COMP2120 Computer Organisation
Revision Notes

Jacob Shing

24 Jan 2024

Remark. These notes were adapted from the lecture slides and the textbook. The author of these notes does not claim any originality. The author is not responsible for any errors or typos in these notes. Opinions expressed in these notes are not necessarily those of the author(s) of the original materials.
This PDF file was automatically compiled and published at https://github.com/ShingZhanho/COMP2120-Notes. The order/arrangement of the contents in this document reflects neither the order of teaching of the course nor the order of the textbook. The author of these notes has rearranged the contents for his own convenience.

Contents

List of Figures

Remark. For taking the final examination, an A4-sized double-sided cheatsheet is also provided on the same GitHub repository.

1 Evolution of Computers

1.1 Bench-marking Performance

Definition 1.1 (Clock Speed)**.** The pulse frequency (f) by the clock, measured in cycles per second, or Hertz (Hz). Also known as clock rate, clock speed. A cycle is technically a synchronised pulse.

Programs consist of instructions, and each instruction has several cycles. For example, a classic RISC[1] pipeline may have these 5 stages:

- Fetch (IF)
- Decode (ID)
- Execution/Effective Address (EX)
- Memory Access (MEM)
- Writeback (WB)

Definition 1.2 (Average Cycles Per Instruction (CPI))**.**

$$\text{Average CPI} = \frac{\sum_i \text{CPI}_i \times I_i}{I_c}$$

where I_i is the number of instructions of type i, and $I_c = \sum_i I_i$.

Definition 1.3 (Processor Time (T))**.**

$$T = \frac{I_c \times \text{CPI}}{f}$$

Definition 1.4 (Million Instructions Per Second (MIPS))**.**

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

Other useful metrics: Million Floating Point Operations Per Second (MFLOPS).

Remark. MIPS and MFLOPS may not accurately reflect the performance of a computer. A better approach is to measure the time required to do some real jobs. Standard Performance Evaluation Corporation (SPEC) benchmarks are used for this.

[1] RISC: Reduced Instruction Set Computing

2 Digital Logic

2.1 Boolean Algebra

First proposed by George Boole in 1854. Variables in boolean algebra are always either 0 or 1.

Definition 2.1 (Boolean Function)**.** A function in boolean algebra that takes k variables is defined as:

$$f : \{0,1\}^k \to \{0,1\}$$

Definition 2.2 (Boolean Operations)**.** In boolean algebra, the basic operations are denoted as follows:

- **AND**: $A \cdot B$ (or simply AB), gives 1 iff $A = B = 1$;
- **OR**: $A + B$, gives 1 if $A = 1$ or $B = 1$;
- **NOT**: $\overline{A}$, inverts the value of A;
- **NAND**: $\overline{A \cdot B}$, the complement of AND;
- **NOR**: $\overline{A + B}$, the complement of OR;
- **XOR**: $A \oplus B$, gives 1 iff one and only one of A and B is 1.

Remark. $A \oplus B = \overline{A}B + A\overline{B}$;
$\overline{A \oplus B} = \overline{\overline{A}B + A\overline{B}} = (\overline{\overline{A}B})(\overline{A\overline{B}}) = (A + \overline{B})(\overline{A} + B) = A\overline{A} + \overline{A}\overline{B} + AB + B\overline{B} = AB + \overline{A}\overline{B}$

The truth tables of AND, OR, XOR, and NOT are given by:

A	B	$A \cdot B$	$A + B$	$A \oplus B$	$\overline{A}$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Definition 2.3 (Boolean Algebra Laws)**.** The important laws of boolean algebra are:

- **Commutative Law**: $A + B = B + A$, $A \cdot B = B \cdot A$;
- **Identity Elements**: $A + 0 = A$, $A \cdot 1 = A$;
- **Null Law**: $A + 1 = 1$, $A \cdot 0 = 0$;
- **Idempotent Law**: $A + A = A$, $A \cdot A = A$;
- **Inverse Law**: $A + \overline{A} = 1$, $A \cdot \overline{A} = 0$;
- **Associative Law**: $(A + B) + C = A + (B + C)$, $(A \cdot B) \cdot C = A \cdot (B \cdot C)$;
- **Distributive Law**: $A \cdot (B + C) = A \cdot B + A \cdot C$, $A + (B \cdot C) = (A + B) \cdot (A + C)$;

Proof of Distributive Law $A + (B \cdot C) = (A + B) \cdot (A + C)$**.**

$$A + (B \cdot C) = A \cdot A + A \cdot C + B \cdot A + B \cdot C$$

Suppose $A = 1$, then the equation becomes:

$$1 + (B \cdot C) = 1 \cdot 1 + 1 \cdot C + B \cdot 1 + B \cdot C$$
$$1 = 1 + C + B + B \cdot C$$
$$1 = 1 + B \cdot C$$

which is true by the Null Law. Now, suppose $A = 0$, then the equation becomes:

$$0 + (B \cdot C) = 0 \cdot 0 + 0 \cdot C + B \cdot 0 + B \cdot C$$
$$B \cdot C = 0 + 0 + 0 + B \cdot C$$
$$B \cdot C = B \cdot C$$

which is true by the Idempotent Law. □

Theorem 2.4 (De Morgan's Theorem)**.** De Morgan's Theorem states that:

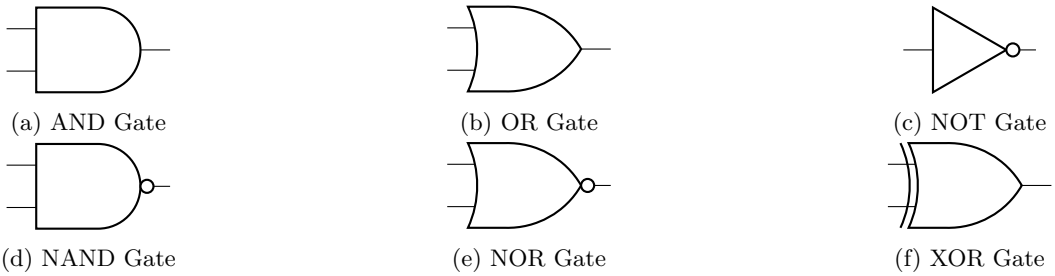
$$\overline{A + B} = \overline{A} \cdot \overline{B}$$
$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

More generally, we have:

$$\overline{A + B + \cdots + N} = \overline{A} \cdot \overline{B} \cdot \cdots \cdot \overline{N}$$
$$\overline{AB \ldots N} = \overline{A} + \overline{B} + \cdots + \overline{N}$$

2.1.1 Logic Gates

Other than expressions, boolean algebra can also be implemented using logic gates. Below is a list of the corresponding gates for the basic boolean operations:



(a) AND Gate (b) OR Gate (c) NOT Gate

(d) NAND Gate (e) NOR Gate (f) XOR Gate

Figure 1: Logic Gates Symbols

2.2 Functional Completeness

Definition 2.5 (Functional Completeness)**.** A set of logic gates is said to be functionally complete if any boolean function can be implemented using only gates from that set.

2.2.1 The `AND`, `OR`, `NOT` Set

The set of `AND`, `OR`, and `NOT` gates is functionally complete, i.e. they can be used to create any arbitrary boolean function.

2.2.2 The `AND`, `NOT` Set

By using De Morgan's Theorem, the OR operation can be implemented using only AND and NOT gates. Therefore, this reduced set is itself functionally complete. We start with the NAND expression in De Morgan's Theorem:

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

Then, we apply the NOT operation to both sides:

$$\overline{\overline{A + B}} = \overline{\overline{A} \cdot \overline{B}}$$
$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

The left hand side is exactly the OR operation, implemented using only AND and NOT gates.

2.2.3 The `OR`, `NOT` Set

Similarly, by using De Morgan's Theorem, the AND operation can be implemented using only OR and NOT gates. Therefore, this reduced set is itself functionally complete.

2.2.4 The `NAND` Set

The NAND operation alone can also be used to implement the AND, OR, and NOT operations. By applying NAND between A and A itself, we have:

$$\overline{A \cdot A} = \overline{A} \text{ (NOT operation)}$$

Similarly, by applying NAND over the value of $\overline{A \cdot B}$, we have $A \cdot B$, which is the AND operation.

Finally, by applying NAND between $\overline{A}$ and $\overline{B}$ (i.e. $\overline{\overline{A} \cdot \overline{B}}$), according to De Morgan's Theorem, we have $A + B$, which is the OR operation.

Since AND, OR, and NOT operations can be implemented using only NAND gates, and the set of AND, OR, and NOT gates is functionally complete, the NAND set is also functionally complete.

2.2.5 The `NOR` Set

Similar to the NAND set, the NOR set is also functionally complete.

2.3 Implementation of Boolean Functions

Suppose we would like to implement a boolean function $F(A, B, C)$ with the following truth table:

A	B	C	$F(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

The technique is to use either the Sum-of-Products (SOP) or the Product-of-Sums (POS) method.

2.3.1 Sum-of-Products (SOP)

SOP usually has the form of $F = XYZ + XYZ + \ldots$, every term should be the product of all the variables, appearing exactly once. Such terms are named "minterms". To implement the function F, first identify the rows in the truth table where $F = 1$, then for each row, construct the minterm by taking the product of all the variables, such that the sum of the minterms will give 1.

Example. From the truth table, we observe that $F = 1$ on the 3rd, 4th, and 7th rows.
For the 3rd row, the product $\overline{A}B\overline{C}$ equals 1.
For the 4th row, the product $\overline{A}BC$ equals 1.
For the 7th row, the product $AB\overline{C}$ equals 1.
Therefore, the function F can be implemented as:

$$F = \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C}$$

This method works because:

- any other input combinations will make all the minterms 0, and the sum of 0's is 0;
- when one of the minterms is 1, the sum will be 1.

2.3.2 Product-of-Sums (POS)

Contrast to SOP, POS usually has the form of $F = (X + Y + Z)(X + Y + Z)\ldots$. The approach is to identify the rows in the truth table where $F = 0$, then for each row, construct a **product** (NOT sum) of all the variables, such that the term give 1, then apply a NOT operation on the product. Connect all the terms together with AND operators, and simplify using De Morgan's Theorem.

Example. From the truth table, we observe that $F = 0$ on rows other than the 3rd, 4th, and 7th.
For the 1st row, we construct the product $\overline{\overline{A} \cdot \overline{B} \cdot \overline{C}}$, which equals 0.
For the 2nd row, we construct the product $\overline{\overline{A} \cdot \overline{B} \cdot C}$, which equals 0.
...
For the 8th row, we construct the product $\overline{A \cdot B \cdot C}$, which equals 0.
Then, connect the terms together with AND operations, we have:

$$F = \left(\overline{\overline{A}\,\overline{B}\,\overline{C}}\right) \cdot \left(\overline{\overline{A}\,\overline{B}\,C}\right) \cdot \left(\overline{A\,\overline{B}\,\overline{C}}\right) \cdot \left(\overline{A\,\overline{B}\,C}\right) \cdot \left(\overline{A\,B\,C}\right)$$

For each product, apply De Morgan's Theorem $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$ to simplify, we have:

$$F = (A + B + C) \cdot (A + B + \overline{C}) \cdot (\overline{A} + B + C) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C})$$

Remark. Whether to use SOP or POS depends on the truth table. Generally, when there are less 1's in the truth table, it is easier to use SOP. When there are less 0's, it is easier to use POS.

2.3.3 Simplification by Boolean Algebra

After constructing the SOP or POS expression, it is possible to simplify the expression using the laws of boolean algebra.

Example. The F we have implemented using SOP can be simplified as follows:

$$\begin{aligned} F &= \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C} \\ &= \overline{A}B\overline{C} + \overline{A}BC + \overline{A}B\overline{C} + AB\overline{C} \\ &= \overline{A}B\left(\overline{C} + C\right) + \left(\overline{A} + A\right)B\overline{C} \\ &= \overline{A}B + B\overline{C} \end{aligned}$$

2.3.4 Simplification by Karnaugh Maps

For boolean expressions with two to four variables, construct Karnaugh maps to simplify the expression. Rule for rows and columns: adjacent rows/columns must differ by only one variable. Rules for grouping 1's: each group-

- should be as large as possible;
- must be rectangular in shape;
- must contain number of 1's that is a power of 2 (1, 2, 4, 8, etc.);
- can overlap or wrap around the edges;
- should contain at least one 1 that is not in any other groups.

Example. To simplify $F = \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C}$, construct the map:

		BC			
		00	**01**	**11**	**10**
A	**0**			1	1
	1				1

They can be separated into two groups - one horizontal and one vertical. For the horizontal group, observe that regardless of the value of C, the value of F is always one, implying an expression of $\overline{A}B$. For the vertical group, the value of F is always one regardless of the value of A, implying an expression of $B\overline{C}$. Therefore, the simplified expression is $\overline{A}B + B\overline{C}$.

2.3.5 Application: Multiplexer

Example. Suppose we have a 2-to-1 multiplexer s defined as:

$$s = \begin{cases} a & \text{if } t = 0 \\ b & \text{if } t = 1 \end{cases}$$

The logical expression of the multiplexer is in fact $s = f(a, b, t)$. To solve for the expression, write down the truth table, and apply either SOP or POS to simplify the expression.

2.4 Adders

Adders can be created using logic gates. When they are chained together, they can be used to perform addition of binary numbers. There are two types of adders:

- **Half Adder**: adds two bits together, produce a sum and a carry (2-in-2-out);
- **Full Adder**: adds two bits and a carry produced from the previous addition, produce a sum and a carry (3-in-2-out).

2.4.1 Half Adder

Consider the truth table of adding two digits A and B:

A	B	Sum (S)	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

We can observe that $S = A \oplus B$ and $C = A \cdot B$. However, the half adder cannot be used to add numbers with more than one digit, because it does not take into account the carry from the previous addition.

