P2 Assembly Language Reference Manual

Complete PASM2 Instruction Set Documentation

June 2026

Version 3.0

Reference Manual Organization

Complete P2 Assembly Language Documentation

Part I: Architecture

- The P2 Execution Model
- The Instruction Format
- Flags and Conditional Execution
- Timing and Determinism
- Special Hardware Overview
- Address Modes

Part II: Language Reference

- Instructions (A-Z)
- Directives
- Constants
- Special Registers

Part III: Appendices (A–J)

Iron Sheep Productions, LLC

P2 Knowledge Base Project

Contents

Copyright and License

Acknowledgments

This manual would not exist without the contributions of many individuals and organizations:

Parallax Inc. for creating the Propeller 2 microcontroller and providing comprehensive reference documentation that forms the foundation of this work.

Chip Gracey for the brilliant design of the P2 architecture and for maintaining detailed technical specifications.

The P2 Community for extensive testing, feedback, and real-world usage that has refined our understanding of the instruction set and identified critical details worth documenting.

Open Source Contributors who have developed tools, compilers, and applications that demonstrate the power and flexibility of PASM2.

This manual is a community-developed resource, created to make the P2's assembly language more accessible to developers at all skill levels.

How to Use This Manual

This manual serves multiple audiences and use cases. The organization is designed to support both learning and reference workflows.

For Different Reader Types

New to P2: Start with Part I, Chapters 1-2 to understand the P2 architecture **AND** instruction format fundamentals. These chapters provide essential context for understanding how PASM2 instructions work. Then explore Part II selectively based on what you need to accomplish.

Experienced P1 Users: See "For P1 Developers" below for a specification comparison and overview of new capabilities. Then use Part II as the primary reference—the instruction-by-instruction format will feel familiar.

Looking Up a Specific Instruction: Go directly to Part II, which is organized alphabetically by instruction name. Each entry provides complete syntax, encoding, behavior, and examples.

Quick Reference Needed: Part III appendices provide dense lookup tables organized by category, encoding pattern, and flag effects for rapid consultation.

For P1 Developers

The Propeller 2 preserves the core Propeller philosophy—eight symmetric COGs sharing Hub memory—while dramatically expanding capabilities.

Specification Comparison

	P1	P2
Clock	80 MHz	180 MHz recommended; 250 MHz typical overclock; 350 MHz absolute max[1]
Clocks/Instruction	4	2
Hub RAM	32 KB	512 KB
COG RAM	512 longs	512 + 512 LUT
I/O	32 pins	64 Smart Pins
Math	Software	CORDIC
Interrupts	None	3 per COG
Instructions	~60	~380

[1] Per P2 Datasheet. Higher frequencies require adequate thermal management.

Architecture That Transfers

- Eight independent COGs with true parallel execution
- Shared Hub memory with round-robin deterministic access

- Private COG RAM for fast local operations
- Wired-OR I/O model preventing pin contention
- Hardware locks for inter-COG synchronization
- Spin/PASM language structure

New in P2

- **Smart Pins** — 64 pins with autonomous ADC, DAC, PWM, serial protocols, USB
- **Lookup RAM** — 512 additional longs per COG for tables and overflow code execution
- **CORDIC** — Hardware math: multiply, divide, square root, trig, logarithms
- **Streamer** — Background DMA between Hub, LUT, and pins
- **Digital Video** — Hardware HDMI/DVI output via Streamer
- **FIFO** — Hardware FIFO for high-bandwidth hub streaming and hub execution
- **Interrupts** — Three levels per COG (plus hidden **DEBUG** interrupt) with 16 event sources
- **Debug Interrupt** — Hidden hardware interrupt for single-stepping and breakpoints
- **COGATN** — Hardware inter-COG attention signaling
- **Register Indirection** — **ALTS**, **ALTD**, **ALTR** for dynamic register addressing
- **Instruction Skipping** — **SKIP**, **SKIPF**, **EXECF** for conditional block execution
- **Hub Execution** — Run code directly from 512 KB Hub RAM

Changed from P1

- **Counters**: CTRA/CTRB replaced by Smart Pin event system
- **Video**: VCFG/VSCL/WAITVID replaced by Streamer and DAC capabilities
- **ROM Tables**: Sine/log/antilog tables replaced by CORDIC operations
- **Boot Pins**: P28-P31 changed to P58-P63

Instruction Format Comparison

The 32-bit instruction **WORD** changed between P1 and P2:

Field	P1	P2	Notes
Condition	Bits 21:18 (4 bits)	Bits 31:28 (4 bits)	Moved to MSBs
Opcode	6 bits	7 bits	Expanded for more instructions
CZI/ZCRI	ZCRI (4 bits)	CZI (3 bits)	R bit removed
D/S	9 bits each	9 bits each	Unchanged

The R (result) bit from P1's ZCRI field was removed in P2. Result writing is now controlled differently depending on the instruction.

Begin with Chapter 1 to understand the P2 execution model. Part II serves as the alphabetical instruction reference—a format familiar from P1 documentation.

Manual Structure

Part I: Architectural Foundation — Six chapters explaining how the P2 works:

- Chapter 1: The P2 Execution Model
- Chapter 2: The Instruction Format
- Chapter 3: Flags and Conditional Execution
- Chapter 4: Timing and Determinism
- Chapter 5: Special Hardware Overview
- Chapter 6: Address Modes

Part II: Language Reference — Complete documentation of all PASM2 elements:

- Instructions (alphabetically organized)
- Directives (assembly-time commands)
- Constants (predefined values)
- Special Registers (hardware registers)

Part III: Appendices — Quick reference materials:

- Appendix A: Instruction Encoding Summary
- Appendix B: Condition Code Reference
- Appendix C: Categorical Instruction Index
- Appendix D: Special Registers Reference
- Appendix E: Predefined Constants
- Appendix F: Smart Pin Mode Constants
- Appendix G: Streamer Mode Constants
- Appendix H: Reserved Words Reference
- Appendix I: Glossary
- Appendix J: Known Bugs

Quick Navigation Guide

"I need to find instruction X" → Part II, Instructions section, alphabetically organized

"I need to understand the architecture" → Part I, read Chapters 1-2 sequentially

"I need encoding details" → Appendix A (encoding summary tables)

"I need to find instructions by category" → Appendix C (grouped by function: arithmetic, logic, memory, etc.)

"I need to understand condition codes" → Appendix B (complete IF_x reference with all aliases)

"I need to know what flags an instruction affects" → Part II (each instruction entry) or Appendix A (Instruction Encoding Summary — C Effect / Z Effect columns)

"I need Smart Pin configuration values" → Appendix F (Smart Pin Mode Constants)

"I need CORDIC operations" → Chapter 5.1 (CORDIC Coprocessor) or Part II instruction entries (QMUL, **QDIV**, etc.)

Conventions Used in This Manual

Typography

`Monospace font` is used for code examples, instruction names in syntax descriptions, register names, and literal values.

Bold text is used for instruction names when mentioned in prose, emphasis of important concepts, and section headings.

Italic text is used for emphasis, the first use of technical terms, and parameter names in descriptions.

UPPERCASE is used for instruction mnemonics, register names (PA, PTRA, DIRA), and condition codes (IF_C, IF_Z).

Code Examples

PASM2 code examples follow standard formatting conventions:

```
label           instruction     D,S             ' Comment
                instruction     D,#immediate    ' Indented code
        IF_C    instruction     D,S     WCZ     ' With condition and effects
```

- Labels are flush left
- Instructions are indented to column 16 (two tabs or 8 spaces)
- Operands follow the instruction
- Conditions precede the instruction; effects follow operands (see Chapter 3)
- Comments start with a single quote (') and explain the operation
- 8-character column alignment for readability

Special Markers

Throughout this manual, special markers highlight important information:

Pitfall: Common mistakes or non-obvious behavior that can cause errors. Pay careful attention to these to avoid debugging challenges.

Tip: Useful techniques, optimization opportunities, or best practices that experienced P2 developers have discovered.

Hardware: Hardware-specific considerations, timing constraints, or interactions with P2 peripherals that affect instruction behavior.

Instruction Encoding Tables

Part II instruction entries include encoding tables with the following columns:

EEEE — Condition code field (4 bits). Determines when instruction executes based on flag states.

Opcode — Opcode bits. The instruction-specific portion of the 32-bit encoding.

CZI — Flag effects field (3 bits). Controls which flags are updated and how.

Dest — Destination register (9 bits). Where the result is written.

Src — Source register or immediate value (9 bits). Second operand for the instruction.

C — Effect on the Carry flag: set (1), cleared (0), modified based on result, or unchanged (—).

Z — Effect on the Zero flag: set (1), cleared (0), modified based on result, or unchanged (—).

Result — What value gets written to the destination register.

Clks — Execution time in system clock cycles.

Cross-References

This manual uses consistent cross-reference formats:

MOV — Hyperlink to a Part II instruction entry (in digital versions)

"See Chapter X" — Reference to Part I chapters for architectural context

"See Appendix X" — Reference to Part III appendices for quick reference tables

"Compare: OTHER_INSTRUCTION" — Points to related or contrasting instructions

About This Manual

This manual represents a comprehensive effort to document the P2 Assembly Language (PASM2) in a format optimized for both human learning and AI-assisted development. The content is derived from official Parallax documentation, community expertise, and extensive verification against the P2 silicon behavior.

The manual is designed to be:

Complete — Every documented instruction, directive, constant, and special register is included with full details.

Accurate — Information has been verified against official sources and tested on actual P2 hardware.

Accessible — Content is organized for multiple skill levels and use cases, from learning to quick reference.

Structured — Consistent formatting enables both human reading and programmatic parsing for tool development.

We welcome feedback, corrections, and suggestions for improvement. This is a living document that will evolve with the P2 community's growing expertise.

You are now ready to explore the P2 Assembly Language. Whether you are learning for the first time or looking up specific details, this manual is designed to support your journey into P2 development.

Part I: Architectural Foundation

Chapter 1: The P2 Execution Model

The Propeller 2 microcontroller implements a unique multi-processor architecture that differs fundamentally from conventional microcontrollers. Understanding this architecture is essential for effective PASM2 programming.

1.1 The Eight-COG Architecture

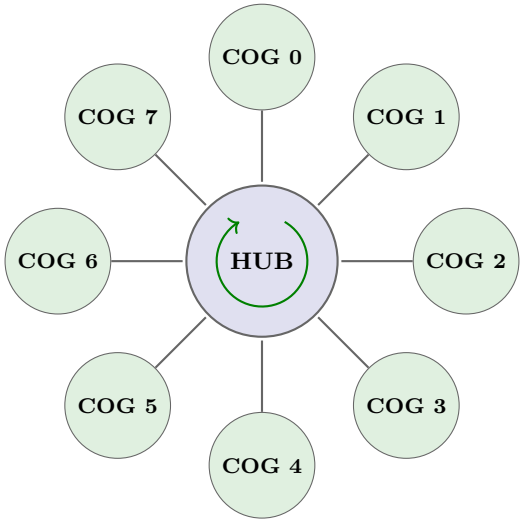


Figure 1.1. Figure 1.1: Eight-COG Architecture Overview

The P2 contains eight identical processors called COGs (Cog Processors). Each COG:

- Executes instructions independently and simultaneously
- Has its own dedicated 512-long register file
- Operates at full clock speed with deterministic timing
- Shares access to a common Hub memory

1.1.1 COG Independence

Unlike conventional microcontrollers that use time-slicing or task switching, the P2 implements true parallel execution. Each COG runs at full clock speed simultaneously with all other COGs. There is no scheduler, no context switching overhead, and no need for traditional interrupts to handle multiple tasks.

This architecture provides deterministic timing. The same code executing on a COG takes exactly the same number of clock cycles every time it runs. This predictability makes the P2 ideal for real-time applications such as video generation, motor control, and protocol implementation where precise timing is essential.

Each COG operates independently. One COG can execute a tight control loop while another manages communications and a third handles user interface tasks. All eight COGs run simultaneously without interfering with each other's timing.

1.1.2 COG Identification

Each COG has a unique identifier from 0 to 7. A COG can determine its own identifier using the `COGID` instruction, which writes the COG number to the destination register. This capability allows the same code to run on multiple COGs while behaving differently based on COG identity.

COGs can communicate with each other through shared Hub memory, hardware locks, and attention signals. The `COGATN` instruction allows one COG to signal other COGs through hardware attention flags, providing fast inter-COG notification without polling shared memory locations.

1.1.3 Starting and Stopping COGs

The `COGINIT` instruction starts a new COG or restarts an existing one. **COGINIT** specifies which COG to start (0-7), where the code resides in Hub memory, and optionally passes a parameter to the new COG. The parameter value appears in the new COG's PTRB register, providing a simple mechanism for initialization data.

The `COGSTOP` instruction halts a running COG. A COG can stop itself or another COG by specifying the target COG number. Stopped COGs consume no power and can be restarted later with different code.

1.2 COG Memory

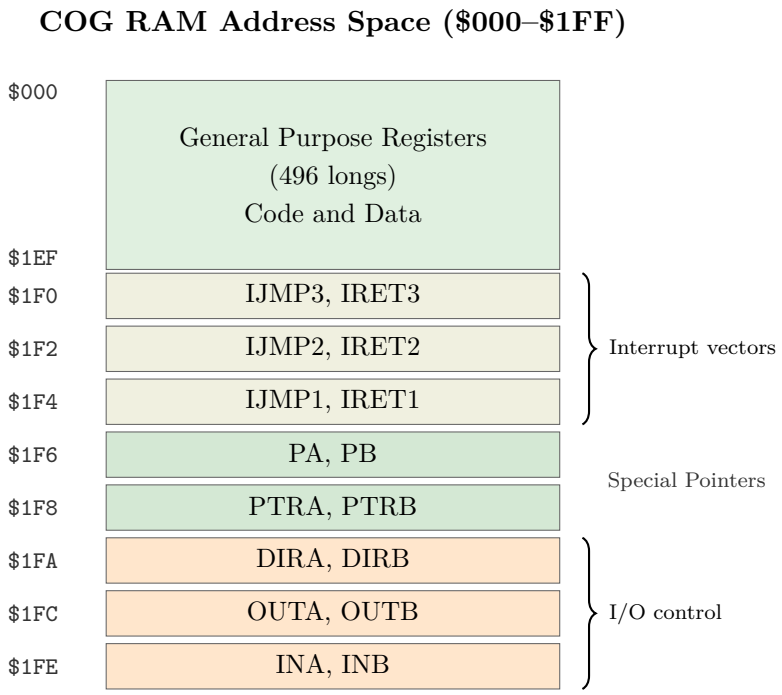


Figure 1.2. Figure 1.2: COG Memory Map

Each COG has 512 longs (2048 bytes) of dedicated RAM addressed from $000 to $1FF. This memory is private to each COG and provides single-cycle read and write access. Unlike Hub memory, COG memory stores 32-bit longs only and uses long-addressing rather than byte-addressing.

1.2.1 General Purpose Registers ($000-$1EF)

The first 496 longs ($000-$1EF) serve as general-purpose registers available for code and data storage. In PASM2, these locations function as registers rather than traditional memory. Instructions specify source and destination operands by register address, and the assembler translates symbolic names to these addresses.

Programs can use this space flexibly. A small program might dedicate most of the space to data storage and lookup tables. A larger program uses more space for code and less for data. The programmer controls this allocation through the assembler's **ORG** directive and **RES** directive for reserving data space.

Registers $1D8-$1DF have predefined symbols PR0-PR7 for Spin2 interoperability. For standalone PASM2 programs, these are ordinary general-purpose registers. See Part II: Special Registers for details on Spin2/PASM2 communication.

1.2.2 Special Purpose Registers ($1F0-$1FF)

The final 16 registers have dedicated hardware functions. Registers $1F0-$1F7 (IJMP3/IRET3, IJMP2/IRET2, IJMP1/IRET1, PA, PB) serve dual purposes: they function as interrupt vectors and **CALL**/return storage when those features are enabled, or as general-purpose RAM otherwise. Registers $1F8-$1FF (PTRA, PTRB, DIRA, DIRB, OUTA, OUTB, INA, INB) are fixed special registers that always provide their hardware I/O and pointer functions.

For complete documentation of each register, see Part II: Special Registers and Appendix C: Special Registers Quick Reference.

1.2.3 Register Addressing

PASM2 instructions use 9-bit fields to specify source (S) and destination (D) register addresses. Nine bits provide 512 possible values, addressing the complete COG RAM space from $000 to $1FF. The instruction encoding dedicates specific bit positions to these address fields, and the assembler automatically encodes symbolic register names into the appropriate bit patterns.

1.3 LUT Memory

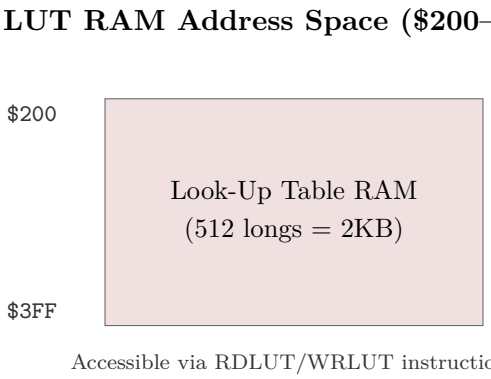


Figure 1.3. Figure 1.3: LUT Memory Map

Each COG has a dedicated 512-long Lookup Table (LUT) providing additional fast memory separate from the main COG RAM space. The LUT serves as auxiliary storage for lookup tables, waveform data, additional code space, or working memory.

1.3.1 LUT Characteristics

LUT memory occupies a separate address space from COG RAM, addressed at $200-$3FF relative to COG addressing. Programs access LUT through dedicated **RDLUT** and **WRLUT** instructions. **RDLUT** takes 3 clock cycles and **WRLUT** takes 2 cycles—both faster than Hub access but slower than direct COG register operations. This separation doubles the available fast memory per COG from 512 longs to 1024 longs total.

LUT RAM can also execute code at the same speed as COG RAM (2 clocks per instruction), making it valuable "overflow" code space when programs exceed COG RAM capacity. When the program counter is in the range $200-$3FF, the COG fetches instructions from LUT memory with the same deterministic timing as COG execution.

The LUT integrates with the P2's streamer and cordic subsystems. The streamer can directly output LUT contents to pins for waveform generation, and cordic operations can store results in LUT memory. This integration makes the LUT particularly valuable for signal generation and digital signal processing applications. A common application is paletted VGA display, where the LUT stores a 256-color palette and the streamer translates 8-bit pixel values to RGB output in real-time.

1.3.2 LUT Instructions

`RDLUT` reads a value from LUT memory to a COG register. `WRLUT` writes a value from a COG register to LUT memory. These instructions work similarly to regular COG memory operations but target the separate LUT address space.

Programs often load the LUT with data from Hub memory at initialization using `SETQ` for burst transfers, then access the LUT repeatedly during time-critical operations. This pattern keeps frequently-accessed data in fast LUT memory while larger datasets remain in Hub memory.

1.3.3 LUT Sharing Between COGs

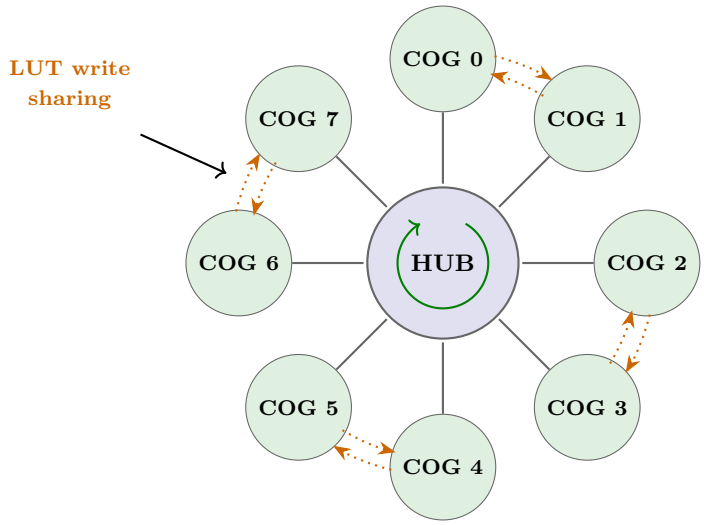


Figure 1.4. Figure 1.4: Eight-COG Architecture with LUT Write Sharing

The SETLUTS instruction activates write-sharing of LUT memory between adjacent COG pairs. When a COG executes SETLUTS #1, the paired COG's WRLUT writes are copied into this COG's LUT via the LUT's second port. This is one-directional; for two-way mirroring both COGs of the pair must execute SETLUTS #1. Adjacent pairs are COGs 0-1, 2-3, 4-5, and 6-7. Each COG retains its own 512-long LUT; **SETLUTS** activates cross-COG write access rather than expanding LUT size. This feature supports producer-consumer patterns where one COG generates data that another COG consumes, eliminating the need to transfer data through Hub memory.

1.4 Hub Memory

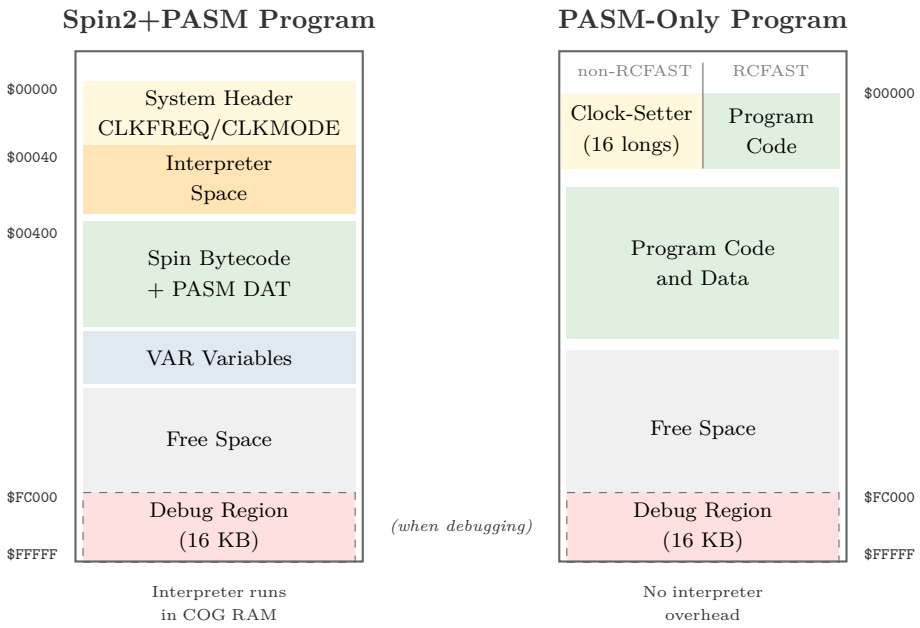


Figure 1.5. Figure 1.5: Hub Memory Layout: Spin2+PASM vs PASM-Only Programs

The Hub provides 512KB of shared RAM accessible by all COGs. Unlike COG memory, Hub memory is byte-addressable and stores programs, data, and resources shared among COGs.

1.4.1 Hub Address Space

Hub memory spans addresses $00000 through $7FFFF, providing 524,288 bytes of storage. All eight COGs can read and write any location in this space. Hub memory stores bytes, words (16-bit), and longs (32-bit) with appropriate address alignment.

Programs use Hub memory to share data between COGs, store large lookup tables, hold program code for Hub execution mode, and buffer data for I/O operations. Each COG accesses Hub memory through dedicated Hub instructions that handle shared access timing.

Hub memory organization is application-defined. Programs allocate space according to their requirements—there is no fixed layout imposed by hardware. Different applications use differ-

ent organizations: some reserve specific regions for communication buffers, others dedicate areas to code overlays, and boot loaders may use particular addresses for compatibility.

Pitfall: Hub addresses below $400 overlap with the region from which COGs load initial code during **COGINIT**. Writing to this area while COGs are being started can cause unpredictable behavior. Programs that dynamically start COGs should avoid using low hub addresses for shared data storage.

1.4.2 Hub Access Timing

Hub RAM is divided into eight "slices"—one per COG. Each slice holds every eighth **LONG** in the composite Hub RAM address space. On every clock cycle, each COG can access the "next" RAM slice in sequence. This arrangement supports continuous bidirectional streaming of 32 bits per clock for sequential addresses.

When a COG accesses a specific Hub address, it must wait up to 7 clocks to reach the initial RAM slice of interest. Once aligned, subsequent sequential locations can be accessed on every clock thereafter for continuous reading or writing of 32-bit longs. This slice architecture differs fundamentally from P1's rotating hub window and provides substantially higher sustained bandwidth.

The hardware FIFO smooths out data flow for non-sequential or variable-rate access. The FIFO can be configured for hub-RAM-read or hub-RAM-write operation, allowing sequential transfers in any combination of bytes, words, or longs at rates up to one **LONG** per clock. The FIFO maintains proper hub slice alignment without programmer intervention.

Hub read instructions (RDBYTE/RDWORD/RDLONG) take 9-16 clocks in COG/LUT execution mode (9-26 in Hub execution mode). Hub write instructions (WRBYTE/WRWORD/WRLONG) take 3-10 clocks in COG/LUT mode (3-20 in Hub execution mode). All ranges are egg-beater hub-window dependent. Hub control instructions (HUBSET, **COGINIT**, LOCK*, CORDIC) have different timing of 2-9 clocks.

Despite the variable initial wait, hub timing remains deterministic. The maximum wait is always seven clocks, and once aligned, sequential access proceeds at one **LONG** per clock. Programs requiring precise timing use COG execution mode for critical sections and Hub memory for data storage and inter-COG communication.

1.4.3 Hub Instructions

PASM2 provides six primary instructions for Hub memory access. `RDBYTE` reads a byte, `RDWORD` reads a word, and `RDLONG` reads a long from Hub memory to a COG register. `WRBYTE`, `WRWORD`, and `WRLONG` write the corresponding data sizes from a COG register to Hub memory.

The `SETQ` instruction enhances Hub access efficiency by configuring burst transfers to COG RAM. **SETQ** followed by a Hub read instruction loads multiple consecutive values in a single operation, amortizing the Hub window wait time across many transfers. Similarly, `SETQ2` configures burst transfers to LUT RAM—use **SETQ**2 before **RDLONG**/WRLONG to transfer blocks directly between Hub and LUT memory.

For high-bandwidth streaming, RDFAST and WRFAST configure the hardware FIFO for continuous Hub transfers. The FIFO prefetches data in the background, hiding Hub access latency from the program. FBLOCK provides dynamic control over FIFO buffer boundaries for seamless ping-pong buffering. These streaming instructions are documented in detail in Chapter 4.

Other hub-related instructions include lock instructions (LOCKNEW, LOCKRET, LOCKTRY, LOCKREL) for inter-COG synchronization, HUBSET for clock and system configuration, and SETLUTS for LUT sharing configuration between adjacent COGs.

The CORDIC coprocessor also interacts with Hub memory. CORDIC operations can read operands from and write results to Hub addresses, enabling efficient processing of large datasets stored in Hub RAM.

1.5 The Execution Pipeline

The P2 implements a five-stage pipelined execution architecture. When the pipeline is full, each instruction effectively takes as little as two clock cycles to execute, providing high throughput while maintaining predictable timing.

Most instructions complete in two clock cycles once the pipeline fills. The first instruction through the pipeline takes five clocks to reach completion. Once the pipeline is full, subsequent instructions complete at a rate of one per two clocks, giving an effective throughput of one instruction every two clocks in steady-state execution.

Hub memory instructions add variable delays waiting for Hub access windows. The hub access rotation means a Hub instruction might execute immediately or wait up to seven clocks for its COG's access slot. This variability affects only Hub memory operations; pure COG operations maintain consistent two-clock timing.

When executing from Hub RAM (Hub execution mode), the COG uses its FIFO hardware to prefetch instructions rather than rotating hub access. The FIFO queues instructions ahead of execution, providing smoother instruction flow. However, this dedicates the FIFO to instruction fetch, making it unavailable for **RDFAST**/WRFAST streaming operations during Hub execution.

Branch instructions incur additional overhead when taken. A conditional branch that is not taken completes in two clocks like other instructions. A taken branch causes the pipeline to be flushed, so the first instruction following the branch takes at least five clock cycles as the pipeline refills from the branch target address.

The P2 handles data dependencies internally through forwarding logic. An instruction that depends on the result of the immediately preceding instruction receives the correct value without requiring explicit programmer intervention or **NOP** insertion. This hardware forwarding removes a major class of pipeline hazards present in simpler architectures (see Chapter 4 for timing detail).

Register indirection instructions (ALTS, **ALTD**, **ALTR**, **ALTB**, **ALTI**) perform dynamic instruction modification within the pipeline. These instructions substitute computed addresses or values into the next instruction's source, destination, or result fields without modifying the actual program

code in memory. The next instruction following any ALT instruction is shielded from interrupts, guaranteeing atomic execution of the ALT+target instruction pair. This pipeline-level modification supports powerful indirect addressing patterns while maintaining deterministic timing.

1.6 Execution Modes

The P2 supports three execution modes based on the program counter address, each offering different trade-offs between speed and capacity. Programs can use any mode exclusively or mix all three modes within a single application.

Mode	PC Range	Characteristics
COG Execution	$00000-$001FF	Fastest, 2 clocks/instruction, 512 longs
LUT Execution	$00200-$003FF	Fast, 2 clocks/instruction, 512 longs overflow
Hub Execution	$00400-$7FFFF	Largest capacity, variable timing, uses FIFO

1.6.1 COG Execution Mode

COG execution mode runs code from COG RAM (PC in range $000-$1FF). Instructions execute in the consistent two-clock pipeline with no additional delays. This mode provides the fastest possible execution and deterministic timing, making it ideal for time-critical code such as communication protocols, motor control loops, and signal generation.

COG execution mode limits programs to the available COG RAM space. After accounting for special registers and data storage, typically 200-400 longs remain for code. Programs that fit in this space achieve maximum performance. Larger programs can overflow into LUT execution or use Hub execution mode.

1.6.2 LUT Execution Mode

LUT execution mode runs code from LUT RAM (PC in range $200-$3FF). Instructions execute at the same speed as COG execution—two clocks per instruction with deterministic timing. LUT execution effectively doubles the available fast code space from 512 to 1024 longs per COG.

LUT execution is ideal for overflow code that doesn't fit in COG RAM but requires deterministic timing. The COG fetches instructions from LUT memory with no additional delays beyond the standard pipeline. There are no special considerations when branching between COG and LUT addresses.

Time-critical inner loops often execute in COG or LUT mode even when the main program runs from Hub memory. The program loads critical code sections to COG/LUT RAM, executes the loop, then returns to Hub-based code. This hybrid approach combines the performance of local execution with the capacity of Hub storage.

1.6.3 Hub Execution Mode

Hub execution mode runs code directly from Hub RAM without loading it to COG memory first. The COG fetches instructions from Hub memory using the FIFO hardware to prefetch and queue instructions for continuous execution. This is distinct from the hub rotation used for random-access data transfers. The FIFO provides smoother instruction flow but **ADDS** variable delay compared to COG mode.

Hub execution mode provides access to the full 512KB Hub address space, enabling programs far larger than COG memory could hold. In practice, Hub-executed code typically resides at addresses $400 and above—the `ORGH` directive defaults to $400, reserving low addresses for COG initialization data. The mode suits applications where code size exceeds available COG RAM and deterministic timing is less critical. User interface code, data processing algorithms, and high-level control logic typically run well in Hub execution mode.

`COGINIT` determines execution mode when starting a COG. The initialization parameter specifies either COG execution (code loaded from Hub to COG RAM, then executed) or Hub execution (code executed directly from Hub RAM). The `ORGH` assembler directive marks code intended for Hub execution, while `ORG` marks code for COG execution.

Pitfall: While executing from Hub RAM, the FIFO hardware is dedicated to instruction prefetch and cannot be used for other purposes. The following instructions are unavailable during Hub execution: **RDFAST**, **WRFAST**, **FBLOCK**, **RFBYTE**, **RFWORD**, **RFLONG**, **RFVAR**, **RFVARS**, **WFBYTE**, **WFWORD**, **WFLONG**, and the streamer FIFO instructions **XINIT**, **XZERO**, and **XCONT** when the streamer mode engages the FIFO. Code requiring these instructions must execute from COG RAM.

1.6.4 Switching Between Modes

Programs switch between execution modes using `CALL` or `JMP` instructions. A COG executing from COG RAM can call or jump to Hub addresses, and Hub-executing code can call or jump to COG addresses. The program counter determines current mode: addresses $000-$3FF indicate COG/LUT execution, while higher addresses indicate Hub execution.

The hardware handles mode transitions transparently. The programmer specifies the target address, and the COG switches to the appropriate execution mode based on the address range. This seamless transition supports hybrid programs that place performance-critical code in COG RAM while maintaining larger program logic in Hub RAM.

Key Concepts

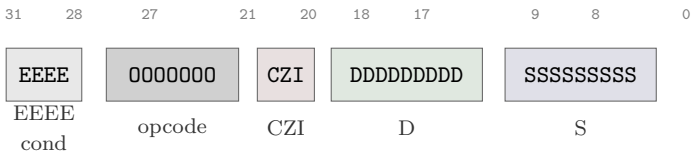
- The P2 has 8 independent COGs executing in true parallel
- Each COG has 512 longs of private RAM plus 512 longs of private LUT
- Hub memory (512KB) is shared among all COGs with deterministic access timing
- Special registers at $1F0-$1FF provide hardware I/O functions
- COGs can execute from COG RAM (fast), LUT RAM (fast), or Hub RAM (larger capacity)—three distinct execution modes
- Hub execution uses FIFO for instruction prefetch; FIFO instructions unavailable in Hub mode
- The pipeline provides two-clock execution for most instructions
- No interrupts are required due to true parallel execution; however, complete interrupt mechanisms are provided

Chapter 2: The Instruction Format

Every PASM2 instruction is encoded in a 32-bit **WORD** with a consistent structure. Understanding this format enables reading the encoding tables in Part II and manually encoding or decoding instructions when needed.

2.1 The 32-Bit Instruction Word

Every PASM2 instruction occupies exactly one 32-bit **LONG** with this structure:



2.1.1 Field Summary

Field	Bits	Width	Purpose
EEEE	31-28	4	Condition code for conditional execution
OOOOOOO	27-21	7	Opcode identifying the instruction
CZI	20-18	3	Flag effects and immediate mode
DDDDDDDDD	17-9	9	Destination register address
SSSSSSSSS	8-0	9	Source operand (register or immediate)

2.1.2 The CZI Field

The three bits at positions 20-18 control flag behavior and operand mode:

Bit	Position	Purpose
C	20	C flag write enable (1 = update C flag)
Z	19	Z flag write enable (1 = update Z flag)
I	18	Immediate mode (1 = S is immediate value)

When WC is specified in source code, the assembler sets bit 20 to 1. When WZ is specified, bit 19 is set. When # prefixes the source operand, bit 18 is set.

2.2 Condition Codes (EEEE Field)

The condition field enables conditional execution of any instruction. The instruction executes only if the specified condition is true based on the current C and Z flags.

2.2.1 Condition Code Summary

The 4-bit EEEE field encodes sixteen conditions:

EEEE	Primary Mnemonic	Condition	Description
0000	_RET_	Always	Execute, then return if no branch
0001	IF_NC_AND_NZ	C=0 AND Z=0	No carry and not zero
0010	IF_NC_AND_Z	C=0 AND Z=1	No carry and zero
0011	IF_NC	C=0	No carry (C flag clear)
0100	IF_C_AND_NZ	C=1 AND Z=0	Carry and not zero
0101	IF_NZ	Z=0	Not zero (Z flag clear)
0110	IF_C_NE_Z	C!=Z	C and Z flags differ
0111	IF_NC_OR_NZ	C=0 OR Z=0	Not both flags set
1000	IF_C_AND_Z	C=1 AND Z=1	Both flags set
1001	IF_C_EQ_Z	C=Z	C and Z flags same
1010	IF_Z	Z=1	Zero (Z flag set)
1011	IF_NC_OR_Z	C=0 OR Z=1	No carry or zero
1100	IF_C	C=1	Carry (C flag set)
1101	IF_C_OR_NZ	C=1 OR Z=0	Carry or not zero
1110	IF_C_OR_Z	C=1 OR Z=1	Either flag set
1111	IF_ALWAYS	Always	Unconditional (when no condition specified)

Complete Reference: Each condition has multiple aliases for different contexts (comparison aliases like IF_GT/IF_A, flag state aliases like IF_00/IF_11, and logical aliases like IF_SAME/IF_DIFF). For the complete alias table and detailed documentation, see **Appendix B: Condition Code Reference**.

2.2.2 The __RET__ Condition

The condition code 0000 (_RET_) has special behavior: it means **"Always execute the instruction, then return if the instruction did not branch."**

When an instruction has EEEE=0000:

1. **The instruction always executes** (condition 0000 means "always" for _RET_)
2. **If the instruction does not branch**: Return by popping stack[19:0] into PC
3. **If the instruction branches** (JMP, **CALL**, etc.): No return occurs—the branch takes precedence
4. **No context restore**: Unlike RET WCZ, the _RET_ prefix does NOT restore C or Z flags

Basic Usage:

```
        _ret_   ADD     x, y            ' ADD then return
        _ret_   DRVNOT  #0              ' Toggle pin 0, then return
        _ret_   MOV     result, temp    ' Copy to result, then return
```

Single-Instruction Subroutines:

The _RET_ prefix enables efficient single-instruction subroutines:

```
toggle_pin0                             ' Subroutine: toggle pin 0
        _ret_   DRVNOT  #0              ' 2 + 2 return = 4 cycles
```

This is significantly faster than a separate instruction followed by **RET**.

Timing: The _RET_ prefix triggers a **RET** (stack-pop) return: +2 cycles incremental return cost in COG/LUT mode. In Hub-exec mode the embedded return costs more due to FIFO refill on the branch — the **RET** hub-exec range is 13…20 cycles (ret.yaml).

> **Complete Reference:** For advanced _RET_ usage including branch behavior, XBYTE bytecode interpreter patterns, and **SKIP**/SKIPF combinations, see **Appendix B: Condition Code Reference**.

2.2.3 Comparison Condition Aliases

When comparing values with **CMP**, **CMPS**, **SUB**, or similar instructions, the resulting C and Z flags can be tested with condition prefixes that express comparison semantics. The P2 provides two equivalent terminology styles for comparison aliases:

Comparison Result	Flag State	Magnitude Style	Arithmetic Style
Greater than	C=0, Z=0	IF_A (Above)	IF_GT (Greater Than)
Greater or equal	C=0	IF_AE (Above or Equal)	IF_GE (Greater or Equal)
Less than	C=1	IF_B (Below)	IF_LT (Less Than)
Less or equal	C=1 OR Z=1	IF_BE (Below or Equal)	IF_LE (Less or Equal)
Equal	Z=1	IF_E	IF_E
Not equal	Z=0	IF_NE	IF_NE

Both styles encode to identical condition codes—the choice is purely stylistic. Use whichever terminology reads best for your code.

Magnitude terminology (A = Above, B = Below) reads naturally with values like addresses, counts, and sizes:

```
1         MOV     addr, ##$80000000       ' addr = 2,147,483,648
2         CMP     addr, #0        wcz     ' Compare
3         if_a    JMP     #addr_is_larger ' "addr is above zero"
```

Arithmetic terminology (GT = Greater Than, LT = Less Than) reads naturally with values like temperatures, positions, and deltas:

```
1         MOV     x, ##-100               ' x = -100 (signed)
2         MOV     y, #50                  ' y = 50
3         CMPS    x, y            wcz     ' Signed compare: -100 vs 50
4         if_lt   JMP     #x_is_smaller   ' "x is less than y"
```

CMP vs. CMPS:

The distinction that matters is the **compare instruction**, not the alias style:

- **CMP** performs unsigned subtraction (for setting flags)
- **CMPS** performs signed subtraction (for setting flags)

After **CMP**, the flags reflect unsigned ordering. After **CMPS**, the flags reflect signed ordering. Either condition code terminology (magnitude aliases like IF_A/IF_B, or arithmetic aliases like IF_GT/IF_LT—see Section 2.2.3) works correctly with either instruction:

```
' Unsigned comparison - either style works
        CMP     a, b            wcz
        if_ae   MOV     result, #1      ' "a is above or equal to b"
        if_ge   MOV     result, #1      ' "a greater or equal to b" (same)

' Signed comparison - either style works
        CMPS    a, b            wcz
        if_ge   MOV     result, #1      ' "a is greater or equal to b"
        if_ae   MOV     result, #1      ' "a is above or equal to b" (same)
```

2.2.4 Conditional Execution Patterns

Conditional execution eliminates branches, providing deterministic timing:

```
' Instead of branching:
                CMP     a, b            wcz
        if_z    JMP     #equal_handler          ' 4 cycles if taken
                MOV     result, #0

' Use conditional execution:
                CMP     a, b            wcz
        if_z    MOV     result, #1              ' Always 2 cycles
        if_nz   MOV     result, #0              ' Always 2 cycles
```

Common patterns:

Minimum/Maximum:

```
                CMP     a, b            wc      ' Compare unsigned
        if_c    MOV     min, a                  ' min = a if a < b
        if_nc   MOV     min, b                  ' min = b if a >= b
```

Conditional Assignment:

```
                TEST    flags, #MASK    wz      ' Test bit
        if_nz   MOV     mode, #1                ' Set if bit present
```

Multi-way Selection:

```
                CMP     selector, #0    wz
        if_z    MOV     result, value0
                CMP     selector, #1    wz
        if_z    MOV     result, value1
                CMP     selector, #2    wz
        if_z    MOV     result, value2
```

2.3 Reading Encoding Tables

Each instruction entry in Part II includes an encoding table with nine columns. The table shows the instruction's binary encoding on the left and its effects on the right.

2.3.1 Encoding Columns (Left Five)

The left five columns show the 32-bit instruction encoding:

Column	Content	Description
COND	EEEE	Condition field (4 bits, always EEEE for conditional instructions)
INSTR	7 bits	The instruction's unique opcode (positions 27-21)
FX	CZI variant	Flag modification and immediate bits (positions 20-18)
DEST	DDDDDDDDD	Destination field pattern (positions 17-9)
SRC	SSSSSSSSS	Source field pattern (positions 8-0)

2.3.2 Result Columns (Right Four)

The right four columns describe instruction effects:

Column	Content	Description
Write	What's written	Which register(s) receive output (D, PC, etc.)
C Flag	C behavior	How C flag is affected, or "—" for no change
Z Flag	Z behavior	How Z flag is affected, or "—" for no change
Clocks	Cycle count	Execution time in clock cycles

2.3.3 The FX Field Variations

The FX column shows which flag and immediate options are available:

FX Pattern	Meaning
CZI	C modifiable (WC), Z modifiable (WZ), Immediate allowed (#)
0ZI	C not modifiable, Z modifiable, Immediate allowed
C0I	C modifiable, Z not modifiable, Immediate allowed
00I	Neither flag modifiable, Immediate allowed
CZ0	Flags modifiable, Immediate not allowed (register only)
NNI	NN bits encode sub-function (e.g., byte number), Immediate allowed
LLI	LL bits encode sub-function, Immediate allowed

When FX shows fixed bits (like `000` or `01I`), those bits have fixed values and the corresponding options are not available.

2.3.4 Special Values in Columns

Write column:

Value	Meaning
`D`	Destination register is written
`D and PC`	Both destination and program counter written (for jumps/calls); rendered `D + PC*` in the tables
`PC`	Only PC written
`---`	Nothing written, or output goes to Hub/LUT memory rather than a COG register (compare, test, and memory-write instructions)
`OUTx`	Pin output state written
`DIR bit`	A pin direction bit is written
`OUT bit`	A pin output bit is written
`DIRx, OUTx`	Pin direction and output state written
† / *	Footnote markers flagging conditional or qualified write behavior

Flag columns:

Value	Meaning
`---`	Flag is not changed
Descriptive text	Describes condition that sets/clears the flag

Clocks column:

Value	Meaning
`2`	Always 2 clock cycles
`2+`	Minimum 2 cycles, may be more
`2 or 4`	2 if condition false/not taken, 4 if true/taken
`2 / 8-23`	COG mode cycles / Hub mode cycles
`9..35`	Variable range depending on operands

2.4 Understanding Multiple Encoding Rows

Some instruction entries show multiple rows in the encoding table. Each row represents a unique machine code encoding.

2.4.1 Instruction Families

When related instructions share an entry (e.g., **DIRZ**/DIRNZ), each instruction gets its own row:

DIRZ / DIRNZ

EEEE	Opcode	CZI	D	S	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000100	DIRx	—	DIR bit	2
EEEE	1101011	CZL	DDDDDDDDD	001000101	DIRx	—	DIR bit	2

The first row is **DIRZ** (S = 001000100), the second is **DIRNZ** (S = 001000101). Both share the same opcode but differ in the SRC field.

2.4.2 Multiple Syntax Forms

When one instruction has multiple syntax forms with different encodings:

GETBYTE

Syntax 1: `GETBYTE Dest, {#}Src, #Num`

Syntax 2: `GETBYTE Dest`

EEEE	Opcode	CZI	D	S	C	Z	Result	Clks
EEEE	1000111	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1000111	000	DDDDDDDDD	000000000	—	—	D	2

The first row shows the standard form with Src and Num operands (NN encodes the byte number 0-3). The second row is the ALTGB-driven form (GETBYTE Dest = GETBYTE Dest,0,#0): a prior ALTGB instruction rewrites this instruction's pipelined Src and Num fields to point at the next byte in Reg RAM, and GETBYTE writes that byte into Dest.

2.4.3 Key Principle

Each unique machine code encoding = one table row. If two mnemonics produce different bit patterns, they appear as separate rows. If one mnemonic has multiple valid encodings (different syntax forms), each encoding appears as a row.

2.5 Destination and Source Fields

2.5.1 The Destination Field (D)

The 9-bit D field (bits 17-9) addresses a COG register from $000 to $1FF:

- **Read and written:** Most ALU instructions read D, compute, and write result back to D
- **Read only:** Compare instructions (CMP, **CMPS**, **TEST**) read D but do not modify it
- **Write only:** Some move instructions write D without reading its previous value

The D field can also specify:

- Hub addresses (for **ALTD**-modified instructions)
- LUT addresses (for LUT instructions)
- Pin numbers (for certain I/O instructions)

2.5.2 The Source Field (S)

The 9-bit S field (bits 8-0) has two modes controlled by the I bit:

Register mode (I = 0):

- S is a COG register address ($000-$1FF)
- The value in that register is used as the operand

Immediate mode (I = 1):

- S is a 9-bit unsigned value (0-511)
- This value is used directly as the operand

```
1         ADD     result, counter         ' S = register address (I=0)
2         ADD     result, #100            ' S = immediate 100 (I=1)
```

2.5.3 When S is Fixed

Some encodings show fixed S values instead of SSSSSSSSS. These instructions use the S field to encode which specific operation to perform:

EEEE	Opcode	CZI	D	S
EEEE	1101011	CZI	DDDDDDDDD	001000100

Fixed value selects DIRZ

The fixed value distinguishes this instruction from others sharing the same opcode. The programmer does not specify this value; it is implicit in the instruction mnemonic.

2.6 Immediate Operands

2.6.1 The # Prefix (9-bit Immediate)

The `#` prefix before an operand indicates an immediate value:

```
1          ADD     result, #100            ' Add immediate 100
2          ADD     result, value           ' Add contents of register 'value'
3          MOV     x, #$1FF                ' Load maximum 9-bit value (511)
```

When `#` is used:

- The assembler sets the I bit (bit 18) to 1
- The S field contains the 9-bit value

2.6.2 Immediate Range

9-bit immediates can represent:

- Unsigned: 0 to 511 ($000 to $1FF)
- Signed (when interpreted): -256 to +255

Values outside this range require augmentation (see Section 2.7).

2.6.3 The $ Prefix for Current Address

The `$` symbol represents the current assembly address:

```
1 loop     ADD     counter, #1
2          DJNZ    count, #$-1             ' Jump back one instruction
```

When used with `#`, it becomes an immediate representing the address.

2.7 Augmented Immediates

2.7.1 The ## Prefix (32-bit Immediate)

The `##` prefix indicates a full 32-bit immediate value:

```
        MOV     dest, ##$12345678       ' Load full 32-bit value
        ADD     counter, ##1000000      ' Add 1 million
        MOV     ptr, ##hub_data         ' Load 20-bit Hub address
```

2.7.2 AUGS and AUGD Instructions

The assembler implements 32-bit immediates by inserting AUG instructions:

- **AUGS** - Augments the Source field for the following instruction
- **AUGD** - Augments the Destination field for the following instruction

The AUG instruction provides the upper 23 bits, which combine with the lower 9 bits from the next instruction:

```
' What the programmer writes:
        MOV     dest, ##$12345678

' What the assembler generates:
        AUGS    #$12345678              ' Upper 23 bits (bits [31:9])
        MOV     dest, #$078             ' Provides lower 9 bits: $078
                                        ' Combined result: $12345678
```

2.7.3 Augmentation Behavior

The AUG instruction must immediately precede the instruction it augments:

1. The AUG executes, storing the 23-bit value internally
2. The next instruction combines this with its 9-bit field
3. The combined 32-bit value is used for that instruction only
4. The augmentation is consumed (one-shot)

If any instruction intervenes (including a conditional **NOP**), the augmentation is lost.

Timing Overhead:

Each AUG instruction **ADDS +2 clock cycles** to the total execution time. When using `##` notation:

Operands	AUG Instructions	Additional Cycles
`##Src` only	1 (AUGS)	+2 cycles
`##Dest` only	1 (AUGD)	+2 cycles
`##Dest, ##Src`	2 (AUGD + AUGS)	+4 cycles

```
1         MOV     x, #100                 ' 2 cycles (no augmentation)
2         MOV     x, ##100000             ' 4 cycles (2 + 2 for AUGS)
3         WRLONG  ##dest, ##addr          ' 6 cycles (AUGD+AUGS+instr)
```

Critical Timing Note: In time-critical code, consider keeping values in registers rather than using repeated `##` augmentation, especially inside loops.

2.7.4 When Augmentation is Required

Augmentation is needed when:

- Values exceed 9 bits (> 511 for unsigned)
- Hub addresses are used (20-bit address space)
- 32-bit constants are needed
- Pin masks exceed 9 bits

```
1         WRLONG  value, ##$1000          ' Hub address $1000 (> 511)
2         MOV     mask, ##$FFFF0000       ' 32-bit mask
3         WAITX   ##1000000               ' Delay > 511 cycles
```

2.8 How to Use This Manual

2.8.1 Looking Up an Instruction

1. **Find the instruction** alphabetically in Part II
2. **Read the syntax** to understand valid operand forms
3. **Check the Result** line for what the instruction produces
4. **Review Parameters** for operand requirements and constraints
5. **Use the Encoding table** when you need machine code details
6. **Read Related** instructions for alternatives and family members
7. **Study Explanation** for complete behavioral description

2.8.2 Visual Anchors: Color Bars

Each entry in Part II has a colored bar on the left edge of its header block. These color bars serve as visual anchors, making it easy to locate entry boundaries when scanning through pages.

The colors indicate entry type:

Color	Entry Type	Description
Red	Instruction	PASM2 machine instructions (the majority of entries)
Amber	Directive	Assembler directives like ORG, BYTE, LONG
Violet	Constant	Pre-defined constants like smart pin mode values

The color bar spans the three-line identity block at the top of each entry:

1. **Mnemonic** — The instruction, directive, or constant name
2. **Expansion** — What the mnemonic stands for (e.g., "Add Signed, Extended")
3. **Category** — The functional category with a brief description

When flipping through Part II, these color bars help you quickly identify entry boundaries and distinguish between instructions, directives, and constants.

2.8.3 Example: Understanding ADD

Consider the **ADD** instruction entry:

ADD — Math Instruction — Add two unsigned values.

`ADD Dest, {#}Src {WC|WZ|WCZ}`

Result: Sum of unsigned Src and unsigned Dest is stored in Dest.

- Dest is a register containing the value to add Src to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is added into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	D	S	C	Z	Result	Clks
EEEE	0001000	CZI	DDDDDDDDD	SSSSSSSSS	carry of (D + S)	Result = 0	D	2

From this entry:

- **Category:** Math Instruction - this is arithmetic
- **Syntax:** `{#}Src` means Src can be register or immediate; `{WC|WZ|WCZ}` means flag effects are optional
- **Result:** The sum goes into Dest (Dest is modified)
- **Encoding:** Opcode is 0001000 (7 bits); FX is CZI meaning all options available; takes 2 cycles
- **C Flag:** Set if addition overflows (unsigned carry)
- **Z Flag:** Set if result is zero

2.8.4 Using Categories for Discovery

Instructions are grouped by category in Appendix C. When looking for "an instruction that does X," consult the categorical index:

- **Math Instructions:** ADD, **SUB**, **MUL**, etc.
- **Logic Instructions:** AND, OR, **XOR**, etc.
- **Branch/Jump Instructions: JMP**, **CALL**, **DJNZ**, etc.
- **Hub Memory Instructions: RDLONG**, **WRLONG**, etc.

Tip: In the PDF version, the category name in each entry's header block is a clickable link that jumps directly to that category's listing in Appendix C.

2.8.5 Navigating with Links

The PDF version of this manual includes extensive cross-reference links to help you navigate efficiently. Links appear in blue text and are clickable:

In the entry header block:

- The **Category name** links to Appendix C's categorical listing

In the Related line:

> **Related:** ADDX, ADDS, ADDSX, SUB

Each instruction name in the Related section is a clickable link that jumps directly to that instruction's entry. This makes it easy to explore instruction families:

- **ADDX**: ADD with carry-in (for multi-precision)
- **ADDS**: Signed addition
- **ADDSX**: Signed addition with carry-in
- **SUB**: The opposite operation

Navigation tip: Use your PDF reader's "back" function (often Alt+Left Arrow or `Cmd+[`) to return to where you were after following a link.

2.9 Constant Expressions and Operators

PASM2 allows constant expressions anywhere a numeric value is expected. These expressions are evaluated at assembly time—the resulting value is encoded into the instruction, not computed at runtime. This enables readable, self-documenting code using symbolic calculations.

2.9.1 Where Constant Expressions Apply

Constant expressions can appear in:

- **Immediate operands:** `MOV x, #(BUFFER_SIZE - 1)`
- **CON block definitions:** `MAX_COUNT = 1000 * 60`
- **Data declarations:** `LONG $FF << 24 | $80 << 16`
- **ORG/ORGH directives:** `ORG $100 + HEADER_SIZE`
- **Repeat counts:** `REP @loop_end, #(TABLE_SIZE / 4)`

2.9.2 Operator Reference

Operators are listed from highest to lowest precedence within each category.

Unary Operators (highest precedence)

Operator	Description	Example
`!`	Bitwise NOT (invert all bits)	`!$FF` → `$FFFFFF00`
`+`	Positive (no effect, explicit sign)	`+5` → `5`
`-`	Negate (two's complement)	`-1` → `$FFFFFFFF`

Bitwise Operators

Operator	Description	Example
`>>`	Shift right	`$80 >> 4` → `$08`
`<<`	Shift left	`1 << 8` → `$100`
`&`	Bitwise AND	`$FF & $0F` → `$0F`
`|`	Bitwise OR	`$F0 | $0F` → `$FF`
`^`	Bitwise XOR	`$FF ^ $0F` → `$F0`

Arithmetic Operators

Operator	Description	Example
`+`	Addition	`100 + 50` → `150`
`-`	Subtraction	`100 - 50` → `50`
`*`	Multiplication (lower 32 bits, signed)	`1000 * 1000` → `1000000`
`/`	Division quotient (signed)	`-100 / 3` → `-33`
`+/`	Division quotient (unsigned)	`$FFFFFFFF +/ 2` → `$7FFFFFFF`
`//`	Division remainder/modulo (signed)	`-100 // 3` → `-1`
`+//`	Division remainder (unsigned)	`$FFFFFFFF +// 16` → `15`

Limit Operators

Operator	Description	Example
`#>`	Limit minimum (signed)	`x #> 0` — ensures x 0
`<#`	Limit maximum (signed)	`x <# 255` — ensures x 255

Comparison Operators

Comparison operators return -1 (true, all bits set) or 0 (false).

Operator	Description	Signed/Unsigned
`<`	Less than	Signed
`+<`	Less than	Unsigned
`>`	Greater than	Signed
`+>`	Greater than	Unsigned
`<=`	Less than or equal	Signed
`+<=`	Less than or equal	Unsigned
`>=`	Greater than or equal	Signed
`+>=`	Greater than or equal	Unsigned
`==`	Equal	(n/a)
`<>`	Not equal	(n/a)

Boolean Operators

Operator	Description	Example
`!!`	Boolean NOT (0→-1, non-zero→0)	`!!5` → `0`
`&&`	Boolean AND	`(a > 0) && (b > 0)`
`||`	Boolean OR	`(a == 0) || (b == 0)`
`^^`	Boolean XOR	`(a > 0) ^^ (b > 0)`
`<=>`	Three-way compare (returns -1, 0, or 1)	`5 <=> 3` → `1`

Ternary Operator (lowest precedence)

Operator	Description	Example
`? :`	Conditional selection	`(x > 0) ? x : -x` — absolute value

2.9.3 Signed vs. Unsigned Comparisons

The `+` prefix on comparison operators indicates unsigned comparison. This matters when comparing values that may have the high bit set:

```
' Signed comparison: $80000000 is negative (-2147483648)
        IF  $80000000 < 0       ' True: negative < 0

' Unsigned comparison: $80000000 is positive (2147483648)
        IF  $80000000 +< 0      ' False: 2147483648 is not < 0
```

Use signed comparisons (`<`, `>`, etc.) for values representing signed quantities. Use unsigned comparisons (`+<`, `+>`, etc.) for addresses, bit patterns, or values that should never be negative.

2.9.4 Practical Examples

Bit field construction:

```
CON
  PIN_MODE  = %01 << 5 | %11 << 3 | %1 << 0   ' Combine fields
  MASK_BITS = (1 << NUM_BITS) - 1              ' Create bit mask
```

Buffer calculations:

```
CON
  BUFFER_END = BUFFER_START + BUFFER_SIZE - 1
  WRAP_MASK  = BUFFER_SIZE - 1                 ' For power-of-2 buffers
```

Conditional assembly values:

```
CON
  DELAY_MS = (CLKFREQ / 1000) #> 1              ' At least 1 tick
  TIMEOUT  = (MAX_WAIT < 1000) ? MAX_WAIT : 1000  ' Clamp to 1000
```

2.10 Labels and Symbol Scoping

PASM2 supports two scoping levels for labels within DAT blocks: global labels and local labels. This scoping mechanism enables reuse of common label names (such as `loop`, `done`, `exit`) without naming collisions across different routines.

2.10.1 Global Labels

Global labels are defined by placing an identifier at the start of a line without any prefix character.

Syntax:

```
labelname       instruction     operands        ' comment
```

Global labels have these characteristics:

- Visible throughout the entire DAT block
- Can be referenced from Spin2 code using `@labelname`
- Defining a new global label resets the local label scope
- Must begin with a letter (A-Z, a-z) or underscore (_)
- May contain letters, digits (0-9), and underscores
- Maximum length: 30 characters

Example:

```
DAT             ORG

' Global labels - visible everywhere in DAT block
init_routine    MOV     x, #0                   ' Routine entry point
                ADD     x, #1
                RET

data_table      LONG    $DEAD_BEEF              ' Data with global label
                LONG    $CAFE_BABE

math_helper     ABS     x                       ' Another routine
                RET
```

2.10.2 Local Labels

Local labels are defined by prefixing an identifier with either a dot (`.`) or colon (`:`). Both prefix characters are functionally equivalent.

Syntax:

```
.labelname      instruction     operands        ' comment
:labelname      instruction     operands        ' comment
```

Local labels have these characteristics:

- Visible only within the scope of the preceding global label
- Scope ends when the next global label is defined
- The same local name can be reused under different global labels
- Internally mangled by the compiler (e.g., `loop'0001`) for uniqueness
- Must begin with a letter or underscore after the prefix

Example:

```
DAT             ORG
                
send_byte       RDBYTE  x, ptr                   ' Global: send_byte
                CALL    #.wait                  ' Reference local .wait
.loop           TESTP   tx_pin          wc      ' Local .loop in send_byte
        if_nc   JMP     #.loop
                WYPIN   x, tx_pin
.wait           TESTP   tx_pin          wc      ' Local .wait in send_byte
        if_c    JMP     #.wait
                RET
                
recv_byte       TESTP   rx_pin          wc      ' Global: recv_byte
                                                '   (new scope begins)
        if_nc   JMP     #.wait                  ' Different .wait, new scope
.wait           TESTP   rx_pin          wc      ' Local .wait in recv_byte
        if_nc   JMP     #.wait
                RDPIN   x, rx_pin
.loop           SHR     x, #24                  ' Local .loop in recv_byte
                RET
```

The example demonstrates how `.loop` and `.wait` can be reused in both `send_byte` and `recv_byte` without collision. Each global label creates a new local scope.

2.10.3 Label Reference Operators

PASM2 provides several operators for referencing labels in different contexts:

Operator	Meaning	Context
`#label`	Immediate value (COG address)	PASM instructions
`#.local`	Immediate reference to local label	PASM instructions
`#\label`	Absolute COG-relative address	Forces 9-bit COG address
`@label`	Hub address of label	Spin2 or PASM
`@@label`	Object-relative address	Spin2 or PASM
`$`	Current COG address	PASM (ORG mode)
`$$`	Current Hub address	PASM (ORGH mode)

Example:

```
1  DAT             ORG
2
3  routine         JMP     #.SKIP                  ' Jump to local label
4                  LONG    0
5  .SKIP           MOV     x, #routine             ' Load address of global
6                  CALL    #\.helper               ' Absolute call to local
7                  RET
8
9  .helper         NOP
10                 RET
11
12 ' In ORGH (Hub) mode:
13                 ORGH
14 hub_data        BYTE    "Hello", 0
15 hub_routine     LONG    @routine                ' Hub address of COG routine
```

2.10.4 Scope Boundary Rules

Three events create scope boundaries:

1. **Global label definition** — Starts a new local scope
2. **Storage directives** (BYTE, **WORD**, **LONG**, **RES** with a label) — Also start a new local scope
3. **End of DAT block** — Terminates all label scopes

Example:

```
DAT             ORG
                                                
func_a          MOV     x, #1                   ' Global: func_a,
                                                '   scope #1 begins
.loop           DJNZ    x, #.loop               ' Local .loop in scope #1
                                                
data_block      LONG    0, 0, 0, 0              ' Global: data_block,
                                                '   scope #2 begins
                                                
func_b          MOV     y, #2                   ' Global: func_b,
                                                '   scope #3 begins
.loop           DJNZ    y, #.loop               ' Local .loop in scope #3
                                                '   (different)
.done           RET                             ' Local .done in scope #3
```

2.10.5 Best Practices

Use descriptive global names for routine entry points: `send_packet`, `init_uart`, `calc_crc`

Use short local names for flow control: `.loop`, `.done`, `.retry`, `.SKIP`, `.exit`

Prefer dot notation (`.label`) over colon notation (`:label`) for consistency with modern convention

Keep local labels near their references to improve readability

Limit symbol names to 30 characters for compatibility with the PNut compiler

Key Concepts

- Every instruction is exactly 32 bits: 4-bit condition, 7-bit opcode, 3-bit flags, 9-bit D, 9-bit S
- The EEEE condition field enables conditional execution based on C and Z flags
- The I bit (position 18) determines whether S is a register address (0) or immediate value (1)
- 9-bit immediates range 0-511; larger values require ## augmentation
- AUGS/AUGD extend immediates to full 32 bits by inserting an extra instruction before the target
- Encoding tables show both the bit pattern (left 5 columns) and the effects (right 4 columns)
- Multiple table rows indicate instruction families or syntax variants with different encodings
- The _RET_ condition (EEEE=0000) transforms any instruction into a subroutine return
- Global labels are visible throughout a DAT block; local labels (.name or :name) are scoped to the preceding global label

Chapter 3: Flags and Conditional Execution

The P2 has two status flags that enable conditional execution and multi-precision arithmetic. Understanding flag behavior is essential for writing efficient, branching-free code.

The P2's flag system differs from many processors in two important ways. First, flags persist until explicitly modified—an instruction without WC or WZ effects leaves flags unchanged, allowing flag values to be used by multiple subsequent instructions. Second, any instruction can be made conditional using IF_x prefixes, enabling deterministic branchless programming where instruction timing remains constant regardless of data values.

These two features combine to create a powerful programming model where complex decision logic can be expressed without branches, maintaining cycle-accurate timing while reducing code size and improving readability.

3.1 The C and Z Flags

Each COG maintains two independent status flags that track computation results and enable conditional execution. These flags are named C (Carry) and Z (Zero), but their meanings extend beyond these basic interpretations depending on the instruction that sets them.

3.1.1 The C Flag (Carry/Borrow)

The C flag serves multiple purposes depending on instruction context:

For arithmetic operations, C indicates **unsigned overflow** after addition (carry out of bit 31) or **unsigned borrow** after subtraction (when the subtrahend exceeds the minuend). This enables multi-precision arithmetic where carries or borrows propagate between 32-bit operations.

For shift and rotate operations, C captures the **bit value shifted out** of the result. A left shift stores bit 31 in C; a right shift stores bit 0 in C. This enables implementing shifts wider than 32 bits by chaining operations.

For comparison operations, C indicates the **relationship between operands**. After an unsigned comparison (CMP), C=1 means the first operand is below (less than) the second. After a signed comparison (CMPS), C=1 means the first operand is less than the second using signed interpretation.

For logical operations, C indicates **parity**—whether the result contains an odd number of 1 bits. This specialized behavior supports error detection and certain bit manipulation patterns.

3.1.2 The Z Flag (Zero)

The Z flag indicates **zero result** or **equality** across most instructions:

For arithmetic and logical operations, Z=1 when the result equals zero. This enables testing for zero values, detecting exhausted counters, or identifying cleared registers.

For comparison operations, Z=1 when the operands are equal. This works for both signed (CMPS) and unsigned (CMP) comparisons—equality has the same meaning regardless of interpretation.

For bit test operations, Z=1 when the tested bits are all clear. The TEST instruction ANDs its operands and sets Z based on whether the result is zero, effectively testing whether any specified bits are set.

3.1.3 Flag Persistence and Independence

Flags retain their values until explicitly modified by a WC, WZ, or WCZ effect. This persistence is a deliberate design feature that enables powerful programming patterns:

```
1                CMP     a, b            wcz     ' Set flags once
2        if_c    MOV     min, a                  ' Use C here
3        if_nc   MOV     min, b                  ' And here
4        if_z    MOV     equal, #1               ' And use Z here
```

In this example, one comparison sets both flags, and three subsequent instructions each test the preserved flag values. No instruction between them modifies the flags, so the flag state from the comparison remains available.

Each COG maintains its own C and Z flags completely independently. Flag values in COG 0 have no relationship to flag values in COG 1. This independence ensures parallel execution across COGs operates without interference.

3.2 Flag Modification Effects

Every instruction can optionally specify which flags to update using effect modifiers. These modifiers—WC, WZ, and WCZ—control whether the instruction modifies the C flag, the Z flag, both flags, or neither flag. The operation always executes; effects only determine whether flags are updated.

3.2.1 The WC Effect

```
1        ADD     result, value   wc      ' Update C flag based on carry
```

When WC (Write C) is specified, the instruction updates the C flag according to its specific C condition while leaving Z unchanged. For ADD, this means C is set if the addition produces a carry out of bit 31. For **CMP**, this means C is set if the first operand is less than the second. Each instruction defines its own C condition as documented in the instruction reference.

The key insight: WC means "update C according to this instruction's C rule." The rule varies by instruction, but the WC effect itself is consistent—it enables C modification.

3.2.2 The WZ Effect

```
1        ADD     result, value   wz      ' Update Z flag based on result
```

When WZ (Write Z) is specified, the instruction updates the Z flag based on whether the result equals zero, while leaving C unchanged. Z=1 indicates a zero result; Z=0 indicates a non-zero result. This behavior is consistent across nearly all instructions—the Z flag always reflects "is the result zero?"

This consistency makes WZ predictable. After any arithmetic, logical, or shift operation with WZ, checking IF_Z tests whether the result was zero. After a comparison with WZ, checking IF_Z tests whether the operands were equal.

Exception: Extended Instructions (Z AND behavior)

The extended arithmetic instructions—ADDX, **SUBX**, **ADDSX**, **SUBSX**, **CMPX**, **CMPSX**—use a modified Z flag update rule:

```
Z = Z AND (result == 0)
```

Instead of simply replacing Z with the zero **TEST**, these instructions AND the new zero status with the existing Z flag. This behavior is essential for multi-precision arithmetic:

```
' 64-bit addition: [hi:lo] += [bhi:blo]
        ADD     lo, blo         wcz     ' Add low 32 bits, Z = (lo == 0)
        ADDX    hi, bhi         wcz     ' High + carry, Z = Z AND (hi==0)
        ' Z is now 1 only if BOTH lo and hi were zero
        '  (entire 64-bit result is zero)
```

Without this AND behavior, the final Z flag would only reflect the last 32-bit operation, losing information about whether the full multi-precision result was zero. The AND logic accumulates zero detection across all operations in the chain.

Source Verification: CSV v35 documents this as "Z = Z AND (Result = 0)" for all extended instructions.

3.2.3 The WCZ Effect

```
        ADD     result, value   wcz     ' Update both flags
```

When WCZ (Write C and Z) is specified, both flags are updated according to their respective conditions. You can specify WC to update only C, WZ to update only Z, or WCZ to update both—these are the three valid effect options.

WCZ is common after comparisons where both the ordering (C) and equality (Z) matter, or after arithmetic operations where both carry detection and zero detection are needed.

3.2.4 Special Flag Effects (ANDC/ANDZ/ORC/ORZ/XORC/XORZ)

The **TESTB**, **TESTBN**, **TESTP**, and **TESTPN** instructions support additional flag effects that perform bitwise operations on the existing flag value rather than replacing it. These enable testing multiple bits and accumulating the results into a single flag.

Effect	Operation	Description
ANDC	C = C AND bit	AND tested bit into C
ANDZ	Z = Z AND bit	AND tested bit into Z
ORC	C = C OR bit	OR tested bit into C
ORZ	Z = Z OR bit	OR tested bit into Z
XORC	C = C XOR bit	XOR tested bit into C
XORZ	Z = Z XOR bit	XOR tested bit into Z

Unlike WC and WZ which replace the flag value, these effects combine the tested bit with the existing flag value using the specified boolean operation.

Use Case: Testing Multiple Bits

The most common use is testing whether ALL bits in a set are high (AND), or whether ANY bit in a set is high (OR):

```
' Test if ALL of pins 0, 4, and 7 are high (AND pattern)
        TESTP   #0              wc      ' C = pin 0 state
        TESTP   #4              andc    ' C = C AND pin 4 state
        TESTP   #7              andc    ' C = C AND pin 7 state
        ' C = 1 only if ALL three pins are high

' Test if ANY of pins 0, 4, or 7 is high (OR pattern)
        TESTPN  #0              wc      ' C = NOT pin 0 (so C=0 if pin high)
        TESTPN  #4              andc    ' C = C AND NOT pin 4
        TESTPN  #7              andc    ' C = C AND NOT pin 7
        ' C = 0 if ANY pin is high, C = 1 if ALL pins are low
```

TESTB vs TESTP:

- **TESTB** tests a bit within a register: `TESTB reg, #bit_number`
- **TESTP** tests a pin's input state: `TESTP #pin_number`
- **TESTBN** and **TESTPN** test the inverted bit or pin state

Source Verification: CSV v35 documents these as `C/Z = C/Z AND/OR/XOR D[S[4:0]]` for **TESTB** variants and `C/Z = C/Z AND/OR/XOR IN[D[5:0]]` for **TESTP** variants.

3.2.5 No Effect (Default)

```
        ADD     result, value           ' Execute operation, preserve flags
```

When no effect is specified, the instruction executes normally but leaves both C and Z unchanged. This is not a "do nothing" mode—the operation completes, the destination is written, and timing is identical to the flagged version. Only the flags are preserved.

This behavior enables using flag values across multiple instructions without interference:

```
                CMP     a, b            wc      ' Set C based on comparison
                MOV     temp, c                 ' Does not modify C
                ADD     temp, d                 ' Does not modify C
        if_c    MOV     result, temp            ' Tests original C
```

The comparison sets C, and two subsequent operations execute without modifying it. The conditional instruction tests the comparison result even though two operations occurred in between.

3.2.6 Effect Availability

Not all instructions support all effect modifiers. Each instruction defines which effects are valid based on whether its C and Z outputs have meaningful interpretations. The assembler validates effect usage and reports an error when an invalid effect is specified.

Effect Categories:

- **Full support (WC, WZ, WCZ):** Most ALU instructions—ADD, **SUB**, **CMP**, AND, OR, **MOV**, etc.—support all three effects because both flags have independent, meaningful interpretations.
- **WCZ only:** Pin and bit manipulation instructions (DRV, *BIT*, DIR, *FLT*, OUT*) set both flags to the same value—the original state before modification. Using WC or WZ alone is not allowed; use WCZ or omit effects entirely.
- **WC only or WZ only:** Some instructions produce meaningful output for only one flag. For example, **LOCKTRY** sets C to indicate lock acquisition but has no meaningful Z output.
- **Extended effects (no WCZ):** The TEST* instructions (TESTP, **TESTPN**, **TESTB**, **TESTBN**) support WC, WZ, and extended effects (ANDC, ORC, XORC, ANDZ, ORZ, XORZ) for accumulating multiple tests, but reject WCZ.

```
' Examples of effect restrictions
        ADD     x, y            wcz     ' Full support: WC, WZ, or WCZ
        DRVH    #pin            wcz     ' WCZ only: WC or WZ alone not OK
        LOCKTRY #0              wc      ' WC only: WZ and WCZ not allowed
        TESTP   #pin            andc    ' Extended: WC, WZ, ANDC (no WCZ)
```

Each instruction entry in Part II documents its allowed effects in the encoding table. For a complete reference of effect restrictions by instruction category, see **Appendix C: Categorical Instruction Index**.

3.3 Conditional Execution

The P2 allows any instruction to execute conditionally based on the current flag values. This conditional execution mechanism enables branchless programming—expressing decision logic without jump instructions—which maintains deterministic timing and often reduces code size.

3.3.1 The IF_x Prefix

Any instruction can be made conditional by prefixing with an IF_x condition. When the condition is false, the instruction does not execute, but still consumes its normal execution time (2 clock cycles). When the condition is true, the instruction executes normally:

```
1                CMP     a, b            wcz     ' Compare, set flags
2        if_z    MOV     result, #1              ' Only if Z=1 (equal)
3        if_nz   MOV     result, #0              ' Only if Z=0 (not equal)
```

This three-instruction sequence sets `result` to 1 if `a` equals `b`, or 0 if they differ. It takes exactly three clock cycles regardless of the comparison result. The unconditional **CMP** always executes, then exactly one of the two conditional MOVs executes.

The timing predictability is crucial. Traditional branch-based code has variable timing depending on which path is taken. Conditional execution eliminates this variation—the instruction stream is fixed, and timing is constant.

3.3.2 Conditional Execution Timing

When a conditional instruction's condition is false, the instruction does not execute but still consumes 2 clock cycles. This behavior might seem wasteful, but it provides deterministic timing—critical for real-time operations, protocol timing, and cycle-accurate code.

Consider this example:

```
1                TEST    flags, #BIT_READY  wz   ' Check ready bit
2        if_nz   RDLONG  data, ptr               ' Read if ready
3        if_nz   ADD     ptr, #4                 ' Advance if read occurred
```

This sequence takes exactly three clock cycles whether the ready bit is set or clear. If implementing the same logic with branches:

```
1                TEST    flags, #BIT_READY  wz
2        if_z    JMP     #SKIP
3                RDLONG  data, ptr
4                ADD     ptr, #4
5 SKIP
```

The branch version takes 2 cycles when not ready (test + jump) or 4 cycles when ready (test + not-jump + **RDLONG** + add). The timing varies by 100%. The conditional version maintains constant 3-cycle timing.

For real-time code, deterministic timing often matters more than average speed.

3.3.3 Available Conditions

The P2 provides sixteen conditions covering all possible combinations of C and Z flag states. Each condition can be expressed using its primary mnemonic or one of several aliases designed to make code more readable in specific contexts.

The most commonly used conditions are:

- **IF_C / IF_NC** — Test the C flag (set / clear)
- **IF_Z / IF_NZ** — Test the Z flag (set / clear)
- **(no condition)** — When omitted, instructions execute unconditionally (encodes as EEEE=1111)
- **_RET_** — Execute instruction, then return

> **Complete Reference:** For the full table of all sixteen conditions with their EEEE encodings, flag state patterns, and complete alias listings (comparison aliases, flag state aliases, logical aliases, and commutative forms), see **Appendix B: Condition Code Reference**.

3.3.4 Comparison Condition Aliases

After a comparison instruction (CMP or **CMPS**), the C and Z flags can be tested with aliases that express relational operators. Two equivalent terminology styles are available:

Condition	Magnitude Style	Arithmetic Style	Relational	Meaning
IF_C	IF_B	IF_LT	<	a is less than b
IF_NC	IF_AE	IF_GE	>=	a is greater or equal to b
IF_Z	IF_E	IF_E	==	a equals b
IF_NZ	IF_NE	IF_NE	!=	a not equal to b
IF_NC_AND_NZ	IF_A	IF_GT	>	a is greater than b
IF_C_OR_Z	IF_BE	IF_LE	<=	a is less or equal to b

Both styles encode to identical condition codes—the choice is purely stylistic. Use whichever terminology reads best for your code:

- **Magnitude terminology** (A = Above, B = Below) reads naturally with addresses, counts, and sizes
- **Arithmetic terminology** (GT = Greater Than, LT = Less Than) reads naturally with temperatures, positions, and deltas

The compare instruction determines the comparison type:

- **CMP** performs unsigned subtraction—flags reflect unsigned ordering
- **CMPS** performs signed subtraction—flags reflect signed ordering

Either alias style works correctly with either compare instruction. The choice of **CMP** vs. CMPS determines whether $80000000 is treated as a large positive number or a negative number. The alias you use afterward is simply a matter of which terminology reads better in your code.

3.4 Flag Behavior by Instruction Category

Flag meanings vary by instruction category. Understanding these patterns helps predict flag behavior without consulting the instruction reference for each operation.

3.4.1 Arithmetic Instructions

Arithmetic instructions set C based on unsigned overflow (carry or borrow) and set Z when the result equals zero:

Instruction	C Flag (with WC)	Z Flag (with WZ)
ADD	Unsigned carry out of bit 31	Result = 0
ADDS	True sign of result (corrected sign of D+S)	Result = 0
SUB	Unsigned borrow (A < B)	Result = 0
SUBS	True sign of result (corrected sign of D−S)	Result = 0
CMP	Unsigned borrow (A < B)	A = B
CMPS	Signed A < B (true sign of A−B)	A = B

For ADD, C=1 indicates that adding the operands produced a value larger than 32 bits can represent—a carry occurred. For **SUB** and **CMP**, C=1 indicates the first operand is less than the second (a borrow would be required). The result is always written to the destination (for ADD/SUB) or the flags are set (for **CMP**/CMPS).

ADDS and **SUBS** set C to the true sign of the result (result bit 31), not a signed-overflow flag — it is the sign the value would have at full precision after overflow correction. For signed multi-long arithmetic, use ADD/ADDX (SUB/SUBX) for the lower longs and **ADDSX** (SUBSX) for the final **LONG** so C reflects the overall result's sign.

3.4.2 Logic Instructions

Logical instructions set C based on parity and set Z based on whether the result is zero:

Instruction	C Flag (with WC)	Z Flag (with WZ)
AND	Parity (odd # of 1 bits)	Result = 0
OR	Parity (odd # of 1 bits)	Result = 0
XOR	Parity (odd # of 1 bits)	Result = 0
NOT	Inverse of operand bit 31 (!S[31] / !D[31])	Result = 0

Parity means C=1 when the result contains an odd number of 1 bits, and C=0 when the result contains an even number of 1 bits. This enables parity checking for error detection—XOR all data bits together, and C indicates odd parity.

The Z flag behavior is straightforward: Z=1 when the entire 32-bit result is zero. For AND, this occurs when the operands share no common 1 bits. For OR, this occurs when both operands are zero. For **XOR**, this occurs when the operands are identical.

3.4.3 Shift and Rotate Instructions

Shift and rotate instructions capture the bit shifted or rotated out in the C flag:

Instruction	C Flag (with WC)	Z Flag (with WZ)
SHL	Bit 31 (MSB shifted out)	Result = 0
SHR	Bit 0 (LSB shifted out)	Result = 0
ROL	Bit 31 (MSB rotated out)	Result = 0
ROR	Bit 0 (LSB rotated out)	Result = 0

For left operations (SHL, **ROL**), the most significant bit (bit 31) moves into C. For right operations (SHR, **ROR**), the least significant bit (bit 0) moves into C. This enables multi-precision shifts where the bit shifted out of one **WORD** becomes the bit shifted into the next **WORD**.

The difference between shift and rotate: shifts fill the vacated bit position with 0, while rotates fill it with the bit shifted out (creating a circular rotation). Both capture the bit that exits the register in C.

3.4.4 Move and Data Instructions

Move and data manipulation instructions set flags based on the source or result characteristics:

Instruction	C Flag (with WC)	Z Flag (with WZ)
MOV	MSB of source (S[31])	Source = 0
NEG	Result is negative (result bit 31)	Result = 0
ABS	Source was negative	Result = 0
NOT	Inverse of operand bit 31 (!S[31] / !D[31])	Result = 0
ENCOD	Source was non-zero	Result = 0

MOV is notable because its C flag reflects the sign bit of the source value, not the result (which is identical to the source). This enables sign testing without a separate comparison:

```
1         MOV     temp, value     wc      ' Copy value, C = sign bit
2         if_c    JMP     #negative       ' Branch if negative
```

NEG sets C to the sign bit of the result — C=1 if the negated value is negative, C=0 if it is positive (or zero).

ABS sets C=1 if the source was negative, indicating that the absolute value operation inverted the sign. This flag persists even for the special case of **NEGX** ($80000000), whose absolute value cannot be represented in 32 bits.

3.5 Common Flag Patterns

Understanding common flag usage patterns accelerates learning and provides templates for solving typical programming problems. These patterns demonstrate how flags enable elegant, efficient solutions.

3.5.1 Testing a Bit

Testing whether a specific bit is set uses **TEST** with WZ:

```
                TEST    value, #%00000100  wz   ' Test bit 2
        if_nz   JMP     #bit_set               ' Jump if bit is set
```

TEST performs a bitwise AND of its operands but writes the result nowhere—it only sets flags. The mask `%00000100` isolates bit 2. If bit 2 is set, the AND produces a non-zero result (specifically, the value 4), so Z=0. If bit 2 is clear, the AND produces zero, so Z=1.

The condition IF_NZ tests "not zero," which corresponds to "bit is set." This pattern works for testing any single bit or combination of bits—just construct the appropriate mask.

3.5.2 Multi-Precision Addition

Adding values wider than 32 bits requires propagating the carry between **WORD** additions:

```
        ADD     x_lo, y_lo      wc      ' Add low words, capture carry
        ADDX    x_hi, y_hi              ' Add high words plus carry
```

The first ADD **ADDS** the low 32 bits and sets C if the addition carries out. The **ADDX** instruction (Add with Carry) **ADDS** the high 32 bits plus the carry from the first addition. This extends to any number of words:

```
        ADD     x0, y0          wc      ' Add word 0
        ADDX    x1, y1          wc      ' Add word 1 plus carry
        ADDX    x2, y2          wc      ' Add word 2 plus carry
        ADDX    x3, y3                  ' Add word 3 plus carry
```

Each **ADDX** uses the carry from the previous addition and generates a new carry for the next addition. The result is 128-bit (4×32-bit) addition with correct carry propagation.

3.5.3 Conditional Assignment

Selecting between two values based on a comparison uses conditional moves:

```
                CMP     a, b            wc      ' Compare a and b
        if_c    MOV     result, a               ' If a < b, result = a
        if_nc   MOV     result, b               ' If a >= b, result = b
```

This implements `result = min(a, b)` without branches. The comparison sets C if `a < b` (unsigned). Exactly one of the two conditional moves executes, storing the smaller value in result. The sequence takes exactly three clock cycles regardless of which value is smaller.

For maximum of two values, invert the conditions:

```
1                CMP     a, b            wc      ' Compare a and b
2        if_c    MOV     result, b               ' If a < b, result = b
3        if_nc   MOV     result, a               ' If a >= b, result = a
```

3.5.4 Branchless Absolute Value

Computing the absolute value of a signed number uses the ABS instruction with conditional negation:

```
1                ABS     result, value   wc      ' Abs value, C = negative
2        if_c    NEG     result                  ' Correct if was negative
```

Wait—this looks wrong. If **ABS** already computes the absolute value, why negate it afterward?

The issue is a quirk of two's complement: the most negative value (-2,147,483,648 or $8000_0000) has no positive representation in 32 bits. Its absolute value cannot be represented. The ABS instruction handles this by leaving the value unchanged and setting C to indicate the exceptional case.

For all other negative values, **ABS** correctly computes the absolute value and clears C. For -2,147,483,648, **ABS** leaves it unchanged and sets C, and the conditional **NEG** negates it back to itself (since negating $8000_0000 produces $8000_0000).

Most code doesn't care about this edge case and can simply use `ABS result, value` without the conditional correction.

3.5.5 Conditional Increment/Decrement

Updating a counter only when a condition is met uses conditional arithmetic:

```
1                TEST    flags, #FLAG_READY  wz   ' Test ready flag
2        if_nz   ADD     count, #1               ' Increment if ready
```

This increments `count` only when the ready flag is set. No branches are needed, and timing is deterministic—two clock cycles regardless of flag state.

3.5.6 Bounds Checking

Checking whether a value falls within a range combines comparison and logical conditions:

```
1                CMP     value, min      wc      ' Check if value < min
2        if_c    JMP     #out_of_range           ' Too small
3                CMP     value, max      wc      ' Check if value >= max
4        if_nc   JMP     #out_of_range           ' Too large
5                ' Value is in range [min, max)
```

This checks whether `value` is in the range `[min, max)`. The first comparison tests for too small; the second tests for too large. If either condition fails, the value is out of range.

3.6 Advanced Flag Usage

Beyond basic conditional execution, the P2 provides specialized instructions for manipulating flags directly and using flags to control data flow. These advanced techniques enable sophisticated flag-based algorithms.

3.6.1 Direct Flag Manipulation

The **MODC** and **MODZ** instructions modify flags directly without performing computations:

```
1         MODC    _set    wc      ' Set C flag to 1
2         MODZ    _clr    wz      ' Clear Z flag to 0
```

MODC sets C according to a 4-bit modifier constant, and **MODZ** sets Z similarly. The WC and WZ effects are required for the modification to take effect; without them, the result is computed but discarded. Common modifier constants include `_set` (always 1), `_clr` (always 0), `_c` (current C), and `_z` (current Z).

The **MODCZ** instruction can modify both flags simultaneously:

```
1         MODCZ   _clr, _set  wcz ' Clear C, set Z
2         MODCZ   _set, _set  wcz ' Set both flags
```

MODCZ accepts two operands specifying operations for C and Z respectively. The WC, WZ, or WCZ effect must be specified for the flags to be modified. Modifier constants include `_clr` (clear to 0), `_set` (set to 1), `_nc` (inverted C), `_nz` (inverted Z), and others that enable complex flag manipulation in a single instruction.

3.6.2 Flag-Based Bit Manipulation

The MUX family of instructions uses flag values to conditionally modify individual bits:

```
1         MUXC    value, #mask    ' C=1: set bits; C=0: clear bits
2         MUXNC   value, #mask    ' C=0: set bits; C=1: clear bits
3         MUXZ    value, #mask    ' Z=1: set bits; Z=0: clear bits
4         MUXNZ   value, #mask    ' Z=0: set bits; Z=1: clear bits
```

These instructions conditionally set or clear bits based on flag values. For example, **MUXC** sets the masked bits if C=1, or clears them if C=0. This enables building up bit patterns based on multiple flag tests:

```
1         TEST    input, #BIT0    wc      ' Test bit 0 of input
2         MUXC    output, #%0001          ' Copy bit 0 to output bit 0
3         TEST    input, #BIT1    wc      ' Test bit 1 of input
4         MUXC    output, #%0010          ' Copy bit 1 to output bit 1
```

This pattern extracts and repositions bits based on flag tests, enabling bit-field manipulation.

3.6.3 Flag Preservation Patterns

Sometimes you need to preserve flag values across operations that might modify them. The P2 does not provide a dedicated flag save/restore mechanism, but you can use register operations:

```
        ' Save flags
        WRC     cflag           ' cflag = {31'b0, C}  (C in bit 0)
        WRZ     zflag           ' zflag = {31'b0, Z}  (Z in bit 0)

        ' ... operations that modify flags ...

        ' Restore flags
        TESTB   cflag, #0       wc      ' C from cflag bit 0
        TESTB   zflag, #0       wz      ' Z from zflag bit 0
```

WRC sets the destination register to {31'b0, C} — C in bit 0, all other bits cleared — overwriting the whole register; **WRZ** does the same with Z (also in bit 0). Neither takes a bit-select operand. Because each overwrites the entire register, save C and Z into separate registers (or combine them explicitly, e.g. wrc tmp then **SHL**/or with Z) rather than into two bits of one register. **TESTB** tests a specific bit and sets C or Z accordingly, effectively restoring the saved flag values.

An alternative approach uses **MODCZ** with computed values, but the **TESTB** pattern is more common and more readable.

3.6.4 Flag-Driven State Machines

Flags can encode state transitions in compact state machines. Instead of comparing state variables and branching, use flags to select the next state:

```
                ' Current state determines which flags are set
                TEST    state, #STATE_IDLE      wz
        if_z    JMP     #handle_idle
                TEST    state, #STATE_ACTIVE    wz
        if_z    JMP     #handle_active
                TEST    state, #STATE_DONE      wz
        if_z    JMP     #handle_done
```

This pattern tests state bits and branches to handlers. Each **TEST** sets Z if the state bit is set, and the conditional jump executes for that state. While this uses jumps (not purely branchless), it demonstrates using flags to encode complex state without comparison operations.

3.7 Multi-Long Arithmetic Operations

The P2's flag system enables arithmetic operations on values wider than 32 bits. By chaining instructions that propagate carry/borrow through the C flag and accumulate zero-detection through the Z flag, you can perform addition, subtraction, and comparison on 64-bit, 96-bit, 128-bit, or arbitrarily wide values.

3.7.1 Instruction Family Overview

The P2 provides four variants each for ADD, **SUB**, and **CMP** operations:

Addition Instructions:

Instruction	Operation	C Flag	Z Flag
ADD D, S	D = D + S	Carry out	D result == 0
ADDX D, S	D = D + S + C	Carry out	Z AND (D result == 0)
ADDS D, S	D = D + S	True sign of result	D result == 0
ADDSX D, S	D = D + S + C	True sign of result	Z AND (D result == 0)

Subtraction Instructions:

Instruction	Operation	C Flag	Z Flag
SUB D, S	D = D - S	Borrow	D result == 0
SUBX D, S	D = D - S - C	Borrow	Z AND (D result == 0)
SUBS D, S	D = D - S	True sign of result	D result == 0
SUBSX D, S	D = D - S - C	True sign of result	Z AND (D result == 0)

Comparison Instructions:

Instruction	Operation	C Flag	Z Flag
CMP D, S	X = D - S	Borrow	X == 0
CMPX D, S	X = D - S - C	Borrow	Z AND (X == 0)
CMPS D, S	X = D - S	True sign of X	X == 0
CMPSX D, S	X = D - S - C	True sign of X	Z AND (X == 0)

The key distinctions:

- **Base instructions** (ADD, **SUB**, **CMP**) start a new operation and reset Z
- **X variants** (ADDX, **SUBX**, **CMPX**) propagate carry/borrow and AND the zero result
- **S variants** (ADDS, **SUBS**, **CMPS**) report the true sign instead of carry
- **SX variants** (ADDSX, **SUBSX**, **CMPSX**) combine both: propagate C, AND-accumulate Z, report true sign

3.7.2 The Chaining Pattern

Multi-long operations follow a consistent pattern:

1. **First long:** Use base instruction (ADD, **SUB**, **CMP**) with WCZ
2. **Middle longs:** Use X variant (ADDX, **SUBX**, **CMPX**) with WCZ
3. **Final long:** Use X variant for unsigned, SX variant for signed

The X variants are critical because they:

- Add/subtract the incoming C flag (carry/borrow from previous **LONG**)
- AND the Z result with the previous Z (tracking if all longs are zero)
- Output carry/borrow for the next **LONG**

3.7.3 Unsigned Multi-Long Examples

64-bit unsigned addition (A = A + B):

```
        ADD     A0, B0    WCZ     ' Add low longs, C = carry, Z = (A0 == 0)
        ADDX    A1, B1    WCZ     ' Add high longs + carry, C = carry,
                                  '  Z = Z AND (A1 == 0)
        ' After: C = overflow, Z = (entire 64-bit result == 0)
```

128-bit unsigned addition (A = A + B):

```
        ADD     A0, B0    WCZ     ' A0 = A0 + B0
        ADDX    A1, B1    WCZ     ' A1 = A1 + B1 + carry
        ADDX    A2, B2    WCZ     ' A2 = A2 + B2 + carry
        ADDX    A3, B3    WCZ     ' A3 = A3 + B3 + carry
        ' After: C = overflow beyond 128 bits, Z = (128-bit result == 0)
```

64-bit unsigned subtraction (A = A - B):

```
        SUB     A0, B0    WCZ     ' Subtract low longs, C = borrow
        SUBX    A1, B1    WCZ     ' Subtract high longs - borrow
        ' After: C = underflow (B > A), Z = (result == 0)
```

64-bit unsigned comparison (compare A to B):

```
        CMP     A0, B0    WCZ     ' Compare low longs
        CMPX    A1, B1    WCZ     ' Compare high longs with borrow
        ' After: C = (A < B), Z = (A == B)
        ' Use IF_B (below) or IF_AE (above/equal) for unsigned branches
```

3.7.4 Signed Multi-Long Examples

For signed operations, the final instruction must be an SX variant to correctly report the sign of the overall result.

64-bit signed addition (A = A + B):

```
        ADD     A0, B0    WCZ     ' Add low longs (unsigned, gen carry)
        ADDSX   A1, B1    WCZ     ' Add high longs + carry, C = true sign
        ' After: C = true sign of result (1 = negative), Z = (result == 0)
```

128-bit signed addition (A = A + B):

```
        ADD     A0, B0    WCZ     ' Unsigned add for low long
        ADDX    A1, B1    WCZ     ' Unsigned add + carry for middle longs
        ADDX    A2, B2    WCZ     ' Unsigned add + carry
        ADDSX   A3, B3    WCZ     ' Signed add for high long, C = true sign
        ' After: C = 1 if result is negative, Z = (result == 0)
```

64-bit signed comparison (compare A to B):

```
        CMP     A0, B0    WCZ     ' Compare low longs
        CMPSX   A1, B1    WCZ     ' Compare high, C = sign of difference
        ' After: C = (A < B) signed, Z = (A == B)
        ' Use IF_LT (less than) or IF_GE (greater/equal) for signed branches
```

3.7.5 Understanding "True Sign"

The S and SX variants report the "true sign" of the result rather than carry/borrow. This is the conceptual bit above the MSB—the sign the result would have if computed with infinite precision.

For signed operations:

- If the result is negative (would be negative with more bits), C = 1
- If the result is non-negative, C = 0

This differs from carry/borrow, which indicates overflow in unsigned arithmetic. For signed comparisons, the true sign tells you the sign of (A - B), directly indicating whether A < B.

3.7.6 Practical Pattern Summary

Operation	First Long	Middle Longs	Final Long (Unsigned)	Final Long (Signed)
Add	ADD WCZ	ADDX WCZ	ADDX WCZ	ADDSX WCZ
Subtract	SUB WCZ	SUBX WCZ	SUBX WCZ	SUBSX WCZ
Compare	CMP WCZ	CMPX WCZ	CMPX WCZ	CMPSX WCZ

After a multi-long comparison:

- **Magnitude terminology:** IF_B (below), IF_AE (above/equal), IF_A (above), IF_BE (below/equal)

- **Arithmetic terminology:** IF_LT (less than), IF_GE (greater/equal), IF_GT (greater), IF_LE (less/equal)
- **Equality (either style):** IF_Z (equal), IF_NZ (not equal)

Both terminology styles encode to identical condition codes—choose whichever reads best for your code. The choice of **CMP** vs. CMPS (not the alias style) determines whether values are compared as unsigned or signed.

Key Concepts

- The C flag indicates carry, borrow, bit shifted out, or parity depending on instruction category
- The Z flag indicates a zero result or equality across nearly all instructions
- Flags persist until explicitly modified—instructions without WC/WZ/WCZ preserve flag values
- WC, WZ, and WCZ effects control which flags are updated; the operation always executes
- Special effects ANDC/ANDZ/ORC/ORZ/XORC/XORZ combine tested bits with existing flags (TESTx instructions only)
- Any instruction can be conditional using IF_x prefixes for deterministic branchless programming
- 16 conditions cover all combinations of C and Z states, with comparison-friendly aliases
- Conditional instructions consume 2 clock cycles whether they execute or not, maintaining deterministic timing
- Multi-precision arithmetic chains flag results between instructions using ADDX and SUBX
- Flag-based bit manipulation (MUXC, MUXZ) enables building bit patterns from sequential flag tests
- Each COG maintains independent C and Z flags with no cross-COG interaction

Chapter 4: Timing and Determinism

The P2 provides deterministic instruction timing, enabling precise real-time control. Understanding timing characteristics is essential for time-critical applications and optimizing code performance.

4.1 Clock Sources and Configuration

Before examining instruction timing, understanding clock configuration is essential—the system clock frequency determines all timing calculations. The P2 supports multiple clock sources, from simple internal oscillators to PLL-multiplied crystals running at 320 MHz.

4.1.1 Available Clock Sources

The P2 provides four clock source options, each suited to different application requirements:

RCFAST is the internal fast RC oscillator, running at 20 MHz or higher (nominally ~24 MHz, characterized 20-30 MHz across process, voltage, and temperature). This is the default clock source at power-on and reset. RCFAST requires no external components and provides immediate operation, though its frequency varies with temperature and process. Use RCFAST for applications where precise timing is not critical or as a bootstrap clock while configuring a more accurate source.

RCSLOW is the internal slow RC oscillator, running at approximately 20 kHz. This ultra-low-power clock serves sleep modes and real-time clock applications. RCSLOW frequency varies significantly with temperature (±50%), making it unsuitable for precision timing but ideal for power-sensitive applications.

Crystal oscillator mode connects an external crystal (typically 10-20 MHz) between the XI and XO pins. The P2 includes internal feedback resistors and programmable loading capacitors, simplifying crystal circuit design. Crystal sources provide the stability needed for precise timing, communication protocols, and frequency synthesis.

External clock mode accepts an external clock signal on the XI pin, supporting frequencies up to the device's rated system-clock maximum (180 MHz typical, 320 MHz extended per the spec sheet). Note that 350 MHz is the PLL overclock ceiling (VCO/1 mode, see §4.1.2), not the direct external-input range. This mode allows the P2 to synchronize with external timing sources or use specialized oscillators.

4.1.2 PLL Multiplication

The Phase-Locked Loop (PLL) multiplies a reference clock to achieve higher frequencies. The PLL takes the crystal or external clock as input and produces an output frequency according to three parameters:

- **Input divider** (1-64): Divides the reference frequency before the VCO
- **VCO multiplier** (1-1024): Multiplies to produce the VCO frequency
- **Post divider** (1, or 2-30 even): Divides the VCO output. Values 2, 4, 6…30 divide the VCO; value 1 passes VCO frequency directly (no division).

The output frequency follows the equation: f_out = (f_ref / input_div) × multiplier / post_div

For example, a 20 MHz crystal with input divider 1, multiplier 16, and post divider 2 produces: (20 MHz / 1) × 16 / 2 = 160 MHz.

The VCO operates optimally between 100-200 MHz for stability. For overclocking, the PLL can be pushed to 350 MHz using VCO/1 mode (%PPPP = 15), though stability becomes application-dependent.

