Summer of Science
Final Report 2025

Computer Architecture

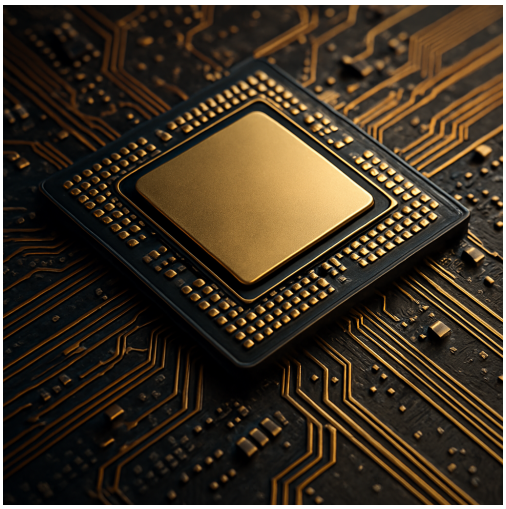
Keshaw Ranjan
Roll No: 24B0908
Mentor: Mitul Tandon

July 20, 2025

Contents

Chapter 1

Introduction: The Digital Odyssey



Computer architecture represents the fascinating intersection of mathematics, physics, and engineering where abstract Boolean algebra manifests as tangible computation. This report chronicles the journey from fundamental digital storage elements made using the simple logic gates (OR, AND, NOT, etc) to sophisticated superscalar processors and Graphical Processing Units (GPUs).

We begin with *flip-flops* and *latches*—the atomic units of digital memory that trap electrons to remember bits. From these primitive cells, we construct *registers* and *combinational circuits* that perform logic operations without memory. This naturally evolves into *sequential logic* where circuits gain the ability to remember past states, culminating in *finite state machines* (FSMs)—the digital brains that orchestrate complex behaviors.

The *Arithmetic Logic Unit* (ALU) then emerges as the computational heart of processors, executing operations from simple addition to bitwise comparisons. Our exploration of the *MIPS architecture* reveals how Reduced Instruction Set Computing (RISC) principles enable elegant hardware implementations. Then, we dissect two execution paradigms: the straightforward *single-cycle* and the hardware-efficient *multi-cycle* datapaths. With this, we start the journey of **PARALLELISM** to make the processor more and more efficient. We discuss the *pipelined* processors and its complication which gives way for *Out-Of-Order* execution which can execute instruction in an order different from how it is fed.This is merged with the superscalar technique, in which multiple instructions can be fed concurrently to form the processors that are used today. A brief overview and inner workings of the GPU is then given highlighting how processors can be made much powerful in a fixed domain. Then we shift ways discussing about *computer memories*, its working and importance in a strive to get an ever fast processor. Finally we conclude the journey showing the current workings in this field.

Chapter 2

Flip-Flops and Latches: Memory's Foundation

2.1 The SR Latch: Digital Memory's Genesis

At its core, digital memory stems from cross-coupled logic gates. The SR (Set-Reset) latch, built from two NOR gates, creates the simplest bistable memory cell. This elegant circuit demonstrates how feedback loops create memory:

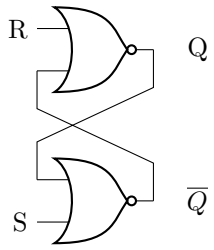


Figure 2.1: SR latch implementation using NOR gates

Behavioral Analysis:

- *S=1, R=0:* Forces Q=1 regardless of previous state. The NOR gate with S input outputs 0 when S=1, which becomes the input to the other NOR gate. With R=0, this causes Q to be set to 1.
- *S=0, R=1:* Resets Q=0 through similar gate interactions.
- *S=0, R=0:* Maintains previous state through the feedback loop—the first instance of digital memory.
- *S=1, R=1:* Creates a forbidden state where both outputs become 0, violating the fundamental requirement that Q and $\overline{\text{Q}}$ should be complements.

This circuit embodies memory through feedback—the output reinforces its own state. The forbidden state (S=R=1) creates a quantum-like uncertainty where Q and $\overline{\text{Q}}$ both become 0, a condition known as *metastability* that can cause unpredictable behavior in digital systems.

To prevent the metastable state, a little modification is made by which the input R and S are made dependent such that $R = \overline{Q}$, which is known by the name of **D latch**.

2.2 Evolution to Edge-Triggered Flip-Flops

While latches are level-sensitive (transparent when enabled), flip-flops introduce *edge-triggering*—updating state only at clock transitions. The master-slave D flip-flop achieves this through two cascaded D latches:

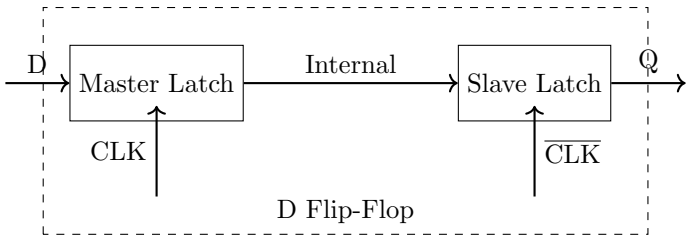


Figure 2.2: Master-slave D flip-flop architecture

Timing Analysis:

1. When CLK=0: Master latch transparent, slave latch opaque
2. Rising CLK edge: Master latches input, slave becomes transparent
3. CLK=1: Master opaque, slave shows latched value

Edge triggering becomes important because of these:
1. *Synchronization:* All state changes occur predictably at clock edges, enabling coordinated system behavior.
2. *Hazard Prevention:* Input glitches between edges are ignored, preventing erratic state changes.
3. *Pipelining Foundation:* Creates clean timing boundaries essential for staged instruction processing.

Without edge-triggering, modern processors couldn't achieve gigahertz clock speeds—signals would ripple chaotically through circuits like marbles in a maze. The disciplined timing of flip-flops transforms digital chaos into computational symphony.

Chapter 3

Combinational Logic

Combinational circuits produce outputs *instantaneously* from current inputs, with no memory of past events. They form the computational fabric of digital systems, transforming data through Boolean algebra.

3.1 Multiplexers

A multiplexer (MUX) selects one of N inputs to route to its output, controlled by $\log_2 N$ selection lines. The 4:1 MUX implements the Boolean logic:

$$\text{Output} = \overline{S_1 S_0} D_0 + \overline{S_1} S_0 D_1 + S_1 \overline{S_0} D_2 + S_1 S_0 D_3$$

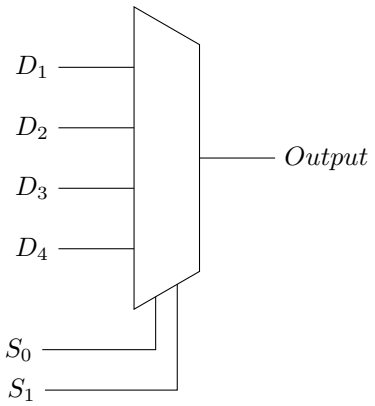


Figure 3.1: 4:1 multiplexer symbol and gate-level implementation

In CPUs, multiplexers perform critical routing:

- *Register File:* Select between multiple register outputs
- *ALU Inputs:* Choose between immediate values or register contents
- *Data Paths:* Route results to memory, registers, or output

They act as the digital equivalent of railroad switches, directing data flows through the processor's intricate pathways.