2.4.2 Full Adder

Now take into account the carry from the previous addition (C'), we have the following truth table:

A	B	C'	Sum (S)	Carry (C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Apply SOP, we have:

$$\begin{aligned} S &= \overline{A}\overline{B}C' + \overline{A}B\overline{C'} + A\overline{B}\overline{C'} + ABC' \\ &= C'(\overline{A}\overline{B} + AB) + \overline{C'}(\overline{A}B + A\overline{B}) \\ &= C'(\overline{A \oplus B}) + \overline{C'}(A \oplus B) \\ &= C' \oplus (A \oplus B) \end{aligned}$$

and

$$\begin{aligned} C &= AB\overline{C'} + \overline{A}BC' + A\overline{B}C' + ABC' \\ &= AB(\overline{C'} + C') + C'(\overline{A}B + A\overline{B}) \\ &= AB + C'(A \oplus B) \end{aligned}$$

3 Number Representation

- Numbers are represented in binary form in computers.
- Mathematical laws do not necessarily hold in computer arithmetic. (e.g. $3.14 + 1e20 - 1e20 \neq 3.14 + (1e20 - 1e20)$ due to precision and rounding errors)
- Numbers in computers are Finite Precision Numbers.

Definition 3.1 (Finite Precision Numbers)**.** Finite Precision Numbers have the following characteristics:

- Limited number of bits to represent a number. (e.g. 32-bit integers)
- Limited range of numbers that can be represented. (e.g. 2^{32} for `unsigned int`)
- Overflow and underflow occur when the number is too large or too small to be represented.

3.1 Positional Number System and Radix

Definition 3.2 (Positional Number System)**.** A positional number system is a system in which the position of a digit in a number determines its value. Numbers are represented in a string of digits in the form of:

$$(a_n a_{n-1} \dots a_2 a_1 a_0 . a_{-1} a_{-2} \dots)_r$$

where:

- $n \in \mathbb{Z}$,
- $r \in [2, +\infty) \cap \mathbb{Z}$ is the radix (base) of the number system, which determines the value of each digit at position i as r^i,
- $a_i \in [0, r) \cap \mathbb{Z}$.

The value of the number being represented is given by:

$$\sum_i (a_i r^i) \quad \textbf{(The Direct Approach)} \tag{1}$$

Generally, the radix value is one of 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal).

Example. The number $1A6.BE_{16}$ is a fractional number in hexadecimal system. Its value in decimal system is:

$$\begin{aligned} 1A6.BE_{16} &= 1 \times 16^2 + 10 \times 16^1 + 6 \times 16^0 + 11 \times 16^{-1} + 14 \times 16^{-2} \\ &= 422.7421875_{10} \end{aligned}$$

Theorem 3.3 (Iterative Approach of Evaluating the Value of a Positional Number)**.** The value of a number in a positional number system can be evaluated alternatively by:

$$r\left(r\left(r\left(r \cdot a_n + a_{n-1}\right) + a_{n-2}\right) + a_{n-3} \dots\right) + a_0 \quad \textbf{(The Iterative Approach)} \tag{2}$$

which is more efficient than the direct approach.

Proof. Consider only the integer case. For the direct approach, for each digit a_i, its value is calculated by

$$a_i \underbrace{\times r \times r \times \cdots \times r}_{i \text{ times}}$$

Therefore, for $i = n$, the number of multiplications performed is n times. The total number of multiplications performed to convert a number of n digits is

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

For the iterative approach, the left most digit (a_n) is first multiplied by r and added to the next digit (a_{n-1}). This sum is then multiplied by r and added to the next digit (a_{n-2}), and so on. The total number of multiplications performed is n times. Therefore, the iterative approach has a linear complexity $O(n)$.
Hence, the iterative approach is more efficient than the direct approach. □

3.2 Integer Representation

3.2.1 Unsigned Integer Representation

Definition 3.4. For a sequence of n bits $(a_{n-1} a_{n-2} \dots a_1 a_0)$, it represents a nonnegative integer of value A, where

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

3.2.2 Sign-and-Magnitude Representation

Definition 3.5. For a sequence of n bits, the MSB[2]is used to represent the sign of the number. The rest $n-1$ bits represent the magnitude of the number. The value of the number is given by:

$$A = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} 2^i a_i$$

Example. For 4-bit integers, we have:

$$\mathbf{0}0110101_2 = +53_{10}$$
$$\mathbf{1}0110101_2 = -53_{10}$$

This representation has two pitfalls: 1. It has two representations for zero (0000_2) and (1000_2). 2. It is inconvenient for arithmetic operations. Therefore, this representation is rarely used to represent integers.

3.2.3 One's and Two's Complement Representation

Definition 3.6 (One's Complement)**.** For each positive integer, its one's complement representation is unchanged. For each negative number, its one's complement representation is obtained by flipping all the bits of its corresponding positive number.

Example. For 8-bit representation of $+8$, its one's complement is $0000\,1000_2$. For -8, its one's complement is obtained by flipping every bit of $+8$, which is $1111\,0111_2$.

One's Complement representation still has two representations for zero (0000_2) and (1111_2). Therefore, Two's Complement representation is more commonly used.

Definition 3.7 (Two's Complement)**.** For each positive integer, its two's complement representation is unchanged. For each negative number, its two's complement representation is obtained by adding 1 to its one's complement.

Example. For 8-bit representation of $+13$, its two's complement is $0000\,1101_2$. For -13, its two's complement is obtained by adding 1 to its one's complement, which is $1111\,0010_2 + 0000\,0001_2 = 1111\,0011_2$.

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

For the "negative zero", by the definition of two's complement, its 8-bit representation will be $-0_{10} = 1111\,1111_2 + 0000\,0001_2 = 1\,0000\,0000_2$. However, since the result is 9 bits, the carry bit is discarded, and the result is $0000\,0000_2$, which is the same as the representation of the "positive zero". The two's complement has only one representation for zero.

For arithmetic operations, simply add the two numbers together, and discard any carry from the MSB, the result will be the correct answer.

Remark. Like the Sign-and-Magnitude representation, the One's and Two's Complement representations use the MSB as the sign bit.

Theorem 3.8 (Overflow Rule for Two's Complement)**.** When two numbers of the same sign are added, overflow occurs iff the result has an opposite sign.

3.2.4 Range Extension

It is sometimes useful to store an integer that requires n bits in m bits ($m > n$). Different methods are needed for different representations.

- **Unsigned Integer**: Add more bits to the left and fill them with zeros.
- **Sign-and-Magnitude**: Add more bits to the left. Move the sign bit to the new MSB, and fill the rest with zeros.
- **Two's Complement**: Add more bits to the left. Fill the new bits with the sign bit. (**Sign Extension**)

[2]Most Significant Bit (the leftmost bit)

3.3 Integer Arithmetic Operations

3.3.1 Negation of Two's Complement

Definition 3.9 (Negation of Two's Complement)**.** To negate a number in Two's Complement representation, take the Two's Complement of the number.

3.3.2 Addition and Subtraction

Addition of two numbers in Two's Complement is the same as if they were unsigned integers. Refer to Theorem 3.8 for overflow rule. For subtraction, negate the second operand and perform addition.

3.3.3 Multiplication

To make explanations clear, we call the first operand the **multiplicand** and the second operand the **multiplier**.

For **unsigned integers**, perform:

1. For the i-th bit of the multiplier, if it is 1, shift the multiplicand left by i bits and add it to the partial sum. (a_i as in $a_n a_{n-1} \dots a_1 a_0$)
2. If the i-th bit of the multiplier is 0, do nothing.
3. Return the partial sum as the result.

For **two's complement integers with both operands being positive**, perform the same steps as for unsigned integers. For **two n-bit two's complement integers with one or both operands being negative**, perform:

1. For the i-th bit of the multiplier, where $i \in [0, n-1]$, if it is 1, shift the multiplicand left by i bits, then sign-extend it to $2n$ bits, and add it to the partial sum.
2. For the MSB of the multiplier,
   - If it is 1, take the two's complement of the multiplicand, sign-extend it to $2n$ bits, and add it to the partial sum.
   - If it is 0, do nothing.
3. Sum the partial sums, ignore the carry bit, and return the result.

Example (Multiplication of -11 and -13 in 8-bit two's complement)**.**

1111 0011	(multiplicand -13)
$\times$ 1111 0101	(multiplier -11)
1111 1111 1111 0011	(left shift by 0, sign-extend)
1111 1111 1100 11	(left shift by 2, sign-extend)
1111 1111 0011	$\cdots$
1111 1110 011	
1111 1100 11	
+ 0000 0110 1	(two's complement of multiplicand, sign-extend)
~~101~~ 0000 0000 1000 1111	(product +143)

Theorem 3.10. Multiplying an n-bit integer by an m-bit integer produces a product of at most $(n+m)$ bits.

3.3.4 Division

Not covered in this course.

3.4 Floating-Point Representation

3.4.1 Excess-K (Bias) Representation

Definition 3.11 (Excess-K Representation)**.** In an n-bit Excess-K representation, the values that are represented are in the interval of $[0-K, 2^n-1-K]$, where the smallest value is represented by all bits being 0, and the greatest value is represented by all bits being 1.
The K is referred to as an offset, or bias, as it is subtracted from the *bit pattern value* to obtain the represented *true value*.

Example. In a 3-bit Excess-5 representation, we have:

Bit Pattern	Bit Pattern Value	True Value
111	7	2 (= 7 − 5)
110	6	1
101	5	0
100	4	-1
011	3	-2
010	2	-3
001	1	-4
000	0	-5

The K is typically chosen to be 2^{n-1} or $2^{n-1}-1$, with the latter being more common.
The Excess-$2^{n-1}-1$ representation is used for representing a floating-point number as regulated by the IEEE[3]standard.

3.4.2 IEEE 754-2008 Floating-Point Number Representation

Definition 3.12 (Format of a Floating-Point Number)**.** A floating-point number is represented in the form of:

$$\pm \text{Significand} \times 2^{\pm \text{(Biased) Exponent}}$$

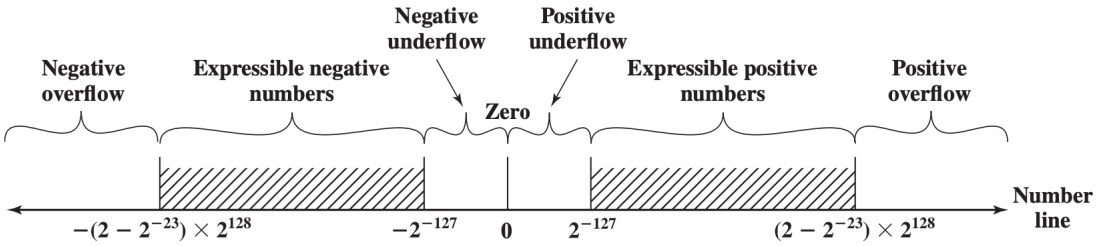
In a general 32-bit IEEE floating-point number, it is stored in memory by:

The floating-point number is always normalised before being stored. A normal number is the one with the MSB being 1. The convention is to always make the radix point to the right of the MSB, i.e. the MSB is always 1. Therefore, the MSB is never stored in memory. A normal nonzero number takes the form:

$$\pm 1.\underbrace{\text{bbb}\ldots\text{b}}_{23 \text{ bits}} \times 2^{\pm \text{E}}$$

The exponent is stored in Excess-127 ($2^{8-1}-1$). Meaning that 127 is added to the true exponent before being stored.

Note that since there is always a leading 1, the significand is always in the range of $[1, 2)$. This implies that when the floating-point number is too far away from or too close to zero, it cannot be represented accurately. The following diagram shows the range of representable values of a 32-bit floating-point number (**not the IEEE standard**):



The diagram shows 0 is in the range of underflow, which can be inconvenient. Therefore, special values are defined in the IEEE standard to represent 0, $\pm\infty$, and `NaN` (Not a Number).

- ± 0: Biased exponent and significand are all 0. The sign bit determines the sign.
- $\pm\infty$: Biased exponent is all 1, and significand is all 0.
- `NaN`: Biased exponent is all 1, and significand is not all 0.
- Subnormal numbers: Biased exponent is all 0, and significand is not all 0. ($\pm 2^{-126}(0.\text{S})$) Allows gradual underflow.

Remark. In assignments and exams, unless specified, the above special meanings are not considered.

IEEE 754-2008 defines three types of binary floating-point numbers - single precision (`Binary32`), double precision (`Binary64`), and quadruple precision (`Binary128`). The following table shows the parameters of each type:

[3]Institute of Electrical and Electronics Engineers

Parameter	Format		
	Binary32	**Binary64**	**Binary128**
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	−126	−1022	−16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Note: * Not including implied bit and not including sign bit.

Example (Manual conversion of a decimal number to Binary32 floating-point number)**.** To convert -23.875 to a 32-bit floating-point number, we have these steps:

- Convert -23.875 to binary: $-23.875_{10} = -10111.111_2$
- Normalise the binary number: $-10111.111_2 = -1.0111111_2 \times 2^4$
- Convert the exponent to biased exponent: $4 + 127 = 131 = 10000011_2$
- Set the sign bit, biased exponent, and significand: $\underbrace{1}_{\text{sign}}\ \underbrace{1000\,0011}_{\text{biased exponent}}\ \underbrace{0111\,1110\,0000\,0000\,0000\,000}_{\text{significand}}$

Theorem 3.13. Some important properties of floating-point numbers:

- The number of representable values of a n-bit floating-point number is NOT greater than that of a n-bit integer (both being 2^n).
- The representable values of a floating-point number are NOT uniformly distributed.

3.5 Floating-Point Arithmetic Operations

3.5.1 Addition and Subtraction

Steps for floating-point number addition and subtraction:

1. For the number with the smaller exponent, shift the significand right by the difference in the exponents to align their radix points.
2. Add or subtract the significands, then determine the sign of the result.
3. Normalise the result. Truncate the significand to the largest number of bits allowed.

