DDCA Speedrun - Shivram Sambhus @ cs.shivi.io

1. Number Systems & Encodings

The foundation for representing all data and instructions in a digital system. It's how we speak the computer's language.

Binary & Hexadecimal

- **Binary (Base-2):** The native language of digital logic, using only digits 0 and 1 (bits).
- **Hexadecimal (Base-16):** A human-friendly shorthand for binary. Uses digits 0-9 and A-F.
- **Key Relationship:** 1 hex digit represents exactly 4 bits. This makes conversion trivial. E.g., $1011_2 = B_{\text{hex}}$.

Two's Complement

The standard method for representing signed integers. It's chosen because it makes arithmetic simple.

- **Sign Bit:** The Most Significant Bit (MSB) tells you the sign: 0 for positive, 1 for negative.
- **Negation:** The core trick. To get $-X$ from X, you **invert all the bits and add 1**.

Example (4-bit):

- $+3 = 0011_2$
- To get -3: Invert (1100) and add 1, giving 1101_2.
- **Range (N bits):** -2^{N-1} to $2^{N-1} - 1$. Notice one more negative number than positive.

Key Intuition: This system works because `A - B` can be computed as `A + (-B)`. This means the hardware for addition and subtraction is identical, saving silicon area.

2. Logic & Transistors

Digital systems are built on a hierarchy of abstraction, from physics to algorithms. We start at the logical-physical boundary: the transistor as a switch and the Boolean algebra that governs it.

2.1. Introduction & Core Principles

Hierarchy of Abstraction

Problem Domain → **Algorithm** → **System Software** → **ISA** (Instruction Set Arch.) → **Microarchitecture** → **Logic** → **Devices** → **Physics**. This course bridges Logic, Microarchitecture, and the Instruction Set Architecture (ISA).

Von Neumann Model

The classic blueprint for a computer. It has three main parts:

- **Computation:** The brain, an Arithmetic/Logic Unit (ALU) that does the math.
- **Communication:** The nervous system, buses that move data around.
- **Storage:** Memory that holds both **instructions** and **data** in the same place (this is a key feature!).

Key Design Tradeoffs

Computer Architecture is an art of balancing conflicting goals:

- **Performance:** How fast can we get work done? (Speed, throughput, latency).
- **Energy Efficiency:** How much work per watt of power? Critical for everything from phones to data centers.
- **Cost:** How much silicon area does it take? How complex is the design?

You can't maximize all three. A faster design usually costs more power and area.

2.2. Transistors: The Switches

MOSFETs

Metal-Oxide-Semiconductor Field-Effect Transistors are the fundamental building block. Think of them as tiny, perfect, electrically-controlled switches.

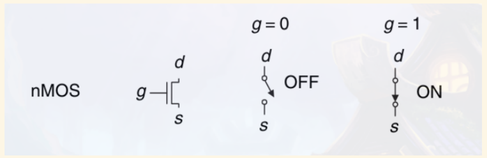
- **Terminals:** Gate (G), Source (S), Drain (D).
- **Logic:** Voltage at the Gate controls if a path exists between Source and Drain.
- We call high voltage (V_{dd}) a logic '1', and low voltage (GND) a logic '0'.

nMOS Transistor

Operation:

- Gate HIGH ($V_G = V_{\text{dd}}$) → Switch is **ON** (a closed circuit).
- Gate LOW ($V_G = \text{GND}$) → Switch is **OFF** (an open circuit).

Key Characteristic: It's great at **pulling the output down** to 0. It is a **strong '0' passer**. It's bad at passing a '1' (it's "weak").

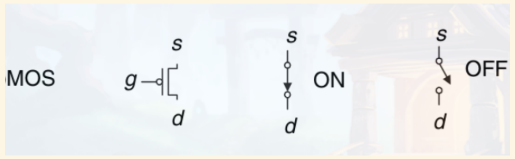


pMOS Transistor

Operation: It's the opposite of an nMOS.

- Gate LOW ($V_G = \text{GND}$) → Switch is **ON**.
- Gate HIGH ($V_G = V_{\text{dd}}$) → Switch is **OFF**.

Key Characteristic: It's great at **pulling the output up** to 1. It is a **strong '1' passer**. It's bad at passing a '0'.



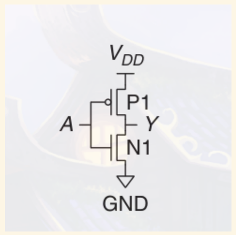
2.3. CMOS Logic

CMOS: Complementary MOS

The Big Idea: Pair up nMOS and pMOS transistors to get the best of both worlds.

- **Pull-Up Network (PUN):** Built with pMOS transistors. Connects the output to V_{dd} to create a strong '1'.
- **Pull-Down Network (PDN):** Built with nMOS transistors. Connects the output to GND to create a strong '0'.
- **Key Design Rule:** The PUN and PDN are **complementary**. For any given input, only one network is active at a time. This prevents a direct short from power to ground and is why CMOS is so power-efficient.

CMOS Inverter (NOT Gate)

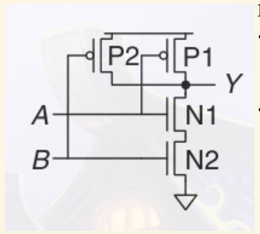


The simplest CMOS gate.

