RISCBoy Documentation

Luke Wren

List of Figures

1 Introduction

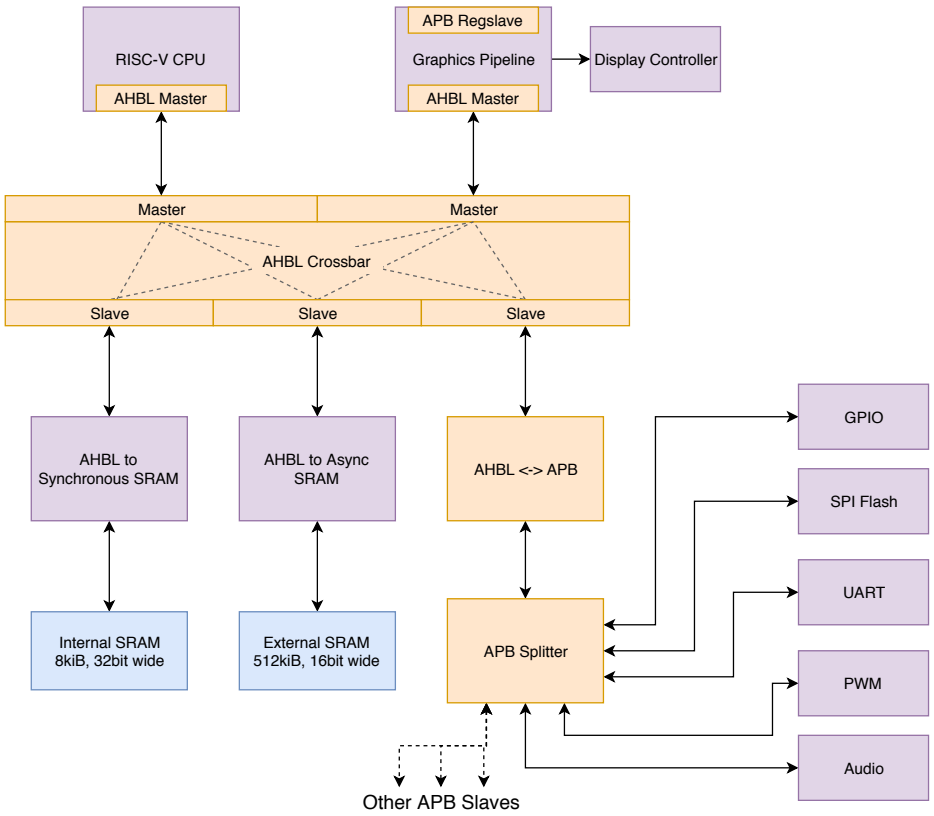
RISCBoy is an open source portable games console, designed from scratch:

- An open source CPU, graphics and bus architecture
- Based on the RISC-V open source instruction set
- FPGA synthesis, place and route with icestorm open source FPGA toolchain
- An open source PCB layout
- PCB designed with KiCAD open source PCB software
- It's open source

  If you say open source one more time I'm gonna nut instantly - Oscar Wilde

1.1 Digital Design

Figure 1: System-level block diagram



The heart of the design is a Lattice iCE40-HX8k FPGA, containing 7680 LUT4s and flipflops. The logic was designed in synthesisable Verilog, with no dependencies on FPGA vendor IP; the contents of this GitHub repository could be taped out onto a chip. This includes:

- RV32IC-compatible 32-bit CPU design
  - RISC-V instruction set
  - 32I: base integer ISA profile
  - M: hardware multiply/divide extension

  - C: compressed instruction extension, for higher code density
  - Vectored interrupts (save/restore of PC, RA only)
  - 5-stage pipeline, similar to textbook RISC
  - Single AHB-Lite master port
- Graphics pipeline
  - Don't expect much, it's about as powerful as a Gameboy Advance
  - Includes some MODE7-like functionality which allows drawing perspective-mapped textured planes, by providing per-scanline affine texture transformation. Think MarioKart
- AMBA 3 AHB-Lite compatible multi-master busfabric
- Peripherals:
  - DMA master
  - External asynchronous SRAM controller (GS74116 or similar)
  - Display controller (ILI9341)
  - GPIO (bitbanging, peripheral muxing)
  - SD card controller
  - UART
  - PWM
  - Basic audio: voices + samples, noise-shaped PWM output

This document attempts to describe some of these, but if you need nitty-gritty detail, the best documentation is the files ending with `.v`.

That a free synthesis tool can cram this into one of the cheapest FPGAs on the market is tremendous. I hope for a situation like software compilers, where free tools such as GCC and LLVM are industry standards.

1.2 Licensing

The Verilog source to this project has no dependencies, and is distributed under the DWTFPL version 3. This is a *very* permissive open-source licence, and its text is included in full at the top of the source files. This license is very similar to the original DWTFPL, which more readers may be familiar with, but has an added indemnification clause.

This license is also known by its more formal name, as the "Do What The Fuck You Want To And Don't Blame Us Public License".

2 CPU Architecture

Hazard5 is a 32-bit processor based on the RISC-V instruction set architecture. It accesses the system through a single AMBA 3 AHB-Lite master port. Those familiar with the textbook 5-stage RISC pipeline will find Hazard5 mostly straightforward, but hopefully will still find some interesting tricks. We will use the following symbols to refer to the 5 stages:

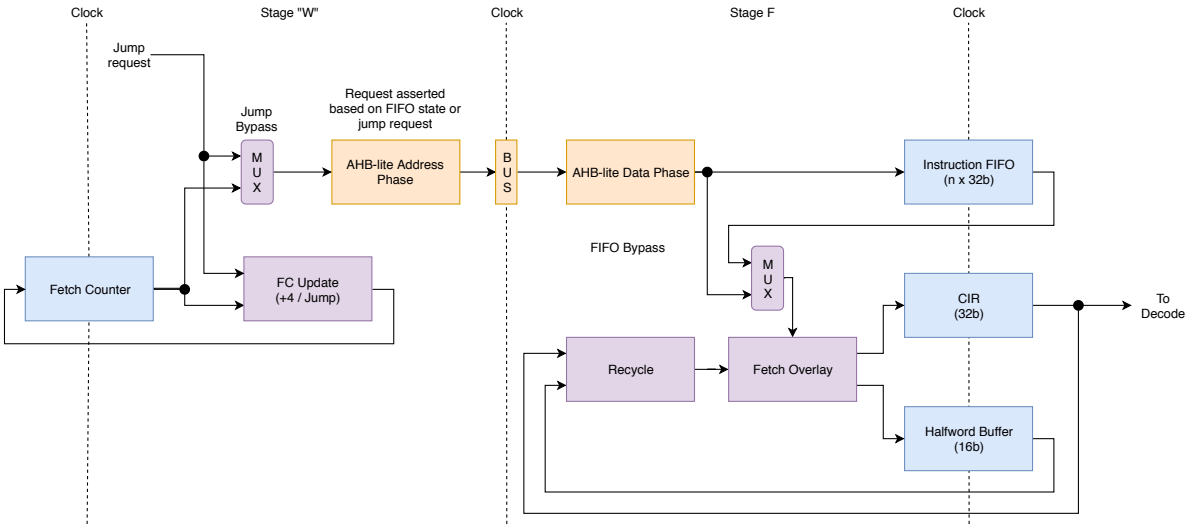
- `F`: fetch
- `D`: decode
- `X`: execute
- `M`: memory access (load/store)
- `W`: register writeback, fetch address generation

Hazard5 supports the RV32IC instruction set, whose encoding is variable-width. The C extension typically reduces instruction bandwidth by ~25%, which helps to maintain performance when sharing bus access between fetch and load/store.

Branches are speculated, but there is currently no dynamic branch predictor. Instead, we use the static prediction scheme described in the RV ISA manual (based on sign of branch offset).

2.1 Frontend

Figure 2: Hazard5 processor frontend, block diagram



The frontend (figure 2) consists of stage `F` and an additional stage which performs the AHB address phase, and can be considered part of `W`. Its purpose is to feed `D` with instructions, whilst meeting the following constraints:

- No combinatorial path from AHB-Lite data phase to address phase (e.g. `hready` → `htrans`)
- AHB-Lite compliant: no unaligned transfers, no deassertion or change of active requests
- Provide up to 32 bits of instruction data per clock in steady state, even if instructions are unaligned
- 0-cycle jump/flush to AHB address phase assertion (with minimal logic on this path)
- No performance penalty for unaligned jump to 16-bit instruction
- Attempt to maintain performance when competing with the load/store unit and AHB-Lite busmaster peers

The main source of complexity is that a RISC-V *C instruction stream is not naturally-aligned, i.e. instruction address modulo instruction size is not always zero. We spend gates here to optimise the common case of sequential execution, and to lessen the effects of fetch starvation due to load-store activity.

To meet these constraints, the frontend performs almost exclusively word accesses, which must be aligned. The only exception is a jump (or similar, e.g. mispredict recovery) to a non-word-aligned address. In this case, a halfword fetch from the target address is performed.

2.1.1 Prefetch Queue

The frontend queues up fresh instruction data which is waiting to be decoded. The pipelined nature of AHB-Lite means that the bus transfers run ahead of `D` by at least two clocks, and the prefetch queue is able to buffer these in-flight transfers if `D` stalls against a later pipe stage. The queue also decouples `D`'s stall logic (which is a function of `hready`) from the address phase request, and finally, the queue helps keep `D` supplied with instructions while the busmaster is busy with load/stores from `X`.

There are three parts to the queue:

- A 32-bit FIFO. The depth is configurable, and can be as little as 1 word.
- A halfword buffer which may store the higher-addressed half of a recently-popped FIFO word
- The upper half of the current instruction register (`CIR`), if the previous instruction was 16-bit

These three sources should service the majority of instruction fetches, and fresh bus data is written only to the FIFO. However, following jumps, flushes, or fetch starvation (either due to load/store activity or bus wait states), bus data can be forwarded directly to `CIR`.

2.1.2 Program Counter

Hazard5 does *not* use the program counter (`PC`) for code fetching, during sequential execution. `PC` is used exclusively for the link value in `JAL(R)`, mispredict recovery, and PC-relative addressing; it is physically located in `D`.

The frontend fetches instruction data from consecutive word-aligned addresses, paced by backpressure from the instruction FIFO; `PC` is not involved. However, as a special case, it *does* need the full jump target address (which becomes the new `PC`), as unaligned jumps require special attention.

2.1.3 Arbitration of Fetch and Load/Store

The single AHB master port asserts transactions from two sources: the frontend, whose address phase is in `W`, and the load/store unit, whose address phase is in `X`. Frontend requests may be linear or non-linear (e.g jumps). The rules are:

1. If a jump or mispredict recovery is asserted by `M`, this wins.
    - Any requests from earlier stages are logically later in program order.
    - If `M` wants to jump then these instructions are being executed in error, so should certainly not be permitted to access the bus.
2. Else if a load/store is asserted by `X`, this wins.
    - Stalling instruction fetch *may* be covered by the prefetch queue, in which case we've lost nothing
    - Stalling a load/store will always increase execution time
    - If instead `X` stalled, and instruction fetch ran ahead, what would we do with the fetched instructions?
3. Otherwise, perform any other access requested by the frontend.
    - Always Be Fetching

The fetch and load/store interfaces are well-decoupled; it would be simple to remove the arbiter and create a 2-master processor configuration. (TODO: add a wrapper that does this!)

2.1.4 Jumps and Branches

Due to the pipelined nature of AHB, we are unable to jump or to take branches in fewer than 2 cycles (without adding sophisticated prediction):

- Cycle 0: AHB data phase for fetch of jump/branch. Next instruction is in address phase concurrently.
- Cycle 1: Jump/branch instruction is now available to `D`
  - (Quickly) use to control the new address phase
  - The immediately following instruction is already in data phase
- Cycle 2: Data phase for jump target instruction
- Cycle 3: Jump target is presented to `D` and decoded.

We knew the jump target on cycle 1, but did not begin decoding the targeted instruction until cycle 3. We also made one wasted code fetch. This is suboptimal, but fetching the jump target *before* decoding the jump is tricky, and there are lower-hanging fruit in terms of performance per LUT.

Jumps physically occur in `W`, directly in front of the fetch address generator. There are two reasons to jump:

- Inspecting the `CIR` in `F/D` pipe register (`JAL`, speculated taken branches)
- Inspecting `X/M` pipe register (`JALR`, branch mispredict recovery)

`JALR` (indirect jump) is taken later because it uses the register file and the ALU to compute its target.

If both of these sources attempt to jump in the same cycle, `X/M` takes priority, since it is executing an older instruction. In both cases, the part of the pipeline in the hazard shadow is invalidated; i.e., `W` → `F`, or `W` → `X`. Invalidation is performed by clobbering the pipeline control signals in such a way that these instructions will have no side effects.

The branch prediction scheme is static: take backward branches, and do not take forward branches. The cycle costs are as follows:

Jump Type	Cycles (Execution + Penalty)
Direct jump	2
Predicted, non-taken branch	1
Predicted, taken branch (same as jump)	2
Indirect jump	4
Branch mispredict	4

Upon jumping, we need some mechanism to invalidate parts of the pipeline: this is described in section 2.2.2.

2.1.5 CIR Locking

There is a landmine in the following tableau:

- `M` contains a load (in data phase), and the bus is stalled
- `X` contains an instruction dependent on the load result; say an `AND`
- `D` contains a branch which is mispredicted taken

For example, say we are polling a status bit in an IO register, looping until some bit is high. The instruction in `X` must stall for at least two cycles due to the bus stall and load-use hazard. Due to a frontend design constraint from 2.1 – "No combinatorial path from AHB-Lite data phase to address phase (e.g. `hready` → `htrans`)" – we must not use `X`'s stall signal to gate the jump request, as this would create such a path. However, *not* gating the jump request is fatal, as new fetches will clobber `CIR` before the branch instruction can proceed into `X`, so the mispredict will never recover. `JAL` has the same problem: it will jump, but not produce a link address.

Hazard5 resolves this with CIR locking. D signals to F that CIR and its validity count must not change on the next clock edge, and uses this same signal to inhibit repeated assertion of its own jump request. In the fetch path, this is achieved by steering the controls on the existing shift/overlay logic in the frontend, and requires no additional muxing.

Whilst the CIR is locked, the frontend is still free to act on the jump request, and fetch ahead along the new code path (more useful for JAL); this data is buffered in the FIFO. Once the roadblock ahead of D clears, the branch instruction proceeds down the pipeline, and the lock is released simultaneously.

