# Intro to Software Engineering
# EK327

Giacomo Cappelletto

21/1/25

# Contents

# Chapter 1

# Digital Logic

## 1.1 Adding Two Single-Bit Binaries Using XOR and AND Gates

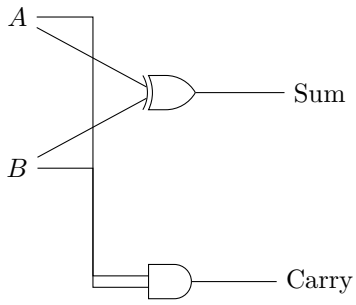
To add two single-bit binary numbers $A$ and $B$, we need to calculate: 1. The **sum bit**, which is the XOR of $A$ and $B$. 2. The **carry bit**, which is the AND of $A$ and $B$.

**Logic - Sum Bit:** Sum = $A \oplus B$ - **Carry Bit:** Carry = $A \cdot B$

The following truth table summarizes the operation:

| $A$ | $B$ | Sum | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

The addition is implemented using XOR and AND gates as shown below:




The XOR gate produces the sum bit, while the AND gate produces the carry bit. This forms the fundamental building block of a full binary adder.

## 1.2 Adding Two Single-Byte Binary Numbers Using XOR and AND

Binary addition can be performed on two single-byte numbers using bitwise operations. Specifically: - The **XOR (Exclusive OR)** operation gives the sum of two bits without considering the carry. - The **AND** operation identifies where a carry will occur. - The carry is then shifted left by one position and added to the result in subsequent iterations.

**Algorithm:** 1. Compute the sum without carry using XOR:

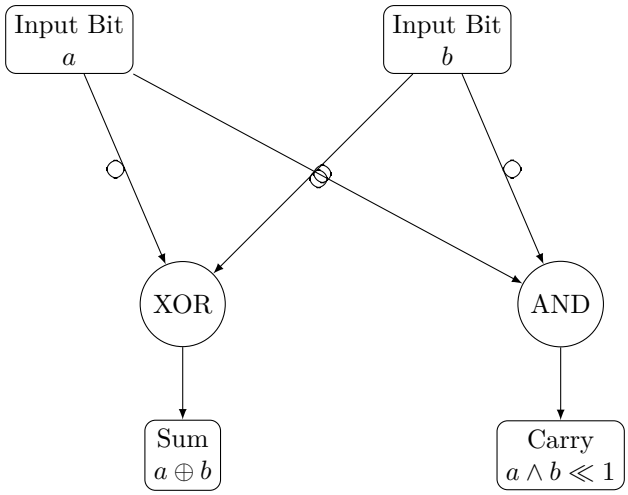
$$\text{sum} = a \oplus b$$

2. Compute the carry using AND and shift it left by one bit:

$$\text{carry} = a \wedge b \ll 1$$

3. Repeat the process by adding the carry to the sum until there is no carry left.

**Diagram:** Below is a diagram that illustrates the process for a single bit addition.




**Example:** Consider two single-byte binary numbers:

$$a = 01101101, \quad b = 10101001$$

1. Compute the XOR for the sum:

$$\text{sum} = a \oplus b = 11000100$$

2. Compute the AND for the carry and shift left:

$$\text{carry} = a \wedge b \ll 1 = 00001010$$

3. Add the sum and carry: - New sum: sum = $11000100 \oplus 00001010 = 11001110$ - New carry: carry = $11000100 \wedge 00001010 \ll 1 = 00000000$

4. Final result: 11001110.

## 1.3 One's Complement for a Single Byte

The **one's complement** of a binary number is obtained by flipping all the bits, changing every `1` to `0` and every `0` to `1`. This operation is commonly used in binary arithmetic, particularly in representing negative numbers in early computing systems.

**Steps to Compute One's Complement:** 1. Write down the binary number. 2. Flip all bits: - Change each `0` to `1`. - Change each `1` to `0`.

**Example:** Consider the binary number $a = 01101101$.

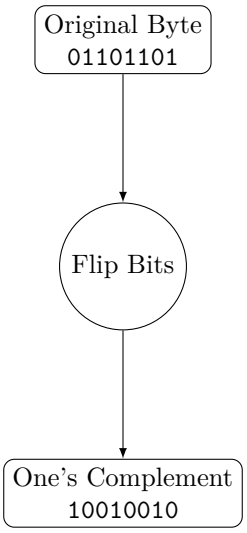
1. Original binary number:

$$a = 01101101$$

2. One's complement (flip all bits):

$$\text{one's complement of } a = 10010010$$

**Diagram:** The following diagram illustrates the transformation:




**Properties of One's Complement:** - A binary number and its one's complement always sum to all 1s (i.e., 11111111 for a single byte). - One's complement is useful in representing signed integers: - Positive numbers are represented as-is. - Negative numbers are represented by the one's complement of their positive counterparts.

**Example with Signed Integers:** For a single byte: 1. 5 in binary: 00000101 2. $-5$ in one's complement: 11111010

**Verification:** Adding 5 and $-5$ in one's complement arithmetic:

00000101 11111010 = 11111111 all bits are 1, representing zero in one's complement arithmetic.

## 1.4 Carrying When Adding Negatives in One's Complement

In one's complement representation, negative numbers are represented by flipping all the bits of their positive counterparts. When adding two negative numbers, a carry might be generated, which needs to be added back to the result to obtain the correct answer.

**Example:** Adding $-3$ and $-4$.

1. Represent $-3$ and $-4$ in one's complement for a single byte:

$$3 = 00000011, \quad -3 = \text{one's complement of } \ 3 = 11111100$$

$$4 = 00000100, \quad -4 = \text{one's complement of } \ 4 = 11111011$$

2. Add the two numbers:

$$\begin{array}{r} 11111100 \\ 11111011 \\ \hline 11110111 \end{array}$$

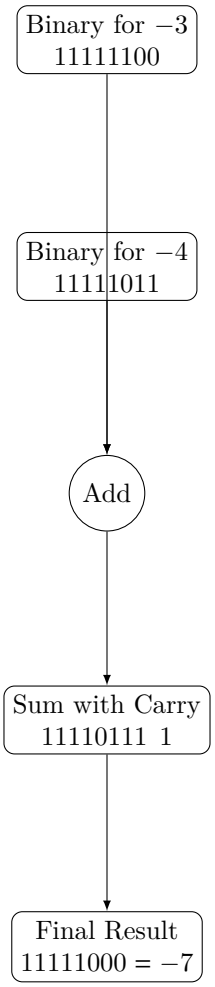
3. Handle the carry: - The result of the addition is 11110111, with a carry bit of 1. - Add the carry back to the least significant bit:

$$11110111 \ 1 = 11111000$$

4. Final result:

$$11111000 = \text{one's complement of } \ 7 = -7$$

**Explanation:** - When the sum generates a carry, it must be added back to the result to comply with one's complement rules. - The result of $-3$ $-4 = -7$, as expected.




**Note:** This approach works for signed integers in one's complement and highlights the importance of handling the carry bit to ensure accurate results.

## 1.5 Two's Complement for a Single Byte

The **two's complement** of a binary number is obtained by flipping all the bits (as in one's complement) and then adding 1 to the result. This method is widely used in modern computer systems to represent signed integers.

**Steps to Compute Two's Complement:** 1. Write down the binary number. 2. Flip all bits (as in one's complement). 3. Add 1 to the flipped number.

**Example:** Consider the binary number $a = 01101101$.

1. Original binary number:

$$a = 01101101$$

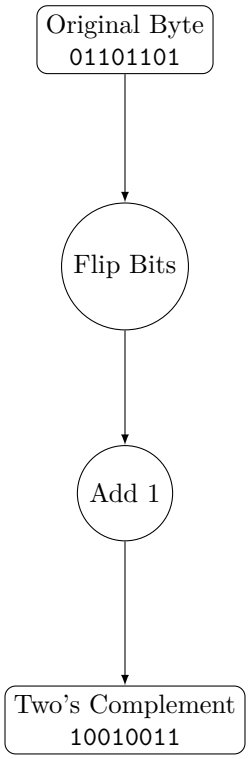
2. Flip all bits (one's complement):

$$10010010$$

3. Add 1 to the result:

$$\text{two's complement of } a = 10010010\ \ 1 = 10010011$$

**Diagram:** The following diagram illustrates the transformation:




**Properties of Two's Complement:** - The two's complement of 0 is 0, and the two's complement of the maximum negative value is itself. - Negative numbers are represented by their two's complement. - Addition and subtraction with two's complement do not require separate subtraction logic, simplifying arithmetic operations.

## 1.6 Carrying When Adding Negatives in Two's Complement

In two's complement representation, negative numbers are represented by flipping all bits of the positive number and adding 1. When adding two negative numbers, the carry generated during addition is discarded.

**Example:** Adding $-3$ and $-4$.

1. Represent $-3$ and $-4$ in two's complement for a single byte:

$$3 = 00000011, \quad -3 = \text{two's complement of } \ 3 = 11111101$$

$$4 = 00000100, \quad -4 = \text{two's complement of } \ 4 = 11111100$$

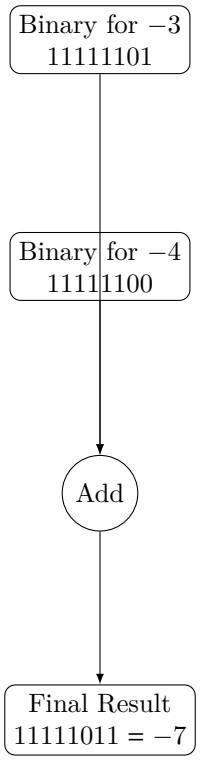
2. Add the two numbers:

$$\begin{array}{r} 11111110 \\ 11111100 \\ \hline 11111011 \end{array}$$

3. Handle the carry: - The result of the addition is 11111011. - In two's complement, the carry is ignored, so the final result is:

$$11111011 = -7$$

**Explanation:** - In two's complement, the carry bit is discarded, unlike in one's complement. - The result of $-3$ $-4 = -7$, as expected.