3.2 Decoders

Decoders convert compact binary codes into encoded outputs, the most common of which is **one-hot**. A 2:4 decoder activates exactly one of four outputs based on a 2-bit input:

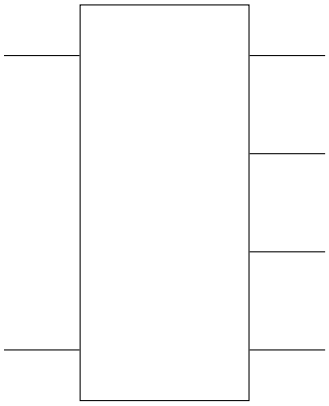


Figure 3.2: 2:4 decoder operation

In instruction decoding, a 6-bit opcode decoder identifies whether an instruction is `add`, `lw`, or `beq`, activating corresponding control signals. This is the processor's "interpreter" that translates binary codes into hardware actions.

3.3 Adders

Arithmetic operations form the computational core of processors. Two primary adder architectures demonstrate the speed-complexity tradeoff:

Ripple-Carry Adder

The simplest but slowest approach—carry propagates sequentially:

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

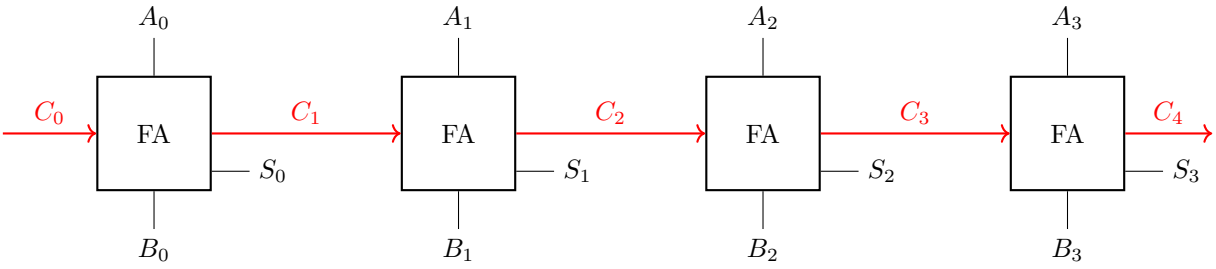


Figure 3.3: 4-bit ripple-carry adder

Delay: $2n$ gate delays for n-bit addition. For 32 bits, this becomes prohibitively slow.

Carry-Lookahead Adder (CLA)

Precomputes carry bits using generate ($G_i = A_i B_i$) and propagate ($P_i = A_i \oplus B_i$) terms:

$$C_{i+1} = G_i + P_i \cdot C_i$$

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

CLA reduces delay to $\log_2 n$ gates—essential for high-speed ALUs. The hardware cost is justified in processors where addition is the most frequent operation.

Chapter 4

Sequential Logic

Sequential circuits marry combinational logic with memory elements (flip-flops), enabling state-dependent behavior. Their outputs depend on both current inputs *and* past history—a fundamental requirement for complex computation.

4.1 The Sequential Paradigm

Consider a vending machine: it dispenses a drink only after receiving coins *and* remembering cumulative value. This requires:

1. **State Storage:** Flip-flops holding current status (e.g., coins inserted)
2. **Next-State Logic:** Combinational circuit computing new state
3. **Output Logic:** Generating actions based on state

Mathematically:

$$\text{Next State} = f(\text{Current State}, \text{Inputs})$$

$$\text{Outputs} = g(\text{Current State}, \text{Inputs}) \quad \text{(Mealy Machine)}$$

or

$$\text{Outputs} = g(\text{Current State}) \quad \text{(Moore Machine)}$$

4.2 Finite State Machines

FSMs formalize sequential logic into states and transitions. We'll explore the finite machine through the example of combinational lock:

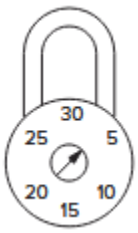


Figure 4.1: Combinational Lock: A finite state machine

Suppose the lock opens when rotated in the order: R13, L22, R3, R13. The state diagram of the lock will look like:

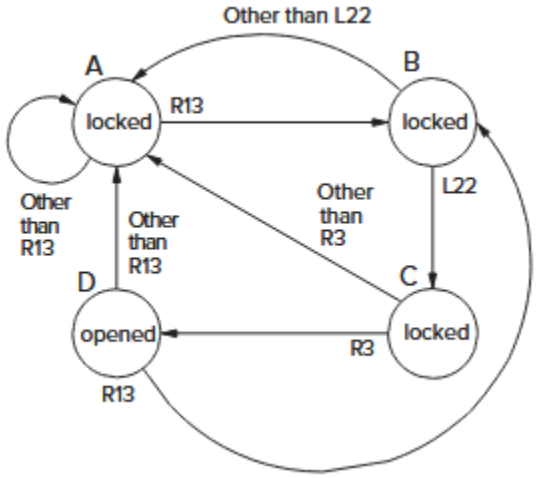


Figure 4.2: State diagram of the combination lock

This FSM demonstrates how complex behaviors emerge from simple states and transitions. Each state can be encoded in binary, stored in flip-flops, with transitions governed by combinational logic.

FSMs form the backbone of processor control units, managing instruction execution through states like Fetch, Decode, Execute, etc. They transform static silicon into dynamic computation.

Chapter 5

Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the computational engine at the heart of every processor. A 32-bit MIPS ALU handles over 20 operations, but we'll focus on its core capabilities.

5.1 A model ALU

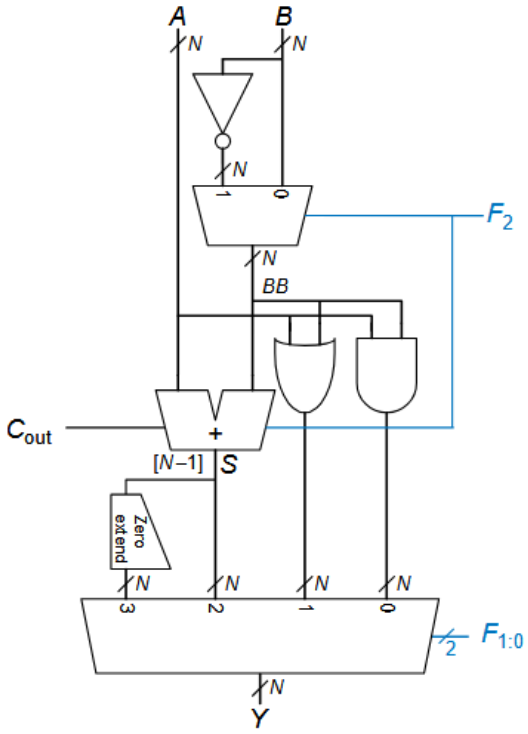


Figure 5.1: 3-bit ALU

This ALU is controlled by three bit control instruction. For example, when the control bit is 010, the **add** operation is performed, and similarly.

5.2 Flag Generation

The ALU outputs status flags critical for decision-making:

- *Zero:* NOR of all result bits (1 if result=0)
- *Overflow:* $C_{in} \oplus C_{out}$ for signed arithmetic (detects invalid results)
- *Carry:* C_{out} for unsigned operations
- *Negative:* MSB of result (sign bit)

This compact unit executes everything from $1 + 1$ to bitwise comparisons—proving complexity emerges from simplicity. The ALU is the processor's calculator, transforming data through mathematical and logical operations at incredible speeds.

