2021

Computer Organization

CPSC 2105

LEE, SUK

COLUMBUS STATE UNIVERSITY | 4225 University Ave, Columbus, GA 31907

Contents

Chapter 1: Introduction

In this chapter, we present an overview of basic computer organization and representation of data in computers.

Objectives

By the end of this chapter you should be able to:

- Explain the decimal numbers and binary numbers systems.
- Explain the number conversion among binary, decimal and hexadecimal numbers.
- Understand a basic binary addition and the overflow due to a fixed number of bits.
- Represent binary negative numbers to sign/magnitude numbers and two's complement numbers.
- Demonstrate basic skill in taking two's complement numbers.

1.1. Overview on Computing Hardware

Basic Definitions

Hardware: Physical parts of a computer. Everything you can touch.

Transistor: A tiny electrically operated switch that can alternate between “on” and “off”. Fig. 1-1 shows a transistor which has three pins.

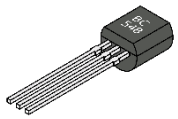


Fig. 1-1. Transistor

Chip (Microchip): A tiny piece of silicon that contains millions of transistors and other electronic components, as shown in Fig. 1-2. Your CPU (Central Processing Unit) is one of Microchips.

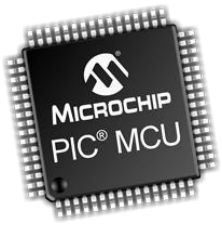


Fig. 1-2. Chip (Microchip)

Hardware Overview

Typical Personal Computer System consists of lots of components, as shown in Fig. 1-3:

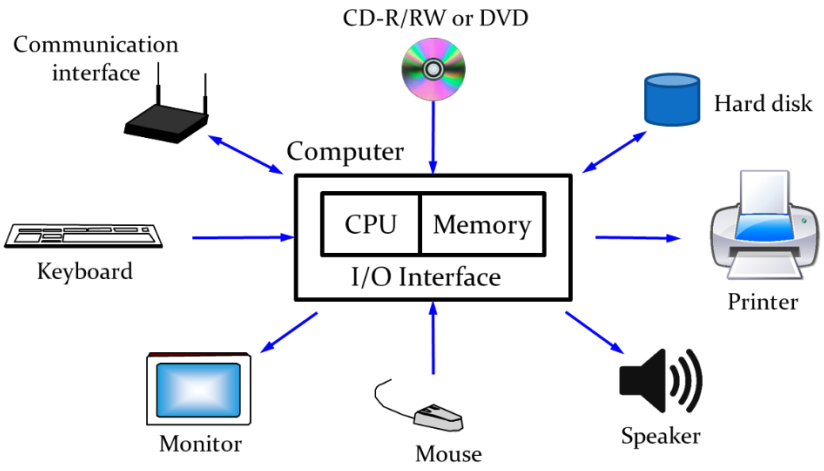


Fig. 1-3. Personal Computer System

- System unit: Motherboard is the main circuit board for the computer, which includes CPU, memory, ports, etc.
- Secondary storage devices can "permanently" hold data and information. Some examples include Floppy disk, hard disk drives, Magnetic tape, CD-ROM, CD-R, CD-RW, DVD-ROM, DVD-R, DVD-RW.
  - CD-ROM stands for Compact Disc - Read Only Memory.
  - CD-R stands for Compact Disc - Recordable and can be written to only once.
  - CD-RW stands for Compact Disc - Re-writeable (or Read/Write).
  - DVD-ROM stands for Digital Versatile Disc - Read Only Memory.
  - DVD-R stands for Digital Versatile Disc - Recordable and can be written to only once.
  - DVD-RW stands for Digital Versatile Disc - Re-writeable (or Read/Write).
- Input devices translate data into a form the computer can understand.
  - Keyboard, mouse, trackball, and touchpad
- Output devices translate information into a form human can understand.
  - Monitor (or Display Screen), Printer, Speaker
- Communications devices send/receive data to/from other computers
  - Modem, network card

1.2. Digital Discipline

Binary Values

In digital discipline, there are two discrete values:

- 1, TRUE, HIGH
- 0, FALSE, LOW

1 and 0 can be represented with voltage levels. If the voltage level is high, it represents 1 bit. If the voltage level is low, it represents 0 bit. The two discrete values can be also represented with rotating gears, fluid levels, etc. Digital circuits of your computer use voltage levels to represent 1 and 0. This is a binary digit so we simply call it “bit”.

Number Systems

Decimal numbers can be expressed as shown in Fig. 1-4. The rightmost digit represents 1’s column. As the digit moves forward to left-side, the weight of each digit increases as power of 10. In the figure, we can read the number, as follows: five thousand, three hundred, seven ten, and four one.

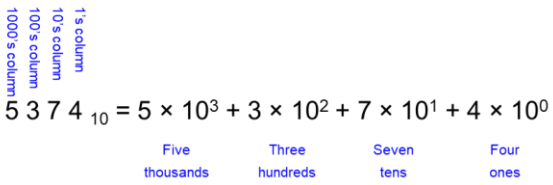


Fig. 1-4. Decimal number

Binary number also can be expressed as shown in Fig. 1-5. Here, the rightmost digit represents 1’s column. As the digit moves forward to left-side, the weight of each digit increases as power of 2. In the figure, we have a binary number 1101. Each bit represents a different weight, the first bit (1) for 8’s column, the second one (1) for 4’s column, the third one (0) for 2’s column, and the last one (1) for 1’s column.

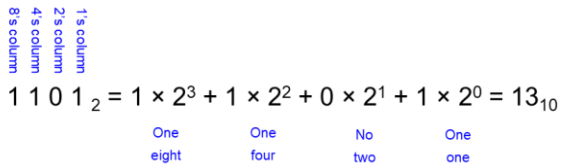


Fig. 1-5. Binary number

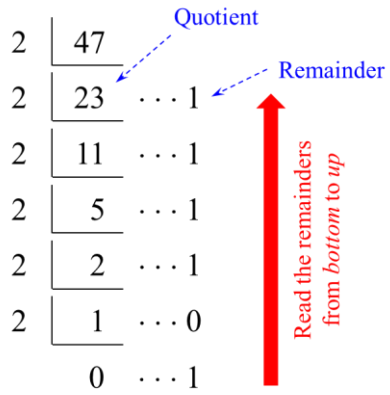


Fig. 1-6. Number Conversion

Let's convert the decimal number 47 to a binary number. We can start to divide the number with the divisor, where the divisor is always 2. You will the quotient 23 and a remainder 1. Keep repeat this process until the dividend becomes zero, as shown in Fig. 1-6. Now let's read the remainders from bottom to up, 101111 is the binary representation of the decimal number 47.

1.3. Definitions

The bit is the most basic unit of information in computing and digital communications.

```
1 0 0 1 0 1 1 0
```

The above 8 bits show an example of the binary number. The first bit we call it most significant bit (msb), whereas the last bit we call it least significant bit (lsb). A group of 8-bit, we call it a byte.

The large powers of two can be expressed as shown below:

- 2^{10} = 1 kilo: 2^{10} (1024) is approximately equal to 1000
- 2^{20} = 1 mega: 2^{20} (1,048,576) is approximately equal to 1 million
- 2^{30} = 1 giga: 2^{30} (1,073,741,824) is approximately equal to 1 billion
- 2^{40} = 1 tera: 2^{40} (1,099,511,627,776) is approximately equal to 1 trillion

1.4. Signed Binary Numbers

There are two ways of representation of signed numbers, i.e. sign-magnitude form and two's complement form.

Sign-Magnitude

The sign-magnitude binary format is the simplest conceptual format. To represent a number in sign-magnitude, we simply use the leftmost bit to represent the sign, where 0 means positive and 1 means negative. The remaining bits represent the magnitude (absolute value).

For example, let's represent +6 and -6 with 4-bit sign-magnitude form. The absolute value for both numbers is equal to |6| = 110. The sign bit for +6 is 0, whereas the sign bit for -6 is 1. We can express both +6 and -6 as shown below:

- +6 = 0110
- -6 = 1110

One of problem in this form is that the addition doesn't work. If you add these two number as shown below, the result is not correct.

```
  1110
+ 0110
1 0100  (wrong~!)
```

Another issue is that there are two representation of 0, i.e. 1000 (-0) and 0000 (+0). That reduces the possible number representation.

Two's Complement

Two's complement is the most common method of representing signed integers on computers. The msb has value of -2^{N-1}, where N is the total number of bits. for example, if you have 4-bit two's complement, the most positive 4-bit number is 0111 that is equal to 7. The most negative 4-bit number is 1000 that is equal to -8. In contrast to sign-magnitude form, addition works in two's complement form and there is single representation for 0.

You can find some YouTube video how to convert the number into two's complement number in the following link:

Two's Complement Representation of Negative Numbers

Chapter 2: Overview on Computing Hardware

In this chapter, we explore basic logic gates that take one or more binary inputs and produce a binary output. In addition, we cover how CMOS transistors are used to implement logic gates.

Objectives

By the end of this chapter you should be able to:

- Explain the basic logic gates and logic levels.
- Explain what noise is and what noise margin is.
- Understand two types of transistors and how to build logic gates from these transistors.
- Demonstrate basic logic gates from the corresponding CMOS gates.

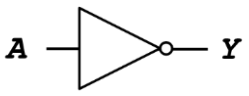
2.1. Logic Gates

Basically, logic gates perform logic functions in the computer system.

Single-Input Logic Gates

The single-input logic gates take a single input and produce an output, which include a logic NOT gate and a buffer. The logic NOT gate is the most basic of all the logic gates and flips an input value. If the input A is "0" or LOW, the NOT gate produces the output "1" or HIGH. If the input A is "1" or HIGH, then it produces the output "0" or LOW, giving us the Boolean expression of: $Y = \bar{A}$. The following figure shows the symbol and the truth table of the logic NOT gate.

Symbol



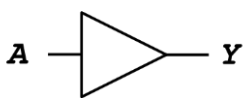
Truth Table

A	Y
0	1
1	0

Fig. 2-1. Logic NOT gate

A buffer is a basic logic gate that passes its input, unchanged, to its output. It just repeats the input signal, giving us the Boolean expression of: $Y = A$. The main purpose of a buffer is to regenerate the input, usually using a strong high and a strong low. The following figure shows the symbol and the truth table of the buffer.

Symbol



Truth Table

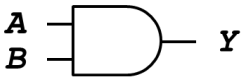
A	Y
0	0
1	1

Fig. 2-2. Buffer gate

Two-Input Logic Gates

For a two-input AND gate, the output Y is true if both input A and input B are "1" or HIGH, giving us the Boolean expression of: $Y = A \cdot B$. Note that the Boolean expression for a two-input AND gate can be written as: $A \cdot B$ or just simply AB without the point. The following figure shows the symbol and the truth table of the two-input AND gate.

Symbol



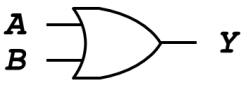
Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 2-3. Two-input AND gate

For a two-input OR gate, the output Y is true or HIGH if either input A or input B is "1" or HIGH, giving us the Boolean expression of: $Y = A + B$. Note that it produces the output Y = 0 if only if both of inputs are "0" or LOW. The following figure shows the symbol and the truth table of the two-input OR gate.

Symbol



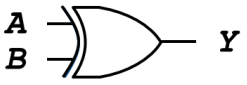
Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 2-4. Two-input OR gate

For a two-input XOR gate, the output Y is true or HIGH if either input A or input B is true, but not both, giving us the Boolean expression of: $Y = A \cdot \bar{B} + \bar{A} \cdot B = A \oplus B$. The following figure shows the symbol and the truth table of the two-input XOR gate.

Symbol



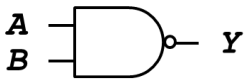
Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 2-5. Two-input XOR gate

For a two-input NAND gate, the output Y is NOT true if both input A and input B are "1" or HIGH, giving us the Boolean expression of: $Y = \overline{A \cdot B}$. The following figure shows the symbol and the truth table of the two-input NAND gate.

Symbol



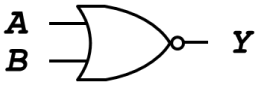
Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 2-6. Two-input NAND gate

For a two-input NOR gate, the output Y is true if both input A and input B are not true, giving us the Boolean Expression of: $Y = \overline{A + B}$. The following figure shows the symbol and the truth table of the two-input NOR gate.

Symbol



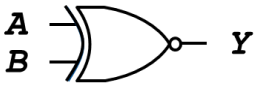
Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 2-7. Two-input NOR gate

For a two-input XNOR gate, the output Y is true if both input A and input B are the same, either true or false, giving us the Boolean expression of: $Y = (A \cdot B) + (\bar{A} \cdot \bar{B}) = \overline{A \oplus B}$. The following figure shows the symbol and the truth table of the two-input XNOR gate.

Symbol



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 2-8. Two-input XNOR gate

2.2. Noise

Anything that degrades the signal can be noise. The noise includes resister, power supply noise, coupling to neighboring wires, etc. The following figure shows how the noise affect the signal strength. There are two buffers connected serially. The output of one buffer connected to the input of the other one. Assume the output voltage of the first one is 5V. The input voltage of the second one may be 4.5 V due to the wire noise which can degrade the signal strength.

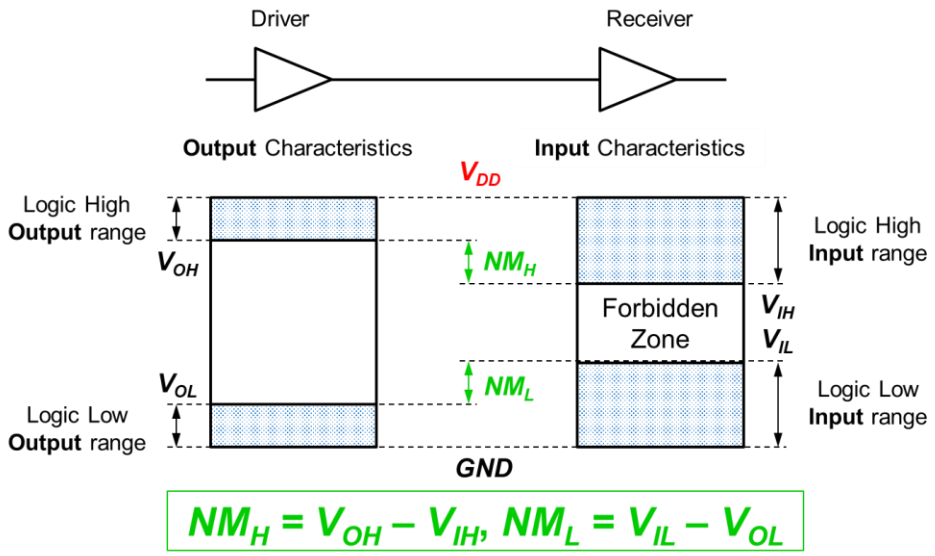


Fig. 2-9. Noise between Driver and Receiver

In a digital circuit or system, with logically valid inputs, every circuit element must produce logically valid outputs, called static discipline. Integrated circuits use limited ranges of voltages to represent discrete values as follows:

The output characteristics:

- The logic high output ranges from V_{DD} to V_{OH}
- The logic low output ranges from V_{OL} to GND

The input Characteristics:

- The logic high input ranges from V_{DD} to V_{IH}
- The logic low input ranges from V_{IL} to GND

The voltage level difference between the logic output high (V_{OH}) and the logic input high (V_{IH}) is called the noise margin for the logic, whereas the voltage level difference between the logic input low (V_{IL}) and the logic output low high (V_{OL}) is called the noise margin for the logic.

In 1970's and 1980's, V_{DD} was 5V. Nowadays V_{DD} has dropped so we can void frying tiny transistors and save the power in the computer system. When you connect chips with different supply voltages, you should be careful; otherwise, you may burn the chip!

2.3. Transistors and Logic Gates

We can build logic gates (AND, OR, XOR, etc.) from transistors. The transistor is a 3-ported voltage-controlled switch with g: gate, d: drain, s: source. Two ports, i.e. drain and source, connected depending on the gate voltage. If the gate voltage is LOW, the switch is OFF. If the gate voltage is HIGH, the switch is on.

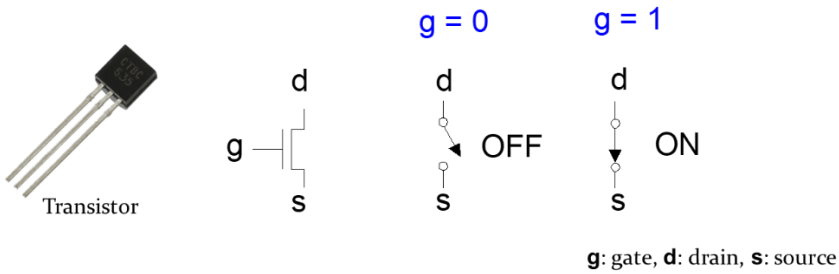


Fig. 2-10. Transistor with g = 0 and g = 1

The Metal oxide silicon (MOS) transistor has the polysilicon (used to be metal) on the gate. The oxide (silicon dioxide) insulator isolates the substrate (p-type silicon: a positively charges silicon) from the polysilicon.

If the gate voltage is low, the polysilicon gate has the negative voltage and the substrate has the positive feature. There is nothing happened, meaning the source and the drain are not connected. If the gate voltage is HIGH, the polysilicon gate has the positive voltage and the substrate also has the positive feature. The positives push each other. It attracts the negative feature (electron) on the surface of the silicon dioxide insulator, which creates a channel to connect the source and the drain, meaning the source and the drain are connected.

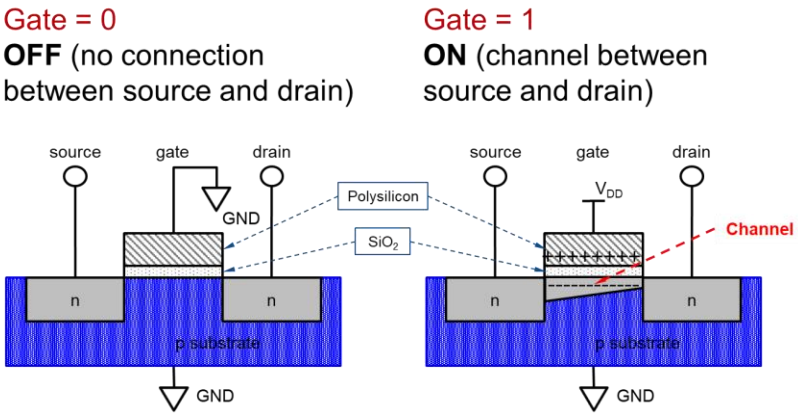


Fig. 2-11. Transistors: nMOS

The following figure shows the pMOS transistor. The pMOS is working in a opposite manner. If the gate voltage is LOW, the switch is on, meaning that the source and the drain are connected. If the gate voltage is HIGH, the switch is off, meaning that the source and the drain are disconnected.

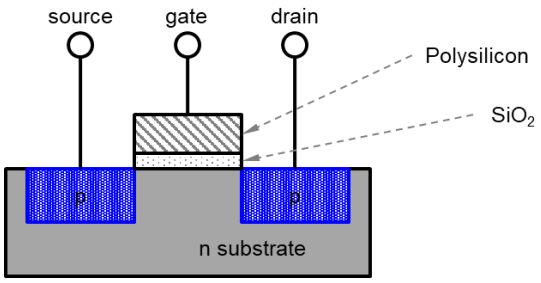


Fig. 2-12. Transistors: pMOS

In summary, the nMOS transistor has the following features:

- If the gate voltage is LOW, the switch is OFF, meaning that the source and the drain are disconnected.
- If the gate voltage is HIGH, the switch is ON, meaning that the source and the drain are connected.

The pMOS transistor has the following features:

- If the gate voltage is LOW, the switch is ON, meaning that the source and the drain are connected.
- If the gate voltage is HIGH, the switch is OFF, meaning that the source and the drain are disconnected.

The nMOS transistor is a good component to pass 0's, so the source port should be connected to GND. The pMOS transistor is a good component to pass 1's, so the source port should be connected to VDD. The drain ports of both nMOS and pMOS transistors can be connected to the output port.

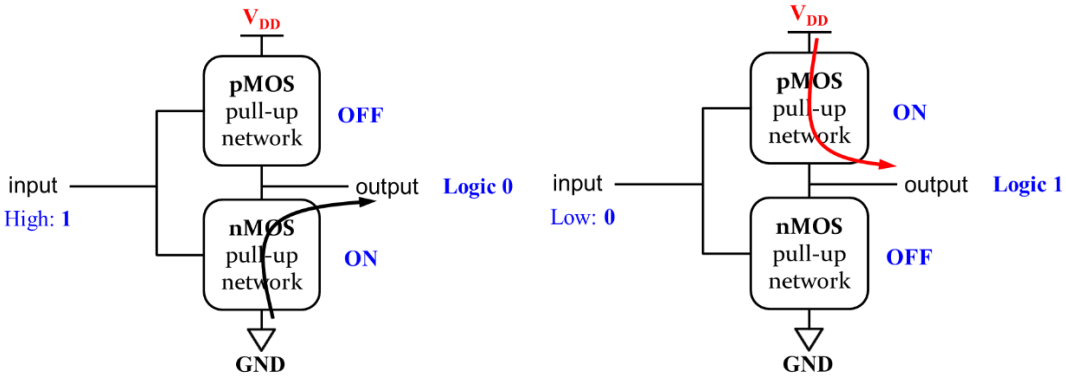


Fig. 2-13. Transistor function

If the gate voltages of both nMOS and pMOS transistors are HIGH (logic "1"), the pMOS transistor is OFF and the nMOS transistor is ON. The output port has the GND voltage (logic "0"). If the gate voltages of both nMOS and pMOS transistors are LOW (logic "0"), the pMOS transistor is ON and the nMOS transistor is OFF. The output port has the V_{DD} voltage (logic "1").

The NOT logic gate can be designed by connecting two transistors, nMOS and pMOS transistors, as follows:

- The logic input A connected to the gate ports of both transistors.
- The source port of the nMOS transistor connected to GND.
- The source port of the pMOS transistor connected to V_{DD}.
- The drain ports of both transistors connected to the output port Y.

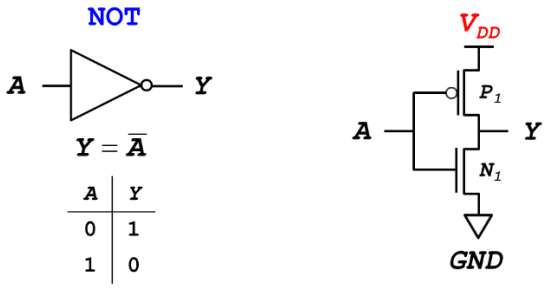


Fig. 2-14. CMOS Gates: NOT Gate

If the logic input A = 0, the gate voltage of nMOS transistor (N1) is LOW so that the nMOS transistor is OFF. On the other hand, the gate voltage of pMOS transistor (P1) is HIGH so that the pMOS transistor is ON. Since the pMOS transistor (P1) is ON, the logic output voltage has the V_{DD} voltage. That means the logic output Y = 1.

If the logic input A = 1, the gate voltage of nMOS transistor (N1) is HIGH so that the nMOS transistor is ON. On the other hand, the gate voltage of pMOS transistor (P1) is LOW so that the pMOS transistor is OFF. Since the nMOS transistor (N1) is ON, the logic output voltage has the GND voltage. That means the logic output Y = 0.

The NAND logic gate can be designed by connecting four transistors, where two nMOS transistors (N1, N2) are connected serially and two pMOS transistors (P1, P2) are connected parallelly, as follows:

- The logic input A connected to the gate ports of both pMOS transistor (P1) and nMOS transistor (N1).
- The logic input B connected to the gate ports of both pMOS transistor (P2) and nMOS transistor (N2).
- The source port of nMOS transistor (N1) connected to the drain port of nMOS transistor (N2).
- The source port of the nMOS transistor (N2) connected to the GND.
- The source ports of both pMOS transistors (P1, P2) connected to V_{DD}.
- The drain ports of both pMOS transistors (P1, P2) and the drain port of nMOS transistor (N1) connected to the output Y

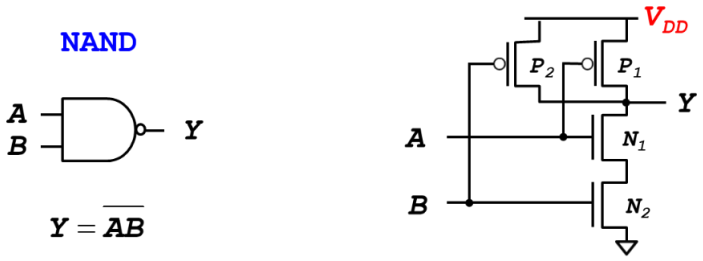


Fig. 2-15. CMOS Gates: NAND Gate

If the logic inputs A = 0 and B = 0, then

- Both nMOS transistors (N1, N2) are OFF. The output port Y has no access to the GND voltage.
- Both pMOS transistors (P1, P2) are ON. The output port Y has the voltage V_{DD} (logic 1).

If the logic inputs A = 0 and B = 1, then

- One nMOS transistor (N1) is OFF and the other nMOS transistor (N2) is ON. Two transistors connected serially, the output port Y has no access to the GND voltage.
- One pMOS transistor (P1) are ON and the other pMOS transistor (P2) is OFF. One of pMOS switches is on. The output port Y has the voltage V_{DD} (logic 1).

If the logic inputs A = 1 and B = 0, then

- One nMOS transistor (N1) is ON and the other nMOS transistor (N2) is OFF. Two transistors connected serially, the output port Y has no access to the GND voltage.
- One pMOS transistor (P1) are OFF and the other pMOS transistor (P2) is ON. One of pMOS switches is on. The output port Y has the voltage V_{DD} (logic 1).

If the logic inputs A = 1 and B = 1, then

- Both nMOS transistors (N1, N2) are ON. Since two transistors connected serially, the output port Y can access the GND voltage (logic 0).
- Both pMOS transistors (P1, P2) are OFF. The output port Y has no access to the voltage V_{DD}.

The following table summarizes the operation of all the transistors with respect to the two inputs:

A	B	P_1	P_2	N_1	N_2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

How do you build a two-input AND gate?

By connecting the output port of CMOS NAND logic gate to the input port of CMOS NOT logic gate, we can design AND gate, as shown in the following figure:

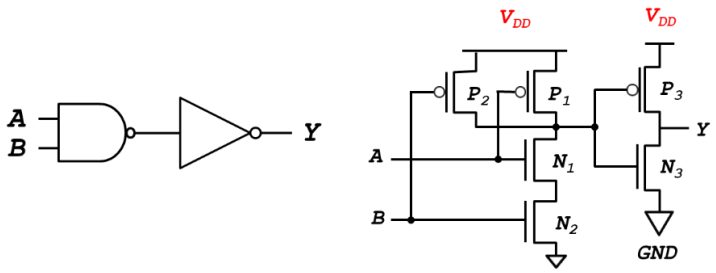


Fig. 2-16. CMOS Gates: AND Gate

The NOR logic gate can be designed by connecting four transistors, where two nMOS transistors (N1, N2) are connected parallelly and two pMOS transistors (P1, P2) are connected serially, as follows:

- The logic input A connected to the gate ports of both pMOS transistor (P1) and nMOS transistor (N1).
- The logic input B connected to the gate ports of both pMOS transistor (P2) and nMOS transistor (N2).
- The source ports of both nMOS transistors (N1, N2) connected to GND.
- The source port of the pMOS transistor (P1) connected to the VDD.
- The drain port of pMOS transistor (P1) connected to the source of pMOS transistor (P2).
- The drain port of pMOS transistor (P2) and the drain ports of both nMOS transistor (N1) and nMOS transistor (N2) connected to the output Y.

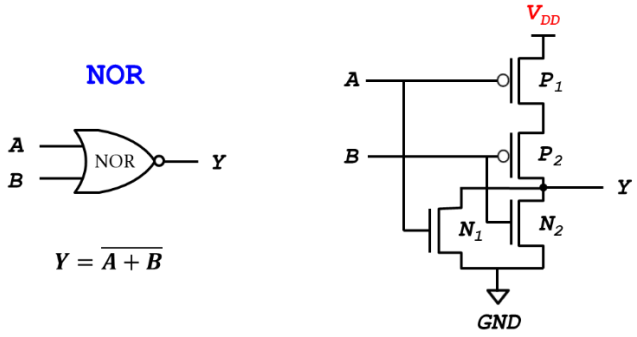


Fig. 2-17. CMOS Gates: NOR Gate

If the logic inputs A = 0 and B = 0, then

- Both nMOS transistors (N1, N2) are OFF. The output port Y has no access to the GND voltage.
- Both pMOS transistors (P1, P2) are ON. The output port Y has the voltage V_{DD} (logic 1).

If the logic inputs A = 0 and B = 1, then

- One pMOS transistor (P1) is ON and the other pMOS transistor (P2) is OFF. Two transistors connected serially, the output port Y has no access to the V_{DD} voltage.
- One nMOS transistor (N1) are OFF and the other nMOS transistor (N2) is ON. One of nMOS switches is on. The output port Y has the GND voltage (logic 0).

If the logic inputs A = 1 and B = 0, then

- One pMOS transistor (P1) is OFF and the other pMOS transistor (P2) is ON. Two transistors connected serially, the output port Y has no access to the V_{DD} voltage.
- One nMOS transistor (N1) are ON and the other nMOS transistor (N2) is OFF. One of nMOS switches is on. The output port Y has the GND voltage (logic 0).

If the logic inputs A = 1 and B = 1, then

- Both pMOS transistors (P1, P2) are OFF. The output port Y has no access to the V_{DD} voltage.
- Both nMOS transistors (N1, N2) are ON. The output port Y has the GND voltage (logic 0).

The following table summarizes the operation of all the transistors with respect to the two inputs:

A	B	P_1	P_2	N_1	N_2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	0
1	0	OFF	ON	ON	OFF	0
1	1	OFF	OFF	ON	ON	0

How do you build a two-input OR gate?

By connecting the output port of CMOS NOR logic gate to the input port of CMOS NOT logic gate, we can design OR gate, as shown in the following figure:

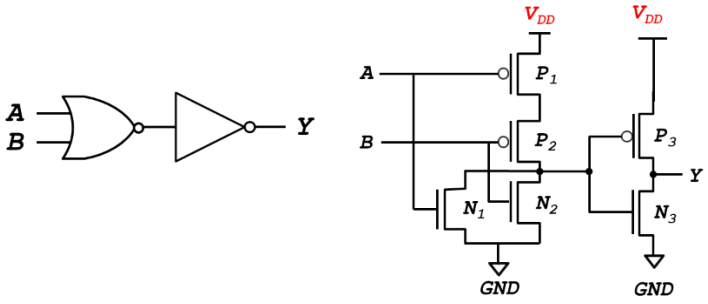


Fig. 2-18. CMOS Gates: OR Gate

Chapter 3: Boolean Expressions

In this chapter, we learn how to write a Boolean expression given a truth table and use Boolean algebra to simplify Boolean equations. De Morgan's Theorem is a particularly powerful tool in digital design, which explains that the complement of the product of all the term is equal to the sum of the complement of each term.

Objectives

By the end of this chapter you should be able to:

- Explain how to derive a Boolean equation from any truth table.
- Express a Boolean equation for any truth table by summing each of the minterms for the output.
- Understand how to use Boolean algebra to simplify equations.
- Demonstrate De Morgan's Theorem to simplify a Boolean equation.

3.1. Boolean Equations

A Boolean equation is a functional specification of outputs in terms of inputs, which are expressed as a logical statement that is either TRUE or FALSE. The following figure exemplifies a functional specification with two inputs A and B and the output Y.

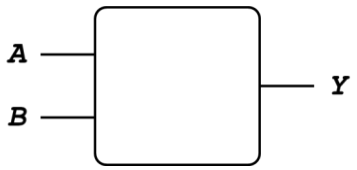


Fig. 3-1. Functional specification with two inputs A and B and the output Y

Let's assume that the functional specification of the above figure can be expressed as the following truth tables:

A	B	Y
0	0	0
0	1	1
1	0	0
1	1	1

$Y = B$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = A \cdot B$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	1

$Y = 1$

Fig. 3-2. Boolean Equations with Truth Tables

In the first truth table, the output Y is always equal to the input B regardless of the input A. We can simplify the Boolean equation with the truth table, such as $Y = B$.

In the second truth table, the output Y produces TRUE if only if both two inputs A and B are TRUE. Either input A or B is FALSE, the output Y is False. Here, the output Y values are equivalent to the AND operation with two inputs A and B. We can simplify the Boolean equation with the truth table, such as $Y = A \cdot B = AB$.

In the last truth table, the output Y is always TRUE regardless of two inputs A and B. We can simplify the Boolean equation with the truth table, such as $Y = 1$.

3.2. Boolean Algebra

In mathematics and mathematical logic, Boolean algebra is the branch of algebra in which the values of the variables are the truth values true and false, usually denoted 1 and 0, respectively. You can simplify the Boolean equations using a variety of axioms and theorems. Like the regular algebra, the Boolean algebra has numerical operations, but it is simpler than the regular algebra because valuables have only two values, i.e. 1 or 0. The main operations of Boolean algebra are the AND operation, the OR operation and the negative or NOT operation.

Boolean Axioms and Theorems

Table 3-1 summarizes the Boolean axioms with its duality, where ANDs and ORs, 0's and 1's are interchanged.