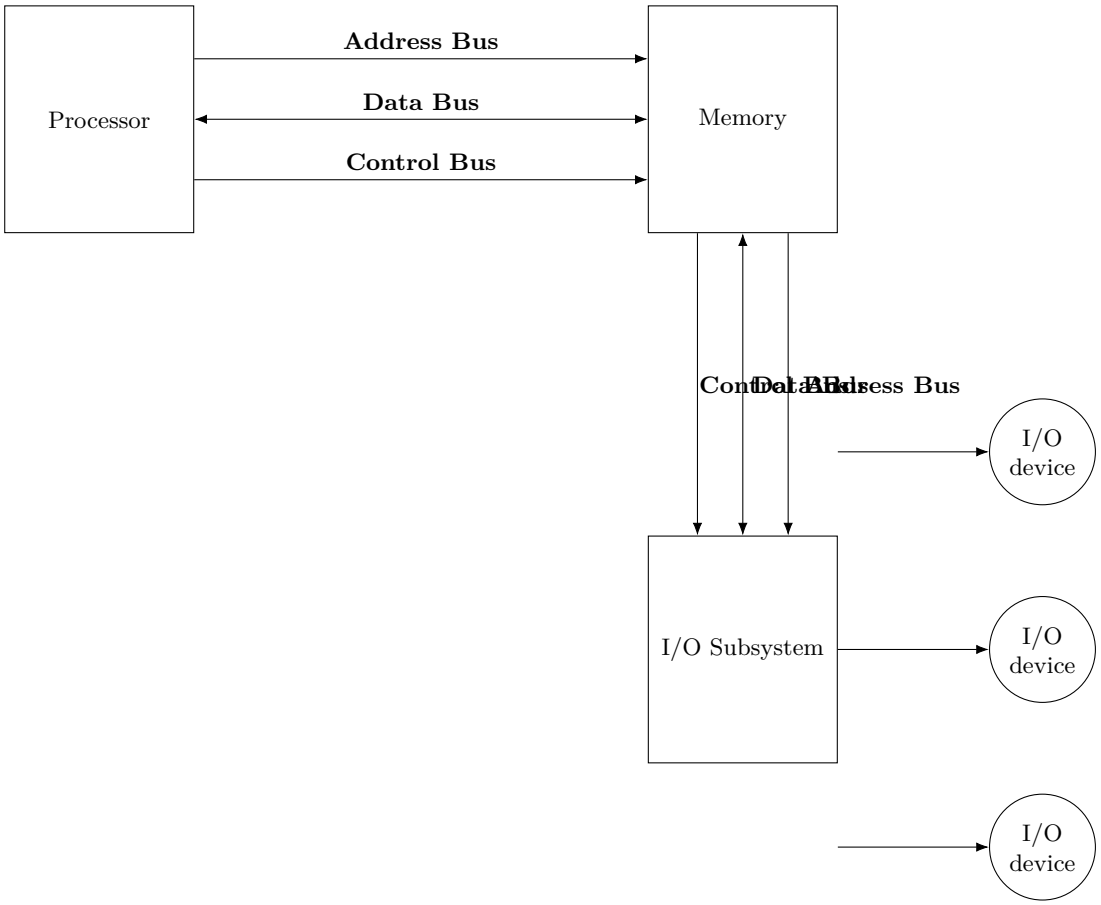



**Note:** Two's complement simplifies arithmetic operations by eliminating the need to add back the carry. It is the most commonly used method for representing signed integers in modern computing systems.

# Chapter 2

# Microarchitecture

## 2.1 Computer Architecture Overview




### 2.1.1 Address Bus

The address bus is a critical component in a computer's architecture, responsible for specifying the memory addresses that the processor will read from or write to. The width of the address bus determines the maximum amount of memory that can be addressed by the system.

**Address Bus Width and Memory Capacity:**

- **8-bit Address Bus:** - Can address $2^8 = 256$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 256 bytes.

- **16-bit Address Bus:** - Can address $2^{16} = 65,536$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 65,536 bytes (or 64 KB).

- **32-bit Address Bus:** - Can address $2^{32} = 4,294,967,296$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 4,294,967,296 bytes (or 4 GB).

- **64-bit Address Bus:** - Can address $2^{64} = 18,446,744,073,709,551,616$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 18,446,744,073,709,551,616 bytes (or 16 exabytes).

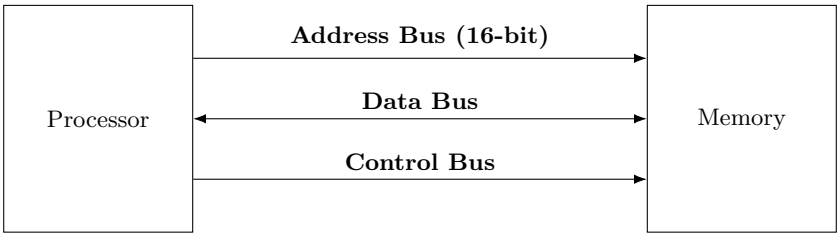
**Example: 16-bit Address Bus**

For a 16-bit address bus, the number of unique addresses is calculated as follows:

$$2^{16} = 65,536 \text{ addresses}$$

Each address corresponds to a unique memory cell, allowing the processor to access up to 65,536 different memory locations. If each memory location holds 1 byte, the total addressable memory is 65,536 bytes (or 64 KB).

**Diagram:**




The address bus plays a crucial role in determining the system's memory capacity and overall performance. A wider address bus allows for more memory to be accessed, which is essential for modern applications and operating systems.

### 2.1.2 Data Bus

The data bus is a bidirectional pathway that transfers data between the processor, memory, and I/O devices. It plays a crucial role in the fetch-execute cycle, enabling the processor to read instructions and data from memory and write results back to memory.

**Multidirectionality in the Fetch-Execute Cycle:** 1. **Fetch:** The processor uses the data bus to read an instruction from memory. 2. **Decode:** The instruction is decoded internally by the processor. 3. **Execute:** The processor may read or write data to/from memory or I/O devices using the data bus.

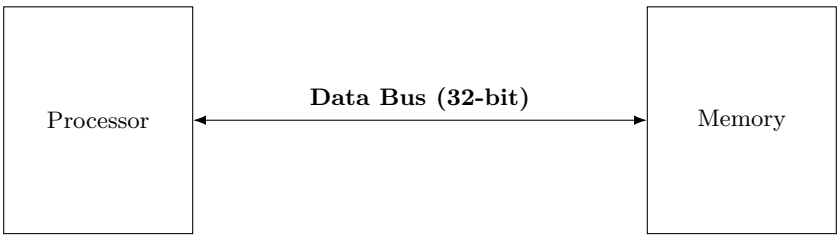
**Bandwidth and Cell Size:** - The width of the data bus determines the amount of data transferred in one cycle. - A wider data bus can transfer more data per cycle, increasing overall system performance. - For example, a 32-bit data bus can transfer 4 bytes per cycle.

**Example: 32-bit Data Bus**

For a 32-bit data bus, the amount of data transferred per cycle is calculated as follows:

$$\text{Data per cycle} = 32 \text{ bits} = 4 \text{ bytes}$$

**Diagram:**




The data bus's bidirectional nature and bandwidth are critical for efficient data transfer, directly impacting the system's speed and performance.

### 2.1.3 Processor Components

The processor, also known as the Central Processing Unit (CPU), is the brain of the computer. It performs calculations, executes instructions, and manages data flow within the system. Key components of the processor include registers, the program counter (PC), and the EFLAGS register.

### Registers

Registers are small, fast storage locations within the CPU used to hold data temporarily during execution. They are crucial for the processor's operation, allowing quick access to frequently used values. In this architecture, we have general-purpose registers and special-purpose registers.

**General-Purpose Registers:** - The CPU has six general-purpose registers, named $r0$ to $r5$. - Each register can store 16 bits of data. - These registers are used for various operations, such as arithmetic, logic, and data manipulation.

**Example:**

| Register | Value (Binary) |
|---|---|
| $r0$ | 0000000000000001 |
| $r1$ | 0000000000000010 |
| $r2$ | 0000000000000011 |
| $r3$ | 0000000000000100 |
| $r4$ | 0000000000000101 |
| $r5$ | 0000000000000110 |

### Program Counter (PC)

The Program Counter (PC) is a special-purpose register that holds the address of the next instruction to be executed. It plays a critical role in the fetch-execute cycle, ensuring the CPU processes instructions in the correct sequence.

**Functionality:** - The PC is automatically incremented after each instruction fetch, pointing to the next instruction in memory. - If a jump or branch instruction is executed, the PC is updated to the target address, altering the flow of execution.

**Example:**

PC = 0000000000001000 Next instruction address

### EFLAGS Register

The EFLAGS register is a special-purpose, read-only register that holds the results of recent operations. It contains several flags that provide information about the state of the CPU and the outcome of arithmetic and logical operations.

**Key Flags:** - **Zero Flag (ZF):** Indicates whether the result of an operation is zero. - Set to 1 if the result is zero. - Set to 0 if the result is non-zero.