- Input A=0: pMOS is ON, nMOS is OFF. The output Y is pulled up to V_{dd} ('1').
- Input A=1: pMOS is OFF, nMOS is ON. The output Y is pulled down to GND ('0').

Final Output: $Y = \overline{A}$

CMOS NAND Gate



Logic: $Y = \overline{AB}$

- **PUN (pMOS):** In **Parallel**. The output is pulled up to '1' if A is '0' **OR** B is '0'.
- **PDN (nMOS):** In **Series**. The output is pulled down to '0' only if A is '1' **AND** B is '1'.

CMOS NOR Gate

Logic: $Y = \overline{A + B}$

- **PUN (pMOS):** In **Series**. The output is pulled up to '1' only if A is '0' **AND** B is '0'.
- **PDN (nMOS):** In **Parallel**. The output is pulled down to '0' if A is '1' **OR** B is '1'.

Universal Gates: NAND and NOR are special because any logic function can be built using only one of them. This simplifies manufacturing.

From NAND:

- NOT: $\overline{A} = A \text{ nand } A$
- AND: $AB = (A \text{ nand } B) \text{ nand } (A \text{ nand } B)$
- OR: $A + B = (A \text{ nand } A) \text{ nand } (B \text{ nand } B)$

From NOR:

- NOT: $\overline{A} = A \text{ nor } A$
- AND: $AB = (A \text{ nor } A) \text{ nor } (B \text{ nor } B)$
- OR: $A + B = (A \text{ nor } B) \text{ nor } (A \text{ nor } B)$

2.4. Boolean Algebra

Notation & Key Theorems

The mathematical rules for digital logic. AND: AB. OR: $A + B$. NOT: $\overline{A}$.

Theorem	Expression
Identity	$A + 0 = A, A \cdot 1 = A$
Complement	$A + \overline{A} = 1, A\overline{A} = 0$
Commutative	$A + B = B + A$
Associative	$A + (B + C) = (A + B) + C$
Distributive	$A(B + C) = AB + AC$
Distributive Dual	$A + BC = (A + B)(A + C)$
Uniting	$AB + A\overline{B} = A$
Uniting Dual	$(A + B)\left(A + \overline{B}\right) = A$
De Morgan's	$\overline{A + B} = \overline{A}\,\overline{B}$
De Morgan's	$\overline{AB} = \overline{A} + \overline{B}$
Consensus	$AB + \overline{A}C + BC = AB + \overline{A}C$

De Morgan's Trick: "Break the bar, change the operator." Visually, this is like pushing an inverter bubble through a gate.

Consensus Intuition: The term BC is redundant because if $BC = 1$, its job is already covered by either the AB term (if A=1) or the $\overline{A}C$ term (if A=0). It's a freebie.

Logic Simplification

- **Goal:** Use fewer/simpler gates to implement a function. This makes the circuit faster, smaller, and lower power.
- **Karnaugh Maps (K-maps):** A visual grid method for applying the Uniting theorem. You group adjacent 1s in powers of 2 to find the simplest terms.
- **"Don't Cares" (X):** These are input conditions that will never happen, or where we don't care about the output. Use them as wildcards in K-maps to make your groups of 1s bigger, which leads to simpler logic.

Canonical Forms

Standard ways to write any Boolean function.

- **Minterm (m_i):** An AND term that is true for exactly one row of the truth table.
- **Maxterm (M_i):** An OR term that is false for exactly one row of the truth table.
- **Sum-of-Products (SOP):** ORing together all the minterms where the function's output is '1'. Corresponds to a two-level AND-OR logic structure.
- **Product-of-Sums (POS):** ANDing together all the maxterms where the function's output is '0'. Corresponds to a two-level OR-AND logic structure.

3. Combinational Circuits

*These circuits are "memoryless." Their output at any instant depends **only** on the current input values. They do the active work of computation.*

3.1. Design & Common Components

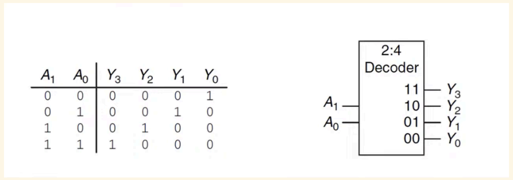
Design Process

1. **Specification:** Understand the problem. Write a truth table or equation.
2. **Simplify:** Use K-maps or Boolean algebra to find the simplest SOP/POS form.
3. **Implement:** Draw the circuit diagram using logic gates.

Decoder (n-to-2^n)

- **Function:** Takes an n-bit binary number and activates exactly one of its 2^n outputs. This is called a "one-hot" output.
- **Logic:** Each output Y_i is simply the minterm m_i for that input combination.