4.1.3 The HUBSET Instruction

Note for Spin2 Programs: Spin2 programs typically configure the clock using CON constants (`_clkfreq`, `_xtlfreq`, `_xinfreq`). The compiler automatically generates the appropriate **HUBSET** calls at program initialization. Direct **HUBSET** use is primarily for: - Pure PASM2 programs without Spin2 - Dynamic clock changes at runtime - Advanced clock configurations not supported by CON constants

Clock configuration uses the **HUBSET** instruction with a 32-bit configuration value:

```
1         HUBSET  ##config_value          ' Configure clock system
```

The configuration value contains fields for clock source selection, crystal configuration, and PLL parameters. The full PLL-mode layout is `%0000_xxxE_DDDD_DDMM_MMMM_MMMM_PPPP_CCSS`:

Bits	Field	Purpose
1:0	SS	Source select (RCFAST/RCSLOW/crystal/PLL)
3:2	CC	Crystal configuration (XI/XO loading and feedback)
7:4	PPPP	Post divider (value → VCO/2..30; 15 = VCO/1)
17:8	MMMMMMMMMM	VCO multiplier (1..1024 = stored value + 1)
23:18	DDDDDD	XI input divider (1..64 = stored value + 1)
24	E	PLL enable
27:25, 31:28	-	Reserved (0)

4.1.4 Clock Switching Sequence

Switching clock sources requires a careful sequence to ensure glitch-free transitions:

1. **Enable the new source**: Configure crystal oscillator or PLL, but keep the current clock source active
2. **Wait for stabilization**: Crystal oscillators need approximately 10 ms to stabilize; PLL lock requires approximately 10 µs
3. **Switch sources**: Change the SS field to select the new clock source
4. **Optionally disable the old source**: Turn off unused oscillators to save power

```
1         HUBSET  ##%0000_0000_0000_0000_0000_0000_0001_0010  ' Enable xtal
2         WAITX   ##20_000_000/100                            ' Wait ~10ms
3         HUBSET  ##%0000_0000_0000_0000_0000_0000_0010_0010  ' Switch
```

The P2 provides automatic fallback to RCFAST if the selected clock source fails, preventing system lockup from clock problems.

4.1.5 Power Considerations

Clock frequency directly affects power consumption. Lower frequencies reduce power but also reduce performance. For battery-powered applications, consider:

- Use RCSLOW during sleep periods when only basic timekeeping is needed
- Disable the PLL when not required—it consumes power even when not selected
- Run at the lowest frequency that meets timing requirements
- Stop unused COGs to eliminate their clock-related power consumption

4.2 Instruction Timing

4.2.1 The System Clock

The P2 operates from a system clock that can run up to 320 MHz. All instruction execution, memory access, and I/O operations occur in relation to this master clock. The clock source can be an internal RC oscillator for standalone operation, an external crystal for precision timing, or a PLL-multiplied clock for maximum performance.

Every timing measurement in the P2 is expressed in clock cycles. At 320 MHz, one clock cycle represents 3.125 nanoseconds. This means that a two-cycle instruction completes in 6.25 nanoseconds—fast enough for demanding real-time applications like video generation, high-speed communication protocols, and precision motor control.

Understanding cycle counts is fundamental to P2 programming because the processor provides cycle-accurate timing guarantees. When a program executes the same instruction sequence under the same conditions, it takes exactly the same number of clock cycles every time. This determinism distinguishes the P2 from processors with caches, speculative execution, or variable-latency memory systems.

4.2.2 Instruction Cycle Counts

Most COG instructions execute in exactly 2 clock cycles. This consistency simplifies timing calculations and makes hand-optimized assembly code practical. The processor can execute one instruction per two-cycle period, achieving an effective instruction rate of 160 million instructions per second at 320 MHz.

The following table shows typical cycle counts for different instruction categories:

Instruction Type	**Typical Cycles**
Register-to-register ALU	2
Immediate ALU	2
Branches (not taken)	2
Branches (taken)	4
Hub access	2-16+
CORDIC operations	2…9 (start), 55 (wait)

Register operations like ADD, **SUB**, AND, and OR complete in 2 cycles whether they operate on registers or immediate values. This uniformity means that choosing between a register operand and an immediate operand has no performance impact—the decision is purely about code clarity and register pressure.

Branch instructions take 2 cycles when the branch is not taken and 4 cycles when taken. This predictable variation allows precise timing of both paths through conditional code. Programmers can eliminate this variation entirely by using conditional execution instead of branches.

Hub memory access instructions have variable timing because they must wait for the COG's hub access window. The base instruction time is 2 cycles, but the wait for hub access **ADDS** 0 to 7 additional cycles depending on when the instruction executes relative to the hub rotation pattern.

CORDIC operations use a two-phase execution model. The instruction that starts a CORDIC operation (like **QMUL** for multiplication) completes in 2 clocks when the COG's hub slot is current, and up to 9 clocks (2 base + up to 7 slot-wait, on an 8-COG P2) when it must wait for its hub slot. The result is not available until 55 clocks after the operation starts. Programs can perform other work during this 55-clock computation period and retrieve the result later with **GETQX** or **GETQY**.

4.2.3 Reading Cycle Counts

The instruction encoding table in the P2 documentation provides precise cycle counts in its Clocks column. Understanding the notation used in this column is essential for accurate timing analysis:

Notation	Meaning
2	Always 2 cycles
2+	Minimum 2 cycles, may be more
2 or 4	2 if not taken, 4 if taken
4 (cog) / 13-20 (hub-exec)	Taken branch: COG mode / hub-execution mode
4…11	Variable range (bounded)

A simple "2" means the instruction always takes exactly 2 cycles regardless of operands or conditions. This applies to most arithmetic, logical, and data movement instructions.

The "2+" notation indicates a base time of 2 cycles plus additional variable time, where the "+" represents an instruction-specific variable delay. Hub data-access instructions are instead documented with an explicit range—RDLONG, for example, is listed as "9…16" (cog mode), the variation being the hub-window slot-wait.

Branch instructions show "2 or 4" to reflect their dual timing behavior. When the branch condition is false, the processor continues to the next instruction in 2 cycles. When the condition is true, the processor loads a new program counter and takes 4 cycles total.

The slash notation—shown here as "4 (cog) / 13-20 (hub-exec)"—separates COG-execution timing (left) from hub-execution timing (right) for instructions whose timing genuinely differs between the two modes. This is **not** a per-instruction fetch penalty. In hub-execution mode the prefetch FIFO streams sequential instructions ahead of execution, so straight-line code runs at the same 2 cycles per instruction as COG mode (see §4.8). The two modes diverge at **taken branches**: hub execution must refill the FIFO from the new address, costing a minimum of 13 clocks (13-20 including the refill's hub-window wait) versus 4 clocks in COG mode. Hub data-access instructions (RDLONG and friends) likewise carry a two-mode entry because the access pipeline is longer in hub-execution mode. **CALLA**/CALLB and **RETA**/RETB are other instructions documented with a `cog / hub-exec` pair.

Variable range notation like "4…11" indicates that execution time varies within fixed bounds depending on the processor state when the instruction runs. **LOCKNEW**, for example, is listed as "4…11" clocks: the hub's shared locks are a hub resource, so allocating one is serviced by the hub and the exact cycle count depends on where the COG sits in the hub rotation at that moment. (By contrast **REP**, despite governing a repeated block, is itself a fixed 2-cycle instruction—it only loads the hardware repeat counter; the variable time is spent in the repeated instructions, not in **REP**.)

4.3 Hub Access Timing

4.3.1 Hub Access Rotation

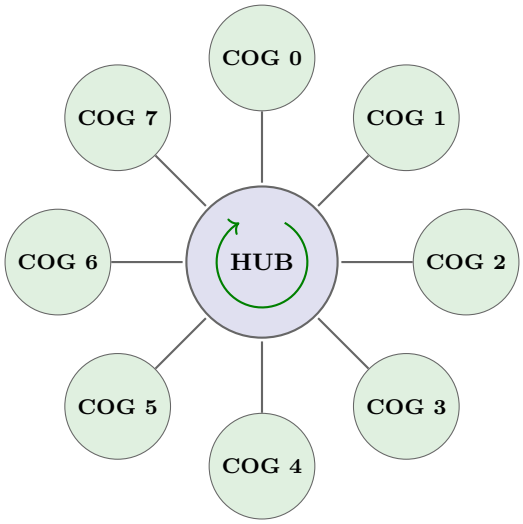


Figure 4.1. Figure 4.1: Hub Access Rotation ("Egg Beater")

Hub memory access uses round-robin arbitration that gives each COG fair access to the shared hub RAM. This rotating pattern is commonly called the "egg beater" due to its visual similarity to rotating blades, with each COG's access window spinning through the sequence in turn.

The hub controller divides time into eight-cycle periods. Within each period, every COG gets exactly one cycle to access hub memory. The access windows rotate continuously through COGs 0, 1, 2, 3, 4, 5, 6, 7, then back to COG 0, repeating this pattern indefinitely. This rotation never stops and never changes—it runs continuously from the moment the chip powers on.

When a COG executes an instruction that accesses hub memory (RDLONG, **WRLONG**, **RDWORD**, **WRWORD**, **RDBYTE**, or **WRBYTE**), the instruction waits until that COG's window arrives, performs the memory access during the window, then completes. The wait time depends on when the instruction executes relative to the rotation pattern.

This deterministic rotation means hub access timing is predictable. While the wait time varies from 0 to 7 cycles, the variation follows a fixed pattern. A program that knows its phase relationship to the hub rotation can achieve minimum wait times by scheduling hub access to align with its windows.

4.3.2 Hub Access Latency

When a COG executes a hub instruction, the actual wait time depends on timing relative to the hub rotation. Three scenarios illustrate the range of possibilities:

Best case: The instruction executes just as the COG's hub window arrives, with zero slot-wait. A standalone **RDLONG** in COG mode then completes in 9 clocks total. The 0-cycle figure is only the slot-wait *component*; the 9-clock floor reflects the hub-access pipeline (FIFO arbitration plus read latency), which a simple "2 base + 1 access" model omits.

Worst case: The instruction executes just after the COG's hub window has passed. The instruction must wait for the rotation to complete—seven more COGs take their turns before this COG's window comes around again. This **ADDS** 7 cycles of slot-wait, for 16 clocks total (a standalone **RDLONG** in COG mode). In hub-execution mode the same access ranges 9…26 clocks.

Average case: On average, an instruction that executes at a random time relative to the hub rotation waits 3.5 cycles of slot-wait for its hub window, landing mid-range in the 9…16 span. This average assumes no deliberate scheduling to align with windows.

The hub access latency directly impacts program performance when hub memory access is frequent. Programs that minimize hub access (by keeping frequently-accessed data in COG registers or COG RAM) avoid this latency. Programs that must access hub memory frequently achieve better performance by organizing hub access into bursts, which amortize the window wait time across multiple memory transfers.

4.3.3 Hub Burst Transfers

SETQ enables burst transfers that read or write multiple consecutive longs in a single hub access sequence. This feature dramatically improves hub memory throughput by eliminating the window wait time for all but the first transfer.

The **SETQ** instruction takes one parameter specifying how many additional longs to transfer. The hub access instruction that follows **SETQ** performs a burst of that many consecutive transfers:

```
1         SETQ    #15                    ' Transfer 16 longs total
2         RDLONG  buffer, ptr            ' Burst read from Hub
```

This code reads 16 consecutive longs from hub memory starting at address `ptr` and stores them in COG RAM starting at address `buffer`. The first **LONG** experiences the normal hub access (9…16 clocks, including its slot-wait), but each subsequent **LONG** transfers in just one additional cycle. The whole burst completes in roughly 2 (SETQ) + 9…16 (first **RDLONG**) + 15 (subsequent longs) 26-33 cycles—far faster than 16 separate **RDLONG** instructions, each of which costs 9…16 clocks for a total on the order of 144-256 clocks (nominally ~10-12 each).

Burst transfers work because once a COG has started transferring data during its hub window, it can continue occupying subsequent windows in the rotation. The hub controller grants consecutive windows to a COG performing a burst, allowing continuous transfers without interruption.

SETQ affects only the next hub instruction. If that instruction is not a hub access instruction, **SETQ** has no effect (some non-hub instructions use **SETQ** for other purposes). After the hub instruction completes, **SETQ** must be reissued to enable another burst.

4.3.4 FIFO Operations

The P2 includes a hardware FIFO (First In, First Out) buffer that provides the highest-bandwidth method for sequential hub data transfer. Unlike individual hub access instructions that wait for hub windows, the FIFO continuously moves data between hub memory and the COG in the background. The hardware prefetches data before the COG needs it (for reads) or buffers data until hub windows become available (for writes), hiding hub access latency from the program.

FIFO Architecture:

Each COG has access to a shared FIFO buffer that can operate in either read mode or write mode (not both simultaneously). The FIFO contains (cogs+11) stages—with all 8 COGs active, this provides 19 stages of buffering. When in read mode, the FIFO loads continuously whenever fewer than (cogs+7) stages are filled, after which up to 5 more longs may stream in, potentially filling all stages. These metrics ensure the FIFO never underflows under any reading scenario.

Setting Up the Read FIFO:

RDFAST configures the FIFO for reading from hub memory. The D operand provides a block count (number of 64-byte blocks before wrapping), and the S operand provides the starting hub address:

```
1         RDFAST  #0, ptr                ' Start continuous read FIFO
2 loop
3         RFLONG  data                   ' Read from FIFO (fast, no hub wait)
4         ' ... process data ...
5         JMP     #loop                  ' Continue reading
```

The **RFLONG**, **RFWORD**, and **RFBYTE** instructions read from the FIFO without waiting for hub windows—if data is available in the FIFO buffer, the read completes immediately. The FIFO refills automatically in the background using whatever hub windows become available.

Wait Mode vs. No-Wait Mode:

RDFAST and **WRFAST** each have two modes controlled by bit 31 of the D operand:

D[31]	Behavior
0	Wait for any previous WRFAST to finish, then reconfigure FIFO. For RDFAST, also wait until FIFO begins receiving data. Ready to use immediately after instruction completes.
1	No-wait mode—takes only 2 clocks. Code must allow sufficient time before accessing FIFO data.

The no-wait mode is useful when you need to reconfigure the FIFO quickly and can guarantee enough cycles will pass before the first FIFO access.

Setting Up the Write FIFO:

WRFAST configures the FIFO for writing to hub memory:

```
        WRFAST  #0, ptr                 ' Start continuous write FIFO
loop
        ' ... generate data ...
        WFLONG  data                    ' Write to FIFO (fast, no hub wait)
        JMP     #loop                   ' Continue writing
```

The **WFLONG**, **WFWORD**, and **WFBYTE** instructions write to the FIFO buffer. If buffer space is available, the write completes immediately without waiting for a hub window. The FIFO drains to hub memory automatically.

Important: If a COG has been writing to hub via **WRFAST** and wants to immediately **COGSTOP** itself, execute `WAITX #20` first to allow time for any lingering FIFO data to be written to hub memory.

Circular Buffer Mode:

The FIFO supports circular buffer operation for continuous streaming. When configured with a non-zero block count, the FIFO wraps back to the starting address after transferring the specified number of 64-byte blocks:

```
        RDFAST  #16, audio_buffer       ' Read 16 blocks (1KB), then wrap
```

For wrapping mode, the hub start address must be long-aligned (address ends in %00) since there won't be an extra cycle to read/write a partial **LONG** at block boundaries. Use 0 for block count when you don't want wrapping—the FIFO will sequence through the entire 1MB hub map before wrapping.

Dynamic Buffer Management with FBLOCK:

The **FBLOCK** instruction provides dynamic control over the FIFO's wrap behavior. It sets a new start address and block count that take effect when the current blocks are fully read or written:

```
        RDFAST  #16, buffer_a           ' Start reading from buffer A
        ' ... reading proceeds ...
        FBLOCK  #16, buffer_b           ' Queue buffer B for when A done
        ' ... FIFO seamlessly transitions to buffer B on wrap
```

FBLOCK can be executed after **RDFAST**, **WRFAST**, or a FIFO block wrap event. Coordinating **FBLOCK** with streamer activity enables dynamic, seamless streaming between hub RAM and pins/DACs—essential for continuous audio/video output where buffer switches must be glitch-free.

Variable-Length Data: RFVAR and RFVARS:

For bytecode interpreters and compact data formats, **RFVAR** and **RFVARS** read 1-4 bytes of variable-length encoded data from the FIFO. The encoding uses the MSB of each byte to indicate whether more bytes follow:

First Byte	Additional Bytes	RFVAR Returns	RFVARS Returns
%0xxxxxxx	none	7-bit value, zero-extended	7-bit value, sign-extended
%1xxxxxxx	1 more (%0xxxxxxx)	14-bit value, zero-extended	14-bit value, sign-extended
%1xxxxxxx	2 more	21-bit value, zero-extended	21-bit value, sign-extended
%1xxxxxxx	3 more	28-bit value, zero-extended	28-bit value, sign-extended

This encoding provides memory-efficient storage for bytecode constants and offset addresses—small values use 1 **BYTE**, larger values expand as needed. **RFVAR** returns unsigned (zero-extended) values; **RFVARS** returns signed (sign-extended) values.

FIFO Events:

The FIFO generates events that programs can monitor for buffer management:

- **EVENT_FBW** (FIFO Block Wrap) signals when the FIFO wraps around in circular buffer mode. Programs use this event to know when to refill the next section of a circular buffer or to synchronize with buffer boundaries.

Programs can wait for this event using WAITSE or poll it using POLLSE after configuring a selectable event source. This enables efficient ping-pong buffering where one COG fills buffers while another consumes them.

Hub Execution Restriction:

The FIFO cannot be used while the COG is executing from hub RAM. During hub execution mode, the FIFO hardware is dedicated to spooling instructions, so these instructions cannot be used:

- **RDFAST** / **WRFAST** / **FBLOCK**
- **RFBYTE** / **RFWORD** / **RFLONG** / **RFVAR** / **RFVARS**
- **WFBYTE** / **WFWORD** / **WFLONG**
- **XINIT** / **XZERO** / **XCONT** (when streamer mode engages the FIFO)

To use FIFO operations, ensure your code executes from COG or LUT RAM.

FIFO and the Streamer:

The Streamer subsystem (described in Chapter 5) uses the FIFO for high-bandwidth data transfer to and from I/O pins. When the Streamer is active, it shares the FIFO with FIFO access instructions. **RDFAST**/WRFAST configure the FIFO source or destination in hub memory; the Streamer then moves data between the FIFO and pins at rates matching the system clock. This combination enables video generation, audio streaming, and high-speed data acquisition without per-sample CPU intervention.

Performance Considerations:

FIFO access provides near-instantaneous data transfer from the program's perspective—no hub window waiting, no variable latency. However, the FIFO has finite depth. If a program reads faster than the FIFO can refill (or writes faster than it can drain), the FIFO stalls waiting for hub access. For sustained maximum throughput, balance data production/consumption rate with the hub's aggregate bandwidth.

The FIFO access instructions (RFLONG, **RFWORD**, **RFBYTE**, **WFLONG**, **WFWORD**, **WFBYTE**) complete in 2 cycles when the FIFO has data available or space available, respectively. This makes FIFO access ideal for streaming applications: video pixel generation, audio sample processing, high-speed communication protocols, and bulk data movement.

4.4 Deterministic Timing

4.4.1 What Determinism Means

The P2's deterministic timing guarantees that the same instruction sequence, executing under the same conditions, takes exactly the same number of clock cycles every time it runs. This guarantee holds across all executions—there are no cache misses, no speculative execution failures, no memory controller delays, and no unpredictable pipeline stalls.

Determinism provides several critical benefits for embedded systems programming:

Predictable performance: When a routine takes 1,000 cycles during testing, it takes 1,000 cycles in production. Performance measurements made during development remain accurate in the deployed system.

Reliable timing: Real-time systems can meet hard timing deadlines because worst-case execution time equals actual execution time. If an interrupt handler must complete within 500 cycles, testing that it does so once proves it always will.

Reproducible behavior: Timing-related bugs are reproducible because timing is consistent. A race condition that appears during development will appear in the same way in production, making debugging practical.

Simplified analysis: Programmers can calculate execution time by hand, adding up cycle counts from the instruction table. This makes optimization straightforward—identify the critical path, count cycles, improve the slow parts.

The P2 achieves determinism through architectural choices: no instruction cache (COG RAM provides fast local storage without cache complexity), no data cache (hub access uses predictable round-robin scheduling), no branch prediction (conditional execution eliminates branches), and no speculative execution (instructions execute in program order).

4.4.2 Sources of Timing Variation

While the P2 provides deterministic timing, four sources of variation exist. These variations are predictable and controllable, not random like cache misses or memory arbitration in complex processors:

Source	Variation	Mitigation
Hub access wait	0-7 cycles	Loop alignment, careful scheduling
Branches	2 vs 4 cycles	Conditional execution instead
CORDIC wait	Up to 55 clocks	Interleave other work
WAITX	Variable	Intentional delays

Hub access wait varies from 0 to 7 cycles depending on when a hub instruction executes relative to the hub rotation. This variation is deterministic—if a program executes a hub instruction at the same point in the rotation cycle, the wait time is identical. Programs can eliminate this variation by scheduling hub access to occur at aligned points in loops, ensuring the loop body is a multiple of 8 cycles so hub access always occurs at the same phase of the rotation.

Branch timing varies because taken branches require 4 cycles while not-taken branches require only 2 cycles. This variation is completely predictable—the same branch decision always takes the same time. Programs can

eliminate this variation by using conditional execution instead of branches, trading the variable 2-or-4-cycle branch for a fixed 2-cycle conditional instruction.

CORDIC wait varies because different CORDIC operations take different amounts of time to compute. Multiplication, division, square root, and trigonometric functions each have specific completion times. The variation is deterministic—the same operation always takes the same time. Programs hide CORDIC latency by issuing the operation early and performing other work during the computation period.

WAITX provides intentional variable delay. This is the only case where variation is desired rather than avoided—WAITX exists specifically to introduce precise, controlled timing delays for applications like bit-banging protocols or pulse generation.

4.4.3 Eliminating Branches

Conditional execution provides an alternative to branching that eliminates timing variation. Instead of using a compare instruction followed by a conditional jump, code can use a compare instruction followed by conditionally-executed instructions.

The branching approach introduces timing variation:

```
' With branch (2 or 4 cycles):
        CMP     a, b            wz
        if_z    JMP     #equal_case
        ' Not-equal path continues here
```

When `a` equals `b`, this code takes 2 (CMP) + 4 (JMP taken) = 6 cycles. When `a` differs from `b`, the code takes 2 (CMP) + 2 (JMP not taken) = 4 cycles. The 2-cycle variation complicates timing analysis.

The conditional execution approach provides constant timing:

```
' Without branch (2 cycles always):
        CMP     a, b            wz
        if_z    MOV     result, #1
        if_nz   MOV     result, #0
```

This code takes 2 (CMP) + 2 (first **MOV**, executed if Z set) + 2 (second **MOV**, executed if Z clear) = 6 cycles when Z is set, or 2 (CMP) + 2 (first **MOV**, skipped) + 2 (second **MOV**, executed) = 6 cycles when Z is clear. Both paths take exactly 6 cycles.

The key insight is that conditionally-skipped instructions still consume their execution time slot—the processor evaluates the condition and skips the instruction's effect, but the instruction still occupies 2 cycles. This behavior ensures that all execution paths through conditionally-executed code take the same time.

Conditional execution works for simple cases where both branches are short. For longer code sequences or cases where only one branch performs work, traditional branching may be more efficient despite the timing variation. The choice depends on whether consistent timing or shorter average time is more important for the specific application.

4.5 Synchronization

4.5.1 WAITX - Precise Delays

WAITX provides precise, cycle-accurate delays by pausing execution for a specified number of clock cycles:

```
1         WAITX   ##100                  ' Wait exactly 100 cycles
```

The instruction accepts a value specifying the delay duration. Execution resumes exactly after that many cycles have elapsed. This precision makes **WAITX** essential for timing-critical operations like bit-banging communication protocols, generating precise pulse widths, or synchronizing with external events.

WAITX delays are relative to when the instruction executes. If a program needs to generate a pulse every 1,000 cycles, using **WAITX** alone accumulates timing drift because the **WAITX** instruction itself consumes time, and the instructions between **WAITX** calls add additional cycles. For precise periodic timing without drift, the counter-based wait instructions provide better alternatives.

4.5.2 Counter-Based Waiting

The P2 provides a global cycle counter that increments every clock cycle. COGs can read this counter with **GETCT** and wait for specific counter values using the WAITCT family of instructions. This mechanism enables drift-free periodic timing.

Each COG has three independent counter match registers (CT1, CT2, CT3). Programs load target counter values into these registers using ADDCT1, ADDCT2, or ADDCT3, then wait for the counter to reach those values using WAITCT1, WAITCT2, or WAITCT3:

```
1         GETCT   time                   ' Read current time
2         ADDCT1  time, ##1000           ' Set CT1 = time + 1000
3         ' ... do work ...
4         WAITCT1                        ' Wait until counter reaches CT1
```

This pattern ensures that the wait completes exactly 1,000 cycles after the **GETCT** instruction, regardless of how long the intervening work takes. If the work completes in 800 cycles, WAITCT1 waits 200 more cycles. If the work takes 1,200 cycles, WAITCT1 returns immediately (the deadline has already passed).

For periodic operations, adding a fixed delta to the counter match register each iteration eliminates drift:

```
1         GETCT   time                   ' Initialize time base
2 loop
3         ADDCT1  time, ##1000           ' Next deadline = previous + 1000
4         ' ... generate pulse or process data ...
5         WAITCT1                        ' Wait for next period
6         JMP     #loop
```

Each iteration runs exactly 1,000 cycles from the previous iteration, maintaining perfect periodicity regardless of small variations in the work performed each cycle.

4.5.3 Pin-Based Synchronization

Several instructions synchronize with pin state changes, enabling precise timing relative to external events:

WAITATN waits for any pin to make a low-to-high transition (attention flag). Smart Pins can be configured to set their ATN flags on specific conditions, making **WAITATN** useful for waiting on external events with minimal COG overhead.

WAITSE1, WAITSE2, WAITSE3, WAITSE4 wait for selectable events SE1-SE4. Each is configured via the corresponding SETSE1-SETSE4 instruction to fire on a chosen source—a pin edge or level, a LUT-address access, or a hub-lock event. A selected event can also be polled (POLLSE1-4), branched on (JSE/JNSE), or used as an interrupt source. (Streamer-driven activity such as a FIFO block-wrap is observed by routing it through one of these selectable event sources, not by a streamer-specific wait.)

WAITPAT waits for a pin pattern match. Programs configure a pattern and mask, then **WAITPAT** suspends execution until the pin states match the specified pattern. This enables synchronization with parallel interfaces or detection of specific pin combinations.

POLLATN, POLLCT1, POLLCT2, POLLCT3 provide polling-based alternatives to waiting. Instead of blocking until a condition occurs, these instructions check whether an event has occurred and set flags accordingly. This allows code to perform useful work while watching for events, rather than waiting idly.

4.6 Timing-Critical Patterns

4.6.1 Cycle-Exact Loops

Many real-time applications require loops that execute with precise, predictable timing. The P2's deterministic instruction timing makes cycle-exact loops practical and reliable.

Consider a loop that reads data from hub memory, processes it, and repeats:

```
1 loop
2         RDLONG  data, ptr               ' 9...16 cycles (hub-window dep.)
3         ADD     ptr, #4                 ' 2 cycles
4         DJNZ    count, #loop            ' 4 cycles (taken)
```

This loop body must account for hub access timing variation. If the loop starts aligned with the COG's hub window, **RDLONG** incurs 0 slot-wait (9 cycles) and the loop takes 9 + 2 + 4 = 15 cycles. If the loop starts just after the hub window, **RDLONG** incurs 7 cycles of slot-wait (16 cycles) and the loop takes 16 + 2 + 4 = 22 cycles.

For truly cycle-exact timing, loops must either eliminate hub access or align hub access with the hub rotation. One approach uses COG RAM for all data, avoiding hub access entirely:

```
1 loop
2         ADD     data, #1                ' 2 cycles
3         DJNZ    count, #loop            ' 4 cycles (taken)
4         ' Exactly 6 cycles per iteration
```

Another approach aligns the loop body to an 8-cycle boundary and ensures hub access occurs at the same phase each iteration:

```
loop
        RDLONG  data, ptr               ' 9...16; settles to 14 once aligned
        ADD     result, data            ' 2 cycles
        ADD     ptr, #4                 ' 2 cycles
        DJNZ    count, #loop            ' 4 cycles (taken)
        NOP                             ' 2 cycles - padding
        ' Loop body = 24 cycles (3× hub period)
```

Once the loop stabilizes—after its first iteration—RDLONG sees a constant 5 cycles of slot-wait every pass, because the 24-cycle loop body is a whole multiple of the 8-cycle hub period and so re-presents **RDLONG** to the rotation at the same phase each time. Every iteration after the first then takes exactly 24 cycles. (Determinism here does not actually depend on the padding **NOP**: a loop containing a single hub access always self-aligns to the next multiple of the hub period after one iteration—the **NOP** only shifts the constant slot-wait, in this case from 7 cycles down to 5.)

4.6.2 Pipelined Hub Access

Programs can hide hub access latency by overlapping computation with hub waiting. Instead of waiting for one hub operation to complete before starting the next computation, a program can issue a hub access and immediately begin computing with data already available, allowing the hub access to proceed in parallel.

The **SETQ**-based burst transfer provides one form of pipelining—while later longs transfer, the program can begin processing earlier longs. A more general approach separates hub access from computation:

```
loop
        RDLONG  next_data, next_ptr     ' Start fetching next data
        ADD     next_ptr, #4
        ' Process current_data while hub fetch proceeds
        ADD     result, current_data
        SUB     current_data, offset
        MOV     current_data, next_data ' Previous fetch is now ready
        DJNZ    count, #loop
```

This pattern keeps hub access and computation overlapped—the **RDLONG** for iteration N+1 occurs while iteration N's computation proceeds. The technique works best when computation time roughly equals hub access time, maximizing overlap.

4.6.3 CORDIC Pipelining

CORDIC operations take 55 clocks to compute results, but the instruction that starts a CORDIC operation completes in just 2…9 clocks (2 when the COG's hub slot is current, up to 9 when it must wait for its hub slot). This creates an opportunity for pipelining: start a CORDIC operation, perform other work during the 55-clock computation period, then retrieve the result.

A simple example shows the pattern:

```
1         QMUL    a, b                    ' Start multiply
2         ' ... 55 clocks of other work ...
3         GETQX   result                  ' Get result (low 32 bits)
```

For maximum efficiency, interleave multiple CORDIC operations with other work:

```
1         QMUL    a1, b1                  ' Start first multiply
2         ' ... some work ...
3         QMUL    a2, b2                  ' Start second multiply
4         ' ... more work ...
5         GETQX   result1                 ' Get first result
6         ' ... more work ...
7         GETQX   result2                 ' Get second result
```

GETQX returns in 2 clocks if the CORDIC result is already available (or the CORDIC is empty); otherwise it automatically stalls the COG until the result is ready—it never returns a partial result (worst case approaching the 55-clock latency). To test readiness without stalling, poll the CORDIC-empty (QMT) event rather than calling **GETQX** blindly. If **GETQX** executes later than the result, the result remains available—CORDIC results persist until the next CORDIC operation starts.

Multiple CORDIC operations can be in flight simultaneously, with results retrieved in order. Starting a new CORDIC operation does not invalidate results from previous operations until their results have been read.

4.6.4 Deterministic I/O

Bit-banging—directly controlling I/O pins with software timing—requires cycle-accurate execution. The P2's deterministic timing makes bit-banging practical for protocols like WS2812 LED control, custom serial formats, or precise pulse generation.

A WS2812 LED protocol example demonstrates the precision required:

```
1 ' WS2812 requires precise pulse widths:
2 ' 0 bit: 400ns high, 850ns low
3 ' 1 bit: 800ns high, 450ns low
4 ' At 200 MHz (5ns per cycle):
5 ' 0 bit: 80 cycles high, 170 cycles low
6 ' 1 bit: 160 cycles high, 90 cycles low
7
```

continues on next page →

↪ *continued from previous page*

```
send_bit
        TEST    data, #31       wc      ' Get high bit into C flag
        DRVH    pin                     ' Start pulse (high)
        if_c    WAITX   ##160           ' 1-bit: wait 160 cycles
        if_nc   WAITX   ##80            ' 0-bit: wait 80 cycles
        DRVL    pin                     ' End pulse (low)
        if_c    WAITX   ##90            ' 1-bit: wait 90 cycles
        if_nc   WAITX   ##170           ' 0-bit: wait 170 cycles
        ROL     data, #1                ' Shift to next bit
        DJNZ    count, #send_bit
```

This code generates precise pulse widths using **WAITX** for delays and conditional execution to avoid branch timing variation. The **DRVH** and **DRVL** instructions change pin states, and the **WAITX** instructions maintain exact timing between transitions.

Deterministic timing eliminates the jitter and uncertainty common in systems with caches or interrupts. Each pulse width is exactly the specified duration, enabling reliable communication with timing-sensitive devices.

4.7 Measuring Execution Time

4.7.1 The Cycle Counter

The P2 provides a global 64-bit cycle counter (Rev B/C silicon) that increments every clock cycle. This counter runs continuously from power-on. COGs read the counter using the **GETCT** instruction, which returns the lower 32 bits by default. The lower 32 bits wrap around after reaching their maximum value.

Measuring code execution time involves reading the counter before and after the code section of interest:

```
        GETCT   start_time              ' Read cycle counter
        ' ... code to measure ...
        GETCT   end_time                ' Read cycle counter again
        SUB     end_time, start_time    ' Elapsed cycles
```

The difference between the two readings gives the exact number of cycles elapsed. This measurement includes the cycles consumed by **GETCT** itself (2 cycles each), so precise measurements should account for this overhead.

For short code sequences, the measurement overhead matters. Measuring a 10-cycle sequence with two **GETCT** instructions reports 14 cycles (2 + 10 + 2). For longer sequences, the 4-cycle overhead becomes negligible.

The cycle counter is global across all COGs—all COGs read the same counter value. This enables synchronization and coordination between COGs. One COG can mark a time value and pass it to another COG via hub memory, allowing the second COG to measure time relative to events in the first COG.

4.7.2 Counter Wrap-Around

The lower 32 bits of the cycle counter wrap around every 2^{32} cycles. At 320 MHz, this occurs every 13.4 seconds. Code that measures elapsed time using the lower 32 bits must handle wrap-around correctly.

Subtraction using unsigned arithmetic naturally handles wrap-around. When end_time is less than start_time (because wrap-around occurred), the subtraction `end_time - start_time` produces the correct elapsed time due to modular arithmetic:

```
        MOV     start_time, ##$FFFF_FFF0  ' Near wrap-around
        MOV     end_time,   ##$0000_0010  ' After wrap-around
        SUB     end_time, start_time      ' Result: $20 (32 cycles)
```

This automatic wrap-around handling works for elapsed times up to 2^{31} cycles (half the counter range). For longer measurements, code must count wrap-around events explicitly or use multiple counter values.

4.7.3 Profiling Techniques

GETCT enables detailed performance profiling of assembly code. By measuring execution time for different code paths, programmers can identify performance bottlenecks and verify that optimizations achieve expected speedups.

A common profiling pattern measures loop iteration time:

```
        MOV     iterations, ##1000
        GETCT   start_time
loop
        ' ... code to profile ...
        DJNZ    iterations, #loop
        GETCT   end_time
        MOV     elapsed, end_time
        SUB     elapsed, start_time
```

The total elapsed time divided by the iteration count gives the average time per iteration. For more detailed profiling, place multiple **GETCT** measurements within the loop to identify which parts of the loop consume the most time:

```
loop
        GETCT   time1
        ' ... section A ...
        GETCT   time2
        ' ... section B ...
        GETCT   time3
        MOV     timeA, time2
        SUB     timeA, time1              ' Section A timing
        MOV     timeB, time3
```

continues on next page →

↪ continued from previous page

```
10          SUB     timeB, time2              ' Section B timing
11          ' Store or accumulate timing data
12          DJNZ    iterations, #loop
```

This approach provides cycle-accurate timing for each code section, enabling precise optimization. The overhead of **GETCT** instructions affects absolute timing but not the relative timing between sections.

Profiling can reveal unexpected timing variations. If a loop shows inconsistent timing across iterations, the variation likely comes from hub access timing, branch behavior, or CORDIC latency. Identifying these variations guides optimization efforts toward the actual bottlenecks rather than presumed slow code.

4.8 COG vs Hub Execution Mode Timing

4.8.1 COG Execution Mode

COG execution mode—often called "COG mode"—executes instructions from the COG's local 512-long (2KB) RAM. This provides the fastest possible execution because instruction fetch occurs from the COG's private memory without any shared resource contention.

In COG mode, most instructions complete in exactly 2 clock cycles. The processor fetches an instruction and executes it without waiting for memory access arbitration, cache lookups, or bus conflicts. This predictable timing makes COG mode ideal for timing-critical code like interrupt handlers, real-time control loops, and I/O bit-banging.

COG mode execution begins when a COG starts via **COGINIT** with a COG RAM address (0-$1FF). The program counter points to COG RAM locations, **AND** instruction fetch proceeds at full speed. All 512 longs of COG RAM are available for code and data, though programs typically reserve some locations for data and use the remainder for code.

The limitation of COG mode is size—only 512 longs of code and data combined. Programs that need more code space must use hub execution mode or carefully manage code overlays.

4.8.2 Hub Execution Mode

Hub execution mode—often called "HUBEXEC mode"—executes instructions from hub RAM. This allows programs to exceed the 512-long COG RAM size limit, supporting much larger code bases at the cost of a branch-refill penalty (sequential throughput is unchanged).

In hub execution mode, sequential straight-line code executes at 2 cycles per instruction—identical throughput to COG mode. The (cogs+11) = 19-stage prefetch FIFO streams instructions ahead of execution, hiding hub latency so there is no per-instruction hub-window wait. The only **HUBEXEC** penalty occurs at branches: a taken branch forces a FIFO refill, costing a minimum of 13 clocks (one more if the target is not long-aligned), versus 4 clocks for a COG-mode branch.

Hub execution mode begins when a COG starts via **COGINIT** with a hub RAM address ($400 or higher). The program counter points to hub RAM locations, and the processor fetches instructions through the FIFO prefetch mechanism. Code can utilize the full 512 KB of hub RAM.

Despite the branch-refill penalty, hub mode remains useful for several scenarios:

Large programs: When code exceeds 512 longs, hub mode is the only option short of implementing code overlays.

Non-critical code: Initialization routines, background tasks, and other code without tight timing requirements run acceptably in hub mode.

Mixed execution: Programs can start in hub mode and copy time-critical sections to COG RAM for execution at full speed. **COGINIT** can switch a running COG between hub and COG mode dynamically.

4.8.3 Timing Comparison

The following table shows typical execution times for common operations in both execution modes:

Operation	COG Mode	Hub Mode
Simple ALU	2 cycles	2 cycles
Branch taken	4 cycles	min 13 cycles (+1 if target not long-aligned)
Hub access	2 + hub wait	2 + hub wait
CORDIC start	2…9 clocks	2…9 clocks

Simple ALU operations (ADD, **SUB**, AND, OR, etc.) take 2 cycles in both modes. In sequential straight-line code the FIFO prefetches instructions ahead of execution, so **HUBEXEC** instruction fetch **ADDS** no per-instruction hub-window wait—throughput matches COG mode.

Branch instructions take 4 cycles in COG mode when taken. In hub mode, a taken branch forces the prefetch FIFO to refill from the new address, costing a minimum of 13 clocks (one more if the target is not long-aligned). This branch-refill penalty—not per-instruction fetch—is where **HUBEXEC** loses time relative to COG mode.

Hub access instructions show essentially the same data-access timing in both modes because the data access (as opposed to instruction fetch) uses the hub window mechanism regardless of where the instruction itself came from. A **RDLONG** takes 9…16 clocks in COG mode (9…26 in hub-execution mode), the variation being the hub-window slot-wait.

CORDIC operations start in 2…9 clocks in both modes (the slot-wait component reflects waiting for the COG's hub slot, **NOT** instruction fetch; the 55-clock computation time is the same in both modes). The CORDIC-issue instruction is sequential and streamed by the FIFO, so it incurs no extra **HUBEXEC** fetch penalty.

Because branch-heavy code pays the FIFO-refill penalty on every taken branch, COG mode remains strongly preferred for timing-critical, tightly-looped code. Programs typically keep inner loops, interrupt handlers, and time-sensitive operations in COG RAM while using hub mode for larger, less-critical code sections.

Key Concepts

- System clock configurable from 20 kHz (RCSLOW) to 320 MHz (PLL) via HUBSET
- Most COG instructions execute in exactly 2 clock cycles
- Branch instructions take 2 cycles if not taken, 4 cycles if taken
- Hub access uses round-robin timing with 0-7 cycle wait for window
- Burst transfers (via SETQ) amortize Hub access overhead
- The P2 provides deterministic timing with no cache or speculative execution
- Conditional execution eliminates branch timing variation
- GETCT reads the cycle counter for precise timing measurement
- Hub execution mode adds instruction fetch latency

Chapter 5: Special Hardware Overview

The P2 includes specialized hardware subsystems that extend beyond basic instruction execution. Understanding these subsystems enables advanced applications: the CORDIC coprocessor accelerates mathematical operations, Smart Pins provide programmable I/O peripherals, the Streamer enables high-speed data movement, events support responsive programming, hardware locks coordinate multi-COG applications, and **DEBUG** hardware assists development. This chapter provides an overview of each subsystem; detailed instruction usage is covered in Part II, and complete subsystem documentation is available in specialized manuals.

5.1 CORDIC Coprocessor

The CORDIC (Coordinate Rotation Digital Computer) coprocessor provides hardware-accelerated mathematical operations. While the P2's instruction set includes basic arithmetic, the CORDIC handles operations that would otherwise require hundreds of instructions: 32×32-bit multiplication producing 64-bit results, division with quotient and remainder, square root extraction, trigonometric computations, and logarithmic functions. The CORDIC operates as a queue-based coprocessor—your code initiates an operation, performs other useful work for 55 clock cycles while the CORDIC computes, then retrieves the results.

5.1.1 CORDIC Capabilities

The CORDIC provides eight categories of operations, each accessed through dedicated queue instructions:

Operation	Instruction	Output
Multiply	QMUL	64-bit product (low 32 bits in X, high 32 bits in Y)
Divide	QDIV	Quotient in X, remainder in Y
Fractional divide	QFRAC	Fractional quotient in X, remainder in Y
Square root	QSQRT	Integer square root in X
Rotate	QROTATE	Rotated X coordinate, rotated Y coordinate
Vector	QVECTOR	Magnitude in X, angle in Y (Cartesian to polar)
Logarithm	QLOG	Base-2 logarithm (5:27 fixed-point) in X
Exponential	QEXP	e^{}x approximation in X

Each operation produces one or two 32-bit results, retrieved through GETQX and GETQY instructions. The multiply operation (QMUL) is particularly valuable for fixed-point arithmetic, providing the full 64-bit product that would otherwise require complex multi-instruction sequences.

5.1.2 CORDIC Operation Flow

CORDIC operations follow a three-step pattern: queue the operation, wait for computation, retrieve results. The critical timing constraint is the 55-clock computation period—attempting to retrieve results before this period completes produces undefined values.

```
        QMUL    multiplicand, multiplier    ' Start 32x32 multiply
        ' ... 55 clocks of other useful work ...
        GETQX   product_lo                  ' Get low 32 bits
        GETQY   product_hi                  ' Get high 32 bits
```

The 55-clock computation period is fixed for all CORDIC operations. Efficient code interleaves CORDIC computations with other processing, ensuring the CPU remains productive while the coprocessor works. The CORDIC operates independently once queued, allowing the COG to execute unrelated instructions during the computation period.

5.1.3 CORDIC Pipelining

The CORDIC is a fully pipelined, shared resource accessed through hub rotation—the same arbitration mechanism used for hub RAM. Each COG receives a CORDIC access slot every 8 clocks. The pipeline is 54 stages deep; results are available 55 clocks after queuing (1 clock to enter the pipeline, 54 clocks to process). With 8-clock access intervals, a single COG can have 6-7 operations in flight simultaneously (54 ÷ 8 6.75). This deep pipelining enables sustained high throughput when processing multiple values.

5.1.4 The Pipeline Phases

Effective CORDIC usage follows a three-phase pattern: fill, steady-state, and drain.

Fill Phase: Submit multiple operations before expecting any results. During this phase, you queue operations without retrieving results, filling the pipeline:

```
        ' Fill phase - queue first 6 operations
        QMUL    a0, b0                      ' Operation 0 enters pipeline
        QMUL    a1, b1                      ' Operation 1 (8 clocks later)
        QMUL    a2, b2                      ' Operation 2
        QMUL    a3, b3                      ' Operation 3
        QMUL    a4, b4                      ' Operation 4
        QMUL    a5, b5                      ' Operation 5
        ' Pipeline now filling, first result not ready yet
```

Steady-State Phase: Once the pipeline fills, retrieve one result and submit one new operation each access slot. This phase achieves maximum throughput—one result per 8 clocks:

```
        ' Steady state - retrieve previous, submit next
.loop   GETQX   result_lo                   ' Get result from ~55 clocks ago
        GETQY   result_hi
        QMUL    a_next, b_next              ' Submit next operation
        ' ... process result, prepare next operands ...
        DJNZ    count, #.loop
```

Drain Phase: After submitting the final operation, continue retrieving remaining results without submitting new operations:

```
        ' Drain phase - retrieve final results
        GETQX   result_lo                   ' Get remaining results
        GETQY   result_hi
        ' ... repeat for each operation still in pipeline ...
```

5.1.5 Result Retrieval Timing

The **GETQX** and **GETQY** instructions retrieve results in submission order. If a result is not yet ready when **GETQX** or **GETQY** executes, the COG stalls until the result becomes available. This automatic stalling simplifies programming—you need not count cycles precisely—but can impact performance if you retrieve too early.

For non-blocking result checking, use **POLLQMT** to test whether the CORDIC pipeline is empty:

```
        POLLQMT             wc              ' C=1 if pipeline empty,
                                            '  C=0 if results pending
        if_nc   GETQX   result              ' Retrieve if available
```

The CORDIC generates Event 15 when **GETQX** or **GETQY** executes with no results available. This event can trigger an interrupt or be polled, useful for detecting programming errors where retrieval occurs before any operations were queued.

5.1.6 Practical Pipelining Example

This example processes an array of coordinate pairs, rotating each by a fixed angle. The pipeline keeps multiple rotations in flight:

```
' Rotate 16 coordinate pairs by angle
' Input: point_array (pairs of X,Y longs), angle
' Output: rotated coordinates written back to array
rotate_points
        MOV     count, #16
        MOV     ptra, ##point_array         ' Read pointer
        MOV     ptrb, ##point_array         ' Write pointer (same array)

        ' Fill phase - start first 6 rotations
        CALL    #queue_rotation             ' Queue op 0
        CALL    #queue_rotation             ' Queue op 1
        CALL    #queue_rotation             ' Queue op 2
        CALL    #queue_rotation             ' Queue op 3
        CALL    #queue_rotation             ' Queue op 4
        CALL    #queue_rotation             ' Queue op 5
        SUB     count, #6

        ' Steady state - retrieve one, queue one
```

continues on next page →

↪ *continued from previous page*

```
.loop   GETQX   rotated_x                   ' Get previous result
        GETQY   rotated_y
        WRLONG  rotated_x, ptrb++           ' Store result
        WRLONG  rotated_y, ptrb++
        CALL    #queue_rotation             ' Queue next
        DJNZ    count, #.loop

        ' Drain phase - retrieve final 6 results
        REP     @.drain_end, #6
        GETQX   rotated_x
        GETQY   rotated_y
        WRLONG  rotated_x, ptrb++
        WRLONG  rotated_y, ptrb++
.drain_end
        RET

' Helper: queue one rotation from point array
queue_rotation
        RDLONG  x, ptra++
        RDLONG  y, ptra++
        SETQ    y                           ' Y coordinate to Q register
        QROTATE x, angle                    ' Start rotation
        RET
```

This pattern achieves one rotation result every ~20 instructions (the loop body), rather than waiting 55 clocks per rotation. For 16 points, the pipelined version completes in roughly 320 clocks versus 864 clocks for sequential processing—nearly 3× faster.

5.1.7 CORDIC Instructions Reference

Queue Operations: QMUL, QDIV, QFRAC, QSQRT, QROTATE, QVECTOR, QLOG, QEXP

Result Retrieval: GETQX, GETQY

Full instruction details, including operand formats and result interpretations, appear in Part II under each instruction's entry.

5.1.8 Protecting Critical CORDIC Sequences

Note: This section applies only to PASM2 code with interrupts enabled. Spin2 operators that use CORDIC (such as `*`, `/`, `SQRT`, `QSIN`, `QCOS`, etc.) are already protected by the Spin2 interpreter—no additional protection is needed when using Spin2.

The 55-clock delay between queuing a CORDIC operation and retrieving its result creates a timing window. In PASM2 applications using interrupts, an interrupt that fires during this window could delay result retrieval or queue additional operations that interfere with the expected sequence. For timing-critical applications, this can cause incorrect results or undefined behavior.

The P2 provides a simple protection mechanism using **REP** with a single iteration:

```
' Protect CORDIC operation from interrupts
        REP     @.protect, #1         ' Execute block atomically
        QMUL    multiplicand, multiplier
        ' ... other CORDIC work (up to 55 clocks) ...
        GETQX   result_lo
        GETQY   result_hi
.protect
```

This idiom works because **REP** stalls interrupt handling until all repeated instructions complete—even with just one iteration. The entire sequence from **QMUL** through **GETQY** executes without interruption.

When to use interrupt protection:

- **DSP inner loops:** Where CORDIC operations must maintain precise timing relationships
- **Fixed-point arithmetic chains:** Where one CORDIC result feeds immediately into another calculation
- **Real-time control:** Where interrupt latency could cause result retrieval timing errors

When protection is unnecessary:

- **Spin2 code:** The Spin2 interpreter already protects CORDIC operations internally
- **PASM2 without interrupts:** When no interrupts are enabled (SETINT not used)
- **Background calculations:** Where the 55-clock window has explicit **NOP** padding or other work
- **Pipelined processing:** Where the fill-steady-drain pattern naturally handles timing

For longer critical sequences, use a large **REP** block count with one iteration:

```
' Extended interrupt-free zone
        REP     #99, #1               ' 99 instructions, 1 iteration
        QSQRT   value, #0             ' CORDIC operations
        QLOG    value
        QEXP    value
        ' ... up to 99 total instructions ...
        GETQX   result
_ret_   MOV     output, result        ' REP exits at _ret_
```

The large instruction count (99) creates an interrupt-free zone that terminates at the first `RET`, `_ret_`, or branch instruction encountered.

5.2 Smart Pins

The P2 provides 64 Smart Pins, one per I/O pin, each containing a complete programmable peripheral. Smart Pins eliminate the need for external support chips in many applications—a single Smart Pin can implement a UART transmitter and receiver, generate PWM signals, measure pulse widths, read quadrature encoders, or convert analog signals. Each Smart Pin contains local state machines, DAC and ADC hardware, timing circuits, and configuration registers, all controlled through PASM2 instructions. The Smart Pin architecture

offloads I/O processing from the COG, allowing precise timing and continuous operation without software intervention.

5.2.1 Smart Pin Architecture

Each Smart Pin integrates multiple hardware components that work together to implement various I/O functions:

- **Configurable I/O circuitry:** Programmable pull-up/down resistors, output drivers, and high-impedance (floating) modes
- **Mode selection logic:** 32 distinct operating modes covering digital, analog, serial, and timing applications
- **Local state machine:** Autonomous operation once configured, generating events when data is ready
- **DAC hardware:** 8-bit digital-to-analog converter for analog output and sigma-delta modulation
- **ADC hardware:** Analog-to-digital conversion using sigma-delta and comparator techniques
- **Timing hardware:** Counters and comparators for precise edge detection and pulse generation

The Smart Pin's autonomous operation is particularly significant. Once configured, a Smart Pin operates independently of the COG—a UART Smart Pin transmits and receives bytes, a PWM Smart Pin generates continuous waveforms, an encoder Smart Pin tracks position changes, all without ongoing CPU attention. The COG interacts with Smart Pins only when new data arrives or new output is needed.

5.2.2 Smart Pin Modes

Smart Pins support 32 distinct modes organized into functional categories. Each mode transforms the pin into a specialized peripheral:

Category	Example Modes	Typical Applications
Digital I/O	Repository mode, registered input, long pulse accumulator	Debounced buttons, event counting, pulse measurement
Serial	UART transmit/receive, synchronous serial, SPI	Communication with peripherals and other systems
PWM	PWM/duty mode, triangle/sawtooth mode, incremental mode	Motor control, LED dimming, audio generation
Analog	DAC output, ADC sampling, comparator	Sensor interfacing, analog signal generation
Timing	Period measurement, pulse width measurement, timeout	Frequency measurement, event timing, watchdog
Quadrature	Quadrature encoder input	Rotary encoder reading, motor position feedback

Mode selection determines the pin's complete behavior: input vs. output, edge sensitivity, data format, timing parameters, and event generation. The mode value, written through **WRPIN**, configures all aspects of the Smart Pin's operation.

5.2.3 Smart Pin Instructions

Smart Pin operation involves three phases: configuration, communication, and direction/output control. PASM2 provides dedicated instructions for each phase.

Configuration Instructions:

Configuration establishes the Smart Pin's operating mode and parameters:

- **WRPIN** - Write pin mode (selects one of 32 operating modes)
- **WXPIN** - Write X parameter (mode-specific configuration value)
- **WYPIN** - Write Y parameter (mode-specific configuration value or output data)

The three-register configuration pattern (mode, X, Y) provides each mode with sufficient parameters. For example, UART mode uses X for bit timing and Y for transmit data; PWM mode uses X for period and Y for duty cycle.

Communication Instructions:

Communication instructions transfer data between the COG and Smart Pin:

- **RDPIN** - Read Smart Pin data and acknowledge (clears ready flag)
- **RQPIN** - Read Smart Pin data without acknowledge (preserves ready flag)
- **AKPIN** - Acknowledge only (clears ready flag without reading)

The read-and-acknowledge pattern prevents missing data. A Smart Pin sets its ready flag when new data arrives; **RDPIN** retrieves the data and clears the flag in one atomic operation. **RQPIN** allows checking values without consuming data, useful for monitoring inputs.

Direction and Output Control Instructions:

Direction and output control manage the physical pin state. The P2 provides four instruction families (DIR, OUT, FLT, DRV), each with eight suffix variants (L, H, C, NC, Z, NZ, NOT, RND):

- **DIR** family - Set pin direction (input vs. output)
- **OUT** family - Set output value (when pin is output)
- **FLT** family - Float pin to high-impedance (tri-state)
- **DRV** family - Drive pin (opposite of float)

Each family includes suffix variants: `L` (DIR/OUT bit := 0), `H` (:= 1), `C` (:= C flag), `NC` (:= !C flag), `Z` (:= Z flag), `NZ` (:= !Z flag), `NOT` (toggle the bit), `RND` (:= a random bit). This provides fine-grained control: `DIRL` forces the pin to input (DIR=0), while `DIRZ` sets the pin's direction to the current Z flag value (Z=1 → output, Z=0 → input).

5.2.4 Smart Pin Documentation

Smart Pin modes vary significantly in configuration and operation. The mode value, X parameter, and Y parameter have different meanings for each mode—UART mode parameters differ completely from PWM mode parameters. Complete Smart Pin mode documentation, including configuration values, timing diagrams, and usage examples, appears in the **P2 Smart Pins Tutorial** (`p2-smart-pins-tutorial`). That manual provides essential reference material for Smart Pin programming.

5.3 Streamer

The Streamer provides DMA-like high-speed data movement between Hub memory and I/O pins. While Smart Pins handle byte-level serial I/O, the Streamer specializes in bulk data transfer at rates matching the system clock—transferring pixels to displays, streaming audio samples to DACs, generating complex waveforms, or receiving high-speed ADC data. The Streamer operates autonomously once configured, fetching data from Hub memory and delivering it to output pins (or capturing from input pins) without COG intervention. This frees the COG to perform computations while data flows continuously.

5.3.1 Streamer Capabilities

The Streamer excels at applications requiring continuous data flow at precise timing:

- **RGB/pixel streaming:** Driving LED panels, VGA displays, or other parallel pixel interfaces requiring continuous refresh
- **ADC/DAC streaming:** Audio applications where sample streams flow continuously between Hub memory and audio hardware
- **Waveform generation:** Creating complex analog waveforms through DAC output, including modulated signals
- **High-speed data acquisition:** Capturing parallel data from external ADCs or digital sensors

The Streamer's key characteristic is autonomy—once initialized with a Hub memory address and transfer parameters, it fetches and outputs data without further CPU involvement. The COG can prepare the next buffer, perform signal processing on captured data, or execute unrelated tasks while the Streamer handles data movement.

5.3.2 Streamer Instructions

Streamer operation involves configuration, initiation, and control. The instruction set provides precise control over transfer timing and data flow.

Configuration and Control:

- **SETXFRQ** - Set streamer frequency (controls output sample rate)
- **XINIT** - Initialize streamer transfer (configures mode and starts first transfer)
- **XCONT** - Continue streamer operation (starts next transfer using current configuration)
- **XZERO** - Zero-fill streamer output (outputs zeros without fetching Hub data)
- **XSTOP** - Stop streamer (halts transfer operation)

The typical pattern initializes the Streamer with **XINIT** for the first buffer, then uses **XCONT** to chain subsequent buffers. **SETXFRQ** establishes the output timing, critical for audio sample rates or display refresh timing. **XZERO** allows inserting silence in audio streams or blanking periods in video signals without transferring Hub data.

5.3.3 Streamer Modes

The Streamer supports multiple operating modes, each optimized for specific data transfer patterns:

Mode	Purpose	Typical Application
LUT mode	Transfer data through lookup table	Color palette mapping, gamma correction
NCO mode	Numerically controlled oscillator	Waveform synthesis, signal generation
RF mode	Radio frequency output generation	RF signal generation, modulation
Goertzel mode	DSP filtering during transfer	Frequency detection, tone decoding

Mode selection appears in the **XINIT** instruction's mode parameter, along with configuration bits controlling data width, pin selection, and transfer direction. Each mode interprets Hub memory data differently—LUT mode uses data as lookup indices, NCO mode uses data as frequency control words, RF mode uses data as modulation patterns.

5.3.4 Streamer Configuration

Streamer commands are built by combining mode constants using OR operations. The constants follow a naming convention that encodes the data flow:

- **X_IMM_** - Immediate data modes (data passed directly)
- **X_RFBYTE/RFWORD/RFLONG_** - Read from FIFO (hub RAM) with specified data width
- **X_..._WFBYTE/WFWORD/WFLONG** - Write to FIFO (hub RAM) for capture operations
- **X_DACS_** - DAC channel selection and configuration
- **X_PINS_ON/OFF** - Enable/disable pin outputs
- **X_WRITE_ON/OFF** - Enable/disable hub RAM writes

The naming pattern `X_[source][size]_[pins]P_[dacs]DAC[bits]` describes the complete data path. For example, `X_RFBYTE_RGB8` reads bytes from hub RAM and interprets them as RGB 3:3:2 color values.

Complete X_* constant documentation, including all 78 mode constants with values and descriptions, appears in Appendix F (Streamer Mode Constants). That appendix provides the detailed reference needed to configure the Streamer for specific applications, including usage examples for video streaming, audio DAC output, and ADC capture.

5.4 Events and Interrupts

The P2 supports event-driven programming through a comprehensive event system. Events notify code when specific conditions occur: counters reach target values, I/O pins match patterns, the Streamer completes transfers, the CORDIC finishes computations, or other COGs request attention. The P2 provides two response mechanisms: polling (checking event flags in code) and interrupts (automatic vectoring to handler code). The architecture favors polling—with 8 COGs available, dedicating one COG to event monitoring often provides better response than interrupt overhead. Interrupts remain available when needed, offering three priority levels for nested interrupt handling.

5.4.1 Event Sources

The P2 defines numerous event sources, each representing a distinct hardware condition:

Event	Source	Typical Use
INT1, INT2, INT3	Software-triggered interrupts	Inter-COG signaling, priority events
CT1, CT2, CT3	Counter events	Periodic timing, scheduled events
SE1, SE2, SE3, SE4	Selectable events	Pin edges, lock status, configurable conditions
PAT	Pattern match on pins	Multi-pin state detection, port monitoring
FBW	FIFO block wrap	Set up next FIFO block at circular-buffer boundary (via FBLOCK)
XMT	Streamer ready for new command	Command-buffer-empty (streamer-empty) notification
XFI	Streamer finished (no pending command)	Wait for streamer completion / streamer idle
XRO	Streamer rollover	Circular buffer management
XRL	Streamer read LUT $1FF	LUT-wrap timing event
ATN	Attention from another COG	Inter-COG communication
QMT	CORDIC operation complete	Math coprocessor completion

Each event source sets a corresponding flag when its condition occurs. Code responds to events through wait instructions (blocking until event occurs), poll instructions (testing event flag without blocking), or interrupt configuration (automatic handler invocation).

5.4.2 Event Configuration

Event configuration establishes which conditions trigger events and how events invoke responses.

Selectable Event Configuration:

The four selectable events (SE1-SE4) can monitor various conditions:

- **SETSE1, SETSE2, SETSE3, SETSE4** - Configure selectable event sources

Each SETSE instruction selects one condition from dozens of options: pin edges (rising/falling on any pin), lock states (locked/unlocked), counter comparisons, or other hardware events. This flexibility allows tailoring event detection to application requirements.

Interrupt Configuration:

Interrupt setup involves two steps: configuring the interrupt source and enabling interrupt processing:

- **SETINT1, SETINT2, SETINT3** - Select the interrupt event source (4-bit code in Dest[3:0]). The handler address is set separately by writing the IJMP1/2/3 registers ($1F4/$1F2/$1F0).
- **STALLI** - Stall (disable) interrupt processing
- **ALLOWI** - Allow (enable) interrupt processing (default on COG start)

Each interrupt level (1, 2, 3) has independent configuration. Level 3 can interrupt level 2; level 2 can interrupt level 1; level 1 can interrupt normal execution. This provides priority-based interrupt handling when multiple urgent events require service.

5.4.3 Event Waiting

Wait instructions block execution until the specified event occurs. The COG halts, consuming minimal power, until the event flag sets:

- **WAITSE1, WAITSE2, WAITSE3, WAITSE4** - Wait for selectable event
- **WAITINT** - Wait for any interrupt to occur
- **WAITCT1, WAITCT2, WAITCT3** - Wait for counter event
- **WAITATN** - Wait for attention from another COG
- **WAITPAT** - Wait for pin pattern match

Wait instructions provide deterministic event response—the next instruction executes immediately after the event occurs. This pattern works well for COGs dedicated to event handling, where blocking behavior is acceptable.

5.4.4 Event Polling

Poll instructions **TEST** event flags without blocking. If the event has occurred, the instruction sets condition flags; if not, execution continues immediately:

- **POLLSE1, POLLSE2, POLLSE3, POLLSE4** - Poll selectable event status
- **POLLINT** - Poll interrupt status
- **POLLCT1, POLLCT2, POLLCT3** - Poll counter event status
- **POLLATN** - Poll attention status
- **POLLPAT** - Poll pattern match status

Polling enables responsive event handling within loops. Code can check multiple events in sequence, responding to whichever occurred, without blocking on any single event:

```
1                POLLSE1         wc          ' Test event 1, C if occurred
2        if_c    JMP     #handler               ' Branch to handler only if
3                                               '  event fired
```

This pattern branches to handler code only when the event occurred.

5.4.5 Interrupt Philosophy

The P2's 8-COG architecture fundamentally changes interrupt philosophy. Traditional single-processor systems use interrupts because no other mechanism provides responsive event handling—the single CPU must interrupt current work to handle urgent events. The P2 offers an alternative: dedicate a COG to event monitoring. A COG waiting for events responds with zero latency when events occur, requires no context save/restore overhead, and introduces no interrupt-related bugs. The COG dedicated to event handling becomes the "interrupt handler," continuously available.

Interrupts remain valuable in specific scenarios:

- **Emergency response:** Hardware failure detection requiring immediate response across all COGs
- **Resource constraints:** When 8 COGs are fully utilized and event handling must share a COG
- **Legacy patterns:** When porting code from single-processor architectures

When interrupts are necessary, the P2's three priority levels enable nested interrupt handling. A high-priority interrupt can preempt a low-priority handler, ensuring critical events receive immediate attention even during other interrupt processing.

5.5 Locks and Synchronization

The P2 provides 16 hardware locks for inter-COG synchronization. When multiple COGs access shared resources—Hub memory data structures, Smart Pin configurations, or hardware peripherals—locks ensure mutual exclusion, preventing race conditions and data corruption. Hardware locks offer atomic test-and-set operations that software alone cannot provide. A COG attempting to acquire a held lock receives immediate notification rather than unknowingly accessing contested resources. The 16 locks support complex applications where multiple COGs coordinate access to numerous shared resources.

5.5.1 Lock Operations

Four instructions manage the complete lock lifecycle: allocation, acquisition, release, and deallocation.

Instruction	Purpose	Condition Flag Behavior
LOCKNEW	Allocate a new lock from the pool	C=0 if lock allocated, C=1 if pool empty
LOCKRET	Return a lock to the pool	Lock becomes available for reallocation
LOCKTRY	Try to acquire a lock	C=0 if already held/failed, C=1 if now acquired
LOCKREL	Release a held lock	Lock becomes available for other COGs

The allocation model prevents lock ID conflicts. **LOCKNEW** returns a lock ID from the pool of available locks; **LOCKRET** returns the lock for reuse. This ensures lock IDs remain valid—if COG A uses lock 5, no other COG receives lock 5 from **LOCKNEW** until COG A returns it via **LOCKRET**.

5.5.2 Lock Usage Pattern

Typical lock usage follows a four-phase pattern: allocate, acquire-use-release loop, deallocate:

```
                LOCKNEW lock_id         wc      ' Allocate lock from pool
        if_c    JMP     #no_locks               ' Handle pool exhaustion

critical_section
                LOCKTRY lock_id         wc      ' Try to acquire lock
        if_nc   JMP     #critical_section       ' Retry if lock held

                ' ... exclusive access to shared resource ...
                WRLONG  data, hub_addr          ' Safe: we hold the lock

```

continues on next page →

↪ continued from previous page

```
                LOCKREL lock_id                 ' Release for other COGs

                ' ... additional work ...
                JMP     #critical_section       ' Repeat access cycle

done            LOCKRET lock_id                 ' Return lock to pool
```

The **LOCKTRY**/LOCKREL pair forms the critical section boundary. Between **LOCKTRY** success and **LOCKREL**, this COG has exclusive access—all other COGs executing **LOCKTRY** on the same lock will fail (C=0) until **LOCKREL** executes. The retry loop (`if_nc JMP #critical_section`) implements busy-waiting, appropriate when lock hold times are short.

5.5.3 Lock Synchronization Use Cases

Locks solve multiple classes of multi-COG coordination problems:

Shared Data Structures:

When multiple COGs read and modify Hub memory data structures (queues, buffers, linked lists), locks prevent partial updates:

```
                LOCKTRY queue_lock      wc
        if_nc   JMP     #retry
                RDLONG  head, queue_head        ' Read
                ADD     head, #1                ' Modify
                WRLONG  head, queue_head        ' Write back
                LOCKREL queue_lock              ' Complete atomic update
```

Without the lock, two COGs might simultaneously read the same `head` value, increment independently, and write back the same result—losing one increment.

Hardware Resource Arbitration:

When multiple COGs share hardware resources (specific Smart Pin, display controller, audio output), locks coordinate exclusive access:

```
                LOCKTRY display_lock    wc      ' Acquire display
        if_nc   JMP     #retry
                ' ... draw graphics, write text ...
                LOCKREL display_lock            ' Release for other COGs
```

Producer/Consumer Synchronization:

Lock status serves as a signaling mechanism. A producer holds a lock while data is invalid; releasing the lock signals data ready. A consumer waits via **LOCKTRY**, acquiring the lock when data becomes valid.

The 16-lock limit rarely constrains applications—complex systems typically need fewer than 16 distinct critical sections. Applications requiring more synchronization points often combine locks with other mechanisms (event flags, shared memory flags) for fine-grained coordination.

5.6 XBYTE Bytecode Engine

XBYTE is a hardware bytecode dispatch mechanism. When a **RET** or _RET_ instruction returns to address $1FF, the hardware automatically fetches a bytecode from the FIFO, looks up a dispatch entry in LUT RAM, and branches to the handler routine. Total dispatch overhead is 6 clock cycles.

5.6.1 Dispatch Cycle

XBYTE executes as a phantom instruction triggered by returning to $1FF. The return does not pop the hardware stack, so repeated **RET**/_RET_ instructions fetch successive bytecodes.

Clock	Phase	Activity
1	go	RFBYTE bytecode, SKIPF #0
2	get	MOV PA,bytecode, RDLUT
3	go	RDLUT complete
4	get	EXECF begin
5	go	MOV PB,(GETPTR), MODCZ, branch
6-7		Pipeline flush/reload
8	get	First instruction of handler

A handler ending with `_RET_` **ADDS** 2 clocks, making the minimum cycle 8 clocks total.

5.6.2 LUT Entry Format

Each 32-bit LUT entry contains:

Bits	Content
[9:0]	Handler address in COG/LUT RAM
[31:10]	SKIPF pattern (22 bits)

EXECF simultaneously branches and applies the **SKIP** pattern.

5.6.3 Configuration Summary

XBYTE is configured via `_RET_ SETQ {#}D` with $1FF on the stack:

Mode	LUT Entries	Index Source
Full 8-bit	256	bytecode[7:0]
7-bit	128	bytecode[6:0] or [7:1]
6-bit	64	bytecode[5:0] or [7:2]
5-bit	32	bytecode[4:0] or [7:3]
4-bit	16	bytecode[3:0] or [7:4]

Smaller modes conserve LUT space. A compressed mode allows mixing individual and shared handlers.

5.6.4 Handler Requirements

- **Location:** COG RAM ($000-$1FF) or LUT RAM ($200-$3FF)
- **Exit:** Must end with **RET** or _RET_
- **Registers:** PA contains bytecode value; PB contains FIFO pointer

See: SETQ, **SETQ**2 for configuration; **EXECF**, **SKIPF** for dispatch mechanism; **RFBYTE**, **RDFAST** for FIFO operations; **GETBRK** for debugging state

5.7 Boot Process

When the P2 powers on or receives a hardware reset, it begins a deterministic boot sequence that loads and executes user code. Understanding this sequence is essential for embedded applications—it explains why programs must configure the clock, how the chip finds your code, and what state the hardware is in when your program starts executing.

5.7.1 Initial Chip State

At reset, the P2 initializes to a known state before any user code executes:

Resource	Initial State
Clock source	RCFAST (~20-30 MHz (typically ~24 MHz) internal RC oscillator)
All COGs	Stopped (except COG 0)
Hub RAM	Undefined contents
I/O pins	High-impedance (floating)
64-bit counter	Cleared to zero
PRNG	Seeded with thermal noise

The internal RC oscillator (RCFAST) provides the initial clock. This oscillator is guaranteed to run at least 20 MHz under all conditions, ensuring reliable serial communication during boot. The exact frequency varies with temperature and manufacturing, typically ~24 MHz. Programs requiring precise timing must configure an external crystal or the PLL after boot.

The boot ROM seeds the Xoroshiro128** pseudo-random number generator with true random data. The ROM reads thermal noise from pin 63 (configured in ADC calibration mode) fifty times, using each 31-bit sample to seed the PRNG through **HUBSET**. This establishes high-quality randomness available immediately when user code starts—there is no need to seed the PRNG again, though programs may do so if desired.

5.7.2 Boot Source Selection

The P2 determines its boot source by sensing external pull-up resistors on pins P59-P61. This hardware detection occurs automatically and requires no software configuration.

P61	P60	P59	Boot Behavior
none	none	none	Serial only (60s window)
pull-up	none	none	Serial 100 ms window, then SPI flash; serial 60s on flash failure
pull-up	any	pull-down	SPI flash only (fast boot), no serial; shutdown on failure
none	pull-up	none	SD card, then serial (60s) on failure
none	pull-up	pull-down	SD card only, shutdown on failure
any	any	pull-up	Serial only (60s window); no flash or SD boot

The pull-up detection uses internal sensing—no software reads these pins. The boot ROM checks pin states immediately after reset and branches to the appropriate loader. Development boards typically include jumpers or switches to select boot mode; production designs hard-wire the appropriate resistor configuration.

5.7.3 Boot Pin Assignments

The boot process uses pins P58-P63 for communication with external boot sources:

Serial Boot (P62-P63):

Pin	Function	Direction
P63	Serial RX	Input
P62	Serial TX	Output

SPI Flash Boot (P58-P61):

Pin	Function	Direction
P61	Chip Select (active low)	Output
P60	Clock	Output
P59	Data Out (MOSI)	Output
P58	Data In (MISO)	Input

SD Card Boot (P58-P61):

Pin	Function	Direction
P61	Chip Select (directly active low)	Output
P60	Clock	Output
P59	Data Out (MOSI)	Output
P58	Data In (MISO)	Input

After boot completes, ROM control of these pins ends and user code takes over. However, the boot source hardware typically remains physically connected:

- **SPI Flash (P58-P61):** The flash chip remains attached. User programs commonly continue using these pins to access flash storage for code snippets, lookup tables, audio files, or data logging.
- **SD Card (P58-P61):** The SD card socket remains attached. User programs commonly continue using these pins for file system access.
- **Serial (P62-P63):** On development boards, these pins typically remain connected to the USB-serial interface for debugging and host communication.

The pins are available for user code to configure and use—but practical usage depends on what external hardware is connected to them.

5.7.4 The Boot Sequence

After reset, COG 0 loads and executes the boot ROM program (ROM_Booter.spin2). The boot sequence proceeds as follows:

Step 1: Check for SPI Flash

If an external pull-up is detected on P61, the booter attempts SPI flash boot:

1. Load the first 1024 bytes (256 longs) from SPI flash into hub RAM at $00000
2. Compute the 32-bit sum of these 256 longs
3. If the sum equals "Prop" ($706F7250), the data is valid:
    - Copy the 256 longs from hub to COG 0 registers $000-$0FF
    - If P59 is pulled down: execute immediately (`JMP #$000`)
    - Otherwise: wait for serial commands (100ms timeout), then execute

Step 2: Serial Loader Window

If SPI boot is not configured or fails checksum validation, the booter enters serial loader mode:

1. Wait for serial commands on P63 (RX pin)
2. Auto-detect baud rate from incoming data (9600 to 2,000,000 baud)
3. Accept commands for up to 60 seconds
4. If a valid program loads: execute via `COGINIT #0,#0`
5. If timeout expires with no valid program: switch to RCSLOW (~20 kHz) and halt COG 0

Step 3: Program Execution

Once valid code is loaded, the booter launches it:

- For SPI/SD boot: `JMP #$000` executes code now in COG 0's registers
- For serial boot: `COGINIT #0,#0` relaunches COG 0 from hub address $00000

In both cases, user code begins executing with the clock still in RCFAST mode. The program must configure the desired clock source if different timing is required.

5.7.5 Serial Loading Protocol

The serial loader provides a text-based protocol for loading code during development. The protocol auto-detects baud rate by measuring bit timing from received characters, supporting rates from 9,600 to 2,000,000 baud.

Auto-Baud Detection:

The loader calibrates timing from ">" characters ($3E) in the data stream. Send ">" (greater-than followed by space) before the first command and periodically throughout data to maintain accurate baud detection against the drifting internal RC oscillator.

Commands:

Command	Purpose
`Prop_Chk`	Verify communication, returns chip version
`Prop_Clk`	Configure clock source before loading
`Prop_Hex`	Load program data in hexadecimal format
`Prop_Txt`	Load program data in Base64 format

Each command includes mask fields for selecting specific chips when multiple P2s share a serial bus. For single-chip loading, use zero masks: `Prop_Chk 0 0 0 0`.

Data Validation:

Loaded programs must include a validation header. The loader computes a 32-bit sum of all loaded longs; if the sum equals "Prop" ($706F7250), the data is considered valid and execution proceeds. Compilers and loaders automatically generate this checksum.

5.7.6 Clock Configuration After Boot

User code starts executing with the RCFAST clock source—an internal RC oscillator running approximately 20-30 MHz (typically ~24 MHz). For applications requiring precise timing, configure an external crystal or the PLL early in your program:

```
' Configure 20 MHz crystal with PLL for 160 MHz operation
                ' Enable crystal oscillator with 15pF caps
                HUBSET  ##%0000_0001_0000_0000_0000_0000_00_10
                ' Wait 10ms for crystal stabilization
                WAITX   ##20_000_000/100
                ' Switch to crystal clock source
                HUBSET  ##%0000_0001_0000_0000_0000_0000_10_10
```

continues on next page →

↪ continued from previous page

```
                ' Configure PLL: /1 * 8 / 1 = 160MHz
                HUBSET  ##%0000_0001_0000_1000_0000_0010_00_10
                ' Wait 100µs for PLL lock
                WAITX   ##20_000_000/10000
                ' Switch to PLL output
                HUBSET  ##%0000_0001_0000_1000_0000_0010_00_11
```

The **ASMCLK** directive provides a convenient shorthand when using standard crystal configurations. It generates the appropriate **HUBSET** sequence based on the _clkfreq and _clkmode constants defined in your program.

Why Clock Setup Is Required:

The boot ROM cannot know what clock source your hardware provides. Some boards use 20 MHz crystals, others use 25 MHz, and some applications run directly from the internal oscillator. By starting in RC-FAST mode, the P2 boots reliably on any hardware. Your program then configures the actual clock source appropriate for your design.

5.7.7 Rebooting from Software

The **HUBSET** instruction can trigger a hardware reset, returning the chip to the boot sequence:

```
                HUBSET  ##$1000_0000                ' Generate reset pulse,
                                                    '  reboot chip
```

This performs a full hardware reset—all COGs stop, all I/O returns to high-impedance, the clock reverts to RCFAST, and the boot ROM executes from the beginning. Use this for implementing watchdog recovery, firmware updates, or returning to the boot loader.

5.8 DEBUG Output

DEBUG is a compile-time directive that generates serial output code. When enabled, **DEBUG** statements transmit formatted data over the serial connection to the development host, where the debug window displays values, text, and graphical visualizations.

5.8.1 Basic Usage

DEBUG statements output text strings and formatted values:

```
                DEBUG("Starting motor control")     ' Text message
                DEBUG("Speed: ", udec(speed))       ' Decimal value
                DEBUG("Status: ", uhex_(status))    ' Hex without name
```

The serial connection typically runs at 2 Mbaud. When **DEBUG** is disabled via compiler option, statements generate no code.

5.8.2 Value Formatters

DEBUG provides formatters for numeric display. Each has unsigned (U prefix) and signed (S prefix) variants:

Base	Formatters	Output Example
Decimal	UDEC, SDEC	`counter = 42`
Hexadecimal	UHEX, SHEX	`addr = $0400`
Binary	UBIN, SBIN	`flags = %10110`

Underscore suffix (UDEC_, UHEX_, etc.) outputs only the value, omitting the variable name.

Size suffixes (_BYTE, _WORD, _LONG) control display width. Array variants (_BYTE_ARRAY, etc.) display multiple consecutive values.

5.8.3 Visual Debug Displays

DEBUG supports graphical display windows including: - **SCOPE** — Oscilloscope waveform display - **PLOT** — Data plotting and charts - **LOGIC** — Logic analyzer view - **TERM** — Dedicated terminal window - **BITMAP** — Pixel display

Visual displays use a two-phase pattern: creation statement (with display type) establishes the window, update statements (backtick + name) send data points.

5.8.4 Multi-COG Programs

When multiple COGs execute **DEBUG** statements, the system automatically prefixes each message with the COG number (Cog0: through Cog7:). This applies to text output only; visual displays are typically dedicated to specific COGs.

5.8.5 Performance Considerations

Pitfall: DEBUG transmits data serially—each statement can consume hundreds of microseconds. Never place **DEBUG** inside performance-critical loops. Use **DEBUG** before or after loops, or sample infrequently with conditional statements.

For production builds, disable **DEBUG** via compiler option. Statements compile to nothing—zero runtime impact.

See: DEBUG instruction in Part II for complete syntax; P2 **DEBUG** Window Manual for visual display configuration, advanced formatters, and professional debugging techniques.

5.8.6 Debug Configuration

The debug system operates at three distinct levels, each controlled by CON constants:

- **Code Instrumentation (Compile-Time): DEBUG**_DISABLE and **DEBUG**_MASK control whether **DEBUG** statements generate code
- **Output Infrastructure (Runtime): DEBUG**_COGS, **DEBUG**_BAUD, and related constants configure the debug serial system

- **Breakpoint Configuration: DEBUG**_MAIN and **DEBUG**_COGINIT configure automatic breaks for single-step debugging

Selective Debug with debugN:

The `DEBUG[N]()` form categorizes **DEBUG** statements into channels (0-31) that compile selectively based on **DEBUG**_MASK:

```
CON
  DBG_INIT  = 0
  DBG_ERROR = 3
  DEBUG_MASK = (1 << DBG_INIT) | (1 << DBG_ERROR)

DAT
        ORG
entry   DEBUG[DBG_INIT]("Starting")   ' COMPILED - bit 0 set
        DEBUG[1]("Motor status")       ' NOT compiled - bit 1 clear
        DEBUG[DBG_ERROR]("Fault!")     ' COMPILED - bit 3 set
```

Disabled channels produce zero code—no runtime overhead exists. Standard `DEBUG()` statements without channel numbers are unaffected by **DEBUG**_MASK and compile whenever **DEBUG** is enabled.

Compile-Time vs Runtime Filtering:

DEBUG_MASK and **DEBUG**_COGS operate at different levels:

Constant	Level	Controls
DEBUG_MASK	Compile-time	Whether `debug[N]()` generates code
DEBUG_COGS	Runtime	Whether a COG can produce debug output

For a debug statement to produce output, both conditions must be met: the statement must compile (DEBUG_MASK permits it), and the executing COG must have its bit set in **DEBUG**_COGS.

See: Appendix E (Debug Configuration Constants) for complete constant documentation including **DEBUG**_DELAY, **DEBUG**_TIMESTAMP, **DEBUG**_BAUD, and breakpoint configuration.

Key Concepts

- The CORDIC coprocessor provides 55-clock hardware math (multiply, divide, sqrt, trig)
- Smart Pins are 64 programmable I/O peripherals with local state machines
- The Streamer enables DMA-like high-speed data movement
- Events provide non-interrupt notification; interrupts are available when needed
- 16 hardware locks enable safe inter-COG synchronization
- XBYTE provides 6-cycle bytecode dispatch for interpreters and VMs
- The P2 boots from RCFAST (~20 MHz) and detects boot source via pin pull-ups
- User code must configure the desired clock source after boot
- DEBUG provides serial output with formatters; can be disabled for production
- The 8-COG architecture often removes the need for interrupts (see Chapter 4: each COG runs deterministically; dedicate a COG to a task instead of interrupting one)
- Each subsystem is controlled through dedicated PASM2 instructions

Chapter 6: Address Modes

PASM2 provides several addressing modes that determine how instruction operands are specified and how memory is accessed. Understanding these modes is essential for writing efficient code that accesses registers, immediate values, and Hub memory correctly.

This chapter covers all addressing modes from simple register access through the sophisticated pointer expressions used for Hub memory operations. Each mode has specific use cases, encoding requirements, and performance characteristics.

6.1 Direct Register Addressing

The most basic addressing mode specifies COG registers directly by address. Both source and destination operands can use direct register addressing.

6.1.1 Register as Destination

The destination field (D) in every instruction specifies a 9-bit COG register address ($000-$1FF). The instruction reads from and/or writes to this register:

```
1         ADD     result, value           ' result is destination register
2         MOV     counter, #0             ' counter is destination register
3         TEST    flags, #MASK    wz      ' flags is destination (read here)
```

The assembler translates symbolic register names to their addresses. Programmers define registers using labels or the **RES** directive:

```
1 result          RES     1               ' Reserve one long here
2 counter         RES     1
3 flags           RES     1
```

6.1.2 Register as Source

When the I bit (bit 18) is clear, the source field (S) specifies a register address. The instruction reads the value from that register:

```
1         ADD     x, y                    ' y is source register (I=0)
2         MOV     dest, source            ' source is register (I=0)
3         CMP     a, b            wc      ' b is source register (I=0)
```

Direct register addressing provides single-cycle access to COG RAM. Both operands are read simultaneously during instruction execution, making register-to-register operations the fastest possible.

6.1.3 Special Register Addresses

Addresses $1F0-$1FF access special-purpose registers with hardware functions:

Address	Register	Purpose
$1F0-$1F7	IJMP3/IRET3 through PA/PB	Interrupt and scratch registers
$1F8	PTRA	Pointer A for Hub addressing
$1F9	PTRB	Pointer B for Hub addressing
$1FA-$1FB	DIRA/DIRB	Pin direction control
$1FC-$1FD	OUTA/OUTB	Pin output control
$1FE-$1FF	INA/INB	Pin input (read-only)

These registers function like ordinary registers for most purposes but have additional hardware significance.

6.2 Immediate Addressing

Immediate addressing embeds a constant value directly in the instruction rather than reading from a register.

6.2.1 The # Prefix (9-bit Immediate)

The `#` prefix before an operand indicates an immediate value:

```
1         ADD     x, #100                 ' Add immediate value 100
2         MOV     counter, #0             ' Load zero
3         CMP     value, #255     wc      ' Compare against 255
```

When `#` is used:

- The assembler sets the I bit (bit 18) to 1
- The 9-bit S field contains the immediate value
- Valid range: 0 to 511 ($000 to $1FF)

6.2.2 Immediate Range and Signedness

For data instructions the 9-bit immediate field is always zero-extended and treated as unsigned (0-511). Sign-extension of the 9-bit immediate applies only to relative-branch instructions, where the immediate is a signed offset in the range -256..+255:

```
1         MOV     x, #$1FF                ' x = 511 (9-bit, zero-extended)
2         ADD     x, #1                   ' Add 1
3         SUB     x, #10                  ' Subtract 10
```

For relative branches, the same 9-bit immediate is interpreted as a signed offset:

```
        JMP     #$-1                    ' Relative branch back 1 (signed)
```

Values outside the 0-511 range require augmentation (see Section 6.3).

6.2.3 Current Address ($)

The `$` symbol represents the current assembly address:

```
loop    ADD     counter, #1
        DJNZ    count, #$-1             ' Jump back one instruction (to ADD)
        JMP     #$                      ' Infinite loop (jump to self)
```

When used with `#`, it becomes an immediate value representing the address. This is useful for relative branches and self-referencing code.

6.3 Augmented Immediate Addressing

When values exceed 9 bits, PASM2 uses augmentation to provide full 32-bit immediates.

6.3.1 The ## Prefix (32-bit Immediate)

The `##` prefix indicates a full 32-bit immediate value:

```
        MOV     dest, ##$12345678       ' Load full 32-bit value
        ADD     counter, ##1000000      ' Add one million
        MOV     ptr, ##hub_buffer       ' Load 20-bit Hub address
```

6.3.2 How Augmentation Works

The assembler implements `##` by inserting an **AUGS** or **AUGD** instruction before the target instruction:

```
' What the programmer writes:
        MOV     dest, ##$12345678

' What the assembler generates:
        AUGS    #$12345678              ' Provides upper 23 bits [31:9]
        MOV     dest, #$078             ' Provides lower 9 bits: $078
                                        ' Combined result: $12345678
```

The AUG instruction provides bits 31-9, which combine with the 9-bit field from the next instruction to form the complete 32-bit value.

6.3.3 AUGS vs. AUGD

Two augmentation instructions exist:

- **AUGS** augments the Source field of the following instruction
- **AUGD** augments the Destination field of the following instruction

Both operands can be augmented simultaneously:

```
' What the programmer writes:
        WRLONG  ##value, ##address      ' Both operands augmented

' What the assembler generates:
        AUGD    #value_upper            ' Augment D field
        AUGS    #address_upper          ' Augment S field
        WRLONG  #value_lower, #address_lower
```

6.3.4 Augmentation Timing

Each AUG instruction **ADDS +2 clock cycles** to execution:

Augmentation	Additional Cycles
`##Src` only	+2 cycles (AUGS)
`##Dest` only	+2 cycles (AUGD)
`##Dest, ##Src`	+4 cycles (AUGD + AUGS)

```
        MOV     x, #100                 ' 2 cycles
        MOV     x, ##100000             ' 4 cycles (2 + 2 for AUGS)
        WRLONG  ##data, ##addr          ' 6+ cycles (2+2+2: AUGD+AUGS+instr)
```

Performance Note: In time-critical code, large constants should be loaded into registers once and reused, rather than using `##` repeatedly inside loops.

6.3.5 Augmentation is One-Shot

The augmented value applies only to the immediately following instruction. If any instruction intervenes (including a conditional instruction that doesn't execute), the augmentation is consumed:

```
        AUGS    #$12345678
        NOP                             ' This consumes the AUGS!
        MOV     x, #$078                ' Gets only $078, NOT $12345678

        AUGS    #$12345678
```

continues on next page →

↪ continued from previous page

```
6           if_z    MOV     x, #$078        ' Even if Z=0, MOV skipped,
7                                           '  AUGS is still consumed
```

The assembler handles this automatically when `##` notation is used. Manual **AUGS**/AUGD usage requires careful attention to instruction sequencing.

6.4 Pointer Register Addressing (PTRA/PTRB)

The P2 provides two dedicated pointer registers—PTRA ($1F8) and PTRB ($1F9)—that enable sophisticated Hub memory addressing with automatic increment, decrement, and indexing.

6.4.1 Basic Pointer Access

The simplest pointer usage reads or writes Hub memory at the address in PTRA or PTRB:

```
1           MOV     ptra, ##hub_buffer      ' Set PTRA to Hub address
2           RDBYTE  x, ptra                 ' Read byte from Hub at PTRA
3           WRLONG  y, ptrb                 ' Write long to Hub at PTRB
```

6.4.2 The SCALE Factor

Critical Concept: Pointer operations are scaled by the instruction's data size:

Instruction	SCALE	Description
RDBYTE, WRBYTE	1	Byte operations
RDWORD, WRWORD	2	Word (16-bit) operations
RDLONG, WRLONG, WMLONG	4	Long (32-bit) operations

All pointer increments, decrements, and index offsets are multiplied by SCALE. This means:

- `RDBYTE x, PTRA++` increments PTRA by **1 byte**
- `RDWORD x, PTRA++` increments PTRA by **2 bytes**
- `RDLONG x, PTRA++` increments PTRA by **4 bytes**

This automatic scaling makes sequential memory access natural—each operation advances to the next element regardless of element size.

6.4.3 Post-Increment and Post-Decrement

Post-modify modes use the current pointer value for the memory access, then update the pointer afterward:

```
1           RDBYTE  x, ptra++               ' Read byte at PTRA, then PTRA += 1
2           RDWORD  y, ptrb++               ' Read word at PTRB, then PTRB += 2
```

continues on next page →

↪ continued from previous page

```
        RDLONG  z, ptra--               ' Read long at PTRA, then PTRA -= 4
        WRBYTE  x, ptrb--               ' Write byte at PTRB, then PTRB -= 1
```

Execution sequence for `RDLONG x, PTRA++`: 1. Read **LONG** from Hub address in PTRA 2. Store value in register x 3. Add 4 (SCALE for **LONG**) to PTRA

Post-modify is ideal for sequential forward or backward traversal:

```
' Read 10 bytes sequentially
        MOV     ptra, ##source
        REP     @.end, #10
        RDBYTE  x, ptra++               ' Read byte, advance pointer
        ' ... process x ...
.end

' Write longs in reverse order
        MOV     ptrb, ##buffer_end
        REP     @.done, #count
        WRLONG  value, ptrb--           ' Write long, move backward
.done
```

6.4.4 Pre-Increment and Pre-Decrement

Pre-modify modes update the pointer first, then use the new value for memory access:

```
        RDBYTE  x, ++ptra               ' PTRA += 1, then read byte there
        RDWORD  y, ++ptrb               ' PTRB += 2, then read word there
        RDLONG  z, --ptra               ' PTRA -= 4, then read long there
        WRBYTE  x, --ptrb               ' PTRB -= 1, then write byte
```

Execution sequence for `RDLONG x, ++PTRA`: 1. Add 4 (SCALE for **LONG**) to PTRA 2. Read **LONG** from Hub address in updated PTRA 3. Store value in register x

Pre-modify is useful for stack operations and accessing elements relative to a base:

```
' Push onto stack (stack grows upward)
        WRLONG  value, ptra++           ' Post: write here, then advance

' Pop from stack
        RDLONG  value, --ptra           ' Pre: back up first, then read

' Skip first element, read second
        MOV     ptra, ##array
        RDLONG  x, ++ptra               ' Skip element 0, read element 1
```

6.4.5 Indexed Pointer Access (Non-Updating)

Indexed mode accesses memory at an offset from the pointer without modifying the pointer:

```
        RDLONG  x, ptra[0]              ' Read at PTRA + 0*4 = PTRA
        RDLONG  y, ptra[5]              ' Read at PTRA + 5*4 = +20 bytes
        RDBYTE  z, ptrb[-3]             ' Read at PTRB - 3 bytes
        WRWORD  w, ptra[10]             ' Write at PTRA + 20 bytes
```

The index is multiplied by SCALE:

Expression	Instruction	Effective Address
`PTRA[5]`	RDBYTE	PTRA + 5 bytes
`PTRA[5]`	RDWORD	PTRA + 10 bytes
`PTRA[5]`	RDLONG	PTRA + 20 bytes

Index Range (non-updating): -32 to +31 (6-bit signed)

Indexed mode is ideal for accessing structure fields or array elements:

```
' Access structure fields
        MOV     ptra, ##my_struct
        RDLONG  id, ptra[0]             ' First field (offset 0)
        RDLONG  flags, ptra[1]          ' Second field (offset 4)
        RDLONG  data, ptra[2]           ' Third field (offset 8)

' Access array element
        MOV     ptra, ##long_array
        RDLONG  x, ptra[index]          ' Read array[index]
```

6.4.6 Indexed Pointer with Update (Compound Forms)

Compound forms combine indexing with pointer update:

```
        RDLONG  x, ptra++[5]            ' Read at PTRA, then PTRA += 20
        RDLONG  y, ptra--[3]            ' Read at PTRA, then PTRA -= 12
        RDLONG  z, ++ptra[5]            ' PTRA += 5*4, then read at new PTRA
        RDLONG  w, --ptra[3]            ' PTRA -= 3*4, then read at new PTRA
```

Index Range (updating): -16 to +16 (positive 1-16 for `++`/`++[]`, negative -16 to -1 for `--`/`--[]`; value 16 encoded as 0)

These forms enable strided access patterns:

```
' Read every 4th long (stride of 16 bytes)
        MOV     ptra, ##data
        REP     @.end, #count
        RDLONG  x, ptra++[4]            ' Read, advance by 4 longs
        ' ... process x ...
.end

' Read structure array (12-byte structures as 3 longs)
        MOV     ptra, ##struct_array
.loop   RDLONG  field1, ptra++[3]       ' Read field1, skip to next struct
        ' ... (to read all fields, use indexed without update
        '      for field2, field3)
```

6.4.7 Complete PTRx Expression Summary

Expression	Memory Address	Pointer Update
`PTRA`	PTRA	None
`PTRA[index]`	PTRA + index*SCALE	None
`PTRA++`	PTRA	PTRA += 1*SCALE
`PTRA--`	PTRA	PTRA -= 1*SCALE
`++PTRA`	PTRA + 1*SCALE	PTRA += 1*SCALE
`--PTRA`	PTRA - 1*SCALE	PTRA -= 1*SCALE
`PTRA++[index]`	PTRA	PTRA += index*SCALE
`PTRA--[index]`	PTRA	PTRA -= index*SCALE
`++PTRA[index]`	PTRA + index*SCALE	PTRA += index*SCALE
`--PTRA[index]`	PTRA - index*SCALE	PTRA -= index*SCALE

All expressions work identically with PTRB.

6.4.8 Extended Index with AUGS

For index values beyond the 5-bit or 6-bit limits, use `##` to invoke **AUGS**:

```
        RDLONG  x, ptra[##1000]         ' Index 1000 = 1000-byte offset
                                        ' (AUGS index is unscaled)
        RDBYTE  y, ++ptrb[##$12345]     ' 20-bit index with update
```

With **AUGS**, the index becomes a 20-bit value, and the index is **not scaled**—it represents the actual **BYTE** offset:

```
' Without AUGS: index is scaled
        RDLONG  x, ptra[10]             ' Offset = 10 * 4 = 40 bytes

' With AUGS: index is NOT scaled (direct byte offset)
        RDLONG  x, ptra[##40]           ' Offset = 40 bytes (same result)
```

6.5 Block Transfers with SETQ and Pointers

The **SETQ** instruction enables efficient multi-long transfers between Hub memory and COG/LUT RAM.

6.5.1 Basic Block Transfer

```
        SETQ    #15                     ' Transfer 16 longs (count - 1)
        RDLONG  first_reg, ptra         ' Read 16 consecutive longs
```

SETQ specifies the count minus one. The transfer moves `count+1` longs at one **LONG** per clock cycle.

6.5.2 Block Transfer with Pointer Update

When using PTRx with **SETQ** block transfers, the pointer updates by the **total transfer size**:

```
' Post-increment: read from current PTRA, then advance by transfer size
        SETQ    #15                     ' 16 longs
        RDLONG  buffer, ptra++          ' Read 16 longs, PTRA += 64

' Post-decrement: read from current PTRA, then move back
        SETQ    #15
        RDLONG  buffer, ptra--          ' Read 16 longs, PTRA -= 64 bytes

' Pre-increment: advance first, then read
        SETQ    #15
        RDLONG  buffer, ++ptra          ' PTRA += 64, then read 16 longs

' Pre-decrement: move back first, then read
        SETQ    #15
        RDLONG  buffer, --ptra          ' PTRA -= 64, then read 16 longs
```

Critical: With **SETQ** block transfers, the index field is **overridden** by the block count. An arbitrary index cannot be specified:

```
' This does NOT work as expected:
        SETQ    #15
        RDLONG  buffer, ptra++[5]       ' Index [5] IGNORED! Uses count
```

6.5.3 SETQ2 for LUT Transfers

SETQ2 works like **SETQ** but transfers to/from LUT RAM instead of COG RAM:

```
        SETQ2   #31                     ' Transfer 32 longs
        RDLONG  lut_addr, ptra++        ' Read 32 longs into LUT
```

6.5.4 Hardware Bug: ALTx/AUGS Between SETQ and Transfer

SILICON BUG: Do not place ALTx, **AUGS**, or **AUGD** instructions between **SETQ**/SETQ2 and the block transfer instruction when using PTRx expressions.

```
' BUGGY CODE - PTRx update is wrong!
        SETQ    #15                     ' Ready to transfer 16 longs
        ALTD    dest_reg                ' ALTD cancels block PTRx delta!
        RDLONG  0, ptra++               ' PTRA += 4 (1 long), NOT 64!

' CORRECT CODE - No intervening instruction
        SETQ    #15
        RDLONG  dest_reg, ptra++        ' PTRA correctly increments by 64
```

Impact: The data transfer completes correctly (16 longs are read), but PTRA only increments by the normal single-operation amount (4 bytes) instead of the block amount (64 bytes).

Workaround: Never place ALTx, **AUGS**, or **AUGD** between **SETQ**/SETQ2 and the subsequent **RDLONG**/WRLONG/WMLONG when using PTRx expressions.

6.6 ALTx Modified Addressing

The ALT instructions modify how the following instruction interprets its operands, enabling computed addresses and self-modifying code patterns.

Hub-Exec Compatibility: All ALTx instructions (ALTI, **ALTS**, **ALTD**, **ALTR**, **ALTB**, **ALTSN**, **ALTSB**, **ALTSW**, **ALTGN**, **ALTGB**, **ALTGW**) operate identically in cog-exec and hub-exec modes. The ALTx mechanism acts on the next pipelined instruction regardless of its source (cog/LUT memory or the hub-prefetch FIFO), enabling dynamic register-substitution patterns in hub-resident code blocks.

6.6.1 ALTD (Alter Destination)

ALTD modifies the destination field of the next instruction:

```
        ALTD    index, #base            ' Next D = base + index
        MOV     0-0, value              ' Actually writes to base[index]
```

The assembler uses `0-0` as a placeholder for the modified destination.

6.6.2 ALTS (Alter Source)

ALTS modifies the source field of the next instruction:

```
        ALTS    index, #table           ' Next S = table + index
        MOV     result, 0-0             ' Actually reads from table[index]
```

6.6.3 ALTI (Alter Both)

ALTI can modify both destination and source fields, plus the instruction opcode:

```
        ALTI    index, #template        ' Modify D, S, and opcode
        ADD     0-0, 0-0                ' Both operands modified
```

6.6.4 ALTx with AUGS Interaction

SILICON BUG: When an ALTx instruction with an immediate operand follows **AUGS**, the **AUGS** value affects both the ALTx and its intended target.

```
' BUGGY CODE - AUGS affects both instructions
        AUGS    #$12340000
        ALTD    index, #$100            ' #$100 becomes #$12340100! (bug)
        MOV     0-0, #$078              ' #$078 becomes #$12340078

' CORRECT CODE - Use register for ALTx operand
        MOV     base, #$100             ' Put base in register
        AUGS    #$12340000
        ALTD    index, base             ' Register not affected by AUGS
        MOV     0-0, #$078              ' Only this augments to #$12340078
```

Workaround: When using ALTx near **AUGS**, use a register for the ALTx S operand instead of an immediate.