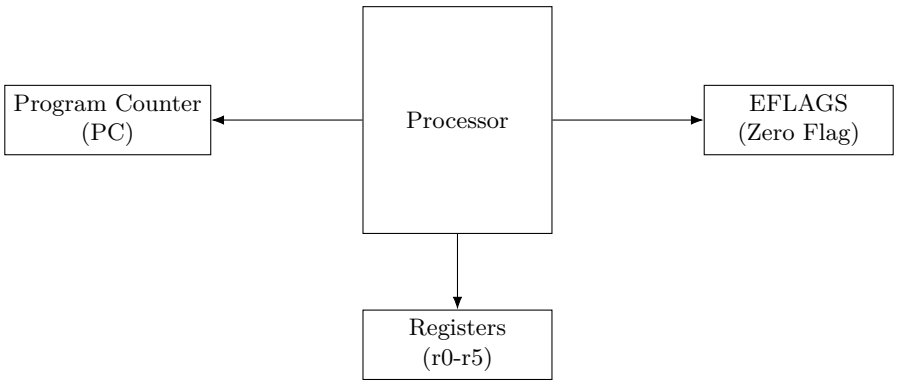
**Example:**

EFLAGS =

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ZF |

**Usage:** - The Zero Flag is commonly used in conditional branching. For example, a jump instruction may depend on whether the Zero Flag is set, allowing the CPU to make decisions based on the results of previous operations.

**Diagram:**




These components work together to enable the processor to execute instructions efficiently, manage data flow, and make decisions based on the results of operations.

# Chapter 3

# Instruction Set

- We will go over all of the instructions that we will cover in this class by example, including their assembly code.
- We will then learn how to represent each instruction in binary, so that the hardware can understand it.
- We will work with them in binary only to start.
- Later, when we get to the Assembly level, we will be able to represent the instructions in more readable form.

## 3.1 Assembly Pseudocode Conventions

- $R_n$: register $n$ (where $n$ is 0 through 5)
- $*R_n$: data in register $n$
- $R_n$: memory cell at the address stored in register $n$
- $R_n \leftarrow x$: contents of register $n$ are now $x$

**Example 3.1.1** (Addition)
assembly:

$$addR_n, R_m$$

pseudocode:

$$R_n \leftarrow *R_n \ *R_m$$

**Example 3.1.2** (Subtraction)
assembly:

$$subR_n, R_m$$

pseudocode:

$$R_n \leftarrow *R_n - *R_m$$

**Example 3.1.3** (XOR)
assembly:

$$xorR_n, R_m$$

pseudocode:

$$R_n \leftarrow *R_n^* R_m$$

**Example 3.1.4** (Compare)

assembly:

$$cmpR_n, R_m$$

pseudocode:

$$EFLAGS \leftarrow compare{*}R_n - {*}R_m$$

**Example 3.1.5** (increment)

assembly:

$$incrementR_n$$

pseudocode:

$$R_n \leftarrow {*}R_n \ 1$$

**Example 3.1.6** (Move: copy data from register $R_n$ into the absolute address in memory stored in $R_m$)

assembly:

$$movR_n, R_m$$

pseudocode:

$$Mem{*}R_n \leftarrow {*}R_m$$

Intuition for busses: The address bus carries the memory address from the processor to the memory. The data bus transfers the actual data between the processor and memory. The control bus carries control signals from the processor to coordinate the operations of the memory and I/O devices.

**Example 3.1.7** (Move: copy data from the absolute address in memory stored in $R_m$ into register $R_n$)

assembly:

$$movR_n, R_m$$

pseudocode:

$$R_n \leftarrow Mem{*}R_m$$

Intuition for busses: The address bus carries the memory address from the processor to the memory. The data bus transfers the actual data between the processor and memory. The control bus carries control signals from the processor to coordinate the operations of the memory and I/O devices.

**Example 3.1.8** (Jump if equals)

assembly:

$$jeqR_n$$

pseudocode:

$$ifZF == 1 \text{ then } PC \leftarrow {*}R_n$$

Intuition: read (jump to that address) $R_n$ if the zero flag is 1.

**Example 3.1.9** (move num to $R_n$)

assembly:

$$movR_n, num$$

pseudocode:

$$R_n \leftarrow num$$

**Question 1: Starting Conditions and Execution Flow**

Given the following initial program state and instructions:

- PC = 0010
- Mem0010 = `mov R0, 1010`
- Mem0011 = `mov R1, 0001`
- Mem0100 = `mov R4, 0110`
- Mem0101 = `mov R5, 1011`
- Mem0110 = `inc R1`
- Mem0111 = `cmp R0, R1`
- Mem1000 = `je R5`
- Mem1001 = `cmp R0, R0`
- Mem1010 = `je R4`
- Mem1011 = `halt`

Explain, step by step, how this program executes until `halt`.

***Solution:*** Here is the execution walkthrough:

1. **PC = 0010:** `mov R0, 1010`. This loads `1010` (binary) into $R_0$. Then PC ← 0011.
2. **PC = 0011:** `mov R1, 0001`. Loads `0001` into $R_1$. Then PC ← 0100.
3. **PC = 0100:** `mov R4, 0110`. Loads `0110` into $R_4$. Then PC ← 0101.
4. **PC = 0101:** `mov R5, 1011`. Loads `1011` into $R_5$. Then PC ← 0110.
5. **PC = 0110:** `inc R1`. Increments $R_1$ from `0001` to `0010`. Then PC ← 0111.
6. **PC = 0111:** `cmp R0, R1`. Compares $R_0$ (`1010`) with $R_1$ (`0010`). Since they differ, the zero/equal flag is cleared. Then PC ← 1000.
7. **PC = 1000:** `je R5` (jump if equal). The zero flag is 0, so no jump occurs. Then PC ← 1001.
8. **PC = 1001:** `cmp R0, R0`. Compares $R_0$ with itself, which sets the zero flag to 1. Then PC ← 1010.
9. **PC = 1010:** `je R4`. Because the zero flag is 1, we jump to the address stored in $R_4$, which is `0110`. So PC ← 0110, looping back.
10. The loop repeats from `inc R1` until eventually $R_1$ equals $R_0$ (`1010`). At that point, the compare (`cmp R0, R1`) sets the zero flag, and `je R5` at PC = 1000 finally jumps to $R_5$ = `1011`, which contains the `halt` instruction.

    This mechanism effectively keeps incrementing $R_1$ until it matches $R_0$, then halts.

## 3.2 Instruction Encoding for 16-bit Memory Cells

Have been talking about 16-bit memory cells. Thus, our instructions and their parameters need to be able to be described and differentiated in 16-bits.

### Encoding Rules

- **Zero-argument instructions**
  - `halt` (only one): `0000 0000 0000 0000`
- **Single-argument instructions**
  - First two bits are always `11`.
  - Use the first five bits to encode the instruction.
  - Use the remaining eleven bits to represent the argument.
- **Two-argument instructions**
  - Use the first four bits to encode the instruction.
  - Use the next six bits for the first argument.
  - Use the final six bits for the second argument.

## 3.3 Assemble

To assemble a program is to convert it from human-readable instructions to their binary representations

| Instruction | Binary Code | Description | Arguments | Example |
|---|---|---|---|---|
| `halt` | `0000 0000 0000 0000` | halt execution | none | `0000 0000 0000 0000` |
| `inc R_n` | `1100 0` | register R_n gets (contents of R_n) + 1 | 11-bit argument representing *n* | `inc R3`<br>`1100 0000 0000 0011` |
| `jmp num` | `1100 1` | jump to *relative* address *num* by adding *num* to the program counter | 11-bit argument representing *num* | `jmp 25`<br>`1100 1000 0001 1001` |
| `jne num` | `1101 0` | jump to relative address *num* if last `cmp` was unequal | 11-bit argument representing *num* | `jne 13`<br>`1101 0000 0000 1101` |
| `je [R_n]` | `1101 1` | jump to *absolute* address in R_n if last `cmp` was equal | 11-bit argument representing *n* | `je [R3]`<br>`1101 1000 0000 0011` |
| `add R_n, R_m` | `0001` | R_n gets (R_n + R_m) | Two 6-bit arguments (first = n, next = m) | `add R3, R5`<br>`0001 0000 1100 0101` |
| `sub R_n, R_m` | `0010` | R_n gets (R_n - R_m) | Two 6-bit arguments (first = n, next = m) | `sub R3, R2`<br>`0010 0010 0100 1000` |
| `xor R_n, R_m` | `0011` | R_n gets bitwise XOR of R_n and R_m | Two 6-bit arguments (first = n, next = m) | `xor R2, R3`<br>`0011 0000 1000 0011` |
| `cmp R_n, R_m` | `0100` | compare R_n and R_m; set flags accordingly | Two 6-bit arguments (first = n, next = m) | `cmp R5, R3`<br>`0100 0001 1100 0011` |
| `mov R_n, num` | `0101` | R_n gets the value *num* | Two 6-bit arguments (first = n, next = m) | `mov R5, 61`<br>`0101 0001 1111 1101` |
| `mov R_n, R_m` | `0110` | R_n gets the contents of R_m | Two 6-bit arguments (first = n, next = m) | `mov R3, R1`<br>`0110 0001 0100 0001` |
| `mov [R_n], R_m` | `0111` | copy contents of R_m to memory address stored in R_n | Two 6-bit arguments (first = n, next = m) | `mov [R3], R1`<br>`0111 0001 0100 0001` |
| `mov R_n, [R_m]` | `1000` | copy from memory address in R_m to R_n | Two 6-bit arguments (first = n, next = m) | `mov R3, [R1]`<br>`1000 0001 0100 0001` |

Table 3.1: Summary of instructions, binary codes, and examples.