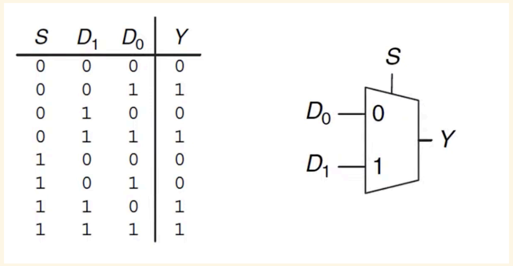
- **Application:** Selecting a memory location (address decoding), or figuring out which instruction to execute.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Multiplexer (MUX, 2^n-to-1)

- **Function:** A data selector. It uses an n-bit "select" signal to choose one of 2^n data inputs and route it to a single output.
- **Logic:** For a 2-to-1 MUX: $Y = \overline{S}D_0 + SD_1$.
- **Analogy:** Think of a train track switch, directing one of several incoming trains to a single destination track.
- **Application:** The workhorse of datapath control. Used everywhere to select which data goes to the ALU, which value is written to a register, etc.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Full Adder (FA)

- **Function:** The basic building block for addition. It adds three 1-bit inputs (A, B, and a Carry-in from the previous bit, C_{in}) and produces a 2-bit result: a Sum bit (S) and a Carry-out bit (C_{out}).
- **Logic:**
  - $S = A \oplus B \oplus C_{\text{in}}$ (XOR of all inputs)
  - $C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}}$ (Majority function)

Ripple-Carry Adder (N-bit Adder)

- **Structure:** To add N-bit numbers, you chain N Full Adders together.
- **Problem:** It's slow. The carry bit has to "ripple" from the least significant bit (LSB) all the way to the most significant bit (MSB). The delay grows linearly with N. Faster adders (like Carry-Lookahead) fix this but are more complex.

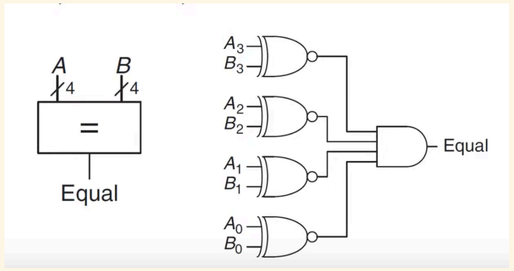
Arithmetic Logic Unit (ALU)

- **Function:** A combo-logic block that performs a whole set of operations (like ADD, SUB, AND, OR, SLT). A control signal (opcode) tells the ALU which operation to perform. It's the core computational unit of a processor.
- **Structure:** Internally, it has all the circuits for its functions, and a big MUX at the end to select the correct result based on the opcode.

3.2. More Combinational Blocks

Comparator

- **Function:** Compares two binary numbers, A and B, and outputs signals like A > B, A = B, A < B.
- **Logic (1-bit):**
  - $A = B : \overline{A \oplus B}$ (XNOR)
  - $A > B : A\overline{B}$
- **Application:** The core of branch instructions (beq, bne).

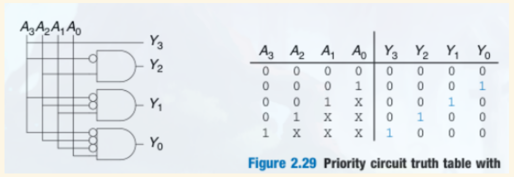


Tri-State Buffer

- **Function:** A special kind of gate with a data input (A), an output (Y), and an enable (E).
- **States:**
  - E=1: It acts like a normal wire (Y = A).
  - E=0: The output is electrically disconnected. This is called the **High-Impedance (Hi-Z)** state. It's not 0 or 1, it's just floating.
- **Application:** Essential for creating shared data buses. Multiple devices can connect to the same wire, but only one is enabled to "drive" the bus at a time. All others are in Hi-Z.

Priority Encoder

- **Function:** The opposite of a decoder. It takes multiple input lines and outputs the binary index of the highest-priority input that is currently active.
- **Application:** Used in interrupt controllers to decide which interrupt is the most urgent to handle.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with

Programmable Logic Array (PLA)

- **Function:** A general-purpose logic chip. It has a programmable AND plane followed by a programmable OR plane.
- **Intuition:** You can "burn in" the connections to create any set of SOP functions you want. It's flexible but less efficient than a custom-designed chip. Shared product terms between outputs are its main advantage.

4. Sequential Circuits

*These circuits have **memory**. Their output depends not only on current inputs but also on the **state** (values stored from past inputs). This is how we build registers, memory, and state machines.*

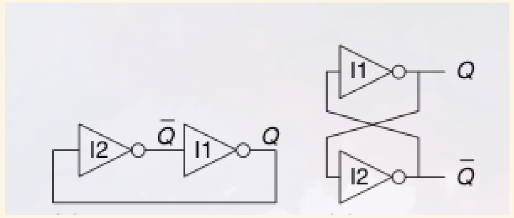
4.1. Storage Elements

Hierarchy of Storage

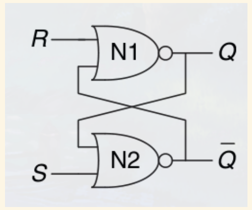
Fastest/Smallest: Latches & Flip-Flops (the building blocks). **On-Chip:** SRAM (Static RAM). Used for Caches and Register Files. Fast but takes up space. **Main Memory:** DRAM (Dynamic RAM). Slower, but very dense and cheap. This is your computer's "RAM".

The Bistable Element

- **Structure:** Two inverters whose outputs feed back to each other's inputs.
- **Property:** This loop has two stable states. It will happily hold either a '0' or a '1' forever. This is the simplest possible memory, but we can't control it.



SR Latch (Set-Reset)



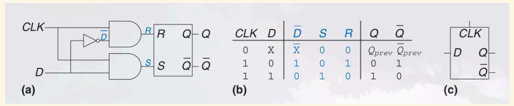
- **Function:** A controllable 1-bit memory.
- **Logic (NOR gates):**

S	R	Action
0	0	Hold (store Q)
0	1	Reset (Q->0)
1	0	Set (Q->1)
1	1	Forbidden! (Q=Q'=0)

- **Problem:** The forbidden state $S = R = 1$ is ambiguous and must be avoided.

