Digital Design and Computer Architecture

Laurin Seeholzer

June 1, 2026

Contents

1 Fundamentals

1.1 Metal-Oxide-Semiconductor Transistors

Definition 1.1. An **n-type** MOS transistor acts as a switch that is **closed (ON)** when the Gate voltage is High (1) and **open (OFF)** when the Gate voltage is Low (0). It is efficient at passing a logic 0 (strong 0) but inefficient at passing a logic 1 (weak 1).

Definition 1.2. A **p-type** MOS transistor is the dual of nMOS. It is **closed (ON)** when the Gate voltage is Low (0) and **open (OFF)** when the Gate voltage is High (1). It is efficient at passing a logic 1 (strong 1) but inefficient at passing a logic 0 (weak 0).

Definition 1.3. In **Complementary MOS (CMOS)** logic, transistors are organized into two networks:

- **Pull-Up Network (PUN):** Built with pMOS to connect the output to V_{DD} (1).
- **Pull-Down Network (PDN):** Built with nMOS to connect the output to Ground (0).

This ensures the output is always driven strongly to either 1 or 0 without drawing significant static power.

Example 1.4. The simplest CMOS gate is the NOT gate. It consists of one pMOS in the PUN and one nMOS in the PDN, both connected to the same input A.

$$Y = \overline{A}$$

1.2 Logic Gates

Definition 1.5. NOT: The simplest gate; it flips the input.

- **Logic:** $Y = \overline{A}$
- **Transistors:** 2 (1 pMOS, 1 nMOS)

Definition 1.6. NAND: Outputs 0 only if both inputs are 1.

- **Logic:** $Y = \overline{A \cdot B}$
- **Transistors:** 4 (2 pMOS parallel, 2 nMOS series)

Definition 1.7. NOR: Outputs 1 only if both inputs are 0.

- **Logic:** $Y = \overline{A + B}$
- **Transistors:** 4 (2 pMOS series, 2 nMOS parallel)

Remark 1.8. AND/OR are typically implemented as NAND/NOR followed by an inverter because inverting gates are more natural in CMOS. (**NAND/NOR** are universally used because they are efficient to build with 4 transistors for 2 inputs.)

Remark 1.9. XOR/XNOR: Used for comparisons and parity.

- **XOR:** $Y = A \oplus B$ (1 if inputs are **different**)
- **XNOR:** $Y = \overline{A \oplus B}$ (1 if inputs are the **same**)

Definition 1.10. Buffer: The output equals the input ($Y = A$). It is used for signal amplification or delay and is physically implemented as two NOT gates in series.

1.3 Boolean Algebra

Definition 1.11. Boolean algebra (and therfore logic gates) follow specific mathematical rules:

- **Identity:** $A \cdot 1 = A$ and $A + 0 = A$
- **Complement:** $A \cdot \overline{A} = 0$ and $A + \overline{A} = 1$
- **Distributive:** $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

Theorem 1.12. DeMorgan's theorems allow for the conversion between AND/OR and inverted logic:

$$\text{Law 1: } \overline{A \cdot B} = \overline{A} + \overline{B}$$

$$\text{Law 2: } \overline{A + B} = \overline{A} \cdot \overline{B}$$

Important 1.13. A set of gates is **logically complete** if any Boolean function can be realized using only those gates. The most common complete sets are $\{AND, OR, NOT\}$, $\{NAND\}$, and $\{NOR\}$.

Definition 1.14. Minterm: A product (AND) term containing all input variables in either true or complemented form. **Sum of Products (SOP)**: A function expressed as the OR of minterms for which the output is 1.

Algorithm 1.15

```
To simplify logic using a Karnaugh Map (K-Map)

1. Map the truth table values into a grid where adjacent cells
   differ by only one bit (Gray code).
2. Circle groups of 2^n adjacent 1s (rectangular blocks).
3. Each circle represents a simplified product term where
   variables that change within the circle are eliminated.
```

1.4 Combinational Logic

Definition 1.16. Combinational Logic is a circuit where the output is a pure function of the current inputs. It has no memory of past states.

1.4.1 Combinational Building Blocks

Definition 1.17. An n-to-2^n **decoder** interprets an n-bit binary input and sets exactly one of its 2^n outputs to 1. It is primarily used for memory addressing and instruction decoding.

Definition 1.18. A **Multiplexer** uses select lines (S) to choose which data input (D) is routed to the single output (Y). For N inputs, $log_2 N$ select lines are required.

Example 1.19. A MUX can implement any logic function by treating its inputs as a **Lookup Table (LUT)**. By wiring the data inputs to constants (0 or 1) and using variables as select signals, the MUX reproduces the truth table.

Definition 1.20. A **Full Adder** is the building block of arithmetic units. It takes inputs A, B, and a carry-in C_{in}, producing:

$$\text{Sum: } S = A \oplus B \oplus C_{in}$$

$$\text{Carry-out: } C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$$

Important 1.21. An n-bit **Ripple Carry Adder (RCA)** is constructed by chaining n Full Adders. The C_{out} of one stage becomes the C_{in} of the next. The critical

path (speed bottleneck) is the "ripple" effect as the carry bit travels from the LSB to the MSB.

Lemma 1.22. A **Tri-state buffer** has three possible output states: $0, 1$, and **High-Z** (High Impedance). When the enable signal is 0, the buffer is electrically disconnected. This allows multiple components to share a single **Bus** wire.

Example 1.23. A **Programmable Logic Array (PLA)** is a structured logic device consisting of a programmable **AND-plane** (creating minterms) and a programmable **OR-plane** (summing the minterms). It is a physical implementation of the Sum-of-Products (SOP) form.

1.5 Sequential Logic

1.5.1 Fundamental Building Blocks

Definition 1.24. Unlike combinational logic, where the output depends solely on current inputs, sequential logic produces outputs based on both current and past input values. These circuits incorporate storage elements to maintain a "state".

Definition 1.25. The most basic storage element consists of two inverters connected in a feedback loop. It has two stable states ($Q = 1$ or $Q = 0$) and a potential metastable state where outputs oscillate.

Definition 1.26. A Reset-Set (R-S) latch is typically built from cross-coupled NAND gates.

- **Set ($S = 0$):** Forces the output Q to 1.
- **Reset ($R = 0$):** Forces the output Q to 0.
- **Forbidden ($S = 0, R = 0$):** This state is invalid as it breaks the $Q = \overline{Q'}$ invariant and can lead to metastability.

Definition 1.27. The Gated D Latch adds a Write Enable (WE) signal to an R-S latch to control when data is stored. When $WE = 1$, the output Q follows the data input D; otherwise, it holds the previous value (Q_{prev}).

Important 1.28. Latches are "transparent" while their enable signal is high, meaning any change in D immediately propagates to Q. This is undesirable for state registers in synchronous systems because next-state values could overwrite current-state values before the cycle ends.