## 3.4 Binary to Hexadecimal Conversion

Converting binary numbers to hexadecimal is straightforward because each hexadecimal digit represents four binary digits (bits). Here are the steps:

1. **Group the binary digits into sets of four**, starting from the right. Add leading zeros if necessary to make a complete set.
2. **Convert each group of four binary digits to its hexadecimal equivalent** using the following table:

| Binary | Hexadecimal |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

**Example:** Convert the binary number 11010110 to hexadecimal.

1. Group the binary digits: 1101 0110.
2. Convert each group:
$$1101 = D, \quad 0110 = 6$$
3. Combine the hexadecimal digits: $D6$.

Thus, $11010110_2 = D6_{16}$.

## 3.5 Machine Code Decoding

### 3.5.1 Hexadecimal Machine Code

```
5000 504A 5094 7040 C000 4042 D7FC 0000
```

### 3.5.2 Instruction Breakdown

- **5000**
  - Binary: 0101 0000 0000 0000
  - Instruction: `MOV R0, 0`
- **504A**
  - Binary: 0101 0000 0100 1010
  - Instruction: `MOV R1, 10`
- **5094**

- Binary: 0101 0000 1001 0100
- Instruction: `MOV R2, 20`

- **7040**
  - Binary: 0111 0000 0100 0000
  - Instruction: `MOV [R1], R0`

- **C000**
  - Binary: 1100 0000 0000 0000
  - Instruction: `INC R0`

- **C001**
  - Binary: 1100 0000 0000 0001
  - Instruction: `INC R1`

- **4042**
  - Binary: 0100 0000 0100 0010
  - Instruction: `CMP R1, R2`

- **D7FC**
  - Binary: 1101 0111 1111 1100
  - Instruction: `JNE -4`

- **0000**
  - Binary: 0000 0000 0000 0000
  - Instruction: `HALT`

### 3.5.3 Program Behavior

1. `MOV R0, 0` sets register `R0` to 0.
2. `MOV R1, 10` sets register `R1` to 10.
3. `MOV R2, 20` sets register `R2` to 20.
4. `MOV [R1], R0` writes the contents of `R0` into memory at the address held by `R1`.
5. `INC R0` increments `R0`.
6. `INC R1` increments `R1`.
7. `CMP R1, R2` compares `R1` and `R2`.
8. `JNE -4` jumps back 4 instructions if `R1` is not equal to `R2`.
9. `HALT` ends execution.

**Summary:** This program uses a loop to:

- Increment `R0` and `R1`
- Store `R0`'s value at memory address `R1`
- Stop once `R1` reaches `R2` (i.e., 20)

In effect, it writes an increasing sequence of values $(0, 1, 2, \ldots)$ into consecutive memory locations starting at address 10, and halts once it has written the value 9 at address 19 (because at that point `R1` becomes 20, matching `R2`, so the loop ends).

## 3.6 Fetch-Decode-Execute-Store Cycle

The fetch-decode-execute-store cycle is the fundamental process by which a computer executes instructions. This cycle involves several key components: the Program Counter (PC), Instruction Register (IR), and various other registers and buses.

### 3.6.1 Fetch

1. The PC holds the address of the next instruction to be executed. 2. The address is sent to memory via the address bus. 3. The instruction at that address is fetched from memory and loaded into the IR.

### 3.6.2 Decode

1. The control unit reads the instruction in the IR. 2. The instruction is decoded to determine the operation to be performed and the operands involved.

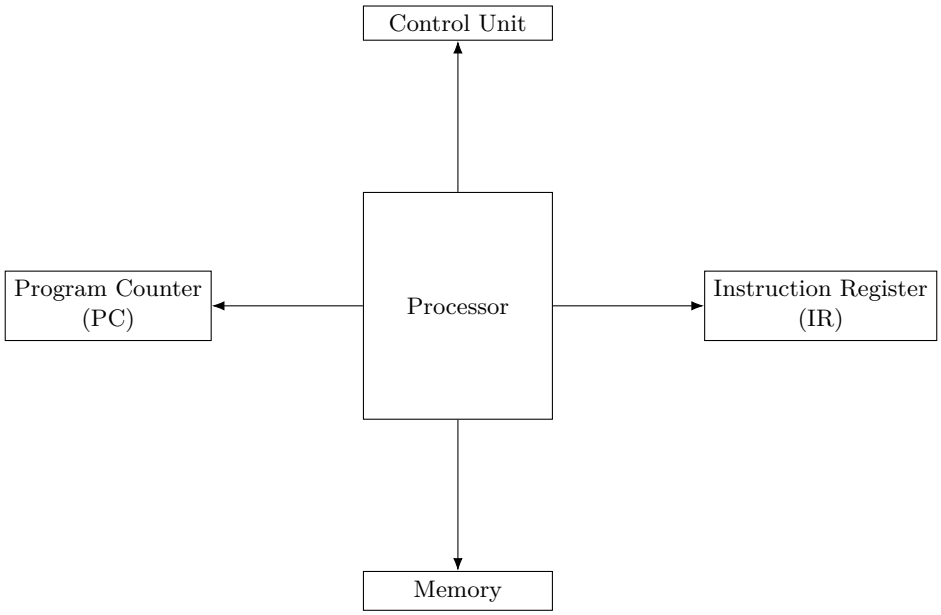
### 3.6.3 Execute

1. The control unit signals the appropriate components (ALU, registers, etc.) to perform the operation. 2. The ALU or other execution units carry out the operation using the operands.

### 3.6.4 Store

1. The result of the operation is stored in the appropriate location (register or memory). 2. The PC is updated to point to the next instruction, and the cycle repeats.

**Diagram:**




This cycle ensures that instructions are processed sequentially and efficiently, allowing the computer to perform complex tasks by breaking them down into simpler operations.

## 3.7 Five-Stage Pipeline for Operations

The five-stage pipeline is a common technique used in computer architecture to improve instruction throughput. Each instruction passes through five distinct stages: Fetch, Decode, Execute, Memory Access, and Write Back. Below are the details of each stage:

### 3.7.1 1. Fetch (IF)

**Instruction Fetch:** - The instruction is fetched from memory. - The Program Counter (PC) holds the address of the instruction to be fetched. - The fetched instruction is stored in the Instruction Register (IR).

**Key Points:** - The PC is incremented to point to the next instruction. - This stage involves reading from memory, which can introduce delays if the memory access time is significant.

### 3.7.2 2. Decode (ID)

**Instruction Decode:** - The fetched instruction is decoded to determine the operation and the operands. - The control unit generates the necessary control signals based on the decoded instruction. - The source operands are read from the register file.

**Key Points:** - This stage involves interpreting the binary instruction and preparing the necessary data for execution. - The control signals generated will guide the subsequent stages.

### 3.7.3 3. Execute (EX)

**Execution:** - The actual operation specified by the instruction is performed. - This could involve arithmetic or logical operations performed by the Arithmetic Logic Unit (ALU). - For branch instructions, the target address is calculated.

**Key Points:** - The ALU performs the required computation. - The result of the computation is temporarily stored for the next stage.

### 3.7.4 4. Memory Access (MEM)

**Memory Access:** - If the instruction involves memory access (e.g., load or store), the memory address is accessed. - For load instructions, data is read from memory and stored in a temporary register. - For store instructions, data is written to the specified memory address.

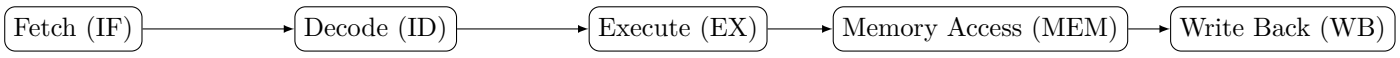
**Key Points:** - This stage is crucial for instructions that interact with memory. - Memory access can introduce delays due to varying memory access times.

### 3.7.5 5. Write Back (WB)

**Write Back:** - The result of the instruction is written back to the register file. - This updates the destination register with the computed value or the data read from memory.

**Key Points:** - This stage ensures that the results of the instruction are stored for future use. - The pipeline is now ready to process the next instruction.

**Diagram:**




The five-stage pipeline allows for overlapping execution of multiple instructions, significantly improving the instruction throughput and overall performance of the processor.

### 3.7.6 Example of parallel processing in 5 stage pipeline

## 3.8 Stalling and Forwarding in a 5-Stage Pipeline

In a 5-stage pipeline, conflicts can arise when instructions that depend on the results of previous instructions are executed concurrently. Two common methods to overcome these conflicts are stalling and forwarding.