6.7 Hub Address Expressions

Hub memory instructions accept several address expression forms:

6.7.1 Register Address

A register containing a Hub address:

```
        MOV     addr, ##$1000
        RDLONG  x, addr                 ' Read from Hub address in register
```

6.7.2 Immediate Address

An 8-bit immediate Hub address (limited range):

```
1         RDLONG  x, #$80                ' Read from Hub address $80
```

6.7.3 Augmented Immediate Address

A 20-bit Hub address using **AUGS**:

```
1         RDLONG  x, ##$12345            ' Read from Hub address $12345
```

6.7.4 Pointer Expressions

Any of the PTRx forms described in Section 6.4:

```
1         RDLONG  x, ptra                ' Basic pointer
2         RDLONG  x, ptra++              ' With update
3         RDLONG  x, ptra[5]             ' With index
```

6.8 Address Mode Selection Guide

Need	Recommended Mode
Local variable access	Direct register
Small constants (0-511)	9-bit immediate (#)
Large constants, Hub addresses	Augmented immediate (##)
Sequential Hub access	PTRx with ++/–
Random Hub access	PTRx with index
Structure field access	PTRx with fixed index
Block transfers	SETQ + PTRx
Computed register access	ALTx instructions

6.8.1 Performance Considerations

Fastest: Direct register addressing (2 cycles)

Fast: 9-bit immediate (2 cycles)

Moderate: Augmented immediate (+2 cycles per AUG instruction)

Variable: Hub operations (9-16 clocks in COG/LUT mode, 9-26 clocks in HUB mode)

> **Timing Note:** Hub operations require ~9 base clocks plus 0-7 clocks waiting for the hub window (with 8 cogs). In HUB execution mode, the FIFO is busy fetching instructions, adding contention that extends the maximum to 26 clocks.

For time-critical inner loops: - Frequently-used values should reside in COG registers - Large constants should be pre-loaded before entering the loop - Sequential Hub access benefits from PTRx with ++/– - Bulk data movement is most efficient with block transfers (SETQ)

Key Concepts

- Direct register addressing uses 9-bit fields to access COG RAM at addresses $000-$1FF
- The # prefix creates 9-bit immediates (0-511); ## creates 32-bit immediates via AUGS/AUGD
- Each AUG instruction adds +2 clock cycles; augmentation is consumed by the next instruction
- PTRA and PTRB support post-modify (PTRx++), pre-modify (++PTRx), and indexed (PTRx[n]) forms
- The SCALE factor (1/2/4) depends on instruction: byte=1, word=2, long=4
- Non-updating index range: -32 to +31; updating index range: -16 to +16
- SETQ block transfers override the index field; pointer updates by total transfer size
- SILICON BUG: ALTx/AUGS between SETQ and PTRx transfer breaks pointer update
- SILICON BUG: AUGS affects immediate operands in intervening ALTx instructions

Part II: Instruction Set Reference

Instruction Categories

This chapter defines the instruction categories used throughout Part II. Each category groups instructions by their primary function. Click any category name in the instruction entries to return here for an overview, or click any instruction mnemonic to jump to its detailed reference.

> **Reading the Encoding Tables:** For help understanding the instruction encoding tables in this section (EEEE condition codes, CZI flag effects, opcode fields), see Chapter 2: The Instruction Format.

Arithmetic Operations

Arithmetic instructions perform mathematical and logical operations on register values. This includes addition, subtraction, multiplication, comparisons, bitwise operations (AND, OR, **XOR**), bit manipulation, shifts, rotates, and data movement. This is the largest instruction category.

Data Movement: MOV, LOC

Addition/Subtraction: ADD, ADDS, ADDSX, ADDX, SUB, SUBR, SUBS, SUBSX, SUBX

Negation/Absolute: ABS, NEG, NEGC, NEGNC, NEGNZ, NEGZ

Multiplication: MUL, MULS, SCA, SCAS

Comparisons: CMP, CMPM, CMPR, CMPS, CMPSUB, CMPSX, CMPX, TEST, TESTN

Min/Max: FGE, FGES, FLE, FLES

Modular Arithmetic: INCMOD, DECMOD

Bitwise Logic: AND, ANDN, OR, XOR, NOT, XORO32

Bit Field Operations: BITC, BITH, BITL, BITNC, BITNOT, BITNZ, BITRND, BITZ, TESTB, TESTBN

Bit Utilities: BMASK, DECOD, ENCOD, ONES, REV, SIGNX, ZEROX

Shifts: SHL, SHR, SAL, SAR

Rotates: ROL, ROR, RCL, RCR, RCZL, RCZR

Byte/Word/Nibble Access: GETBYTE, GETNIB, GETWORD, SETBYTE, SETNIB, SETWORD, ROLBYTE, ROLNIB, ROLWORD

Byte/Word Packing: MOVBYTS, SPLITB, SPLITW, MERGEB, MERGEW

Mux Operations: MUXC, MUXNC, MUXNZ, MUXZ, MUXQ, MUXNIBS, MUXNITS

Conditional Sum: SUMC, SUMNC, SUMNZ, SUMZ

Flag Operations: WRC, WRNC, WRNZ, WRZ, MODC, MODZ, MODCZ

Instruction Field Modification: SETD, SETS, SETR

CRC: CRCBIT, CRCNIB

Graphics: RGBEXP, RGBSQZ

Shuffling: SEUSSF, SEUSSR

Branching and Flow Control

Branch instructions control program flow by modifying the program counter. This category includes conditional and unconditional jumps, subroutine calls using stack or pointer registers, returns from subroutines and interrupts, **AND** instruction skipping/repeating mechanisms.

CALL, CALLA, CALLB, CALLD, CALLPA, CALLPB, DJF, DJNF, DJNZ, DJZ, EXECF, IJNZ, IJZ, JMP, JMPREL, REP, RESI0, RESI1, RESI2, RESI3, RET, RETA, RETB, RETI0, RETI1, RETI2, RETI3, SKIP, SKIPF, TJF, TJNF, TJNS, TJNZ, TJS, TJV, TJZ

Hub Memory Access

Hub memory instructions transfer data between cog registers and the shared 512KB hub RAM. This includes **BYTE**, **WORD**, and **LONG** access with various addressing modes, pointer-based operations using PTRA/PTRB, and high-speed FIFO streaming for bulk data transfers.

FBLOCK, GETPTR, POPA, POPB, PUSHA, PUSHB, RDBYTE, RDFAST, RDLONG, RDWORD, RFBYTE, RFLONG, RFVAR, RFVARS, RFWORD, WFBYTE, WFLONG, WFWORD, WMLONG, WRBYTE, WRFAST, WRLONG, WRWORD

Lookup Table

Lookup table (LUT) instructions access the 512-long LUT memory private to each cog. The LUT provides fast table lookups, additional register storage, and can be shared between adjacent cog pairs for inter-cog communication.

RDLUT, SETLUTS, WRLUT

Pin I/O and Smart Pins

Pin instructions control the P2's 64 I/O pins. Basic pin operations set direction (input/output) and output level (high/low). Smart pin instructions configure and communicate with the autonomous smart pin state machines that can perform complex I/O functions independent of cog processing.

Direction Control: DIRC, DIRH, DIRL, DIRNC, DIRNOT, DIRNZ, DIRRND, DIRZ

Output Control: OUTC, OUTH, OUTL, OUTNC, OUTNOT, OUTNZ, OUTRND, OUTZ

Drive (Direction + Output): DRVC, DRVH, DRVL, DRVNC, DRVNOT, DRVNZ, DRVRND, DRVZ

Float (Input with Preset): FLTC, FLTH, FLTL, FLTNC, FLTNOT, FLTNZ, FLTRND, FLTZ

Pin Testing: TESTP, TESTPN

Smart Pin Control: AKPIN, RDPIN, RQPIN, WRPIN, WXPIN, WYPIN

Oscilloscope/DAC: GETSCP, SETSCP, SETDACS

Events and Timing

Event instructions monitor and respond to system events including counter/timer triggers, smart pin signals, FIFO status, streamer conditions, and inter-cog attention signals. They provide configuration, polling, waiting, and conditional branching mechanisms for synchronization.

Configuration: ADDCT1, ADDCT2, ADDCT3, SETPAT, SETSE1, SETSE2, SETSE3, SETSE4

Inter-COG: COGATN

Polling: POLLATN, POLLCT1, POLLCT2, POLLCT3, POLLFBW, POLLINT, POLLPAT, POLLQMT, POLLSE1, POLLSE2, POLLSE3, POLLSE4, POLLXFI, POLLXMT, POLLXRL, POLLXRO

Waiting: WAITATN, WAITCT1, WAITCT2, WAITCT3, WAITFBW, WAITINT, WAITPAT, WAITSE1, WAITSE2, WAITSE3, WAITSE4, WAITXFI, WAITXMT, WAITXRL, WAITXRO

Branch on Event Set: JATN, JCT1, JCT2, JCT3, JFBW, JINT, JPAT, JQMT, JSE1, JSE2, JSE3, JSE4, JXFI, JXMT, JXRL, JXRO

Branch on Event Clear: JNATN, JNCT1, JNCT2, JNCT3, JNFBW, JNINT, JNPAT, JNQMT, JNSE1, JNSE2, JNSE3, JNSE4, JNXFI, JNXMT, JNXRL, JNXRO

Interrupts

Interrupt instructions control the cog's three-level interrupt system (INT1, INT2, INT3) plus the debug interrupt (INT0). This includes enabling/disabling interrupts, configuring interrupt sources, triggering software interrupts, and managing breakpoints for debugging.

ALLOWI, BRK, COGBRK, GETBRK, NIXINT1, NIXINT2, NIXINT3, SETINT1, SETINT2, SETINT3, STALLI, TRGINT1, TRGINT2, TRGINT3

COG Control and Locks

COG control instructions manage cog operations including starting and stopping cogs, querying cog identity, and configuring hub-level system settings. Lock instructions provide mutex-style synchronization primitives for safe inter-cog resource sharing.

COGID, COGINIT, COGSTOP, HUBSET, LOCKNEW, LOCKREL, LOCKRET, LOCKTRY

CORDIC Coprocessor

CORDIC (Coordinate Rotation Digital Computer) instructions provide hardware-accelerated mathematical operations. The dedicated coprocessor performs multiplication, division, square root, trigonometric functions, logarithms, and coordinate transformations with high precision.

GETQX, GETQY, QDIV, QEXP, QFRAC, QLOG, QMUL, QROTATE, QSQRT, QVECTOR

Streamer

Streamer instructions control the cog's dedicated DMA engine that autonomously transfers data between hub memory, LUT, and I/O pins. The streamer is essential for high-bandwidth applications like video output, audio streaming, and bulk data movement.

GETXACC, SETXFRQ, XCONT, XINIT, XSTOP, XZERO

Color Space and Pixel Operations

Color space and pixel instructions provide hardware-accelerated graphics processing. The colorspace converter transforms between color representations (RGB, YUV). The pixel mixer performs alpha blending, color addition, and format conversions for video and graphics applications.

ADDPIX, BLNPIX, MIXPIX, MULPIX, SETCFRQ, SETCI, SETCMOD, SETCQ, SETCY, SETPIV, SETPIX

Instruction Modification

Instruction modification instructions (also known as register indirection) dynamically alter subsequent instructions by changing their source, destination, or bit index fields before execution. They enable register arrays, computed addressing, and self-modifying code patterns essential for efficient data structure access.

ALTB, ALTD, ALTGB, ALTGN, ALTGW, ALTI, ALTR, ALTS, ALTSB, ALTSN, ALTSW

Miscellaneous

Miscellaneous instructions provide utility functions including immediate value extension (AUGS/AUGD), stack operations, random number generation, system timer access, and delay insertion.

AUGD, AUGS, GETCT, GETRND, NOP, POP, PUSH, SETQ, SETQ2, WAITX

Instructions: A

This section contains all PASM2 instructions beginning with the letter A.

ABS

Absolute Value
Arithmetic Operations - Returns the absolute (non-negative) value of a signed number.

ABS *Dest, {#}Src* **{WC|WZ|WCZ}**

ABS *Dest* **{WC|WZ|WCZ}**

Result: Absolute Src (or Dest) value is stored in Dest.

- Dest is the register in which to write the absolute value of Dest or Src.
- Src is an optional register, 9-bit literal, or 32-bit augmented literal whose absolute value is written to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0110010	CZI	DDDDDDDDD	SSSSSSSSS	S[31]	result == 0	D	2
EEEE	0110010	CZ0	DDDDDDDDD	DDDDDDDDD	D[31]	result == 0	D	2

Related: NEG

Explanation:

ABS determines the absolute value of Src or Dest and writes the result into Dest. The first syntax form computes the absolute value of Src, while the second syntax form (without Src) computes the absolute value of Dest itself.

If the WC or WCZ effect is specified, the C flag is set (1) if the original Src or Dest value was negative (the sign bit was 1), or is cleared (0) if it was positive. This preserves information about the original sign of the value.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result is zero, or is cleared (0) if it is non-zero.

Literal Src values are zero-extended, so **ABS** is best used with register Src (or augmented Src) values for meaningful signed operations.

ADD

Add Unsigned
Arithmetic Operations - **ADDS** two unsigned 32-bit values.

ADD *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Sum of unsigned Src and unsigned Dest is stored in Dest.

- Dest is a register containing the value to add Src to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is added into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001000	CZI	DDDDDDDDD	SSSSSSSSS	carry of (D + S)	result == 0	D	2

Related: ADDX, ADDS, ADDSX, SUB

Explanation:

ADD sums the two unsigned values of Dest and Src together and stores the result into the Dest register.

If the WC or WCZ effect is specified, the C flag is set (1) if the summation results in a 32-bit overflow (unsigned carry), or is cleared (0) if no overflow. This indicates that the result exceeded the maximum unsigned 32-bit value of $FFFF_FFFF.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result of Dest + Src equals zero, or is cleared (0) if it is non-zero.

To add unsigned multi-long values (64-bit or larger), use ADD for the least significant **LONG**, then **ADDX** for each subsequent **LONG**. **ADDX** carries the overflow from the previous addition into the current one. For example, to add two 64-bit values:

```
1         ADD     value_lo, addend_lo  wc    ' Add low longs, capture carry
2         ADDX    value_hi, addend_hi        ' Add high longs with carry-in
```

ADD and **ADDX** are also used for adding signed multi-long values, with **ADDSX** ending the sequence to properly handle sign extension.

ADDCT1 / ADDCT2 / ADDCT3

Add and Set Counter Event Trigger

Events and Timing - Sets counter event trigger to Dest + Src for time-based events.

ADDCT1 *Dest, {#}Src*

ADDCT2 *Dest, {#}Src*

ADDCT3 *Dest, {#}Src*

Result: The Src value is added into Dest and the result is also stored in the hidden CTn event trigger register.

- Dest is a register containing the value to add Src to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is added into Dest.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010011	00I	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1010011	01I	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1010011	10I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: POLLCT1/2/3, WAITCT1/2/3, JCT1/2/3, JNCT1/2/3

Explanation:

ADDCT1, ADDCT2, and ADDCT3 set their respective hidden counter event trigger registers to the value of Dest + Src. The result is also written to Dest. These instructions are used to schedule time-based events that will trigger when the System Counter (CT) reaches the specified value.

The P2 provides three independent counter event triggers (CT1, CT2, CT3), allowing a cog to manage multiple simultaneous time-based operations. Use the corresponding POLLCTn, WAITCTn, JCTn, and JNCTn instructions to process each counter's time-based events. This enables precise timing control for periodic operations, delays, and synchronized activities.

ADDPIX

Add Pixels
Color Space and Pixel Operations - **ADDS** color channel bytes with saturation.

ADDPIX *Dest, {#}Src*

Result: Src color value bytes are added into Dest color value bytes with full saturation.

- Dest is a register containing the RGB color value to add Src to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose RGB color value bytes are added into Dest.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010010	00I	DDDDDDDDD	SSSSSSSSS	—	—	D	7

Related: MULPIX, BLNPIX, MIXPIX

Explanation:

ADDPIX sums the individual **BYTE** fields of Src into those of Dest and stores the result in the Dest register. Each of the four bytes of the 32-bit register is treated as a separate field — for 8:8:8:8 pixel data these are the red, green, blue, and alpha/fourth bytes — and each is saturated independently to prevent wraparound.

Saturation means that if the sum of a color channel exceeds 255, the result is clamped to 255 rather than wrapping around to a low value. This prevents color distortion when combining bright colors and produces visually correct results for color blending operations.

The instruction processes all four **BYTE** fields (the three RGB color channels plus the alpha/fourth **BYTE**) in parallel, completing in 7 clock cycles.

ADDS

Add Signed
Arithmetic Operations - **ADDS** two signed 32-bit values.

ADDS *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Sum of signed Src and signed Dest is stored in Dest.

- Dest is a register containing the value to add Src to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is added into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001010	CZI	DDDDDDDDD	SSSSSSSSS	correct sign of (D + S)	result == 0	D	2

Related: ADD, ADDX, ADDSX, SUBS

Explanation:

ADDS sums the two signed values of Dest and Src together and stores the result into the Dest register.

If Src is a 9-bit literal, its value is interpreted as positive (0-511; it is not sign-extended). Use ##Value (or insert a prior **AUGS** instruction) for a 32-bit signed value, negative or positive.

If the WC or WCZ effect is specified, the C flag is set (1) if the summation results in a signed overflow (signed carry), or is cleared (0) if no overflow. Signed overflow occurs when the result cannot be represented in 32 bits using two's complement notation.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result of Dest + Src is zero, or is cleared (0) if it is non-zero.

To add signed multi-long values, use ADD (not **ADDS**) followed possibly by **ADDX**, and finally **ADDSX** as the last operation to properly handle sign extension.

ADDSX

Add Signed Extended
Arithmetic Operations - Extended signed addition for multi-long values.

ADDSX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Sum of signed Src plus C and signed Dest is stored in Dest.

- Dest is a register containing the value to add Src plus C to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value plus C is added into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001011	CZI	DDDDDDDDD	SSSSSSSSS	correct sign of (D + S + C)	Z AND (result == 0)	D	2

Related: ADD, ADDX, ADDS, SUBSX

Explanation:

ADDSX sums the signed values of Dest and Src plus C together and stores the result into the Dest register. The **ADDSX** instruction is used to perform signed multi-long (extended) addition, such as 64-bit addition.

If the WC or WCZ effect is specified, the C flag is set (1) if the result is negative (Result[31] = 1), or is cleared (0) if positive. Use WC or WCZ on preceding ADD and **ADDX** instructions for proper final C flag state.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Z was previously set and the result of Dest + Src + C is zero, or it is cleared (0) if non-zero. Use WZ or WCZ on preceding ADD and **ADDX** instructions for proper final Z flag state. This allows detection of a zero result across the entire multi-long value.

To add signed multi-long values, use ADD (not **ADDS**) followed possibly by **ADDX**, and finally **ADDSX** as the last operation. **ADDSX** properly handles the sign extension for the most significant portion of the multi-long value.

ADDX

Add Unsigned Extended
Arithmetic Operations - Extended unsigned addition for multi-long values.

ADDX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Sum of unsigned Src plus C and unsigned Dest is stored in Dest.

- Dest is a register containing the value to add Src plus C to, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value plus C is added into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001001	CZI	DDDDDDDDD	SSSSSSSSS	carry of (D + S + C)	Z AND (result == 0)	D	2

Related: ADD, ADDS, ADDSX, SUBX

Explanation:

ADDX sums the unsigned values of Dest and Src plus C together and stores the result into the Dest register. The **ADDX** instruction is used to perform unsigned multi-long (extended) addition, such as 64-bit addition.

If the WC or WCZ effect is specified, the C flag is set (1) if the summation resulted in an unsigned carry, or is cleared (0) if no carry. Use WC or WCZ on preceding ADD and **ADDX** instructions for proper final C flag state. If C is set after the last **ADDX** in a multi-long addition, it indicates unsigned overflow.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Z was previously set and the result of Dest + Src + C is zero, or it is cleared (0) if non-zero. Use WZ or WCZ on preceding ADD and **ADDX** instructions for proper final Z flag state. This allows detection of a zero result across the entire multi-long value.

To add unsigned multi-long values, use ADD followed by one or more **ADDX** instructions. Each **ADDX** carries the overflow from the previous addition into the current one.

AKPIN

Acknowledge Smart Pin
Pin I/O and Smart Pins - Acknowledges Smart Pin(s) to allow future events.

AKPIN *{#}Src*

Result: One or more Smart Pins is acknowledged; lowering their corresponding IN signal(s).

- Src is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the Smart Pin(s) to acknowledge.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100000	01I	000000001	SSSSSSSSS	—	—	—	2

Related: WRPIN, WXPIN, WYPIN, RDPIN

Explanation:

AKPIN acknowledges the Smart Pin(s) designated by Src. This lowers the corresponding IN signal(s) so that future Smart Pin events may raise them again later.

Src[5:0] indicates the pin number (0-63). For a range of Smart Pins, Src[5:0] indicates the first pin number (0-63) and Src[10:6] indicates how many contiguous pins beyond the first should be affected (1-31).

A 9-bit literal Src is enough to express the starting pin (Src[5:0]) and a range of up to 8 contiguous pins (Src[8:6]). If needed, use the augmented literal feature (##Src) to augment Src to the required 11-bit literal value, which automatically inserts an **AUGS** instruction prior.

When Src is a register, the register's value bits [10:0] are used as-is to form the 11-bit Smart Pin range, unless a **SETQ** instruction immediately precedes the **AKPIN** instruction; in that case, **SETQ**'s Dest[4:0] substitutes for value bits[10:6] for **AKPIN**'s use.

The range calculation (from Src[5:0] up to Src[5:0]+Src[10:6]) wraps within the same 32-pin group (DIRA or DIRB); it will not cross the port boundary.

ALLOWI

Allow Interrupts
Interrupts - Re-enables interrupt handling after **STALLI**.

ALLOWI

Result: Any stalled and future interrupts are allowed.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	000100000	000100100	—	—	—	2

Related: STALLI

Explanation:

ALLOWI re-enables interrupt branching; the default on COG start. **ALLOWI** is the complement of the **STALLI** instruction. Both are used to protect short, vital sections of main code from timing jitter or state loss caused by asynchronous interrupt handling.

When **ALLOWI** is executed, any interrupts that were stalled by a previous **STALLI** instruction are allowed to proceed, and future interrupts are also enabled. This allows the COG to respond to interrupt events normally.

ALTB

Alter Bit

Register Indirection - Alters next BITxxx instruction's target bit address.

ALTB *Dest, {#}Src*

ALTB *Dest*

Result: The next instruction's pipelined Dest value is altered to be (Src + Dest[13:5]) & $1FF, or just Dest[13:5] for syntax 2.

- Dest is the register whose 14-bit value is the index, or the full bit address, for the BITxxx instruction to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base **LONG** address (Src[8:0]; added to index (Dest[13:5]) for BITxxx) and also an optional auto-indexer value (Src[17:9]; added to Dest at the end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001100	11I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001100	111	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTD, ALTS, ALTR, ALTI

Explanation:

ALTB should be followed by a BITxxx instruction. It modifies the BITxxx instruction's Dest value, enabling code to iterate through multiple bits of data across a range of register RAM.

BITxxx's Dest value is changed to (Src + Dest[13:5]) & $1FF (for syntax 1), or to Dest[13:5] (for syntax 2). Dest[13:5] corresponds to the target **LONG** register's 9-bit address and Dest[4:0] is the bit ID within it; values of 0-31 identify individual bits, by position, in least-significant bit order.

Iteratively executing **ALTB** followed by a BITxxx instruction, and each time incrementing **ALTB**'s 14-bit Dest value by one, effectively writes a stream of bit values to register RAM as if it were all made of bit-sized registers.

Warning: BITxxx instructions optionally operate on a range of bits, encoded in the Src value. They don't limit themselves to only reading Src[4:0] for the bit number. For this reason, care must be taken when using **ALTB** with BITxxx or the index value (often used for the Src of the altered instruction) will be misinterpreted as multiple bits to affect. One way to solve this is to use a **SETQ** #0 followed by the **ALTB** then BITxxx instructions to force BITxxx's Src[9:5] bits to 0; that is, no extra bits beyond the single bit described by Src[4:0].

In syntax 1, Src consists of two 9-bit fields: a base address (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base is the register RAM address where the series of bits begins. **ALTB ADDS** the long index (Dest[13:5]) to the base (Src[8:0]) to locate the register holding the target bit. The bit ID (Dest[4:0]) identifies the bit's position within that **LONG** register. At the end of **ALTB** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the 14-bit index (Dest) for a future **ALTB**+BITxxx iteration.

In syntax 2, Dest serves as the full bit address. It is the same format as in syntax 1, but represents the target **LONG**'s absolute address and its bit index instead of the long's relative index (to add to a base) and bit index.

The instruction following **ALTB** is shielded from interrupt. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTD

Alter Destination
Register Indirection - Alters next instruction's Dest field.

ALTD *Dest, {#}Src*

ALTD *Dest*

Result: The next instruction's pipelined Dest value is altered to be (Src + Dest) & $1FF, or just Dest[8:0] in syntax 2.

- Dest is the register whose 9-bit value is the offset, or the full value, for the next instruction to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base (Src[8:0]; added to offset (Dest) for the next instruction) and also an optional auto-indexer value (Src[17:9]; added to Dest at the end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001100	01I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001100	011	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTS, ALTR, ALTB, ALTI

Explanation:

ALTD modifies the next instruction's Dest value to be (Src + Dest) & $1FF (for syntax 1), or to Dest[8:0] (for syntax 2).

In syntax 1, Src consists of two 9-bit fields: a base value (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base represents a starting point. **ALTD ADDS** the offset (Dest[8:0]) to the base (Src[8:0]) to determine the next instruction's Dest value. At the end of **ALTD** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the offset (Dest) for a future **ALTD**+instruction iteration.

In syntax 2, Dest serves as the full value. It is used as-is for the next instruction's substitute Dest value.

The instruction following **ALTD** is shielded from interrupt. **ALTD** alters the next instruction regardless of its kind. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

Pitfall (Silicon Bug): **ALTD** placed between **SETQ**/SETQ2 and **RDLONG**/WRLONG/WMLONG cancels the block-size PTRx delta calculation. The block transfer completes correctly, but PTRx advances by only a single-long delta.

Pitfall (Silicon Bug): When **ALTD** uses an immediate #S operand and an **AUGS** is active (targeting a later instruction), **ALTD**'s #S operand also receives the augmented value without canceling it. Use a register for **ALTD**'s S operand when **AUGS** is active.

ALTGB

Alter Get **BYTE**

Register Indirection - Alters next **GETBYTE**/ROLBYTE instruction's target **BYTE**.

ALTGB *Dest, {#}Src*

ALTGB *Dest*

Result: The next instruction's pipelined Src and Num fields are altered to be (Src + Dest[10:2]) & $1FF, or just Dest[10:2] for syntax 2, and Dest[1:0], respectively.

- Dest is the register whose 11-bit value is the index, or the full **BYTE** address, for the **GETBYTE** / **ROLBYTE** instruction to read.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base **LONG** address (Src[8:0]; added to index (Dest[10:2]) for **GETBYTE** / **ROLBYTE**) and also an optional auto-indexer value (Src[17:9]; added to Dest at end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001011	01I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001011	011	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTGN, ALTGW, ALTSB, GETBYTE, ROLBYTE

Explanation:

ALTGB should be followed by **GETBYTE** or **ROLBYTE**. It modifies the **GETBYTE** / **ROLBYTE** instruction's Src and Num values, enabling code to iterate through multiple bytes of data across a range of register RAM.

GETBYTE / **ROLBYTE**'s Src value is changed to (Src + Dest[10:2]) & $1FF (for syntax 1), or to Dest[10:2] (for syntax 2), and its Num value is changed to Dest[1:0]. Dest[10:2] corresponds to the target **LONG** register's 9-bit address and Dest[1:0] is the byte ID within it; values of 0-3 identify individual bytes, by position, in least-significant byte order.

Iteratively executing **ALTGB** followed by **GETBYTE** or **ROLBYTE**, and each time incrementing **ALTGB**'s 11-bit Dest value by one, effectively reads a stream of byte values from register RAM as if it were all made of byte-sized registers.

In syntax 1, Src consists of two 9-bit fields: a base address (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base is the register RAM address where the series of bytes begins. **ALTGB ADDS** the long index (Dest[10:2]) to the base (Src[8:0]) to locate the register holding the target **BYTE**. The byte ID (Dest[1:0])

identifies the byte's position within that **LONG** register. At the end of **ALTGB** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the 11-bit index (Dest) for a future **ALTGB**+GETBYTE or **ROLBYTE** iteration.

In syntax 2, Dest serves as the full **BYTE** address. It is the same format as in syntax 1, but represents the target **LONG**'s absolute address and its **BYTE** index instead of the long's relative index (to add to a base) and **BYTE** index.

The instruction following **ALTGB** is shielded from interrupt. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTGN

Alter Get Nibble
Register Indirection - Alters next **GETNIB**/ROLNIB instruction's target nibble.

ALTGN *Dest, {#}Src*

ALTGN *Dest*

Result: The next instruction's pipelined Src and Num values are altered to be (Src + Dest[11:3]) & $1FF, or just Dest[11:3] for syntax 2, and Dest[2:0], respectively.

- Dest is the register whose 12-bit value is the index, or the full nibble address, for the next **GETNIB** / **ROLNIB** instruction to read.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base **LONG** address (Src[8:0]; added to index (Dest[11:3]) for **GETNIB** / **ROLNIB**) and also an optional auto-indexer value (Src[17:9]; added to Dest at end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001010	11I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001010	111	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTGB, ALTGW, ALTSN, GETNIB, ROLNIB

Explanation:

ALTGN should be followed by **GETNIB** or **ROLNIB**. It modifies the **GETNIB** / **ROLNIB** instruction's Src and Num values, enabling code to iterate through multiple nibbles of data across a range of register RAM.

GETNIB / **ROLNIB**'s Src value is changed to (Src + Dest[11:3]) & $1FF (for syntax 1), or to Dest[11:3] (for syntax 2), and its Num value is changed to Dest[2:0]. Dest[11:3] corresponds to the target **LONG** register's 9-bit address and Dest[2:0] is the nibble ID within it; values of 0-7 identify individual nibbles, by position, in least-significant nibble order.

Iteratively executing **ALTGN** followed by **GETNIB** or **ROLNIB**, and each time incrementing **ALTGN**'s 12-bit Dest value by one, effectively reads a stream of nibble values from register RAM as if it were all made of nibble-sized registers.

In syntax 1, Src consists of two 9-bit fields: a base address (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base is the register RAM address where the series of nibbles begins. **ALTGN ADDS** the long index (Dest[11:3]) to the base (Src[8:0]) to locate the register holding the target nibble. The nibble ID (Dest[2:0]) identifies the nibble's position within that **LONG** register. At the end of **ALTGN** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the 12-bit index (Dest) for a future **ALTGN**+GETNIB or **ROLNIB** iteration.

In syntax 2, Dest serves as the full nibble address. It is the same format as in syntax 1, but represents the target **LONG**'s absolute address and its nibble index instead of the long's relative index (to add to a base) and nibble index.

The instruction following **ALTGN** is shielded from interrupt. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTGW

Alter Get **WORD**

Register Indirection - Alters next **GETWORD**/ROLWORD instruction's target **WORD**.

ALTGW *Dest, {#}Src*

ALTGW *Dest*

Result: The next instruction's pipelined Src and Num fields are altered to be (Src + Dest[9:1]) & $1FF, or just Dest[9:1] for syntax 2, and Dest[0], respectively.

- Dest is the register whose 10-bit value is the index, or the full **WORD** address for the **GETWORD** / **ROLWORD** instruction to read.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base **LONG** address (Src[8:0]; added to index (Dest[9:1]) for **GETWORD** / **ROLWORD**) and also an optional auto-indexer value (Src[17:9]; added to Dest at end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001011	11I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001011	111	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTGB, ALTGN, ALTSW, GETWORD, ROLWORD

Explanation:

ALTGW should be followed by **GETWORD** or **ROLWORD**. It modifies the **GETWORD** / **ROLWORD** instruction's Src and Num values, enabling code to iterate through multiple words of data across a range of register RAM.

GETWORD / **ROLWORD**'s Src value is changed to (Src + Dest[9:1]) & $1FF (for syntax 1), or to Dest[9:1] (for syntax 2), and its Num value is changed to Dest[0]. Dest[9:1] corresponds to the target **LONG** register's 9-bit address and Dest[0] is the word ID within it; values of 0-1 identify individual words, by position, in least-significant **WORD** order.

Iteratively executing **ALTGW** followed by **GETWORD** or **ROLWORD**, and each time incrementing **ALTGW**'s 10-bit Dest value by one, effectively reads a stream of word values from register RAM as if it were all made of word-sized registers.

In syntax 1, Src consists of two 9-bit fields: a base address (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base is the register RAM address where the series of words begins. **ALTGW ADDS** the long index (Dest[9:1]) to the base (Src[8:0]) to locate the register holding the target **WORD**. The word ID (Dest[0]) identifies the word's position within that **LONG** register. At the end of **ALTGW** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the 10-bit index (Dest) for a future **ALTGW**+GETWORD or **ROLWORD** iteration.

In syntax 2, Dest serves as the full **WORD** address. It is the same format as in syntax 1, but represents the target **LONG**'s absolute address and its **WORD** index instead of the long's relative index (to add to a base) and **WORD** index.

The instruction following **ALTGW** is shielded from interrupt. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTI

Alter Instruction
Register Indirection - Alters multiple fields of the next instruction.

ALTI *Dest, {#}Src*

ALTI *Dest*

Result: The next instruction's pipelined field values are substituted from the Dest template, and Dest is modified per Src configuration.

- Dest is the register whose value contains one or more of the next instruction's field substitutes or an entire 32-bit opcode for full substitution.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value describes the substitutions and Dest modifications to perform.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001101	00I	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001101	001	DDDDDDDDD	101100100	—	—	D	2

Related: SETD, SETS, SETR, ALTD, ALTS, ALTR

Explanation:

ALTI substitutes fields from Dest for one or more of the next instruction's pipelined Dest, Src, Result, Instr, FX, and/or Cond values, and ALTI's Dest is then modified per Src configuration (syntax 1), or the entire Dest opcode (instruction) is executed in place of the next instruction (syntax 2).

The Dest register contains the **ALTI** template; a 32-bit value with format similar to an opcode with Condition (31:28), Result (27:19), Indirect I (18), Dest D (17:9), and Source S (8:0) fields.

In syntax 1, Src consists of six 3-bit fields (%rrr_ddd_sss_RRR_DDD_SSS) that describe field substitution and Dest modification. The mask size fields (%rrr, %ddd, %sss) control increment/decrement masking from

unlimited 9-bit (000) to 2-bit (111). The control fields (%RRR, %DDD, %SSS) control field substitution and adjustment.

In syntax 2, Dest serves as the full opcode value. It is executed as-is in place of the next instruction and Dest remains unaltered afterward.

The instruction following **ALTI** is shielded from interrupt. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTR

Alter Result
Register Indirection - Alters next instruction's result write address.

ALTR *Dest, {#}Src*

ALTR *Dest*

Result: The next instruction's pipelined Result address (Dest address by default) is altered to be (Src + Dest) & $1FF, or just Dest[8:0] in syntax 2.

- Dest is the register whose 9-bit value is the offset, or the full value, for the next instruction to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base (Src[8:0]; added to offset (Dest) for the next instruction) and also an optional auto-indexer value (Src[17:9]; added to Dest at the end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001100	00I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001100	001	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTD, ALTS, ALTB, ALTI

Explanation:

ALTR modifies the next instruction's Result address to be (Src + Dest) & $1FF (for syntax 1), or to Dest[8:0] (for syntax 2).

The Result address is the Dest address by default. It identifies where the result value from the instruction's execution is written at the end of execution. During execution, the pipeline holds an instruction's Dest address and the Result address as two separate entities, normally set to the same location. **ALTR** causes the next instruction's Result to redirect to a different address; changing an instruction from a destructive (operand overwriting) operation to a non-destructive (operand preserving) operation.

In syntax 1, Src consists of two 9-bit fields: a base value (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base represents a starting point. **ALTR ADDS** the offset (Dest[8:0]) to the base (Src[8:0]) to determine the next instruction's Result address. At the end of **ALTR** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the offset (Dest) for a future **ALTR**+instruction iteration.

In syntax 2, Dest serves as the full value. It is used as-is for the next instruction's substitute Result address.

The instruction following **ALTR** is shielded from interrupt. **ALTR** alters the next instruction regardless of its kind. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTS

Alter Source

Register Indirection - Alters next instruction's Src field.

ALTS *Dest, {#}Src*

ALTS *Dest*

Result: The next instruction's pipelined Src value is altered to be (Src + Dest) & $1FF, or just Dest[8:0] in syntax 2.

- Dest is the register whose 9-bit value is the offset, or the full value, for the next instruction to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base (Src[8:0]; added to offset (Dest) for the next instruction) and also an optional auto-indexer value (Src[17:9]; added to Dest at the end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001100	10I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001100	101	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTD, ALTR, ALTB, ALTI

Explanation:

ALTS modifies the next instruction's Src value to be (Src + Dest) & $1FF (for syntax 1), or to Dest[8:0] (for syntax 2).

In syntax 1, Src consists of two 9-bit fields: a base value (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base represents a starting point. **ALTS ADDS** the offset (Dest[8:0]) to the base (Src[8:0]) to determine the next instruction's Src value. At the end of **ALTS** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the offset (Dest) for a future **ALTS**+instruction iteration.

In syntax 2, Dest serves as the full value. It is used as-is for the next instruction's substitute Src value.

The instruction following **ALTS** is shielded from interrupt. **ALTS** alters the next instruction regardless of its kind. Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTSB

Alter Set **BYTE**

Register Indirection - Alters next **SETBYTE** instruction's target **BYTE**.

ALTSB *Dest, {#}Src*

ALTSB *Dest*

Result: The next instruction's pipelined Dest and Num values are altered to be (Src + Dest[10:2]) & $1FF (syntax 1), or just Dest[10:2] (syntax 2), and Num is set to Dest[1:0]. Dest is post-adjusted by auto-indexer.

- Dest is the register whose 11-bit value is the index (Dest[10:2] = **LONG** address, Dest[1:0] = **BYTE** ID) or the full **BYTE** address for **SETBYTE** to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal containing base **LONG** address (Src[8:0]) and optional auto-indexer value (Src[17:9]).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001011	00I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001011	001	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTGB, ALTSN, ALTSW, SETBYTE

Explanation:

ALTSB should be followed by **SETBYTE**. It modifies the **SETBYTE** instruction's Dest and Num values, enabling code to iterate through multiple bytes across register RAM.

SETBYTE's Dest is changed to (Src + Dest[10:2]) & $1FF (syntax 1), or to Dest[10:2] (syntax 2), and its Num value is changed to Dest[1:0]. Dest[10:2] is the target **LONG** register's 9-bit address; Dest[1:0] is the byte ID (0-3) within it.

Iteratively executing **ALTSB** followed by **SETBYTE** while incrementing the 11-bit Dest value writes a stream of bytes to register RAM as if it were byte-sized registers.

In syntax 1, Src contains a base address (Src[8:0]) and signed auto-indexer (Src[17:9]). In syntax 2, Dest serves as the full **BYTE** address.

The instruction following **ALTSB** is shielded from interrupt. **ALTSB** alters the next instruction regardless of its kind (intended for **SETBYTE**). Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTSN

Alter Set Nibble

Register Indirection - Alters next **SETNIB** instruction's target nibble.

ALTSN *Dest, {#}Src*

ALTSN *Dest*

Result: The next instruction's pipelined Dest and Num values are altered to be (Src + Dest[11:3]) & $1FF, or just Dest[11:3] for syntax 2, and Dest[2:0], respectively.

- Dest is the register whose 12-bit value is the index, or the full nibble address, for the **SETNIB** instruction to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base **LONG** address (Src[8:0]; added to index (Dest[11:3]) for **SETNIB**) and also an optional auto-indexer value (Src[17:9]; added to Dest at the end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001010	10I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001010	101	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTGN, ALTSB, ALTSW, SETNIB

Explanation:

ALTSN should be followed by **SETNIB**. It modifies the **SETNIB** instruction's Dest and Num values, enabling code to iterate through multiple nibbles of data across a range of register RAM.

SETNIB's Dest value is changed to (Src + Dest[11:3]) & $1FF (for syntax 1), or to Dest[11:3] (for syntax 2), and its Num value is changed to Dest[2:0]. Dest[11:3] corresponds to the target **LONG** register's 9-bit address and Dest[2:0] is the nibble ID within it; values of 0-7 identify individual nibbles, by position, in least-significant nibble order.

Iteratively executing **ALTSN** followed by **SETNIB**, and each time incrementing **ALTSN**'s 12-bit Dest value by one, effectively writes a stream of nibble values to register RAM as if it were all made of nibble-sized registers.

In syntax 1, Src consists of two 9-bit fields: a base address (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base is the register RAM address where the series of nibbles begins. **ALTSN ADDS** the long index (Dest[11:3]) to the base (Src[8:0]) to locate the register holding the target nibble. The nibble ID (Dest[2:0]) identifies the nibble's position within that **LONG** register. At the end of **ALTSN** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the 12-bit index (Dest) for a future **ALTSN**+SETNIB iteration.

In syntax 2, Dest serves as the full nibble address. It is the same format as in syntax 1, but represents the target **LONG**'s absolute address and its nibble index instead of the long's relative index (to add to a base) and nibble index.

The instruction following **ALTSN** is shielded from interrupt. **ALTSN** alters the next instruction regardless of its kind (intended for **SETNIB**). Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

ALTSW

Alter Set **WORD**
Register Indirection - Alters next **SETWORD** instruction's target **WORD**.

ALTSW *Dest, {#}Src*

ALTSW *Dest*

Result: The next instruction's pipelined Dest and Num fields are altered to be (Src + Dest[9:1]) & $1FF, or just Dest[9:1] for syntax 2, and Dest[0], respectively.

- Dest is the register whose 10-bit value is the index, or the full **WORD** address, for the **SETWORD** instruction to operate on.
- Src is an optional register, 9-bit literal, or 18-bit augmented literal whose value contains a base **LONG** address (Src[8:0]; added to index (Dest[9:1]) for **SETWORD**) and also an optional auto-indexer value (Src[17:9]; added to Dest at end of execution).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001011	10I	DDDDDDDDD	SSSSSSSSS	—	—	D†	2
EEEE	1001011	101	DDDDDDDDD	000000000	—	—	D†	2

† Dest is post-adjusted by the auto-indexer value; the sign-extended Src[17:9]. In syntax 2, the auto-indexer value is 0.

Related: ALTGW, ALTSB, ALTSN, SETWORD

Explanation:

ALTSW should be followed by **SETWORD**. It modifies the **SETWORD** instruction's Dest and Num values, enabling code to iterate through multiple words of data across a range of register RAM.

SETWORD's Dest value is changed to (Src + Dest[9:1]) & $1FF (for syntax 1), or to Dest[9:1] (for syntax 2), and its Num value is changed to Dest[0]. Dest[9:1] corresponds to the target **LONG** register's 9-bit address and Dest[0] is the word ID within it; values of 0-1 identify individual words, by position, in least-significant **WORD** order.

Iteratively executing **ALTSW** followed by **SETWORD**, and each time incrementing **ALTSW**'s 10-bit Dest value by one, effectively writes a stream of word values to register RAM as if it were all made of word-sized registers.

In syntax 1, Src consists of two 9-bit fields: a base address (Src[8:0]) and a signed auto-indexer (Src[17:9]). The base is the register RAM address where the series of words begins. **ALTSW ADDS** the long index (Dest[9:1]) to the base (Src[8:0]) to locate the register holding the target **WORD**. The word ID (Dest[0]) identifies the word's position within that **LONG** register. At the end of **ALTSW** execution, the optional auto-indexer value (usually 0, 1, or -1) is added to the 10-bit index (Dest) for a future **ALTSW**+SETWORD iteration.

In syntax 2, Dest serves as the full **WORD** address. It is the same format as in syntax 1, but represents the target **LONG**'s absolute address and its **WORD** index instead of the long's relative index (to add to a base) and **WORD** index.

The instruction following **ALTSW** is shielded from interrupt. **ALTSW** alters the next instruction regardless of its kind (intended for **SETWORD**). Field value modification occurs in the instruction pipeline only; code is not altered, values do not persist. **SETQ**/SETQ2 does not affect ALTx instructions; the Q value passes through to the next instruction.

AND

Bitwise And
Arithmetic Operations - Performs bitwise AND between two values.

AND *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Bitwise AND of Dest and Src is stored in Dest.

- Dest is the register containing the value to bitwise AND with Src and is the destination in which to write the result.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value will be bitwise ANDed with Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0101000	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2

Related: ANDN, OR, XOR, TEST

Explanation:

AND performs a bitwise AND of the value in Src into that of Dest, storing the result in Dest. Each bit in the result is 1 only if the corresponding bits in both Dest and Src are 1.

Dest	Src	Result
0	0	0
0	1	0
1	0	0
1	1	1

If the WC or WCZ effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits, or is cleared (0) if it contains an even number of high bits. This parity calculation is useful for error detection.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if it is non-zero.

ANDN

And Not
Arithmetic Operations - Clears bits in Dest where Src bits are set.

ANDN *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Bitwise AND of Dest with inverse of Src is stored in Dest.

- Dest is the register containing the value to bitwise AND with the inverse of Src and is the destination in which to write the result.

- Src is a register, 9-bit literal, or 32-bit augmented literal whose inverse value will be bitwise ANDed with Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0101001	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2

Related: AND, OR, XOR, TEST

Explanation:

ANDN performs a bitwise AND of Dest with the inverse of Src (!Src), storing the result in Dest. This effectively clears bits in Dest wherever the corresponding bits in Src are set.

Dest	Src	Result
0	0	0
0	1	0
1	0	1
1	1	0

ANDN is particularly useful for clearing specific bits while leaving others unchanged. For example, to clear bits 7:4 of a register while preserving all other bits, use **ANDN** with a mask that has 1s in positions 7:4.

If the WC or WCZ effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits, or is cleared (0) if it contains an even number of high bits.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if it is non-zero.

ASMCLK

Set Clock Mode
COG Control and Locks - Configures system clock from CON symbols.

ASMCLK

Result: Configures the P2 system clock according to clock setup CON symbols.

- No operands. Clock configuration is read from CON symbols (`_clkfreq`, `_xtlfreq`, `_xinfreq`, `_rcslow`, `_rcfast`).
- Can be used with conditional prefix (IF_C, IF_NC, etc.).

Note: ASMCLK is a pseudo-instruction (macro) that expands to 1–6 real PASM instructions depending on the clock mode. It is not a hardware instruction with a fixed encoding.

Expansion:

Clock Mode	Expands To	Instructions
External crystal/oscillator with PLL	HUBSET, WAITX, HUBSET	3–6
RCSLOW (internal slow RC)	HUBSET #1	1
RCFAST (internal fast RC)	HUBSET #0	1

For external clock modes, the expansion sequence is:

```
1                HUBSET  ##clkmode_ & !%11       ' Start ext clock, RCFAST
2                WAITX   ##20_000_000/100        ' Wait ~10ms for stability
3                HUBSET  ##clkmode_              ' Switch to target mode
```

Related: HUBSET, WAITX

Explanation:

ASMCLK is a pseudo-instruction for PASM-only programs that sets the system clock mode based on clock configuration symbols defined in a CON block. When assembled, **ASMCLK** expands to the appropriate **HUBSET** and **WAITX** instructions needed to configure the clock.

The clock configuration is determined by these CON symbols:

- `_clkfreq` — Target clock frequency in Hz
- `_xtlfreq` — External crystal frequency (for crystal modes)
- `_xinfreq` — External clock input frequency (for external oscillator modes)
- `_rcslow` — Use internal slow RC oscillator (~20 kHz)
- `_rcfast` — Use internal fast RC oscillator (~20 MHz, default)

Modern Usage (v35v and later):

As of compiler version v35v (September 2022), **ASMCLK** is typically unnecessary. The compiler automatically prepends a 16-long clock-setter program to PASM-only programs that use non-RCFAST clock modes. This clock-setter configures the clock, relocates your program down by 16 longs, then executes it via `COGINIT #0,#0`.

To disable the automatic clock-setter and use **ASMCLK** manually, define:

```
1 CON
2   _AUTOCLK = 0                  ' Disable automatic clock-setter
```

Example:

```
CON
  _clkfreq = 200_000_000            ' 200 MHz target
  _xtlfreq = 20_000_000             ' 20 MHz crystal

DAT
                ORG     0
                ASMCLK              ' Set clock to 200 MHz
                ' ... program continues
```

AUGD

Augment Destination

Miscellaneous - Extends next literal Dest to 32 bits.

AUGD *#Dest*

Result: The 23-bit value formed from Dest is queued to prefix the next literal Dest occurrence (#Dest) to form a 32-bit literal for that instruction; interrupts are also temporarily disabled.

- Dest is a 32-bit literal whose upper 23 bits are prepended to the next literal Dest occurrence.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	11111DD	DDD	DDDDDDDDD	DDDDDDDDD	—	—	—	2

Related: AUGS

Explanation:

AUGD is an assistant instruction to aid with literal values that exceed 9 bits. Most PASM2 instructions have 9 bits available for literal Dest values; enough for many uses, but not all. **AUGD** augments the next occurrence of a literal Dest value to be a full 32-bits.

When the instruction with the soon-to-be-augmented literal is later executed, the COG uses the lower 9 bits encoded in the instruction's Dest field and prepends **AUGD**'s 23 bits to it.

All instructions following **AUGD** are shielded from interrupt until after the instruction with the newly-augmented literal Dest value is executed. Dest value augmentation occurs in the instruction pipeline only; code is not altered, value does not persist. **SETQ**/SETQ2 does not affect **AUGD**; the Q value passes through to the next instruction.

Though **AUGD** may be manually entered wherever needed, the Parallax P2 compiler supports a convenient way to use this feature. In the target instruction's Dest field, use "##" followed by the desired 32-bit literal (instead of "#" followed by a 9-bit literal); the compiler will automatically invoke **AUGD** immediately before. When counting clock cycles, make sure to account for 2 extra clock cycles for instructions containing ## augmented literals.

Pitfall (Silicon Bug): **AUGD** placed between **SETQ**/SETQ2 and **RDLONG**/WRLONG/WMLONG cancels the block-size PTRx delta calculation. The block transfer completes correctly, but PTRx advances by only a single-long delta.

AUGS

Augment Source

Miscellaneous - Extends next literal Src to 32 bits.

AUGS *#Src*

Result: The 23-bit value formed from Src is queued to prefix the next literal Src occurrence (#Src) to form a 32-bit literal for that instruction; interrupts are also temporarily disabled.

- Src is a 32-bit literal whose upper 23 bits are prepended to the next literal Src occurrence.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	11110SS	SSS	SSSSSSSSS	SSSSSSSSS	—	—	—	2

Related: AUGD

Explanation:

AUGS is an assistant instruction to aid with literal values that exceed 9 bits. Most PASM2 instructions have 9 bits available for literal Src values; enough for many uses, but not all. **AUGS** augments the next occurrence of a literal Src value to be a full 32-bits.

When the instruction with the soon-to-be-augmented literal is later executed, the COG uses the lower 9 bits encoded in the instruction's Src field and prepends **AUGS**'s 23 bits to it.

All instructions following **AUGS** are shielded from interrupt until after the instruction with the newly-augmented literal Src value is executed. Src value augmentation occurs in the instruction pipeline only; code is not altered, value does not persist. **SETQ**/SETQ2 does not affect **AUGS**; the Q value passes through to the next instruction.

Though **AUGS** may be manually entered wherever needed, the Parallax P2 compiler supports a convenient way to use this feature. In the target instruction's Src field, use "##" followed by the desired 32-bit literal (instead of "#" followed by a 9-bit literal); the compiler will automatically invoke **AUGS** immediately before. When counting clock cycles, make sure to account for 2 extra clock cycles for instructions containing ## augmented literals.

Pitfall (Silicon Bug): Intervening ALTx instructions with an immediate #S operand between **AUGS** and its intended target instruction will also receive the augmented value—without canceling it. Both the ALTx and the target instruction use the **AUGS** value. To avoid this, use a register for the ALTx instruction's S operand instead of an immediate.

Pitfall (Silicon Bug): **AUGS** placed between **SETQ**/SETQ2 and **RDLONG**/WRLONG/WMLONG cancels the block-size PTRx delta calculation. The block transfer completes correctly, but PTRx advances by only a single-long delta.

Instructions: B

This section contains all PASM2 instructions beginning with the letter B.

BITC / BITNC / BITZ / BITNZ

Set Bit to Flag State

Arithmetic Operations - Sets bits to match flag state.

BITC *Dest, {#}Src* **{WCZ}**

BITNC *Dest, {#}Src* **{WCZ}**

BITZ *Dest, {#}Src* **{WCZ}**

BITNZ *Dest, {#}Src* **{WCZ}**

Result: Dest bit(s) designated by Src are set to the corresponding flag state. Optionally updates C and Z to the original bit state.

- Dest is a register whose value will have bit(s) set to the flag state.
- Src identifies the bit(s) to modify: Src[4:0] = bit number, Src[9:5] = additional contiguous bits.
- WCZ is an optional effect to update C and Z flags to the original bit state.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100010	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2
EEEE	0100011	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2
EEEE	0100100	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2
EEEE	0100101	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2

Related: BITH, BITL, BITNOT, BITRND

Explanation:

These instructions set designated bit(s) in Dest to the specified flag value:

Instruction	Sets bits to
BITC	C flag value
BITNC	!C (inverted C)
BITZ	Z flag value
BITNZ	!Z (inverted Z)

BITC and **BITZ** copy the direct flag state; **BITNC** and **BITNZ** copy the inverted flag state.

Src[4:0] indicates the bit number (0-31). For a range, Src[9:5] specifies additional contiguous bits (1-31). A **SETQ** instruction preceding these can substitute its Dest[4:0] for Src[9:5].

If WCZ is specified, both C and Z flags are set to the original base bit value—set (1) if the original base bit was set, or cleared (0) if it was clear.

BITH

Bit High
Arithmetic Operations - Sets specified bits to high (1).

BITH *Dest, {#}Src* **{WCZ}**

Result: Dest bit(s) designated by Src are set to high (1).

- Dest is a register whose value will have one or more bits set high.
- Src is a register, 9-bit literal, or 10-bit augmented literal whose value identifies the bit(s) to modify.
- WCZ is an optional effect to update the C and Z flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100001	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2

Related: BITL, BITNOT, BITC, BITNC, BITZ, BITNZ

Explanation:

BITH sets the Dest bit(s) designated by Src to high (1). All other bits are left unchanged.

Src[4:0] indicates the bit number (0-31). For a range of bits, Src[4:0] indicates the base bit number and Src[9:5] indicates how many contiguous bits beyond the base should be affected (1-31). A 9-bit literal Src is enough to express the base bit (Src[4:0]) and a range of up to 16 contiguous bits (Src[8:5]). If needed, use the augmented literal feature (##Src) to augment Src to a 10-bit literal value.

When Src is a register, the register's value bits [9:0] are used as-is, unless a **SETQ** instruction immediately precedes **BITH**, substituting **SETQ**'s Dest[4:0] in place of value bits[9:5].

If the WCZ effect is specified, both the C and Z flags are set (1) if the original Dest base bit (before modification) was set, or are cleared (0) if it was clear. This preserves information about the original bit state before it was set high.

BITL

Bit Low
Arithmetic Operations - Sets specified bits to low (0).

BITL *Dest, {#}Src* **{WCZ}**

Result: Dest bit(s) designated by Src are set to low (0).

- Dest is a register whose value will have one or more bits set low.
- Src is a register, 9-bit literal, or 10-bit augmented literal whose value identifies the bit(s) to modify.
- WCZ is an optional effect to update the C and Z flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100000	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2

Related: BITH, BITNOT, BITC, BITNC, BITZ, BITNZ

Explanation:

BITL sets the Dest bit(s) designated by Src to low (0). All other bits are left unchanged.

Src[4:0] indicates the bit number (0-31). For a range of bits, Src[4:0] indicates the base bit number and Src[9:5] indicates how many contiguous bits beyond the base should be affected (1-31). A 9-bit literal Src is enough to express the base bit (Src[4:0]) and a range of up to 16 contiguous bits (Src[8:5]). If needed, use the augmented literal feature (##Src) to augment Src to a 10-bit literal value.

When Src is a register, the register's value bits [9:0] are used as-is, unless a **SETQ** instruction immediately precedes **BITL**, substituting **SETQ**'s Dest[4:0] in place of value bits[9:5].

If the WCZ effect is specified, both the C flag and the Z flag are set (1) if the original Dest base bit (before modification) was set, or are cleared (0) if it was clear. This preserves information about the original bit state before it was cleared to low.

BITNOT

Bit Not

Arithmetic Operations - Toggles specified bits to opposite state.

BITNOT *Dest, {#}Src* **{WCZ}**

Result: Dest bit(s) designated by Src are toggled to their opposite state(s).

- Dest is a register whose value will have one or more bits toggled.
- Src is a register, 9-bit literal, or 10-bit augmented literal whose value identifies the bit(s) to modify.
- WCZ is an optional effect to update the C and Z flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100111	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2

Related: BITH, BITL, BITC, BITNC, BITZ, BITNZ, BITRND

Explanation:

BITNOT alters the Dest bit(s) designated by Src to their inverse state. All other bits are left unchanged.

Src[4:0] indicates the bit number (0-31). For a range of bits, Src[4:0] indicates the base bit number and Src[9:5] indicates how many contiguous bits beyond the base should be affected (1-31). A 9-bit literal Src is enough to express the base bit (Src[4:0]) and a range of up to 16 contiguous bits (Src[8:5]). If needed, use the augmented literal feature (##Src) to augment Src to a 10-bit literal value.

When Src is a register, the register's value bits [9:0] are used as-is, unless a **SETQ** instruction immediately precedes **BITNOT**, substituting **SETQ**'s Dest[4:0] in place of value bits[9:5].

If the WCZ effect is specified, the C and Z flags are set (1) if the original Dest base bit (before modification) was set, or are cleared (0) if it was clear. This preserves information about the original bit state.

BITRND

Bit Random

Arithmetic Operations - Sets specified bits to random states.

BITRND *Dest, {#}Src* **{WCZ}**

Result: Dest bit(s) designated by Src are each set randomly to low or high.

- Dest is a register whose value will have one or more bits set randomly low or high.
- Src is a register, 9-bit literal, or 10-bit augmented literal whose value identifies the bit(s) to modify.
- WCZ is an optional effect to update the C and Z flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100110	CZI	DDDDDDDDD	SSSSSSSSS	original D[S[4:0]]	original D[S[4:0]]	D	2

Related: BITZ, BITNZ, BITC, BITNC, BITH, BITL, BITNOT

Explanation:

BITRND alters the Dest bit(s) designated by Src to each be an independent random low or high value, based on bit(s) from the Xoroshiro128** PRNG. All other bits are left unchanged.

Src[4:0] indicates the bit number (0-31). For a range of bits, Src[4:0] indicates the base bit number and Src[9:5] indicates how many contiguous bits beyond the base should be affected (1-31). A 9-bit literal Src is enough to express the base bit (Src[4:0]) and a range of up to 16 contiguous bits (Src[8:5]). If needed, use the augmented literal feature (##Src) to augment Src to a 10-bit literal value.

When Src is a register, the register's value bits [9:0] are used as-is, unless a **SETQ** instruction immediately precedes **BITRND**, substituting **SETQ**'s Dest[4:0] in place of value bits[9:5].

If the WCZ effect is specified, the C and Z flags are set (1) if the original Dest base bit (before modification) was set, or are cleared (0) if it was clear. This preserves information about the original state of the base bit before randomization.

Each bit in the range is set independently from the Xoroshiro128** PRNG, producing pseudo-random values suitable for randomization, dithering, and simulation applications. The PRNG is not cryptographically secure and should not be used to generate cryptographic key material or IVs.

BLNPIX

Blend Pixels

Color Space and Pixel Operations - Alpha-blends color values using **SETPIV** factor.

BLNPIX *Dest, {#}Src*

Result: Src color value bytes are alpha-blended into Dest color value bytes using the SETPIV blend factor.

- Dest is a register containing the RGB color value to blend Src into, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose RGB color value bytes are blended into Dest.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010010	10I	DDDDDDDDD	SSSSSSSSS	—	—	D	7

Related: ADDPIX, MULPIX, MIXPIX, SETPIV

Explanation:

BLNPIX alpha-blends the individual RGB (red, green, blue) color values of Src into that of Dest and stores the result in the Dest register. The blend factor is set by a previous **SETPIV** instruction.

The alpha-blending operation combines the two color values based on the blend factor, allowing smooth color transitions and transparency effects. A blend factor of 0 leaves Dest unchanged, while a blend factor of 255 completely replaces Dest with Src. Values between 0 and 255 produce proportional blends.

The instruction processes all three color channels (and alpha if present) in parallel, completing in 7 clock cycles. This enables efficient pixel manipulation for graphics applications, user interfaces, and visual effects.

BMASK

Bit Mask
Arithmetic Operations - Generates an LSB-justified bit mask.
BMASK *Dest, {#}Src*

BMASK *Dest*

Result: Bit mask of size Src+1, or Dest+1 (1 to 32 bits) is stored into Dest.

- Dest is a register in which to store the generated bit mask and optionally contains the 5-bit mask size (second syntax).
- Src is a register or 5-bit literal whose value is the size of the bit mask to generate.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001110	01I	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001110	010	DDDDDDDDD	DDDDDDDDD	—	—	D	2

Related: ENCOD, DECOD, ONES, ZEROX

Explanation:

BMASK generates an LSB-justified bit mask (all ones) of Src+1 or Dest+1 length and stores it in Dest. The size value, whether specified by Src or Dest, is in the range 0-31 to generate 1 to 32 bits of bit mask.

In effect, Dest becomes (%10 « size) - 1 via the **BMASK** instruction. A size value of 0 generates a 1-bit mask (%1), a size value of 5 generates a 6-bit mask (%111111), and a size value of 15 generates a 16-bit mask (%1111_1111_1111_1111).

A bit mask is often useful in bitwise operations (AND, OR, **XOR**) to filter out or affect special groups of bits. For example:

```
1         BMASK   mask, #7              ' Create 8-bit mask ($FF)
2         AND     data, mask            ' Keep only lower 8 bits
```

The first syntax form uses Src to specify the size, while the second syntax form (without Src) uses the value already in Dest to determine the mask size. Both forms write the resulting mask back to Dest.

BRK

Breakpoint
Interrupts - Triggers a debug breakpoint in the current COG.

BRK *{#}Dest*

Result: If debug interrupts are enabled, a debug interrupt is triggered in the current COG and Dest's value becomes the debug code or the next debug condition.

- Dest is a register, 9-bit literal, or 32-bit augmented literal whose value becomes the debug code or condition depending on the state of execution (outside or inside of a Debug ISR).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110110	—	—	—	2

Related: GETBRK, COGBRK

Explanation:

BRK triggers a breakpoint in the current COG and either defines a breakpoint code or the next breakpoint condition(s). The COG must have **DEBUG** interrupts enabled, and if **BRK** is to be executed within the normal program (outside the Debug ISR), the "BRK instruction" interrupt must first be enabled from within a prior **DEBUG** ISR.

During normal program execution, the **BRK** instruction is used to generate a debug interrupt with an 8-bit code (from Dest[7:0]) which can be read within the Debug ISR using a **GETBRK** instruction. This allows the program to communicate **DEBUG** information or trigger specific breakpoint handlers.

During a Debug ISR, the **BRK** instruction is used instead to establish the next **DEBUG** interrupt condition(s) and to select INA/INB, instead of the IJMP0/IRET0 registers exposed during the ISR, so that the pins' inputs states may be read.

The format of Dest for **DEBUG** ISR use is %AAAAAAAAAAAAAAAAAAAA_BCDEFGHIJKLM where A is the 20-bit breakpoint address or 4-bit event code, and bits B-M control various interrupt enable conditions.

BRK is essential for interactive debugging, allowing precise control over program execution and inspection of program state at specific points or conditions.

Instructions: C

This section contains all PASM2 instructions beginning with the letter C.

CALL

CALL Subroutine
Branching and Flow Control - Calls a subroutine and pushes return info to stack.

CALL *#Addr*

CALL *#\Addr*

CALL *Dest* **{WC|WZ|WCZ}**

Result: Current C and Z flags and address of the next instruction are pushed onto the hardware stack, PC is set to the new address, and optionally C and Z are updated to new states.

- Addr is a symbolic reference to the target subroutine; the location to set PC to. Relative addressing is the default; use '\' to force absolute addressing.
- Dest is a register containing the 20-bit absolute address to set PC to and optional new C and Z states.
- WC, WZ, or WCZ are optional effects to update the flags from Dest's upper bit states.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101101	RAA	AAAAAAAAA	AAAAAAAAA	—	—	—	4 / 13-20
EEEE	1101011	CZ0	DDDDDDDDD	000101101	D[31]	D[30]	—	4 / 13-20

Related: RET, CALLA, CALLB, CALLD, CALLPA, CALLPB

Explanation:

CALL records the current state of the C and Z flags and the address of the next instruction (PC + 1 if COG/LUT execution; PC + 4 if Hub execution) by pushing to the stack (K), potentially updates the C and Z flags with new given states, and jumps to the given address or offset. The routine at the new address should eventually execute a **RET** instruction, or an instruction with a _RET_ condition, to return to the recorded address (the instruction following the CALL) and optionally restore the C and Z flag state as it was prior.

In the first syntax form, `#Addr` and `#\Addr` encodes the instruction with relative and absolute addressing, respectively. The relative form (the default) is vital for creating relocatable code. In either case, use symbolic references for Addr and the assembler will encode it properly. Examples: `CALL #SendBit` or `CALL #\DebugStatus`.

In the second syntax form, the format of the value at Dest is `CZxxxxxx_xxxxAAAA_AAAAAAAA_AAAAAAAA`. C is the new C flag state, Z is the new Z flag state, A is the new 20-bit address to jump to, and x are don't-care bits. This syntax effectively swaps the flags and PC with the value in the Dest register (and **RET** swaps them back), making it convenient for switching between two threads.

If the WC or WCZ effect is specified, the C flag is updated to match D[31], after its original state is recorded.

If the WZ or WCZ effect is specified, the Z flag is updated to match D[30], after its original state is recorded.

The instruction takes 4 cycles for COG/LUT execution, or 13-20 cycles for Hub execution.

CALLA

CALL Subroutine via PTRA
Branching and Flow Control - Calls subroutine using PTRA as stack pointer.

CALLA *#Addr*

CALLA *#\Addr*

CALLA *Dest* **{WC|WZ|WCZ}**

Result: Current C and Z flags and address of the next instruction are written to Hub RAM at PTRA, PTRA is incremented by 4, PC is set to the new address, and optionally C and Z are updated to new states.

- Addr is a symbolic reference to the target subroutine; the location to set PC to. Relative addressing is the default; use '\' to force absolute addressing.
- Dest is a register containing the 20-bit absolute address to set PC to and optional new C and Z states.
- WC, WZ, or WCZ are optional effects to update the flags from Dest's upper bit states.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101110	RAA	AAAAAAAAA	AAAAAAAAA	—	—	—	5-12 / 14-32
EEEE	1101011	CZ0	DDDDDDDDD	000101110	D[31]	D[30]	—	5-12 / 14-32

Related: CALL, CALLB, CALLD, RETA

Explanation:

CALLA writes the current C and Z flags and the address of the next instruction into the 4-byte Hub RAM location at PTRA, then increments PTRA by 4, sets PC to the new relative or absolute address, and optionally updates C and Z to new states.

In the first syntax form, `#Addr` and `#\Addr` encodes the instruction with relative and absolute addressing, respectively. The relative form (the default) is vital for creating relocatable code. In either case, use symbolic references for Addr and the assembler will encode it properly.

In the second syntax form, the format of the value at Dest is `CZxxxxxx_xxxxAAAA_AAAAAAAA_AAAAAAAA`. C is the new C flag state, Z is the new Z flag state, A is the new 20-bit address to jump to, and x are don't-care bits.

If the WC or WCZ effect is specified, the C flag is set to D[31] after the original state is recorded.

If the WZ or WCZ effect is specified, the Z flag is set to D[30] after the original state is recorded.

CALLA is used for subroutine calls when Hub RAM is being used as the call stack instead of the hardware stack. This is useful for deep nesting or when preserving the hardware stack for other purposes. The instruction takes 5-12 cycles for COG/LUT execution, or 14-32 cycles for Hub execution.

CALLB

CALL Subroutine via PTRB

Branching and Flow Control - Calls subroutine using PTRB as stack pointer.

CALLB *#Addr*

CALLB *#\Addr*

CALLB *Dest* **{WC|WZ|WCZ}**

Result: Current C and Z flags and address of the next instruction are written to Hub RAM at PTRB, PTRB is incremented by 4, PC is set to the new address, and optionally C and Z are updated to new states.

- Addr is a symbolic reference to the target subroutine; the location to set PC to. Relative addressing is the default; use '\' to force absolute addressing.
- Dest is a register containing the 20-bit absolute address to set PC to and optional new C and Z states.
- WC, WZ, or WCZ are optional effects to update the flags from Dest's upper bit states.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101111	RAA	AAAAAAAAA	AAAAAAAAA	—	—	—	5-12 / 14-32
EEEE	1101011	CZ0	DDDDDDDDD	000101111	D[31]	D[30]	—	5-12 / 14-32

Related: CALL, CALLA, CALLD, RETB

Explanation:

CALLB writes the current C and Z flags and the address of the next instruction into the 4-byte Hub RAM location at PTRB, then increments PTRB by 4, sets PC to the new relative or absolute address, and optionally updates C and Z to new states.

In the first syntax form, `#Addr` and `#\Addr` encodes the instruction with relative and absolute addressing, respectively. The relative form (the default) is vital for creating relocatable code. In either case, use symbolic references for Addr and the assembler will encode it properly.

In the second syntax form, the format of the value at Dest is `CZxxxxxx_xxxxAAAA_AAAAAAAA_AAAAAAAA`. C is the new C flag state, Z is the new Z flag state, A is the new 20-bit address to jump to, and x are don't-care bits.

If the WC or WCZ effect is specified, the C flag is set to D[31] after the original state is recorded.

If the WZ or WCZ effect is specified, the Z flag is set to D[30] after the original state is recorded.

CALLB operates identically to **CALLA** except it uses PTRB as the stack pointer instead of PTRA. This allows for maintaining separate **CALL** stacks or using both pointers for different purposes. The instruction takes 5-12 cycles for COG/LUT execution, or 14-32 cycles for Hub execution.

CALLD

CALL with Destination Register

Branching and Flow Control - Calls subroutine saving return info to a register.

CALLD *PA|PB|PTRA|PTRB, #Addr*

CALLD *PA/PB/PTRA/PTRB, #\Addr*

CALLD *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Current C and Z flags and address of the next instruction are written to the specified register (PA, PB, PTRA, PTRB, or Dest), PC is set to the new address, and optionally C and Z are updated to new states.

- PA|PB|PTRA|PTRB is the special register to store the current C and Z flags and next address into.
- Addr is a symbolic reference to the target subroutine; the location to set PC to. Relative addressing is the default; use '\' to force absolute addressing.
- Dest is a register to write the current C and Z flags and the address of the next instruction into.
- Src is a register, 9-bit literal, or 32-bit augmented literal that contains the relative or absolute address to set PC to and optional new C and Z states. Use # for relative addressing; omit # for absolute addressing.
- WC, WZ, or WCZ are optional effects to update the flags from Src's upper bit states.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	11100WW	RAA	AAAAAAAAA	AAAAAAAAA	—	—	—	4 / 13-20
EEEE	1011001	CZI	DDDDDDDDD	SSSSSSSSS	S[31]	S[30]	—	4 / 13-20

Related: CALL, CALLPA, CALLPB, RET, PA, PB, PTRA, PTRB

Explanation:

CALLD records the current state of the C and Z flags and the address of the next instruction (PC + 1 if COG/LUT execution; PC + 4 if Hub execution) by writing them to the PA, PB, PTRA, PTRB, or Dest register, potentially updates the C and Z flags with new given states, and jumps to the given address or offset. The routine at the new address should eventually execute another **CALLD** instruction to return to the recorded address (the instruction following the original **CALLD**), optionally restore the C and Z flag state as it was prior, and optionally prep for another **CALLD**.

This instruction is typically used for the P2 **DEBUG** function.

In the first syntax form, `#Addr` and `#\Addr` encodes the instruction with relative and absolute addressing, respectively. The relative form (the default) is vital for creating relocatable code. In either case, use symbolic references for Addr and the assembler will encode it properly. Examples: `CALLD PA, #SendBit` or `CALLD PB, #\DebugStatus`.

In the second syntax form, the format of the value at Src is `CZxxxxxx_xxxxAAAA_AAAAAAAA_AAAAAAAA`. C is the new C flag state, Z is the new Z flag state, A is the new 20-bit address to jump to, and x are don't-care bits. If Src is a 9-bit literal (immediate), it will be sign-extended to 20 bits and used as a relative offset, giving a range of -256 to +255 instructions relative to the instruction following the **CALLD**. When relative, PC is adjusted by signed(Src) if COG/LUT execution, or by signed(Src * 4) if Hub execution.

If the WC or WCZ effect is specified, the C flag is updated to match S[31], after its original state is recorded.

If the WZ or WCZ effect is specified, the Z flag is updated to match S[30], after its original state is recorded.

The instruction takes 4 cycles for COG/LUT execution, or 13-20 cycles for Hub execution.

CALLPA

CALL Subroutine with PA Parameter
Branching and Flow Control - Calls subroutine and loads parameter into PA.

CALLPA *{#}Dest, {#}Src*

Result: Current C and Z flags and address of the next instruction are pushed onto the hardware stack, Dest is copied to PA, and PC is set to the address specified by Src.

- Dest is a register, 9-bit literal, or 32-bit augmented literal whose value is copied to PA.
- Src is a register, 9-bit literal, or 32-bit augmented literal that contains the relative or absolute address to set PC to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011010	0LI	DDDDDDDDD	SSSSSSSSS	—	—	K, PA and PC	4 / 13-20

Related: CALL, CALLPB, CALLD, RET, PA

Explanation:

CALLPA records the current state of the C and Z flags and the address of the next instruction (PC + 1 if COG/LUT execution; PC + 4 if Hub execution) by pushing to the stack (K), copies the value of Dest to PA, and jumps to the address specified by Src. The routine at the new address should eventually execute a **RET** instruction to return to the recorded address and restore the flags.

This instruction is useful for passing a parameter to a subroutine via the PA register while simultaneously calling that subroutine. The parameter can be an immediate value, making it convenient for subroutines that need a single argument.

The Src operand determines the target address. If Src is preceded by #, it is treated as a relative address; otherwise it is an absolute address. If Src is a register, its lower 20 bits specify the absolute address to jump to.

The instruction takes 4 cycles for COG/LUT execution, or 13-20 cycles for Hub execution.

CALLPB

CALL Subroutine with PB Parameter
Branching and Flow Control - Calls subroutine and loads parameter into PB.

CALLPB *{#}Dest, {#}Src*

Result: Current C and Z flags and address of the next instruction are pushed onto the hardware stack, Dest is copied to PB, and PC is set to the address specified by Src.

- Dest is a register, 9-bit literal, or 32-bit augmented literal whose value is copied to PB.
- Src is a register, 9-bit literal, or 32-bit augmented literal that contains the relative or absolute address to set PC to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011010	1LI	DDDDDDDDD	SSSSSSSSS	—	—	K, PB and PC	4 / 13-20

Related: CALL, CALLPA, CALLD, RET, PB

Explanation:

CALLPB records the current state of the C and Z flags and the address of the next instruction (PC + 1 if COG/LUT execution; PC + 4 if Hub execution) by pushing to the stack (K), copies the value of Dest to PB, and jumps to the address specified by Src. The routine at the new address should eventually execute a **RET** instruction to return to the recorded address and restore the flags.

This instruction operates identically to **CALLPA** except it uses the PB register instead of PA. This is useful for passing a parameter to a subroutine via the PB register, or when both PA and PB need to be set by using **CALLPA** followed by **CALLPB**, or when the subroutine convention uses PB for parameters.

The Src operand determines the target address. If Src is preceded by #, it is treated as a relative address; otherwise it is an absolute address. If Src is a register, its lower 20 bits specify the absolute address to jump to.

The instruction takes 4 cycles for COG/LUT execution, or 13-20 cycles for Hub execution.

CMP

Compare Unsigned

Arithmetic Operations - Compares two unsigned values and sets flags.

CMP *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Greater/lesser and equality status is optionally written to the C and Z flags.

- Dest is the register containing the value to compare with that of Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is compared to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010000	CZI	DDDDDDDDD	SSSSSSSSS	Unsigned (D < S)	(D == S)	—	2

Related: CMPR, CMPX, CMPS, CMPSX, CMPM

Explanation:

CMP compares the unsigned values of Dest and Src by subtracting Src from Dest and optionally setting the C and Z flags accordingly. The result of the subtraction is discarded; only the flags are affected. Dest is not modified.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest is less than Src (unsigned comparison), or is cleared (0) if Dest is greater than or equal to Src. This indicates that the subtraction would require a borrow.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Dest equals Src, or is cleared (0) if they are not equal.

To compare unsigned multi-long values (64-bit or larger), use **CMP** for the least significant **LONG**, then **CMPX** for each subsequent **LONG**. For example, to compare two 64-bit values:

```
CMP     value_lo, other_lo  wc    ' Compare low longs
CMPX    value_hi, other_hi  wcz   ' Compare high longs with borrow
' C and Z now reflect the 64-bit comparison result
```

CMP is fundamental for implementing conditional logic and control flow based on numeric comparisons.

CMPM

Compare Most Significant Bit
Arithmetic Operations - Compares values with C set to MSB of difference.

CMPM *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Greater/lesser and equality status is optionally written to the C and Z flags.

- Dest is the register containing the value to compare with that of Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is compared to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010101	CZI	DDDDDDDDD	SSSSSSSSS	MSB of (D-S)	(D == S)	—	2

Related: CMP, CMPS

Explanation:

CMPM compares the unsigned values of Dest and Src by subtracting Src from Dest and optionally setting the C and Z flags accordingly. The result of the subtraction is discarded; only the flags are affected. Dest is not modified.

If the WC or WCZ effect is specified, the C flag is updated to the MSB (bit 31) of the result of Dest - Src. This is different from **CMP**, which sets C based on whether a borrow occurred. **CMPM**'s C flag directly reflects the sign bit of the difference.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Dest equals Src, or is cleared (0) if they are not equal.

CMPM is useful when the most significant bit of the difference carries semantic meaning for the algorithm being implemented, such as certain mathematical operations or specialized comparison logic.

CMPR

Compare Reverse
Arithmetic Operations - Compares values with reversed operand order.

CMPR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Greater/lesser and equality status is optionally written to the C and Z flags.

- Dest is the register containing the value to compare with that of Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is compared to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010100	CZI	DDDDDDDDD	SSSSSSSSS	borrow of (S - D)	D == S	—	2

Related: CMP

Explanation:

CMPR compares the unsigned values of Dest and Src by subtracting Dest from Src (the reverse of **CMP**) and optionally setting the C and Z flags accordingly. The result of the subtraction is discarded; only the flags are affected. Dest is not modified.

If the WC or WCZ effect is specified, the C flag is set (1) if Src is less than Dest (unsigned comparison), or is cleared (0) if Src is greater than or equal to Dest. This is the opposite condition from **CMP**.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Dest equals Src, or is cleared (0) if they are not equal.

CMPR is useful when the natural order of operands in your code is reversed from what **CMP** expects, avoiding the need to swap operands or reverse the logic. Note that for unsigned multi-long comparisons, use **CMP** (not **CMPR**) followed by **CMPX**.

CMPS

Compare Signed
Arithmetic Operations - Compares two signed values and sets flags.

CMPS *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Greater/lesser and equality status is optionally written to the C and Z flags.

- Dest is the register containing the value to compare with that of Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is compared to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010010	CZI	DDDDDDDDD	SSSSSSSSS	correct sign of (D - S)	(D == S)	—	2

Related: CMP, CMPX, CMPSX

Explanation:

CMPS compares the signed values of Dest and Src by subtracting Src from Dest and optionally setting the C and Z flags to indicate the comparison and operation results. The result of the subtraction is discarded; only the flags are affected. Dest is not modified.

If the WC or WCZ effect is specified, the C flag is set (1) if signed Dest is less than signed Src, or is cleared (0) if signed Dest is greater than or equal to signed Src. The comparison properly accounts for the sign bit.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Dest equals Src, or is cleared (0) if they are not equal.

To compare signed multi-long values (64-bit or larger), use **CMP** (not **CMPS**) for the least significant **LONG**, optionally followed by **CMPX** for middle longs, and finally **CMPSX** for the most significant **LONG**. The final **CMPSX** accounts for sign extension properly. For example, to compare two 64-bit signed values:

```
CMP     value_lo, other_lo  wc    ' Compare low longs unsigned
CMPSX   value_hi, other_hi  wcz   ' Compare high signed w/borrow
' C and Z now reflect the signed 64-bit comparison result
```

CMPSUB

Compare and Subtract

Arithmetic Operations - Conditionally subtracts if Dest is greater or equal.

CMPSUB *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Dest is decremented by Src unless it is less than Src, and the comparison results are optionally written to the C and Z flags.

- Dest is the register containing the value to compare with Src and is the destination written to if a subtraction is performed.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is compared with and possibly subtracted from Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010111	CZI	DDDDDDDDD	SSSSSSSSS	D >= S	result == 0	D †	2

† Dest is only written if D >= S (subtraction was performed).

Related: CMP, SUB

Explanation:

CMPSUB compares the unsigned values of Dest and Src, and if Src is less than or equal to Dest, then Src is subtracted from Dest. Optionally, the C and Z flags are set to indicate the comparison and operation results.

The operation performs the comparison first. If Dest >= Src (unsigned), then Dest is updated to Dest - Src. If Dest < Src, then Dest is left unchanged.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was greater than or equal to Src (subtraction was performed), or is cleared (0) if Dest was less than Src (no subtraction).

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals 0, or is cleared (0) if non-zero. Note that if no subtraction was performed (Dest < Src), Z reflects whether Dest was already zero.

CMPSUB is particularly useful for implementing division algorithms, modulo operations, and other mathematical routines where conditional subtraction based on magnitude is needed.

CMPSX

Compare Signed Extended
Arithmetic Operations - Extended signed comparison for multi-long values.

CMPSX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Greater/lesser and equality status is optionally written to the C and Z flags.

- Dest is the register containing the value to compare with that of Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is compared to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010011	CZI	DDDDDDDDD	SSSSSSSSS	correct sign of (D - (S + C))	Z AND (D == S + C)	—	2

Related: CMP, CMPX, CMPS

Explanation:

CMPSX compares the signed values of Dest and Src plus C by subtracting Src + C from Dest and optionally setting the C and Z flags accordingly. The **CMPSX** instruction is used to perform signed multi-long comparisons, such as 64-bit comparisons. The result of the subtraction is discarded; only the flags are affected. Dest is not modified.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest is less than Src + C (as multi-long signed values), or is cleared (0) otherwise. Use WC or WCZ on preceding **CMP** and **CMPX** instructions for proper final C flag. The comparison properly accounts for sign extension.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Z was previously set and the result of Dest - (Src + C) is zero, or it is cleared (0) if non-zero. This allows the Z flag to cascade through multi-long comparisons, remaining set only if all compared longs are equal.

For signed multi-long comparisons, use **CMP** for the least significant **LONG**, optionally **CMPX** for middle longs, and **CMPSX** for the most significant **LONG**:

```
1         CMP     value_lo, other_lo  wc    ' Compare low longs
2         CMPSX   value_hi, other_hi  wcz   ' Compare high signed w/borrow
3         ' C=1 if signed value < other, Z=1 if equal
```

CMPX

Compare Unsigned Extended
Arithmetic Operations - Extended unsigned comparison for multi-long values.

CMPX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Greater/lesser and equality status is optionally written to the C and Z flags.

- Dest is a register containing the value to compare with that of Src plus C.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value plus C is compared to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010001	CZI	DDDDDDDDD	SSSSSSSSS	borrow of (D - (S + C))	Z AND (D == S + C)	—	2

Related: CMP, CMPS, CMPSX

Explanation:

CMPX compares the unsigned values of Dest and Src plus C by subtracting Src + C from Dest and optionally setting the C and Z flags accordingly. The **CMPX** instruction is used to perform unsigned multi-long comparisons, such as 64-bit comparisons. The result of the subtraction is discarded; only the flags are affected. Dest is not modified.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest is less than Src plus C (unsigned comparison), or is cleared (0) otherwise. Use WC or WCZ on preceding **CMP** and **CMPX** instructions for proper final C flag.

If the WZ or WCZ effect is specified, the Z flag is set (1) if Z was previously set and Dest equals Src + C, or it is cleared (0) otherwise. This allows the Z flag to cascade through multi-long comparisons, remaining set only if all compared longs are equal.

For unsigned multi-long comparisons, use **CMP** for the least significant **LONG**, then **CMPX** for each subsequent **LONG**:

```
1         CMP     value_lo, other_lo  wc    ' Compare low longs
2         CMPX    value_hi, other_hi  wcz   ' Compare high longs with borrow
3         ' C=1 if unsigned value < other, Z=1 if equal
```

COGATN

Cog Attention
Events and Timing - Signals attention to one or more cogs.

COGATN *{#}Dest*

Result: The attention signal of one or more cogs is strobed.

- Dest is the register or 9-bit literal whose value (lower 8-bit pattern) indicates which cogs to signal.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111111	—	—	—	2

Related: POLLATN, WAITATN, JATN, JNATN

Explanation:

COGATN strobes the attention signal for one or more cogs. Dest bit positions 7:0 represent cogs 7 through 0; high (1) bits indicate the cog(s) to signal. The receiving cog(s) then latch the signal, setting an internal flag, and can use any of the attention monitor instructions (JATN, **JNATN**, **POLLATN**, **WAITATN**) or interrupts to respond and clear the flag.

In the intended use case, the cog receiving an attention request knows which other cog is strobing it and how to respond. In cases where multiple cogs may request the attention of a single cog, some messaging structure may need to be implemented in Hub RAM to differentiate requests.

For example, to signal cog 3:

```
1         COGATN  #%0000_1000           ' Signal cog 3 (bit 3 = 1)
```

To signal multiple cogs simultaneously:

```
1         COGATN  #%0001_0010           ' Signal cogs 1 and 4
```

COGATN is useful for implementing inter-cog communication, synchronization, and event notification without requiring polling of shared memory.

COGBRK

Cog Breakpoint
Interrupts - Triggers a breakpoint in a specified cog.

COGBRK *{#}Dest*

Result: If in the Debug ISR, trigger an asynchronous breakpoint in cog identified by Dest.

- Dest is the register or 9-bit literal whose value (lower 3-bits) indicates which cog to trigger.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110101	—	—	—	2

Related: CALLD, BRK

Explanation:

COGBRK triggers an asynchronous breakpoint in a designated cog. The **COGBRK** instruction must be executed from within a Debug ISR (interrupt service routine) and the designated cog must already have its asynchronous breakpoint interrupt enabled. Dest[2:0] indicates the ID of the desired cog (0-7).

This instruction is part of the P2's debugging infrastructure and is typically used by **DEBUG** monitors or development tools to halt a running cog for inspection. When executed, the target cog will interrupt its current execution and vector to its **DEBUG** interrupt handler, allowing the debugging system to examine or modify the cog's state.

For example, to trigger a breakpoint in cog 2:

```
1         COGBRK  #2                    ' Break cog 2 (must be in debug ISR)
```

COGBRK is a specialized instruction primarily used by development and debugging tools rather than in typical application code.

COGID

Cog Identification
COG Control and Locks - Gets current cog ID or checks if a cog is running.

COGID *{#}Dest* **{WC}**

Result: Current cog's ID is written to Dest or C is set (1) or cleared (0) if the Dest cog is running or stopped.

- Dest is the register where the current cog's ID will be written, or is the register or 9-bit literal whose value (lower 3-bits) indicates which cog to get the status for.
- WC is an optional effect to update the C flag with the Dest cog's running status.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	C0L	DDDDDDDDD	000000001	Cog D[3:0] running	—	D †	2...9, +2 if result

† Result written only if D is register and WC not specified.

Related: COGINIT, COGSTOP

Explanation:

COGID writes the current cog's ID into Dest (if Dest is a register and WC is omitted) or sets/clears the C flag according to the running/stopped state of the cog indicated by Dest[2:0] (if WC is given).

When used without the WC effect, **COGID** stores the current cog's ID (0-7) in the Dest register. This is useful when code needs to know which cog it is running on, for example when accessing cog-specific resources or implementing cog-aware algorithms.

When used with the WC effect, **COGID** checks the status of the cog specified by Dest[2:0]. If the WC effect is specified, the C flag is set (1) if the specified cog is running, or is cleared (0) if stopped. In this mode, Dest is not written.

For example, to get the current cog's ID:

```
1         COGID   myid                  ' Store this cog's ID in myid
```

To check if cog 3 is running:

```
1         COGID   #3            wc    ' C=1 if cog 3 is running
```

COGINIT

Cog Initialize

COG Control and Locks - Starts a cog to execute code from Hub RAM.

COGINIT *{#}Dest, {#}Src* **{WC}**

Result: Target cog is started according to Dest to execute code from Src. The code pointer (Src) is written to the target cog's PTRB, and optionally a data pointer is written to its PTRA if SETQ preceded COGINIT.

- Dest is the register or 9-bit literal describing the type of launch and possibly the ID of the desired cog to launch. If Dest is a register and WC is given, Dest is also where the ID of the launched cog will be written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value (lower 20 bits) is the target RAM address (for code) and the new cog's PTRB value.
- WC is an optional effect to update the C flag with the success (0) or fail (1) status and triggers Dest to be overwritten with new cog's ID.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100111	CLI	DDDDDDDDD	SSSSSSSSS	No cog available	—	D †	2…9, +2 if result

† Result written only if D is register and WC specified; contains launched cog ID.

Related: COGID, COGSTOP

Explanation:

COGINIT starts a new (unused) cog, a new pair of cogs (that may share LUT memory), or a specific cog by ID, to load code from Hub RAM to be executed within COG/LUT RAM or to be executed right from Hub RAM.

The format of Dest is `%E_N_xVVV` where:

- E controls loading (0=load from Hub, 1=no load/Hub exec)
- N controls target selection (0=specific cog ID, 1=find free cog)
- VVV is the cog ID or mode

The following predefined constants encode these bit patterns:

Constant	Target	Execution	Description
COGEXEC + id	Specific COG	COG RAM	Load 496 longs from Hub to COG RAM, execute from COG
HUBEXEC + id	Specific COG	Hub RAM	Execute directly from Hub RAM (no load)
COGEXEC_NEW	Any free COG	COG RAM	Auto-select available COG, load and execute
HUBEXEC_NEW	Any free COG	Hub RAM	Auto-select available COG, execute from Hub
COGEXEC_NEW_PAIR	Adjacent pair	COG RAM	Auto-select adjacent COG pair for LUT sharing
HUBEXEC_NEW_PAIR	Adjacent pair	Hub RAM	Auto-select adjacent COG pair, Hub execution

For specific COG targeting, add the cog ID (0-7) to **COGEXEC** or **HUBEXEC**. The _NEW variants automatically select available resources.

The lower 20 bits of Src is the code address; the entire 32-bit Src is written to the target cog's PTRB. If **COGINIT** is preceded by **SETQ**, that value is written to the target cog's PTRA.

If the WC effect is specified, C is set (1) on failure or cleared (0) on success. When WC is given and Dest is a register, Dest receives the launched cog's ID (or $F on failure).

Common usage examples:

Load and start a specific cog from Hub RAM:

```
1         COGINIT #1, #$100             ' Load and start cog 1 from Hub $100
```

Start a free cog:

```
1                 COGINIT #COGEXEC_NEW, addr  wc   ' Find free cog, load, start
2         if_c    JMP     #no_cog_available        ' Branch if no cog available
```

SKIP load and execute from Hub RAM:

```
1         COGINIT #HUBEXEC+3, addr      ' Cog 3 hub exec mode
```

Start a cog pair for LUT sharing:

```
1         COGINIT #HUBEXEC_NEW_PAIR, addr   ' Start free cog pair
```

COGSTOP

Cog Stop

COG Control and Locks - Stops and terminates a running cog.

COGSTOP *{#}Dest*

Result: Cog indicated by Dest is terminated (stopped).

- Dest is the register or 9-bit literal indicating (in lowest 3 bits) which cog to stop.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000000011	—	—	—	2-9

Related: COGINIT, COGID

Explanation:

COGSTOP terminates the cog identified by Dest[2:0]. In this dormant state, the cog ceases to execute code and power consumption is greatly reduced.

The cog specified by the lower 3 bits of Dest (0-7) is immediately halted. All registers and state in that cog are lost. The cog can be restarted later using **COGINIT**, which will reload it with new code and reset its state.

For example, to stop cog 4:

```
1         COGSTOP #4                    ' Stop cog 4
```

To stop the current cog (terminate self):

```
1         COGID   myid                  ' Get my cog ID
2         COGSTOP myid                  ' Stop myself
```

COGSTOP is useful for managing cog resources dynamically, shutting down cogs that are no longer needed, or resetting a cog before restarting it with new code. Note that stopping a cog does not free any Hub memory it may have been using.

CRCBIT

CRC Iterate Bit

Arithmetic Operations - Computes one bit iteration of a CRC calculation.

CRCBIT *Dest, {#}Src*

Result: Dest is updated with the next CRC iteration using the C flag and polynomial in Src.

- Dest is a register containing the current CRC value and is where the updated CRC is written.
- Src is a register or 9-bit literal containing the CRC polynomial.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001110	10I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: CRCNIB, REV

Explanation:

CRCBIT iterates the CRC value in Dest using the current C flag and the polynomial in Src. This instruction is designed for computing cyclic redundancy check (CRC) values bit by bit.

The operation performs a single bit iteration of a CRC calculation. The C flag represents the input bit, and Src contains the CRC polynomial. Dest contains the running CRC value and is updated with the result of this iteration.

The exact algorithm follows the standard CRC bit-wise computation: 1. Shift the CRC value in Dest left by one bit 2. If the original MSB **XOR** the input bit (C) is 1, **XOR** with the polynomial in Src

CRCBIT is typically used in a loop to process data one bit at a time:

```
1         MOV     crc, #0               ' Initialize CRC
2 .loop   RCL     data, #1        wc    ' Get next bit into C
3         CRCBIT  crc, poly             ' Update CRC with bit
4         DJNZ    count, #.loop         ' Repeat for all bits
```

For processing nibbles (4 bits) at a time instead, use **CRCNIB**.

CRCNIB

CRC Iterate Nibble
Arithmetic Operations - Computes four bit iterations of a CRC calculation.

CRCNIB *Dest, {#}Src*

Result: Dest is updated with CRC iterations for a nibble, and Q is shifted left by 4 bits.

- Dest is a register containing the current CRC value and is where the updated CRC is written.
- Src is a register or 9-bit literal containing the CRC polynomial.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001110	11I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: CRCBIT, REV

Explanation:

CRCNIB iterates the CRC value in Dest for a nibble (4 bits) using the polynomial in Src, and shifts the Q register left by 4 bits. This instruction accelerates CRC calculations by processing 4 bits per instruction instead of 1.

CRCNIB performs four CRC bit-iterations in sequence, consuming the input bits from Q[31:28] (the high nibble), then shifts Q left by 4 bits to bring the next nibble into Q[31:28] for the following **CRCNIB**.

The typical usage pattern is:

```
        SETQ    data                    ' Load data into Q
        MOV     crc, #0                 ' Initialize CRC
.loop   CRCNIB  crc, poly               ' Process 4 bits from Q[31:28]
        ' Q is automatically shifted left by 4
        DJNZ    count, #.loop           ' Repeat for all nibbles
```

CRCNIB is more efficient than **CRCBIT** when processing byte-oriented data, providing a 4x speedup for CRC calculations. The automatic Q shift simplifies the loop logic for multi-nibble processing.

Instructions: D

This section contains all PASM2 instructions beginning with the letter D.

Conditional Jump Timing Convention: Conditional jumps in this section (DJZ, **DJNZ**, **DJF**, **DJNF**) show their `Clks` field as `NOT-taken / taken`. The *taken* value depends on execution context:

Context	Clocks when taken
COG / LUT execution	4
Hub execution	13…20

So `2 OR 4 / 2 OR 13-20` reads as: 2 cycles when the jump is not taken, 4 cycles when taken in cog/LUT, 13–20 cycles when taken in hub execution.

DECMOD

Decrement Modulus
Arithmetic Operations - Decrements with modulus wrap-around from zero to a maximum.

DECMOD *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: If Dest was not equal to 0, it is decremented by 1; otherwise Dest is reset to Src.

- Dest is a register containing the value to decrement down to 0 with modulus, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is the modulus limit to apply to Dest's decrement operation.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0111001	CZI	DDDDDDDDD	SSSSSSSSS	D was 0	result == 0	D	2

Related: INCMOD

Explanation:

DECMOD compares Dest with 0—if not equal, it decrements Dest; otherwise it sets Dest equal to Src. If Dest begins in the range 0 to Src, iterations of **DECMOD** will decrement Dest repetitively from Src to 0.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was equal to 0 and subsequently reset to Src, or is cleared (0) if not reset. This indicates that the modulus wrapping occurred.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result is zero, or is cleared (0) if it is non-zero.

DECMOD does not limit Dest within the specified range—if Dest begins as greater than Src, iterations will continue to decrement it down through Src before cycling from Src to 0. This instruction is useful for implementing circular buffers and modular counters that wrap from 0 back to a maximum value.

DECOD

Decode Bit Position

Arithmetic Operations - Generates a bitmask with a single bit set at the specified position.

DECOD *Dest, {#}Src*

DECOD *Dest*

Result: A 32-bit value, with the bit position corresponding to Src or Dest value (0-31) set high, is stored in Dest.

- Dest is the register in which to store the decoded value and optionally begins by containing the 5-bit bit position value it is requesting (syntax 2).
- Src is an optional register or 5-bit literal whose value is the bit position to set high in the decoded value.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001110	00I	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001110	000	DDDDDDDDD	DDDDDDDDD	—	—	D	2

Related: ENCOD, BMASK

Explanation:

DECOD generates a 32-bit value with just one bit high, corresponding to the Src or Dest value (0-31) and stores that result in Dest. In effect, Dest becomes %1 « value via the **DECOD** instruction, where value is Src[4:0] or Dest[4:0].

Examples of decoded values:

- A value of 0 generates %00000000_00000000_00000000_00000001
- A value of 5 generates %00000000_00000000_00000000_00100000
- A value of 15 generates %00000000_00000000_10000000_00000000

The first syntax form uses Src to specify the bit position, while the second syntax form uses Dest[4:0] as both the input bit position and the destination for the result.

DECOD is the complement of **ENCOD**. It is commonly used to generate bit masks for testing or setting individual bits within registers or memory locations.

DIRC / DIRNC

Set Pin Direction by C Flag {#dirnc}

Pin I/O and Smart Pins - Sets pin direction based on C flag state.

DIRC *{#}Dest* **{WCZ}**

DIRNC *{#}Dest* **{WCZ}**

Result: The I/O pin direction bit(s), described by Dest, are set to output/input according to C or !C; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output or input.

- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000010	DIR bit†	DIR bit†	DIR bit	2
EEEE	1101011	CZL	DDDDDDDDD	001000011	DIR bit†	DIR bit†	DIR bit	2

† Original direction state of the base pin (D[5:0]) before instruction executes.

Related: DIRZ, DIRNZ, DIRL, DIRH, DIRNOT, DIRRND

Explanation:

DIRC or **DIRNC** alters the direction register's bit(s) designated by Dest to equal the state, or inverse state, of the C flag. All other bits are left unchanged.

DIRC sets the pin(s) to the direction indicated by the C flag: high (1) sets the pin(s) to output, low (0) to input. **DIRNC** inverts this relationship, setting the pin(s) according to the inverse of the C flag (!C).

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **DIRC** or **DIRNC** instruction; substituting **SETQ**'s Dest[4:0] in place of value bits[10:6], for **DIRC** or **DIRNC**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group (DIRA or DIRB); it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of DIRA / DIRB's base bit, identified by Dest.