Chapter 6

MIPS ISA

6.1 RISC vs. CISC: The Architectural Divide

The processor design landscape is dominated by two philosophies:

Feature	**RISC (MIPS, ARM, RISC-V)**	**CISC (x86, 68000)**
Philosophy	Hardware simplicity enables high clock speeds	Complex instructions reduce code size
Instructions	Fewer (50-100), fixed length	Many (200+), variable length
Execution	Mostly single-cycle	Often microcoded (multi-cycle)
Registers	Many (32+ GPRs)	Fewer (8-16)
Memory Access	Load/store only (reg-reg architecture)	Memory-to-memory operations
Pipelining	Easier to implement	Complex due to variable instructions
Examples	MIPS R3000, ARM Cortex, Apple M1	Intel Core i9, AMD Ryzen

Table 6.1: RISC vs. CISC architectural comparison

RISC prioritizes simplicity in design and operation—a philosophy that dominates modern architectures (ARM, RISC-V). While this eases the work for the processor designer, it increases and complexes the work for compiler-designer as they have to handle all the complex operations using only the basic operations. The MIPS architecture, developed in the 1980s, became the textbook example of RISC principles.

6.2 MIPS Instruction Set Anatomy

All instructions are 32-bit, with three uniform formats enabling simple decoding:

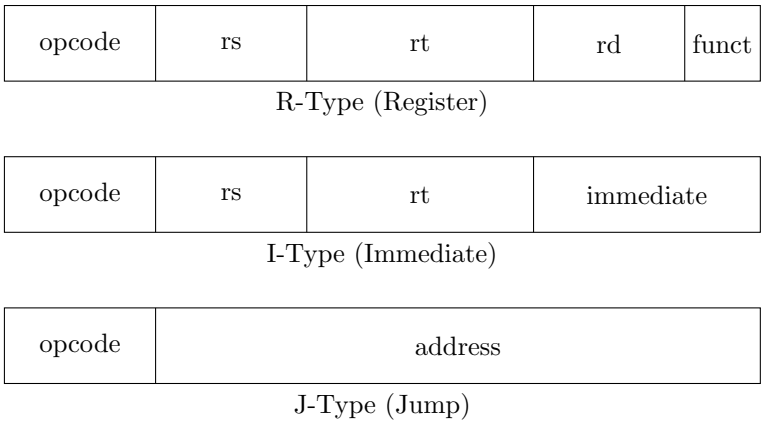


Figure 6.1: MIPS instruction formats

6.3 Instruction Gallery with Encodings

```
# Arithmetic/Logical (R-Type)
add $s0, $s1, $s2   # 000000 10001 10010 10000 00000 100000
sub $t0, $t1, $t2   # 000000 01001 01010 01000 00000 100010
and $t3, $t4, $t5   # 000000 01100 01101 01011 00000 100100

# Immediate Operations (I-Type)
addi $s0, $s1, 42   # 001000 10001 10000 0000000000101010
ori  $t0, $t1, 0xFF  # 001101 01001 01000 0000000011111111

# Memory Access (I-Type)
lw $t0, 8($s1)      # 100011 10001 01000 0000000000001000
sw $t2, 12($s3)     # 101011 10011 01010 0000000000001100

# Control Flow (I-Type and J-Type)
beq $t0, $t1, Label # 000100 01000 01001 0000000000001100
j Loop              # 000010 00000000000000000011110010
```

Listing 6.1: Essential MIPS instructions

The elegance of MIPS lies in its regularity: all instructions are the same length, source registers are always in the same bit positions, and arithmetic operations always target a register. This simplicity enables efficient pipelined implementations.

Chapter 7

Single-Cycle Datapath

Combining all the single components (ALU, decoder, multiplexer and memory) gives the basic framework of the processor. All that differs then is the implemenation using these components. One of the primitive one in the processor journey is the single-cycle processor. The single-cycle processor executes every instruction in one clock cycle. All hardware is active simultaneously, with the clock period set by the slowest instruction.

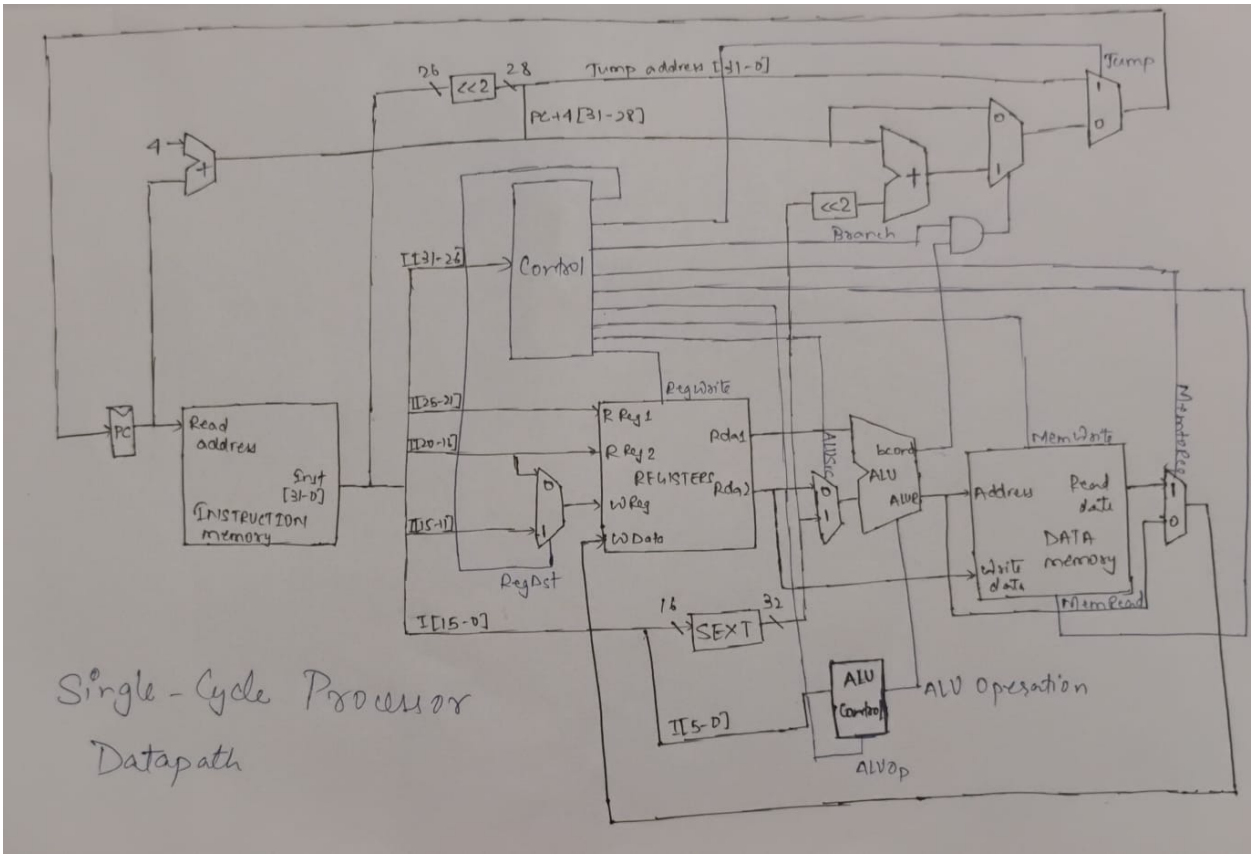


Figure 7.1: Single-cycle MIPS datapath

7.1 Component Inventory

- **Instruction Memory:** Stores program code

- **Register File:** 32 registers with 2 read ports, 1 write port
- **ALU:** Performs arithmetic/logic operations
- **Data Memory:** Stores variables and data
- **Sign Extender:** Converts 16-bit immediate to 32-bit
- **Control Unit:** Generates signals based on opcode