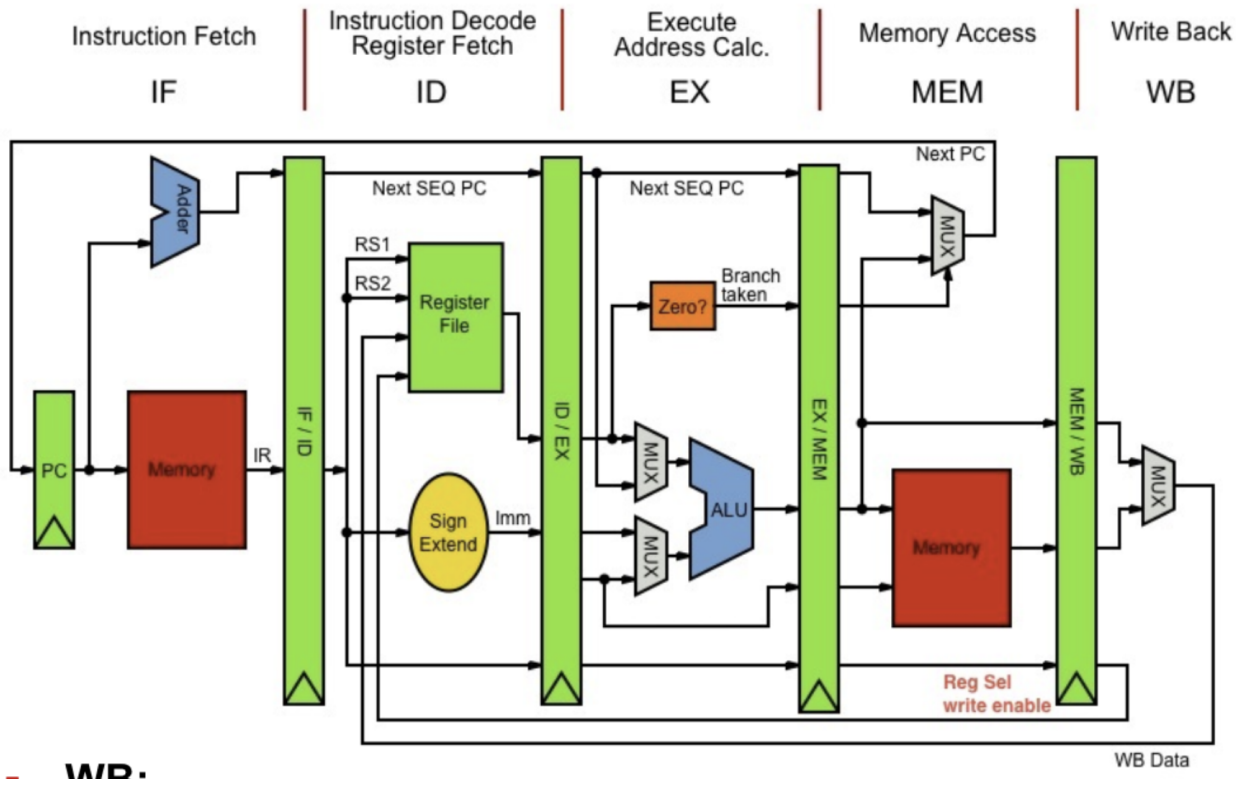



Figure 3.1: Five-Stage Pipeline for Operations

| Cycle | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| 1 | MOV R1 [R2] | | | | |
| 2 | ADD R3 R4 | MOV R1 [R2] | | | |
| 3 | MOV [R5] R6 | ADD R3 R4 | MOV R1 [R2] | | |
| 4 | | MOV [R5] R6 | ADD R3 R4 | MOV R1 [R2] | |
| 5 | | | MOV [R5] R6 | ADD R3 R4 | MOV R1 [R2] |
| 6 | | | | MOV [R5] R6 | ADD R3 R4 |
| 7 | | | | | MOV [R5] R6 |

Table 3.2: Pipeline Execution Table

### 3.8.1 Stalling

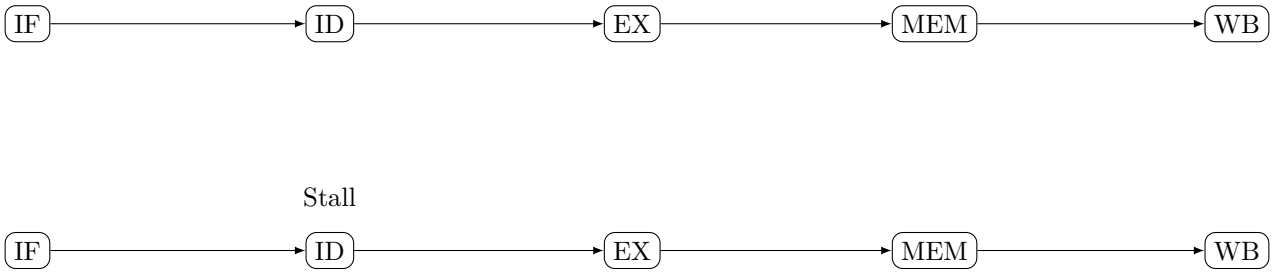
Stalling, also known as pipeline interlock, involves pausing the pipeline until the data required by an instruction is available. This method ensures that instructions are executed in the correct order, but it can reduce the overall performance due to idle cycles.

**Example:** Consider the following sequence of instructions:

```
1. MOV R1, [R2]
2. ADD R3, R1
```

In this case, the ADD instruction depends on the result of the MOV instruction. To resolve this conflict, the pipeline can be stalled until the MOV instruction completes.

**Diagram:**




### 3.8.2 Forwarding

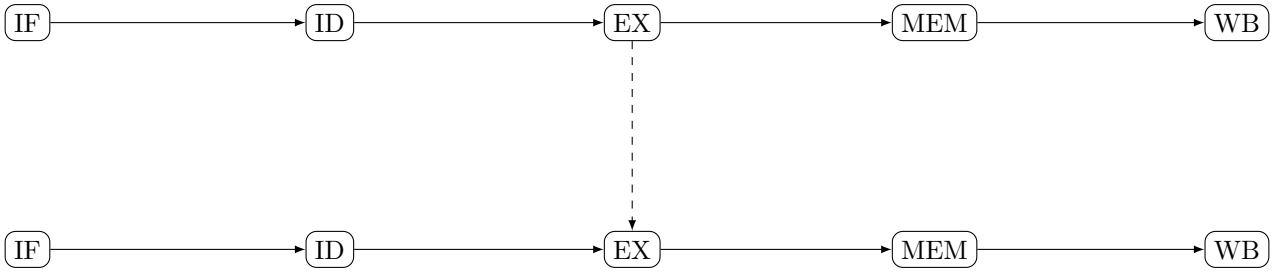
Forwarding, also known as bypassing, involves passing the result of an instruction directly to a subsequent instruction that needs it, without waiting for it to be written back to the register file. This method reduces the number of idle cycles and improves performance.

**Example:** Consider the following sequence of instructions:

```
1. MOV R1, [R2]
2. ADD R3, R1
```

In this case, the result of the MOV instruction can be forwarded directly to the ADD instruction, allowing it to proceed without stalling.

**Diagram:**




By using stalling and forwarding, the pipeline can handle data hazards and ensure correct execution of instructions while maintaining high performance.

## 3.9 Handling Simultaneous Memory Reads in IF and WB Stages

In a pipelined processor, simultaneous memory reads in the Instruction Fetch (IF) and Write Back (WB) stages can lead to conflicts and performance degradation. Two common approaches to address this issue are stalling and using the Harvard architecture.

### 3.9.1 Stalling

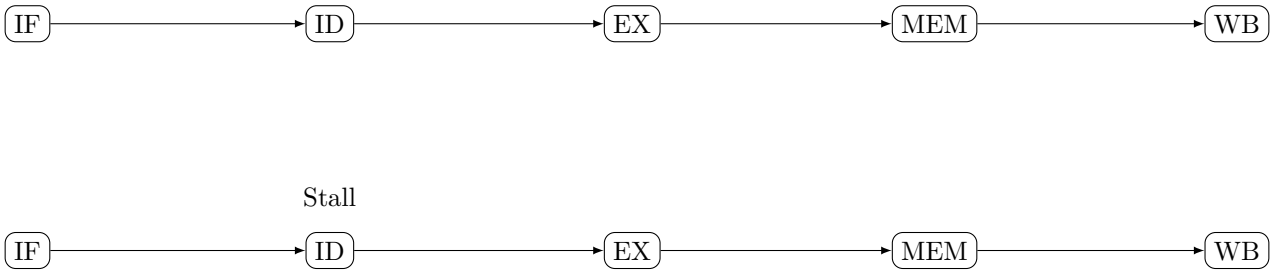
Stalling involves pausing the pipeline to resolve memory access conflicts. When a memory read conflict is detected, the pipeline is stalled until the memory access is completed. This ensures that the correct data is read, but it can reduce overall performance due to idle cycles.

**Example:** Consider the following sequence of instructions:

```
1. MOV R1, [R2]
2. ADD R3, R1
```

If both instructions require memory access at the same time, the pipeline can be stalled to resolve the conflict.

**Diagram:**

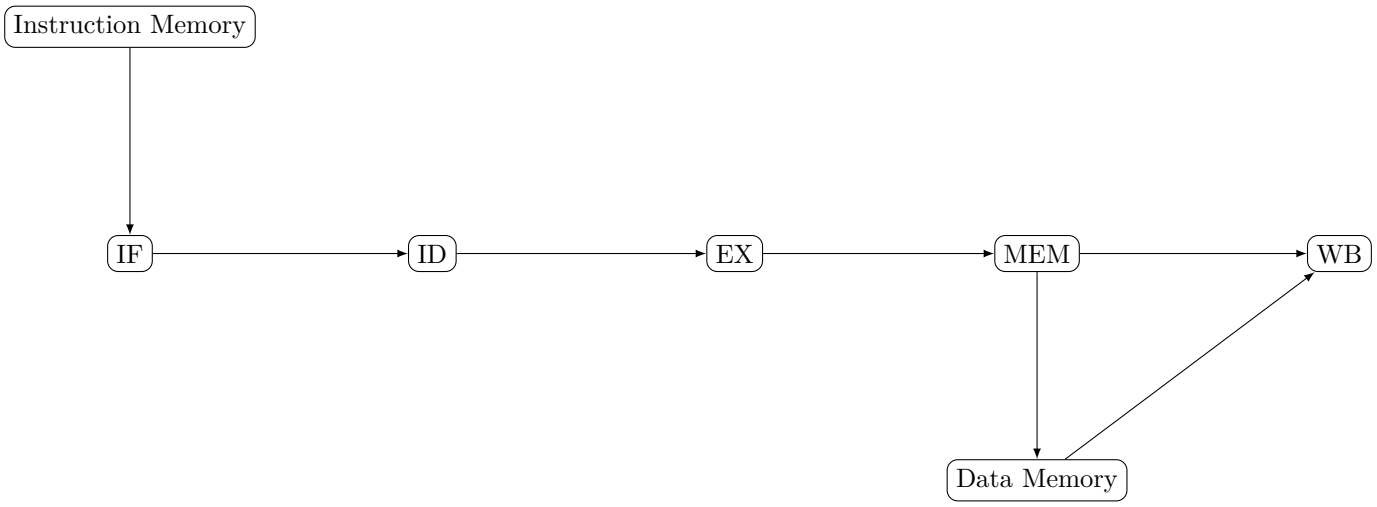



### 3.9.2 Harvard Architecture

The Harvard architecture addresses memory access conflicts by using separate memory spaces for instructions and data. This allows simultaneous access to both instruction and data memory, eliminating conflicts and improving performance.