Generally, there are five phrases in floating-point addition and subtraction:

1. **Check for zeros**;
2. **Alignment** of significands;
3. **Addition/Subtraction** of significands;
4. **Normalisation** of the result;
5. **Rounding**.

A typical floating-point addition/subtraction is illustrated in the following flowchart:

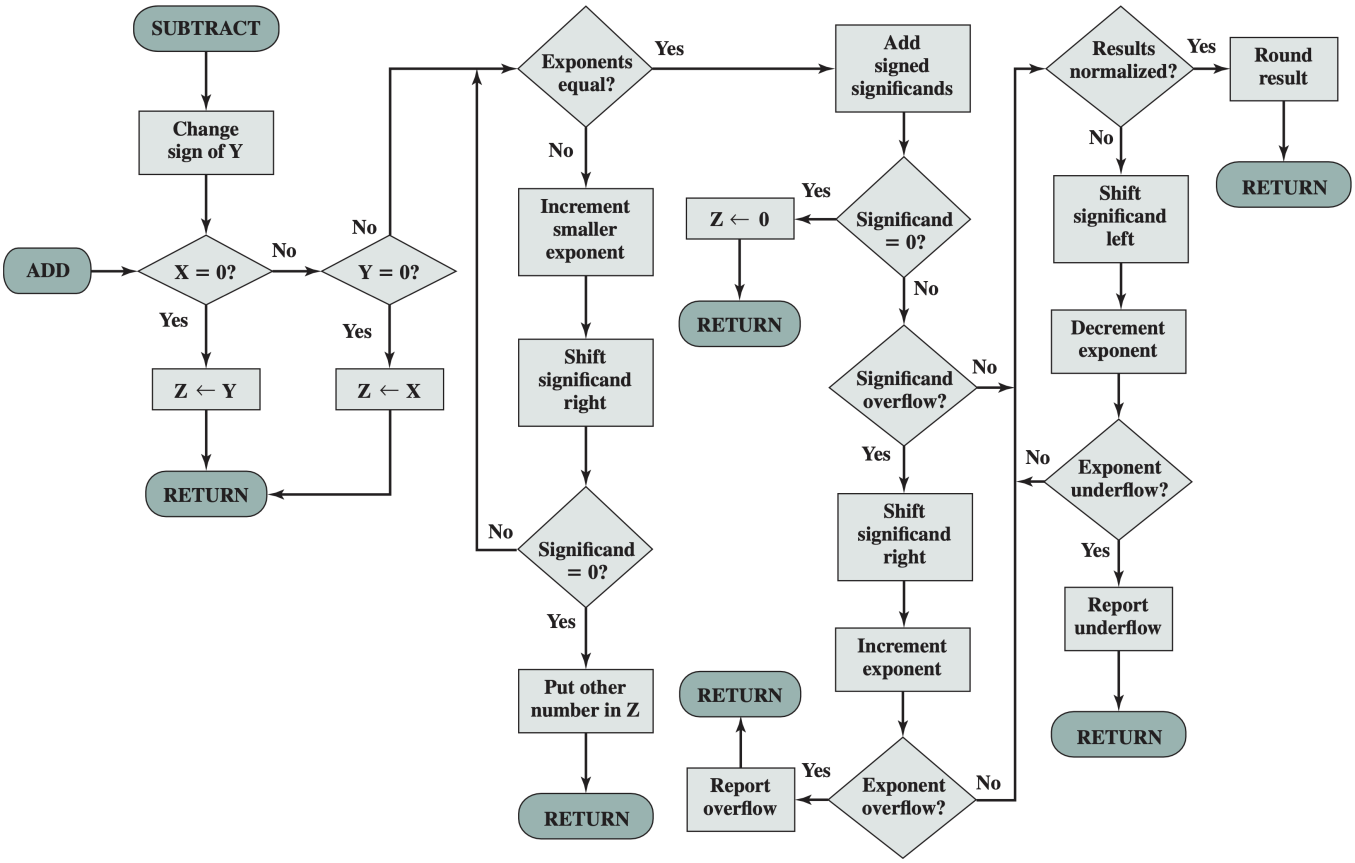


Figure 2: Floating-Point Addition and Subtraction Flowchart

3.5.2 Multiplication

Consider the multiplication illustrated in the equation:

$$\pm m_1 \times 2^{\text{exp}_1} \times \pm m_2 \times 2^{\text{exp}_2} = \pm m_1 \times m_2 \times 2^{\text{exp}_1 + \text{exp}_2}$$

where exp is the real exponent value stored in Excess-K, whose bit pattern is $\text{e} := \text{exp} + K$. To perform multiplication, the exponents are added, and the significands are multiplied. The result is then normalised.

Consider the addition of the exponents' bit pattern, we have:

$$\begin{aligned} \text{exp}_1 + \text{exp}_2 &= (\text{e}_1 - K) + (\text{e}_2 - K) \\ \text{exp}_{\text{sum}} &= \text{e}_1 + \text{e}_2 - 2K \\ \text{e}_{\text{sum}} - K &= \text{e}_1 + \text{e}_2 - 2K \\ \text{e}_{\text{sum}} &= \text{e}_1 + \text{e}_2 - K \end{aligned}$$

Therefore, the bias K is subtracted from the sum of the exponents' bit patterns. Then, determine the sign of the result, and normalise and round the result.

3.5.3 Division

Division of floating-point numbers is similar to multiplication. It is defined by:

$$\frac{\pm m_1 \times 2^{\text{exp}_1}}{\pm m_2 \times 2^{\text{exp}_2}} = \pm \frac{m_1}{m_2} \times 2^{\text{exp}_1 - \text{exp}_2}$$

For the division of the exponents' bit patterns, we have:

$$\begin{aligned} \text{exp}_1 - \text{exp}_2 &= (\text{e}_1 - K) - (\text{e}_2 - K) \\ \text{exp}_{\text{diff}} &= \text{e}_1 - \text{e}_2 \\ \text{e}_{\text{diff}} - K &= \text{e}_1 - \text{e}_2 \\ \text{e}_{\text{diff}} &= \text{e}_1 - \text{e}_2 + K \end{aligned}$$

Therefore, the bit patterns of the exponents are subtracted, and the bias K is added to the result. Then, determine the sign of the result, and normalise and round the result.

Flowcharts showing floating-point multiplication and division are shown below.

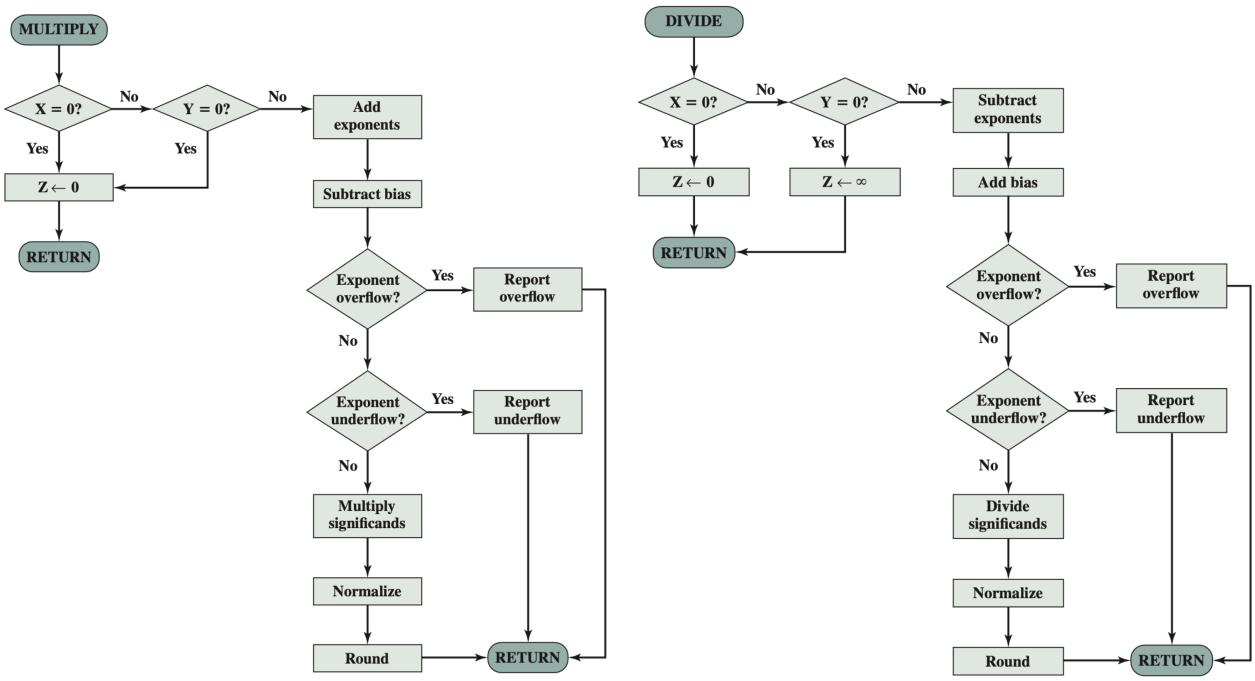


Figure 3: Floating-Point Multiplication and Division Flowcharts

3.5.4 Approximation of Floating-Point Arithmetic

Floating-point numbers are prone to precision errors due to the following reasons:

- Not all numbers can be represented precisely in binary, e.g. $0.2_{10} = 0.00110011\ldots_2$.
- Round-off errors: some digits of the significand is lost to the left or right end of the significand when being shifted.

Due to errors, the Associative Law do not necessarily hold, especially when a very large number is calculated with a very small number. In such cases, different orders of operations may yield different results.

Different approaches for rounding a floating-point number include 1. round to nearest, 2. round towards zero, 3. round towards $\pm\infty$.

4 Instruction Execution Cycle

4.1 Terminologies and Basic Information

Terminologies:

- **Byte**: 8 bits, smallest addressable unit of memory.
- **Word**: Unit of organisation of memory, varies from system to system. e.g. 32-bit, 64-bit, etc.
- **Register**: Small, fast storage location within the CPU.
- **Word Addressing**: Addresses of memory on a computer that uniquely identify a word.
- **Byte Addressing**: Addresses of memory on a computer that uniquely identify a byte.

Registers inside a CPU:

- **PC (Program Counter)**: Holds the address of the next instruction to be fetched.
- **IR (Instruction Register)**: Holds the current instruction.
- **MAR (Memory Address Register)**: Holds the address of the memory location to be accessed.
- **MBR (Memory Buffer Register)**: Holds the data to be written to or read from memory. (Also, MDR)
- **I/O AR** and **I/O BR**: Similar to MAR and MBR, but for I/O operations.

Data are transferred through system bus. The source register put the data on the bus, then the destination register pulls the data from it. At any given time, only one data transfer can be performed on one bus.

4.2 Instruction Execution Cycle

Remark. This note assumes the computer uses 32-bit word.

A typical instruction execution cycle consists of these stages:

1. Instruction Fetch (IF);
2. Instruction Decode (ID);
3. Instruction Execution, which includes:
   (a) Calculate Operand Address (CO);
   (b) Operand Fetch (OF);
   (c) Execution (EI);
   (d) Write Operand (WO);

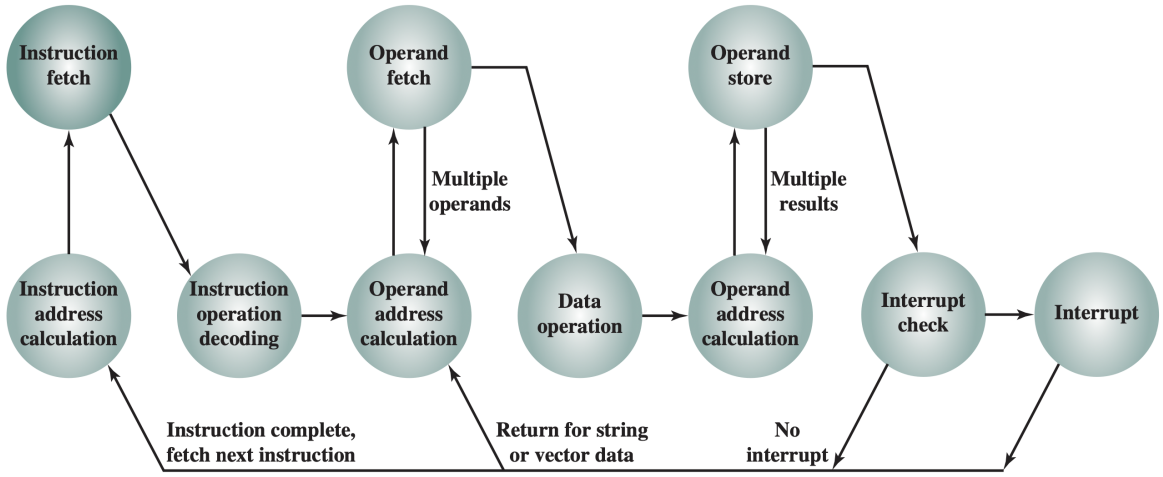


Figure 4: Instruction Execution Cycle with Interrupts Handling

4.2.1 Instruction Format

An instruction can be one-word or multi-word. A typical form of instruction may have the format of:

Byte 0	Byte 1	Byte 2	Byte 3
Opcode	Source Operand 1	Source Operand 2	Destination Operand

where opcode stands for operation code, for example, `ADD (0x00)`, `SUB (0x01)`, `AND (0x02)`, `OR (0x03)`, `NOT (0x04)` etc.

Types of Operations: **(1)** Data transfer (e.g. `MOV`, `LD`, `ST`); **(2)** Arithemtic (e.g. `ADD`, `SUB`); **(3)** Logical (e.g. `AND`, `OR`, `NOT`); **(4)** Transfer of control (e.g. `BR`, `CALL`); **(5)** Input/Output (see Port-mapped I/O in Section 6); **(6)** Data conversion (e.g. `NEG` - negate, `SXT` - sign extend).