Definition 1.29. A D Flip-Flop solves the transparency problem by using a master-slave configuration of two D latches. It is **edge-triggered**, capturing the value of D only at the rising edge of the clock ($0 \to 1$ transition) and maintaining it for the entire cycle.

1.5.2 Timing and the Clock

Definition 1.30.

- **Asynchronous:** State transitions occur immediately in response to inputs with no global coordination.
- **Synchronous:** State transitions are synchronized to a global **clock** signal and occur at fixed units of time.

Definition 1.31. Clock (CLK): A periodic signal alternating between 0 and 1 that triggers state transitions across all sequential elements in a system. The clock period must be long enough to accommodate the worst-case delay of the combinational logic.

Important 1.32. Standard synchronous registers use **rising-clock-edge** triggered flip-flops. This ensures that the state changes only once per clock cycle, making system behavior predictable and easier to design than level-triggered latch-based systems.

1.5.3 Finite State Machines (FSM)

Definition 1.33. Finite State Machine (FSM): A discrete-time model of a stateful system consisting of a finite number of states, external inputs, external outputs, state transitions, and output logic.

Theorem 1.34. FSMs are categorized by how they generate outputs:

- **Moore FSM:** Outputs depend **only** on the current state.
- **Mealy FSM:** Outputs depend on **both** the current state and the current inputs.

Remark 1.35. Every FSM implementation consists of three parts:

1. **State Register:** Sequential circuit storing the current state.
2. **Next State Logic:** Combinational circuit determining the next state based on current state and inputs.
3. **Output Logic:** Combinational circuit generating system outputs.

Example 1.36. States must be represented as binary values. Common methods include:

- **Binary Encoding:** Uses the minimum number of bits (log_2(states)); minimizes flip-flops but may lead to complex combinational logic.
- **One-Hot Encoding:** Uses one bit per state; simplifies next-state logic but maximizes flip-flop usage.
- **Output Encoding:** Uses the output signals themselves as the state bits; minimizes output logic.

1.5.4 Memory and Registers

Definition 1.37. Register: A structure that stores multiple bits (e.g., 4 or 32 bits) by grouping D flip-flops together with a common clock or write-enable signal.

Definition 1.38. Memory: An organization of storage locations indexed by addresses.

- **Address Space:** The total number of unique locations (2^n for n address bits).
- **Addressability:** The number of bits stored at each location.
- **Hardware:** Uses an **Address Decoder** to select a location and a **Multiplexer** to route the selected data to the output.

1.6 Timing

1.6.1 Combinational Circuit Timing

Definition 1.39. The timing behavior of a combinational circuit is bounded by two parameters:

- **Propagation Delay (t_{pd}):** The **maximum** time from when an input changes until the output has reached its final value (longest path).
- **Contamination Delay (t_{cd}):** The **minimum** time from when an input changes until the output **starts** to change (shortest path).

Remark 1.40. Transistors take finite time to switch due to capacitance, resistance, and the finite speed of light. Actual delays vary based on temperature, supply voltage, circuit aging, and whether the signal is rising or falling.

Example 1.41. A **glitch** occurs when a single input transition causes multiple transitions on the output before it settles. This is caused by different paths through the logic having different delays. While often ignored in steady-state logic, glitches can be fixed by adding redundant "consensus terms" in a K-map to cover transitions between prime implicants.

1.6.2 Sequential Circuit Timing

Definition 1.42. For a flip-flop to reliably sample data at the rising clock edge, the input D must be stable within the **aperture time** ($t_a = t_{setup} + t_{hold}$):

- **Setup Time (t_{setup}):** The time **before** the clock edge that data must be stable.
- **Hold Time (t_{hold}):** The time **after** the clock edge that data must remain stable.

Important 1.43. If the setup or hold time is violated, the flip-flop may enter a **metastable** state where the output is stuck between 0 and 1 before eventually settling non-deterministically. This can cause system failure.

Definition 1.44.

- **Propagation Clock-to-Q Delay (t_{pcq}):** The **maximum** time after the clock edge until the output Q is stable.
- **Contamination Clock-to-Q Delay (t_{ccq}):** The **minimum** time after the clock edge until Q **starts** to change.

Theorem 1.45. To ensure a synchronous system operates correctly, two conditions must be met:

- **Setup Time Constraint:** The clock period T_c must be long enough for the signal to propagate through the critical path:

  $$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

  This determines the **maximum operating frequency** ($f_{max} = 1/T_{c,min}$).
- **Hold Time Constraint:** The signal must not reach the next flip-flop too quickly (shortest path):

  $$t_{ccq} + t_{cd} > t_{hold}$$

  If violated, data "shuffles" through multiple flip-flops in one cycle. This **cannot** be fixed by slowing the clock.

1.6.3 Clock Skew and Performance

Definition 1.46. Clock skew (t_{skew}) is the time difference between the arrival of the clock signal at two different flip-flops. It is caused by the physical delay of the clock distribution network.

Important 1.47. Clock skew worsens both timing constraints:

- **Setup:** $T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$ (effectively increases required period).
- **Hold:** $t_{ccq} + t_{cd} > t_{hold} + t_{skew}$ (effectively increases required min delay).

2 Processor Design

2.1 Von-Neumann Architecture

Definition 2.1. The **Von Neumann model**, proposed in 1946, defines a universal architecture for general-purpose computers consisting of five fundamental components:

- **Memory:** Stores both the program (instructions) and the data.
- **Processing Unit:** Performs arithmetic and logical operations. It contains the Arithmetic Logic Unit (ALU) and small temporary storage (registers).
- **Control Unit:** Orchestrates the execution of instructions. It contains the Program Counter (PC), which tracks the address of the next instruction, and the Instruction Register (IR), which holds the current instruction.
- **Input:** Devices to get information into the computer (e.g., keyboard).
- **Output:** Devices to get information out (e.g., monitor).

Important 2.2. Modern computing is based on two key properties of this model:

- **Stored Program:** Instructions are stored in a linear memory array alongside data. The hardware interprets a memory value as an instruction or data depending on which phase of the cycle it is in.
- **Sequential Processing:** Instructions are processed one at a time in sequence, unless a control flow instruction explicitly changes the Program Counter.

Remark 2.3. The process of executing a single instruction follows a specific sequence of phases:

- 1. **FETCH:** Obtain instruction from memory into the IR and increment the PC.
- 2. **DECODE:** Identify the opcode and determine the control signals needed.
- 3. **EVALUATE ADDRESS:** Calculate the memory address of operands if needed (e.g., for loads).
- 4. **FETCH OPERANDS:** Get the required source values from memory or registers.
- 5. **EXECUTE:** Perform the actual operation in the ALU.
- 6. **STORE RESULT:** Write the final value into the destination register or memory.

Definition 2.4. Address Space: The total number of uniquely identifiable memory locations. (LC-3: 2^{16}; MIPS: 2^{32}).

Definition 2.5. Addressability: The number of bits stored at each address (e.g., byte-addressable (8 bit) vs. word-addressable (32 bit)).

