Cloud Infrastructure Numerical & Like

Numericals and Fundamentals

Relation Between Clock and Instruction Cycle

- $\mathbf{F_{osc}}$ = oscillator frequency or clock frequency.
- $\mathbf{F_{cy}}$ = instruction cycle frequency. How many full instruction cycles (fetch–decode–execute–store) are executed per second.
- $\mathbf{T_{cy}}$ = instruction cycle period = $1/F_{cy}$.The time required to complete one full instruction cycle.
- Typically, $\mathbf{F_{cy} = F_{osc} \div N}$, where *N* is an implementation-dependent divider.

Timing Characteristics

- One instruction cycle requires multiple clock ticks (depending on architecture).
- Simple instructions may require 1 cycle.
- Memory access or complex arithmetic may require multiple cycles.
- Performance = F_{cy} × average instructions per cycle (IPC).

Amdahl's Law

Programs have two parts:

- A **serial portion** that must run on a single processor.
- A **parallel portion** that can be divided among multiple processors.

Adding processors only helps the parallel portion. The serial portion takes the same time no matter how many processors are available.

Overall speedup is limited by the serial portion. If even a small fraction of the program is serial, it puts a hard cap on the maximum possible improvement.

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

f = fraction of program that can be parallelised
N = number of processors
(1−f) = serial part, unchanged by parallelism

Problem

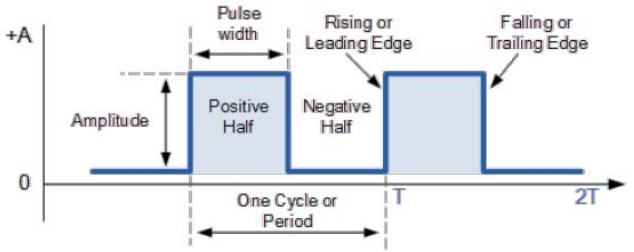
A program has been updated with 80% parallelizable code and 20% sequential code. The parallelization is done using 6 processors. What is the speedup in percentage?

Parallel fraction (f): 80% = 0.8
Sequential fraction (1 – f): 20% = 0.2
Processors (N): 6
Speedup: ~3.0

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

Review of Processor Operations

Every operation a processor performs — fetching an instruction, decoding it, loading data, and executing arithmetic or logic — is driven by a system clock. The clock generates pulses that act as the rhythm for the processor. Each operation starts with a clock pulse, and the overall speed of the processor is tied to how frequently these pulses occur, measured in Hertz (Hz).



Clock Cycles and Speed

For example, a 1 GHz (10^9) processor generates one billion pulses per second. Each pulse represents a clock cycle, and the time between pulses is called the *cycle period*.

The higher the clock speed, the more operations the processor can start each second.

However, performance isn't only about clock speed — memory access delays, pipeline efficiency, and instruction parallelism also determine how much work gets done per cycle.

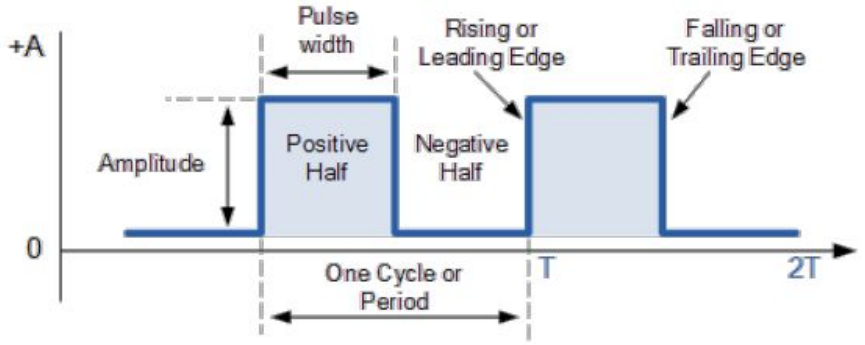
Clock Signal: The Processor's Timing Pulse

Amplitude (+A to 0):
This is the voltage difference of the clock signal. +A is the "high" state, and 0 is the "low" state. The processor recognises these two states as the basis for timing.