Gated D Latch

- **Function:** Fixes the SR latch problem. It has one data input (D) and an enable input (G, sometimes called CLK).
- **Behavior:**
  - **G=1 (Transparent):** The latch is "open". The output Q simply follows the input D.
  - **G=0 (Opaque):** The latch is "closed". The output Q holds whatever value it had when G went low.
- **Key Property:** It is **level-sensitive**. The output can change at **any time** while the gate signal is high. This can be problematic.

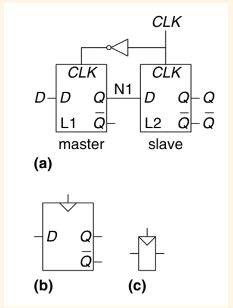


CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

4.2. Flip-Flops & Synchronous Systems

D Flip-Flop (DFF)

- **Function:** The heart of all synchronous (clocked) digital systems.
- **Behavior:** It only captures the value of the D input at a very specific moment: the **edge** of the clock (CLK) signal.
  - **Positive-edge-triggered:** Captures D on the clock's low-to-high ($0 \rightarrow 1$) transition.
- **Key Property:** It is **edge-triggered**. This is the crucial difference from a latch. It ensures that state changes across the entire system happen at the same instant, preventing data from "racing" through logic uncontrollably.



Flip-Flop Timing Parameters

- t_{su} **(Setup Time):** D must be stable for some amount of time **before** the clock edge arrives.
- t_h **(Hold Time):** D must remain stable for some amount of time **after** the clock edge.
- t_{cq} **(Clock-to-Q Delay):** How long it takes for the new value to appear on the Q output after the clock edge.

Metastability: If you violate setup or hold time, the flip-flop enters a "confused" state. Its output may oscillate or settle to an invalid voltage level for an unpredictable amount of time. This is a major source of failure in digital systems.

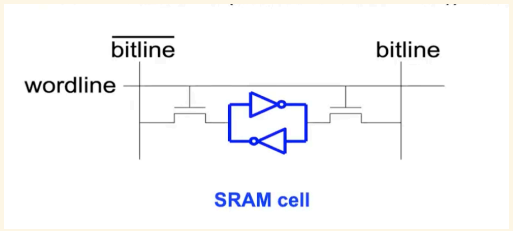
4.3. Registers, Memory, and FSMs

Registers

- **Structure:** Simply a group of N D Flip-Flops that share a common clock signal.
- **Function:** To store an N-bit word of data. Examples include the Program Counter (PC) and general-purpose registers.

Static RAM (SRAM) Cell

- **Structure (6T Cell):** Basically a bistable element (latch) with two extra "access" transistors.
- **Control:** A **Word Line (WL)** acts like an enable for an entire row of cells.
- **Data I/O:** Data is read from or written to the cell via two wires: a **Bit Line (BL)** and its inverse ($\overline{\text{BL}}$).



Finite State Machines (FSM)

The design pattern for any sequential circuit. **Components:**

1. **State Register (DFFs):** Stores the machine's **current state**.
2. **Next-State Logic (Combinational):** A block of logic that decides the **next state** based on the **current state** and the **inputs**.
3. **Output Logic (Combinational):** A block of logic that computes the machine's **outputs**.

Feature	Moore Machine	Mealy Machine
Output Logic	Depends **only** on the **current** state. Outputs are registered.	Depends on **current state** AND **current inputs**. Outputs are combinational.
Output Timing	Changes synchronously with the clock. Stable for the entire cycle.	Can change asynchronously if inputs change, potentially causing glitches.
Complexity	May need more states.	Often needs fewer states.

5. Verilog & Timing

Verilog is a Hardware Description Language (HDL). It's how we describe hardware to tools that simulate and synthesize it into real gates. Timing analysis proves it will work at speed.

5.1. Verilog Fundamentals

HDL vs. Programming Language

- **Key Difference:** An HDL **describes** concurrent, parallel hardware. A programming language describes a sequence of steps. All `always` blocks and `assign` statements in Verilog are "executing" at the same time.
- **Synthesis:** The process of turning descriptive HDL code into a gate-level circuit.

Module & Data Types

- `module`: The basic container for a circuit.
- `wire`: Represents a physical wire. It has no memory and must be continuously driven by something (like a gate output or an `assign` statement).
- `reg`: Represents a variable that can hold its value. **Crucially, `reg` does NOT always mean a flip-flop!** It becomes a flip-flop only if assigned within a clocked `always` block. Otherwise, it might become a wire or, dangerously, a latch.

always Blocks

- **For Combinational Logic:** `always @*` (or `always @(all inputs)`). The synthesis tool will build combinational logic. **Rule:** You must assign a value to every output in every possible path to avoid unintentionally creating a latch.
- **For Sequential Logic:** `always @(posedge clk)`. This construct explicitly tells the synthesis tool to create flip-flops.

Blocking vs. Non-Blocking

This is a common point of confusion.