Logical Operations vs Arithmetic Operations:

1. **Arithmetic and Logical Operations**:
   Consider this operation: `ADD R1, R2, R3` (or in pseudocode, `R3 = R1 + R2`), the instruction is represented as `0x00010203`. Since `ADD` requires two source operands and one destination operand, all fields are used.
   Consider `NOT R1, R2` (or in pseudocode, `R2 = NOT R1`), the instruction is represented as `0x04010002`. Since `NOT` uses only one source operand, the second source operand is set as `0x00`.
2. **`LD` and `ST` Instructions**:
   For instructions that uses memory, a two-word instruction is needed as the memory address cannot fit in the operand. Assume that `LD (0x06)` and `ST (0x07)`. For example, `LD P1(0x0000003c), R1`, which loads the content of memory address `P1` to register `R1`, the instruction is represented as `0x0600ff01 0000003c`. The format is:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Opcode (`LOAD`)	Source (`0x00`)	Addressing Mode (`0xff`)	Destination	Memory Address			
Word 0				Word 1			

   Remark. For simplicity, there is only one addressing mode (`0xff`) used in this section.

3. **Branching Instructions**:
   A branching instruction has the following format:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Opcode (Branch)	Condition Code	Address/Addressing Mode	(not used)	Address of Destination Instruction			
Word 0				Word 1			

   Examples of condition codes are:

Instruction	Condition Code	Meaning
BR	0x00	Unconditional branch
BZ	0x01	Branch if zero
BNZ	0x02	Branch if not zero

   By saying "zero", it means to check the zero flag of the ALU to determine if the previous operation resulted in zero. Other flags may also be used.

   Example. Consider this code (on the right side is the machine code in hexadecimal):

```
        LD  P2, R2      ;                       0000: 0600ff02 00000034
        LD  P1, R1      ;                       0008: 0600ff01 00000030
        LD  P3, R3      ;                       0010: 0600ff03 00000038
        MOV R2, R4      ;                       0018: 05020004
    L:  ADD R2, R3, R2  ; Increment R2 by 1     001C: 00020302
        SUB R1, R2, R4  ; R4 = R1 - R2          0020: 01010204
        BNZ L           ; If R4 != 0, go to L   0024: 0802ff00 0000001C
        HLT             ;                       002C: 09000000
   P1:  .WORD 5         ;                       0030: 00000005
   P2:  .WORD 0         ;                       0034: 00000000
   P3:  .WORD 1         ;                       0038: 00000001
   P:   .WORD           ;                       003C: 00000000
```

   The `BNZ` instruction is used to create a loop that increments `R2` by 1 until `R1 - R2` is zero. The program halts when the condition is met.

4. **Halt Instruction**:
   The `HLT` instruction is used to halt the program. It does not use any operands and those fields are set to `0x00`.

Remark. Read Section 7.1: More on Instruction Format for an extension on this topic.

4.2.2 Instruction Fetch

Address to the next instruction is stored in PC register, which is incremented automatically during execution. For a two-word instruction, the first word is fetched first, then PC is incremented by 1 word to point to the second word. Then

the second word is fetched, and PC is incremented again to point to the next instruction.

Note that PC is will be changed when branching happens.

This process is called Instruction Address Calculation (IAC).

During IF, the following data transfer happens:

$$\begin{aligned} \text{MAR} &\leftarrow \text{PC} \\ \text{PC} &\leftarrow \text{PC} + 1 \\ \text{MBR} &\leftarrow \text{Memory[MAR]} \end{aligned}$$

4.2.3 Instruction Decode

The control unit will decode the instruction and setup the ALU and other components for appropriate operations (e.g. memory read/write, data transfer, etc.). These actions are carried out at appropriate times by the control unit.

4.2.4 Operand Fetch

If the operands are in registers, data are moved from registers to ALU.

If the operands are in memory, then the instruction would be a two-word instruction. Note that if PC points to the second word of a two-word instruction, after the above process, MBR will contain an **address** at which data of the operand is stored, not the content. Therefore, another memory read is needed to fetch the content, by:

$$\begin{aligned} \text{MAR} &\leftarrow \text{MBR} \\ \text{MBR} &\leftarrow \text{Memory[MAR]} \end{aligned}$$

4.2.5 Execution

The ALU performs the operation specified by the instruction. The result is stored in some temporary register.

4.2.6 Result Store

Similar to operand fetch, if the destination is in register, RF write is performed. If the destination is in memory, then operand address calculation is first performed. Then the data is written to memory.

4.2.7 Interrupt Handling

Interruptions are important as:

- They improve efficiency.
- When an I/O arrives, it may need immediate attention, or data may be lost. e.g. incoming data from a network.
- Other programs may also need the CPU's attention. e.g. on a time-sharing system.

When interruption is required, I/O device sends a signal to the CPU. The CPU will need to remember the current state of the program, and then jump to serve the interrupt. The CPU will then return to the original program and continue execution as if nothing happened.

Interrupt handlers can either be hardware or software.

Interrupt signals are checked at the end of one complete instruction cycle, minimising the registers that need to be saved/restored. Before each interrupt, the following information is saved (by pushing to the stack):

- Flag register – to remember the state of the ALU.
- PC – to remember the address of the next instruction.
- Modified register files.
- The current address of the instruction – to remember where the program was interrupted, while the address of the interruption program is loaded to PC.

Other registers (like MAR, MBR, IOAR, etc.) are not saved as they are only meaningful during the current instruction cycle.

5 Memory

5.1 Memory Hierarchy

Different types of memory exhibit different performance and impose different costs of production. To achieve a balance between performance and cost, a hierarchy of memory is introduced.

1. **Inbound Memory**: The fastest memory in the hierarchy.
   - **Registers**: inside the processor.
   - **On-chip Cache**: on the CPU chip.
   - **Cache Memory**: on the motherboard.
   - **Main Memory**: RAM, on the motherboard.
2. **Outbound Storage**: Secondary memory. e.g. hard disk, SSD, DVD, etc.
3. **Off-line Storage**: magnetic tapes, etc.

Going from top to bottom, we observe the following trends:

- **Capacity**: increases.
- **Cost per bit**: decreases.
- **Access time**: increases.
- **Frequency of access**: decreases.

5.1.1 Principle of Locality

Definition 5.1 (Principle of Locality)**.** The Principle of Locality states that programs do not access all memory locations uniformly. Some memory locations have higher tendency to be accessed than others.

There are two types of locality:

Definition 5.2 (Temporal Locality)**.** If a memory location is accessed, it is likely to be accessed again soon.

Definition 5.3 (Spatial Locality)**.** If a memory location is accessed, it is likely that nearby memory locations will be accessed soon.

Example. Consider the following code:

```
for (int i = 0; i < 100; i++) {
    sum += arr[i];
        // sum is accessed in every iteration -> temporal locality
        // consecutive memory locations of arr[] are accessed -> spatial locality
}
```

5.1.2 Memory Organisation

For multi-byte data, they are stored differently in memory on different architectures.

- **Big Endian Mode**: Stored in memory from left to right. (e.g. IBM mainframes)
- **Little Endian Mode**: Stored in memory from right to left. (e.g. Intel x86)

Example. For storing the 4-byte integer `0x12345678` in memory:

Address	101	102	103	104
Big Endian	12	34	56	78
Little Endian	78	56	34	12

5.1.3 Unit of Transfer

CPU reads data not from main memory but from cache memory, as main memory is much slower. Between cache and main memory, data are transferred in blocks, which contain multiple bytes (e.g. 4 KBytes).

5.1.4 Access Methods

1. **Sequential Access**: Data is accessed from the beginning to the end.
2. **Random Access**: Data can be accessed in any order directly, by providing the address. Has a constant latency.

3. **Associative Access**: Data can be accessed by providing the content of the data, not the address. Used in cache memory and Content-Addressable Memory (CAM).

5.2 Internal Memory

This section presents some common CMOS[4] memory technologies and their characteristics.

5.2.1 Read-Only Memory (ROM)

- Fabricated like integrated circuit chips.
- Non-volatile memory, i.e., retains data even when power is turned off.
- **Programmable ROM (PROM)**: Can be programmed once. Usually done by supplier or customer after chip is manufactured.
- Content cannot be changed.

5.2.2 Read-Mostly Memory

- **Erasable Programmable ROM (EPROM)**: Can be erased and reprogrammed multiple times. Erasing is done by exposing the chip to ultraviolet light for a specified time.
- **Electrically Erasable PROM (EEPROM)**: Can be erased and reprogrammed in place. Only the bytes addressed are changed. Erasing process is slow.

5.2.3 Flash Memory

- Non-volatile memory.
- Between EPROM and EEPROM.
- Faster write than EEPROM.
- Have limited number of write cycles.
- Usage: USB drives, SSD, storage of BIOS (in recent years).

5.2.4 Random Access Memory (RAM)

Characteristic	**Dynamic RAM**	**Static RAM**
Storage Technology	Use transistors to store electric charges.	Use logic gates (latches).
Refreshing	Required (every few ms due to leaking charge)	Not required
Speed	Slower (delay due to capacitance)	Faster
Usage	Main memory	Cache memory
Cost	Cheap	Very expensive

5.2.5 Comparison of Memory Types

Memory Type	**Category**	**Erasure**	**Write Mechanism**	**Volatility**
RAM	Read-write	Electrically, byte-level	Electrically	Volatile
ROM	Read-only	Not possible	Masks	Nonvolatile
PROM			Electrically	
EPROM	Read-mostly	UV light, chip-level		
EEPROM		Electrically, byte-level		
Flash		Electrically, block-level		

5.2.6 Bench-marking Memory Performance

Key metrics for memory performance include:

- **Access Time**: Time taken to read/write data.
- **Bandwidth/Transfer Rate**: Rate at which data can be read/written.

[4] Complementary Metal-Oxide-Semiconductor

- **Memory Cycle Time**: (Access Time + Transfer Time).

Example. A two-level memory system has an upper level with 0.01μs access time and a lower level with 0.1μs. The upper level has a hit rate of 95%. The average time to access memory is:

$$0.01 \times 0.95 + (0.01 + 0.1) \times 0.05 = 0.015\mu\text{s}$$

5.2.7 Error Detection and Correction

Errors may arise from various sources such as spike in voltage (lightning), electromagnetic interference (cosmic ray), power supply problems, etc. Extra bits are required to detect errors. (Usually in secondary storage devices but not in main memory as it is volatile and less likely to have errors.)

Common error detection and correction methods include:

- **Parity Bit**: Extra bit added to data to make number of 1s even/odd. (Cannot be used to correct errors when the location of the error is unknown. Cannot detect $2n$-bit errors.)

Definition 5.4 (Even Parity Bit)**.** Suppose there are a series of bits $(b_1 b_2 b_3 \cdots b_{n-1})$, then the even parity bit b_n is given by:

$$b_n = b_1 \oplus b_2 \oplus b_3 \oplus \cdots \oplus b_{n-1}$$

Theorem 5.5 (Error Correction by Parity Bits)**.** Parity Bits can be used to recover errors iff the number of errors is one and the location of the error is known.

Example. [5.5]
Suppose we have a bit pattern $b_1 b_2 b_3 p$, where p is the parity bit. By Definition 5.4, we have

$$p = b_1 \oplus b_2 \oplus b_3$$

Given that b_2 is lost, we can recover b_2 by applying $\oplus b_2 \oplus p$ on both sides of the equation, then we have:

$$p \oplus b_2 \oplus p = b_1 \oplus b_2 \oplus b_3 \oplus b_2 \oplus p$$

$$b_2 = \boxed{b_1 \oplus b_3 \oplus p}$$

- **Hamming Code**
- **Repetition Code**

5.3 Cache Memory

Cache memory is transparent (hidden) to the software and is managed by the hardware. It stores copies of frequently accessed data to speed up subsequent access to that data.

There can be one or more layers between the CPU and main memory. The transfer between the CPU and L1 cache is the fastest, and the speed decreases as the distance from the CPU increases.

5.3.1 Cache Memory Organisation

A word-addressable main memory with n-bit addresses has 2^n words. Divide the main memory into blocks of K words each, then the memory has $M = \frac{2^n}{K}$ blocks. Suppose the cache has m blocks, called **lines**. Generally, we have $m \ll M$. Each line has K words, a tag, and several control bits. The length of the line, excluding the tag and the control bits is called the **line size** or **block length**.

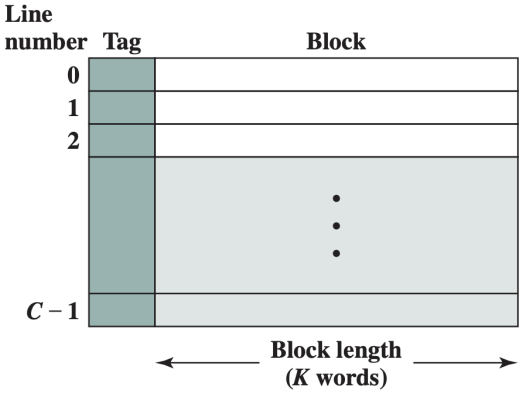


Figure 5: Cache Memory Organisation

5.3.2 Cache Memory Read

The process of reading from the cache is roughly described as follows:

1. Obtain the address of the word to be read from the CPU.
2. Check if the block containing the word is in the cache.
   (a) If the block is in the cache, read the word from the cache.
   (b) If the block is not in the cache, load the block from the main memory into the cache, and **at the same time** deliver the word to the CPU.

5.3.3 Address Mapping

There are three ways to map the main memory to the cache:

1. **Direct Mapping**: Each block of main memory maps to exactly one line in the cache. The mapping is given as
$$i = j \bmod m$$
where i is the cache line number, j is the main memory block number, and m is the number of cache lines.