Positive Half:
The duration when the clock signal is high (+A). This is often when certain operations (like latching data) are triggered.

Negative Half:
The duration when the clock signal is low (0). Some circuits are designed to react during this phase.

Pulse Width:
The time the signal stays in the high state during one cycle. It is a portion of the total cycle.



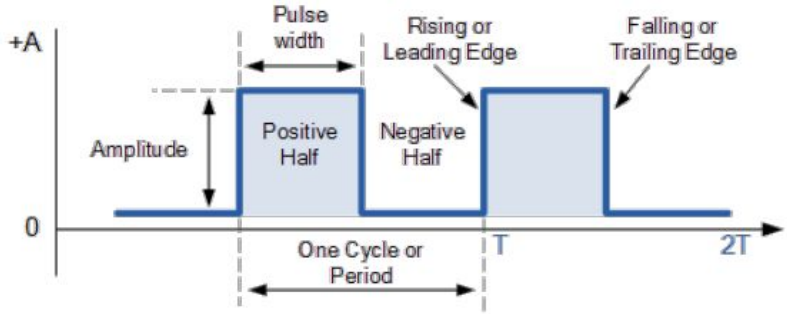
Clock Signal: The Processor's Timing Pulse

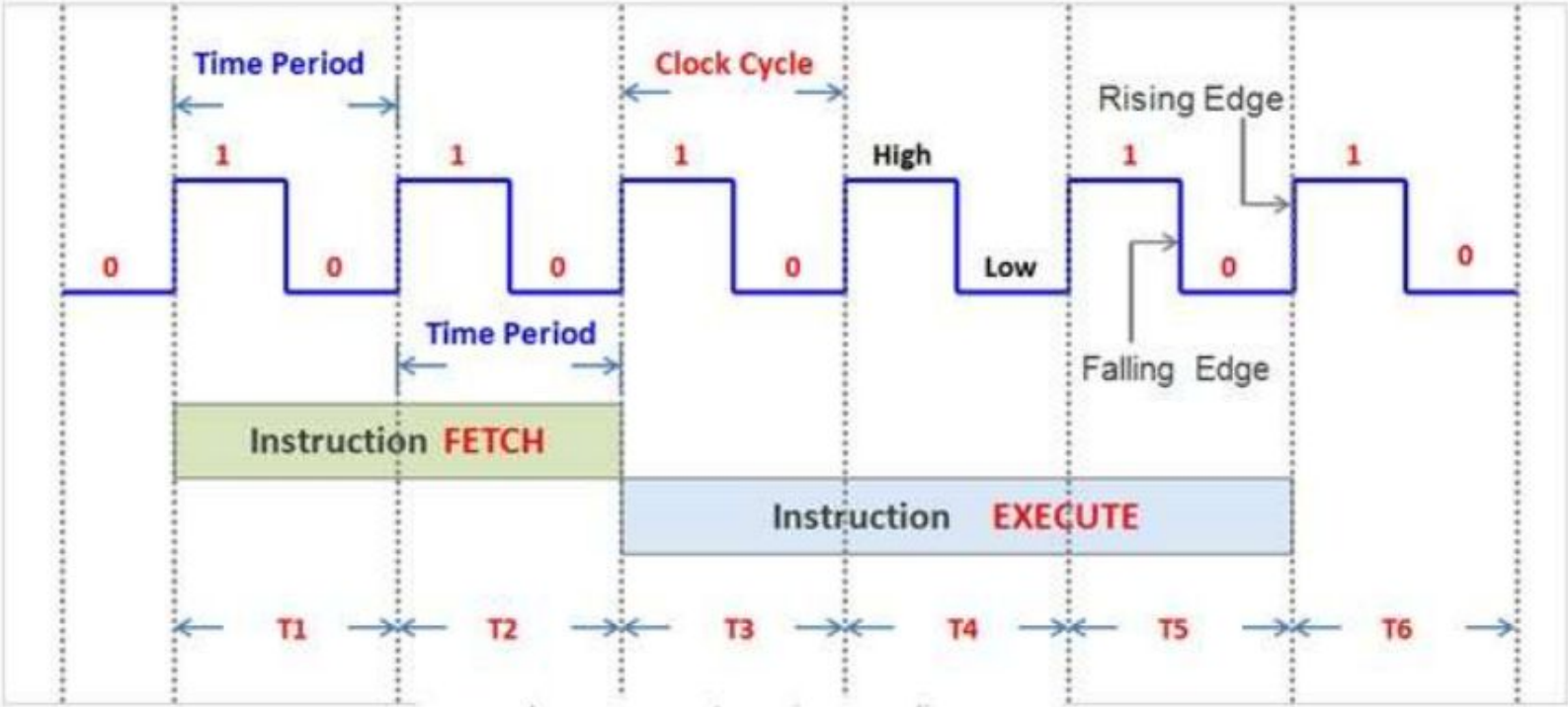
Rising Edge (Leading Edge):
The transition point where the signal goes from low (0) to high (+A). Many digital circuits, including processors, start new operations on this edge.