Note also that a bus stall does not cause the frontend to block jump requests, as this too would create a combinatorial path, since requests are forwarded straight to the bus (due to another constraint from 2.1). The frontend's response to bus stall on jump request is simply to not increment the target before storing to FC, so that the target address continues to be asserted on subsequent cycles. Registered feedback blocks *subsequent* jump requests until the first request completes its address phase.

2.1.6 Instruction Barrier (FENCE.I)

If the program stores to instruction addresses about to be executed from, which potentially exist in the prefetch queue, a stale instruction will be executed. FENCE.I cannot be decoded as a nop, and requires special handling.

Hazard5 decodes FENCE.I as "jump to PC + 4". The jump invalidates the prefetch queue, and the following instruction will be re-fetched from memory.

Timing analysis shows that calculating a jump target, between CIR and the address bus, is generally on the critical path. To avoid more logic on this path, FENCE.I is instead implemented by spoofing a branch-taken mispredict, which has the same effect. This adds two cycles to the execution time, but performance of this instruction is decidedly noncritical.

2.2 Backend

The backend is the hardware which manipulates the processor's architectural state according to the stream of instruction data from the frontend. Architectural state refers to the state you will read about in the ISA manual, and includes the register file and program counter; the backend also contains some non-architectural state, e.g. holding intermediate results in between pipeline stages.

The backend consists of stages D, X, M. Its overall structure is depicted by figure 3. This is a pretty large figure – too large to print really – so you can view it separately at https://github.com/Wren6991/RISCBoy/raw/master/doc/diagrams/cpu_backend.pdf

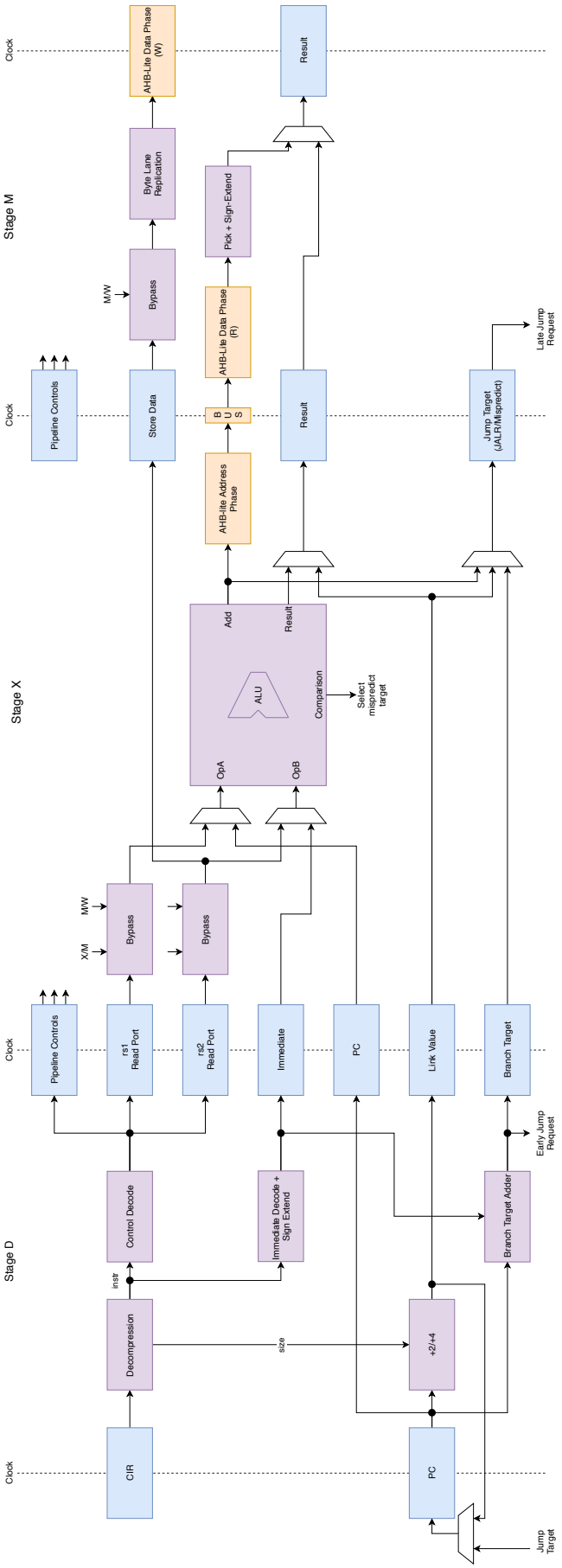


Figure 3: Hazard5 processor backend, block diagram

2.2.1 Operand Bypass

Hazard5 possesses an operand bypass (forwarding) network. Register writes must always be visible to later instructions, even before an earlier instruction reaches register writeback. One solution is to detect that one instruction depends on the result of an earlier instruction (a read-after-write, or RAW hazard) and stall the later instruction until the earlier one completes. This is safe, but incurs a hefty performance penalty; a more elegant solution is to pluck the result of the earlier instruction straight from the pipeline, before it finishes executing (but after the result is valid!).

Locations of bypasses are shown in figure 3. The following bypasses are available:

- `X/M` → `X`
- `M/W` → `X`
- `M/W` → `M`
- `W/X` → `X`

The last is in lieu of a write-to-read bypass in the register file; some tools (not Yosys!) struggle with inferring memories from transparent memory models, so it's better to be explicit.

To control the bypassing, some of the register specifiers from `CIR` are passed down the pipeline alongside the data. `rs1, rs2, rd` (operand sources and destination) are passed down as far as `X`. `rs2, rd` make it to `M`, and only `rd` makes it to `W`.

The upshot is:

- Back-to-back ALU operations execute at 1 CPI
- Loads insert 1 stall cycle if immediately required by the ALU. 1 CPI otherwise.
- Stores execute at 1 CPI (bus stall notwithstanding)
- In a load + store pair, the load takes only one cycle, since the `M` stage has self-forwarding

Various interesting strategies can alleviate load-use penalty in in-order pipelines, such as adding a second, "late" ALU in the `M` stage. In our case we judge this to not be worth the LUTs.

Another interesting strategy is to add muxing at the register file write port, to select which stage to retire from. This has the dual benefit of firstly simplifying the operand bypass, and secondly reducing dynamic power, since there is no need to pass data through unused stages (and they can be clock gated). Hazard5 does not use this strategy either – maybe the next project.

2.2.2 Pipeline Stalling and Flushing

Our terminology: stalling means a pipeline stage does not advance its state until some blocking condition has cleared. The instruction residing in this stage will not progress to the next stage, and the previous stage will not write *its* instruction into this stage. Flushing is when in-flight instructions in some stages are replaced with NOPs, and their results are discarded.

The frontend is decoupled from other stages' stall logic via the prefetch queue. This is important: `hready` is an input to that stall logic, and the the frontend's address-phase request must not be a function of `hready`.

The frontend may not be able to immediately accept a jump request, which may cause other pipe stages to stall if it is low. One cause is the frontend holding an existing address-phase request stable until the cycle *after* `hready`, which is required for AHB-Lite compliance.

For the backend, the stall logic is more intricate, as signals such as `hready` are used in-cycle to determine whether an instruction progresses to the next pipeline stage:

- `D`:
  - `CIR` does not contain a valid instruction (either no data, or half of a 32-bit instruction)

  - `D` asserts jump, but frontend rejects the jump request
  - `X` is stalled
- `X`:
  - `hready` low and `X` address-phase request asserted
  - RAW hazard on `M` (load-use)
  - `M` is stalled
- `M`:
  - `hready` low and data-phase active
  - `M` asserts jump, but frontend rejects the jump request
- `W`: does not stall

If a given stage is stalled, but the following stage is not, it must insert a bubble. Bubbles are created by zeroing out control fields, such as `rd`, so that the instruction cannot affect system or processor state.

There are two cases where we must flush:

- Branch/jump taken from `D`; frontend invalidates prefetched data
- Jump/mispredict taken from `M`; must flush frontend, `D`, `X`

And the flushing mechanisms for each stage are as follows:

- `D`: destination register `rd` cleared, which makes result invisible to register file and operand bypass. `memop`, `branchcond` pipe flags are cleared.
- `X`: same as `D` (except for `branchcond`, which does not pass on to `M` anyway.

Flushing and bubble insertion are very similar in mechanism.

2.3 Unaligned Memory Accesses

Alignment is the constraint that the address of a memory access be equal to zero, modulo some size. Where no size is specified, we refer to *natural* alignment, i.e. modulo the size of this particular memory operation. RISC-V requires that memory is byte-addressable.

The frontend goes to some length (section 2.1) to maintain high throughput. RV-C instruction streams are *always* unaligned, and *every* instruction must be fetched before it is executed, so Amdahl says it's worth it. On the other hand load/stores are less than 100% of all instructions, and the vast majority are unaligned; consequently, Hazard5 does not have hardware support for unaligned load/stores. These are trapped (TODO) and handled in software if and when they occur.

2.4 Control and Status Registers (CSRs)

Hazard5 possesses the standard `Zicsr` extension, which provides atomic access to the CSRs. Only M-mode CSRs are implemented. Access to an unimplemented CSR (e.g. a U-mode or S-mode CSR) causes an illegal instruction exception. The implementation is minimally legal: WARL fields are widely exploited, to reduce logic and state overhead. Module parameters can reduce CSR support, or remove it entirely, as per relative importance of compliance versus area in your application. Note that some features, such as interrupts and exceptions, require CSR support.

When the counter CSRs are present, `mtime` and `mcycle` are aliased to the same counter. 64-bit counters are prohibitively large in a compact 32-bit processor. The `mcountinhibit` (WARL) register is tied to zero: `mtime` must run freely, hence the `mcycle` alias of this counter cannot be halted either. `minstret` is fully implemented, but its `mcountinhibit` bit is also tied low. The width of the counters can be reduced from 64 bits to save logic (*noncompliant*), although the full 64 CSR bits remain writeable, which leaves some opportunity for software emulation.

TODO: full table of CSRs we implement, and to what extent they are implemented/WARL'd

2.5 Interrupts and Exceptions

Figure 4: Hazard5 trap vector table layout

Trap Table Offset	`mcause`	Trap Description
0x00	0x00000000	Misaligned instruction access
0x04	0x00000001	Instruction access fault
0x08	0x00000002	Illegal instruction
0x0c	0x00000003	Breakpoint
0x10	0x00000004	Misaligned load address
0x14	0x00000005	Load access fault
0x18	0x00000006	Misaligned store address
0x1c	0x00000007	Store access fault
0x20	0x00000008	*Reserved*
0x24	0x00000009	*Reserved*
0x28	0x0000000a	*Reserved*
0x2c	0x0000000b	M-mode ECALL
0x30	0x0000000c	*Reserved*
...	...	...
0x3c	0x0000000f	*Reserved*
0x40	0x80000000	*Reserved*
0x44	0x80000001	*Reserved*
0x48	0x80000002	*Reserved*
0x4c	0x80000003	Machine software interrupt
0x50	0x80000004	*Reserved*
0x54	0x80000005	*Reserved*
0x58	0x80000006	*Reserved*
0x5c	0x80000007	Machine timer interrupt
0x60	0x80000008	*Reserved*
...	...	...
0x7c	0x8000000f	*Reserved*
0x80	0x80000010	External IRQ 0
...	...	...
0xbc	0x8000001f	External IRQ 15

Hazard5 has simple exception and interrupt support, compliant with the RISC-V privileged ISA spec (M-mode only). Here we follow the terminology set out in the RISC-V user-level spec: an *exception* is an anomalous condition caused by executing an instruction, which requires control transfer; an *interrupt* request is some external, asynchronous event, which requires control transfer; a *trap* is the transfer of control to a handler, used to service either an interrupt or an exception. Hazard5 implements interrupts and exceptions as follows:

- Trap entry causes a jump to a location in the trap vector table (addressed by `mtvec`), simultaneously stashing the PC in `mepc`
  - Other CSR state is also modified, e.g. `mcause`, `mstatus`
  - The physical mechanism for this jump is the same as a branch mispredict
- Trap exit, via the `mret` instruction, causes a jump to `mepc`
  - `mret` also modifies some diagnostic CSR state, e.g. `mstatus`, as per the privileged ISA specification
- Only vectored mode is available. I.e., the LSBs of `mtvec` are tied to 0x1.
- When written, `mtvec` is rounded down to a 4kB boundary. This saves an adder when generating the trap vector address.
- Besides PC, the hardware saves/restores no architectural state on entry/exit. Software is responsible for stacking/unstacking the GPRs.

External IRQs can be observed and masked in bits 31:16 of `mip` and `mie`. This limits the number of IRQs to 16, but this limit could be increased by additional nonstandard CSRs to observe and mask more external interrupts.

The layout of the trap table is in figure 4. Where a standard exception or interrupt is unimplemented (S/U mode) or irrelevant (page faults), it is marked as *reserved*. In particular, the standard machine external interrupt vector (`mcause` = 0x8000000b) is not used, and external interrupts are instead separately routed to `mcause` = 0x80000010 and above. This layout has been arranged to produce an obvious mapping between `mcause` and trap vector address, which unfortunately bloats the table with reserved vectors.

To save space, it is reasonable for initial bootcode to include only the first 12 entries in the table (the exceptions), and run with interrupts disabled until it has populated a more complete table elsewhere in memory, which it can then point to by modifying `mtvec`.

Trap handlers do not preempt one another. After a trap returns, the highest-priority active trap request is selected, and its handler entered. Normal execution continues while there are no active trap requests. Exceptions take priority over IRQs, and lower-numbered IRQs are higher priority. An illegal instruction exception encountered while handling an exception causes a lockup of the core (details TBD). TODO: table of the exception/interrupt space

IRQ inputs are level-sensitive. It is recommended to clear an interrupt at its source upon entering its handler. If the interrupt reasserts before the handler `mret`s, the handler still returns, and the interrupt competes with other trap requests for reentry. Note that, if many register stages are present on the IRQ signal, there is a potential race between clearing the IRQ and executing the `mret`, causing spurious reentry; in practice, restoration of caller-saves should provide ample time for IRQ deassertion to be observed, before the `mret` is reached.

2.6 Plugin Interface

Plugins are pieces of hardware attached to the Hazard5 pipeline to extend its functionality, and instruction set. For example, the M extension (multiply and divide) is implemented using a plugin.

Plugins interact with three pipe stages:

- `D`
  - Plugins see the current instruction, so that they can decode it
  - Plugins notify `D` that an instruction it does not recognise is, in fact, valid
- `X`
  - Plugins can see the `rs1` and `rs2` operand bypasses
  - Additional control signals provide functionality such as killing an in-progress operation so that e.g. an interrupt can run
- `M`
  - All plugins retire their results into `M`.
  - This adds a 1-cycle RAW stall if the immediately-following instruction is dependent, but on current iCE40 implementations, `X` is already near-critical-path and very mux heavy.
  - Plugins can stall `M` if they need more cycles to produce a result.