Table 3-1. Boolean Axioms with Its Duality

	Axiom	Dual	Name
(1)	$A = 0$ if $A \neq 1$	$A = 1$ if $A \neq 0$	Binary field
(2)	$\bar{0} = 1$	$\bar{1} = 0$	NOT
(3)	$0 \cdot 0 = 0$	$1 + 1 = 1$	AND/OR
(4)	$1 \cdot 1 = 1$	$0 + 0 = 0$	AND/OR
(5)	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND/OR

(1) Since the Boolean algebra has the binary field, the value A will be 0 if the value A is not 1. If the value A is not 0, the value A will be 1. (2) The negative or NOT operation can be denoted as a bar over the variable. The negative operation is equivalent to the complement or inverse of the variable.

	Axiom	Dual
(3)	0 ANDed with O is equal to 0.	1 ORed with 1 is equal to 1.
(4)	1 ANDed with 1 is equal to 1.	0 ORed with 0 is equal to 0.
(5)	0 ANDed with 1 is equal to 0.	0 ORed with 1 is equal to 1.

The identify theorem exists in the Boolean algebra. A variable $A \in \{1, 0\}$ ANDed with 1 is always equal to itself. This operation executes in AND gate, as follows:

- A = 0 ANDed with 1 equal to Y = 0,

- A = 1 ANDed with 1 equal to Y = 1.

In a similar manner, A variable $A \in \{1, 0\}$ ORed with 0 is always equal to itself. This operation executes in OR gate, as follows:

- A = 1 ORed with 0 equal to Y = 1,
- A = 0 ORed with 0 equal to Y = 0.

The Fig. 3-3 visualizes the identify theorem.

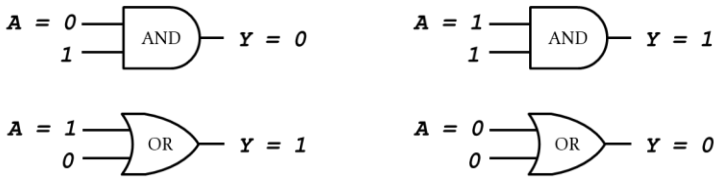


Fig. 3-3. Identify Theorem

The null element theorem exists in the Boolean algebra. A variable $A \in \{1, 0\}$ ANDed with 0 is always equal to 0. This operation executes in AND gate, as follows:

- A = 1 ANDed with 0 equal to Y = 0,
- A = 0 ANDed with 0 equal to Y = 0.

In a similar manner, A variable A ∈ {1, 0} ORed with 1 is always equal to 1. This operation executes in OR gate, as follows:

- A = 1 ORed with 1 equal to Y = 1,
- A = 0 ORed with 1 equal to Y = 1.

The Fig. 3-4 visualizes the null element theorem.

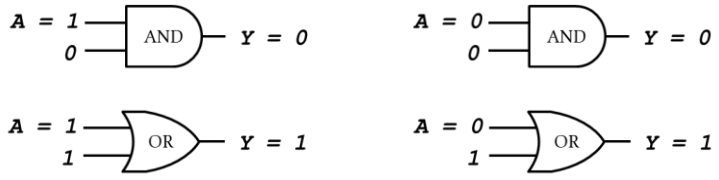


Fig. 3-4. Null Element Theorem

The idempotency theorem exists in the Boolean algebra. A variable $A \in \{1, 0\}$ ANDed with itself is always equal to the variable. This operation executes in AND gate, as follows:

- A = 1 ANDed with itself (1) equal to Y = 1

- A = 0 ANDed with itself (0) equal to Y = 0

In a similar manner, A variable $A \in \{1, 0\}$ ORed with itself is always equal to the variable. This operation executes in OR gate, as follows:

- A = 1 ORed with itself (1) equal to Y = 1,
- A = 0 ORed with itself (0) equal to Y = 0.

The Fig. 3-5 visualizes the idempotency theorem.

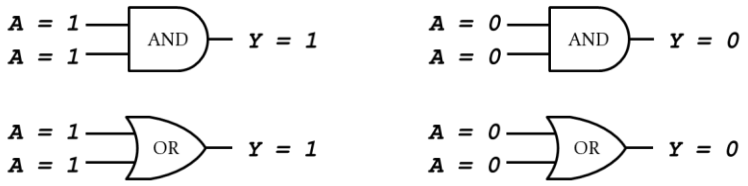


Fig. 3-5. Idempotency Theorem

The complement theorem exists in the Boolean algebra. A variable $A \in \{1, 0\}$ ANDed with its complement $\bar{A}$ is always equal to 0. This operation executes in AND gate, as follows:

- A = 1 ANDed with its complement ($\bar{A} = 0$) is always equal to 0,
- A = 0 ANDed with its complement ($\bar{A} = 1$) is always equal to 0.

In a similar manner, A variable $A \in \{1, 0\}$ ORed with its complement $\bar{A}$ is always equal to 1. This operation executes in OR gate, as follows:

- A = 1 ORed with its complement ($\bar{A} = 0$) is always equal to 1,
- A = 0 ORed with its complement ($\bar{A} = 1$) is always equal to 1.

The Fig. 3-6 visualizes the complement theorem.

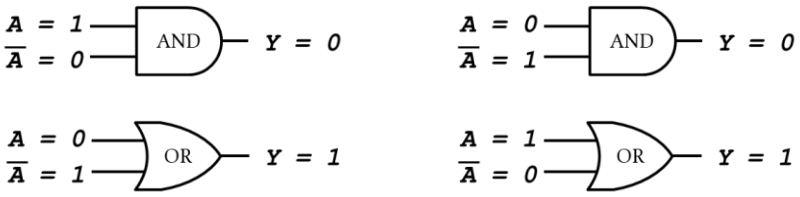


Fig. 3-6. Complement Theorem

The double complement law exists in the Boolean algebra. The double complement (negation) of a variable $A \in \{1, 0\}$ is always equal to the variable. This operation executes with by connecting two NOT gates serially, as follows:

- A = 0 double complement is always equal to 0,
- A = 1 double complement is always equal to 1.

The Fig. 3-7 visualizes the double complement law.

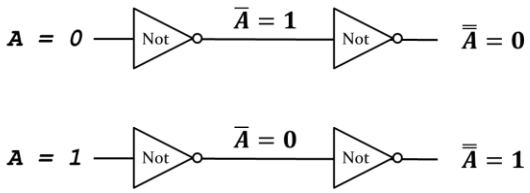


Fig. 3-7. Double Complement Law

The commutative law exists in the Boolean algebra.

- $A \cdot B = B \cdot A$; A ANDed with B is equal to B ANDed with A
- $A + B = B + A$; A ORed with B is equal to B ORed with A

The associative law exists in the Boolean algebra. When we execute AND or OR gates with more than 2 inputs, the order doesn't matter.

- $A(BC) = (AB)C$; A ANDed with BC is equal to C ANDed with AB.
- $A + (B + C) = (A + B) + C$; A ORed with (B+C) is equal to C ORed with (A+B).

The distributive law exists in the Boolean algebra.

- $A(B + C) = AB + AC$; A ANDed with (B+C) is equal to AB ORed with AC.

The absorption law exists in the Boolean algebra.

- $A + AB = A$

With the distributive law, the left-hand side of the equation can be expressed as follows:
$A + AB = A(1 + B)$, where the round bracket is further simplified as $(1 + B) = 1$ due to the identify theorem. A variable A ANDed with 1 is equal to A $(A \cdot 1 = A)$, which is identical to the right-hand side of the above equation.

- $A(A + B) = A$

With the distributive law, the left-hand side of the equation can be expressed as follows:
$A(A + B) = AA + AB$, where AA is equal to A $(AA = A)$ due to the idempotency theorem. Now the above equation is expressed as $A + AB = A$. You can also simplify the equation by drawing the truth table, as follows:

A	B	$A+B$	$A(A+B)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Exercises

Simplifying the following Boolean Equations:

(1) $Y = AB + \bar{A}B$

Answer:

$Y = AB + \bar{A}B = B(A + \bar{A}) = B\,(1) = B$

(2) $Y = A(AB + ABC)$

Answer:

$Y = A(AB + ABC)$

$= A\big(AB(1 + C)\big)$ // distributive law
$= A\big(AB(1)\big)$ // identify theorem
$= A(AB)$
$= (AA)B$
$= AB$

(3) $Y = AB + A(B + C) + B(B + C)$

Answer:

$Y = AB + A(B + C) + B(B + C)$
$= AB + AB + AC + BB + BC$ // distributive law
$= AB + AB + AC + B + BC$ // idempotency theorem
$= AB + AC + B + BC$ // $B + BC = B$
$= AB + AC + B$ // $AB + B = B$
$= B + AC$

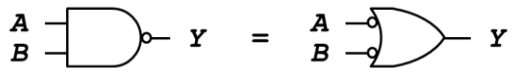
3.3. De Morgan's Theorems

De Morgan's Theorems are a pair of transformation rules that are both valid rules of inference. The rules can be expressed as:

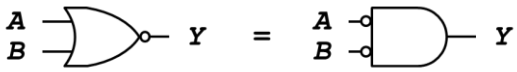
- The complement of the intersection of two sets is the same as the union of their complements; and
- the complement of the union of two sets is the same as the intersection of their complements,

where the intersection and union operations are expressed as AND and OR gates respectively in the digital logic systems. In the Boolean algebra, these are written formally as follows:

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



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



Bubble pushing is a technique to apply De Morgan's theorem directly to the logic diagram. There are two steps to use the bubble pushing, as follows:

- Change the logic gate (AND to OR and OR to AND).
- Add bubbles to the inputs and outputs where there were none, and remove the original bubbles.

For example, the backward bubble pushing is applied to NAND gate, pushing the bubble in the output side back to input side. After changing AND gate to OR gate, add bubbles to two inputs A and B, as shown in the following figures:

Fig. 3-8. Backward Bubble Pushing

The forward bubble pushing is applied to the logic gate which has the bubbles at all the inputs, by pushing the bubbles at the input side A & B to the output side Y. After changing OR gate to AND gate, add a bubble to the output Y, as shown in the following figures:

Fig. 3-9. Forward Bubble Pushing

There are rules for the bubble pushing, as listed:

- Begin at output, then work toward inputs
- Push bubbles on final output back
- Draw gates in a form so bubbles cancel

There are four inputs, A, B, C, and D in Fig. 3-10. Two inputs A & B are fed into NOR gate. It's output and the input C are fed into a NAND gate. The output of NAND gate and the input D are fed into another NAND gate, and those two inputs produce the output Y.

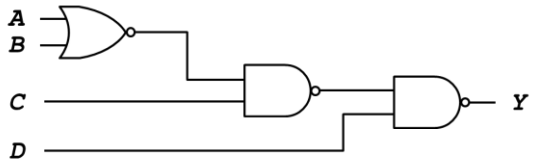


Fig. 3-10. Bubble Pushing Example

The last NAND gate can be changed with the backward bubble pushing, as shown in Fig. 3-11.

- Body changes from AND to OR gate.
- Adds bubbles to inputs. No bubble at the output.

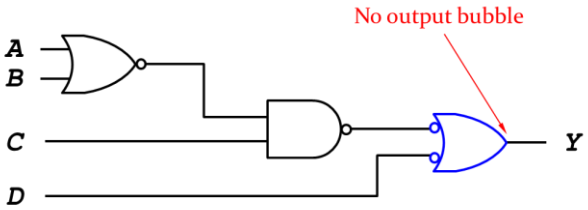


Fig. 3-11. Bubble Pushing Example - No Output Bubble-1

Two bubbles, i.e. the output bubble of NAND gate and the input bubble produced with backward, are now put in the same line. These two bubbles canceled each other, because the double complement of a variable is always equal to the variable.

NOR gate with two inputs A & B can be changed with De Morgan's Theorems, as shown in Fig. 3-12:

- Body changes from OR to AND gate.
- Adds bubbles to inputs. No bubble at the output.

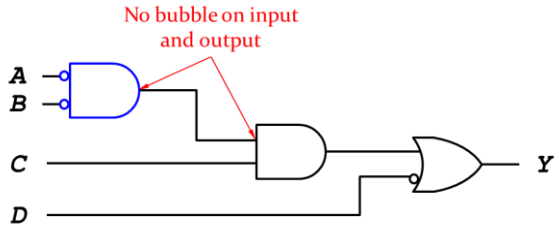


Fig. 3-12. Bubble Pushing Example - No Output Bubble-2

From the above figure, we can draw the following Boolean equation: $Y = \bar{A} \cdot \bar{B} \cdot C + \overline{D}$. Note that we will get the Boolean equation $Y = \overline{\overline{\overline{A + B} \cdot \overline{C} \cdot D}}$ without De Morgan's theorems.

Exercise

1) There are five inputs, A, B, C, D, and E with four NAND gates, as shown in Fig. 3-13:

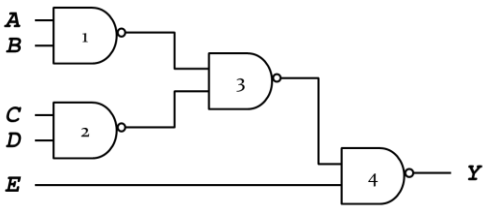


Fig. 3-13. Quiz 1 Figure

- NAND_1 gate has two inputs A and B. NAND_2 gate has two inputs C and D.
- The two outputs of NAND gates fed into NAND_3 gate.
- The out of the NAND_3 gate and input E fed into the NAND_4 gate, produce the output Y.

Simply the logic gates with De Morgan's theorems and write the corresponding Boolean equation.

2) There are five inputs, A, B, C, D, and E with three NAND gates and one NOR gate, as shown in Fig. 3-14:

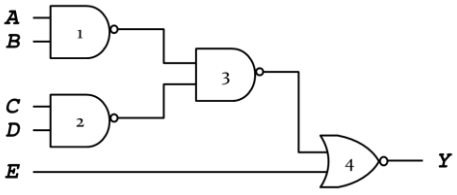


Fig. 3-14. Quiz 2 Figure

- NAND_1 gate has two inputs A and B. NAND_2 gate has two inputs C and D.
- The two outputs of NAND gates fed into NAND_3 gate.
- The out of the NAND_3 gate and input E fed into the NOR_4 gate, produce the output Y

Simply the logic gates with De Morgan's theorems and write the corresponding Boolean equation.

3) There are four inputs, A, B, C, and D with three NAND gates, as shown in Fig. 3-15:

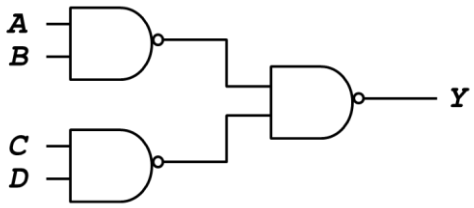


Fig. 3-15. Quiz 3 Figure

- One NAND gate has two inputs A and B. Another NAND gate has two inputs C and D.
- The two outputs of NAND gates fed into the other NAND gate, and produce output Y.

Simply the logic gates with De Morgan's theorems and write the corresponding Boolean equation.

4) Simplify the following Boolean expression to a minimum number of literals:

$$Y = \bar{A}\bar{B} + \bar{A}B\bar{C} + \overline{(A + \bar{C})}$$

Answer)

$$Y = \bar{A}\bar{B} + \bar{A}B\bar{C} + \overline{(A + \bar{C})}$$

$$= \bar{A}\bar{B} + \bar{A}B\bar{C} + \bar{A}C \quad \text{(De Morgan)}$$

$$= \bar{A}(\bar{B} + B\bar{C} + C)$$

$$= \bar{A}(\bar{B} + B\bar{C} + C(\bar{B} + B))$$

$$= \bar{A}(\bar{B} + B\bar{C} + \bar{B}C + BC)$$

$$= \bar{A}(\bar{B}(1 + C) + B(\bar{C} + C))$$

$$= \bar{A}(\bar{B} + B) = \bar{A}$$

Chapter 4: Boolean Expressions and Combinational Circuits

In this chapter, we learn how to express Boolean equation with combinational circuits, circuit schematics rules for combinational circuits, and one of the multiple-output circuits - priority circuit. By understanding the meanings of Contention X (don't care) and floating Z, we can apply the rules of Karnaugh maps for simplifying Boolean equations.

Objectives

By the end of this chapter you should be able to:

- Express Boolean equation with combinational circuits.
- Recognize circuit schematics rules for combinational circuits.
- Recall multiple-output circuits - priority circuit.
- Demonstrate the meanings of Contention X (don't care) and floating Z.
- Summarize the rules of Karnaugh maps.
- Apply Karnaugh maps for simplifying Boolean equations.

4.1 Circuit Schematics Rules

Digital systems are constructed by using logic gates which are abstract representations of real devices. We can represent the Boolean algebra with two-level logic, ANDs followed by ORs. For example, we have a Boolean equation: $Y = \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C}$. This equation can be designed with logic gates, as shown in the following figure:

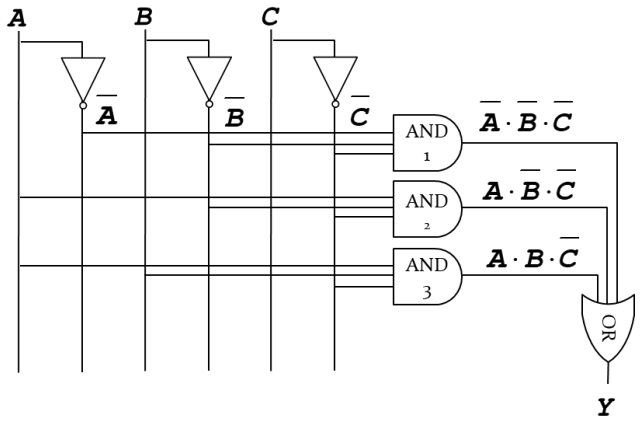


Fig. 4-1. From Logic to Gates

In Fig. 4-1, There are three AND gates. These AND gates have the following inputs,

- AND_1: A complement, B complement, C complement
- AND_2: A, B complement, C complement

- AND_3: A, B, C complement

The outputs of the three AND gates feed into OR gate. The OR gate produces the output Y.

There are some rules for circuit schematics. Wires always connect at a T junction, as shown in Fig. 4-2. A dot where wires cross indicates a connection between the wires. Wires crossing without a dot make no connection.

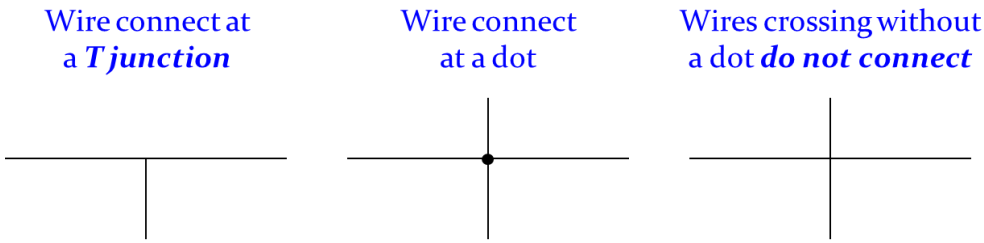


Fig. 4-2. Circuit Schematics Rules

The following figures show that there are T junctions between the wires. That means the wires are connected.

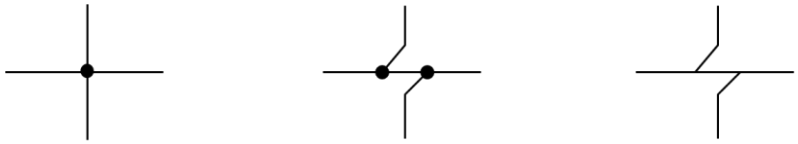


Fig. 4-3. Some examples of Junction

The following figures show that there is no T junction between the wires. That means the wires are not connected.

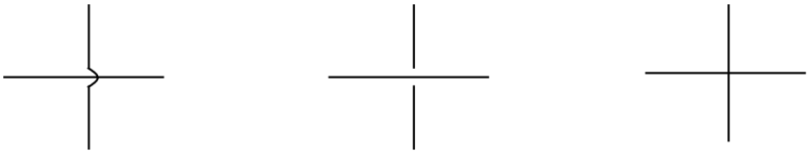


Fig. 4-4. Some examples of No Junction

The following figure, Fig. 4-5, designed a Boolean equation. There are five AND gates. These AND gates have the following inputs,

- AND_1: A complement, D
- AND_2: B, D
- AND_3: A, C complement, D
- AND_4: A, B complement, C
- AND_5: A, B, C, D

The outputs of four AND gates (AND_1, AND_3, AND_4, AND_5) feed into OR gate. The OR gate produces the output Y. We can express the corresponding Boolean equation as follows: $Y = \bar{A} \cdot D + A \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot D + A \cdot B \cdot C \cdot D$.

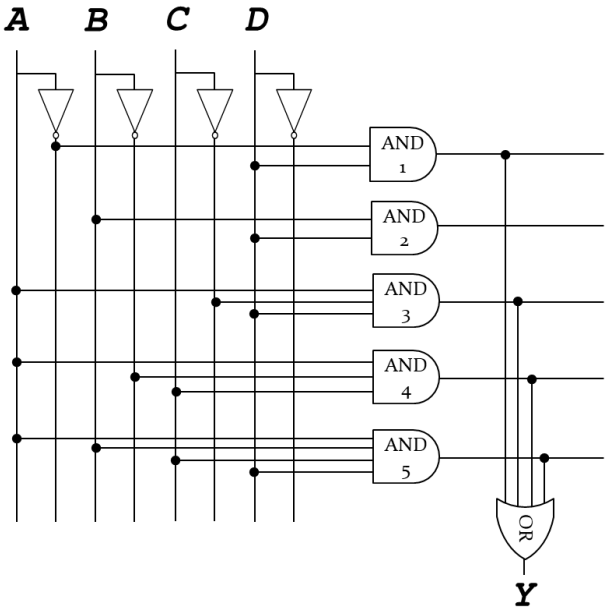


Fig. 4-5. Some examples of Boolean equation

Multiple-Output Circuits

Circuits that we have previously discussed have only one output. Here we will discuss how multiple output systems are analyzed. A priority encoder is a circuit or algorithm that compresses multiple binary inputs into a smaller number of outputs. The priority circuit produces an output asserted corresponding to the most significant TRUE input. The following figure shows the truth table and the hardware design of the priority circuit.

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

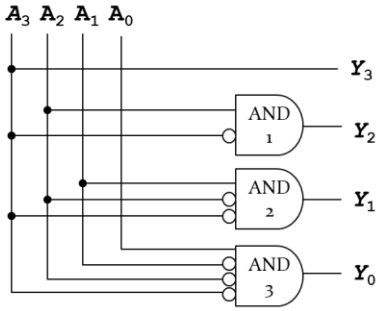


Fig. 4-6. Truth Table and Hardware Design of Priority Circuit

The priority circuit has:

- Input A3 directly connected to Y3
- AND_1 gate produces the output Y2 with two inputs, A3 complement and A2
- AND_2 gate produces the output Y1 with three inputs, i.e. A3 complement, A2 complement, and A1
- AND_3 gate produces the output Y0 with four inputs, i.e. A3 complement, A2 complement, A1 complement and A0

where the bubble symbol represents a complement. The above truth table can be simplified as the following table.

Table 4-1. Truth Table of Priority Circuit with Don't Cares (X)

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

Let's look at the case when the output $\mathbf{Y_1}$ is TRUE. A_3 and A_2 didn't assert, meaning that both A_3 and A_2 are FALSE. In this case, A_1 has a priority among the input values. Once A_1 asserts TRUE, the input A_0 doesn't matter whether it is TRUE or FALSE. Only $\mathbf{Y_1}$ is TRUE and the other outputs are all FALSE.

Let's look at the case when the output $\mathbf{Y_2}$ is TRUE. A_3 didn't assert, meaning that A_3 is FALSE. In this case, A_2 has a priority among the input values. Once A_2 asserts TRUE, the two inputs A_0 and A_1 don't matter whether they are TRUE or FALSE. Only $\mathbf{Y_2}$ is TRUE and the other outputs are all FALSE.

In the last case where the output Y3 is TRUE, A3 has a priority among the input values. Once A_3 asserts TRUE, the other inputs A_0, A_1 and A_2 don't matter whether they are TRUE or FALSE. Only $\mathbf{Y_3}$ is TRUE and the other outputs are all FALSE. Here, a don't-care term (**X**) for a function is an input-sequence (a series of bits) for which the function output does not matter.

With the contention **X**, circuit tries to drive an output to 1 and 0. The actual value of the contention could be 0, 1, or in forbidden zone. That might change with voltage, temperature, time and noise. The contention often causes excessive power dissipation. However, the contention usually indicates a bug. The symbol, **X**, is used for "don't care" and contention. With this don't care, we can find out some way to simplify the Boolean equation when we design the digital circuits.

The floating Z, high impedance, is driven neither HIGH nor LOW. The floating Z might be 0, 1 or somewhere in between. The tristate buffer is one of example of this floating Z which has three possible output states: HIGH (1), LOW (0) and floating (Z).

The output Y determined by both the input A and the enable E.

- If enable E = 0, then the tristate buffer outputs floating Z regardless of the value of input A.
- If enable E = 1, then the tristate buffer outputs the same as the value of input A.

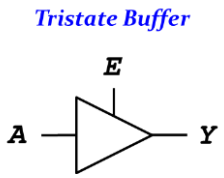


Fig. 4-7. Tristate Buffer

The floating nodes are used in tristate busses, where many different drivers use the shared bus with processor, video, Ethernet, memory, etc.

4.2 Karnaugh Maps (K-Maps)

The Karnaugh map (K-map) is a graphical method of simplifying Boolean algebra expressions, where the Boolean expressions can be minimized by combining terms.

Let's look at a truth table, where the output **Y** is TRUE if only if the two inputs are **A** = 0 and **B** = 1, or **A** = 1 and **B** = 1.

A	B	Y
0	0	0
0	1	**1**
1	0	0
1	1	**1**

The above table can be expressed in the K-map as the following figure, where each value of the squares is corresponding to the value of Y:

- If A = 0 and B = 0, Y = 0 (*i.e.*, $Y = \bar{A}\bar{B}$).
- If A = 0 and B = 1, Y = 1 (*i.e.*, $Y = \bar{A}B$).
- If A = 1 and B = 1, Y = 0 (*i.e.*, $Y = A\bar{B}$).
- If A = 1 and B = 1, Y = 0 (*i.e.*, $Y = AB$).

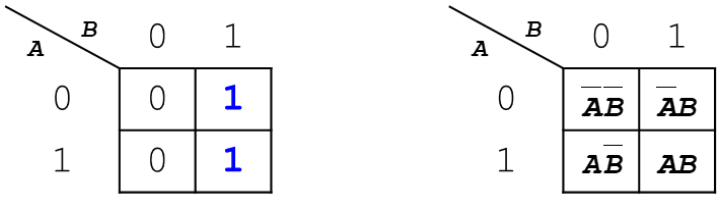


Fig. 4-8. K-Map Representation

In order to simplify the Boolean expression with K-map, we can circle 1's in adjacent squares, where the most left column and the first row represent the input variables, inputs A and B.

In the following figure, the circle is located in the most right column, whose literal corresponds to the input B. Let's look closely at the circle to draw the corresponding literal. The circle takes two rows (A, $\bar{A}$), i.e. one for the input A and another for the input $\bar{A}$. These two literals, A and $\bar{A}$, are cancelled each. We can only write the literal B, $Y = B$ that simplify the Boolean equation.

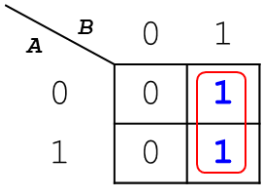


Fig. 4-9. K-Map Representation with a Circle

The following figure shows that the circles must span a power of 2.

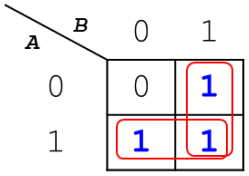


Fig. 4-10. K-Map Representation with a Circle

Fig. 4-10 has two red circles, one located at the most right column and another located at the last row. The first circle takes a single column (B) and two rows (A, $\bar{A}$), where A and $\bar{A}$ are cancelled each other. We can draw only the literal B from this circle. The latter circle takes a single row (A) and two columns (B, $\bar{B}$), where B and $\bar{B}$ are cancelled each other. We can draw only the literal A from the circle. These two literals are ORed, so we can draw the corresponding Boolean equation: $Y = A + B$ from Fig. 4-10.

The 3-input K-Map can be drawn in the following figure:

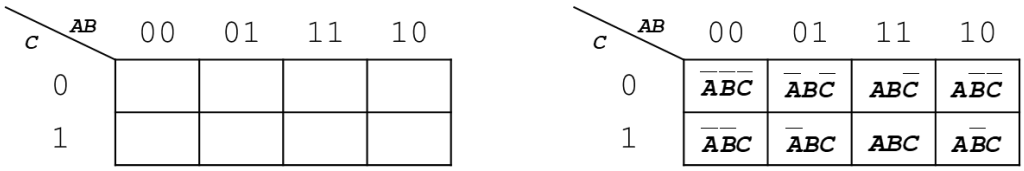


Fig. 4-11. 3-Input K-Map Representation

In this K-map, the first row and the most left column represent the input variables.

- First row: inputs A and B
- Most-left column: input C

In the first row, we can identify four different combinations of two inputs; AB=00, AB=01, AB=11, and AB=10, where only one-bit change in value from one adjacent column to the next column is allowed.

Let's see how to use this 3-input K-Map to simplify the Boolean equation. The following truth table can be mapped to K-map, as shown in Fig. 4-12.

A	B	C	Y
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	0
1	1	1	**1**

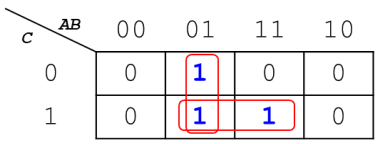


Fig. 4-12. 3-Input K-Map Representation with Circles

In the K-map of Fig. 4-12, we can identify only 3 squares filled with '1' bit; mapping to $\bar{A}B\bar{C}$, $\bar{A}BC$, and ABC. We can circle 1's in adjacent squares, and have two red circles; one located at the second column of the squares and another located at the last row of the squares. The first circle takes a single column ($\bar{A}B$) and two rows (C, $\bar{C}$), where C and $\bar{C}$ are cancelled each other. We can draw the literal $\bar{A}B$ from the circle. The latter circle takes a single row (C) and two columns ($\bar{A}B$, AB), where A and $\bar{A}$ are cancelled each other, leaving only the literal B. We can draw the literals BC from the circle. These two implicants (product of literals) are ORed, so we can draw the corresponding Boolean equation: $Y = \bar{A}B + BC$ from Fig. 4-12.

When we draw a circle in the K-map, we need to draw a circle as big as possible so that the corresponding implicant can be minimized. We called a **prime implicant** whose implicant corresponding to the largest circle in a K-map.

There are some rules when we draw a circle in a K-map:

- **Every 1** must be circled at least once
- Each circle must span a **power of 2** (*i.e.* 1, 2, 4) squares in each direction
- Each circle must be **as large as possible**
- A circle may **wrap around the edges**
- A "don't care" (X) is circled **only if it helps minimize** the equation

Let's look at another example of 3-input K-map in Fig. 4-13.

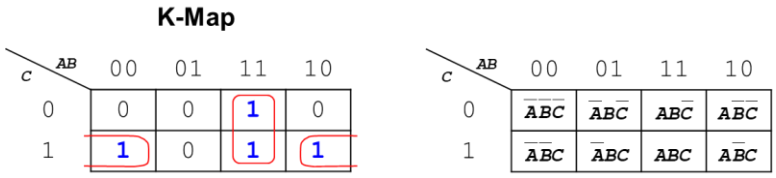


Fig. 4-13. 3-Input K-Map Representation with wrap around edges

In Fig. 4-13, we have the two circles, one for the column AB and another for the bottom edges of the squares. The first circle takes a single column (AB) and two rows (C, $\bar{C}$), where C and $\bar{C}$ are cancelled each other. We can draw the literals AB from the circle. The latter circle takes a single row (C) and two columns ($\bar{A}\bar{B}$, $A\bar{B}$), where A and $\bar{A}$ are cancelled each other, leaving only the literal $\bar{B}$. We can draw the literals $\bar{B}C$ from the circle. These two implicants (product of literals) are ORed, so we can draw the corresponding Boolean equation: $Y = AB + \bar{B}C$ from Fig. 4-13.

The following table shows a 4-input truth table. From the truth table, we can fill out K-map. In the K-map, the first row and the most left column represent the input variables.

- First row: inputs A and B have four different combinations AB=00, AB=01, AB=11, and AB=10, where only one-bit change in value from one adjacent column to the next column is allowed.
- Most-left column: inputs C and D have four different combinations CD=00, CD=01, CD=11, and CD=10, where only one-bit change in value from one adjacent column to the next column is allowed.