Falling Edge (Trailing Edge):
The transition where the signal drops from high (+A) back to low (0). Other circuits may use this edge to trigger their operations.

One Cycle (Period, T):
A full cycle is the combination of one positive half and one negative half. It is the repeating unit of the clock signal.

- Measured in **time (seconds)**.
- The **frequency** is the inverse of this period (f = 1/T). For example, if T = 1 nanosecond, f = 1 GHz.




Time Period
Clock Cycle
Rising Edge
1
1
1
High
1
1
0
0
0
0
Low
0
0
Time Period
Falling Edge
Instruction FETCH
Instruction EXECUTE
T1
T2
T3
T4
T5
T6

Understanding Clock Speed and Cycle Time

A **1-GHz processor** means the chip gets **1 billion clock pulses every second**.

- The number of pulses per second is called the **clock speed**.
- Each single pulse is a **clock cycle**.
- The tiny gap between two pulses is the **cycle time**.

Cycles Per Instruction (CPI)

Instruction count (I_c) → total number of instructions executed by the program.

CPI_i → cycles per instruction for a specific instruction type *i*.

I_i → number of instructions of type *i*.

CPI (average) → weighted average of cycles per instruction across the program.

Cycle time (τ) → duration of one clock cycle (*1/f*).

Execution time formula:
Execution Time=(Instruction Count)×(Average CPI)×(Cycle Time)

Program runtime is influenced by (1) number of instructions, (2) how many cycles instructions need, and (3) clock speed.

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c}$$

$$T = I_c \times CPI \times \tau$$

MIPS (Millions of Instructions Per Second)

Definition: MIPS measures how many million instructions a processor executes per second.

Variables:

- I_c → number of instructions executed.
- T → total execution time.
- f → processor clock frequency.
- CPI → average cycles per instruction.

$$MIPS = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

Formula meaning:

- Higher **clock speed (f)** increases MIPS.
- Lower **CPI** improves MIPS (fewer cycles per instruction).

Case Study

The difference in RAM speed (4400 MT/s vs. 6000 MT/s). Whether changing the RAM from 4400 MT/s to 6000 MT/s will do?

- Simply upgrading RAM from **4400 MT/s to 6000 MT/s** may **not work** unless both the **motherboard** and the **processor** support the higher speed.
- **Motherboard dependency:** The memory controller on the motherboard (chipset + BIOS) must allow 6000 MT/s. If it only supports up to 4400 MT/s, faster RAM will downclock to 4400 MT/s.
- **Processor dependency:** Modern CPUs have integrated memory controllers. If the CPU only supports up to 4800 or 5200 MT/s, installing 6000 MT/s RAM won't run at full speed.
- Changing RAM alone is not enough. You may need to change the **motherboard, processor, or both**, depending on their supported memory speeds.

Case Study

How does the difference in RAM speed (4400 MT/s vs.6000 MT/s) influence the overall system performance in tasks such as gaming, content creation, and AI/ML workloads, given that all other hardware components are identical?