7.2 Instruction Walkthroughs

`add $s0, $s1, $s2` (R-Type)

1. **IF:** Instruction fetched from address in PC
2. **Decode:** Control unit recognizes R-type (opcode=000000)
3. **Register Read:** Values of $s1 and $s2 read from register file
4. **ALU Operation:** ALU adds two values (ALUOp=10, funct=100000)
5. **Write Back:** Result written to $s0 (RegWrite=1)

`lw $t0, 4($s1)` (I-Type)

1. **IF:** Instruction fetched
2. **Decode:** Control sets MemRead=1, RegWrite=1
3. **Address Calc:** ALU adds $s1 and sign-extended 4
4. **Memory Read:** Data memory accessed at computed address
5. **Write Back:** Value written to $t0

`beq $t0, $t1, Label`

(I-Type)

1. **IF:** Instruction fetched
2. **Decode:** Control sets Branch=1
3. **Compare:** ALU subtracts $t0 - $t1, checks Zero flag
4. **Branch Calc:** If Zero=1, PC = PC + 4 + (imm ¡¡ 2)

Critical Weakness: As the clock period is same for all the instructions, it is decided by the slowest one. `lw` requires memory access ≈ 200ps + ALU 100ps + register access 50ps = 350ps, while `add` completes in 150ps. This inefficiency wastes significant time and the processor remains idle for most of the part of the cycle for the fast instructions. It was clearly inefficient, although it was simple to design.

Chapter 8

Multi-Cycle Datapath

The inefficiency of single cycle datapath was soon realized and it paved way for other designs where the clock cycle was broken down into quantas small enough to accomodate the fastest instruction in a multiple of a single quanta (the quanta, then, become the clock cycle). It was called the multi-cycle approach.

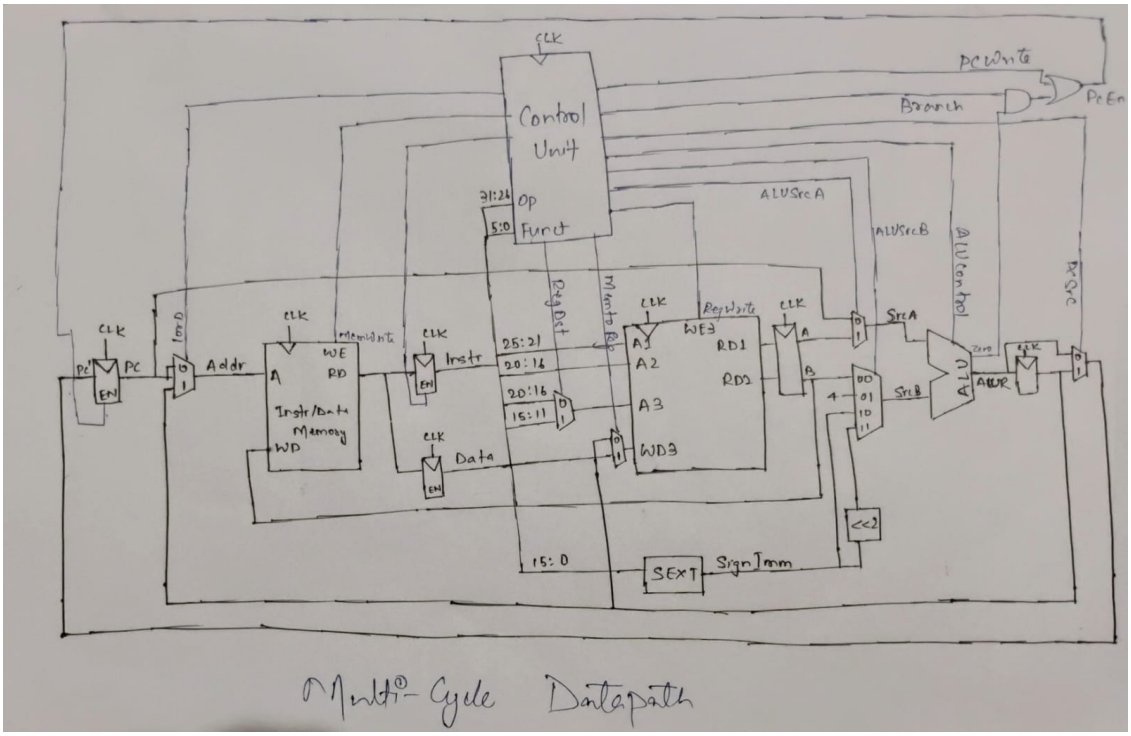


Figure 8.1: Multi-cycle MIPS datapath with shared functional units

8.1 Execution Stages

Stage	Cycle	Action	Key Control Signals
Instruction Fetch (IF)	1	Read instruction, PC=PC+4	IorD=0, IRWrite=1
Instruction Decode (ID)	2	Read registers, compute branch target	ALUSrcA=0, ALUSrcB=01
Execute (EX)	3	ALU operation per instruction	Varies by instruction
Memory Access (MEM)	4	Load/store memory if needed	MemRead/MemWrite
Write Back (WB)	5	Write to register file	RegWrite

8.2 Instruction Timelines

- `add`: IF → ID → EX → WB (4 cycles)
- `lw`: IF → ID → EX → MEM → WB (5 cycles)
- `beq`: IF → ID → EX (3 cycles)

8.3 Performance Analysis

Advantages: - Faster clock possible (shorter critical path) - Hardware reuse reduces complexity

Disadvantages: - Higher CPI increases latency - Complex control FSM required

The multi-cycle design can be said to be the architectural equivalent of carpooling—saving hardware resources by sharing components across time.