**Example:** In the Harvard architecture, the instruction memory and data memory are accessed independently, allowing the IF stage to fetch instructions while the WB stage writes data.

**Diagram:**




By using separate memory spaces for instructions and data, the Harvard architecture allows for more efficient memory access and reduces the need for stalling, leading to improved overall performance.

## 3.10 Memory Pointers

- When we get into C (and C++ later), we will talk a **lot** about memory indirection (pointers).
- But you have already been exposed to the concept with our minimal Instruction Set!
- **What is memory indirection??**
- Which one feels like memory indirection?
  - `MOV R1 [0x27]`
  - `MOV R1 [R2]`
- In the first case, we are reading an absolute memory address into `register-1`.
- In the second case, we are reading the memory address specified by the value of `register-2` into `register-1`.
- **The second case is memory indirection, and `register-2` is acting as a pointer!**

## 3.11 Memory Indirection in C++

```
int numbers[] = {10, 20, 30, 40, 50};
int size = sizeof(numbers) / sizeof(numbers[0]);
// Pointer to the first element of the array
int *ptr = numbers;
for (int i = 0; i < size; i++) {
 printf("Element %d: %d\n", i, *(ptr + i));
}
```

**Explanation:**

This code demonstrates memory indirection using pointers in C. Here's a breakdown of how it works:

1. `int numbers[] = {10, 20, 30, 40, 50};` - An array of integers is declared and initialized with values.

2. `int size = sizeof(numbers) / sizeof(numbers[0]);` - The size of the array is calculated by dividing the total size of the array by the size of one element.

3. `int *ptr = numbers;` - A pointer `ptr` is declared and initialized to point to the first element of the array.

4. `for (int i = 0; i < size; i++) {` - A loop is used to iterate through the array elements.

5. The value of each element is accessed using the pointer `ptr` with memory indirection (`*(ptr + i)`) and printed.

**Memory Indirection:** - The expression `*(ptr + i)` uses memory indirection to access the value at the memory address pointed to by `ptr` plus `i`. - This allows accessing array elements using pointer arithmetic, demonstrating how pointers can be used to indirectly access memory locations.

# Chapter 4

# OS and Assembly

## 4.1 Why is an OS needed

## 4.2 Why is an OS needed

An operating system (OS) is essential for several reasons:

1. **Resource Management:** - The OS manages hardware resources such as CPU, memory, and I/O devices, ensuring efficient and fair allocation among multiple programs.

2. **Program Execution:** - The OS provides an environment for executing programs, handling tasks such as loading programs into memory, scheduling CPU time, and managing program execution.

3. **Memory Management:** - The OS handles memory allocation and deallocation, ensuring that programs do not interfere with each other's memory space. This prevents issues such as memory corruption, where programs might write data into memory cells containing instructions, leading to unpredictable behavior and system crashes.

4. **File System Management:** - The OS manages files and directories on storage devices, providing a structured way to store, retrieve, and organize data.

5. **Security and Access Control:** - The OS enforces security policies, controlling access to system resources and protecting against unauthorized access and malicious activities.

6. **Error Handling:** - The OS detects and handles errors, providing mechanisms for recovering from hardware and software failures.

7. **User Interface:** - The OS provides a user interface, such as a command-line interface (CLI) or graphical user interface (GUI), allowing users to interact with the system and perform tasks.

8. **Device Management:** - The OS manages device drivers, facilitating communication between the system and peripheral devices such as printers, keyboards, and network adapters.

Overall, the OS acts as an intermediary between users, applications, and hardware, ensuring the smooth and efficient operation of the computer system.

## 4.3 Memory Virtualization

Memory virtualization is a technique used by operating systems to provide an abstraction of physical memory, allowing each process to have its own virtual address space. This improves security, isolation, and efficient use of memory. Here are the key concepts:

1. **Virtual Address Space:** - Each process is given a virtual address space, which is a contiguous range of addresses that the process can use. - The virtual address space is mapped to physical memory by the OS.

    - **Virtual Memory** allows the OS to use disk space as an extension of physical memory by assigning more memory to its programs than it has in RAM.

2. **Page Tables:** - The OS uses page tables to keep track of the mapping between virtual addresses and physical addresses. - Each entry in the page table corresponds to a page (a fixed-size block of memory) and contains the physical address of the page.

3. **Paging:** - Memory is divided into fixed-size pages (e.g., 4 KB). - Virtual memory is also divided into pages of the same size. - When a process accesses a virtual address, the OS translates it to a physical address using the page table.

4. **Page Faults:** - If a process accesses a virtual address that is not currently mapped to physical memory, a page fault occurs. - The OS handles the page fault by loading the required page from disk into physical memory and updating the page table.

5. **Swapping:** - When physical memory is full, the OS may swap out pages that are not currently in use to disk, freeing up physical memory for other pages. - Swapped-out pages are stored in a swap space on disk and can be swapped back into physical memory when needed.

6. **Protection:** - Memory virtualization provides isolation between processes, preventing one process from accessing the memory of another process. - The OS enforces access control by setting permissions on pages (e.g., read, write, execute).

7. **Translation Lookaside Buffer (TLB):** - The TLB is a cache used to speed up the translation of virtual addresses to physical addresses. - It stores recent translations, reducing the need to access the page table for every memory access.

## 4.4 Context Switching

Context switching is the process of saving and restoring the state of a CPU so that multiple processes can share the CPU effectively. Even if the number of cores that are available are greater than the number of processes, context switching is still necessary to handle interrupts and system calls. Here are the key steps involved in context switching:

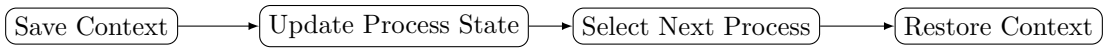
1. **Save the Context:** - The OS saves the state of the currently running process, including the contents of registers, program counter, and other CPU state information. - This state is stored in a data structure called the process control block (PCB).

2. **Update the Process State:** - The OS updates the state of the currently running process to indicate that it is no longer running (e.g., changing its state to ”waiting” or ”ready”).

3. **Select the Next Process:** - The OS selects the next process to run based on a scheduling algorithm (e.g., round-robin, priority-based). - The selected process's state is updated to indicate that it is now running.

4. **Restore the Context:** - The OS restores the state of the selected process from its PCB, including the contents of registers and program counter. - The CPU is now ready to execute the selected process.

**Diagram:**



# Chapter 5

# C and C++

## 5.1 C Compilation

- **Preprocessing:**
  - The preprocessor processes directives (e.g., `#include`, `#define`) and expands macros.
  - The output is a translation unit with all preprocessor directives processed.
- **Compilation:**
  - The compiler translates the preprocessed source code into assembly code or object code.
  - The output is an object file containing machine code instructions.
- **Assembly:**
  - The assembler translates the object code into machine code.
  - The output is an executable file that can be run on the target platform.
- **Linking:**
  - The linker combines object files and libraries to create an executable file.
  - It resolves external references, assigns memory addresses, and generates the final executable.
- **Execution:**
  -
  - The operating system loads the executable into memory and starts its execution.

**Compiled languages:** Translated to machine code before execution.
**Interpreted Languages:** Translated at runtime.

### 5.1.1 Types and Casting

```
int c = a / b;
printf("c is %d \n", c);
```

**Example 5.1.1** (Integer Division Truncation)
Dividing two integers `a` (5) and `b` (2) results in truncation to an integer. The decimal part is discarded, leaving `c` with value 2.

```
float d = 8 / 3;
printf("d is %f \n", d);
```

**Example 5.1.2** (Implicit Integer Division)

The expression `8/3` performs integer division first (resulting in 2), then assigns to float `d`. This demonstrates that division occurs before type conversion.

```
float e = (float)a / b;
printf("e is %f \n", e);
```

**Example 5.1.3** (Explicit Casting for Floating-Point Division)

Casting `a` to `float` before division forces floating-point arithmetic. This produces a precise result (2.5) instead of truncation.

**Definition 5.1.1: Type Conversion in Division**

When at least one operand of the division operator (`/`) is a floating-point type, the result will be a floating-point value. Explicit casting can enforce this behavior.

```
int g = a;
printf("g is %d \n", g);
```

**Note:-**

Assigning an integer to another integer preserves its value. This example appears redundant but demonstrates direct assignment between matching types.

```
float h = int(f) / 2;
printf("h is %f \n", h);
```

**Example 5.1.4** (Double Truncation Example)

First truncates `f` (3.14159) to integer 3 via explicit cast, then performs integer division `3/2 = 1`. Finally assigns to float `h` as 1.0, showing two truncation steps.

**Theorem 5.1.1** Truncation Hierarchy

Truncation occurs at each explicit cast or integer division operation. The order of operations determines whether decimal values are preserved or discarded.

**Question 2: Why does `h` equal 1.0?**

Explain why `int(f)/2` produces 1.0 instead of 1.5 when stored in a float.