- **Blocking (=):** Assignments execute sequentially, one after another, within the block. Use for **combinational logic**. `b = a; c = b;`
- **Non-Blocking (<=):** All right-hand sides are evaluated first, and then all assignments happen "at the same time" at the end of the step. Use for **sequential logic**. `b <= a; c <= b;`

Golden Rule: Use `=` in `always @*`. Use `<=` in `always @(posedge clk)`.

5.2. Timing Analysis

Combinational Delays

- t_{pd} **(Propagation Delay):** The **maximum** time it takes for an output to settle after an input changes.
- t_{cd} **(Contamination Delay):** The **minimum** time before an output might start to change.
- **Critical Path:** The path through the combinational logic with the longest total propagation delay. This path determines the maximum possible clock frequency of the entire system.

Setup Time Constraint

The data signal must arrive at a flip-flop's D input at least t_{su} **before** the next clock edge.

$$T_{clk} \geq t_{pcq} + t_{pd\_comb} + t_{su} - t_{skew}$$

Intuition: The clock period (T_{clk}) must be long enough for the data to leave the first flip-flop (t_{pcq}), travel through the slowest logic path (t_{pd_comb}), and arrive at the second flip-flop with enough time to spare (t_{su}).

Hold Time Constraint

The data at a flip-flop's D input must not change until t_h **after** the clock edge. This prevents the new data from "contaminating" the current operation.

$$t_{ccq} + t_{cd\_comb} \geq t_h + t_{skew}$$

Intuition: The data from the **next** cycle, traveling through the **fastest** possible logic path (t_{cd_comb}), must not arrive too early and corrupt the data being latched in the **current** cycle. Notice this is independent of the clock speed.

6. ISA & Processor Basics

The Instruction Set Architecture (ISA) is the contract between software and hardware. The microarchitecture is the implementation that fulfills that contract.

6.1. ISA Concepts

ISA vs. Microarchitecture

- **ISA (What):** The programmer's view of the machine. It defines the set of instructions, registers, memory addressing modes, etc. It's an **abstraction**.
- **Microarchitecture (How):** The specific hardware implementation (datapath, control, pipeline). It's a **concrete design**. Different microarchitectures can implement the same ISA (e.g., Intel and AMD both make x86 processors).

Instruction Categories

1. **Operate:** Arithmetic and logical operations (ADD, AND, XOR). Usually work on registers.
2. **Data Movement:** Load data from memory into registers (e.g., `lw` - load word) and store data from registers to memory (e.g., `sw` - store word).
3. **Control Flow:** Instructions that change the Program Counter (PC).
   - **Unconditional:** Jumps (`j`, `jr`).
   - **Conditional:** Branches (`beq` - branch if equal, `bne` - branch if not equal).
   - **Procedure Calls:** `jal` (jump and link) jumps to a function and saves the return address.

Endianness

How bytes are ordered within a multi-byte word in memory.

- **Big-Endian:** The "big end" (most significant byte) comes first, at the lowest memory address.
- **Little-Endian:** The "little end" (least significant byte) comes first. (Used by x86).

6.2. Basic Microarchitectures

Single-Cycle Microarchitecture

- **Concept:** Every instruction takes exactly one, very long, clock cycle to execute.
- **Pros:** Extremely simple design and control logic.
- **Cons:** The clock speed is dictated by the **slowest possible instruction** (usually a load from memory). This makes it very inefficient, as a simple ADD instruction has to wait for the same long cycle as a load.

Multi-Cycle Microarchitecture

- **Concept:** Break each instruction down into multiple steps (like IF, ID, EX, MEM, WB), where each step takes one shorter clock cycle.
- **Pros:** Allows for a much faster clock. Simpler instructions (like ADD) can finish in fewer cycles than complex ones (like LW). Allows hardware to be reused across cycles (e.g., one ALU can be used for address calculation and later for arithmetic).
- **Cons:** Requires a more complex controller, typically a Finite State Machine (FSM), to keep track of which step the current instruction is in.

7. Pipelining

Pipelining is the key technique for improving processor throughput. It works by overlapping the execution of multiple instructions, like an assembly line.

7.1. Pipelining & Hazards

5-Stage MIPS Pipeline

1. **IF:** Instruction Fetch (get instruction from memory)
2. **ID:** Instruction Decode & Register Read
3. **EX:** Execute / Address Calculation
4. **MEM:** Memory Access (for loads/stores)
5. **WB:** Write Back (write result to register file)

Pipeline Registers: These are registers (made of DFFs) that sit between each stage (e.g., IF/ID, ID/EX). They hold all the data and control signals for an instruction as it moves down the pipeline, isolating the stages from each other. **Analogy:** Think of doing laundry. While one load is in the dryer (EX), another can be in the washer (ID), and you can be folding a third (WB). Pipelining doesn't make any single load of laundry faster (latency), but you finish more loads per hour (throughput).

Structural Hazards

- **Problem:** Two instructions need the same hardware resource at the same time. Classic example: a unified memory is needed by IF (for instruction fetch) and MEM (for data access) simultaneously.
- **Solution:** Duplicate the resource (e.g., have separate Instruction and Data Caches) or stall the pipeline.

Data Hazards (RAW)

- **Problem (Read-After-Write):** An instruction needs a result from a previous instruction that hasn't been written back to the register file yet.

```
add $r1, $r2, $r3
sub $r4, $r1, $r5
```

- **Solution 1: Stall.** The simplest solution. Just pause the pipeline and insert "bubbles" (NOPs) until the data is ready. This is slow.
- **Solution 2: Forwarding/Bypassing.** The best solution. Add extra wires (datapaths) to send the result directly from where it's created (e.g., ALU output) back to where it's needed (e.g., ALU input) without waiting for the WB stage. This is a "shortcut" for data.