There is no hard limit to the number of plugins that can be added, but the additional muxing and fanout is not free, which constrains practical implementations.

2.6.1 M-small Plugin

The M-small plugin implements the RISC-V M standard extension (multiply, divide and modulo). To save area, the plugin employs a combined multiply/divide unit which performs all required calculations, at 1 bit per clock. If timing permits, the datapath can be unrolled to perform multiple iterations per clock. e.g. radix-4 divide/multiply.

2.6.2 M-fast Plugin

The Mfast plugin also implements the M extension, but uses a fast Wallace tree to perform multiplication, allowing a throughput of 1 32-bit multiply per clock.

Other details TBD

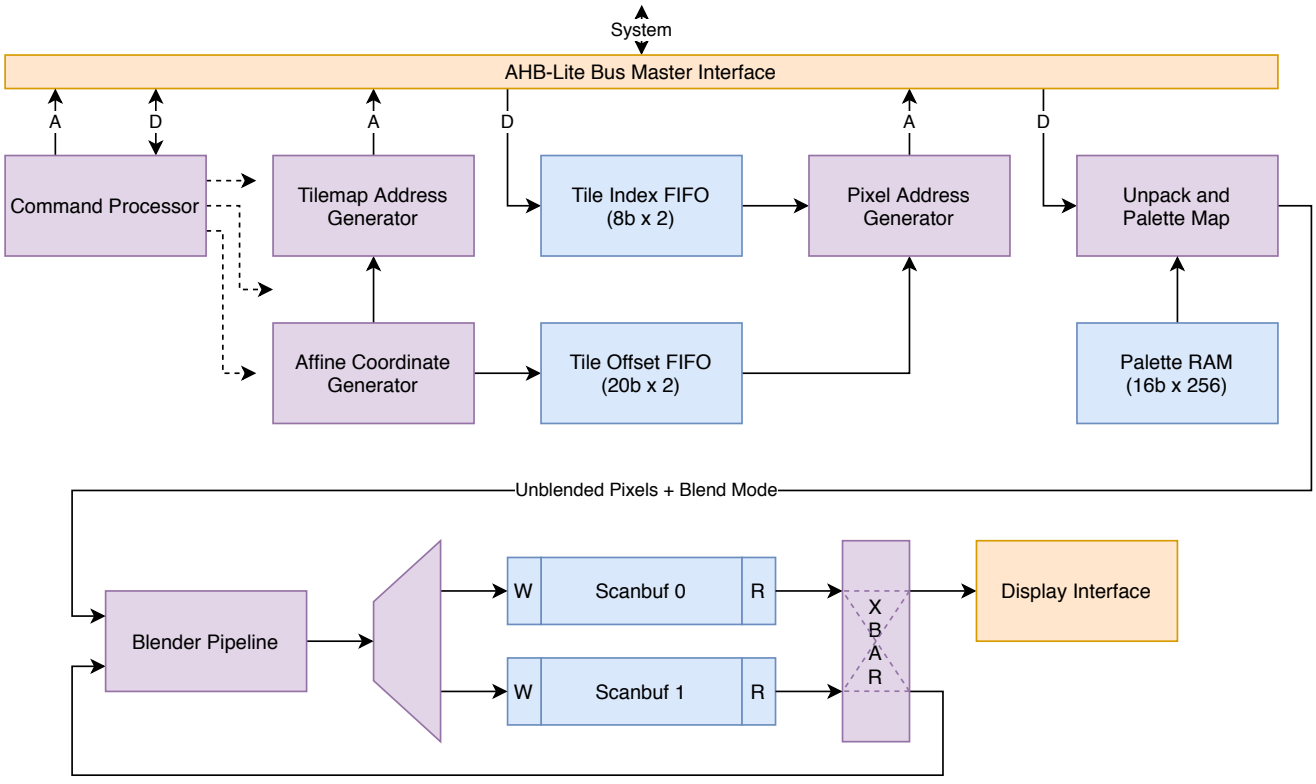
3 Pixel Processing Unit (PPU)

Figure 5 shows the high-level structure of the PPU. This is the second-generation architecture – the previous version had a number of sprite and tile engines operating in parallel, with shared bus access. This was a mistake. Many of these units were idle much of the time, and they were stripped down to the minimum to meet strict area requirements; this in turn led to poor bus utilisation and a lack of flexibility. The second problem was partially solved by the Poker, but at the expense of high software complexity for common use cases (e.g. many sprites on screen). The high level goals of PPU2 are, in order of precedence:

1. Always Be Fetching
2. Avoid idle or duplicated base functional units (e.g. pixel unpacker) so that more expensive functions can be provided (e.g. random access to pixels)
3. Be highly programmable, and allow features to be combined in interesting ways
4. Serve the simple use cases (layered, tiled backgrounds with sprites) with minimum program complexity
5. Mario Kart: Super Circuit

Fundamentally we are fetching pixels from memory (over a 16 bit bus on RISCBoy), and putting them on the screen in interesting orders. If we saturate the bus, and do not waste what we fetch, our performance is as good as it possibly can be.

Figure 5: Pixel processing unit, block-level diagram



At any point in time the PPU is either rendering to one of the internal scanline buffers, or waiting for a buffer to be freed and passed back by the display interface hardware. The exact sequence of operations the PPU performs while rendering is defined by a program in memory, executed by the command processor. The instruction set is small, and specialised to the task at hand – for example the `SYNC` instruction marks the current scanline buffer as ready for display, and waits for a buffer be freed by the display hardware, and the `FILL` instruction writes a constant-colour span to some x range of pixels.

Common memory access patterns (for nonpaletted pixels):

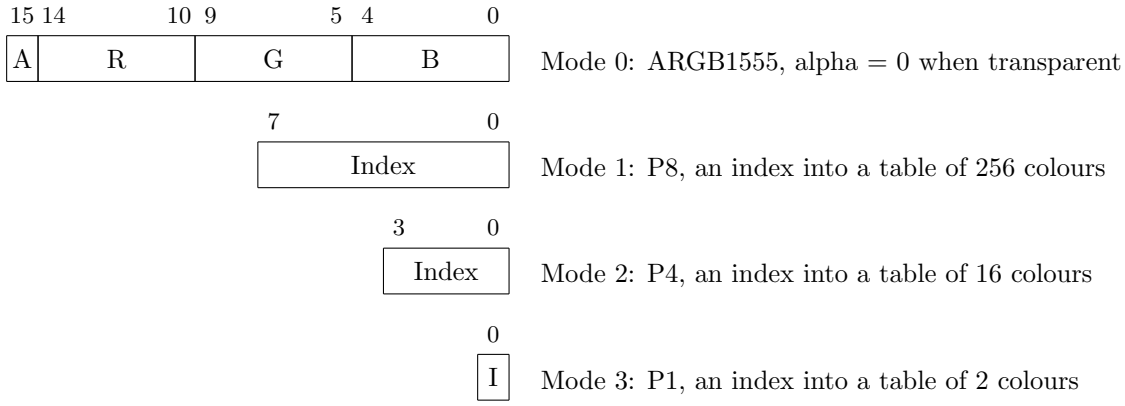
- Sprite blitting: fetch a pixel from memory every cycle
- Tiling: fetch a tile index after every n pixels, then n pixel fetches
- Affine-mapped tiling: fetch tile indices and pixels on alternate cycles

The display controller attaches to a dedicated interface on the PPU. Through a simple handshake, the display controller can acquire and release dirty scanbuffers. The display interface routes read addresses from the display controller to the correct scanline buffer, and routes pixel data back to the controller. Currently two display controllers are available to connect to this interface: one for serial LCDs such as ILI9341, and another for direct DVI output. The PPU display interface is synchronous, and any clock crossing (to SCK domain for serial LCDs, or pixel clock domain for DVI) is handled inside the display controller.

3.1 Pixel Formats

Internally, the PPU uses a single native pixel format, namely ARGB 1555 (see figure 6), but it can stream pixels from memory in a variety of formats, and convert internally. PPU memory accesses are **always little-endian**: for performance reasons the PPU performs the widest possible fetch, yielding multiple pixels each, which are numbered least-significant-first.

Figure 6: PPU pixel formats



For pixels smaller than one byte, the pixel order continues to be defined in a little-endian fashion, i.e. the least-significant pixel will be the first to be displayed. The PPU also requires image base addresses to be word-aligned, which implies all pixels are naturally aligned.

3.2 Palettes

The PPU contains a single hardware palette memory (PRAM), which is large enough to store 256 colours in ARGB1555 format. Each pixel in a paletted image (see figure 6) consists of an index into PRAM. The PPU looks these indices up before passing pixels to the screen; wide colour range is maintained at reduced bits per pixel.

Since PRAM contents is in ARGB1555 format, there is no special convention for indicating whether a paletted pixel is transparent: the hardware always performs the palette lookup, and the resulting colour may or may not have its Alpha bit set.

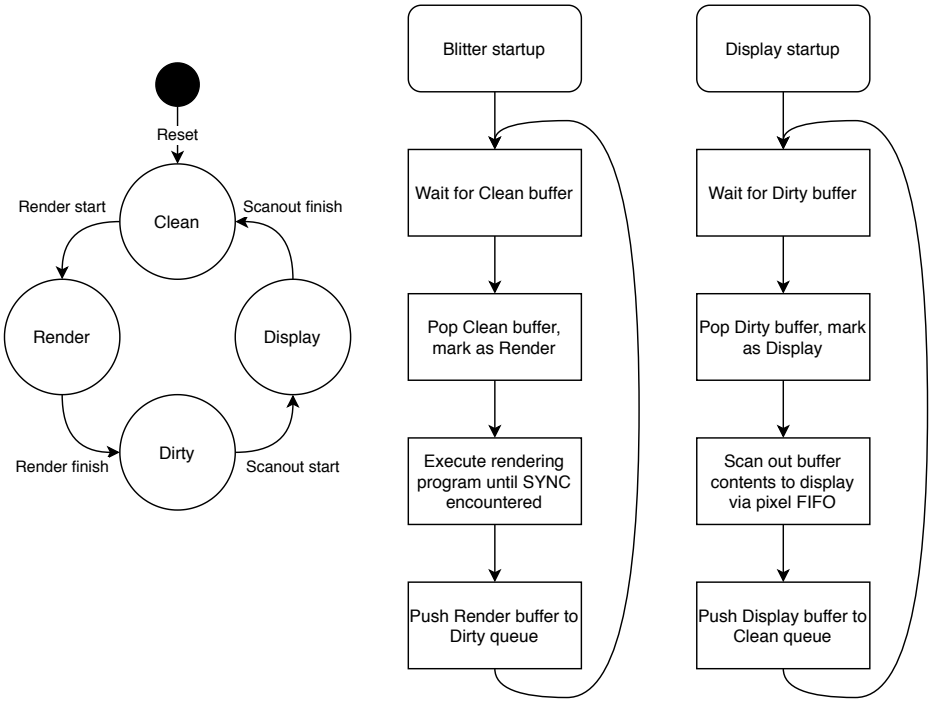
Although there is only a single hardware palette, 256 colours in size, `TILE` and `BLIT` commands can supply an offset, which is added to each paletted pixel before palette lookup. PRAM may be initialised with e.g. multiple 16-colour tables at different (potentially overlapping) locations, giving effectively independent palettes.

PRAM can be written to (but not read) through the PPU's configuration interface.

3.3 Scanline Buffers

The scanline buffers are implemented as 1-read 1-write memories, which allows the blending pipeline to easily achieve a 1 pixel per clock throughput, keeping the bottleneck on the system bus. This could also be implemented as a double-width 1RW memory, but not a compelling tradeoff on FPGA.

Figure 7: PPU scanline buffer states



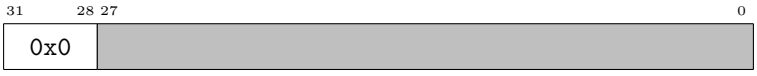
On RISCBoy these are 15b×512 memories, each composed of a pair of iCE40 4 kb block RAMs. This is sufficient to store one QVGA RGB555 scanline in each buffer. The blitter and display pass each other buffers through a pair of queues, one containing clean buffers for the blitter, and the other containing dirty buffers for the display. After reset, both buffers are in the clean queue.

3.4 Instruction Set

Each instruction consists of one or more words in memory; the blitter processes each instruction in turn, progressing in a linear fashion unless a `POPJ` is encountered, or the PPU is manually vectored to a new program by system software. This section describes the encoding of blitter instructions, and their operation. Some fields are marked as *reserved*, denoted by a grey colour fill in the bit diagrams. Reserved fields are not used by the current hardware, and should be written all-zeroes by software.

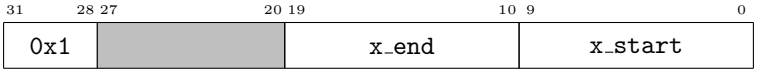
For simple purposes (e.g. tiled backgrounds with some sprites on top), the blitter can execute the same program on each scanline: the coordinate systems of `BLIT/TILE` commands take the current raster y coordinate into account (see section 3.5). However, the instruction set is quite flexible, and the blitter's features can be combined in interesting ways, with fancy-looking results.

SYNC



Present the current scanline buffer to the display, then stall until a clean buffer becomes available. If `CSR_HALT_HSYNC` is set, or `CSR_HALT_VSYNC` is set and this is the last line of the frame, halt the PPU.

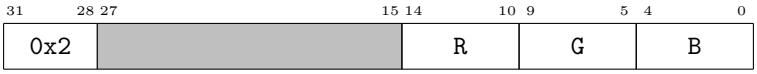
CLIP



Set the active region for rendering. Pixels at x coordinates less than `x_start` or greater than `x_end` will be unaffected by subsequent `FILL`, `[A]TILE` or `[A]BLIT` operations. This region remains in effect until another `CLIP` is executed.

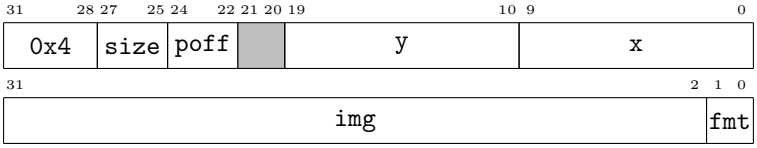
Often it is sufficient to perform an initial `CLIP` to the screen width, and never change from this value, but `CLIP` has numerous applications when combined with pixel-filling operations: for example using `CLIP` + `ABLIT` pairs to render a sequence of short affine-textured spans.