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

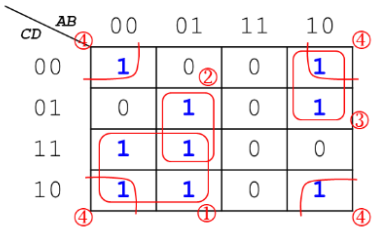
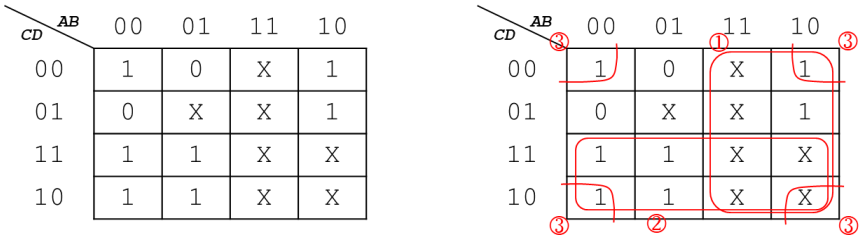


Fig. 4-14. 4-Input K-Map Representation

Let's see how to use this 4-input K-Map to simplify the Boolean equation. We can draw four circles in the above figure using the rules. The first circle① takes two columns ($\bar{A}\bar{B}$, $\bar{A}B$) and two rows (CD, $C\bar{D}$), where B and $\bar{B}$ are cancelled each other, and D and $\bar{D}$ are cancelled in the same manner. We can draw the literals $\bar{A}C$ from the circle. The second circle② takes a single column ($\bar{A}B$) and two rows ($\bar{C}D$, CD), where C and $\bar{C}$ are cancelled each other. We can draw the literals $\bar{A}BD$ from the circle. The third circle③ takes a single column ($A\bar{B}$) and two rows ($\bar{C}\bar{D}$, $\bar{C}D$), where D and $\bar{D}$ are cancelled each other. We can draw the literals $A\bar{B}\bar{C}$ from the circle. The last circle④ is located around the corner of edges and takes two columns ($\bar{A}\bar{B}$, $A\bar{B}$) and two rows ($\bar{C}\bar{D}$, $C\bar{D}$), where A and $\bar{A}$ are cancelled each other, and C and $\bar{C}$ are cancelled in the same manner. We can draw the literals $\bar{B}\bar{D}$ from the circle. These four implicants (product of literals) are ORed, so we can draw the corresponding Boolean equation: $Y = \bar{A}C + \bar{A}BD + A\bar{B}\bar{C} + \bar{B}\bar{D}$ from Fig. 4-14.

Fig. 4-15 shows an example of 4-input K-map with "don't cares (**X**)". We need to circle every '1' bit at least once. We can also circle "don't cares (**X**)" if they help minimize the equation by making the circle as large as possible. If the don't care (**X**) doesn't help to maximize the circle, it can be considered as a '0' bit. The first circle① takes two columns (AB, $A\bar{B}$) and four rows ($\bar{C}\bar{D}$, $\bar{C}D$, CD, $C\bar{D}$), where all the literals are cancelled each other (B and $\bar{B}$, C and $\bar{C}$, and D and $\bar{D}$) except the literal A. We can draw the literal A from the circle. The second circle② takes four columns ($\bar{A}\bar{B}$, $\bar{A}B$, AB, $A\bar{B}$), two rows (CD, $C\bar{D}$), where all the literals are cancelled each other (A and $\bar{A}$, B and $\bar{B}$, and D and $\bar{D}$) except the literal C. We can draw the literal C from the second. The last circle③ is located around the corner of edges and takes two columns ($\bar{A}\bar{B}$, $A\bar{B}$) and two rows ($\bar{C}\bar{D}$, $C\bar{D}$), where A and $\bar{A}$ are cancelled each other, and C and $\bar{C}$ are cancelled in the same manner. We can draw the literals $\bar{B}\bar{D}$ from the circle. These three implicants

(product of literals) are ORed, so we can draw the corresponding Boolean equation: $Y = A + C + \bar{B}\bar{D}$ from Fig. 4-15.



CD \ AB	00	01	11	10
00	1	0	X	1
01	0	X	X	1
11	1	1	X	X
10	1	1	X	X

Fig. 4-15. 4-Input K-Map Representation with Don't Cares

Exercise

Simply the Boolean equation with K-map:

CD \ AB	00	01	11	10
00	1	0	X	1
01	0	0	X	0
11	0	0	X	X
10	1	1	X	X

Answer: You can get the following equation: $Y = C\bar{D} + \bar{B}\bar{D}$

4.3 Combinational Circuits

Combinational Circuits are circuits made up of different types of logic gates and produce outputs by combining the values of the inputs at any given time. The circuits do not make use of any memory or storage device.

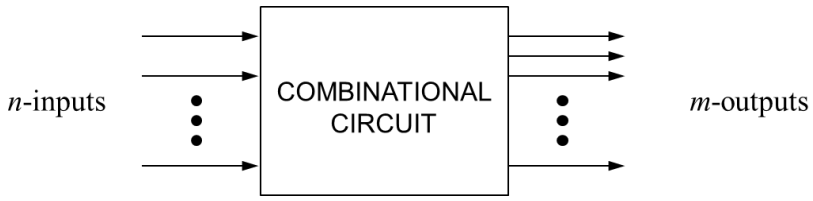


Fig. 4-16. Combinational Circuit Description

For *n* input variables, there are 2^n possible binary input combinations, and for each binary combination of the input variables, there is one possible output.

The combinational circuit is like a black box but it can be described with the truth table, which gives one possible output for each binary combination of the input variables. We will take a look at some popular combinational circuits throughout this section.

1-Bit Half Adders

A 1-bit half adder is used for adding together the two least significant digits in a binary sum. It has two inputs A and B, and two outputs, the sum S and the carryout C_{out}. The following table describe the 1-bit half adder. The sum is the output of exclusive OR gate, which has $S = \bar{A}B + A\bar{B} = A \oplus B$. The C_{out} (carryout) is the output of AND gate, which has two inputs, A and B, i.e. $S = A \cdot B$.

Table 4-2. Truth Table of 1-bit Half Adder

A	B	C_{out}	S
0	0	0	**0**
0	1	0	**1**
1	0	0	**1**
1	1	1	**0**

For the above truth table, we can fill the box of Fig. 4-16 with a combinational circuit of 1-bit half adder in the following figure:

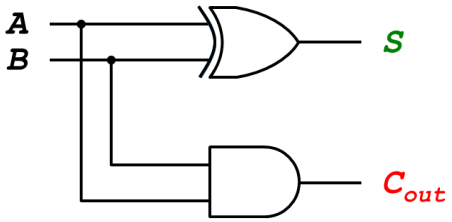


Fig. 4-17. Combinational Circuit of 1-bit Half Adder

Multiplexer

A multiplexer (or Mux), also known as a data selector, is a device that selects one of N analog or digital inputs and forwards the selected input to a single output line. If the mux has the two inputs, it needs a $\log_2 2$ control input. If the mux has N inputs, it needs $\log_2 N$ control inputs. The following figure shows a 2-to-1 multiplexer which has two inputs (D_0 and D_1), one output (Y), and a control input (S).

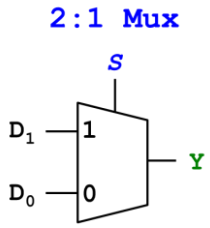


Fig. 4-18. 2-to-1 Multiplexer

If the control input S is 0, the input D_0 is forwarded to the output Y. If the control input S is 1, the input D_1 is forwarded to the output Y. The following table describes the 2-to-1 multiplexer.

Table 4-3. Truth Table of 2-to-1 Multiplexer

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

We can simply draw the truth table as follows:

S	Y
0	D_0
1	D_1

The multiplexer can have more than two inputs. The 4-to-1 multiplexer has four inputs (D_0, D_1, D_2, and D_3), one output (Y), and two select inputs (S_0 and S_1), as shown in the following figure:

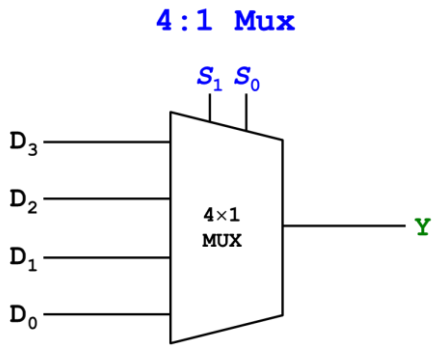


Fig. 4-19. 4-to-1 Multiplexer

The multiplexer operates as follows:

- If the select inputs S_1 S_0 is 00, the input D_0 is forwarded to the output Y. In this case, the other inputs D_3, D_2, and D_1 don't matter. Only the input D_0 determines the output Y.
- If the select inputs S_1 S_0 is 01, the input D_1 is forwarded to the output Y. In this case, the other inputs D_3, D_2, and D_0 don't matter. Only the input D_1 determines the output Y.
- If the select inputs S_1 S_0 is 10, the input D_2 is forwarded to the output Y. In this case, the other inputs D_3, D_1, and D_0 don't matter. Only the input D_2 determines the output Y.
- If the select inputs S_1 S_0 is 11, the input D_3 is forwarded to the output Y. In this case, the other inputs D_2, D_1, and D_0 don't matter. Only the input D_3 determines the output Y.

Accordingly, we can draw the truth table of 4-to-1 multiplexer as follows:

Table 4-4. Truth Table of 4-to-1 Multiplexer

S_1	S_0	D_3	D_2	D_1	D_0	Y
0	0	X	X	X	0	**0**
0	0	X	X	X	1	**1**
0	1	X	X	0	X	**0**
0	1	X	X	1	X	**1**

1	0	X	0	X	X	0
1	0	X	1	X	X	1
1	1	0	X	X	X	0
1	1	1	X	X	X	1

where 'X' represents the don't care term.

Fig. 4-20 shows how the 4-to-1 multiplexer operates when the two select inputs S_1S_0 = 00. The input sequence D_0 = 11111111 is forwarded to the output Y. The other inputs don't affect the sequence of the output Y.

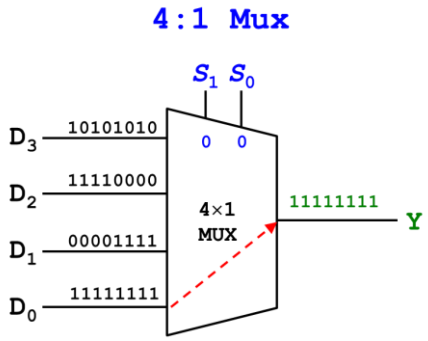


Fig. 4-20. 4-to-1 Multiplexer with S_1S_0 = 00

Fig. 4-21 shows how the 4-to-1 multiplexer operates when the two select inputs S_1S_0 = 01. The input sequence D_1 = 00001111 is forwarded to the output Y. The other inputs don't affect the sequence of the output Y.

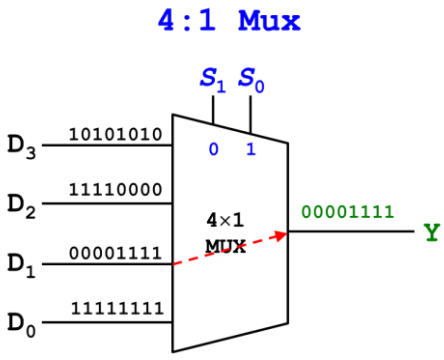


Fig. 4-21. 4-to-1 Multiplexer with S_1S_0 = 01

Fig. 4-22 shows how the 4-to-1 multiplexer operates when the two select inputs $S_1S_0 = 10$. The input sequence $D_2 = 11110000$ is forwarded to the output Y. The other inputs don't affect the sequence of the output Y.

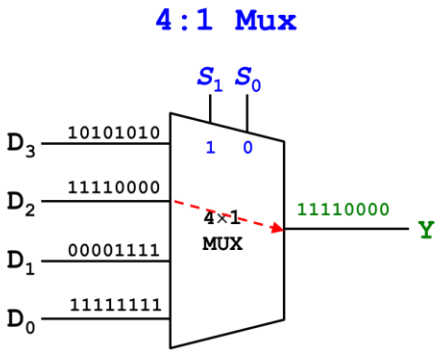


Fig. 4-22. 4-to-1 Multiplexer with $S_1S_0 = 10$

Fig. 4-23 shows how the 4-to-1 multiplexer operates when the two select inputs $S_1S_0 = 11$. The input sequence $D_3 = 10101010$ is forwarded to the output Y. The other inputs don't affect the sequence of the output Y.

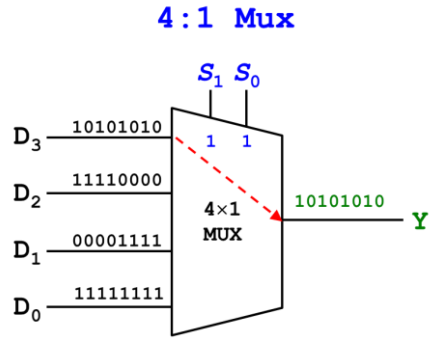


Fig. 4-23. 4-to-1 Multiplexer with $S_1S_0 = 11$

The following figure shows the 8-to-1 multiplexer which has eight inputs, D_0 through D_7. Since it has 8 inputs, $\log_2 8 = 3$ select bits (select inputs) required.

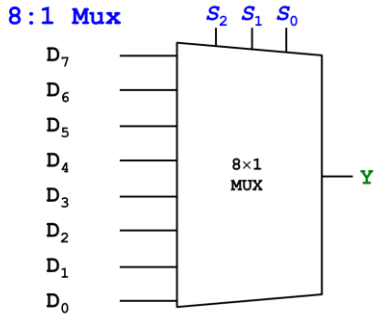


Fig. 4-24. 8-to-1 Multiplexer

The output Y is determined by the three select inputs, i.e. S_2, S_1, and S_0. The following table describes the operation of the 8-to-1 multiplexer.

Table 4-5. Truth Table of 8-to-1 Multiplexer

S_2	S_1	S_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

Encoder

In general, encoders convert motion to an electrical signal that can be read by some type of control devices. One of very popular encoders you know is a keyboard. When you press a button of the keyboard, the keyboard coverts this motion to an 8-bit digital signal. Your computer can read the value you pressed. The encoder is an inverse operation of a decoder. If you have 2^N inputs, the encoder produces a total of N outputs so that it generates the binary code corresponding to the input value.

The following figure shows a 4-to-2 encoder, where there are four inputs, i.e. D_3, D_2, D_1, D_0, and two outputs (binary code), i.e. B_1 and B_0. In the encoder, only one input is high or "1" and the other inputs are low or "0".

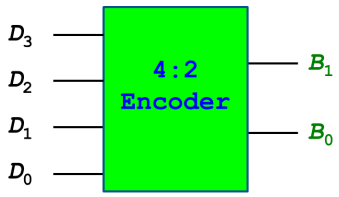


Fig. 4-25. 4-to-2 Encoder

The encoder generates the binary code corresponding to the input value. For example, if the input D_0 is high and the other inputs are low, the encoder generates the binary code B_1 B_0 = 00. If the input D_3 is high and the other inputs are low, the encoder generates the binary code B_1 B_0 = 11. The following table describe the operation of the 4-to-2 encoder.

Table 4-6. Truth Table of 4-to-2 Encoder

Inputs				Outputs	
D_3	D_2	D_1	D_0	B_1	B_0
0	0	0	1	**0**	**0**
0	0	1	0	**0**	**1**
0	1	0	0	**1**	**0**
1	0	0	0	**1**	**1**

The following figure shows a 8-to-3 encoder, where there are eight inputs, i.e. D_7 through D_0, and three outputs (binary code), i.e. B_2, B_1 and B_0. This encoder operates in a similar manner. Only one input is high or "1" and the other inputs are low or "0".

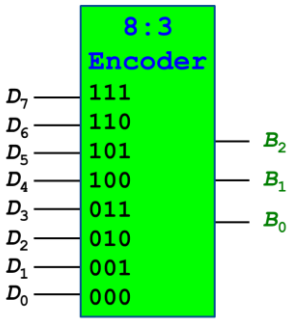


Fig. 4-26. 8-to-3 Encoder

The encoder generates the binary code corresponding to the input value. For example, if the input D_7 is high and the other inputs are low, the encoder generates the binary code B_2 B_1 B_0 = 111. If the input D_4 is high and the other inputs are low, the encoder generates the binary code B_2 B_1 B_0 = 100.

Table 4-7. Truth Table of 8-to-3 Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	B_2	B_1	B_0
0	0	0	0	0	0	0	1	**0**	**0**	**0**
0	0	0	0	0	0	1	0	**0**	**0**	**1**
0	0	0	0	0	1	0	0	**0**	**1**	**0**
0	0	0	0	1	0	0	0	**0**	**1**	**1**
0	0	0	1	0	0	0	0	**1**	**0**	**0**
0	0	1	0	0	0	0	0	**1**	**0**	**1**
0	1	0	0	0	0	0	0	**1**	**1**	**0**
1	0	0	0	0	0	0	0	**1**	**1**	**1**

Binary Decoder

The binary decoder translates the binary value into the decimal value. Fig. 4-27 shows a block diagram of 2-bit binary decoder.

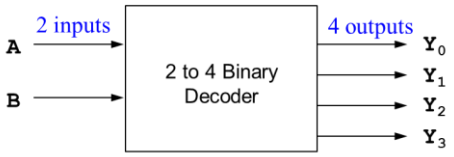


Fig. 4-27. Block Diagram of 2-bit Binary Decoder

The 2-bit binary decoder has the two inputs, A and B, and four output, Y_0, Y_1, Y_2, and Y_3. The following table shows the truth table of the decoder.

Table 4-8. Truth Table of 2-bit Binary Decoder

A	B	Y_0	Y_1	Y_2	Y_3
0	0	**1**	0	0	0
0	1	0	**1**	0	0
1	0	0	0	**1**	0
1	1	0	0	0	**1**

The following figure shows 3-to-8 binary decoder, where there are three inputs (binary code), i.e. A_2, A_1, and A_0, and eight outputs, i.e. Y_7 through Y_0.

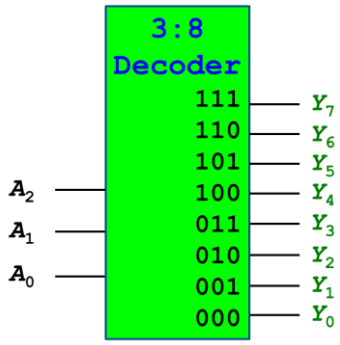


Fig. 4-28. 3-to-8 Decoder

The decoder converts the binary code into a decimal value (outputs), where only one output is high or "1" and the other outputs are low or "0". The decoder generates a decimal value corresponding to the input binary code. For example, if the binary inputs are A_2 A_1 A_0 = 110, only the output Y_6 is HIGH, and the other outputs are all LOW. The following table describes the operation of the decoder.

Table 4-9. Truth Table of 3-to-8 Decoder

Inputs			Outputs							
A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Priority Circuit

This chapter introduced the priority circuit in Section 4.1. This section describes the priority circuit in detail. Fig. 4-6 shows the truth table of the priority circuit. We will show how to design the circuit hardware from the truth table using K-map.

The following figure shows the block of the priority circuit. If the inputs A_3 A_2 A_1 A_0 are "0000", the circuit produces the outputs Y_3 Y_2 Y_1 Y_0 = 0000.

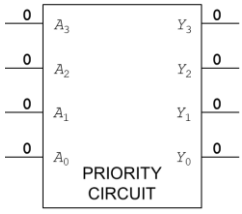


Fig. 4-29. Priority Circuit with inputs: A_3 A_2 A_1 A_0 = 0000

In the following figure, the inputs A_3 A_2 A_1 A_0 are "0001". Only A_0 asserts TRUE and the other input values are all FALSE. The circuit produces the outputs Y_3 Y_2 Y_1 Y_0 = 0001.

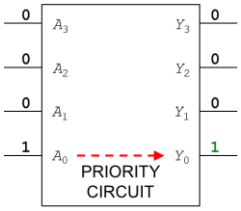


Fig. 4-30. Priority Circuit with inputs: A_3 A_2 A_1 A_0 = 0001

In the following figure, the inputs A_3 A_2 A_1 A_0 are “001X”, where the term ‘X’ represents ‘don’t care’. The higher priority inputs A_3 and A_2 didn’t assert. Since only A_1 asserts TRUE, the lower priority input A_0 doesn’t matter whether it is TRUE or FALSE. Only the output Y_1 is TRUE and the other outputs are all FALSE.

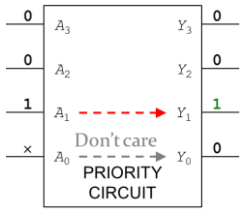


Fig. 4-31. Priority Circuit with inputs: A_3 A_2 A_1 A_0 = 001X

In the following figure, the inputs A_3 A_2 A_1 A_0 are “01XX”, where the term ‘X’ represents ‘don’t care’. The higher priority input A_3 didn’t assert. Since the next higher priority A_2 asserts TRUE, the lower priority inputs A_1 and A_0 don’t matter whether they are TRUE or FALSE. Only the output Y_2 is TRUE and the other outputs are all FALSE.

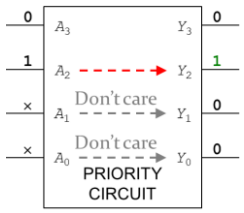


Fig. 4-32. Priority Circuit with inputs: A_3 A_2 A_1 A_0 = 01XX

In the following figure, the inputs A_3 A_2 A_1 A_0 are “1XXX”, where the term ‘X’ represents ‘don’t care’. The highest priority input A_3 asserts TRUE. The lower priority inputs A_2, A_1 and A_0 don’t matter whether they are TRUE or FALSE. Only the output Y_3 is TRUE and the other outputs are all FALSE.

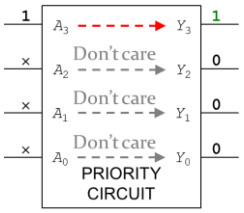


Fig. 4-33. Priority Circuit with inputs: A_3 A_2 A_1 A_0 = 1XXX

We can summarize the operation of the priority circuit in the following table:

Table 4-10. Summary of Priority Circuit Operation

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	**1**
0	0	1	×	0	0	**1**	0
0	1	×	×	0	**1**	0	0
1	×	×	×	**1**	0	0	0

where the symbol 'X' represents 'don't care".

Let's design the hardware from the truth table. The following figures show how to simplify the output Y_3 with the input variables, A_3, A_2, A_1, and A_0. With the red circuit, we can simplify the Boolean equation and draw the corresponding equation: $Y_3 = A_3$.

Y_3 A_1A_0 \ A_3A_2	00	01	11	10
00	0	0	**1**	**1**
01	0	0	**1**	**1**
11	0	0	**1**	**1**
10	0	0	**1**	**1**

Fig. 4-34. Priority Circuit Design of Y_3 using K-map

The following figures show how to simplify the output Y_2 with the input variables, A_3, A_2, A_1, and A_0. With the red circuit, we can simplify the Boolean equation and draw the corresponding equation: $Y_2 = \overline{A_3}A_2$.

Y_2 A_1A_0 \ A_3A_2	00	01	11	10
00	0	**1**	0	0
01	0	**1**	0	0
11	0	**1**	0	0
10	0	**1**	0	0

Fig. 4-35. Priority Circuit Design of Y_2 using K-map

The following figures show how to simplify the output Y_1 with the input variables, A_3, A_2, A_1, and A_0. With the red circuit, we can simplify the Boolean equation and draw the corresponding equation: $Y_1 = \overline{A_3}\,\overline{A_2}A_1$.

Y_1 A_3A_2 / A_1A_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	0	0	0
10	1	0	0	0

Fig. 4-36. Priority Circuit Design of Y_1 using K-map

The following figures show how to simplify the output Y_0 with the input variables, A_3, A_2, A_1, and A_0. With the red circuit, we can simplify the Boolean equation and draw the corresponding equation: $Y_0 = \overline{A_3}\,\overline{A_2}\,\overline{A_1}A_0$.

Y_0 A_3A_2 / A_1A_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	0	0	0	0
10	0	0	0	0

Fig. 4-37. Priority Circuit Design of Y_0 using K-map

With the above figures, from Fig. 4-34 to Fig. 4-37, we can design the priority circuit as follows:

- Input A3 directly connected to Y3
- AND1 gate produces the output Y2 with two inputs, A3 complement and A2
- AND2 gate produces the output Y1 with three inputs, i.e. A3 complement, A2 complement, and A1
- AND3 gate produces the output Y0 with four inputs, i.e. A3 complement, A2 complement, A1 complement and A0

The following figure shows the priority circuit with logic gates.

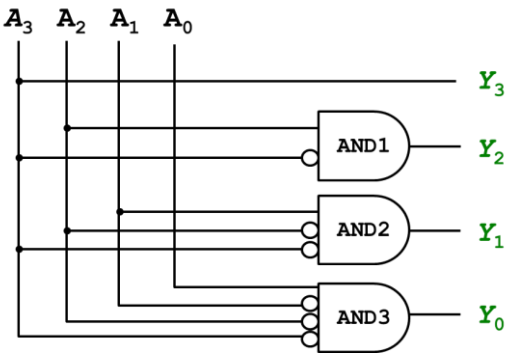


Fig. 4-38. Priority Circuit with Logic Gates

where the bubble symbol at the input side of AND gates represents a complement.

Chapter 5: Binary Number Formats

In this chapter, we introduce fixed- and floating-point number systems that can represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

Objectives

By the end of this chapter you should be able to:

- Use the fixed- and floating-point number systems to represent rational numbers.
- Demonstrate signed fixed- and floating-point numbers.
- Recall how to convert decimal numbers to binary numbers.
- Express rational numbers into scientific notations.
- Identify the biased exponent for IEEE 754 representation.
- Demonstrate the floating-point precision.

5.1 Number Systems for Binary Representations

Computers operate on both integers and fractions. So far, the numbers we can represent using binary representations include positive and negative integer numbers. Positive integer numbers are represented with unsigned binary numbers, whereas negative integer numbers are represented with two's complement and sign/magnitude numbers. How can we represent fractions? There are two common notations to represent numbers with fractions; (1) fixed point notation and (2) floating point notation. In fixed point notation, the location of decimal point is fixed and there are a fixed number of digits after the decimal point. On the other hand, floating point number allows for a varying number of digits after the decimal point, meaning that the decimal point floats to the right of the most significant '1' bit.

5.2 Fixed-Point Number Representation

The decimal number can be expressed as the sum of the products of each digit times the weight for that digit. Thus, the decimal number $\mathbf{123.45}_{10}$, can be expressed as

$$\mathbf{123.45}_{10} = (\mathbf{1} \times 10^{2}) + (\mathbf{2} \times 10^{1}) + (\mathbf{3} \times 10^{0}) + (\mathbf{4} \times 10^{-1}) + (\mathbf{5} \times 10^{-2})$$

The weight of digits moving towards left increases by a factor of 10, whereas the weight of digits moving towards right decreases by a factor of 10.

Now, let's look at binary number representation. The binary number can be expressed as the sum of the products of each digit times the weight for that digit in a similar manner. Thus, the binary number $\mathbf{101.11}_2$, can be expressed as

$$\mathbf{101.11}_2 = (\mathbf{1} \times 2^2) + (\mathbf{0} \times 2^1) + (\mathbf{1} \times 2^0) + (\mathbf{1} \times 2^{-1}) + (\mathbf{1} \times 2^{-2})$$

$$= 4 + 0 + 1 + 0.5 + 0.25 = 5.75_{10}$$

In the binary number representation, the weight of digits moving towards left increases by a factor of 2, whereas the weight of digits moving towards right decreases by a factor of 2.

For example, what decimal number does the binary number 1011.1011_2 represent? We can find the decimal number value of the binary number 1011.1011_2 with the sum of the products of each digit times the weight for that digit, such as $1011.1011_2 = 1\times2^3+0\times2^2+1\times2^1+1\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4} = 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 11.6875$.

What about the reverse process? Let's convert the decimal number 6.75_{10} to a fixed-point binary number. First, we need to split the value into the integral part and the fractional part; integral 6 and fractional 0.75. The integral part will be converted into the binary number by repeating the division, 6 = 110. The fractional part 0.75 will be converted into the binary number by repeating the multiplication as shown below:

- $0.75 \times 2 = 1.5 \rightarrow$ remove overflow digit **1**
- $0.50 \times 2 = 1.0 \rightarrow$ remove overflow digit **1**

By collecting all the overflow digits from top to bottom, we can represent the decimal number 6.75 into the binary number 110.11_2.

Exercises

Represent the decimal number 12.6875 as the binary number using 4 integer bits and 4 fraction bits.

After splitting the decimal number into the integer part and the fractional part, we can get the integral part as shown below:

- Integral part: $12 \rightarrow 1100$

The fractional part 0.6875 will be converted into the binary number by repeating the multiplication as shown below:

- $0.6875 \times 2 = 1.375 \rightarrow$ remove overflow digit 1
- $0.375 \times 2 = 0.75 \rightarrow$ no overflow 0
- $0.75 \times 2 = 1.5 \rightarrow$ remove overflow digit 1
- $0.5 \times 2 = 1.0 \rightarrow$ remove overflow digit 1

By collecting all the overflow digits from top to bottom, we can represent the decimal number 12.6875 into the binary number 1100.1011_2.

Signed Fixed-Point Numbers

The fixed-point number can represent the positive and negative values with two's complement and sign/magnitude number systems.

For example, let's represent the decimal number -7.5_{10} as a signed fixed-point binary number using 4 integer and 4 fraction bits. In the sign/magnitude number system, the first bit always represents the sign. Since the decimal number -7.5_{10} is a negative value, the sign bit should be '1'. The rest of integer bits can represent the integer value, so that we can convert the integer part 7 into the binary value 111. The fractional part 0.5 will be converted into the binary number by repeating the multiplication, 0.5 × 2 = 1.0 (overflow digit: 1). The number -7.5_{10} can be converted into the signed fixed-point number, 1111.1000, where the decimal point is fixed.

Now, let's represent the decimal number -7.5_{10} as a two's complement number. First, we will find a positive representation of the number and then we will negate the value, meaning that we convert the positive value to the negative value. We can find a positive representation of the number 7.5, by splitting the value into the integral part and the fractional part; integral 0111 and fractional 1000. Let's negate the positive representation 01111000 by inverting all the bits and adding 1 to lsb (least significant bit), as shown below:

- +7.5: `01111000`
- Invert bits: `10000111`
- Add 1 to lsb: `+ 1`
- -7.5: `10001000`

In the two's complement number system, the first digit always represents a negative value. The other bits are regular binary numbers, as shown below:

Digits	1	0	0	0	1	0	0	0
Weights	-8	4	2	1	0.5	0.25	0.125	0.0625

That means the first digit '1' represents -8 and the fifth digit '1' represents 0.5. the sum of these two digits represents -7.5 (= -8 + 0.5) which we got from the above operation.

Exercises

Convert the following two's complement binary fixed-point numbers to base 10. The implied binary point is explicitly shown to aid in your interpretation.

- 0101.1000 =

The integer part is **0101**, so that we can get the integer part as follows: **0** × -2^3 + **1** × 2^2 + **0** × 2^1 + **1** × 2^0 = 1 × 4 + 1 × 1 = 5. The fractional part is **1000**, so that we can get the fractional part as follows: **1** × 2^{-1} + **0** × 2^{-2} + **0** × 2^{-3} + **0** × 2^{-4} = 0.5. The sum of integer part and the fractional part is 5.5.

- 1111.1111 =

The integer part is **1111**, so that we can get the integer part as follows: **1** × -2^3 + **1** × 2^2 + **1** × 2^1 + **1** × 2^0 = -8 + 4 + 2 + 1 = -1. The fractional part is **1111**, so that we can get the fractional part as follows: **1** × 2^{-1} + **1** × 2^{-2} + **1** × 2^{-3} + **1** × 2^{-4} = 0.5 + 0.25 + 0.125 + 0.0625 = 0.9375. The sum of integer part and the fractional part is -1 + 0.9375 = -0.0625.

- 1000.000 =

There is only the integer part 1000. We can get the integer value, -8.

5.3 Floating-Point Number Representation

In the floating-point number, the binary point position is assumed always just before the most significant digit, which is very similar to decimal scientific notation. Before we dive into the binary number, let's look at a decimal number 273_{10}. We can write the decimal number 273_{10} in scientific notation: 273 = 2.73 × 10^2.

In general, a number is written in scientific notation as follows:

$$\pm M \times B^E$$

where the symbol M defines the mantissa (fraction), the symbol B defines the base, and the symbol E defines the exponent. In the example, the mantissa M is 2.73, the base B is 10, and the exponent E is 2.

The binary number can be written in scientific notation as shown above, where the base is 2. Once we got the scientific notation, we can store the binary number in 32 bits, as shown in Fig. 5-1. The first bit stores the sign. If the sign bit is 0, the number is positive; otherwise the number is negative. The next exponent field (8 bits) stores the exponent value. The mantissa field (23 bits) stores all the digit of the number.

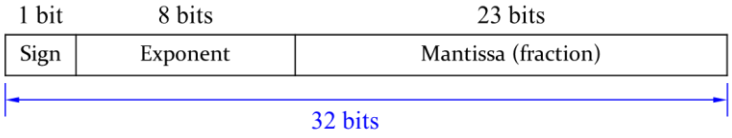


Fig. 5-1. Floating-Point Number Representation

We will show you how to represent the decimal number 228_{10} using a 32-bit floating point representation. There are three versions. The final version is called the IEEE 754 floating-point standard.

First, we need to convert the decimal number to the binary number.

- $228_{10} = 11100100_2$

Second, we can write the binary number in "binary scientific notation".

- $11100100_2 = 1.11001_2 \times 2^7$

where, we can identify the mantissa $M = 111001$, the base $B = 2$, and the exponent $E = 7$. Let's fill in each field of the 32-bit floating point number:

- The sign bit is positive (0)
- The 8-bit exponent represents the value 7: 00000111
- The remaining 23 bits are the mantissa: 11100100000000000000000

Since the mantissa has a total of 6 digits in the given example. The rest of the mantissa will be filled with '0'. The following figure show the first representation of the floating-point number.

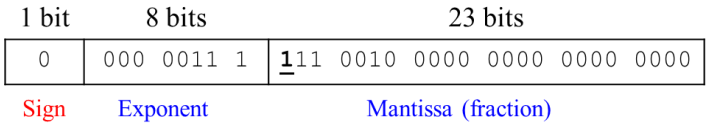


Fig. 5-2. Floating-Point Number Representation 1

The first bit of the mantissa is always '1'. The implicit leading one is not included in the 23-bit mantissa for efficiency. We only store the fraction bits in 23-bit field except the leading one. The following figure shows the second representation of the floating-point number.