Impact of RAM Speed on System Performance

- **Gaming:**
  - Most modern games are more dependent on GPU and CPU performance.
  - Faster RAM (6000 MT/s vs. 4400 MT/s) can improve *minimum frame rates* and reduce stuttering, but average FPS gains are often modest (~3–10%).
- **Content Creation (e.g., video editing, 3D rendering):**
  - Applications that handle large datasets (like 4K/8K video timelines or huge texture assets) benefit more from higher memory bandwidth.
  - You may see noticeable performance improvements in *export times and real-time previews*.
- **AI/ML Workloads:**
  - These workloads often involve large matrix multiplications and memory-bound operations.
  - Higher RAM speed can give significant boosts in *training throughput* and *data preprocessing speed*.
- **Overall:**
  - Faster RAM gives diminishing returns if the workload is CPU/GPU bound.
  - Workloads that are memory-intensive benefit more.
  - Performance uplift varies from *minor (gaming)* to *moderate/significant (AI/ML, heavy content creation)*.

Case Study

An organization intends to procure a high-performance server to meet its extensive processing and GPU requirements (currently around 500 GB). Also, a separate redundance storage solution is required. The server will be utilized to create virtual machines (VMs) that will be allocated to various associated organizations, with the flexibility to reallocate them based on evolving configuration needs. Additionally, the organization seeks a scalable solution that allows for future infrastructure expansion while maintaining the core system configuration. What would be the most suitable server solution to meet these requirements?

Key Requirements in the Problem

High Processing and GPU Power

- The organization has heavy computational needs (around 500 GB of GPU memory capacity).
- This suggests workloads like **AI/ML, big data processing, or GPU-intensive simulations**.

Separate Redundant Storage

- They don’t just want performance, but also **reliability**.
- Redundant storage means a system like **RAID arrays, SAN (Storage Area Network), or NAS with redundancy** to avoid data loss.

Virtual Machines (VMs)

- The server must support **virtualization**.
- VMs will be allocated to different organizations (multi-tenancy).
- There should be flexibility to **reallocate VMs dynamically** as needs change.

Scalability

- The solution should **scale in the future** without redesigning the entire infrastructure.
- Core configuration (CPU, GPU, memory structure) should remain intact while allowing easy expansion.

Average Memory Access Time (AMAT)

The performance of the memory system can be summarised with the formula:

AMAT = Hit time + (Miss rate × Miss penalty)

This averages the fast accesses (hits) with the slow ones (misses).

- A lower AMAT means better performance.
- Because miss penalties are usually far larger than hit times, the best ways to reduce AMAT are to:
  - Lower the **miss rate** (by increasing cache size or improving replacement policies).
  - Reduce the **miss penalty** (for example, by using multi-level caches).

Performance Example

Suppose:

- 33% of instructions are data accesses.
- Cache hit ratio = 97% → Miss rate = 3% (0.03).
- Hit time = 1 cycle.
- Miss penalty = 20 cycles.

Step 1: Compute AMAT for data accesses

AMAT_data = Hit time + (Miss rate × Miss penalty)
= 1 + (0.03 × 20)
= 1 + 0.6 = **1.6 cycles**

Step 2: Combine with overall workload

- Data accesses = 33% of instructions → cost = 0.33 × 1.6 = 0.528
- Other instructions (67%) → cost = 0.67 × 1 = 0.67

So, AMAT_overall = 0.528 + 0.67 = **1.198 cycles**

This means, on average, each memory access effectively takes about **1.2 cycles**, thanks to the high hit ratio.

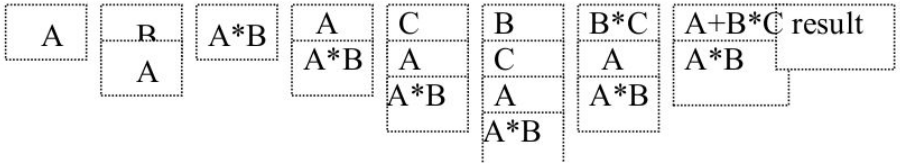
Stack Architecture Example

A * B – (A + C * B)

- **Rule**
- A binary operation (`mul`, `add`, `sub`) **pops** the top two values, performs the operation, and **pushes** the result back.

push A	**[A]**
push B	**[B, A]**
mul	**[A*B]**
push A	**[A, A*B]**
push C	**[C, A, A*B]**
push B	**[B, C, A, A*B]**
mul	**[C*B, A, A*B]**
add	**[A + C*B, A*B]**
sub	**[A*B − (A + C*B)]**

- No need to name registers explicitly.
- Operations always use the top of the stack.
- Instruction sequence is longer because of repeated push/pop.