***Solution:*** The cast operation `int(f)` happens first, converting 3.14159 to integer 3. Then integer division `3/2` produces 1. Finally, this integer result is implicitly converted to float 1.0 during assignment to `h`.

## 5.2 Fractions

**Definition 5.2.1: Floating-Point Representation in C++**

In C++, floating-point numbers are represented according to the IEEE 754 standard. A single-precision (float) number consists of:

- **Sign bit (1 bit)**: Determines if the number is positive or negative.
- **Exponent (8 bits)**: Encodes the exponent in a biased format.
- **Mantissa (23 bits)**: Also known as the significand; represents the precision bits of the number.

The value of a floating-point number is computed as:

$$-1^{\text{sign}} \times 2^{\text{exponent}-\text{bias}} \times 1 \text{ mantissa}$$

For double-precision (double) numbers, the sizes differ (1 sign bit, 11 exponent bits, and 52 mantissa bits).

**Note:-**

Floating-point arithmetic can introduce precision errors due to the finite number of bits used to represent real numbers. Operations that are mathematically exact might yield unexpected results in computer arithmetic.

**Example 5.2.1** (Demonstration of Floating-Point Precision Loss)

Consider the following code that demonstrates the loss of precision when performing floating-point arithmetic:

```
void showFloatLossiness()
{
    double a = 0.1;
    double b = 0.2;
    double c = a + b;
    double d = c - b;
    printf("a: %.20f\n", a);
    printf("b: %.20f\n", b);
    printf("c: %.20f\n", c);
    printf("d: %.20f\n", d);
    printf("a == d: %s\n", a == d ? "true" : "false");
}
```

When running this code, you might get the following output:

```
a: 0.10000000000000000555
b: 0.20000000000000001110
c: 0.30000000000000004441
d: 0.10000000000000000555
a == d: true
```

This output shows that even simple arithmetic operations can result in floating-point numbers that have small differences from their expected values due to representation errors.

**Note:-**

The loss of precision occurs because certain decimal fractions cannot be represented exactly in binary floating-point format. Numbers like 0.1 and 0.2 have infinite repeating binary representations, so they are approximated in a finite number of bits.

**Definition 5.2.2: Converting Decimal Fractions to Binary**

Converting a decimal fraction to binary involves multiplying the fractional part by 2 and recording the integer part at each step. This process repeats until the fractional part becomes zero or until a sufficient level of precision is achieved.

**Example 5.2.2** (Example: Binary Representation of 12.5 as a Float)

Let's consider the decimal number 12.5 and examine its binary floating-point representation:

1. **Sign**: Positive, so the sign bit is 0.
2. **Binary Form**: $12.5_{10} = 1100.1_2$.
3. **Normalized Form**: Shift the binary point to after the first 1: $1.1001_2 \times 2^3$.
4. **Exponent**: The exponent is 3. Adding the bias (127 for single-precision float): 3 127 = 130, which is $10000010_2$ in binary.
5. **Mantissa**: The mantissa is the fractional part after the leading 1: 1001, padded with zeros to fill 23 bits.

Thus, the complete IEEE 754 binary representation is:

Sign bit : 0 Exponent : 10000010 Mantissa : 10010000000000000000000

**Note:-**

Due to limited precision, only a finite number of decimal fractions can be represented exactly in binary floating-point format. This limitation often leads to small rounding errors in calculations.

**Example 5.2.3** (Converting a Decimal Number to a Binary Fraction in C++)

The following function converts a decimal number to its binary string representation, including the fractional part:

```
std::string convertToBinaryFraction(double num)
{
    // Check input bounds.
    if (num < 0)
    {
        return "Negative numbers not supported!";
    }
    if (num > 4294967295.0)
    {
        return "Input number too large!";
    }

    unsigned int intPart = static_cast<unsigned int>(num);
    double fracPart = num - intPart;

    // Convert integer part to binary.
    std::string intStr;
    if (intPart == 0)
    {
        intStr = "0";
    }
    else
```

```
    {
        while (intPart > 0)
        {
            intStr = std::to_string(intPart % 2) + intStr;
            intPart /= 2;
        }
    }

    // Convert fractional part to binary.
    std::string fracStr = ".";
    if (fracPart > 0)
    {
        int count = 0;
        while (fracPart > 0 && count < (32 - intStr.length()))
        {
            fracPart *= 2;
            if (fracPart >= 1)
            {
                fracStr += "1";
                fracPart -= 1;
            }
            else
            {
                fracStr += "0";
            }
            count++;
        }
    }
    else
    {
        fracStr += "0";
    }

    return intStr + fracStr;
}
```

This function handles:

- **Input validation**: Checks for negative numbers and excessively large inputs.
- **Integer part conversion**: Converts the integer part to binary by repeatedly dividing by 2.
- **Fractional part conversion**: Converts the fractional part to binary by multiplying by 2 and extracting the integer part.
- **Precision control**: Limits the number of iterations to prevent infinite loops with non-terminating binary fractions (e.g., 13).

**Note:-**

When converting fractions like 13 to binary, the fractional part does not terminate, resulting in infinite binary expansions. The function limits the precision based on the length of the integer part to mitigate excessive computation while providing a reasonable approximation.

**Theorem 5.2.1** Limitations of Binary Fraction Conversion

Certain decimal fractions have infinite, non-repeating binary representations, similar to how 13 is infinite and non-repeating in decimal. Consequently, these numbers cannot be represented exactly in binary, leading to approximation errors in computations.

## Types and Type Safety

```
unsigned int aa = 0x12345678;
printf("Size of x is %zu bytes \n", sizeof(aa));
```

**Definition 5.2.3: Type Safety**

Degree to which a programming language prevents type errors. Contains two approaches:

- Weak typing: Flexible type interactions (may cause unexpected behaviors)
- Strong typing: Strict type enforcement (C++ uses this approach)
  - Prevents operations between incompatible types
  - Enables better performance through optimized type handling

**Definition 5.2.4: Type Checking**

Process of verifying type constraints. Two main approaches:

- Static checking (compile-time):
  - Used in C++
  - Catches errors before execution
  - Enables optimizations
- Dynamic checking (runtime):
  - Type verification during execution
  - Adds runtime overhead

```
int x = 2;
double y = 3.5;
double result = x + y;
printf("Result is %f \n", result);
```

**Example 5.2.4** (Implicit Type Conversion)

Demonstrates C++'s lenient type safety through *arithmetic conversions*:

- `int x` promoted to `double` before addition
- Conversion hierarchy: `int` → → `double`
- Result maintains highest precision type (`double`)

```
int a = -1;
unsigned int b = 1;
if (a < b) {
printf("a is less than b \n");
} else {
printf("a is not less than b \n");
}
```

**Example 5.2.5** (Unsigned Conversion Pitfall)

Illustrates unexpected behavior with signed/unsigned comparisons:

- `int a` converted to `unsigned int` via *integral promotions*
- -1 becomes 4294967295 (assuming 4-byte int)
- Comparison 4294967295 ¡ 1 evaluates false

**Theorem 5.2.2** Conversion Hierarchy Rule

When operands have different types, the compiler will:

1. Promote smaller types to larger equivalents (`char` → → `int`)
2. Convert signed to unsigned if unsigned operand present
3. Convert to floating-point if either operand is floating-point

```
bool c = x < y;
printf(c ? "true" : "false");
```

**Note:-**

C++ allows comparisons between different arithmetic types:

- Compiler performs implicit conversions before comparison
- Conversion follows same hierarchy as arithmetic operations
- Result always `bool` type regardless of operand types

**Question 3: Why does -1 appear larger than 1?**

Explain the counterintuitive result in the signed/unsigned comparison example.

***Solution:*** The comparison converts both operands to unsigned integers. The signed value -1 becomes $2^{32} - 1$ (maximum unsigned int value) in two's complement systems, making it numerically larger than 1. Always use explicit casts when comparing signed and unsigned values.

## 5.3 Pointers

```
int square(int n) {
int result = n * n;
return result;
}
// ... other function definitions ...
int exampleCallStack() {
int a = 3, b = 4, c = 5;
int result = sumOfSquares(a, b, c);
std::cout << result << std::endl;
return 0;
}
```

**Definition 5.3.1: Memory Segments in C++**

Program memory is divided into three main segments:

- **Call Stack:**
  - Stores function call frames with arguments, local variables, and return addresses
  - Automatically managed (LIFO structure)
  - Fast allocation but limited size
- **Heap:**
  - Manual memory management via `new/delete`
  - Larger capacity but slower access
  - Memory persists until explicitly freed
- **Data Segment:**
  - Contains global and static variables
  - Allocated for program's entire lifetime

**Example 5.3.1** (Call Stack Operation)

Demonstrates stack frame creation during nested function calls:

- Each function call creates a new frame
- Frames contain parameters and local variables
- Return values propagate back through frames
- Frames destroyed in reverse creation order

**Note:-**

Recursive Functions

In recursion, each recursive call creates a new stack frame. Deep recursion without proper base cases can lead to:

- Stack overflow from excessive frame accumulation
- Memory corruption and program crashes

```
int *counterPointer = &counter;
std::cout << "value at counterPointer:" << *counterPointer;
```

**Example 5.3.2** (Pointer Basics)

Illustrates fundamental pointer operations:

- Address-of operator (&) gets variable's memory location
- Dereference operator (*) accesses value at address
- Pointers enable indirect memory manipulation

**Definition 5.3.2: Pointer Characteristics**

- Fixed size (typically 4/8 bytes depending on architecture)
- Enable pass-by-reference semantics
- Require explicit dereferencing for value access
- Can be reassigned to point to different variables

```
*counterPointer = 15; // Direct memory modification
counterPointer = \&maxValue; // Pointer reassignment
```

**Example 5.3.3** (Pointer Manipulation)

Demonstrates two key pointer capabilities:

- Value modification through dereferenced pointers
- Address reassignment to different variables
- Shows pointer/variable synchronization

```
void increment(int *count) { *count += 1; }
```

**Example 5.3.4** (Pass-by-Reference)

Implements modified parameter passing:

- Avoids copying large data structures
- Requires explicit address-of operator (&) at call site
- Enables direct memory modification in callee

**Note:-**

Pointer Safety

Always ensure:

- Pointers remain valid during dereferencing
- No returns of pointers to stack-allocated variables
- Proper null-checking before dereferencing

```
int *roundedResult = roundUp(input);
std::cout << *roundedResult; // Dangerous!
```

**Example 5.3.5** (Dangling Pointer Example)

Demonstrates common pointer error:

- Returns address of stack-allocated variable
- Memory becomes invalid after function return
- Accessing invalid memory causes undefined behavior

**Theorem 5.3.1** Pointer Lifetime Rule

A pointer must never outlive:

1. The memory region it points to
2. The scope where the referenced variable was declared
3. Any parent stack frames in call hierarchy

**Question 4: Why does roundUp() produce garbage values?**

Explain the memory safety issue in the roundUp() function example.