DIRH

Set Pin Direction High
Pin I/O and Smart Pins - Sets pins to output direction.

DIRH *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to output direction; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000001	DIR bit†	DIR bit†	DIR bit	2

† Original direction state of the base pin (D[5:0]) before instruction executes.

Related: DIRL, DIRC, DIRNC, DIRZ, DIRNZ

Explanation:

DIRH sets the direction register's bit(s) designated by Dest to high (1), making the pin(s) outputs. All other direction bits are left unchanged.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

If the WCZ effect is specified, the C flag is set to the original state of the base direction bit, and Z is set to the same value.

DIRL

Set Pin Direction Low
Pin I/O and Smart Pins - Sets pins to input direction.

DIRL *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to input direction; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to input.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000000	DIR bit†	DIR bit†	DIR bit	2

† Original direction state of the base pin (D[5:0]) before instruction executes.

Related: DIRH, DIRC, DIRNC, DIRZ, DIRNZ

Explanation:

DIRL alters the direction register's bit(s) designated by Dest to be low (0), setting the I/O pin(s) to input mode. The rest of the direction bits are left as-is.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

If the WCZ effect is specified, the C flag is set to the original state of the base direction bit, and Z is set to the same value.

DIRNOT

Direction Not
Pin I/O and Smart Pins - Toggles pin direction to opposite state.

DIRNOT *{#}Dest* **{WCZ}**

Result: The I/O pin direction bit(s), described by Dest, are toggled to their opposite state(s); the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to toggle to the opposite direction.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000111	DIR bit†	DIR bit†	DIR bit	2

† Original direction state of the base pin (D[5:0]) before instruction executes.

Related: DIRRND, DIRL, DIRH, DIRC, DIRNC, DIRZ, DIRNZ

Explanation:

DIRNOT alters the direction register's bit(s) designated by Dest to their inverse state. All other bits are left unchanged. Pins that were outputs become inputs, and pins that were inputs become outputs.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **DIRNOT** instruction; substituting **SETQ**'s Dest[4:0] in place of value bits[10:6], for **DIRNOT**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group (DIRA or DIRB); it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of DIRA / DIRB's base bit, identified by Dest.

DIRZ / DIRNZ

Set Pin Direction by Z Flag {#dirnz}
Pin I/O and Smart Pins - Sets pin direction based on Z flag state.

DIRZ *{#}Dest* **{WCZ}**

DIRNZ *{#}Dest* **{WCZ}**

Result: The I/O pin direction bit(s), described by Dest, are set to output/input according to Z or !Z; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output or input.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000100	DIR bit†	DIR bit†	DIR bit	2
EEEE	1101011	CZL	DDDDDDDDD	001000101	DIR bit†	DIR bit†	DIR bit	2

† Original direction state of the base pin (D[5:0]) before instruction executes.

Related: DIRC, DIRNC, DIRNOT, DIRRND, DIRL, DIRH

Explanation:

DIRZ or **DIRNZ** alters the direction register's bit(s) designated by Dest to equal the state, or inverse state, of the Z flag. All other bits are left unchanged.

DIRZ sets the pin(s) to the direction indicated by the Z flag: high (1) sets the pin(s) to output, low (0) to input. **DIRNZ** inverts this relationship, setting the pin(s) according to the inverse of the Z flag (!Z).

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **DIRZ** or **DIRNZ** instruction; substituting **SETQ**'s Dest[4:0] in place of value bits[10:6], for **DIRZ** or **DIRNZ**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group (DIRA or DIRB); it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of DIRA / DIRB's base bit, identified by Dest.

DIRRND

Direction Random
Pin I/O and Smart Pins - Sets pin direction to random state.

DIRRND *{#}Dest* **{WCZ}**

Result: The I/O pin direction bit(s), described by Dest, are each set randomly low or high (input or output); the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set randomly to input or output.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000110	DIR bit†	DIR bit†	DIR bit	2

† Original direction state of the base pin (D[5:0]) before instruction executes.

Related: DIRC, DIRNC, DIRZ, DIRNZ, DIRNOT, DIRL, DIRH

Explanation:

DIRRND alters the direction register's bit(s) designated by Dest to be random low and high (input and output), based on bit(s) from the Xoroshiro128** PRNG. All other bits are left unchanged.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **DIRRND** instruction; substituting **SETQ**'s Dest[4:0] in place of value bits[10:6], for **DIRRND**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group (DIRA or DIRB); it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of DIRA / DIRB's base bit, identified by Dest, before the random modification occurs.

DJF

Decrement and Jump If Full
Branching and Flow Control - Decrements and jumps if result wraps to $FFFFFFFF.

DJF *Dest, {#}Src*

Result: Dest is decremented. If the result equals $FFFF_FFFF (full), PC is set to a new relative (#Src) or absolute (Src) address; otherwise execution continues with the next instruction.

- Dest is a register whose value is decremented and tested for full or not full.
- Src is a register, 9-bit literal, or 20-bit augmented literal whose value is the absolute or relative address to set PC to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011011	10I	DDDDDDDDD	SSSSSSSSS	—	—	D	2 or 4 / 2 or 13-20

Related: DJNF, DJZ, DJNZ

Explanation:

DJF decrements the value in Dest, writes the result, and jumps to the address described by Src if the result is full ($FFFF_FFFF, or -1 signed).

This instruction is useful for implementing loops that count down until a register wraps from 0 to -1. Use # prefix on Src for relative addressing; omit # for absolute addressing.

The instruction executes in 2 clock cycles when the branch is not taken. When taken, it executes in 4 clock cycles during cog/LUT execution, or 13-20 clock cycles during hub execution.

DJNF

Decrement and Jump If Not Full

Branching and Flow Control - Decrements and jumps if result does not wrap.

DJNF *Dest, {#}Src*

Result: Dest is decremented. If the result does NOT equal $FFFF_FFFF (not full), PC is set to a new relative (#Src) or absolute (Src) address; otherwise execution continues with the next instruction.

- Dest is a register whose value is decremented and tested for full or not full.
- Src is a register, 9-bit literal, or 20-bit augmented literal whose value is the absolute or relative address to set PC to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011011	11I	DDDDDDDDD	SSSSSSSSS	—	—	D	2 or 4 / 2 or 13-20

Related: DJF, DJZ, DJNZ

Explanation:

DJNF decrements the value in Dest, writes the result, and jumps to the address described by Src if the result is NOT full (not equal to $FFFF_FFFF).

This instruction is useful for implementing loops that continue until a register wraps from 0 to -1 (full). Use # prefix on Src for relative addressing; omit # for absolute addressing.

Dest is always written with the decremented value. PC is written only when the result in Dest is not full.

The instruction executes in 2 clock cycles when the branch is not taken. When taken, it takes 4 clock cycles in COG/LUT execution, or 13–20 clock cycles in hub execution.

DJZ / DJNZ

Decrement and Jump If Zero {#djnz}

Branching and Flow Control - Decrements and conditionally jumps based on zero result.

DJZ *Dest, {#}Src*

DJNZ *Dest, {#}Src*

Result: Dest is decremented by 1, and conditionally jumps based on the result.

- Dest is a register whose value is decremented and tested.
- Src is the jump address: use # for relative, omit for absolute.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011011	00I	DDDDDDDDD	SSSSSSSSS	—	—	D + PC*	2 or 4 / 2 or 13-20
EEEE	1011011	01I	DDDDDDDDD	SSSSSSSSS	—	—	D + PC*	2 or 4 / 2 or 13-20

*PC is written only when the jump condition is met.

Related: DJF, DJNF, IJZ, IJNZ, TJZ, TJNZ

Explanation:

DJZ and **DJNZ** decrement Dest and conditionally jump based on whether the result is zero or non-zero:

Instruction	Jumps when
DJZ	result == 0
DJNZ	Result != 0

DJNZ is one of the most commonly used loop instructions—it continues looping while the counter is non-zero.

Example loop:

```
1         MOV     count, #10              ' Set loop counter to 10
2 .loop   ' loop body here
3         DJNZ    count, #.loop           ' Decrement and loop if not zero
```

Takes 2 clocks when not jumping; when jumping, 4 clocks in cog/LUT execution or 13–20 clocks during hub execution (pipeline flush).

DRVC / DRVNC

Drive Pins by C Flag {#drvnc}
Pin I/O and Smart Pins - Drives pins high or low based on C flag state.

DRVC *{#}Dest* **{WCZ}**

DRVNC *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the output direction and to an output level of low/high according to C or !C; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output direction and output levels of low or high.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001011010	OUT bit†	OUT bit†	OUT bit	2
EEEE	1101011	CZL	DDDDDDDDD	001011011	OUT bit†	OUT bit†	OUT bit	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: DRVZ, DRVNZ, DRVH, DRVL, DRVNOT, DRVRND

Explanation:

DRVC or **DRVNC** sets the I/O pin(s) designated by Dest to the output direction and to a low/high output level according to the C flag or its inverse (!C). All other pins are left unchanged.

DRVC sets the pin(s) to the output direction and to the level indicated by the C flag: high (1) for high output, low (0) for low output. **DRVNC** inverts this relationship, setting the output level according to the inverse of the C flag (!C).

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group; it will not cross the port boundary.

If the WCZ effect is specified, the C flag is set to the original state of the base OUT bit, and Z is set to the same value.

DRVH

Drive Pins High
Pin I/O and Smart Pins - Sets pins to output direction and drives high.

DRVH *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the output direction and to an output level of high; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output direction and high output level.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001011001	OUT bit†	OUT bit†	OUT bit	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: DRVL, DRVC, DRVNC, DRVZ, DRVNZ

Explanation:

DRVH sets the I/O pin(s) designated by Dest to the output direction and to a high output level. All other pins are left unchanged.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group; it will not cross the port boundary.

If the WCZ effect is specified, the C flag is set to the original state of the base OUT bit, and Z is set to the same value.

DRVL

Drive Pins Low
Pin I/O and Smart Pins - Sets pins to output direction and drives low.

DRVL *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the output direction and to an output level of low; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output direction and low output level.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001011000	OUT bit†	OUT bit†	OUT bit	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: DRVH, DRVC, DRVNC, DRVZ, DRVNZ

Explanation:

DRVL sets the I/O pin(s) designated by Dest to the output direction and to a low output level. All other pins are left unchanged.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group; it will not cross the port boundary.

If the WCZ effect is specified, the C flag is set to the original state of the base OUT bit, and Z is set to the same value.

Note that the new DIRx state is not data-forwarded; the next pipelined instruction sees the old state.

DRVNOT

Drive Not

Pin I/O and Smart Pins - Sets pins to output direction and toggles output level.

DRVNOT *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the output direction and to their opposite output level(s); the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to the output direction and toggle to opposite output levels.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001011111	OUT bit†	OUT bit†	OUT bit	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: DRVRND, DRVH, DRVL, DRVC, DRVNC, DRVZ, DRVNZ

Explanation:

DRVNOT sets the I/O pin(s) designated by Dest to the output direction and toggles their output level(s) to the opposite state. All other pins are left unchanged. This instruction achieves the same effect as two instructions—OUTNOT followed by **DIRH**.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **DRVNOT** instruction; substituting **SETQ**'s Dest[4:0] in place of value bits[10:6], for **DRVNOT**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group (DIRA or DIRB and OUTA or OUTB); it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of OUTA / OUTB's base bit, identified by Dest.

Note that the new DIRx state is not data-forwarded; the next pipelined instruction sees the old state.

DRVZ / DRVNZ

Drive Pins by Z Flag {#drvnz}

Pin I/O and Smart Pins - Drives pins high or low based on Z flag state.

DRVZ *{#}Dest* **{WCZ}**

DRVNZ *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the output direction and to an output level of low/high according to Z or !Z; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to output direction and output levels of low or high.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001011100	OUT bit†	OUT bit†	OUT bit	2
EEEE	1101011	CZL	DDDDDDDDD	001011101	OUT bit†	OUT bit†	OUT bit	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: DRVC, DRVNC, DRVH, DRVL, DRVNOT, DRVRND

Explanation:

DRVZ or **DRVNZ** sets the I/O pin(s) designated by Dest to the output direction and to a low/high output level according to the Z flag or its inverse (!Z). All other pins are left unchanged.

DRVZ sets the pin(s) to the output direction and to the level indicated by the Z flag: high (1) for high output, low (0) for low output. **DRVNZ** inverts this relationship, setting the output level according to the inverse of the Z flag (!Z).

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group; it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are set to the original state of the base OUT bit.

DRVRND

Drive Random
Pin I/O and Smart Pins - Sets pins to output direction with random output levels.

DRVRND *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the output direction and each output level is set randomly low or high; the rest are left as-is.

- Dest is the register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to the output direction and with output level(s) set randomly to low or high.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001011110	OUT bit†	OUT bit†	DIRx, OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: DRVH, DRVL, DRVC, DRVNC, DRVZ, DRVNZ, DRVNOT

Explanation:

DRVRND sets the I/O pin(s) designated by Dest to the output direction and with output level(s) set randomly low and high, based on bit(s) from the Xoroshiro128** PRNG. All other pins are left unchanged. This instruction can affect one or more of the bits within the DIRA or DIRB and OUTA or OUTB registers.

DRVRND achieves the same effect as two instructions—OUTRND followed by **DIRH**.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value—this inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **DRVRND** instruction; substituting **SETQ**'s Dest[4:0] in place of value bits[10:6], for **DRVRND**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) will wrap within the same 32-pin group (DIRA or DIRB and OUTA or OUTB); it will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of OUTA / OUTB's base bit, identified by Dest, before the random modification occurs.

Note that the new DIRx state is not data-forwarded; the next pipelined instruction sees the old state.

Instructions: E

This section contains all PASM2 instructions beginning with the letter E.

ENCOD

Encode Bit Position
Arithmetic Operations - Returns the position of the highest set bit.

ENCOD *Dest, {#}Src* **{WC|WZ|WCZ}**

ENCOD *Dest* **{WC|WZ|WCZ}**

Result: The bit position value of the top-most high bit (1) in Src, or Dest, is stored in Dest.

- Dest is a register in which to store the encoded bit position value and optionally contains the 32-bit value to encode (syntax 2).
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is to be encoded into a bit position.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0111100	CZI	DDDDDDDDD	SSSSSSSSS	S != 0	result == 0	D	2
EEEE	0111100	CZ0	DDDDDDDDD	DDDDDDDDD	Original D != 0	result == 0	D	2

Related: DECOD

Explanation:

ENCOD stores the bit position value (0-31) of the top-most high bit (1) of Src, or Dest, into Dest. The instruction scans from the most significant bit (bit 31) down to the least significant bit (bit 0) and returns the position of the first 1 bit encountered.

If the WC or WCZ effect is specified, the C flag is set (1) if Src (or original Dest in syntax 2) was not zero, or is cleared (0) if it was zero. This allows distinguishing between an input value of 1 (which encodes to 0) versus an input value of 0 (which also produces a result of 0).

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if not zero.

For example:

- `%00000000_00000000_00000000_00000001` encodes to 0 (bit position of the only 1)
- `%00000000_00000000_00000000_00100000` encodes to 5 (bit position 5 is the top-most 1)
- `%00000000_00000000_10000001_01000000` encodes to 15 (bit position 15 is the top-most 1)
- `%00000000_00000000_00000000_00000000` encodes to 0 with C flag cleared to 0

If the value to encode may be 0, use the WC or WCZ effect and check the resulting C flag to distinguish between the cases of input = 1 versus input = 0. Without this flag check, both cases would produce a Dest value of 0.

ENCOD is the complement of **DECOD**. Where **DECOD** converts a bit position (0-31) into a 32-bit value with a single bit set, **ENCOD** performs the reverse operation, converting a 32-bit value into the position of its highest set bit.

EXECF

Execute with **SKIP** Pattern
Branching and Flow Control - Jumps to address with **SKIP** pattern for conditional execution.

EXECF *{#}Dest*

Result: PC is set to Dest[9:0] and the SKIPF pattern is set to Dest[31:10].

- Dest is a register or immediate value 0-511 (augmentable to a full 32-bit value via **AUGD**). Bits [9:0] of the resulting Dest value specify the target COG/LUT address and bits [31:10] specify the 22-bit **SKIP** pattern.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110011	—	—	—	4

Related: CALL, SKIPF, SKIP

Explanation:

EXECF performs a combined jump and **SKIP** pattern operation. The instruction sets the program counter (PC) to the 10-bit address specified in Dest[9:0] and simultaneously loads the **SKIPF** pattern register with the value from Dest[31:10].

The PC is set to the address formed by zero-extending Dest[9:0] to create a COG/LUT address: PC = {10'b0, Dest[9:0]}. This allows jumping to any location within the 1024-address COG/LUT memory space (addresses 0-511 for COG, 512-1023 for LUT).

The **SKIPF** pattern in Dest[31:10] provides a 22-bit pattern that controls which subsequent instructions will be skipped after the jump. Like **SKIPF**, this allows the PC to leap over instructions rather than cancelling them, providing fast conditional execution without the overhead of traditional branch instructions.

EXECF combines the functionality of **CALL** (jumping to a new address) and **SKIPF** (setting a **SKIP** pattern), enabling efficient implementation of computed branches with conditional execution. This is particularly useful for jump tables and state machines where both the target address and subsequent execution pattern need to be determined dynamically.

The instruction takes 4 clock cycles to execute, regardless of whether it executes from COG/LUT or Hub memory.

Instructions: F

This section contains all PASM2 instructions beginning with the letter F.

FBLOCK

Set Next FIFO Block
Hub Memory Access - Configures the next block for FIFO wraparound operations.

FBLOCK *{#}Dest, {#}Src*

Result: The next block parameters are configured for FIFO wraparound operations.

- Dest is a register or 9-bit literal whose value specifies the block size in 64-byte units (0 = maximum size).
- Src is a register or 9-bit literal whose value specifies the block start address in Hub memory.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100100	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: RDFAST, WRFAST, RFLONG, WFLONG

Explanation:

FBLOCK configures the parameters for the next Hub FIFO block that will be used when the current block wraps around. This instruction is used to set up circular buffering in Hub memory for streaming read and write operations.

Dest[13:0] specifies the block size in 64-byte units. A value of 0 represents the maximum block size. The block size determines how many bytes can be transferred before the FIFO wraps to the beginning of the block.

Src[19:0] specifies the starting address of the block in Hub memory. This address marks where the FIFO will wrap to when it reaches the end of the current block.

FBLOCK is typically used in conjunction with **RDFAST**/WRFAST for setting up high-throughput data streaming between Hub memory and COG/LUT memory. The block configuration takes effect when the current FIFO operation completes and wraps around.

FGE

Force Greater or Equal
Arithmetic Operations - Forces unsigned Dest to be at least Src (minimum clamp).

FGE *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Unsigned Dest is set to unsigned Src if Dest was less than Src.

- Dest is a register containing the unsigned value to limit to a minimum of unsigned Src, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose unsigned value is the lower limit to force upon Dest.

- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0011000	CZI	DDDDDDDDD	SSSSSSSSS	limit enforced†	result == 0	D	2

† C = 1 if limit was enforced (D changed), else C = 0 (D unchanged).

Related: FLE, FGES, FLES

Explanation:

FGE sets unsigned Dest to unsigned Src if Dest is less than Src. This is a limit minimum function that prevents Dest from sinking below the value of Src. If Dest is already greater than or equal to Src, Dest remains unchanged.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was limited (Dest was less than Src and is now equal to Src), or is cleared (0) if not limited (Dest was already greater than or equal to Src).

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero.

FGE is useful for clamping values to a minimum threshold, ensuring that a value never falls below a specified floor. This is commonly used in digital signal processing, graphics calculations, and boundary checking where values must stay within valid ranges.

FGES

Force Greater or Equal Signed
Arithmetic Operations - Forces signed Dest to be at least Src (minimum clamp).

FGES *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Signed Dest is set to signed Src if Dest was less than Src.

- Dest is a register containing the signed value to limit to a minimum of signed Src, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose signed value is the lower limit to force upon Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0011010	CZI	DDDDDDDDD	SSSSSSSSS	limit enforced†	result == 0	D	2

† C = 1 if limit was enforced (D changed), else C = 0 (D unchanged).

Related: FLES, FGE, FLE

Explanation:

FGES sets signed Dest to signed Src if Dest is less than Src. This is a limit minimum function that prevents Dest from sinking below the signed value of Src. If Dest is already greater than or equal to Src, Dest remains unchanged. The comparison and limiting are performed treating both operands as signed 32-bit values.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was limited (Dest was less than Src and is now equal to Src), or is cleared (0) if not limited (Dest was already greater than or equal to Src).

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero.

FGES is the signed counterpart to **FGE** and is used when working with signed values that need to be clamped to a minimum threshold. This is particularly useful in audio processing, control systems, and any application where signed values must be constrained within bounds.

FLE

Force Less or Equal
Arithmetic Operations - Forces unsigned Dest to be at most Src (maximum clamp).

FLE *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Unsigned Dest is set to unsigned Src if Dest was greater than Src.

- Dest is a register containing the unsigned value to limit to a maximum of unsigned Src, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose unsigned value is the upper limit to force upon Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0011001	CZI	DDDDDDDDD	SSSSSSSSS	limit enforced†	result == 0	D	2

† C = 1 if limit was enforced (D changed), else C = 0 (D unchanged).

Related: FGE, FLES, FGES

Explanation:

FLE sets unsigned Dest to unsigned Src if Dest is greater than Src. This is a limit maximum function that prevents Dest from rising above the value of Src. If Dest is already less than or equal to Src, Dest remains unchanged.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was limited (Dest was greater than Src and is now equal to Src), or is cleared (0) if not limited (Dest was already less than or equal to Src).

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero.

FLE is useful for clamping values to a maximum threshold, ensuring that a value never exceeds a specified ceiling. This is commonly used in digital signal processing, graphics calculations, and boundary checking where values must stay within valid ranges.

FLES

Force Less or Equal Signed
Arithmetic Operations - Forces signed Dest to be at most Src (maximum clamp).

FLES *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Signed Dest is set to signed Src if Dest was greater than Src.

- Dest is a register containing the signed value to limit to a maximum of signed Src, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose signed value is the upper limit to force upon Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0011011	CZI	DDDDDDDDD	SSSSSSSSS	limit enforced†	result == 0	D	2

† C = 1 if limit was enforced (D changed), else C = 0 (D unchanged).

Related: FGES, FLE, FGE

Explanation:

FLES sets signed Dest to signed Src if Dest is greater than Src. This is a limit maximum function that prevents Dest from rising above the signed value of Src. If Dest is already less than or equal to Src, Dest remains unchanged. The comparison and limiting are performed treating both operands as signed 32-bit values.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was limited (Dest was greater than Src and is now equal to Src), or is cleared (0) if not limited (Dest was already less than or equal to Src).

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero.

FLES is the signed counterpart to **FLE** and is used when working with signed values that need to be clamped to a maximum threshold. This is particularly useful in audio processing, control systems, and any application where signed values must be constrained within bounds.

FLTC / FLTNC / FLTZ / FLTNZ

Float with Output Preset by Flag

Pin I/O and Smart Pins - Sets pins to input direction with output preset by flag state.

FLTC *{#}Dest* **{WCZ}**

FLTNC *{#}Dest* **{WCZ}**

FLTZ *{#}Dest* **{WCZ}**

FLTNZ *{#}Dest* **{WCZ}**

Result: The I/O pins are set to input direction with output preset according to flag state. Optionally sets Z to original output state.

- Dest identifies the I/O pin(s): Dest[5:0] = base pin (0-63), Dest[10:6] = additional contiguous pins.
- WCZ is an optional effect to set Z to the original output state.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001010010	OUT bit†	OUT bit†	OUTx	2
EEEE	1101011	CZL	DDDDDDDDD	001010011	OUT bit†	OUT bit†	OUTx	2
EEEE	1101011	CZL	DDDDDDDDD	001010100	OUT bit†	OUT bit†	OUTx	2
EEEE	1101011	CZL	DDDDDDDDD	001010101	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: FLTH, FLTL, FLTNOT, FLTRND

Explanation:

These instructions set pin(s) to input direction (floating) while pre-setting the output register based on flag state:

Instruction	Presets output high when
FLTC	C = 1
FLTNC	C = 0
FLTZ	Z = 1
FLTNZ	Z = 0

When the pin is later driven as output, it will immediately be at the desired level. **FLTC** and **FLTZ** preset output high when their flag is set; **FLTNC** and **FLTNZ** preset output high when their flag is clear.

If WCZ is specified, the C and Z flags are set to the original output state of the base pin.

Pipeline Note: The new DIRx state is not data-forwarded to subsequent instructions; only the OUTx state is forwarded (the P2 has only one forwarding path, and OUT was prioritized). Any instruction that reads or modifies DIRx should be placed at least two instructions after a FLT instruction to see the updated direction state.

FLTH

Float High
Pin I/O and Smart Pins - Sets pins to input direction with output preset high.

FLTH *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the input direction and to an output level of high.

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to input direction and output level of high.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001010001	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: FLTL, FLTC, FLTNC, FLTZ, FLTNZ

Explanation:

FLTH sets the I/O pin(s) designated by Dest to the input direction (floating) and to a high output level. All other pins are left unchanged. This pre-sets the output register so that when the pin is later driven as output, it will immediately be high.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value, which inserts an **AUGD** instruction prior.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) wraps within the same 32-pin group and will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are set to the original state of the OUTA/OUTB base bit identified by Dest.

Pipeline Note: The new DIRx state is not data-forwarded to subsequent instructions; only the OUTx state is forwarded (the P2 has only one forwarding path, and OUT was prioritized). Any instruction that reads or modifies DIRx should be placed at least two instructions after **FLTH** to see the updated direction state.

FLTL

Float Low
Pin I/O and Smart Pins - Sets pins to input direction with output preset low.

FLTL *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the input direction and to an output level of low.

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to input direction and output level of low.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001010000	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: FLTH, FLTC, FLTNC, FLTZ, FLTNZ

Explanation:

FLTL sets the I/O pin(s) designated by Dest to the input direction (floating) and to a low output level. All other pins are left unchanged. This pre-sets the output register so that when the pin is later driven as output, it will immediately be low.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value, which inserts an **AUGD** instruction prior.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) wraps within the same 32-pin group and will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are set to the original state of the OUTA/OUTB base bit identified by Dest.

Pipeline Note: The new DIRx state is not data-forwarded to subsequent instructions; only the OUTx state is forwarded (the P2 has only one forwarding path, and OUT was prioritized). Any instruction that reads or modifies DIRx should be placed at least two instructions after **FLTL** to see the updated direction state.

FLTNOT

Float Not
Pin I/O and Smart Pins - Sets pins to input direction with output toggled.

FLTNOT *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the input direction and to their opposite output level(s).

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to the input direction and toggle to opposite output levels.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001010111	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: FLTC, FLTNC, FLTZ, FLTNZ, FLTRND

Explanation:

FLTNOT sets the I/O pin(s) designated by Dest to the input direction (floating) and toggles their output level(s) to the opposite state. All other pins are left unchanged. **FLTNOT** achieves the same effect as two instructions: **DIRL** followed by **OUTNOT**.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value, which inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **FLTNOT** instruction, in which case **SETQ**'s Dest[4:0] substitutes in place of value bits[10:6] for **FLTNOT**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) wraps within the same 32-pin group (DIRA or DIRB and OUTA or OUTB) and will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of OUTA/OUTB's base bit identified by Dest.

Pipeline Note: The new DIRx state is not data-forwarded to subsequent instructions; only the OUTx state is forwarded (the P2 has only one forwarding path, and OUT was prioritized). Any instruction that reads

or modifies DIRx should be placed at least two instructions after **FLTNOT** to see the updated direction state.

FLTRND

Float Random
Pin I/O and Smart Pins - Sets pins to input direction with random output levels.

FLTRND *{#}Dest* **{WCZ}**

Result: The I/O pins described by Dest are set to the input direction and each output level is set randomly low or high.

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to the input direction and with output level(s) set randomly to low or high.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001010110	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: FLTC, FLTNC, FLTZ, FLTNZ, FLTH, FLTL, FLTNOT

Explanation:

FLTRND sets the I/O pin(s) designated by Dest to the input direction and with output level(s) set randomly low and high, based on bit(s) from the Xoroshiro128** PRNG. All other pins are left unchanged. This instruction can affect one or more of the bits within the DIRA or DIRB and OUTA or OUTB registers.

FLTRND achieves the same effect as two instructions: **DIRL** followed by **OUTRND**.

Dest[5:0] indicates the pin number (0-63). For a range of pins, Dest[5:0] indicates the base pin number (0-63) and Dest[10:6] indicates how many contiguous pins beyond the base should be affected (1-31).

A 9-bit literal Dest is enough to express the base pin (Dest[5:0]) and a range of up to 8 contiguous pins (Dest[8:6]). If needed, use the augmented literal feature (##Dest) to augment Dest to an 11-bit literal value, which inserts an **AUGD** instruction prior.

When Dest is a register, the register's value bits [10:0] are used as-is to form the 11-bit ID range, unless a **SETQ** instruction immediately precedes the **FLTRND** instruction, in which case **SETQ**'s Dest[4:0] substitutes in place of value bits[10:6] for **FLTRND**'s use.

The range calculation (from Dest[5:0] up to Dest[5:0]+Dest[10:6]) wraps within the same 32-pin group (DIRA or DIRB and OUTA or OUTB) and will not cross the port boundary.

If the WCZ effect is specified, the C and Z flags are updated to the original state of OUTA/OUTB's base bit identified by Dest.

Pipeline Note: The new DIRx state is not data-forwarded to subsequent instructions; only the OUTx state is forwarded (the P2 has only one forwarding path, and OUT was prioritized). Any instruction that reads or modifies DIRx should be placed at least two instructions after **FLTRND** to see the updated direction state.

Instructions: G

This section contains all PASM2 instructions beginning with the letter G.

GETBRK

Get Breakpoint Status
Interrupts - Retrieves breakpoint or COG status information.

GETBRK *Dest* **{WC|WZ|WCZ}**

Result: Breakpoint or COG status information is retrieved into Dest based on the flag effect specified.

- Dest is a register where the status information is written.
- WC, WZ, or WCZ are optional effects that determine which status information is retrieved.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000110101	—	—	D	2

Related: BRK, COGBRK

Explanation:

GETBRK retrieves various breakpoint and COG status information into the Dest register. The specific information retrieved depends on which flag effect is specified.

When the WCZ effect is specified, **GETBRK** retrieves the full 32-bit ISR **CALL** address into Dest. This is the address where the debug interrupt service routine will resume execution after handling the breakpoint.

When the WC effect is specified, **GETBRK** retrieves the 8-bit COG ID into Dest[7:0]. This identifies which COG triggered the breakpoint, useful in multi-COG debugging scenarios where a debug ISR needs to determine the calling COG.

When the WZ effect is specified, **GETBRK** retrieves the 8-bit breakpoint code into Dest[7:0]. This code was set by the **BRK** instruction and can be used for conditional breakpoint handling or to distinguish between different types of breakpoints.

When no flag effects are specified, **GETBRK** retrieves the 16-bit **SKIP** pattern into Dest[15:0]. This pattern is used with the **SKIPF** instruction to selectively execute or **SKIP** subsequent instructions, typically within an ISR context.

GETBRK is essential for implementing **DEBUG** infrastructure and coordinating multi-COG debugging systems. It works in conjunction with **BRK** and SETBRK to provide comprehensive breakpoint support.

GETBYTE

Get **BYTE**
Arithmetic Operations - Extracts a specified **BYTE** from a 32-bit value.

GETBYTE *Dest, {#}Src, #Num*

GETBYTE *Dest*

Result: Byte Num (0-3) of Src, or a byte from a source described by prior ALTGB instruction, is written to Dest.

- Dest is the register in which to store the byte.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value contains the target **BYTE** to read.
- Num is a 2-bit literal identifying the byte ID (0-3) of Src to read.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1000111	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1000111	000	DDDDDDDDD	000000000	—	—	D	2

Related: ALTGB, GETNIB, GETWORD, SETBYTE, ROLBYTE

Explanation:

GETBYTE reads the byte identified by Num (0-3) from Src and writes it to Dest. The Num parameter identifies which of the four bytes in the 32-bit value to extract, numbered in least-significant byte order.

Num 0 selects bits [7:0], Num 1 selects bits [15:8], Num 2 selects bits [23:16], and Num 3 selects bits [31:24]. The extracted **BYTE** is zero-extended to 32 bits when written to Dest.

The second syntax form (GETBYTE Dest) is intended for use after an **ALTGB** instruction. This form is useful in loops that iteratively read a series of byte values from contiguous **LONG** registers. The **ALTGB** instruction modifies the subsequent **GETBYTE** instruction's source register and **BYTE** index automatically, enabling efficient sequential **BYTE** extraction without explicitly specifying the source and index on each iteration.

GETCT

Get System Counter
Miscellaneous - Retrieves the current value of the system counter.

GETCT *Dest* **{WC}**

Result: The current value of the system counter CT is written to Dest.

- Dest is a register where the system counter value is written.
- WC is an optional effect to retrieve the upper 32 bits of the 64-bit counter (Rev B/C silicon).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	C00	DDDDDDDDD	000011010	—	—	D (CT[31:0], or CT[63:32] if WC)	2

Related: ADDCT1/2/3, WAITCT1/2/3

Explanation:

GETCT retrieves the current value of the system counter CT into the Dest register. On **REV** B/C silicon, the system counter is a 64-bit counter that is reset to zero on system reset and increments by one on every clock cycle. By default, the lower 32 bits (CT[31:0]) are returned in Dest.

The CT counter provides a continuous, monotonic time reference. The lower 32 bits wrap around from $FFFF_FFFF to $0000_0000 approximately every 21.5 seconds at 200 MHz. This counter is shared across all COGs and provides the foundation for timing operations and synchronization.

64-bit Counter (Rev B/C): If the WC effect is specified, the upper 32 bits of the 64-bit counter (CT[63:32]) are written to Dest instead of the lower 32 bits. To capture a full 64-bit timestamp, use two consecutive **GETCT** instructions:

```
1         GETCT   low_word        ' Get lower 32 bits (CT[31:0])
2         GETCT   high_word wc    ' Get upper 32 bits (CT[63:32])
```

GETCT is commonly used with the ADDCT and WAITCT instruction families to implement precise timing, delays, and event scheduling. The retrieved counter value serves as a time reference for calculating future wait points or measuring elapsed time intervals.

GETNIB

Get Nibble

Arithmetic Operations - Extracts a specified nibble from a 32-bit value.

GETNIB *Dest, {#}Src, #Num*

GETNIB *Dest*

Result: Nibble Num (0-7) of Src, or a nibble from a source described by prior ALTGN instruction, is written to Dest.

- Dest is the register in which to store the nibble.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value contains the target nibble to read.
- Num is a 3-bit literal identifying the nibble ID (0-7) of Src to read.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	100001N	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1000010	000	DDDDDDDDD	000000000	—	—	D	2

Related: ALTGN, GETBYTE, GETWORD, SETNIB, ROLNIB

Explanation:

GETNIB reads the nibble identified by Num (0-7) from Src and writes it to Dest. The Num parameter identifies which of the eight nibbles in the 32-bit value to extract, numbered in least-significant nibble order.

Num 0 selects bits [3:0], Num 1 selects bits [7:4], Num 2 selects bits [11:8], and so on up to Num 7 which selects bits [31:28]. The extracted nibble is zero-extended to 32 bits when written to Dest.

The second syntax form (GETNIB Dest) is intended for use after an **ALTGN** instruction. This form is useful in loops that iteratively read a series of nibble values from contiguous **LONG** registers. The **ALTGN** instruction modifies the subsequent **GETNIB** instruction's source register and nibble index automatically, enabling efficient sequential nibble extraction without explicitly specifying the source and index on each iteration.

GETPTR

Get FIFO Hub Pointer

Hub Memory Access - Retrieves the current FIFO hub pointer position.

GETPTR *Dest*

Result: The current FIFO hub pointer is written to Dest.

- Dest is a register where the FIFO hub pointer is written.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	000110100	—	—	D	2

Related: RDFAST, WRFAST, RFBYTE, RFWORD, RFLONG, WFBYTE, WFWORD, WFLONG

Explanation:

GETPTR retrieves the current position of the hub FIFO pointer into the Dest register. This pointer tracks the current hub memory address for FIFO read and write operations initiated by **RDFAST** or **WRFAST**.

The hub FIFO pointer advances automatically as data is read from or written to the hub FIFO using the **RFBYTE**, **RFWORD**, **RFLONG**, **WFBYTE**, **WFWORD**, or **WFLONG** instructions. Each FIFO access increments the pointer by the size of the data transferred (1 **BYTE**, 2 bytes, or 4 bytes).

GETPTR is useful for monitoring FIFO transfer progress, calculating how much data has been transferred, or determining the current position within a buffer. The retrieved pointer value represents the hub memory address that will be accessed by the next FIFO read or write operation.

GETQX

Get CORDIC X Result

CORDIC Coprocessor - Retrieves the X result from the CORDIC solver.

GETQX *Dest* **{WC|WZ|WCZ}**

Result: The CORDIC X result is written to Dest after waiting if necessary for the computation to complete.

- Dest is a register where the CORDIC X result is written.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000011000	X[31]	result == 0	D	2…58

Related: GETQY, QROTATE, QVECTOR, QMUL, QDIV, QFRAC, QSQRT, QLOG, QEXP

Explanation:

GETQX retrieves the X result from the CORDIC solver into the Dest register. If the CORDIC computation is not yet complete when **GETQX** executes, the instruction waits until the result is ready before retrieving it and continuing execution.

The CORDIC solver performs various mathematical operations including rotation, vectoring, multiplication, division, square root, logarithm, and exponentiation. Each operation produces two results, X and Y, which are retrieved using **GETQX** and **GETQY** respectively.

If the WC or WCZ effect is specified, the C flag is set to X[31], which is the sign bit of the result. This allows immediate determination of whether the result is negative (C = 1) or non-negative (C = 0) when interpreting the result as a signed value.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero.

GETQX takes 2 clocks if the result is already available. If the result is not yet ready, **GETQX** waits until the CORDIC computation completes (up to 58 clocks from when the operation was queued).

GETQY

Get CORDIC Y Result
CORDIC Coprocessor - Retrieves the Y result from the CORDIC solver.

GETQY *Dest* **{WC|WZ|WCZ}**

Result: The CORDIC Y result is written to Dest after waiting if necessary for the computation to complete.

- Dest is a register where the CORDIC Y result is written.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000011001	Y[31]	result == 0	D	2…58

Related: GETQX, QROTATE, QVECTOR, QMUL, QDIV, QFRAC, QSQRT, QLOG, QEXP

Explanation:

GETQY retrieves the Y result from the CORDIC solver into the Dest register. If the CORDIC computation is not yet complete when **GETQY** executes, the instruction waits until the result is ready before retrieving it and continuing execution.

The CORDIC solver performs various mathematical operations including rotation, vectoring, multiplication, division, square root, logarithm, and exponentiation. Each operation produces two results, X and Y, which are retrieved using **GETQX** and **GETQY** respectively.

If the WC or WCZ effect is specified, the C flag is set to Y[31], which is the sign bit of the result. This allows immediate determination of whether the result is negative (C = 1) or non-negative (C = 0) when interpreting the result as a signed value.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero.

GETQY takes 2 clocks if the result is already available. If the result is not yet ready, **GETQY** waits until the CORDIC computation completes (up to 58 clocks from when the operation was queued).

GETRND

Get Random Value

Miscellaneous - Retrieves a pseudo-random value from the COG's RNG.

GETRND *Dest* **{WC|WZ|WCZ}**

GETRND {WC|WZ|WCZ}

Result: The current pseudo-random value is written to Dest, or the random bits are stored in the C and Z flags.

- Dest is a register where the full 32-bit random value is written (first syntax).
- WC, WZ, or WCZ are optional effects to retrieve random bits into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000011011	RND[31]	RND[30], unique per cog	D	2
EEEE	1101011	CZ1	000000000	000011011	RND[31]	RND[30], unique per cog	—	2

Related: SETQ, SETQ2

Explanation:

GETRND retrieves the current value from the pseudo-random number generator (RNG) that is unique to each COG. Each COG maintains its own independent RNG state that advances continuously.

The first syntax form (GETRND Dest) writes the full 32-bit random value to the Dest register. This provides a complete random **WORD** for applications requiring random data, random seeds, or probabilistic algorithms.

The second syntax form (GETRND without Dest) is used when only random flag bits are needed. This form requires at least one flag effect to be specified, otherwise the instruction has no visible effect.

If the WC or WCZ effect is specified, the C flag is set to RND[31], which is the most significant bit of the current random value.

If the WZ or WCZ effect is specified, the Z flag is set to RND[30]. Notably, RND[30] is unique per COG, meaning each COG's RNG produces independent bit sequences at this position, useful for multi-COG systems requiring independent randomness.

The random value is produced by the P2's Xoroshiro128** pseudo-random number generator, which has 128 bits of state, advances every clock cycle, and has an extremely long period (2^{}128 - 1).

GETSCP

Get Oscilloscope Samples

Pin I/O and Smart Pins - Retrieves four 8-bit oscilloscope samples.

GETSCP *Dest*

Result: Four 8-bit oscilloscope samples are written to Dest as D = {ch3[7:0], ch2[7:0], ch1[7:0], ch0[7:0]}.

- Dest is a register where the four oscilloscope samples are written.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001110001	—	—	D	2

Related: SETSCP, RDPIN, WXPIN

Explanation:

GETSCP retrieves the current samples from the four-channel digital oscilloscope into the Dest register. The oscilloscope continuously samples four independent channels and packs the 8-bit sample values into a single 32-bit **WORD**.

The four samples are arranged in the Dest register with channel 0 in bits [7:0], channel 1 in bits [15:8], channel 2 in bits [23:16], and channel 3 in bits [31:24]. Each channel provides an 8-bit unsigned sample value ranging from 0 to 255.

The oscilloscope is configured using the **SETSCP** instruction to specify which pins or signals each channel monitors. Once configured, the oscilloscope continuously updates its samples based on the monitored signals, and **GETSCP** can retrieve the latest samples at any time.

This instruction is useful for real-time signal monitoring, debugging, and creating oscilloscope-like functionality for analyzing digital signals or pin states within the P2 system.

GETWORD

Get **WORD**

Arithmetic Operations - Extracts a specified **WORD** from a 32-bit value.

GETWORD *Dest, {#}Src, #Num*

GETWORD *Dest*

Result: Word Num (0-1) of Src, or a word from a source described by prior ALTGW instruction, is written to Dest.

- Dest is the register in which to store the word.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value contains the target **WORD** to read.
- Num is a 1-bit literal identifying the word ID (0-1) of Src to read.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001001	1NI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001001	100	DDDDDDDDD	000000000	—	—	D	2

Related: ALTGW, GETNIB, GETBYTE, SETWORD, ROLWORD

Explanation:

GETWORD reads the word identified by Num (0-1) from Src and writes it to Dest. The Num parameter identifies which of the two words in the 32-bit value to extract, numbered in least-significant **WORD** order.

Num 0 selects bits [15:0] (the lower **WORD**), and Num 1 selects bits [31:16] (the upper **WORD**). The extracted **WORD** is zero-extended to 32 bits when written to Dest.

The second syntax form (GETWORD Dest) is intended for use after an **ALTGW** instruction. This form is useful in loops that iteratively read a series of word values from contiguous **LONG** registers. The **ALTGW** instruction modifies the subsequent **GETWORD** instruction's source register and **WORD** index automatically, enabling efficient sequential **WORD** extraction without explicitly specifying the source and index on each iteration.

GETXACC

Get Goertzel Accumulators

Streamer - Retrieves Goertzel X and Y accumulators from the streamer.

GETXACC *Dest*

Result: The streamer's Goertzel X accumulator is written to Dest, the Y accumulator is written to the next instruction's S field, and both accumulators are cleared.

- Dest is a register where the Goertzel X accumulator value is written.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	000011110	—	—	D	2

Related: XCONT, XINIT, XZERO

Explanation:

GETXACC retrieves the two Goertzel accumulators from the streamer, which are used for frequency detection and digital signal processing applications. The Goertzel algorithm accumulates signal correlation data that can be used to detect specific frequencies in an input signal.

The X accumulator value is written directly to the Dest register. The Y accumulator value is written to the S field of the immediately following instruction, utilizing the P2's next-instruction operand modification capability. After both values are retrieved, the X and Y accumulators are automatically cleared to zero.

This dual-retrieval mechanism allows both accumulator values to be captured in a compact instruction sequence. The following instruction must have an S field that can receive the Y accumulator value. Typically, this is a **MOV** or similar instruction where the S operand receives the Y accumulator data.

GETXACC is used in conjunction with the streamer's Goertzel mode, configured via **XINIT** and controlled via **XCONT**. The retrieved accumulator values represent the correlation between the input signal and the reference frequency configured in the Goertzel algorithm.

Instructions: H

This section contains all PASM2 instructions beginning with the letter H.

HUBSET

Set Hub Configuration
Hub Control - Configures hub clock system, crystal, and PLL settings.

HUBSET *{#}D*

Result: Hub configuration is updated according to the value in D, controlling clock source, crystal settings, and PLL configuration.

- D is a register or 9-bit literal (or 32-bit augmented literal) containing the configuration value for the hub system.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000000000	—	—	—	2…9

Related: COGINIT, COGID

Explanation:

HUBSET configures the P2's clock system and hub parameters. The 32-bit value in D specifies clock source selection, crystal oscillator settings, and PLL configuration to control the system clock frequency.

The D value contains multiple fields that control different aspects of the clock system:

Clock Source Selection (D[1:0]): - `%00` - RCFAST internal oscillator (~20-25 MHz, boot default) - `%01` - RCSLOW internal oscillator (~20 kHz, low power mode) - `%10` - Crystal or external clock on XI pin - `%11` - PLL output

Crystal Configuration (D[3:2]): - `%00` - XI/XO pins disabled (Hi-Z) - `%01` - XI/XO with 1MΩ feedback, no capacitors - `%10` - XI/XO with 1MΩ feedback, 15pF capacitors - `%11` - XI/XO with 1MΩ feedback, 30pF capacitors

PLL Configuration: - D[23:18] - Input divider (DDDDDD field, divides XI input by 1-64; stored as divider-1) - D[17:8] - VCO multiplier (MMMMMMMMMM, 10-bit; multiplies by 1-1024; stored as multiplier-1) - D[7:4] - Post divider (PPPP field): VCO/2, VCO/4, …, VCO/30 for PPPP=0..14, and VCO/1 for PPPP=15 (the fast-overclock mode) - D[24] - PLL power enable (E) - Note: the XI oscillator is enabled by the crystal-config field CC != %00, not by a dedicated bit.

System Reset: - D[31] - Write 1 to reset the entire chip

The clock switching is glitch-free, and the system automatically falls back to RCFAST if the selected clock source fails. Proper timing must be observed when switching clock sources to allow for oscillator stabilization.

Example: Enable a 20 MHz crystal with 15pF capacitors:

```
        HUBSET  ##%10_00              ' Enable 15pF crystal, stay RCFAST
        WAITX   ##20_000_000/100      ' Wait 10ms for stabilization
        HUBSET  ##%10_10              ' Switch to crystal clock
```

Example: Configure PLL to generate 160 MHz from a 20 MHz crystal:

```
        HUBSET  ##%10_00          ' Enable 15pF crystal, stay RCFAST
        WAITX   ##20_000_000/100  ' Wait 10ms
        HUBSET  ##%10_10          ' Switch to crystal
        ' PLL on, /1 * 16 / 2, stay on XI while PLL locks:
        HUBSET  ##%1_000000_0000001111_0000_10_10
        WAITX   ##20_000_000/10000  ' Wait 100µs for PLL lock
        ' Switch to PLL output:
        HUBSET  ##%1_000000_0000001111_0000_10_11
```

In this PLL example, the VCO runs at 20 MHz * 16 = 320 MHz, then the post divider divides by 2 to produce 160 MHz system clock.

HUBSET takes 2-9 clock cycles to execute depending on Hub window alignment. Switching to a new clock source may take additional time for oscillator stabilization and PLL lock. Always allow appropriate wait periods when changing clock sources.

Instructions: I

This section contains all PASM2 instructions beginning with the letter I.

Conditional Jump Timing Convention: Conditional jumps in this section (IJZ, **IJNZ**) show their `Clks` field as `NOT-taken / taken`. The *taken* value depends on execution context:

Context	Clocks when taken
COG / LUT execution	4
Hub execution	13…20

So `2 OR 4 / 2 OR 13-20` reads as: 2 cycles when the jump is not taken, 4 cycles when taken in cog/LUT, 13–20 cycles when taken in hub execution.

IJZ / IJNZ

Increment and Jump If Zero {#ijnz}
Branching and Flow Control - Increments and conditionally jumps based on the result.

IJZ *Dest, {#}Src*

IJNZ *Dest, {#}Src*

Result: Dest is incremented by 1, and conditionally jumps based on the result.

- Dest is a register whose value is incremented and tested.
- Src is the jump address: use # for relative, omit for absolute.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011100	00I	DDDDDDDDD	SSSSSSSSS	—	—	D + PC*	2 or 4 / 2 or 13-20
EEEE	1011100	01I	DDDDDDDDD	SSSSSSSSS	—	—	D + PC*	2 or 4 / 2 or 13-20

*PC is written only when the jump condition is met.

Related: DJZ, DJNZ, TJZ, TJNZ

Explanation:

IJZ and **IJNZ** increment Dest and conditionally jump based on whether the result is zero or non-zero:

Instruction	Jumps when
IJZ	result == 0
IJNZ	Result != 0

IJZ is useful for counting until overflow to zero (from $FFFF_FFFF to 0). **IJNZ** is useful for counting up from a negative value until reaching zero.

Takes 2 clocks when not jumping, 4 clocks when jumping (pipeline flush).

INCMOD

Increment Modulus

Arithmetic Operations - Increments with modulus wrap-around.

INCMOD *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: If Dest was not equal to Src, it is incremented by 1; otherwise Dest is reset to 0.

- Dest is a register containing the value to increment up to Src with modulus, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is the modulus limit to apply to Dest's increment operation.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0111000	CZI	DDDDDDDDD	SSSSSSSSS	D was S (wrapped)	result == 0	D	2

Related: DECMOD, ADDCT1/2/3

Explanation:

INCMOD compares Dest with Src. If they are not equal, **INCMOD** increments Dest by 1. If they are equal, **INCMOD** sets Dest to 0. This provides automatic wrap-around behavior for circular counting sequences.

If Dest begins in the range 0 to Src, repeated iterations of **INCMOD** will increment Dest cyclically from 0 to Src, then wrap back to 0, over and over. This makes **INCMOD** ideal for round-robin scheduling, circular buffer indexing, and other modulo-arithmetic operations.

If the WC or WCZ effect is specified, the C flag is set (1) if Dest was equal to Src and subsequently reset to 0 (the modulus was triggered), or is cleared (0) if Dest was simply incremented. This allows detecting when the cycle completes.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if it is non-zero.

INCMOD does not limit Dest within the specified range. If Dest begins at a value greater than Src, iterations of **INCMOD** will continue to increment it through the 32-bit rollover point ($FFFF_FFFF wrapping to $0000_0000) before it will effectively cycle from 0 to Src.

A common usage pattern for **INCMOD** is managing circular buffers:

```
1                ' Increment tail index with modulo for circular buffer
2                INCMOD  tail_idx, #BUF_SIZE-1  wc
3        if_c    JMP     #buffer_wrapped
```

continues on next page →

↪ *continued from previous page*

```
4
5                 ' Safe to add data at tail
6                 ADD     buffer_ptr, tail_idx
7                 WRBYTE  new_data, buffer_ptr
```

INCMOD is also ideal for round-robin scheduling across a fixed number of resources:

```
1                 ' Round-robin through 8 ports (0-7)
2 .loop
3                 ' Service current port
4                 ' ... port service code ...
5
6                 ' Move to next port
7                 INCMOD  portctr, #7            wc
8         if_nc   JMP     #.loop
9
10                ' All ports serviced, continue
```

Instructions: J

This section contains all PASM2 instructions beginning with the letter J.

Conditional Jump Timing Convention: Conditional jumps (including event-jumps and counter-jumps) show their `Clks` field as `NOT-taken / taken`. The *taken* value depends on execution context:

Context	Clocks when taken
COG / LUT execution	4
Hub execution	13…20

So `2 OR 4 / 2 OR 13-20` reads as: 2 cycles when the jump is not taken (either context), 4 cycles when taken in cog/LUT, 13–20 cycles when taken in hub execution.

JATN / JNATN

Jump If Attention Set / Clear {#jnatn}
Events and Timing - Jumps based on ATN event flag state.

JATN *{#}S*

JNATN *{#}S*

Result: JATN jumps if the ATN event flag is set; JNATN jumps if the ATN event flag is clear.

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001110	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011110	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

PC is written only when the condition is met (flag set for **JATN**, flag clear for **JNATN**).

Related: COGATN, POLLATN

Explanation:

JATN checks the ATN (attention) event flag and conditionally jumps if the flag is set. **JNATN** performs the opposite **TEST**, jumping if the flag is clear. The ATN event flag indicates that one or more other cogs are requesting this cog's attention via the **COGATN** instruction.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the condition is not met, execution continues with the next instruction.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

These instructions are useful for implementing inter-cog communication mechanisms where one cog needs to signal and get the attention of another cog for coordination or data exchange purposes.

JCT1 / JCT2 / JCT3 / JNCT1 / JNCT2 / JNCT3

Jump If Counter Event Set / Clear
Events and Timing - Jumps based on counter event flag state.

JCT1 *{#}S*

JCT2 *{#}S*

JCT3 *{#}S*

JNCT1 *{#}S*

JNCT2 *{#}S*

JNCT3 *{#}S*

Result: JCTn jumps if the CTn event flag is set; JNCTn jumps if the CTn event flag is clear.

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000000001	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000000010	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000000011	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010001	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010010	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010011	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

PC is written only when the condition is met (flag set for JCTn, flag clear for JNCTn).

Related: ADDCT1/2/3, POLLCT1/2/3, WAITCT1/2/3

Explanation:

JCT1, JCT2, and JCT3 check their respective counter event flags and conditionally jump to the address specified by S if the flag is set. JNCT1, JNCT2, and JNCT3 perform the opposite **TEST**, jumping if the flag is clear. Each CTn event flag is automatically set when the system counter reaches the CTn target value that was previously configured using the corresponding ADDCTn instruction.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the condition is not met, execution continues with the next instruction.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

The P2 provides three independent hardware counters for timing operations, allowing a cog to manage multiple simultaneous time-based events without software overhead. JCTn instructions are commonly used for timing loops that wait until a counter fires, while JNCTn instructions enable polling loops that continue until a counter event occurs.

JFBW / JNFBW

Jump If FIFO Block Wrap Set / Clear {#jnfbw}
Events and Timing - Jumps based on FIFO block wrap event flag state.

JFBW *{#}S*

JNFBW *{#}S*

Result: JFBW jumps if the FIFO block wrap event flag is set; JNFBW jumps if the flag is clear.

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001001	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011001	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

PC is written only when the condition is met.

Related: RFBYTE, WFBYTE, SETQ2

Explanation:

JFBW checks the FIFO interface block wrap event flag and jumps if set. **JNFBW** performs the opposite **TEST**, jumping if clear. This event flag is set when a FIFO read or write operation wraps around the configured block boundary.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the condition is not met, execution continues with the next instruction.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

These instructions are useful for implementing circular buffer operations and managing block-based data transfers through the FIFO interface.

JINT / JNINT

Jump If Interrupt Set / Clear {#jnint}

Events and Timing - Jumps based on INT event flag state.

JINT *{#}S*

JNINT *{#}S*

Result: JINT jumps if the INT event flag is set; JNINT jumps if the flag is clear.

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000000000	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010000	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

PC is written only when the condition is met.

Related: POLLINT, SETINT1/2/3

Explanation:

JINT checks the INT (interrupt) event flag and jumps if set. **JNINT** performs the opposite **TEST**, jumping if clear. The INT event flag indicates that a hardware interrupt condition is pending, as configured by one of the SETINT instructions.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the condition is not met, execution continues with the next instruction.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

These instructions provide a polling-based mechanism for handling hardware interrupts, allowing code to check for interrupt conditions at convenient points in the program flow.

JMP

Jump

Branching and Flow Control - Unconditionally jumps to a new address.

JMP *D* **{WC/WZ/WCZ}**

JMP *#A*

JMP *#\A*

Result: PC is set to the address specified by D or A.

- D is a register containing the absolute jump address, and optionally flag values in bits [31:30].
- A is a 20-bit absolute or PC-relative address. Use \ prefix to force absolute addressing when using #.
- WC, WZ, or WCZ are optional effects to set C flag to D[31] and/or Z flag to D[30].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000101100	D[31]	D[30]	PC	4 / 13-20 †
EEEE	1101100	RAA	AAAAAAAAA	AAAAAAAAA	—	—	PC	4 / 13-20 †

† **Timing varies by execution context:**

Context	Clocks
COG / LUT execution	4
Hub execution	13…20

Related: CALL, RET, JMPREL, CALLD

Explanation:

JMP performs an unconditional jump to a new address, setting the PC to either the value in register D or the immediate address A.

The first syntax form (JMP D) reads the jump address from register D and sets PC to that value. When the WC or WCZ effect is specified, the C flag is set to bit 31 of D. When the WZ or WCZ effect is specified, the Z flag is set to bit 30 of D. This allows flags to be restored as part of a jump, which is useful for return-from-subroutine operations where both PC and flags need to be restored.

The second syntax form (JMP #A) jumps to an immediate address. The R bit in the encoding determines whether the address is PC-relative (R=1) or absolute (R=0). By default, the assembler uses PC-relative addressing for # jumps. The backslash prefix () forces absolute addressing: **JMP** #\address.

For PC-relative jumps in COG execution mode, the 20-bit address field is added to PC. For Hub execution mode, the lower 18 bits are shifted left by 2 (multiplied by 4) before being added to PC, since Hub addresses are long-aligned.

The instruction executes in 4 clock cycles in COG execution mode. In Hub execution mode, jumps take 13-20 clock cycles depending on Hub access timing.

JMPREL

Jump Relative

Branching and Flow Control - Jumps by adding a signed offset to the PC.

JMPREL *{#}D*

Result: PC is incremented or decremented by the value in D.

- D is a register or 9-bit literal specifying the signed offset in instructions. For COG execution, PC += D[19:0]. For Hub execution, PC += D[17:0] « 2.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110000	—	—	PC	4 / 13-20 †

† **Timing varies by execution context:**

Context	Clocks
COG / LUT execution	4
Hub execution	13…20

Related: JMP, CALL, DJNZ, IJMP1/2/3

Explanation:

JMPREL performs a relative jump by adding or subtracting the value in D to the current PC value. This allows position-independent code that can jump forward or backward by a specified number of instructions without knowing the absolute address.

For COG execution mode, the lower 20 bits of D are added to PC. Positive values jump forward, negative values (in two's complement) jump backward. The offset is in units of instructions (longs).

For Hub execution mode, the lower 18 bits of D are shifted left by 2 bits (multiplied by 4) before being added to PC. This accounts for the fact that Hub addresses are **BYTE** addresses and each instruction occupies 4 bytes. The offset is still conceptually in units of instructions.

The instruction executes in 4 clock cycles in COG execution mode. In Hub execution mode, jumps take 13-20 clock cycles depending on Hub access timing.

JMPREL is useful for implementing position-independent code, jump tables, and dynamic control flow where the jump offset is computed at runtime.

JSE1 / JSE2 / JSE3 / JSE4 / JNSE1 / JNSE2 / JNSE3 / JNSE4

Jump If Selectable Event Set / Clear

Events and Timing - Jumps based on selectable event flag state.

JSE1 *{#}S*

JSE2 *{#}S*

JSE3 *{#}S*

JSE4 *{#}S*

JNSE1 *{#}S*

JNSE2 *{#}S*

JNSE3 *{#}S*

JNSE4 *{#}S*

Result: JSEn jumps if the SEn event flag is set; JNSEn jumps if the SEn event flag is clear.

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000000100	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000000101	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000000110	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000000111	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010100	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010101	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010110	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000010111	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

PC is written only when the condition is met (flag set for JSEn, flag clear for JNSEn).

Related: SETSE1/2/3/4, POLLSE1/2/3/4, WAITSE1/2/3/4

Explanation:

JSE1, JSE2, JSE3, and JSE4 check their respective selectable event flags and conditionally jump to the address specified by S if the flag is set. JNSE1, JNSE2, JNSE3, and JNSE4 perform the opposite **TEST**, jumping if the flag is clear. Each selectable event can be configured to detect various hardware conditions using the corresponding SETSE instruction.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the condition is not met, execution continues with the next instruction.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

The P2 provides four independent selectable event sources, enabling multiple concurrent hardware event detection mechanisms for sophisticated event-driven applications. JSEn instructions are commonly used for event-triggered actions, while JNSEn instructions enable polling loops that continue until an event occurs.

JPAT / JNPAT

Jump If Pattern Match Event Set / Clear {#jnpat}
Events and Timing - Jumps based on PAT event flag state.

JPAT *{#}S*

JNPAT *{#}S*

Result: PC is set to the address specified by S if the PAT event flag is set (JPAT) or clear (JNPAT).

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001000	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011000	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

Related: SETPAT, POLLPAT

Explanation:

JPAT and **JNPAT** check the PAT (pattern match) event flag and conditionally jump to the address specified by S. **JPAT** jumps if the flag is set; **JNPAT** jumps if it is clear. The PAT event flag is set when the I/O pins match a pattern previously configured with the **SETPAT** instruction.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the flag is in the opposite state, execution continues with the next instruction and the jump is not taken.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

JPAT is useful for implementing hardware-triggered control flow where code execution branches based on specific pin state patterns. **JNPAT** is useful for polling loops that wait until a specific pattern appears on the I/O pins.

JQMT / JNQMT

Jump If CORDIC Empty Event Set / Clear {#jnqmt}

Events and Timing - Jumps based on CORDIC-read-but-empty event flag state.

JQMT *{#}S*

JNQMT *{#}S*

Result: PC is set to the address specified by S if the CORDIC-read-but-empty event flag is set (JQMT) or clear (JNQMT).

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001111	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011111	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

Related: QMUL, QROTATE, GETQX, GETQY

Explanation:

JQMT and **JNQMT** check the CORDIC-read-but-empty event flag and conditionally jump to the address specified by S. **JQMT** jumps if the flag is set; **JNQMT** jumps if it is clear. This event flag is set when code attempts to read CORDIC results before the calculation has completed, indicating a timing error.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the flag is in the opposite state, execution continues with the next instruction and the jump is not taken.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

JQMT is useful for error handling in CORDIC operations, allowing code to detect and respond to premature reads of calculation results. **JNQMT** is useful for ensuring CORDIC results are read at the correct time, helping to detect and handle timing errors in mathematical operations.

JXFI / JNXFI

Jump If Streamer Finished Event Set / Clear {#jnxfi}

Events and Timing - Jumps based on XFI event flag state.

JXFI *{#}S*

JNXFI *{#}S*

Result: PC is set to the address specified by S if the XFI event flag is set (JXFI) or clear (JNXFI).

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001011	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011011	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

Related: XINIT, XCONT, POLLXFI

Explanation:

JXFI and **JNXFI** check the XFI (streamer finished) event flag and conditionally jump to the address specified by S. **JXFI** jumps if the flag is set; **JNXFI** jumps if it is clear. The XFI event flag is set when the streamer completes its current operation.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the flag is in the opposite state, execution continues with the next instruction and the jump is not taken.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

JXFI is useful for chaining streamer operations or triggering code execution immediately when a streaming operation completes. **JNXFI** is useful for polling loops that wait until the streamer completes its operation.

JXMT / JNXMT

Jump If Streamer Empty Event Set / Clear {#jnxmt}
Events and Timing - Jumps based on XMT event flag state.

JXMT *{#}S*

JNXMT *{#}S*

Result: PC is set to the address specified by S if the XMT event flag is set (JXMT) or clear (JNXMT).

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001010	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011010	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

Related: XINIT, XCONT, POLLXMT

Explanation:

JXMT and **JNXMT** check the XMT (streamer empty) event flag and conditionally jump to the address specified by S. **JXMT** jumps if the flag is set; **JNXMT** jumps if it is clear. The XMT event flag is set when the streamer's internal buffer becomes empty and needs to be refilled.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the flag is in the opposite state, execution continues with the next instruction and the jump is not taken.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

JXMT is useful for implementing continuous streaming operations where the code needs to reload data into the streamer when the buffer empties. **JNXMT** is useful for maintaining continuous streamer operation by reloading data only when the streamer buffer still contains data.

JXRL / JNXRL

Jump If Streamer LUT Rollover Event Set / Clear {#jnxrl}
Events and Timing - Jumps based on XRL event flag state.

JXRL *{#}S*

JNXRL *{#}S*

Result: PC is set to the address specified by S if the XRL event flag is set (JXRL) or clear (JNXRL).

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001101	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011101	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

Related: XINIT, XCONT, POLLXRL

Explanation:

JXRL and **JNXRL** check the XRL (streamer LUT RAM rollover) event flag and conditionally jump to the address specified by S. **JXRL** jumps if the flag is set; **JNXRL** jumps if it is clear. The XRL event flag is set when the streamer's LUT RAM address pointer rolls over from the end back to the beginning of the configured range.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the flag is in the opposite state, execution continues with the next instruction and the jump is not taken.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

JXRL is useful for implementing circular buffer operations with the streamer using LUT RAM, detecting when a complete cycle through the buffer has occurred. **JNXRL** is useful for detecting when a buffer boundary has not yet been crossed.

JXRO / JNXRO

Jump If Streamer NCO Rollover Event Set / Clear {#jnxro}
Events and Timing - Jumps based on XRO event flag state.

JXRO *{#}S*

JNXRO *{#}S*

Result: PC is set to the address specified by S if the XRO event flag is set (JXRO) or clear (JNXRO).

- S is a register, 9-bit literal, or 20-bit augmented literal specifying the absolute or relative address to jump to. Use # for relative addressing; omit # for absolute addressing.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	01I	000001100	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20
EEEE	1011110	01I	000011100	SSSSSSSSS	—	—	PC	2 or 4 / 2 or 13-20

Related: XINIT, XCONT, POLLXRO

Explanation:

JXRO and **JNXRO** check the XRO (streamer NCO rollover) event flag and conditionally jump to the address specified by S. **JXRO** jumps if the flag is set; **JNXRO** jumps if it is clear. The XRO event flag is set when the streamer's numerically controlled oscillator (NCO) rolls over, which occurs at regular intervals determined by the NCO frequency setting.

When the # prefix is used with S, the jump is relative to the current PC value. When # is omitted, the jump is to the absolute address specified by S. If the flag is in the opposite state, execution continues with the next instruction and the jump is not taken.

The instruction executes in 2 clock cycles if the jump is not taken, or 4 clock cycles if the jump is taken (in COG execution mode). In Hub execution mode, taken jumps require 13-20 clock cycles depending on Hub timing.

JXRO is useful for timing-critical streamer applications where code needs to synchronize with the NCO rollovers. **JNXRO** is useful for detecting the absence of NCO rollovers in the streaming operation.

Instructions: L

This section contains all PASM2 instructions beginning with the letter L.

LOC

Load Address
Branching and Flow Control - Loads an address into a pointer register (PA, PB, PTRA, or PTRB).

LOC *PA/PB/PTRA/PTRB, #A*

LOC *PA/PB/PTRA/PTRB, #\A*

Result: Address is loaded into the specified pointer register.

- PA, PB, PTRA, or PTRB is the destination pointer register.
- A is a 20-bit address value.
- The optional backslash () prefix forces absolute addressing (R=0). Without it, relative addressing is used (R=1).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	11101WW	RAA	AAAAAAAAA	AAAAAAAAA	—	—	—	2

Related: PA, PB, PTRA, PTRB, CALLD, CALLPA, CALLPB

Explanation:

LOC loads an address into one of the four pointer registers: PA, PB, PTRA, or PTRB. These pointer registers are used by various memory operations and **CALL** instructions.

The instruction supports two addressing modes, controlled by the R bit in the encoding. By default, **LOC** uses relative addressing (R=1), where the address is calculated as PC + A. This allows position-independent code, as the address is computed relative to the current program counter. To force absolute addressing (R=0), prefix the address with a backslash (), making the address equal to A directly.

The WW field in the encoding selects which pointer register to load: 00 for PA, 01 for PB, 10 for PTRA, and 11 for PTRB. The address field A is 20 bits wide, providing access to the full Hub memory space.

LOC is commonly used to set up pointer registers before memory operations, **CALL** sequences, or when establishing base addresses for data structures. The relative addressing mode is particularly useful for creating position-independent code blocks that can execute correctly regardless of where they are loaded in Hub memory.

LOCKNEW

Allocate New Lock
COG Control and Locks - Requests an available lock from the hardware pool.

LOCKNEW *D* **{WC}**

Result: D is written with an available lock number (0-15), or remains unchanged if no lock is available.

- D is a register where the allocated lock number is written.

- WC is an optional effect to update the C flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	C00	DDDDDDDDD	000000100	No LOCK available	—	D	4…11

Related: LOCKTRY, LOCKREL, LOCKRET

Explanation:

LOCKNEW requests a lock from the P2's hardware lock pool. The P2 provides 16 hardware locks (numbered 0-15) for inter-COG synchronization and resource protection. **LOCKNEW** searches the lock pool for an available lock and, if one is found, returns its number in the D register.

If the WC effect is specified, the C flag is set (1) if no lock is available, or cleared (0) if a lock was successfully allocated. This allows the calling code to detect allocation failure and take appropriate action.

Once a lock is allocated with **LOCKNEW**, it remains assigned until explicitly returned to the pool with **LOCKRET**. The allocated lock can then be used with **LOCKTRY** to acquire exclusive access and **LOCKREL** to release it. This allocation-try-release-return pattern ensures proper resource management in multi-COG systems.

LOCKNEW is essential for dynamic lock allocation in systems where the number of required locks is not known at compile time, or where locks are allocated and deallocated as resources are created and destroyed. The instruction completes in 4 to 11 clock cycles depending on lock availability and contention.

LOCKREL

Release Lock
COG Control and Locks - Releases a lock for other COGs to acquire.

LOCKREL *{#}D* **{WC}**

Result: The lock specified by D[3:0] is released for other COGs to acquire.

- D is a register or 4-bit literal (0-15) specifying the lock number to release.
- When D is a register and WC is specified, D is written with the previous owner's COG ID and the C flag indicates lock status.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	C0L	DDDDDDDDD	000000111	LOCK status	—	—	2…9, +2 if result

Related: LOCKTRY, LOCKNEW, LOCKRET, COGID

Explanation:

LOCKREL releases a lock that was previously acquired with **LOCKTRY**, making it available for other COGs to acquire. The lock to release is specified by the lower 4 bits of D (D[3:0]), allowing lock numbers 0 through 15.

When D is a register (not an immediate) and the WC effect is specified, **LOCKREL** performs an additional operation: it writes the COG ID of the previous lock owner into D and sets the C flag based on whether the

lock was held. This diagnostic feature allows verification of lock ownership and debugging of synchronization issues.

LOCKREL is safe to call even if the lock was not held by the current COG. Releasing an unheld lock simply has no effect. This property simplifies error recovery code, as locks can be released without checking ownership first.

Proper lock management requires that every **LOCKTRY** that successfully acquires a lock is balanced with a corresponding **LOCKREL**. Failure to release locks leads to deadlocks and resource starvation. The instruction completes in 2 to 9 clock cycles, with an additional 2 cycles if the result is written back to D.

LOCKRET

Return Lock To Pool

COG Control and Locks - Returns a lock to the pool for reallocation by **LOCKNEW**.

LOCKRET *{#}D*

Result: The lock specified by D[3:0] is returned to the pool and becomes available for LOCKNEW.

- D is a register or 4-bit literal (0-15) specifying the lock number to return.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000000101	—	—	—	2...9

Related: LOCKNEW, LOCKTRY, LOCKREL

Explanation:

LOCKRET returns a lock to the hardware lock pool, making it available for future allocation by **LOCKNEW**. This instruction completes the lifecycle of a dynamically allocated lock: first allocated with **LOCKNEW**, then used with **LOCKTRY** and **LOCKREL** for synchronization, and finally returned with **LOCKRET** when no longer needed.

The lock to return is specified by the lower 4 bits of D (D[3:0]), allowing lock numbers 0 through 15. Unlike **LOCKREL**, which only releases ownership of a lock while keeping it allocated, **LOCKRET** deallocates the lock entirely, allowing **LOCKNEW** to assign it to a different purpose.

LOCKRET should only be called on locks that are not currently held by any COG. Before returning a lock, ensure it has been released with **LOCKREL**. Returning a lock that is still held can cause synchronization failures in other COGs that may be waiting for or using that lock.

The proper pattern for dynamic lock usage is: **LOCKNEW** to allocate, **LOCKTRY**/LOCKREL for each critical section, and **LOCKRET** when the lock is no longer needed for any purpose. This ensures efficient use of the limited pool of 16 hardware locks. The instruction completes in 2 to 9 clock cycles depending on Hub access contention.

LOCKTRY

Try To Acquire Lock

COG Control and Locks - Attempts to acquire a lock using atomic test-and-set.

LOCKTRY *{#}D* **{WC}**

Result: Attempts to acquire the lock specified by D[3:0]. The C flag indicates success.

- D is a register or 4-bit literal (0-15) specifying the lock number to acquire.
- WC is an optional effect to update the C flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	C0L	DDDDDDDDD	000000110	1 if got LOCK	—	—	2…9, +2 if result

Related: LOCKREL, LOCKNEW, LOCKRET, COGID

Explanation:

LOCKTRY attempts to acquire a lock using an atomic test-and-set operation. The lock to acquire is specified by the lower 4 bits of D (D[3:0]), allowing lock numbers 0 through 15. The P2 provides 16 hardware locks for inter-COG synchronization and resource protection.

If the WC effect is specified, the C flag is set (1) if the lock was successfully acquired, or cleared (0) if the lock is already held by another COG. This non-blocking behavior allows the calling code to make immediate decisions: proceed with the protected operation if the lock was acquired, or take alternative action if it was not.

LOCKTRY implements the critical section entry point in the standard lock pattern: try to acquire the lock, and only proceed if successful. The lock must be released with **LOCKREL** when the critical section completes. This ensures mutual exclusion, preventing multiple COGs from simultaneously accessing shared resources.

The instruction is non-blocking and returns immediately regardless of lock availability. For spin-lock behavior (waiting until the lock is acquired), **LOCKTRY** must be called repeatedly in a loop. Lock 15 is traditionally reserved for **DEBUG** monitor use. The instruction completes in 2 to 9 clock cycles, with an additional 2 cycles if a result is returned.

Instructions: M

This section contains all PASM2 instructions beginning with the letter M.

MERGEB

Merge Bits Of Bytes
Arithmetic Operations - Rearranges bits by extracting one bit from each byte and merging them.

MERGEB *D*

Result: Bits from each byte in D are rearranged into a specific merged pattern.

- D is a register containing the value whose **BYTE** bits will be merged.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100001	—	—	D	2

Related: MERGEW, SPLITB, SPLITW

Explanation:

MERGEB rearranges the bits within D by extracting one bit from each byte and merging them into a specific pattern. The result is: D = {D[31], D[23], D[15], D[7], D[30], D[22], D[14], D[6], …, D[24], D[16], D[8], D[0]}.

This operation takes the most significant bit from each of the four bytes in D and places them in the upper nibble of the result, then the next most significant bit from each byte into the next nibble, and so on. Each group of four bits in the result contains one bit from each of the four original bytes.

MERGEB is useful for bit-plane conversions, graphics operations, and data transformations where bits need to be regrouped across **BYTE** boundaries. It performs the inverse operation of **SPLITB**, which distributes bits back into their original **BYTE** positions.

MERGEW

Merge Bits Of Words
Arithmetic Operations - Rearranges bits by interleaving from the two 16-bit words.

MERGEW *D*

Result: Bits from each word in D are rearranged into a specific merged pattern.

- D is a register containing the value whose **WORD** bits will be merged.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100011	—	—	D	2

Related: MERGEB, SPLITB, SPLITW

Explanation:

MERGEW rearranges the bits within D by extracting corresponding bits from each of the two 16-bit words and interleaving them. The result is: D = {D[31], D[15], D[30], D[14], D[29], D[13], …, D[17], D[1], D[16], D[0]}.

This operation interleaves the bits from the upper and lower words of D, alternating between taking a bit from the upper **WORD** and a bit from the lower **WORD**. The most significant bit of the result comes from the most significant bit of the upper **WORD**, the next bit from the most significant bit of the lower **WORD**, and so on.

MERGEW is useful for word-level bit-plane conversions, graphics operations requiring word-aligned data transformations, and encoding operations. It performs the inverse operation of **SPLITW**, which de-interleaves the bits back into their original **WORD** positions.

MIXPIX

Mix Pixels

Color Space and Pixel Operations - Blends pixel bytes according to **SETPIX** and **SETPIV** configuration.

MIXPIX *D,{#}S*

Result: Bytes of S are blended into bytes of D according to the SETPIX and SETPIV configuration.

- D is a register containing the destination pixel bytes to be modified.
- S is a register, 9-bit literal, or 32-bit augmented literal containing the source pixel bytes.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010010	11I	DDDDDDDDD	SSSSSSSSS	—	—	D	7

Related: SETPIX, SETPIV, ADDPIX, MULPIX, BLNPIX

Explanation:

MIXPIX performs pixel blending operations on the four bytes of D using the four bytes of S, according to the mixing parameters previously configured by **SETPIX** and **SETPIV** instructions. Each byte is treated as a separate pixel component (typically used for red, green, blue, and alpha channels in RGBA color format).

The **SETPIX** instruction configures the pixel mixer mode, which determines how the source and destination bytes are combined (such as multiply, add, or blend operations). The **SETPIV** instruction provides additional configuration values that affect the mixing calculation.

This instruction executes in 7 clock cycles to perform the pixel arithmetic on all four bytes in parallel. The exact blending formula depends on the mode set by **SETPIX**, but typically implements standard pixel compositing operations used in graphics rendering, such as alpha blending, color multiplication, or additive blending.

MIXPIX is essential for high-performance graphics operations, enabling real-time color mixing, transparency effects, and color space transformations without requiring multiple individual **BYTE** operations.

MODC

Modify C Flag

Arithmetic Operations - Sets or clears C flag based on a modifier and current flag states.

MODC *c* **{WC}**

Result: The C flag is set or cleared according to the modifier and current C and Z flag states.

- c is a 4-bit modifier constant (such as `_set`, `_clr`, `_c`, `_z`) that selects which combination of current C and Z flag states produces a 1 result for the C flag.
- WC must be specified for the C flag modification to take effect; without it, the result is computed but not written to the flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	C01	0cccc0000	001101111	cccc[C,Z]	—	—	2

Related: MODZ, MODCZ, TESTB, TESTBN

Explanation:

MODC provides conditional modification of the C flag based on a 4-bit modifier value and the current state of both the C and Z flags. The modifier value acts as a lookup table, where each of the four bits corresponds to one of the four possible combinations of the current C and Z flag states: 00, 01, 10, and 11.

The modifier is applied as: C = cccc[{C,Z}], where {C,Z} forms a 2-bit index into the 4-bit modifier value. For example, if the current C flag is 1 and Z flag is 0, the index is binary 10 (2 decimal), and the C flag is set to bit 2 of the modifier value.

Common modifier values enable useful operations: $F (binary 1111) always sets C to 1, $0 (binary 0000) always clears C to 0, $C (binary 1100) copies C to itself (C unchanged, independent of Z), and $3 (binary 0011) sets C to the inverse of the current C (NC), independent of Z.

MODC is typically used after comparison or **TEST** instructions to create complex conditional logic without branching. It provides a mechanism to compute a boolean result based on multiple flag conditions in a single instruction.

The WC effect must be specified for the modification to take effect. Without WC, the instruction computes the result but does not write it to the C flag, rendering the instruction ineffective for most purposes.

MODCZ

Modify C And Z Flags
Arithmetic Operations - Sets or clears both C and Z flags based on modifiers.

MODCZ *c,z* **{WC/WZ/WCZ}**

Result: Both C and Z flags are set or cleared according to their modifiers and the current C and Z flag states.

- c is a 4-bit modifier constant (such as `_set`, `_clr`, `_c`, `_z`) that selects which combination of current C and Z flag states produces a 1 result for the C flag.
- z is a 4-bit modifier constant that selects which combination of current C and Z flag states produces a 1 result for the Z flag.
- WC, WZ, or WCZ must be specified for the flag modifications to take effect; without them, results are computed but not written.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ1	0cccczzzz	001101111	cccc[C,Z]	zzzz[C,Z]	—	2

Related: MODC, MODZ, TESTB, TESTBN

Explanation:

MODCZ provides simultaneous conditional modification of both the C and Z flags based on 4-bit modifier values and the current state of both flags. Each modifier value acts as a lookup table, where each of the four bits corresponds to one of the four possible combinations of the current C and Z flag states: 00, 01, 10, and 11.

The modifiers are applied as: C = cccc[{C,Z}] and Z = zzzz[{C,Z}], where {C,Z} forms a 2-bit index into each 4-bit modifier value. Both flags are updated simultaneously based on the same initial C and Z states, allowing complex boolean operations to be computed in parallel.

This instruction enables sophisticated conditional logic operations without branching. For example, modifier values can implement logical operations like AND, OR, **XOR** between the flags, or conditional moves where one flag's new value depends on the other flag's current state.

Common uses include implementing state machines where both flags represent state bits, performing multi-condition tests after comparison operations, and creating compact conditional code sequences that would otherwise require multiple instructions or branches.

The WC, WZ, or WCZ effect must be specified for the modifications to take effect. Without these effects, the instruction computes results but does not write them to the flags, rendering the instruction ineffective for most purposes.

The simultaneous update of both flags makes **MODCZ** more powerful than using separate **MODC** and **MODZ** instructions, as it allows each flag's new value to be based on the same initial flag state rather than having one flag update affect the other's calculation.

Modifier Constants:

Value	Binary	Mnemonic	Description
0	0000	_CLR	Always clear (result = 0)
1	0001	_NC_AND_NZ	C=0 AND Z=0
2	0010	_NC_AND_Z	C=0 AND Z=1
3	0011	_NC	Copy inverse of C (not C)
4	0100	_C_AND_NZ	C=1 AND Z=0
5	0101	_NZ	Copy inverse of Z (not Z)
6	0110	_C_NE_Z	C XOR Z (C not equal to Z)
7	0111	_NC_OR_NZ	C=0 OR Z=0 (NAND)
8	1000	_C_AND_Z	C=1 AND Z=1 (AND)
9	1001	_C_EQ_Z	NOT(C XOR Z) (C equals Z)
10	1010	_Z	Copy Z
11	1011	_NC_OR_Z	C=0 OR Z=1
12	1100	_C	Copy C
13	1101	_C_OR_NZ	C=1 OR Z=0
14	1110	_C_OR_Z	C=1 OR Z=1 (OR)
15	1111	_SET	Always set (result = 1)

```
1         MODCZ   _CLR, _SET      ' Clear C, set Z
2         MODCZ   _SET, _CLR      ' Set C, clear Z
3         MODCZ   _C, _Z          ' C and Z unchanged (copy to themselves)
4         MODCZ   _Z, _C          ' Swap C and Z values
5         MODCZ   _NC, _NZ        ' Invert both flags
```

MODZ

Modify Z Flag

Arithmetic Operations - Sets or clears Z flag based on a modifier and current flag states.

MODZ *z* {WZ}

Result: The Z flag is set or cleared according to the modifier and current C and Z flag states.

- z is a 4-bit modifier constant (such as `_set`, `_clr`, `_c`, `_z`) that selects which combination of current C and Z flag states produces a 1 result for the Z flag.
- WZ must be specified for the Z flag modification to take effect; without it, the result is computed but not written to the flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	0Z1	00000zzzz	001101111	—	zzzz[C,Z]	—	2

Related: MODC, MODCZ, TESTB, TESTBN

Explanation:

MODZ provides conditional modification of the Z flag based on a 4-bit modifier value and the current state of both the C and Z flags. The modifier value acts as a lookup table, where each of the four bits corresponds to one of the four possible combinations of the current C and Z flag states: 00, 01, 10, and 11.

The modifier is applied as: Z = zzzz[{C,Z}], where {C,Z} forms a 2-bit index into the 4-bit modifier value. For example, if the current C flag is 0 and Z flag is 1, the index is binary 01 (1 decimal), and the Z flag is set to bit 1 of the modifier value.

Common modifier values enable useful operations: $F (binary 1111) always sets Z to 1, $0 (binary 0000) always clears Z to 0, $A (binary 1010) copies Z to itself (preserving current state), and $C (binary 1100) sets Z if C=1.

MODZ is typically used after comparison or **TEST** instructions to create complex conditional logic without branching. It provides a mechanism to compute a boolean result based on multiple flag conditions in a single instruction.

The WZ effect must be specified for the modification to take effect. Without WZ, the instruction computes the result but does not write it to the Z flag, rendering the instruction ineffective for most purposes.

MOV

Move

Arithmetic Operations - Copies a value from source to destination register.

MOV *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The Src value is stored in Dest.

- Dest is a register where the Src value will be written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is copied to Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0110000	CZI	DDDDDDDDD	SSSSSSSSS	S[31]	result == 0	D	2

Related: MOVBYTS, MUXNIBS, MUXNITS, SETQ

Explanation:

MOV copies the value from Src into the Dest register, providing the fundamental data movement operation in PASM2. This is one of the most frequently used instructions, enabling register initialization, value copying, and data transfer between registers.

If the WC or WCZ effect is specified, the C flag is set to the most significant bit of the source value (Src[31]), which represents the sign bit when Src is interpreted as a signed 32-bit value. This allows **MOV** to simultaneously copy a value and **TEST** its sign.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result written to Dest equals zero, or is cleared (0) if the result is non-zero. This enables immediate testing of whether the moved value is zero without requiring a separate comparison instruction.

MOV with immediate values is commonly used for register initialization:

```
1         MOV     counter, #100           ' Initialize counter to 100
2         MOV     mask, ##$FFFF_0000      ' Load 32-bit constant using AUGS
```

MOV between registers is used for preserving values and working with temporary copies:

```
1         MOV     temp, value             ' Save value in temp
2         ADD     value, increment        ' Modify value
3         MOV     result, value           ' Copy final result
```

When combined with flag effects, **MOV** enables efficient value testing:

```
1                 MOV     data, source  wz        ' Copy and test if zero
2         if_nz   CALL    #process                ' Process only if non-zero
3                 MOV     signed, value  wc       ' Copy and test sign bit
4         if_c    NEG     signed, signed          ' Make positive if negative
```

MOVBYTS

Move Bytes

Arithmetic Operations - Rearranges bytes within a register according to a selection pattern.

MOVBYTS *D,{#}S*

Result: Bytes within D are rearranged according to the byte selection pattern in S.

- D is a register containing the bytes to be rearranged.
- S is a register, 9-bit literal, or 32-bit augmented literal containing the byte selection pattern.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001111	11I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: MERGEB, SPLITB, ROLBYTE

Explanation:

MOVBYTS rearranges the four bytes within D according to a selection pattern specified in the lower 8 bits of S. The result is: D = {D.BYTE[S[7:6]], D.BYTE[S[5:4]], D.BYTE[S[3:2]], D.BYTE[S[1:0]]}.

Each 2-bit field in S selects which of the four original bytes in D will appear in each position of the result. S[1:0] selects the byte for the least significant position, S[3:2] for the second **BYTE**, S[5:4] for the third **BYTE**, and S[7:6] for the most significant **BYTE**. The 2-bit values 0, 1, 2, and 3 select bytes 0 (bits 7:0), 1 (bits 15:8), 2 (bits 23:16), and 3 (bits 31:24) respectively.

For example, to swap the high and low words of D, use S = $4E (binary 01_00_11_10), which places **BYTE** 2 in position 0, **BYTE** 3 in position 1, **BYTE** 0 in position 2, and **BYTE** 1 in position 3. To reverse all four bytes, use S = $1B (binary 00_01_10_11).

MOVBYTS is useful for byte-order conversions (endianness swapping), color channel reordering in pixel data, and general **BYTE** permutation operations. It executes in 2 clock cycles, making it an efficient alternative to multiple shift and mask operations.

Common patterns include:

- S = $E4 (binary 11_10_01_00): No change (identity)
- S = $1B (binary 00_01_10_11): Reverse bytes (big/little endian swap)
- S = $B1 (binary 10_11_00_01): Swap bytes within each word
- S = $4E (binary 01_00_11_10): Swap words

MUL

Multiply
Arithmetic Operations - Multiplies two 16-bit unsigned values, producing 32-bit result.

MUL *Dest, {#}Src* **{WZ}**

Result: The 32-bit unsigned product of the lower 16 bits of Dest and Src is stored in Dest.

- Dest is a register containing the 16-bit value to multiply with Src, and is where the 32-bit result is written.
- Src is a register, 9-bit literal, or 16-bit augmented literal whose lower 16 bits are multiplied with Dest.
- WZ is an optional effect to update the Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010000	0ZI	DDDDDDDDD	SSSSSSSSS	—	(S == 0) OR (D == 0)	D	2

Related: MULS, QMUL, SCA, SCAS

Explanation:

MUL performs an unsigned 16-bit by 16-bit multiplication, taking only the lower 16 bits from each of Dest and Src, multiplying them together, and storing the full 32-bit unsigned product into Dest. This is a fast 2-clock multiplication operation suitable for small integer arithmetic and fixed-point calculations.

The operation is: D = unsigned(D[15:0] * S[15:0]). The upper 16 bits of both Dest and Src are ignored during the multiplication, but the full 32-bit result can utilize all bits in the destination register. For example, multiplying $0001_8000 by $0002_4000 produces $2000_0000 (using only the $8000 and $4000 values).

If the WZ effect is specified, the Z flag is set (1) if either Dest or Src equals zero before the multiplication, or is cleared (0) if both are non-zero. Note that this tests the pre-multiplication values, not the result, providing a quick way to detect zero operands.

MUL is commonly used for scaling operations in fixed-point arithmetic:

```
1         MOV     value, ##1000           ' Value = 1000
2         MUL     value, #25              ' Multiply by 25: value = 25000
```

For fixed-point math with 16-bit fractional parts:

```
        ' Multiply two 16.16 fixed-point numbers
        ' Result in upper 16 bits needs shifting
        MOV     temp, frac1
        MUL     temp, frac2             ' temp = product (low 16 of each)
        SHR     temp, #16               ' Adjust for fixed-point scale
```

For multiplications larger than 16x16 bits, use the CORDIC solver **QMUL** instruction, which can multiply full 32-bit values and produces a 64-bit result accessible through the upper and lower result registers. **MUL**'s 2-clock speed makes it ideal when the operands are known to fit in 16 bits.

MULPIX

Multiply Pixels

Color Space and Pixel Operations - Multiplies corresponding pixel bytes in parallel.

MULPIX *D,{#}S*

Result: Each byte of S is multiplied with the corresponding byte of D, with results stored in D.

- D is a register containing four pixel bytes to be multiplied.
- S is a register, 9-bit literal, or 32-bit augmented literal containing four pixel bytes as multipliers.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010010	01I	DDDDDDDDD	SSSSSSSSS	—	—	D	7

Related: ADDPIX, BLNPIX, MIXPIX, SETPIX

Explanation:

MULPIX performs parallel multiplication on four **BYTE** pairs, treating each byte as a fractional value where $FF represents 1.0 and $00 represents 0.0. Each of the four bytes in S is multiplied with the corresponding **BYTE** in D, and the results replace the bytes in D.

The multiplication treats bytes as 8-bit fractional values in the range 0.0 to 1.0. For each byte position, the operation computes: D.BYTEn = (D.BYTEn * S.BYTEn) / 255. The division by 255 is implicit in the fractional representation, where $FF * $FF = $FF (1.0 * 1.0 = 1.0).

This instruction is essential for pixel color multiplication operations used in graphics rendering. For example, multiplying an RGB color by a brightness value: if D contains $80_60_40_20 (RGBA values) and S contains $80_80_80_FF (50% brightness on RGB, full alpha), each color component is reduced to 50% of its original value.

MULPIX executes in 7 clock cycles to perform all four parallel multiplications. This is significantly faster than performing four separate multiply and scale operations, making it practical for real-time graphics processing.

Common uses include:

- Color modulation (tinting): Multiply each color channel by a tint value
- Brightness adjustment: Multiply RGB by a brightness factor

- Alpha premultiplication: Multiply RGB by alpha for compositing
- Texture filtering: Combine texel colors with interpolation weights

The instruction treats all bytes independently, so it can be used for any four-byte parallel multiply operation, not just color processing.

MULS

Multiply Signed
Arithmetic Operations - Multiplies two signed 16-bit values, producing signed 32-bit result.

MULS *Dest, {#}Src* **{WZ}**

Result: The 32-bit signed product of the signed lower 16 bits of Dest and Src is stored in Dest.

- Dest is a register containing the signed 16-bit value to multiply with Src, and is where the signed 32-bit result is written.
- Src is a register, 9-bit literal, or signed 16-bit augmented literal whose lower 16 bits are multiplied with Dest.
- WZ is an optional effect to update the Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010000	1ZI	DDDDDDDDD	SSSSSSSSS	—	(S == 0) OR (D == 0)	D	2

Related: MUL, QMUL, SCA, SCAS

Explanation:

MULS performs a signed 16-bit by 16-bit multiplication, taking only the lower 16 bits from each of Dest and Src as signed values, multiplying them together, and storing the full signed 32-bit product into Dest. This is a fast 2-clock multiplication operation suitable for signed integer arithmetic and signed fixed-point calculations.

The operation is: D = signed(D[15:0] * S[15:0]). The upper 16 bits of both Dest and Src are ignored during the multiplication. The lower 16 bits are treated as signed values (using two's complement representation), so values from $8000 (-32768) to $7FFF (+32767) are valid inputs. The 32-bit result is properly sign-extended to represent the full range of products.

For example, multiplying $FFFF_8000 (-32768 in lower 16 bits) by $0000_0002 (+2) produces $FFFF_0000 (-65536 as a signed 32-bit value). The upper 16 bits of the operands are ignored, and the result is correctly signed.

If the WZ effect is specified, the Z flag is set (1) if either Dest or Src equals zero before the multiplication, or is cleared (0) if both are non-zero. Note that this tests the pre-multiplication values, not the result, providing a quick way to detect zero operands.

MULS is commonly used for signed arithmetic and physics calculations:

```
        MOV     velocity, signed_speed
        MULS    velocity, time          ' velocity = speed * time (signed)
```

For signed fixed-point math with 16-bit fractional parts:

```
        ' Multiply two signed 16.16 fixed-point numbers
        MOV     temp, signed_frac1
        MULS    temp, signed_frac2      ' Signed multiplication
        SAR     temp, #16               ' Arithmetic shift to preserve sign
```

MULS differs from **MUL** only in that it treats the 16-bit operands as signed values rather than unsigned. The choice between them depends on whether the values being multiplied represent signed or unsigned quantities.

For multiplications larger than 16x16 bits, use the CORDIC solver **QMUL** instruction, which can multiply full signed 32-bit values and produces a signed 64-bit result accessible through the upper and lower result registers.

MUXC / MUXNC / MUXZ / MUXNZ

Multiplex Flag To Bits

Arithmetic Operations - Sets selected bits to a flag value based on mask.

MUXC *D,{#}S* **{WC|WZ|WCZ}**

MUXNC *D,{#}S* **{WC|WZ|WCZ}**

MUXZ *D,{#}S* **{WC|WZ|WCZ}**

MUXNZ *D,{#}S* **{WC|WZ|WCZ}**

Result: Each bit position in D where S has a 1 is set to the specified flag value. Optionally sets C to parity and Z if result is zero.

- D is a register whose bits will be set to the flag value where S has 1 bits.
- S is a register, 9-bit literal, or 32-bit augmented literal that selects which bits to modify.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0101100	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2
EEEE	0101101	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2
EEEE	0101110	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2
EEEE	0101111	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2

Related: MUXQ, TESTB, TESTBN

Explanation:

These instructions modify selected bits in D based on a flag value:

Instruction	Sets bits to
MUXC	C flag value
MUXNC	!C (inverted C)
MUXZ	Z flag value
MUXNZ	!Z (inverted Z)

For each bit position where S contains a 1, the corresponding bit in D is replaced with the flag value (or its inverse). All other bits in D remain unchanged. The operation is: D = (!S & D) | (S & {32{flag}}).

MUXC and **MUXZ** copy the direct flag value; **MUXNC** and **MUXNZ** copy the inverted flag value.

Example: Conditionally set bits based on a comparison:

```
1         CMP     value, limit  wc        ' Set C if value < limit
2         MUXC    status, #$01            ' Set bit 0 if less than
3         MUXNC   status, #$02            ' Set bit 1 if greater or equal
```

If the WC or WCZ effect is specified, the C flag is set to the parity of the result. If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero.

These instructions provide an efficient alternative to conditional branches when setting bits based on flag states.

MUXNIBS

Multiplex Nibbles
Arithmetic Operations - Replaces nibbles in Dest where Src nibbles are non-zero.

MUXNIBS *Dest, {#}Src*

Result: Each non-zero nibble in Src replaces the corresponding nibble in Dest.

- Dest is a register whose nibbles will be updated from Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal containing nibble values to copy.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001111	01I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: MUXNITS, MUXQ, MOVBYTS, SPLITB

Explanation:

MUXNIBS selectively copies nibbles (4-bit fields) from Src to Dest based on whether each nibble in Src is non-zero. For each of the eight nibble positions, if the nibble in Src is non-zero, that nibble value is copied to the corresponding position in Dest. If the nibble in Src is zero, the corresponding nibble in Dest remains unchanged.

For example, if Dest = $1234_5678 and Src = $0A00_0C0D, the result is Dest = $1A34_5C7D. The nibbles at positions 6 ($A), 2 ($C), and 0 ($D) from Src are copied because they are non-zero, while positions 7, 5, 4, 3, and 1 remain unchanged in Dest because the corresponding Src nibbles are zero.

This instruction is useful for sparse updates where only certain nibbles need modification:

```
' Update only the changed nibbles in a configuration register
MOV     config, current_config
MUXNIBS config, changes         ' Apply non-zero changes only
```

MUXNIBS is commonly used in graphics operations for palette updates, bit-field modifications where fields are naturally nibble-aligned, and efficient sparse data updates. It provides a single-instruction way to perform selective nibble replacement that would otherwise require multiple mask and merge operations.

The instruction treats nibbles independently, enabling parallel conditional updates across all eight nibble positions in a single 2-clock operation.

MUXNITS

Multiplex Nits
Arithmetic Operations - Replaces bit pairs in Dest where Src bit pairs are non-zero.

MUXNITS *Dest, {#}Src*

Result: Each non-zero bit pair in Src replaces the corresponding bit pair in Dest.

- Dest is a register whose bit pairs will be updated from Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal containing bit pair values to copy.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001111	00I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: MUXNIBS, MUXQ, MOVBYTS, SPLITB

Explanation:

MUXNITS selectively copies bit pairs (2-bit fields, called "nits") from Src to Dest based on whether each bit pair in Src is non-zero. For each of the sixteen bit pair positions, if the bit pair in Src is non-zero (01, 10, or 11), that bit pair value is copied to the corresponding position in Dest. If the bit pair in Src is zero (00), the corresponding bit pair in Dest remains unchanged.

For example, if Dest = $5555_5555 (binary 01_01_01_01… in bit pairs) and Src = $00A0_0002 (containing non-zero bit pairs at positions 11, 10, and 0), only those three bit pairs are updated in Dest while the others remain as 01.

This instruction is particularly useful for pixel graphics operations where 2-bit values represent pixel data (such as in 4-color graphics modes), sparse bit-field updates, and state machine implementations where state variables are represented as 2-bit fields.

MUXNITS provides parallel conditional updates across all sixteen bit pair positions in a single 2-clock operation:

```
        ' Update specific 2-bit fields in a packed structure
        MOV     state, current_state
        MUXNITS state, updates          ' Apply non-zero updates only
```

The name "nits" comes from "nibble bits" or 2-bit fields, representing the next smaller grouping after nibbles (4-bit fields). This instruction complements **MUXNIBS** by operating at a finer granularity.

MUXQ

Multiplex Q
Arithmetic Operations - Copies bits from Src to Dest at positions where Q has 1 bits.

MUXQ *Dest, {#}Src*

Result: Bits from Src are copied to Dest at positions where Q has 1 bits.

- Dest is a register whose bits will be updated from Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal containing bit values to copy.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001111	10I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: SETQ, MUXC, MUXZ, MUXNIBS, MUXNITS

Explanation:

MUXQ performs selective bit copying from Src to Dest based on a mask previously loaded into the Q register using **SETQ**. For each bit position where Q contains a 1, the corresponding bit from Src is copied into Dest. For bit positions where Q contains a 0, the corresponding bit in Dest remains unchanged. The operation is: D = (!Q & D) | (Q & S).