Chapter 9

Pipelined Processor

With the demand for more computational speed, a new dimension of '*PARALLELISM*' was explored. In as early as the 1960s, inspired by the assembly line in industries, the idea of *pipelining* came into effect in processors like **IBM 7030 Stretch**. Pipelining overlaps instruction execution, achieving near 1-instruction/cycle throughput. The classic MIPS pipeline has five stages:

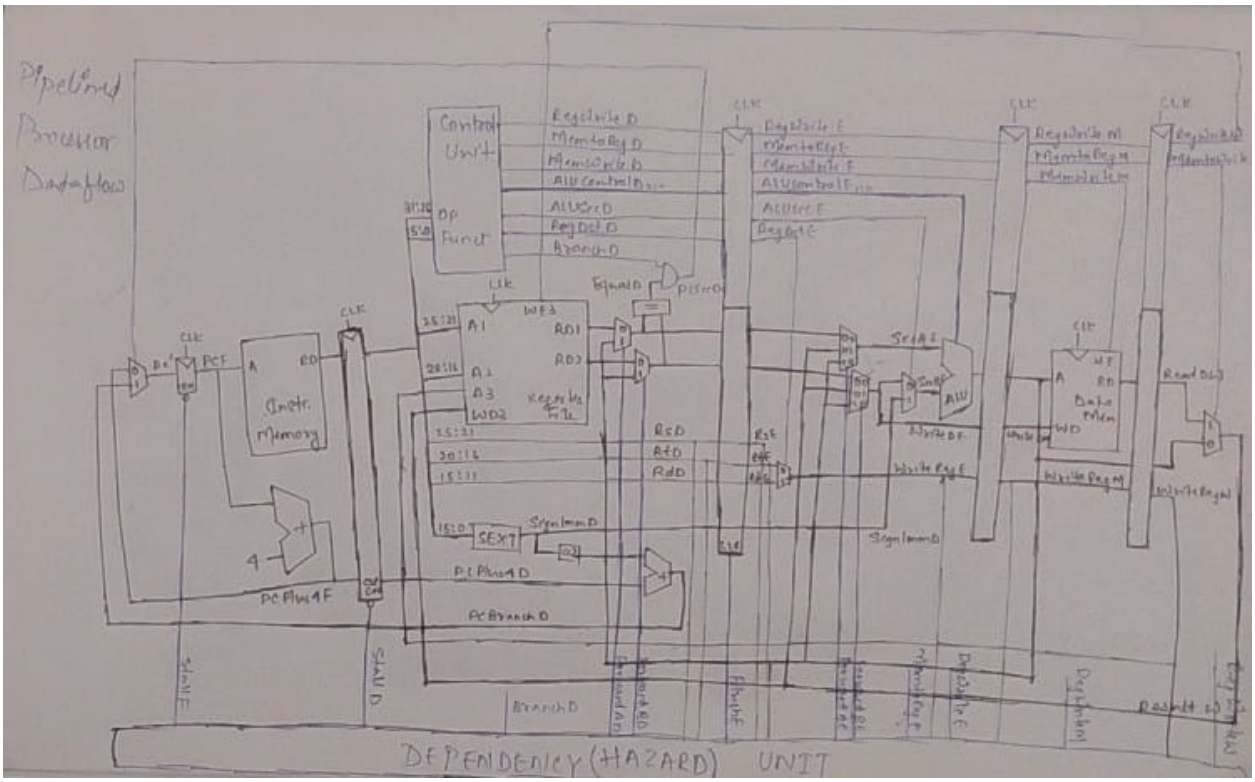


Figure 9.1: MIPS pipelined datapath with pipeline registers

9.1 The Five-Stage Pipeline

Stage	Abbr.	Key Actions
Instruction Fetch	IF	Read instruction, PC+4
Instruction Decode	ID	Read registers, sign-extend immediate
Execute	EX	ALU operation, branch decision
Memory Access	MEM	Load/store data
Write Back	WB	Write to register file

9.2 Ideal Execution Timeline

Cycle	1	2	3	4	5
Instr1	IF	ID	EX	MEM	WB
Instr2		IF	ID	EX	MEM
Instr3			IF	ID	EX
Instr4				IF	ID
Instr5					IF

Throughput: 1 instruction/cycle after pipeline fill.

9.3 Hazards: Pipeline Bubbles

A big limitation in the continuous flow of instructions in the pipeline is dependency (commonly known as hazards) between the instructions. When a conditional instruction was fetched, the pipeline had to halt before it can decide whether it will take the branch or not! Three types of dependecies are observed:

- Data Dependencies
- Control Dependencies
- Structural Dependencies

9.3.1 Data Hazard

Dependencies (or hazards) from data occurs when one instruction is dependent on data produced from an instruction before it. It arises on the register level and can be of three types -

1. **WAR** (Write After Read)
2. **WAW** (Write After Write)
3. **RAW** (Read After Write)

The dependencies *WAR* and *WAW* are mainly due to the absence of enough registers or bad compiling and are called *name* or *false-dependencies*.

Read After Write (RAW)

```
1 add $s0, $s1, $s2
2 sub $t0, $s0, $t1  # Needs $s0 not yet written
```

It can be solved to some extent by directly making the data available (if it is available) without going through all the stages of the pipeline. It is called *Data Forwarding (Bypassing)*.

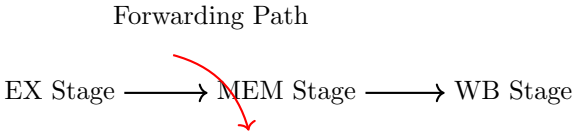


Figure 9.2: Forwarding data from EX/MEM to ALU inputs

9.3.2 Control Hazard

Also known as *Branch Hazard*, it occurs mainly when enough information is not available on whether to take the branch or not!

```
beq $t0, $t1, Target
add $s1, $s2, $s3  # Fetched before branch resolved
```

Some ways to deal with are:
1. *Stall:* Insert bubbles until branch resolved 2. *Branch Prediction:* Predict not taken (flush if wrong) 3. *Delayed Branch:* Execute next instruction regardless

9.3.3 Structural Hazard

It occurs due to the limitation on the number of components (registers, ALU, etc) available, when multiple instructions try to access the same component. One way to mitigate (for memory access) is to fetch data and instruction from different memory (cache). It is more efficient as data and instructions are handled differently by the processor.

Chapter 10

Tomasulo's Algorithm: Out-Of-Order Execution

10.1 False Dependencies

The hazards (dependencies) in the pipelined processors was blocking a non-dependent instruction from executing which otherwise would have executed without any halt. Of the three types of dependencies WAR (Write-after-Read), RAW (Read-after-Write) and WAW (Write-after-Write), only RAW is a true one. The other two, called name or false-dependencies can be removed by using a different register for writing.

10.2 The Algorithm

In 1967, Robert Tomasulo gave an algorithm to overcome the false dependencies and make execution faster.

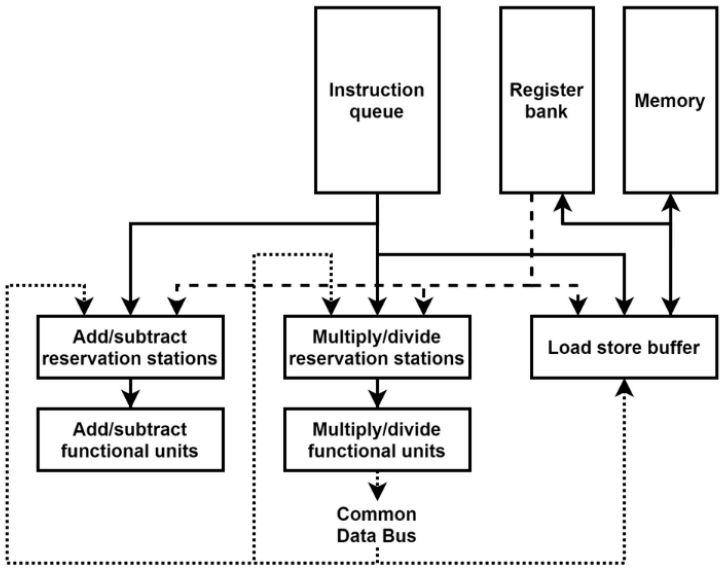


Figure 10.1: Tomasulo's Algorithm

The main component comprising Tomasulo's Algorithm are (1) Reservation Station, (2) Common Bus and (3) Reorder Buffer (added in later modification, to implement *Precise Interrupt*). The distinct feature of the algorithm is that instructions are executed out-of-order, in the order in which operands are available

and in that sense it approachs the Data-Flow paradigm (in which operations are executed as soon as the operands for it are available).