Control Hazards (Branches)

- **Problem:** When we fetch a branch instruction, we don't know which way it will go (taken or not-taken) until the EX stage. But by then, we've already fetched several instructions from the wrong path!
- **Misprediction Penalty:** The number of cycles wasted flushing the incorrect instructions and fetching the correct ones.
- **Solutions:**
  1. **Stall:** Wait until the branch outcome is known. Safe but very slow.
  2. **Predict:** Guess the outcome and speculatively fetch instructions. If you guess right, no penalty! If you guess wrong, you flush and pay the penalty.
  3. **Delayed Branch:** An old ISA trick. The instruction in the "delay slot" right after a branch is **always** executed, regardless of the branch outcome. Shifts the burden to the compiler to find useful work to put there.

Analogy: Branch Prediction as a Fork in the Road You're running a race on a path that splits. Instead of stopping to read the map (resolving the branch), you guess and take the left path. If you're right, you've saved time. If you're wrong, you must run all the way back to the fork and take the right path. This wasted time is the **misprediction penalty**.

Branch Prediction

- **Goal:** Guess the branch outcome with high accuracy to keep the pipeline full.
- **1-bit Predictor:** Remembers the last outcome (Taken/Not-Taken). Fails twice on a simple loop (at the last iteration and the first time back).
- **2-bit Saturating Counter:** Uses four states (Strongly NT, Weakly NT, Weakly T, Strongly T). It takes two wrong predictions in a row to flip from one strong state to the other. This adds **hysteresis** and correctly predicts typical loops with only one misprediction.
- **Correlating Predictors:** Use the history of recent branches to predict the current one. A **Branch History Register (BHR)** stores the outcomes of the last N branches, and this pattern is used to look up a prediction in a **Pattern History Table (PHT)**.

8. Advanced Microarchitecture

Modern techniques to find and exploit Instruction-Level Parallelism (ILP), pushing performance beyond what a simple pipeline can do.

8.1. Superscalar & Out-of-Order Execution

Superscalar Execution

- **Concept:** Fetch, decode, and execute **more than one** instruction per clock cycle.
- **Hardware:** Requires a "wider" pipeline: wider fetch, multiple decoders, multi-ported register file, and multiple parallel execution units (ALUs, FPUs, etc.).
- **Goal:** Achieve an Instructions Per Cycle (IPC) greater than 1.

Out-of-Order (OoO) Execution

- **Concept:** Don't stall the whole pipeline for one slow instruction. Let later, independent instructions run ahead while the stalled one waits for its data. Instructions are executed based on data availability, not program order.
- **Key Enabler: Register Renaming**
  - **Problem:** Sometimes instructions aren't truly dependent, they just happen to use the same register name (a "false" dependency like WAR or WAW).
  - **Solution:** The hardware renames the architectural registers (e.g., `$r1`) to a larger pool of hidden **physical registers**.
  - **Mechanism:** A **Register Alias Table (RAT)** keeps track of the mapping. Every time an instruction writes to a register, it gets a fresh physical register, breaking the false dependency.

Core OoO Components (Tomasulo-like)

1. **In-Order Front-End:** Fetches and decodes instructions in program order.
2. **Reservation Stations (RS) / Issue Queue (IQ):** A buffer where renamed instructions wait for their operands to become available.
3. **Common Data Bus (CDB):** When an execution unit finishes, it broadcasts the result and the physical register tag on the CDB. Reservation stations are "snooping" this bus to see if the data they're waiting for is now available.
4. **Reorder Buffer (ROB):** This is the key to maintaining correctness. It tracks instructions in their original program order. Instructions **commit** (update the architectural state, like registers or memory) from the head of the ROB. This ensures **in-order retirement**, even though execution was out-of-order. This also enables **precise exceptions**.

Analogy: Out-of-Order as a Deli Counter

- You place your order (instruction). The cashier gives you a ticket number (`ROB` entry).
- Sandwich makers (functional units) look at a board of pending orders (`RS`).
- A chef shouts "Turkey is ready!" (`CDB` broadcast). Any sandwich maker waiting for turkey grabs it.
- A maker sees your order now has all its ingredients. They make your sandwich (execute instruction).
- You only pick up your food when your number is called, in the order you paid (`in-order commit`), even if someone behind you in line ordered a simpler sandwich and got it made first.

9. Parallel & Specialized Architectures

Moving beyond single-core performance by exploiting other forms of parallelism, often for specific types of problems.

9.1. VLIW, SIMD, and Systolic Arrays

VLIW (Very Long Instruction Word)

- **Concept:** The **compiler** is the smart one. It finds multiple independent operations and packs them into one giant instruction.
- **Hardware:** The hardware is very simple. It just peels off the operations from the VLIW and sends each to its own functional unit.
- **Tradeoff:** Very simple, low-power hardware, but totally dependent on a clever compiler. If the compiler can't find parallel work, it must fill the instruction with NOPs, wasting space.

Systolic Arrays

- **Concept:** A grid of simple, identical Processing Elements (PEs). Data flows rhythmically through the grid, with each PE doing a small calculation and passing the result to its neighbor.