MUXQ must be preceded by **SETQ** to load the mask into Q:

```
        SETQ    mask                    ' Load mask into Q
        MUXQ    dest, source            ' Copy masked bits from source
```

This provides atomic masked bit updates that are more efficient than separate AND and OR operations:

```
        ' Traditional approach (4 instructions):
        MOV     temp, source            ' Copy source
        AND     temp, mask              ' Extract source bits
        ANDN    dest, mask              ' Clear masked bits in dest
        OR      dest, temp              ' Merge into dest

        ' MUXQ approach (2 instructions):
        SETQ    mask                    ' Set mask
        MUXQ    dest, source            ' Atomic masked copy
```

MUXQ is critical for parallel I/O operations, especially driving multiple pins simultaneously:

```
        ' Update multiple RGB LED pins atomically
        SETQ    rgb_mask                ' Mask for RGB pins
        MUXQ    outa, rgb_data          ' Update all RGB pins together
```

The Q register mask enables sophisticated bit manipulation:

```
        ' Update specific configuration bits
        SETQ    ##$00FF_FF00            ' Mask for middle bytes
        MUXQ    config, new_values      ' Update only those bytes
```

MUXQ is particularly valuable for HUB75 RGB panel driving and other applications requiring atomic multi-pin updates. It executes in 2 clock cycles, providing high-performance parallel bit operations essential for real-time graphics and control applications.

Unlike **MUXC** and **MUXZ** which replicate a single flag bit to all selected positions, **MUXQ** copies the actual corresponding bits from the source, enabling true parallel bit transfer operations.

Instructions: N

This section contains all PASM2 instructions beginning with the letter N.

NEG

Negate
Arithmetic Operations - Negates a value, flipping its sign.

NEG *Dest, {#}Src* **{WC|WZ|WCZ}**

NEG *Dest* **{WC|WZ|WCZ}**

Result: The Src or Dest value is negated and stored into Dest.

- Dest is a register to receive the -Src value (syntax 1), or contains the value to negate (syntax 2).
- Src is an optional register, 9-bit literal, or 32-bit augmented literal whose negated value is stored into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0110011	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2
EEEE	0110011	CZ0	DDDDDDDDD	DDDDDDDDD	MSB of result	result == 0	D	2

Related: ABS, NEGC, NEGNC, NEGZ, NEGNZ

Explanation:

NEG negates the value in Src (syntax 1) or Dest (syntax 2) and stores the result in the Dest register. The negation flips the value's sign; for example, 78 becomes -78, or -306 becomes 306.

When using syntax 1, **NEG** negates the Src operand and stores the result into Dest. When using syntax 2 (where Src is omitted), **NEG** negates the value already in Dest and stores the result back into Dest.

If the WC or WCZ effect is specified, the C flag is set (1) if the result is negative, or is cleared (0) if positive.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if it is non-zero.

NEGC / NEGNC / NEGZ / NEGNZ

Conditional Negate
Arithmetic Operations - Conditionally negates a value based on flag state.

NEGC *Dest, {#}Src* **{WC|WZ|WCZ}**

NEGC *Dest* **{WC|WZ|WCZ}**

NEGNC *Dest, {#}Src* **{WC|WZ|WCZ}**

NEGNC *Dest* **{WC|WZ|WCZ}**

NEGZ *Dest, {#}Src* **{WC|WZ|WCZ}**

NEGZ *Dest* **{WC|WZ|WCZ}**

NEGNZ *Dest, {#}Src* **{WC|WZ|WCZ}**

NEGNZ *Dest* **{WC|WZ|WCZ}**

Result: The Src or Dest value, conditionally negated based on flag state, is stored into Dest. Optionally sets C to sign and Z if result is zero.

- Dest is a register to receive the result.
- Src is an optional register, 9-bit literal, or 32-bit augmented literal.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0110100	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2
EEEE	0110100	CZ0	DDDDDDDDD	DDDDDDDDD	MSB of result	result == 0	D	2
EEEE	0110101	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2
EEEE	0110101	CZ0	DDDDDDDDD	DDDDDDDDD	MSB of result	result == 0	D	2
EEEE	0110110	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2
EEEE	0110110	CZ0	DDDDDDDDD	DDDDDDDDD	MSB of result	result == 0	D	2
EEEE	0110111	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2
EEEE	0110111	CZ0	DDDDDDDDD	DDDDDDDDD	MSB of result	result == 0	D	2

Related: NEG

Explanation:

These instructions conditionally negate the value in Src (two-operand form) or Dest (single-operand form) based on the specified flag condition:

Instruction	Negates when
NEGC	C = 1
NEGNC	C = 0
NEGZ	Z = 1
NEGNZ	Z = 0

If the condition is true, the value is negated (sign flipped) before being stored in Dest. If the condition is false, the value is stored unchanged.

NEGC and **NEGZ** negate when their flag is set (1). **NEGNC** and **NEGNZ** negate when their flag is clear (0), providing complementary behavior.

If the WC or WCZ effect is specified, the C flag is set (1) if the result is negative, or cleared (0) if positive.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result is zero, or cleared (0) if non-zero.

NIXINT1 / NIXINT2 / NIXINT3

Cancel Interrupt

Events and Timing - Cancels any pending interrupt event for the specified level.

NIXINT1
NIXINT2
NIXINT3

Result: The specified interrupt event (INT1, INT2, or INT3) is cancelled.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	000100101	000100100	—	—	—	2
EEEE	1101011	000	000100110	000100100	—	—	—	2
EEEE	1101011	000	000100111	000100100	—	—	—	2

Related: SETINT1/2/3, TRGINT1/2/3, RETI0/1/2/3, RESI0/1/2/3

Explanation:

NIXINT1, NIXINT2, and NIXINT3 cancel any pending interrupt events for their respective interrupt levels. These instructions prevent the interrupt from occurring even if its event condition has been met.

The P2 provides three independent interrupt levels, and each NIXINT instruction cancels only its corresponding level. Use these instructions when an interrupt that was previously configured is no longer needed or when the program needs to explicitly clear a pending interrupt condition before it can trigger cog execution flow changes.

NOP

No Operation

Miscellaneous - Consumes two clock cycles without any operation.

NOP

Result: Two clock cycles are consumed.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
0000	0000000	000	000000000	000000000	—	—	—	2

Related: WAITX, WAITCT1/2/3

Explanation:

NOP simply consumes two clock cycles without performing any operation. No registers are modified, no flags are affected, and no memory is accessed.

NOP is primarily used for timing adjustments, creating precise delays, or as a placeholder during development. It can also be used to align code for performance optimization or to fill instruction slots in pipelined operations.

NOT

Bitwise Not
Arithmetic Operations - Inverts all bits in a value.

NOT *Dest, {#}Src* **{WC|WZ|WCZ}**

NOT *Dest* **{WC|WZ|WCZ}**

Result: The bitwise NOT of Src or Dest is stored in Dest.

- Dest is the register containing the value to bitwise NOT (syntax 2) or to be replaced by the bitwise NOT of Src (syntax 1).
- Src is an optional register, 9-bit literal, or 32-bit augmented literal whose value will be bitwise NOTed and stored into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0110001	CZI	DDDDDDDDD	SSSSSSSSS	!S[31]	result == 0	D	2
EEEE	0110001	CZ0	DDDDDDDDD	DDDDDDDDD	!D[31]	result == 0	D	2

Related: AND, OR, XOR, ANDN

Explanation:

NOT performs a bitwise **NOT** operation, inverting all bits of the value in Src (syntax 1) or Dest (syntax 2), and stores the result into Dest. Each 0 bit becomes 1, and each 1 bit becomes 0.

Input	Result
0	1
1	0

When using syntax 1, NOT inverts the Src operand and stores the result into Dest. When using syntax 2 (where Src is omitted), NOT inverts the value already in Dest and stores the result back into Dest.

If the WC or WCZ effect is specified, the C flag is set to the inverse of bit 31 of the source operand. For syntax 1, this is the inverse of S[31]; for syntax 2, this is the inverse of D[31].

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if it is non-zero.

Instructions: O

This section contains all PASM2 instructions beginning with the letter O.

ONES

ONES
Arithmetic Operations - Counts the number of high bits (1s) in a value.

ONES *Dest, {#}Src* **{WC|WZ|WCZ}**

ONES *Dest* **{WC|WZ|WCZ}**

Result: The number of high bits (1s) in Src, or Dest, is stored in Dest.

- Dest is a register where the count of high bits is stored, and optionally contains the value to check (second syntax form).
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is checked for **ONES**.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0111101	CZI	DDDDDDDDD	SSSSSSSSS	Result is odd	result == 0	D	2
EEEE	0111101	CZ0	DDDDDDDDD	DDDDDDDDD	Result is odd	result == 0	D	2

Related: TEST, TESTB, TESTBN, BITNOT

Explanation:

ONES tallies the number of high bits (1s) in the specified value and stores the count in Dest. This is a population count (popcount) operation commonly used for bit manipulation and analysis.

When Src is provided in the first syntax form, **ONES** counts the high bits in Src and stores the result (0 to 32) in Dest. When Src is omitted in the second syntax form, **ONES** counts the high bits in Dest itself and replaces Dest with the count.

If the WC or WCZ effect is specified, the C flag is set (1) if the count is odd, or is cleared (0) if the count is even. This provides a parity check on the number of high bits.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero (no high bits were found), or is cleared (0) if the result is non-zero (at least one high bit exists).

ONES is useful for analyzing bit patterns, counting enabled flags, and implementing parity checks in data transmission protocols.

OR

Bitwise Or
Arithmetic Operations - Performs bitwise OR between two values.

OR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Dest OR Src is stored in Dest.

- Dest is a register containing the value to bitwise OR with Src, and is where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is bitwise ORed into Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0101010	CZI	DDDDDDDDD	SSSSSSSSS	Parity of Result	result == 0	D	2

Related: AND, XOR, ANDN, NOT

Explanation:

OR performs a bitwise **OR** operation between the values in Dest and Src, storing the result in Dest. Each bit position in the result is set (1) if the corresponding bit in either Dest or Src (or both) is set, and is cleared (0) only if both corresponding bits are cleared.

The bitwise **OR** operation follows this truth table for each bit position:

Dest	Src	Result
0	0	0
0	1	1
1	0	1
1	1	1

If the WC or WCZ effect is specified, the C flag is set (1) if the result contains an odd number of high bits, or is cleared (0) if it contains an even number of high bits. This provides a parity indication of the result.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if the result is non-zero. Note that the result can only be zero if both Dest and Src were zero.

OR is commonly used for setting specific bits in a value, combining bit masks, and implementing logical operations in algorithms.

OUTC / OUTNC / OUTZ / OUTNZ

Output By Flag State
Pin I/O and Smart Pins - Sets pin output level based on flag state.

OUTC *{#}Dest* **{WCZ}**

OUTNC *{#}Dest* **{WCZ}**

OUTZ *{#}Dest* **{WCZ}**

OUTNZ *{#}Dest* **{WCZ}**

Result: The I/O pin output level bit(s) described by Dest are set according to the flag state. Optionally sets C and Z to the original output state.

- Dest identifies the I/O pin(s): Dest[5:0] = base pin (0-63), Dest[10:6] = additional contiguous pins.

- WCZ is an optional effect to set C and Z to the original output state.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001001010	OUT bit†	OUT bit†	OUTx	2
EEEE	1101011	CZL	DDDDDDDDD	001001011	OUT bit†	OUT bit†	OUTx	2
EEEE	1101011	CZL	DDDDDDDDD	001001100	OUT bit†	OUT bit†	OUTx	2
EEEE	1101011	CZL	DDDDDDDDD	001001101	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: OUTH, OUTL, OUTNOT, OUTRND

Explanation:

These instructions set pin output level(s) based on flag state:

Instruction	Drives high when
OUTC	C = 1
OUTNC	C = 0
OUTZ	Z = 1
OUTNZ	Z = 0

OUTC and **OUTZ** drive high when their flag is set; **OUTNC** and **OUTNZ** drive high when their flag is clear.

If WCZ is specified, both the C flag and the Z flag are set to the original output state of the base pin before modification.

OUTH

Output High
Pin I/O and Smart Pins - Sets pin output level to high (1).

OUTH *{#}Dest* **{WCZ}**

Result: The I/O pin output level bit(s) described by Dest are set high (1).

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set high.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001001001	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: OUTL, OUTNOT, OUTC, OUTNC, DIRH

Explanation:

OUTH sets the output level of the pin(s) specified by Dest to high (1), driving them to the high voltage level. All other output level bits remain unchanged.

Dest[5:0] specifies the base pin number (0-63). For controlling a single pin, only these lower 6 bits matter. For controlling a range of contiguous pins, Dest[10:6] specifies how many additional pins beyond the base should be affected (0-31, where 0 means just the base pin, 1 means base plus one additional pin, etc.).

A 9-bit literal Dest can express the base pin (bits [5:0]) and up to 7 additional pins (bits [8:6]). To specify a wider range, use the augmented literal prefix (##Dest) to provide an 11-bit value, which allows controlling up to 32 contiguous pins.

If the WCZ effect is specified, the C flag is set to the original state of the output level bit for the base pin, and Z is set to the same value, before the instruction executes.

OUTH is commonly used to turn on LEDs, assert control signals, or drive pins high for any digital output purpose. For the output level change to affect the actual pin voltage, the pin must also be configured as an output using the direction control instructions.

OUTL

Output Low
Pin I/O and Smart Pins - Sets pin output level to low (0).

OUTL *{#}Dest* **{WCZ}**

Result: The I/O pin output level bit(s) described by Dest are set low (0).

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set low.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001001000	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: OUTH, OUTNOT, OUTC, OUTNC, DIRL

Explanation:

OUTL sets the output level of the pin(s) specified by Dest to low (0), driving them to the low voltage level (typically ground). All other output level bits remain unchanged.

Dest[5:0] specifies the base pin number (0-63). For controlling a single pin, only these lower 6 bits matter. For controlling a range of contiguous pins, Dest[10:6] specifies how many additional pins beyond the base should be affected (0-31, where 0 means just the base pin, 1 means base plus one additional pin, etc.).

A 9-bit literal Dest can express the base pin (bits [5:0]) and up to 7 additional pins (bits [8:6]). To specify a wider range, use the augmented literal prefix (##Dest) to provide an 11-bit value, which allows controlling up to 32 contiguous pins.

If the WCZ effect is specified, the C flag is set to the original state of the output level bit for the base pin, and Z is set to the same value, before the instruction executes.

OUTL is commonly used to turn off LEDs, de-assert control signals, or drive pins low for any digital output purpose. For the output level change to affect the actual pin voltage, the pin must also be configured as an output using the direction control instructions.

OUTNOT

Output Not (Toggle)
Pin I/O and Smart Pins - Toggles pin output level to opposite state.

OUTNOT *{#}Dest* **{WCZ}**

Result: The I/O pin output level bit(s) described by Dest are toggled to their opposite state(s).

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to toggle.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001001111	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: OUTH, OUTL, OUTRND, NOT, DRVNOT

Explanation:

OUTNOT toggles the output level of the pin(s) specified by Dest to their opposite state. Pins that were high (1) become low (0), and pins that were low become high. All other output level bits remain unchanged.

Dest[5:0] specifies the base pin number (0-63). For controlling a single pin, only these lower 6 bits matter. For controlling a range of contiguous pins, Dest[10:6] specifies how many additional pins beyond the base should be affected (0-31, where 0 means just the base pin, 1 means base plus one additional pin, etc.).

A 9-bit literal Dest can express the base pin (bits [5:0]) and up to 7 additional pins (bits [8:6]). To specify a wider range, use the augmented literal prefix (##Dest) to provide an 11-bit value, which allows controlling up to 32 contiguous pins.

If the WCZ effect is specified, the C flag is set to the original state of the output level bit for the base pin, and Z is set to the same value, before the instruction executes.

OUTNOT is commonly used for blinking LEDs, generating clock signals, or toggling any output that needs to alternate states. It is particularly efficient for creating square waves or implementing state machines that alternate between two states.

OUTRND

Output Random
Pin I/O and Smart Pins - Sets pin output level to random state from PRNG.

OUTRND *{#}Dest* **{WCZ}**

Result: The I/O pin output level bit(s) described by Dest are each set randomly to low or high.

- Dest is a register, 9-bit literal, or 11-bit augmented literal whose value identifies the I/O pin(s) to set to random output levels.
- WCZ is an optional effect to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001001110	OUT bit†	OUT bit†	OUTx	2

† Original output state of the base pin (D[5:0]) before instruction executes.

Related: OUTC, OUTNC, OUTZ, OUTNZ, OUTH, OUTL, OUTNOT

Explanation:

OUTRND sets the output level of the pin(s) specified by Dest to random low and high states, using bits from the hardware Xoroshiro128** pseudo-random number generator (PRNG). Each affected pin is independently set to either low (0) or high (1) based on successive bits from the PRNG. All other output level bits remain unchanged.

Dest[5:0] specifies the base pin number (0-63). For controlling a single pin, only this lower 6-bit value matters. For controlling a range of contiguous pins, Dest[10:6] specifies how many additional pins beyond the base should be affected (0-31, where 0 means just the base pin, 1 means base plus one additional pin, etc.).

A 9-bit literal Dest can express the base pin (bits [5:0]) and up to 7 additional pins (bits [8:6], allowing control of 1 to 8 contiguous pins). To specify a wider range, use the augmented literal prefix (##Dest) to provide an 11-bit value, which allows controlling up to 32 contiguous pins.

When Dest is a register, the register's bits [10:0] are used directly to form the 11-bit pin range specification. However, if a **SETQ** instruction immediately precedes **OUTRND**, then **SETQ**'s Dest[4:0] is substituted for the register's bits [10:6], allowing dynamic control of the pin range.

If the WCZ effect is specified, both the C and Z flags are set to the original state of the output level bit for the base pin, before the instruction executes.

OUTRND is useful for generating random visual patterns on LEDs, creating noise signals for testing or audio applications, or implementing randomized control sequences. The quality of randomness depends on proper initialization of the PRNG using the SETRAND instruction.

Instructions: P

This section contains all PASM2 instructions beginning with the letter P.

POLLATN

Poll Attention Event
Events and Timing - Polls and clears the inter-cog attention event flag.

POLLATN {WC|WZ|WCZ}

Result: Attention event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001110	000100100	ATN Event	ATN Event	—	2

Related: COGATN, WAITATN, JATN, JNATN

Explanation:

POLLATN copies the state of the attention event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the attention event flag prior to clearing it.

The attention event flag is set whenever another cog issues an attention request for this cog using **COGATN**. The flag is cleared upon cog start, or execution of **POLLATN**, **WAITATN**, **JATN**, or **JNATN** instructions.

This instruction enables inter-cog communication by allowing a cog to check whether another cog has requested its attention without blocking execution.

POLLCT1 / POLLCT2 / POLLCT3

Poll Counter Event
Events and Timing - Polls and clears the system counter event flag.

POLLCT1 {WC|WZ|WCZ}

POLLCT2 {WC|WZ|WCZ}

POLLCT3 {WC|WZ|WCZ}

Result: CTn event flag state is optionally copied into C and/or Z, then the flag is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000000001	000100100	CT1 Event	CT1 Event	—	2
EEEE	1101011	CZ0	000000010	000100100	CT2 Event	CT2 Event	—	2
EEEE	1101011	CZ0	000000011	000100100	CT3 Event	CT3 Event	—	2

Related: ADDCT1/2/3, WAITCT1/2/3, JCT1/2/3, JNCT1/2/3

Explanation:

POLLCT1, POLLCT2, and POLLCT3 copy the state of their respective counter event flags into C and/or Z and then clear the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the counter event flag prior to clearing it.

Each counter event flag is set whenever the System Counter (CT) passes the value in that counter's event trigger register; that is, the MSB of (CT - CTn) is 0. The counter event flag is cleared upon execution of ADDCTn, POLLCTn, WAITCTn, JCTn, or JNCTn.

These instructions enable time-based event polling without blocking execution. The P2 provides three independent counter event triggers (CT1, CT2, CT3) allowing a cog to simultaneously track multiple timing requirements.

POLLFBW

Poll FIFO Block Wrap Event
Events and Timing - Polls and clears the FIFO block wrap event flag.

POLLFBW {WC|WZ|WCZ}

Result: FIFO-interface-block-wrap event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001001	000100100	FBW Event	FBW Event	—	2

Related: RDFAST, WRFAST, FBLOCK, WAITFBW, JFBW, JNFBW

Explanation:

POLLFBW copies the state of the FIFO-interface-block-wrap event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The FIFO-interface-block-wrap event flag is set whenever the Hub RAM FIFO interface exhausts its block count and reloads its block count and start address. The flag is cleared upon execution of **RDFAST**, **WRFAST**, **FBLOCK**, **POLLFBW**, **WAITFBW**, **JFBW**, or **JNFBW** instructions.

This instruction enables circular buffer management for high-speed Hub RAM transfers.

POLLINT

Poll Interrupt Event

Events and Timing - Polls and clears the interrupt-occurred event flag.

POLLINT {WC|WZ|WCZ}

Result: Interrupt-occurred event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000000000	000100100	INT Event	INT Event	—	2

Related: WAITINT, JINT, JNINT

Explanation:

POLLINT copies the state of the interrupt-occurred event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The interrupt-occurred event flag is set whenever interrupt 1, 2, or 3 occurs. **DEBUG** interrupts are ignored. The flag is cleared upon cog start, or execution of **POLLINT**, **WAITINT**, **JINT**, or **JNINT** instructions.

This instruction enables non-blocking interrupt handling.

POLLPAT

Poll Pin Pattern Event

Events and Timing - Polls and clears the pin pattern match event flag.

POLLPAT {WC|WZ|WCZ}

Result: Pin-pattern-detected event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001000	000100100	PAT Event	PAT Event	—	2

Related: SETPAT, WAITPAT, JPAT, JNPAT

Explanation:

POLLPAT copies the state of the pin-pattern-detected event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The pin-pattern-detected event flag is set whenever the masked input pins match or don't match the pattern described by a previous **SETPAT** instruction. The flag is cleared upon execution of **SETPAT**, **POLLPAT**, **WAITPAT**, **JPAT**, or **JNPAT** instructions.

This instruction enables non-blocking pattern detection on input pins.

POLLQMT

Poll CORDIC Empty Event

Events and Timing - Polls and clears the CORDIC empty event flag.

POLLQMT {WC|WZ|WCZ}

Result: CORDIC-read-but-empty event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001111	000100100	QMT Event	QMT Event	—	2

Related: GETQX, GETQY, JQMT, JNQMT

Explanation:

POLLQMT copies the state of the CORDIC-read-but-empty event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The CORDIC-read-but-empty event flag is set whenever **GETQX** or **GETQY** executes without any CORDIC results available or in progress. The flag is cleared upon cog start or execution of **POLLQMT**, WAITQMT, **JQMT**, or **JNQMT** instructions.

This instruction enables error detection for CORDIC operations.

POLLSE1 / POLLSE2 / POLLSE3 / POLLSE4

Poll Selectable Event

Events and Timing - Polls and clears a configurable selectable event flag.

POLLSE1 {WC|WZ|WCZ}

POLLSE2 {WC|WZ|WCZ}

POLLSE3 {WC|WZ|WCZ}

POLLSE4 {WC|WZ|WCZ}

Result: SEn event flag state is optionally copied into C and/or Z, then the flag is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000000100	000100100	SE1 Event	SE1 Event	—	2
EEEE	1101011	CZ0	000000101	000100100	SE2 Event	SE2 Event	—	2
EEEE	1101011	CZ0	000000110	000100100	SE3 Event	SE3 Event	—	2
EEEE	1101011	CZ0	000000111	000100100	SE4 Event	SE4 Event	—	2

Related: SETSE1/2/3/4, WAITSE1/2/3/4, JSE1/2/3/4, JNSE1/2/3/4

Explanation:

POLLSE1, POLLSE2, POLLSE3, and POLLSE4 copy the state of their respective selectable event flags into C and/or Z and then clear the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the selectable event flag prior to clearing it.

Each selectable event flag is set whenever the corresponding configured event occurs. The flag is cleared upon execution of SETSEn, POLLSEn, WAITSEn, JSEn, or JNSEn instructions.

The P2 provides four independent selectable event generators that can be configured to monitor various hardware conditions.

POLLXFI

Poll Streamer Finished Event
Events and Timing - Polls and clears the streamer finished event flag.

POLLXFI {WC|WZ|WCZ}

Result: Streamer-finished event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001011	000100100	XFI Event	XFI Event	—	2

Related: XINIT, XZERO, XCONT, WAITXFI, JXFI, JNXFI

Explanation:

POLLXFI copies the state of the streamer-finished event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The streamer-finished event flag is set whenever the streamer runs out of commands to process. The flag is cleared upon execution of **XINIT**, **XZERO**, **XCONT**, **POLLXFI**, **WAITXFI**, **JXFI**, or **JNXFI** instructions.

This instruction enables non-blocking management of the streamer subsystem.

POLLXMT

Poll Streamer Empty Event
Events and Timing - Polls and clears the streamer empty event flag.

POLLXMT {WC|WZ|WCZ}

Result: Streamer-empty event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001010	000100100	XMT Event	XMT Event	—	2

Related: XINIT, XZERO, XCONT, WAITXMT, JXMT, JNXMT

Explanation:

POLLXMT copies the state of the streamer-empty event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The streamer-empty event flag is set whenever the streamer is ready for a new command. The flag is cleared upon execution of **XINIT**, **XZERO**, **XCONT**, **POLLXMT**, **WAITXMT**, **JXMT**, or **JNXMT** instructions.

This instruction enables pipelined streamer operations.

POLLXRL

Poll Streamer LUT Rollover Event
Events and Timing - Polls and clears the streamer LUT rollover event flag.

POLLXRL {WC|WZ|WCZ}

Result: Streamer-LUT-RAM-rollover event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001101	000100100	XRL Event	XRL Event	—	2

Related: XINIT, XZERO, XCONT, WAITXRL, JXRL, JNXRL

Explanation:

POLLXRL copies the state of the streamer-LUT-RAM-rollover event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The streamer-LUT-RAM-rollover event flag is set whenever location $1FF of the Lookup RAM is read by the streamer. The flag is cleared upon cog start or upon execution of **POLLXRL**, **WAITXRL**, **JXRL**, or **JNXRL** instructions.

This instruction enables circular buffer management when using LUT RAM as a streamer data source.

POLLXRO

Poll Streamer NCO Rollover Event
Events and Timing - Polls and clears the streamer NCO rollover event flag.

POLLXRO {WC|WZ|WCZ}

Result: Streamer-NCO-rollover event flag is optionally copied into C and/or Z, then it is cleared.

- WC, WZ, or WCZ are optional effects to capture the event state into flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000001100	000100100	XRO Event	XRO Event	—	2

Related: XINIT, XZERO, XCONT, WAITXRO, JXRO, JNXRO

Explanation:

POLLXRO copies the state of the streamer NCO rollover event flag into C and/or Z and then clears the flag (unless it's being set again by the event sensor). If the WC, WZ, or WCZ effect is specified, the C flag and/or Z flag is updated to the state of the event flag prior to clearing it.

The streamer-NCO-rollover event flag is set whenever the streamer's numerically-controlled oscillator (NCO) rolls over. The flag is cleared upon execution of **XINIT**, **XZERO**, **XCONT**, **POLLXRO**, **WAITXRO**, **JXRO**, or **JNXRO** instructions.

This instruction enables precise timing control for streamer operations that use the NCO for rate control.

POP

POP From Internal Stack
Miscellaneous - Pops a value from the internal K register stack.

POP *Dest* **{WC|WZ|WCZ}**

Result: Dest receives the value from the K register.

- Dest is the register to receive the popped value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000101011	K[31]	result == 0	D	2

Related: PUSH, POPA, POPB

Explanation:

POP pops the internal stack register K into the destination register Dest. The P2 provides a single-level internal stack register K that is automatically used by **CALL** instructions to store the return address.

If the WC or WCZ effect is specified, the C flag is set to bit 31 of the popped value.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the popped value equals zero, or is cleared (0) if non-zero.

POP retrieves this value, typically as part of a return sequence, though it can also be used to retrieve any value previously stored with **PUSH**.

POPA

POP From Hub Stack A
Hub Memory Access - Pops a long from Hub memory using PTRA as stack pointer.

POPA *Dest* **{WC|WZ|WCZ}**

Result: Dest receives the long value from Hub address –PTRA.

- Dest is the register to receive the popped value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011000	CZ1	DDDDDDDDD	101011111	MSB of long	result == 0	D	9…16

Related: PUSHA, POPB, POP

Explanation:

POPA reads a long from Hub address –PTRA into the destination register Dest. PTRA is automatically decremented by 4 before the read occurs (pre-decrement). Paired with **PUSHA**'s post-increment write to PTRA++, this implements an ascending stack that grows upward in memory (toward higher addresses).

If the WC or WCZ effect is specified, the C flag is set to the MSB (bit 31) of the popped value.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the popped value equals zero, or is cleared (0) if non-zero.

This instruction enables Hub RAM-based stacks for deep subroutine nesting and large temporary storage.

POPB

POP From Hub Stack B
Hub Memory Access - Pops a long from Hub memory using PTRB as stack pointer.

POPB *Dest* **{WC|WZ|WCZ}**

Result: Dest receives the long value from Hub address –PTRB.

- Dest is the register to receive the popped value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011000	CZ1	DDDDDDDDD	111011111	MSB of long	result == 0	D	9…16

Related: PUSHB, POPA, POP

Explanation:

POPB reads a long from Hub address –PTRB into the destination register Dest. PTRB is automatically decremented by 4 before the read occurs (pre-decrement). Paired with **PUSHB**'s post-increment write to PTRB++, this implements an ascending stack that grows upward (toward higher addresses) in memory.

If the WC or WCZ effect is specified, the C flag is set to the MSB (bit 31) of the popped value.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the popped value equals zero, or is cleared (0) if non-zero.

Having two independent Hub stack pointers (PTRA and PTRB) allows a cog to manage separate stacks for different purposes.

PUSH

PUSH To Internal Stack

Miscellaneous - Pushes a value onto the internal K register stack.

PUSH *{#}Dest*

Result: The value from Dest (or immediate value) is stored in the K register.

- Dest is a register or 9-bit immediate value (0-511) to push.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000101010	—	—	—	2

Related: POP, PUSHA, PUSHB

Explanation:

PUSH pushes the value in Dest (or an immediate value 0-511) onto the internal stack register K. This instruction does not affect any flags.

The P2 provides a single-level internal stack register K that is automatically used by **CALL** instructions to store the return address. **PUSH** can be used to save other values in K, though this overwrites any return address that may be stored there.

PUSHA

PUSH To Hub Stack A

Hub Memory Access - Pushes a long to Hub memory using PTRA as stack pointer.

PUSHA *{#}Dest*

Result: The long value from Dest is written to Hub address PTRA++.

- Dest is a register or 9-bit immediate value to push.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100011	0L1	DDDDDDDDD	101100001	—	—	—	3…10

Related: POPA, PUSHB, PUSH

Explanation:

PUSHA writes the long value in Dest (or a 9-bit immediate value) to Hub address PTRA++. PTRA is automatically incremented by 4 after the write occurs (post-increment).

This instruction does not affect any flags. The post-increment model means PTRA always points to the next available stack location after the **PUSH** operation.

PUSHA paired with **POPA** implements an ascending stack in Hub RAM (the pointer advances to higher addresses on each **PUSH**).

PUSHB

PUSH To Hub Stack B

Hub Memory Access - Pushes a long to Hub memory using PTRB as stack pointer.

PUSHB *{#}Dest*

Result: The long value from Dest is written to Hub address PTRB++.

- Dest is a register or 9-bit immediate value to push.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100011	0L1	DDDDDDDDD	111100001	—	—	—	3…10

Related: POPB, PUSHA, PUSH

Explanation:

PUSHB writes the long value in Dest (or a 9-bit immediate value) to Hub address PTRB++. PTRB is automatically incremented by 4 after the write occurs (post-increment).

This instruction does not affect any flags. The post-increment model means PTRB always points to the next available stack location after the **PUSH** operation.

Having two independent Hub stack pointers (PTRA and PTRB) allows a cog to manage separate stacks for different purposes.

Instructions: Q

This section contains all PASM2 instructions beginning with the letter Q. The Q instructions are part of the CORDIC coprocessor family.

QDIV

Queue Divide
CORDIC Coprocessor - Divides 64-bit by 32-bit, producing quotient and remainder.

QDIV *{#}Dest, {#}Src*

Result: Divides a 64-bit numerator by a 32-bit denominator, producing a 32-bit quotient (GETQX) and remainder (GETQY) 55 clocks later.

- Dest is a register or literal containing the lower 32 bits of the 64-bit numerator.
- Src is a register or literal containing the 32-bit denominator (divisor).
- Use **SETQ** before **QDIV** to specify the upper 32 bits of the numerator (defaults to 0 if not used).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101000	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2...9

Related: GETQX, GETQY, SETQ, QFRAC, QMUL

Explanation:

QDIV performs high-precision unsigned division using the P2's 54-stage pipelined CORDIC solver. It divides a 64-bit numerator by a 32-bit denominator, producing both a 32-bit quotient and 32-bit remainder.

The 64-bit numerator is formed by concatenating the **SETQ** value (or 0 if **SETQ** not used) as the upper 32 bits with the Dest operand as the lower 32 bits: {SETQ, Dest}. The denominator is specified in the Src operand. After 55 clocks, the quotient can be retrieved using **GETQX** and the remainder using **GETQY**.

```
1         QDIV    ##1000000, #3  ' {0, 1000000} / 3
2         ' Wait 55 clocks...
3         GETQX   quotient       ' Get 333333
4         GETQY   remainder      ' Get 1
```

Division by zero produces undefined results. Each cog can issue one CORDIC instruction per hub window (every 8 clocks).

QEXP

Queue Exponential
CORDIC Coprocessor - Converts logarithm to integer (antilog/exponential).

QEXP *{#}Dest*

Result: Converts a 5:27-bit logarithm format into a 32-bit unsigned integer, retrieved via GETQX 55 clocks later.

- Dest is a register or literal containing the 5:27-bit logarithm (5-bit exponent in bits [31:27], 27-bit fraction in bits [26:0]).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000001111	—	—	—	2...9

Related: GETQX, QLOG, QMUL

Explanation:

QEXP performs logarithm to integer conversion using the P2's 54-stage pipelined CORDIC solver. It converts a 5:27-bit logarithm format into a 32-bit unsigned integer, effectively computing the exponential (antilog) of the input.

The instruction takes the logarithm value in the Dest operand, which must be in P2's 5:27 format where bits [31:27] contain the 5-bit whole exponent and bits [26:0] contain the 27-bit fractional exponent. After 55 clocks, the integer result can be retrieved using **GETQX**.

QEXP is the complement of **QLOG** and is commonly used together with **QLOG** to perform power calculations.

```
1         QEXP    log_value      ' Begin exponential conversion
2         ' Wait 55 clocks...
3         GETQX   integer_result ' Get 32-bit integer
```

QFRAC

Queue Fractional Divide
CORDIC Coprocessor - Divides 64-bit by 32-bit with reversed operand arrangement.

QFRAC *{#}Dest, {#}Src*

Result: Divides a 64-bit numerator by a 32-bit denominator, producing a 32-bit quotient (GETQX) and remainder (GETQY) 55 clocks later.

- Dest is a register or literal containing the upper 32 bits of the 64-bit numerator.
- Src is a register or literal containing the 32-bit denominator (divisor).
- Use **SETQ** before **QFRAC** to specify the lower 32 bits of the numerator (defaults to 0 if not used).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101001	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2...9

Related: GETQX, GETQY, SETQ, QDIV, QMUL

Explanation:

QFRAC performs fractional division using the P2's 54-stage pipelined CORDIC solver. It divides a 64-bit numerator by a 32-bit denominator, but differs from **QDIV** in the operand arrangement: Dest forms the upper 32 bits while **SETQ** (or 0) forms the lower 32 bits.

The 64-bit numerator is formed as {Dest, SETQ}. This arrangement makes QFRAC particularly suitable for fractional arithmetic where the integer part is in Dest and the fractional part is in SETQ.

```
        SETQ    ##$C0000000     ' 0.75 in 32-bit fraction format
        QFRAC   #5, #2          ' {5, 0.75} / 2 = 2.875
        ' Wait 55 clocks...
        GETQX   quotient        ' Get integer quotient
        GETQY   remainder       ' Get fractional remainder
```

QLOG

Queue Logarithm

CORDIC Coprocessor - Converts 32-bit integer to logarithm format.

QLOG *{#}Dest*

Result: Converts a 32-bit unsigned integer into a 5:27-bit logarithm format, retrieved via GETQX 55 clocks later.

- Dest is a register or literal containing the 32-bit unsigned integer input.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000001110	—	—	—	2...9

Related: GETQX, QEXP

Explanation:

QLOG performs integer to logarithm conversion using the P2's 54-stage pipelined CORDIC solver. It converts a 32-bit unsigned integer into a 5:27-bit logarithm format, where the result contains a 5-bit whole exponent in bits [31:27] and a 27-bit fractional exponent in bits [26:0].

The instruction takes the unsigned integer value in the Dest operand. After 55 clocks, the logarithm result can be retrieved using **GETQX**.

```
        QLOG    ##1000          ' Begin log conversion
        ' Wait 55 clocks...
        GETQX   log_result      ' Get 5:27 logarithm
```

QMUL

Queue Multiply

CORDIC Coprocessor - Multiplies two 32-bit values, producing 64-bit result.

QMUL *{#}Dest, {#}Src*

Result: Multiplies two 32-bit unsigned values, producing a 64-bit result with lower 32 bits via GETQX and upper 32 bits via GETQY, 55 clocks later.

- Dest is a register or literal containing the first 32-bit multiplicand.
- Src is a register or literal containing the second 32-bit multiplicand.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101000	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2...9

Related: GETQX, GETQY, QDIV, QFRAC

Explanation:

QMUL performs high-precision unsigned multiplication using the P2's 54-stage pipelined CORDIC solver. It multiplies two 32-bit unsigned integers (Dest × Src) and produces a full 64-bit product, avoiding the precision loss that would occur with standard 32-bit multiplication.

After 55 clocks, the 64-bit result can be retrieved using **GETQX** for the lower 32 bits and **GETQY** for the upper 32 bits.

```
1         QMUL    ##1000000, ##2000000
2         ' Wait 55 clocks...
3         GETQX   lower_32        ' Get lower 32 bits
4         GETQY   upper_32        ' Get upper 32 bits
```

Each cog can issue one CORDIC instruction per hub window (every 8 clocks), allowing efficient pipelining.

QROTATE

Queue Rotate
CORDIC Coprocessor - Rotates coordinate pair around origin by specified angle.

QROTATE *{#}Dest, {#}Src*

Result: Rotates a coordinate pair around the origin, producing new X (GETQX) and Y (GETQY) coordinates 55 clocks later.

- Dest is a register or literal containing the X coordinate (32-bit signed).
- Src is a register or literal containing the rotation angle in P2 angle units ($00000000 = 0°, $40000000 = 90°, $80000000 = 180°, $C0000000 = 270°).
- Use **SETQ** before **QROTATE** to specify the Y coordinate (defaults to 0 if not used).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101010	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2...9

Related: GETQX, GETQY, SETQ, QVECTOR

Explanation:

QROTATE performs point rotation using the P2's 54-stage pipelined CORDIC solver. It rotates a 32-bit signed (X, Y) coordinate pair around the origin (0, 0) by a specified angle, producing new 32-bit signed (X, Y) results.

The instruction takes the X coordinate from Dest and the Y coordinate from the **SETQ** value (or 0 if **SETQ** was not used). The rotation angle is specified in Src using P2's standard angle units.

This instruction can also be used for polar to cartesian conversion by setting X (Dest) to the length, Y (SETQ) to 0, and the angle (Src) to the desired angle.

```
        SETQ    #200            ' Set Y coordinate
        QROTATE #100, ##$20000000 ' X=100, angle=45 degrees
        ' Wait 55 clocks...
        GETQX   new_x           ' Get rotated X
        GETQY   new_y           ' Get rotated Y
```

QSQRT

Queue Square Root
CORDIC Coprocessor - Calculates square root of a 64-bit value.

QSQRT *{#}Dest, {#}Src*

Result: Calculates the square root of a 64-bit value, producing a 32-bit result via GETQX 55 clocks later.

- Dest is a register or literal containing the lower 32 bits of the 64-bit input value.
- Src is a register or literal containing the upper 32 bits of the 64-bit input value.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101001	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2...9

Related: GETQX, QMUL

Explanation:

QSQRT performs square root calculation using the P2's 54-stage pipelined CORDIC solver. It calculates the square root of a 64-bit unsigned value and produces a 32-bit result.

The 64-bit input is formed by concatenating the Src operand as the upper 32 bits with the Dest operand as the lower 32 bits, creating the value {Src, Dest}. After 55 clocks, the 32-bit square root result can be retrieved using **GETQX**.

The result is the largest integer whose square does not exceed the input value.

```
        QSQRT   ##1000000, #0  ' sqrt(1000000) = 1000
        ' Wait 55 clocks...
        GETQX   sqrt_result    ' Get 1000
```

For 32-bit square roots, use Src=0.

QVECTOR

Queue Vector
CORDIC Coprocessor - Converts cartesian coordinates to polar form.

QVECTOR *{#}Dest, {#}Src*

Result: Converts cartesian coordinates to polar form, producing length (GETQX) and angle (GETQY) 55 clocks later.

- Dest is a register or literal containing the X coordinate (32-bit signed).
- Src is a register or literal containing the Y coordinate (32-bit signed).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101010	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2...9

Related: GETQX, GETQY, QROTATE

Explanation:

QVECTOR performs cartesian to polar coordinate conversion using the P2's 54-stage pipelined CORDIC solver. It converts a 32-bit signed (X, Y) cartesian coordinate pair into a 32-bit (length, angle) polar coordinate pair.

The instruction takes the X coordinate in Dest and Y coordinate in Src, both as 32-bit signed values. After 55 clocks, the results can be retrieved using **GETQX** for the length and **GETQY** for the angle.

The angle result uses P2's standard angle units where $00000000 = 0°, $40000000 = 90°, $80000000 = 180°, and $C0000000 = 270°.

QVECTOR is the inverse operation of **QROTATE**.

```
1         QVECTOR #100, #200     ' Begin conversion
2         ' Wait 55 clocks...
3         GETQX   length         ' Get polar length
4         GETQY   angle          ' Get polar angle
```

Instructions: R

This section contains all PASM2 instructions beginning with the letter R.

RCL

Rotate Carry Left
Arithmetic Operations - Shifts bits left, inserting carry flag as new LSBs.

RCL *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted left by Src bits, inserting C as new LSBs.

- Dest is a register containing the value to rotate carry left.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to rotate.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000101	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[31].

Related: RCR, ROL, ROR

Explanation:

RCL shifts Dest's binary value left by Src places (0-31 bits) and sets the new LSBs to C. The carry flag acts as an extension of the register, allowing 33-bit rotations.

If the WC or WCZ effect is specified, the C flag is updated to the value of the last bit shifted out if Src is 1-31, or to Dest[31] if Src is 0.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if non-zero.

This instruction is useful for multi-precision arithmetic operations where the carry from one **WORD** needs to be propagated into the next **WORD**.

RCR

Rotate Carry Right
Arithmetic Operations - Shifts bits right, inserting carry flag as new MSBs.

RCR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted right by Src bits, inserting C as new MSBs.

- Dest is a register containing the value to rotate carry right.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to rotate.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000100	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[0].

Related: RCL, ROL, ROR

Explanation:

RCR shifts Dest's binary value right by Src places (0-31 bits) and sets the new MSBs to C. The carry flag acts as an extension of the register, allowing 33-bit rotations.

If the WC or WCZ effect is specified, the C flag is updated to the value of the last bit shifted out if Src is 1-31, or to Dest[0] if Src is 0.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if non-zero.

This instruction is useful for multi-precision arithmetic operations where the carry needs to be propagated through multiple words.

RCZL

Rotate Carry And Zero Left
Arithmetic Operations - Shifts bits left by two, inserting C and Z as new LSBs.

RCZL *Dest* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted left by two places and C and Z are inserted as new LSBs.

- Dest is a register containing the value to rotate.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	001101011	D[31]	D[30]	D	2

Related: RCZR, RCL, RCR

Explanation:

RCZL shifts Dest's binary value left by two places and sets Dest[1] to C and Dest[0] to Z.

If the WC or WCZ effect is specified, the C flag is updated to the original Dest[31] state.

If the WZ or WCZ effect is specified, the Z flag is updated to the original Dest[30] state.

This instruction provides a compact way to shift two flag states into a register while simultaneously extracting two bits from the register into the flags, enabling efficient state serialization and deserialization.

RCZR

Rotate Carry And Zero Right
Arithmetic Operations - Shifts bits right by two, inserting C and Z as new MSBs.

RCZR *Dest* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted right by two places and C and Z are inserted as new MSBs.

- Dest is a register containing the value to rotate.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	001101010	D[1]	D[0]	D	2

Related: RCZL, RCL, RCR

Explanation:

RCZR shifts Dest's binary value right by two places and sets Dest[31] to C and Dest[30] to Z.

If the WC or WCZ effect is specified, the C flag is updated to the original Dest[1] state.

If the WZ or WCZ effect is specified, the Z flag is updated to the original Dest[0] state.

This instruction provides a compact way to shift two flag states into a register while simultaneously extracting two bits from the register into the flags, enabling efficient state serialization and deserialization.

RDBYTE

Read **BYTE** From Hub

Hub Memory Access - Reads a zero-extended **BYTE** from Hub memory into a register.

RDBYTE *Dest, {#}Src/Ptr* **{WC|WZ|WCZ}**

Result: A zero-extended byte from Hub address Src or pointer (PTRA/PTRB) is loaded into Dest.

- Dest is the register to receive the byte value.
- Src/Ptr is a Hub address from register, immediate value, or pointer register (PTRA/PTRB).
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010110	CZI	DDDDDDDDD	SSSSSSSSS	MSB of byte	result == 0	D	9…16 †

† **Timing varies by execution context:**

Context	Clocks
COG execution	9…16
Hub execution	9…26
COG with interrupts	9…24
Hub with interrupts	9…44

Related: RDWORD, RDLONG, WRBYTE

Explanation:

RDBYTE reads a byte from Hub memory at the address specified by Src (or pointer register) and loads it into Dest with zero extension (bits 31:8 are cleared to 0). Timing depends on execution context: 9-16 cycles

for COG execution, 9-26 for Hub execution, with additional latency when interrupts are enabled (9-24 for COG, 9-44 for Hub). The cog must wait for its Hub access window.

If preceded by a **SETQ** instruction, burst reads of multiple bytes can be performed.

If the WC or WCZ effect is specified, C is set to the MSB of the byte.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

Hub memory operations follow a round-robin access pattern where each cog gets a regular time slot. The actual latency depends on when the request arrives relative to the cog's assigned slot.

RDFAST

Read Fast Via FIFO

Hub Memory Access - Begins fast Hub read operation via FIFO for high-throughput streaming.

RDFAST *{#}Dest, {#}Src*

Result: A fast read operation begins, filling the FIFO with data from Hub memory starting at address Src.

- Dest is a configuration value: Dest[31] = no-wait mode, Dest[13:0] = block size in 64-byte units (0 = maximum).
- Src is the Hub memory start address (Src[19:0]) for the read operation.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100011	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2 or WRFAST finish + 10…17 †

† **Timing varies by execution context:**

Context	Clocks
COG execution	2 or WRFAST finish + 10…17
Hub execution	*Not available—FIFO in use*
COG with interrupts	2 or WRFAST finish + 10…25
Hub with interrupts	*Not available—FIFO in use*

Note: FIFO operations require COG execution mode. When code runs from Hub memory, the FIFO is used for instruction fetch and cannot be redirected for data streaming.

Related: RFBYTE, RFWORD, RFLONG, WRFAST, FBLOCK

Explanation:

RDFAST begins a new fast Hub read operation via the FIFO. The instruction configures automatic sequential reading from Hub memory with background FIFO refill, enabling high-throughput streaming data processing. This instruction is only available when executing from COG/LUT memory, not Hub memory.

Dest[31] = 1 enables no-wait mode, which prevents stalls when the FIFO is being filled. Dest[13:0] specifies the block size in 64-byte units, with 0 indicating maximum size (16384 longs). Src[19:0] specifies the starting Hub address. The FIFO automatically wraps at the block boundary.

After **RDFAST** is executed, subsequent **RFBYTE**, **RFWORD**, or **RFLONG** instructions read data from the FIFO. The FIFO is automatically refilled in the background, making this ideal for checksums, CRC calculations, data processing, and block copy operations.

RDLONG

Read **LONG** From Hub

Hub Memory Access - Reads a 32-bit **LONG** from Hub memory into a register.

RDLONG *Dest, {#}Src/Ptr* **{WC|WZ|WCZ}**

Result: A long from Hub address Src or pointer (PTRA/PTRB) is loaded into Dest.

- Dest is the register to receive the long value.
- Src/Ptr is a Hub address from register, immediate value, or pointer register (PTRA/PTRB).
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011000	CZI	DDDDDDDDD	SSSSSSSSS	MSB of long	result == 0	D	9…16 †

† **Timing varies by execution context:**

Context	Clocks
COG execution	9…16
Hub execution	9…26
COG with interrupts	9…24
Hub with interrupts	9…44

Related: RDBYTE, RDWORD, WRLONG

Explanation:

RDLONG reads a long from Hub memory at the address specified by Src (or pointer register) and loads it into Dest. Timing depends on execution context: 9-16 cycles for COG execution, 9-26 for Hub execution, with additional latency when interrupts are enabled (9-24 for COG, 9-44 for Hub). The cog must wait for its Hub access window.

If preceded by a **SETQ** instruction, burst reads of multiple longs can be performed.

If the WC or WCZ effect is specified, C is set to the MSB of the long.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

Hub memory operations follow a round-robin access pattern where each cog gets a regular time slot.

Pitfall (Silicon Bug): When using **SETQ**/SETQ2 for block transfers with PTRx expressions, do NOT place any ALTx, **AUGS**, or **AUGD** instruction between **SETQ**/SETQ2 and **RDLONG**. Such intervening

instructions cancel the block-size PTRx delta calculation—the data transfers correctly, but PTRx advances by only a single-long delta (4 bytes) instead of the full block size. This leads to corrupted subsequent operations if you expect PTRx to point past the block.

RDLUT

Read From LUT

Lookup Table - Reads data from the cog's lookup table memory.

RDLUT *Dest, {#}Src/Ptr* **{WC|WZ|WCZ}**

Result: Data from LUT address Src or pointer (PTRA/PTRB) is loaded into Dest.

- Dest is the register to receive the data.
- Src/Ptr is a LUT address from register, immediate value, or pointer register (PTRA/PTRB).
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010101	CZI	DDDDDDDDD	SSSSSSSSS	MSB of data	result == 0	D	3

Related: WRLUT, RDLONG

Explanation:

RDLUT reads data from the Lookup Table at the address specified by Src (or pointer register) and loads it into Dest. The LUT is a 512-long (2KB) memory area in each cog that can be used for lookup tables, buffers, or general-purpose memory. The operation takes 3 clock cycles.

If the WC or WCZ effect is specified, C is set to the MSB of the data.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

The LUT provides fast local memory access for frequently accessed data structures, making it ideal for sin/cos tables, gamma correction tables, and small data buffers.

RDPIN

Read Smart Pin

Pin I/O and Smart Pins - Reads Smart Pin result and acknowledges, clearing the ready flag.

RDPIN *Dest, {#}Src* **{WC}**

Result: Smart Pin Src[5:0] result is loaded into Dest, and the pin is acknowledged.

- Dest is the register to receive the pin result.
- Src is a register or literal identifying the pin number (Src[5:0]) to read from.
- WC is an optional effect to write the modal result to C.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010100	C1I	DDDDDDDDD	SSSSSSSSS	Modal result	—	D	2

Related: RQPIN, WRPIN, WXPIN, WYPIN

Explanation:

RDPIN reads the result value from the specified Smart Pin and acknowledges the pin, clearing its "ready" flag. The result value depends on the pin's configured mode and represents measurement data such as pulse width, period, edge count, ADC value, or serial data.

If the WC effect is specified, the C flag is set to the modal result, which provides mode-specific status information.

Smart Pins are powerful autonomous I/O processors that can measure timing, count edges, perform A/D conversion, generate PWM, and communicate serially without continuous CPU intervention. **RDPIN** retrieves the measured or received data after the pin signals completion.

RDWORD

Read **WORD** From Hub

Hub Memory Access - Reads a zero-extended **WORD** from Hub memory into a register.

RDWORD *Dest, {#}Src/Ptr* **{WC|WZ|WCZ}**

Result: A zero-extended word from Hub address Src or pointer (PTRA/PTRB) is loaded into Dest.

- Dest is the register to receive the word value.
- Src/Ptr is a Hub address from register, immediate value, or pointer register (PTRA/PTRB).
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010111	CZI	DDDDDDDDD	SSSSSSSSS	MSB of word	result == 0	D	9…16 †

† **Timing varies by execution context:**

Context	Clocks
COG execution	9…16
Hub execution	9…26
COG with interrupts	9…24
Hub with interrupts	9…44

Related: RDBYTE, RDLONG, WRWORD

Explanation:

RDWORD reads a word from Hub memory at the address specified by Src (or pointer register) and loads it into Dest with zero extension (bits 31:16 are cleared to 0). Timing depends on execution context: 9-16 cycles for COG execution, 9-26 for Hub execution, with additional latency when interrupts are enabled (9-24 for COG, 9-44 for Hub). The cog must wait for its Hub access window.

If preceded by a **SETQ** instruction, burst reads of multiple words can be performed.

If the WC or WCZ effect is specified, C is set to the MSB of the word.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

REP

Repeat Block
Branching and Flow Control - Creates a zero-overhead hardware loop for repeated execution.

REP *{#}Dest, {#}Src*

REP *@.label, {#}Src*

Result: The next Dest[8:0] instructions are executed Src times.

- Dest is the number of instructions to repeat (Dest[8:0], 0-511). If Dest[8:0] = 0, nothing repeats.
- Src is the number of repetitions. If Src = 0, instructions repeat infinitely.
- Alternatively, `@.label` calculates the instruction count automatically from a local label.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100110	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: DJNZ, JNCT1/2/3

Explanation:

REP creates a hardware-implemented loop that executes the next Dest[8:0] instructions Src times. If Src = 0, the instructions repeat infinitely (useful for main loops). If Dest[8:0] = 0, nothing repeats.

The **REP** instruction itself takes 2 cycles, and the repeated instructions execute with zero overhead—no jump penalty, no counter decrement. This makes **REP** ideal for time-critical inner loops.

REP blocks cannot be nested. The P2 hardware uses a single internal counter for **REP** execution; starting a new **REP** while one is active overwrites the existing repeat state. For nested iteration, use **REP** for the inner loop and branch instructions (DJNZ) for outer loops. Interrupts are blocked during **REP** execution to maintain timing precision. The zero-overhead nature of **REP** makes it essential for high-performance applications like DSP algorithms, graphics rendering, and precise timing operations.

Critical Restrictions:

- **Branches cancel REP:** Any branch instruction (JMP, **CALL**, **DJNZ**, **TJZ**, etc.) executed within the repeated block immediately cancels **REP** activity. The branch executes normally, but repetition stops. This includes conditional branches that are taken.
- **Hub memory overhead:** When **REP** executes from Hub memory (ORGH section), it remains functional but is no longer zero-overhead: each iteration's hidden return-jump pays the hub-branch refill cost (13+ clocks). For zero-overhead inner loops, execute **REP** from COG or LUT memory; for non-time-critical loops, hub-exec **REP** works correctly with the documented per-iteration penalty.

Forbidden instructions in REP blocks: - Branch instructions: **JMP**, **CALL**, **CALLA**, **CALLB**, **CALLD** - Conditional branches: **DJNZ**, **DJZ**, **TJZ**, **TJNZ**, **IJZ**, **IJNZ** - Any instruction that modifies PC

Using Labels Instead of Counts:

The `@.label` syntax enables **REP** to automatically calculate the instruction count from a local label placed after the repeated block. The assembler computes the distance between **REP** and the label at assembly

time. This approach is preferred over hardcoded counts because it remains correct when instructions are added or removed.

Example using instruction count (fragile):

```
' Hardcoded count - breaks if code changes
                REP     #3, count               ' Repeat next 3 instructions
                RDLONG  x, ptr                  ' 1st
                ADD     ptr, #4                 ' 2nd
                ADD     sum, x                  ' 3rd
                ' If you add code here, the count becomes wrong!
```

Example using local label (preferred):

```
' Label-based count - automatically correct
process_data    REP     @.end, count            ' Repeat until .end label
                RDLONG  x, ptr                  ' Instructions between REP
                ADD     ptr, #4                 ' and label are counted
                ADD     sum, x                  ' automatically
.end                                            ' Empty label marks end

' Alternative using the # prefix with local label:
fill_buffer     REP     #(.done - $), #256      ' Expression = count
                WRBYTE  value, ptr
                ADD     ptr, #1
.done
```

Pitfall: When using the label form, place the label immediately after the last repeated instruction. The label must be within the same local scope (same enclosing global label). See Chapter 2.10 for label scoping rules.

Extended Count Capability:

Both the instruction count (D) and repetition count (S) can exceed the 9-bit immediate limit of 0-511 using two methods:

Form	Limit	Mechanism
`#count`	0-511	9-bit immediate field
`##count`	0 to 2^{}32-1	AUGD/AUGS prefix emitted automatically
`register`	0 to 2^{}32-1	Register value used at runtime

```
' Extended repetition examples
                REP     @.end, ##1000          ' 1000 reps (AUGS prefix)
```

continues on next page →

↪ *continued from previous page*

```
3                   REP     @.end, big_count        ' Register-based count
4                   REP     ##1000, ##2000          ' Both extended (rare)
```

Memory Mode Constraints (for @label form):

The `@label` end position is constrained by both the execution mode and the 9-bit encoding limit:

Memory Mode	Address Range	@label Constraint
COG only	$000-$1FF	min(511 instructions, $1FF - current)
COG + LUT	$000-$3FF	min(511 instructions, $3FF - current)
LUT only	$200-$3FF	min(511 instructions, $3FF - current)
Hub (ORGH)	$00000-$7FFFF	511 instructions (encoding limit)

REP blocks can span from COG RAM into LUT RAM when executing in combined COG+LUT mode.

Interrupt Protection Pattern:

A common PASM2 idiom uses **REP** with repetition count = 1 to stall interrupts during critical operations. (Note: This pattern is only needed in PASM2 code with interrupts enabled; Spin2 operators are already protected by the interpreter.)

```
1  ' Protect CORDIC operation from interrupts
2                   REP     @.stall, #1             ' Run block once, atomically
3                   QMUL    y, x                    ' CORDIC multiply
4                   GETQX   x                       ' Get result
5                   GETQY   y                       ' Get overflow
6  .stall
```

This works because **REP** stalls interrupt handling until all repeated instructions complete, even with just one iteration.

Extended Interrupt Stall:

For longer critical sequences, use a large instruction count with repetition = 1:

```
1  ' Stall interrupts until ret/_ret_ is encountered
2  op_quna          REP     #99, #1                 ' Large count, exits on ret
3                   QSQRT   x, #0                   ' CORDIC operations...
4                   QLOG    x
5                   QEXP    x
6                   ...
7          _ret_    MOV     result, x               ' REP ends at _ret_
```

The large instruction count (99) with repetition count of 1 creates an interrupt-free zone that terminates at the first `RET`, `_ret_`, or branch instruction.

Conditional REP:

REP itself can be conditionally executed:

```
                TESTP   pin                 wc
    if_c        REP     @.end, #5           ' Only repeat if C set
                ADD     sum, #1
.end
```

Instructions within the **REP** block can also be conditional:

```
                REP     @.end, #4
                ADD     sum, #1
                TEST    sum, #1             wz
    if_z        ADD     result, #1          ' Conditional within block
.end
```

Bit-Bang I2C Pattern:

```
' Output 8 bits, MSB first
.wr_byte        REP     #8, #8              ' 8 instructions, 8 times
                SHL     data, #1            wc
                DRVC    sda                 ' Drive SDA with carry
                DRVH    SCL                 ' Clock high
                WAITX   delay
                DRVL    SCL                 ' Clock low
                WAITX   delay
                NOP
                NOP
```

Array Operations:

```
' Fill array with incrementing values
                MOV     counter, #0
                LOC     ptra, #\hub_array
                REP     @.arr_end, #8
                ADD     counter, #1
                WRLONG  counter, ptra++
.arr_end
```

RESI0 / RESI1 / RESI2 / RESI3

Resume From Interrupt

Interrupts - Resumes execution from an interrupted location.

RESI0
RESI1
RESI2
RESI3

Result: Execution resumes from the interrupted location for the specified interrupt level.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011001	110	111111110	111111111	—	—	—	4 (COG), 13…20 (Hub)
EEEE	1011001	110	111110100	111110101	—	—	—	4 (COG), 13…20 (Hub)
EEEE	1011001	110	111110010	111110011	—	—	—	4 (COG), 13…20 (Hub)
EEEE	1011001	110	111110000	111110001	—	—	—	4 (COG), 13…20 (Hub)

Related: RETI0/1/2/3, SETINT1/2/3, NIXINT1/2/3

Explanation:

RESI0, RESI1, RESI2, and RESI3 resume execution from their respective interrupt levels. Each instruction is functionally equivalent to a **CALLD** instruction that restores the program counter, C flag, and Z flag from the corresponding interrupt return address registers.

Unlike RETIx instructions which return from the interrupt handler, RESIx instructions resume interrupted execution, used when an interrupt handler needs to yield to another interrupt priority level before completion.

RET

Return From Subroutine
Branching and Flow Control - Returns from subroutine by popping the hardware stack.

RET {WC|WZ|WCZ}

Result: The program counter, C flag, and Z flag are restored from the top of the hardware stack.

- WC, WZ, or WCZ are optional effects to restore flags from the stack.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ1	000000000	000101101	K[31]	K[30]	—	4 / 13-20 †

† **Timing varies by execution context:**

Context	Clocks
COG / LUT execution	4
Hub execution	13…20

Related: CALL, CALLA, CALLB, RETA, RETB

Explanation:

RET returns from a subroutine by popping the hardware stack (K register). The program counter is restored from K[19:0].

If the WC or WCZ effect is specified, the C flag is restored from K[31].

If the WZ or WCZ effect is specified, the Z flag is restored from K[30].

The operation takes 4 cycles in cog/LUT execution, or 13–20 cycles in hub execution (the hub-branch refill cost when the return target resides in hub memory).

The P2 provides an 8-level hardware stack for fast subroutine calls. **RET** is paired with **CALL**, **CALLPA**, **CALLPB**, **CALLA**, and **CALLB** instructions.

RETA

Return Via PTRA Stack
Branching and Flow Control - Returns from subroutine using PTRA as software stack pointer.

RETA {WC|WZ|WCZ}

Result: The program counter, C flag, and Z flag are restored from Hub memory at –PTRA.

- WC, WZ, or WCZ are optional effects to restore flags from the stack.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ1	000000000	000101110	L[31]	L[30]	—	11…18 †

† **Timing varies by execution context:**

Context	Clocks
COG execution	11…18
Hub execution	20…40
COG with interrupts	11…26
Hub with interrupts	20…70

Related: CALLA, RET, RETB

Explanation:

RETA returns from a subroutine by reading a Hub **LONG** from –PTRA. PTRA is pre-decremented by 4 bytes, then a long is read from that address. The program counter is restored from L[19:0].

If the WC or WCZ effect is specified, the C flag is restored from L[31].

If the WZ or WCZ effect is specified, the Z flag is restored from L[30].

RETA is paired with **CALLA** for implementing software stacks in Hub memory, enabling deep **CALL** nesting beyond the 8-level hardware stack limit.

RETB

Return Via PTRB Stack
Branching and Flow Control - Returns from subroutine using PTRB as software stack pointer.

RETB {WC|WZ|WCZ}

Result: The program counter, C flag, and Z flag are restored from Hub memory at –PTRB.

- WC, WZ, or WCZ are optional effects to restore flags from the stack.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ1	000000000	000101111	L[31]	L[30]	—	11…18 †

† **Timing varies by execution context:**

Context	Clocks
COG execution	11…18
Hub execution	20…40
COG with interrupts	11…26
Hub with interrupts	20…70

Related: CALLB, RET, RETA

Explanation:

RETB returns from a subroutine by reading a Hub **LONG** from –PTRB. PTRB is pre-decremented by 4 bytes, then a long is read from that address. The program counter is restored from L[19:0].

If the WC or WCZ effect is specified, the C flag is restored from L[31].

If the WZ or WCZ effect is specified, the Z flag is restored from L[30].

RETB is paired with **CALLB** for implementing software stacks in Hub memory, enabling deep **CALL** nesting beyond the 8-level hardware stack limit.

RETI0 / RETI1 / RETI2 / RETI3

Return From Interrupt

Interrupts - Returns from interrupt handler to interrupted location.

RETI0
RETI1
RETI2
RETI3

Result: Execution returns from the specified interrupt level to the interrupted location.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011001	110	111111111	111111111	—	—	—	4 (COG), 13…20 (Hub)
EEEE	1011001	110	111111111	111110101	—	—	—	4 (COG), 13…20 (Hub)
EEEE	1011001	110	111111111	111110011	—	—	—	4 (COG), 13…20 (Hub)
EEEE	1011001	110	111111111	111110001	—	—	—	4 (COG), 13…20 (Hub)

Related: RESI0/1/2/3, SETINT1/2/3, NIXINT1/2/3

Explanation:

RETI0, RETI1, RETI2, and RETI3 return from their respective interrupt handlers. Each instruction is functionally equivalent to a **CALLD** instruction that restores the program counter, C flag, and Z flag from the corresponding interrupt return address registers.

The P2 provides four interrupt levels (INT0-INT3), with INT0 being the lowest priority and INT3 being the highest. Each RETI instruction completes its interrupt handler and resumes normal execution at the point where the interrupt occurred.

REV

Reverse Bits

Arithmetic Operations - Reverses all 32 bits in a register.

REV *Dest*

Result: The 32-bit pattern in Dest is reversed (bits 31:0 become bits 0:31).

- Dest is the register containing the bit pattern to reverse.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001101001	—	—	D	2

Related: ROL, ROR, ZEROX

Explanation:

REV performs a complete bitwise reverse of the value in Dest, storing the result back into Dest. Bit 31 becomes bit 0, bit 30 becomes bit 1, and so on through bit 0 becoming bit 31. The operation takes 2 cycles and does not affect any flags.

This instruction is useful for processing binary data in different MSB/LSB order than it is transmitted with, such as serial protocols that send least-significant bit first but need processing in most-significant bit first order. It is also used in bit-reversal algorithms for FFT operations.

RFBYTE

Read **BYTE** Via FIFO
Hub Memory Access - Reads a zero-extended **BYTE** from the **RDFAST** FIFO.

RFBYTE *Dest* **{WC|WZ|WCZ}**

Result: A zero-extended byte from the FIFO is loaded into Dest.

- Dest is the register to receive the byte value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000010000	MSB of byte	result == 0	D	2

Related: RDFAST, RFWORD, RFLONG, RFVAR

Explanation:

RFBYTE is used after **RDFAST** to read zero-extended bytes from the FIFO. The byte is loaded into Dest with bits 31:8 cleared to 0.

If the WC or WCZ effect is specified, C is set to the MSB of the byte.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

The operation takes 2 cycles when the FIFO has data available. The FIFO is automatically refilled in the background by the **RDFAST** operation.

RFLONG

Read **LONG** Via FIFO

Hub Memory Access - Reads a 32-bit **LONG** from the **RDFAST** FIFO.

RFLONG *Dest* **{WC|WZ|WCZ}**

Result: A long from the FIFO is loaded into Dest.

- Dest is the register to receive the long value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000010010	MSB of long	result == 0	D	2

Related: RDFAST, RFBYTE, RFWORD, RFVAR

Explanation:

RFLONG is used after **RDFAST** to read longs from the FIFO.

If the WC or WCZ effect is specified, C is set to the MSB of the long.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

The operation takes 2 cycles when the FIFO has data available. The FIFO is automatically refilled in the background by the **RDFAST** operation.

RFVAR

Read Variable Via FIFO

Hub Memory Access - Reads a zero-extended 1-4 byte value from the **RDFAST** FIFO.

RFVAR *Dest* **{WC|WZ|WCZ}**

Result: A zero-extended 1-4 byte value from the FIFO is loaded into Dest.

- Dest is the register to receive the variable-length value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000010011	0	result == 0	D	2

Related: RDFAST, RFBYTE, RFVARS

Explanation:

RFVAR is used after **RDFAST** to read variable-length values (1-4 bytes) from the FIFO with zero extension. The value is loaded into Dest with upper bits cleared to 0.

If the WC or WCZ effect is specified, C is always cleared to 0.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

The length of each value read is determined by the streamer configuration set up before the **RDFAST** operation.

RFVARS

Read Signed Variable Via FIFO
Hub Memory Access - Reads a sign-extended 1-4 byte value from the **RDFAST** FIFO.

RFVARS *Dest* **{WC|WZ|WCZ}**

Result: A sign-extended 1-4 byte value from the FIFO is loaded into Dest.

- Dest is the register to receive the sign-extended value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000010100	MSB of value	result == 0	D	2

Related: RDFAST, RFVAR, RFBYTE

Explanation:

RFVARS is used after **RDFAST** to read variable-length values (1-4 bytes) from the FIFO with sign extension. The value is loaded into Dest with upper bits set according to the MSB of the value (sign extension).

If the WC or WCZ effect is specified, C is set to the MSB of the value.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

RFWORD

Read **WORD** Via FIFO
Hub Memory Access - Reads a zero-extended **WORD** from the **RDFAST** FIFO.

RFWORD *Dest* **{WC|WZ|WCZ}**

Result: A zero-extended word from the FIFO is loaded into Dest.

- Dest is the register to receive the word value.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	DDDDDDDDD	000010001	MSB of word	result == 0	D	2

Related: RDFAST, RFBYTE, RFLONG, RFVAR

Explanation:

RFWORD is used after **RDFAST** to read zero-extended words from the FIFO. The word is loaded into Dest with bits 31:16 cleared to 0.

If the WC or WCZ effect is specified, C is set to the MSB of the word.

If the WZ or WCZ effect is specified, Z is set (1) if the result equals zero, or is cleared (0) if non-zero.

The operation takes 2 cycles when the FIFO has data available.

RGBEXP

Expand RGB Color
Color Space and Pixel Operations - Expands 5:6:5 RGB color to 8:8:8 format.

RGBEXP *Dest*

Result: The 5:6:5 RGB value in Dest[15:0] is expanded into 8:8:8 format in Dest[31:8].

- Dest contains 5:6:5 RGB in Dest[15:0], receives 8:8:8 RGB in Dest[31:8].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100111	—	—	D	2

Related: RGBSQZ

Explanation:

RGBEXP expands a compact 5:6:5 RGB color value (commonly used in 16-bit color displays) into full 8:8:8 RGB format (24-bit true color). The input 5:6:5 value is in Dest[15:0] with 5 bits red, 6 bits green, and 5 bits blue. The output 8:8:8 value is placed in Dest[31:8] with 8 bits each for red, green, and blue. The expansion replicates the most significant bits into the lower bits to maintain color accuracy.

This instruction is useful when converting between 16-bit and 24-bit color formats for graphics processing.

RGBSQZ

Squeeze RGB Color
Color Space and Pixel Operations - Compresses 8:8:8 RGB color to 5:6:5 format.

RGBSQZ *Dest*

Result: The 8:8:8 RGB value in Dest[31:8] is compressed into 5:6:5 format in Dest[15:0].

- Dest contains 8:8:8 RGB in Dest[31:8], receives 5:6:5 RGB in Dest[15:0].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100110	—	—	D	2

Related: RGBEXP

Explanation:

RGBSQZ compresses a full 8:8:8 RGB color value (24-bit true color) into compact 5:6:5 format (16-bit color). The input 8:8:8 value is in Dest[31:8] with 8 bits each for red, green, and blue. The output 5:6:5 value is placed in Dest[15:0] with 5 bits red, 6 bits green, and 5 bits blue. The compression keeps the most significant bits of each color channel.

This instruction is useful when converting from 24-bit to 16-bit color formats for display output.

ROL

Rotate Left

Arithmetic Operations - Rotates bits left, wrapping MSBs to LSBs.

ROL *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are rotated left by Src positions; departing MSBs are moved into LSBs.

- Dest is the register containing the value to rotate left.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to rotate.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000001	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[31].

Related: ROR, RCL, RCR, SHL

Explanation:

ROL rotates Dest's binary value left by Src places (0-31 bits). All MSBs rotated out are moved into the new LSBs.

If the WC or WCZ effect is specified, the C flag is updated to the value of the last bit rotated out if Src is 1-31, or to Dest[31] if Src is 0.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if non-zero. Since no bits are lost by this operation, the result will only be zero if Dest started at zero.

Rotation is useful for bit manipulation, circular buffers, hash functions, and cryptographic operations.

ROLBYTE

Rotate **BYTE** Left Into Register

Arithmetic Operations - Rotates a byte from source into destination register.

ROLBYTE *Dest, {#}Src, #N*

ROLBYTE *Dest*

Result: Byte N (0-3) of Src, or a byte from a source described by prior ALTGB instruction, is rotated left into Dest.

- Dest is the register into which the byte is rotated.
- Src is a register, 9-bit literal, or 32-bit augmented literal containing the target **BYTE**.
- N is a 2-bit literal (0-3) identifying the byte position in Src.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001000	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001000	000	DDDDDDDDD	000000000	—	—	D	2

Related: ROLNIB, ROLWORD, GETBYTE, SETBYTE, ALTGB

Explanation:

ROLBYTE reads the byte identified by N (0-3) from Src, or a byte from the source described by a prior **ALTGB** instruction, and rotates it left into Dest. **ROLBYTE** achieves the same effect as two instructions: an 8-bit **SHL** followed by **SETBYTE** into **BYTE** 0.

The second syntax form is intended for use after an **ALTGB** instruction in a loop to iteratively read a series of byte values within contiguous **LONG** registers.

ROLNIB

Rotate Nibble Left Into Register
Arithmetic Operations - Rotates a nibble from source into destination register.

ROLNIB *Dest, {#}Src, #N*

ROLNIB *Dest*

Result: Nibble N (0-7) of Src, or a nibble from a source described by prior ALTGN instruction, is rotated left into Dest.

- Dest is the register into which the nibble is rotated.
- Src is a register, 9-bit literal, or 32-bit augmented literal containing the target nibble.
- N is a 3-bit literal (0-7) identifying the nibble position in Src.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	100010N	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1000100	000	DDDDDDDDD	000000000	—	—	D	2

Related: ROLBYTE, ROLWORD, GETNIB, SETNIB, ALTGN

Explanation:

ROLNIB reads the nibble identified by N (0-7) from Src, or a nibble from the source described by a prior **ALTGN** instruction, and rotates it left into Dest. **ROLNIB** achieves the same effect as two instructions: a 4-bit **SHL** followed by **SETNIB** into nibble 0.

The second syntax form is intended for use after an **ALTGN** instruction in a loop to iteratively read a series of nibble values within contiguous **LONG** registers.

ROLWORD

Rotate **WORD** Left Into Register
Arithmetic Operations - Rotates a word from source into destination register.

ROLWORD *Dest, {#}Src, #N*

ROLWORD *Dest*

Result: Word N (0-1) of Src, or a word from a source described by prior ALTGW instruction, is rotated left into Dest.

- Dest is the register into which the word is rotated.
- Src is a register, 9-bit literal, or 32-bit augmented literal containing the target **WORD**.

- N is a 1-bit literal (0-1) identifying the word position in Src.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001010	0NI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001010	000	DDDDDDDDD	000000000	—	—	D	2

Related: ROLBYTE, ROLNIB, GETWORD, SETWORD, ALTGW

Explanation:

ROLWORD reads the word identified by N (0-1) from Src, or a word from the source described by a prior **ALTGW** instruction, and rotates it left into Dest. **ROLWORD** achieves the same effect as two instructions: a 16-bit **SHL** followed by **SETWORD** into **WORD** 0.

The second syntax form is intended for use after an **ALTGW** instruction in a loop to iteratively read a series of word values within contiguous **LONG** registers.

ROR

Rotate Right
Arithmetic Operations - Rotates bits right, wrapping LSBs to MSBs.

ROR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are rotated right by Src positions; departing LSBs are moved into MSBs.

- Dest is the register containing the value to rotate right.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to rotate.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000000	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[0].

Related: ROL, RCL, RCR, SHR

Explanation:

ROR rotates Dest's binary value right by Src places (0-31 bits). All LSBs rotated out are moved into the new MSBs.

If the WC or WCZ effect is specified, the C flag is updated to the value of the last bit rotated out if Src is 1-31, or to Dest[0] if Src is 0.

If the WZ or WCZ effect is specified, the Z flag is set (1) if the result equals zero, or is cleared (0) if non-zero. Since no bits are lost by this operation, the result will only be zero if Dest started at zero.

Rotation is useful for bit manipulation, circular buffers, hash functions, and cryptographic operations.

RQPIN

Read Smart Pin Without Acknowledge

Pin I/O and Smart Pins - Reads Smart Pin result without clearing the ready flag.

RQPIN *Dest, {#}Src* **{WC}**

Result: Smart Pin Src[5:0] result is loaded into Dest without clearing the pin's ready flag.

- Dest is the register to receive the pin result.
- Src is a register or literal identifying the pin number (Src[5:0]) to read from.
- WC is an optional effect to write the modal result to C.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010100	C0I	DDDDDDDDD	SSSSSSSSS	Modal result	—	D	2

Related: RDPIN, WRPIN, WXPIN, WYPIN

Explanation:

RQPIN reads the result value from the specified Smart Pin without acknowledging the pin. Unlike **RDPIN**, this instruction does not clear the pin's "ready" flag, allowing the same result to be read multiple times or checked before being consumed.

If the WC effect is specified, the C flag is set to the modal result, which provides mode-specific status information.

This instruction is useful when you need to check a pin's result value without consuming it, such as polling for completion before actually processing the result.

Instructions: S

This section contains all PASM2 instructions beginning with the letter S.

SAL

Shift Arithmetic Left
Arithmetic Operations - Shifts bits left, extending the original LSB into new rightmost bits.

SAL *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted left by Src bits, extending Dest[0] into new rightmost bits.

- Dest is a register containing the value to arithmetically left shift.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to shift.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000111	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[31].

Related: SAR, SHL, SHR

Explanation:

SAL shifts the destination's binary value left by the source number of places (0-31 bits) and sets the new LSBs to that of the original Dest[0]. **SAL** is the complement of **SAR** for bit streams but not for math operations. For swift 32-bit integer multiplication by a power-of-two, use **SHL** instead.

```
1         SAL     data, #4       ' Shift left 4 bits, extending LSB
```

SAR

Shift Arithmetic Right
Arithmetic Operations - Shifts bits right, preserving the sign bit for signed division.

SAR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted right by Src bits, extending Dest[31] (the sign bit) into new leftmost bits.

- Dest is a register containing the value to arithmetically right shift.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to shift.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000110	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[0].

Related: SAL, SHL, SHR

Explanation:

SAR shifts the destination's binary value right by the source number of places (0-31 bits) and sets the new MSBs to that of the original Dest[31], preserving the sign of a signed integer. This is useful for bit stream manipulation and for swift division. It is similar to **SHR** for swift division by a power-of-two, but is safe for both signed and unsigned integers.

```
1         SAR     value, #3      ' Divide signed value by 8
```

SCA

Scale

Arithmetic Operations - Scales unsigned 16-bit values by multiplying and right-shifting.

SCA *Dest, {#}Src* **{WZ}**

Result: The upper 16 bits of the unsigned product from the 16-bit Dest and Src multiplication is substituted as the next instruction's S value.

- Dest is a register containing the 16-bit value to multiply with Src.
- Src is a register, 9-bit literal, or 16-bit augmented literal to multiply with Dest.
- WZ is an optional effect to update the Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010001	0ZI	DDDDDDDDD	SSSSSSSSS	—	Product = 0	—	2

Related: SCAS

Explanation:

SCA multiplies the lower 16 bits of each of Dest and Src together, right shifts the 32-bit product by 16 (to scale down the result), and substitutes this value as the next instruction's S value. This is useful for creating scaled unsigned 16-bit values for subsequent operations.

The instruction following **SCA** is shielded from interrupts. This ensures the scaled value is correctly applied to the next instruction's S operand before any interrupt can occur.

```
1         SCA     factor, ##$8000  ' Scale by 0.5 (32768/65536)
2         ADD     result, #0      ' Add scaled value
```

SCAS

Scale Signed

Arithmetic Operations - Scales signed 16-bit values by multiplying and right-shifting.

SCAS *Dest, {#}Src* **{WZ}**

Result: The upper 18 bits of the signed product from the 16-bit Dest and Src multiplication is substituted as the next instruction's S value.

- Dest is a register containing the signed 16-bit value to multiply with Src.
- Src is a register, 9-bit literal, or signed 16-bit augmented literal to multiply with Dest.
- WZ is an optional effect to update the Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010001	1ZI	DDDDDDDDD	SSSSSSSSS	—	result == 0	—	2

Related: SCA

Explanation:

SCAS multiplies the lower signed 16 bits of each of Dest and Src together, right shifts the 32-bit product by 14 (to scale down the result), and substitutes this value as the next instruction's S value. This is useful for creating scaled signed values for subsequent operations.

The instruction following **SCAS** is shielded from interrupts. This ensures the scaled value is correctly applied to the next instruction's S operand before any interrupt can occur.

SETBYTE

Set **BYTE**

Arithmetic Operations - Writes an 8-bit value to a specific **BYTE** position within a register.

SETBYTE *Dest, {#}Src, #N*

SETBYTE *{#}Src*

Result: Src[7:0] is written to byte N (0-3) of Dest, or to another register byte described by prior ALTSB instruction.

- Dest is a register in which to modify a byte.
- Src is a register or 8-bit literal whose bits [7:0] will be stored in the designated location.
- N is a 2-bit literal (0-3) identifying the byte of Dest to modify.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1000110	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1000110	00I	000000000	SSSSSSSSS	—	—	D*	2

*Dest and **BYTE** ID specified by prior **ALTSB** instruction.

Related: ALTSB, SETNIB, SETWORD, GETBYTE

Explanation:

SETBYTE stores Src[7:0] into the byte identified by N within Dest, or the byte and register described by a prior ALTSB instruction. No other bits are modified. N (0-3) identifies a value's individual bytes by position in least-significant byte order. The second syntax is intended for use after an ALTSB instruction in a loop to iteratively affect a series of byte values within contiguous long registers.

```
1         SETBYTE data, #$FF, #2  ' Set byte 2 of data to $FF
```

SETCFRQ

Set Colorspace Converter Frequency
Color Space and Pixel Operations - Configures the frequency parameter for colorspace conversion hardware.

SETCFRQ *{#}Dest*

Result: The colorspace converter CFRQ parameter is set to Dest[31:0].

- Dest is a register or literal value (0-511) to set as CFRQ parameter.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111011	—	—	—	2

Related: SETCI, SETCMOD, SETCQ, SETCY

Explanation:

Sets the colorspace converter CFRQ parameter to the value in Dest. This instruction configures the frequency parameter for the colorspace conversion hardware.

SETCI

Set Colorspace Converter CI
Color Space and Pixel Operations - Configures the CI parameter for colorspace conversion hardware.

SETCI *{#}Dest*

Result: The colorspace converter CI parameter is set to Dest[31:0].

- Dest is a register or literal value (0-511) to set as CI parameter.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111001	—	—	—	2

Related: SETCFRQ, SETCMOD, SETCQ, SETCY

Explanation:

Sets the colorspace converter CI parameter to the value in Dest. This instruction configures the CI parameter for the colorspace conversion hardware.

SETCMOD

Set Colorspace Converter Mode

Color Space and Pixel Operations - Configures the mode parameter for colorspace conversion hardware.

SETCMOD *{#}Dest*

Result: The colorspace converter CMOD parameter is set to Dest[8:0].

- Dest is a register or literal value (0-511) to set as CMOD parameter.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111100	—	—	—	2

Related: SETCFRQ, SETCI, SETCQ, SETCY

Explanation:

Sets the colorspace converter CMOD parameter to Dest[8:0]. This instruction configures the mode parameter for the colorspace conversion hardware.

SETCQ

Set Colorspace Converter CQ

Color Space and Pixel Operations - Configures the CQ parameter for colorspace conversion hardware.

SETCQ *{#}Dest*

Result: The colorspace converter CQ parameter is set to Dest[31:0].

- Dest is a register or literal value (0-511) to set as CQ parameter.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111010	—	—	—	2

Related: SETCFRQ, SETCI, SETCMOD, SETCY

Explanation:

Sets the colorspace converter CQ parameter to the value in Dest. This instruction configures the CQ parameter for the colorspace conversion hardware.

SETCY

Set Colorspace Converter CY

Color Space and Pixel Operations - Configures the CY parameter for colorspace conversion hardware.

SETCY *{#}Dest*

Result: The colorspace converter CY parameter is set to Dest[31:0].

- Dest is a register or literal value (0-511) to set as CY parameter.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111000	—	—	—	2

Related: SETCFRQ, SETCI, SETCMOD, SETCQ

Explanation:

Sets the colorspace converter CY parameter to the value in Dest. This instruction configures the CY parameter for the colorspace conversion hardware.

SETD

Set Destination Field
Register Indirection - Sets the D field of a template for use with **ALTI** instruction.

SETD *Dest, {#}Src*

Result: The D field [17:9] of template Dest is set to Src[8:0].

- Dest is a register whose 32-bit value is a template for use with an **ALTI** instruction.
- Src is a register or 9-bit literal whose value (Src[8:0]) is copied to the D field of Dest.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001101	10I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: SETS, SETR, ALTI

Explanation:

SETD copies Src[8:0] to the D field of the template Dest to be used with an **ALTI** instruction. Bits outside the D field remain unaffected. The D field holds the address of a register (or sometimes a literal value) for the instruction to use as its destination value, and usually as its result destination, during its execution.

SETD can also be used in self-modifying register RAM code. Unlike with ALTx instructions, when used this way, field value modification occurs in the program code itself (not the instruction pipeline); code is altered, values persist. Due to the instruction pipeline nature, after modifying a code register, it is necessary to elapse at least two instructions before executing the modified register.

SETDACS

Set DACs
Pin I/O and Smart Pins - Sets all four DAC channels simultaneously from a single register.

SETDACS *{#}Dest*

Result: DAC3 = Dest[31:24], DAC2 = Dest[23:16], DAC1 = Dest[15:8], DAC0 = Dest[7:0].

- Dest is a register or literal value (0-511) containing four 8-bit DAC values.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000011100	—	—	—	2

Explanation:

Sets all four DAC channels simultaneously from the four bytes in Dest. DAC3 receives bits [31:24], DAC2 receives bits [23:16], DAC1 receives bits [15:8], and DAC0 receives bits [7:0].

SETINT1 / SETINT2 / SETINT3

Set Interrupt Source (1, 2, Or 3)

Interrupts - Configures which event triggers the specified interrupt level.

SETINT1 *{#}Dest*

SETINT2 *{#}Dest*

SETINT3 *{#}Dest*

Result: The specified interrupt source (INT1, INT2, or INT3) is set to Dest[3:0].

- Dest is a register or literal value (0-511) containing interrupt source in bits [3:0].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000100101	—	—	—	2
EEEE	1101011	00L	DDDDDDDDD	000100110	—	—	—	2
EEEE	1101011	00L	DDDDDDDDD	000100111	—	—	—	2

Related: NIXINT1/2/3, TRGINT1/2/3, RETI0/1/2/3, RESI0/1/2/3

Explanation:

SETINT1, SETINT2, and SETINT3 configure which event will trigger their respective interrupt levels. The interrupt source is specified in Dest[3:0].

The P2 provides three configurable interrupt levels (INT1-INT3), each of which can be independently configured to respond to different event sources.

SETLUTS

Set LUT Sharing

Lookup Table - Enables or disables LUT sharing between adjacent cog pairs.

SETLUTS *{#}Dest*

Result: If Dest[0] = 1, LUT sharing is enabled where LUT writes within the adjacent odd/even companion cog are copied to this cog's LUT.

- Dest is a register or literal value (0-511) with enable bit in Dest[0].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110111	—	—	—	2

Related: RDLUT, WRLUT

Explanation:

Enables or disables LUT sharing based on Dest[0]. When enabled (Dest[0] = 1), LUT writes within the adjacent odd/even companion cog are automatically copied to this cog's LUT, allowing cogs to share lookup table data.

SETNIB

Set Nibble

Arithmetic Operations - Writes a 4-bit value to a specific nibble position within a register.

SETNIB *Dest, {#}Src, #N*

SETNIB *{#}Src*

Result: Src[3:0] is written to nibble N (0-7) of Dest, or to another register nibble described by prior ALTSN instruction.

- Dest is a register in which to modify a nibble.
- Src is a register or 4-bit literal whose bits [3:0] will be stored in the designated location.
- N is a 3-bit literal (0-7) identifying the nibble of Dest to modify.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	100000N	NNI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1000000	00I	000000000	SSSSSSSSS	—	—	D*	2

*Dest and nibble ID specified by prior **ALTSN** instruction.

Related: ALTSN, SETBYTE, SETWORD, GETNIB

Explanation:

SETNIB stores Src[3:0] into the nibble identified by N within Dest, or the nibble and register described by a prior ALTSN instruction. No other bits are modified. N (0-7) identifies a value's individual nibbles by position in least-significant nibble order. The second syntax is intended for use after an ALTSN instruction in a loop to iteratively affect a series of nibble values within contiguous long registers.

```
1         SETNIB  data, #$A, #5   ' Set nibble 5 of data to $A
```

SETPAT

Set Pin Pattern

Pin I/O and Smart Pins - Configures pin pattern matching for PAT event detection.

SETPAT *{#}Dest, {#}Src*

Result: Pin pattern for PAT event is configured. C selects INA/INB, Z selects =/!=, Dest provides mask value, Src provides match value.

- Dest is a register or immediate containing mask value.
- Src is a register or immediate containing match value.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011111	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: POLLPAT, WAITPAT

Explanation:

Sets pin pattern for PAT event detection. The C flag selects INA or INB for monitoring, the Z flag selects equality (=) or inequality (!=) matching, Dest provides the mask value to select which pins to monitor, and Src provides the match value to compare against.

SETPIV

Set Pixel Blend Factor
Color Space and Pixel Operations - Sets the blend factor for **BLNPIX** and **MIXPIX** pixel operations.

SETPIV *{#}Dest*

Result: BLNPIX/MIXPIX blend factor is set to Dest[7:0].

- Dest is a register or literal value (0-511) containing 8-bit blend factor in bits [7:0].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111101	—	—	—	2

Related: SETPIX, BLNPIX, MIXPIX

Explanation:

Sets the blend factor for **BLNPIX** and **MIXPIX** operations to Dest[7:0]. This controls the blending ratio for pixel mixing operations.

SETPIX

Set Pixel Mixer Mode
Color Space and Pixel Operations - Configures the **MIXPIX** operating mode for pixel combining.

SETPIX *{#}Dest*

Result: MIXPIX mode is set to Dest[5:0].

- Dest is a register or literal value (0-511) containing 6-bit mode in bits [5:0].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000111110	—	—	—	2

Related: SETPIV, MIXPIX

Explanation:

Sets the **MIXPIX** operating mode to Dest[5:0]. This configures how the pixel mixer combines pixel values.

SETQ

Set Q Register

Hub Memory Access - Loads the Q register for block transfers and multi-parameter instructions.

SETQ *{#}Dest*

Result: Q register is set to Dest.

- Dest is a register or literal value (0-511) to load into Q.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000101000	—	—	—	2

Related: SETQ2, RDLONG, WRLONG

Explanation:

Sets Q register to Dest. Use before **RDLONG**/WRLONG/WMLONG to set block transfer count. Also used before **MUXQ**/COGINIT/QDIV/QFRAC/QROTATE/WAITxxx instructions to provide additional parameters.

```
1         SETQ    #16-1          ' Set up for 16-long block transfer
2         RDLONG  buffer, ptra   ' Read 16 longs from hub
```

Pitfall (Silicon Bug): Intervening ALTx, **AUGS**, or **AUGD** instructions between **SETQ** and **RDLONG**/WRLONG/WMLONG cancel the block-size PTRx delta calculation. The correct number of longs transfers, but PTRx advances by only a single-long delta instead of the full block size. Avoid placing any ALTx or AUGx instruction between **SETQ** and the block transfer instruction, or manually adjust PTRx afterward.

SETQ2

Set Q For LUT Transfers

Hub Memory Access - Loads the Q register for LUT-to-hub block transfers.

SETQ2 *{#}Dest*

Result: Q register is set to Dest for LUT block transfers.

- Dest is a register or literal value (0-511) to load into Q.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000101001	—	—	—	2

Related: SETQ, RDLONG, WRLONG, RDLUT, WRLUT

Explanation:

Sets Q register to Dest. Use before **RDLONG**/WRLONG/WMLONG to set LUT block transfer. **SETQ**2 enables block transfers to/from LUT RAM instead of COG RAM: **SETQ**2 + **RDLONG** performs block read from HUB to LUT, while **SETQ**2 + **WRLONG** performs block write from LUT to HUB. This is essential for fast bulk data movement for lookup tables, waveform tables, and large datasets.

```
1         SETQ2   #256-1          ' Set up for 256-long LUT transfer
2         RDLONG  0, ptra         ' Read 256 longs from hub into LUT
```

Pitfall (Silicon Bug): Same as **SETQ**—intervening ALTx, **AUGS**, or **AUGD** instructions between **SETQ**2 and **RDLONG**/WRLONG/WMLONG cancel the block-size PTRx delta calculation. The data transfers correctly, but PTRx advances by only a single-long delta instead of the full block size. Avoid placing any ALTx or AUGx instruction between **SETQ**2 and the block transfer instruction.

SETR

Set Result Field

Register Indirection - Sets the Result field of a template for use with **ALTI** instruction.

SETR *Dest, {#}Src*

Result: The Result field [27:19] of template Dest is set to Src[8:0].

- Dest is a register whose 32-bit value is a template for use with an **ALTI** instruction.
- Src is a register or 9-bit literal whose value (Src[8:0]) is copied to the Result field of Dest.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001101	01I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: SETD, SETS, ALTI

Explanation:

SETR copies Src[8:0] to the Result field of the template Dest to be used with an **ALTI** instruction. Bits outside the Result field remain unaffected. The Result field does not exist in instruction opcodes, but takes its value from the D field, holding the address of a register for the instruction to use as its result destination upon execution.

SETR can also be used in self-modifying register RAM code, though it affects the Instr field and upper two bits of the FX field rather than a non-existent Register field. Unlike with ALTx instructions, when used this way, field value modification occurs in the program code itself (not the instruction pipeline); code is altered, values persist. Due to the instruction pipeline nature, after modifying a code register, it is necessary to elapse at least two instructions before executing the modified register.

SETS

Set Source Field

Register Indirection - Sets the S field of a template for use with **ALTI** instruction.

SETS *Dest, {#}Src*

Result: The S field [8:0] of template Dest is set to Src[8:0].

- Dest is a register whose 32-bit value is a template for use with an **ALTI** instruction.
- Src is a register or 9-bit literal whose value (Src[8:0]) is copied to the S field of Dest.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001101	11I	DDDDDDDDD	SSSSSSSSS	—	—	D	2

Related: SETD, SETR, ALTI

Explanation:

SETS copies Src[8:0] to the S field of the template Dest to be used with an **ALTI** instruction. Bits outside the S field remain unaffected. The S field holds the address of a register or literal value for an instruction to use as its source value during its execution.

SETS can also be used in self-modifying register RAM code. Unlike with ALTx instructions, when used this way, field value modification occurs in the program code itself (not the instruction pipeline); code is altered, values persist. Due to the instruction pipeline nature, after modifying a code register, it is necessary to elapse at least two instructions before executing the modified register.

SETSCP

Set Oscilloscope
Pin I/O and Smart Pins - Configures the four-channel hardware oscilloscope for debugging.

SETSCP *{#}Dest*

Result: Four-channel oscilloscope enable is set to Dest[6] and input pin base is set to Dest[5:2].

- Dest is a register or literal value (0-511) containing enable bit [6] and pin base [5:2].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	001110000	—	—	—	2

Explanation:

Sets the four-channel oscilloscope enable to Dest[6] and sets the input pin base to Dest[5:2]. This configures the hardware oscilloscope feature for debugging and signal monitoring.

SETSE1 / SETSE2 / SETSE3 / SETSE4

Set Selectable Event (1, 2, 3, Or 4)
Events and Timing - Configures the detection criteria for selectable events.

SETSE1 *{#}Dest*

SETSE2 *{#}Dest*

SETSE3 *{#}Dest*

SETSE4 *{#}Dest*

Result: The specified selectable event configuration (SE1-SE4) is set to Dest[8:0].

- Dest is a register or literal value (0-511) containing event configuration in bits [8:0].

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000100000	—	—	—	2
EEEE	1101011	00L	DDDDDDDDD	000100001	—	—	—	2
EEEE	1101011	00L	DDDDDDDDD	000100010	—	—	—	2
EEEE	1101011	00L	DDDDDDDDD	000100011	—	—	—	2

Related: POLLSE1/2/3/4, WAITSE1/2/3/4, JSE1/2/3/4, JNSE1/2/3/4

Explanation:

SETSE1, SETSE2, SETSE3, and SETSE4 configure their respective selectable event's detection criteria. The Dest[8:0] operand specifies which condition will trigger the event. Configuring SETSEn also clears the corresponding SEn event flag.

The P2 provides four independent selectable events, each of which can be configured to detect various conditions including pin states, hub operations, CORDIC completion, and other system events. Once configured, these events can be polled with POLLSEn, waited upon with WAITSEn, or used for conditional jumps with JSEn and JNSEn.

SETWORD

Set **WORD**

Arithmetic Operations - Writes a 16-bit value to a specific **WORD** position within a register.

SETWORD *Dest, {#}Src, #N*

SETWORD *{#}Src*

Result: Src[15:0] is written to word N (0-1) of Dest, or to another register word described by prior ALTSW instruction.

- Dest is a register in which to modify a word.
- Src is a register, 9-bit literal, or 16-bit augmented literal whose bits [15:0] will be stored in the designated location.
- N is a 1-bit literal (0-1) identifying the word of Dest to modify.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1001001	0NI	DDDDDDDDD	SSSSSSSSS	—	—	D	2
EEEE	1001001	00I	000000000	SSSSSSSSS	—	—	D*	2

*Dest and **WORD** ID specified by prior **ALTSW** instruction.

Related: ALTSW, SETNIB, SETBYTE, GETWORD

Explanation:

SETWORD stores Src[15:0] into the word identified by N within Dest, or the word and register described by a prior ALTSW instruction. No other bits are modified. N (0-1) identifies a value's individual words by position in least-significant word order. The second syntax is intended for use after an ALTSW instruction in a loop to iteratively affect a series of word values within contiguous long registers.

```
1           SETWORD data, ##$ABCD, #1   ' Set high word of data to $ABCD
```

SETXFRQ

Set Streamer Frequency
Streamer - Sets the NCO frequency that controls streamer data output rate.

SETXFRQ *{#}Dest*

Result: Streamer NCO frequency is set to Dest.

- Dest is a register or literal value (0-511) containing frequency value.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000011101	—	—	—	2

Related: XINIT, XCONT

Explanation:

Sets the streamer NCO (Numerically Controlled Oscillator) frequency to Dest. This controls the frequency at which the streamer outputs data.

SEUSSF

Seuss Forward
Arithmetic Operations - Transforms bits by relocating and inverting for pseudo-random scrambling.

SEUSSF *Dest*

Result: Dest is transformed by relocating and periodically inverting bits. Returns to original value on 32nd iteration.

- Dest is a register to transform.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100100	—	—	D	2

Related: SEUSSR

Explanation:

Relocates and periodically inverts bits within Dest using a forward pattern. The transformation returns to the original value after 32 iterations. This is useful for pseudo-random bit scrambling and data obfuscation.

SEUSSR

Seuss Reverse
Arithmetic Operations - Reverse transforms bits for pseudo-random scrambling, inverse of **SEUSSF**.

SEUSSR *Dest*

Result: Dest is transformed by relocating and periodically inverting bits. Returns to original value on 32nd iteration.

- Dest is a register to transform.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100101	—	—	D	2

Related: SEUSSF

Explanation:

Relocates and periodically inverts bits within Dest using a reverse pattern. The transformation returns to the original value after 32 iterations. This is useful for pseudo-random bit scrambling and data obfuscation, providing the inverse operation of **SEUSSF**.