Instructions are fetched in-order in the fetching stage. In the decode stage, the false dependencies are removed by the technique of **Register Renaming** in which a new register (from a set of registers) is assigned if data is to be written to a register assigned by the instruction, which is detected by a information bit added to the register. After that, instructions are sent to the **Reservation Station**, which always listens for data on the **Common Bus** and assigns any data available to the needed operands. Once all the operands are available, the instruction is issued to the **Execution Unit** which executes it and announces the result on the **Common Bus** as well as writing the result on the **Reorder Buffer**, which takes care of the order in which instructions were fetched and commits the instruction to register in that order. So it maintains an overall picture of sequential execution of instructions. This helps in the implementation of **Precise Exception** in which all instructions executed before the faulting instruction are guaranteed to be completed, and all instructions after it are discarded or can be restarted without side effects.

Chapter 11

Superscalar Processors

Uptil now, only one instruction was fetched at a time and it was called **Scalar** processors. In the late 1980s, many processors implemented the technique of fetching multiple instructions with the help of replicated hardware. It came to be known as **Superscalar** processor and it increased the IPC (instructions per cycle), but not linearly with the hardware.

11.1 Modern Superscalar Processors

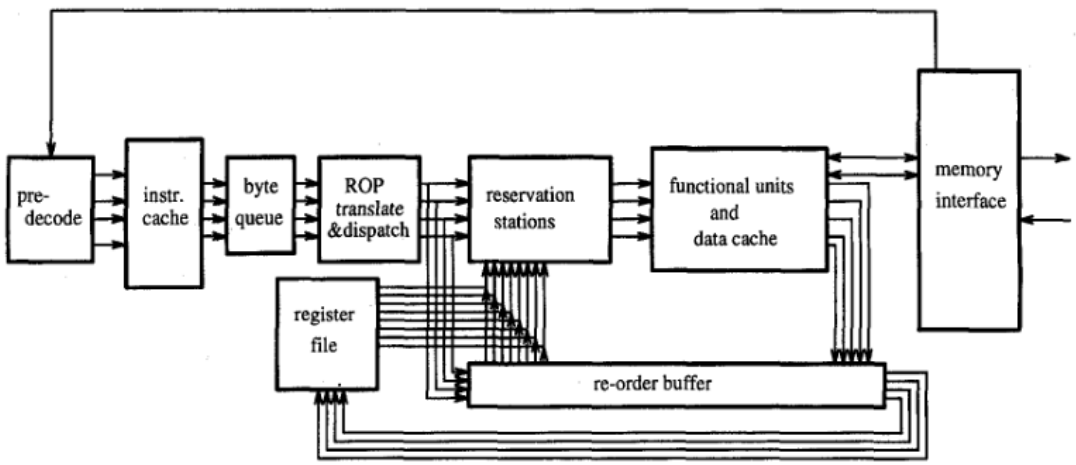


Figure 11.1: AMD K5 superscalar implementation of the Intel x86 instruction set

In the early 1990s, the technique of multiple instructions fetching and Tomasulo's algorithm are combined to form the processors which show efficiency much greater than the efficiency of any one of them. All the modern processors (eg., Intel, AMD, ARM, IBM, etc) uses the same technique.

11.2 Branch Prediction

When multiple instructions are fetched in one cycle, there is a high possibility of it containing branch instructions. When a branch instruction is encountered, it halts the flow of instructions till the time its directions and destination address are calculated (The same problem also appears in the pipeline architecture). This

hampers the efficiency.
To avoid this, techniques are used to predict the branch direction and address in advance while the normal execution for direction and address continues. If the predicted branch and address are correct, the normal execution flow continues. If the direction is wrong, all the instructions on the wrong path are flushed and Instruction Pointer moves to the calculated address. Various techniques are used to predict the direction and address of the branch. They are divided into two types on the basis of which information is used for prediction.

11.2.1 Static Branch Prediction

It predicts the outcome of a branch based solely on the branch instruction. All decisions about the direction of the branch are made at compile time, which can be assisted by adding extra information by the programmers (*pragmas*). Techniques include predicting that a conditional branch will always be taken, or always not taken. A more advanced version of static predictor assumes that forward branches are always not taken while backward branches are always taken (which might be useful in the case of loops).

11.2.2 Dynamic Branch Prediction

It uses the information about taken or not taken branches gathered at run-time to predict the outcome of a branch. It comes in varied form. Some uses a few bits (2 is commonly used) to store information about the previous branch and saturately increments it if the prediction is correct and saturately decrements it otherwise and takes the prediction decision according to the value of the bit. Two-Level Branch Prediction seems to be most effective in this case as it uses the correlation between the different branches for prediction. It uses local or global histroy of past n branch directions (in a register, called *BHSR* (Branch History Shift Register)) to index the *PHT* (Pattern History Table), the value at which predicts the direction of the current branch.

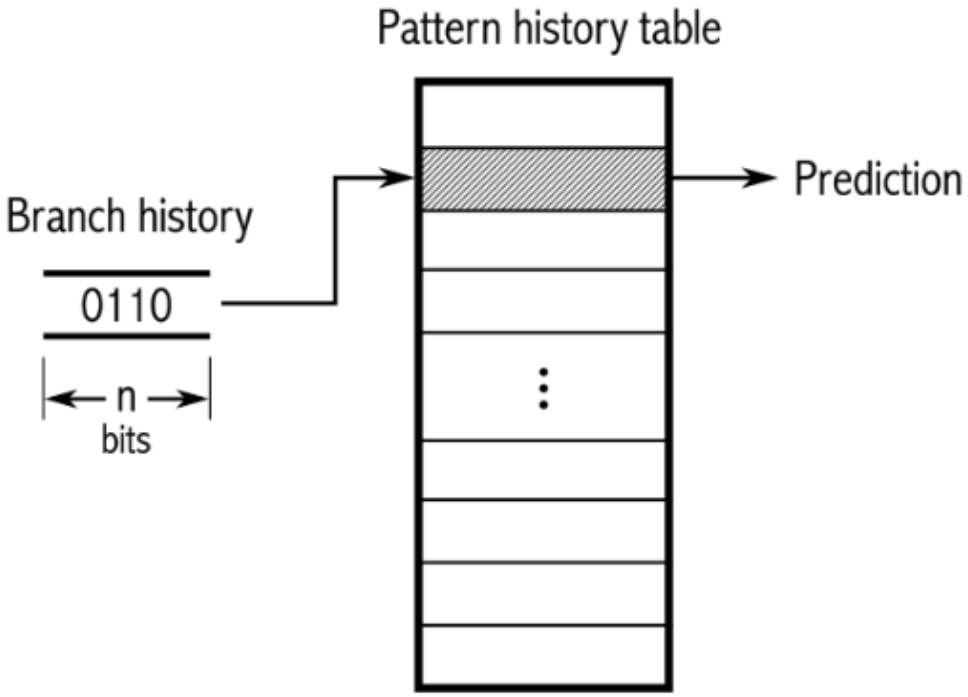


Figure 11.2: Two-Level Adaptive Branch Predictor

11.2.3 Mixed Branch Prediction

Modern processors uses a mix of various Dynamic Branch Predictors and also keeps Static Branch Predictors as a fallback option. The reasoning behind this is that various predictors give different accuracy on different types of branches. So, the mix gives the good of every predictor which ultimately increases prediction accuracy.