FILL



Fill the entire clipped region with a single colour. Note that scanline buffers retain their old contents when passed back to the blitter by the display hardware; if you want a solid background colour, you must `FILL` it into each scanline, or include it in your background tileset.

BLIT

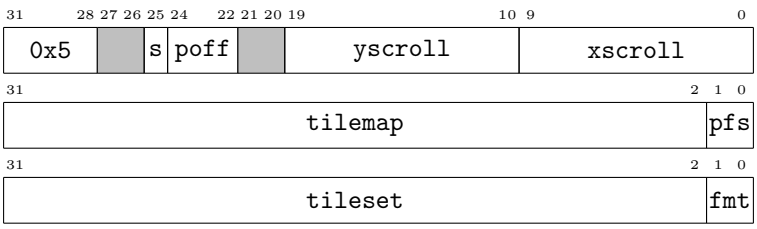


Block image transfer: paste a square image (e.g. a sprite) over the current scanline. If this image does not intersect the current scanline due to its position and size, or if it is fully outside of the clipped region on this scanline, the `BLIT` command has no effect, and completes immediately. Generally `BLIT` is called on each scanline with the same arguments, to build up the full image. For example, each game sprite may correspond to a `BLIT` command which runs on every scanline.

`BLIT` also supports efficient framebuffer graphics: in this case each scanline will have a different `BLIT` command with a `y` equal to that scanline's position, and a `img` pointer that is one scanline further advanced into a software framebuffer. This allows a framebuffer to be packed in memory as a flat width × height array of pixels.

- `img` is a pointer to the source image, which is assumed to be word-aligned (hence the missing LSBs).
- `fmt` is the pixel format, as described in figure 6. The values are: mode 0 ARGB1555, mode 1 P8, mode 2 P4, mode 3 P1.
- `size` defines the size of the source image: width = height = $2^{\texttt{size}+3}$, so square images from 8 to 1024 pixels are supported.
- `poff` is the palette offset. This is left-shifted by 5 and added to each paletted pixel, with wrap on overflow, before the pixel is looked up in the palette RAM.

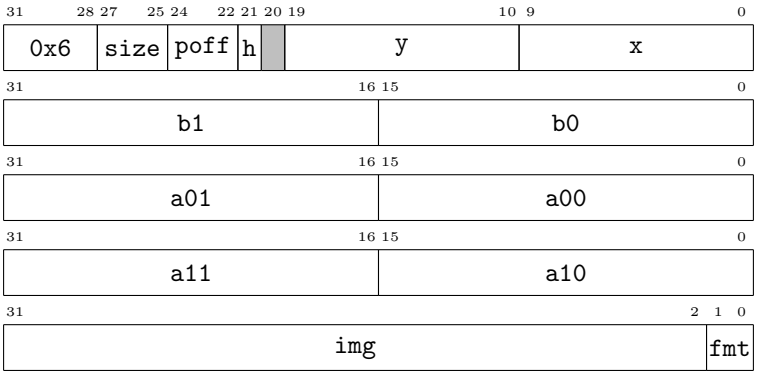
TILE



Render a tiled background span (see section 3.6).

- `xscroll` and `yscroll` define the horizontal and vertical scroll of the tiled region. (TBD this may need to be redefined slightly for new PPU).
- `s` defines the size of the source image: width = height = 2^{s+3}, so square images of 8×8 or 16×16 are supported.
- `poff` is the palette offset. This is left-shifted by 5 and added to each paletted pixel, with wrap on overflow, before the pixel is looked up in the palette RAM.
- `tileset` is a pointer to an array of 8×8 or 16×16 px images
- `fmt` is the pixel format of the tileset images, as described in figure 6. The values are: mode 0 ARGB1555, mode 1 P8, mode 2 P4, mode 3 P1.
- `tilemap` is a pointer to a 2D array of 8 bit tile indices. Each index identifies which image from `tileset` occupies that 8×8 or 16×16 px square.
- `pfs` is the playfield size in pixels, width = height = 2^{size+7}, so 128 to 1024 px. See section 3.6.

ABLIT



Like `BLIT`, except screen coordinates are transformed before each pixel lookup:

$$\mathbf{u} = \mathbf{A}(\mathbf{s} - \mathbf{s}_0) + \mathbf{b}$$

Where

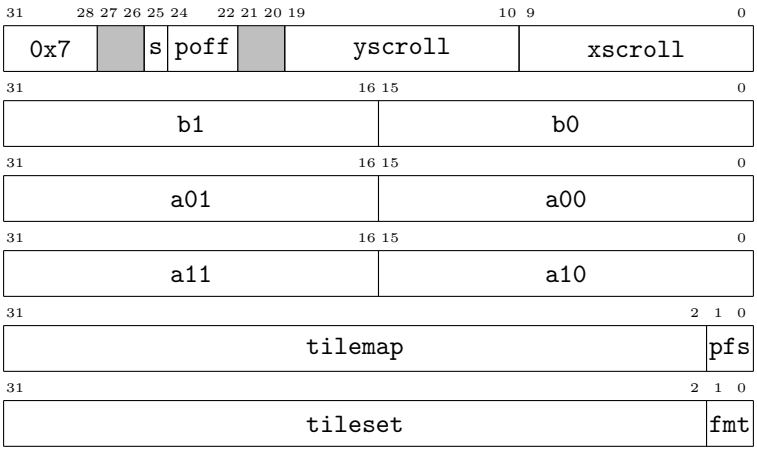
$$\mathbf{u} = \begin{bmatrix} u & v \end{bmatrix}^T \quad \text{(texture coordinates)}$$
$$\mathbf{s} = \begin{bmatrix} s_x & s_y \end{bmatrix}^T \quad \text{(screen coordinates)}$$
$$\mathbf{s}_0 = \begin{bmatrix} x & y \end{bmatrix}^T \quad \text{(blit target position } \texttt{x}, \texttt{y}\text{)}$$
$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad \text{(scale/rotate/shear matrix)}$$
$$\mathbf{b} = \begin{bmatrix} b_0 & b_1 \end{bmatrix}^T \quad \text{(translation vector)}$$

This affine transform is described in detail in section 3.5. It allows a range of geometric manipulations of the source image. `ABLIT` renders to the same square region as `BLIT`, defined by the `size`, `x` and `y` parameters, as well as the current `CLIP` region. It is the *texture lookup* for each rendered pixel that differs.

The `a` components are signed 8.8 fixed point numbers, and `b` components are unsigned 10.6 fixed point.

- `h`: half-size flag: indicate the texture is only half the size of the rasterized region in each axis. This allows e.g. a 32×32 px texture to be rotated, without scaling, by 45° and blitted to a 64×64 px region, so that its corners are not clipped. The effect of the half-size flag when `size` = 0 is undefined.
- `img` is a pointer to the source image, which is assumed to be word-aligned (hence the missing LSBs).
- `fmt` is the pixel format, as described in figure 6. The values are: mode 0 ARGB1555, mode 1 P8, mode 2 P4, mode 3 P1.
- `size` defines the size of both the source image and the blit target area: width = height = $2^{\texttt{size}+3}$, so square images from 8 to 1024 pixels are supported.
- `poff` is the palette offset. This is left-shifted by 5 and added to each paletted pixel, with wrap on overflow, before the pixel is looked up in the palette RAM.

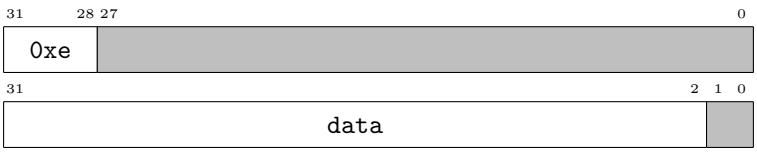
ATILE



Like `TILE` but affine transform applied before each tile and pixel lookup: $\mathbf{u} = \mathbf{A}(\mathbf{s} - \mathbf{s}_0) + \mathbf{b}$. As with `TILE`, the entire `CLIP` region is rendered to. $\mathbf{s}_0$ is the `xscroll`, `yscroll` offset, the same as `TILE`. All in all, `TILE` can be thought of as a simple case of `ATILE`, where $\mathbf{A}$ is the identity matrix and $\mathbf{b}$ is the zero vector.

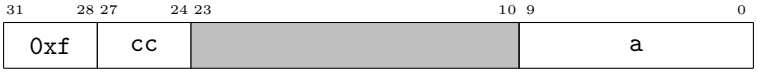
- `s` defines the size of the source image: width = height = $2^{\texttt{s}+3}$, so square images of 8×8 or 16×16 are supported.
- `poff` is the palette offset. This is left-shifted by 5 and added to each paletted pixel, with wrap on overflow, before the pixel is looked up in the palette RAM.
- `tileset` is a pointer to an array of 8×8 or 16×16 px images
- `fmt` is the pixel format of the tileset images, as described in figure 6. The values are: mode 0 ARGB1555, mode 1 P8, mode 2 P4, mode 3 P1.
- `tilemap` is a pointer to a 2D array of 8 bit tile indices. Each index identifies which image from `tileset` occupies that 8×8 or 16×16 px square.
- `pfs` is the playfield size in pixels, width = height = $2^{\texttt{size}+7}$, so 128 to 1024 px. See section 3.6.

PUSH



Push literal `data` onto the stack. The stack is popped only by a `POPJ` instruction, is 8 words deep, and wraps on empty/full.

POPJ



Always pop an address from the stack. If `cc` is true, branch to this address. Values for `cc`:

- 0: always
- 1: YLT, current raster beam $y <$ `a`
- 2: YGE, current raster beam $y \geq$ `a`

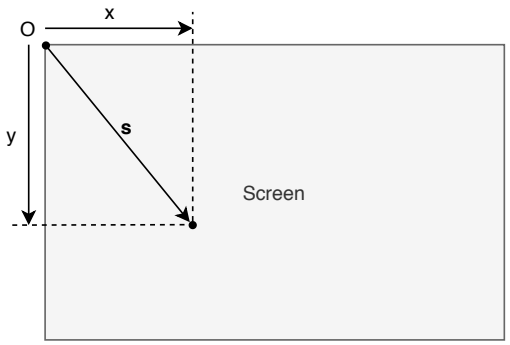
A regular jump is formed with `PUSH addr; POPJ`, and a subroutine call is formed with `PUSH ret; PUSH target; POPJ; label ret:` followed by a `POPJ` at the end of the target routine.

Subroutines are useful for when a part of your command list is repeated every scanline (e.g. a list of sprites, represented by `BLIT` commands) and a part is not (e.g. `FILL`ing a different background colour every scanline to produce a gradient). Such a program can be structured as a larger per-frame command list which repeatedly calls into a per-scanline subroutine.

3.5 Coordinate Systems

The origin of the screen is at the top left. x coordinates increase to the right. y coordinates increase going downward.

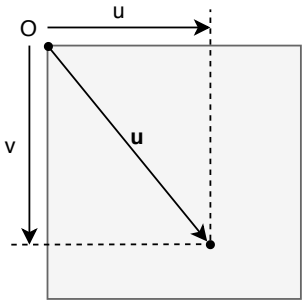
Figure 8: PPU screen coordinate system



When using a vector to refer to a screen-space coordinate, we will typically use the symbol **s**.

Texture Coordinates

Figure 9: PPU texture coordinate system



Textures (e.g. sprite images) have completely independent coordinates. We will conventionally refer to pixels in texture space with the vector $\mathbf{u} = \begin{bmatrix} u & v \end{bmatrix}^T$. Textures are stored in memory as a flat array of pixels, going first in u order and then stepping down to the next row.

When drawing a sprite, the PPU does the following:

- Iterate over each pixel in some horizontal span of screen space
- For each pixel, transform the screen-space coordinate $\mathbf{s}$ into a texture-space coordinate $\mathbf{u}$
- Look up the texture pixel at $\mathbf{u}$ and draw it on the screen at $\mathbf{s}$

This transformation may be simple, as in the case of the `BLIT` command:

$$\mathbf{u} = \mathbf{s} - \mathbf{s}_0$$

Where $\mathbf{s}_0$ is the position the texture is being blitted to (`x`, `y` in the `BLIT` instruction), $\mathbf{s}$ is the position of next pixel to be rendered on the screen, and $\mathbf{u}$ is the position in texture space which determines the colour of the screen pixel. The transformation may be more complex, as in the case of the `ABLIT` command:

$$\mathbf{u} = \mathbf{A}(\mathbf{s} - \mathbf{s}_0) + \mathbf{b}$$

Where the matrix $\mathbf{A}$ and vector $\mathbf{b}$ are constants supplied by the `ABLIT` instruction, permitting any combination of translation, rotation, scale and shear. Note the direction of the definition: this is a mapping *from* screen space *to* texture space. If you find it more intuitive to think about mapping the texture onto the screen, you must note that this is the *inverse* of the above mapping, and you have some linear algebra to do.

Internally, the PPU represents $\mathbf{u}$-space coordinates as pairs of unsigned 10.8 fixed point numbers (a range of 0 to $1023 + \frac{255}{256}$); the largest supported texture size is therefore 1024×1024 px.

As far as coordinates are concerned, there is no difference between e.g. a 512×512 px texture and a 512×512 px tiled background. The difference is that a texture's pixels are defined by a single large image (512×512 px in this example), whereas the tiled area is defined by a number of small e.g. 16×16 px images called the *tileset*, and a 2D array of tile indices called the *tilemap* which defines which 16×16 px image should be at which 16×16 px region of $\mathbf{u}$-space.