Accumulator Architecture: Rules

Rule:

One accumulator register is the main working area for all arithmetic operations.
Binary operations (like `add A`, `sub A`, `mul A`, `div A`) always use:

- The value in the accumulator **and**
- The value from memory (operand A).
- Result is stored back in the accumulator.

To save intermediate results, you use `store X`.
To bring a value from memory into the accumulator, use `load X`.

Example: A * B – (A + C * B)

1. **load B** → Acc = B
2. **mul C** → Acc = B * C
3. **add A** → Acc = A + (B * C)
4. **store D** → Save (A + B * C) into D (for later use)
5. **load A** → Acc = A
6. **mul B** → Acc = A * B
7. **sub D** → Acc = (A * B) – (A + B * C)

Final result in Accumulator = A * B – (A + C * B)

Question

A 2 word instruction is stored in memory at an address designated by the symbol W. The address field of instruction (stored at W +1) is designated by symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. State how Z is calculated for different addressing modes.

We are given a two-word instruction stored in memory starting at address **W**. The first word contains the opcode, and the second word (at **W+1**) holds the address field, denoted as **Y**. The actual operand needed during execution is located at some effective address **Z**. Depending on the addressing mode, **Z** is computed differently. An index register holds a value **X**, which is used in the indexed mode. The problem asks us to express how **Z** is calculated under direct, indirect, relative, and indexed addressing modes.

Answer

How Z is Calculated as the following:

- **Direct addressing**: The operand is directly at the address Y. → **Z = Y**
- **Indirect addressing**: The operand's address is found in memory at Y. → **Z = M[Y]**
- **Relative addressing**: The operand's address is given relative to the program counter (next instruction at W+2). → **Z = Y + (W+2)**
- **Indexed addressing**: The operand's address is obtained by adding the index register to Y. → **Z = Y + X**

Question

An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate; (c) relative; (d) register indirect; (e) index with R1 as the index register.

The problem is about **effective address calculation** under different addressing modes.

1. The **instruction** itself is stored at **memory location 300**.
2. Its **address field** (the part that may contain the operand or an address) is stored at **location 301**.
3. This **address field** has the value **400**. Depending on the addressing mode, this "400" may be treated as an actual memory address, an immediate constant, or combined with other values.
4. A **processor register R1** is also given, which currently contains the value **200**. In some addressing modes (like indexed), this register will be added to the address field.

The task: For each addressing mode—direct, immediate, relative, register indirect, and index with R1—you must work out how the effective address (**EA**) is computed.

The **Effective Address (EA)** is the actual memory address where the operand is located (except in immediate mode, where the operand is part of the instruction itself).

(a) Direct addressing

In direct addressing, the address field itself gives the memory location of the operand. No extra calculation is required.

Here, address field = 400 → **EA = 400**.

So the operand will be fetched from memory location 400.

(b) Immediate addressing

In immediate addressing, the address field does not represent a memory location at all; it is the operand value itself.

Here, address field = 400 → **operand = 400**.

So the processor uses 400 directly as data, and there is **no EA** to compute.

(c) Relative (PC-relative) addressing

In relative addressing, the operand's effective address is obtained by adding the displacement in the address field to the Program Counter (PC) of the next instruction.

Here, PC = 302 (since current instruction occupies 300–301) → **EA = 302 + 400 = 702**.

So the operand will be fetched from memory location 702.

(d) Register indirect addressing

In register indirect, the register itself contains the memory address of the operand. The instruction specifies the register, not the address.

Here, register R1 = 200 → **EA = 200**.

So the operand will be fetched from memory location 200.

(e) Indexed addressing (with R1)

In indexed mode, the effective address is obtained by adding the contents of an index register to the address field.

Here, address field = 400 and R1 = 200 → **EA = 400 + 200 = 600**.

So the operand will be fetched from memory location 600.