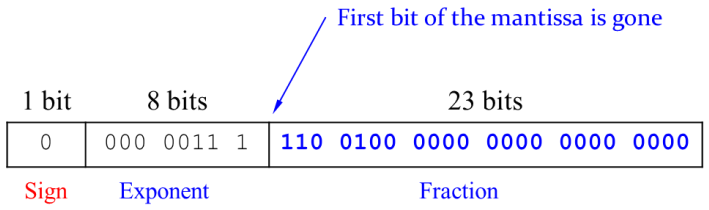


Fig. 5-3. Floating-Point Number Representation 2

Notice that the first bit of the mantissa is gone. Now we only store the fraction.

The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a biased exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127. The exponent of 7 is stored as a biased exponent that is equal to the sum of the bias (127) and the original exponent (7), i.e., 127 + 7 = 134 = 10000110_2. The IEEE 754 32-bit floating-point representation of 228_{10} is shown in the following figure:

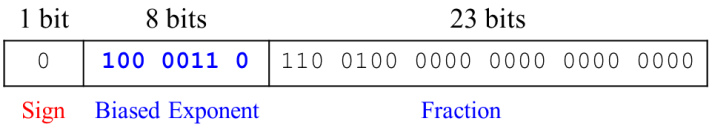


Fig. 5-4. Floating-Point Number Representation 3 – IEEE 754

The hexadecimal representation of the number is `0x43640000`.

Exercises

Write the decimal number -58.25$_{10}$ in floating point of IEEE 754 format.

First, we need to convert the decimal number to the binary number, as shown below:

- 58.25$_{10}$ = **111010.01$_2$**

Second, we can write the binary number in "binary scientific notation".

- 1.1101001×2^5

where, we can identify the mantissa $M = 11101001$, the base $B = 2$, and the exponent $E = 5$. Let's fill in each field of the 32-bit floating point number:

- The sign bit is negative (1)
- The 8-bit biased exponent bits: (127 + 5) = 132 = 10000100$_2$
- 23 fraction bits: 110 1001 0000 0000 0000 0000

Note that the first bit of the mantissa is gone and we have 23 fraction bits.

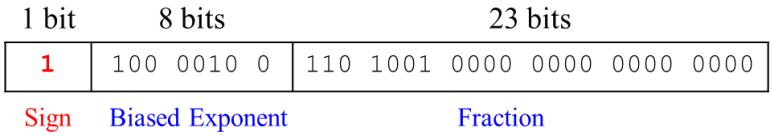


Fig. 5-5. Floating-Point Number Representation with IEEE 754 Format

The hexadecimal representation of the number is `0xC2690000`.

Special Cases

The IEEE 754 floating-point format has special cases to represent numbers such as zero, positive and negative infinity, and illegal results. The following figure show special cases of these values.

Table 5-1. Special Cases of IEEE 754 Standard Format

Number	sign	Exponent (8 bits)	Fraction (23 bits)
0	x	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
-∞	1	11111111	00000000000000000000000
NaN	x	11111111	Non-zero

We have showed 32-bit floating-point numbers. When you declare a float variable in your program language, the variable is stored with the format we have discussed so far in the computer system. The format is also called single-precision (float) or single. The IEEE 754 standard also defines 64-bit double-precision numbers (also called doubles) that can provide greater precision and range.

The following table shows the number of bits used for the fields in each format.

Table 5-2. Single-Precision and Double-Precision Formats of IEEE 754 Standard

Format	Total bits	Sign bits	Exponent bits	Bias value	Fraction bits
Single-Precision	32	1	8	127	23
Double-Precision	64	1	11	1023	52

Recall that a number overflows when its magnitude is too large to be represented. Likewise, a number underflows when it is too tiny to be represented. Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: round down, round up, round toward zero, and round to nearest. The default rounding mode is round to nearest.

For example, round the value 1.100101 (1.578125) to only 3 fraction bits. If the round down mode is applied, the value rounds '1.100'. If the round up mode is applied, the value rounds '1.101'. If the round toward zero is applied, the value rounds '1.100'. If the round to a neighboring number is applied, the value rounds '1.101', because 1.625_{10} (1.101_2) is closer to 1.578125^{10} (1.100101_2) than 1.5^{10} (1.1_2) is.

Chapter 6: Computer Arithmetic

In this chapter, we introduce arithmetic circuits which are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shift, multiplication and division. This module describes hardware implementations for all of these operations.

Objectives

By the end of this chapter you should be able to:

- Demonstrate knowledge of 1-bit half and full adders
- Demonstrate knowledge of four-bit adder and subtractor
- Recall how to operate four-bit adder-subtractor
- Evaluate the arithmetic operation with four-bit adder-subtractor
- Execute arithmetic logic operations with Arithmetic logic unit
- Differentiate logical shift and arithmetic shift
- Apply arithmetic and shift operations for multiplication and division

6.1 Boolean Addition

Let's look at how the computer execute the Boolean addition, 5 + 6, with binary numbers.
We assume an 8-bit computer. The decimal number 5 will be converted to an 8-bit binary number, 0000 0101. The decimal number 6 will be converted to an 8-bit binary number, 0000 0110, as shown in the below figure:

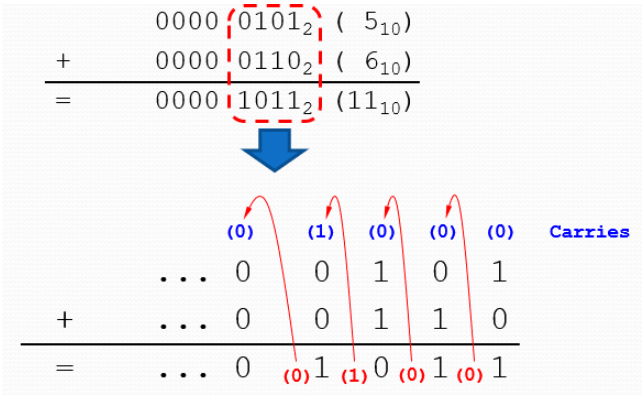


Fig. 6-1. Boolean Addition: 5 + 6

As we calculate the decimal addition, the rightest digits will be added first and then the next digits will be executed. The sum of 1 and 0 is 1 with a carry 0. In the second column from the right side, the sum of 0, 1, and the carry 0 is 1 with a carry 0. In the third column from the right side, the sum of 1, 1 and the carry 0 is 0 with a carry 1. In the fourth column, the sum of 0, 0, and the carry 1 is 1 with a carry 0.

We can execute the Boolean addition using 1-bit full adders. A 1-bit half adder is used for adding together the two least significant digits in a binary sum. It has two inputs A and B, and two outputs, Sum and C_{out}. But it lacks a C_{in} (carry) input.

1-Bit Full Adders

1-bit full adder can perform addition of numbers, where it has such as inputs and outputs:

- Inputs: A, B, Carry in (Cin)
- Outputs: Carry out (Cout), Sum (S)

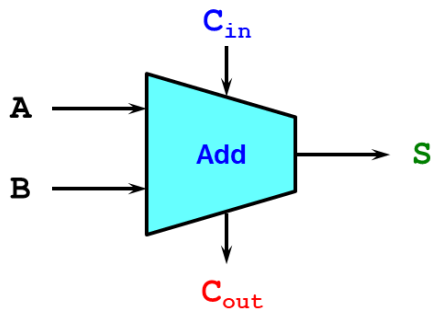


Fig. 6-2. 1-Bit Full Adder

The following table describes the operation of the adder. Here, the sum S will be '1' if the number of the input '1' is odd. For example, if the inputs A, B, and C_{in} are 001, the sum S is '1'. If the inputs A, B, and C_{in} are 110, the sum S is '0'. The C_{out} will be '1' if the number of the input '1' is greater than or equal to 2. For example, if the inputs A, B, and C_{in} are 110, the C_{out} is '1'.

Table 6-1. Truth Table of 1-bit Full Adder

A	B	C_{in}	S	C_{out}
0	0	0	**0**	0
0	0	1	**1**	0
0	1	0	**1**	0
0	1	1	**0**	1
1	0	0	**1**	0
1	0	1	**0**	1
1	1	0	**0**	1
1	1	1	**1**	1

We can simplify the outputs S and C_{out} using K-map. The following figure shows K-map representation of the output S.

C_{in} \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Fig. 6-3. K-map representation of the output S

From the above figure, we can express the output *S* in terms of input variables as follows:

$$S = A\bar{B}\bar{C}_{in} + \bar{A}B\bar{C}_{in} + \bar{A}\bar{B}C_{in} + ABC_{in} = A \oplus B \oplus C_{in}$$

In order to produce the output S, there are four AND gates and one OR gate needed as shown in Fig. 6-4:

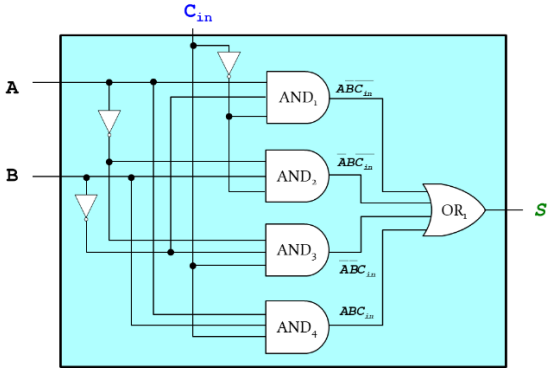


Fig. 6-4. Logic gate circuit for the Output S

Each AND gate has the following inputs:

- AND_1: A, $\bar{B}$, and $\bar{C}_{in}$
- AND_2: $\bar{A}$, B, and $\bar{C}_{in}$
- AND_3: $\bar{A}$, $\bar{B}$, and C_{in}
- AND_4: A, B, and C_{in}

All outputs of the AND gates fed into the OR gate that produces the sum *S*.

The following figure shows K-map representation of the output C_{out}.

C_{in} \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Fig. 6-5. K-map representation of the output C_{out}

From the above figure, we can express the output C_{out} in terms of input variables as follows:

$$C_{out} = AC_{in} + BC_{in} + AB$$

In order to produce the output C_{out}, there are three AND gates and one OR gate needed as shown in Fig. 6-6:

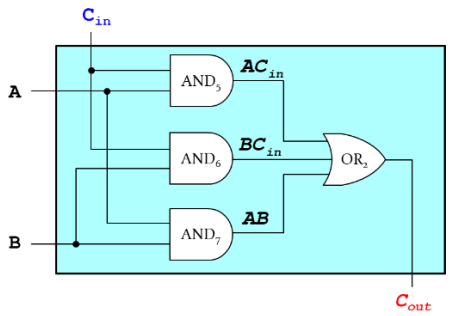


Fig. 6-6. Logic gate circuit for the Output C_{out}

Each AND gate has the following inputs:

- AND_5: A and C_{in}
- AND_6: B and C_{in}
- AND_7: A and B

All outputs of the AND gates fed into the OR gate that produces the carryout C_{out}.

By combining Figs. 6-4 and 6-6, we can draw the 1-bit full adder with the carryin C_{in}, as shown in the following figure:

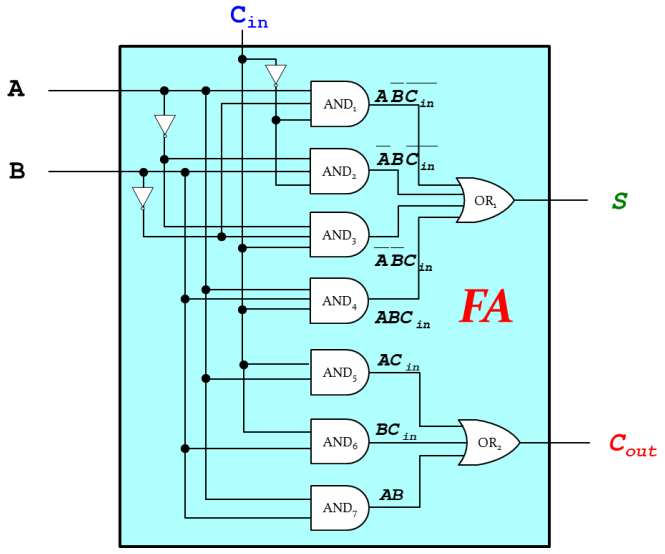


Fig. 6-7. Logic gate circuit for the 1-Bit Full Adder

In the above figure, the 1-bit full adder has three inputs, A, B and C_{in}; and two outputs, S and C_{out}. All the logic gates for the outputs S and C_{out} placed in a single block. The sum S is produced with four AND gates and one OR gate in the same way to generate the output S in Fig. 6-4. The C_{out} is produced with three AND gates and one OR gate in the same way to generate the output C_{out} in Fig. 6-6.

Four-Bit Adders

How can we design four-bit adders? The four-bit adder is designed with four 1-bit full adders by connecting them in a parallel manner. The carryout C_{out} of the first full adder connected to the carry in C_{in} of the second full adder, C_{out} of the second full adder connected to C_{in} of the third fuller adder, and C_{out} of the third full adder connected to C_{in} of the fourth fuller adder, as shown in the following figure:

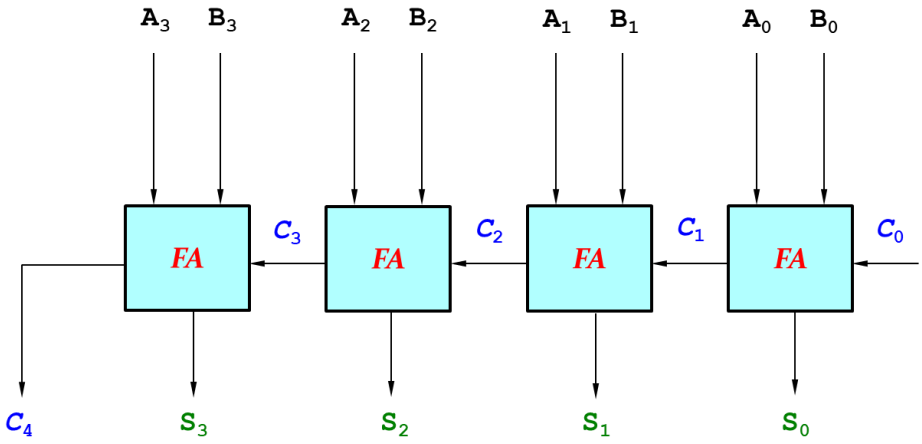


Fig. 6-8. Four-bit Adder

where the binary input A includes A_3, A_2, A_1 and A_0, the binary input B includes B_3, B_2, B_1 and B_0, C_{in} is C_0, the output S includes S_3, S_2, S_1 and S_0, and the C_{out} is C_4.

Exercises

Add 3 and 4 using the four-bit adder.

- 3 = 0011 (A), 4 = 0100 (B)

Inputs A (0011) and B (0100) are fed into 4-bit full adder. The sum of A_0, B_0 and C_0 is 1 (S_0), the sum of A_1, B_1 and C_1 is 1 (S_1), the sum of A_2, B_2 and C_2 is 1 (S_2), and the sum of A_3, B_3 and C_3 is 0 (S_3). By collecting all the bits S_3 – S_0, it produces 0111 (=7) which is the sum of 3 and 4.

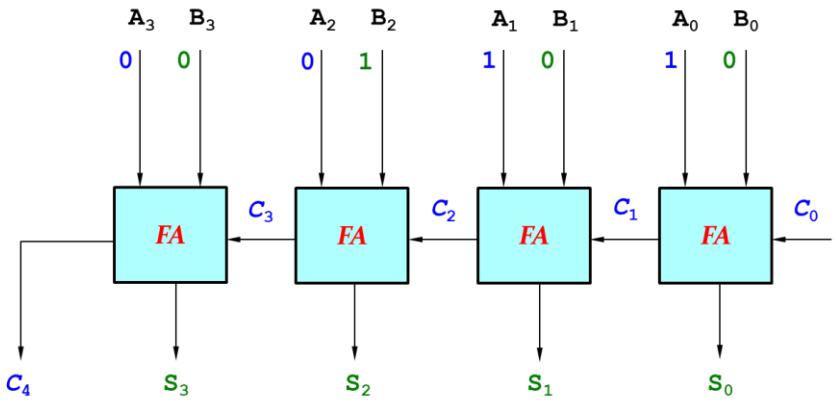


Fig. 6-9. Add 3 and 4 using four-bit adder

6.2 Boolean Subtraction

Let's look at how the computer execute the Boolean subtraction, 12 - 5, with binary numbers. We assume an 8-bit computer. The decimal number 12 will be converted to an 8-bit binary number, 0000 1100. The decimal number 5 will be converted to an 8-bit binary number, 0000 0101, as shown in the below:

$$\begin{array}{ll} & 0000\ 1100_2\ (12_{10}) \\ - & 0000\ 0101_2\ (5_{10}) \\ \hline = & 0000\ 0111_2\ (7_{10}) \end{array}$$

The logic operation of the above is not appreciate with the logic gates. Instead of the above operation, the Boolean subtraction is converted to the addition by converting the subtracted value to negate using two's complement.

We can negate the number 5 by flipping all the bits and adding one to the lsb (least significant bit), as shown below:

$$\begin{array}{ll} & 1111\ 1010_2 \\ + & \qquad 1 \\ \hline & 1111\ 1011_2\ (-5_{10}) \end{array}$$

The Boolean subtraction is now converted to the addition, as shown below:

$$\begin{array}{ll} 1 & 1111\ 0000\ \ \text{carries} \\ & 0000\ 1100_2\ (12_{10}) \\ + & 1111\ 1011_2\ (-5_{10}) \\ \hline = & 0000\ 0111_2\ (7_{10}) \end{array}$$

The most left carry bit (9th bit) will be ignored because the operation is executed in an 8-bit computer. The overflow may occur when there are insufficient bits in a binary number representation to portray the result of an arithmetic operation.

Four-Bit Subtractor

Four-bit subtractor can be designed with 4-bit full adder, by adding NOT gates and carryin (C_{in}) = 1, as shown in the below figure:

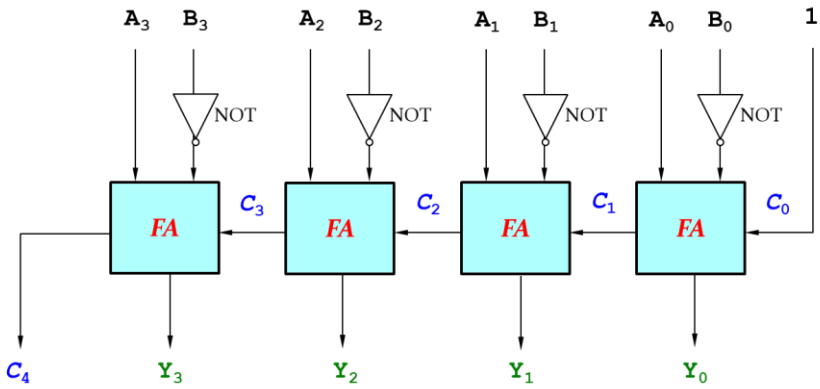


Fig. 6-10. Four-bit Subtractor

where the input bits A_3 through A_0 are fed into the full adder directly, the input bits B_3 through B_0 are fed into the full adder after flipping the bits with NOT gates, and the first carryin C_0 set to 1, $C_0 = 1$. The Boolean equation of the above block diagram is expressed as shown below:

$$Y = A - B = A + (\bar{B} + 1)$$

The subtractor produces the output Y, Y_3 through Y_0, and the carryout C_4.

Let's subtract 3 from 7 using the four-bit subtractor.

- 7 = 0111 (A); $A_3 = 0$, $A_2 = 1$, $A_1 = 1$, $A_0 = 1$,
- 3 = 0011 (B); $B_3 = 0$, $B_2 = 0$, $B_1 = 1$, $B_0 = 1$.

The binary inputs A and B ($A_3 – A_0$ and $B_3 – B_0$) are fed into the subtractor as shown below:

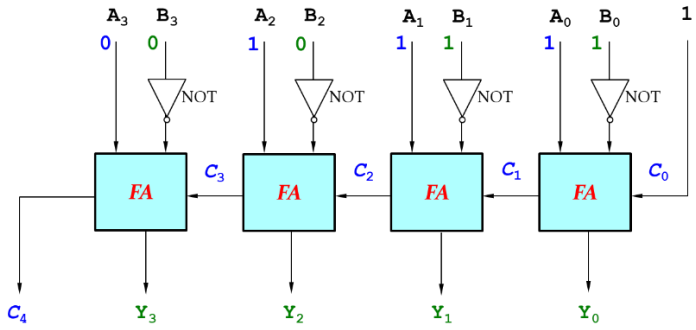


Fig. 6-11. Subtract 3 from 7

Starting from the rightest side, we can execute the full adder operation with A_0 = 1, $\bar{B}_0$ = 0, and C_0 = 1, producing the outputs Y_0 = 0 and C_1 = 1, where C_1 become the carryin for the next full adder. The second full adder executes with the following inputs; A_1 = 1, $\bar{B}_1$ = 0, and C_1 = 1, producing the outputs Y_1 = 0 and C_2 = 1, where C_2 become the carryin for the next full adder. The third and fourth full adders execute in a similar manner: the inputs A_2 = 1, $\bar{B}_2$ = 1, and C_2 = 1 produce the outputs Y_2 = 1 and C_3 = 1; and the inputs A_3 = 0, $\bar{B}_3$ = 1, and C_3 = 1 produce the outputs Y_3 = 0 and C_4 = 1, where the carryout C_4 will be ignored because this is a 4-bit computer system. The output bits $Y_3Y_2Y_1Y_0$ (= 0100) is equal to 4.

6.3 Adder-Subtractor

The hardware configure of the adder is very similar to that of the subtractor. Since the hardware configuration is related to the cost. We can design this two hardware by sharing some components, i.e. full adders. A four-bit Adder-Subtractor can be designed with 4-bit full adder, four exclusive OR gates and a mode bit M, as shown in the below:

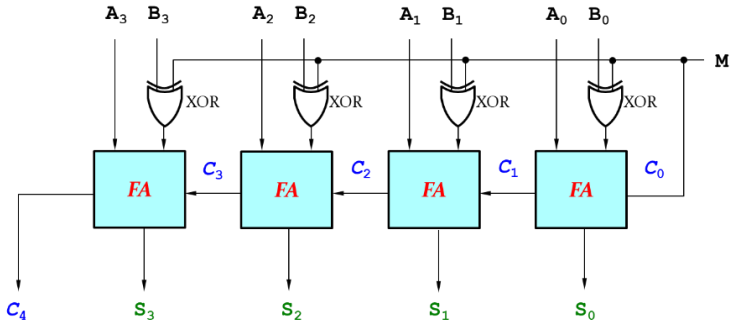


Fig. 6-12. Four-bit Adder-Subtractor

where the input bits A_3 through A_0 are fed into the full adder directly, the input bits B_3 through B_0 are fed into the full adder after executing exclusive OR operation with a mode bit M. The mode bit M

determines either an adder (M = 0) or a subtractor (M = 1). The Adder-Subtractor produces the output bits S_3 through S_0, and the carryout C_4, where the carryout C_4 is used for overflow detection.

The following figure shows the case when the mode bit M is equal to 0, executing the add operation.

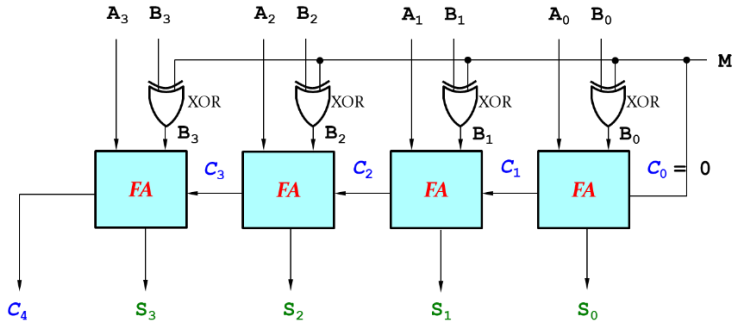


Fig. 6-13. Four-bit Adder-Subtractor: M = 0

Since the mode bit M = 0, the outputs of exclusive OR gates are equal to B_3, B_2, B_1 and B_0. Then the four-bit Adder-Subtractor produces the output bits, S_3 through S_0, and the carryout C_4. It works as a 4-bit full adder.

If the mode bit M = 1, as shown in Fig. 6-14, the outputs of exclusive OR gates are equal to $\bar{B}_3$, $\bar{B}_2$, $\bar{B}_1$, and $\bar{B}_0$, where the carryin C_1 of the first full adder is equal to 1.

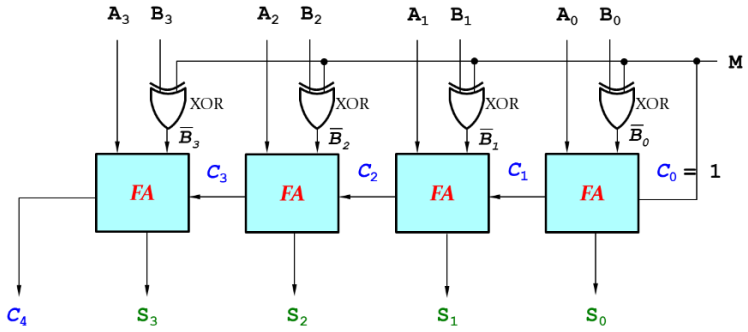


Fig. 6-14. Four-bit Adder-Subtractor: M = 1

Then the four-bit Adder-Subtractor produces the output bits, S_3 through S_0, and the carryout C_4. It works as a 4-bit subtractor.

Let's look at how this circuit works with the following example inputs;

- A = 5 (0101)
- B = 7 (0111)
- The mode bit M = 1

The input bits A_3 through A_0 are directly fed into the adders. On the other hand, the input bits B_3 through B_0 are fed into the exclusive OR gates. Since the mode bit M = 1, the outputs of the exclusive OR gates are $\bar{B}$ (B complement).

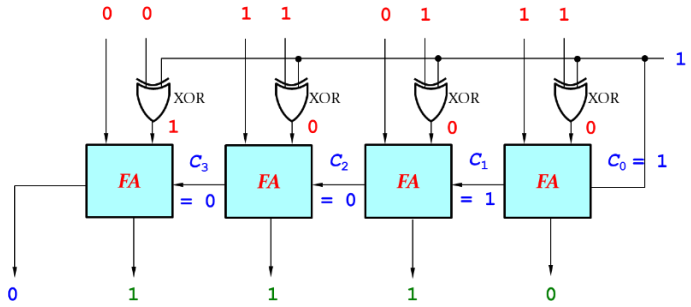


Fig. 6-15. Four-bit Adder-Subtractor with Example Inputs: A = 5, B = 7, and M = 1

From the rightest side of the above figure,

- Three inputs A_0 = 1, $\bar{B}_0$ = 0, carryin C_0 = 1 fed into the first FA, and then produce S_0 = 0 and carryout C_1 = 1.
- Three inputs A_1 = 0, $\bar{B}_1$ = 0, carryin C_1 = 1 fed into the second FA and then produce S_1 = 1 and carryout C_2 = 0.
- Three inputs A_2 = 1, $\bar{B}_2$ = 0, carryin C_2 = 0 fed into the third FA, and then produce S_2 = 1 and carryout C_3 = 0.
- Three inputs A_3 = 0, $\bar{B}_3$ = 1, carryin C_3 = 0 fed into the fourth FA, and then produce S_3 = 1 and carryout C_4 = 0.

Let's collect all the sum bits, $S_3S_2S_1S_0$ (= 1110), which is equal to -2 (two's complete number).

Exercises

Eight-bit Adder-Subtractor can be designed with eight 1-bit full adders by connecting them in a parallel manner.

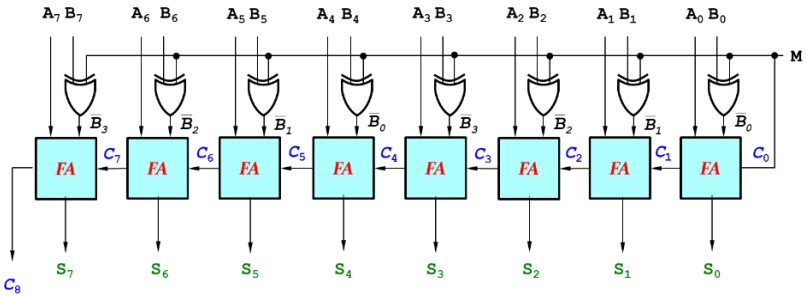


Fig. 6-16. Eight-bit Adder-Subtractor

- C_{out} (C_1) of the first full adder connected to C_{in} of the second fuller adder, C_{out} (C_2) of the second full adder connected to C_{in} of the third fuller adder, etc.
- Binary input A includes A_7 through A_0
- Binary input B includes B_7 through B_0
- C_{in}: C_0
- Output S includes S_7 through S_0.
- C_{out}: C_4

If A = 32 (`00100000`), B = 63 (`00111111`), and M = `1`, what is the output S?

6.4 Comparators

A 1-bit comparator is designed with an exclusive OR gate, as shown in Fig. 6-17.

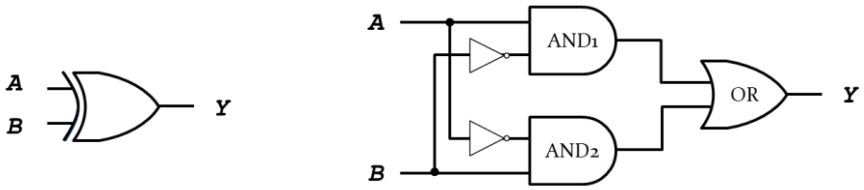


Fig. 6-17. Eight-bit Adder-Subtractor

There are two AND gates and one OR gate. AND_1 gate has two inputs, A and $\bar{B}$. AND_2 gate has also two inputs, $\bar{A}$ and B. The two outputs of AND gates are fed into the OR gate that produces the output Y.

Equality

1-bit equality comparator is simply designed with exclusive NOR gate. What about 4-bit equality comparator? 4-bit equality comparator is designed by connecting 4 exclusive NOR gates parallelly, as shown below:

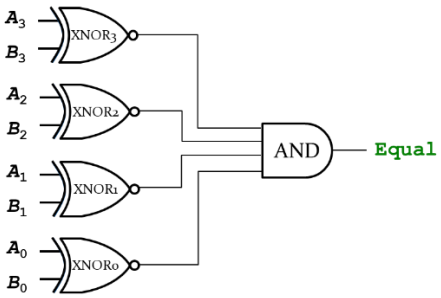


Fig. 6-18. Four-bit Equality Comparator

Each XNOR gate compares the inputs A and B as follows:

- $XNOR_3$ for two inputs A_3 and B_3
- $XNOR_2$ for two inputs A_2 and B_2
- $XNOR_1$ for two inputs A_1 and B_1
- $XNOR_0$ for two inputs A_0 and B_0

The outputs of all XNOR gates are fed into 4-input AND gate, which produces the logic "1" if only if all the inputs are "1".

Less Than

The Less Than comparator can be designed with magnitude comparison by computing A – B and looking at the sign bit (msb: most significant bit), as shown in the below:

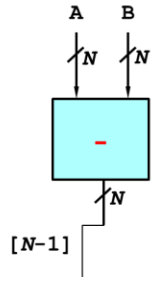


Fig. 6-19. Less Than Comparator

where the term 'N' means the M-bit input or output, and [N-1] represents the sign bit, i.e. the most significant bit. If the sign bit is `1`, A is less than B. If the sign bit is `0`, A is greater than or equal to B.

6.5 Arithmetic Logic Unit

An ALU (Arithmetic Logic Unit) combines a variety of mathematical and logical operations, e.g. addition, subtraction, magnitude comparison, AND operation, OR operation, etc., into a single unit. The following figure shows a simplified ALU.

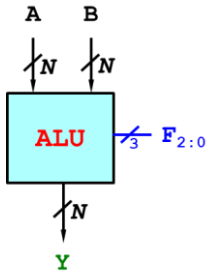


Fig. 6-20. Simplified ALU

There are the two inputs A and B. Both of them are N bits. The operation of ALU determined by the function bits, F_2 F_1 F_0. For example, if F_2 F_1 F_0 = 000, ALU executes AND operation. If F_2 F_1 F_0 = 001, ALU executes OR operation. The following table explains the operation of the ALU with the function bits.

Table 6-2. ALU Operation with Function Bits

F_2	F_1	F_0	Function
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A + B
0	1	1	Not used
1	0	0	A AND ~B
1	0	1	A OR ~B
1	1	0	A - B
1	1	1	SLT

N-bit ALU

The simplified N-bit ALU of Fig. 6-20 is designed with 2-to-1 Multiplexer, full adder, zero extend, NOT gate, OR gate, AND gate, and 4-to-1 Multiplexor, as shown in Fig. 6-21, where all the units are N bits.