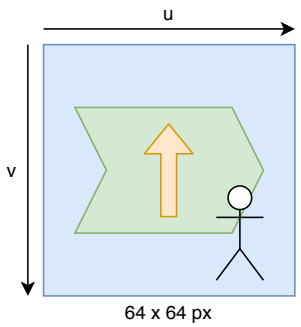
Affine Transforms

We gave the following definition for the transformation from screen space to texture space used by the `ABLIT` and `ATILE` commands:

$$\mathbf{u} = \mathbf{A}(\mathbf{s} - \mathbf{s}_0) + \mathbf{b}$$

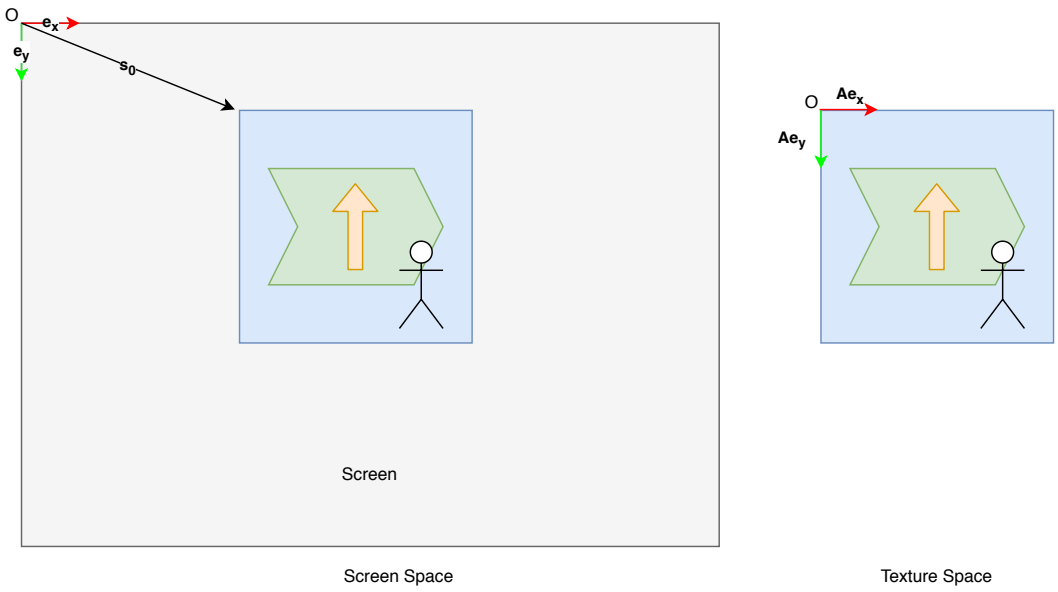
All of our examples will use the following texture which, for argument's sake, is 64×64 px in size:

Figure 10: PPU texture example



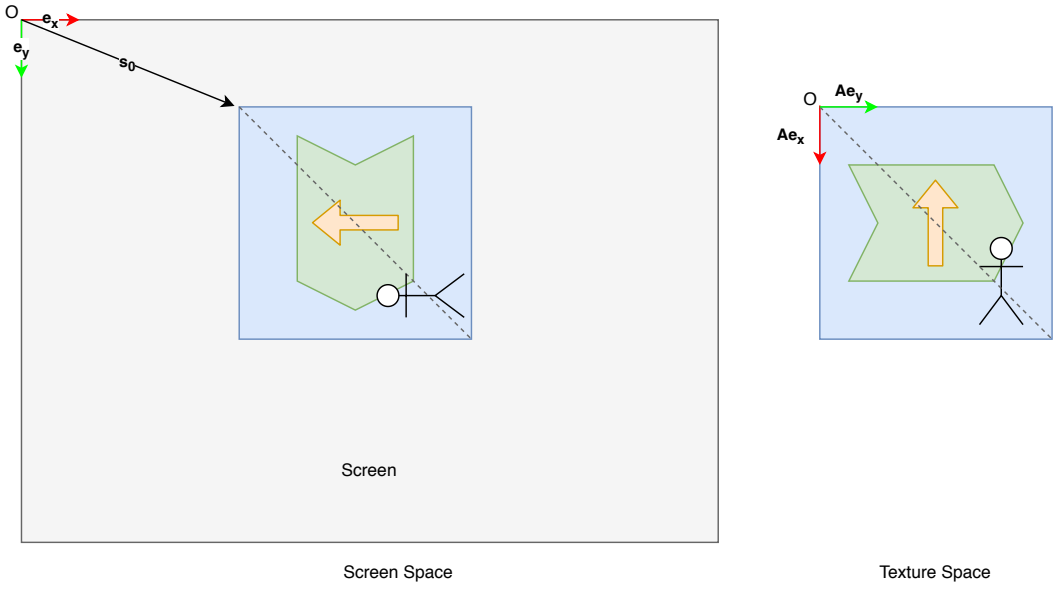
Like `BLIT`, `ABLIT` renders to a square area of the screen, starting at $\mathbf{s}_0$ (defined by the `x`, `y` arguments to `ABLIT`), and extending rightward and downward by the texture size. If we set $\mathbf{A} = \mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$, then the texture coordinate $\mathbf{u}$ is equal to the raster coordinate $\mathbf{s} - \mathbf{s}_0$, and the `ABLIT` behaves the same as a `BLIT`:

Figure 11: PPU affine blit with identity mapping



Setting $\mathbf{A}$ to the identity matrix maps the screen-space unit vectors $\mathbf{e}_x$ and $\mathbf{e}_y$ to the texture space's unit vectors $\mathbf{e}_u$, $\mathbf{e}_v$. The texture is undistorted. If we instead set $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$:

Figure 12: PPU affine blit with reflection in u=v

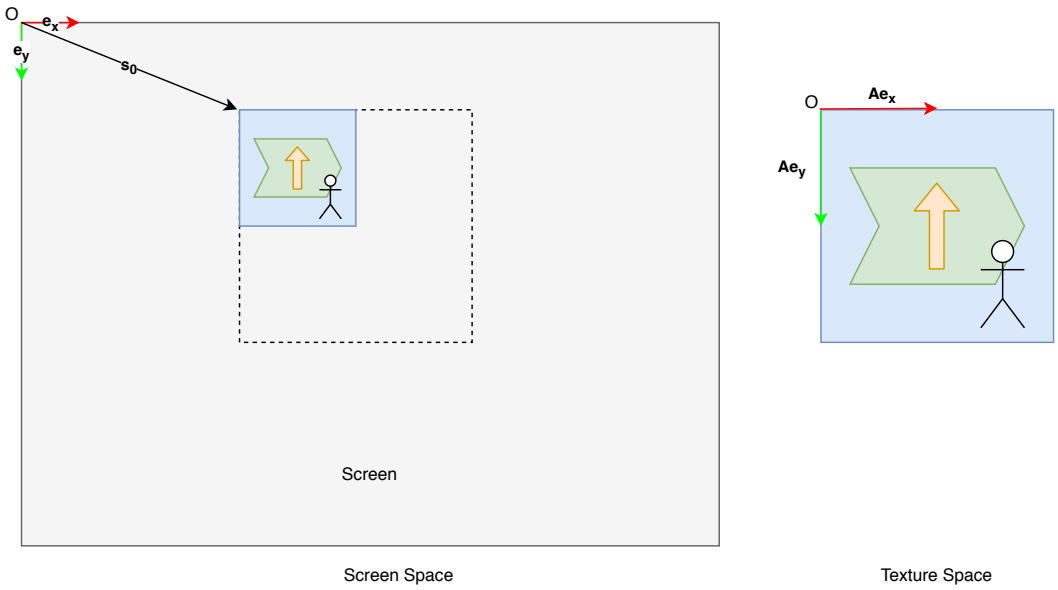


The horizontal axis in screen space ($\mathbf{e}_x$) has instead been mapped to the vertical axis in texture space ($\mathbf{e}_v$), and vice versa. Texture lookups have been reflected across the line $u = v$.

We can perform a uniform scale by setting $\mathbf{A} = \lambda\mathbf{I}$, which maps the screen-space unit vectors onto *larger* vectors in texture space, so that texture samples of neighbouring screen pixels are further apart on the texture. E.g. take

$\lambda = 2$:

Figure 13: PPU affine blit with uniform scale



The $\mathbf{b}$ vector is used to translate texture lookups. It can be understood as the point in texture space to which the upper-left of the rasterised region is mapped, since at this point on the screen $\mathbf{s} = \mathbf{s}_0$, so $\mathbf{u} = \mathbf{A}(\mathbf{s}_0 - \mathbf{s}_0) + \mathbf{b} = \mathbf{b}$. For example, we could set $\mathbf{b} = \begin{bmatrix} -32 & -32 \end{bmatrix}^T$, so that the sample corresponding to $\mathbf{s}_0$ is higher and further left in texture space:

Figure 14: PPU affine blit with uniform scale and translation

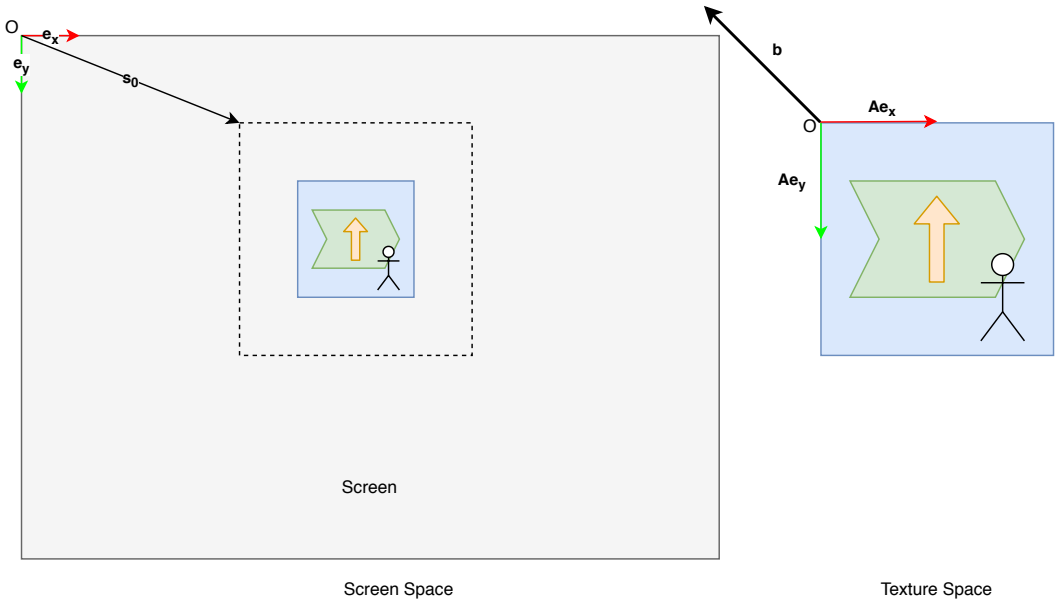
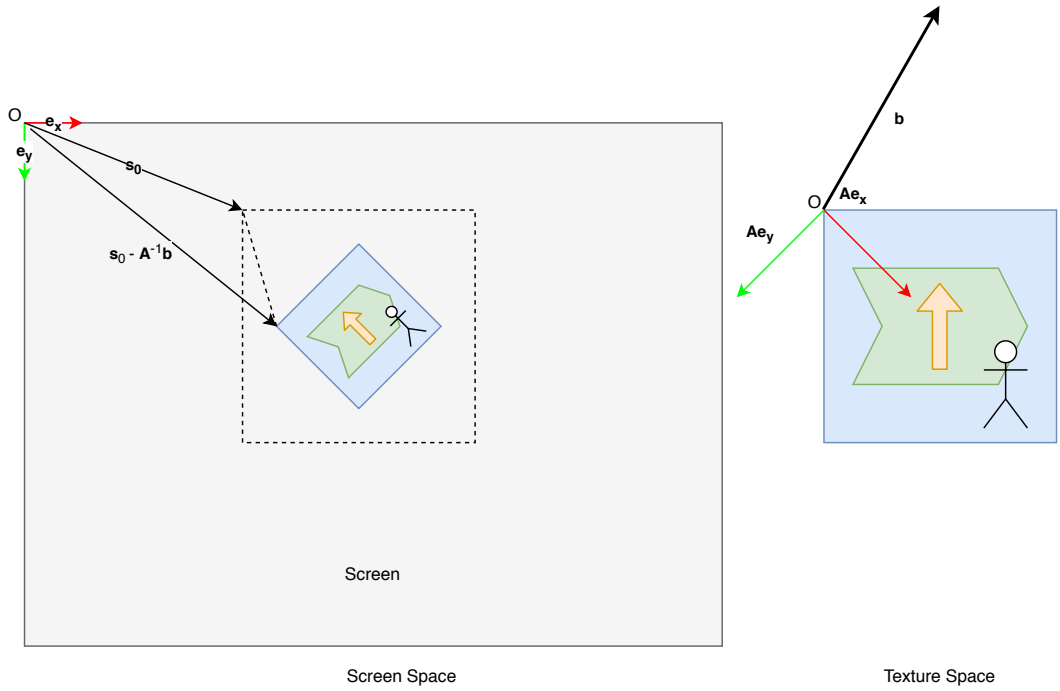


Figure 14 shows $\mathbf{b}$ plotted in texture space, and the resulting shift of the sampled texture as seen in the rasterised area. For a final example, the matrix

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Defines, in our coordinate system, a clockwise rotation by θ. Setting $\theta = 45°$, scaling by 2 and carefully choosing the $\mathbf{b}$ vector, we can make the texture appear to rotate *counterclockwise* about its centre in screen space.

Figure 15: PPU affine blit with rotation

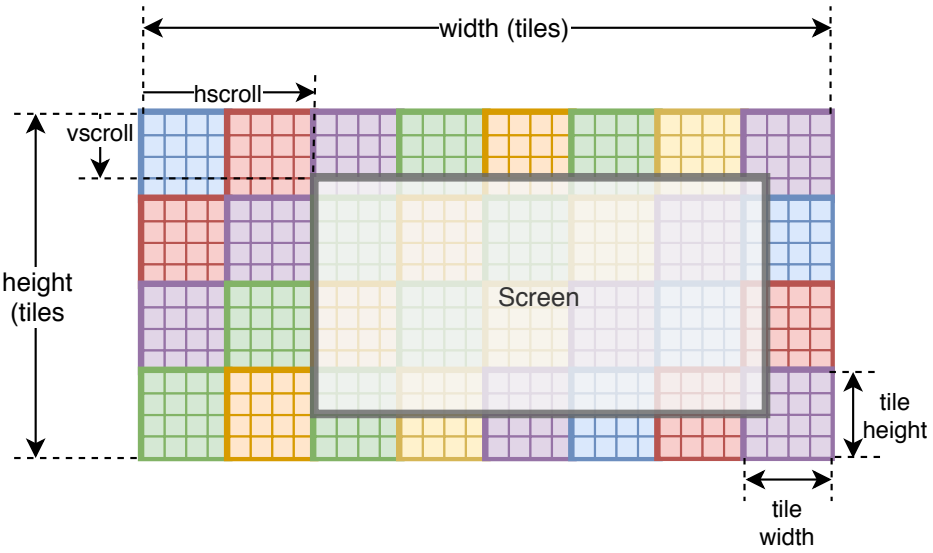


Note that, without a scale factor, the corners of the texture would be clipped to the rasterised region of the screen. It is quite inefficient to take a large texture and always show it in a scaled-down form; `ABLIT`'s half-size flag reduces the texture size to half of the rasterised region, so a 64×64 px `ABLIT` would expect a 32×32 px texture, which would occupy the upper-left quadrant of the rasterised region if $\mathbf{A} = \mathbf{I}$ and $\mathbf{b} = \mathbf{0}$. This could then be rotated without scaling, and provided it is appropriately translated, the corners of the texture would not be clipped.