SHL

Shift Left

Arithmetic Operations - Shifts bits left, inserting zeros for fast multiplication by powers of two.

SHL *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted left by Src bits, inserting zeros (0) as new rightmost bits.

- Dest is a register containing the value to left shift.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to shift.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000011	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[31].

Related: SHR, SAL, SAR, ROL

Explanation:

SHL shifts the destination's binary value left by the source number of places (0-31 bits) and sets the new LSBs to 0. This is useful for bit-stream manipulation as well as for swift multiplication; signed or unsigned 32-bit integer multiplication by a power-of-two. Care must be taken for power-of-two multiplications since upper bits shift through the MSB (sign bit), mangling large signed values.

```
1         SHL     value, #2      ' Multiply by 4
```

SHR

Shift Right

Arithmetic Operations - Shifts bits right, inserting zeros for fast unsigned division.

SHR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The bits of Dest are shifted right by Src bits, inserting zeros (0) as new leftmost bits.

- Dest is a register containing the value to right shift.
- Src is a register or 5-bit literal (0-31) specifying the number of bit positions to shift.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0000010	CZI	DDDDDDDDD	SSSSSSSSS	Last bit out†	result == 0	D	2

† If S[4:0] > 0, C receives the last bit shifted out. If S[4:0] = 0 (no shift), C receives D[0].

Related: SHL, SAR, ROR

Explanation:

SHR shifts the destination's binary value right by the source number of places (0-31 bits) and sets the new MSBs to 0. This is useful for bit-stream manipulation as well as for swift division; unsigned 32-bit integer division by a power-of-two. For similar division of a signed value, use **SAR** instead.

```
1         SHR     value, #3      ' Divide unsigned by 8
```

SIGNX

Sign Extend
Arithmetic Operations - Sign-extends a value above the specified bit position.

SIGNX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The Dest value is sign-extended above the bit indicated by Src and is stored in Dest. Optionally the C and Z flags are updated to the resulting MSB and zero status.

- Dest is a register containing the value to sign-extend above bit Src[4:0] and where the result is written.
- Src is a register or 9-bit literal whose value (lower 5 bits) identifies the bit of Dest to sign-extend beyond.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0111011	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2

Related: ZEROX

Explanation:

SIGNX fills the bits of Dest above the bit indicated by Src[4:0] with the value of that identified bit, i.e. sign-extending the value. This is handy when converting encoded or received signed values from a small bit width to a large bit width, i.e. 32 bits.

```
        SIGNX   value, #7       ' Sign-extend 8-bit value to 32 bits
```

SKIP

SKIP Instructions
Branching and Flow Control - Cancels subsequent instructions based on a bitmask pattern.

SKIP *{#}Dest*

Result: Subsequent instructions 0-31 are cancelled for each '1' bit in Dest[0]-Dest[31].

- Dest is a register or literal value (0-511) containing **SKIP** pattern bitmask.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110001	—	—	—	2

Related: SKIPF

Explanation:

Skips instructions based on Dest bitmask. Subsequent instructions 0-31 get cancelled for each '1' bit in Dest[0]-Dest[31]. Each set bit causes the corresponding sequential instruction to be cancelled (replaced with **NOP**).

```
        SKIP    #%10101         ' Skip instructions 0, 2, 4
        NOP                     ' Skipped (bit 0)
        ADD     x, #1           ' Executed (bit 1 = 0)
        NOP                     ' Skipped (bit 2)
```

SKIPF

SKIP Instructions Fast
Branching and Flow Control - Leaps over instructions based on a bitmask for faster skipping.

SKIPF *{#}Dest*

Result: Program counter leaps over cog/LUT instructions based on Dest bitmask.

- Dest is a register or literal value (0-511) containing **SKIP** pattern bitmask.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000110010	—	—	—	2

Related: SKIP

Explanation:

Like **SKIP**, but instead of cancelling instructions, the PC leaps over them. This provides faster execution when skipping multiple instructions, as the skipped instructions are never fetched or executed.

CRITICAL: COG/LUT Memory Only

SKIPF can ONLY leap over instructions when executing from **COG or LUT memory**. When **SKIPF** is executed from Hub memory, it automatically **reverts to SKIP behavior** (cancelling instructions in the pipeline instead of stepping over them). This is a hardware limitation—the Hub memory FIFO can only provide sequential instructions; random PC stepping requires the random-access capability of COG/LUT memory.

Best Practice: Use **SKIP** for code in Hub memory (ORGH sections), **SKIPF** for code in COG/LUT memory (ORG sections).

REP Compatibility: - **SKIP** is fully compatible with **REP**—cancellation maintains instruction counts - **SKIPF** works with **REP** ONLY if all **SKIP** patterns result in identical instruction counts - Recommendation: Use **SKIP** within **REP** blocks for predictable behavior

SPLITB

Split Bits To Bytes
Arithmetic Operations - Redistributes every 4th bit into separate bytes.

SPLITB *Dest*

Result: Dest = {Dest[31], Dest[27], Dest[23], Dest[19], …Dest[12], Dest[8], Dest[4], Dest[0]}.

- Dest is a register to transform.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100000	—	—	D	2

Related: SPLITW, MERGEB

Explanation:

Splits every 4th bit of Dest into bytes. The bits at positions 0, 4, 8, 12, 16, 20, 24, 28 become the new low **BYTE**, the bits at positions 1, 5, 9, 13, 17, 21, 25, 29 become the second **BYTE**, and so on. This is useful for bit reordering and data unpacking operations.

SPLITW

Split Bits To Words
Arithmetic Operations - Separates odd and even bits into separate words.

SPLITW *Dest*

Result: Dest = {Dest[31], Dest[29], Dest[27], Dest[25], …Dest[6], Dest[4], Dest[2], Dest[0]}.

- Dest is a register to transform.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001100010	—	—	D	2

Related: SPLITB, MERGEW

Explanation:

Splits odd and even bits of Dest into separate words. The even bits (0, 2, 4, ...30) become the low **WORD**, and the odd bits (1, 3, 5, ...31) become the high **WORD**. This is useful for bit reordering and data unpacking operations.

STALLI

Disallow Interrupts

Interrupts - Disables interrupt branching to protect critical code sections.

STALLI

Result: All future interrupts are disallowed.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	000100001	000100100	—	—	—	2

Related: ALLOWI

Explanation:

STALLI disables interrupt branching. **STALLI** is the complement of the **ALLOWI** instruction; both are used to protect short, vital sections of main code from timing jitter or state loss caused by asynchronous interrupt handling.

```
1         STALLI                ' Disable interrupts
2         ' Critical section...
3         ALLOWI                ' Re-enable interrupts
```

SUB

Subtract

Arithmetic Operations - Subtracts unsigned Src from unsigned Dest.

SUB *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Difference of unsigned Dest and unsigned Src is stored in Dest and optionally the C and Z flags are updated to the borrow and zero status.

- Dest is a register containing the value to subtract Src from, and where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is subtracted from Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001100	CZI	DDDDDDDDD	SSSSSSSSS	Borrow of (D - S)	result == 0	D	2

Related: SUBX, SUBS, SUBSX, SUBR, ADD

Explanation:

SUB subtracts the unsigned Src from the unsigned Dest and stores the result into the Dest register. To subtract unsigned multi-long values, use **SUB** followed by **SUBX** as described in Subtracting Two Multi-

Long Values. **SUB** and **SUBX** are also used in subtracting signed multi-long values with **SUBSX** ending the sequence.

```
1          SUB     count, #1 WZ    ' Decrement count, set Z if zero
```

SUBR

Subtract Reverse
Arithmetic Operations - Subtracts unsigned Dest from unsigned Src (reverse order).

SUBR *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Difference of unsigned Src and unsigned Dest is stored in Dest and optionally the C and Z flags are updated to the borrow and zero status.

- Dest is a register containing the value to subtract from Src, and where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is subtracted by Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0010110	CZI	DDDDDDDDD	SSSSSSSSS	Borrow of (S - D)	result == 0	D	2

Related: SUB

Explanation:

SUBR subtracts the unsigned Dest from the unsigned Src and stores the result into the Dest register. This is the reverse of the subtraction order of **SUB**, computing Src - Dest instead of Dest - Src.

SUBS

Subtract Signed
Arithmetic Operations - Subtracts signed Src from signed Dest.

SUBS *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Difference of signed Dest and signed Src is stored in Dest and optionally the C and Z flags are updated to the sign and zero status.

- Dest is a register containing the value to subtract Src from, and where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value is subtracted from Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001110	CZI	DDDDDDDDD	SSSSSSSSS	correct sign of (D - S)	result == 0	D	2

Related: SUB, SUBX, SUBSX

Explanation:

SUBS subtracts the signed Src from the signed Dest and stores the result into the Dest register. If Src is a 9-bit literal, its value is interpreted as positive (0-511; it is not sign-extended). Use ##Value (or insert a prior **AUGS** instruction) for a 32-bit signed value; negative or positive. To subtract signed multi-long values, use **SUB** (not **SUBS**) followed possibly by **SUBX**, and finally **SUBSX**.

SUBSX

Subtract Signed Extended

Arithmetic Operations - Subtracts signed Src plus C from signed Dest for multi-long operations.

SUBSX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Difference of signed Dest and signed Src (plus C) is stored in Dest and optionally the C and Z flags are updated to the extended sign and zero status.

- Dest is a register containing the value to subtract Src plus C from, and where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value plus C is subtracted from Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001111	CZI	DDDDDDDDD	SSSSSSSSS	correct sign of (D - (S + C))	Z AND (result == 0)	D	2

Related: SUB, SUBX, SUBS

Explanation:

SUBSX subtracts the signed value of Src plus C from the signed Dest and stores the result into the Dest register. The **SUBSX** instruction is used to perform signed multi-long (extended) subtraction, such as 64-bit subtraction. Use WC or WCZ on preceding **SUB** and **SUBX** instructions for proper final C flag. Use WZ or WCZ on preceding **SUB** and **SUBX** instructions for proper final Z flag. To subtract signed multi-long values, use **SUB** (not **SUBS**) followed possibly by **SUBX**, and finally **SUBSX**.

SUBX

Subtract Extended

Arithmetic Operations - Subtracts unsigned Src plus C from unsigned Dest for multi-long operations.

SUBX *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Difference of unsigned Dest and unsigned Src (plus C) is stored in Dest and optionally the C and Z flags are updated to the extended borrow and zero status.

- Dest is a register containing the value to subtract Src plus C from, and where the result is written.
- Src is a register, 9-bit literal, or 32-bit augmented literal whose value plus C is subtracted from Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0001101	CZI	DDDDDDDDD	SSSSSSSSS	Borrow of (D - (S + C))	Z AND (result == 0)	D	2

Related: SUB, SUBSX

Explanation:

SUBX subtracts the unsigned value of Src plus C from the unsigned Dest and stores the result into the Dest register. The **SUBX** instruction is used to perform unsigned multi-long (extended) subtraction, such as 64-bit subtraction. Use WC or WCZ on preceding **SUB** and **SUBX** instructions for proper final C flag. If C is set after the last **SUBX** in a multi-long subtraction, it indicates unsigned underflow. Use WZ or WCZ on preceding **SUB** and **SUBX** instructions for proper final Z flag. To subtract unsigned multi-long values, use **SUB** followed by one or more **SUBX** instructions.

SUMC / SUMNC / SUMZ / SUMNZ

Conditional Sum

Arithmetic Operations - Conditionally **ADDS** or subtracts based on flag state.

SUMC *Dest, {#}Src* **{WC|WZ|WCZ}**

SUMNC *Dest, {#}Src* **{WC|WZ|WCZ}**

SUMZ *Dest, {#}Src* **{WC|WZ|WCZ}**

SUMNZ *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: Conditionally adds or subtracts Src from Dest based on flag state.

- Dest is a register containing the value to adjust.
- Src is a register, 9-bit literal, or 32-bit augmented literal.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0011100	CZI	DDDDDDDDD	SSSSSSSSS	Sign	result == 0	D	2
EEEE	0011101	CZI	DDDDDDDDD	SSSSSSSSS	Sign	result == 0	D	2
EEEE	0011110	CZI	DDDDDDDDD	SSSSSSSSS	Sign	result == 0	D	2
EEEE	0011111	CZI	DDDDDDDDD	SSSSSSSSS	Sign	result == 0	D	2

Explanation:

These instructions conditionally add or subtract Src from Dest based on the specified flag state:

Instruction	Subtracts when	Adds when
SUMC	C = 1	C = 0
SUMNC	C = 0	C = 1
SUMZ	Z = 1	Z = 0
SUMNZ	Z = 0	Z = 1

The C flag (with WC) is updated to reflect the correct sign of the result.

SUMC and **SUMZ** subtract when their flag is set (1). **SUMNC** and **SUMNZ** subtract when their flag is clear (0), providing complementary behavior.

Instructions: T

This section contains all PASM2 instructions beginning with the letter T.

Conditional Jump Timing Convention: Conditional jumps in this section (TJZ, **TJNZ**, **TJF**, **TJNF**, **TJV**, **TJS**, **TJNS**) show their `Clks` field as `NOT-taken / taken`. The *taken* value depends on execution context:

Context	Clocks when taken
COG / LUT execution	4
Hub execution	13…20

So `2 OR 4 / 2 OR 13-20` reads as: 2 cycles when the jump is not taken, 4 cycles when taken in cog/LUT, 13–20 cycles when taken in hub execution.

TEST

TEST

Arithmetic Operations - Tests parity and zero state of a value.

TEST *Dest* **{WC|WZ|WCZ}**

TEST *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The parity and zero-state of Dest, or of Dest bitwise ANDed with Src, is stored in the C and Z flags.

- Dest is a register whose value will be tested.
- Src is an optional register, 9-bit literal, or 32-bit augmented literal to AND with Dest.
- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks		
EEEE	0111110	CZ0	DDDDDDDDD	DDDDDDDDD	Parity of D	(D == 0)	—	2		
EEEE	0111110	CZI	DDDDDDDDD	SSSSSSSSS	Parity of (D	S)	((D	S) == 0)	—	2

Related: TESTN, TESTB, TESTBN, TESTP, TESTPN

Explanation:

TEST determines the parity (number of high bits) and the zero or non-zero state of Dest, or of Dest bitwise ANDed with Src, and stores the results in the C and/or Z flag.

If the WC or WCZ effect is specified, the C flag is set to 1 if the number of high bits in Dest (or Dest ANDed with Src) is odd, or is cleared to 0 if it is even.

If the WZ or WCZ effect is specified, the Z flag is set to 1 if Dest (or Dest ANDed with Src) is zero, or is cleared to 0 if it is not zero.

TEST is non-destructive—it does not modify Dest.

```
1         TEST    flags WCZ      ' Test all bits for parity and zero
2         TEST    value, #$FF WZ ' Test low byte for zero
```

TESTB

TEST Bit

Arithmetic Operations - Tests a specific bit and optionally combines with flag.

TESTB *Dest, {#}Src* **WC/WZ**
TESTB *Dest, {#}Src* **ANDC/ANDZ**
TESTB *Dest, {#}Src* **ORC/ORZ**
TESTB *Dest, {#}Src* **XORC/XORZ**

Result: The state of bit Src[4:0] of Dest is read and either stored as-is, or bitwise ANDed, ORed, or XORed into C or Z.

- Dest is a register whose bit will be tested.
- Src is a register or 5-bit literal (0-31) specifying bit position.
- WC/WZ writes bit state directly to C or Z flag.
- ANDC/ANDZ ANDs bit state with C or Z flag.
- ORC/ORZ ORs bit state with C or Z flag.
- XORC/XORZ XORs bit state with C or Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100000	CZI	DDDDDDDDD	SSSSSSSSS	D[S[4:0]]	D[S[4:0]]	—	2
EEEE	0100010	CZI	DDDDDDDDD	SSSSSSSSS	C/Z AND D[S[4:0]]	C/Z AND D[S[4:0]]	—	2
EEEE	0100100	CZI	DDDDDDDDD	SSSSSSSSS	C/Z OR D[S[4:0]]	C/Z OR D[S[4:0]]	—	2
EEEE	0100110	CZI	DDDDDDDDD	SSSSSSSSS	C/Z XOR D[S[4:0]]	C/Z XOR D[S[4:0]]	—	2

Related: TESTBN, TESTP, TESTPN

Explanation:

TESTB reads the state (0 or 1) of a bit in Dest designated by Src, and either stores it as-is, or bitwise ANDs, ORs, or XORs it into C or Z. The bit position is specified by Src[4:0] (0-31). The WC, WZ, ANDC, ANDZ, ORC, ORZ, XORC, or XORZ effect determines how the bit state is applied to the selected flag.

TESTB is useful for examining individual bits without modifying the register value.

```
1         TESTB   flags, #7 WC   ' Test bit 7, store in C
2         TESTB   mask, #3 ANDC  ' AND bit 3 with current C
```

TESTBN

TEST Bit Negated

Arithmetic Operations - Tests a specific bit inverted and optionally combines with flag.

TESTBN *Dest, {#}Src* **WC/WZ**
TESTBN *Dest, {#}Src* **ANDC/ANDZ**
TESTBN *Dest, {#}Src* **ORC/ORZ**
TESTBN *Dest, {#}Src* **XORC/XORZ**

Result: The inverted state of bit Src[4:0] of Dest is read and either stored as-is, or bitwise ANDed, ORed, or XORed into C or Z.

- Dest is a register whose bit will be tested.
- Src is a register or 5-bit literal (0-31) specifying bit position.
- WC/WZ writes inverted bit state to C or Z flag.
- ANDC/ANDZ ANDs inverted bit state with C or Z flag.
- ORC/ORZ ORs inverted bit state with C or Z flag.
- XORC/XORZ XORs inverted bit state with C or Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0100001	CZI	DDDDDDDDD	SSSSSSSSS	!D[S[4:0]]	!D[S[4:0]]	—	2
EEEE	0100011	CZI	DDDDDDDDD	SSSSSSSSS	C/Z AND !D[S[4:0]]	C/Z AND !D[S[4:0]]	—	2
EEEE	0100101	CZI	DDDDDDDDD	SSSSSSSSS	C/Z OR !D[S[4:0]]	C/Z OR !D[S[4:0]]	—	2
EEEE	0100111	CZI	DDDDDDDDD	SSSSSSSSS	C/Z XOR !D[S[4:0]]	C/Z XOR !D[S[4:0]]	—	2

Related: TESTB, TESTP, TESTPN

Explanation:

TESTBN reads the state (0 or 1) of a bit in Dest designated by Src, inverts that result, and either stores it as-is, or bitwise ANDs, ORs, or XORs it into C or Z. The bit position is specified by Src[4:0] (0-31). The WC, WZ, ANDC, ANDZ, ORC, ORZ, XORC, or XORZ effect determines how the inverted bit state is applied to the selected flag.

TESTBN is useful for testing whether a bit is clear (0) rather than set (1).

TESTN

TEST Not

Arithmetic Operations - Tests parity and zero state with inverted mask.

TESTN *Dest, {#}Src* **{WC|WZ|WCZ}**

Result: The parity and zero-state of Dest bitwise ANDed with !Src is stored in the C and Z flags.

- Dest is a register whose value will be tested.
- Src is a register, 9-bit literal, or 32-bit augmented literal to invert and AND with Dest.

- WC, WZ, or WCZ are optional effects to update flags.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks		
EEEE	0111111	CZI	DDDDDDDDD	SSSSSSSSS	Parity of (D	!S)	((D	!S) == 0)	—	2

Related: TEST, TESTB, TESTBN

Explanation:

TESTN determines the parity (number of high bits) and the zero or non-zero state of Dest bitwise ANDed with !Src and stores the results in the C and/or Z flag.

If the WC or WCZ effect is specified, the C flag is set to 1 if the number of high bits in Dest ANDed with !Src is odd, or is cleared to 0 if it is even.

If the WZ or WCZ effect is specified, the Z flag is set to 1 if Dest ANDed with !Src is zero, or is cleared to 0 if it is not zero.

TESTN is non-destructive—it does not modify Dest. It is useful for testing which bits in Dest are set while masking out specific bits defined by Src.

TESTP / TESTPN

TEST Pin / **TEST** Pin Negated {#testpn}
Pin I/O and Smart Pins - Tests I/O pin state and optionally combines with flag.

TESTP *{#}Dest* **WC/WZ**
TESTP *{#}Dest* **ANDC/ANDZ**
TESTP *{#}Dest* **ORC/ORZ**
TESTP *{#}Dest* **XORC/XORZ**

TESTPN *{#}Dest* **WC/WZ**
TESTPN *{#}Dest* **ANDC/ANDZ**
TESTPN *{#}Dest* **ORC/ORZ**
TESTPN *{#}Dest* **XORC/XORZ**

Result: The state (TESTP) or inverted state (TESTPN) of the I/O pin described by Dest is read and either stored as-is, or bitwise ANDed, ORed, or XORed into C or Z.

- Dest is a register or 6-bit literal (0-63) identifying the I/O pin.
- WC/WZ writes pin state to C or Z flag.
- ANDC/ANDZ ANDs pin state with C or Z flag.
- ORC/ORZ ORs pin state with C or Z flag.
- XORC/XORZ XORs pin state with C or Z flag.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	001000000	IN	IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000001	!IN	!IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000010	C/Z AND IN	C/Z AND IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000011	C/Z AND !IN	C/Z AND !IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000100	C/Z OR IN	C/Z OR IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000101	C/Z OR !IN	C/Z OR !IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000110	C/Z XOR IN	C/Z XOR IN	—	2
EEEE	1101011	CZL	DDDDDDDDD	001000111	C/Z XOR !IN	C/Z XOR !IN	—	2

IN = pin state at Dest[5:0]; !IN = inverted pin state.

Related: TESTB, TESTBN, DRVL, DRVH

Explanation:

TESTP reads the state (0 or 1) of the I/O pin designated by Dest, and either stores it as-is, or bitwise ANDs, ORs, or XORs it into C or Z. TESTPN does the same but inverts the pin state first. The pin number is specified by Dest[5:0] (0-63). The WC, WZ, ANDC, ANDZ, ORC, ORZ, XORC, or XORZ effect determines how the pin state is applied to the selected flag.

Both instructions read the actual pin state from the IN register, not the output register. This makes them useful for reading sensor inputs, detecting edges, and building multi-bit values from pin states. **TESTPN** is particularly useful for active-low signals where a low pin state (0) indicates an active condition.

```
1         TESTP   #10 WC          ' Read pin 10 state into C
2         TESTP   sensor_pin WZ   ' Test sensor pin, store in Z
3         TESTPN  #button WC      ' C=1 if active-low button pressed
```

TJF / TJNF

TEST And Jump If Full / Not Full {#tjnf}
Branching and Flow Control - Tests for all bits set and conditionally jumps.

TJF *Dest, {#}Src*

TJNF *Dest, {#}Src*

Result: Dest is tested and conditionally jumps based on full state.

- Dest is a register whose value is tested for full state.
- Src is a register, 9-bit literal, or 20-bit augmented literal specifying jump address.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011101	00I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20
EEEE	1011101	01I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20

Related: TJZ, TJNZ, TJS, TJNS, TJV

Explanation:

TJF and **TJNF TEST** Dest for "full" state ($FFFF_FFFF = -1 = all bits set) and conditionally jump:

Instruction	Jumps when
TJF	Dest = $FFFF_FFFF (full)
TJNF	Dest != $FFFF_FFFF (not full)

The address (Src) can be absolute or relative. To specify an absolute address, Src must be a register containing a 20-bit address value. To specify a relative address, use #Label for a 9-bit signed offset or use ##Label for a 20-bit signed offset. Offsets are relative to the instruction following the **TJF**/TJNF.

Takes 2 clocks when not jumping, 4 clocks when jumping (pipeline flush).

TJS / TJNS

TEST And Jump If Signed / Not Signed {#tjns}
Branching and Flow Control - Tests sign bit and conditionally jumps.

TJS *Dest, {#}Src*

TJNS *Dest, {#}Src*

Result: Dest is tested and conditionally jumps based on sign bit state.

- Dest is a register whose value is tested for sign bit.
- Src is a register, 9-bit literal, or 20-bit augmented literal specifying jump address.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011101	10I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20
EEEE	1011101	11I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20

Related: TJZ, TJNZ, TJF, TJNF, TJV

Explanation:

TJS and **TJNS** test the sign bit (bit 31) of Dest and conditionally jump:

Instruction	Jumps when
TJS	Dest[31] = 1 (negative/signed)
TJNS	Dest[31] = 0 (positive/unsigned)

The address (Src) can be absolute or relative. To specify an absolute address, Src must be a register containing a 20-bit address value. To specify a relative address, use #Label for a 9-bit signed offset or use ##Label for a 20-bit signed offset. Offsets are relative to the instruction following the **TJS**/TJNS.

Takes 2 clocks when not jumping, 4 clocks when jumping (pipeline flush).

TJZ / TJNZ

TEST And Jump If Zero / Not Zero {#tjnz}
Branching and Flow Control - Tests for zero and conditionally jumps.

TJZ *Dest, {#}Src*

TJNZ *Dest, {#}Src*

Result: Dest is tested (not modified), and conditionally jumps based on zero/non-zero result.

- Dest is a register whose value is tested (unchanged).
- Src is the jump address: use # for relative, omit for absolute.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011100	10I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20
EEEE	1011100	11I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20

*PC is written only when the jump condition is met.

Related: TJF, TJNF, TJS, TJNS, TJV, DJZ, DJNZ

Explanation:

TJZ and **TJNZ TEST** Dest (without modifying it) and conditionally jump based on whether the value is zero or non-zero:

Instruction	Jumps when
TJZ	Dest = 0
TJNZ	Dest != 0

Unlike **DJZ**/DJNZ which decrement before testing, these instructions only **TEST**.

```
        TJNZ    count, #loop   ' Loop while count <> 0
        TJZ     count, #done   ' Exit when count = 0
```

Takes 2 clocks when not jumping, 4 clocks when jumping (pipeline flush).

TJV

TEST And Jump If Overflow

Branching and Flow Control - Tests for signed overflow and conditionally jumps.

TJV *Dest, {#}Src*

Result: Dest is tested against C and if it has overflowed (Dest[31] != C), PC is set to a new relative (#Src) or absolute (Src) address.

- Dest is a register whose value is tested for overflow.
- Src is a register, 9-bit literal, or 20-bit augmented literal specifying jump address.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1011110	00I	DDDDDDDDD	SSSSSSSSS	—	—	PC*	2 or 4 / 2 or 13-20

Related: ADDS, ADDSX, SUBS, SUBSX

Explanation:

TJV tests the value in Dest against C and jumps to the address described by Src if Dest has overflowed (Dest[31] != C). This instruction requires that C be updated (to the correct sign) by the previous **ADDS**, **ADDSX**, **SUBS**, **SUBSX**, **CMPS**, **CMPSX**, or SUMx instruction. The address (Src) can be absolute or relative.

The instruction takes 2 cycles if the jump is not taken, or 4 cycles if taken.

```
        ADDS    result, delta WC  ' Signed add, update C
        TJV     result, #overflow_handler
```

TRGINT1 / TRGINT2 / TRGINT3

Trigger Interrupt (1, 2, Or 3)

Interrupts - Software-triggers an interrupt handler.

TRGINT1
TRGINT2
TRGINT3

Result: The specified interrupt handler (INT1, INT2, or INT3) is triggered regardless of STALLI mode.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	000100010	000100100	—	—	—	2
EEEE	1101011	000	000100011	000100100	—	—	—	2
EEEE	1101011	000	000100100	000100100	—	—	—	2

Related: SETINT1/2/3, NIXINT1/2/3, RETI0/1/2/3, RESI0/1/2/3

Explanation:

TRGINT1, TRGINT2, and TRGINT3 software-trigger their respective interrupt handlers, regardless of **STALLI** mode. This allows code to explicitly invoke interrupt service routines without waiting for external events.

The P2 provides three independent interrupt levels, and each TRGINT instruction triggers only its corresponding level. Use these instructions when you need to invoke an interrupt handler programmatically.

Instructions: W

This section contains all PASM2 instructions beginning with the letter W.

WAITATN

Wait For Attention
Events and Timing - Waits for an attention event from another cog.

WAITATN {WC|WZ|WCZ}

Result: Waits for an attention event to occur (unless the event flag is already set), then clears the event flag (unless it's being set again by the event sensor) and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011110	000100100	Timeout	Timeout	—	2+

Related: COGATN, POLLATN, JATN, JNATN

Explanation:

WAITATN waits for an attention event to occur, stalling the pipeline until the event flag is set. The attention event flag is set whenever another cog issues an attention request for this cog using **COGATN**. The flag is cleared upon cog start or execution of **POLLATN**, **WAITATN**, **JATN**, or **JNATN** instructions.

To set an optional timeout, insert a **SETQ** instruction (with a future System Counter target value) immediately before **WAITATN**. The WC, WZ, or WCZ effect is recommended only when timeout is specified. Flags are set (1) if timeout occurred before the event, or cleared (0) if the event occurred before timeout.

During a wait, the pipeline is stalled—no instructions execute and no interrupts are processed in the cog until the wait condition ends.

```
1         WAITATN                 ' Wait for attention from another cog
```

WAITCT1 / WAITCT2 / WAITCT3

Wait For Counter Event
Events and Timing - Waits for a counter event flag to be set.

WAITCT1 {WC|WZ|WCZ}

WAITCT2 {WC|WZ|WCZ}

WAITCT3 {WC|WZ|WCZ}

Result: Waits for the specified counter event flag (CT1, CT2, or CT3) to be set, then clears the flag (unless it's being set again by the event sensor) and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000010001	000100100	Timeout	Timeout	—	2+
EEEE	1101011	CZ0	000010010	000100100	Timeout	Timeout	—	2+
EEEE	1101011	CZ0	000010011	000100100	Timeout	Timeout	—	2+

Related: ADDCT1, ADDCT2, ADDCT3, POLLCT1, POLLCT2, POLLCT3, JCT1, JCT2, JCT3

Explanation:

WAITCT1, WAITCT2, and WAITCT3 wait for counter events 1, 2, or 3 respectively, stalling the pipeline until the corresponding event flag is set. Each counter event flag is set whenever the System Counter (CT) passes the value in the corresponding event trigger register (CT1, CT2, or CT3). Specifically, the flag is set when the MSB of (CT - CTx) equals 0, providing a precise mathematical definition of "passes" that handles counter wraparound correctly.

The flags are cleared by execution of ADDCT*n*, POLLCT*n*, WAITCT*n*, JCT*n*, or JNCT*n* instructions (where *n* is 1, 2, or 3).

To set an optional timeout, insert a **SETQ** instruction immediately before the WAITCTn instruction.

WAITFBW

Wait For FIFO Block Wrap
Events and Timing - Waits for a FIFO block wrap event.

WAITFBW {WC|WZ|WCZ}

Result: Waits for a FIFO-interface-block-wrap event to occur, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011001	000100100	Timeout	Timeout	—	2+

Related: RDFAST, WRFAST, FBLOCK, POLLFBW

Explanation:

WAITFBW waits for a FIFO-interface-block-wrap event to occur, stalling the pipeline until the event flag is set. The FIFO-interface-block-wrap event flag is set whenever the Hub RAM FIFO interface exhausts its block count and reloads its block count and start address.

The FIFO-interface-block-wrap event flag is cleared upon execution of **RDFAST**, **WRFAST**, **FBLOCK**, **POLLFBW**, **WAITFBW**, **JFBW**, or **JNFBW** instructions.

WAITINT

Wait For Interrupt
Events and Timing - Waits for an interrupt event to occur.

WAITINT {WC|WZ|WCZ}

Result: Waits for an interrupt-occurred event, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000010000	000100100	Timeout	Timeout	—	2+

Related: POLLINT, JINT, JNINT

Explanation:

WAITINT waits for an interrupt-occurred event to occur, stalling the pipeline until the event flag is set. The interrupt-occurred event flag is set whenever interrupt 1, 2, or 3 occurs—debug interrupts are ignored.

The interrupt-occurred event flag is cleared upon cog start or execution of **POLLINT**, **WAITINT**, **JINT**, or **JNINT** instructions.

WAITPAT

Wait For Pattern
Events and Timing - Waits for a pin pattern match event.

WAITPAT {WC|WZ|WCZ}

Result: Waits for a pin-pattern-detected event, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011000	000100100	Timeout	Timeout	—	2+

Related: SETPAT, POLLPAT, JPAT, JNPAT

Explanation:

WAITPAT waits for a pin-pattern-detected event to occur, stalling the pipeline until the event flag is set. The pin-pattern-detected event flag is set whenever the masked input pins match or don't match the pattern described by a previous **SETPAT** instruction.

The pin-pattern-detected event flag is cleared upon execution of **SETPAT**, **POLLPAT**, **WAITPAT**, **JPAT**, or **JNPAT** instructions.

```
1         SETPAT  mask, pattern  ' Set up pattern detector
2         WAITPAT                ' Wait for pattern match
```

WAITSE1 / WAITSE2 / WAITSE3 / WAITSE4

Wait For Selectable Event (1, 2, 3, Or 4)
Events and Timing - Waits for a selectable event flag to be set.

WAITSE1 {WC|WZ|WCZ}

WAITSE2 {WC|WZ|WCZ}

WAITSE3 {WC|WZ|WCZ}

WAITSE4 {WC|WZ|WCZ}

Result: Waits for the specified selectable event flag (SE1-SE4) to be set, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000010100	000100100	Timeout	Timeout	—	2+
EEEE	1101011	CZ0	000010101	000100100	Timeout	Timeout	—	2+
EEEE	1101011	CZ0	000010110	000100100	Timeout	Timeout	—	2+
EEEE	1101011	CZ0	000010111	000100100	Timeout	Timeout	—	2+

Related: SETSE1/2/3/4, POLLSE1/2/3/4, JSE1/2/3/4, JNSE1/2/3/4

Explanation:

WAITSE1, WAITSE2, WAITSE3, and WAITSE4 wait for their respective selectable events to occur, stalling the pipeline until the corresponding SE flag is set.

Each selectable event flag is cleared by execution of its respective SETSEn, POLLSEn, WAITSEn, JSEn, or JNSEn instruction.

WAITX

Wait Cycles

Miscellaneous - Stalls the cog for a precise number of clock cycles.

WAITX *{#}Dest* **{WC|WZ|WCZ}**

Result: Stalls the cog for 2 + Dest clock cycles. If WC/WZ/WCZ is specified, waits 2 + (Dest AND RND) clocks for a randomized delay and clears C and Z to 0 after completion.

- Dest is the delay value; total wait is 2 + Dest cycles (0-511 for immediate).
- WC, WZ, or WCZ enable randomized delay mode; C and Z are set to 0 after completion.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZL	DDDDDDDDD	000011111	0	0	—	2 + D

Related: WAITCT1, WAITCT2, WAITCT3

Explanation:

WAITX stalls the cog for 2 + Dest clock cycles. When WC, WZ, or WCZ is specified, the delay becomes randomized: 2 + (Dest AND RND) clocks, where RND is a random value. This randomized mode is useful for avoiding timing-based interference between cogs. **WAITX** is critical for bit-banging protocols, PWM generation, and timing-sensitive operations where precise delays are required.

WAITX blocks cog execution completely—no instructions execute and no interrupts are processed during the wait period. For **LONG** delays, consider using WAITCT instructions instead.

```
1         WAITX   #99             ' Wait 101 clock cycles (2 + 99)
```

WAITXFI

Wait For Streamer Finished

Events and Timing - Waits for the streamer to finish all commands.

WAITXFI {WC|WZ|WCZ}

Result: Waits for a streamer-finished event to occur, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011011	000100100	Timeout	Timeout	—	2+

Related: WAITXMT, WAITXRL, WAITXRO, XINIT, XCONT

Explanation:

WAITXFI waits for a streamer-finished event to occur, stalling the pipeline until the event flag is set. The streamer-finished event flag is set whenever the streamer runs out of commands to process.

The streamer-finished event flag is cleared upon execution of **XINIT**, **XZERO**, **XCONT**, **POLLXFI**, **WAITXFI**, **JXFI**, or **JNXFI** instructions.

WAITXMT

Wait For Streamer Empty

Events and Timing - Waits for the streamer to be ready for a new command.

WAITXMT {WC|WZ|WCZ}

Result: Waits for a streamer-empty event to occur, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011010	000100100	Timeout	Timeout	—	2+

Related: WAITXFI, WAITXRL, WAITXRO, XINIT, XCONT

Explanation:

WAITXMT waits for a streamer-empty event to occur, stalling the pipeline until the event flag is set. The streamer-empty event flag is set whenever the streamer is ready for a new command.

The streamer-empty event flag is cleared upon execution of **XINIT**, **XZERO**, **XCONT**, **POLLXMT**, **WAITXMT**, **JXMT**, or **JNXMT** instructions.

WAITXRL

Wait For Streamer LUT Rollover

Events and Timing - Waits for the streamer LUT RAM rollover event.

WAITXRL {WC|WZ|WCZ}

Result: Waits for a streamer-LUT-RAM-rollover event to occur, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011101	000100100	Timeout	Timeout	—	2+

Related: WAITXFI, WAITXMT, WAITXRO, POLLXRL

Explanation:

WAITXRL waits for a streamer-LUT-RAM-rollover event to occur, stalling the pipeline until the event flag is set. The streamer-LUT-RAM-rollover event flag is set whenever location $1FF of the Lookup RAM is read by the streamer.

The streamer-LUT-RAM-rollover event flag is cleared upon cog start or execution of **POLLXRL**, **WAITXRL**, **JXRL**, or **JNXRL** instructions.

WAITXRO

Wait For Streamer NCO Rollover
Events and Timing - Waits for the streamer NCO rollover event.

WAITXRO {WC|WZ|WCZ}

Result: Waits for a streamer-NCO-rollover event to occur, then clears the flag and resumes execution.

- WC, WZ, or WCZ are optional effects to set flags on timeout.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	CZ0	000011100	000100100	Timeout	Timeout	—	2+

Related: WAITXFI, WAITXMT, WAITXRL, POLLXRO

Explanation:

WAITXRO waits for a streamer-NCO-rollover event to occur, stalling the pipeline until the event flag is set. The streamer-NCO-rollover event flag is set whenever the streamer's numerically-controlled oscillator (NCO) rolls over.

The streamer-NCO-rollover event flag is cleared upon execution of **XINIT**, **XZERO**, **XCONT**, **POLLXRO**, **WAITXRO**, **JXRO**, or **JNXRO** instructions.

WFBYTE

Write FIFO **BYTE**
Hub Memory Access - Writes a byte to the Hub FIFO interface.

WFBYTE *{#}Dest*

Result: Writes the byte in Dest[7:0] into the FIFO. Must be used after WRFAST has configured the FIFO.

- Dest is the byte value to write (bits 7:0 used).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000010101	—	—	—	2

Related: WFWORD, WFLONG, WRFAST

Explanation:

WFBYTE writes a byte from Dest[7:0] into the Hub FIFO interface. This instruction must be used after **WRFAST** has configured the FIFO for fast Hub memory writes.

Only the lower 8 bits of Dest are written. **WFBYTE** executes in 2 clock cycles when the FIFO is ready. If the FIFO is full, execution stalls until space becomes available.

WFLONG

Write FIFO **LONG**
Hub Memory Access - Writes a long to the Hub FIFO interface.

WFLONG *{#}Dest*

Result: Writes the long in Dest[31:0] into the FIFO. Must be used after WRFAST has configured the FIFO.

- Dest is the long value to write (all 32 bits used).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000010111	—	—	—	2

Related: WFBYTE, WFWORD, WRFAST

Explanation:

WFLONG writes a long (32-bit value) from Dest[31:0] into the Hub FIFO interface. This instruction must be used after **WRFAST** has configured the FIFO for fast Hub memory writes.

All 32 bits of Dest are written. **WFLONG** executes in 2 clock cycles when the FIFO is ready. If the FIFO is full, execution stalls until space becomes available.

WFWORD

Write FIFO **WORD**
Hub Memory Access - Writes a word to the Hub FIFO interface.

WFWORD *{#}Dest*

Result: Writes the word in Dest[15:0] into the FIFO. Must be used after WRFAST has configured the FIFO.

- Dest is the word value to write (bits 15:0 used).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	00L	DDDDDDDDD	000010110	—	—	—	2

Related: WFBYTE, WFLONG, WRFAST

Explanation:

WFWORD writes a word (16-bit value) from Dest[15:0] into the Hub FIFO interface. This instruction must be used after **WRFAST** has configured the FIFO for fast Hub memory writes.

Only the lower 16 bits of Dest are written. **WFWORD** executes in 2 clock cycles when the FIFO is ready. If the FIFO is full, execution stalls until space becomes available.

WMLONG

Write Masked **LONG**
Hub Memory Access - Writes only non-zero bytes to Hub RAM.

WMLONG *Dest, {#}Src/P*

Result: Writes only non-$00 bytes in Dest[31:0] to hub address Src/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.

- Dest is the long value with bytes to write (non-zero bytes only).
- Src/P is the hub address or pointer (PTRA/PTRB).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1010011	11I	DDDDDDDDD	SSSSSSSSS	—	—	—	3…10

Related: WRLONG, WRBYTE, WRWORD

Explanation:

WMLONG writes only non-zero bytes from Dest to Hub RAM at address Src. Each byte in Dest is examined: if the byte is $00, that **BYTE** position in Hub RAM is not modified; if the byte is non-zero, it is written to Hub RAM.

This masked write capability is useful for sprite graphics, text overlay, and other applications where selective pixel/byte updates are needed without affecting other data in the same **LONG**.

Prior execution of **SETQ** or **SETQ**2 invokes cog or LUT block transfer mode.

WRBYTE

Write **BYTE**
Hub Memory Access - Writes a byte to Hub RAM.

WRBYTE *{#}Dest, {#}Src/P*

Result: Writes the byte in Dest[7:0] to hub address Src/PTRx.

- Dest is the byte value to write (bits 7:0 used).
- Src/P is the hub address or pointer (PTRA/PTRB).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100010	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	3…10 †

† **Timing varies by execution context:**

Context	Clocks
COG / LUT execution	3…10
Hub execution	3…20

Related: WRWORD, WRLONG, RDBYTE

Explanation:

WRBYTE writes the byte in Dest[7:0] to Hub RAM at address Src/PTRx. Only the lower 8 bits of Dest are written.

The instruction takes 3–10 cycles in cog/LUT execution, or 3–20 cycles in hub execution, depending on hub-window alignment. When Src specifies PTRA or PTRB, the pointer value is used as the Hub address. Pointer auto-increment modes can be applied for sequential access.

```
1         WRBYTE  value, ptra++  ' Write byte, increment pointer
```

WRC / WRNC / WRZ / WRNZ

Write Flag To Register

Arithmetic Operations - Writes 0 or 1 to register based on flag state.

WRC *Dest*

WRNC *Dest*

WRZ *Dest*

WRNZ *Dest*

Result: Writes 0 or 1 to Dest based on the specified flag condition:

Instruction	Dest value
WRC	1 if C=1, else 0
WRNC	1 if C=0, else 0
WRZ	1 if Z=1, else 0
WRNZ	1 if Z=0, else 0

- Dest is the destination register. Upper 31 bits are cleared to zero.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001101100	—	—	D	2
EEEE	1101011	000	DDDDDDDDD	001101101	—	—	D	2
EEEE	1101011	000	DDDDDDDDD	001101110	—	—	D	2
EEEE	1101011	000	DDDDDDDDD	001101111	—	—	D	2

Explanation:

These instructions copy flag states to a register, providing a convenient way to convert flag conditions into numeric values for computation or storage.

WRC and **WRZ** write the direct flag state (C or Z), while **WRNC** and **WRNZ** write the inverted flag state. The result is always 0 or 1; the upper 31 bits of Dest are cleared.

WRFAST

Write FIFO Setup
Hub Memory Access - Configures the Hub FIFO for fast writes.

WRFAST *{#}Dest, {#}Src*

Result: Initializes the Hub FIFO for fast writes. Dest[31] = no wait, Dest[13:0] = block size in 64-byte units (0 = max), Src[19:0] = block start address.

- Dest contains configuration: bit 31 = nowait, bits 13:0 = block size.
- Src contains Hub RAM start address (bits 19:0).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100100	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2 or WRFAST finish + 3

Related: WFBYTE, WFWORD, WFLONG, RDFAST

Explanation:

WRFAST configures the Hub FIFO interface for fast streaming writes to Hub RAM. After **WRFAST** executes, use **WFBYTE**, **WFWORD**, or **WFLONG** to write data through the FIFO.

Dest[13:0] specifies the block size in 64-byte units. A value of 0 selects the maximum block size. Dest[31] controls wait behavior: if set, FIFO writes proceed without stalling.

Src[19:0] specifies the starting Hub RAM address. The FIFO automatically increments the address as data is written.

```
1         WRFAST  #0, buffer_addr  ' Set up FIFO write to buffer
2         WFLONG  data              ' Write data to FIFO
```

WRLONG

Write **LONG**
Hub Memory Access - Writes a long to Hub RAM.

WRLONG *{#}Dest, {#}Src/P*

Result: Writes the long in Dest[31:0] to hub address Src/PTRx. Prior SETQ/SETQ2 invokes cog/LUT block transfer.

- Dest is the long value to write (all 32 bits used).
- Src/P is the hub address or pointer (PTRA/PTRB).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100011	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	3…10 †

† **Timing varies by execution context:**

Context	Clocks
COG / LUT execution	3…10
Hub execution	3…20

Related: WRBYTE, WRWORD, WMLONG, RDLONG

Explanation:

WRLONG writes the 32-bit value in Dest to Hub RAM at address Src/PTRx. All 32 bits of Dest are written.

The instruction takes 3–10 cycles in cog/LUT execution, or 3–20 cycles in hub execution, depending on hub-window alignment (minimum 3 cycles when the window is hit). When Src specifies PTRA or PTRB, the pointer value is used as the Hub address. Pointer auto-increment modes can be applied for sequential access.

Prior execution of **SETQ** or **SETQ**2 invokes block transfer mode, writing multiple longs from cog or LUT RAM to Hub RAM in a burst transfer.

```
1         SETQ    #16-1          ' Set up for 16-long block transfer
2         WRLONG  buffer, ptra   ' Write 16 longs to hub
```

Pitfall (Silicon Bug): When using **SETQ**/SETQ2 for block transfers with PTRx expressions, do NOT place any ALTx, **AUGS**, or **AUGD** instruction between **SETQ**/SETQ2 and **WRLONG**. Such intervening instructions cancel the block-size PTRx delta calculation—the data transfers correctly, but PTRx advances by only a single-long delta (4 bytes) instead of the full block size.

WRLUT

Write LUT

Lookup Table - Writes a value to Lookup Table RAM.

WRLUT *{#}Dest, {#}Src/P*

Result: Writes Dest to LUT address Src/PTRx.

- Dest is the value to write.
- Src/P is the LUT address or pointer (PTRA/PTRB).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100001	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: RDLUT, WRLONG, SETQ

Explanation:

WRLUT writes the value in Dest to the Lookup Table (LUT) at address Src/PTRx. The LUT is a 512-long (2KB) fast memory space.

When Src specifies PTRA or PTRB, the pointer value is used as the LUT address. Only the lower 9 bits of the address are used (0-511).

WRLUT executes in 2 clock cycles, providing fast access to LUT RAM for lookup tables, buffers, and temporary storage.

```
        WRLUT   value, #100    ' Write to LUT address 100
```

WRPIN

Write Pin Mode
Pin I/O and Smart Pins - Configures the operating mode of a Smart Pin.

WRPIN *{#}Dest, {#}Src*

Result: Sets the mode of smart pins Src[10:6]+Src[5:0]..Src[5:0] to Dest, acknowledges smart pins. Wraps within A/B pins. Prior SETQ overrides Src[10:6].

- Dest is the smart pin mode configuration.
- Src is the pin number or pin range.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100000	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: WXPIN, WYPIN, RDPIN, AKPIN

Explanation:

WRPIN configures the operating mode of one or more Smart Pins. Each of the P2's 64 pins has a dedicated Smart Pin module capable of autonomous operation for PWM, serial I/O, pulse measurement, ADC, and many other functions.

CRITICAL REQUIREMENT: Smart pins MUST be reset (DIR=0) before configuring with **WRPIN**.

The standard configuration sequence is: 1. **DIRL** pin — Reset smart pin (required) 2. **WRPIN** mode, pin — Configure smart pin mode 3. **WXPIN** x, pin — Set X parameter 4. **WYPIN** y, pin — Set Y parameter 5. **DIRH** pin — Enable smart pin

WRPIN #0, pin clears all smart pin configuration.

```
        DIRL    #10            ' Reset pin 10
        WRPIN   pwm_mode, #10  ' Configure for PWM
        WXPIN   period, #10    ' Set period
        DIRH    #10            ' Enable
```

WRWORD

Write **WORD**

Hub Memory Access - Writes a word to Hub RAM.

WRWORD *{#}Dest, {#}Src/P*

Result: Writes the word in Dest[15:0] to hub address Src/PTRx.

- Dest is the word value to write (bits 15:0 used).
- Src/P is the hub address or pointer (PTRA/PTRB).

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100010	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	3…10 †

† **Timing varies by execution context:**

Context	Clocks
COG / LUT execution	3…10
Hub execution	3…20

Related: WRBYTE, WRLONG, RDWORD

Explanation:

WRWORD writes the word (16-bit value) in Dest[15:0] to Hub RAM at address Src/PTRx. Only the lower 16 bits of Dest are written.

The instruction takes 3–10 cycles in cog/LUT execution, or 3–20 cycles in hub execution, depending on hub-window alignment. When Src specifies PTRA or PTRB, the pointer value is used as the Hub address. Pointer auto-increment modes can be applied for sequential access.

WXPIN

Write Pin X Parameter

Pin I/O and Smart Pins - Sets the X parameter of a Smart Pin.

WXPIN *{#}Dest, {#}Src*

Result: Sets the X register of smart pins Src[10:6]+Src[5:0]..Src[5:0] to Dest, acknowledges smart pins. Wraps within A/B pins. Prior SETQ overrides Src[10:6].

- Dest is the X parameter value.
- Src is the pin number or pin range.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100000	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: WRPIN, WYPIN, RDPIN

Explanation:

WXPIN sets the X parameter of one or more Smart Pins. The X register meaning depends on the smart pin mode:

- For PWM modes: Sets frame period or duty cycle parameter
- For serial modes: Controls bit timing and configuration
- For pulse measurement: Sets measurement parameters
- For transition modes: Controls timebase

Writing the X register also acknowledges the smart pin, clearing any completion flags.

WYPIN

Write Pin Y Parameter
Pin I/O and Smart Pins - Sets the Y parameter of a Smart Pin.

WYPIN *{#}Dest, {#}Src*

Result: Sets the Y register of smart pins Src[10:6]+Src[5:0]..Src[5:0] to Dest, acknowledges smart pins. Wraps within A/B pins. Prior SETQ overrides Src[10:6].

- Dest is the Y parameter value.
- Src is the pin number or pin range.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100001	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: WRPIN, WXPIN, RDPIN

Explanation:

WYPIN sets the Y parameter of one or more Smart Pins. The Y register serves multiple purposes depending on smart pin mode:

- For PWM modes: Sets the base period
- For SPI/serial modes: Controls data to transmit
- For counter modes: Sets count value
- For ADC modes: Initiates conversions

Writing the Y register also acknowledges pin completion, clearing any completion flags. This dual purpose makes **WYPIN** essential for continuous smart pin operation—it both provides new data and signals that previous results have been processed.

```
1         WYPIN   pwm_value, #10  ' Set PWM duty and acknowledge
```

Instructions: X

This section contains all PASM2 instructions beginning with the letter X. The X instructions include the **XOR** logic operation, the xoroshiro32+ PRNG instruction, and the streamer control family.

XCONT

Execute Continue
Streamer - Buffers a streamer command continuing from current phase.

XCONT *{#}Dest, {#}Src*

Result: Buffers a new streamer command to execute when the current command completes its final NCO rollover, continuing from current phase.

- Dest is the streamer mode configuration.
- Src is the data value or hub address for the streamer operation.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100110	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2+

Related: XINIT, XZERO, XSTOP, WAITXFI

Explanation:

XCONT buffers a new streamer command that executes automatically when the current command completes. Unlike **XINIT** and **XZERO**, **XCONT** preserves the phase accumulator, allowing seamless continuation of streamer operations without phase discontinuities.

This instruction enables chaining multiple streamer operations together while maintaining phase coherence. The buffered command waits for the current command's NCO (numerically controlled oscillator) to complete its final rollover before activation.

The mode **WORD** in Dest specifies the streamer configuration including pin assignments, data direction, and transfer format. The Src parameter provides either immediate data or a hub memory address depending on the mode configuration.

XINIT

Execute Initialize
Streamer - Issues a streamer command immediately with phase reset to zero.

XINIT *{#}Dest, {#}Src*

Result: Issues a streamer command immediately with the phase accumulator reset to zero.

- Dest is the streamer mode configuration.
- Src is the data value or hub address for the streamer operation.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100101	0LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2

Related: XCONT, XZERO, XSTOP, WAITXFI, SETXFRQ

Explanation:

XINIT starts a streamer operation immediately, resetting the phase accumulator to zero. This provides a clean starting point for high-speed data transfers between the cog and hub memory or I/O pins.

The streamer operates as a hardware DMA engine, transferring data without CPU intervention. The mode **WORD** in Dest configures critical parameters:

- Transfer direction (input from pins to hub, output from hub to pins, or cog-only operations)
- Number of pins involved in the transfer
- Data formatting (bit order, **BYTE** packing, **WORD** sizes)

The Src parameter provides either the data source (for immediate transfers) or a hub memory address (for hub-based transfers).

XINIT commonly coordinates with smart pins to achieve maximum I/O throughput:

```
1         XINIT   mode, data          ' Start data transfer
2         WYPIN   count, #clk_pin     ' Start clock generation
3         WAITXFI                     ' Wait for completion
```

This parallel operation eliminates CPU intervention, enabling sustained high-speed data rates limited only by the configured clock frequency.

XOR

Exclusive Or

Arithmetic Operations - Performs bitwise exclusive OR of Dest and Src.

XOR *Dest, {#}Src* **{WC/WZ/WCZ}**

Result: Dest XOR Src is stored in Dest. Optionally sets C to parity of result and Z if result equals zero.

- Dest is the register containing the value to **XOR** with Src.
- Src is a register, 9-bit literal, or 32-bit augmented literal (##) whose value is XORed with Dest.
- WC sets C to the parity (odd number of 1 bits) of the result.
- WZ sets Z if the result equals zero.
- WCZ sets both C and Z.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0101011	CZI	DDDDDDDDD	SSSSSSSSS	parity of result	result == 0	D	2

Related: AND, OR, ANDN, TEST

Explanation:

XOR performs a bitwise exclusive **OR** operation between Dest and Src, storing the result in Dest. Each bit position in the result is set to 1 if the corresponding bits in Dest and Src differ, or 0 if they match.

Dest	Src	Result
0	0	0
0	1	1
1	0	1
1	1	0

The exclusive **OR** operation has several important properties:

- XORing a value with itself produces zero (useful for clearing registers)
- XORing a value with all 1s produces the bitwise complement
- XORing twice with the same value returns the original (useful for simple encryption)
- **XOR** is commutative: A **XOR** B equals B **XOR** A

When the WC effect is specified, the C flag receives the parity of the result—set to 1 if the result contains an odd number of 1 bits, or cleared to 0 for an even number. This provides a fast parity calculation.

When the WZ effect is specified, the Z flag is set if the result equals zero (meaning Dest and Src were identical), or cleared if the result is non-zero (Dest and Src differ in at least one bit).

XORO32

Xoroshiro 32

Arithmetic Operations - Generates next pseudo-random number using xoroshiro32+ algorithm.

XORO32 *Dest*

Result: Dest is updated with the next PRNG state. The generated random value is placed into the S field of the next instruction.

- Dest is the register containing the 32-bit PRNG state.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1101011	000	DDDDDDDDD	001101000	—	—	D	2

Related: GETRND, SETQ

Explanation:

XORO32 implements one iteration of the xoroshiro32+ algorithm, a fast, high-quality pseudo-random number generator. The instruction updates the generator state in Dest and simultaneously makes the generated random value available to the next instruction by injecting it into that instruction's S field.

The xoroshiro32+ algorithm provides excellent statistical properties for a 32-bit generator:

- Long period (2^{}32 - 1 values before repeating)
- Good distribution across all output bits
- Fast execution (2 clocks per random number)
- Small state requirement (single 32-bit value)

```
        MOV     seed, initial_value  ' Initialize with non-zero seed

.loop   XORO32  seed                 ' Advance PRNG state
        MOV     random_val, 0        ' Next instruction receives random in S
        ' Process random_val...
```

The random value appears in the S field of the instruction immediately following XORO32. This means the next instruction must be one that reads from S, and the value specified for S in that instruction's encoding is ignored—it gets replaced by the random value.

The seed value in Dest must be non-zero. A seed of zero will produce only zero values. For best results, initialize the seed with a value from **GETRND** or another entropy source.

XSTOP

Execute Stop

Streamer - Immediately halts the active streamer operation.

XSTOP

Result: The currently active streamer operation terminates immediately.

- Takes no operands.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100101	011	000000000	000000000	—	—	—	2

Related: XINIT, XCONT, XZERO, WAITXFI

Explanation:

XSTOP immediately halts any active streamer operation. This provides programmatic control to abort streamer transfers before completion.

When **XSTOP** executes, the streamer hardware stops all data movement and pin activity. Any buffered streamer command (from **XCONT** or **XZERO**) is also discarded.

XSTOP is useful when:

- Error conditions require aborting a transfer
- Dynamic control flow needs to terminate streaming based on data content
- Cleanup is required before reconfiguring the streamer

After **XSTOP**, the streamer remains idle until a new **XINIT** command is issued. **XSTOP** is itself an alias for **XINIT** #0,#0, so it leaves the phase accumulator zeroed. To restart, issue **XINIT** (which begins a new command with phase reset to zero); **XCONT** cannot be used to restart from idle because it only buffers behind an active command.

XZERO

Execute Zero

Streamer - Buffers a streamer command with phase reset to zero.

XZERO *{#}Dest, {#}Src*

Result: Buffers a new streamer command to execute when the current command completes, resetting phase to zero.

- Dest is the streamer mode configuration.
- Src is the data value or hub address for the streamer operation.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	1100101	1LI	DDDDDDDDD	SSSSSSSSS	—	—	—	2+

Related: XINIT, XCONT, XSTOP, WAITXFI

Explanation:

XZERO buffers a new streamer command that executes automatically when the current command completes, with the phase accumulator reset to zero. This combines the buffering behavior of **XCONT** with the phase-zeroing behavior of **XINIT**.

The buffered command waits for the current streamer operation's NCO (numerically controlled oscillator) to complete its final rollover before activation. When activation occurs, the phase accumulator resets to zero, providing a clean starting point for the new operation.

This instruction enables chaining multiple streamer operations where each operation should start from a known phase state. This is particularly useful when switching between different streamer modes or when phase coherence between operations is not required.

The mode **WORD** in Dest specifies the streamer configuration including pin assignments, data direction, and transfer format. The Src parameter provides either immediate data or a hub memory address depending on the mode configuration.

Instructions: Z

This section contains all PASM2 instructions beginning with the letter Z. There is currently one Z instruction: **ZEROX** for zero extension.

ZEROX

Zero Extend
Arithmetic Operations - Zero-extends a value above the specified bit position.

ZEROX *Dest, {#}Src* **{WC/WZ/WCZ}**

Result: Dest is zero-extended above the bit indicated by Src[4:0]. Optionally sets C to MSB of result and Z if result equals zero.

- Dest is the register containing the value to zero-extend.
- Src is a register or 9-bit literal identifying the bit position (0-31) beyond which to zero-extend.
- WC sets C to the MSB (bit 31) of the result.
- WZ sets Z if the result equals zero.
- WCZ sets both C and Z.

EEEE	Opcode	CZI	Dest	Src	C	Z	Result	Clks
EEEE	0111010	CZI	DDDDDDDDD	SSSSSSSSS	MSB of result	result == 0	D	2

Related: SIGNX

Explanation:

ZEROX fills the bits of Dest, above the bit indicated by Src[4:0], with zeros, effectively zero-extending the value. This is useful when converting encoded or received unsigned values from a smaller bit width to 32 bits.

For example, if Dest contains $FFFF_FFFF and Src contains 7, **ZEROX** clears bits 31 down to bit 8, leaving only bits 7-0 intact. The result in Dest becomes $0000_00FF.

The instruction examines only the lower 5 bits of Src (Src[4:0]), allowing bit positions 0 through 31 to be specified. This makes **ZEROX** particularly useful for extracting and zero-extending bit fields from packed data structures or network protocols.

```
1         ' Extract lower byte and zero-extend
2         MOV     data, big_value
3         ZEROX   data, #7          ' Keep bits 7-0, clear bits 31-8
4                                   ' If big_value was $FFFF_FFFF,
5                                   ' data becomes $0000_00FF
```

ZEROX is the complement to **SIGNX**. While **ZEROX** fills upper bits with zeros (for unsigned values), **SIGNX** fills upper bits with the value of the designated bit (for signed values). Use **ZEROX** when working with unsigned data, and **SIGNX** when working with signed data that needs proper sign extension.

Assembler Directives

Assembler directives control the assembly process itself. Unlike instructions that generate executable code, directives guide the assembler in organizing memory, reserving space, and verifying code constraints. Directives execute at assembly time, not runtime.

The P2 assembler provides 15 directives organized into seven functional categories: origin control, memory definition, size verification, alignment, code replication, space management, and inline assembly control.

Origin Control Directives

Origin directives set the memory address where subsequent code or data will be assembled. The P2 distinguishes between COG RAM (0-$1FF), LUT RAM ($200-$3FF), and Hub RAM addresses.

The $ Symbol (Current Origin)

Within DAT blocks, the `$` symbol represents the current origin address:

- **In COG mode** (after **ORG**): `$` returns the current COG address in longs (0-$3FF)
- **In Hub mode** (after **ORGH**): `$` returns the current Hub address in bytes

```
1  DAT
2          ORG     0
3          ' $ = 0 (COG address 0)
4          NOP
5          ' $ = 1 (COG address 1)
6
7          ORGH    $400
8          ' $ = $400 (Hub address $400)
9          BYTE    0
10         ' $ = $401 (Hub address $401)
```

COG/LUT Memory Regions

Address Range	Memory	Notes
$000 - $1EF	COG RAM	General purpose registers
$1F0 - $1FF	COG RAM	Special purpose registers (PTRA, DIRA, etc.)
$200 - $3FF	LUT RAM	Lookup table / additional code space

ORG

Set Origin
Sets assembly origin to a specific COG/LUT RAM address.

Set the assembly origin to a specific COG or LUT RAM address. All subsequent instructions assemble starting from this address.

Syntax

```
        ORG                     ' Reset to COG address 0, limit $1F8
        ORG     address         ' Set COG address, auto-calculate limit
        ORG     address, limit  ' Set COG address and limit
```

Parameters

Parameter	Range	Description
address	0 to $400	Starting COG/LUT address (in longs)
limit	0 to $400	Maximum address for FIT checking (optional)

Auto-Limit Behavior

1. **Without parameters** (`ORG`):
   - Sets COG address to 0
   - Sets limit to $1F8 (standard COG RAM limit, before special registers)
2. **With address only** (`ORG address`):
   - Sets COG address to specified value
   - Auto-calculates limit:
     - If address < $200: limit = $200 (COG RAM boundary)
     - If address >= $200: limit = $400 (LUT RAM boundary)
3. **With address and limit** (`ORG address, limit`):
   - Sets COG address and limit to specified values

Usage

Use **ORG** to position code or data at specific COG/LUT RAM addresses. This is essential for creating interrupt vectors, placing time-critical code at optimal locations, organizing cog memory layout, or positioning code in LUT RAM.

Example

```
        ORG     0               ' Start at COG address 0
entry   JMP     #main           ' First instruction at address 0

        ORG     $100            ' Start at COG address $100
table   LONG    1, 2, 3         ' Data table at specific address

        ORG     $200            ' Start in LUT RAM
```

continues on next page →

↪ *continued from previous page*

```
lut_code
        MOV     PA, #0          ' LUT address $200
        RET                     ' LUT address $201
        FIT     $400            ' Verify fits in LUT
```

Restrictions

Restriction	Error Message
Inside inline assembly	`ORG not allowed within inline assembly code`
Inside DITTO block	`ORG not allowed within a DITTO block`
Address > $400	`Cog address exceeds $400 limit`
Cannot precede with symbol	`This directive cannot be preceded by a symbol`

Notes

- **ORG** affects COG/LUT RAM addresses (range 0-$3FF)
- For Hub RAM addresses, use **ORGH**
- To fill gaps between addresses with zeros, use ORGF
- **ORG** sets the address counter without generating any bytes
- DAT blocks start in Hub mode by default; use **ORG** to switch to COG mode

Pitfall: Forgetting that **ORG** without parameters defaults to limit $1F8 (not $200) can cause unexpected **FIT** errors when code approaches the special register area.

Related Directives

- ORGH — Set hub RAM origin
- ORGF — Set origin with zero-fill
- FIT — Verify code fits within address limit

ORGF

Set Origin With Fill

Advances to specified address, filling with zeros.

Set origin with fill—advance to specified address, filling intervening space with zeros. Unlike **ORG** which only sets the address counter, ORGF fills the gap between the current address and the target address with zero bytes.

Syntax

```
        ORGF    address
```

Parameters

Parameter	Description
address	Target COG address to advance to (0-$1FF), filling intervening space with zeros

Usage

Use ORGF for contiguous binary output with guaranteed zero-filled gaps. ORGF ensures data structures start at exact addresses while maintaining a complete memory image. Essential for interrupt vector tables, memory-mapped structures, and fixed-layout binary formats.

Example

```
DAT
        ORG     0
entry   JMP     #main
        ' ... some code ...

        ORGF    $100            ' Fill with zeros up to address $100
table   LONG    1, 2, 3         ' Table starts exactly at $100

        ' Create fixed-size code block
        ORG     0
block_start
        ' ... code ...
        ORGF    block_start + 64   ' Ensure block is exactly 64 longs
block_end
```

Restrictions

Restriction	Error Message
In ORGH mode	`ORGF is not allowed in ORGH mode`
Target < current	`Origin already exceeds target`
Target > limit	`Cog address exceeds limit`
Cannot precede with symbol	`This directive cannot be preceded by a symbol`

Notes

- ORGF fills the gap with zero bytes/longs to reach the target address
- ORGF is only valid in COG mode (after **ORG**), not in Hub mode
- Generates assembly error if target address is less than current address
- **ORG** only changes the address counter without filling
- Useful for creating fixed-layout binary structures
- Essential for interrupt vector tables and memory-mapped structures

Pitfall: ORGF only works in COG mode. Attempting to use ORGF after **ORGH** produces an error. For hub address gaps, use explicit **BYTE** or **LONG** declarations with zero values.

Related Directives

- ORG — Set origin without fill
- ORGH — Set Hub RAM origin
- FIT — Verify code fits
- RES — Reserve space without initialization

ORGH

Set Hub Origin
Sets assembly origin to a Hub RAM address.

Set the assembly origin to a Hub RAM address. All subsequent code and data assemble for hub execution starting at the specified address.

Syntax

```
1         ORGH                    ' Reset to current hub position (or $400)
2         ORGH    address         ' Set hub address
3         ORGH    address, limit  ' Set hub address and limit
```

Parameters

Parameter	Range	Description
address	$400 to $100000	Starting hub address (in bytes)
limit	address to $100000	Maximum address for FIT checking (optional)

Behavior by Context

1. **Without parameters** (`ORGH`):
   - In Spin2 objects: Sets hub address to $400 (after interpreter)
   - In PASM-only objects: Sets hub address to current object position
   - Sets limit to $100000 (1MB)
2. **With address only** (`ORGH address`):
   - Sets hub address to specified value
   - In PASM-only mode: Pads with zeros to reach the address
   - Sets limit to $100000
3. **With address and limit** (`ORGH address, limit`):
   - Sets hub address and limit to specified values

Address Constraints

Context	Minimum	Maximum
Spin2 objects	$400	$100000
PASM-only objects	0	$100000

The $400 minimum for Spin2 objects reserves space for the Spin2 interpreter.

Usage

Use **ORGH** when switching from cog-exec code to hub-exec code, or when defining data that resides in Hub RAM. DAT blocks start in Hub mode by default. Use **ORGH** to explicitly set hub addresses or to switch back to Hub mode after using **ORG**.

Example

```
        ORGH    $400            ' Start at hub address $400
        ' Hub-exec code here

        ORGH                    ' Default: start at hub $400

        ORGH    $1000           ' Start at hub address $1000
hubData LONG    $DEADBEEF       ' Hub address $1000
        LONG    $CAFEBABE       ' Hub address $1004

        ORGH    $400, $800      ' Hub from $400 to $800 limit
        BYTE    0[1024]         ' 1KB of data
        FIT     $800            ' Verify fits within limit
```

Mode Switching

A DAT block can switch between COG and Hub modes multiple times:

```
DAT
        ORGH                    ' Hub mode: bytecode tables
dispatch_table
        WORD    @routine1
        WORD    @routine2
        ALIGNL

        ORG     $100            ' COG mode: register code
routine1
        MOV     PA, #1
        RET

```

continues on next page →

↪ *continued from previous page*

```
13          ORGH                    ' Back to hub mode
14  hub_data
15          LONG    $12345678
```

Restrictions

Restriction	Error Message
Inside inline assembly	`ORGH not allowed within inline assembly code`
Inside DITTO block	`ORGH not allowed within a DITTO block`
Address < $400 (Spin2)	`Hub address below $400 limit`
Address > $100000	`Hub address exceeds $100000 ceiling`
Address decrease (PASM)	`Hub address cannot decrease`
Limit < address	`Hub address exceeds limit`
Cannot precede with symbol	`This directive cannot be preceded by a symbol`

Notes

- **ORGH** sets Hub RAM addresses for hub-exec code and hub data
- Default address is $400 if not specified (in Spin2 objects)
- Hub-exec code executes directly from Hub RAM without loading into COG
- After **ORGH**, use **ORG** to switch to COG RAM addresses
- DAT blocks start in Hub mode by default

Tip: Use `@label` to get the hub address of any label, regardless of whether that label is in COG or Hub mode.

Related Directives

- ORG — Set COG RAM origin
- ORGF — Set origin with fill
- FIT — Verify code fits within limit

Memory Definition Directives

Memory definition directives allocate and initialize data in memory. Each directive specifies the size of data elements (byte, **WORD**, or **LONG**) and their initial values.

BYTE

Declare **BYTE** Data

Stores 8-bit values at the current address.

Declare **BYTE** data in memory. Stores 8-bit values at the current address.

Syntax

```
[label] BYTE    value[, value...]
[label] BYTE    value[count]
```

Parameters

Parameter	Description
value	8-bit value or string literal
count	Repetition count (creates *count* copies of *value*)

Usage

Use **BYTE** to define individual bytes, **BYTE** arrays, or strings. Each value occupies exactly 1 **BYTE**. Strings are stored as individual bytes in sequence. **BYTE** provides no automatic alignment—data appears at the current address.

The repetition syntax `value[count]` creates multiple copies of the same value, useful for initializing buffers or padding.

Example

```
text    BYTE    "Hello P2", 0   ' String with null terminator
data    BYTE    $FF, $00, $55   ' Hex values
nums    BYTE    1, 2, 3, 4, 5   ' Decimal values
zeros   BYTE    0[256]          ' 256 zero bytes (buffer initialization)
pattern BYTE    $AA[16], $55[16] ' Alternating pattern: 16 $AA, then 16 $55
```

Notes

- Each value occupies exactly 1 **BYTE**
- Strings are stored as individual bytes without alignment
- No automatic alignment—use **ALIGNW** or **ALIGNL** if needed
- Values outside 0-255 range will be truncated to 8 bits
- The `[count]` syntax repeats the preceding value, useful for buffer initialization

Related Directives

- WORD — Declare 16-bit **WORD** data
- LONG — Declare 32-bit **LONG** data
- BYTEFIT — Declare **BYTE** data with range validation
- RES — Reserve uninitialized space

LONG

Declare **LONG** Data

Stores 32-bit values at the current address.

Declare **LONG** data in memory. Stores 32-bit values at the current address.

Syntax

```
[label] LONG    value[, value...]
[label] LONG    value[count]
```

Parameters

Parameter	Description
value	32-bit value, expression, or address reference
count	Repetition count (creates *count* copies of *value*)

Usage

Use **LONG** to define 32-bit integers, addresses, or any data requiring full 32-bit precision. Each value occupies 4 bytes. No automatic alignment—data packs sequentially; use **ALIGNL** before **LONG** if alignment is needed for optimal access efficiency.

The repetition syntax `value[count]` creates multiple copies of the same value, useful for initializing register buffers or lookup tables.

Example

```
counter LONG    0               ' Single long
table   LONG    $1234_5678      ' Hex value with underscores for readability
ptrs    LONG    @start, @end    ' Address pointers
buffer  LONG    0[32]           ' 32 zero longs (128 bytes)
clkfreq LONG    160_000_000[8]  ' Initialize 8 entries with clock frequency
```

Notes

- Each value occupies 4 bytes
- No automatic alignment—data packs sequentially; use **ALIGNL** if alignment needed

- Supports full 32-bit range (0 to $FFFFFFFF)
- Standard size for P2 registers and instructions
- The `[count]` syntax repeats the preceding value

Related Directives

- BYTE — Declare 8-bit **BYTE** data
- WORD — Declare 16-bit **WORD** data
- ALIGNL — Force **LONG** alignment
- RES — Reserve uninitialized longs

WORD

Declare **WORD** Data

Stores 16-bit values at the current address.

Declare **WORD** data in memory. Stores 16-bit values at the current address.

Syntax

```
[label] WORD    value[, value...]
[label] WORD    value[count]
```

Parameters

Parameter	Description
value	16-bit value or expression
count	Repetition count (creates *count* copies of *value*)

Usage

Use **WORD** to define 16-bit integers or data elements. Each value occupies 2 bytes. Data packs sequentially without automatic alignment—use **ALIGNW** if **WORD** alignment is needed for efficient access.

The repetition syntax `value[count]` creates multiple copies of the same value, useful for initializing tables or buffers.

Example

```
counts  WORD    1000, 2000, 3000    ' Decimal values
addr    WORD    @buffer             ' Address reference (lower 16 bits)
zeros   WORD    0[64]               ' 64 zero words (128 bytes)
sine    WORD    $8000[256]          ' Init sine table with midpoints
```

Notes

- Each value occupies 2 bytes
- No automatic alignment—data packs sequentially; use **ALIGNW** if alignment needed
- Range: 0 to 65535 (unsigned)
- Values outside this range will be truncated to 16 bits
- The `[count]` syntax repeats the preceding value

Related Directives

- BYTE — Declare 8-bit **BYTE** data
- LONG — Declare 32-bit **LONG** data
- WORDFIT — Declare **WORD** data with range validation
- ALIGNW — Force **WORD** alignment

FILE

Include Binary File

Includes raw binary file data at the current address.

Include the contents of a binary file at the current assembly address. The raw bytes from the specified file are inserted directly into the assembled output.

Syntax

```
[label] FILE    "filename"
```

Parameters

Parameter	Description
filename	Filename enclosed in double quotes (no path separators allowed)

Filename Requirements

The filename must not contain path separator characters. The following characters are invalid in filenames:

Character	Description
/	Forward slash
:	Colon
*	Asterisk
?	Question mark
"	Double quote
<	Less than
>	Greater than
|	Pipe

The compiler searches for the file in the following order: 1. **Current directory** — The directory containing the source file 2. **Library directory** — The compiler's built-in library location 3. **Include directories** — Directories specified via compiler options†

† *Include directory support varies by compiler. PNut_ts supports `-I` options; other P2 compilers may have different or no include directory mechanisms.*

Usage

Use FILE to embed binary resources directly into your program—font data, lookup tables, images, audio samples, or any pre-computed binary content. The file is read at assembly time and its raw bytes are inserted at the current address. A label preceding FILE becomes a byte pointer to the start of the included data.

FILE is only allowed in DAT blocks, not in inline PASM code within PUB or PRI methods.

Example

```
DAT
' Include a font file for VGA text display
font_data   file    "8x8_font.bin"      ' 2KB font bitmap
font_end                                 ' Label marks end for size calc

' Include pre-computed sine table
sine_table  file    "sine_256.dat"      ' 256-entry sine lookup

' Include raw image data
splash      file    "logo.raw"          ' Splash screen bitmap

' Calculate included file size at assembly time
            LONG    @font_end - @font_data  ' Store font size in bytes
```

Example: Text File Inclusion

```
DAT
' Include text file for display
text_data   file    "message.txt"
text_end

PUB ShowText() | ptr, len
    ptr := @text_data
    len := @text_end - @text_data
    ' Process text bytes...
```

Notes

- FILE reads the file at assembly time—the file must exist during compilation
- File contents are included as raw bytes without modification
- A label before FILE provides a byte-addressable pointer to the data
- Place a label after the FILE directive to calculate the included file's size
- FILE is only allowed in DAT blocks (not in inline PASM code)
- Maximum filename length: 253 characters
- Filename case-matching follows the host OS filesystem (case-insensitive on Windows; case-sensitive on Linux and case-sensitive macOS volumes)
- Common uses: fonts, lookup tables, images, audio samples, pre-computed data

Related Directives

- BYTE — Declare individual **BYTE** data
- LONG — Declare **LONG** data
- ORGH — Set hub origin (FILE data resides in hub RAM)

Inline Type Mixing

BYTE, **WORD**, and **LONG** declarations can be mixed within a single data block to create packed data structures. Each type specifier affects only the values that follow it until the next type specifier or end of line.

Example: Protocol Packet Header

```
DAT
' Packet header: 1-byte type, 2-byte length, 4-byte timestamp
packet_hdr
        BYTE    $01             ' Packet type (1 byte)
        WORD    $0100           ' Length field (2 bytes)
        LONG    0               ' Timestamp placeholder (4 bytes)
```

Example: Mixed Data Block

```
DAT
' Sensor configuration block with mixed sizes
sensor_cfg
        BYTE    $42             ' Sensor ID
        BYTE    $03             ' Channel count
        WORD    1000            ' Sample rate (Hz)
        LONG    @callback       ' Callback address
        BYTE    "SENS", 0       ' Name string with terminator
```

Notes

- Data elements pack contiguously regardless of size
- No automatic padding is inserted between different-sized elements
- Use **ALIGNW** or **ALIGNL** when subsequent access requires alignment
- This technique is useful for protocol buffers, hardware register layouts, and memory-mapped structures

For Spin2-declared structures (STRUCT) accessed from PASM2, refer to the Spin2 Reference Manual for structure memory layout and the SIZEOF() operator.

Size Verification Directives

Size verification directives provide compile-time checking that values fit within specified bit ranges. These directives generate assembly errors when constraints are violated, catching overflow errors before runtime.

BYTEFIT

Declare **BYTE** Data With Range Validation

Stores byte values with compile-time range checking.

Declare **BYTE** data with compile-time range validation. Works identically to **BYTE** for storage, but generates an assembly error if any value exceeds the valid **BYTE** range. This catches potential truncation errors during compilation.

Syntax

```
[label] BYTEFIT  value [, value...]
[label] BYTEFIT  value[count]
```

Parameters

Parameter	Description
value	Constant value or expression that must fit in byte range
count	Repetition count (creates *count* copies of *value*)

Valid Range

Representation	Minimum	Maximum
Hexadecimal	-$80	$FF
Decimal (signed)	-128	127
Decimal (unsigned)	0	255

The combined range allows both signed (-128 to +127) and unsigned (0 to 255) byte values.

Usage

Use BYTEFIT instead of **BYTE** for compile-time verification that values fit in 8 bits. BYTEFIT catches overflow errors during assembly rather than silently truncating values. Particularly valuable when values derive from calculations or constants subject to change.

Example

```
DAT
' Valid BYTEFIT values
byteData    BYTEFIT   -$80              ' Minimum signed value: -128
            BYTEFIT   $FF               ' Maximum unsigned value: 255
            BYTEFIT   0, 100, 200, 255  ' Multiple values
            BYTEFIT   -128, -1, 0, 127  ' Signed values
            BYTEFIT   0[100]            ' 100 bytes of value 0

' Lookup table with validation
gammaTable  BYTEFIT   0, 1, 2, 3, 4, 5, 7, 9, 12, 15
            BYTEFIT   18, 22, 27, 32, 38, 44, 51, 58

' The following would cause compile errors:
'           BYTEFIT   256               ' ERROR: 256 > 255
'           BYTEFIT   -129              ' ERROR: -129 < -128
```

Error Message

When values exceed the valid range, the compiler produces:

```
BYTEFIT values must range from -$80 to $FF
```

Notes

- Compile-time validation only—no runtime overhead
- Storage is identical to **BYTE** (8 bits per value)
- Unlike **BYTE**, does not silently truncate out-of-range values
- Useful for lookup tables, configuration data, and calculated offsets
- Can only be used in DAT blocks

Related Directives

- WORDFIT — Declare **WORD** data with range validation
- BYTE — Declare **BYTE** data (no range checking)

WORDFIT

Declare **WORD** Data With Range Validation

Stores word values with compile-time range checking.

Declare **WORD** data with compile-time range validation. Works identically to **WORD** for storage, but generates an assembly error if any value exceeds the valid **WORD** range. This catches potential truncation errors during compilation.

Syntax

```
[label] WORDFIT  value [, value...]
[label] WORDFIT  value[count]
```

Parameters

Parameter	Description
value	Constant value or expression that must fit in word range
count	Repetition count (creates *count* copies of *value*)

Valid Range

Representation	Minimum	Maximum
Hexadecimal	-$8000	$FFFF
Decimal (signed)	-32768	32767
Decimal (unsigned)	0	65535

The combined range allows both signed (-32768 to +32767) and unsigned (0 to 65535) word values.

Usage

Use WORDFIT instead of **WORD** for compile-time verification that values fit in 16 bits. WORDFIT catches overflow errors during assembly rather than silently truncating values. Particularly valuable when values derive from calculations or constants subject to change.

Example

```
DAT
' Valid WORDFIT values
wordData    WORDFIT   -$8000            ' Minimum signed value: -32768
            WORDFIT   $FFFF             ' Maximum unsigned value: 65535
            WORDFIT   1000, 30000       ' Multiple values
            WORDFIT   -32768, 0, 32767  ' Signed values
            WORDFIT   $ABCD[50]         ' 50 words of value $ABCD

' ADC calibration values
adcOffsets  WORDFIT   -1024, -512, 0, 512, 1024
adcGains    WORDFIT   32768, 33000, 32500, 32768

' The following would cause compile errors:
'           WORDFIT   65536             ' ERROR: 65536 > 65535
'           WORDFIT   -32769            ' ERROR: -32769 < -32768
```

Error Message

When values exceed the valid range, the compiler produces:

```
WORDFIT values must range from -$8000 to $FFFF
```

Notes

- Compile-time validation only—no runtime overhead
- Storage is identical to **WORD** (16 bits per value)
- Unlike **WORD**, does not silently truncate out-of-range values
- Useful for lookup tables, calibration data, and calculated offsets
- Can only be used in DAT blocks

Related Directives

- BYTEFIT — Declare **BYTE** data with range validation
- WORD — Declare **WORD** data (no range checking)

Alignment Directives

Alignment directives insert padding bytes to align the next data **OR** instruction to specified boundaries. Proper alignment improves memory access efficiency and is required for certain P2 operations.

ALIGNL

Align To **LONG** Boundary
Inserts padding bytes for 4-byte alignment.

Align to **LONG** boundary (4-byte alignment). Inserts zero bytes as needed to align the next data **OR** instruction to a long boundary.

Syntax

```
DAT
   code_and_data_statements
   ALIGNL
   data_statements
```

Result: The next data element is long-aligned in Hub RAM by emitting up to three bytes (each $00) prior.

- *code_and_data_statements* are leading program code and/or data.
- *data_statements* begin long-aligned in Hub RAM.

Explanation

ALIGNL aligns the next data element to the beginning of the next **LONG** of Hub RAM. **ALIGNL** is important to use when code requires certain data to begin on a long boundary (for access convenience and speed).

ALIGNL is only allowed in DAT blocks, not in in-line PASM.

Example

The following creates a data table of a byte ($11), a word ($BBAA), and a long ($44332211) meant for access from Hub RAM.

```
DAT
     T1       BYTE    $11
     T2       WORD    $BBAA
              LONG    $44332211
```

This data is emitted into the Hub memory image as shown below. The actual starting address depends on preceding code and data; the relative layout remains constant. The L#, W#, and B# labels denote contiguous **LONG**, **WORD**, and **BYTE** boundaries. Note that P2 is little-endian, so the word $BBAA stores as bytes $AA, $BB and the long $44332211 stores as bytes $11, $22, $33, $44 in memory order.

	L0							L1							L2					
	W0		W1		W2		W3		W4		W5									
	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11								
	$11	$AA	$BB	$11	$22	$33	$44	--	--	--	--	--								

Figure 6.1. Figure D.1: Memory Layout Before ALIGNL

Notice how each data element packs immediately after the previous one without any automatic padding or alignment. The word at T2 starts at **BYTE** offset 1 (misaligned), and the long starts at **BYTE** offset 3

(also misaligned). If the code that is meant to access Table T2 expects it to align with a long boundary (i.e. for convenient long-sized access or pointer alignment), the **ALIGNL** directive achieves this, as follows.

```
1 DAT
2     T1      BYTE    $11
3
4             ALIGNL
5     T2      WORD    $BBAA
6             LONG    $44332211
```

In comparison, this data will be emitted as follows:

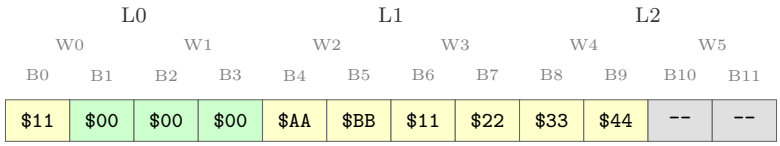


Figure 6.2. Figure D.2: Memory Layout After ALIGNL

In this case, the **ALIGNL** directive causes three zero ($00) bytes to emit after Table T1 to pad and align the start of Table T2 to the boundary of L1. After T2, the word and **LONG** pack sequentially—the **LONG** at offset 6 is still misaligned. To long-align the long as well, another **ALIGNL** would be needed before it.

Notes

- Inserts 0-3 bytes of padding as needed to reach next 4-byte boundary
- P2 requires **LONG** alignment for certain operations
- Critical for hub memory access efficiency
- No effect if already on a long boundary

Related Directives

- ALIGNW — Align to **WORD** boundary
- LONG — Declare **LONG** data
- ORG — Set origin address

ALIGNW

Align To **WORD** Boundary
Inserts padding bytes for 2-byte alignment.

Align to **WORD** boundary (2-byte alignment). Inserts zero bytes as needed to align the next data **OR** instruction to a word boundary.

Syntax

```
DAT
  code_and_data_statements
  ALIGNW
  data_statements
```

Result: The next data element is word-aligned in Hub RAM by emitting zero or one byte ($00) prior.

- *code_and_data_statements* are leading program code and/or data.
- *data_statements* begin word-aligned in Hub RAM.

Explanation

ALIGNW aligns the next data element to the beginning of the next **WORD** of Hub RAM. **ALIGNW** is important to use when code requires certain data to begin on a word boundary (for access convenience and speed).

ALIGNW is only allowed in DAT blocks, not in in-line PASM.

Example

The following creates a data table of a byte ($11), two bytes ($AA, $BB), and a long ($44332211) meant for access from Hub RAM.

```
DAT
    T1      BYTE    $11
    T2      BYTE    $AA, $BB
            LONG    $44332211
```

This data is emitted into the Hub memory image as shown below. The actual starting address depends on preceding code and data; the relative layout remains constant. The L#, W#, and B# labels denote contiguous **LONG**, **WORD**, and **BYTE** boundaries. Note that P2 is little-endian, so the long $44332211 stores as bytes $11, $22, $33, $44 in memory order.

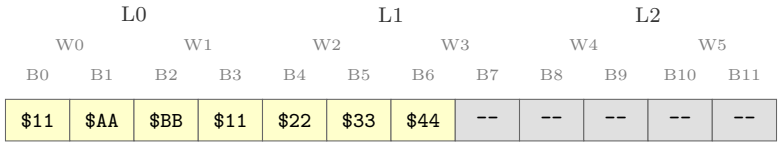


Figure 6.3. Figure D.3: Memory Layout Before ALIGNW

Notice how each data element, regardless of size, is packed right next to the data before it. If the code that is meant to access Table T2 expects it to align with a word boundary (i.e. for convenient word-sized access), the **ALIGNW** directive achieves this, as follows.

```
DAT
    T1      BYTE    $11

            ALIGNW
    T2      BYTE    $AA, $BB
            LONG    $44332211
```

In comparison, this data will be emitted as follows:

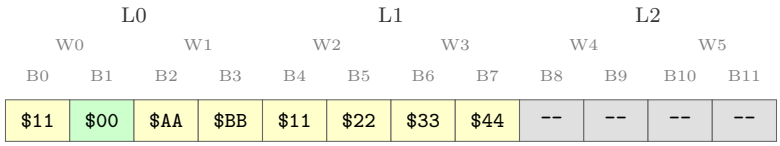


Figure 6.4. Figure D.4: Memory Layout After ALIGNW

In this case, the **ALIGNW** directive causes one zero ($00) **BYTE** to emit after Table T1 to pad and align the start of Table T2 to the boundary of W1. This allows T2 to be accessed as a word-aligned address. Note that the long after T2 packs sequentially at offset 4—it happens to be long-aligned here only because T2 is exactly 2 bytes; this is coincidental, not automatic.

Notes

- Inserts 0-1 bytes of padding as needed to reach next 2-byte boundary
- Important for 16-bit data access efficiency
- No effect if already on a word boundary

Related Directives

- ALIGNL — Align to **LONG** boundary
- WORD — Declare **WORD** data
- ORG — Set origin address

Code Replication Directive

The code replication directive generates multiple copies of instruction or data blocks at compile time. Unlike runtime repetition (REP instruction), code replication expands during assembly, producing distinct instruction copies with optional iteration-based variation.

DITTO

Replicate Code/Data Block

Repeats a block of code or data with iteration index access.

Replicate a block of instructions or data a specified number of times at compile time. The special `$$` symbol provides access to the current iteration index within the block.

Syntax

```
DAT
        DITTO   count           ' Start block, repeat count times
        ' ... code or data ...
        DITTO   END             ' End block
```

Parameters

Parameter	Description
count	Number of iterations (0 or more); zero skips the block entirely
$$	Special symbol evaluating to current iteration index (0 to count-1)

Usage

Use DITTO to generate repetitive code or data patterns without manual duplication. The $$ symbol allows each iteration to produce different values based on the iteration index. This is particularly useful for pin initialization sequences, lookup table generation, and multi-channel configurations. DITTO requires Spin2 v50 or later; place the {Spin2_v50} version directive at the start of the source file.

Example

```
{Spin2_v50}                     ' Required: must be the first line

CON
  NumChannels = 8
  BasePin = 16

DAT
        ORG     0

' Initialize 8 consecutive pins using DITTO
        DITTO   NumChannels
        DRVH    #BasePin + $$   ' Drive pins 16, 17, 18, ... 23 high
        DITTO   END

' Generate indexed data table
        DITTO   4
        LONG    $$ * 100        ' Produces: 0, 100, 200, 300
        DITTO   END

' Multi-instruction block per iteration
        DITTO   NumChannels
        WRPIN   ##PinMode, #BasePin + $$
        WXPIN   ##PinX, #BasePin + $$
```

continues on next page →

↪ *continued from previous page*

```
24          DRVL    #BasePin + $$
25          DITTO   END
```

Zero Count Behavior

When count is 0, the entire block is skipped with no output generated:

```
1  CON
2    MotorCount = 0                ' No motors in this build
3
4  DAT
5          DITTO   MotorCount      ' Block skipped entirely
6          ' ... motor init code ...
7          DITTO   END
```

Restrictions

Restriction	Error Message
ORG inside DITTO	`ORG not allowed within a DITTO block`
ORGH inside DITTO	`ORGH not allowed within a DITTO block`
`$$` outside DITTO	`"$$" (DITTO index) is only allowed within a DITTO block, inside a DAT block`
Negative count	`DITTO count must be a positive integer or zero`
Missing END	`Expected DITTO END`

Notes

- Requires Spin2 v50 or later — add {Spin2_v50} at the top of the file
- Requires the {Spin2_v50} version directive at the start of the source file (first line, before any CON/-DAT) — examples omitting it will not compile
- Works in COG, LUT, and **ORGH** (hub) modes
- `$$` can be used in any expression: `$$ * 2`, `1 << $$`, `BasePin + $$`
- Replication occurs at compile time—no runtime overhead
- Use constants for count to enable configuration: `DITTO NumChannels`
- Each iteration generates its own instructions/data with `$$` evaluated fresh

Related Directives

- **REP** instruction — Hardware-assisted runtime instruction repeat
- ORG — Set origin address (not allowed inside DITTO)
- ORGH — Set hub origin (not allowed inside DITTO)

Space Management Directives

Space management directives control memory allocation and verify size constraints. **FIT** verifies that code fits within specified address limits, while **RES** reserves COG/LUT RAM space without initialization.

FIT

Verify Code Fits

Generates error if current address exceeds limit.

Verify at compile time that the current address has not exceeded a specified limit. **FIT** is a safety check that produces an error if code or data is too large.

Syntax

```
1         FIT     limit           ' Verify current address <= limit
```

Parameters

Parameter	Description
limit	Maximum address (in longs for COG mode, bytes for Hub mode)

Behavior by Mode

In COG Mode (after ORG): - `limit` is a long address (0 to $400) - Error: `Cog address exceeds FIT limit`

In Hub Mode (after ORGH): - `limit` is a byte address - Error: `Hub address exceeds FIT limit`

Common Limit Values

Limit	Meaning
`$1F0`	User COG RAM (before special registers)
`$1F8`	COG RAM (with some special registers)
`$200`	Full COG RAM
`$400`	COG + LUT RAM
`496`	Decimal equivalent of $1F0

Usage

Use **FIT** to verify that code does not exceed available space. This is essential for COG code, which must fit within 512 longs (addresses 0-$1FF). **FIT** generates an assembly error if the current address exceeds the specified limit, catching size overflow during assembly rather than at runtime.

FIT does nothing if the limit is not exceeded—it is purely a compile-time check.

Example: Standard COG Program

```
DAT
        ORG     0

entry   ASMCLK                  ' Set clock
        ' ... main code ...
        JMP     #entry

vars    RES     10

        FIT     $1F0            ' Ensure user area only
```

Example: Split COG/LUT Program

```
DAT
        ORG     0

        ' COG code
        MOV     PA, #1
        CALL    #lut_routine
        JMP     #$

        FIT     $200            ' Must fit in COG before LUT

        ORG     $200            ' LUT code

lut_routine
        MOV     PB, #2
        RET

        FIT     $400            ' Must fit in LUT
```

Example: Hub Data Table

```
DAT
        ORGH    $400

sinTable
        LONG    0[256]          ' Sine lookup table

        FIT     $800            ' Table must not exceed $800
```

Example: Calculated Limits

```
CON
  OVERLAY_END = $300

DAT
        ORG     0
        ' ... overlay code ...
        FIT     OVERLAY_END     ' Must fit before overlay area
```

Restrictions

Restriction	Error
Cannot have a preceding label	`This directive cannot be preceded by a symbol`
Address exceeds COG limit	`Cog address exceeds FIT limit`
Address exceeds Hub limit	`Hub address exceeds FIT limit`

Notes

- **FIT** generates an assembly error if the limit is exceeded
- Essential for COG code size verification
- Special registers occupy COG addresses $1F0-$1FF
- Use **FIT** $1F0 to ensure code does not overwrite special registers
- **FIT** works in both COG mode and Hub mode

Tip: Always add **FIT** after COG code to catch overflow early. It costs nothing at runtime and prevents hard-to-debug overwrites of special registers or adjacent code.

Related Directives

- ORG — Set origin address
- RES — Reserve space
- ORGF — Fill to address

RES

Reserve Space
Allocates COG/LUT RAM without initialization.

Reserve space in COG or LUT RAM without initializing. Allocates memory space but generates no object code.

Syntax

```
[label] RES     count           ' Reserve 'count' longs
[label] RES     0               ' Create label here, no space reserved
```

Parameters

Parameter	Description
label	Symbol name for the reserved space (optional but typical)
count	Number of longs to reserve (can be 0)

Key Characteristics

1. **COG Mode Only** - **RES** only works after **ORG**, not in **ORGH** mode
2. **No Object Code** - **RES** advances the COG address counter but produces no bytes in the object file
3. **Uninitialized** - Reserved space contains whatever was previously in COG RAM
4. **Long-Aligned** - **RES** advances to the next **LONG** boundary before reserving

Usage

Use **RES** to allocate variables and buffers in COG RAM without initializing them. This advances the address counter by the specified number of longs without generating any bytes in the binary. **RES** is only valid in COG/LUT RAM—Hub RAM variables must use **LONG** with initial values or be allocated at runtime.

Example

```
DAT
          ORG     0

entry     MOV     temp, #100
          ADD     temp, value
          RET

temp      RES     1               ' Reserve 1 long for temporary variable
value     RES     1               ' Reserve 1 long for value storage
buffer    RES     16              ' Reserve 16 longs for buffer
```

Zero-Count Label (Alias Technique)

RES with a count of 0 creates a label at the current address without reserving any space. This technique creates aliases—multiple names for the same register:

```
DAT
        ORG     0

' Create aliases - both point to same register
ma      RES     0               ' ma is alias for x (RES 0 = no space)
x       RES     1               ' x occupies 1 long

' Both ma and x refer to the same COG address
```

Tip: Use **RES** 0 aliases to give meaningful names for overlapping register uses—for example, `float_a` and `int_x` can be aliases when the same register serves different purposes at different times.

RES vs LONG for Data

Aspect	RES count	LONG 0[count]
Initializes memory	No	Yes (to 0)
Generates object code	No	Yes
Valid in ORGH mode	No	Yes
Use case	COG working registers	Initialized data

Working with Spin2 Structures

When reserving space for Spin2-declared structures, use the SIZEOF() operator to calculate the correct size in longs:

```
' Reserve space for a Spin2 structure (structure defined in CON block)
mystruct        RES     SIZEOF(point) / 4       ' Reserve longs for point
```

The SIZEOF() operator returns the structure size in bytes, so divide by 4 to convert to longs for **RES**. For complete documentation of Spin2 structures and the SIZEOF() operator, refer to the Spin2 Reference Manual.

Restrictions

Restriction	Error Message
Used in ORGH mode	`RES is not allowed in ORGH mode`
Exceeds limit	`Cog address exceeds limit`

Notes

- **RES** only reserves space in COG/LUT RAM (not Hub RAM)
- No Hub memory is allocated or affected

- Useful for variables and buffers that will be initialized at runtime
- Advances address counter by count longs without generating binary data
- Use **LONG** to declare initialized data in Hub RAM
- SIZEOF() enables correct sizing when working with Spin2 structures

Pitfall: **RES** cannot be used in Hub mode (after **ORGH**). For hub-resident uninitialized buffers, use `LONG 0[count]` which does generate object code.

Related Directives

- LONG — Declare initialized **LONG** data
- ORG — Set origin address
- FIT — Verify space fits within limit

Inline Assembly Directives

Inline assembly allows PASM2 code to be embedded directly within Spin2 PUB and PRI methods. The END directive marks the boundary where inline assembly ends and Spin2 code resumes.

END

End Inline Assembly

Terminates an inline assembly block within a Spin2 method.

Terminate an inline assembly block and return to Spin2 execution. The compiler automatically inserts a **RET** instruction at the END location.

Syntax

```
PUB/PRI MethodName() | locals
  ' Spin2 code

  ORG                           ' Begin inline PASM (COG execution)
  ' ... PASM instructions ...
  END                           ' End inline PASM, implicit RET

  ' Spin2 code continues
```

Parameters

END takes no parameters. It must appear alone on its line.

Usage

Use END to mark the conclusion of an inline assembly block that began with **ORG** or **ORGH** within a PUB or PRI method. Inline assembly enables time-critical operations to execute at full PASM speed within Spin2 methods.