> **Example.** The cache logic treats the main memory address in three parts as follows:
> $$\underbrace{\overbrace{0000\,0001}^{(s-r)\text{ bits (tag)}}\ \overbrace{1111\,1111\,1111\,11}^{r\text{ bits (line number)}}}_{\text{block number}}\ \overbrace{00}^{w\text{ bits (word)}}$$
> The least significant w bits identify the word within the block, where the block size is 2^w words. The next r bits identify the line number within the cache memory, where the cache memory has 2^r lines. The most significant $(s-r)$ bits are the tag bits, which are used to distinguish between the different main memory blocks that map to the same line, where the main memory has 2^s blocks.
>
> > **Remark.** If the main memory is byte-addressable, then the word bits should be referred to as byte offsets, and they correspond to the number of 2^w bytes in a block.

To perform a read operation, the line number first identifies the line in the cache. Then, the tag bits in the line are compared with the tag bits in the address. If the tags match, the word is read from the cache according to the word bits in the address. If the tags do not match, the block is read from the main memory into the cache, and the word is read from the cache.

Advantages:
- Only need to check one cache line, fast.
- No selection is required, less use of logic gates, inexpensive.

Disadvantages:
- When two blocks map to the same line are accessed alternatively, constant cache misses occur.
- The cache is not fully utilised.

2. **Fully Associative Mapping**: Any block of main memory can be loaded into any line of the cache. The main memory address is treated as two parts only – the tag and the word bits. Again, for a main memory address of $(s+w)$ bits, and a cache memory with block length of 2^w words, the word bits are the least significant w bits, and the tag bits are the remaining s bits.
To perform a read operation, the cache controller searches the entire cache for the desired tag. If it is a miss, the block is read from the main memory into the cache.

Advantages:
- More flexible use of cache than direct mapping.
- Higher hit rate.

Disadvantages:
- Requires more complex logic and circuits for tag comparison, more expensive.
- Must simultaneously search all cache lines, slower.

3. **Set Associative Mapping**: The cache consists of a number of sets, and each sets consists of a number of lines. Their relationship is given by
$$m = v \times k$$
$$i = j \bmod v$$
where i is the set number, j is the main memory block number, m is the number of lines in cache, v is the number of sets, and k is the number of lines in each set. (k is usually 2, the maximum is 8.) Also referred to as k-way set associative mapping.
Block B_j in main memory maps to any of the lines in set j in the cache. There are two ways of implementing a set associative mapping, either as
   (a) v associative-mapped caches (usually used for high associativity, i.e. larger k). Each cache is called a **set**.
   (b) k direct-mapped caches (for lower associativity). Each cache is called a **way**.

The cache control logic treats the $(s+w)$-bit main memory address in three parts: tag, set, and word. The d bits of set identifies the set in cache, where $v = 2^d$. The least significant w bits identify the word. The remaining $(s-d)$ bits are the tag bits.
With this method, the tag size is much smaller than in fully associative mapping, and each tag is only compared with k tags in a single set.

Advantages:

- Fewer misses than direct mapping.

Disadvantages:

- Complex selection and comparison logic, slightly slower.

Example. Suppose a computer has a byte-addressable main memory with addresses of 32 bits, a 64 KB 2-way set associative cache memory, and the block size is 128 bytes. Find the number of bits in the tag, set, and word fields of the main memory address.

Solution. Since the block size is 128 (2^7) bytes, then the word field is 7 bits.
Number of blocks in cache memory is $64 \times 2^{10} \div 2^7 = 512$ blocks.
Number of sets in cache memory is $512 \div 2 = 256 = 2^8$ sets. Therefore, the set field is 8 bits.
The remaining bits are the tag field, which is $32 - 8 - 7 = 17$ bits.

Tag	Set	Word
17	8	7

5.3.4 Replacement Algorithms

When all lines are occupied and a new block needs to be loaded into the cache, a line must be selected to be replaced. The different replacement algorithms are:

1. **Random Replacement** (RR): A random line is selected to be replaced. This is the simplest method, but it does not guarantee the best performance. Not used in practice.
2. **First-In-First-Out** (FIFO): The line that has been in the cache the longest is replaced. It does not consider the frequency of use of the block.
3. **Least Recently Used** (LRU): The line that has not been used for the longest time is replaced.
4. **Least Frequently Used** (LFU): The line that has the fewest references is replaced, often implemented with a counter.

Note that replacement algorithms do not apply to direct-mapped caches, as there is only one possible line to replace.

5.3.5 Write Policies

To maintain consistency between the cache and main memory, the main memory must be updated whenever the cache line to be replaced has been modified. There are two write policies:

1. **Write-Through**: The main memory is updated whenever the cache is updated. This ensures that the main memory is always up-to-date, but it is slower as the CPU must wait for the main memory to be updated.
2. **Write-Back**: The line is associated with a **dirty bit** that is set whenever the line is modified. The main memory is only updated when the line is replaced and the dirty bit is set. This method minimises the number of main memory writes and is faster. The main drawback is that the main memory may be inconsistent.

5.3.6 Performance

Definition 5.6 (Average Access Time). The average access time of a cache memory is given by

$$\begin{aligned}\text{Average Access Time} &= \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Average Time When Miss} \\ &= \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}\end{aligned}$$

Example. Assume the access time of main memory is 50 ns, the L1 cache has a miss rate and access time of 10% and 1 ns, respectively, the L2 cache has a miss rate and access time of 5% and 5 ns, respectively, and the L3 cache has a miss rate and access time of 2% and 10ns, respectively. Calculate the average access time of the memory system.

Solution. When there is no cache, access time = 50 ns.
When there is only L1 cache, access time $= 1 + 10\% \times 50 = 6$ ns.
When there are L1 and L2 caches, access time $= 1 + 10\% \times (5 + 5\% \times 50) = 1.75$ ns.
When there are L1 to L3 caches, access time $= 1 + 10\% \times (5 + 5\% \times (10 + 2\% \times 50)) = \boxed{1.555 \text{ ns}}$.

Example. Consider a hypothetical machine with 512 words of cache memory. They are in two-way set associative organisation, with cache block size of 64 words, using LRU replacement. Suppose the cache hit time is 8 ns, and the time to transfer the first word from main memory to cache is 60 ns, while subsequent words require 10 ns each.

1. What is the cache miss penalty?
2. If there is a read sequence of 28 blocks accessed that has 15 cache misses, what is the cache hit rate?
3. What is the average memory access time?

Solution.
1. Cache miss penalty is the time to transfer one block from main memory to cache. $60 + 63 \times 10 = \boxed{690 \text{ ns}}$
2. Cache hit rate $= 1 - \frac{15}{28 \times 32} = \boxed{98.33\%}$.
3. Average memory access time $= 8 + (1 - 0.9833) \times 690 = \boxed{19.52 \text{ ns}}$.

5.3.7 Unified Cache and Split Cache

1. **Split Cache**: The cache is divided into instruction cache and data cache. Size for each cache is fixed. No pipeline hazard. The main trend is to use split cache.
2. **Unified Cache**: The cache is used for both instructions and data. Instructions and data are automatically balanced. Has contention problem on parallel and pipeline execution that imposes bottleneck on performance.

5.4 Virtual Memory

In multitasking operating systems, the demand of memory is often greater than the available physical memory. To solve this problem, virtual memory is introduced.

Physical Address vs **Logical Address**:

- **Physical Address**: The address used to actually access the physical memory, which can be smaller (e.g. 1 GiB).
- **Logical Address**: The addressing space that a program/process sees, which can be larger. (e.g. 4 GiB if 32-bit)

Mapping between logical and physical addresses is done by the **Memory Management Unit (MMU)**. Similar to Cache memory, the MMU is based on the principle of locality. It maps a logical address (of a program) to a physical address (of the memory).

5.4.1 Paging & Page Table

Physical memory is divided into fixed-size blocks called **frames**, and logical memory is divided into blocks of the same size called **pages**.

Each process has its own logical address space, hence each process will be divided into several pages. The OS maintains a **page table** for each process.

Each page in logical address space has a corresponding page table entry (PTE). The format of a PTE is roughly:

1 bit	several bits	1 bit	serveral bits
Valid bit	**P**rotection Bits	**D**irty Bit	Physical Frame #

- **Valid Bit**: Indicates whether the page is in memory or not.
- **Protection Bits**: Indicate the access rights of the page (read, write, execute, user/kernel access).
- **Dirty Bit**: Indicates whether the page has been modified or not.
- **Physical Frame #**: The frame number in physical memory where the page is stored.

5.4.2 Demand Paging & Page Fault

When a program is loaded, not all its pages are loaded at once. The OS only loads the pages needed, i.e. on demand, hence the name **demand paging**. **Pros**: fast response since only a few pages are loaded, and less memory usage. **Cons**: a lot of page faults until a stable set of pages is loaded.

When the program branches to another instruction on a page that is not in memory, a **page fault** occurs. The program will be suspended and the OS will take over to load the required page, then restart the program.

When the memory is full and no free frames are available, the OS will use different replacement algorithms to select a frame to be replaced. The algorithms are the same as those introduced in Section 5.3.4.

5.4.3 Address Translation & TLB

The logical address has a page # and an offset. The page number will be used to look up the page table to find the corresponding physical frame #. Then, after adding the offset, the physical address can be obtained and data can be fetched from the memory.

This process involves two memory accesses: page table lookup and data access. To speed up, a **Translation Lookaside Buffer (TLB)** is used. The TLB acts like a cache, which maintains a small number of valid PTEs. If a PTE is not in TLB, then the page table is consulted.