3.6 Tiles

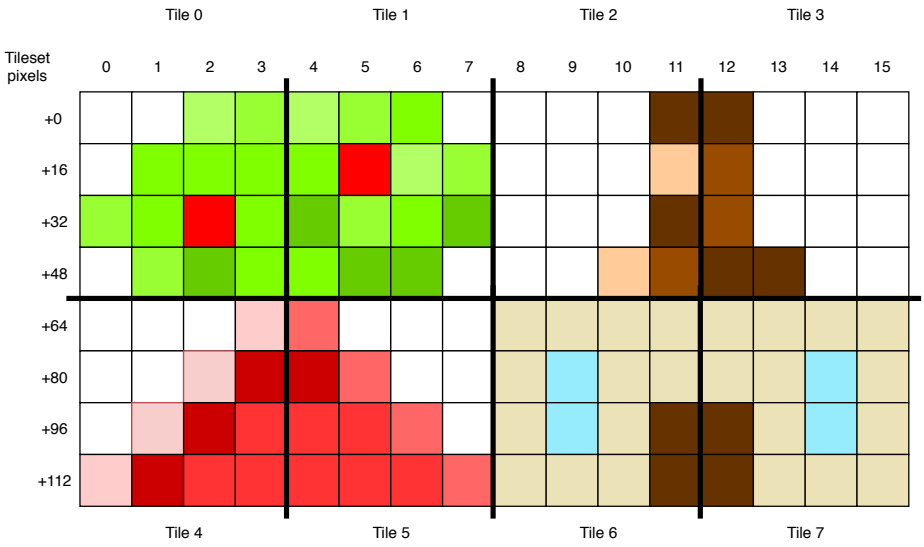
To reduce memory footprint, the PPU is able to assemble scenes on the fly from an array of small images, known as tiles. TODO much of this section refers to old hw with dedicated tiled backgrounds engines

Figure 16: PPU background coordinate system



Each tile is a square image, the width and height of which (measured in pixels) is configured per-background, and is always a power of two. Tiles are stored as part of a larger image, known as the tileset. Tiles are numbered in a row-major order, starting at the top left of the tileset. An example tileset is shown in figure 17: this is a tileset of 8 tiles, each 4×4 pixels in size.

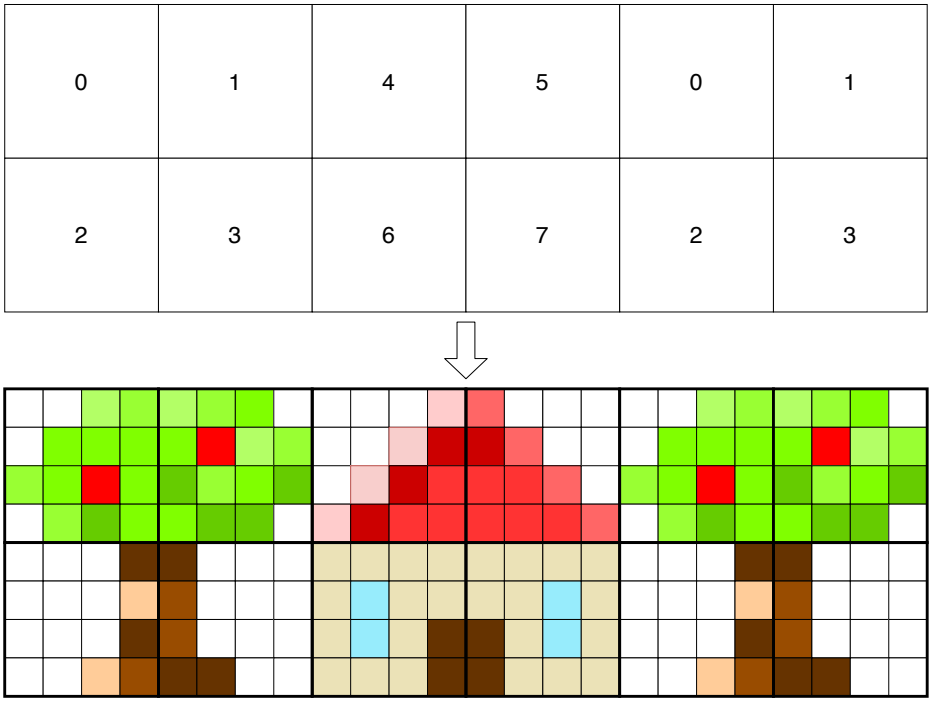
Figure 17: Example PPU tileset



Each pixel row of the tileset is stored as a packed array of pixels in memory, and the next row follows. Note that all tiles must be the same size, and have the same pixel format.

The complete background image is assembled from these tiles. The arrangement is specified the tilemap: a grid of numbers, each naming a tile from the tileset. For each pixel on the screen, the tilemap tells the PPU which tile should be at that location, and the corresponding tile image in the tileset tells the PPU the colour of each pixel in that screen tile. This is shown in figure 18.

Figure 18: Example PPU tilemap



The screen origin is offset into the background by configuring the horizontal and vertical scroll. This allows the game view to move smoothly over a fixed background. If the screen area overhangs the background, coordinates are wrapped.

In total, a background is defined by:

- Width and height, measured in pixels
- Horizontal and vertical scroll, measured in pixels
- Tileset:
  - Tile pixel format
  - Tile size in pixels (power of two)
  - Tileset width in pixels (power of two)
  - Pointer to the tileset image (aligned to row size in bytes)
- Tilemap:
  - Pointer to the tilemap buffer

This tilemapping process allows detailed images to be displayed with a minimal memory footprint, compared with a framebuffer, which must store the colour of every single pixel on the screen.

4 Audio Processing Unit (APU)

Note: this section is even less complete than the others

The APU is a lightweight processing core, designed to synthesise polyphonic audio, stream PCM or ADPCM audio from the system, or a mixture of the two, whilst blending/panning the audio channels. It runs in hard real time: each audio sample is calculated in one sample period, so no data buffering is required between APU and DACs. The design goals are:

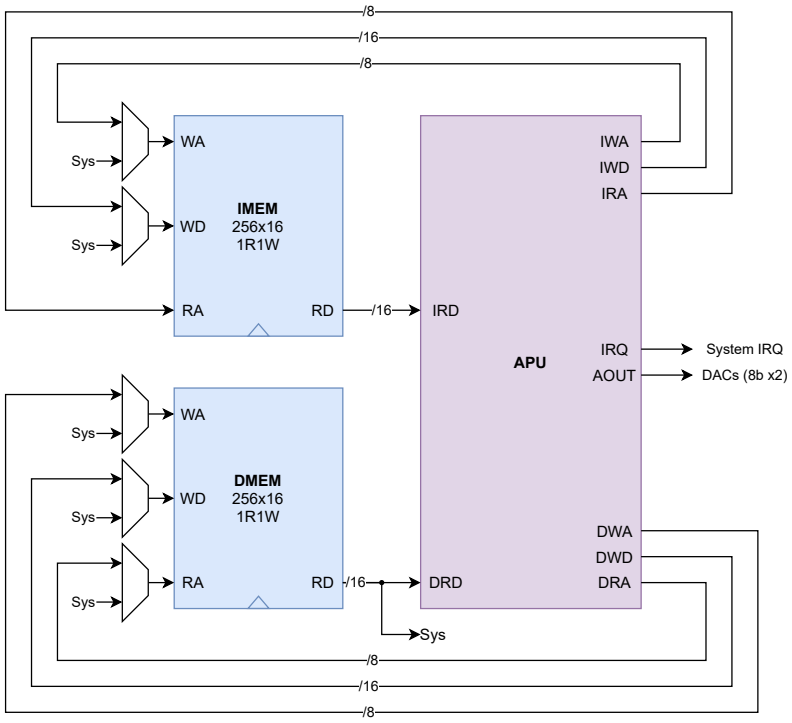
1. Low resource utilisation (100 LUTs for the processing core)
2. Output 48 kSa/s 8-bit stereo audio with a 12 MHz clock
3. Provide similar interface to an 8-bit era games console with default APU code, so no programming required
4. Streaming PCM audio from the system should be trivial (ring buffer + IRQ)
5. Be fun to program: very focused on a single task, so quirky but easy to learn

Architectural Overview

Memory Architecture

The APU is designed around a pair of iCE40 block RAMs: each is 16 bits wide by 256 entries deep, and has an independent read port and write port. The architecture and microarchitecture are designed around these parameters, for efficient use of the memory, and in particular for high utilisation of the memory ports. Figure **??** shows this in overview.

Figure 19: Audio processing unit memory architecture



- IMEM:
  - Contains the APU program instructions. Each instruction is 16 bits in size.
  - Upper 8 slots contain a copy of the APU's general purpose registers

  - The system has write-only access when the APU is halted, and no access when the APU is running

- DMEM:
  - Contains sample ring buffers used for wave tables or for buffering PCM data from the system
  - Contains program temporary variables that don't fit into registers or need to be indexed
  - Upper 8 slots contain a copy of the APU's general purpose registers
  - The system has read-write access at all times, at a lower priority than the APU (system access stalls when the APU is accessing DMEM)

8 general-purpose registers are available to APU programs. The register contents are mirrored across IMEM and DMEM, so that two independent register reads can take place simultaneously. Register writeback of one instruction is overlapped with instruction fetch of the next instruction. A typical register-to-register instruction (e.g. an add) executes in two cycles, performs 3 memory reads and 2 memory writes.

Arithmetic

There are two data types used by APU instructions:

- 16-bit integers
- Packed pairs of 8-bit integers (SIMD)

The former is useful for control variables and for phase accumulation of digital oscillators. The latter is used for SIMD processing of stereo sampling pairs. All 8-bit arithmetic is signed-saturating. 16-bit arithmetic can generally be either signed-saturating or normal modular arithmetic.

Input/Output

The APU performs all calculations necessary to produce a 2×8-bit stereo sample pair, then outputs this to the external DACs (probably some kind of sigma delta thing) with an `out` instruction. The APU then stalls until the end of the audio sample period, whereupon it begins calculating the next samples.

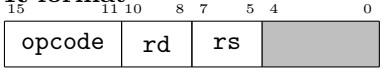
For PCM sample streaming, the APU accesses ring buffers in DMEM, in the same way that it access DMEM wavetables for local synthesis. The system tops up the ring buffers by writing to DMEM, triggered by APU interrupts.

The `irq` instruction can generate a system interrupt, which is mainly useful for half-empty or quarter-empty interrupts for PCM ring buffers.

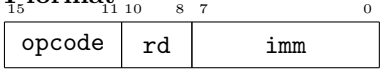
4.1 Instruction Set

The two main instruction formats are:

R format



I format



Example Programs

Wave channel

```
.imem                       ; List instruction memory contents
        ld r1, [#freq]          ; Initialise frequency from data memory
        li r0, #0               ; Initialise phase to 0
loop:
        addm16 r0, r1           ; Modular 16-bit addition to advance phase
        ldw r2, [r0, #sine]     ; Look up 2x8-bit samples in sine wave table
        out r2                  ; Output samples to DAC, stall until next sample
        b loop                  ; Repeat forever

.dmem                       ; List data memory contents
sine:
        ...
freq:
        .hword 0x1234
```

4.1.1 Wave channel, controllable frequency

```
.imem
loop:
        ld r1, [#freq]          ; Fetch frequency stored in memory
        ll r0, [#phase]         ; Fetch phase stored in memory, and set exclusive flag
        addm16 r0, r1           ; Increment phase by frequency
        sc r0, [#phase]         ; Store phase back into memory if still exclusive
        ldw r0, [r0, #sine]     ; Load wave sample from sine table
        out r0                  ; Output 2x8-bit samples from r0
        b loop                  ; Always jump to loop

.dmem
sine:
        ...
freq:
        .hword 0x1234
phase:
        .hword 0
```

4.1.2 Wave channel, controllable volume with decay

```
.imem
loop:
        ld r1, [#freq]          ; Atomic update phase based on frequency
        ll r0, [#phase]
        addm16 r0, r1
        sc r0, [#phase]
        ld r2, [#decay]         ; Atomic update volume based on linear decay
        ll r1, [#volume]
        sub16 r1, r2
        sc r1, [#volume]
        ldw r0, [r0, #sine]     ; Look up wave table based on phase
        movhl r1, r1            ; Duplicate upper byte of volume
        mul48 r0, r1            ; Multiply each sample by 4 volume MSBs
        out
        b loop

.dmem
sine:
        ...
freq:
        .hword 0x1234
```

```
phase:
        .hword 0
decay:
        .hword 0xabcd
volume:
        .hword 0
```

4.1.3 Multiple wave channels

5 Bus Fabric and Memory Subsystem

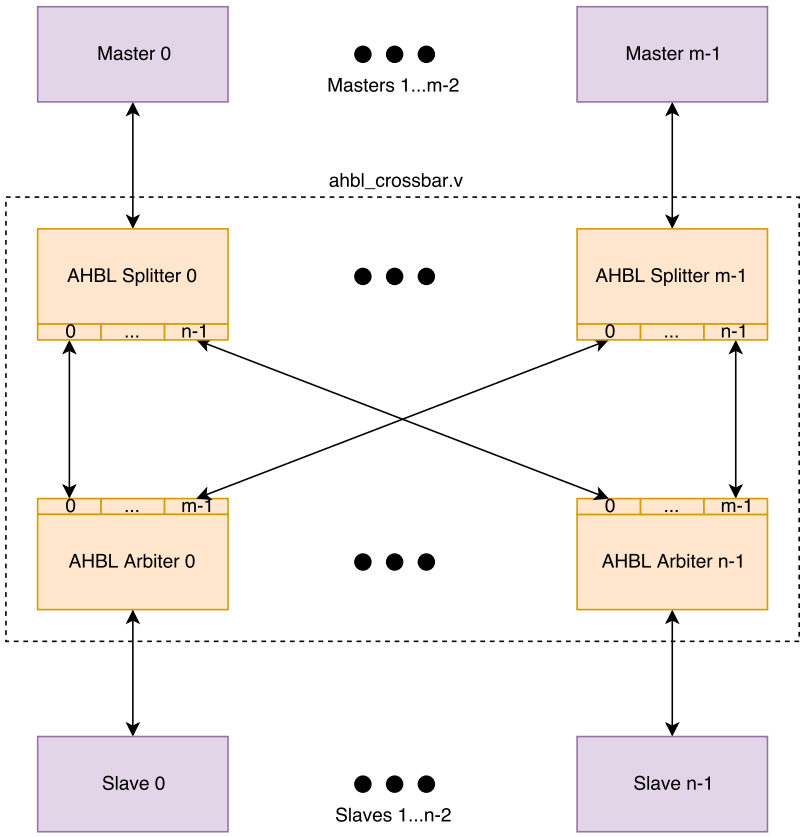
Bus fabric is digital plumbing. A master, such as a processor, requests a read or write on some address; the bus fabric routes the request to the correct slave device, and routes the response back. RISCBoy implements two bus fabric standards:

- AMBA 3 AHB-Lite connects masters to high-performance devices such as SRAM controllers
- AMBA 3 APB connects to simple devices such as a UART

Figure 20 shows the structure of the AHB-Lite crossbar (`ahbl_crossbar.v`). The crossbar is shown in context in figure 1. An independent AHB-Lite datapath connects each of m masters to each of n slaves. One master can address one slave at a time, and one slave can be in data-phase with one master at a time; subject to these constraints, up to $\min(m, n)$ independent transfers can take place in a single machine clock cycle.