Definition 2.6. Endianness: The order of bytes in a word. **Big Endian** stores the Most Significant Byte (MSB) at the lowest address, while **Little Endian** stores the Least Significant Byte (LSB) at the lowest address.

Example 2.7. To read from memory:

1. Load the **Memory Address Register (MAR)** with the target address.
2. The memory hardware places the data from that address into the **Memory Data Register (MDR)**.

To write, both MAR (address) and MDR (data) are loaded before activating the write-enable signal.

2.2 Instruction Set Architecture (ISA)

Definition 2.8. The Instruction Set Architecture (ISA) serves as the "contract" or interface between the software (compiler/programmer) and the hardware (microarchitecture). It specifies:

- **Memory Organization:** Address space and addressability.
- **Register Set:** Number, size, and usage of registers (e.g., 8 GPRs in LC-3, 32 in MIPS).
- **Instruction Set:** Opcodes, data types, and addressing modes.

Important 2.9. Instructions are encoded as binary "words" containing:

- **Opcode:** Specifies the operation to perform (e.g., ADD, LD).
- **Operands:** Specify where to find the source data and where to put the result.

Lemma 2.10. The semantic gap refers to the distance between high-level language constructs and the ISA.

- **CISC (Complex Instruction Set Computing):** Small semantic gap; instructions do a lot of work (e.g., x86).

- **RISC (Reduced Instruction Set Computing):** Large semantic gap; simple instructions serve as primitives (e.g., MIPS, LC-3).

Definition 2.11. Mechanisms used to specify where an operand is located:

- **Immediate:** The operand is a constant within the instruction.
- **Register:** The operand is located in a general-purpose register.
- **PC-Relative:** The address is calculated as $PC + offset$.
- **Indirect:** The instruction points to a memory location that contains the *actual* address of the operand.
- **Base + Offset:** The address is calculated as $Register + offset$.

2.3 Assembly

Definition 2.12. Machine code is the binary representation (0s and 1s) processed by hardware. **Assembly language** is a human-readable representation using mnemonics (e.g., 'ADD') and labels for memory locations.

Example 2.13. High-level: $a = b + c$

- **LC-3 Assembly:** 'ADD R0, R1, R2' (R0 is destination).
- **MIPS Assembly:** 'add $s0, $s1, $s2' ($s0 is destination).

Important 2.14. In RISC (Reduced Instruction Set Computing) architectures, operands must be moved into registers before processing.

- **Load (LD/LDR/lw):** Reads data from memory into a register.
- **Store (ST/STR/sw):** Writes data from a register into memory.

Remark 2.15. The sequential execution order can be altered using:

- **Conditional Branches:** Executed only if a condition is met. LC-3 uses **Condition Codes** (N, Z, P) set by previous instructions. MIPS often compares two registers directly (e.g., 'beq $s0,s1, Label').
- **Unconditional Jumps:** Always change the PC (e.g., 'JMP' in LC-3, 'j' in MIPS).

2.4 Microarchitecture

Definition 2.16. Microarchitecture is the specific implementation of an Instruction Set Architecture (ISA) under given design constraints and goals. It describes how hardware structures, such as registers, ALUs, and buses, are organized to execute the instructions defined by the ISA.

Remark 2.17. While the ISA acts as the functional contract visible to the software, the microarchitecture is the internal logic that performs the work.

Important 2.18. A single ISA (such as x86, ARM, or MIPS) **can be implemented by many different microarchitectures**. This allows manufacturers to create different chips that run the same software but vary in performance, power consumption, or cost. The software remains unaware of whether the hardware uses a single-cycle, multi-cycle, or out-of-order execution strategy.

Remark 2.19. Microarchitects evaluate their designs based on several critical metrics:

- **Performance:** Usually measured by execution time or throughput.
- **Cost:** Often determined by the silicon die area and circuit complexity.
- **Power/Energy:** Critical for thermal management and battery-operated devices.
- **Reliability:** The ability of the system to operate correctly despite aging or environmental interference.

2.4.1 Single-Cycle Microarchitecture

Definition 2.20. In a single-cycle microarchitecture, every instruction is executed in exactly one clock cycle. This requires all phases of the instruction cycle, Fetch, Decode, Execute, Memory Access, and Write-back, to be completed within a single period of the global clock.

Important 2.21. The timing and performance of a single-cycle processor are characterized by:

- **CPI (Cycles Per Instruction):** The CPI is always 1.
- **Clock Cycle Time (T_c):** The clock period must be long enough to allow the **slowest** instruction in the ISA to complete.

Remark 2.22. In most RISC architectures, the lw (load word) instruction defines the critical path. It requires an instruction fetch, a register read, an ALU operation to calculate the address, a data memory read, and a register write. Because the clock must accommodate this long path, faster instructions like add or jump waste a significant portion of the clock cycle waiting for the next edge.

2.4.2 Multi-Cycle Microarchitecture

Definition 2.23. A multi-cycle microarchitecture breaks the execution of a single instruction into multiple, shorter clock cycles. Each instruction takes a variable number of cycles depending on its specific requirements (e.g., an add might take 3 cycles, while a lw takes 5).

Important 2.24. Multi-cycle designs allow for significant hardware reuse and require internal storage:

- **Functional Unit Reuse:** A single ALU can be used in different cycles of the same instruction (e.g., once to increment the PC and later to perform the arithmetic).
- **Microarchitectural Registers:** Since an instruction spans multiple cycles, intermediate results must be saved in registers that are not visible to the ISA, such as the Instruction Register (IR), Memory Data Register (MDR), and ALUOut.

Example 2.25. The phases of a multi-cycle instruction typically include:

- **Cycle 1 (Fetch):** Instruction is fetched and PC is incremented.
- **Cycle 2 (Decode):** Opcode is identified and registers are read.
- **Cycle 3 (Execute):** ALU performs the calculation or address computation.
- **Cycle 4 (Memory):** Data memory is accessed (only for loads/stores).
- **Cycle 5 (Write-back):** Result is written back to the register file.

Theorem 2.26. The execution time of a program is defined by the following equation:

$$\text{Time} = \text{Instruction Count} \times \text{Average CPI} \times \text{Clock Cycle Time}$$

While a multi-cycle design has a higher average CPI than a single-cycle design, it can achieve better performance because the clock cycle time (T_c) is much shorter, as it only needs to cover the delay of the single longest *stage* rather than the entire instruction.

Remark 2.27. The **control unit** in a multi-cycle processor is implemented as a Finite State Machine (FSM). It must keep track of the current step of an instruction to ensure the correct control signals (like ALUSel or MemWrite) are asserted at the correct time.

Lemma 2.28. The primary **downside** of multi-cycle designs is the **"sequencing overhead"**. Each cycle transition adds a delay due to register setup times and clock-to-Q propagation. If instructions are broken into too many cycles, this overhead can eventually negate the performance benefits of a high clock frequency.