11.3 Memory Access

Memory access takes time, especially if there is a *cache* miss. This problem is handled by the architecture of the Superscalar processors. The instruction for which memory access is required is put in the reservation station till the time memory is accessed. It then normaly continues the execution. While this works, it leads to high latency and there are other methods to solve this.

Chapter 12

Graphical Processing Unit

With the advent and rise of Machine Learning, Image Processing and huge amounts of data produced in fields like astronomy, the need for fast computation of numbers in bulk grew rapidly. While processors are great for general-purpose processing, to handle such large and bulk calculations, special-purpose processors were thought of that can do a particular task with much high speed and efficiency.

12.1 Single Instruction, Multiple Data

The early processors were operating on the principal of **SISD** (Single Instruction, Single Data), i.e., a single instruction was operating on a single (pair) data. Soon it was realized that in the fields of Machine Learning and other such fields, a particular instruction acts on a large set of number (arrays or matrices). So instead of fetching the same instruction for a large set of data, an instruction was fetched and executed on the large set of data procured in once. That way, the overhead for fetching the instruction on every data was removed and it greatly increased the speed of execution.

12.2 GPU

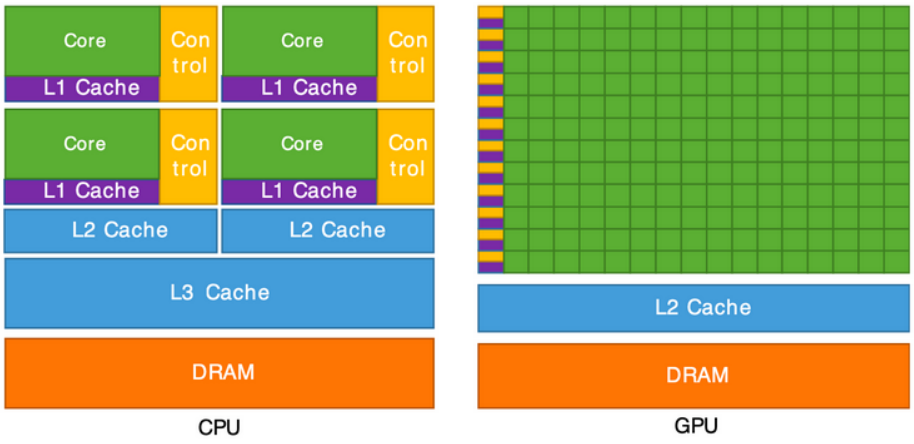


Figure 12.1: CPU vs GPU architecture

GPUs were created on the principal of SIMD, originally for the processing of images in the gaming industry. With the advent of ML, it was adapted as the processor for processing data in hugh amount (matrix multiplication, for example). **NVIDIA** and **AMD** are the leading companies in the production of GPUs. For

example, *CUDA* technology from **NVIDIA** allows developers to use GPUs for general-purpose computations and significantly speeds up the task by executing the code in parallel on the multiple processing cores found in *NVIDIA GPUs*.

Multiple Instruction, Single Data

In the path of ever higher speed, different techniques are explored and exploited. One of them is *MISD* (Multiple Instruction, Single Data) which is employed in *Systolic Arrays* in which multiple instruction are executed on a single data so that the need to fetch the same data is reduced. This works in some specific field, though.

Chapter 13

Computer Memory

The Computer Memory is the storage house from where data are fetched and stored back after some computation on it. It is also the storehouse for the instructions that are executed in the processor. In a continuous effort to increase the execution speed of the processor, the time required for fetching and storing of data from the memory becomes critical. Various attempts have been made to decrease the time required by the processor to fetch the data and instruction from the memory. // There mainly exists two types of memory:

1. **Cache:** Memory which are fast and very expensive, made using SRAM technology
2. **Main Memory:** Memory which are slow and cheap, made using DRAM technology

They cannot be used singularly, so a tradeoff is reached in which both are used together to get the best of both, small cache for fast access and big main memory for large storage at a reasonable cost.

13.1 Memory Hierarchy

In the modern processors, memory system consists of a multilevel (3-4 levels) of cache (with decreasing access speed) and a main memory. So they form a hierarchy, from the *L1* cache at the top to the *Main memory* at the bottom.

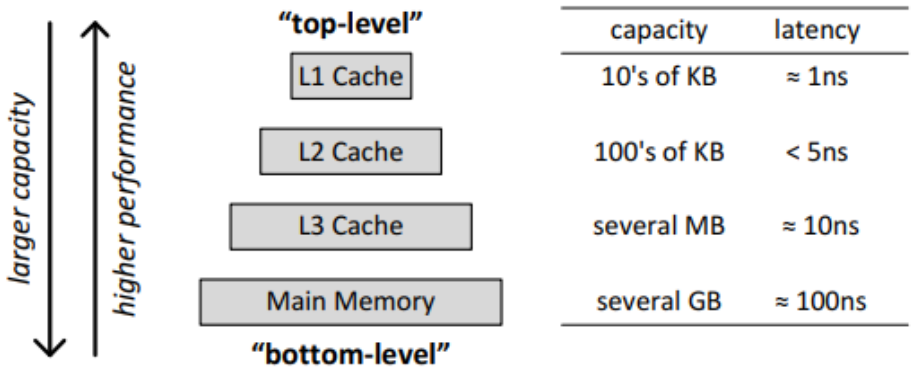


Figure 13.1: Memory Hierarchy with three levels of cache

Many computer programs exhibits *locality* in their memory access behavior. Locality exits in two forms: *temporal locality* in which the same piece of data is likely to be accessed again and again and *spatial locality* in which the data in the neighbor of an accessed data is likely to be accessed in the near future. The memory hierarchy leverages this locality in the memory access behavior by storing the recently accessed data and carrying a block of data next to the data being accessed in the cache. This way, locality in memory access can be satisfied by accessing data from cache without any delay.