Analogy: Systolic Array as an Assembly Line Think of a car assembly line. Each worker (a PE) does one simple task (e.g., add a bolt) and passes the car to the next worker. Data flows through the PEs just like the car flows down the line. It's extremely efficient for repetitive tasks like matrix multiplication. This is the core idea behind Google's TPUs.

SIMD (Single Instruction, Multiple Data)

- **Concept:** One instruction operates on many pieces of data at the same time.
- **Vector Processor:** A specific type of SIMD machine. It has large **vector registers** that hold many data elements. A single instruction like `VADD` will add all the elements of two vector registers.
- **Modern CPUs:** Use short-vector SIMD extensions (like SSE, AVX on x86, or NEON on ARM) to get a taste of this performance boost.

9.2. GPUs

GPU Architecture

- **Execution Model (SIMT):** Single Instruction, Multiple Threads. This is a subtle but important evolution of SIMD.
- **Warp/Wavefront:** The hardware groups 32 threads together into a **warp**. All threads in a warp execute the same instruction in lockstep. The warp is the fundamental unit of scheduling.
- **Hierarchy:** A GPU contains many **Streaming Multiprocessors (SMs)**. Each SM has many simple cores, a shared local memory, and schedulers that manage a large number of warps.

Latency Hiding in GPUs

- **The GPU Superpower:** GPUs don't try to reduce memory latency with big caches like CPUs do. Instead, they **hide** it with massive multithreading.
- **Mechanism:** An SM has many more warps than it has execution units. If one warp stalls (e.g., waiting for data from slow DRAM), the scheduler instantly swaps it out for another warp that is ready to run. This keeps the execution units constantly busy, hiding the long memory delay.

Branch Divergence

- **Problem:** What happens if threads within a single warp take different paths at a branch? They can't execute different instructions because the whole warp must execute the same instruction.
- **Solution (Masking):** The hardware serializes execution. It executes the 'if' path, disabling (masking) the threads that didn't take it. Then it executes the 'else' path, disabling the threads that took the 'if' path. This gets the job done but kills SIMT efficiency.

10. The Memory System

The memory hierarchy is a system of caches designed to bridge the massive speed gap between the fast CPU and slow, large DRAM.

10.1. DRAM & Memory Organization

The Memory Problem

- The "Memory Wall": Processor speeds have improved much faster than memory speeds. Getting data from memory is often the biggest performance bottleneck.
- Data movement is also very expensive in terms of energy.

Memory Technology

- **SRAM (Static RAM):** Built from latches (6 transistors). Very fast, but not dense and expensive. Used for on-chip caches.
- **DRAM (Dynamic RAM):** Built from a single transistor and a tiny capacitor. The charge on the capacitor represents the bit. Very dense and cheap, but the charge leaks away, so it needs to be periodically **refreshed**. Used for main memory.

DRAM Bank Operation

The key to understanding DRAM performance is the row buffer.

1. **ACTIVATE (ACT):** Select a row and copy its entire contents into a special register called the **row buffer**. This is the **slow** step.
2. **READ/WRITE (CAS):** Select a column from the already-active row buffer. This is very **fast**. A subsequent access to the same open row is called a **row buffer hit**.
3. **PRECHARGE (PRE):** Close the current row, writing it back to the main array. You must do this before you can open a **different** row in the same bank. An access that requires a PRECHARGE and then an ACTIVATE is a **row buffer conflict** and is very slow.

10.2. Caching

Caching Principles

- **Goal:** Create the illusion of a single, large, fast memory.
- **Locality of Reference:** The principle that makes caches work.
  - **Temporal Locality:** If you access something, you're likely to access it again soon.
  - **Spatial Locality:** If you access something, you're likely to access its neighbors soon.
- **Operation:** When the CPU needs data, it checks the cache first. A **hit** means the data is there (fast). A **miss** means it's not; we must fetch a **cache block/line** from main memory (slow).

Cache Organization (Placement)

Where can a memory block be placed in the cache?

- **Direct-Mapped:** A block can go in exactly one spot. The `index` bits of the address determine the cache line. Simple and fast, but suffers from **conflict misses** if two frequently used addresses map to the same line.
- **N-Way Set-Associative:** A compromise. The `index` bits determine a **set**, and the block can go in any of the N "ways" within that set.
- **Fully-Associative:** A block can go anywhere. Requires comparing the address tag with every tag in the cache, making it expensive.

Cache Management Policies

- **Replacement (on a miss to a full set):**
  - **LRU (Least Recently Used):** Evict the block that hasn't been accessed for the longest time. Good performance, but hard to implement perfectly for high associativity.
  - **Random:** Simple and avoids worst-case behavior.
- **Write Policy (on a write hit):**
  - **Write-Through:** Write to both the cache and main memory immediately. Simple, but uses a lot of memory bandwidth.
  - **Write-Back:** Write only to the cache and set a **dirty bit**. The block is only written back to memory when it's evicted. More efficient.
- **Write Miss Policy:**
  - **Write-Allocate:** On a write miss, fetch the block into the cache first, then write to it. Usually paired with write-back.

Cache Miss Types (The 3 Cs)

1. **Compulsory (Cold):** The very first time a block is accessed. Unavoidable.
2. **Capacity:** The cache is too small to hold the program's entire working set.
3. **Conflict:** Two blocks repeatedly kick each other out because they map to the same set in a set-associative or direct-mapped cache.