ORG vs ORGH for Inline Assembly:

Directive	Execution Location	Speed	Address Space
ORG	COG RAM	Fastest	$000-$11F (limited)
ORGH	Hub RAM	Fast	Larger

Example: Pin Toggle

```
PUB FastToggle(pin) | mask

  mask := 1 << pin              ' Spin2 code

  ORG                           ' Begin inline PASM (COG execution)
                DRVNOT  mask    ' Toggle the pin
  END                           ' End inline PASM, implicit RET

  ' Execution returns here
```

Example: I2C Start Sequence

```
PUB start() | SCL, sda, tix

  longmove(@SCL, @sclpin, 3)    ' Copy pins & timing to locals

  ORG
                DRVH    sda     ' SDA high
                DRVH    SCL     ' SCL high
                WAITX   tix     ' Delay

                DRVL    sda     ' SDA low (start condition)
                WAITX   tix     ' Delay
                DRVL    SCL     ' SCL low
                WAITX   tix     ' Delay
  END
```

Example: Local Variable Access

Inline PASM accesses local variables by name:

```
PUB Example() | value, result

  value := 100

```

continues on next page →

↪ *continued from previous page*

```
5   ORG
6                   MOV     result, value    ' Read local variable
7                   ADD     result, #50      ' Modify
8   END
9
10  ' result now contains 150
```

Restrictions

Restriction	Error Message
Missing END after ORG/ORGH in method	`Expected END`
ORG inside inline (nested)	`ORG not allowed within inline assembly code`
ORGH inside inline (nested)	`ORGH not allowed within inline assembly code`
ALIGNW/ALIGNL inside inline	`ALIGNW/ALIGNL not allowed within inline assembly code`

END vs RET

Aspect	END	RET instruction
Purpose	End inline block	Return from PASM subroutine
Automatic RET	Compiler adds RET	Manual
Returns to	Spin2 code	PASM caller
Context	Inline assembly only	Any PASM code

Notes

- END is only valid within inline assembly blocks (after **ORG** or **ORGH** in PUB/PRI methods)
- The compiler automatically inserts a **RET** instruction at the END location
- Inline assembly is limited in scope—complex PASM routines belong in DAT blocks
- Local variables declared in the method are accessible by name within inline PASM
- END does not apply to DAT blocks—DAT assembly has no explicit terminator

Variable vs Code Limits in Inline PASM

The Spin2 interpreter handles inline PASM in two separate copy operations:

1. The first 16 **LONG** variables (method parameters, result, and locals) are copied to cog registers `$1E0..$1EF`. **The 16-long limit applies to variables only — not to the PASM code itself.**
2. The PASM code is copied separately into cog registers starting at the **ORG** address (default `$000`).

With no multitasking in use, the inline code area is `$000..$11F` — 288 longs of code space, which is far more than the variable limit suggests.

Multitasking and Inline Code Space Overlap

When a cog uses Spin2 multitasking, the interpreter maintains a taskptr table in cog registers `$100..$11F`. The taskptr for task 31 occupies `$11F`, task 30 occupies `$11E`, and so on, filling downward. **This range is the upper portion of the inline-PASM code area** (`$000..$11F`) — multitasking and large inline-PASM blocks compete for the same space.

Programs using fewer than 32 software tasks leave the *lower* portion of `$100..$11F` free for inline code. Programs using all 32 tasks consume the full table. Plan inline-PASM size accordingly, or place large inline blocks in `ORGH` (hub-exec mode) to avoid this conflict entirely.

Pitfall: Programs using both inline PASM and multitasking can silently lose code space without compile-time warning. If an inline block compiles but behaves unexpectedly with multitasking enabled, suspect taskptr-table overlap and move the block to `ORGH`.

Tip: Keep inline assembly short and focused. For complex PASM routines, define them in a DAT block and launch with **COGINIT** or **CALL** from hub-exec code.

Related Directives

- ORG — Set COG/LUT origin (begins inline block in methods)
- ORGH — Set hub origin (begins hub-exec inline block)

Summary

The P2 assembler's 15 directives provide complete control over memory layout and assembly constraints:

Origin Control: **ORG**, **ORGH**, ORGF set assembly addresses **Memory Definition**: **BYTE**, **WORD**, **LONG** allocate and initialize data; FILE includes binary files **Size Verification**: BYTEFIT, WORDFIT declare data with compile-time range validation **Alignment**: **ALIGNL**, **ALIGNW** optimize memory access **Code Replication**: DITTO generates multiple copies of instruction/data blocks at compile time **Space Management**: **RES**, **FIT** control allocation and verify constraints **Inline Assembly**: END terminates inline PASM blocks within Spin2 methods

These directives execute at assembly time, shaping the binary output without affecting runtime execution. Understanding and using directives effectively is essential for efficient P2 assembly programming.

Special Registers

The P2 provides a set of special-purpose registers that enable critical system functions including Hub RAM access, I/O control, interrupt handling, and timing operations. These registers fall into three categories: dual-purpose registers that can also serve as general RAM, fixed special registers with dedicated hardware functions, and non-memory-mapped registers accessed through specific instructions.

Register Architecture

The P2's special register architecture provides a balance between functionality and flexibility. Each cog has its own independent copy of all special registers, allowing parallel operation without interference. Changes to these registers take effect immediately, enabling precise control over timing-critical operations.

Memory Map ($1F0-$1FF)

The top 16 locations of cog RAM are reserved for special registers:

Special Registers ($1F0–$1FF)

Address	Register	Function
$1F0	IJMP3	Interrupt 3 jump address
$1F1	IRET3	Interrupt 3 return address
$1F2	IJMP2	Interrupt 2 jump address
$1F3	IRET2	Interrupt 2 return address
$1F4	IJMP1	Interrupt 1 jump address
$1F5	IRET1	Interrupt 1 return address
$1F6	PA	Parameter A (COGINIT)
$1F7	PB	Parameter B (COGINIT)
$1F8	PTRA	Hub pointer A
$1F9	PTRB	Hub pointer B
$1FA	DIRA	Direction P0–P31
$1FB	DIRB	Direction P32–P63
$1FC	OUTA	Output P0–P31
$1FD	OUTB	Output P32–P63
$1FE	INA	Input P0–P31 (read-only)
$1FF	INB	Input P32–P63 (read-only)

Figure 6.5. Figure R.1: Special Registers Memory Map (1*F*0–1FF)

Dual-Purpose vs. Fixed Registers

Dual-purpose registers ($1F0-$1F7) can be used as general-purpose cog RAM when their special functions are not enabled. This provides eight additional general-purpose registers for programs that do not use interrupts or the PA/PB facilities.

Fixed special registers ($1F8-$1FF) always provide their special functions when accessed. These registers implement hardware behaviors that activate whenever the register is read or written.

Dual-Purpose Registers

IJMP3

Address $1F0. Interrupt 3 **CALL** address. Stores the address where execution jumps when interrupt 3 is triggered.

Access: Read/Write

Usage: When the INT3 event is triggered, the cog saves the current PC in IRET3 and jumps to the address stored in IJMP3. This register can be used as general RAM when interrupt 3 is not enabled.

Example:

```
        MOV     IJMP3, ##int3_handler   ' Set INT3 handler address
        SETINT3 #event_ct1              ' Enable INT3 for CT1 event
```

Related: IRET3, SETINT3, RETI3

IRET3

Address $1F1. Interrupt 3 return address. Stores the return address when interrupt 3 is triggered.

Access: Read/Write

Usage: When INT3 is triggered, the hardware automatically saves the interrupted PC value to this register. The RETI3 instruction uses this address to return from the interrupt handler. This register can be used as general RAM when interrupt 3 is not enabled.

Example:

```
int3_handler
        ' Handle interrupt...
        RETI3                           ' Return to saved address in IRET3
```

Related: IJMP3, SETINT3, RETI3

IJMP2

Address $1F2. Interrupt 2 **CALL** address. Stores the address where execution jumps when interrupt 2 is triggered.

Access: Read/Write

Usage: When the INT2 event is triggered, the cog saves the current PC in IRET2 and jumps to the address stored in IJMP2. This register can be used as general RAM when interrupt 2 is not enabled.

Example:

```
        MOV     IJMP2, ##int2_handler   ' Set INT2 handler address
        SETINT2 #event_ct2              ' Enable INT2 for CT2 event
```

Related: IRET2, SETINT2, RETI2

IRET2

Address $1F3. Interrupt 2 return address. Stores the return address when interrupt 2 is triggered.

Access: Read/Write

Usage: When INT2 is triggered, the hardware automatically saves the interrupted PC value to this register. The RETI2 instruction uses this address to return from the interrupt handler. This register can be used as general RAM when interrupt 2 is not enabled.

Example:

```
int2_handler
        ' Handle interrupt...
        RETI2                           ' Return to saved address in IRET2
```

Related: IJMP2, SETINT2, RETI2

IJMP1

Address $1F4. Interrupt 1 **CALL** address. Stores the address where execution jumps when interrupt 1 is triggered.

Access: Read/Write

Usage: When the INT1 event is triggered, the cog saves the current PC in IRET1 and jumps to the address stored in IJMP1. This register can be used as general RAM when interrupt 1 is not enabled.

Example:

```
        MOV     IJMP1, ##int1_handler   ' Set INT1 handler address
        SETINT1 #event_ct3              ' Enable INT1 for CT3 event
```

Related: IRET1, SETINT1, RETI1

IRET1

Address $1F5. Interrupt 1 return address. Stores the return address when interrupt 1 is triggered.

Access: Read/Write

Usage: When INT1 is triggered, the hardware automatically saves the interrupted PC value to this register. The RETI1 instruction uses this address to return from the interrupt handler. This register can be used as general RAM when interrupt 1 is not enabled.

Example:

```
int1_handler
        ' Handle interrupt...
        RETI1                           ' Return to saved address in IRET1
```

Related: IJMP1, SETINT1, RETI1

PA

Address $1F6. Multi-purpose register A. Serves multiple special functions or can be used as general RAM.

Access: Read/Write

Usage: PA serves three primary special functions:

1. **CALLD immediate return address storage**: When using **CALLD** with PA as the destination, return information is stored here.
2. **CALLPA parameter passing**: The **CALLPA** instruction copies a value to PA before calling a routine.
3. **LOC address storage**: The **LOC** instruction can store an address in PA.

When these functions are not needed, PA can be used as general-purpose cog RAM.

Example:

```
        CALLD   PA, #subroutine         ' Return info in PA, call
        CALLPA  param, #handler         ' Copy param to PA, call
        LOC     PA, #label              ' Store label address in PA

        ' Using PA as general RAM
        MOV     PA, #42                 ' Regular register usage
```

Related: PB, CALLD, CALLPA, LOC

PB

Address $1F7. Multi-purpose register B. Serves multiple special functions or can be used as general RAM.

Access: Read/Write

Usage: PB serves three primary special functions:

1. **CALLD immediate return address storage**: When using **CALLD** with PB as the destination, return information is stored here.
2. **CALLPB parameter passing**: The **CALLPB** instruction copies a value to PB before calling a routine.
3. **LOC address storage**: The **LOC** instruction can store an address in PB.

When these functions are not needed, PB can be used as general-purpose cog RAM.

Example:

```
        CALLD   PB, #subroutine         ' Return info in PB, call
        CALLPB  param, #handler         ' Copy param to PB, call
        LOC     PB, #label              ' Store label address in PB

        ' Using PB as general RAM
        MOV     PB, ##hub_addr          ' Regular register usage
```

Related: PA, CALLD, CALLPB, LOC

Communication Registers (PR0-PR7)

Addresses $1D8-$1DF. Eight general-purpose registers with predefined symbols.

Access: Read/Write

Memory Map:

Address	Register
$1D8	PR0
$1D9	PR1
$1DA	PR2
$1DB	PR3
$1DC	PR4
$1DD	PR5
$1DE	PR6
$1DF	PR7

Usage: For standalone PASM2 programs, these are ordinary general-purpose registers available for any purpose. The compiler reserves the symbols PR0-PR7 as aliases for these addresses.

Note: The PR0-PR7 symbols exist primarily for Spin2 inline assembly interoperability. See the Spin2 Language Manual for Spin2/PASM2 communication patterns.

Fixed Special Registers

PTRA

Address $1F8. Pointer A to Hub RAM. Primary pointer register for Hub RAM access with automatic increment/decrement support.

Access: Read/Write

Usage: PTRA is the primary pointer for Hub RAM operations. It supports indexed addressing modes with automatic pre- and post-increment/decrement, making it ideal for sequential memory access patterns. PTRA is a 32-bit register; its low 20 bits address the full 1 MB Hub RAM space. On **COGINIT**, the target cog's PTRA receives the **SETQ** value (typically a parameter-block hub address) if a **SETQ** was executed immediately before the **COGINIT**; otherwise PTRA is cleared to 0. This is the standard P2 mechanism for passing a 32-bit parameter or data-structure pointer to a launched cog.

Addressing Modes:

The increment/decrement amount (SCALE) depends on the instruction:

Instruction	SCALE	Increment/Decrement
RDBYTE, WRBYTE	1	1 byte
RDWORD, WRWORD	2	2 bytes
RDLONG, WRLONG, WMLONG	4	4 bytes

- `PTRA++` — Post-increment by SCALE bytes
- `PTRA--` — Post-decrement by SCALE bytes
- `++PTRA` — Pre-increment by SCALE bytes
- `--PTRA` — Pre-decrement by SCALE bytes
- `PTRA[index]` — Indexed access: address = PTRA + (index × SCALE)
- `PTRA++[index]` — Post-update indexed: use PTRA, then PTRA += index × SCALE
- `++PTRA[index]` — Pre-update indexed: PTRA += index × SCALE, then use PTRA

Index ranges: -32 to +31 for non-updating indexed; 1 to 16 for updating forms.

Example:

```
        MOV     ptra, ##hub_buffer      ' Set PTRA to Hub address
        RDLONG  data, ptra++            ' Read long, PTRA += 4 (SCALE=4)
        RDBYTE  char, ptra++            ' Read byte, PTRA += 1 (SCALE=1)
        WRLONG  data, ptra[4]           ' Address: PTRA + (4 × 4) = PTRA+16

        ' Block transfer using SETQ
        SETQ    #15                     ' Transfer 16 longs
        RDLONG  cog_buffer, ptra++      ' Read 16 longs, auto-inc
```

Related: PTRB, RDLONG, WRLONG, RDBYTE, RDWORD, SETQ

PTRB

Address $1F9. Pointer B to Hub RAM. Secondary pointer register for Hub RAM access with automatic increment/decrement support.

Access: Read/Write

Usage: PTRB is the secondary pointer for Hub RAM operations, providing the same capabilities as PTRA. Having two independent pointers enables efficient dual-buffer operations and complex memory access patterns. **COGINIT** writes the code start address to the target cog's PTRB, enabling position-independent code.

Addressing Modes:

PTRB supports the same addressing modes as PTRA, with SCALE determined by instruction type (see PTRA for details):

- `PTRB++` — Post-increment by SCALE bytes
- `PTRB--` — Post-decrement by SCALE bytes
- `++PTRB` — Pre-increment by SCALE bytes
- `--PTRB` — Pre-decrement by SCALE bytes
- `PTRB[index]` — Indexed access: address = PTRB + (index × SCALE)
- `PTRB++[index]` — Post-update indexed: use PTRB, then PTRB += index × SCALE
- `++PTRB[index]` — Pre-update indexed: PTRB += index × SCALE, then use PTRB

Example:

```
1         MOV     ptrb, ##hub_source      ' Set PTRB to source address
2         RDLONG  data, ptrb++            ' Read long, PTRB += 4 (SCALE=4)
3         RDWORD  wval, ptrb++            ' Read word, PTRB += 2 (SCALE=2)
4         WRLONG  data, ptrb[8]           ' Address: PTRB + (8 × 4) = PTRB+32
5
6         ' COGINIT sets PTRB in launched cog
7         COGINIT cognumber, ##code_addr  ' PTRB in target cog gets code_addr
```

Related: PTRA, RDLONG, WRLONG, COGINIT

DIRA

Address $1FA. Direction register A for pins 0-31. Controls whether each pin is an input or output.

Access: Read/Write

Bit Field:

Bits	Name	Description
31:0	DIR	Direction for each pin: 1 = output, 0 = input

Usage: DIRA controls the direction of pins 0-31. Setting a bit to 1 configures the corresponding pin as an output, while 0 configures it as an input. Changes take effect immediately. When a pin is configured as an

output, the value in the corresponding OUTA bit is driven onto the pin. When configured as an input, the pin state can be read from INA.

Example:

```
        MOV     DIRA, ##$00FF_0000      ' Set pins 16-23 as outputs
        OR      DIRA, #1                ' Set pin 0 as output
        ANDN    DIRA, ##$0000_00FF      ' Set pins 0-7 as inputs

        ' Atomic direction change
        MOV     DIRA, new_directions    ' Change all 32 directions
```

Related: DIRB, OUTA, INA, DIRC, DIRH, DIRL

DIRB

Address $1FB. Direction register B for pins 32-63. Controls whether each pin is an input or output.

Access: Read/Write

Bit Field:

Bits	Name	Description
31:0	DIR	Direction for each pin: 1 = output, 0 = input

Usage: DIRB controls the direction of pins 32-63. Setting a bit to 1 configures the corresponding pin as an output, while 0 configures it as an input. The bit positions map to pins 32-63, where bit 0 controls pin 32 and bit 31 controls pin 63.

Example:

```
        MOV     DIRB, #0                ' Set all pins 32-63 as inputs
        OR      DIRB, ##$8000_0000      ' Set pin 63 as output
        ANDN    DIRB, ##$0000_FFFF      ' Set pins 32-47 as inputs
```

Related: DIRA, OUTB, INB

OUTA

Address $1FC. Output register A for pins 0-31. Sets the output state for pins configured as outputs.

Access: Read/Write

Bit Field:

Bits	Name	Description
31:0	OUT	Output state for each pin: 1 = high, 0 = low

Usage: OUTA sets the output state for pins 0-31. Only affects pins configured as outputs via DIRA. Reading OUTA returns the current output register state, not the actual pin states (use INA to read pin states). When multiple cogs drive the same pin, the outputs are OR'd together—if any cog outputs high, the pin goes high.

Example:

```
        MOV     OUTA, #0                ' Clear all outputs 0-31
        OR      OUTA, #1                ' Set pin 0 high
        XOR     OUTA, ##$0000_00FF      ' Toggle pins 0-7
        ANDN    OUTA, pin_mask          ' Clear specific outputs

        ' Atomic pattern change
        MOV     OUTA, new_pattern       ' Change all 32 outputs atomically
```

Related: OUTB, DIRA, INA, OUTC, OUTH, OUTL

OUTB

Address $1FD. Output register B for pins 32-63. Sets the output state for pins configured as outputs.

Access: Read/Write

Bit Field:

Bits	Name	Description
31:0	OUT	Output state for each pin: 1 = high, 0 = low

Usage: OUTB sets the output state for pins 32-63. Only affects pins configured as outputs via DIRB. The bit positions map to pins 32-63, where bit 0 controls pin 32 and bit 31 controls pin 63. When multiple cogs drive the same pin, the outputs are OR'd together.

Example:

```
        MOV     OUTB, pattern           ' Set output pattern for pins 32-63
        ANDN    OUTB, mask              ' Clear specific outputs
        OR      OUTB, ##$8000_0000      ' Set pin 63 high
        XOR     OUTB, toggle_mask       ' Toggle specific pins
```

Related: OUTA, DIRB, INB

INA

Address $1FE. Input register A for pins 0-31. Reads the current state of pins regardless of direction setting.

Access: Read-only for pin states (overlaid as IJMP0, R/W, during a debug ISR)

Bit Field:

Bits	Name	Description
31:0	IN	Current state of each pin: 1 = high, 0 = low

Usage: INA returns the actual electrical state of pins 0-31, regardless of whether they are configured as inputs or outputs. This allows output pins to be read back to verify their state. Reading INA captures the pin states at the moment the instruction executes, providing a consistent snapshot of all 32 pins. During a debug ISR, $1FE is overlaid as IJMP0 — the debug-interrupt jump address — and becomes read/write (it is initialized to $1F8, the debug-ISR-entry routine, on **COGINIT**).

Example:

```
1                   MOV     state, INA              ' Read all pins 0-31
2                   TEST    INA, #1             wz  ' Test if pin 0 is high
3           if_nz   JMP     #pin_high
4
5                   AND     inputs, INA             ' Mask input pins
6
7                   ' Wait for pin high
8   .wait           TEST    INA, pin_mask       wz
9           if_z    JMP     #.wait
```

Related: INB, DIRA, OUTA

INB

Address $1FF. Input register B for pins 32-63. Reads the current state of pins regardless of direction setting.

Access: Read-only for pin states (overlaid as IRET0, R/W, during a debug ISR)

Bit Field:

Bits	Name	Description
31:0	IN	Current state of each pin: 1 = high, 0 = low

Usage: INB returns the actual electrical state of pins 32-63, regardless of whether they are configured as inputs or outputs. The bit positions map to pins 32-63, where bit 0 represents pin 32 and bit 31 represents pin 63. During a debug ISR, $1FF is overlaid as IRET0 (the debug-interrupt return address) and becomes read/write.

Example:

```
                MOV     state, INB              ' Read all pins 32-63
                TEST    INB, ##$8000_0000   wz  ' Test if pin 63 is high
        if_z    JMP     #pin_low

                ' Copy input pattern to output
                MOV     OUTB, INB
```

Related: INA, DIRB, OUTB

Non-Memory-Mapped Registers

Several critical registers exist outside the cog RAM address space and are accessed only through specific instructions.

Program Counter (PC)

The program counter is a 20-bit register that holds the Hub RAM address of the currently executing instruction.

Access: Read via GETPC, modified implicitly by jumps and calls

Range: $00000-$FFFFF (full Hub address space)

Usage: The PC automatically increments by 4 after each instruction execution, pointing to the next long-aligned instruction in Hub RAM. Jump and **CALL** instructions modify the PC to change program flow. The PC wraps at the 20-bit boundary when incremented beyond $FFFFF.

Example:

```
        getpc   current_addr            ' Read current PC value

        ' PC modified by control flow
        JMP     #target                 ' Sets PC to target address
        CALL    #subroutine             ' Saves PC+4, jumps to subroutine
```

Related: GETPC, JMP, CALL, CALLD

Q Register

The Q register is a 32-bit auxiliary register used for CORDIC operations, division results, and block transfer setup.

Access: Read via **GETQX**/GETQY, write via **SETQ**/SETQ2

Usage: The Q register serves multiple purposes:

1. **CORDIC results**: After CORDIC operations (QROTATE, **QVECTOR**, etc.), results are read from Q using **GETQX** and **GETQY**.
2. **Division quotient**: Division instructions place the quotient in Q.

3. **Block operations**: **SETQ** and **SETQ2** configure the Q register to enable multi-long transfers with RDxxxx/WRxxxx instructions.

The Q register contents are volatile—CORDIC and division operations overwrite previous values. Read results immediately after the operation completes.

Example:

```
        SETQ    y                       ' Y coordinate via Q
        QROTATE x, angle                ' Rotate (X, Y) by angle
        GETQX   result_x                ' Get X result from Q
        GETQY   result_y                ' Get Y result from Q

        ' Block transfer setup
        SETQ    #15                     ' Setup for 16-long transfer
        RDLONG  buffer, ptra++          ' Read 16 longs using Q count

        ' Division
        QDIV    dividend, divisor       ' Quotient goes to Q
        GETQX   quotient                ' Read quotient from Q
        GETQY   remainder               ' Read remainder from Q
```

Related: GETQX, GETQY, SETQ, SETQ2, QROTATE, QVECTOR, QDIV

System Counter (CT)

The system counter is a free-running 64-bit counter (Rev B/C silicon) that increments on every system clock cycle. It is global across all cogs—all cogs reading CT simultaneously receive the same value. **GETCT** returns the lower 32 bits by default, or the upper 32 bits with WC.

Access: Read via **GETCT**, used by ADDCT1/ADDCT2/ADDCT3 and WAITCT1/WAITCT2/WAITCT3

Resolution: System clock cycles (typically 200 MHz = 5ns resolution)

Usage: CT provides precise timing for delays, timeouts, and event synchronization. The lower 32 bits wrap approximately every 21.5 seconds at 200 MHz. For precise waits, read the current CT value, add the desired delay to compute a target time, and wait for CT to reach that target. This approach compensates for instruction execution time between reading CT and initiating the wait.

Example:

```
                GETCT   target                  ' Get current time
                ADDCT1  target, ##delay_cycles  ' target = now + delay
                WAITCT1                         ' Wait for CT to reach it

                ' Timeout pattern
                GETCT   timeout
                ADD     timeout, ##max_cycles
.loop           ' ... do work ...
```

continues on next page →

↪ *continued from previous page*

```
                GETCT   now
                CMP     now, timeout        wc  ' Check if timeout exceeded
        if_nc   JMP     #timed_out
                JMP     #.loop
```

Related: GETCT, ADDCT1, ADDCT2, ADDCT3, WAITCT1, WAITCT2, WAITCT3

Hardware Random Number Generator (RANDOM)

The hardware random number generator produces true random numbers based on thermal noise, providing a new random value on each read.

Access: Read via **GETRND**

Features: True random number generation (not pseudo-random), continuously generates new values

Usage: Each execution of **GETRND** returns a new 32-bit random value. The generator runs continuously in hardware, so consecutive reads produce different values. The randomness quality is suitable for cryptographic applications.

Example:

```
        GETRND  random_value            ' Get 32-bit random number

        ' Generate random in range 0-99
        GETRND  temp
        QMUL    temp, #100              ' Multiply by 100
        GETQY   random_0_99             ' High 32 bits = value*100/2^32

        ' Random bit
        GETRND  temp
        SHR     temp, #31               ' Get bit 31 (random 0 or 1)
```

Related: GETRND, QMUL (for scaling random values)

C and Z Flags

The carry (C) and zero (Z) flags are 1-bit condition flags that store the results of tests and arithmetic operations.

Access: Set by instructions with WC, WZ, or WCZ effects; tested by conditional instruction execution

Persistence: Flags maintain their values until explicitly modified by another instruction with WC/WZ/WCZ

Usage: The C and Z flags enable conditional execution and branching. Most ALU instructions can update these flags based on their results. Conditional prefixes (IF_Z, IF_NZ, IF_C, IF_NC, etc.) determine whether an instruction executes based on flag states.

Flag Setting:

- **WZ**: Sets Z flag based on result (Z=1 if result is zero)

- **WC**: Sets C flag based on operation (carry out, bit shifted out, etc.)
- **WCZ**: Sets both flags

Example:

```
                CMP     value, #100         wz   ' Compare, set Z if equal
        if_z    JMP     #equal

                TEST    flags, ##$8000_0000 wc   ' Test bit 31, put in C
        if_c    JMP     #bit_set

                ADD     sum, addend         wc   ' Add, set C if overflow
        if_c    JMP     #overflow

                SHR     data, #1            wc   ' Shift right, C = bit out
```

Related: All conditional execution (IF_xx), CMP, TEST, and ALU instructions with WC/WZ/WCZ

Common Usage Patterns

Pin Control

Toggle a pin:

```
        XOR     OUTA, pin_mask          ' Toggle pin atomically
```

Wait for pin high:

```
.wait           TEST    INA, pin_mask       wz
        if_z    JMP     #.wait
```

Copy inputs to outputs:

```
        MOV     OUTA, INA               ' Mirror inputs to outputs
```

Set multiple pins atomically:

```
        MOV     OUTA, new_pattern       ' All 32 pins change simultaneously
```

Hub RAM Access

Block read with pointer:

```
        MOV     ptra, ##hub_buffer
        SETQ    #count-1                ' Transfer 'count' longs
        RDLONG  cog_buffer, ptra++      ' Read block, auto-increment PTRA
```

Dual buffer operation:

```
        MOV     ptra, ##source_buffer
        MOV     ptrb, ##dest_buffer
        SETQ    #15                     ' Transfer 16 longs
        RDLONG  temp, ptra++            ' Read from PTRA
        SETQ    #15
        WRLONG  temp, ptrb++            ' Write to PTRB
```

Interrupt Setup

Configure interrupt handler:

```
        MOV     IJMP1, ##handler_addr   ' Set handler address
        SETINT1 #event_ct1              ' Enable INT1 for CT1 event

handler_addr
        ' ... handle interrupt ...
        RETI1                           ' Return to interrupted code
```

Timing Operations

Precise delay:

```
        GETCT   target                  ' Get current time
        ADDCT1  target, ##delay_cycles  ' Add delay
        WAITCT1                         ' Wait until target time
```

Timeout detection:

```
                GETCT   deadline
                ADD     deadline, ##max_time
.loop           ' ... do work ...
                GETCT   now
```

continues on next page →

↪ *continued from previous page*

```
5                   CMP     now, deadline       wc
6           if_nc   JMP     #timeout
7                   ' ... continue if not timed out ...
8                   JMP     #.loop
```

Important Behaviors

Multi-Cog Pin Control: When multiple cogs drive the same pin as an output, the pin outputs are OR'd together. If any cog outputs high, the pin goes high. This enables cooperative control but requires coordination to avoid conflicts.

Smart Pin Override: When a pin is configured for smart pin operation, the smart pin mode overrides the basic DIRA/OUTA/INA functions for that pin. The pin is controlled through smart pin registers and commands rather than the basic I/O registers.

Immediate Effect: Changes to DIR and OUT registers take effect immediately—the hardware updates pin states on the same clock cycle as the register write.

Input Reading: INA and INB always return actual pin states, regardless of direction settings. This allows outputs to be read back for verification.

Pointer Auto-Modification: When using PTRA++ or PTRB++ addressing modes, the pointer update occurs after the memory access completes. The modification affects subsequent operations using that pointer.

PC Wrap Behavior: The program counter wraps at the 20-bit boundary ($FFFFF → $00000). Code executing near the top of Hub RAM must account for this wrap behavior.

Per-Cog Independence: Each cog has its own independent copy of all special registers. Changes in one cog do not affect other cogs' registers, enabling parallel independent operation.

Part III: Reference Tables

Appendix A: Instruction Encoding Master Table

This appendix provides the complete encoding reference for all PASM2 instructions in alphabetical order.

Reading This Table

Column	Description
Instruction	Mnemonic name
Opcode	7-bit binary pattern (bits 21-27 of instruction word) (bits 28-31 are the EEEE condition-code field; see Appendix B)
CZI	Available effects (C=WC, Z=WZ, I=immediate)
Cycles	Execution time in clock cycles
C Effect	What C flag indicates after instruction execution
Z Effect	What Z flag indicates after instruction execution

Flag Effect Notation:

- `---` indicates the flag is not affected by the instruction
- `Result = 0` means the flag is set if the result equals zero
- Specific conditions are described where applicable

Instruction Encodings

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
ABS	0110010	CZI	2	S[31]	Result = 0
ADD	0001000	CZI	2	carry of (D + S)	Result = 0
ADDCT1	1010011	—	2	—	—
ADDCT2	1010011	—	2	—	—
ADDCT3	1010011	—	2	—	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
ADDPIX	1010010	—	7	—	—
ADDS	0001010	CZI	2	sign of (D + S)	Result = 0
ADDSX	0001011	CZI	2	sign of (D+S+C)	Z AND (Result = 0)
ADDX	0001001	CZI	2	carry of (D + S + C)	Z AND (result == 0)
AKPIN	1100000	—	2	—	—
ALLOWI	1101011	—	2	—	—
ALTB	1001100	—	2	—	—
ALTD	1001100	—	2	—	—
ALTGB	1001011	—	2	—	—
ALTGN	1001010	—	2	—	—
ALTGW	1001011	—	2	—	—
ALTI	1001101	—	2	—	—
ALTR	1001100	—	2	—	—
ALTS	1001100	—	2	—	—
ALTSB	1001011	—	2	—	—
ALTSN	1001010	—	2	—	—
ALTSW	1001011	—	2	—	—
AND	0101000	CZI	2	parity of result	Result = 0
ANDN	0101001	CZI	2	parity of result	Result = 0
ASMCLK	---	—	—	—	—
AUGD	1111100	—	2	—	—
AUGS	1111000	—	2	—	—
BITC	0100010	CZI	2	—	original D[S[4:0]]
BITH	0100001	CZI	2	—	original D[S[4:0]]
BITL	0100000	CZI	2	—	original D[S[4:0]]
BITNC	0100011	CZI	2	—	original D[S[4:0]]
BITNOT	0100111	CZI	2	—	original D[S[4:0]]
BITNZ	0100101	CZI	2	—	original D[S[4:0]]
BITRND	0100110	CZI	2	Original D base bit	Original D base bit
BITZ	0100100	CZI	2	—	original D[S[4:0]]
BLNPIX	1010010	—	7	—	—
BMASK	1001110	—	2	—	—
BRK	1101011	—	2	—	—
CALL	1101101	—	4 / 13-20	—	—
CALLA	1101011	CZ	5…12 *	D[31]	D[30]
CALLB	1101011	CZ	5…12 *	D[31]	D[30]

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
CALLD	1011001	CZI	4 / 13-20	—	—
CALLPA	1011010	—	4 / 13–20	—	—
CALLPB	1011010	—	4 / 13–20	—	—
CMP	0010000	CZI	2	Unsigned (D < S)	D=S
CMPM	0010101	CZI	2	Result[31]	D=S
CMPR	0010100	CZI	2	borrow of (S - D)	(D == S)
CMPS	0010010	CZI	2	Signed (D < S)	D=S
CMPSUB	0010111	CZI	2	Unsigned(D => S)	Result = 0
CMPSX	0010011	CZI	2	correct sign of (D - (S + C))	Z AND (D == S + C)
CMPX	0010001	CZI	2	borrow of (D - (S + C))	Z AND (D == S + C)
COGATN	1101011	—	2	—	—
COGBRK	1101011	—	2	—	—
COGID	1101011	C	2–9, +2 if result	Cog Running	—
COGINIT	1100111	C	2–9, +2 if result	No cog available	—
COGSTOP	1101011	—	2–9	—	—
CRCBIT	1001110	—	2	—	—
CRCNIB	1001110	—	2	—	—
DEBUG	---	—	—	—	—
DECMOD	0111001	CZI	2	Modulus triggered	Result = 0
DECOD	1001110	—	2	—	—
DIRC	1101011	CZ	2	—	DIR bit
DIRH	1101011	CZ	2	—	DIR bit
DIRL	1101011	CZ	2	—	DIR bit
DIRNC	1101011	CZ	2	—	DIR bit
DIRNOT	1101011	CZ	2	—	DIR bit
DIRNZ	1101011	CZ	2	—	DIR bit
DIRRND	1101011	CZ	2	Original DIRx base bit	Original DIRx base bit
DIRZ	1101011	CZ	2	—	DIR bit
DJF	1011011	—	2 or 4	—	—
DJNF	1011011	—	2 or 4	—	—
DJNZ	1011011	—	2 or 4	—	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
DJZ	1011011	—	2 or 4	—	—
DRVC	1101011	CZ	2	—	OUT bit
DRVH	1101011	CZ	2	—	OUT bit
DRVL	1101011	CZ	2	—	OUT bit
DRVNC	1101011	CZ	2	—	OUT bit
DRVNOT	1101011	CZ	2	—	OUT bit
DRVNZ	1101011	CZ	2	—	OUT bit
DRVRND	1101011	CZ	2	Original OUTx base bit	Original OUTx base bit
DRVZ	1101011	CZ	2	—	OUT bit
ENCOD	0111100	CZI	2	S != 0	Result = 0
EXECF	1101011	—	4	—	—
FBLOCK	1100100	—	2	—	—
FGE	0011000	CZI	2	limit enforced	Result = 0
FGES	0011010	CZI	2	limit enforced	Result = 0
FLE	0011001	CZI	2	limit enforced	Result = 0
FLES	0011011	CZI	2	limit enforced	Result = 0
FLTC	1101011	CZ	2	—	OUT bit
FLTH	1101011	CZ	2	—	OUT bit
FLTL	1101011	CZ	2	—	OUT bit
FLTNC	1101011	CZ	2	—	OUT bit
FLTNOT	1101011	CZ	2	—	OUT bit
FLTNZ	1101011	CZ	2	—	OUT bit
FLTRND	1101011	CZ	2	Original OUTx base bit	Original OUTx base bit
FLTZ	1101011	CZ	2	—	OUT bit
GETBRK	1101011	CZ	2	—	—
GETBYTE	1000111	—	2	—	—
GETCT	1101011	C	2	same	—
GETNIB	1000010	—	2	—	—
GETPTR	1101011	—	2	—	—
GETQX	1101011	CZ	2…58	X[31]	Result = 0
GETQY	1101011	CZ	2…58	Y[31]	Result = 0
GETRND	1101011	CZ	2	RND[31]	RND[30], unique per cog
GETSCP	1101011	—	2	—	—
GETWORD	1001001	—	2	—	—
GETXACC	1101011	—	2	—	—
HUBSET	1101011	—	2…9	—	—
IJNZ	1011100	—	2 or 4	—	—
IJZ	1011100	—	2 or 4	—	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
INCMOD	0111000	CZI	2	1, else D = D + 1 and C = 0	Result = 0
JATN	1011110	—	2 or 4	—	—
JCT1	1011110	—	2 or 4	—	—
JCT2	1011110	—	2 or 4	—	—
JCT3	1011110	—	2 or 4	—	—
JFBW	1011110	—	2 or 4	—	—
JINT	1011110	—	2 or 4	—	—
JMP	1101011	CZ	4	D[31]	D[30]
JMPREL	1101011	—	4	—	—
JNATN	1011110	—	2 or 4	—	—
JNCT1	1011110	—	2 or 4	—	—
JNCT2	1011110	—	2 or 4	—	—
JNCT3	1011110	—	2 or 4	—	—
JNFBW	1011110	—	2 or 4	—	—
JNINT	1011110	—	2 or 4	—	—
JNPAT	1011110	—	2 or 4	—	—
JNQMT	1011110	—	2 or 4	—	—
JNSE1	1011110	—	2 or 4	—	—
JNSE2	1011110	—	2 or 4	—	—
JNSE3	1011110	—	2 or 4	—	—
JNSE4	1011110	—	2 or 4	—	—
JNXFI	1011110	—	2 or 4	—	—
JNXMT	1011110	—	2 or 4	—	—
JNXRL	1011110	—	2 or 4	—	—
JNXRO	1011110	—	2 or 4	—	—
JPAT	1011110	—	2 or 4	—	—
JQMT	1011110	—	2 or 4	—	—
JSE1	1011110	—	2 or 4	—	—
JSE2	1011110	—	2 or 4	—	—
JSE3	1011110	—	2 or 4	—	—
JSE4	1011110	—	2 or 4	—	—
JXFI	1011110	—	2 or 4	—	—
JXMT	1011110	—	2 or 4	—	—
JXRL	1011110	—	2 or 4	—	—
JXRO	1011110	—	2 or 4	—	—
LOC	1110100	—	2	—	—
LOCKNEW	1101011	C	4…11	1 if no LOCK available	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
LOCKREL	1101011	C	2…9, +2 if result	—	—
LOCKRET	1101011	—	2…9	—	—
LOCKTRY	1101011	C	2…9, +2 if result	1 if got LOCK	—
MERGEB	1101011	—	2	—	—
MERGEW	1101011	—	2	—	—
MIXPIX	1010010	—	7	—	—
MODC	1101011	—	2	cccc[C,Z]	—
MODCZ	1101011	—	2	cccc[C,Z]	zzzz[C,Z]
MODZ	1101011	—	2	—	zzzz[C,Z]
MOV	0110000	CZI	2	S[31]	Result = 0
MOVBYTS	1001111	—	2	—	—
MUL	1010000	I	2	—	(D = 0) OR (S = 0)
MULPIX	1010010	—	7	—	—
MULS	1010000	I	2	—	(D = 0) OR (S = 0)
MUXC	0101100	CZI	2	parity of result	Result = 0
MUXNC	0101101	CZI	2	parity of result	Result = 0
MUXNIBS	1001111	—	2	—	—
MUXNITS	1001111	—	2	—	—
MUXNZ	0101111	CZI	2	parity of result	Result = 0
MUXQ	1001111	—	2	—	—
MUXZ	0101110	CZI	2	parity of result	Result = 0
NEG	0110011	CZI	2	Sign of result	Result = 0
NEGC	0110100	CZI	2	Sign of result	Result = 0
NEGNC	0110101	CZI	2	Sign of result	Result = 0
NEGNZ	0110111	CZI	2	Sign of result	Result = 0
NEGZ	0110110	CZI	2	Sign of result	Result = 0
NIXINT1	1101011	—	2	—	—
NIXINT2	1101011	—	2	—	—
NIXINT3	1101011	—	2	—	—
NOP	0000000	—	2	—	—
NOT	0110001	CZI	2	!S[31]	Result = 0
ONES	0111101	CZI	2	Result is odd	Result = 0
OR	0101010	CZI	2	Parity of Result	Result = 0
OUTC	1101011	CZ	2	—	OUT bit
OUTH	1101011	CZ	2	—	OUT bit

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
OUTL	1101011	CZ	2	—	OUT bit
OUTNC	1101011	CZ	2	—	OUT bit
OUTNOT	1101011	CZ	2	—	OUT bit
OUTNZ	1101011	CZ	2	—	OUT bit
OUTRND	1101011	CZ	2	Original OUTx base bit	Original OUTx base bit
OUTZ	1101011	CZ	2	—	OUT bit
POLLATN	1101011	—	2	ATN Event	ATN Event
POLLCT1	1101011	—	2	CT1 Event	CT1 Event
POLLCT2	1101011	—	2	CT2 Event	CT2 Event
POLLCT3	1101011	—	2	CT3 Event	CT3 Event
POLLFBW	1101011	—	2	FBW Event	FBW Event
POLLINT	1101011	—	2	INT Event	INT Event
POLLPAT	1101011	—	2	PAT Event	PAT Event
POLLQMT	1101011	—	2	QMT Event	QMT Event
POLLSE1	1101011	—	2	SE1 Event	SE1 Event
POLLSE2	1101011	—	2	SE2 Event	SE2 Event
POLLSE3	1101011	—	2	SE3 Event	SE3 Event
POLLSE4	1101011	—	2	SE4 Event	SE4 Event
POLLXFI	1101011	—	2	XFI Event	XFI Event
POLLXMT	1101011	—	2	XMT Event	XMT Event
POLLXRL	1101011	—	2	XRL Event	XRLEvent
POLLXRO	1101011	—	2	XRO Event	XRO Event
POP	1101011	CZ	2	K[31]	Result = 0
POPA	1011000	CZ	9…16 *	MSB of long	Result = 0
POPB	1011000	CZ	9…16 *	MSB of long	Result = 0
PUSH	1101011	—	2	—	—
PUSHA	1100011	—	3…10*	—	—
PUSHB	1100011	—	3…10*	—	—
QDIV	1101000	—	2…9	—	—
QEXP	1101011	—	2…9	—	—
QFRAC	1101001	—	2…9	—	—
QLOG	1101011	—	2…9	—	—
QMUL	1101000	—	2…9	—	—
QROTATE	1101010	—	2…9	—	—
QSQRT	1101001	—	2…9	—	—
QVECTOR	1101010	—	2…9	—	—
RCL	0000101	CZI	2	last bit shifted out if S[4:0] > 0, else D[31]	Result = 0

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
RCR	0000100	CZI	2	Last bit out1	Result = 0
RCZL	1101011	CZ	2	D[31]	D[30]
RCZR	1101011	CZ	2	D[1]	D[0]
RDBYTE	1010110	CZI	9…16	MSB of byte	Result = 0
RDFAST	1100011	—	2 or WRFAST finish + 10…17	—	—
RDLONG	1011000	CZI	9…16 *	MSB of long	—
RDLUT	1010101	CZI	3	MSB of data	Result = 0
RDPIN	1010100	C	2	modal result	—
RDWORD	1010111	CZI	9…16 *	MSB of word	Result = 0
REP	1100110	—	2	—	—
RESI0	1011001	—	4	—	—
RESI1	1011001	—	4	—	—
RESI2	1011001	—	4	—	—
RESI3	1011001	—	4	—	—
RET	1101011	—	4	K[31]	K[30]
RETA	1101011	—	11…18 *	L[31]	L[30]
RETB	1101011	—	11…18 *	L[31]	L[30]
RETI0	1011001	—	4	—	—
RETI1	1011001	—	4	—	—
RETI2	1011001	—	4	—	—
RETI3	1011001	—	4	—	—
REV	1101011	—	2	—	—
RFBYTE	1101011	CZ	2	MSB of byte	Result = 0
RFLONG	1101011	CZ	2	MSB of long	Result = 0
RFVAR	1101011	CZ	2	0	Result = 0
RFVARS	1101011	CZ	2	MSB of value	Result = 0
RFWORD	1101011	CZ	2	MSB of word	Result = 0
RGBEXP	1101011	—	2	—	—
RGBSQZ	1101011	—	2	—	—
ROL	0000001	CZI	2	last bit shifted out if S[4:0] > 0, else D[31]	Result = 0
ROLBYTE	1001000	—	2	—	—
ROLNIB	1000100	—	2	—	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
ROLWORD	1001010	—	2	—	—
ROR	0000000	CZI	2	last bit shifted out if S[4:0] > 0, else D[0]	Result = 0
RQPIN	1010100	C	2	modal result	—
SAL	0000111	CZI	2	last bit shifted out if S[4:0] > 0, else D[31]	Result = 0
SAR	0000110	CZI	2	last bit shifted out if S[4:0] > 0, else D[0]	Result = 0
SCA	1010001	I	2	—	Product = 0
SCAS	1010001	I	2	—	Result = 0
SETBYTE	1000110	—	2	—	—
SETCFRQ	1101011	—	2	—	—
SETCI	1101011	—	2	—	—
SETCMOD	1101011	—	2	—	—
SETCQ	1101011	—	2	—	—
SETCY	1101011	—	2	—	—
SETD	1001101	—	2	—	—
SETDACS	1101011	—	2	—	—
SETINT1	1101011	—	2	—	—
SETINT2	1101011	—	2	—	—
SETINT3	1101011	—	2	—	—
SETLUTS	1101011	—	2	—	—
SETNIB	1000000	—	2	—	—
SETPAT	1011111	—	2	—	—
SETPIV	1101011	—	2	—	—
SETPIX	1101011	—	2	—	—
SETQ	1101011	—	2	—	—
SETQ2	1101011	—	2	—	—
SETR	1001101	—	2	—	—
SETS	1001101	—	2	—	—
SETSCP	1101011	—	2	—	—
SETSE1	1101011	—	2	—	—
SETSE2	1101011	—	2	—	—
SETSE3	1101011	—	2	—	—
SETSE4	1101011	—	2	—	—
SETWORD	1001001	—	2	—	—
SETXFRQ	1101011	—	2	—	—
SEUSSF	1101011	—	2	—	—
SEUSSR	1101011	—	2	—	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
SHL	0000011	CZI	2	last bit shifted out if S[4:0] > 0, else D[31]	Result = 0
SHR	0000010	CZI	2	last bit shifted out if S[4:0] > 0, else D[0]	Result = 0
SIGNX	0111011	CZI	2	MSB of result	Result = 0
SKIP	1101011	—	2	—	—
SKIPF	1101011	—	2	—	—
SPLITB	1101011	—	2	—	—
SPLITW	1101011	—	2	—	—
STALLI	1101011	—	2	—	—
SUB	0001100	CZI	2	borrow of (D - S)	Result = 0
SUBR	0010110	CZI	2	borrow of (S - D)	Result = 0
SUBS	0001110	CZI	2	sign of (D - S)	Result = 0
SUBSX	0001111	CZI	2	sign of D-(S+C)	Z AND (Result = 0)
SUBX	0001101	CZI	2	borrow of (D - (S + C))	Z AND (result == 0)
SUMC	0011100	CZI	2	1 then D = D - S, else D = D + S. C = correct sign of (D +/- S)	Result = 0
SUMNC	0011101	CZI	2	0 then D = D - S, else D = D + S. C = correct sign of (D +/- S)	Result = 0
SUMNZ	0011111	CZI	2	correct sign of (D +/- S)	0 then D = D - S, else D = D + S
SUMZ	0011110	CZI	2	correct sign of (D +/- S)	1 then D = D - S, else D = D + S
TEST	0111110	CZ	2	Parity of (D & S)	(D & S) = 0
TESTB	0100000	CZI	2	D[S[4:0]]	D[S[4:0]]
TESTBN	0100001	CZI	2	!D[S[4:0]]	!D[S[4:0]]
TESTN	0111111	CZI	2	Parity of (D & !S)	(D & !S) = 0
TESTP	1101011	CZ	2	IN[D[5:0]]	IN[D[5:0]]
TESTPN	1101011	CZ	2	!IN[D[5:0]]	!IN[D[5:0]]
TJF	1011101	—	2 or 4	—	—
TJNF	1011101	—	2 or 4 / 2 or 13-20	—	—
TJNS	1011101	—	2 or 4	—	—
TJNZ	1011100	—	2 or 4	—	—
TJS	1011101	—	2 or 4 / 2 or 13-20	—	—

(continued on next page)

Instruction Encodings (continued)

Instruction	Opcode	CZI	Cycles	C Effect	Z Effect
TJV	1011110	—	2 or 4 / 2 or 13–20	—	—
TJZ	1011100	—	2 or 4	—	—
TRGINT1	1101011	—	2	—	—
TRGINT2	1101011	—	2	—	—
TRGINT3	1101011	—	2	—	—
WAITATN	1101011	—	2+	Timeout Abort	Timeout Abort
WAITCT1	1101011	—	2+	timeout	timeout
WAITCT2	1101011	—	2+	timeout	timeout
WAITCT3	1101011	—	2+	Timeout Abort	Timeout Abort
WAITFBW	1101011	—	2+	Timeout Abort	Timeout Abort
WAITINT	1101011	—	2+	Timeout Abort	Timeout Abort
WAITPAT	1101011	—	2+	timeout	timeout
WAITSE1	1101011	—	2+	timeout	timeout
WAITSE2	1101011	—	2+	timeout	timeout
WAITSE3	1101011	—	2+	timeout	timeout
WAITSE4	1101011	—	2+	timeout	timeout
WAITX	1101011	CZ	2 + D	0	0
WAITXFI	1101011	—	2+	Timeout Abort	Timeout Abort
WAITXMT	1101011	—	2+	timeout	timeout
WAITXRL	1101011	—	2+	Timeout Abort	Timeout Abort
WAITXRO	1101011	—	2+	Timeout Abort	Timeout Abort
WFBYTE	1101011	—	2	—	—
WFLONG	1101011	—	2	—	—
WFWORD	1101011	—	2	—	—
WMLONG	1010011	—	3…10 *	—	—
WRBYTE	1100010	—	3…10	—	—
WRC	1101011	—	2	—	—
WRFAST	1100100	—	2 or WR-FAST finish + 3	—	—
WRLONG	1100011	—	3…10*	—	—
WRLUT	1100001	—	2	—	—
WRNC	1101011	—	2	—	—
WRNZ	1101011	—	2	—	—
WRPIN	1100000	—	2	—	—

(continued on next page)

Instruction Encodings (continued)					
Instruction	Opcode	**CZI**	**Cycles**	**C Effect**	**Z Effect**
WRWORD	1100010	—	3…10*	—	—
WRZ	1101011	—	2	—	—
WXPIN	1100000	—	2	—	—
WYPIN	1100001	—	2	—	—
XCONT	1100110	—	2+	—	—
XINIT	1100101	—	2	—	—
XOR	0101011	CZI	2	Parity of Result	Result = 0
XORO32	1101011	—	2	—	—
XSTOP	1100101	—	2	—	—
XZERO	1100101	—	2+	—	—
ZEROX	0111010	CZI	2	MSB of result	Result = 0

Total Instructions: 359 (357 with a fixed encoding + 2 without: **ASMCLK** and **DEBUG**)

Notes:

- This table shows the primary encoding for each instruction
- Instructions with multiple encoding forms show only the most common variant
- Multi-cycle instructions show ranges (e.g., `2...9`) where timing depends on:
  - Hub synchronization (variable wait for hub access)
  - Operation parameters (CORDIC solver iterations, streamer operations)
  - Memory location (cog vs. LUT vs. hub execution)
- The `*` symbol indicates hub memory access with variable timing
- See Part II (Instruction Reference) for complete encoding details and all variants
- **ASMCLK** is a pseudo-instruction (macro) and **DEBUG** is a debug directive; neither has a single fixed hardware encoding (ASMCLK expands to **HUBSET**/WAITX, **DEBUG** emits a debug **CALL** under -d)

Appendix B: Condition Code Reference

This appendix is the **canonical reference** for all P2 condition codes. The EEEE field (bits 31-28) of every instruction specifies one of sixteen conditions that control whether the instruction executes based on the current C and Z flag states.

Every instruction can be made conditional by prefixing it with one of these condition mnemonics. When the condition is false, the instruction does not execute but still consumes its normal execution time (2 clock cycles for most instructions).

B.1 Complete Condition Code Table

EEEE	Primary Mnemonic	Condition	All Aliases
0000	_RET_	Always + return	—
0001	IF_NC_AND_NZ	C=0 AND Z=0	IF_NZ_AND_NC, IF_GT, IF_A, IF_00
0010	IF_NC_AND_Z	C=0 AND Z=1	IF_Z_AND_NC, IF_01
0011	IF_NC	C=0	IF_GE, IF_AE, IF_0X
0100	IF_C_AND_NZ	C=1 AND Z=0	IF_NZ_AND_C, IF_10
0101	IF_NZ	Z=0	IF_NE, IF_X0
0110	IF_C_NE_Z	C!=Z	IF_Z_NE_C, IF_DIFF
0111	IF_NC_OR_NZ	C=0 OR Z=0	IF_NZ_OR_NC, IF_NOT_11
1000	IF_C_AND_Z	C=1 AND Z=1	IF_Z_AND_C, IF_11
1001	IF_C_EQ_Z	C=Z	IF_Z_EQ_C, IF_SAME
1010	IF_Z	Z=1	IF_E, IF_X1
1011	IF_NC_OR_Z	C=0 OR Z=1	IF_Z_OR_NC, IF_NOT_10
1100	IF_C	C=1	IF_LT, IF_B, IF_1X
1101	IF_C_OR_NZ	C=1 OR Z=0	IF_NZ_OR_C, IF_NOT_01
1110	IF_C_OR_Z	C=1 OR Z=1	IF_Z_OR_C, IF_LE, IF_BE, IF_NOT_00
1111	IF_ALWAYS	Always	—

B.2 Alias Categories

The P2 provides multiple aliases for the same condition codes, enabling programmers to express intent clearly in different contexts.

B.2.1 Comparison Aliases

After a comparison instruction (CMP or **CMPS**), condition aliases express relational comparisons. Two equivalent terminology styles are available—choose whichever reads best for your code:

Relationship	Magnitude Style	Arithmetic Style	Primary	Flag State
Greater than	IF_A (Above)	IF_GT (Greater Than)	IF_NC_AND_NZ	C=0, Z=0
Greater or equal	IF_AE (Above or Equal)	IF_GE (Greater or Equal)	IF_NC	C=0
Less than	IF_B (Below)	IF_LT (Less Than)	IF_C	C=1
Less or equal	IF_BE (Below or Equal)	IF_LE (Less or Equal)	IF_C_OR_Z	C=1 OR Z=1
Equal	IF_E	IF_E	IF_Z	Z=1
Not equal	IF_NE	IF_NE	IF_NZ	Z=0

Magnitude terminology (A = Above, B = Below) reads naturally with unsigned values like addresses, counts, and sizes.

Arithmetic terminology (GT = Greater Than, LT = Less Than) reads naturally with signed values like temperatures, positions, and deltas.

Both styles encode to the same condition codes—the choice is purely stylistic. Use whichever terminology makes your code's intent clearer.

B.2.2 Flag State Aliases

Express exact C/Z bit patterns directly:

Alias	C	Z	Primary
IF_00	0	0	IF_NC_AND_NZ
IF_01	0	1	IF_NC_AND_Z
IF_10	1	0	IF_C_AND_NZ
IF_11	1	1	IF_C_AND_Z
IF_0X	0	*	IF_NC
IF_1X	1	*	IF_C
IF_X0	*	0	IF_NZ
IF_X1	*	1	IF_Z

The asterisk (*) indicates "don't care"—the condition is true regardless of that flag's value.

B.2.3 Logical Aliases

Express logical relationships between flag states:

Alias	Meaning	Primary
IF_SAME	C equals Z	IF_C_EQ_Z
IF_DIFF	C differs from Z	IF_C_NE_Z
IF_NOT_00	Not both clear	IF_C_OR_Z
IF_NOT_01	Not (C=0, Z=1)	IF_C_OR_NZ
IF_NOT_10	Not (C=1, Z=0)	IF_NC_OR_Z
IF_NOT_11	Not both set	IF_NC_OR_NZ

B.2.4 Commutative Forms

These pairs are identical—the operand order in the name is interchangeable:

Form 1	Form 2
IF_NC_AND_NZ	IF_NZ_AND_NC
IF_NC_AND_Z	IF_Z_AND_NC
IF_C_AND_NZ	IF_NZ_AND_C
IF_C_AND_Z	IF_Z_AND_C
IF_NC_OR_NZ	IF_NZ_OR_NC
IF_NC_OR_Z	IF_Z_OR_NC
IF_C_OR_NZ	IF_NZ_OR_C
IF_C_OR_Z	IF_Z_OR_C
IF_C_EQ_Z	IF_Z_EQ_C
IF_C_NE_Z	IF_Z_NE_C

B.3 The _RET_ Condition (EEEE=0000)

The condition code 0000 (`_RET_`) has special behavior that differs from all other conditions. Unlike other condition codes which control whether the instruction executes, `_RET_` means: **"Always execute the instruction, then return if the instruction did not branch."**

B.3.1 Behavior

When an instruction has EEEE=0000:

1. **The instruction always executes** (condition 0000 means "always" for `_RET_`)
2. **If the instruction does not branch**: Return by popping stack[19:0] into PC

3. **If the instruction branches** (JMP, **CALL**, etc.): No return occurs—the branch takes precedence
4. **No context restore**: Unlike `RET WCZ`, the `_RET_` prefix does NOT restore C or Z flags from the stack

This is fundamentally different from the **RET** instruction, which optionally restores C and Z flags when WC/WZ/WCZ effects are specified.

B.3.2 Basic Usage

```
        _ret_   ADD     x, y            ' ADD then return (flags unchanged)
        _ret_   DRVNOT  #0              ' Toggle pin 0, then return
        _ret_   MOV     result, temp    ' Copy to result, then return
```

B.3.3 Branch Behavior

When `_RET_` prefixes a branching instruction, the branch executes normally but no return occurs because the instruction itself changed PC:

```
        _ret_   JMP     #somewhere      ' JMP executes, NO return
        _ret_   CALL    #subroutine     ' CALL executes, NO return
        _ret_   DJNZ    counter, #loop  ' Branch: no return; zero: return
```

For **DJNZ** and similar conditional branches: if the branch is taken, no return occurs; if the branch is not taken (counter reaches zero), the return executes.

B.3.4 XBYTE Bytecode Interpreter

The `_RET_` prefix with **SETQ** and **SETQ2** is essential for the XBYTE bytecode execution mechanism. When the top of the hardware stack holds $1FF, these combinations configure XBYTE mode:

```
' Start XBYTE: SETQ configures mode, returns to $1FF
        PUSH    #$1FF                   ' Push $1FF for XBYTE returns
        _ret_   SETQ    #$100           ' LUT base $100, then return

' Change XBYTE mode permanently
        _ret_   SETQ    ##$200           ' New LUT base for all bytecodes

' Change XBYTE mode for next bytecode only
        _ret_   SETQ2   ##$300           ' Temp LUT base for one bytecode
```

B.3.5 SKIP/SKIPF with _RET_

Both **SKIP** and **SKIPF** can be combined with `_RET_` to branch before a **SKIP** pattern begins:

```
        PUSH    #routine                ' Push target address
        _ret_   SKIPF   pattern         ' SKIPF then branch with skip active
```

B.3.6 Timing

The `_RET_` prefix **ADDS** overhead to the base instruction timing:

Execution Mode	Additional Cycles
COG/LUT	+2 cycles
Hub	+11 to +18 cycles

B.3.7 Single-Instruction Subroutines

The `_RET_` prefix enables efficient single-instruction subroutines:

```
toggle_pin0                             ' Subroutine: toggle pin 0
        _ret_   DRVNOT  #0              ' 2 + 2 return = 4 cycles

read_input                              ' Subroutine: read input
        _ret_   MOV     result, ina     ' MOV, then return
```

This is significantly faster than a separate instruction followed by **RET** (which would take at least 4 additional cycles).

B.4 Conditional Execution Timing

When a conditional instruction's condition is false, the instruction does not execute but still consumes 2 clock cycles. This provides deterministic timing—critical for real-time operations:

```
                CMP     a, b            wcz     ' 2 cycles - always
        if_z    MOV     result, #1              ' 2 cycles - if Z=1 or not
        if_nz   MOV     result, #0              ' 2 cycles - if Z=0 or not
                                                ' Total: always 6 cycles
```

This timing predictability enables branchless programming where instruction timing remains constant regardless of data values.

Appendix C: Categorical Instruction Index

This appendix organizes P2 instructions by functional category, helping you find instructions based on what you want to accomplish rather than by alphabetical order. Each instruction name links to its detailed reference in Part II.

For a quick overview of each category with compact instruction lists, see Instruction Categories in Part II.

Arithmetic Operations

Arithmetic instructions perform mathematical and logical operations on register values. This includes addition, subtraction, multiplication, comparisons, bitwise operations (AND, OR, **XOR**), bit manipulation, shifts, rotates, and data movement. This is the largest instruction category.

Table C.2.

Instruction	Description
ABS	Get absolute value of D into D
ADD	Add S into D
ADDS	Add S into D, signed
ADDSX	Add (S + C) into D, signed and extended
ADDX	Add (S + C) into D, extended
AND	AND S into D
ANDN	AND !S into D
BITC	Bits D[S[9:5]+S[4:0]:S[4:0]] = C
BITH	Bits D[S[9:5]+S[4:0]:S[4:0]] = 1
BITL	Bits D[S[9:5]+S[4:0]:S[4:0]] = 0
BITNC	Bits D[S[9:5]+S[4:0]:S[4:0]] = !C
BITNOT	Toggle bits D[S[9:5]+S[4:0]:S[4:0]]
BITNZ	Bits D[S[9:5]+S[4:0]:S[4:0]] = !Z
BITRND	Bits D[S[9:5]+S[4:0]:S[4:0]] = RNDs
BITZ	Bits D[S[9:5]+S[4:0]:S[4:0]] = Z
BMASK	Get LSB-justified bit mask of size (D[4:0] + 1) into D
CMP	Compare D to S
CMPM	Compare D to S, get MSB of difference into C
CMPR	Compare S to D (reverse)

Table C.2.

Instruction	Description
CMPS	Compare D to S, signed
CMPSUB	Compare and subtract S from D if D >= S
CMPSX	Compare D to (S + C), signed and extended
CMPX	Compare D to (S + C), extended
CRCBIT	Iterate CRC value in D using C and polynomial in S
CRCNIB	Iterate CRC value in D using Q[31:28] and polynomial in S
DECMOD	Decrement with modulus
DECOD	Decode D[4:0] into D
ENCOD	Get bit position of top-most '1' in D into D
FGE	Force D >= S
FGES	Force D >= S, signed
FLE	Force D <= S
FLES	Force D <= S, signed
GETBYTE	Get byte established by prior ALTGB instruction into D
GETNIB	Get nibble established by prior ALTGN instruction into D
GETWORD	Get word established by prior ALTGW instruction into D
INCMOD	Increment with modulus
LOC	Get {12'b0, address[19:0]} into PA/PB/PTRA/PTRB (per W)
MERGEB	Merge bits of bytes in D
MERGEW	Merge bits of words in D
MODC	Modify C according to cccc
MODCZ	Modify C and Z according to cccc and zzzz
MODZ	Modify Z according to zzzz
MOV	Move S into D
MOVBYTS	Move bytes within D, per S
MUL	D = unsigned (D[15:0] * S[15:0])
MULS	D = signed (D[15:0] * S[15:0])
MUXC	Mux C into each D bit that is '1' in S
MUXNC	Mux !C into each D bit that is '1' in S
MUXNIBS	For each non-zero nibble in S, copy that nibble into the corresponding D nibble
MUXNITS	For each non-zero bit pair in S, copy that bit pair into the corresponding D bits

Table C.2.

Instruction	Description
MUXNZ	Mux !Z into each D bit that is ‘1’ in S
MUXQ	Used after SETQ
MUXZ	Mux Z into each D bit that is ‘1’ in S
NEG	Negate D
NEGC	Negate D by C
NEGNC	Negate D by !C
NEGNZ	Negate D by !Z
NEGZ	Negate D by Z
NOT	Get !D into D
ONES	Get number of ’1’s in D into D
OR	OR S into D
RCL	Rotate carry left
RCR	Rotate carry right
RCZL	Rotate C,Z left through D
RCZR	Rotate C,Z right through D
REV	Reverse D bits
RGBEXP	Expand 5:6:5 RGB value in D[15:0] into 8:8:8 value in D[31:8]
RGBSQZ	Squeeze 8:8:8 RGB value in D[31:8] into 5:6:5 value in D[15:0]
ROL	Rotate left
ROLBYTE	Rotate-left byte established by prior ALTGB instruction into D
ROLNIB	Rotate-left nibble established by prior ALTGN instruction into D
ROLWORD	Rotate-left word established by prior ALTGW instruction into D
ROR	Rotate right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SCA	Next instruction’s S value = unsigned (D[15:0] * S[15:0]) » 16
SCAS	Next instruction’s S value = signed (D[15:0] * S[15:0]) » 14
SETBYTE	Set S[7:0] into byte established by prior ALTSB instruction
SETD	Set D field of D to S[8:0]
SETNIB	Set S[3:0] into nibble established by prior ALTSN instruction
SETR	Set R field of D to S[8:0]

Table C.2.

Instruction	Description
SETS	Set S field of D to S[8:0]
SETWORD	Set S[15:0] into word established by prior ALTSW instruction
SEUSSF	Relocate and periodically invert bits within D
SEUSSR	Relocate and periodically invert bits within D
SHL	Shift left
SHR	Shift right
SIGNX	Sign-extend D from bit S[4:0]
SPLITB	Split every 4th bit of D into bytes
SPLITW	Split odd/even bits of D into words
SUB	Subtract S from D
SUBR	Subtract D from S (reverse)
SUBS	Subtract S from D, signed
SUBSX	Subtract (S + C) from D, signed and extended
SUBX	Subtract (S + C) from D, extended
SUMC	Sum +/-S into D by C
SUMNC	Sum +/-S into D by !C
SUMNZ	Sum +/-S into D by !Z
SUMZ	Sum +/-S into D by Z
TEST	Test D
TESTB	Test bit S[4:0] of D, XOR into C/Z
TESTBN	Test bit S[4:0] of !D, XOR into C/Z
TESTN	Test D with !S
WRC	Write 0 or 1 to D, according to C
WRNC	Write 0 or 1 to D, according to !C
WRNZ	Write 0 or 1 to D, according to !Z
WRZ	Write 0 or 1 to D, according to Z
XOR	XOR S into D
XORO32	Iterate D with xoroshiro32+ PRNG algorithm
ZEROX	Zero-extend D above bit S[4:0]

Branching and Flow Control

Branch instructions control program flow by modifying the program counter. This category includes conditional and unconditional jumps, subroutine calls using stack or pointer registers, returns from subroutines and interrupts, **AND** instruction skipping/repeating mechanisms.

Jump Instructions

Instruction	Description
JMP	Jump to A
JMPREL	Jump ahead/back by D instructions

Call Instructions

Instruction	Description
CALL	Call to A by pushing {C, Z, 10'b0, PC[19:0]} onto stack
CALLA	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTRA++
CALLB	Call to A by writing {C, Z, 10'b0, PC[19:0]} to hub long at PTRB++
CALLD	Call to A by writing {C, Z, 10'b0, PC[19:0]} to PA/PB/PTRA/PTRB (per W)
CALLPA	Call to S by pushing return onto stack, copy D to PA
CALLPB	Call to S by pushing return onto stack, copy D to PB

Return Instructions

Instruction	Description
RET	Return by popping stack
RETA	Return by reading hub long at –PTRA
RETB	Return by reading hub long at –PTRB
RETI0	Return from INT0
RETI1	Return from INT1
RETI2	Return from INT2
RETI3	Return from INT3
RESI0	Resume from INT0
RESI1	Resume from INT1
RESI2	Resume from INT2
RESI3	Resume from INT3

Test and Branch Instructions

Table C.3.

Instruction	Description
TJF	Test D and jump to S if D is full ($FFFF_FFFF)
TJNF	Test D and jump to S if D is not full
TJNS	Test D and jump to S if D is not signed (D[31] = 0)
TJNZ	Test D and jump to S if D is not zero
TJS	Test D and jump to S if D is signed (D[31] = 1)
TJV	Test D and jump to S if D overflowed
TJZ	Test D and jump to S if D is zero
DJF	Decrement D and jump to S if result is $FFFF_FFFF
DJNF	Decrement D and jump to S if result is not $FFFF_FFFF
DJNZ	Decrement D and jump to S if result is not zero
DJZ	Decrement D and jump to S if result is zero
IJNZ	Increment D and jump to S if result is not zero
IJZ	Increment D and jump to S if result is zero

Skip and Repeat Instructions

Instruction	Description
SKIP	Skip instructions per D
SKIPF	Skip cog/LUT instructions fast per D
EXECF	Jump to D[9:0] in cog/LUT and set SKIPF pattern to D[31:10]
REP	Execute next D[8:0] instructions S times

Hub Memory Access

Hub memory instructions transfer data between cog registers and the shared 512KB hub RAM. This includes **BYTE**, **WORD**, and **LONG** access with various addressing modes, pointer-based operations using PTRA/PTRB, and high-speed FIFO streaming for bulk data transfers.

Hub RAM Read

Instruction	Description
POPA	Read long from hub address –PTRA into D
POPB	Read long from hub address –PTRB into D
RDBYTE	Read zero-extended byte from hub address into D
RDLONG	Read long from hub address into D
RDWORD	Read zero-extended word from hub address into D

Hub RAM Write

Instruction	Description
PUSHA	Write long in D to hub address PTRA++
PUSHB	Write long in D to hub address PTRB++
WMLONG	Write only non-$00 bytes in D to hub address
WRBYTE	Write byte in D[7:0] to hub address
WRLONG	Write long in D to hub address
WRWORD	Write word in D[15:0] to hub address

Hub FIFO

Instruction	Description
GETPTR	Get current FIFO hub pointer into D
RDFAST	Begin new fast hub read via FIFO
WRFAST	Begin new fast hub write via FIFO
FBLOCK	Set next block for when block wraps
RFBYTE	Read byte from FIFO (after RDFAST)
RFLONG	Read long from FIFO (after RDFAST)
RFVAR	Read variable-length value from FIFO
RFVARS	Read signed variable-length value from FIFO
RFWORD	Read word from FIFO (after RDFAST)
WFBYTE	Write byte to FIFO (after WRFAST)
WFLONG	Write long to FIFO (after WRFAST)
WFWORD	Write word to FIFO (after WRFAST)

Lookup Table

Lookup table (LUT) instructions access the 512-long LUT memory private to each cog. The LUT provides fast table lookups, additional register storage, and can be shared between adjacent cog pairs for inter-cog communication.

Instruction	Description
RDLUT	Read data from LUT address into D
SETLUTS	Enable/disable LUT sharing with adjacent cog
WRLUT	Write D to LUT address

Pin I/O and Smart Pins

Pin instructions control the P2's 64 I/O pins. Basic pin operations set direction (input/output) and output level (high/low). Smart pin instructions configure and communicate with the autonomous smart pin state machines that can perform complex I/O functions independent of cog processing.

Direction Control

Instruction	Description
DIRC	DIR bits of pins = C
DIRH	DIR bits of pins = 1 (output)
DIRL	DIR bits of pins = 0 (input)
DIRNC	DIR bits of pins = !C
DIRNOT	Toggle DIR bits of pins
DIRNZ	DIR bits of pins = !Z
DIRRND	DIR bits of pins = random
DIRZ	DIR bits of pins = Z

Output Control

Instruction	Description
OUTC	OUT bits of pins = C
OUTH	OUT bits of pins = 1 (high)
OUTL	OUT bits of pins = 0 (low)
OUTNC	OUT bits of pins = !C
OUTNOT	Toggle OUT bits of pins
OUTNZ	OUT bits of pins = !Z
OUTRND	OUT bits of pins = random
OUTZ	OUT bits of pins = Z

Drive Control (Direction + Output)

Instruction	Description
DRVC	Set pins to output, level = C
DRVH	Set pins to output high
DRVL	Set pins to output low
DRVNC	Set pins to output, level = !C
DRVNOT	Set pins to output, toggle level
DRVNZ	Set pins to output, level = !Z
DRVRND	Set pins to output, random level
DRVZ	Set pins to output, level = Z

Float Control (Input with Preset)

Instruction	Description
FLTC	Set pins to input, preset output = C
FLTH	Set pins to input, preset output high
FLTL	Set pins to input, preset output low
FLTNC	Set pins to input, preset output = !C
FLTNOT	Set pins to input, toggle preset output
FLTNZ	Set pins to input, preset output = !Z
FLTRND	Set pins to input, random preset output
FLTZ	Set pins to input, preset output = Z

Pin Testing

Instruction	Description
TESTP	Test IN bit of pin, XOR into C/Z
TESTPN	Test !IN bit of pin, XOR into C/Z

Smart Pin Control

Instruction	Description
AKPIN	Acknowledge smart pin (clear flag)
RDPIN	Read smart pin result, acknowledge
RQPIN	Read smart pin result, don't acknowledge
WRPIN	Set mode of smart pin
WXPIN	Set X parameter of smart pin
WYPIN	Set Y parameter of smart pin
SETDACS	Set all four DAC channels
GETSCP	Get four-channel oscilloscope samples
SETSCP	Set oscilloscope enable and input pin base

Events and Timing

Event instructions monitor and respond to system events including counter/timer triggers, smart pin signals, FIFO status, streamer conditions, and inter-cog attention signals. They provide configuration, polling, waiting, and conditional branching mechanisms for synchronization.

Event Configuration

Instruction	Description
ADDCT1	Set CT1 event to trigger on CT = D + S
ADDCT2	Set CT2 event to trigger on CT = D + S
ADDCT3	Set CT3 event to trigger on CT = D + S
SETPAT	Set pin pattern for PAT event
SETSE1	Set SE1 event configuration
SETSE2	Set SE2 event configuration
SETSE3	Set SE3 event configuration
SETSE4	Set SE4 event configuration

Event Polling

Table C.4.

Instruction	Description
POLLATN	Get ATN event flag into C/Z, then clear
POLLCT1	Get CT1 event flag into C/Z, then clear
POLLCT2	Get CT2 event flag into C/Z, then clear
POLLCT3	Get CT3 event flag into C/Z, then clear
POLLFBW	Get FBW event flag into C/Z, then clear
POLLINT	Get INT event flag into C/Z, then clear
POLLPAT	Get PAT event flag into C/Z, then clear
POLLQMT	Get QMT event flag into C/Z, then clear
POLLSE1	Get SE1 event flag into C/Z, then clear
POLLSE2	Get SE2 event flag into C/Z, then clear
POLLSE3	Get SE3 event flag into C/Z, then clear
POLLSE4	Get SE4 event flag into C/Z, then clear
POLLXFI	Get XFI event flag into C/Z, then clear
POLLXMT	Get XMT event flag into C/Z, then clear
POLLXRL	Get XRL event flag into C/Z, then clear
POLLXRO	Get XRO event flag into C/Z, then clear

Event Waiting

Table C.5.

Instruction	Description
WAITATN	Wait for ATN event flag, then clear
WAITCT1	Wait for CT1 event flag, then clear
WAITCT2	Wait for CT2 event flag, then clear
WAITCT3	Wait for CT3 event flag, then clear
WAITFBW	Wait for FBW event flag, then clear
WAITINT	Wait for INT event flag, then clear
WAITPAT	Wait for PAT event flag, then clear
WAITSE1	Wait for SE1 event flag, then clear

Table C.5.

Instruction	Description
WAITSE2	Wait for SE2 event flag, then clear
WAITSE3	Wait for SE3 event flag, then clear
WAITSE4	Wait for SE4 event flag, then clear
WAITXFI	Wait for XFI event flag, then clear
WAITXMT	Wait for XMT event flag, then clear
WAITXRL	Wait for XRL event flag, then clear
WAITXRO	Wait for XRO event flag, then clear

Event Branching

Table C.6.

Instruction	Description
JATN	Jump to S if ATN event flag is set
JCT1	Jump to S if CT1 event flag is set
JCT2	Jump to S if CT2 event flag is set
JCT3	Jump to S if CT3 event flag is set
JFBW	Jump to S if FBW event flag is set
JINT	Jump to S if INT event flag is set
JNATN	Jump to S if ATN event flag is clear
JNCT1	Jump to S if CT1 event flag is clear
JNCT2	Jump to S if CT2 event flag is clear
JNCT3	Jump to S if CT3 event flag is clear
JNFBW	Jump to S if FBW event flag is clear
JNINT	Jump to S if INT event flag is clear
JNPAT	Jump to S if PAT event flag is clear
JNQMT	Jump to S if QMT event flag is clear
JNSE1	Jump to S if SE1 event flag is clear
JNSE2	Jump to S if SE2 event flag is clear
JNSE3	Jump to S if SE3 event flag is clear
JNSE4	Jump to S if SE4 event flag is clear
JNXFI	Jump to S if XFI event flag is clear

Table C.6.

Instruction	Description
JNXMT	Jump to S if XMT event flag is clear
JNXRL	Jump to S if XRL event flag is clear
JNXRO	Jump to S if XRO event flag is clear
JPAT	Jump to S if PAT event flag is set
JQMT	Jump to S if QMT event flag is set
JSE1	Jump to S if SE1 event flag is set
JSE2	Jump to S if SE2 event flag is set
JSE3	Jump to S if SE3 event flag is set
JSE4	Jump to S if SE4 event flag is set
JXFI	Jump to S if XFI event flag is set
JXMT	Jump to S if XMT event flag is set
JXRL	Jump to S if XRL event flag is set
JXRO	Jump to S if XRO event flag is set

Inter-COG Attention

Instruction	Description
COGATN	Strobe attention of cogs whose bits are high in D[15:0]

Interrupts

Interrupt instructions control the cog's three-level interrupt system (INT1, INT2, INT3) plus the debug interrupt (INT0). This includes enabling/disabling interrupts, configuring interrupt sources, triggering software interrupts, and managing breakpoints for debugging.

Table C.7.

Instruction	Description
ALLOWI	Allow interrupts (default)
BRK	If in debug ISR, set next break condition to D
COGBRK	If in debug ISR, trigger breakpoint in cog D[3:0]
GETBRK	Get breakpoint/cog status into D
NIXINT1	Cancel INT1
NIXINT2	Cancel INT2

Table C.7.

Instruction	Description
NIXINT3	Cancel INT3
SETINT1	Set INT1 source to D[3:0]
SETINT2	Set INT2 source to D[3:0]
SETINT3	Set INT3 source to D[3:0]
STALLI	Stall interrupts
TRGINT1	Trigger INT1, regardless of STALLI mode
TRGINT2	Trigger INT2, regardless of STALLI mode
TRGINT3	Trigger INT3, regardless of STALLI mode

COG Control and Locks

COG control instructions manage cog operations including starting and stopping cogs, querying cog identity, and configuring hub-level system settings. Lock instructions provide mutex-style synchronization primitives for safe inter-cog resource sharing.

COG Control

Instruction	Description
COGID	Get cog ID (0 to 15) into D
COGINIT	Start cog selected by D
COGSTOP	Stop cog D[3:0]
HUBSET	Set hub configuration to D

Locks

Instruction	Description
LOCKNEW	Request a lock from the pool
LOCKREL	Release lock D[3:0]
LOCKRET	Return lock D[3:0] for reallocation
LOCKTRY	Try to get lock D[3:0]

CORDIC Coprocessor

CORDIC (Coordinate Rotation Digital Computer) instructions provide hardware-accelerated mathematical operations. The dedicated coprocessor performs multiplication, division, square root, trigonometric functions, logarithms, and coordinate transformations with high precision.

Instruction	Description
GETQX	Retrieve CORDIC result X into D
GETQY	Retrieve CORDIC result Y into D
QDIV	Begin CORDIC unsigned division
QEXP	Begin CORDIC logarithm-to-number conversion
QFRAC	Begin CORDIC fractional division
QLOG	Begin CORDIC number-to-logarithm conversion
QMUL	Begin CORDIC unsigned multiplication
QROTATE	Begin CORDIC rotation of point by angle
QSQRT	Begin CORDIC square root
QVECTOR	Begin CORDIC vectoring of point

Streamer

Streamer instructions control the cog's dedicated DMA engine that autonomously transfers data between hub memory, LUT, and I/O pins. The streamer is essential for high-bandwidth applications like video output, audio streaming, and bulk data movement.

Instruction	Description
GETXACC	Get Goertzel X and Y accumulators, clear them
SETXFRQ	Set streamer NCO frequency to D
XCONT	Buffer new streamer command, continue phase
XINIT	Issue streamer command immediately, zero phase
XSTOP	Stop streamer immediately
XZERO	Buffer new streamer command, zero phase

Color Space and Pixel Operations

Color space and pixel instructions provide hardware-accelerated graphics processing. The colorspace converter transforms between color representations (RGB, YUV). The pixel mixer performs alpha blending, color addition, and format conversions for video and graphics applications.

Color Space Converter

Instruction	Description
SETCFRQ	Set colorspace converter CFRQ parameter
SETCI	Set colorspace converter CI parameter
SETCMOD	Set colorspace converter CMOD parameter
SETCQ	Set colorspace converter CQ parameter
SETCY	Set colorspace converter CY parameter

Pixel Mixer

Instruction	Description
ADDPIX	Add bytes of S into bytes of D with saturation
BLNPIX	Alpha-blend bytes of S into bytes of D
MIXPIX	Mix bytes of S into bytes of D
MULPIX	Multiply bytes of S into bytes of D
SETPIV	Set BLNPIX/MIXPIX blend factor
SETPIX	Set MIXPIX mode

Instruction Modification

Instruction modification instructions (also known as register indirection) dynamically alter subsequent instructions by changing their source, destination, or bit index fields before execution. They enable register arrays, computed addressing, and self-modifying code patterns essential for efficient data structure access.

Instruction	Description
ALTB	Alter D field of next instruction to D[13:5]
ALTD	Alter D field of next instruction to D[8:0]
ALTGB	Alter subsequent GETBYTE/ROLBYTE instruction
ALTGN	Alter subsequent GETNIB/ROLNIB instruction
ALTGW	Alter subsequent GETWORD/ROLWORD instruction
ALTI	Execute D in place of next instruction
ALTR	Alter result register address of next instruction
ALTS	Alter S field of next instruction to D[8:0]
ALTSB	Alter subsequent SETBYTE instruction
ALTSN	Alter subsequent SETNIB instruction
ALTSW	Alter subsequent SETWORD instruction

Miscellaneous

Miscellaneous instructions provide utility functions including immediate value extension (AUGS/AUGD), stack operations, random number generation, system timer access, and delay insertion.

Instruction	Description
AUGD	Extend next instruction's D immediate to 32 bits
AUGS	Extend next instruction's S immediate to 32 bits
GETCT	Get CT[31:0] or CT[63:32] if WC into D
GETRND	Get random number into D and/or C/Z
NOP	No operation
POP	Pop stack into D
PUSH	Push D onto stack
SETQ	Set Q register to D
SETQ2	Set Q register to D (for LUT transfers)
WAITX	Wait 2 + D clocks

Effect Support Reference

Not all instructions support all flag effect modifiers (WC, WZ, WCZ). This section provides a quick reference for effect restrictions. Each instruction entry in Part II also documents its allowed effects.

Important: You cannot write `WC WZ` as separate tokens. Use `WCZ` to update both flags.

Effect Categories

Category	Allowed Effects	Reason
Full	WC, WZ, WCZ	Both flags have independent, meaningful values
WCZ only	WCZ	Both flags set to the same value
WC only	WC	Only C has a defined meaning
WZ only	WZ	Only Z has a defined meaning
Extended	WC, WZ, ANDC, ANDZ, ORC, ORZ, XORC, XORZ (no WCZ)	Bit/pin test with accumulation
None	(no effects)	No meaningful flag output

Instructions by Effect Support

Category	Count	Instructions
Full (WC/WZ/WCZ)	~300	ADD, SUB, CMP, AND, OR, XOR, MOV, SHL, SHR, and most other ALU operations
WCZ only	40	BITC, BITH, BITL, BITNC, BITNOT, BITNZ, BITRND, BITZ, DIRC, DIRH, DIRL, DIRNC, DIRNOT, DIRNZ, DIRRND, DIRZ, DRVC, DRVH, DRVL, DRVNC, DRVNOT, DRVNZ, DRVRND, DRVZ, FLTC, FLTH, FLTL, FLTNC, FLTNOT, FLTNZ, FLTRND, FLTZ, OUTC, OUTH, OUTL, OUTNC, OUTNOT, OUTNZ, OUTRND, OUTZ
WC only	9	COGID, COGINIT, GETCT, LOCKNEW, LOCKREL, LOCKTRY, MODC, RDPIN, RQPIN
WZ only	5	MODZ, MUL, MULS, SCA, SCAS
Extended	4	TESTP, TESTPN, TESTB, TESTBN

WCZ-Only Instructions

The 40 WCZ-only instructions all follow the same pattern: they set both C and Z to the **same value**—the original state of the targeted bit or pin before the instruction modifies it. Because both flags receive identical information, updating only one flag would be meaningless.

These fall into five families of eight instructions each:

Family	Instructions	Operation
BIT*	BITC, BITH, BITL, BITNC, BITNOT, BITNZ, BITRND, BITZ	Modify bit(s) in register
DIR*	DIRC, DIRH, DIRL, DIRNC, DIRNOT, DIRNZ, DIRRND, DIRZ	Set pin direction
DRV*	DRVC, DRVH, DRVL, DRVNC, DRVNOT, DRVNZ, DRVRND, DRVZ	Set pin direction and output
FLT*	FLTC, FLTH, FLTL, FLTNC, FLTNOT, FLTNZ, FLTRND, FLTZ	Float pin (set to input)
OUT*	OUTC, OUTH, OUTL, OUTNC, OUTNOT, OUTNZ, OUTRND, OUTZ	Set pin output level

WC-Only Instructions

These eight instructions produce meaningful output only for the C flag:

Instruction	C Flag Meaning
COGID	1 if cog is running
COGINIT	1 if no free cog available
LOCKNEW	1 if no lock available
LOCKREL	1 if lock is currently taken (held)
LOCKTRY	1 if lock was acquired
MODC	Result of cccc expression
RDPIN	Modal result (depends on Smart Pin mode)
RQPIN	Modal result (depends on Smart Pin mode)

WZ-Only Instructions

These five instructions produce meaningful output only for the Z flag:

Instruction	Z Flag Meaning
MODZ	Result of zzzz expression
MUL	1 if either operand was zero
MULS	1 if either operand was zero
SCA	1 if result equals zero
SCAS	1 if result equals zero

Extended Effect Instructions

The **TESTP**, **TESTPN**, **TESTB**, and **TESTBN** instructions support WC, WZ, and extended effects, but explicitly reject WCZ. The extended effects combine the test result with the existing flag value using logical operations:

Effect	Operation
ANDC	C = C AND test_result
ANDZ	Z = Z AND test_result
ORC	C = C OR test_result
ORZ	Z = Z OR test_result
XORC	C = C XOR test_result
XORZ	Z = Z XOR test_result

These extended effects enable testing multiple bits or pins and accumulating the results into a single flag:

```
' Test if ALL of pins 0, 4, and 7 are high
        TESTP   #0              wc      ' C = pin 0 state
        TESTP   #4              andc    ' C = C AND pin 4 state
        TESTP   #7              andc    ' C = C AND pin 7 state
        if_c    JMP     #all_high       ' Branch if all three are high
```

Appendix D: Special Registers Quick Reference

Address	Hex	Register	Access	Purpose
496	$1F0	IJMP3	R/W	Interrupt 3 jump address
497	$1F1	IRET3	R/W	Interrupt 3 return address
498	$1F2	IJMP2	R/W	Interrupt 2 jump address
499	$1F3	IRET2	R/W	Interrupt 2 return address
500	$1F4	IJMP1	R/W	Interrupt 1 jump address
501	$1F5	IRET1	R/W	Interrupt 1 return address
502	$1F6	PA	R/W	Multi-purpose register A
503	$1F7	PB	R/W	Multi-purpose register B
504	$1F8	PTRA	R/W	Hub pointer A
505	$1F9	PTRB	R/W	Hub pointer B
506	$1FA	DIRA	R/W	Pin direction 0-31
507	$1FB	DIRB	R/W	Pin direction 32-63
508	$1FC	OUTA	R/W	Pin output 0-31
509	$1FD	OUTB	R/W	Pin output 32-63
510	$1FE	INA	R/O	Pin input 0-31
511	$1FF	INB	R/O	Pin input 32-63

For complete documentation including memory map diagram, usage examples, and non-memory-mapped registers, see Part II: Special Registers.

Appendix E: Predefined Constants

PASM2 provides a set of predefined constants that the assembler substitutes at compile time. These constants do not generate code themselves but provide standardized values for common operations including boolean logic, numeric bounds, mathematical calculations, and execution mode control.

Boolean Constants

TRUE

Logical True Constant
All bits set ($FFFFFFFF / -1).

Logical true constant with all bits set.

Value

Representation	Value
Hexadecimal	$FFFFFFFF
Decimal	-1
Binary	%11111111_11111111_11111111_11111111

Description

The TRUE constant represents a boolean true condition with all 32 bits set to 1. In two's complement signed representation, this equals -1. The all-bits-set pattern makes TRUE particularly useful for bitwise masking operations where a true condition must affect all bits.

Usage

```
' Using TRUE in conditional logic
                CMP     x, #0       wz      ' Compare x with 0
                MOV     result, TRUE        ' Default to TRUE
         if_z   MOV     result, FALSE       ' Set to FALSE if x was 0
```

Notes

- Standard boolean true value in PASM2
- Compatible with bitwise operations due to all-bits-set pattern
- Commonly used with conditional execution suffixes

Related Constants

- FALSE — Logical false constant

FALSE

Logical False Constant
All bits cleared ($00000000 / 0).

Logical false constant with all bits cleared.

Value

Representation	Value
Hexadecimal	$00000000
Decimal	0
Binary	%00000000_00000000_00000000_00000000

Description

The FALSE constant represents a boolean false condition with all 32 bits cleared to 0. This zero value serves as the standard false representation in PASM2 and provides a clean starting state for flag initialization.

Usage

```
1 ' Using FALSE for initialization
2         MOV     flag, FALSE     ' Initialize flag to FALSE
3         ' ... some operations ...
4         CMP     x, y        wz  ' Compare x and y
5         if_e MOV  flag, TRUE    ' Set flag to TRUE if equal
```

Notes

- Standard boolean false value in PASM2
- Used for clearing flags and initialization
- All bits cleared makes it safe for bitwise operations

Related Constants

- TRUE — Logical true constant

Numeric Limit Constants

NEGX

Maximum Negative Integer

Most negative 32-bit signed value ($80000000).

Most negative value in 32-bit signed integer representation.

Value

Representation	Value
Hexadecimal	$80000000
Decimal	-2,147,483,648
Binary	%10000000_00000000_00000000_00000000

Description

NEGX represents the maximum negative integer value in 32-bit two's complement representation (-2^{31}). This constant marks the lower boundary of the signed integer range and serves as a critical reference point for underflow detection and saturation arithmetic.

Usage

```
' Checking for negative underflow
                CMPS    value, ##NEGX   wc      ' Check if below min neg
        if_c    JMP     #underflow              ' Jump if underflow

' Using NEGX as lower limit
                MOV     limit, ##NEGX           ' Set limit to max negative
                FGES    value, limit            ' Clamp to not go below NEGX
```

Notes

- Represents -2^{31} in decimal notation
- Bit 31 set, bits 30-0 clear
- Used for saturation arithmetic and bounds checking
- Special case: `ABS(NEGX) = NEGX` due to two's complement representation (no positive equivalent exists)

Related Constants

- POSX — Maximum positive integer constant

POSX

Maximum Positive Integer
Most positive 32-bit signed value ($7FFFFFFF).

Most positive value in 32-bit signed integer representation.

Value

Representation	Value
Hexadecimal	$7FFFFFFF
Decimal	+2,147,483,647
Binary	%01111111_11111111_11111111_11111111

Description

POSX represents the maximum positive integer value in 32-bit two's complement representation (2^{31} - 1). This constant marks the upper boundary of the signed integer range and serves as a critical reference point for overflow detection and saturation arithmetic.

Usage

```
' Checking for positive overflow
                CMP     value, ##POSX   wc      ' Check if exceeds max
        if_nc   JMP     #overflow               ' Jump if overflow

' Using POSX as upper limit
                MOV     limit, ##POSX           ' Set limit to max positive
                FLES    value, limit            ' Clamp to not exceed POSX
```

Notes

- Represents 2^{31} - 1 in decimal notation
- Bit 31 clear, bits 30-0 set
- Used for saturation arithmetic and bounds checking
- One less than 2^{31} due to zero occupying one value in the range

Related Constants

- NEGX — Maximum negative integer constant

Mathematical Constants

PI

Mathematical Pi Constant
IEEE 754 single-precision ($40490FDB).

IEEE 754 single-precision floating-point representation of .

Value

Representation	Value
Hexadecimal	$40490FDB
Decimal	3.141593
Actual Value	3.141592653589793

Description

The PI constant provides the mathematical constant encoded in IEEE 754 single-precision floating-point format. This encoding allows direct use with the P2's CORDIC operations and floating-point calculations without runtime conversion overhead.

Usage

```
' Using PI with CORDIC rotation
        MOV     angle, ##PI         ' Load PI constant
        SHR     angle, #1           ' Divide by 2 for PI/2 (90 degrees)
        QROTATE angle, radius       ' Rotate by PI/2 radians

' Converting radians to degrees using PI
        MOV     x, ##PI             ' Start with PI
        QMUL    x, ##180            ' Multiply PI by 180
        QDIV    x, ##$80000000      ' Divide by 2^31 for scaling
        GETQX   degrees             ' Get degrees conversion factor
```

Notes

- IEEE 754 single-precision format provides approximately 7 decimal digits of precision
- Used primarily with CORDIC and floating-point operations
- For CORDIC angular operations, a full circle equals $80000000 ($2^{31}$)
- The constant stores the floating-point encoding, not a fixed-point representation

Related Constants

None (unique mathematical constant)

Execution Mode Constants

COGEXEC

Cog Execution Mode

Load code from hub to cog RAM and execute.

Execution mode constant for loading code from hub RAM to cog RAM.

Value

Representation	Value
Binary	%0_0_0000
Hexadecimal	$00

Description

COGEXEC specifies cog execution mode for the **COGINIT** instruction. When used, **COGINIT** loads 504 longs from hub RAM into cog RAM registers $000-$1F7 and begins execution at cog address $000. This mode provides maximum execution speed since all instructions execute from fast cog RAM.

Usage

```
' Start specific cog with code load
        COGINIT #COGEXEC+1, #$100   ' Load and start Cog 1 from Hub RAM $100

' Start Cog 5 with code at label
        COGINIT #COGEXEC+5, @code   ' Load and start Cog 5 from @code
```

Syntax

```
COGINIT #COGEXEC+id, #address
```

Where `id` specifies the target cog (0-7) and `address` points to the code in hub RAM.

Notes

- Loads cog RAM registers $000-$1F7 (504 longs) from hub RAM
- Begins execution at cog register address $000
- Must specify target cog ID (0-7)
- Fastest execution mode due to cog RAM access speeds
- Code size limited to 504 longs (2KB minus the 8 special-purpose registers at $1F8-$1FF)

Related Constants

- HUBEXEC — Hub execution mode constant
- COGEXEC_NEW — Auto-select available cog variant
- COGEXEC_NEW_PAIR — Auto-select adjacent cog pair variant

HUBEXEC

Hub Execution Mode

Execute code directly from hub RAM.

Execution mode constant for executing code directly from hub RAM.

Value

Representation	Value
Binary	%1_0_0000
Hexadecimal	$20

Description

HUBEXEC specifies hub execution mode for the **COGINIT** instruction. When used, **COGINIT** starts the target cog executing instructions directly from hub RAM without loading code to cog RAM. This mode removes code size restrictions at the cost of slower instruction fetch times.

Usage

```
' Start specific cog with hub execution
        COGINIT #HUBEXEC+1, ##$400   ' Cog 1 from Hub RAM $400

' Start Cog 5 with hub execution at label
        COGINIT #HUBEXEC+5, @code   ' Cog 5 from @code in hub
```

Syntax

```
COGINIT #HUBEXEC+id, #address
```

Where `id` specifies the target cog (0-7) and `address` points to the code in hub RAM.

Notes

- Executes instructions directly from hub RAM (no cog RAM load required)
- Hub execution allows unlimited code size (not limited to 504 longs)
- Slower than cog execution due to hub RAM access timing and FIFO overhead
- Instruction fetching occurs through FIFO/streamer mechanism
- Must specify target cog ID (0-7)
- Each cog maintains its own program counter for hub execution

Related Constants

- COGEXEC — Cog execution mode constant
- HUBEXEC_NEW — Auto-select available cog variant
- HUBEXEC_NEW_PAIR — Auto-select adjacent cog pair variant

Execution Mode Variants

The execution mode constants include additional variants for automatic cog selection. These variants combine the base execution mode (COGEXEC or **HUBEXEC**) with automatic resource selection flags, eliminating the need to manually specify cog IDs.

COGEXEC_NEW

Auto-Select Cog For Cog Execution
Auto-selects available cog for **COGEXEC** mode.

Execution mode constant for automatically selecting an available cog with COG RAM execution.

Encoding

Combines **COGEXEC** base mode with the N (new cog) flag set. The assembler resolves this to the appropriate bit pattern for **COGINIT**'s Dest operand.

Description

COGEXEC_NEW instructs **COGINIT** to find the next available (stopped) cog, load 504 longs from Hub RAM into that cog's RAM, and begin execution at cog address $000. This mode provides maximum execution speed since all instructions execute from fast cog RAM.

Usage

```
' Start any available cog with code load
                COGINIT #COGEXEC_NEW, ##@cog_code  wc
        if_c    JMP     #no_cog_available
```

Notes

- Use WC to detect if no cog was available (C=1 on failure)
- With WC and register Dest, the launched cog's ID is returned
- Equivalent to **COGEXEC** with N=1 in the %E_N_xVVV encoding

Related Constants

- COGEXEC — Base cog execution mode (specific cog)
- COGEXEC_NEW_PAIR — Auto-select adjacent cog pair variant

COGEXEC_NEW_PAIR

Auto-Select Cog Pair For Cog Execution
Auto-selects adjacent cog pair for **COGEXEC** mode.

Execution mode constant for automatically selecting an adjacent pair of available cogs with COG RAM execution.

Encoding

Combines **COGEXEC** base mode with both the N (new cog) and pair selection flags set.

Description

COGEXEC_NEW_PAIR instructs **COGINIT** to find an adjacent pair of available cogs (0-1, 2-3, 4-5, or 6-7), load code into the first cog, and start execution. Adjacent cog pairs can share their LUT memory via **SETLUTS**, enabling efficient inter-cog communication and data sharing.

Usage

```
' Start a cog pair for LUT sharing
                COGINIT #COGEXEC_NEW_PAIR, ##@pair_code  wc
        if_c    JMP     #no_pair_available
```

Notes

- Requires two adjacent, stopped cogs to succeed
- The returned cog ID is the lower of the pair (0, 2, 4, or 6)
- Adjacent pairs can share LUT memory for fast inter-cog communication
- Use **SETLUTS** to configure LUT sharing after both cogs are running

Related Constants

- COGEXEC — Base cog execution mode
- COGEXEC_NEW — Single cog auto-select variant

HUBEXEC_NEW

Auto-Select Cog For Hub Execution
Auto-selects available cog for **HUBEXEC** mode.

Execution mode constant for automatically selecting an available cog with Hub RAM execution.

Encoding

Combines **HUBEXEC** base mode with the N (new cog) flag set.

Description

HUBEXEC_NEW instructs **COGINIT** to find the next available (stopped) cog and start it executing instructions directly from Hub RAM without loading code to cog RAM. This mode removes the 504-long code size limitation at the cost of slower instruction fetch times due to Hub access latency.

Usage

```
' Start any available cog in hub execution mode
                COGINIT #HUBEXEC_NEW, ##@hub_code  wc
        if_c    JMP     #no_cog_available
```

Notes

- Hub execution allows unlimited code size
- Instruction fetching uses the FIFO/streamer mechanism
- Slower than cog execution due to Hub RAM access timing
- Use WC to detect failure and retrieve the launched cog's ID

Related Constants

- HUBEXEC — Base hub execution mode (specific cog)
- HUBEXEC_NEW_PAIR — Auto-select adjacent cog pair variant

HUBEXEC_NEW_PAIR

Auto-Select Cog Pair For Hub Execution

Auto-selects adjacent cog pair for **HUBEXEC** mode.