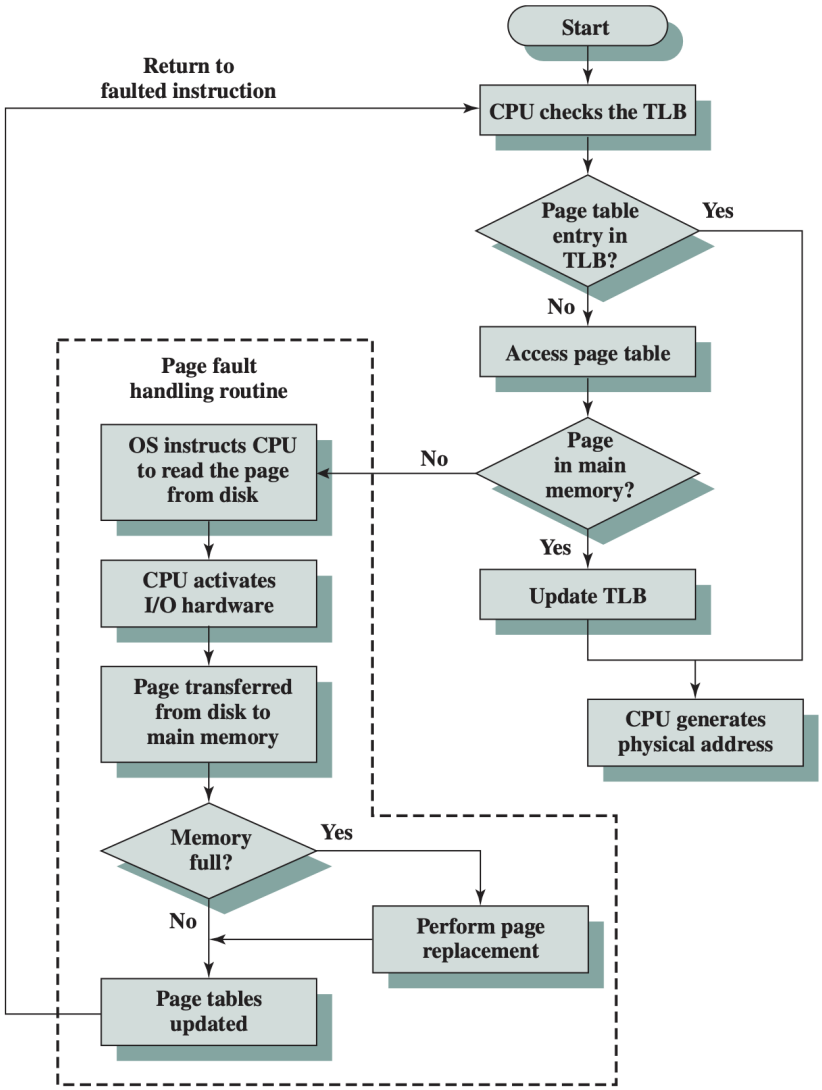


Figure 6: Rough process of paging operation

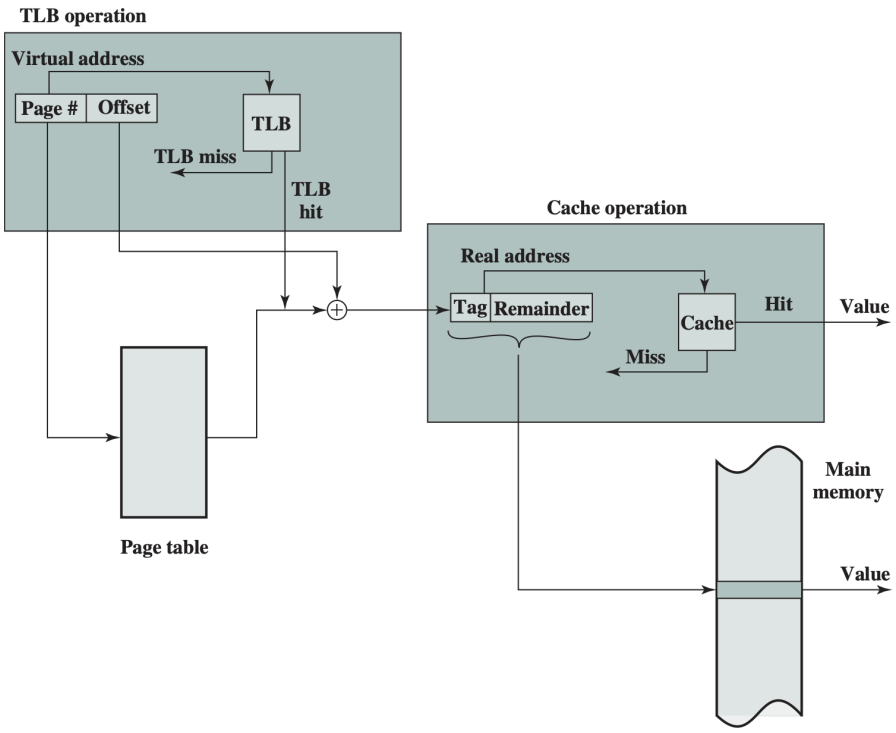


Figure 7: TLB and cache operation

5.5 External Memory

5.5.1 Hard Disk Drives (HDD) - Magnetic Disks

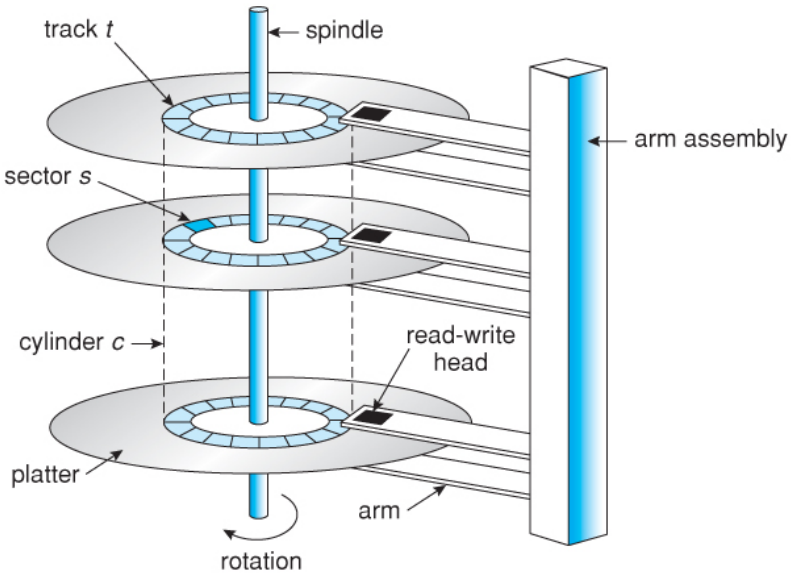


Figure 8: Disk Data Layout

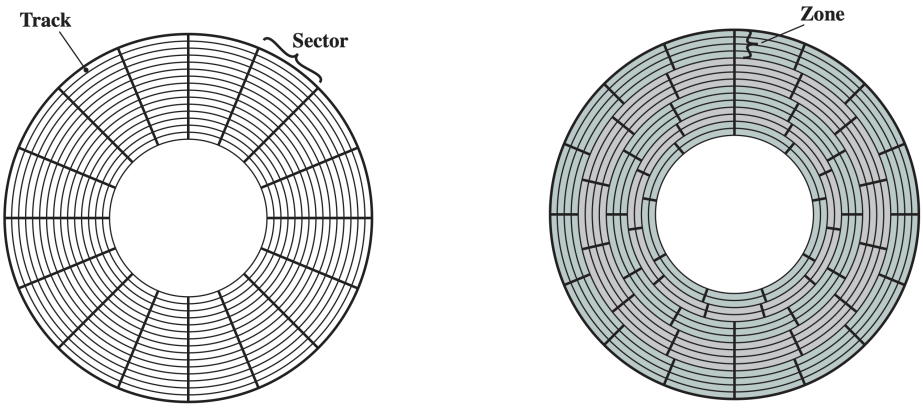
Terminologies about the HDD layout and components

- **Platter**: Magnetically coated disks.
- **Track**: Concentric rings on a platter.
- **Sector**: A segment of a track, usually 512 bytes.
- **Cylinder**: Tracks of different platters that are under the read/write head at the same time.

Formats of Tracks

Tracks contain sectors that hold data and other bits that are useful for the disk controller. The example of a track format can be:

- Each track contains 30 sectors of fixed-length 600 bytes, with 512 bytes for data and 88 bytes for control information.
- Each sector contains several fields:
  - **Gap 1** (17 bytes): Used to separate sectors.
  - **ID Field**: Contains Synch (1 byte), Track, Head, Sector # (4 bytes), and CRC (2 bytes).
  - **Gap 2** (41 bytes): Used to separate ID field and data field.
  - **Data Field** (515 bytes): Contains 1 Synch byte, 512 bytes of data, and 2 CRC bytes.
  - **Gap 3** (40 bytes)



(a) CAV (b) MZR

Figure 9: Disk Layout Methods

Disk Layout Methods

- **Constant Angular Velocity (CAV)**: Blocks of data can be directly addressed by track and sector. Read/write is easy. However, density of data decreases from the inner tracks to the outer tracks, which wastes space.
- **Multiple Zone Recording (MZR)**: Divides the disk into zones, with each zone having a different number of sectors per track. Note that the data density is not exactly the same, but only approximated to be the same. This allows maximised storage capacity.

Disk Access Time

- **Seek Time**: Move the read/write head from one cylinder to another. Depends on start and destination. Typical values: 5 - 15 ms (start up), 0.2 - 1 ms (consecutive tracks).
- **Rotational Latency**: Time for the required sector to rotate under the head. Average latency is the time for half a revolution.

  Example. For a disk rotating at 7200 RPM, the average latency is given by:
  $$\frac{1}{7200 \text{ rotation minute}^{-1}} \times \frac{60 \text{ s}}{1 \text{ minute}} \times 1000 \text{ ms s}^{-1} \times \frac{1}{2} \text{ rotation} = \boxed{4.17 \text{ ms}}$$

- **Data Transfer Time** (t_T): Typically, $t_T \ll$ seek + latency.
  $$t_T = \frac{b}{N} \times \frac{1}{r}$$
  where b is bytes to be transfered, N is bytes per track, and r is the rotation speed in rps.

5.5.2 Redundant Array of Independent Disks (RAID)

Characteristics of RAID

- Several disk drives are arranged together and appear as one single disk to the operating system (a **logical disk**).
- Files are distributed in **strips** across the disks.
- Strips can be in physical blocks, sectors, or other units.
- RAID allows parallel operation.
- Redundant capacity stores parity information which guarantees 24/7 operation.
- Several levels, from RAID 0 to RAID 6.

RAID 0 (Non-redundant)

- Data is written in consecutive sectors in a round-robin fashion.
- Efficient for accessing a block of data.
- One disk failure will cause all strips to be lost without recovery.

RAID 1 (Mirroring)

- Fault tolerant, can recover from multi-disk failure as long as one copy still exists.
- During read, either copy can be used, hence reduce seek time.
- One logical write requires two physical writes, reduces write performance.

RAID 2

- Uses extra disks to store Error Correction Codes (Hamming codes), which is very expensive.
- Number of redundant disks $\approx \log_2$(number of data disks).
- No commercial usage.
- Potential advantage: (when strip size is small) efficient for parallel read.
- Universally controlled spindles (all read/write heads move in parallel without individual control).

RAID 3 (Bit-interleaved Parity)

- One extra disk is used to store parity bit of the data disks.
- Can recover from one disk failure. Refer to Example 5.5 for how parity bit can be used to recover data.
- Writing is also possible with one disk failure, use the above method to alter the parity bit.
- Error correction principle applies from RAID 3 to RAID 6.

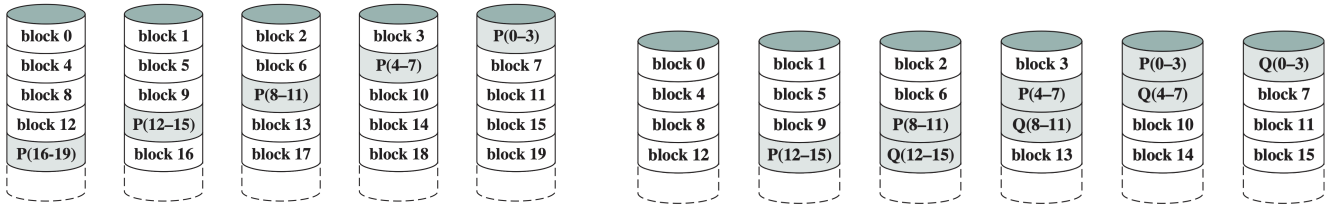
RAID 4 (Block-level Parity)

- Similar to RAID 3, but parity is calculated on a block basis.
- Write to any block would result in the need of recauculating the parity block. Write penalty: 2 reads, 2 writes.
- Methods of writing: (1) Write data, recalculate parity, write parity; or (2) Write data, compare old data with new data, add the difference to parity.
- Individual spindle control.

- No commercial usage.

RAID 5 (Block-level Distributed Parity)

- Uses $(n-1)$ disks to store data.
- Difference from RAID 4: Parity is distributed across all disks.
- Difference from RAID 3: Parity is calculated on a block basis.
- Can endure one disk failure.
- Commonly used in Network Attached Storage (NAS).



(a) RAID Level 5

(b) RAID Level 6

Figure 10: Comparison between RAID 5 and RAID 6

RAID 6 (Dual Redundancy)

- Uses $(n-2)$ disks to store data.
- Similar to RAID 5, but uses two strips for parity, calculated by different methods.
- Parity distributed across different disks, require 2 extra disks.
- Can endure two disk failures.

Table 1: Advantages and Disadvantages of Different RAID Levels

Level	Advantages	Disadvantages
0	I/O performance greatly improved by spreading data across disks. No parity calculation overhead. Very simple design & easy to implement.	One drive failure will result in data in one array to be lost.
1	100% redundancy, no rebuild necessary when disk fails. May sustain multiple simultaneous drive failures. Simplest RAID storage subsystem design.	Highest (100%) disk overhead of all RAID types.
2	High data transfer rate. The higher transfer rate required, the better data disk to ECC disk ratio. Relatively simpler controller design than RAID 3-5.	Expensive. Very high data disk to ECC disk ratio with smaller word sizes.
3	Very high read/write data transfer rate. Disk failure has insignificant impact on throughput. Low ECC to data disk ratio, higher efficiency.	Controller design is faily complex.
4	Very high read data transfer rate. Low ECC to data disk ratio, higher efficiency.	Quite complex controller design. Worst write transaction rate and Write aggregate transfer rate. Difficult and inefficient data rebuild after disk failure.
5	Highest Read data transaction rate. Low ECC to data disk ratio, higher efficiency. Good aggregate transfer rate.	Most complex controller design. Difficult to rebuild data after disk failure as compared to RAID 1.
6	Provides highest data fault tolerance. Can sustain multiple simultaneous drive failures.	More complex controller design than RAID 5. Controller overhead to compute parity is extremely high.

5.5.3 Solid State Drives (SSD)

SSDs are non-volatile storage devices that are based on semiconductor technology. They have limited write cycles, but are faster than HDDs.

Advantages of SDDs over HDDs:

- Faster I/O operation performance.
- Lower power consumption, cooler and quieter.
- Longer lifespan – no mechanical parts for read/write.
- Durability – less susceptible to physical shock.
- Lower access time.

6 Input & Output

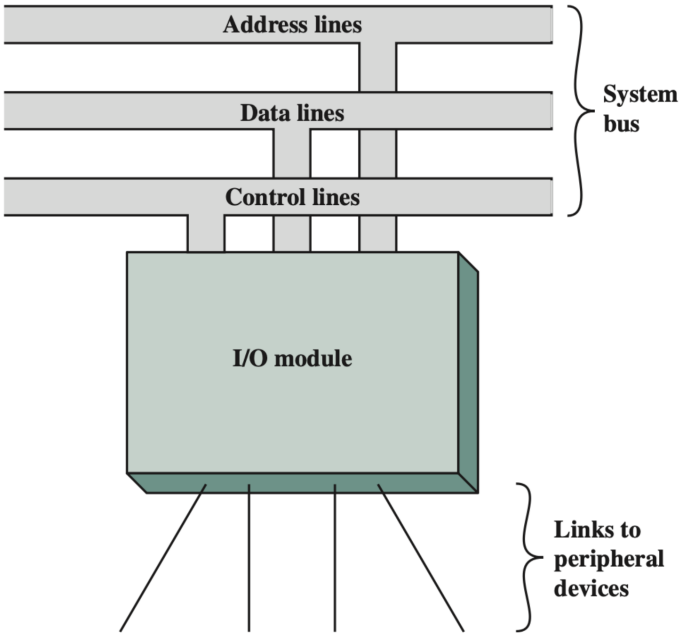


Figure 11: Generic Model of an I/O Module

6.1 Generic Model of I/O Modules

An I/O module is:

- an interface between the processor and memory via the system bus or central switch
- an interface to one ore more peripheral devices through tailored data links

Major requirements on an I/O module:

- **Asynchronous timing**.
- **Command decoding**: interpret the commands sent from the bus, e.g. `SEEK`.
- **Data**: exchange data via the data bus.
- **Status reporting**: e.g. Ready, Busy, Out of Paper (printer).
- **Address recognition**: identify the address of a peripheral device.
- **Data buffering**: for speeding up transaction because the device may be slower.
- **Error detection and correction**.

The CPU operates I/O devices by reading/writing from/to the devices' status/control/data registers. The registers are mapped in two ways:

1. **Memory-mapped I/O**: the I/O device registers are mapped into the same address space as the memory. The CPU can access the I/O device registers as if they were memory locations.
2. **Port-mapped I/O**: the I/O device registers are mapped into a separate address space from the memory. The CPU uses special I/O instructions to access the I/O device registers.

6.2 I/O Techniques

Three types of I/O techniques for interacting with I/O devices:

- Not using interrupts: **Programmed I/O**
- Using interrupts:
  - **Interrupt-driven I/O** (Memory ↔ CPU ↔ I/O)
  - **Direct Memory Access (DMA)** (Memory ↔ I/O)

.

6.2.1 Programmed I/O

The processor executes a program to control the I/O devices via the Control and Status Registers (CSR). When the CPU sends a command to the device, it must wait for the device to complete. This wastes CPU time.