3 Instruction Level Parallelism (ILP)

3.1 Pipelining

Definition 3.1. Pipelining is a microarchitecture implementation technique where **multiple instructions are overlapped** in execution. It functions like an assembly line, dividing the processing of an instruction into a sequence of stages so that a different instruction can be processed in each stage simultaneously.

Important 3.2. The core objective of pipelining is to increase **instruction throughput** (the number of instructions completed per unit of time). While it significantly improves throughput, it generally does not decrease the **latency** of a single instruction; in fact, the latency may increase slightly due to the overhead of pipeline registers.

Theorem 3.3. In an ideal scenario where all stages have equal delay and there are no dependencies, a pipeline with k stages provides a k-fold speedup. However, in reality, the speedup is limited by:

- **Unbalanced stages:** The clock period is determined by the slowest stage.
- **Sequencing overhead:** Pipeline registers add setup time and clock-to-Q delays.
- **Hazards:** Dependencies between instructions that prevent seamless overlapping.

3.1.1 The 5-Stage Instruction Pipeline

Definition 3.4. A standard RISC pipeline (like MIPS or LC-3) typically consists of five stages:

1. **Instruction Fetch (IF):** Fetch the instruction from memory and update the PC.
2. **Instruction Decode (ID):** Decode the instruction and read operands from the register file.
3. **Execute (EX):** Perform an arithmetic/logic operation or calculate a memory address.
4. **Memory Access (MEM):** Read from or write to the data memory.
5. **Write-Back (WB):** Write the result back into the register file.

Important 3.5. To maintain the state of different instructions in different stages, **Pipeline Registers** (IF/ID, ID/EX, EX/MEM, MEM/WB) are placed between the stages. These registers carry the data and control signals forward through the pipeline every clock cycle.

3.1.2 Pipeline Hazards and Dependencies

Definition 3.6. A hazard is a situation that prevents the next instruction in the stream from executing in its designated clock cycle.

- **Structural Hazards:** Conflicts for hardware resources (e.g., using a single memory for both instructions and data).
- **Data Hazards:** An instruction depends on the result of a previous instruction that has not yet completed.
- **Control Hazards:** The next address to fetch is unknown because a branch or jump has not been resolved.

Definition 3.7. Data dependencies are primarily of three types:

- **Flow Dependence (RAW):** Instruction j needs data produced by instruction i.
- **Anti-Dependence (WAR):** Instruction j writes to a location that instruction i reads.
- **Output Dependence (WAW):** Both instructions write to the same location.

In a standard 5-stage in-order pipeline, only RAW hazards require hardware intervention.

3.1.3 Resolving Data Hazards

Lemma 3.8. Forwarding (Bypassing) resolves many data hazards by routing the result of an instruction directly from the EX or MEM stage to the ALU input of a following instruction, bypassing the register file.

Important 3.9. Forwarding cannot resolve a **Load-Use Hazard** (where a load is immediately followed by an instruction using the loaded data). This requires a **Stall**. The hardware must:

1. Disable the PC and IF/ID register updates (holding instructions in place).
2. Flush the ID/EX register (inserting a "bubble" or NOP).

3.1.4 Control Hazards and Branching

Definition 3.10. Control hazards arise from the delay in determining the branch outcome. If the pipeline fetches the wrong instructions, it must **flush** those instructions once the branch is resolved.

Remark 3.11. Common strategies for handling branches include:

- **Static Prediction:** For example, "Always Predict Not Taken."
- **Dynamic Prediction:** Using a Branch History Table to predict based on past execution.
- **Early Branch Resolution:** Moving the branch comparison to the Decode stage to reduce the misprediction penalty.

3.1.5 Advanced Pipeline Issues

Important 3.12. Precise Exceptions: A pipelined processor must be able to stop execution on an exception-causing instruction while ensuring all prior instructions complete and no subsequent instructions have modified the architectural state. This is challenging because instructions finish at different times.

Lemma 3.13. In systems with multi-cycle functional units (e.g., a fast integer ALU and a slow divider), instructions can finish **out of order**. This "out-of-order completion" can break the sequential semantics of the ISA if an exception occurs in the middle of a stream.

Definition 3.14. Fine-Grained Multithreading: A technique where the processor switches to a different thread every clock cycle. This hides the latency of data and control hazards by ensuring that instructions currently in the pipeline are independent of one another.

3.1.6 Precise Exceptions and Interrupts

In a pipelined Architecture, not all instructions take the same amount of time in the execution stage. So what happens if a previous instruction causes an exception and the next instruction has already been written back to the register file? We need to make sure that the state of the processor is consistent with the ISA.

Definition 3.15. Unplanned changes in program exection due to internal problems are called **exceptions** while due to external events they are called **interrupts**.

Important 3.16. When an exception/interrupt occurs all previous instructions should be retired whil no later instructions should be allowed to proceed nor be retired.

Hence when the oldest instruction ready to be retired is detected to have caused an exception we have to:

- Ensure architectural state is consistent (REG-File, PC, Memory)
- Flush all younger instructions in the pipeline
- Save PC and REG
- Redirect to exception handler

Remark 3.17. In Single-Cycle machines this is easy to handle as we only have one instruction in the pipeline at a time which ensures sequential execution semantics.

Remark 3.18. Note that we could make every operation take the same amount of time in the pipeline, this would ensure sequential execution semantics but would be very inefficient.

Remark 3.19. The idea behind a **reorder-buffer** is that we complete instructions out of order but reorder them before making the results visible to the architectural state.

Definition 3.20. A **Reorder-Buffer (ROB)** is a hardware structure that stores the results of instructions as they complete. It allows instructions to be retired in-order, even if they complete out-of-order.

Lemma 3.21. Reordering-Buffer uses a valid bit in the REG to indicate whether the result is valid, if not the REG stores a pointer to the ROB entry where the result will be stored once the instruction completes (ROB also uses a valid bit to indicate readiness).

This allows us to write out of order to the REG and ensure in order retirement where we can check in-order for any exceptions.

3.2 Out of Order Execution

Remark 3.22. An instruction that uses operands of a previous instruction is called a **dependent instruction**. If such a preceding instructions result are not yet available, the dependent instruction must wait for the result to be computed, which stalls an **in-order pipeline**.

Definition 3.23. Out-of-order execution (OoO) is a technique that allows a processor to execute instructions in a different order than they appear in the program. This is done to improve performance.

Remark 3.24. OoO tolerates longer latencies by executing other independent instructions between dependent ones that would otherwise stall the pipeline. This improves instructions per cycle (IPC).

Definition 3.25. The **dataflow principle** states that the execution order of instructions should be determined by the availability of their operands rather than their program order.

Remark 3.26. Typically OoO is only done in the core. Hence instructions are **fetched and decoded in order** then they are **executed out of order** and the results are **written back in order**.

3.2.1 Mechanisms for OoO

Definition 3.27. Register-Renaming eliminates false dependencies (WAW and WAR), which occure because of the reuse of registernames, by decoupling the register names from the actual physical storage locations.