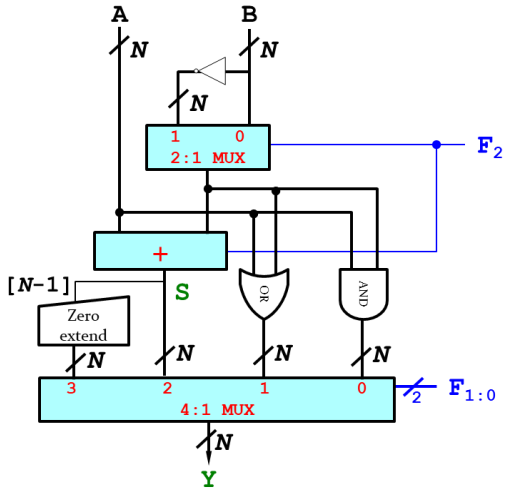


Fig. 6-21. Design of N-bit ALU

Input A (n-bit) directly is fed into the full adder, OR gate, or AND gate. Input B (n-bit) is fed into 2-to-1 Multiplexor with and without NOT gate, where 2-to-1 Multiplexor produce either *B* or $\bar{B}$ depending on the function bit F_2 value. If the function bit F_2 is true (F_2 = 1), the multiplexor produces $\bar{B}$. If the function bit F_2 is false (F_2 = 0), it produces B. The full adder has two inputs A and B (or $\bar{B}$). In a similar manner,

each logic OR or AND gate has two inputs A and B (or $\bar{B}$). Zero extend detects the most significant bit of the full adder output (the sum S) and produces a total of N bits, where the other bits except the least significant bit are filled with '0'.

4-to-1 Multiplexor has the following four inputs:

- the output of zero extend
- the output of the adder
- The output of OR gate
- The output of AND gate

It forward one of inputs to the output Y depending on the two function bits F_1 F_0.

Let's look at an example how the ALU operates with the following inputs:

- Input A = 25
- Input B =32
- Function bits F_2 F_1 F_0 = `111`

where we assume this is a 32-bit ALU. Input A = 25 (32-bit) is directly fed into the full adder, whereas input $\bar{B}$ (32-bit) is fed into the full adder because 2-to-1 Multiplexor produces $\bar{B}$ with F_2 = `1`. Here, the full adder works as subtractor. Note that the carryin of the full adder is F_2 = `1`.

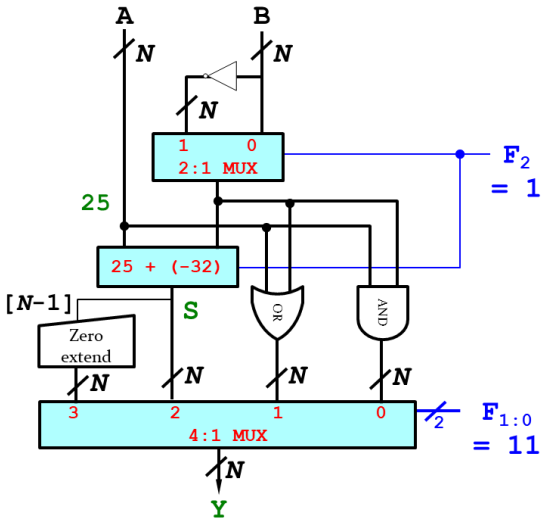


Fig. 6-22. SLT Operation of N-bit ALU

The output of the adder is -7, i.e. A – B = 25 – 32 = -7. The most significant bit (msb) S_{31} is equal to `1` because the output value of the adder is negative. The zero-extend extends the msb and produces `0x00000001`. Since F_1 F_0 = `11`, the output of zero-extend is forwarded to the output of 4-to-1 Multiplexer, i.e. Y = `0x00000001`.

Exercises

Let's execute the designed 32-bit ALU with the following inputs:

- Input A = 16
- Input B =31
- Function bits F_2 F_1 F_0 = `110`

where we assume this is a 32-bit ALU. Input A = 16 (32-bit) directly is fed into the full adder, whereas input $\bar{B}$ (32-bit) is fed into the full adder because 2-to-1 Multiplexor produces $\bar{B}$ with F_2 = `1`. Here, the full adder works as subtractor.

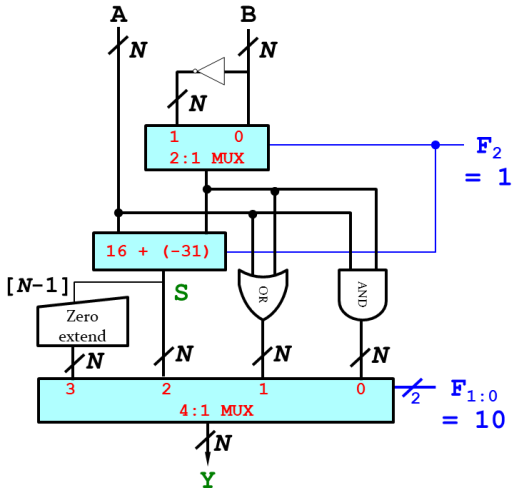


Fig. 6-23. Subtract Operation of N-bit ALU

The adder executes the following operation:

- Binary input A = `0000|0000|0000|0000|0000|0000|0001|0000`
- Binary input $\bar{B}$ = `1111|1111|1111|1111|1111|1111|1110|0000`
- Carryin F_2 = `0000|0000|0000|0000|0000|0000|0000|0001`
- Binary output S = `1111|1111|1111|1111|1111|1111|1111|0001` (=-15)

Since two function bits F_1 F_0 = `10`, the 4-to-1 Multiplexer forwards the binary output S to the output value Y = S = `0xFFFFFFF1`.

Logical Shift

A logical shift is a bitwise operation that shifts all the bits. The two base variants are the logical left shift and the logical right shift. In the logic left shift, shift all the bits to left and fill empty spaces with 0's:

`11001011` LSL 1 = `10010110`

where the underlined zero is added to fill the empty spaces. The most significant bit is discarded.

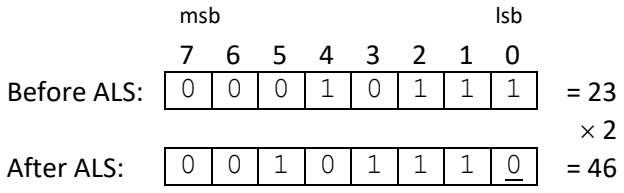
In the logical right shift, shift all the bits to right and fill empty spaces with 0's:

11001011 LSR 2 = 00110010

where the underlined zeros are added to fill the empty spaces. The least significant bits are discarded.

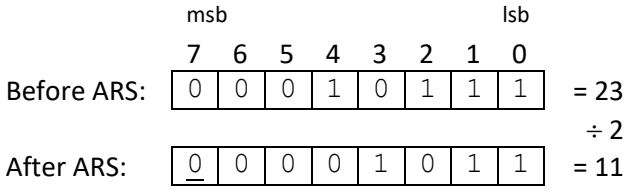
Arithmetic Shift

An arithmetic shift is also a bitwise operation that shifts all the bits. The two base variants are the arithmetic left shift and the arithmetic right shift. The operation of the arithmetic left shift is the same as the logic left shift. The vacant least significant bit is fill with zero and the most significant bit is discarded.



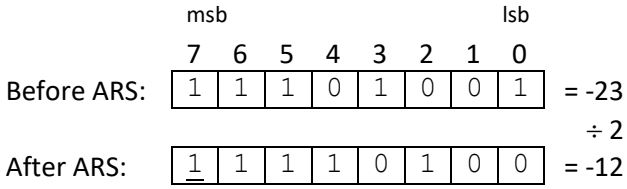
The arithmetic left shift is equivalent to multiplication. After we execute ALS by 1 bit, the original value is multiplied with $2^{1\text{-bit}}$. For example, if you execute ALS by 1 bit with 00010111 (=23), the operation returns 00101110 (= 46). If you execute ALS by 2 bits with 00010111 (=23), it returns 01011100 (=$23 \times 2^2 = 92$).

The arithmetic right shift is equivalent to division. After we execute ARS by 1 bit, the original value is divided by $2^{1\text{-bit}}$. For example, if you execute ARS by 1 bit with 00010111 (=23), the operation returns 00001011 (=11), where the result is always round down.



where the vacant most significant bit is filled with a copy of the original msb zero, where the original number is positive. If the number is negative, the vacant most significant bit is filled with one.

Let's execute ARS by 1 bit with 11101001 (=-23). The operation returns 11110100 (-12), where the vacant msb is filled with a copy of the original msb one. The result is always round down.



Exercises

Execute the logical shift operation of the following values:

- `11001` LSR 2 = `00110`

  The original value `11001` is shifted right. The vacant msb is filled with zero and the lsb is discarded.

- `11001` LSL 2 = `00100`

  The original value `11001` is shifted left. The vacant lsb is filled with zero and the msb is discarded.

Execute the arithmetic shift operation of the following values:

- `11001` ASR 2 = `11110`

  The original value `11001` is shifted right. The vacant msb is filled with a copy of the original msb.

- `11001` ASL 2 = `00100`

  The original value 11001 is shifted left. The vacant lsb is filled with a copy of the original lsb.

Chapter 7: Circuit Designs and Sequential Circuits

In this chapter, we introduce the most commonly used building blocks: multiplexer, decoder, and encoder. The outputs of these combinational logic circuits depend on current input values, hiding the unnecessary gate-level details to emphasize the function of the building block. This chapter also introduces sequential logic circuits, which outputs depend on both current and prior values.

Objectives

By the end of this chapter you should be able to:

- Demonstrate knowledge of multiplexer, decoder, and encoder
- Simplify the Boolean equation with k-map
- Design combinational logic circuits with logic gates
- Differentiate combinational logic circuits and sequential logic circuits
- Recall basic knowledge of SR latch: set, reset, memory and invalid state
- Evaluate internal circuit operations of D latch and D flip-flop

7.1 Combinational Circuit Design

Multiplexer

A multiplexer (or Mux), also known as a data selector, is a device that selects one among N analog or digital inputs and forwards the selected input to a single output line. If the mux has *N* inputs, it needs $\log_2 N$ control inputs. For example, if the mux has the two inputs, it needs a $\log_2 2$ (= 1) control input. The following figure shows 2-to-1 multiplexer which has two inputs (D_0 and D_1), one output (*Y*), and a control input (*S*).

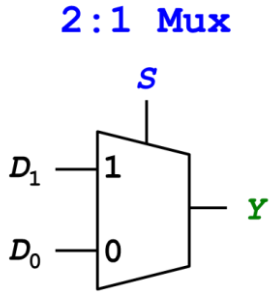


Fig. 7-1. 2-to-1 Multiplexer

If the control input *S* is 0, the input D_0 is forwarded to the output *Y*. If the control input *S* is 1, the input D_1 is forwarded to the output *Y*. The following table describes the 2-to-1 multiplexer.

Table 7-1. Truth Table of 2-to-1 Multiplexer

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

From the above table, we can notice that the value of the output *Y* is determined by two inputs, D_1 and D_0, and the control input *S*. Let's draw K-map to simplify the Boolean equation in terms of the output *Y*, as shown in Fig. 7-2.

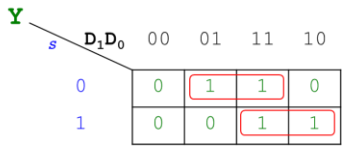


Fig. 7-2. 2-to-1 Multiplexer

The simplified equation is $\boldsymbol{Y} = D_0\bar{S} + D_1S$ which allows us to design logic gates for2-to-1 multiplexer. For the multiplexer design, there needs two AND gates and one OR gate in the block of Fig. 7-1.

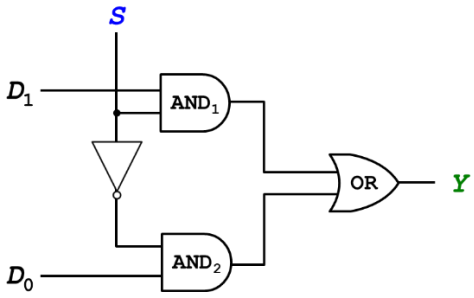


Fig. 7-3. 2-to-1 Multiplexer

As shown in Fig. 7-3, AND_1 gate has two inputs, D_1 and S, whereas AND_2 gate has two inputs, D_0 and $\bar{S}$. The two outputs of the AND gates are fed into the OR gate which produces the output *Y*.

Decoder

The decoder translates the binary value into a decimal value. Fig. 7-4 shows a block diagram of 2-bit binary decoder. The 2-to-4 decoder has two inputs A_1 and A_0, and four outputs Y_3, Y_2, Y_1 and Y_0. Depending on the binary inputs A_1 and A_0, only one output will be TRUE and the other outputs will be FALSE. For example, if the binary inputs A_1A_0 is equal to '11', only one output Y_3 is TRUE and the other outputs Y_2 Y_1 Y_0 are all FALSE.

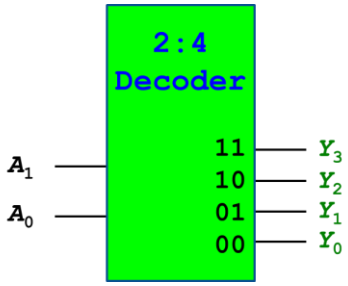


Fig. 7-4. Block Diagram of 2-bit Binary Decoder

The following table shows the truth table of the decoder.

Table 7-2. Truth Table of 2-bit Binary Decoder

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

The truth table of 2-to-4 binary decoder can be mapped to K-map and we can describe the output variables in terms of input variables.

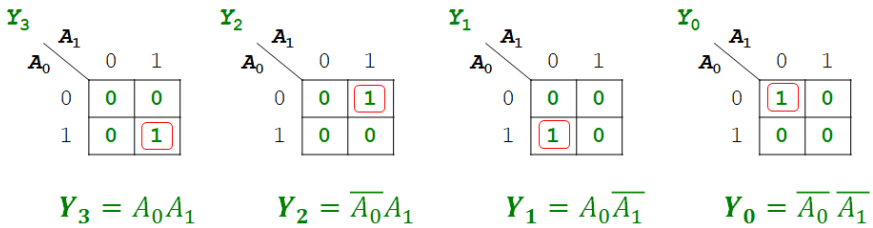


Fig. 7-5. K-Map Representation of 2-bit Binary Decoder

Each output value was expressed with input variables, as shown in the above figure. We can fill the box of Fig. 7-4 with a combinational circuit of 2-bit binary decoder in the following figure:

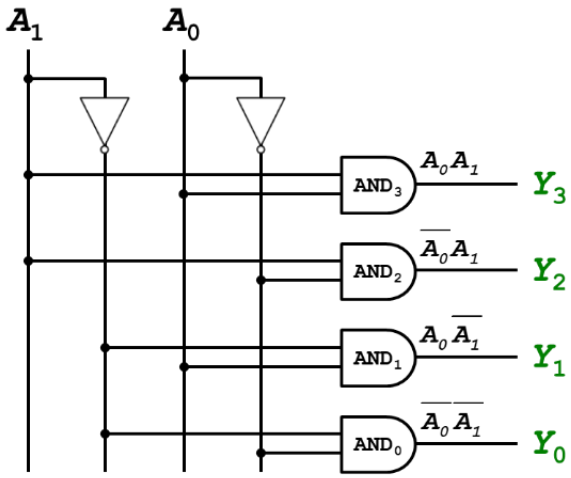


Fig. 7-6. Combinational Circuit of 2-bit Binary Decoder

The 2-to-4 binary decoder has two inputs (A_1, A_0) and four outputs (Y_3, Y_2, Y_1, and Y_0). There are four AND gates and two NOT logic gate in the block.

- AND_3 gate produces the output Y_3 with two inputs, A_0 and A_1.
- AND_2 gate produces the output Y_2 with two inputs, $\bar{A}_0$ and A_1.
- AND_1 gate produces the output Y_1 with two inputs, A_0 and $\bar{A}_1$.
- AND_0 gate produces the output Y_0 with two inputs, $\bar{A}_0$ and $\bar{A}_1$.

Encoder

The encoder is the inverse operation of a decoder. The operation is like the keyboard. Only one input is TURE (press only one button) and the others are FALSE. It generates the binary code corresponding to the input value. A 4-to-2 encoder has four inputs D_3 D_2 D_1 D_0 and two binary outs B_1 B_0, as shown in Fig. 7-7. Only one input is TRUE and the other inputs are FALSE. For example, if the input D_3 is TRUE and the others are FALSE, it generates the corresponding binary code B_1 B_0 = 11.

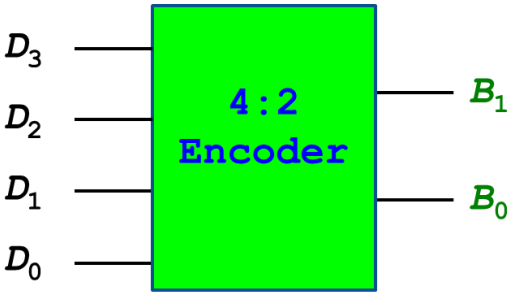


Fig. 7-7. 4-to-2 Encoder

The following table describe the operation of the 4-to-2 encoder.

Table 7-3. Truth Table of 4-to-2 Encoder

Inputs				Outputs	
D_3	D_2	D_1	D_0	B_1	B_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

From the above table, we can notice that the outputs B_1 and B_0 are determined by four inputs, D_3, D_2, D_1 and D_0, respectively. Using the truth table of the encoder, we can create K-map which can simplify the Boolean equation for the encoder.

To simplify the Boolean equation of the output B_1, we have the following Boolean values:

- If the inputs D_3 D_2 D_1 D_0 are equal to '0 0 0 1', the output B_1 = 0
- If the inputs D_3 D_2 D_1 D_0 are equal to '0 0 1 0', the output B_1 = 0
- If the inputs D_3 D_2 D_1 D_0 are equal to '0 1 0 0', the output B_1 = 1
- If the inputs D_3 D_2 D_1 D_0 are equal to '1 0 0 0', the output B_1 = 1

To simplify the Boolean equation of the output B_0, we have the following Boolean values:

- If the inputs D_3 D_2 D_1 D_0 are equal to '0 0 0 1', the output B_0 = 0
- If the inputs D_3 D_2 D_1 D_0 are equal to '0 0 1 0', the output B_0 = 1
- If the inputs D_3 D_2 D_1 D_0 are equal to '0 1 0 0', the output B_0 = 0
- If the inputs D_3 D_2 D_1 D_0 are equal to '1 0 0 0', the output B_0 = 1

The following figure shows the K-map representation of 4-to-2 encoder.

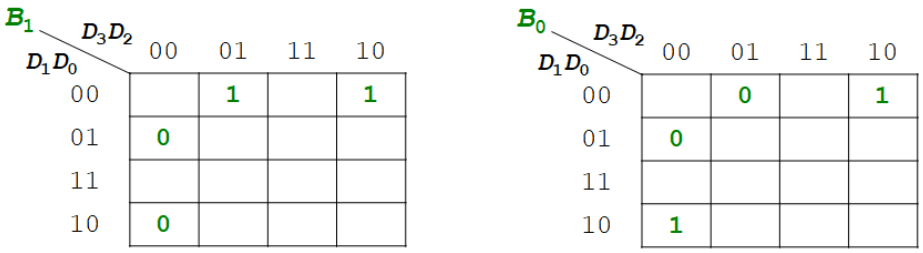


Fig. 7-8. K-map Representation of 4-to-2 Encoder with Empty Cells

The empty cells will be full with *X* (don't care) notation, because we don't care the outputs B_1 and B_0 (either 0 or 1) when the other input combinations of D_3 D_2 D_1 D_0 are fed into the encoder. It seems that more than two buttons pressed simultaneously.

The K-map simplifies the Boolean equation of B_1 and B_0, as shown in Fig. 7-9. We can get the simplified equation as follows:

- $B_1 = D_3 + D_2$
- $B_0 = D_3 + D_1$

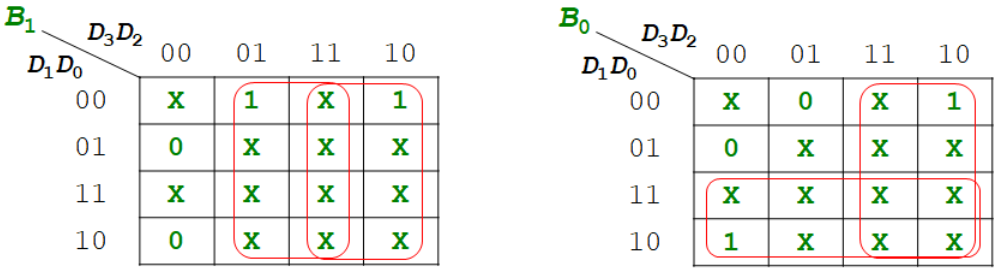


Fig. 7-9. K-map Representation of 4-to-2 Encoder with X (don't care) Notation

Let's design logic gates for the encoder using the above figure:

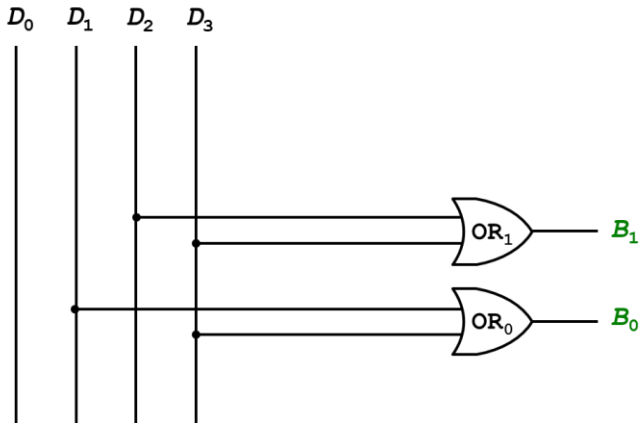


Fig. 7-10. Designed Encoder with logic gates

4-to-2 encoder has four inputs, i.e. D_3, D_2, D_1, D_0, and two outputs (binary code), i.e. B_1 and B_0. There are two OR gates in the block.

- OR_1 gate has two inputs, D_2 and D_3, and produces the output B_1.
- OR_0 gate has two inputs, D_1 and D_0, and produces the output B_0.

7.2 Sequential Circuits

So far, we take a look at the combinational circuit, in which the output is independent of time and only relies on the current input at that particular instant. On the other hand, the sequential circuit is the type of circuit where output not only relies on the current input but also depends on the previous output. The sequential circuit consists of a combinational circuit and storage elements. The previous input/output values are stored in storage elements.

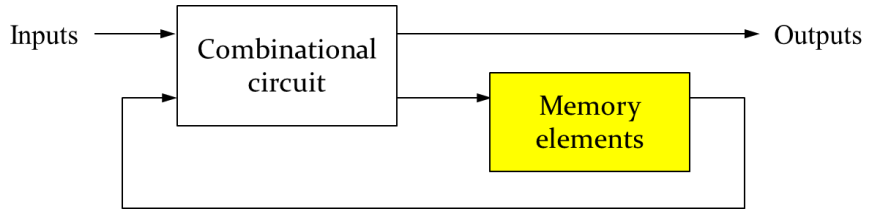


Fig. 7-11. Sequential Circuit

As shown in the above figure, the inputs directly are fed into the combinational circuit block. The combinational circuit produces outputs with current and prior input values. Some states of the combinational circuit are stored in memory elements, e.g. Flip-flops, which will be used as the prior input values. Storage elements maintain a binary state indefinitely as long as power is delivered to the circuit. There are two types of the storage elements: 1) Latch – operated with signal levels, and 2) Flip-Flop – controlled by a clock transition.

SR Latch

SR latch is the most fundamental building block using static gates, where *S* and *R* stand for set and reset. SR latch can be designed with two NOR gates. The inputs *S* and *R* are fed into each NOR gate, and the output of one NOR gate recursively is fed into the input of the other NOR gate, as shown below:

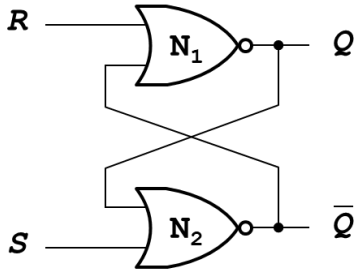


Fig. 7-12. SR Latch

The outputs Q and $\bar{Q}$ represent the value of the stored state and its complement, respectively. In SR latch, there are four possible input cases:

- Case 1: $S = 1$, $R = 0$

Let's look at the case when the inputs $S = 1$ and $R = 0$. The '1' bit is the dominant input of the NOR gate. Since one of the inputs, $S = 1$, is TRUE, the NOR gate (N_2) produces the output FALSE which is fed into one of the inputs in the other NOR gate (N_1). Both two inputs of NOR gate N_1 are FALSE and the NOR gate N_1 produces the output TRUE.

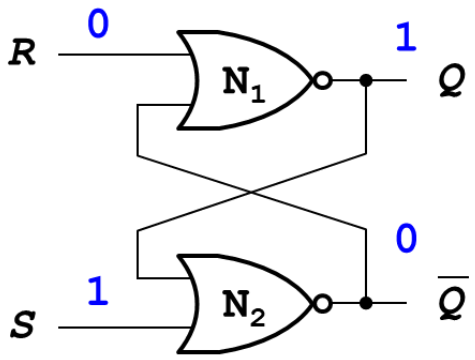


Fig. 7-13. SR Latch When the inputs S = 1 and R = 0

- Case 2: $S = 0$, $R = 1$

Let's look at the case when the inputs $S = 0$ and $R = 1$. The '1' bit is the dominant input of the NOR gate. Since one of the inputs, $R = 1$, is TRUE, the NOR gate (N_1) produces the output FALSE which is fed into one of the inputs in the other NOR gate (N_2). Both two inputs of NOR gate N_2 are FALSE and the NOR gate N_2 produces the output TRUE.

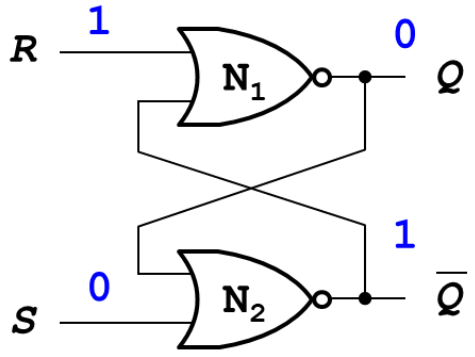


Fig. 7-14. SR Latch When the inputs S = 0 and R = 1

- Case 3: *S* = 0, *R* = 0

Let's look at the case when both inputs *S* and *R* are FALSE. Since both of inputs are FALSE, we need to consider the case whether the previous output Q_{prev} is FALSE or TRUE. In the former case when the previous output Q_{prev} is FALSE, both inputs of the NOR gate N_2 are FALSE and the gate produces the output TRUE. Both inputs of the other NOR gate N_1 has TRUE and FALSE, the gate produces the output FALSE. In the latter case when the previous output Q_{prev} is TRUE, both inputs of the NOR gate N_1 are FALSE and the gate produces the output TRUE. Both inputs of the other NOR gate N_2 has TRUE and FALSE, the gate produces the output FALSE.

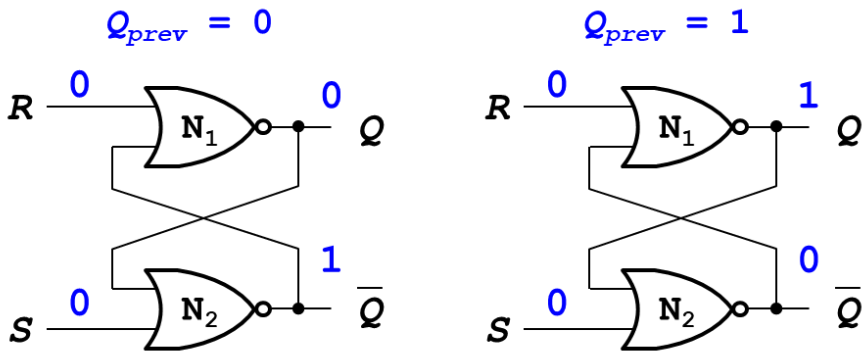


Fig. 7-15. SR Latch When the inputs S = 0 and R = 0

In summary, if the inputs *S* = 0, *R* = 0, and Q_{prev} = 0, then the output Q = 0. If the inputs *S* = 0, *R* = 0, and Q_{prev} = 1, then the output Q = 1. In this case, the latch memorizes the previous state.

- Case 4: *S* = 1, *R* = 1

Let's look at the case when both inputs *S* and *R* are TRUE. Both NOR gates have a dominant input '1' and both outputs Q and $\bar{Q}$ are equal to 0. This is an invalid state. The values of Q and $\bar{Q}$ should be different. We should avoid this state.

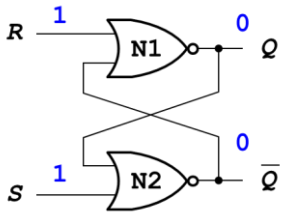


Fig. 7-16. SR Latch When the inputs S = 1 and R = 1

SR latch stores one bit of state (Q). The following figure shows the SR latch symbol.

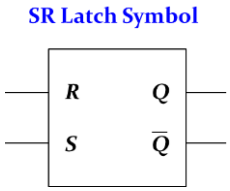
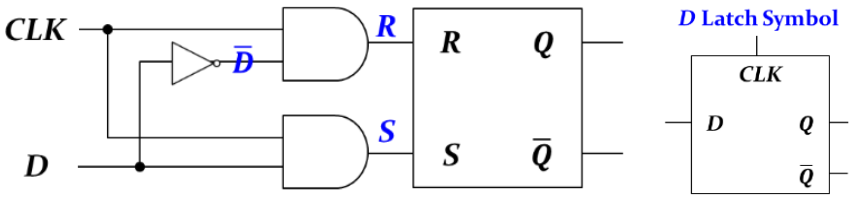


Fig. 7-17. SR Latch Symbol

D Latch

SR latch has an invalid state. We must do something to avoid the invalid state. D latch allows us to avoid this invalid state, where the D latch has two inputs *CLK* and *D*. The *CLK* input controls when the output changes, and the data input *D* controls what the output changes to. The function of D latch is as follows:

- When *CLK* = 1, *D* passes through to Q: called it a state of "transparent"
- When *CLK* = 0, Q holds its previous value: called it a state of "opaque"



(a) Internal Circuit (b) Symbol

Fig. 7-18. D Latch, (a) Internal circuit and (b) Symbol

As shown in the above figure, D Latch Internal Circuit consists of NOT gate, two AND gates, and SR latch. There are two inputs, *CLK* and *D*, and two outputs Q and $\overline{Q}$. The inputs, *CLK* and $\overline{D}$, are fed into one AND gate, and the gate produces the internal value *R*. On the other hand, the inputs, *CLK* and *D,* are fed into the other AND gate, and the gate produces the internal value *S*. The following table summarizes the internal states of the D latch.

Table 7-4. Internal States of D Latch

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

- If *CLK* = 0, both of AND gates produce "0". That means two internal inputs *S* and *R* are "0". The latch produces Q_{prev} and $\overline{Q}_{prev}$, meaning that the current output Q is equal to Q_{prev}.

- If *CLK* = 1 and *D* = 0, the internal input *S* has a value of "0" and the internal input *R* has a value of "1". The latch produces Q = 0 and $\bar{Q}$ = 1.
- If *CLK* = 1 and *D* = 1, the internal input S has a value of "1" and the internal input *R* has a value of "0". The latch produces Q = 1 and $\bar{Q}$ = 0.

The way to respond to the clock signal is slightly different in Latch and Flip-Flop. The latch updates its state when the clock level is positive, as shown in the following figure:

Fig. 7-19. Latch Respond to Positive Level

On the other hand, the flip-flop updates its state when the clock level is in a transitional state, i.e. edge triggered, as shown in the following figure:

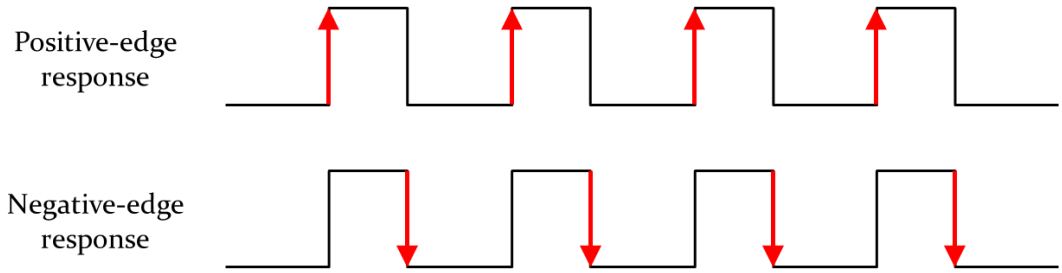


Fig. 7-20. Flip-Flop Respond to Positive or negative-edge in the Clock Cycle

That means that the flip-flop changes the state when the clock level is changed from low to high, referred to as a positive-edge response, or from high to low, referred to as a negative-edge response.

D Flip-Flop

The D flip-flop is created by connecting two gated D latches serially, and inverting the CLK input to one of them. There are two inputs, *CLK* and *D*, and it produces the output values Q and $\bar{Q}$. The following figure shows the symbols of D flip-flop.

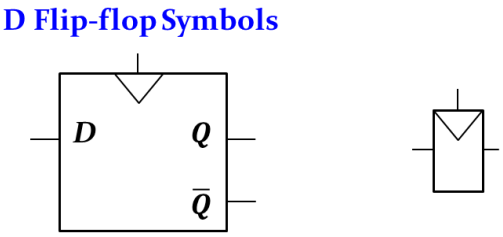


Fig. 7-21. D Flip-Flop Symbols

In D flip-flop, D (data) passes through to Q when *CLK* rises from 0 to 1 (or from 1 to 0); otherwise, Q holds its previous value. Q value changes only on rising edge of *CLK* (from 0 to 1), called edge-triggered, which was represented in the inverted triangle of the above figure.