10.3. Advanced Caching & Prefetching

Software Cache Optimization

- **Loop Interchange:** Reorder nested loops to access data in the same order it's laid out in memory (e.g., row-major vs. column-major access).
- **Blocking/Tiling:** Break a large problem into smaller "tiles" that fit in the cache. This maximizes data reuse (temporal locality).

Multi-Core Caching & Coherence

- **The Problem:** If two cores have their own private caches, what happens if they both cache the same memory location? Core A might write to it, but Core B's copy is now stale/invalid.

Analogy: Cache Coherence as a Shared Google Doc When multiple people edit a Google Doc, the system has to make sure everyone sees the latest version. This is cache coherence. An **invalidate protocol** is like the system telling everyone else, "Your copy is stale, please reload." An **update protocol** is like changes appearing on everyone's screen in real-time.

- **Solution: Coherence Protocols** like MESI (Modified, Exclusive, Shared, Invalid) are hardware protocols that use snooping or directories to ensure all cores see a consistent view of memory.

Prefetching

- **Goal:** Fetch data into the cache **before** the CPU actually asks for it, turning a potential miss into a hit.
- **Hardware Prefetchers:**
  - **Stride Prefetcher:** A simple but effective prefetcher that looks for fixed-stride memory access patterns (e.g., A, A+N, A+2N, ...).
  - **Stream Prefetcher:** A special case that detects sequential memory access (stride = 1).

11. Virtual Memory

The powerful abstraction layer that gives each process its own private address space, isolates processes from each other, and uses main memory as a cache for the disk.

11.1. VM Concepts

Core Idea

- The OS gives each process a huge, private **virtual address space**.
- The hardware (Memory Management Unit, or MMU) and OS work together to translate these **virtual addresses** into **physical addresses** in DRAM.
- This indirection enables protection, sharing, and allows physical memory to act as a fully-associative cache for pages stored on disk.

Mechanism: Page-Based VM

- **Virtual Page (VP):** A fixed-size block of the virtual address space (e.g., 4KB).
- **Physical Frame (PF):** A fixed-size block of physical memory, the same size as a page.
- **Page Table:** A per-process data structure, managed by the OS, that stores the mapping from Virtual Page Numbers (VPNs) to Physical Frame Numbers (PFNs).
- **Page Table Entry (PTE):** An entry in the page table containing:
  - **Valid Bit:** Is this page currently in physical memory?
  - **Physical Frame Number (PFN):** If valid, where is it?
  - **Permission Bits:** Read, Write, Execute, User/Supervisor.
  - **Dirty Bit:** Has this page been written to?
- **Page Fault:** An exception that occurs when a program tries to access a page whose PTE has the valid bit turned off. The OS takes over, finds the page on disk, loads it into a physical frame, updates the PTE, and resumes the program.

11.2. TLB & Protection

Page Table Challenges & Solutions

1. **Size:** Page tables can get enormous. A 64-bit address space would need a petabyte-sized page table.
   - **Solution: Multi-Level Page Tables.** Break the VPN into multiple pieces to index a hierarchy of tables. This saves space as you only need to allocate tables for regions of the address space that are actually in use.
2. **Speed:** Multi-level tables mean every memory access now requires multiple extra memory accesses just to do the translation!
   - **Solution: Translation Lookaside Buffer (TLB).**

Translation Lookaside Buffer (TLB)

- **What it is:** A small, fast, highly-associative hardware **cache for page table entries (PTEs)**.
- **Operation:**
  - **TLB Hit:** The VPN is in the TLB. The translation is done in a single cycle. Fast!
  - **TLB Miss:** The VPN is not in the TLB. This triggers a **page table walk** (either by hardware or a software exception handler) to fetch the PTE from the page table in main memory. The PTE is then loaded into the TLB.

The Full Address Translation Flow

For a given virtual address:

1. Check the TLB for the translation.
2. **If TLB Hit:** Great! You have the Physical Address. Now check the L1 cache.
3. **If TLB Miss:** A page walker (hardware or OS) traverses the page table in memory to find the PTE.
4. **If PTE is valid:** Load the PTE into the TLB. Go back to step 1.
5. **If PTE is invalid (Page Fault):** Trap to the OS. The OS pages the data in from disk, updates the PTE, and resumes the program. The instruction will then restart from step 1.

Cache Interaction (VIPT)

- **Virtually Indexed, Physically Tagged (VIPT):** The standard design for L1 caches.
- **How it works:** The cache lookup and the TLB lookup happen **in parallel**. The virtual address provides the `index` to select a set in the cache. At the same time, the TLB translates the virtual tag to a physical tag. When both are done, the physical tag from the TLB is compared with the physical tag stored in the cache to determine a hit or miss. This is faster than waiting for the full translation before starting the cache access.

Memory Protection

- **How it works:** VM is the foundation of modern OS security.
- **Isolation:** Since each process has its own page table, it's impossible for one process to generate a virtual address that translates to another process's private physical memory.
- **Permissions:** On every single memory access, the hardware (MMU) checks the R/W/X and User/Supervisor permission bits in the TLB/PTE. Any violation (e.g., user code trying to write to a read-only page) triggers a protection fault exception, and the OS terminates the offending program.