Definition 3.28. Reservation Stations (RS) are buffers that store instructions that are waiting for their operands to become available.

Definition 3.29. Tag Broadcast is a mechanism that allows the results of instructions to be broadcast to all reservation stations that are waiting for those operands by broadcasting identification tags of available results via a bus.

3.2.2 Tomasulo's Algorithm

Definition 3.30. Tomasulo's Algorithm is an algorithm for out-of-order execution that uses reservation stations and tag broadcasting to eliminate false dependencies and improve performance.

Algorithm 3.31

```
1       decode(instruction)
2
3       if (free reservation station) {
4
5               reservation station = instruction
6               replace registers with tags from Register-Alias Table (
                    RAT)
7
8               if (operands ready) {
9
10                      execute to functional unit
11                      broadcast result and update RAT
12
13                  } else {
14                      observe common data bus for tags of operands
15                  }
16
17          } else {
18              wait
19          }
```

Important 3.32. The instruction window size (how many instr. can be fetched/decoded and not yet retired) limits the scalability of Tomasulo's algorithm.

Remark 3.33. Moder processors use the already discussed reorder buffer (ROB) to ensure in order commitment of results.

Remark 3.34. The usage of a **Scheduler** with reservation stations and a **reorder buffer** can be visualized as a camel with two humps.

Remark 3.35. OoO allows for larger latency tolerance and exploitation of Irregular parallelism but comes at the cost of increased complexity and power consumption.

3.2.3 Memory Dependencies

Example 3.36. Instruction i writes to memory adress x while the instruction $i+1$ reads from x. In out of order execution it can happen that $i+1$ finishes before i which invalidates its result. Hence we need to handle memory dependencies.

Definition 3.37. Memory Disambiguation is the process of determining whether a load/store is dependent on a previous load/store.

Definition 3.38. Solutions to memory dependencies:

- **Conservative:** Assume all load/stores are dependend and stall the pipeline to ensure correctness.
- **Agressive:** Assume no load/stores are dependent and pay the extra time/-work when a false result is generated.
- **Intelligent:** Try and predict whether a load/store is dependent or not in order to minimize the penalty of false results.

Remark 3.39. Observe that the intelligent approach is a trade-off between the conservative and aggressive approaches and ensures the best performance.

Definition 3.40. Load/Store Queue (LSQ) is a queue that stores load/store results that have not yet been committed to the reorder buffer.

Definition 3.41. Store-to-Load Forwarding is a mechanism that allows the result of a store instruction to be forwarded directly to a load instruction that is waiting for that result. This works by searching the LSQ for a matching address and directly copying the data from there.

3.3 Superscalar Execution

Definition 3.42. A N-**Superscalar Processor** is a processor that can issue more than one (N) instruction per clock cycle.

Remark 3.43. Note that superscalar execution is orthogonal to OoO execution. A processor can be in-order or out-of-order and scalar or superscalar.

Important 3.44. The main challenge of superscalar processors is to find independent instructions to execute in parallel. (For example trough Forwarding, Software-Optimizations, Speciulation or Multi-Threading)

Remark 3.45. The **tradeoffs** of Superscalar execution are IPC vs Complexity.

3.4 Branch Prediction

Remark 3.46. The Goal: Branch prediction handles control dependencies. It attempts to answer the fundamental question: How do we exploit instruction-level parallelism in the presence of control flow?

Remark 3.47. The Cost of Misprediction: Branch mispredictions drastically reduce IPC in deep pipelines. Modern processors therefore target 95%+ accuracy.

Definition 3.48. There are multiple different **Branch Types**: Conditional (if, while), Unconditional (goto), Indirect (register targets), & Function calls/returns. *Conditional* and *indirect* are the hardest to predict.

Remark 3.49. Control dependencies are handled by:

- **Stalling** or **Predicting** the pipeline.
- **Delayed Branching:** Executing adjacent instructions unconditionally.
- **Multi-threading:** Executing other threads.
- **Predicated Execution:** Converting control to data-dependence.
- **Multi-path Execution:** Fetching all branches simultaneously.

3.4.1 Static Branch Prediction

Definition 3.50. Static Branch Prediction: The branch outcome is predicted strictly at compile time. Because it is static, it cannot adapt to dynamic changes in branch behavior during runtime.

Remark 3.51. The simplest strategy is to always predict the same outcome (**Always Taken** or **Always Not-Taken**). This logic requires minimal hardware but yields low accuracy.

Definition 3.52. BTFN (Backward Taken, Forward Not-Taken): Predicts backward branches as taken and forward branches as not-taken.

Remark 3.53. BTFN significantly improves prediction success by intelligently exploiting loop structures. Backward branches are typically loop iterations, which are statistically taken much more often than not.

Definition 3.54. Profile-based Prediction: Predicts the outcome based on branch behavior statistics gathered from a previous profiling run of the program.

Definition 3.55. Program-based Prediction: Uses compiler heuristics (e.g., assuming a check for a negative integer is an error path, hence not-taken).

Definition 3.56. Programmer-based Prediction: Relies on explicit pragmas or hints provided by the programmer directly in the source code.

Code Example 3.57

```
1      // Pragmas as hints for the branch predictor
2      if (likely(x)) {...}
3      if (unlikely(error)) {...}
```

3.4.2 Dynamic Branch Prediction

Definition 3.58. Dynamic Branch Prediction: The branch outcome is predicted at runtime based on the execution history.

Definition 3.59. Branch Target Buffer (BTB): A hardware cache that specifically stores the *target addresses* of recently executed branches. When a branch instruction is fetched, the BTB is accessed to provide the target address immediately, avoiding pipeline stalls while waiting for the ALU to compute the target.

Definition 3.60. Return Address Stack (RAS): Dedicated hardware for predicting function returns. It pushes the return address on a `call` instruction and pops it on a `ret`. It is highly accurate (¿95% with just an 8-entry stack) for predicting indirect returns.

Definition 3.61. Indirect Branch Predictor: Indirect branches map to multiple targets, making standard BTBs inaccurate. These are typically predicted using a dedicated history-based target predictor (e.g., indexing the BTB with the GHR

XORed with the PC).

Definition 3.62. Last-Time Predictor: Predicts the outcome of a branch to be exactly the same as the outcome of the last time that specific branch was executed.

Remark 3.63. Last-Time predictors are simple (one bit per branch). They exploit long loops (e.g. 1000 iterations) with high accuracy but perform extremely poorly on short loops (e.g. 2 iterations) or on branches that alternate heavily.

Definition 3.64. Two-Bit Counter (Bimodal Predictor): Prediction only changes after two consecutive mistakes. This introduces **hysteresis** (via cyclic Strong/Weak states), preventing the prediction from arbitrarily flipping due to a single anomalous outcome.

Remark 3.65. Two-Bit Counters significantly improve prediction accuracy compared to Last-Time predictors (yielding around 0.85-0.90 accuracy).