***Solution:*** The function returns a pointer to a local stack variable (`roundedUp`). When the function returns:

- Its stack frame is destroyed
- The pointer now references invalid memory
- Subsequent dereferencing accesses corrupted data

Proper solution: Allocate heap memory or modify parameter directly.

## 5.4 Structs and Arrays

```
enum Semester { FALL, SPRING };
```

**Definition 5.4.1: Enumeration Type**

Represents a fixed set of named constants:

- Improves code readability
- Provides type safety for discrete values
- Underlying type is integer (implicit conversion possible)

```
struct BUClass {
    string name;
    int year;
    Semester semester;
};
```

**Example 5.4.1** (Class Structure Definition)

Models academic course information:

- Contains name, year, and semester fields
- Demonstrates composition of primitive types and enums
- Memory efficient (fixed size structure)

```
struct Student {
    string firstName;
    string lastName;
    int graduationYear;
    float gpa;
    BUClass *classes[10];  // Array of pointers
    int classesSize = 0;
};
```

**Theorem 5.4.1** Pointer Array Storage

When storing object associations:

- Pointer arrays prevent expensive copies
- Fixed size (10 elements here) requires bounds checking
- Requires explicit size tracking (classesSize counter)

```
void registerForClass(Student *student, BUClass buClass) {
    student->classes[student->classesSize] = &buClass;
    student->classesSize++;
}
```

**Example 5.4.2** (Pass-by-Reference Pitfall)

Illustrates common pointer error:

- Stores address of stack-allocated parameter
- &buClass becomes invalid after function return
- Should store pointers to heap-allocated objects instead

```
int main5() {
    int finalGrades[] = {91, 83, 78, 84, 73};
    printArrayInfo(finalGrades);  // Prints 2 instead of 5
}
```

**Note:-**

Array Parameter Handling

Arrays decay to pointers when passed to functions:

- `sizeof(array)` returns pointer size, not array size
- Always pass array size explicitly
- Alternative: Use `std::array` or `std::vector`

```
void probabilisicallyRegisterForClass(Student *student,
                                      BUClass *buClass,
                                      float probability) {
    if (randomFloatInRange(0.0, 1.0) <= probability) {
        student->classes[student->classesSize] = buClass;
        student->classesSize++;
    }
}
```

**Example 5.4.3** (Probabilistic Registration)

Demonstrates pointer-based association management:

- Uses Monte Carlo simulation for class registration
- Maintains references to global BUClass objects
- Shows pointer array manipulation with size tracking

**Question 5: Why does `printArrayInfo` give wrong size?**

Explain the array size calculation discrepancy in the example.

***Solution:*** Arrays decay to pointers when passed to functions. The `sizeof` operator in `printArrayInfo` returns:

- `sizeof(int*)` = 8 bytes (pointer size)
- `sizeof(int)` = 4 bytes
- 8/4 = 2 instead of actual array size 5

Always pass array size explicitly for functions processing arrays.

**Theorem 5.4.2** Memory Management Rules

When working with pointer arrays:

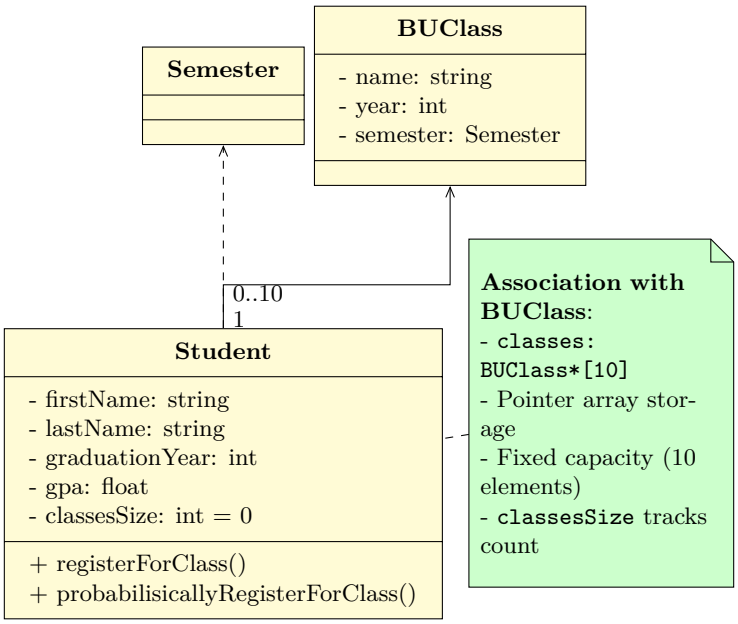
1. Ensure pointed objects outlive the array
2. Prefer stack allocation for short-lived objects
3. Use heap allocation (`new`/`delete`) for dynamic lifetime
4. Never store pointers to temporary values

```
std::cout << "student registered for at least 4 classes: "
          << students[index].lastName << std::endl;
```

**Note:-**

String Concatenation Optimization

- `std::to_string` creates temporary strings
- Multiple concatenations can be expensive
- Consider using `std::ostringstream` for complex formatting



## 5.5 Testing

### 5.5.1 Why is it needed?

- Identify bugs/defects before users do
- Maintain system malleability
- Gain confidence that system maintains functionality after changes are made:
  - small – incremental fixes/features
  - large – a refactoring
- Enables continuous integration/deployment
  - Continuous Testing
    - Rich automated test suite - run it after every change
  - Continuous Integration
    - Merge your code into a single source of truth frequently
  - Continuous Deployment
    - Deploy your changes to production frequently
- Gives us confidence to ship code frequently

### How was it done

**Waterfall-Model:** Development was done by a team, and then that code would move on to test engineers while the dev engineers move on to next task. Issue: if issues are found, the testers have to go back to the devs, interrupts their development and they have lost context.
**Agile-Model:** Development is done by a team, but the team is made up of developers who develop, test and deploy.

### 5.5.2 Types of test

- Unit testing
  - Fast
  - Cheap
  - Deterministic
  - Focused
  - White box
- Service testing
  - Slower
  - Less Deterministic
  - More expensive
  - Broader
  - Black box
- UI testing
  - End to End
  - Much more noise -¿ less deterministic
  - More expensive
  - Broadest
  - Black box

### 5.5.3 Testing in C++

```
//#include <gtest/gtest.h>
//#include "hw2_problem2.h"

TEST(ScaleTest, ZeroScale)
{
    // Create a rectangle with bottom-left corner at (1, 3) and top-right corner at (5, 9).
    // This represents a rectangle with width = 4 (5 - 1) and height = 6 (9 - 3).
    Rectangle rectangle = createRectangle(createPoint(1, 3), createPoint(5, 9));

    // Attempt to scale the rectangle with a horizontal scaling factor of 1.0 (no scaling in x-direction)
    // and a vertical scaling factor of 0.0 (scaling in y-direction by zero).
    // The ‘scale‘ function is expected to return ‘false‘ when the scaling factor is zero,
    // indicating that no scaling operation was performed.
    EXPECT_EQ(false, scale(&rectangle, 1.0, 0.0));

    // Define the expected state of the rectangle after the scaling operation.
    // Since the scaling factor is zero, the rectangle should remain unchanged.
    Rectangle expectedScaled = createRectangle(createPoint(1, 3), createPoint(5, 9));

    // Verify that the rectangle's dimensions and position remain unchanged after the scaling operation.
    // This assertion ensures that the ‘scale‘ function correctly handles the edge case of a zero scaling
    EXPECT_EQ(expectedScaled, rectangle);
}
```

### 5.5.4 Development Order in Test Driven Development

1. Write header files (.h) to define functions
2. Write all of the test cases we want to cover against those header files - we'll be able to compile those test cases but not execute them yet - as there are no implementations yet
3. Write the implementations (.cpp) and progressively execute the test cases against the implementations to measure progress towards completion

# Chapter 6

# OOP

## Operator Overloading Instead of toString

**Example 6.0.1** (Code Example)

```
class Point {
private:
        int x;
        int y;
public:
        Point(int x, int y);
        ~Point();
        int getX() const { return x; }
        int getY() const { return y; }

        friend std::ostream\& operator<<(std::ostream\& os,
        const Point\& point) {
                os << "Point(" << point.x << ",-" << point.y << ")";
                return os;
        }
};
```

**Note:-**

need `friend` here because we are defining this operator on `std::ostream` and it needs to access private state of `Point`