The internal circuit of D Flip-flop composes of two latches (L_1: Master, and L_2: Slave) and NOT gate, as shown in the following figure:

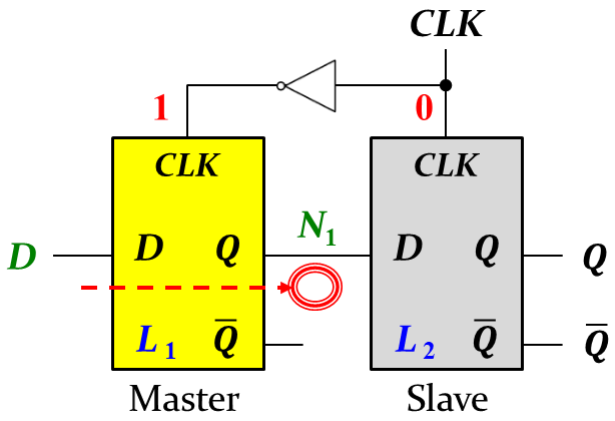


Fig. 7-22. D Flip-Flop Internal Circuit - Master Enabled

CLK value is directly fed into L_2, but connected to L_1 after flipping *CLK* with NOT gate. The input *D* is directly fed into L_1. The output Q of L_1 is directly connected to the internal input N_1 of L_2. When the *CLK* value is zero, L_1 Latch is enabled (transparent) and the input *D* value can pass through L_1. On the other hand, L_2 Latch is disabled (opaque) and the internal input N_1 cannot pass through L_2.

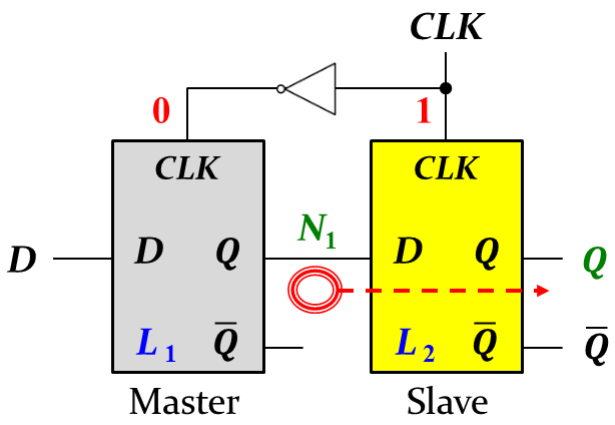


Fig. 7-23. D Flip-Flop Internal Circuit - Slave Enabled

When the *CLK* value is one, L_2 Latch (Slave) is enabled (transparent) and the internal input N_1 can pass through L_2. On the other hand, L_1 Latch (Master) is disabled (opaque) and the input D cannot pass through L_1, as shown in the above figure.

Thus, when the CLK value rises from 0 to 1 (on the edge of the clock), the D value passes through *Q* value.

Since D flip-flop keeps one-bit information, we can design a 4-bit register with four D flip-flops, as follows:

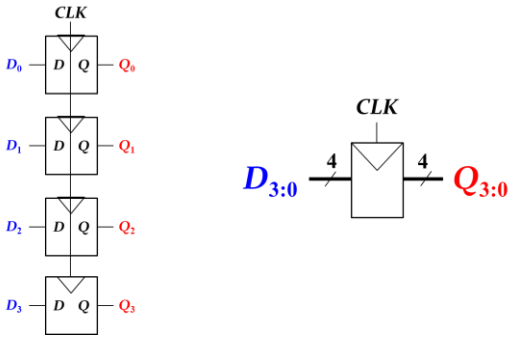


Fig. 7-24. 4-bit Register

- D Flip-Flop$_0$ has the input D_0 and the output Q_0.
- D Flip-Flop$_1$ has the input D_1 and the output Q_1.
- D Flip-Flop$_2$ has the input D_2 and the output Q_2.
- D Flip-Flop$_3$ has the input D_3 and the output Q_3.

Notice that each edge clock CLK of D Flip-Flop is connected to the common CLK individually.

Chapter 8: Basic CPU Organization

In this chapter, we introduce the basic CPU organization and instructions. This module also shows how a CPU is made, what's inside a CPU, how computer memory works, and how a CPU works.

Objectives

By the end of this chapter you should be able to:

- Recognize the history of Intel microprocessors
- Recall how a CPU is made from sand to chip
- List what's inside a CPU
- Demonstrate knowledge of computer memory integrating with a CPU

8.1. Hardware Overview

Typical personal computer systems consist of lots of input/output devices, storage devices and communication interface. The input device includes keyboard, mouse. The output device includes monitor, printer, and speaker. Storage devices include CD-R/RW, DVD, and Hard disk. When you open a desktop computer case, you can see lots of electronic components in the main board. The key components of your main board are CPU and Memory.

History of Intel Processors

The early computers that used vacuum tubes were huge. The ENIAC occupied a whole room. Vacuum also took a long time to warm up and they produce a lot of excess heat and then came transistors. The transistor was invented at Bell Laboratories on December 23, 1947. The following show the history of intel processors (https://www.youtube.com/watch?v=Qu2njWY3Hjk):

Year	Processors	# of Transistors	Clock rate	Memory	Feature size
1971	Intel 4004	2,300	740 KHz	Up to 4,096 bytes	10 microns
1972	Intel 8008	3,500	0.2 to 0.8 MHz	Up to 16 kB	10 microns
1964	Intel 8080	4,500	2 MHz	Up to 64 kB	6 microns
1978	Intel 8086	29,000	5 to 10 MHz	Up to 1 MB	3 microns
1979	Intel 8088	29,000	5 to 10 MHz	Up to 1 MB	3 microns
1982	Intel 80186	55,000	6 to 25 MHz	Up to 1 MB	3 microns
1982	Intel 80286	134,000	6 to 25 MHz	Up to 16 MB	1.5 microns
1985	Intel 80386	275,000	12 to 40 MHz	Up to 4 GB	1.5 microns
1989	Intel 80486	1,180,235	16 to 150 MHz	Up to 4 GB Cache – 8 to 16 kB	1 micron

1993	Intel Pentium 80501	3.1 to 3.3 million	60 to 66 MHz	Up to 4 GB Cache – 8 kB instruction cache, 8 kB cache	0.35 to 0.8 microns
1995	Intel Pentium Pro	5.5 million	150 – 200 MHz	Up to 64 GB L1 Cache – 8 kB instruction cache & 8 kB data cache LS Cache – 512 kB	0.35 to 0.5 microns
1997	Intel Pentium II	7.5 million	233, 266 or 300 MHz	Up to 64 GM L1 Cache – 32 kB L2 Cache – 512 kB	0.35 microns
1999	Intel Pentium II (Dixon)	27.4 million	400 MHz	Up to 64 GB L1 Cache – 32 kB L2 Cache – 256 kB	180 nm
1999	Intel Pentium 3 Katmai	9.5 million	450 to 600 MHz	L1 Cache – 16 kB instruction cache & 16 kB data cache L2 Cache – 512 kB (50% of CPU speed)	250 nm
2001	Intel Pentium 3 Tualatin	45 million	1000 to 1400 MHz	L1 Cache – 16 kB instruction cache 7 16 kB data cache L2 Cache – 256 kB or 512 kB (full speed)	130 nm
2000	Intel Pentium 4 Willamette	42 million	1300 to 2000 MHz	L1 Cache – 20 kB L2 Cache – 256 kB	180 nm
2002	Intel Pentium 4 Northwood	55 million	1600 to 2800 MH	L2 Cache – 512 kB	130 nm
2004	Intel Pentium 4 Prescott	112 million	2400 to 3067 MHz	L2 Cache – 1024 kB	90 nm
2005	Intel Pentium 4 Prescott 2M	169 million	2.8 to 4.00 GHz	L2 Cache – 2 MB	90 nm
2006	Intel Pentium 4 Cedar Mill	184 million	3 to 3.6 GHz	L2 Cache – 2 MB	65 nm

The list of Intel microprocessors can be found in the following link:
https://en.wikipedia.org/wiki/List_of_Intel_microprocessors

How A CPU Is Made

Your CPU made with sand (silicon), UV light, fire (high temperature), and water (cleaning). Intel released all the major steps in a process that normally takes hundreds of stages to complete. See the link to see that Intel shows how a CPU is made: https://www.tomshardware.com/picturestory/514-intel-cpu-processor-core-i7.html

8.2. CPU Organization

What's inside a CPU

Inside every computer is a central processing unit and inside every CPU are small components that carry out all the instructions for every program you run. These components include AND gates, OR gates, NOT gates, Clock, Multiplexer, ALU (arithmetic logic unit), etc. Data bus performs data transfer within a CPU and a computer. As shown in Fig. 8-1, CPU is organized with Program Counter (PC), Instruction Register (IR), Instruction Decoder, Control Unit, Arithmetic Logic Unit (ALU), Registers, and Buses. PC holds the address of the next instruction to be fetched from Memory. IR holds each instruction after it is fetched from Memory. Instruction Decoder decodes and interprets the contents of the IR, and splits a whole instruction into fields for the Control Unit to interpret. Control Unit co-ordinates all activities within the CPU, has connections to all parts of the CPU, and includes a sophisticated timing circuit. ALU carries out arithmetic and logical operations, exemplified with addition, comparison, Boolean AND/OR/NOT operations. Within ALU, input registers hold the input operands and output register holds the result of an ALU operation. Once completing ALU operation, the result is copied from the ALU output register to its final destination.

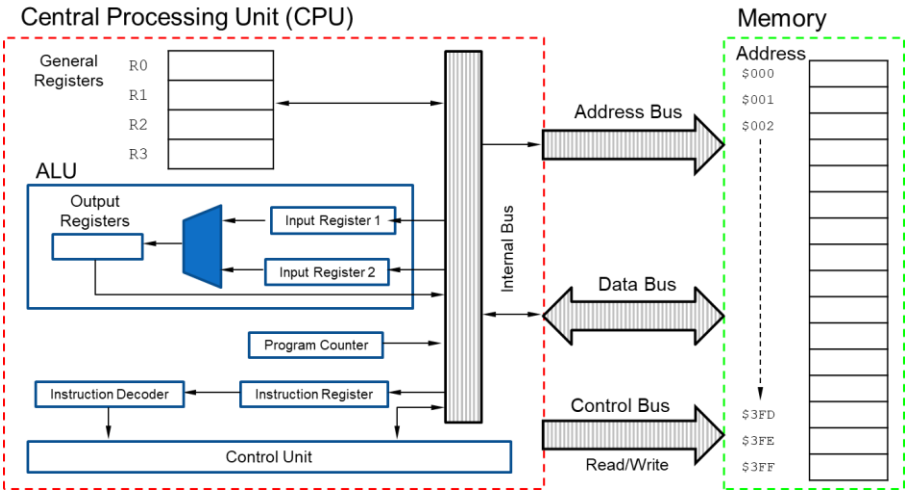


Fig. 8-1. CPU Organization

General-purpose registers are available for the programmer to use in their programs within CPU. Typically, the programmer tries to maximize the use of these registers in order to speed program execution. Busses serve as communication highways for passing information in the computer.

The computer has memory which memorize data in a similar way we remember the past events. The register is the fastest memory which is located within CPU of the computer.

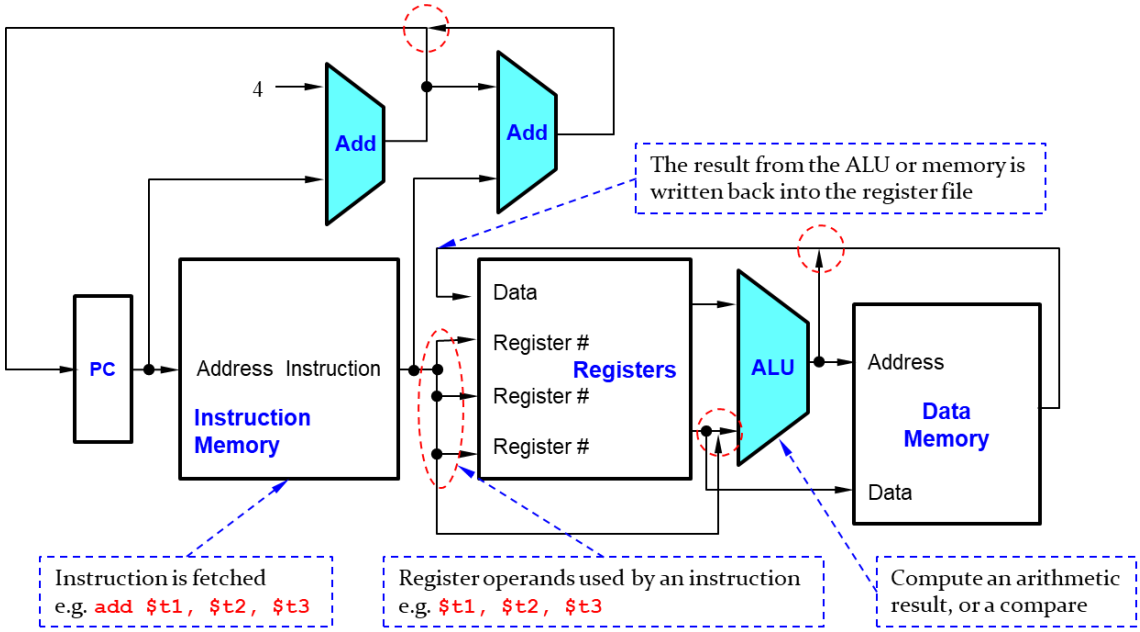


Fig. 8-2. CPU Overview

The above figure shows CPU overview which consists of PC, instruction memory, registers, ALU, and Data memory. PC always holds the address of the next instruction to be fetched from Memory. Instruction, e.g. add $t1, $t2, $t3, is fetched into instruction memory. Register operands are used by an instruction in registers, where $t1 is the first source operand, $t2 is the second source operand, and $t3 is the storage of the result. ALU executes an arithmetic operation, e.g. Sum of $t1 and $t2. The result from the ALU or memory is written back into the register file ($t3). In the figure, ALU results and the output of data memory can't just join wires together. The red dash-dot line can be designed with the multiplexer to put the wires together.

The following figure shows CPU control with multiplexers. The first multiplexer controls what value replaces the PC (PC + 4 or the branch destination address), where the Mux is controlled by the AND gate with the Zero output of ALU and a control signal. The second multiplexer steers the output of the ALU or the output of the data memory. The third one determines whether the second ALU input is from the registers or from the offset field of the instruction (for a load or store).

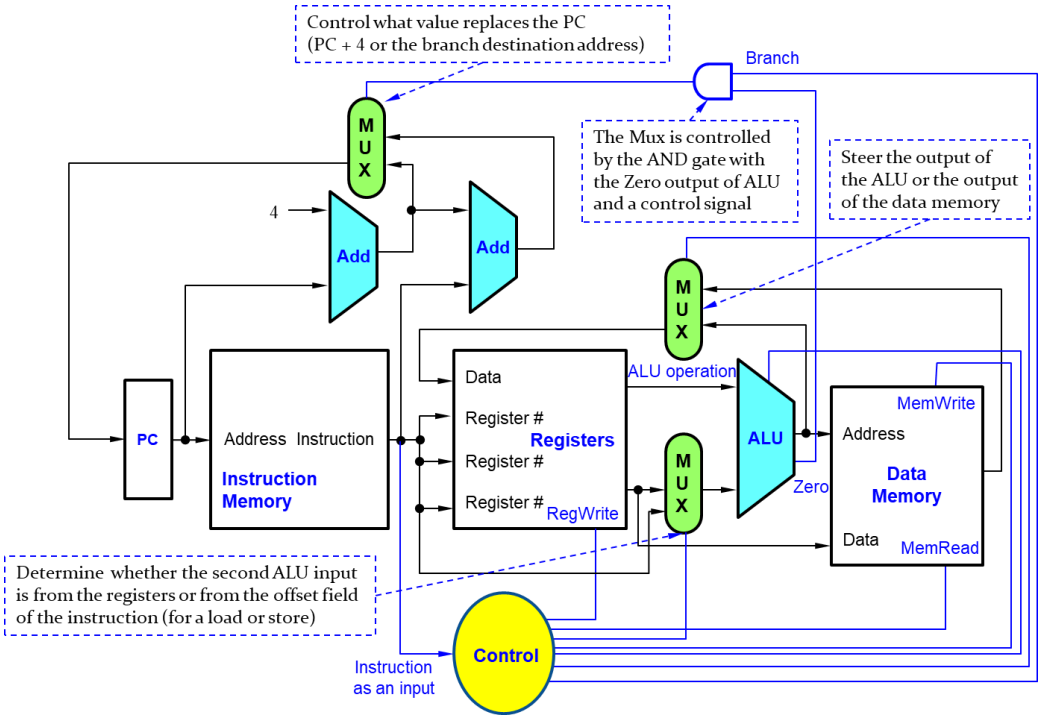


Fig. 8-3. CPU Control with Multiplexer

Chapter 9: Instruction Set Architecture

In this chapter, we introduce the instruction set architecture. The architecture is the programmer's view of a computer, which is defined by instruction set (language) and operand locations (registers and memory). We look at the computer's vocabulary (called the instruction set). Computer instructions indicate both the operation to perform and the operands to use. we look at the operands which come from memory, from registers, or from the instruction itself. This chapter also shows how to interpret assembly language into machine language.

Objectives

By the end of this chapter you should be able to:

- Recognize Assembly language and machine language
- Demonstrate knowledge of MIPS Assembly architecture
- Differentiate the operation to perform and the operands to use
- Summarize features of word-addressable memory
- Deconstruct different types of instructs, i.e. R-type, I-type and J-type
- Translate assembly language into machine code

9.1. Instructions

An instruction is a single operation of a processor defined by the processor instruction set. The size of length of an instruction depends on the processor. The instruction can be written in human-readable formats or computer-readable formats. Assembly language is the human-readable format of instructions, whereas machine language is the computer-readable format (1's and 0's).

Once you've learned one architecture, it's easy to learn others. MIPS (Microprocessor without Interlocked Pipelined Stages) architecture was developed by John Hennessy and his colleagues at Stanford in the 1980's, and used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco. Underlying architecture design principles, as articulated by Hennessy and Patterson are as follows:

- Simplicity favors regularity
- Make the common case fast
- Smaller is faster
- Good design demands good compromises

Let's look at the following instructions for the addition:

High-level Code	MIPS assembly code
`a = b + c`	`add  a, b, c`

where `add` is a mnemonic which indicates operation to perform. `b` and `c` are source operands on which the operation is performed. `a` is a destination operation to which the result is written.

The next instructions show the subtractions in High-Level Code and MIPS assembly code.

High-level Code	MIPS assembly code
`a = b - c`	`sub  a, b, c`

The subtraction is similar to addition, only mnemonic changes.

As shown in the above instructions, MIPS assembly code shows consistent instruction format, has the same number of operands (two source operands and one destination operand), and is easy to encode and handle in hardware. This is the first design principle: ***Simplicity favors regularity***.

More complex code is handled by multiple MIPS instructions. For example, the following High-Level Code can be interpreted into multiple MIPS instructions, as follows:

High-level Code	MIPS assembly code
`a = b + c - d`	`add  t, b, c  # t = b + c` `sub  a, t, d  # a = t - d`

MIPS assembly code includes only simple, commonly used instructions. With this feature, hardware to decode and execute instructions can be simple, small, and fast. More complex instructions (that are less common) are performed using multiple simple instructions. This is the second design principle: ***Make the common case fast***.

Operands

An instruction operates on operands. The instructions need a physical location from which to retrieve the binary data. Operand can be stored in the following locations:

- **Registers** that is located in CPU. The instruction in registers can be accessed quickly.
- **Memory** is located outside of CPU in the computer. It provides large capacity but operate slowly.
- **Constant** (also called immediate) expressions indicate inline values of the instruction.

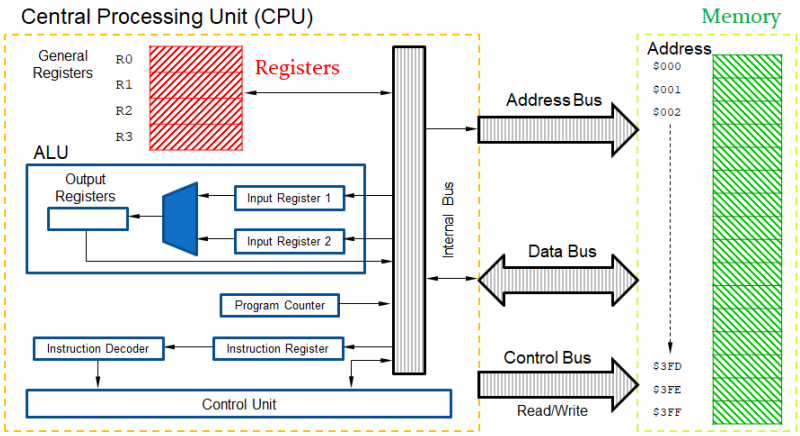


Fig. 9-1. CPU Organization

As shown in the above figure, CPU is organized with Program Counter (PC), Instruction Register (IR), Instruction Decoder, Control Unit, Arithmetic Logic Unit (ALU), general registers, and buses. MIPS has 32 32-bit general registers, which is called the register set or register file. The fewer the registers, the faster they can be accessed. This is related to the third design principle: ***Smaller is faster***. In terms of volume, the registers are much smaller than memory, and located within CPU. That's why the registers are faster than memory. MIPS is also called "32-bit architecture" because it operates on 32-bit data.

The operands are positioned on registers. Typically, the register comes with the symbol `$` before their name. For example, we read the symbol `$0` in "register zero", "dollar zero". The registers are used for specific purposes. The register `$0` always holds the constant value 0. The saved registers, `$s0 – $s7`, are used to hold variables. The temporary registers, `$t0 – $t9`, are used to hold intermediate values during a larger computation process. The following table show the register usage in MIPS assembly system.

Table 9-1. Register Usage

Register number	Register name	Usage
0	zero	Always zero
1	$at	Reserved for the assembler
2 – 3	$v0 - $v1	Function return value
4 – 7	$a0 - $a3	The first four parameters passed to a procedure. (Function arguments)
8 – 15	$t0 - $t7	Temporary variables. Can be overwritten by callee
16 - 23	$s0 - $s7	Saved variables. Must be saved/restored by callee
24 - 25	$t8 - $t9	Temporary variables. Can be overwritten by callee
26 - 27	$k0 - $k1	Reserved for kernel usage (operating system)
28	$gp	Global pointer for static data (pointer to global area)
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Function return address

Now, we can interpret the instructions with registers. The following High-Level codes can be converted to MIPS assembly codes with designated register names:

Example 1)

High-Level code	MIPS assembly code

`a = b + c;`	`# $s0=a, $s1=b, $s2=c`
	
	`add $s0, $s1, $s2`

Example 2)

High-Level code	MIPS assembly code

`a = b + c - d;`	`# $s0=a, $s1=b, $s2=c, $s3=d`
	
	`sub $t0, $s2, $s3 // t = c - d`
	`add $s0, $s1, $t0 // a = b + t`

Word-addressable Memory

When we execute instructions, there are too much data to fit in only 32 registers. The memory has a lot of capacities to store data. The register file is small and fast, whereas memory is large and slow, because the memory is located outside the CPU. Only commonly used variables are kept in registers. The rest of them are kept in memory for a future processing. As shown in the below, each 32-bit data word has a unique 32-bit address. This is called word-addressable memory. Both the 32-bit word address and the 32-bit data value are written in hexadecimal.

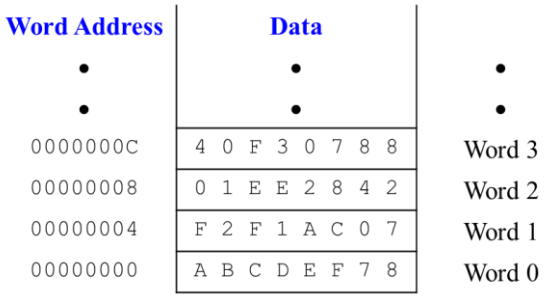


Fig. 9-2. Word-addressable Memory

Exercises

Translate the following high-level code into assembly language. Assume variables `a – c` are held in registers `$s0 – $s2` and `f – j` are in `$s3-$s7`.

```
a = b – c;
```

```
f = (g + h) – (i + j);
```

Answer)

```
# MIPS assembly code
# $s0=a, $s1=b, $s2=c, $s3=f, $s4=g, $s5=h, $s6=i, $s7=j
      sub $s0, $s1, $s2     # a = b - c
      add $t0, $s4, $s5     # $t0 = g + h
      add $t1, $s6, $s7     # $t1 = i + j
      sub $s3, $t0, $t1     # f = (g + h) - (i + j)
```

9.2. Machine Languages

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called machine language. The small number of formats allows some regularity among all the types, and thus simpler hardware, while it can also accommodate different instructions needs.

MIPS Assembly language uses 32-bit instructions that makes the compromise of defining three instruction formats: R-type, I-type, and J-type. This is the fourth design principle: ***Good design demands good compromises***. In MIPS assembly language, multiple instruction formats allow flexibility. For example, `add` and `sub` use 3 register operands, whereas `lw` and `sw` use 2 register operands and a constant. The number of instruction formats kept small to adhere to design principles 1 and 3.

R-type Instruction Format

The name R-type is short for register-type. The following figure shows the R-type instruction fields.

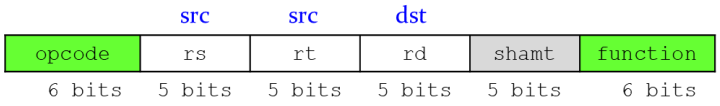


Fig. 9-3. R-type Instruction fields

- `opcode`: operation code (zero value for all R-type)
- `rs`: first source register number
- `rt`: second source register number
- `rd`: destination register number
- `shamt`: shift amount (00000 for now)
- `function`: function code (extends opcode)

Now, let's look at how the computer can interpret a MIPS instruction, `add $s0, $s1, $s2`, into a machine language. In the R-type instruction, the operation code field is all zero. The function field extends the operation code value that define the `add` mnemonic in the function field. The `rs` and `rt` fields are filled with the two source operands, `$s1` and `$s2`. The `rd` field is filled with the destination operand `$s0`. For the `add` mnemonic, the shift amount is unused for now. This field is filled with all 0's, as shown below.

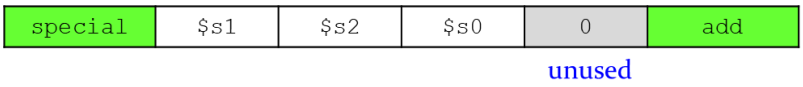


Fig. 9-4. R-type Instruction field with `add $s0, $s1, $s2`

Each instruction set architecture has its own function definition in the following table.

Table 9-2. References for Operation code and Function field

Op (31:26)								
28-26 / 31-29	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0 (000)	R-format	Bltz/ gez	jump	Jump & link	Branch eq	Branch ne	blez	bgtz
1 (001)	add Imm.	addiu	Set less than imm.	Set less than imm. unsigned	andi	ori	xori	Load upper imm.
2 (010)	. . .							
Op (31:26)=000000 (R-format), function(5:0)								
2-0 / 5-3	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0 (000)	Shift left log		Shift right log	sra	sllv		srlv	srav
1 (001)	Jump reg	jalr			syscall	break		
2 (010)	mfhi	mthi	mflo	mflo				
3 (011)	mult	multu	div	divu				
4 (100)	add	addu	subtract	subu	and	or	xor	nor
5 (101)	. . .							

The register number for the register usage is defined in Table 9-1. We can define the register numbers, such as the decimal value 17 for `$s1`, the decimal value 18 for `$s2`, and the decimal value 16 for `$s0`. The R-type instruction field of the `add` instruction is filled with all those decimal values, as shown below:

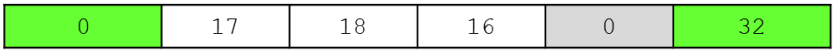


Fig. 9-5. R-type Instruction field with Decimal Representation

The decimal representation is expressed with the binary number representation, i.e. machine code as shown below:

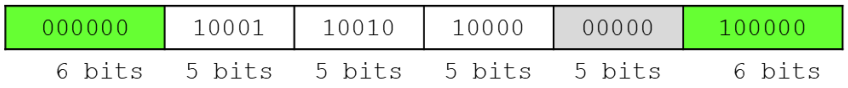


Fig. 9-6. R-type Instruction field with Binary Number Representation

We can express this binary number in the hexadecimal representation: 02328020_{16}.

Let's look at another example with the sub instruction, `sub $t0, $t3, $t5`, and interpret it into a machine language. Since the sub instruction is one of R-type instructions as shown in Table 9-2, the operation code field is all zero. The function field extends the operation code value that define the `sub` mnemonic in the function field. The `rs` and `rt` fields are filled with the two source operands, `$t3` and `$t5`. The `rd` field is filled with the destination operand `$t0`. For the `sub` mnemonic, the shift amount is unused. This field is filled with all 0's, as shown below.

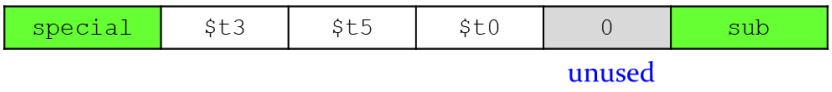


Fig. 9-7. R-type Instruction field with `sub $t0, $t3, $t5`

As shown with the register number in Table 9-1. We can define the register numbers, such as the decimal value 11 for `$t3`, the decimal value 13 for `$t5`, and the decimal value 8 for `$t0`. The R-type instruction field of the `sub` instruction is filled with all those decimal values, as shown below:

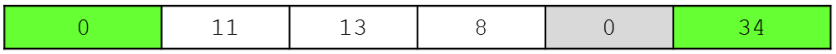


Fig. 9-8. R-type Instruction field with Decimal Representation

The decimal representation is expressed with the binary number representation, i.e. machine code as shown below:

Fig. 9-9. R-type Instruction field with Binary Number Representation

We can express this binary number in the hexadecimal representation: $016D4022_{16}$.

I-type Instruction Format

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. Any instruction that comes with a constant (off) value or memory address can be accommodated with the I-type instruction format. That means the I-type instruction format can be used for the load/store word instruction and the immediate arithmetic instructions which include a constant value. The following figure shows the I-type instruction fields.

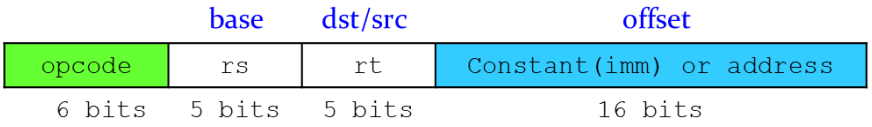


Fig. 9-10. I-type Instruction Fields

- The first three fields, `op`, `rs`, and `rt`, are like those of R-type instructions.
- `rs` and `imm` are always used as source operands.
- `rt` is used as a destination (`addi` and `lw`) or another source (`sw`)
- Constant (`imm`): -2^{15} to $2^{15}-1$
- Address: offset added to base address in `rs`

Now, let's look at how the computer can interpret the following I-type instructions into machine languages.

- Assembly Code

```
addi   rt,   rs, imm      →    addi   $s0, $s1, 5
addi   rt,   rs, imm      →    addi   $t0, $s3, -12
lw     rt,   imm(rs)      →    lw     $t2, 32($0)
sw     rt,   imm(rs)      →    sw     $s1, 4($t1)
```

The `addi` is a I-type instruction, where `rt` is used for the destination register address, `rs` is the base address, and `imm` is the 16-bit immediate value. The `opcode` field of the `addi` is the decimal value 8 (00100) defined in Table 9-2. Both the load word (`lw`) and the store word (`sw`) instructions are I-type instructions. The data positioned in the memory can be loaded to the (destination) register with the load word (`lw`) instruction.

The `opcode` field of the `lw` instruction is the decimal value 35. For the `lw` instruction, the memory address is calculated with the sum of the base register address and the offset value. In the above example, the base register address is `$0` and the offset value is 32. The calculated memory address is 32. After finding the data that is located in the memory (memory address: 32), the data is loaded into the destination register address (`$t2`).

The `opcode` field of the `sw` instruction is the decimal value 43. The data positioned in the register file can be stored to the memory with the store word (`sw`) instruction. For the `sw` instruction, the memory address is calculated in the same way to the `lw` instruction. In the above example, the base register address `$t1` is and the offset value is 4. The memory address is the sum of the value in `$t1` and the offset value, `$t1 + 4`. The value located in the register `$s1` is stored in the memory address `$t1 + 4`.

The following figure shows the field values of the above examples:

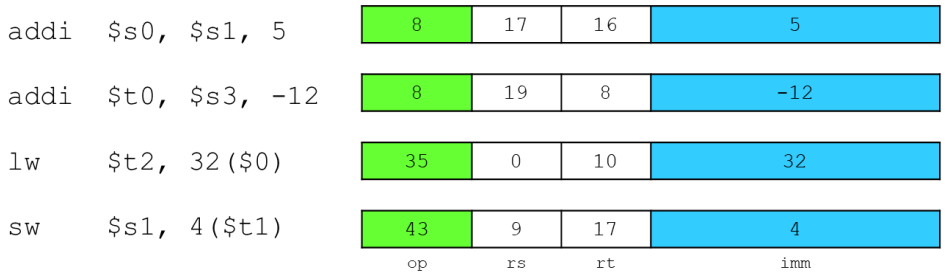


Fig. 9-11. I-type Instruction fields with Decimal Representation

The decimal representations are expressed with binary number representations, i.e. machine code, as shown below:

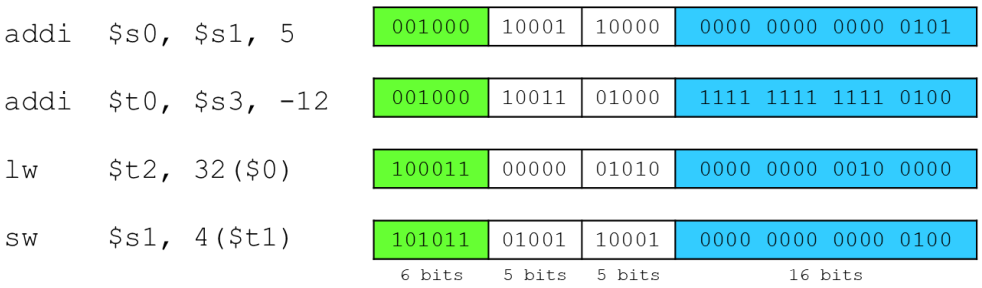


Fig. 9-12. I-type Instruction field with Binary Number Representation

J-type Instructions

The J-type instruction is used to jump the target of the address. The following figure shows the J-type instruction field.