Definition 3.66. Two-Level Prediction: Attempts to correlate the outcome of a branch using history tracking:

- **Global History Register (GHR)**: Tracks the outcome of *recent different branches* (global context).
- **Local History Register (LHR)**: Tracks the outcome of *the same branch* in its previous iterations.

Definition 3.67. Gshare Predictor: XORs the Global History Register (GHR) with the Program Counter (PC) to index the Pattern History Table (PHT), providing branch-specific context and drastically reducing interference.

Definition 3.68. Agree Predictor & **Gskew Predictor**: Advanced techniques beyond Gshare to reduce interference. The **Agree Predictor** predicts whether the outcome will *agree* or *disagree* with a statically determined bias bit. The **Gskew Predictor** uses multiple PHTs with different hash indexing functions and resolves predictions via a majority vote.

Important 3.69. Biased Branches & Branch Filtering: Filtering heavily biased branches (e.g., 99% taken) to simpler predictors saves space in history tables for hard-to-predict branches.

Important 3.70. Tournament Predictor: In many systems, branch prediction is not done using a single predictor. Instead, multiple different predictors run simultaneously, and a meta-predictor chooses whichever predictor has been the most accurate recently.

Remark 3.71. Notable Historical & Modern Implementations:

- **Intel Pentium Pro**: First commercially successful out-of-order processor, utilizing two-level global prediction.
- **Alpha 21264**: Famously utilized a highly accurate Tournament Predictor.
- **Intel Pentium M**: Introduced specialized Loop and Jump predictors alongside traditional predictors.
- **AMD Zen 2 (and newer)**: Employs a highly accurate hybrid **Perceptron + TAGE** multi-level branch predictor.

3.4.3 Advanced Predictors

Definition 3.72. Perceptron Predictor: Uses a single-layer neural network to predict the outcome of a branch.

Remark 3.73. Efficiently implemented as an *adder tree* followed by a sign check. Achieves top-tier accuracy (0.95+) but **only learns linearly separable functions** (cannot learn XOR-type correlations).

Definition 3.74. TAGE (TAgged GEometric history length): Uses multiple global history registers of varying, geometric lengths to predict the outcome of a branch.

Remark 3.75. TAGE predictors represent state-of-the-art branch prediction. They are more complex to implement than perceptron predictors but achieve the highest structural prediction accuracy available (0.97+).

3.4.4 Alternative Control Flow Handling

Definition 3.76. Delayed Branching: The N instructions fetched after the branch (delay slots) execute unconditionally.
Advantage: Keeps pipeline full. *Disadvantage*: Ties ISA to pipeline depth; hard to fill slots with useful independent instructions.

Definition 3.77. Predicated Execution (e.g., CMOV): Eliminates branches via condition codes. An operation executes but only commits if the predicate is true.
Advantage: Eliminates mispredictions, allows larger basic blocks. *Disadvantage*: Wastes energy fetching/executing paths that are discarded.

4 Advanced Architectures

4.1 VLIW Architectures

Definition 4.1. VLIW (Very Long Instruction Word): A parallel architecture where multiple independent operations are grouped into a single "very long" instruction word and executed simultaneously by multiple **processing elements (PE)**.

Remark 4.2. This drops the need for complex dependency checking hardware (like in superscalar processors), as the **compiler is responsible** for ensuring that all operations in a single VLIW instruction are independent.

Important 4.3. The compiler needs to **statically align** the instructions (and insert noops if necessary) for them to be processed by the correct PE. This **significantly reduces portability**, as the binary code is tied to the very specific hardware architecture.

Remark 4.4. The main advantage of VLIW is its simplicity and potential for high instruction-level parallelism.

4.1.1 Compiler optimizations for VLIW

Definition 4.5. Trace Scheduling is a compiler optimization technique that attempts to find the **most frequently executed paths** (the "trace") through a program and schedules instructions along that path to maximize parallelism.

Remark 4.6. Trace scheduling is particularly effective for VLIW architectures because it allows the compiler to pack as many independent instructions as possible into a single VLIW instruction.

Definition 4.7. Superblocks expand the trace to have a single entry point, but multiple exit points.

Remark 4.8. This is done trough tail duplication which eliminates side-entries to the trace which allows for further more agressive optimization.

4.2 SIMD Architectures

Definition 4.9. Single Instruction, Multiple Data (SIMD): Same instruction applied to multiple data elements at once.

Definition 4.10. Data Parallelism: Performing the same operation on multiple data elements in parallel. (opposed to control parallelism: multiple instructions)

Definition 4.11. Array Processor: Instruction operates on multiple data elements at the same time using different spaces (PEs).

Definition 4.12. Vector Processor: Instruction operates on multiple data elements in consecutive time steps (pipelined) using the same space (PE).

Remark 4.13. Vector processing allows for deeper pipelines as it does not require handling of data dependencies or branches but requires regular data structures and a lot of memory bandwidth.

Definition 4.14. Vector Registers (VRs): Registers that store multiple data elements in parallel.

Definition 4.15. Stride: The distance between consecutive memory addresses that are accessed in parallel.

Definition 4.16. Memory Banks: Physical memory modules that can be accessed in parallel.

4.3 Systolic Array Architectures

Remark 4.17. Systolic arrays might be hard to understand intuitively. The key idea is to have a **very specific architecture** of processing elements (PEs) that are connected in a **grid- or array-like structure** and control the flow of data. This allows us to process large amounts of data in parallel without having to load data entries multiple times.

Important 4.18. Systolic arrays are **not a standard pipeline**, they allow for **non linear and multi-dimensional data flow**.

Example 4.19. To convolve a matrix with a kernel, we can use a systolic array to process the data in parallel.

Remark 4.20. The main advantage of systolic arrays is their high efficiency in terms of data movement and processing power as well as the regularity of the design.

Remark 4.21. The main disadvantage is that they are very specialized and not good at exploiting irregular parallelism.

4.4 GPU Architectures

4.4.1 SIMT

Definition 4.22. Single Instruction, Multiple Threads (SIMT) is an execution model where a **single instruction is issued simultaneously to a group of threads**, each operating on its own private data and registers. Unlike SIMD, each thread has its own program counter, so threads can diverge independently.

Remark 4.23. Both SIMT and SIMD apply the same operation to many data elements in parallel. SIMT presents each thread as an independent scalar program, the hardware then groups threads and executes them in lock-step, achieving SIMD-like efficiency without exposing vector width to the programmer.

Definition 4.24. Thread Divergence occurs when threads within the same group take different control-flow paths. The hardware executing each branch with only the relevant threads active.

4.4.2 GPU Architecture

Definition 4.25. A **GPU** is optimized for **high-throughput** parallelism by using large amounts of small arithmetic modules and allowing for long latency with enormous amounts of parallelism.

Definition 4.26. A GPU is organized hierarchically to implement SIMT:

- **Streaming Multiprocessor (SM):** The basic processing block. Each SM contains multiple CUDA cores, a large register file, shared memory, and warp

schedulers.