6.2.2 Interrupt-Driven I/O

This technique attempts to alleviate the issues of Programmed I/O. After sending a command to the I/O module, the CPU will carry on with other tasks. When the I/O module finishes, it sends an interrupt signal to the CPU. The CPU will then handle the interrupt and return to its tasks.

The CPU sends a interrupt acknowledge signal (`INTA`) to the I/O module to indicate receipt of the interrupt signal. The last instruction of the interrupt routine is a return from interrupt instruction (`RETI`). Refer to Section 4.2.7 for interrupt handling procedures.

6.2.3 Direct Memory Access (DMA)

Interrupt-Driven I/O still involves CPU operations, which can be minimised by a special purpose processor – **Input-Output Processor (IOP)**.

To resolve bus conflicts between the CPU and IOP, the IOP steals cycles from the CPU by sending a signal to the

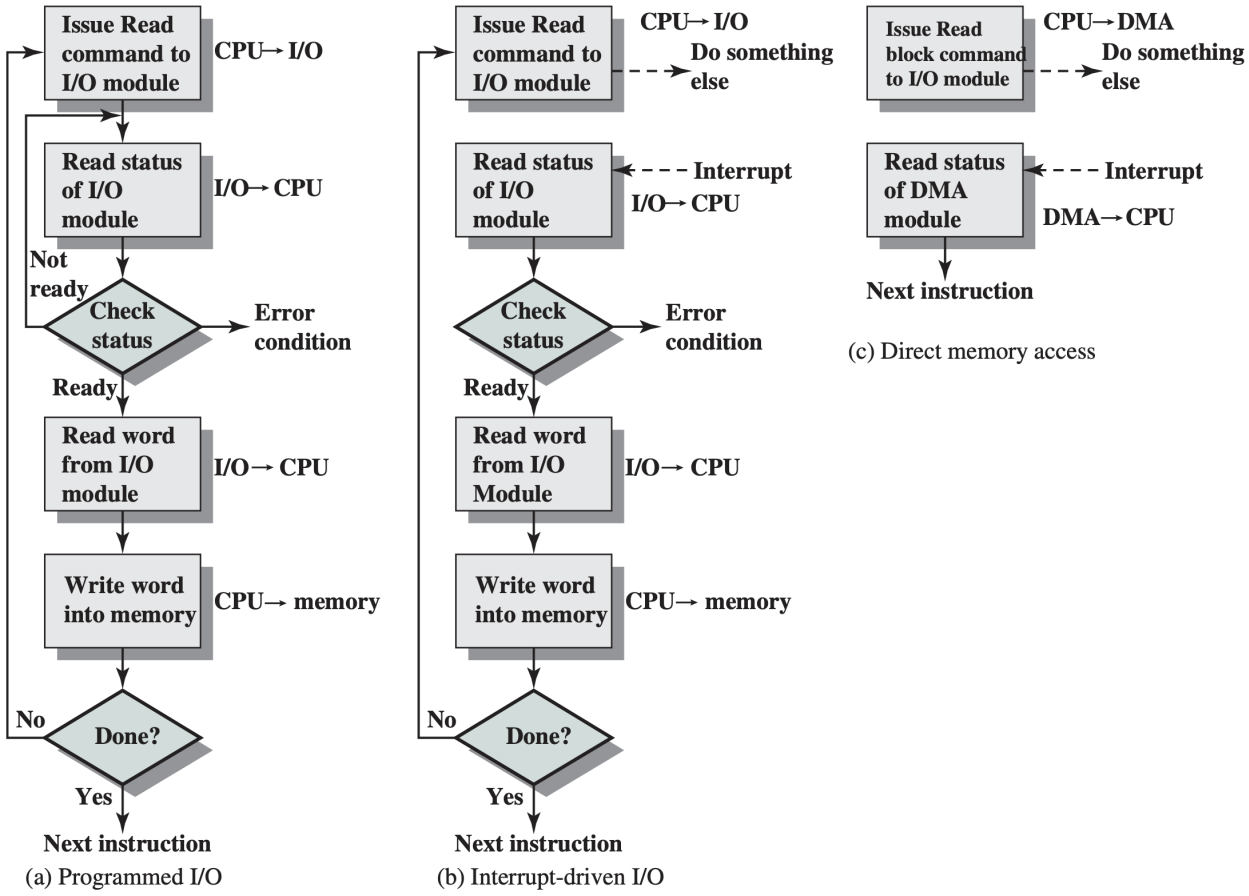


Figure 12: I/O Techniques

CPU. The CPU will now see an elongated clock cycle, and will wait until the cycle ends to continue.

Difference between Interrupt-Driven I/O and DMA: The "interruption" in DMA is **within** one instruction execution cycle, while the interruption in Interrupt-Driven I/O is after the instruction execution cycle, as illustrated in Figure 13.

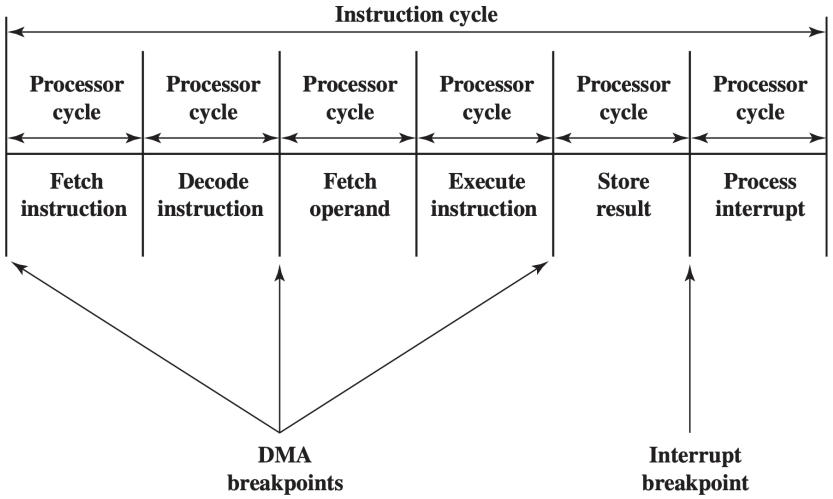


Figure 13: Difference between Interrupt-Driven I/O and DMA

7 Instruction Sets

7.1 More on Instruction Format

7.1.1 Arithmetic & Logical Instructions

Arithmetic operations treat operands as numbers and have to consider the sign of the operands. Arithmetic shifts are equivalent to multiplication (left shift) or division (right shift) by 2 with remainders shifted out. The new bits are filled with sign bits on the left, and 0's on the right.

Logical operations treat operands as bit patterns. Logical shifts simply discard the bits shifted out and replenish the new bits with 0's.

There are also **rotate** operations, which put the bits shifted out back into the other end of the number.

7.1.2 Procedures & Function Calls

A procedure consists of multiple instructions that are executed in sequence. Within a procedure, instructions can be given to execute another procedure. For the CPU to know where to go and where to return after the called procedure is done, the return addresses need to be stored, which is done by a stack. The latest return address will be at the top of the stack, and the CPU will pop it when it reaches a return instruction.

7.1.3 Instruction Operands

In most applications, instructions either have three, two, one, or zero operands (or addresses). Symbolically, they are represented as:

# of Operands	Symbolic Representation	Interpretation
3	`OP A, B, C`	`A` ← `B OP C`
2	`OP A, B`	`A` ← `A OP B`
1	`OP A`	`AC` ← `AC OP A`
0	`OP`	`T` ← `(T-1) OP T`

Note: `AC` = accumulator, `T` = top of stack, `(T-1)` = second element of the stack

Instruction operands can either be in main memory or in registers.

7.1.4 Registers

- **General Purpose Registers**: registers that can be used freely;
- **Dedicated Purpose Registers**: e.g. program counter (PC), instruction register (IR), stack pointer (SP), processor status word (PSW), flag register;

7.1.5 Data Types

Two types of data types: **(1)** Numeric (integer, floating point); **(2)** Non-numeric (character, binary data). The lengths are typically 8, 16, 32, or 64 bits.

For MIPS architecture (a family of reduced instruction set computer (RISC), not ARM or x86), have 9 basic data types: **(1)** signed and unsigned bytes; **(2)** signed and unsigned half-words; **(3)** signed and unsigned words; **(4)** double words; **(5)** single-precision floating point (32 bits); **(6)** double-precision floating point (64 bits).

For ARM architecture, it supports datatypes of **(1)** byte (8 bits); **(2)** half-word (16 bits); and **(3)** word (32 bits) in length. It only provides unsigned integers, nonnegative integers, and two's complement integers. The ARM architecture does not provide floating point hardware and they must be emulated in software.

7.2 Addressing Modes

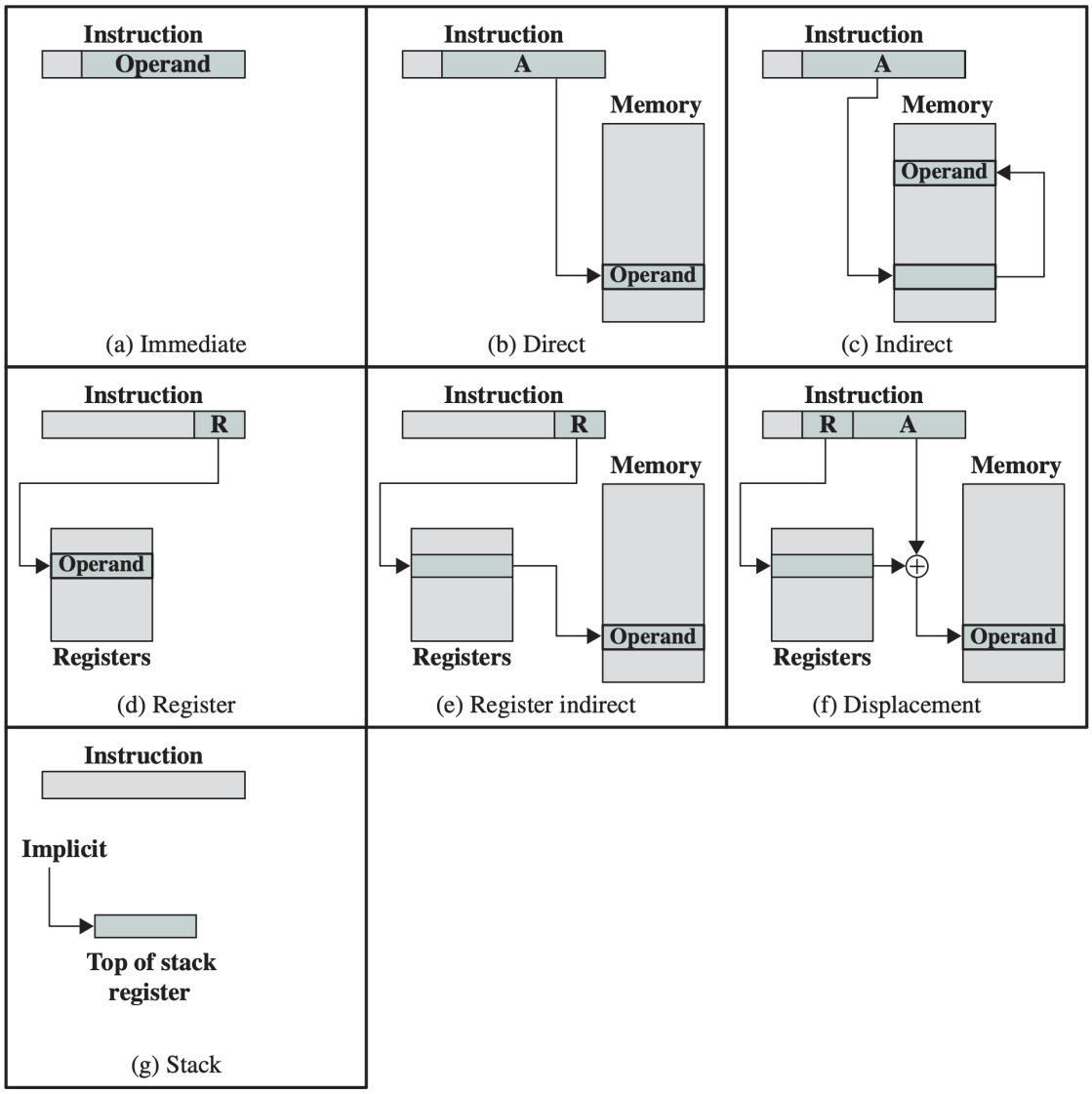


Figure 14: Addressing Modes

Remark. Notations used in this section: `A` = contents of an address (operand) field of an instruction; `R` = contents of an address field that refers to a register; `EA` = effective address (actual address) of the location where the referenced operand is stored; `(X)` = contents of memory location or register `X`.
(The parenthesis notation is similar to a dereference operator of a pointer in C/C++.)

7.2.1 Immediate

- **Notation**: Operand = `A`
- **Advantage**: No memory references needed.
- **Disadvantage**: Limited operand magnitude.

7.2.2 Direct

- **Notation**: `EA = A`
- **Advantage**: Simple and increased operand magnitude.
- **Disadvantage**: Limited address space. (Range of memory addresses accessible to the instruction depends on the size of the address field.)

7.2.3 Indirect

- **Notation**: `EA = (A)`
- **Advantage**: Increased address space.
- **Disadvantage**: Multiple memory references needed.

7.2.4 Register

- **Notation**: `EA = R`
- Similar to direct addressing, but the operand is in a register.
- **Advantage**: Fast access to operands.
- **Disadvantage**: Limited number of registers. (e.g. 32 registers in MIPS)

7.2.5 Register Indirect

- **Notation**: `EA = (R)`
- Similar to indirect addressing, but the operand is in a register.
- **Advantage & Disadvantage**: Same as indirect addressing.

7.2.6 Displacement

- **Notation**: `EA = A + (R)`
- **Usage**: Accessing local variables or parameters in a function call, or accessing an array.
- Registers involved: PC, SP, and base pointer register.
- **Advantage**: Flexible.
- **Disadvantage**: Complex.