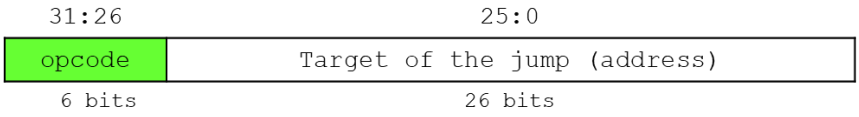


Fig. 9-13. J-type Instruction Fields

Jump instruction uses word address and updates PC with concatenation of the following values (total of 32 bits):

- Top 4 bits of old PC (4 bits)
- 26-bit jump address (26 bits)
- 00 (2 bits)

The following example codes show how the Jump instruction is used in the assembly code.

```
    addi  $s0, $0, 4              # $s0 = 4
```

```
      addi  $s1, $0, 1            # $s1 = 1
      j          target           # jump to target
      addi  $s1, $s1, 1           # not executed
      sub   $s1, $s1, $s0         # not executed
target:
      add   $s1, $s1, $s0         # $s1 = 1 + 4 = 5
```

The first two `addi` instructions execute the immediate arithmetic operations, where the destination register address `$s0` holds the sum of `$0` and 4 (`$s0=4`), and the destination register address `$s1` holds the sum of `$0` and 1 (`$s1=1`). The jump instruction jumps the target of the address and then executes the last `add` instruction. The destination register address `$s1` holds the sum of two register values 1 and 4.

Instruction Fetch and PC

Program Counter (PC) is a 32-bit register which holds the address of the next instruction to be fetched from the memory. PC value is increased by 4 for the next instruction, as shown in the following figure. The instruction memory fetches the instruction from the memory, and forward the instruction to the next step.

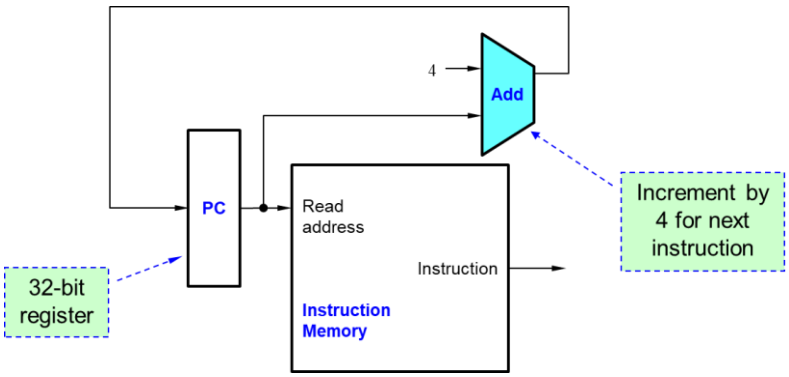


Fig. 9-14. Instruction Fetch and PC Increment

Exercises

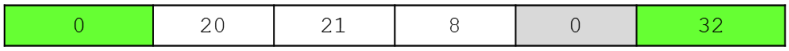
1) Translate the following assembly language into machine language.

```
add $t0, $s4, $s5   // $t0->8, $s4->20, $s5->21
```

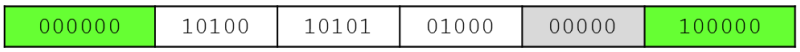
opcode	$s4	$s5	$t0	shamt	add
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Answer)

- Decimal representation (field values):



- Binary number representation (Machine Code):



00000010100101010100000000100000$_2$ = 02954020$_{16}$

2) Translate the following I-type instruction into machine code.

```
                    // lw opcode value: 35
lw $s3, -24($s4)    // $s3 and $s4 are #19 and #20.
```

Answer)

- Decimal representation (field values):

- Binary number representation (Machine Code):

10001110100**10011**1111111111101000$_2$ = 8E93FFE8$_{16}$

3) Convert the following machine language into MIPS assembly language.
0x01094020

Answer)

0000 0001 0000 1001 0100 0000 0010 0000 (32 bits)

0000 0001 0000 1001 0100 0000 00**10 0000**

0 8 9 8 0 32

Opcode src src dst shmt func

add $t0 $t0 $t1

Chapter 10: Assembly

In this chapter, we introduce the assembly language which is the human-readable representation of the computer's native language. We also introduce simple arithmetic instructions and show how these operations are written in Assembly language. We then define the MIPS instruction operands: registers, memory, and constants.

Objectives

By the end of this chapter you should be able to:

- Demonstrate knowledge of word-addressable and memory
- Differentiate word-addressable and byte-addressable memories
- Summarize features of the stored program
- Recall different types of machine code
- Demonstrate knowledge of logic operations
- Carry out conditional operations with arithmetic instructions

10.1. Assembly Languages

Read Word-Addressable Memory

An instruction operates on operands. In MIPS assembly architecture, there are only 32 registers which are not good enough to hold all the data. We can store more data in memory. The register file is small and fast, whereas memory is large and slow. The instructions are stored in memory. Only commonly used variables are kept in registers.

When we store the data in memory, each 32-bit data word has a unique address, as shown below:

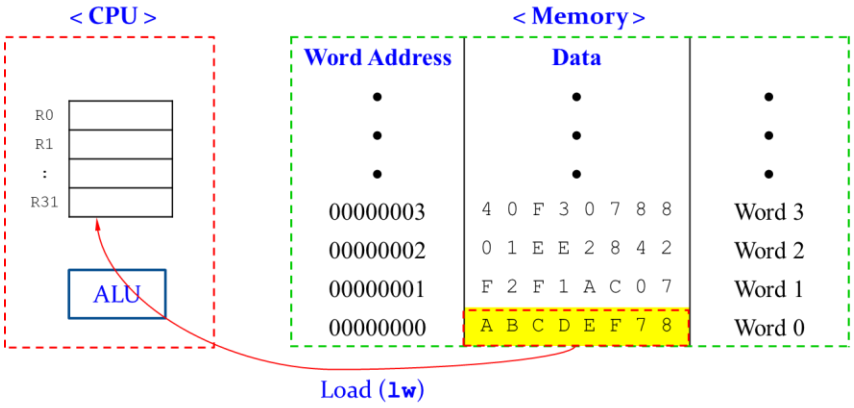


Fig. 10-1. Word-Addressable Memory

We can read the data in memory and load it to one register using the load word (`lw`) instruction, as indicated with the red arrow in the above figure.

The following instruction exemplifies the format of the `lw` instruction.

- `lw $s0, 5(t1)`

where `$s0` is the register address that will hold the data after loading the data from memory. The memory address is calculated by adding the base address (`$t1`) to the offset value (`5`), i.e. `$t1 + 5`.

After executing the `lw` instruction, the register address `$s0` holds the value at the memory address `$t1 + 5`. Any register can be used as the base address.

Let's read a word of data at the memory address `1` into the register address `$s3`. If the `$0` is used for the base address, the memory address is calculated by adding the zero value (`$0`) to the offset (`1`), as shown below:

- `lw $s3, 1($0) # read memory word 1 into $s3`

As a result of this instruction, the register address `$s3` holds the data `0xF2F1AC07` as shown in the figure below:

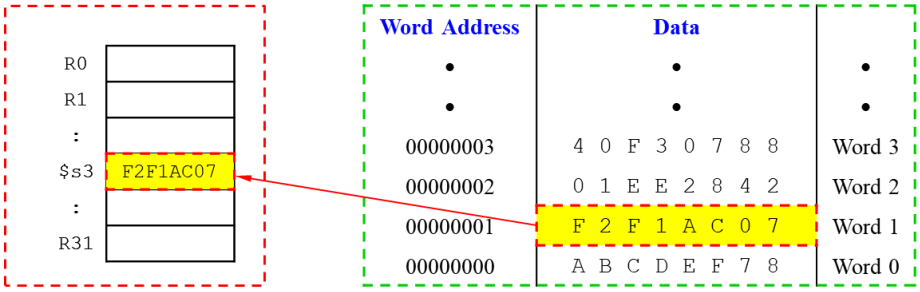


Fig. 10-2. Load data from Memory to Register with lw instruction

The above figure shows the register address `$s3` holds the word of data at the memory address `1` after executing the instruction `lw $s3, 1($0)`.

Write Word-Addressable Memory

We can store the data located at a register into memory with the store word (`sw`) instruction.

The following instruction exemplifies the format of the `sw` instruction.

- `sw $t4, 0x7($0)`

where `$t4` is the register address. We can store the data located at the register address `$t4` into the memory, where the memory address is calculated by adding the base address (`$0`) to the offset value (7), i.e. `$0 + 7`.

Let's store the value of the register address `$t4` into the memory address `7`. If the `$0` is used for the base address, the memory address is calculated by add the zero value (`$0`) to the offset (`7`), as shown below:

- `sw $t4, 0x7($0) # write the value of $t4 into memory word 7`

where the offset can be written in decimal (default) or hexadecimal. As a result of this instruction, the memory address `7` holds the value `0x6A049C04` of the register address `$t4`, as shown below:

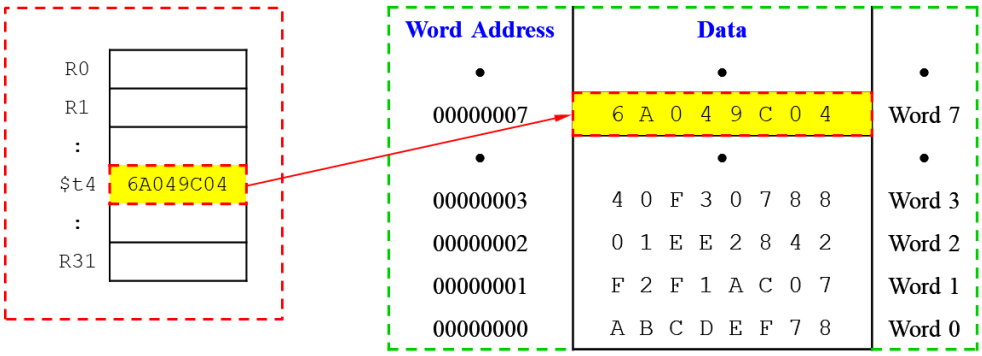


Fig. 10-3. Store the value of Register into Memory with sw instruction

Byte-Addressable Memory

Each data byte has unique address. We can load/store words or single bytes with load byte (`lb`) and store byte (`sb`). Since we are using 32-bit word that is 4 bytes, the word address is increased by 4. That means the address of a memory word must now be multiplied by 4. For example, the address of memory word 2 is 2 × 4 = 8 and the address of memory word 10 is 10 × 4 = 40 (`0x28`). Keep in mind that MIPS is byte-addressed, not word-addressed.

Power of the Stored Program

Both 32-bit instructions and data are stored in memory. The only difference between two applications is the sequence of instructions. We do not require large amounts of time and effort to reconfigure or rewire hardware to run a new program. We only require writing the new program to memory. When executing program, the processor fetches (reads) instructions from memory to instruction register in sequence and performs the specific operation. Even large and complex programs are simplified to a series of memory reads and instruction executions.

For example, the assembly code and the corresponding machine code are given as below:

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

The whole machine codes are stored in memory, as shown in Fig. 10-4. In this example, the first instruction is stored at the memory address `0x00400000`. The next instruction is stored at the memory address `0x00400004`, etc. Note that the memory address is increased by 4 because it is a byte address (4 × 8 = 32). The program counter in the processor keeps track of current instruction. Each instruction is executed in sequence.

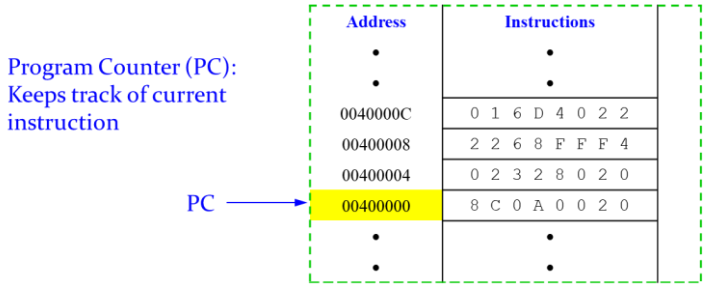


Fig. 10-4. Stored Program

The processor starts to interpreting machine code. The first six bits (`opcode`) tell how parse the rest of them. If `opcode` is all `0`'s, the function field tells the arithmetic/logic operation; otherwise it tells operation.

Exercises

1) The data values in the memory address are drawn below. MIPS Assembly code is given as follows:

```
lw $s0, 0($0)
lw $s1, 8($0)
lw $s2, 0xC($0)
```

What are the register values in `$s0`, `$s1`, and `$s2`?

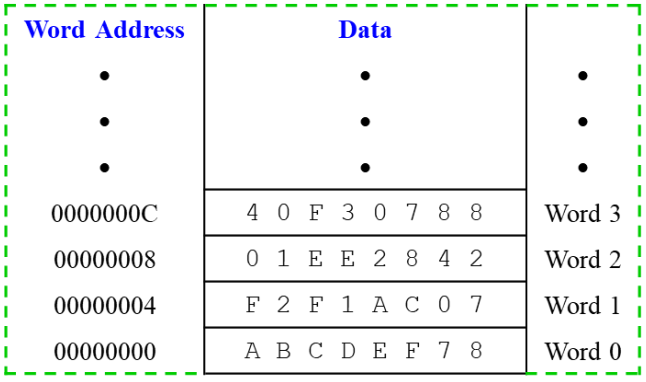


Fig. 10-5. Data Values in Memory Address

Answer)

```
lw $s0, 0($0)       # read data word 0(0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2(0x01EE2842) into $s1
lw $s2, 0xC($0)     # read data word 3(0x40F30788) into $s2
```

2) Translate the following machine language code into assembly language.
`0x2237FFF1`

Answer) the machine language code `0x2237FFF1` is expanded into as below:

```
  2    2    3    7    F    F    F    1       (hexadecimal)

0010 0010 0011 0111 1111 1111 1111 0001      (binary, 32 bits)

0010 0010 0011 0111 1111 1111 1111 0001

   8   17   23          -15

Opcode  rs    rt          imm
```

The corresponding assembly code: `addi $s7, $s1, -15`.

3) What is the assembly language statement corresponding to this machine instruction?
`0x00AF8020`

Answer)

Convert hexadecimal to binary

0	0	A	F	8	0	2	0
0000	0000	1010	1111	1000	0000	0010	0000

Referring to the Table 9-2.

```
 op    rs   rt   rd  shamt  funct

000000  00101  01111  10000  00000  100000

→ add $s0, $a1, $t7
```

10.2. Logic Operations

MIPS instructions execute bitwise manipulation as shown below:

Table 10-1. Instructions for Bitwise Manipulation

Logic operations	C operators	Java operators	MIPS instruction
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

The instructions in the above table are useful for extracting and inserting groups of bits in a word.

Shift Operations

The shift operation is a R-type instruction and the field value is shown below:

Fig. 10-6. The field Value of Shift Operations

where the field `shamt` tells how many positions to shift.

There are two logic shifts, i.e., the shift left logic (`sll`) and the shift right logic (`srl`). The `sll` shifts the bits left and fills the empty bits with 0 bits. `sll` by *i* bits is equivalent to multiply by 2^i. The `srl` shifts the bits right and fills the empty bits with 0 bits. `srl` by *i* bits is equivalent to divide by 2^i (unsigned only).

Let's look at how the shift operations work with some example. The MIPS assembly codes and field values are shown below:

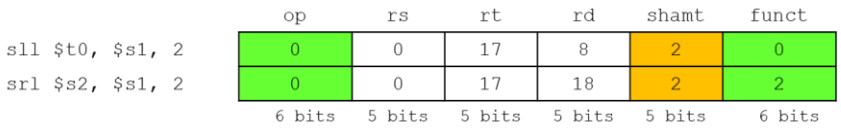
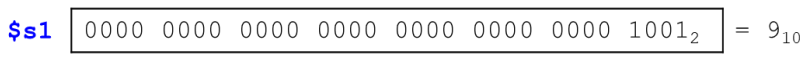
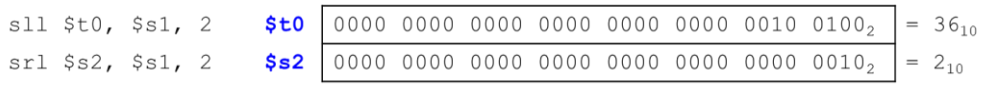


Fig. 10-7. Examples of Shift Operation

The source register address `$s1` has the following field value:



After executing the above MIPS assembly code, the target register addresses `$t0` and `$s0` have the following field values:



AND Operations

AND operation is useful to mask bits in a word. When executing AND operation, some bits are selected if both bits are TRUE; otherwise, it clears others to 0.

For example, let's execute AND operation of the values located in the register addresses `$t1` and `$t2`, and store the result in the register address `$t0`:

- `and $t0, $t1, $t2`

Only the selected bits are set to TRUE (1's), whereas the other bits are set to all FALSE (0's), as shown below:

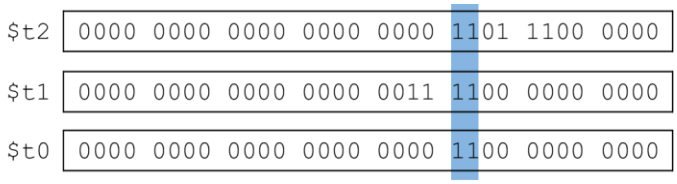


Fig. 10-8. Examples of AND Operation

OR Operations

OR operation is useful to include bits in a word. When executing OR operation, it sets some bits to TRUE (1's) and leaves others unchanged.

For example, let's execute OR operation of the values located in the register addresses `$t1` and `$t2`, and store the result in the register address `$t0`:

- `or $t0, $t1, $t2`

As shown below, some bits are set to TRUE highlighted in blue. The other bits are unchanged.

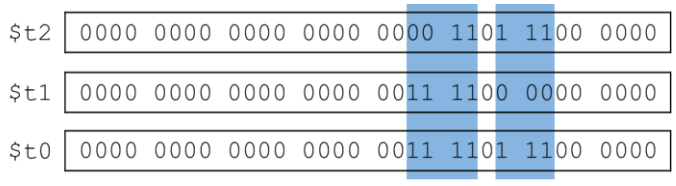


Fig. 10-9. Examples of OR Operation

NOT Operations

NOT operation is useful to invert bits in a word. That means it change 0 bit to 1 bit, and 1 bit to 0 bit. MIPS has a NOR 3-operand instruction that has the same function as the NOT instruction.

- `a NOR b == NOT (a OR b)`

we can invert bits in a word using NOR 3-operand instruction, as shown below:

- `nor $t0, $t1, $zero`

Since the register 0 always holds zero value, the NOR 3-operand instruction can execute the above instruction and return the result of NOT operation as shown below:

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$0	0000 0000 0000 0000 0000 0000 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Fig. 10-10. Examples of NOT Operation

Exercises

1) The source register addresses `$s1` and `$s2` are given below:

$s1	1111 1111 1111 1111 0000 0000 0000 0000
$s2	0100 0110 1010 0001 1111 0000 1011 0111

We would like to execute the following MIPS assembly code:

```
AND $s3, $s1, $s2
OR  $s4, $s1, $s2
NOR $s5, $s1, $s2
XOR $s6, $s1, $s2
```

What field values do the target register addresses, i.e., `$s3`, `$s4`, `$s5`, and `$s6`, hold?

`AND $s3, $s1, $s2`	**$s3**	0100 0110 1010 0001 0000 0000 0000 0000
`OR  $s4, $s1, $s2`	**$s4**	1111 1111 1111 1111 1111 0000 1011 0111
`NOR $s5, $s1, $s2`	**$s5**	0000 0000 0000 0000 0000 1111 0100 1000
`XOR $s6, $s1, $s2`	**$s6**	1011 1001 0101 1110 1111 0000 1011 0111

10.3. Conditional Operation

The conditional operations are used to branch to a labeled instruction if a condition is true. If the condition is false, the instructions are executed sequentially.

The following instructions show the conditional operations:

- `beq rs, rt, L1`

The branch on equal (`beq`) tests the equality of the condition. It branches to the instruction labeled `L1` if the condition (`rs == rt`) is true.

- `bne rs, rt, L1`

The branch on not equal (`bne`) tests the inequality of the condition. It branches to the instruction labeled `L1` if the condition (`rs != rt`) is true.

- `j L1`

In the jump (`j`), it jumps to the instruction labeled `L1` unconditionally.

Let's look at an example how the conditional branch instruction `beq` is used with the following MIPS assembly code:

```
        addi $s0, $0, 4                 # $s0 = 0 + 4 = 4
        addi $s1, $0, 1                 # $s1 = 0 + 1 = 1
        sll  $s1, $s1, 2                # $s1 = 1 << 2 = 4
        beq  $s0, $s1, target           # $s0 == $s1, so branch is taken
        addi $s1, $s1, 1                # not executed
        sub  $s1, $s1, $s0              # not executed

Target:
        add $s1, $s1, $s0               # $s1 = 4 + 4 = 8
```

Fig. 10-11. Examples of conditional instruction `beq`

The first two instructions set the values of the register addresses, `$s0` to 4 and `$s1` to 1. The value of the register address `$s1` is multiplied by 4 (=2^2) using the instruction `sll`. Since the equality of the conditional instruction `beq` is true, the branch is taken. Two instructions, `addi` and `sub`, are not executed. The value of the source register `$s1` is set to 8.

Let's look at the following example how the conditional branch `bne` is used with MIPS assembly codes:

```
        addi $s0, $0, 4                 # $s0 = 0 + 4 = 4
        addi $s1, $0, 1                 # $s1 = 0 + 1 = 1
        sll  $s1, $s1, 2                # $s1 = 1 << 2 = 4
        bne  $s0, $s1, target           # $s0 == $s1, so branch is not taken
        addi $s1, $s1, 1                # $s1 = 4 + 1 = 5
        sub  $s1, $s1, $s0          # $s1 = 5 - 4 = 1

target:
        add $s1, $s1, $s0           # $s1 = 1 + 4 = 5
```

Fig. 10-12. Examples of conditional instruction `bne`

The first three instructions (two `addi` and an `sll`) set the values of the register addresses, `$s0` to 4 and `$s1` to 4. Since the inequality of the conditional instruction `bne` is false, the branch is not taken. Two instructions, `addi` and `sub`, are executed. The value of the source register address `$s1` is set to 5 in this case.

Let's look at an example how the unconditional branch `j` is used with the following MIPS assembly codes:

```
        addi    $s0, $0, 4                  # $s0 = 4
        addi    $s1, $0, 1                  # $s1 = 1
        j       target                      # jump to target
        addi    $s1, $s1, 1                 # not executed
        sub     $s1, $s1, $s0               # not executed

target:
        add     $s1, $s1, $s0               # $s1 = 1 + 4 = 5
```

Fig. 10-13. Examples of unconditional branch `j`

The first two instructions set the values of the register addresses, `$s0` to 4 and `$s1` to 1. The unconditional branch `j` jumps to the target of the instruction. Two instructions, `addi` and `sub`, are not executed in this case. The value of the source register address `$s1` is set to 5 with the last instruction.

The unconditional branch, Jump register (`jr`) is used to jump to the address held in a register. The following MIPS assembly code includes the unique address:

```
0x00002000 addi      $s0, $0, 0x2010          # $s0 = 0x2010
0x00002004 jr  $s0                            # jump to 0x00002010
0x00002008 addi $s1, $0,1                     # not executed
0x0000200c sra  $s1, $s1, 2                   # not executed
0x00002010 lw        $s3, 44($s1)             # executed after jr
```

Fig. 10-14. Examples of unconditional branch `jr`

The fist instruction sets the value of the register address `$s0` to `0x2010`. The second instruction, Jump register (`jr`) jumps to the address `0x00002010` that was held in the register `$s0`.

Conditional Statements

There are conditional statements commonly used in high-level languages, as shown below:

- `if` statements
- `if`/`else` statements
- `while` loops
- `for` loops

Let's look at how those conditional statements are translated into MIPS assembly code.

if statements

The high-level code with `if` statement is shown below:

```
if (i == j)
       f = g + h;

f = f - i;
```

If the condition is true, the code executes the add operation, followed by the subtract operation. If the condition is false, it won't execute the add operation. It only executes the subtract operation. Since all the instruction is executed in sequence, the `if` conditional statement is translated into the MIPS assembly code with `bne` instruction.

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

       bne $s3, $s4, L1     # if i ≠ j
       add $$s0, $s1, $s2   # f = g + h
L1:
       sub $s0, $s0, $s3    # f = f - i
```

The conditional branch `bne` is taken if the register value `$s3` is not equal to `$s4`, and then jumps to the target of the instruction `sub`. If the two register values are same, the conditional branch `bne` is not taken. Both `add` and `sub` instructions are executed.

if/else statements

The high-level code with `if`/`else` statement is shown below:

```
if (i == j)
       f = g + h;

else
       f = f - i;
```

If the condition is true, the code executes the add operation; otherwise it executes the subtract operation. That means it executes either add or sub operation. This conditional statement is translated into MIPS assembly code with both `bne` and `j` instructions.

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

       bne $s3, $s4, else   # if i ≠ j
       add $$s0, $s1, $s2   # f = g + h
       j   done                    # skip else
else:
       sub $s0, $s0, $s3    # f = f - i
done:
```

The conditional branch `bne` is taken if the register value `$s3` is not equal to `$s4`, and then jumps to the target of the instruction line `else`. After executing the instruction `sub`, the program is terminated.

If the two register values are same, the conditional branch `bne` is not taken. It executes the next instruction `add` in sequence, followed by the unconditional branch `j` that jumps to the end of this program.

while loops

The high-level code with `while` loop is shown below:

```
int pow = 1;
int x = 0;


while(pow != 128)
{
   pow = pow * 2;
   x = x + 1;
}
```

If the condition of the `while` statement is true, the code executes all the instructions within the curly bracket {}; otherwise it terminate the program. This statement is translated into MIPS assembly code with both `beq` and `j` instructions.

```
# $s0 = pow, $s1 = x
       addi $s0, $0, 1            # pow = 1
       addi $s1, $0, 0            # x = 0

       addi $t0, $0, 128    # $t0 = 128
while:                      # comparison
       beq  $s0, $t0, done        # if pow=128
       sll  $s0, $s0, 1     # pow=pow*2
       addi $s1, $s1, 1     # x = x + 1
       j    while
done:
```

The conditional branch `beq` is taken if the register value `$s0` is equal to `$t0`, and then jumps to the target of the instruction line `done`. If the register value `$s0` is not equal to `$t0`, the branch is not taken and it executes the next instructions in sequence, followed by the unconditional branch `j` that jumps to the target of the instruction `while` of this program.

for loops

The high-level code with `for` loop is shown below:

```
int sum = 0;
```

```
for(i=0;i!=10;i=i+1)
{
       sum = sum + i;
}
```

The integer variable `sum` is initialized with 0. In the `for` loop, there are three instructions, as shown below:

1) index `i` initialized with 0; `i = 0;`
2) the condition of for loop; `i != 10;`
3) increment/decrement of the index `i`; `i = i + 1`

After initializing the index `i`, the statement checks the condition. If the condition is true, it executes all the instructions within the curly bracket { }. After increasing the value of the index `i` by 1, the statement checks the condition again. If the condition is false, it terminates the program. If the condition is true, it repeats all the previous steps until the condition becomes false. This statement is translated into MIPS assembly code with both `beq` and `j` instructions.

```
# $s0 = i, $s1 = sum
       addi $s1, $0, 0            # sum = 0
       addi $s0, $0, 0            # i = 0
       addi $t0, $0, 10           # $t0 = 10

for:
       beq  $s0, $t0, done        # if i == 10
       add  $s1, $s1, $s0         # sum=sum+i
       addi $s0, $s0, 1           # i = i + 1
       j    for
done:
```

The first three instructions initialize the register values, i.e., `$s0`, `$s1` and `$t0`. If the register value `$s0` is equal to `$t0`, the conditional branch `beq` is taken, and then jumps to the target of the instruction line `done`. If the register value `$s0` is not equal to `$t0`, the branch is not taken and then it executes the next instructions, i.e., `add` and `addi` instructions in sequence. The `add` instruction updates the register value `$s1` that is equivalent to the integer variable `sum`. The `addi` instruction increases the value of the index `i` by 1. The unconditional branch `j` jumps to the target of the instruction line `for` of this program.

Loops using slt

The high-level code with `for` loop is shown below:

```
int sum = 0;



for(i=1;i<101;i=i*2)
{
   sum = sum + i;
```

```
}
```

The integer variable `sum` is initialized with 0. In the `for` loop, there are three instructions, as shown below:

1) index `i` initialized with 1; `i = 1;`
2) the condition of for loop; `i < 101;`
3) increment/decrement of the index `i`; `i = i * 2`

After initializing the index `i`, the statement checks the condition. If the condition is true, it executes all the instructions within the curly bracket { }. After multiplying the value of the index `i` with 2, the statement checks the condition again. If the condition is false, it terminates the program. If the condition is true (the value of the index `i` is less than 101), it repeats all the previous steps until the condition becomes false. Since there is the less than condition in the loop, this statement is translated into MIPS assembly code with `slt`, `beq`, and `j` instructions, where the set less than (`slt`) instruction sets the destination register value to 1 if the first register operand is less than the second register operand.

```
# $s0 = i, $s1 = sum
       addi $s1, $0, 0            # sum = 0
       addi $s0, $0, 1            # i = 1
       addi $t0, $0, 101   # $t0 = 101
loop:
# if (i < 101) $t1=1, else $t1 = 0
       slt  $t1, $s0, $t0
       beq  $t1, $0, done         # if $t1=0
       add  $s1, $s1, $s0         # sum = sum + i
       sll  $s0, $s0, 1           # i = i * 2
       j    loop
done:
```

After initializing the register values, there comes the instruction line `loop`. In the `slt` instruction, the register value `$s0` is compared with the register value `$t0`. If `$s0` is less than `$t0`, it sets the value of the register `$t1` to 1; otherwise sets to 0. The `beq` instruction compares the register value `$t1` with `$0`. If the register value `$t1` is 1 (`$s0 < $t0`), the branch is not taken and it executes the next instructions in sequence and updates the variables `sum` and `i`, followed by the unconditional branch `j` that jumps to the target of the instruction `loop` of this program. If the register value `$t1` is 0 (`$s0 => $t0`), the branch is taken and it terminates the program.

Chapter 11: Pipeline

In this chapter, we introduce microarchitecture, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines, memories, and other logic building blocks needed to implement an architecture. We also define instruction pipelining, hazards, pipelined datapath, and pipelined control.

Objectives

By the end of this chapter you should be able to:

- Identify five stages in MIPS pipeline
- Recognize structure hazards, data hazard, and control hazard
- Demonstrate knowledge of pipelined datapath
- Clarify pipeline usage in a single-clock cycle
- Clarify pipeline operation in multi-cycle pipeline diagram

11.1. Instruction Pipelining

R-Type Instruction

The instruction is fetched from memory, and the PC is incremented by 4 in the instruction fetch (IF) stage, as shown in Fig. 11-1. The fetched instruction is used by other parts of the datapath. Program Counter (PC) always holds the next memory address to be fetched, where PC is a byte address, not bit address. PC value is updated by adding 4 to the previous PC value.

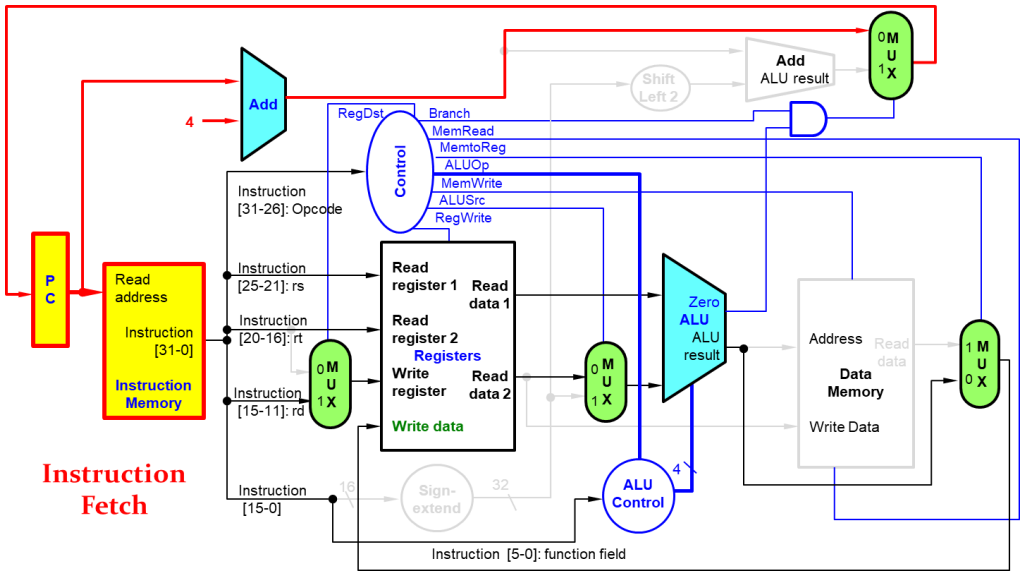


Fig. 11-1. Instruction Fetch Stage of R-Type Instruction

Fig. 11-2 shows the instruction decode (ID) stage of R-Type Instruction. The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. The inputs (RS and RT) carrying the register number to the register file are all 5-bit wide, whereas the lines carrying data values are 32-bit wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4-bit wide.