Execution mode constant for automatically selecting an adjacent pair of available cogs with Hub RAM execution.

Encoding

Combines **HUBEXEC** base mode with both the N (new cog) and pair selection flags set.

Description

HUBEXEC_NEW_PAIR instructs **COGINIT** to find an adjacent pair of available cogs and start them executing from Hub RAM. This combines the unlimited code size of hub execution with the LUT sharing capability of cog pairs.

Usage

```
' Start a cog pair for hub execution with LUT sharing
                COGINIT #HUBEXEC_NEW_PAIR, ##@hub_pair_code  wc
        if_c    JMP     #no_pair_available
```

Notes

- Combines unlimited hub code size with LUT sharing capability
- Requires two adjacent, stopped cogs to succeed
- The returned cog ID is the lower of the pair
- Use **SETLUTS** to configure LUT sharing after both cogs are running

Related Constants

- HUBEXEC — Base hub execution mode
- HUBEXEC_NEW — Single cog auto-select variant

These variants simplify cog management by allowing the system to automatically assign available cogs rather than requiring explicit cog ID specification. Always use WC with **COGINIT** when using these variants to detect allocation failures.

Debug Configuration Constants

The P2's **DEBUG** system operates at three distinct levels, each controlled by CON constants defined in the program. Code instrumentation constants control whether **DEBUG** statements compile into the program. Output infrastructure constants configure the debug serial communication system. Breakpoint constants configure automatic breaks for single-step debugging.

Code Instrumentation Constants

These constants control compile-time behavior. When **DEBUG** statements are disabled, the assembler generates no code for them—zero runtime overhead.

DEBUG_DISABLE

Disable All **DEBUG** Statements
Prevents all **DEBUG** statements from compiling (0 = enabled, non-zero = disabled).

Compile-time constant that globally disables all **DEBUG** statements.

Value

Value	Effect
0 or undefined	DEBUG statements compile normally
Non-zero	All DEBUG statements are omitted from compilation

Description

DEBUG_DISABLE provides a master switch for debug output. When defined as any non-zero value, the assembler skips all **DEBUG** statements entirely—no code is generated, no runtime overhead exists. This enables maintaining **DEBUG** instrumentation in source code while producing release binaries with zero **DEBUG** footprint.

Usage

```
CON
  DEBUG_DISABLE = 1       ' Set to 1 for release, 0 for development

DAT
        ORG
entry   DEBUG("This generates no code when DEBUG_DISABLE = 1")
        ' ... program code ...
```

Notes

- Must be defined as an integer constant in a CON block
- Affects both standard `DEBUG()` and selective `DEBUG[N]()` statements
- The check occurs at compile time; disabled statements produce zero bytes
- Works identically in Spin2 PUB/PRI blocks and PASM2 DAT blocks

Related Constants

- DEBUG_MASK — Selective channel control

DEBUG_MASK

Selective **DEBUG** Channel Mask
32-bit mask controlling which **DEBUG**N channels compile (bit N = channel N).

Compile-time constant enabling selective **DEBUG** channel compilation.

Value

Bit	Channel	Binary Mask
0	debug[0]	%00000000_00000000_00000000_00000001
1	debug[1]	%00000000_00000000_00000000_00000010
2	debug[2]	%00000000_00000000_00000000_00000100
…	…	…
31	debug[31]	%10000000_00000000_00000000_00000000

Description

DEBUG_MASK provides fine-grained control over debug output by channel. Each bit in the 32-bit mask corresponds to a debug channel numbered 0 through 31. The `DEBUG[N]()` statement compiles only if bit N is set in **DEBUG**_MASK. Standard `DEBUG()` statements without a channel number are unaffected by **DEBUG**_MASK.

This mechanism enables categorizing debug output by subsystem, verbosity level, or development phase. Changing a single constant recompiles only the desired **DEBUG** channels.

Usage

```
CON
  ' Channel assignments
  DBG_INIT   = 0              ' Initialization messages
  DBG_MOTOR  = 1              ' Motor control
  DBG_SENSOR = 2              ' Sensor readings
  DBG_ERROR  = 3              ' Error conditions

  ' Enable only initialization and errors
  DEBUG_MASK = (1 << DBG_INIT) | (1 << DBG_ERROR)

DAT
        ORG
entry   DEBUG[DBG_INIT]("Starting")     ' COMPILED - bit 0 set
        DEBUG[DBG_MOTOR]("Motor on")    ' NOT compiled - bit 1 clear
        DEBUG[DBG_SENSOR]("Reading")    ' NOT compiled - bit 2 clear
        DEBUG[DBG_ERROR]("Fault!")      ' COMPILED - bit 3 set
```

Notes

- Must be defined as an integer constant for `DEBUG[N]()` to compile
- If **DEBUG**_MASK is undefined, using `DEBUG[N]()` causes a compile error
- A mask of 0 disables all numbered channels; standard `DEBUG()` still works
- A mask of $FFFF_FFFF (-1) enables all 32 channels
- Channel numbers outside 0-31 cause a compile error

Related Constants

- DEBUG_DISABLE — Global **DEBUG** disable
- DEBUG_COGS — Runtime COG filtering

Output Infrastructure Constants

These constants configure the debug output system that handles all **DEBUG** statement output. They are patched into the debugger binary and affect serial communication parameters and output formatting.

DEBUG_COGS

Debug-Enabled COG Mask

8-bit mask specifying which COGs can produce debug output (bit N = COG N).

Runtime constant controlling which COGs can trigger debug output.

Value

Bit	COG	Binary Mask
0	COG 0	%00000001
1	COG 1	%00000010
2	COG 2	%00000100
3	COG 3	%00001000
4	COG 4	%00010000
5	COG 5	%00100000
6	COG 6	%01000000
7	COG 7	%10000000

Description

DEBUG_COGS controls runtime **DEBUG** capability per COG. If a COG's bit is clear, **DEBUG** statements executing on that COG produce no output—the **DEBUG** interrupt is ignored. This operates independently from **DEBUG**_MASK: **DEBUG**_MASK controls compile-time code generation, while **DEBUG**_COGS controls runtime output permission.

For a DEBUG statement to produce output, both conditions must be met: the statement must compile (DEBUG_MASK allows it or it's a standard `DEBUG()`), and the executing COG must have its bit set in **DEBUG**_COGS.

Usage

```
1 CON
2   DEBUG_COGS = %00000011      ' Only COGs 0 and 1 produce output
3
4 DAT
5         ORG
6 entry   DEBUG("From COG 0")           ' Output appears
7         cogspin(NEWCOG, worker, @stack)
```

continues on next page →

↪ *continued from previous page*

```
8
9 worker  DEBUG("From worker")          ' Output only if on COG 0 or 1
```

Notes

- Default behavior (undefined): all COGs can produce debug output
- Must be defined as an integer constant
- Reduces **DEBUG** overhead in multi-COG applications
- Useful for isolating debug output from specific COGs during development

Related Constants

- DEBUG_MASK — Compile-time channel filtering

DEBUG_DELAY

DEBUG Startup Delay

Milliseconds to wait before **DEBUG** system begins operation.

Startup delay before any debug output occurs.

Value

Type	Range
Integer	0 to practical limit (milliseconds)

Description

DEBUG_DELAY specifies a delay in milliseconds before the debug system begins operation. This delay occurs before the application launches, providing time for serial terminals to connect and synchronize. The delay is calculated as `(CLKFREQ / 1000) * DEBUG_DELAY` and executed during debugger initialization.

Usage

```
1 CON
2   DEBUG_DELAY = 2000          ' Wait 2 seconds for terminal connection
3
4 DAT
5         ORG
6 entry   DEBUG("This appears after 2 seconds")
```

Notes

- Must be defined as an integer constant
- Value is in milliseconds

- The delay occurs before any application code executes
- Useful when the host serial terminal needs connection time

Related Constants

- DEBUG_BAUD — Communication baud rate

DEBUG_TIMESTAMP

Enable **DEBUG** Timestamps
ADDS timing information to all debug output.

Enables timestamps in **DEBUG** messages.

Value

Definition	Effect
Defined (any value)	Timestamps enabled
Undefined	No timestamps

Description

DEBUG_TIMESTAMP enables timing information in all debug output. When defined, each debug message includes a timestamp relative to program start. This aids timing analysis and performance profiling by showing when events occur.

Usage

```
CON
  DEBUG_TIMESTAMP = TRUE

DAT
        ORG
entry   DEBUG("Started")              ' Output includes timestamp
        waitms(100)
        DEBUG("After delay")          ' Timestamp shows ~100ms elapsed
```

Notes

- The value is irrelevant; defining the symbol enables timestamps
- Timestamps appear on all debug output, not selectively
- Useful for profiling and timing-sensitive debugging

Related Constants

- DEBUG_DELAY — Startup delay

DEBUG_PIN_TX

DEBUG Transmit Pin
P2 pin number for **DEBUG** serial transmit.

Configures the debug serial transmit pin.

Value

Type	Default	Range
Integer	62	0-63

Description

DEBUG_PIN_TX specifies which P2 pin transmits **DEBUG** serial data to the host. The default pin 62 matches standard development board configurations where pins 62-63 connect to the USB-serial interface.

Usage

```
CON
  DEBUG_PIN_TX = 62           ' Use default transmit pin
```

Notes

- Must be defined as an integer constant
- **DEBUG**_PIN is an alias for **DEBUG**_PIN_TX
- Default matches Parallax development board pinout

Related Constants

- DEBUG_PIN_RX — Receive pin
- DEBUG_BAUD — Baud rate

DEBUG_PIN_RX

DEBUG Receive Pin
P2 pin number for **DEBUG** serial receive.

Configures the debug serial receive pin.

Value

Type	Default	Range
Integer	63	0-63

Description

DEBUG_PIN_RX specifies which P2 pin receives **DEBUG** serial data from the host. The default pin 63 matches standard development board configurations.

Usage

```
CON
  DEBUG_PIN_RX = 63           ' Use default receive pin
```

Notes

- Must be defined as an integer constant
- Used for bidirectional **DEBUG** communication with host
- Default matches Parallax development board pinout

Related Constants

- DEBUG_PIN_TX — Transmit pin
- DEBUG_BAUD — Baud rate

DEBUG_BAUD

DEBUG Baud Rate

Serial communication speed for debug output.

Configures the debug serial baud rate.

Value

Type	Default	Typical Values
Integer	DOWNLOAD_BAUD	115200, 230400, 921600, 2000000

Description

DEBUG_BAUD sets the serial communication speed for all debug output. Higher baud rates reduce **DEBUG** overhead but require host terminal support. The default uses the same baud rate as the download connection.

Usage

```
CON
  DEBUG_BAUD = 2_000_000      ' 2 Mbaud for fast debug output
```

Notes

- Must be defined as an integer constant
- Higher rates reduce per-statement timing impact
- Host terminal must support the configured rate
- 2 Mbaud is common for development; lower rates for compatibility

Related Constants

- DEBUG_PIN_TX — Transmit pin
- DEBUG_PIN_RX — Receive pin

Breakpoint Configuration Constants

These constants configure automatic breakpoints for single-step debugging. They instruct the debugger to halt execution at specific points, enabling interactive debugging.

DEBUG_MAIN

Break at Program Start

Triggers a breakpoint when the main program begins.

Configures the debugger to break at program entry.

Value

Definition	Effect
Defined (any value)	Break at main entry
Undefined	No automatic break

Description

DEBUG_MAIN instructs the debugger to trigger a breakpoint at the start of the main program. Execution halts before any user code runs, allowing single-stepping from the first instruction. This is essential for debugging initialization issues or understanding program flow from the beginning.

Usage

```
CON
  DEBUG_MAIN                  ' Break at program start

PUB main()
  ' Debugger breaks here before any code executes
  initialize()
```

Notes

- The value is irrelevant; defining the symbol enables the break
- Takes precedence over **DEBUG**_COGINIT if both are defined
- Enables single-stepping from program entry
- Used for debugging startup and initialization code

Related Constants

- DEBUG_COGINIT — Break on COG initialization

DEBUG_COGINIT

Break on COG Initialization
Triggers a breakpoint when any COG is initialized.

Configures the debugger to break on COG startup.

Value

Definition	Effect
Defined (any value)	Break on each COGINIT/COGSPIN
Undefined	No automatic break

Description

DEBUG_COGINIT instructs the debugger to trigger a breakpoint whenever a **COGINIT** or COGSPIN instruction executes. This enables debugging multi-COG applications by providing an opportunity to examine state before each new COG begins execution.

Usage

```
CON
  DEBUG_COGINIT               ' Break on every COG initialization

```

continues on next page →

↪ *continued from previous page*

```
PUB main()
  cogspin(NEWCOG, worker(), @stack)   ' Debugger breaks here
```

Notes

- The value is irrelevant; defining the symbol enables the break
- **DEBUG**_MAIN takes precedence if both are defined
- Useful for debugging COG startup and inter-COG coordination
- Each **COGINIT** or COGSPIN triggers a separate break

Related Constants

- DEBUG_MAIN — Break at program start
- DEBUG_COGS — Runtime COG filtering

Hardware Configuration Constants

The P2 provides extensive predefined constants for configuring its sophisticated hardware subsystems. These constants are documented in dedicated reference sections:

SmartPin Constants

The P2's 64 Smart Pins each function as independent hardware peripherals. Over 50 predefined constants configure input selection, filtering, output control, and the 32 operating modes including DAC, ADC, PWM, serial communication, and counters.

See: SmartPin Configuration Constants

Streamer Constants

The Streamer is the P2's DMA-like engine for high-bandwidth data transfer between hub RAM, LUT, pins, and DAC outputs. Over 80 predefined constants configure data sources, destinations, formats, color modes, and control flags.

See: Streamer Configuration Constants

Constants Summary

Category	Count	Purpose
Boolean	2	TRUE, FALSE for logical operations
Numeric Limits	2	NEGX, POSX for bounds checking
Mathematical	1	PI for CORDIC and floating-point
Execution Mode	6	COGEXEC, HUBEXEC and variants
Debug Configuration	10	DEBUG_DISABLE, DEBUG_MASK, infrastructure
SmartPin	59	Pin configuration and modes
Streamer	85	Data streaming and video
Total	**165**	Core predefined constants

Note: Clock configuration constants (RCFAST, RCSLOW, XI, PLL, XDIV, XMUL*, etc.) add over 1,000 additional symbols for system clock setup.*

Appendix F: Smart Pin Mode Constants

PASM2 provides an extensive set of predefined constants for configuring the P2's 64 Smart Pins. These constants replace complex 32-bit configuration patterns with readable symbolic names, making SmartPin programming practical and maintainable.

SmartPin Configuration Word Structure

Each SmartPin is configured through a 32-bit mode **WORD** with the following structure:

```
Bits [31..0] = %AAAA_BBBB_FFF_PPPPPPPPPPPPP_TT_MMMMM_0
```

Field	Bits	Purpose
AAAA	31-28	A input selector (polarity and source)
BBBB	27-24	B input selector (polarity and source)
FFF	23-21	A/B input logic and filter settings
P	20-8	Low-level pin mode and parameters
TT	7-6	DIR/OUT control mode
MMMMM	5-1	Smart pin operating mode (0-31)
0	0	Reserved (must be 0)

Constants are combined using OR operations to build the complete configuration:

```
        MOV     mode, ##P_PWM_TRIANGLE | P_OE | P_LOCAL_A
        WRPIN   mode, #56
```

A Input Configuration

A Input Polarity (pick one)

Constant	Value	Description
P_TRUE_A	%0000_0000_000_0000000000000_00_00000_0	True A input (default)
P_INVERT_A	%1000_0000_000_0000000000000_00_00000_0	Invert A input

A Input Selection (pick one)

Constant	Value	Description
P_LOCAL_A	%0000_0000_000_0000000000000_00_00000_0	Select local pin for A input (default)
P_PLUS1_A	%0001_0000_000_0000000000000_00_00000_0	Select pin+1 for A input
P_PLUS2_A	%0010_0000_000_0000000000000_00_00000_0	Select pin+2 for A input
P_PLUS3_A	%0011_0000_000_0000000000000_00_00000_0	Select pin+3 for A input
P_OUTBIT_A	%0100_0000_000_0000000000000_00_00000_0	Select OUT bit for A input
P_MINUS3_A	%0101_0000_000_0000000000000_00_00000_0	Select pin-3 for A input
P_MINUS2_A	%0110_0000_000_0000000000000_00_00000_0	Select pin-2 for A input
P_MINUS1_A	%0111_0000_000_0000000000000_00_00000_0	Select pin-1 for A input

B Input Configuration

B Input Polarity (pick one)

Constant	Value	Description
P_TRUE_B	%0000_0000_000_0000000000000_00_00000_0	True B input (default)
P_INVERT_B	%0000_1000_000_0000000000000_00_00000_0	Invert B input

B Input Selection (pick one)

Constant	Value	Description
P_LOCAL_B	%0000_0000_000_0000000000000_00_00000_0	Select local pin for B input (default)
P_PLUS1_B	%0000_0001_000_0000000000000_00_00000_0	Select pin+1 for B input
P_PLUS2_B	%0000_0010_000_0000000000000_00_00000_0	Select pin+2 for B input
P_PLUS3_B	%0000_0011_000_0000000000000_00_00000_0	Select pin+3 for B input
P_OUTBIT_B	%0000_0100_000_0000000000000_00_00000_0	Select OUT bit for B input
P_MINUS3_B	%0000_0101_000_0000000000000_00_00000_0	Select pin-3 for B input
P_MINUS2_B	%0000_0110_000_0000000000000_00_00000_0	Select pin-2 for B input
P_MINUS1_B	%0000_0111_000_0000000000000_00_00000_0	Select pin-1 for B input

A/B Input Logic (pick one)

Constant	Value	Description
P_PASS_AB	%0000_0000_000_0000000000000_00_00000_0	Pass A and B through (default)
P_AND_AB	%0000_0000_001_0000000000000_00_00000_0	A AND B → A, pass B
P_OR_AB	%0000_0000_010_0000000000000_00_00000_0	A OR B → A, pass B
P_XOR_AB	%0000_0000_011_0000000000000_00_00000_0	A XOR B → A, pass B
P_FILT0_AB	%0000_0000_100_0000000000000_00_00000_0	Filter A and B (2-clock sample)
P_FILT1_AB	%0000_0000_101_0000000000000_00_00000_0	Filter A and B (3-clock sample)
P_FILT2_AB	%0000_0000_110_0000000000000_00_00000_0	Filter A and B (5-clock sample)
P_FILT3_AB	%0000_0000_111_0000000000000_00_00000_0	Filter A and B (8-clock sample)

Low-Level Pin Modes

Logic/Schmitt/Comparator Input Modes (pick one)

Constant	Value	Description
P_LOGIC_A	%0000_0000_000_0000000000000_00_00000_0	Logic level A → IN, output OUT (default)
P_LOGIC_A_FB	%0000_0000_000_0001000000000_00_00000_0	Logic level A → IN, output feedback
P_LOGIC_B_FB	%0000_0000_000_0010000000000_00_00000_0	Logic level B → IN, output feedback
P_SCHMITT_A	%0000_0000_000_0011000000000_00_00000_0	Schmitt trigger A → IN, output OUT
P_SCHMITT_A_FB	%0000_0000_000_0100000000000_00_00000_0	Schmitt trigger A → IN, output feedback
P_SCHMITT_B_FB	%0000_0000_000_0101000000000_00_00000_0	Schmitt trigger B → IN, output feedback
P_COMPARE_AB	%0000_0000_000_0110000000000_00_00000_0	A > B → IN, output OUT
P_COMPARE_AB_FB	%0000_0000_000_0111000000000_00_00000_0	A > B → IN, output feedback

ADC Input Modes (pick one)

Constant	Value	Description
P_ADC_GIO	%0000_0000_000_1000000000000_00_00000_0	ADC GIO → IN, output OUT
P_ADC_VIO	%0000_0000_000_1000010000000_00_00000_0	ADC VIO → IN, output OUT
P_ADC_FLOAT	%0000_0000_000_1000100000000_00_00000_0	ADC FLOAT → IN, output OUT
P_ADC_1X	%0000_0000_000_1000110000000_00_00000_0	ADC 1x gain → IN, output OUT
P_ADC_3X	%0000_0000_000_1001000000000_00_00000_0	ADC 3.16x gain → IN, output OUT
P_ADC_10X	%0000_0000_000_1001010000000_00_00000_0	ADC 10x gain → IN, output OUT
P_ADC_30X	%0000_0000_000_1001100000000_00_00000_0	ADC 31.6x gain → IN, output OUT
P_ADC_100X	%0000_0000_000_1001110000000_00_00000_0	ADC 100x gain → IN, output OUT

DAC Output Modes (pick one)

Constant	Value	Description
P_DAC_990R_3V	%0000_0000_000_1010000000000_00_00000_0	DAC 990Ω, 3.3V peak, ADC 1x → IN
P_DAC_600R_2V	%0000_0000_000_1010100000000_00_00000_0	DAC 600Ω, 2.0V peak, ADC 1x → IN
P_DAC_124R_3V	%0000_0000_000_1011000000000_00_00000_0	DAC 123.75Ω, 3.3V peak, ADC 1x → IN
P_DAC_75R_2V	%0000_0000_000_1011100000000_00_00000_0	DAC 75Ω, 2.0V peak, ADC 1x → IN

Level-Comparison Modes (pick one)

Constant	Value	Description
P_LEVEL_A	%0000_0000_000_1100000000000_00_00000_0	A > Level → IN, output OUT
P_LEVEL_A_FBN	%0000_0000_000_1101000000000_00_00000_0	A > Level → IN, output negative feedback
P_LEVEL_B_FBP	%0000_0000_000_1110000000000_00_00000_0	B > Level → IN, output positive feedback
P_LEVEL_B_FBN	%0000_0000_000_1111000000000_00_00000_0	B > Level → IN, output negative feedback

Low-Level Pin Sub-Modes

Sync Mode (pick one)

Constant	Value	Description
P_ASYNC_IO	%0000_0000_000_0000000000000_00_00000_0	Asynchronous I/O (default)
P_SYNC_IO	%0000_0000_000_0000100000000_00_00000_0	Synchronous I/O

IN Polarity (pick one)

Constant	Value	Description
P_TRUE_IN	%0000_0000_000_0000000000000_00_00000_0	True IN bit (default)
P_INVERT_IN	%0000_0000_000_0000010000000_00_00000_0	Invert IN bit

Output Polarity (pick one)

Constant	Value	Description
P_TRUE_OUTPUT	%0000_0000_000_0000000000000_00_00000_0	True output (default)
P_TRUE_OUT	%0000_0000_000_0000000000000_00_00000_0	Alias for P_TRUE_OUTPUT
P_INVERT_OUTPUT	%0000_0000_000_0000001000000_00_00000_0	Invert output
P_INVERT_OUT	%0000_0000_000_0000001000000_00_00000_0	Alias for P_INVERT_OUTPUT

Drive Strength

Drive-High Strength (pick one)

Constant	Value	Description
P_HIGH_FAST	%0000_0000_000_0000000000000_00_00000_0	Drive high fast (30mA) - default
P_HIGH_1K5	%0000_0000_000_0000000001000_00_00000_0	Drive high 1.5kΩ
P_HIGH_15K	%0000_0000_000_0000000010000_00_00000_0	Drive high 15kΩ
P_HIGH_150K	%0000_0000_000_0000000011000_00_00000_0	Drive high 150kΩ
P_HIGH_1MA	%0000_0000_000_0000000100000_00_00000_0	Drive high 1mA current source
P_HIGH_100UA	%0000_0000_000_0000000101000_00_00000_0	Drive high 100 A current source
P_HIGH_10UA	%0000_0000_000_0000000110000_00_00000_0	Drive high 10 A current source
P_HIGH_FLOAT	%0000_0000_000_0000000111000_00_00000_0	Float high (high-impedance)

Drive-Low Strength (pick one)

Constant	Value	Description
P_LOW_FAST	%0000_0000_000_0000000000000_00_00000_0	Drive low fast (30mA) - default
P_LOW_1K5	%0000_0000_000_0000000000001_00_00000_0	Drive low 1.5kΩ
P_LOW_15K	%0000_0000_000_0000000000010_00_00000_0	Drive low 15kΩ
P_LOW_150K	%0000_0000_000_0000000000011_00_00000_0	Drive low 150kΩ
P_LOW_1MA	%0000_0000_000_0000000000100_00_00000_0	Drive low 1mA current sink
P_LOW_100UA	%0000_0000_000_0000000000101_00_00000_0	Drive low 100 A current sink
P_LOW_10UA	%0000_0000_000_0000000000110_00_00000_0	Drive low 10 A current sink
P_LOW_FLOAT	%0000_0000_000_0000000000111_00_00000_0	Float low (high-impedance)

DIR/OUT Control (TT Field)

Constant	Value	Description
P_TT_00	%0000_0000_000_0000000000000_00_00000_0	TT = %00 (default)
P_TT_01	%0000_0000_000_0000000000000_01_00000_0	TT = %01
P_TT_10	%0000_0000_000_0000000000000_10_00000_0	TT = %10
P_TT_11	%0000_0000_000_0000000000000_11_00000_0	TT = %11
P_OE	%0000_0000_000_0000000000000_01_00000_0	Output enable in smart pin mode
P_CHANNEL	%0000_0000_000_0000000000000_01_00000_0	Enable DAC channel (non-smart mode)
P_BITDAC	%0000_0000_000_0000000000000_10_00000_0	Enable BITDAC (non-smart mode)

Smart Pin Operating Modes (32 Modes)

Mode %00000 - %00011: Repository and DAC Dither Modes

Constant	Value	Description
P_NORMAL	%0000_0000_000_0000000000000_00_00000_0	Normal I/O (smart pin disabled)
P_REPOSITORY	%0000_0000_000_0000000000000_00_00001_0	Long repository (non-DAC mode)
P_DAC_NOISE	%0000_0000_000_0000000000000_00_00001_0	DAC noise (DAC mode)
P_DAC_DITHER_RND	%0000_0000_000_0000000000000_00_00010_0	DAC 16-bit random dither
P_DAC_DITHER_PWM	%0000_0000_000_0000000000000_00_00011_0	DAC 16-bit PWM dither

Mode %00100 - %00111: Pulse and NCO Modes

Constant	Value	Description
P_PULSE	%0000_0000_000_0000000000000_00_00100_0	Pulse/cycle output
P_TRANSITION	%0000_0000_000_0000000000000_00_00101_0	Transition output
P_NCO_FREQ	%0000_0000_000_0000000000000_00_00110_0	NCO frequency output
P_NCO_DUTY	%0000_0000_000_0000000000000_00_00111_0	NCO duty cycle output

Mode %01000 - %01011: PWM Modes

Constant	Value	Description
P_PWM_TRIANGLE	%0000_0000_000_0000000000000_00_01000_0	PWM with triangle carrier
P_PWM_SAWTOOTH	%0000_0000_000_0000000000000_00_01001_0	PWM with sawtooth carrier
P_PWM_SMPS	%0000_0000_000_0000000000000_00_01010_0	PWM for switch-mode power supplies
P_QUADRATURE	%0000_0000_000_0000000000000_00_01011_0	A-B quadrature encoder input

Mode %01100 - %01111: Counter Modes

Constant	Value	Description
P_REG_UP	%0000_0000_000_0000000000000_00_01100_0	Inc on A-rise when B-high
P_REG_UP_DOWN	%0000_0000_000_0000000000000_00_01101_0	Inc on A-rise/B-high, dec on A-rise/B-low
P_COUNT_RISES	%0000_0000_000_0000000000000_00_01110_0	Count A-rises, optionally dec on B-rise
P_COUNT_HIGHS	%0000_0000_000_0000000000000_00_01111_0	Count A-highs, optionally dec on B-high

Mode %10000 - %10111: Timing Measurement Modes

Constant	Value	Description
P_STATE_TICKS	%0000_0000_000_0000000000000_00_10000_0	For A-low/high states, count ticks
P_HIGH_TICKS	%0000_0000_000_0000000000000_00_10001_0	For A-high states, count ticks
P_EVENTS_TICKS	%0000_0000_000_0000000000000_00_10010_0	For X A-events, count ticks / timeout
P_PERIODS_TICKS	%0000_0000_000_0000000000000_00_10011_0	For X periods of A, count ticks
P_PERIODS_HIGHS	%0000_0000_000_0000000000000_00_10100_0	For X periods of A, count highs
P_COUNTER_TICKS	%0000_0000_000_0000000000000_00_10101_0	For periods in X+ ticks, count ticks
P_COUNTER_HIGHS	%0000_0000_000_0000000000000_00_10110_0	For periods in X+ ticks, count highs
P_COUNTER_PERIODS	%0000_0000_000_0000000000000_00_10111_0	For periods in X+ ticks, count periods

Mode %11000 - %11011: ADC and USB Modes

Constant	Value	Description
P_ADC	%0000_0000_000_0000000000000_00_11000_0	ADC sample/filter/capture (internal clock)
P_ADC_EXT	%0000_0000_000_0000000000000_00_11001_0	ADC sample/filter/capture (external clock)
P_ADC_SCOPE	%0000_0000_000_0000000000000_00_11010_0	ADC oscilloscope with trigger
P_USB_PAIR	%0000_0000_000_0000000000000_00_11011_0	USB D+/D- pin pair

Mode %11100 - %11111: Serial Communication Modes

Constant	Value	Description
P_SYNC_TX	%0000_0000_000_0000000000000_00_11100_0	Synchronous serial transmit
P_SYNC_RX	%0000_0000_000_0000000000000_00_11101_0	Synchronous serial receive
P_ASYNC_TX	%0000_0000_000_0000000000000_00_11110_0	Asynchronous serial transmit
P_ASYNC_RX	%0000_0000_000_0000000000000_00_11111_0	Asynchronous serial receive

Usage Examples

PWM Output Configuration

```
' Configure pin 56 for triangle PWM output
        MOV     mode, ##P_PWM_TRIANGLE | P_OE
        WRPIN   mode, #56
        WXPIN   ##10000, #56        ' Period = 10000 clocks
        WYPIN   ##5000, #56         ' Duty = 50%
        DIRH    #56                 ' Enable output
```

ADC Input with Gain

```
' Configure pin 32 for ADC with 10x gain
        MOV     mode, ##P_ADC | P_ADC_10X
        WRPIN   mode, #32
        WXPIN   ##14, #32           ' 14-bit resolution
        DIRL    #32                 ' Input mode
```

Open-Drain Output (I2C-style)

```
' Configure for open-drain with 1.5kΩ pull-up
        MOV     mode, ##P_HIGH_FLOAT | P_LOW_1K5
        WRPIN   mode, #44
```

Schmitt Trigger Input with Filter

```
' Debounced button input
        MOV     mode, ##P_SCHMITT_A | P_FILT3_AB
        WRPIN   mode, #0
```

Combining Constants

SmartPin constants are designed to be combined using OR operations. The bit fields are carefully arranged so constants from different categories don't conflict:

```
' Complex config: Async TX, inverted, fast drive
        MOV     mode, ##P_ASYNC_TX | P_OE | P_INVERT_OUTPUT
        OR      mode, ##P_HIGH_FAST | P_LOW_FAST
        WRPIN   mode, pin
```

Related Instructions

- WRPIN — Write SmartPin mode register
- WXPIN — Write SmartPin X register (period, bit timing, etc.)
- WYPIN — Write SmartPin Y register (duty, data, etc.)
- RDPIN — Read SmartPin result and clear flag
- RQPIN — Read SmartPin result without clearing flag
- AKPIN — Acknowledge SmartPin (clear flag only)

Appendix G: Streamer Mode Constants

PASM2 provides predefined constants for configuring the P2's Streamer—a powerful DMA-like engine that transfers data between hub RAM, LUT RAM, pins, and DAC outputs. These constants replace complex bit patterns with readable symbolic names.

Streamer Overview

The Streamer operates in conjunction with the FIFO and can:

- Transfer data from hub RAM to pins/DACs (playback)
- Transfer data from pins/ADCs to hub RAM (capture)
- Perform real-time data transformations (color conversion, bit manipulation)
- Generate video signals with automatic timing

Streamer commands are issued via **XINIT**, **XCONT**, and related instructions.

Command Word Structure

Streamer commands are 32-bit values composed of mode selection and control fields:

```
Bits 31-28: Mode selector; bits 27-16: control/config fields
Bits 15-0:  Transfer count (NCO rollovers); NCO rate is set by SETXFRQ
```

The values shown below are the base constants that get combined with control flags using OR operations.

Immediate to LUT to Pins/DACs

These modes stream immediate data through the LUT to output pins or DAC channels.

Constant	Value	Description
X_IMM_32X1_LUT	%0000_0000_0000_0000 << 16	32×1: 32 bits to LUT, 1 bit per pin
X_IMM_16X2_LUT	%0001_0000_0000_0000 << 16	16×2: 16 bits to LUT, 2 bits per pin
X_IMM_8X4_LUT	%0010_0000_0000_0000 << 16	8×4: 8 bits to LUT, 4 bits per pin
X_IMM_4X8_LUT	%0011_0000_0000_0000 << 16	4×8: 4 bits to LUT, 8 bits per pin

Immediate to Pins/DACs (Direct)

These modes stream immediate data directly to pins and DAC channels.

Constant	Value	Description
X_IMM_32X1_1DAC1	%0100_0000_0000_0000 << 16	32×1 immediate, 1 pin, 1 DAC channel
X_IMM_16X2_2DAC1	%0101_0000_0000_0000 << 16	16×2 immediate, 2 pins, 1 DAC channel
X_IMM_16X2_1DAC2	%0101_0000_0000_0010 << 16	16×2 immediate, 1 pin, 2 DAC channels
X_IMM_8X4_4DAC1	%0110_0000_0000_0000 << 16	8×4 immediate, 4 pins, 1 DAC channel
X_IMM_8X4_2DAC2	%0110_0000_0000_0010 << 16	8×4 immediate, 2 pins, 2 DAC channels
X_IMM_8X4_1DAC4	%0110_0000_0000_0100 << 16	8×4 immediate, 1 pin, 4 DAC channels
X_IMM_4X8_4DAC2	%0110_0000_0000_0110 << 16	4×8 immediate, 4 pins, 2 DAC channels
X_IMM_4X8_2DAC4	%0110_0000_0000_0111 << 16	4×8 immediate, 2 pins, 4 DAC channels
X_IMM_4X8_1DAC8	%0110_0000_0000_1110 << 16	4×8 immediate, 1 pin, 8 DAC channels
X_IMM_2X16_4DAC4	%0110_0000_0000_1111 << 16	2×16 immediate, 4 pins, 4 DAC channels
X_IMM_2X16_2DAC8	%0111_0000_0000_0000 << 16	2×16 immediate, 2 pins, 8 DAC channels
X_IMM_1X32_4DAC8	%0111_0000_0000_0001 << 16	1×32 immediate, 4 pins, 8 DAC channels

RDFAST to LUT to Pins/DACs

These modes read data from hub RAM via **RDFAST** FIFO, process through LUT, and output to pins/DACs.

Constant	Value	Description
X_RFLONG_32X1_LUT	%0111_0000_0000_0010 << 16	Read long, 32×1 to LUT to pins
X_RFLONG_16X2_LUT	%0111_0000_0000_0100 << 16	Read long, 16×2 to LUT to pins
X_RFLONG_8X4_LUT	%0111_0000_0000_0110 << 16	Read long, 8×4 to LUT to pins
X_RFLONG_4X8_LUT	%0111_0000_0000_1000 << 16	Read long, 4×8 to LUT to pins

RDFAST Byte Operations

These modes read bytes from hub RAM and output to pins/DACs with various configurations.

Constant	Value	Description
X_RFBYTE_1P_1DAC1	%1000_0000_0000_0000 << 16	Read byte, 1 pin, 1 DAC channel
X_RFBYTE_2P_2DAC1	%1001_0000_0000_0000 << 16	Read byte, 2 pins, 1 DAC channel
X_RFBYTE_2P_1DAC2	%1001_0000_0000_0010 << 16	Read byte, 1 pin, 2 DAC channels
X_RFBYTE_4P_4DAC1	%1010_0000_0000_0000 << 16	Read byte, 4 pins, 1 DAC channel
X_RFBYTE_4P_2DAC2	%1010_0000_0000_0010 << 16	Read byte, 2 pins, 2 DAC channels
X_RFBYTE_4P_1DAC4	%1010_0000_0000_0100 << 16	Read byte, 1 pin, 4 DAC channels
X_RFBYTE_8P_4DAC2	%1010_0000_0000_0110 << 16	Read byte, 4 pins, 2 DAC channels
X_RFBYTE_8P_2DAC4	%1010_0000_0000_0111 << 16	Read byte, 2 pins, 4 DAC channels
X_RFBYTE_8P_1DAC8	%1010_0000_0000_1110 << 16	Read byte, 1 pin, 8 DAC channels

RDFAST Word/Long Operations

These modes read words or longs from hub RAM for higher bandwidth applications.

Constant	Value	Description
X_RFWORD_16P_4DAC4	%1010_0000_0000_1111 << 16	Read word, 16 pins, 4 DAC channels
X_RFWORD_16P_2DAC8	%1011_0000_0000_0000 << 16	Read word, 16 pins, 8 DAC channels
X_RFLONG_32P_4DAC8	%1011_0000_0000_0001 << 16	Read long, 32 pins, 8 DAC channels

Video and Color Modes

These modes perform color space conversion for video generation.

Constant	Value	Description
X_RFBYTE_LUMA8	%1011_0000_0000_0010 << 16	Read byte as 8-bit luminance (grayscale)
X_RFBYTE_RGBI8	%1011_0000_0000_0011 << 16	Read byte as RGBI 2:2:2:2 (16 colors + intensity)
X_RFBYTE_RGB8	%1011_0000_0000_0100 << 16	Read byte as RGB 3:3:2 (256 colors)
X_RFWORD_RGB16	%1011_0000_0000_0101 << 16	Read word as RGB 5:6:5 (65536 colors)
X_RFLONG_RGB24	%1011_0000_0000_0110 << 16	Read long as RGB 8:8:8 (16M colors)

WRFAST Operations (Capture)

These modes capture data from pins/ADCs and write to hub RAM via **WRFAST** FIFO.

Constant	Value	Description
X_1P_1DAC1_WFBYTE	%1100_0000_0000_0000 << 16	1 pin, 1 DAC to byte, write to hub
X_2P_2DAC1_WFBYTE	%1101_0000_0000_0000 << 16	2 pins, 1 DAC to byte, write to hub
X_2P_1DAC2_WFBYTE	%1101_0000_0000_0010 << 16	1 pin, 2 DACs to byte, write to hub
X_4P_4DAC1_WFBYTE	%1110_0000_0000_0000 << 16	4 pins, 1 DAC to byte, write to hub
X_4P_2DAC2_WFBYTE	%1110_0000_0000_0010 << 16	2 pins, 2 DACs to byte, write to hub
X_4P_1DAC4_WFBYTE	%1110_0000_0000_0100 << 16	1 pin, 4 DACs to byte, write to hub
X_8P_4DAC2_WFBYTE	%1110_0000_0000_0110 << 16	4 pins, 2 DACs to byte, write to hub
X_8P_2DAC4_WFBYTE	%1110_0000_0000_0111 << 16	2 pins, 4 DACs to byte, write to hub
X_8P_1DAC8_WFBYTE	%1110_0000_0000_1110 << 16	1 pin, 8 DACs to byte, write to hub
X_16P_4DAC4_WFWORD	%1110_0000_0000_1111 << 16	16 pins, 4 DACs to word, write to hub
X_16P_2DAC8_WFWORD	%1111_0000_0000_0000 << 16	16 pins, 8 DACs to word, write to hub
X_32P_4DAC8_WFLONG	%1111_0000_0000_0001 << 16	32 pins, 8 DACs to long, write to hub

ADC Sampling Modes

These modes capture ADC samples and optionally write to hub RAM.

Constant	Value	Description
X_1ADC8_0P_1DAC8_WFBYTE	%1111_0000_0000_0010 << 16	1 ADC to 8-bit, 0 pins, 1 DAC, write byte
X_1ADC8_8P_2DAC8_WFWORD	%1111_0000_0000_0011 << 16	1 ADC to 8-bit, 8 pins, 2 DACs, write word
X_2ADC8_0P_2DAC8_WFWORD	%1111_0000_0000_0100 << 16	2 ADCs to 8-bit, 0 pins, 2 DACs, write word
X_2ADC8_16P_4DAC8_WFLONG	%1111_0000_0000_0101 << 16	2 ADCs to 8-bit, 16 pins, 4 DACs, write long
X_4ADC8_0P_4DAC8_WFLONG	%1111_0000_0000_0110 << 16	4 ADCs to 8-bit, 0 pins, 4 DACs, write long

DDS and Goertzel Modes

These modes perform digital signal processing operations.

Constant	Value	Description
X_DDS_GOERTZEL_SINC1	%1111_0000_0000_0111 << 16	DDS/Goertzel with SINC1 filter
X_DDS_GOERTZEL_SINC2	%1111_0000_1000_0111 << 16	DDS/Goertzel with SINC2 filter

Control Flags

These flags modify Streamer behavior and are combined with mode constants using OR.

DAC Channel Selection

The DAC selection constants control which of the four DAC channels (3, 2, 1, 0) are active and how they're configured. In the naming convention, 0 = streamer data channel X0, 1 = data channel X1 (likewise 2 = X2, 3 = X3), X = no override (the **SETDACS** value for that DAC passes through), and the N suffix = one's-complement (inverted) output.

Table G.8.

Constant	Value	Description
X_DACS_OFF	(default - no bits set)	No streamer DAC output (SETDACS values pass through on all channels)
X_DACS_0_0_0_0	%0000_0001_0000_0000 << 16	X0 on all four DAC channels (mono)
X_DACS_X_X_0_0	%0000_0010_0000_0000 << 16	X0 on DAC channels 1 and 0; channels 3,2 not overridden
X_DACS_0_0_X_X	%0000_0011_0000_0000 << 16	X0 on DAC channels 3 and 2; channels 1,0 not overridden
X_DACS_X_X_X_0	%0000_0100_0000_0000 << 16	X0 on DAC channel 0 only
X_DACS_X_X_0_X	%0000_0101_0000_0000 << 16	X0 on DAC channel 1 only
X_DACS_X_0_X_X	%0000_0110_0000_0000 << 16	X0 on DAC channel 2 only
X_DACS_0_X_X_X	%0000_0111_0000_0000 << 16	X0 on DAC channel 3 only
X_DACS_0N0_0N0	%0000_1000_0000_0000 << 16	X0 differential pairs on all four channels: ch3 !X0, ch2 X0, ch1 !X0, ch0 X0
X_DACS_X_X_0N0	%0000_1001_0000_0000 << 16	X0 differential pair on DAC channels 1 and 0: ch1 !X0, ch0 X0
X_DACS_0N0_X_X	%0000_1010_0000_0000 << 16	X0 differential pair on DAC channels 3 and 2: ch3 !X0, ch2 X0
X_DACS_1_0_1_0	%0000_1011_0000_0000 << 16	X1,X0 pairs on all four channels: ch3 X1, ch2 X0, ch1 X1, ch0 X0
X_DACS_X_X_1_0	%0000_1100_0000_0000 << 16	X1,X0 on DAC channels 1 and 0: ch1 X1, ch0 X0
X_DACS_1_0_X_X	%0000_1101_0000_0000 << 16	X1,X0 on DAC channels 3 and 2: ch3 X1, ch2 X0
X_DACS_1N1_0N0	%0000_1110_0000_0000 << 16	X1,X0 differential pairs on all four channels: ch3 !X1, ch2 X1, ch1 !X0, ch0 X0
X_DACS_3_2_1_0	%0000_1111_0000_0000 << 16	X3,X2,X1,X0 --- one streamer word per channel (standard 4-channel)

Pin Output Control

Constant	Value	Description
X_PINS_OFF	(default - no bits set)	Disable pin outputs
X_PINS_ON	%0000_0000_1000_0000 << 16	Enable pin outputs

Write Control

Constant	Value	Description
X_WRITE_OFF	(default - no bits set)	Disable hub RAM writes
X_WRITE_ON	%0000_0000_1000_0000 << 16	Enable hub RAM writes

Alternate Bit Order

Constant	Value	Description
X_ALT_OFF	(default - no bits set)	Normal bit order
X_ALT_ON	%0000_0000_0000_0001 << 16	Alternate bit order for 1/2/4 bit modes

Usage Examples

Video Pixel Streaming

```
' Stream RGB24 video data to VGA pins
        RDFAST  #0, video_buffer       ' Set up FIFO from video buffer
        MOV     mode, ##X_RFLONG_RGB24 | X_PINS_ON
        XINIT   mode, ##25_000_000     ' 25 MHz pixel clock
```

Audio DAC Output

```
' Stream 8-bit audio samples to DAC
        RDFAST  #0, audio_buffer
        MOV     mode, ##X_RFBYTE_1P_1DAC1 | X_DACS_3_2_1_0
        XINIT   mode, ##44100          ' 44.1 kHz sample rate
```

ADC Capture to Memory

```
' Capture ADC samples to hub RAM
        WRFAST  #0, capture_buffer     ' Set up FIFO for writing
        MOV     mode, ##X_1ADC8_0P_1DAC8_WFBYTE | X_WRITE_ON
        XINIT   mode, ##100_000        ' 100 kHz sample rate
```

LUT-Based Color Mapping

```
' Stream bytes through LUT for palette lookup
        RDFAST  #0, sprite_data
        MOV     mode, ##X_RFLONG_8X4_LUT | X_PINS_ON
        SETLUTS #0                      ' Use LUT for color palette
        XINIT   mode, nco_value
```

Mode Naming Convention

Streamer constant names follow a consistent pattern:

```
X_[source][size]_[pins]P_[dacs]DAC[bits]_[dest]
```

Component	Meaning
X_	Streamer constant prefix
RF	Read from FIFO (hub RAM)
WF	Write to FIFO (hub RAM)
IMM	Immediate data
BYTE/WORD/LONG	Data unit size
_nP	Number of pins used
_nDACn	Number of DAC channels, bits per channel
LUT	Data passes through LUT

Combining Constants

Streamer mode and control flags are combined using OR:

```
' Full-featured video mode
        MOV     mode, ##X_RFLONG_RGB24 | X_PINS_ON | X_DACS_3_2_1_0
        XINIT   mode, nco_rate
```

Data Width Modes

The Streamer supports various data packing/unpacking modes:

Mode	Meaning
32x1	32 single-bit values per transfer
16x2	16 2-bit values per transfer
8x4	8 4-bit (nibble) values per transfer
4x8	4 8-bit (byte) values per transfer
2x16	2 16-bit (word) values per transfer
1x32	1 32-bit (long) value per transfer

Related Documentation

Chapter 5.3 (Streamer) provides the architectural overview of the Streamer subsystem, including its relationship with the FIFO, capabilities, and programming model. Refer to that section for conceptual understanding before using these mode constants.

Related Instructions

- XINIT — Initialize Streamer with mode and NCO rate
- XCONT — Continue Streamer with new parameters
- XSTOP — Stop Streamer operation
- XZERO — Zero Streamer and stop
- RDFAST — Set up hub-to-FIFO reading
- WRFAST — Set up FIFO-to-hub writing
- SETLUTS — Configure LUT for Streamer use

Appendix H: Reserved Words Reference

This appendix lists all reserved words recognized by the Propeller 2 compiler. These identifiers cannot be used as user-defined labels, symbols, or variable names. Attempting to use a reserved **WORD** as a label will result in an assembly error.

Important: Since Spin2 and PASM2 share a single compiler, **all reserved words from both languages apply** regardless of whether you are writing pure PASM2 or mixed Spin2/PASM2 code.

Total Reserved Words: 852+ (456 PASM2 + 396 Spin2; P_/X_ hardware constants add ~194 more — see Grand Total below)

Quick Reference Index

Use this alphabetical index to quickly check if a name is reserved. For detailed descriptions and usage context, see the categorized sections that follow.

Note: P_* constants (Smart Pin, ~116 words) are listed in Appendix E. X_* constants (Streamer, ~78 words) are listed in Appendix F. Both prefixes are reserved.

A

```
ABS         ABORT       ADDBITS     ADD         ADDCT1      ADDCT2
ADDCT3      ADDPIX      ADDPINS     ADDS        ADDSX       ADDX
AKPIN       ALIGNL      ALIGNW      ALLOWI      ALT         ALTB
ALTD        ALTGB       ALTGN       ALTGW       ALTI        ALTR
ALTS        ALTSB       ALTSN       ALTSW       AND         ANDC
ANDN        ANDZ        ARCHIVE     ASMCLK      AUGD        AUGS
```

B

```
BACKCOLOR   BITMAP      BITC        BITH        BITL        BITNC
BITNOT      BITNZ       BITRND      BITZ        BLACK       BLNPIX
BLUE        BMASK       BOX         BRK         BYTE        BYTEFILL
BYTEFIT     BYTEMOVE    BYTES_1BIT  BYTES_2BIT  BYTES_4BIT
```

C

```
CALL        CALLA       CALLB       CALLD       CALLPA      CALLPB
CARTESIAN   CASE        CASE_FAST   CHANNEL     CIRCLE      CLEAR
CLKFREQ     CLKMODE     CLKSET      CLOSE       CMP         CMPM
CMPR        CMPS        CMPSUB      CMPSX       CMPX        COGBRK
COGCHK      COGEXEC     COGEXEC_NEW COGEXEC_NEW_PAIR        COGATN
COGID       COGINIT     COGSPIN     COGSTOP     COLOR       CON
CRCBIT      CRCNIB      CYAN
```

D

```
DAT                 DEBUG               DEBUG_BAUD          DEBUG_COGS
DEBUG_COGINIT       DEBUG_DELAY         DEBUG_DISABLE       DEBUG_DISPLAY_LEFT
DEBUG_DISPLAY_TOP   DEBUG_HEIGHT        DEBUG_LEFT          DEBUG_LOG_SIZE
DEBUG_MAIN          DEBUG_MASK          DEBUG_PIN           DEBUG_PIN_RX
DEBUG_PIN_TX        DEBUG_TIMESTAMP     DEBUG_TOP           DEBUG_WIDTH
DEBUG_WINDOWS_OFF
DECMOD      DECOD       DEPTH       DIRA        DIRB        DIRC
DIRH        DIRL        DIRNC       DIRNOT      DIRNZ
DIRRND      DIRZ        DITTO       DJF         DJNF        DJNZ
DJZ         DLY         DOT         DOTSIZE     DRVC        DRVH
DRVL        DRVNC       DRVNOT      DRVNZ       DRVRND      DRVZ
```

E

```
ELSE        ELSEIF      ELSEIFNOT   ENCOD       END         EVENT_ATN
EVENT_CT1   EVENT_CT2   EVENT_CT3   EVENT_FBW   EVENT_INT   EVENT_PAT
EVENT_QMT   EVENT_SE1   EVENT_SE2   EVENT_SE3   EVENT_SE4   EVENT_XFI
EVENT_XMT   EVENT_XRL   EVENT_XRO   EXECF
```

F

```
FABS        FALSE       FBLOCK      FDEC        FDEC_       FDEC_ARRAY
FDEC_ARRAY_ FDEC_REG_ARRAY          FDEC_REG_ARRAY_         FFT
FGE         FGES        FILE        FIT         FLE         FLES
FLOAT       FLTC        FLTH        FLTL        FLTNC       FLTNOT
FLTNZ       FLTRND      FLTZ        FRAC        FROM        FSQRT
FVAR        FVARS
```

G

```
GETBRK      GETBYTE     GETCRC      GETCT       GETMS       GETNIB
GETPTR      GETQX       GETQY       GETREGS     GETRND      GETSCP
GETSEC      GETWORD     GETXACC     GREEN       GREY
```

H

```
HIDEXY      HOLDOFF     HSV8        HSV8W       HSV8X       HSV16
HSV16W      HSV16X      HUBEXEC     HUBEXEC_NEW HUBEXEC_NEW_PAIR
HUBSET
```

I

```
IF          IF_00       IF_0000     IF_0001     IF_0010     IF_0011
IF_01       IF_0100     IF_0101     IF_0110     IF_0111     IF_0X
IF_10       IF_1000     IF_1001     IF_1010     IF_1011     IF_11
IF_1100     IF_1101     IF_1110     IF_1111     IF_1X       IF_A
IF_AE       IF_ALWAYS   IF_B        IF_BE       IF_C        IF_C_AND_NZ
IF_C_AND_Z  IF_C_EQ_Z   IF_C_NE_Z   IF_C_OR_NZ  IF_C_OR_Z   IF_DIFF
IF_E        IF_GE       IF_GT       IF_LE       IF_LT       IF_NC
IF_NC_AND_NZ            IF_NC_AND_Z IF_NC_OR_NZ IF_NC_OR_Z  IF_NE
IF_NOT_00   IF_NOT_01   IF_NOT_10   IF_NOT_11   IF_NZ
IF_NZ_AND_C IF_NZ_AND_NC            IF_NZ_OR_C  IF_NZ_OR_NC IF_SAME
IF_X0       IF_X1       IF_Z        IF_Z_AND_C  IF_Z_AND_NC IF_Z_EQ_C
IF_Z_NE_C   IF_Z_OR_C   IF_Z_OR_NC  IFNOT       IJMP1       IJMP2
IJMP3       IJNZ        IJZ         INA         INB         INCMOD
INT_OFF     IRET1       IRET2       IRET3
```

J

```
JATN        JCT1        JCT2        JCT3        JFBW        JINT
JMP         JMPREL      JNATN       JNCT1       JNCT2       JNCT3
JNFBW       JNINT       JNPAT       JNQMT       JNSE1       JNSE2
JNSE3       JNSE4       JNXFI       JNXMT       JNXRL       JNXRO
JPAT        JQMT        JSE1        JSE2        JSE3        JSE4
JXFI        JXMT        JXRL        JXRO
```

L

```
LINE        LINESIZE    LOC         LOCKCHK     LOCKNEW     LOCKREL
LOCKRET     LOCKTRY     LOGIC       LONG        LONGFILL    LONGMOVE
LONGS_16BIT LONGS_1BIT  LONGS_2BIT  LONGS_4BIT  LONGS_8BIT
LOOKDOWN    LOOKDOWNZ   LOOKUP      LOOKUPZ     LSTR        LSTR_
LUMA8       LUMA8W      LUMA8X      LUT1        LUT2        LUT4
LUT8        LUTCOLORS
```

M

```
MAG         MAGENTA     MERGEB      MERGEW      MIDI        MIXPIX
MODC        MODCZ       MODZ        MOV         MOVBYTS     MUL
MULDIV64    MULPIX      MULS        MUXC        MUXNC       MUXNIBS
MUXNITS     MUXNZ       MUXQ        MUXZ
```

N

```
NAN         NEG         NEGC        NEGNC       NEGNZ       NEGX
NEGZ        NEWCOG      NEXT        NIXINT1     NIXINT2     NIXINT3
NOP         NOT
```

O

```
OBJ          OBOX         ONES         OPACITY      OR           ORANGE
ORC          ORG          ORGF         ORGH         ORIGIN       ORZ
OTHER        OUTA         OUTB         OUTC         OUTH         OUTL
OUTNC        OUTNOT       OUTNZ        OUTRND       OUTZ         OVAL
```

P

```
PA           PB           PC_KEY       PC_MOUSE     PI           PINCLEAR
PINF         PINFLOAT     PINH         PINHIGH      PINL         PINLOW
PINR         PINREAD      PINSTART     PINT         PINTOGGLE    PINW
PINWRITE     PLOT         POLAR        POLLATN      POLLCT       POLLCT1
POLLCT2      POLLCT3      POLLFBW      POLLINT      POLLPAT      POLLQMT
POLLSE1      POLLSE2      POLLSE3      POLLSE4      POLLXFI      POLLXMT
POLLXRL      POLLXRO      POLXY        POP          POPA         POPB
POS          POSX         PR0          PR1          PR2          PR3
PR4          PR5          PR6          PR7          PRECISE      PRECOMPILE
PRI          PTRA         PTRB         PUB          PUSH         PUSHA
PUSHB
```

Q

```
QCOS         QDIV         QEXP         QFRAC        QLOG         QMUL
QROTATE      QSIN         QSQRT        QUIT         QVECTOR
```

R

```
RANGE        RCL          RCR          RCZL         RCZR         RDBYTE
RDFAST       RDLONG       RDLUT        RDPIN        RDWORD       RECV
RED          REG          REGEXEC      REGLOAD      REP          REPEAT
RES          RESI0        RESI1        RESI2        RESI3        RET
RETA         RETB         RETI0        RETI1        RETI2        RETI3
RETURN       REV          RFBYTE       RFLONG       RFVAR        RFVARS
RFWORD       RGB8         RGB16        RGB24        RGBEXP       RGBI8
RGBI8W       RGBI8X       RGBSQZ       ROL          ROLBYTE      ROLNIB
ROLWORD      ROR          ROTXY        ROUND        RQPIN
```

S

```
SAL          SAMPLES      SAR          SAVE         SBIN         SBIN_
SBIN_BYTE    SBIN_BYTE_   SBIN_BYTE_ARRAY           SBIN_BYTE_ARRAY_
SBIN_LONG    SBIN_LONG_   SBIN_LONG_ARRAY           SBIN_LONG_ARRAY_
SBIN_REG_ARRAY            SBIN_REG_ARRAY_           SBIN_WORD    SBIN_WORD_
SBIN_WORD_ARRAY           SBIN_WORD_ARRAY_          SCA          SCAS
SCOPE        SCOPE_XY     SCROLL       SDEC         SDEC_        SDEC_BYTE
SDEC_BYTE_   SDEC_BYTE_ARRAY           SDEC_BYTE_ARRAY_          SDEC_LONG
SDEC_LONG_   SDEC_LONG_ARRAY           SDEC_LONG_ARRAY_          SDEC_REG_ARRAY
SDEC_REG_ARRAY_           SDEC_WORD    SDEC_WORD_   SDEC_WORD_ARRAY
```

```
SDEC_WORD_ARRAY_        SEND        SET         SETBYTE     SETCFRQ
SETCI       SETCMOD     SETCQ       SETCY       SETD        SETDACS
SETINT1     SETINT2     SETINT3     SETLUTS     SETNIB      SETPAT
SETPIV      SETPIX      SETQ        SETQ2       SETR        SETREGS
SETS        SETSCP      SETSE1      SETSE2      SETSE3      SETSE4
SETWORD     SETXFRQ     SEUSSF      SEUSSR      SHEX        SHEX_
SHEX_BYTE   SHEX_BYTE_  SHEX_BYTE_ARRAY         SHEX_BYTE_ARRAY_
SHEX_LONG   SHEX_LONG_  SHEX_LONG_ARRAY         SHEX_LONG_ARRAY_
SHEX_REG_ARRAY          SHEX_REG_ARRAY_         SHEX_WORD   SHEX_WORD_
SHEX_WORD_ARRAY         SHEX_WORD_ARRAY_        SHL         SHR
SIGNED      SIGNX       SIZE        SKIP        SKIPF       SPACING
SPECTRO     SPLITB      SPLITW      SPRITE      SPRITEDEF   SQRT
STALLI      STEP        STRCOMP     STRCOPY     STRING      STRSIZE
STRUCT      SUB         SUBR        SUBS        SUBSX       SUBX
SUMC        SUMNC       SUMNZ       SUMZ
```

T

```
TERM        TEST        TESTB       TESTBN      TESTN       TESTP
TESTPN      TEXT        TEXTANGLE   TEXTSIZE    TEXTSTYLE   TITLE
TJF         TJNF        TJNS        TJNZ        TJS         TJV
TJZ         TO          TRACE       TRGINT1     TRGINT2     TRGINT3
TRIGGER     TRUE        TRUNC
```

U

```
UBIN        UBIN_       UBIN_BYTE   UBIN_BYTE_  UBIN_BYTE_ARRAY
UBIN_BYTE_ARRAY_        UBIN_LONG   UBIN_LONG_  UBIN_LONG_ARRAY
UBIN_LONG_ARRAY_        UBIN_REG_ARRAY          UBIN_REG_ARRAY_
UBIN_WORD   UBIN_WORD_  UBIN_WORD_ARRAY         UBIN_WORD_ARRAY_
UDEC        UDEC_       UDEC_BYTE   UDEC_BYTE_  UDEC_BYTE_ARRAY
UDEC_BYTE_ARRAY_        UDEC_LONG   UDEC_LONG_  UDEC_LONG_ARRAY
UDEC_LONG_ARRAY_        UDEC_REG_ARRAY          UDEC_REG_ARRAY_
UDEC_WORD   UDEC_WORD_  UDEC_WORD_ARRAY         UDEC_WORD_ARRAY_
UHEX        UHEX_       UHEX_BYTE   UHEX_BYTE_  UHEX_BYTE_ARRAY
UHEX_BYTE_ARRAY_        UHEX_LONG   UHEX_LONG_  UHEX_LONG_ARRAY
UHEX_LONG_ARRAY_        UHEX_REG_ARRAY          UHEX_REG_ARRAY_
UHEX_WORD   UHEX_WORD_  UHEX_WORD_ARRAY         UHEX_WORD_ARRAY_
UNTIL       UPDATE
```

V-W

```
VAR         VARBASE     WAITATN     WAITCT      WAITCT1     WAITCT2
WAITCT3     WAITFBW     WAITINT     WAITMS      WAITPAT     WAITSE1
WAITSE2     WAITSE3     WAITSE4     WAITUS      WAITX       WAITXFI
WAITXMT     WAITXRL     WAITXRO     WC          WCZ         WFBYTE
WFLONG      WFWORD      WHITE       WHILE       WINDOW      WITH
WMLONG      WORD        WORDFILL    WORDFIT     WORDMOVE    WORDS_1BIT
WORDS_2BIT  WORDS_4BIT  WORDS_8BIT  WRBYTE      WRC         WRFAST
WRLONG      WRLUT       WRNC        WRNZ        WRPIN       WRWORD
```

```
WRZ         WXPIN       WYPIN       WZ
```

X-Z

```
XCONT       XINIT       XOR         XORC        XORO32      XORZ
XSTOP       XYPOL       XZERO       YELLOW      ZEROX       ZSTR
ZSTR_
```

Underscore-Prefixed Conditions

```
_C          _CLR        _C_AND_NZ   _C_AND_Z    _C_EQ_Z     _C_NE_Z
_C_OR_NZ    _C_OR_Z     _E          _GE         _GT         _LE
_LT         _NC         _NC_AND_NZ  _NC_AND_Z   _NC_OR_NZ   _NC_OR_Z
_NE         _NZ         _NZ_AND_C   _NZ_AND_NC  _NZ_OR_C    _NZ_OR_NC
_RET_       _SET        _Z          _Z_AND_C    _Z_AND_NC   _Z_EQ_C
_Z_NE_C     _Z_OR_C     _Z_OR_NC
```

Categories

Reserved words fall into six main categories:

1. **Instruction Mnemonics** (358 words) - All instruction names
2. **Assembly Directives** (21 words) - Block identifiers and assembly-time directives
3. **Predefined Constants** (11 words) - Built-in constant values
4. **Special Register Names** (16 words) - Special-purpose registers
5. **Condition Keywords** (41 words) - Conditional execution prefixes
6. **Effect Keywords** (9 words) - Flag modification suffixes

Instruction Mnemonics (358 words)

All PASM2 instruction names are reserved. These appear in alphabetical order for quick reference:

```
ABS         ADD         ADDCT1      ADDCT2      ADDCT3      ADDPIX
ADDS        ADDSX       ADDX        AKPIN       ALLOWI      ALTB
ALTD        ALTGB       ALTGN       ALTGW       ALTI        ALTR
ALTS        ALTSB       ALTSN       ALTSW       AND         ANDN
ASMCLK      AUGD        AUGS        BITC        BITH        BITL
BITNC       BITNOT      BITNZ       BITRND      BITZ        BLNPIX
BMASK       BRK         CALL        CALLA       CALLB       CALLD
CALLPA      CALLPB      CMP         CMPM        CMPR        CMPS
CMPSUB      CMPSX       CMPX        COGATN      COGBRK      COGID
COGINIT     COGSTOP     CRCBIT      CRCNIB      DECMOD      DECOD
DIRC        DIRH        DIRL        DIRNC       DIRNOT      DIRNZ
DIRRND      DIRZ        DJF         DJNF        DJNZ        DJZ
DRVC        DRVH        DRVL        DRVNC       DRVNOT      DRVNZ
DRVRND      DRVZ        ENCOD       EXECF       FBLOCK      FGE
FGES        FLE         FLES        FLTC        FLTH        FLTL
```

```
FLTNC       FLTNOT      FLTNZ       FLTRND      FLTZ        GETBRK
GETBYTE     GETCT       GETNIB      GETPTR      GETQX       GETQY
GETRND      GETSCP      GETWORD     GETXACC     HUBSET      IJNZ
IJZ         INCMOD      JATN        JCT1        JCT2        JCT3
JFBW        JINT        JMP         JMPREL      JNATN       JNCT1
JNCT2       JNCT3       JNFBW       JNINT       JNPAT       JNQMT
JNSE1       JNSE2       JNSE3       JNSE4       JNXFI       JNXMT
JNXRL       JNXRO       JPAT        JQMT        JSE1        JSE2
JSE3        JSE4        JXFI        JXMT        JXRL        JXRO
LOC         LOCKNEW     LOCKREL     LOCKRET     LOCKTRY     MERGEB
MERGEW      MIXPIX      MODC        MODCZ       MODZ        MOV
MOVBYTS     MUL         MULPIX      MULS        MUXC        MUXNC
MUXNIBS     MUXNITS     MUXNZ       MUXQ        MUXZ        NEG
NEGC        NEGNC       NEGNZ       NEGZ        NIXINT1     NIXINT2
NIXINT3     NOP         NOT         ONES        OR          OUTC
OUTH        OUTL        OUTNC       OUTNOT      OUTNZ       OUTRND
OUTZ        POLLATN     POLLCT1     POLLCT2     POLLCT3     POLLFBW
POLLINT     POLLPAT     POLLQMT     POLLSE1     POLLSE2     POLLSE3
POLLSE4     POLLXFI     POLLXMT     POLLXRL     POLLXRO     POP
POPA        POPB        PUSH        PUSHA       PUSHB       QDIV
QEXP        QFRAC       QLOG        QMUL        QROTATE     QSQRT
QVECTOR     RCL         RCR         RCZL        RCZR        RDBYTE
RDFAST      RDLONG      RDLUT       RDPIN       RDWORD      REP
RESI0       RESI1       RESI2       RESI3       RET         RETA
RETB        RETI0       RETI1       RETI2       RETI3       REV
RFBYTE      RFLONG      RFVAR       RFVARS      RFWORD      RGBEXP
RGBSQZ      ROL         ROLBYTE     ROLNIB      ROLWORD     ROR
RQPIN       SAL         SAR         SCA         SCAS        SETBYTE
SETCFRQ     SETCI       SETCMOD     SETCQ       SETCY       SETD
SETDACS     SETINT1     SETINT2     SETINT3     SETLUTS     SETNIB
SETPAT      SETPIV      SETPIX      SETQ        SETQ2       SETR
SETS        SETSCP      SETSE1      SETSE2      SETSE3      SETSE4
SETWORD     SETXFRQ     SEUSSF      SEUSSR      SHL         SHR
SIGNX       SKIP        SKIPF       SPLITB      SPLITW      STALLI
SUB         SUBR        SUBS        SUBSX       SUBX        SUMC
SUMNC       SUMNZ       SUMZ        TEST        TESTB       TESTBN
TESTN       TESTP       TESTPN      TJF         TJNF        TJNS
TJNZ        TJS         TJV         TJZ         TRGINT1     TRGINT2
TRGINT3     WAITATN     WAITCT1     WAITCT2     WAITCT3     WAITFBW
WAITINT     WAITPAT     WAITSE1     WAITSE2     WAITSE3     WAITSE4
WAITX       WAITXFI     WAITXMT     WAITXRL     WAITXRO     WFBYTE
WFLONG      WFWORD      WMLONG      WRBYTE      WRC         WRFAST
WRLONG      WRLUT       WRNC        WRNZ        WRPIN       WRWORD
WRZ         WXPIN       WYPIN       XCONT       XINIT       XOR
XORO32      XSTOP       XZERO       ZEROX
```

Assembly Directives (21 words)

Directives control the assembly process and code organization:

Block/Section Identifiers (7)

These keywords define the major sections of a Spin2/PASM2 source file:

- **CON** - Constants block (define named constants)
- **DAT** - Data block (contains PASM2 code and data)
- **FILE** - Include binary file in DAT section
- **OBJ** - Objects block (instantiate child objects)
- **PRI** - Private method block
- **PUB** - Public method block
- **VAR** - Variables block (instance variables)

Assembly-Time Directives (14)

- **ALIGNL** - Align to next **LONG** boundary (4-byte alignment)
- **ALIGNW** - Align to next **WORD** boundary (2-byte alignment)
- **BYTE** - Reserve/initialize byte-sized data
- **BYTEFIT** - Verify code fits in specified **BYTE** count
- **DEBUG** - Insert **DEBUG** statements (Spin2 feature)
- **DITTO** - Repeat previous instruction encoding
- **FIT** - Verify code fits in COG memory
- **LONG** - Reserve/initialize long-sized data (32 bits)
- **ORG** - Set assembly origin (COG address)
- **ORGF** - Set assembly origin with fill
- **ORGH** - Set assembly origin (Hub address)
- **RES** - Reserve uninitialized registers/memory
- **WORD** - Reserve/initialize word-sized data (16 bits)
- **WORDFIT** - Verify code fits in specified **WORD** count

Predefined Constants (11 words)

Built-in constants that can be used in assembly expressions:

Basic Constants (5)

- **FALSE** - Boolean false value (`$00000000`, decimal 0)
- **NEGX** - Most negative signed 32-bit value (`$80000000`, decimal -2147483648)
- **PI** - Fixed-point pi value for CORDIC operations
- **POSX** - Most positive signed 32-bit value (`$7FFFFFFF`, decimal 2147483647)
- **TRUE** - Boolean true value (`$FFFFFFFF`, decimal -1)

Execution Mode Constants (6)

Used with the **COGINIT** instruction to specify execution mode:

- **COGEXEC** - Execute from COG RAM (base mode, `%0_0_0000`)
- **COGEXEC_NEW** - Auto-select available COG, execute from COG RAM

- **COGEXEC_NEW_PAIR** - Auto-select COG pair, execute from COG RAM
- **HUBEXEC** - Execute from Hub RAM (base mode, `%0_1_0000`)
- **HUBEXEC_NEW** - Auto-select available COG, execute from Hub RAM
- **HUBEXEC_NEW_PAIR** - Auto-select COG pair, execute from Hub RAM

Note: The `_NEW` and `_NEW_PAIR` variants are bit patterns that modify the base `COGEXEC` and `HUBEXEC` constants for use with **COGINIT**'s automatic COG selection feature.

Special Register Names (16 words)

Special-purpose registers mapped to COG RAM addresses `$1F0-$1FF`:

Dual-Purpose Registers ($1F0-$1F7)

Can be used as general RAM or special registers depending on enabled features:

- **IJMP3** - Interrupt 3 jump address ($1F0, 496)
- **IRET3** - Interrupt 3 return address ($1F1, 497)
- **IJMP2** - Interrupt 2 jump address ($1F2, 498)
- **IRET2** - Interrupt 2 return address ($1F3, 499)
- **IJMP1** - Interrupt 1 jump address ($1F4, 500)
- **IRET1** - Interrupt 1 return address ($1F5, 501)
- **PA** - Multi-purpose register A ($1F6, 502)
- **PB** - Multi-purpose register B ($1F7, 503)

Fixed Special Registers ($1F8-$1FF)

Always provide special functions when accessed:

- **PTRA** - Pointer A to Hub RAM ($1F8, 504)
- **PTRB** - Pointer B to Hub RAM ($1F9, 505)
- **DIRA** - Direction register for pins 0-31 ($1FA, 506)
- **DIRB** - Direction register for pins 32-63 ($1FB, 507)
- **OUTA** - Output register for pins 0-31 ($1FC, 508)
- **OUTB** - Output register for pins 32-63 ($1FD, 509)
- **INA** - Input register for pins 0-31 ($1FE, 510)
- **INB** - Input register for pins 32-63 ($1FF, 511)

Condition Keywords (41 words)

Conditional execution prefixes (IF_xxx) that can be applied to any instruction. These test the C (Carry) and Z (Zero) flags:

Primary Condition Codes (16)

These are the canonical condition names:

- **IF_ALWAYS** - Always execute (EEEE=1111; this is the encoding used when no condition is specified)
- **_RET_** - Execute instruction, then return if no branch (EEEE=0000; note: P1's IF_NEVER does NOT exist in P2)
- **IF_C** - Execute if C=1
- **IF_NC** - Execute if C=0
- **IF_Z** - Execute if Z=1
- **IF_NZ** - Execute if Z=0
- **IF_C_AND_Z** - Execute if C=1 AND Z=1
- **IF_C_AND_NZ** - Execute if C=1 AND Z=0
- **IF_NC_AND_Z** - Execute if C=0 AND Z=1
- **IF_NC_AND_NZ** - Execute if C=0 AND Z=0
- **IF_C_OR_Z** - Execute if C=1 OR Z=1
- **IF_C_OR_NZ** - Execute if C=1 OR Z=0
- **IF_NC_OR_Z** - Execute if C=0 OR Z=1
- **IF_NC_OR_NZ** - Execute if C=0 OR Z=0
- **IF_C_EQ_Z** - Execute if C equals Z
- **IF_C_NE_Z** - Execute if C not equal to Z

Comparison Aliases (15)

Convenient aliases for post-comparison conditional execution. Two equivalent terminology styles are available—both encode to identical condition codes:

Magnitude terminology aliases:

- **IF_A** - Above (same as IF_NC_AND_NZ)
- **IF_AE** - Above or equal (same as IF_NC)
- **IF_B** - Below (same as IF_C)
- **IF_BE** - Below or equal (same as IF_C_OR_Z)
- **IF_E** - Equal (same as IF_Z)
- **IF_NE** - Not equal (same as IF_NZ)

Arithmetic terminology aliases:

- **IF_GE** - Greater or equal (same as IF_NC)
- **IF_GT** - Greater than (same as IF_NC_AND_NZ)
- **IF_LE** - Less or equal (same as IF_C_OR_Z)
- **IF_LT** - Less than (same as IF_C)

Other aliases:

- **IF_DIFF** - Different (same as IF_C_NE_Z)
- **IF_SAME** - Same (same as IF_C_EQ_Z)
- **IF_NZ_AND_C** - Not zero and carry (same as IF_C_AND_NZ)
- **IF_NZ_AND_NC** - Not zero and no carry (same as IF_NC_AND_NZ)

- **IF_Z_AND_C** - Zero and carry (same as IF_C_AND_Z)

Special Return Condition (1)

- **_RET_** - Always execute instruction, then return if no branch (no flag restore)

Symmetric Alternatives (9)

Additional aliases that express the same conditions in reverse order:

- **IF_Z_AND_NC** - Same as IF_NC_AND_Z
- **IF_Z_OR_C** - Same as IF_C_OR_Z
- **IF_Z_OR_NC** - Same as IF_NC_OR_Z
- **IF_NZ_OR_C** - Same as IF_C_OR_NZ
- **IF_NZ_OR_NC** - Same as IF_NC_OR_NZ

Note: Many conditions have multiple valid names (aliases). For example, `IF_C`, `IF_B`, and `IF_LT` all represent the same condition code but provide semantic clarity depending on context.

Effect Keywords (9 words)

Effect suffixes control flag updates after instruction execution:

Basic Effect Modifiers (3)

- **WC** - Write result to Carry flag
- **WZ** - Write result to Zero flag
- **WCZ** - Write result to both Carry and Zero flags

Logical Effect Modifiers (6)

Combine instruction result with existing flag using logic operation:

- **ANDC** - AND result with C flag
- **ANDZ** - AND result with Z flag
- **ORC** - OR result with C flag
- **ORZ** - OR result with Z flag
- **XORC** - **XOR** result with C flag
- **XORZ** - **XOR** result with Z flag

Usage: Effect keywords appear after the instruction's operands:

```
1 ADD   x, y  WC      ' Update C flag with carry
2 CMP   a, b  WCZ     ' Update both C and Z flags
3 TEST  val, mask  ANDZ   ' AND test result with Z flag
```

Avoiding Reserved Words

When naming labels, variables, and symbols in your PASM2 code:

1. **Check this reference** before choosing identifiers
2. **Use descriptive names** that clearly differ from reserved words
3. **Add prefixes/suffixes** to avoid conflicts (e.g., `my_add`, `loop_counter`)
4. **Case sensitivity:** PASM2 is case-insensitive - `MOV`, `MOV`, and `MOV` are all reserved

Common Naming Strategies

- Add application-specific prefixes: `uart_receive`, `led_toggle`
- Add type suffixes: `count_value`, `delay_ms`
- Use underscores: `_start`, `main_loop`, `temp_reg`
- Combine words: `blink_rate`, `max_count`

Example Conflicts to Avoid

```
' WRONG - uses reserved words as labels
ADD         MOV   x, #1      ' Error: 'add' is instruction
OR          JMP   #loop      ' Error: 'or' is instruction
BYTE        LONG  $0         ' Error: 'byte' is directive
```

```
' CORRECT - uses valid label names
add_routine     MOV   x, #1
choice_or       JMP   #loop
byte_data       LONG  $0
```

Summary

The Propeller 2 compiler reserves **852+ identifiers** across PASM2 and Spin2:

PASM2-Specific Reserved Words (456):

Category	Count	Purpose
Instructions	358	All instruction mnemonics
Directives	21	Block identifiers and assembly-time directives
Constants	11	Predefined constant values
Special Registers	16	Hardware-mapped registers
Conditions	41	Conditional execution prefixes
Effects	9	Flag modification suffixes
PASM2 Subtotal	**456**	

Spin2-Specific Reserved Words (396):

Category	Count	Purpose
Language Keywords	20	Core Spin2 constructs
DEBUG Parameters	120	Debug output formatting
Graphics/Color	34	Color names and display
String/Data Methods	22	Memory/string manipulation
Math/Conversion	11	Math functions
Event Constants	16	Event source identifiers
Pin Methods	14	High-level pin control
Condition Shortcuts	32	Underscore-prefixed conditions
IF_ Variants	28	Extended condition patterns
Shared Registers	8	PR0-PR7 communication
System/I/O	27	System control methods
Graphics Drawing	32	Graphics primitives
Text/Display	12	Text rendering
Lookup/Misc	20	Table lookup and other
Spin2 Subtotal	**396**	

Hardware Constants (194+):

Category	Count	Purpose
Smart Pin (P_*)	~116	Pin configuration
Streamer (X_*)	~78	Streamer modes
Constants Subtotal	**~194**	

Grand Total: 1,046+ reserved identifiers

Cross-References:

- **Part II** — Complete documentation of instructions, directives, constants, and special registers
- **Chapter 3** — Detailed explanation of condition codes and effect modifiers
- **Appendix E** — Smart Pin mode constants (P_* symbols, approximately 116 constants)
- **Appendix F** — Streamer mode constants (X_* symbols, approximately 78 constants)

Note on P_* and X_* Constants: The Smart Pin configuration constants (P_*) and Streamer mode constants (X_*)* are predefined symbols that function as reserved words when programming the P2's Smart Pins and Streamer hardware. These are documented in their own appendices due to their specialized nature and extensive count. While not included in the 456-word count above, they are effectively reserved and cannot be used as user-defined symbols.

Spin2 Reserved Words

Since the Propeller 2 uses a single compiler for both Spin2 and PASM2, **all Spin2 reserved words are also reserved in PASM2**. You cannot use any of these identifiers as labels, symbols, or variable names in your assembly code, even when writing pure PASM2.

Total Spin2-Only Reserved Words: 396

The following sections list Spin2 reserved words organized by category.

Language Keywords (20 words)

Core Spin2 language constructs (block names CON, DAT, VAR, PUB, PRI, OBJ are listed under PASM2 Assembly Directives):

```
ABORT       CASE        CASE_FAST   ELSE        ELSEIF      ELSEIFNOT
END         FROM        IF          IFNOT       NEXT        OTHER
QUIT        REPEAT      RETURN      STRUCT      TO          UNTIL
WHILE       WITH
```

Note: STRUCT requires Spin2 v45 or later; WITH is the REPEAT positive-count loop-counter binding (`REPEAT <count> WITH <var>`).

DEBUG Command Parameters (120 words)

Debug output formatting commands and their variants:

Configuration Symbols:

```
DEBUG_BAUD          DEBUG_COGS          DEBUG_COGINIT       DEBUG_DELAY
DEBUG_DISABLE       DEBUG_DISPLAY_LEFT  DEBUG_DISPLAY_TOP   DEBUG_HEIGHT
DEBUG_LEFT          DEBUG_LOG_SIZE      DEBUG_MAIN          DEBUG_MASK
DEBUG_PIN           DEBUG_PIN_RX        DEBUG_PIN_TX        DEBUG_TIMESTAMP
DEBUG_TOP           DEBUG_WIDTH         DEBUG_WINDOWS_OFF
```

Signed decimal (SDEC) variants:

```
SDEC        SDEC_       SDEC_BYTE       SDEC_BYTE_      SDEC_BYTE_ARRAY
SDEC_BYTE_ARRAY_    SDEC_LONG       SDEC_LONG_      SDEC_LONG_ARRAY
SDEC_LONG_ARRAY_    SDEC_REG_ARRAY  SDEC_REG_ARRAY_  SDEC_WORD
SDEC_WORD_          SDEC_WORD_ARRAY SDEC_WORD_ARRAY_
```

Unsigned decimal (UDEC) variants:

```
UDEC        UDEC_       UDEC_BYTE       UDEC_BYTE_      UDEC_BYTE_ARRAY
UDEC_BYTE_ARRAY_    UDEC_LONG       UDEC_LONG_      UDEC_LONG_ARRAY
UDEC_LONG_ARRAY_    UDEC_REG_ARRAY  UDEC_REG_ARRAY_  UDEC_WORD
UDEC_WORD_          UDEC_WORD_ARRAY UDEC_WORD_ARRAY_
```

Signed hex (SHEX) variants:

```
SHEX        SHEX_       SHEX_BYTE        SHEX_BYTE_       SHEX_BYTE_ARRAY
SHEX_BYTE_ARRAY_      SHEX_LONG        SHEX_LONG_       SHEX_LONG_ARRAY
SHEX_LONG_ARRAY_      SHEX_REG_ARRAY   SHEX_REG_ARRAY_  SHEX_WORD
SHEX_WORD_            SHEX_WORD_ARRAY  SHEX_WORD_ARRAY_
```

Unsigned hex (UHEX) variants:

```
UHEX        UHEX_       UHEX_BYTE        UHEX_BYTE_       UHEX_BYTE_ARRAY
UHEX_BYTE_ARRAY_      UHEX_LONG        UHEX_LONG_       UHEX_LONG_ARRAY
UHEX_LONG_ARRAY_      UHEX_REG_ARRAY   UHEX_REG_ARRAY_  UHEX_WORD
UHEX_WORD_            UHEX_WORD_ARRAY  UHEX_WORD_ARRAY_
```

Signed binary (SBIN) variants:

```
SBIN        SBIN_       SBIN_BYTE        SBIN_BYTE_       SBIN_BYTE_ARRAY
SBIN_BYTE_ARRAY_      SBIN_LONG        SBIN_LONG_       SBIN_LONG_ARRAY
SBIN_LONG_ARRAY_      SBIN_REG_ARRAY   SBIN_REG_ARRAY_  SBIN_WORD
SBIN_WORD_            SBIN_WORD_ARRAY  SBIN_WORD_ARRAY_
```

Unsigned binary (UBIN) variants:

```
UBIN        UBIN_       UBIN_BYTE        UBIN_BYTE_       UBIN_BYTE_ARRAY
UBIN_BYTE_ARRAY_      UBIN_LONG        UBIN_LONG_       UBIN_LONG_ARRAY
UBIN_LONG_ARRAY_      UBIN_REG_ARRAY   UBIN_REG_ARRAY_  UBIN_WORD
UBIN_WORD_            UBIN_WORD_ARRAY  UBIN_WORD_ARRAY_
```

Floating-point decimal (FDEC) variants:

```
FDEC        FDEC_       FDEC_ARRAY       FDEC_ARRAY_      FDEC_REG_ARRAY
FDEC_REG_ARRAY_
```

Graphics and Color Constants (34 words)

Color names and graphics-related constants:

```
BACKCOLOR   BLACK       BLUE        COLOR       CYAN        DEPTH
GREEN       GREY        MAGENTA     OPACITY     ORANGE      RED
WHITE       YELLOW
```

HSV color conversion:

```
HSV8        HSV8W       HSV8X       HSV16       HSV16W      HSV16X
```

RGB color formats:

```
RGB8        RGB16       RGB24       RGBI8       RGBI8W      RGBI8X
```

Luminance and LUT:

```
LUMA8       LUMA8W      LUMA8X      LUT1        LUT2        LUT4
LUT8        LUTCOLORS
```

String and Data Methods (22 words)

Memory and string manipulation:

```
BYTEFILL    BYTEMOVE    LONGFILL    LONGMOVE    STRCOMP     STRCOPY
STRING      STRSIZE     WORDFILL    WORDMOVE
```

Bit-packing constants:

```
BYTES_1BIT  BYTES_2BIT  BYTES_4BIT
WORDS_1BIT  WORDS_2BIT  WORDS_4BIT  WORDS_8BIT
LONGS_1BIT  LONGS_2BIT  LONGS_4BIT  LONGS_8BIT  LONGS_16BIT
```

Math and Conversion Methods (11 words)

Mathematical functions available in Spin2:

```
FABS        FLOAT       FRAC        FSQRT       MULDIV64    NAN
QCOS        QSIN        ROUND       SQRT        TRUNC
```

Event Constants (16 words)

Event source identifiers for WAITSE and POLLSE:

```
EVENT_ATN   EVENT_CT1   EVENT_CT2   EVENT_CT3   EVENT_FBW   EVENT_INT
EVENT_PAT   EVENT_QMT   EVENT_SE1   EVENT_SE2   EVENT_SE3   EVENT_SE4
EVENT_XFI   EVENT_XMT   EVENT_XRL   EVENT_XRO
```

Pin Methods (14 words)

High-level pin manipulation methods:

```
PINCLEAR    PINF        PINFLOAT    PINH        PINHIGH     PINL
PINLOW      PINR        PINREAD     PINSTART    PINT        PINTOGGLE
PINW        PINWRITE
```

Condition Code Shortcuts (32 words)

Spin2 uses underscore-prefixed condition codes as shortcuts:

```
_C          _CLR        _E          _GE         _GT         _LE
_LT         _NC         _NE         _NZ         _SET        _Z
```

Compound conditions:

```
_C_AND_NZ   _C_AND_Z    _C_EQ_Z     _C_NE_Z     _C_OR_NZ    _C_OR_Z
_NC_AND_NZ  _NC_AND_Z   _NC_OR_NZ   _NC_OR_Z    _NZ_AND_C   _NZ_AND_NC
_NZ_OR_C    _NZ_OR_NC   _Z_AND_C    _Z_AND_NC   _Z_EQ_C     _Z_NE_C
_Z_OR_C     _Z_OR_NC
```

MODCZ Operand Values:

These mnemonics are used with the **MODCZ** instruction to modify C and Z flags. Each mnemonic represents a 4-bit value that selects the flag modification logic:

Value	Binary	Mnemonic	Description
0	0000	_CLR	Always clear (result = 0)
1	0001	_NC_AND_NZ	C=0 AND Z=0
2	0010	_NC_AND_Z	C=0 AND Z=1
3	0011	_NC	Copy inverse of C (not C)
4	0100	_C_AND_NZ	C=1 AND Z=0
5	0101	_NZ	Copy inverse of Z (not Z)
6	0110	_C_NE_Z	C XOR Z (C not equal to Z)
7	0111	_NC_OR_NZ	C=0 OR Z=0 (NAND)
8	1000	_C_AND_Z	C=1 AND Z=1 (AND)
9	1001	_C_EQ_Z	NOT(C XOR Z) (C equals Z)
10	1010	_Z	Copy Z
11	1011	_NC_OR_Z	C=0 OR Z=1
12	1100	_C	Copy C
13	1101	_C_OR_NZ	C=1 OR Z=0
14	1110	_C_OR_Z	C=1 OR Z=1 (OR)
15	1111	_SET	Always set (result = 1)

Common MODCZ Usage:

```
        MODCZ   _CLR, _SET      ' Clear C, set Z
        MODCZ   _SET, _CLR      ' Set C, clear Z
        MODCZ   _C, _Z          ' C and Z unchanged (copy to themselves)
        MODCZ   _Z, _C          ' Swap C and Z values
        MODCZ   _NC, _NZ        ' Invert both flags
```

Cross-Reference: See Part II **MODCZ** instruction for complete behavior description.

Additional IF_ Condition Variants (28 words)

Extended condition code patterns for bit-testing:

```
IF          IF_00       IF_0000     IF_0001     IF_0010     IF_0011
IF_01       IF_0100     IF_0101     IF_0110     IF_0111     IF_0X
IF_10       IF_1000     IF_1001     IF_1010     IF_1011     IF_11
IF_1100     IF_1101     IF_1110     IF_1111     IF_1X       IF_NOT_00
IF_NOT_01   IF_NOT_10   IF_NOT_11   IF_X0       IF_X1       IF_Z_EQ_C
IF_Z_NE_C   IFNOT
```

Shared Registers (8 words)

PASM2 to Spin2 communication registers:

```
PR0         PR1         PR2         PR3         PR4         PR5
PR6         PR7
```

System and I/O Methods (27 words)

System control and I/O operations (FILE is listed under PASM2 Assembly Directives):

```
CLKFREQ     CLKMODE     CLKSET      CLOSE       COGCHK      COGSPIN
GETCRC      GETMS       GETREGS     GETSEC      INT_OFF     LOCKCHK
NEWCOG      RECV        REG         REGEXEC     REGLOAD     SEND
SETREGS     UPDATE      VARBASE     WAITCT      WAITMS      WAITUS
WINDOW
```

Graphics Drawing Methods (32 words)

Graphics primitives and display control:

```
BITMAP      BOX         CARTESIAN   CIRCLE      CLEAR       DOT
DOTSIZE     FFT         HIDEXY      HOLDOFF     LINE        LINESIZE
LOGIC       OBOX        ORIGIN      OVAL        PC_KEY      PC_MOUSE
PLOT        POLAR       POLLCT      POLXY       POS         RANGE
ROTXY       SAMPLES     SAVE        SCOPE       SCOPE_XY    SCROLL
SPECTRO     XYPOL
```

Text and Display (12 words)

Text rendering parameters:

```
SPACING     SPRITE      SPRITEDEF   TERM        TEXT        TEXTANGLE
TEXTSIZE    TEXTSTYLE   TITLE       TRACE       TRIGGER     ZSTR
ZSTR_
```

Lookup and Miscellaneous (20 words)

Table lookup and other Spin2 features:

```
ADDBITS     ADDPINS     ALT         ARCHIVE     CHANNEL     DLY
FVAR        FVARS       LOOKDOWN    LOOKDOWNZ   LOOKUP      LOOKUPZ
LSTR        LSTR_       MAG         MIDI        PRECISE     PRECOMPILE
SET         SIGNED      SIZE        SQRT        STEP
```

Smart Pin Constants (P_*)

The complete list of Smart Pin configuration constants (116 constants) is documented in **Appendix E: Smart Pin Constants**. These include:

- Pin mode constants (P_ASYNC_TX, P_ASYNC_RX, P_SYNC_TX, etc.)
- DAC configuration (P_DAC_*, P_BITDAC)
- ADC configuration (P_ADC_*)
- Filter and logic modes (P_FILT*, P_LOGIC_*, P_COMPARE_*)
- Output drive strength (P_HIGH_*, P_LOW_*)
- Many more specialized pin configurations

All P_* constants are reserved words and cannot be used as user-defined symbols.

Streamer Constants (X_*)

The complete list of Streamer mode constants (78 constants) is documented in **Appendix F: Streamer Constants**. These include:

- Immediate mode constants (X_IMM_*)
- RF **BYTE**/word/long modes (X_RFBYTE_, *X_RFWORD_*, X_RFLONG_*)
- DAC output configurations (X_*DAC*)
- Control flags (X_PINS_ON, X_PINS_OFF, X_WRITE_ON, X_WRITE_OFF, etc.)

All X_* constants are reserved words and cannot be used as user-defined symbols.

Appendix I: Glossary of Encoding Terms

This glossary defines the terms used throughout the instruction encoding tables, syntax descriptions, and opcode documentation in this manual.

Encoding Field Terms

A / Addr A 20-bit relative or absolute value used to change PC (the program counter). This field appears in branch and **CALL** instructions where the 20-bit address occupies the two low bits of the CZI/FX field (positions 19-18) together with the D and S fields; the R bit (position 20) selects relative (PC += A) vs. absolute (PC = A) addressing.

C / Carry Flag A 1-bit persistent flag value representing a special state before or after instruction execution. Traditionally, the C flag indicates that an arithmetic operation resulted in a carry (addition) or borrow (subtraction). The P2 extends this with instruction-specific meanings for both input and output. When C appears in an instruction's opcode encoding, it indicates optional flag writing governed by the WC or WCZ effect.

CZI / FX Field The three bits at positions 20-18 in the instruction **WORD**. Bit 20 (C) enables writing to the C flag. Bit 19 (Z) enables writing to the Z flag. Bit 18 (I) indicates immediate mode for the S operand. Some instructions repurpose these bits for other functions, documented in the FX column of opcode tables.

D / Dest / Destination The target register that an instruction ultimately affects. Usually a 9-bit register address (0-511), but may be a 32-bit augmented value when preceded by an **AUGD** instruction. The destination register is often read, manipulated, and overwritten during instruction execution. The final value written is also called the Result.

EEEE / Condition Field The four bits at positions 31-28 that specify the execution condition. Default value 1111 means "always execute." Other values **TEST** combinations of C and Z flags—the instruction executes only if the condition is true.

Flag and State Terms

H / Hub Long A Hub RAM **LONG** (4 bytes) used to store subroutine calling context states. This includes the C and Z flags plus the return address, allowing nested subroutine calls to preserve and restore processor state.

I / Immediate Flag When set (I=1), the S field contains a literal value rather than a register address. When clear (I=0), the S field is a register address and the instruction reads from that register. The `#` prefix in source code sets this bit.

K / Stack The 8-level hardware stack used for subroutine calls and temporary storage. On CALL, the stack stores C, Z, and PC (return address). **PUSH** and **POP** provide general-purpose 32-bit value storage. Stack overflow/underflow wraps silently—there is no trap or error indication.

L / Literal Flag When set (L=1), the D field contains a literal value rather than a register address. This is less common than immediate S operands and appears in specific instructions. The `#` prefix on the destination in source code sets this bit where valid.

N / Index Number A small index value (typically 0-1, 0-3, or 0-7) used as a third operand in some instructions. Examples include interrupt numbers (1-3), event selector indices, and bit position specifiers.

PC / Program Counter A dedicated internal register that determines the next instruction address. Automatically increments by 1 (COG/LUT execution) or 4 (Hub execution) after each instruction unless altered by a branch. Not directly accessible but affected by **JMP**, **CALL**, **RET**, and conditional branches.

R / Relative Flag When set (R=1), the address field is interpreted relative to the current PC. When clear (R=0), the address is absolute. Relative addressing enables position-independent code. The `\` prefix forces absolute addressing; its absence allows relative.

Result The value written at the end of instruction execution. Usually stored in the Destination register, but some instructions write to special registers or memory instead. The Result value determines the Z flag when WZ is specified.

Z / Zero Flag A 1-bit persistent flag value traditionally indicating that an operation produced a zero result. The P2 extends this with instruction-specific meanings. When Z appears in an instruction's opcode encoding, it indicates optional flag writing governed by the WZ or WCZ effect. The Z flag is also used for equality testing in comparisons.

Operand Terms

S / Src / Source The origin value that instructions operate with. Can be a 9-bit literal value (when I=1), a register address (when I=0), or a 32-bit augmented value (when preceded by **AUGS** or the `##` prefix). The S field occupies bits 8-0 of the instruction **WORD**.

W / Write Register A 2-bit field (values 00-11) that selects which special register to write in certain instructions. The values map to PA (00), PB (01), PTRA (10), and PTRB (11). This appears in instructions that can target pointer registers.

Opcode Table Columns

Column	Description
COND	Bits 31-28: Execution condition (EEEE pattern)
INSTR	Bits 27-21: Instruction opcode (7 bits)
FX	Bits 20-18: Flag effects and immediate mode (CZI or special)
DEST	Bits 17-9: Destination operand (9 bits)
SRC	Bits 8-0: Source operand (9 bits)
Write	What the instruction modifies (register, memory, flags)
C Flag	How the C flag is affected (if WC specified)
Z Flag	How the Z flag is affected (if WZ specified)
Clocks	Execution time in system clock cycles

Related Documentation

- **Chapter 2** — Detailed explanation of instruction encoding format
- **Chapter 3** — Complete coverage of flag behavior and conditional execution
- **Appendix A** — Encoding summary tables with complete opcode bit patterns

Appendix J: Known Silicon Bugs

This appendix documents known hardware bugs in the P2 silicon that affect instruction behavior. These bugs cannot be fixed in software updates—they are permanent characteristics of the P2X8C4M64P **REV** B/C silicon.

ALTx/AUGx Interference with SETQ Block Transfers

Affected Instructions: SETQ, **SETQ**2, **RDLONG**, **WRLONG**, **WMLONG** with PTRx expressions

Bug Description:

When **SETQ** or **SETQ**2 precedes **RDLONG**, **WRLONG**, or **WMLONG** to set up a block transfer, intervening ALTx, **AUGS**, or **AUGD** instructions cancel the special-case block-size PTRx delta calculation. The expected number of longs transfers correctly, but PTRx is modified according to normal PTRx expression behavior rather than the block-adjusted delta.

Example of Bug:

```
1         SETQ    #16-1           ' Ready to load 16 longs
2         ALTD    start_reg       ' BUG: Cancels block-size PTRx delta!
3         RDLONG  0, ptra++       ' ptra += 4 (not 64!)
```

Expected Behavior: After reading 16 longs with `ptra++`, ptra should advance by 64 bytes (16 × 4).

Actual Behavior: ptra advances by only 4 bytes (1 **LONG**) because the **ALTD** instruction between **SETQ** and **RDLONG** cancels the block-size adjustment.

Workaround:

Manually adjust PTRx after the block transfer, or restructure code to avoid ALTx/AUGx instructions between **SETQ**/SETQ2 and the subsequent **RDLONG**/WRLONG/WMLONG.

```
1         ' Workaround: Adjust pointer manually after transfer
2         SETQ    #16-1           ' Ready to load 16 longs
3         ALTD    start_reg       ' Alter start register
4         RDLONG  0, ptra++       ' ptra only advances by 4
5         ADD     ptra, #(16-1)*4 ' Manually add remaining 60 bytes
```

AUGS Leakage to Intervening ALTx Instructions

Affected Instructions: **AUGS**, **ALTD**, **ALTS**, **ALTR**, and all ALTx variants

Bug Description:

When **AUGS** precedes an instruction with an immediate #S operand (its intended target), intervening ALTx instructions that also have an immediate #S operand will consume the **AUGS** value without canceling it. Both the intervening ALTx and the intended target instruction receive the augmented value.

Example of Bug:

```
1         AUGS    #$FFFFF123      ' Intended for ADD instruction
2         ALTD    index, #base    ' WARNING: #base also receives AUGS value!
3         ADD     0-0, #$123      ' #$123 is augmented, cancels AUGS
```

Expected Behavior: **AUGS** should only affect the **ADD** instruction's #$123 operand.

Actual Behavior: **AUGS** affects both `#base` in the **ALTD** instruction AND `#$123` in the **ADD** instruction. The `#base` value becomes `#$FFFFF000 + base` (augmented), which is almost certainly not the intended behavior.

Workaround:

Use a register instead of an immediate for the ALTx instruction's S operand when an **AUGS** is active.

```
1         ' Workaround: Use register instead of immediate in ALTx
2         MOV     temp, #base     ' Load base into register first
3         AUGS    #$FFFFF123      ' Intended for ADD instruction
4         ALTD    index, temp     ' Register operand - unaffected by AUGS
5         ADD     0-0, #$123      ' Only ADD gets the augmented value
```

Summary Table

Bug	Trigger Condition	Consequence	Workaround
ALTx cancels block PTRx delta	ALTx/AUGx between SETQ and RD/WR/WMLONG	PTRx advances by single-long delta instead of block delta	Manually adjust PTRx after transfer
AUGS leaks to ALTx	ALTx with #S between AUGS and target	ALTx receives unintended augmented value	Use register for ALTx S operand

---

These bugs are documented in the official Parallax P2 documentation and affect all P2X8C4M64P Rev B/C silicon.