7.2.7 Stack

- **Notation**: `EA` = Top of Stack
- Uses the stack pointer register (SP). This is implied in the instruction.
- **Usage**: Mainly for `PUSH` and `POP` instructions.
- **Advantage**: No memory reference.
- **Disadvantage**: Limited applicability.

7.3 Assembly Language Programming

Remark. Assembly Language is very instruction set architecture (ISA) dependent. The language differs from one architecture to another. This course focuses on a hypothetical machine, with the following features:

- Comments start with a `#`[5]and continue to the end of the line;
- Destination operands are on the **right** of the operands list;
- Instructions are case insensitive;

7.3.1 Syntax

Definition 7.1 (Assembly Language Syntax)**.** Each line of assembly language consists of:

```
LABEL: OPERATION_MNEMONIC OPERAND_1, OPERAND_2, ..., OPERAND_N ; COMMENT
```

- `LABEL`: optional, used to identify a location in the program;
- `OPERATION_MNEMONIC`: the operation to be performed;
- `OPERAND_1, OPERAND_2, ..., OPERAND_N`: the operands for the operation;
- `COMMENT`: optional, used to explain the purpose of the instruction.

7.3.2 Assembler Directives

Like compiler directives in C/C++, assembly language also has assembler directives. Assembler directives are for the assembler to perform an action or change a setting, they are not translated into machine code. Assembler directives start with a dot.

Assembler Directive	Description
`.DATA`	Adds the subsequent data to the data segment.
`.TEXT`	Adds the subsequent code to the text (program) segment.
`.GLOBAL NAME`	Makes `NAME` available to external files.
`.SPACE EXPRESSION`	Reserves spaces with the amount specified by the value of `EXPRESSION` in bytes. Reserved space is filled with 0's.
`.WORD VALUE_1[, VALUE_2, ...]`	Puts the values in successive memory locations.

7.3.3 Flow Control

1. **`IF...THEN...ELSE...` structure**

   Example. The following C/C++ code:

   ```
   if (a[0] > a[1]) x = a[0];
   else x = a[1];
   ```

   is equivalent to the following assembly code:

   ```
           .DATA   ; declares the data segment
   a:      .WORD 1 ; a[0] = 1
           .WORD 3 ; a[1] = 3
   x:      .WORD 4 ; x = 4
           .TEXT   ; declares the program segment
   main:
           LD      [a], R8     ; load address of a into R8
           LD      0(R8), R9   ; load a[0] into R9 (displacement addressing)
           LD      4(R8), R10  ; load a[1] into R10 (displacement addressing)
           BGT     R9, R10, f1 ; if a[0] > a[1], branch to f1
           ST      R10, x      ; else: x = a[1]
   ```

[5]For using LaTeX's `minted` package for syntex highlighting, the comment character in these notes will be ";".

```
        BR      f2
f1:     ST      R9, x       ; x = a[0]
f2:     RET                 ; return to the caller
```

2. **`FOR` loop**

   Example. The following C/C++ code:

   ```
   a = 0;
   for (i = 0; i < 10; i++) a += i;
   ```

   is equivalent to the following assembly code:

   ```
           .DATA   ; declares the data segment
   a:      .WORD 0
           .TEXT
   main:
           SUB     R8, R8, R8      ; prepare R8 = 0 as the counter
           LD      0xa, R9         ; constant 10 in R9 (hexadecimal)
           LD      0x1, R10        ; constant 1 in R10 for incrementing R8
           SUB     R11, R11, R11   ; use R11 as sum
   L:      ADD     R11, R8, R11    ; R11 += R8
           ADD     R8, R10, R8     ; R8++
           BGT     R9, R8, L       ; branch to L if R9 (10) > R8 (counter)
           ST      R11, [a]
           RET
   ```

3. **`WHILE` loop** – very similar to `FOR` loops.

4. **Function calls**
   Use `CALL` to make a function call, and `RET` to return from the function. Input/return parameters are passed in registers and need to be documented by the developer. If the function modifies some other registers, either document this behaviour, or use `PUSH` and `POP` to save and restore the registers before and after the function.

   Example. The following assembly code shows how to call a function `c_mult` that multiplies two complex numbers. ($\mathrm{Re}(a \cdot b) = \mathrm{Re}(a) \cdot \mathrm{Re}(b) - \mathrm{Im}(a) \cdot \mathrm{Im}(b)$ and $\mathrm{Im}(a \cdot b) = \mathrm{Re}(a) \cdot \mathrm{Im}(b) + \mathrm{Im}(a) \cdot \mathrm{Re}(b)$)

   ```
           .DATA
   ar:     .WORD 1     ; Re(a) = 1 (real part of complex number a)
   ai:     .WORD 5     ; Im(a) = 5 (imaginary part of complex number a)
   br:     .WORD 2
   bi:     .WORD 3
   cr:     .WORD 0
   ci:     .WORD 0
           .TEXT
   main:
           LD      [ar], R8    ; Prepare parameters for c_mult
           LD      [ai], R9
           LD      [br], R10
           LD      [bi], R11
           CALL    c_mult      ; Call c_mult
           ST      R12, [cr]   ; Store the result
           ST      R13, [ci]
           RET
   c_mult:
           ; multiply two complex numbers
           ; input: R8 = Re(a), R9 = Im(a), R10 = Re(b), R11 = Im(b)
           ; output: R12 = Re(a*b), R13 = Im(a*b)
           PUSH    R14         ; use R14 as a temporary register
           MUL     R8, R10, R12
           MUL     R9, R11, R14
           SUB     R12, R14, R12
           MUL     R8, R11, R13
           MUL     R9, R10, R14
           ADD     R13, R14, R13
           POP     R14
           RET
   ```

7.4 Operating System Support

The operating system (OS) is a software that controls the execution of programs and manages hardware resources. It allows a computer to be used efficiently and conveniently.

Services provided by the OS:

- **Program Creation** – through compilers, assemblers, editors, debuggers etc.
- **Program Execution** – through loading programs into memory and preparing resources for the program
- **I/O Access** – through providing a uniform I/O interface while the implementation is left to the OS
- **File System Management**
- **System Access** – control access to system resources to prevent unauthorised users
- **Error Detection and Response**
- **Accounting** – collecting usage statistics and monitoring performance parameters

7.4.1 OS Protection Scheme

Most OSs uses two modes of operation: **user mode** and **kernel mode**. The CPU will execute in different modes to facilitate protection. Some OSs may use up to four modes. Some resources are only accessible in kernel mode.

An OS is supposed to be well-tested, while bugs may exist in user programs.

OS functions are usually accessed via special entry points called **system calls**. Upon a system call, the CPU will switch from user mode to kernel mode.

7.4.2 Multitasking, Time Sharing & Process Scheduling

To fully utilise the CPU, the CPU is shared among multiple processes. Each process is given a time slice to execute. When it uses up its time slice, the CPU will suspend the execution and switch to another process.

The OS maintains a queue of processes in which the order of execution depends on several factors, such as priority, waiting time, whether the process has used up its time slice (i.e. CPU-bound jobs), the current system load, etc.

7.5 Processor Organisation

The processor executes instructions via moving data to the desired location and performing data transformations/processing using the ALU. These operations are controlled by the control signals generated by the control unit (CU).

7.5.1 Data Movement and Transformation

Recall the stages of an instruction execution cycle at Section 4.2: **(1)** IF, **(2)** ID, **(3)** CO, **(4)** OF, **(5)** EI, **(6)** WO. For modern processors, only `LD` and `ST` instructions are related to memory operands and they do not need the `EI` stage. Other instructions usually do not need the `CO` stage.

Example. Suppose the instruction `ADD A, B, C` uses direct addressing mode, and A, B, and C are supplied in the words following the instruction. Write down the involved data movement.

```
; IF - Instruction Fetch
MAR ← PC
PC ← PC + 4 ; PC now points to A
IR ← mem[MAR]
; ID - Instruction Decode
; CO(A) - Calculate Operand (A)
MAR ← PC ; MAR has address to address of A
PC ← PC + 4 ; PC now points to B
MBR ← mem[MAR] ; MBR has the address of A
; FO(A) - Fetch Operand (A)
MAR ← MBR ; MAR has the address of A
MBR ← mem[MAR] ; MBR has the value of A
ALU_IN_1 ← MBR ; ALU_IN_1 has the value of A
; CO(B) - Calculate Operand (B)
MAR ← PC ; MAR has address to address of B
PC ← PC + 4 ; PC now points to C
MBR ← mem[MAR] ; MBR has the address of B
; FO(B) - Fetch Operand (B)
MAR ← MBR ; MAR has the address of B
MBR ← mem[MAR] ; MBR has the value of B
ALU_IN_2 ← MBR ; ALU_IN_2 has the value of B
; EI - Execute Instruction
ALU_OUT ←ALU ALU_IN_1 + ALU_IN_2
; WO - Write Operand
MAR ← PC ; MAR has address to address of C
PC ← PC + 4 ; PC points to next instruction
MBR ← mem[MAR] ; MBR has the address of C
MAR ← MBR ; MAR has the address of C
MBR ← ALU_OUT ; MBR has the value of A + B
mem[MAR] ← MBR ; Writes the result to C
```

7.5.2 Instruction Pipeline

Main purpose: to increase the throughput of instruction execution.

Suppose an instruction has 5 stages, and each stage takes exactly 1 clock cycle to execute, then, an ideal pipeline will

finish executing n instructions in $(5+n-1)$ clock cycles, as shown in Figure 15, as opposed to $5n$ clock cycles if executed sequentially.

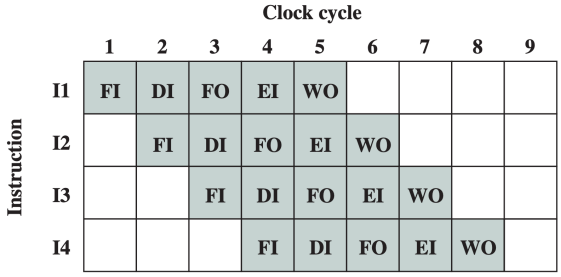


Figure 15: Ideal 5-stage Pipeline Time Diagram

However, in practice, this pipelining strategy will cause problems, such as when I_2 operates on data produced by I_1, but the IF stage already took place before WO of I_1 is performed, incorrect data will arise. These problems are called **pipeline hazards**.

Pipeline hazards can be classified into three categories:

1. **Resource Hazard** (or *structural hazard*): e.g. when PC increment and an ALU operation happen at the same time, the ALU (the hardware resource) will be in conflict.
   Solution: Add more resources. E.g. a dedicated incrementer for the PC, separate data/instruction caches (split cache), etc.
2. **Data Hazard**: instruction waiting for the result from previous instructions due to data dependency. Three types of data hazards:
   (a) **Read After Write (RAW)**: I_1 writes to a register, and I_2 reads from the same register. Occurs if the read happens before the write is done.
   (b) **Write After Read (WAR)**: I_1 reads from a register, and I_2 writes to the same register. Occurs if the write happens before the read is done.
   (c) **Write After Write (WAW)**: I_1 and I_2 write to the same register. Occurs if the second write happens before the first write is done.

   Note that only RAW occurs in pipeline (refer to Figure 15). The others occur in parallel systems.
   Solutions:
   - **Stalling**: wait until the dependent instruction is done. This will put the processor in idle, which wastes time.
   - **Rearrange Instructions**: rearrange the instructions so that the instruction that depends on the result of the previous instruction is only executed after the previous instruction is done. This is not always possible.
   - **Data Forwarding**: (a hardware solution) by forwarding the result from `ALU_out` to `ALU_in` when the next instruction is executed.
3. **Control Hazard**: occurs when the `IF` stage of the next instruction cannot start until the previous branch is resolved.
   Solution: Unsolvable. The CPU must either wait, or guess the branch target by **branch prediction**. Two types of branch prediction:
   (a) **Static Prediction**: always predict the same branch target. E.g. always predict the branch will be taken, or will not be taken. 50% correct on average, in some cases like `for` loops, higher accuracy since most of the time the branch will be taken. Potential problem: when the branch target is on a different page, the branch penalty will be higher to involve the time taken to resolve the page fault.
   (b) **Dynamic Prediction**:
   - **1-bit Prediction**: use a single bit for prediction strategy. If the prediction is correct, keep predicting the same. If the prediction is wrong, change the prediction. Problem: at the end of a `for` loop, the prediction must be wrong, if another `for` loop follows, the prediction will be wrong again.
   - **2-bit Prediction**: requires two consecutive wrong predictions to change the prediction.

7.6 Reduced Instruction Set Computer (RISC) Architecture

Characteristics of RISC architecture:

- **Load/Store Architecture**: all instructions are register-to-register, with the only exceptions being `LD` and `ST` instructions, which access memory.
- **Fixed-length and Simple, Fixed-format Instructions**: all instructions are of the same length, and have the same format. `OF` can always be done directly since the operands are always in the same place. ($\Rightarrow$ high clock rate, low clock cycle time)

- **Fewer Addressing Modes**: allows simple CPU and faster clock rate. Pipeline is easier to implement since there are less cases to consider ($\Rightarrow$ low CPI).
- **More Instructions**: due to the fixed-length and simple instructions, more instructions are needed to perform the same task. However, empirical study shows that the increase in instruction count does not significantly affect the performance. (e.g. increase in instruction count by 20% $\Rightarrow$ CPI reduced by 2 to 4 times)
- **Extensive Software and Hardware Pipelining**: use instruction-level parallelism and extensive software and hardware techniques to eliminate pipeline hazards.
- **Compiler Optimisation**: relies on the compiler to optimise the code.

Remark. The RISC characteristics can be applied to any processor design, not just RISC. The RISC is a design philosophy rather than a specific architecture. Some hybrid designs feature both RISC and CISC[6]design principles.

[6]Complex Instruction Set Computer