13.2 Virtual Memory

A memory system does not directly expose its address space to the user, but instead provides the illusion of an extremely large address space that is separate for each individual programs. This is done to hide the actual size of the physical memory and also to allow multiple programs to run concurrently on the processor without overwriting each other's data. This illusive memory is called *Virtual Memory.* //

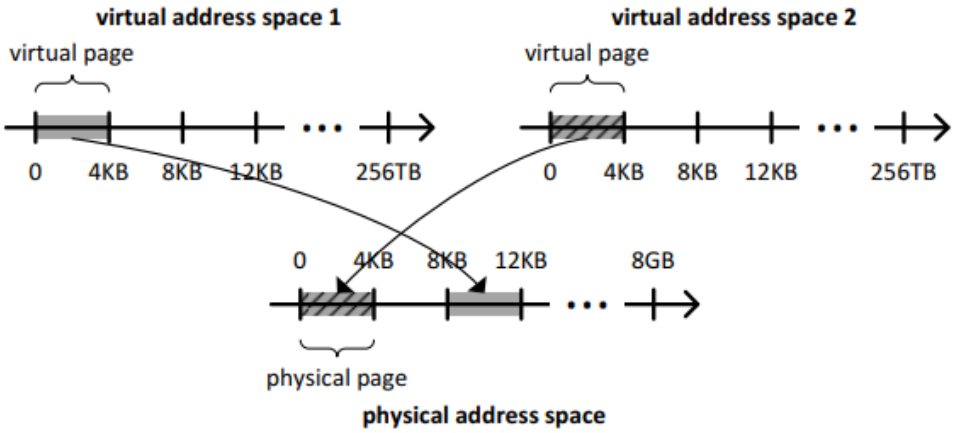


Figure 13.2: Virtual Memory System

A virtual memory address, just by itself, does not represent any real storage location unless it is actually backed up by a physical memory address. It is the job of the operating system to substantiate a virtual address by mapping it to a physical address. // The operating system keeps track of the mapping between Virtual and Physical memory address by storing it in a data structure called the *page table.* In a memory reference during instructon execution, virtual memory is referenced which is translated by the *Translation Lookaside Buffer (TLB)* (a cache storing instance of the page table) into the actual memory address which is used to access the data or instruction at the physical memory.

13.3 Caches

A cache is any structure that stores data that is likely to be accessed again. Data stored in a cache can be accessed quickly by the processor. The effectiveness of a cache depends on whether a large fraction of the memory accesses "hits" in the cache and, as a result, are able to avoid being served by the much slower main memory.// A cache is divided into many small pieces, called *cache blocks*, and memory transfers data to the cache in this granularity. A cache block has a *tag* associated with it which is used to index the data stored and it signifies the address of the data.

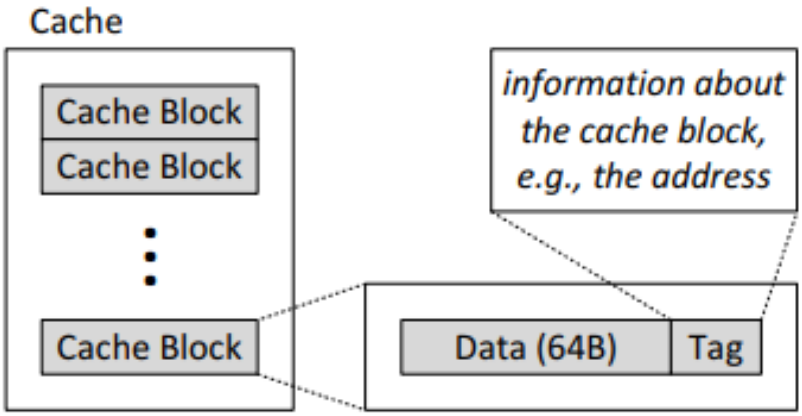


Figure 13.3: Cache

A mapping between the memory chunks and the cache blocks is required for the transfer of data. There are several ways of mapping:

- **Fully-Associative:** In this mapping, a new chunk can be placed in any empty cache-block. If a block is not empty, an occupied block is replaced by the new block (with the data written back to memory, if it has been changed). Different ways are used for choosing the block to replace such as LRU (Least Recently Used), FIFO or Random.
- **Direct-Mapped:** In this mapping, a new chunk can be placed only in a specific block of cache (which can be calculated by some specific way, such as modulus). If the block is filled, it is evicted and replaced by the new chunk.
- **Set-Associative:** It is a mix of the two mapping. It uses a *set* which is a group of few cache blocks which is fixed but any block can be chosen inside the set to place the new chunk.

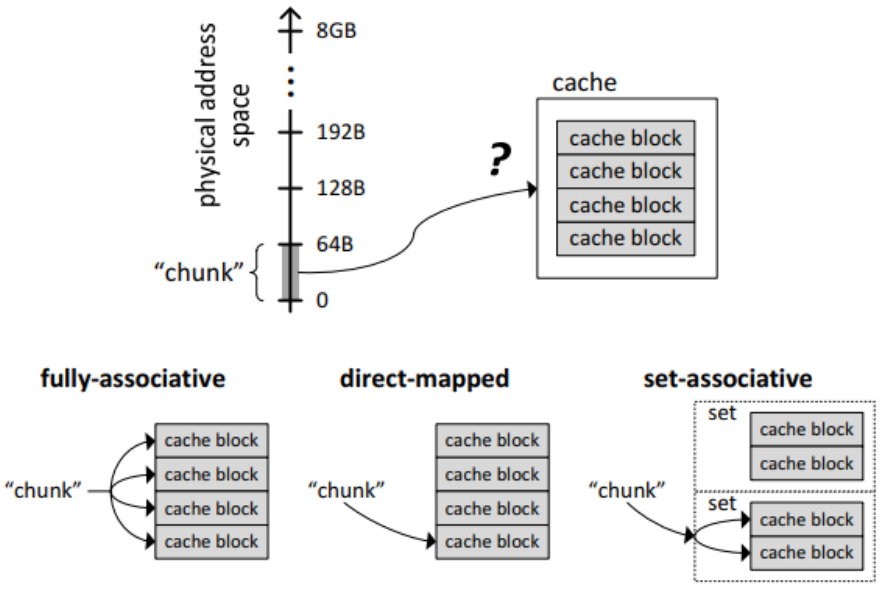


Figure 13.4: Mapping of cache with physical memory

Cache for Virtual Memory

The TLB (Translation Lookaside Buffer) used in the mapping of virtual to physical memory address is a cache which refrences data from the page table which itself acts as a cache for the actual mapping that is stored in the secondary memory.

13.4 Prefetching

Prefetching is a technique used in fetching instruction or data from memory and putting it in the cache to increase the computational speed. In this, instructions or data that might be needed a few steps away in the future is brought and placed in the cache so that the demand for it can be fulfilled without any delay. A good prefetching algorithm is required, or otherwise it will fill the cache with data that might never be needed and thus hampering the performance.

Chapter 14

Current Trend and the Way Forward

Computing is bottlenecked by data. Due to the big gap between the place where data is stored and where data is used for computation, a huge amount of time and energy goes into data movement across the entire computing system, with some going as high as 60%. While various researchs are going on to make the memory cheaper and access time lower, many researchers are talking about a change in paradigm to give more preference to memory.
The paradigm that is being followed currently is processor-centeric, which creates a clear dichotomy between processing and memory/storage. Data has to be brought from storage and memory units to computation units, which are far away from the memory/storage units, before any processing can be done on the data. Research is going on the memory-centeric paradigm according to which processing is either done *using the memory* or *close to it*. This way the travelling cost might be greatly eliminated.
Whatever the direction be, the quest for more computational power will never stop!

References

- Omur Mutlu *Digital Design & Computer Architecture*
- James E. Smith, Gurindar S. Sohi *The Microarchitecture of Superscalar Processors*
- Tse-Yu Yeh, Yale N. Patt *Two-Level Adaptive Training Branch Prediction*
- Yoongu Kim, Onur Mutlu *Memory Systems*
- Scott McFarling *Combining Branch Predictors*