Some claim AHB-Lite does not "support" multi-master arbitration. Their problem is a lack of enthusiasm: motorbikes do not "support" wheelies by design, but are excellent at it.

Figure 20: AHB-Lite crossbar, module-level block diagram



Each master is under the illusion that it is the only master in the system, but that slaves sometimes take longer to respond. During this waiting period, the slave may actually have fielded multiple transactions from higher-priority masters; this interaction is handled by the slave's AHB-Lite arbiter, and is transparent to the masters.

One of the crossbar's slave ports is attached to an AHBL-APB bridge. This bridge appears as a slave to the AHB portion of the bus fabric, and as a master to the APB portion. There are three main benefits to this scheme:

- APB is fundamentally simpler
  - This keeps peripheral gate count down
  - The peripherals on the APB bus do not need the full AHB-Lite bandwidth anyway

- Fewer AHB-Lite slaves
  - There is a nonlinear area scaling associated with adding slaves to the AHB-Lite fabric
  - This would also add extra gate delays to a fairly critical data path
- One APB master
  - AHB-Lite masters get arbitrated down to one inside the AHB-Lite crossbar. APB slaves do not care who is addressing them.
  - Different masters accessing different APB slaves will have to queue to use the bridge, even though they could theoretically proceed simultaneously
  - However, area/complexity vs performance tradeoff is more than worth it for slow peripherals
  - Multi-master APB is easy to implement, but never used in practice, due to the above tradeoff

The splitter and arbiter modules in the AHB-Lite crossbar can also be used on their own. Arbitrary multi-layer busfabric topologies should be possible with these two components.

Currently, the RISCBoy busfabric does not support AHB-Lite bursts (TODO), and the masters do not use them.

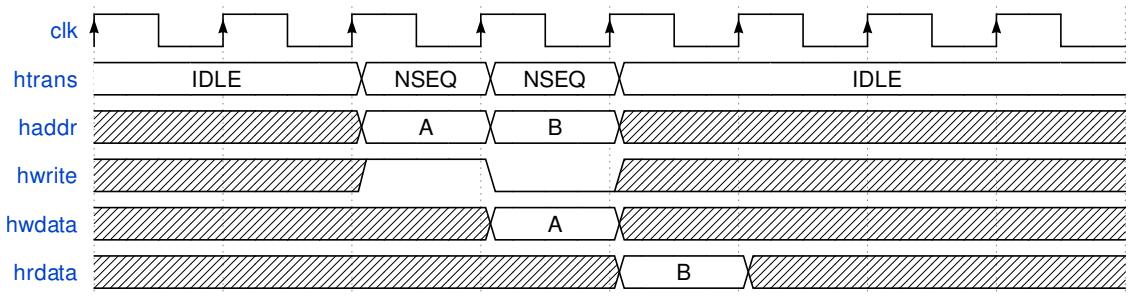
5.1 AHB-Lite Primer

For a full understanding of the bus standard used by RISCBoy, read through ARM's AMBA 3 AHB-Lite spec. This document is mirrored in the `reference` folder in the GitHub repository, and gives a clear and comprehensive breakdown of AHB-Lite. However, the following overview should provide sufficient understanding of the standard to read through the Verilog.

Transactions take place in two phases, named the address phase and the data phase. During the address phase, the master asserts signals which control the nature of the transfer, such as the address, whether the transfer is a read or write, protection/permission information, the width of the data, and so on. During the data phase, data is asserted on either the read or write data bus (`hrdata` and `hwdata`), but never both. `hrdata` is driven by the slave, and `hwdata` by the master.

The central conceit of AHB-Lite is that these two phases are *pipelined*. Whilst the master is asserting or accepting data for an earlier transaction (currently in data phase), it concurrently asserts address and control information for a later transaction (currently in address phase). As is generally the case with pipelining, the goal is to enable higher clock frequencies with undiminished work-per-clock.

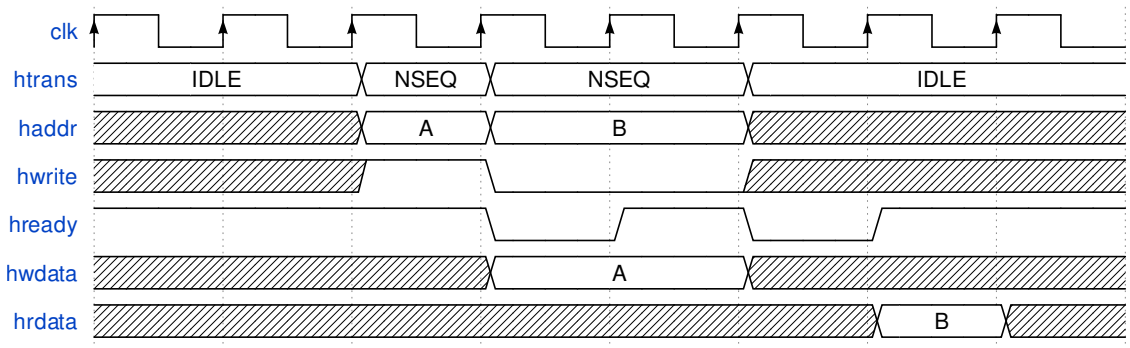
Figure 21: AHB-Lite transfers, a simple example



In figure 21, a master carries out two AHB-Lite transactions: a write to address A, followed by a read from address B. Only a subset of AHB-Lite signals are shown on the diagram. `htrans`, `haddr`, and `hwrite` are driven by the master, during the address phase; the other two are data phase signals. `htrans` indicates the type of transfer the master next wishes to perform, which is one of `IDLE`, `NSEQ` (non-sequential), `SEQ` and `BUSY`. The latter two are exclusive to burst transactions, which are not used in RISCBoy. (See the ARM spec if you do want details.)

Sometimes a slave is unable to service a request immediately, as shown in figure 22. `hready` is a data phase signal, which signifies the end of the current data phase.

Figure 22: AHB-Lite transfers, simple example with stalling



This slave needs two cycles to perform each data phase; perhaps it is an SRAM capable of running only at half the system clock speed. Therefore, `hready` is low for one cycle, and high for the second (last) cycle of each data phase. The master drives `hwdata` for the duration of A's data phase, waiting for the slave to signal completion. `hrdata`, on the other hand, is invalid until the final cycle of a read data phase.

Note that an address phase does not complete until the preceding transfer's data phase completes. In figure 22, address B (and associated address-phase signals) continue to be driven until the A data phase completes. `IDLE` transactions do have a data phase, which always completes immediately. Consequently, `hready` idles high while the bus is in an idle state. This is why A's address phase completes immediately in the figure.

In a practical system, there are multiple slaves. Each drives a signal called `hreadyout`, to indicate that *that slave* is ready. The bus fabric tracks which slave the master is currently accessing in the data phase, and selects that slave's `hreadyout` to be the global `hready`. To see why this is necessary, think about the situation where a master is in data phase with one slave, and address phase with a different slave.

5.2 Multi-Master Operation

In a single-master busfabric, `hready` is a global signal, which causes the entire AHB-Lite state machine (masters, slaves, fabric, the lot) to advance. Where multiple masters are concerned, `hready` is more subtle; in part it is a per-master stall signal. At this point we need to be more specific about the relationship between `hreadyout` and `hready`.

Any AHB-Lite slave port (of which there is one on the master side of the splitter, n on the master side of the arbiter, and one on each slave device) has an output called `hreadyout`, which indicates the slave's readiness. Each of these ports also has an input called `hready`, which indicates that the data phase is ending for the master who is connected to this slave (which does not mean that it is in data phase with *this* slave; it may be addressing this slave while in data phase with another). `hready` is a function of `hreadyout`s and bus state. The connections between masters, splitters, arbiters and slaves are shown in figure 20.

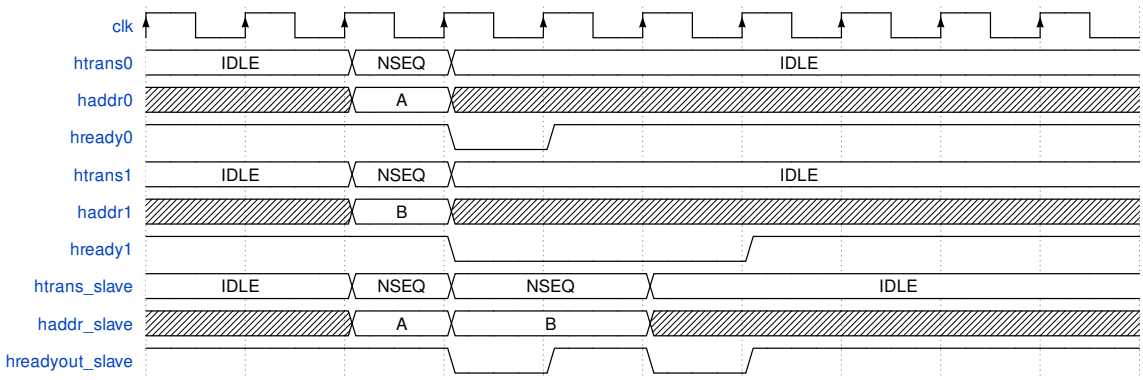
In the single-layer crossbar on RISCBoy, each system AHB-Lite slave is the slave of an arbiter, which is the slave of several splitters, each of which is the slave of a system master. As a general rule, the busfabric must filter system slaves' `hreadyout`s up to each system master, tie `hreadyout`s across to `hready`s at the very top of the busfabric, and then distribute these `hready` signals down to the correct system slaves.

5.2.1 Multiple Masters, One Slave

The arbiters are the most complex busfabric component, so it is instructive to consider interactions between multiple masters and a single slave, which are mediated by one arbiter. There are additional complexities when we combine arbiters and splitters to build a crossbar, which are discussed in the next section.

In figure 23, two masters attempt to access a single slave simultaneously. Assume that master 0 always wins address-phase arbitration:

Figure 23: AHB-Lite transfers, two masters access one slave



Again, we assume the slave requires 2 cycles to complete each data phase.

If we look at each master's trace, there is no indication at all that there is more than one master in the system: they present an address, and subsequently the transaction completes. Likewise, the slave neither knows nor cares that there are multiple masters: it simply carries out transactions according to the address-phase signals it sees. All of the smoke, mirrors and machinery are inside of the arbiter.

One odd feature of this trace is that, when the slave sees the address B, no master is asserting this address.

1. Initially, both masters assert `IDLE`; `IDLE` data phases complete in one cycle
2. `IDLE` data phases are concurrent with A, B address phases, so these *also* complete immediately
3. From the master 1's point of view, transaction B proceeds immediately to data phase.
4. From both the master 0's and the slave's point of view, transaction A proceeds immediately to data phase
5. Whilst the slave is in data phase for A, it is simultaneously in address phase for B
6. When A data phase completes, master 0 is signaled, and B proceeds to data phase at the slave
7. When B data phase completes, master 1 is signaled

More concisely put, the first clock cycle of a given transaction's data phase may differ between the slave and master, but the *last* cycle of that data phase is always the same clock cycle. The slave address phase will occur some time between the master address phase starting, and the slave data phase starting. These are strong enough guarantees for correct operation.

Based on this discussion, the AHB-Lite arbiters need the facility to buffer one address-phase request, per master. A buffered request will be applied before any new requests from that master, but after any higher-priority requests. There is a nonzero hardware cost to this buffering, but there are clear engineering benefits to keeping this complexity confined to the arbiters, as they are the only component in the busfabric which is explicitly "multi master".

Figure 24: AHB-Lite transfers, two masters access one slave, with low-priority back-to-back

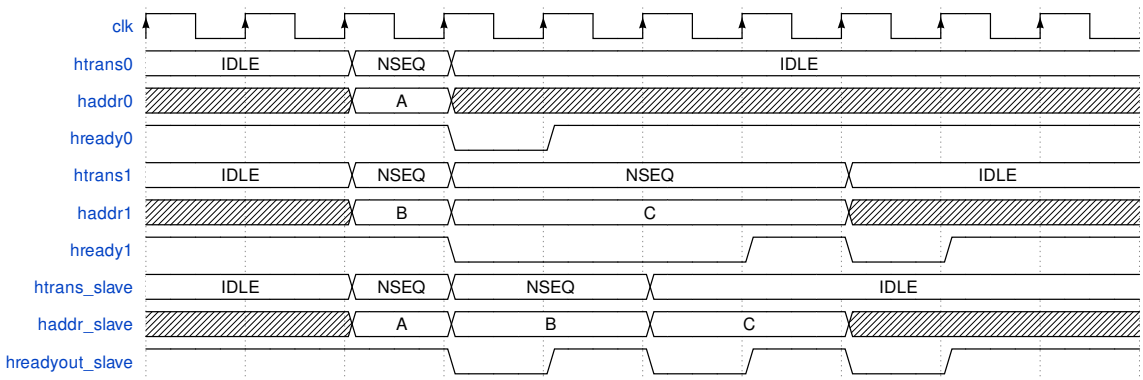
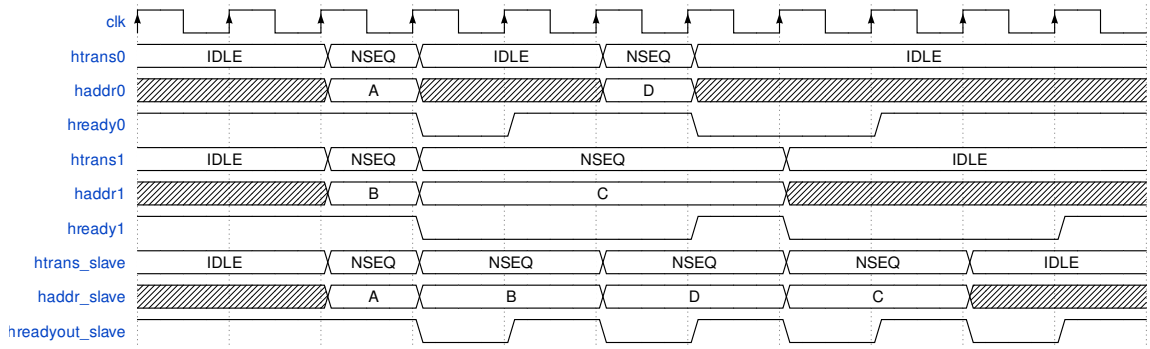


Figure 24 shows the same sequence of events as figure 23, except master 1 now performs two back-to-back transactions. Once B's slave address phase completes, the arbiter's request buffer is cleared, and the C request passes transparently through the arbiter to the slave. Again, the only indication to master 1 of any master 0 activity is increased latency.