- **Warps:** Groups of 32 threads scheduled and executed together in lock-step.
- **Thread Blocks:** Groups of warps assigned to the same SM; threads within a block share fast on-chip shared memory.
- **Grid:** The full set of thread blocks launched for a kernel, distributed across all SMs.

Remark 4.27. Each SM runs many warps simultaneously. The warp scheduler selects a ready warp each cycle to issue instructions, hiding the latency of stalled warps without any out-of-order hardware.

4.4.3 Warps

Definition 4.28. A **warp** is the fundamental unit of thread scheduling on a GPU. It consists of threads that always execute the same instruction at the same cycle on different data (SIMT).

Remark 4.29. Warps exist to amortize control overhead: one instruction fetch, decode, and issue serves 32 threads simultaneously. Managing thousands of threads as groups of 32 is far more practical than scheduling individual threads. The warp granularity also matches the width of the underlying SIMD execution units.

4.4.4 Advantages and Disadvantages of Warp Execution

Theorem 4.30. Advantages of warp-based execution:

- **Amortized control overhead:** One instruction fetch and issue covers 32 threads.
- **Latency hiding:** With many warps per SM, the scheduler switches to a ready warp when another stalls, keeping functional units busy without out-of-order logic.
- **Memory coalescing:** When all threads in a warp access contiguous, aligned addresses, the memory controller merges requests into a single wide transaction, maximizing bandwidth.

Theorem 4.31. Disadvantages of warp-based execution:

- **Thread divergence:** Conditional branches cause warps to serialize – in the worst case only 1 of 32 lanes is active, yielding $\frac{1}{32}$ of peak throughput.
- **Uncoalesced memory access:** Scattered (non-contiguous) accesses by threads in a warp result in multiple separate memory transactions, wasting bandwidth.

- **Resource pressure:** Large register footprints reduce occupancy, limiting latency hiding.
- **Tail effects:** If the thread count is not a multiple of 32, the last warp is partially filled, wasting execution lanes.

4.4.5 Fine-Grained Multithreading and Latency Hiding

Definition 4.32. Fine-Grained Multithreading (FGMT) is a technique where the processor switches between warps **every clock cycle**. While one warp is stalled (e.g., waiting on a memory response), another ready warp executes, keeping functional units occupied.

Theorem 4.33. Latency Hiding: To fully hide a memory latency of L cycles, the SM must have enough independent warps to keep functional units busy during that latency. Informally:

$$N_{\text{warps}} \geq \frac{L}{\text{arithmetic instructions between memory accesses}}$$

Example 4.34. For example, global memory latency on modern GPUs is ≈ 600 cycles, requiring $\sim$20–30 active warps per SM to maintain full throughput.

Important 4.35. The GPU's strategy is to **tolerate latency through parallelism** rather than reduce it. This works only when there is sufficient occupancy (enough independent warps) and regular memory access patterns.

4.4.6 Memory and Performance Trade-Offs

Remark 4.36. The GPU memory hierarchy has drastically different latencies at each level:

- **Registers:** $\sim$1 cycle. Private per thread; total capacity is shared among all resident warps.
- **Shared Memory / L1:** $\sim$20–30 cycles. On-chip, per SM; shared by all threads in a block. Must be explicitly managed.
- **L2 Cache:** $\sim$100–200 cycles. Shared across all SMs.
- **Global Memory (HBM/GDDR):** $\sim$400–800 cycles. High bandwidth ($\sim$1–3 TB/s) but must be hidden by FGMT.

Definition 4.37. Memory Coalescing: The GPU merges the memory requests of all threads in a warp into one wide DRAM transaction when addresses are contiguous and aligned. Uncoalesced access (scattered addresses) results in many separate

transactions and a proportional loss of bandwidth.

Definition 4.38. Bank Conflicts occur in shared memory when multiple threads access different addresses in the same memory bank simultaneously. The accesses are serialized, reducing effective bandwidth. Avoided by careful data layout (padding, transposition).

5 Memory

Remark 5.1. Memory is the **limiting factor** in computation. It limits computation in time, uses by far most of the space and consumes most of the energy.

Remark 5.2. The robustness decreases with increased memory capacity. (Ex. Row-Hammering-Attack)

Remark 5.3. What the programmer sees is **virtual memory addresses** that are mapped to **physical memory** behind the scenes. This simplifies programming.

Definition 5.4. Ideal memory has zero latency, cost, energy and infinite capacity and bandwidth

Definition 5.5. Memory types are defined by the trade-offs of cost and performance.

- FlipFlops and Latches: (tens of transistors per bit)
- Static Random Access Memory (SRAM): (6 transistors per bit)
- Dynamic Random Access Memory (DRAM): (cheaper but slower, needs refreshing)
- Storage: (slow but cheap, 1 transistor per bit, non-volatile)

5.1 organization of Memory

Definition 5.6. Memory is an array of cells with adressing and output logic.

Remark 5.7. To increase performance memory is divided into multiple banks that allow concurrent access to independent adresses.

Definition 5.8. A typical **DRAM hierarchy** has

- Multiple-channels that leave/enter the memory controller
- Dual in-line memory modules (DIMM) which are mounted on the motherboard
- Ranks most likely front/back of card which are independently adressable bank groups with that share command and data busses.

- Chips that have multiple banks and share data bus out of the chip
- Bank can be made of multiple mats and has a global row/collumn organization

Definition 5.9. A bank hase a **row-buffer** where a whole row is loaded and from where the column bit is selected.

Remark 5.10. If a row is already in the row-buffer it is a row hit, else a row-miss.

Remark 5.11. To load a row into the row-buffer we use a row of sense amplifiers that sense the bit in the capacitors, amplify and then restore them.

Remark 5.12. On a high level,SRAM is organized similarly but uses more transistors and is faster.

5.2 Memory Technologies

Definition 5.13. Static Random Access Memory (SRAM) is a low latency memory built from two cross-coupled inverters that uses 6 transistors and does not need any refresh cycles.

Definition 5.14. Dynamic Random Access Memory (DRAM) uses only 1 transistor and a capacitor to store a bit. It is cheaper but slower and needs to be refreshed periodically.

Definition 5.15. Phase change memory (PCM) is a new non-volatile memory technology that uses chalcogenide glass in its Amorphous and Crystalline forms to store bits. The form of the glass can be changed by applying heat using current to the material.

Remark 5.16. PCM is slow, energy hungry and has limited endurance (the material degrades over time). However, it is non-volatile and very dense.

5.3 Memory Hierarchy

Remark 5.17. The reason to having multiple levels of memory are the trade-offs between cost, capacity, latency and energy consumption. We can not have fast and large memory at the same time hence we have a small fast memory (caches) and a large slow memory (RAM).

Remark 5.18. By intelligently selecting and moving chunks of data form the main memory to the caches we get nearly the performance of a large and fast memory.

Definition 5.19. Data Locality is the property of data that allows us to predict where it will be accessed next.

- **Temporal locality:** If a data item is accessed, it is likely to be accessed again soon.
- **Spatial locality:** If a data item is accessed, it is likely that nearby data items will be accessed soon.