There is a different case which requires the arbiter's request buffer, shown in figure 25.

Figure 25: AHB-Lite arbiter: simultaneous request buffer writes

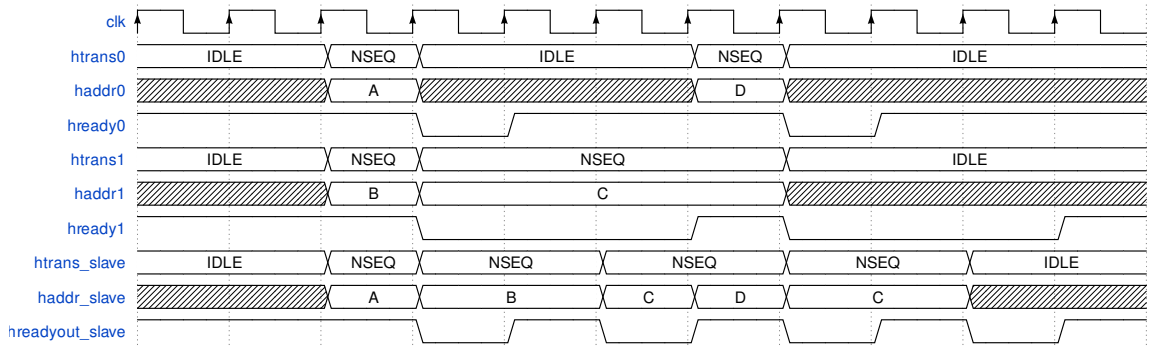


At the instant where D address phase is asserted, `hready0` is high, because master 0 previously asserted an `IDLE` transfer. However, the slave is not ready. In this case, the arbiter needs to buffer master 0's request, even though it is the highest-priority master. The buffered request is cleared once its slave address phase completes, as usual.

On the next cycle, B's data phase completes, and master 1 also considers this to be the end of the C address phase. The arbiter must write the C request into master 1's request buffer. Master 0's buffered request will continue to take priority over master 1's buffered request, until the first buffer is cleared.

There is one final case, for two masters accessing one slave, which is worth being aware of (figure 26).

Figure 26: AHB-Lite arbiter: late arrival of high priority request



Whilst `hreadyout` is low, the C address briefly appears on the slave bus, before being replaced by the higher-priority D request. **This is a departure from the AHB-Lite standard**, which stipulates the address must be constant during this time. This is deliberate, and easily amended. Slaves are generally insensitive to address-phase request during this time (as there is no performance benefit to latching APR before `hreadyout`, due to the way the bus operates), and this avoids a priority inversion, reducing average latency for higher-priority masters. If you find something that this breaks, write me an angry email! I would be interested to see such a slave.

The D request causes the low-priority C request to be buffered; the B data phase completes on this cycle, hence, from master 0's point of view, the C address phase does too.

5.2.2 Full Crossbar

The previous section discussed some cases where multiple masters access a single slave, and showed how the arbiter safely navigates them. There are yet more issues to consider when multiple masters and multiple slaves are involved,

which must be handled without added latency cycles, and with minimal extra gate delay.

For example, a master may be engaged in address phase with one arbiter and data phase with another arbiter simultaneously, via a splitter, and these two arbiters will not necessarily signal `hreadyout` at the same time. Consequently, a master may have a positive `hready`, filtered from its data phase arbiter, when its address phase arbiter has a negative `hreadyout`, which requires action on the arbiter's part.

There is also the issue that being in data phase with an arbiter does not mean you are genuinely in data phase with the arbitrated slave; in fact, a very simple sequence of events (all masters `IDLE` $\rightarrow$ all masters `NSEQ`) will put all masters simultaneously in data phase with the same arbiter. The arbiter behaviour described in the previous section should allow us to abstract this away, provided we can deal with the first issue safely.

Splitters will filter their slaves' `hreadyout`s based on which is currently in data phase, and present it on their own slave port. Arbiters will present their slave's `hreadyout` on any master-facing ports which are in data phase with the arbiter, and will present `hreadyout` = 1 on any idle ports.

Splitters will fan their `hready` signal out to all of their slaves; a low `hready` directed at a slave you are not engaged with is harmless.

5.3 Memories

RISCBoy possesses two memory areas for data and code: an external SRAM (512 kiB, 16 b wide, asynchronous), and internal RAM (8 kiB, 32 b wide, synchronous), assembled from FPGA block RAMs. The former is the main system memory, which most games will simply load into as a flat image; the latter is intended to be used for processor stack, and some critical code sections, including ISRs. Internal RAM also contains the first-stage bootloader, as it can be initialised during FPGA configuration, so is a convenient place to put the first few thousand instructions the processor will execute at start-of-day. SRAM was chosen for the external memory due to the low initial access latency – this permits reasonable performance without building complex cache hierarchies into the system bus masters. On a larger FPGA we could consider something like HyperRAM, or even DDRx SDRAM.

Both memories need to be interfaced to the AHB-lite system bus. SRAMs aren't too complex, but they can be a little fiddly to use efficiently due to the timing of AHB-lite – in particular, the alignment of `haddr` and `hwdata` is more or less the opposite of what you want for a synchronous SRAM.

The internal SRAM can perform one 32-bit read or write per cycle. The external SRAM can also service a 32-bit AHB-lite read every cycle, because it is double-pumped; the controller performs two back-to-back 16 bit SRAM reads in one clock cycle. However, writes to external SRAM are limited to 16 bits per clock cycle, due to the timing of the `WE` pin, which is deasserted part way through each access. This is reasonable – reads are more common than writes, for almost any workload. However, special cases like the processor stack will benefit from the improved write performance of IRAM.

5.3.1 Internal RAM

Synchronous single-port SRAMs typically behave as follows:

- Address, write data and write enable are presented on one clock
- Read data is available on the next clock (and the whole process is pipelined)
- Byte-enables allow narrow writes to be performed without a read-modify-write sequence.

This is shown in figure 27. However, in AHB-lite, read and write data are both presented during the data phase, which begins on the cycle after the address phase completes. The timing of this is shown in figure 28. Note the difference in timing between `wdata` in figure 27 and `hwdata` in figure 28.

Figure 27: Timing of synchronous SRAM interface

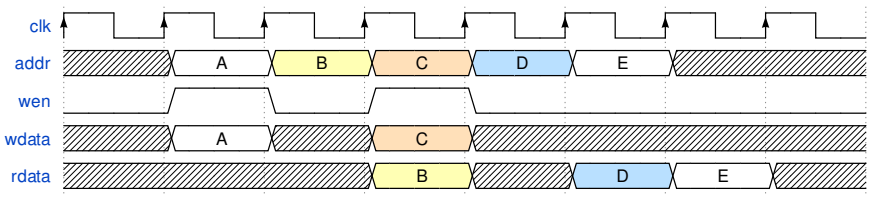
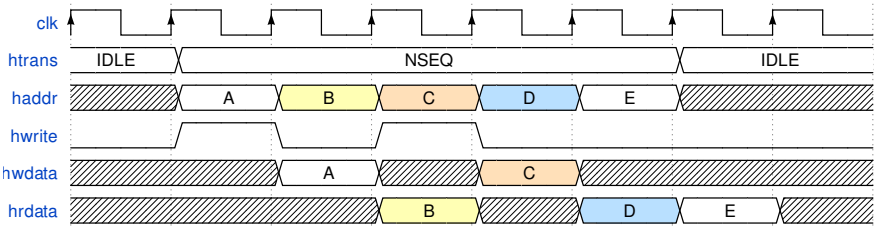
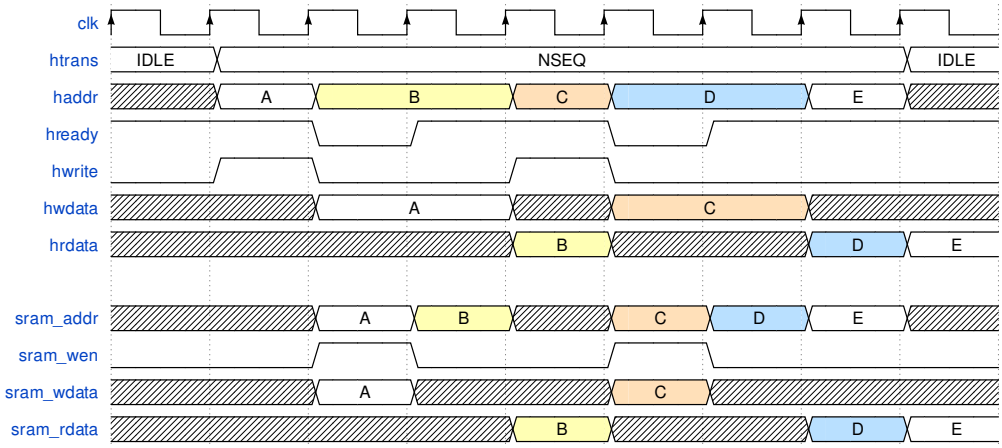


Figure 28: Timing of AHB-lite SRAM accesses, showing write data misalignment



The issue is that, as we can't source signals from the future, the SRAM write cycle must be delayed so that write address is aligned with write data. However, if the next transfer is a read, there is a collision: the delayed write address would need to be presented on the same cycle as the read address, which is impossible for a single-port SRAM. One way to resolve this is to simply insert wait states on write access, shown in figure 29.

Figure 29: AHB-lite SRAM access: resolving address collision with wait states

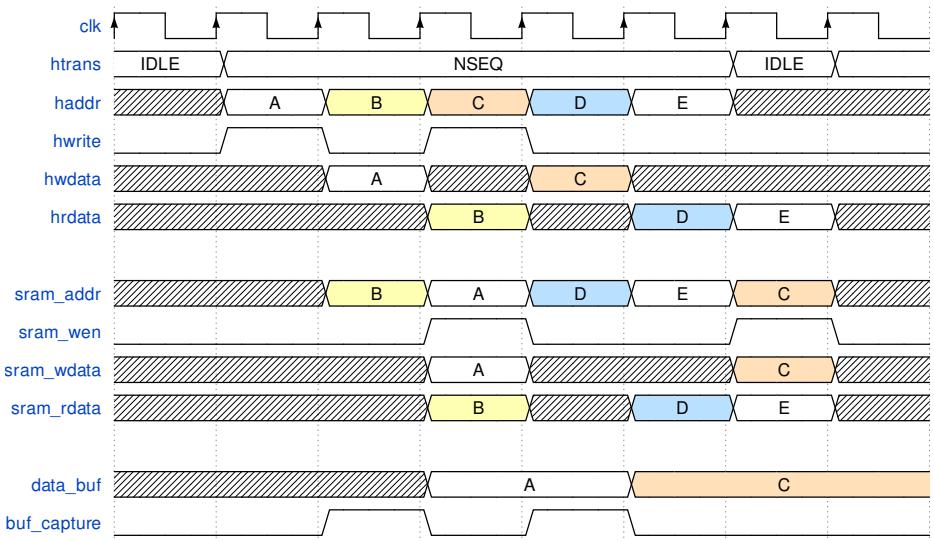


This works, but the performance cost is significant if you are constantly swapping back and forth between reads and writes (like, say, a processor!). This needn't be so. Noting that:

- Multiple consecutive writes are fine; all addresses are delayed equally, so do not collide
- Write-to-read has a single collision, and no more
- A run of consecutive reads is always followed by either an idle cycle or a write cycle (by definition)
- Read-to-write leaves the SRAM idle for one cycle; this is seen in figure 29

If we are able to buffer a single write, and hold onto it until the SRAM is idle (either AHB idle or read-to-write), we never need wait states. The only additional wrinkle is that, if a read is issued to an address we are currently holding a buffered write to, we will need to merge the buffered write data into the read data on-the-fly. As the write may be narrower than the SRAM, this needs to be done on a byte-by-byte basis. Use of a write buffer is shown in figure 30.

Figure 30: AHB-lite SRAM access: resolving address collision with a write buffer



It's probably worth staring at for a short while. Key observations for figure 30:

- SRAM reads are always aligned with AHB-lite reads, to avoid adding latency
- SRAM writes are in-order with respect to other SRAM writes (otherwise we'd need more buffering)
- An SRAM write can be deferred for any number of read cycles, after which the SRAM address bus will be free
- The data buffer is only needed for write-to-read, not write-to-write or write-to-idle.

5.3.2 Main Memory

Main memory consists of a 16-bit-wide, 512 kiB asynchronous SRAM. The rough timing of read and write cycles is shown below (just signal alignment, no timing parameters):

(TODO)

Currently the controller requires a single cycle for each 16-bit access, and services 32-bit accesses from the busfabric as two back-to-back external access cycles; a single wait state is inserted via `hready`. At RISCBoy's target clock frequency of 36 MHz, this cuts the available bandwidth down to 72 MiB/s.

The part used has a 10 ns access time, and RISCBoy is targeting around a 36 MHz system clock period (~28 ns), so with some care we may be able to perform two reads in the same clock cycle, using a DDR output on the least-significant address bit, and an optional negedge capture fed into the lower half of the AHB-lite data bus. This means we can service one 32-bit AHB-lite read each cycle, which is ideal for a 32-bit processor with up to 32-bit instructions!

Writes are more problematic: due to the timing of the `WE` pin, we need a full cycle for each 16-bit write cycle, so 32-bit writes will stall for one cycle. Write-intensive memory regions such as the processor stack should be located in IRAM if possible, to avoid this performance penalty.