Definition 5.20. Manually managed memory is when the programmer has to manually tell the system when to put what into which cache. These are often refered to as shared memory or scratchpads.

5.4 Cache

Definition 5.21. Caches can be more or less strict in terms of where and how data is stored (mapped).

- **Direct Mapped:** The simplest form of cache where each memory address can only be mapped to one specific cache line.
- **Set Associative:** Each memory address can be mapped to one of multiple cache lines within a specific set.
- **Fully Associative:** Each memory address can be mapped to any cache line.

Definition 5.22. Replacement Strategies are needed in case we access a cache line and it is already full. In that case we need to replace a cache line with the new one. Common strategies are:

- **Random:** Randomly select a cache line to replace.
- **FIFO:** First In First Out.

- **LRU:** Least Recently Used.
- **LFU:** Least Frequently Used.

Definition 5.23. Beladys OPT evicts the cache line that will be used furthest in the future. This is achieves optimal number of cache misses but not optimal performance because cache misses might cost different amounts of time depending on where they are in the memory hierarchy. Furthermore it would require perfect knowledge of future accesses.

Definition 5.24. Write-Policy defines what happens to data when it is written to the cache.

- **Write-Back:** The data is only written to the cache and not to the main memory.
- **Write-Through:** The data is written to both the cache and the main memory.

Remark 5.25. Cache compression if cached data is close to each other we can save a base adress and then only the offsets of the other elements in the cache line. This effecively shrinks cache space usage.

5.4.1 Prefetching

Definition 5.26. Prefetching fetches data before the program needs it to hide memory latency. It can eliminate compulsory misses.

Remark 5.27. Correctness: Prefetching mispredictions do not affect program correctness. No state recovery is needed because unused prefetched data is simply ignored.

Definition 5.28. Performance Metrics:

- **Accuracy:** Used prefetches / total sent prefetches.
- **Coverage:** Prefetched misses / all misses.
- **Timeliness:** On-time prefetches / used prefetches.
- **Bandwidth/Cache Usage:** Extra bandwidth consumed and **Cache Pollution** (displacing useful demand data).

Remark 5.29. Four Questions: What (address), When (initiation time), Where (cache level/buffer), and How (who operates it).

Definition 5.30. Hardware Prefetchers monitor accesses to find patterns:

- **Instruction-Based (Stride):** Tracks strides per individual instruction using the Program Counter (PC).
- **Memory-Region-Based:** Detects patterns across multiple instructions within a region.
- **Stream Buffers:** Special case of region-based stride prefetching where stride $N = 1$.

Definition 5.31. Software Prefetching: Programmer or compiler inserts ISA-specific instructions (e.g., x86 $PREFETCH$).

Remark 5.32. Software prefetching consumes execution bandwidth and offers limited portability as the prefetch distance depends on hardware latency.

Definition 5.33. Execution-Based Prefetching pre-executes program pieces to find misses using Runahead Execution. When the oldest instruction in the pipeline is a long-latency miss, the processor checkpoints architectural state and speculatively executes ahead to generate prefetches.

Remark 5.34. Runahead execution allows to detect future cache misses while waiting for the fetch of the current cache miss. This allows for memory level parallelism without the imense hardware of a Reorder Buffer (ROB).

Remark 5.35. Runahead exectuion can not parallelize dependent misses (Pointer-Chasing). We can use **Adress-Value Delta (AVD) Prediction** to predict the value of a memory access and prefetch the data.

Remark 5.36. Modern Techniques:

- **Pythia:** Uses Reinforcement Learning to select prefetch offsets based on rewards and current system state to increase accuracy and bandwidth efficiency.
- **Hermes:** Uses a Perceptron to predict if a load will go off-chip, allowing it to

bypass the slow on-chip cache hierarchy to find data faster.

Remark 5.37. Challenges in Multi-Core Prefetching: Prefetching of core A can interfere with Cache and Bus bandwidth or the access to the DRAM (banks, ranks, channels...) of core B. To avoid contention/pollution of cache/bus capacity and DRAM accesses, prefetching has to be throttled and coordinated across cores.

5.5 Virtual Memory

Definition 5.38. Virtual Memory is an automatic memory management technique that provides each process with the illusion of a private, very large address space.

- **Programmer Abstraction**: The programmer works with virtual addresses (VA) and does not deal with physical memory addresses.
- **Indirection**: Requires a mapping mechanism to translate virtual addresses to physical addresses (PA).
- **Isolation**: Each process has its own address space, ensuring one process cannot overwrite or access the memory of another.

Lemma 5.39. Address Translation occurs at a page granularity. A virtual address is split into a Virtual Page Number (VPN) and a Page Offset. Only the VPN is translated into a Physical Page Number (PPN), while the offset remains identical in the physical address.

Definition 5.40. A **Page Table** is a per-process data structure stored in physical memory that maintains the virtual-to-physical mappings.

- **PTE (Page Table Entry)**: Contains the mapping (PPN) and status bits such as Present, Read/Write, and User/Supervisor.
- **PDE (Page Directory Entry)**: Used in multi-level structures to point to the next level of the page table.

Remark 5.41. Multi-Level Page Tables address the size issue of single-level tables by only requiring the first-level table to be permanently in physical memory. Modern x86-64 systems use a 4-level page table structure.

Definition 5.42. The **Translation Lookaside Buffer (TLB)** is a small hardware cache within the processor that stores recently used PTEs to speed up translation and avoid frequent memory accesses to the page table.

Example 5.43. Handling TLB Misses:

- **Hardware-Managed (e.g., x86)**: A hardware Page Table Walker automatically traverses the table levels.
- **Software-Managed (e.g., MIPS)**: The hardware raises an exception, and the OS performs the walk.

Definition 5.44. The **CLOCK Page Replacement Algorithm** is an approximation of LRU used when physical memory is full. It uses a circular list and a "reference (R) bit" in the PTE. The algorithm clears R-bits during traversal and replaces the first page found with an R-bit of 0.

5.5.1 Memory Protection

Remark 5.45. Access Control is enforced during translation by storing permission bits in the PTE.

- **Privilege Levels**: x86 uses four protection rings (Ring 0 for Kernel/Supervisor, Ring 3 for Applications/User).
- **Enforcement**: If an access type (Read/Write) or privilege level violates the bits in the PDE/PTE, the hardware generates an exception.

Remark 5.46. RowHammer Security Issue: This hardware failure allows an attacker to induce bit flips in adjacent memory rows. An attacker can use this to flip bits in a PTE, potentially gaining write access to their own page table and subsequently the entire physical memory.

Lemma 5.47. Cache-VM Interaction: Systems must decide if caches are virtual or physical.

- **Virtually Addressed**: Faster access but suffers from **Homonyms** (same VA, different PA across processes) and **Synonyms** (different VAs, same PA).
- **Physically Addressed**: Avoids aliasing but requires translation (TLB) before the cache can be fully accessed.