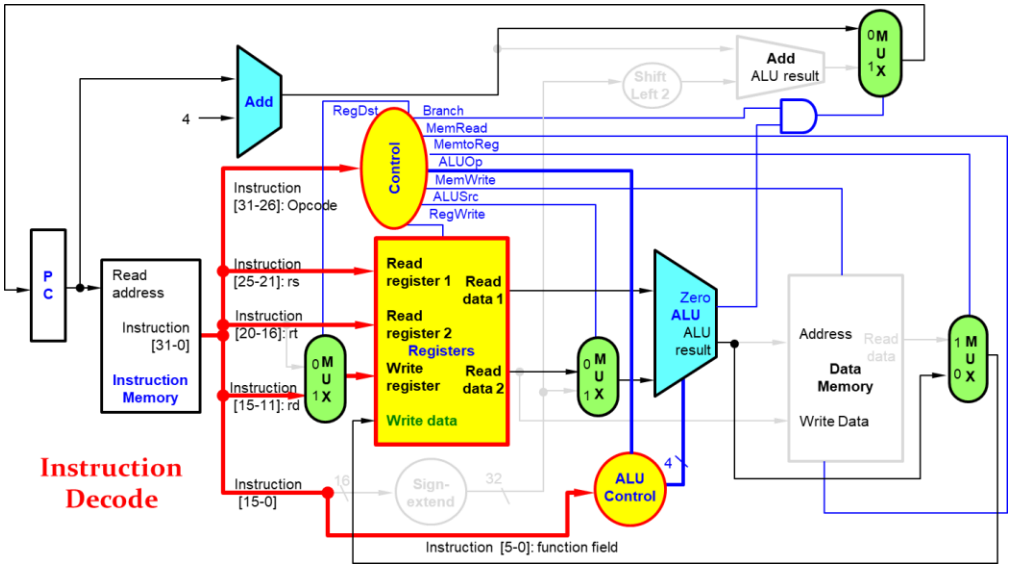


Fig. 11-2. Instruction Decode Stage of R-Type Instruction

The arithmetic operations are executed in the execute (EX) stage, as shown in Fig. 11-3. Two 32-bit wide inputs from register files are fed into ALU to execute logic operations.

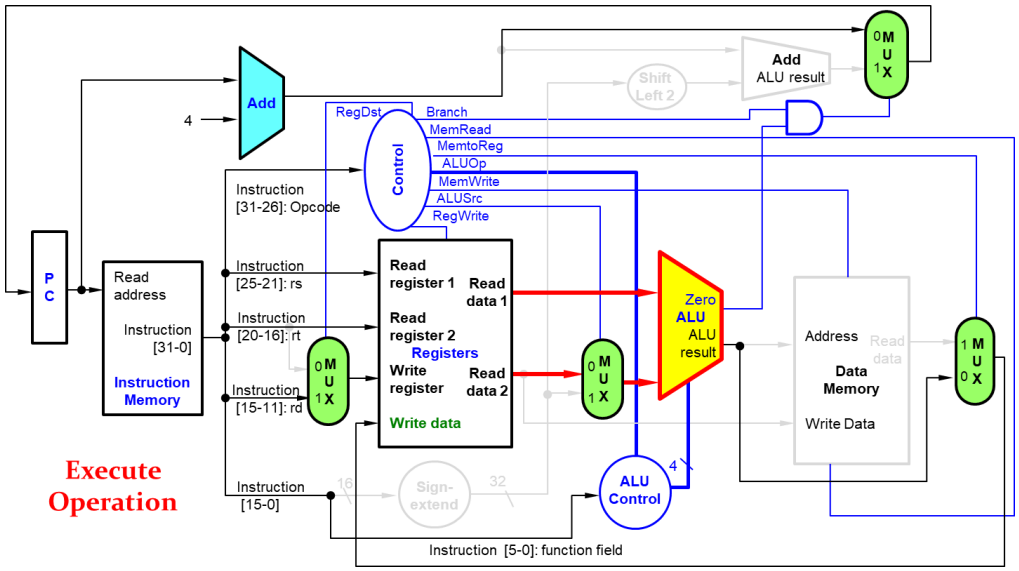


Fig. 11-3. Execute Stage of R-Type Instruction

There is nothing happening in memory access stage in R-type instruction.

In the write back (WB) stage of Fig. 11-4, the result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register.

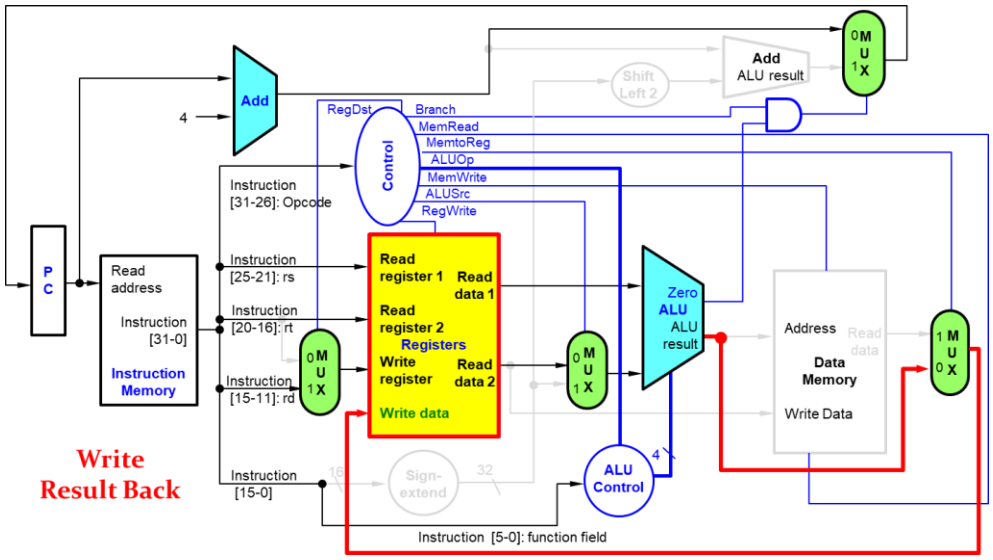


Fig. 11-4. Write Back Stage of R-Type Instruction

Load Instruction

In load instruction of Fig. 11-5, the instruction is fetched from memory, and PC value is increased by 4, which is the same as R-type instruction.

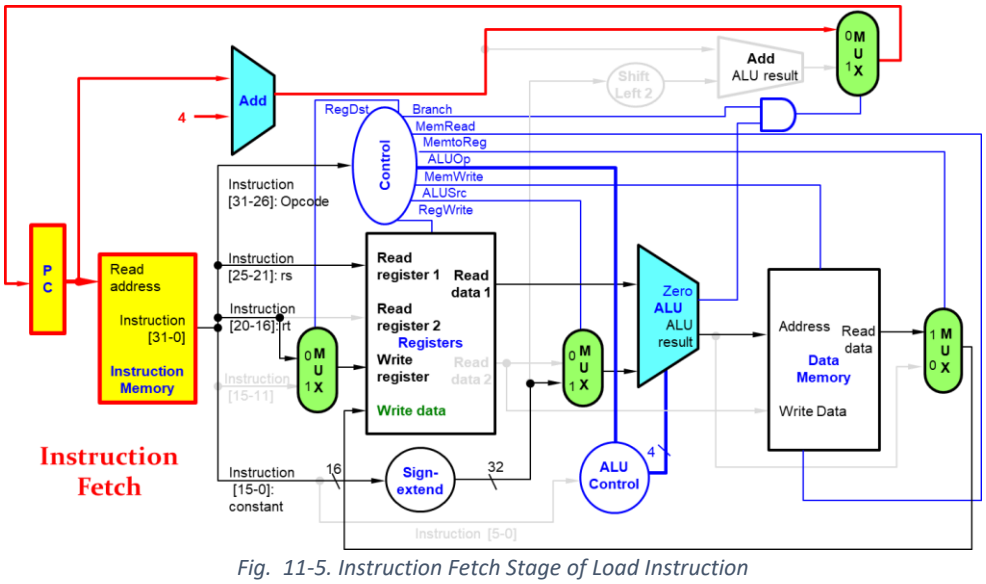


Fig. 11-5. Instruction Fetch Stage of Load Instruction

The fetched instruction is used by other parts of the datapath. Program Counter (PC) always holds the next memory address to be fetched, where PC is a byte address, not bit address. PC value is updated by adding 4 to the previous PC value.

Fig. 11-6 shows the ID stage of Load instruction. In this stage the instruction field value [25 – 21] is fed into the register files and produces Read data 1 (32 bits), whereas the instruction field value [15 – 0] is fed into sign-extend function and produces a 32-bit constant/address value.

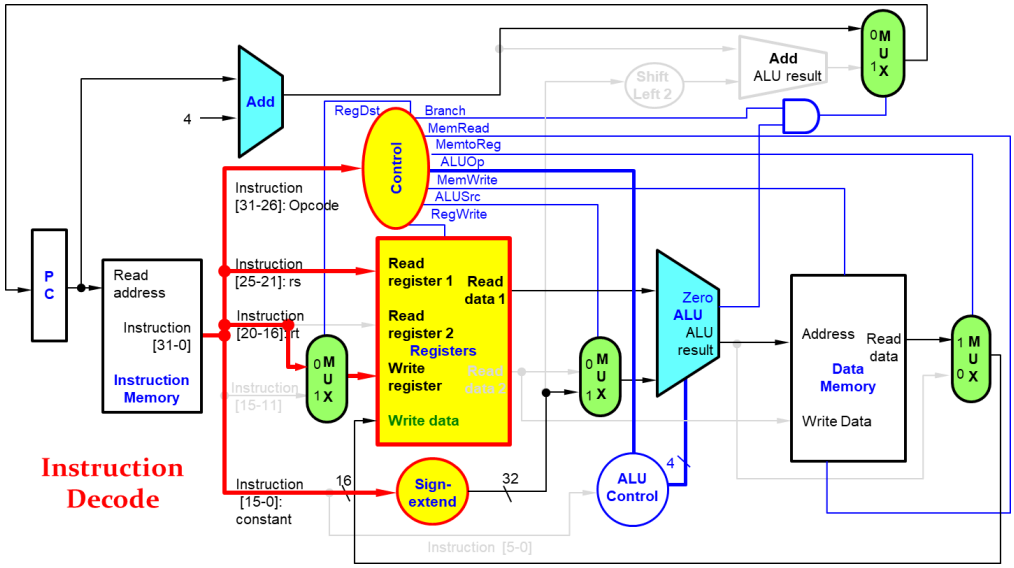


Fig. 11-6. Instruction Decode Stage of Load Instruction

The memory address is calculated with two 32-bit values in the execute stage of Fig. 11-7.

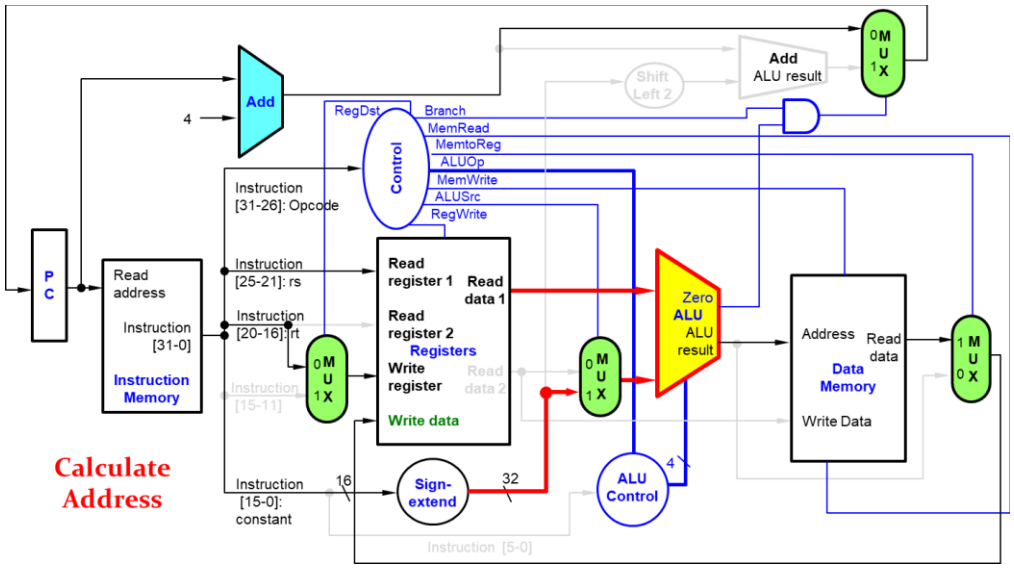


Fig. 11-7. Execute Stage of Load Instruction

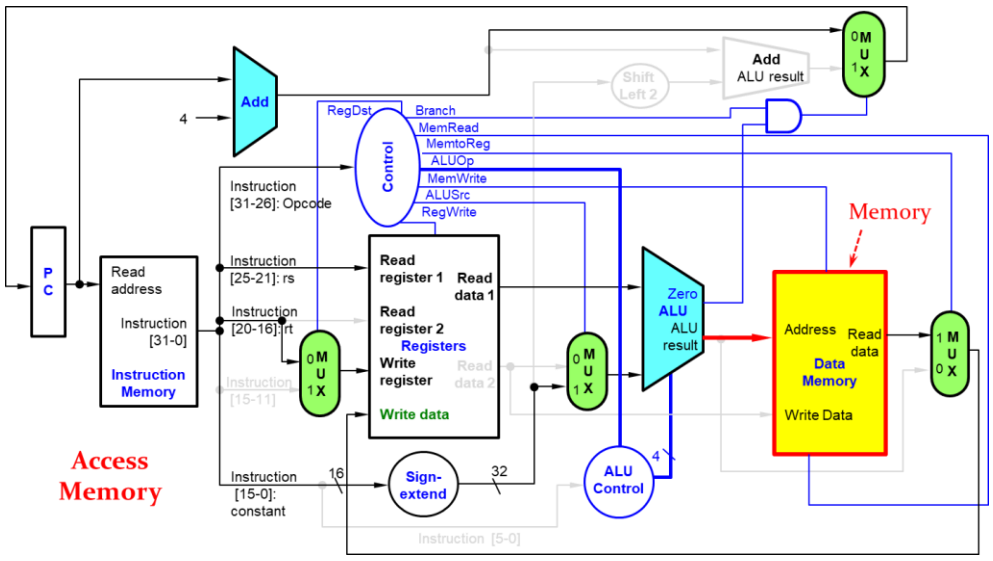


Fig. 11-8. Memory Access Stage of Load Instruction

Fig. 11-8 shows the memory access stage of Load instruction. In this stage, the control bit for MemWrite is set to 1. Data memory contents designated by the address input are replaced by the value on the Write data input.

As shown in the following figure, the control bit for MemtoReg is set to 1 in the write back stage. The value fed to the register Write data input comes from the data memory.

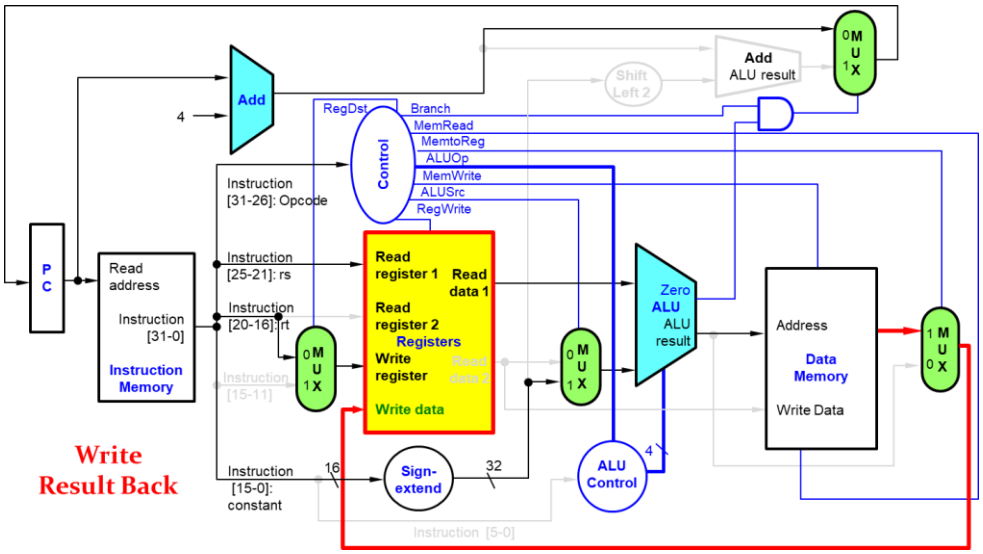


Fig. 11-9. Write Back Stage of Load Instruction

Performance Issues

Historically early computers with very simple instruction sets did use this implementation technique. Pipelining improves efficiency by executing multiple instruction simultaneously.

The longest delay determines clock period in the pipeline. In the MIPS instruction sets, the load instruction is the critical path because it includes the following stage:

- Instruction memory (IF) → register file (ID) → ALU (EX) → data memory (MEM) → register file (WB)

It is not feasible to vary period for different instructions, because that violates design principle, making the common case fast. We can improve performance by pipelining, meaning that each instruction is executed in a different stage simultaneously in the processor.

With pipeline, we can overlap the execution. It is the similar concept to improve the performance with parallelism. The laundry analogy exemplified this parallelism. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. Assume there are total four laundries and four steps for each laundry, i.e. washer, dryer, folding clothes, and clothes closet. Each step needed 30 minutes to complete. A sequential laundry takes 8 hours for 4 loads of wash, i.e. 4 loads × 2 hours, whereas a pipelined laundry takes just 3.5 hours, i.e. 1.5 hours + 30 minutes × 4).

MIPS pipeline has five stages, one step per stage:

- IF: Instruction fetch from memory
- ID: Instruction decode & register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register

Let's assume that the time for stages is as follows:

- 100 ps for register read or write
- 200 ps for other stages

In the following table, we can compare the total time of the pipelined datapath with a single-cycle datapath:

Table 11-1. Pipelined DataPath

Instruction	Instruction fetch	Register read	ALU op	Memory access	Register Write	Total Time
`lw`	200 *ps*	100 *ps*	200 *ps*	200 *ps*	100 *ps*	800 *ps*
`sw`	200 *ps*	100 *ps*	200 *ps*	200 *ps*		700 *ps*
`R-format`	200 *ps*	100 *ps*	200 *ps*		100 *ps*	600 *ps*
`beg`	200 *ps*	100 *ps*	200 *ps*			500 *ps*

The load instruction includes all the pipeline stage so that the total time of the pipelined datapath is 800 ps, whereas the R-type instruction has a total time of 700 ps because it doesn't include the memory access stage.

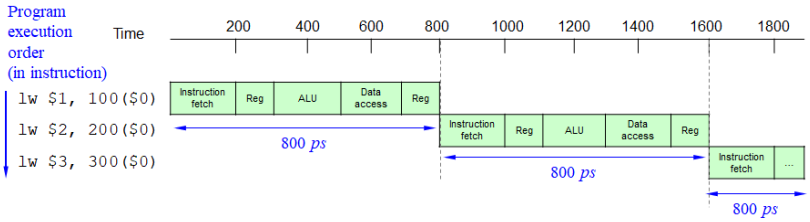


Fig. 11-10. Nonpipelined Execution of Three Load Word Instruction

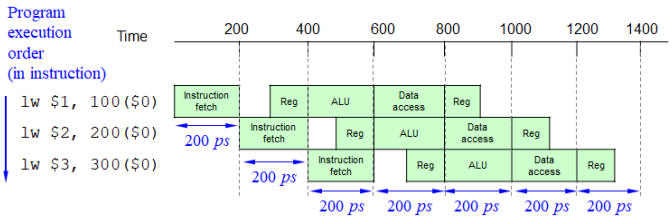


Fig. 11-11. Pipelined Execution of Three Load Word Instruction

Figs. 11-10 and 11-11 compare nonpipelined and pipelined execution of three load word instructions. In the nonpipelined execution, a single-cycle is 800ps, thus the total time to execute three load instructions is 3 × 800 ps or 2400 ps in the nonpipelined design. On the other hand, in the pipelined execution, a clock cycle is 200 ps, and the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, even though some stages take only 100 ps. The total time to execute three load instructions is 200 ps × 5 + 200 ps × 2 or 1400 ps. Notice that the pipelined execution time (1400 ps) is faster than the nonpipelined execution time (2400 ls).

What would happen if we added 1,000,000 instructions in the pipelined and non-pipelined process in the above examples?

For the pipelined process, each instruction adds 200 ps to the total execution time. The total time will be as follows:

- 1,000,000 × 200 ps + 1400 ps = 200,001,400 ps

For the nonpipelined process, each instruction adds 800 ps to the total execution time. The total time will be as follows:

- 1,000,000 × 800 *ps* + 2400 *ps* = 800,002,400 *ps*

The ratio of total execution times for real programs on nonpipelined to pipelined processors will be like

$$\frac{800{,}002{,}400\ ps}{200{,}002{,}400\ ps} \cong \frac{800\ ps}{200\ ps} = 4.00$$

If all stages are balanced, i.e., all stage take the same time, the total time of the pipelined process can be faster (× number of stages) than the total time of the nonpipelined process. If all stages are not

balanced, speedup is less. Note that this speedup is due to the increased throughput. The time for each instruction (latency) doesn't decrease.

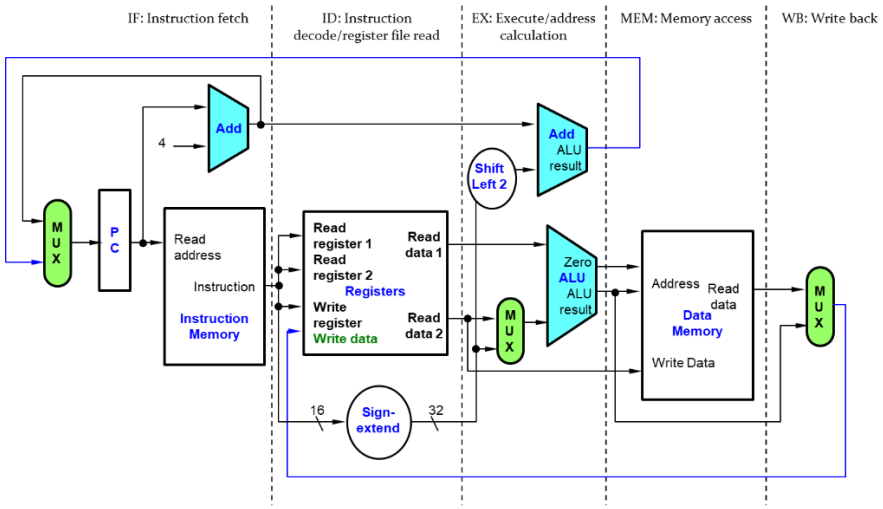


Fig. 11-12. MIPS Pipelined Datapath

As shown in the above figure, MIPS Pipelined Datapath has IF (Instruction fetch), ID (Instruction decode/register file read), EX (Execute/address calculation), MEM (Memory access), and WB (Write back). Each step of the instruction can be mapped onto the datapath from left to right.

The update of the PC and the write-back step sends either the ALU result or the data from memory to the left to be written into the register file.

11.2. Pipelined Datapath

The pipelined datapath needs registers between stages. The pipeline registers separate each pipeline stage, as shown in the following figure.

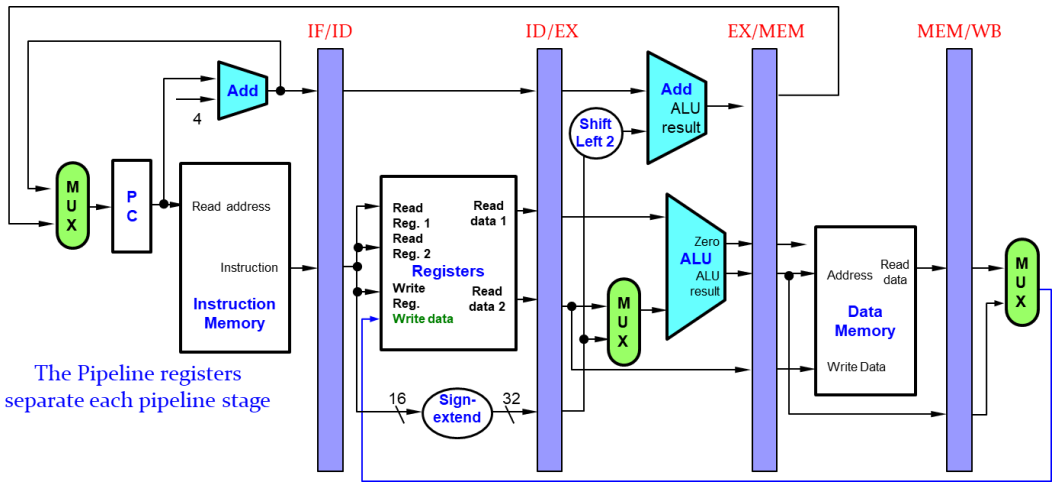


Fig. 11-13. Pipeline Registers

The pipeline registers are labeled by the stages that they separate; for example, the first is labeled IF/ID because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. The pipeline operates cycle-by-cycle flow of instructions through the pipelined datapath. The single-clock-cycle pipeline diagram shows pipelined usage in a single cycle and highlight resources used in the pipeline, whereas the multi-clock-cycle diagram shows the graph of operation over time.

Let's look at "single-clock-cycle" diagrams for load and store instructions.

Single-clock-cycle Pipeline Diagram

Fig. 11-14 shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.

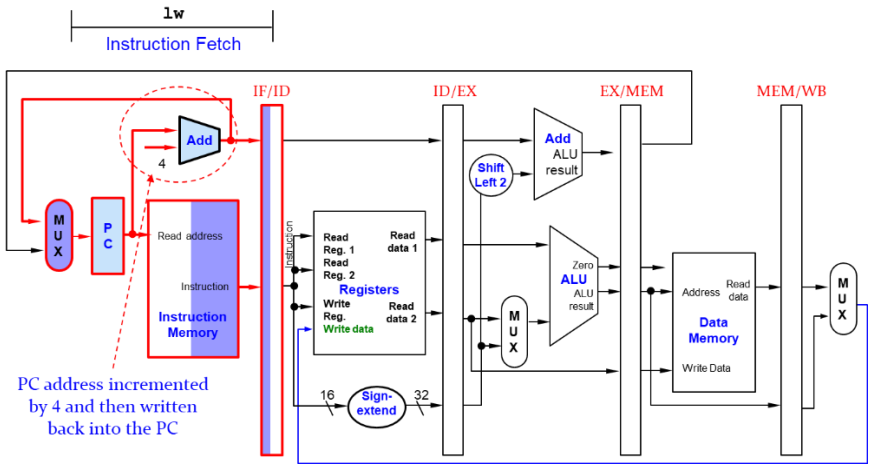


Fig. 11-14. Instruction Fetch Stage for Load and Store

The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

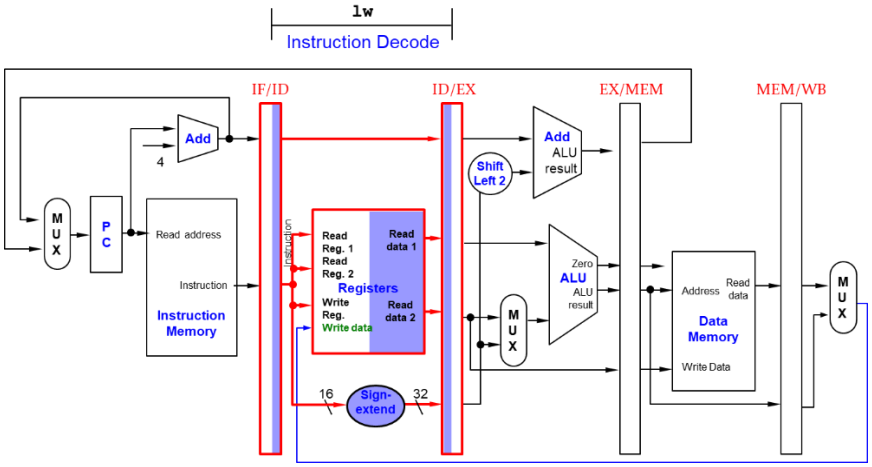


Fig. 11-15. Instruction Decode Stage for Load and Store

Fig. 11-15 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

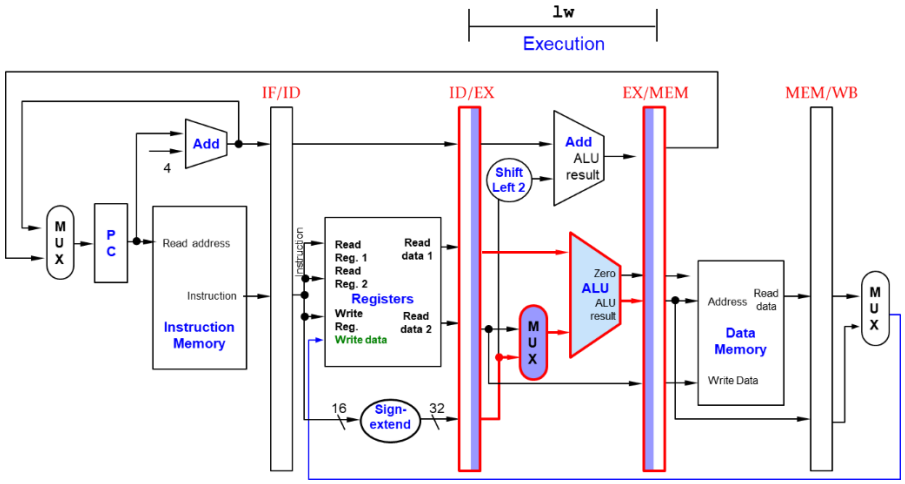


Fig. 11-16. Execute Stage for Load

Fig. 11-16 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.

Fig. 11-17 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

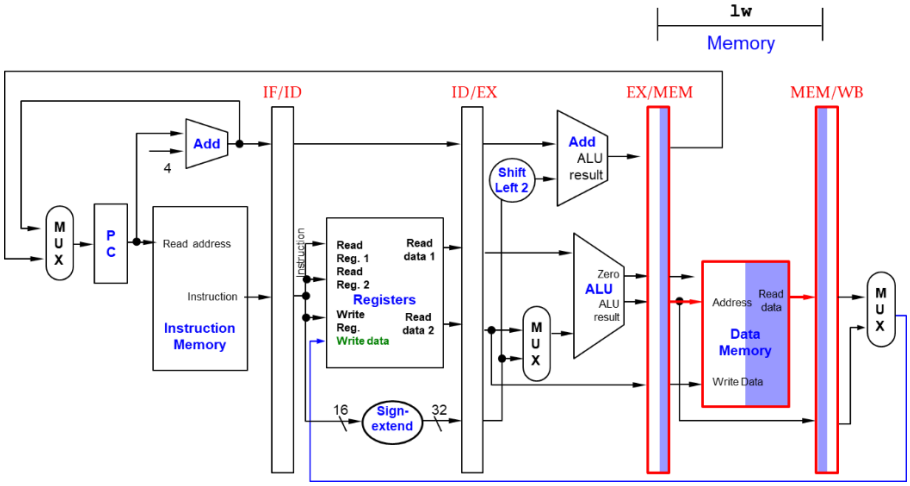


Fig. 11-17. Memory Access Stage for Load

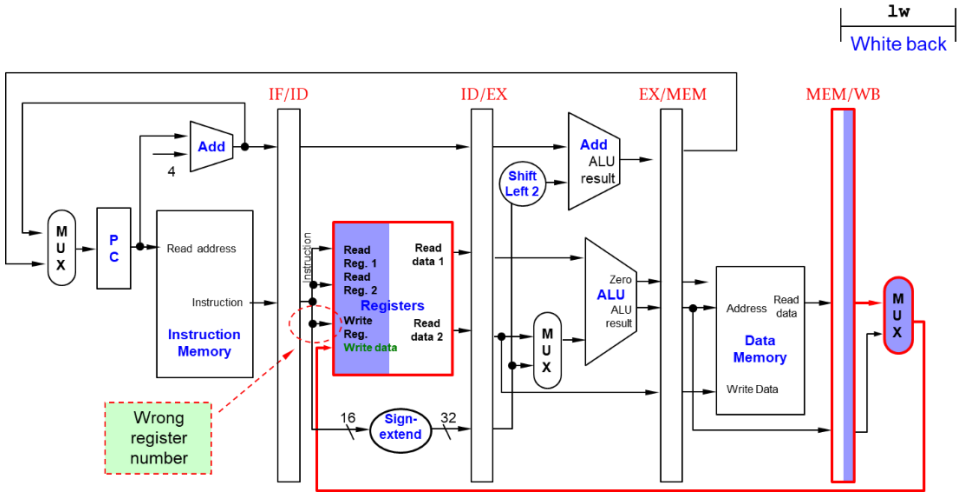


Fig. 11-18. Write Back Stage for Load

Fig. 11-18 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure. When the processor executes WB stage of Load instruction, the write register number is not corresponding to the load instruction, because other instructions were executed for the ID stage.

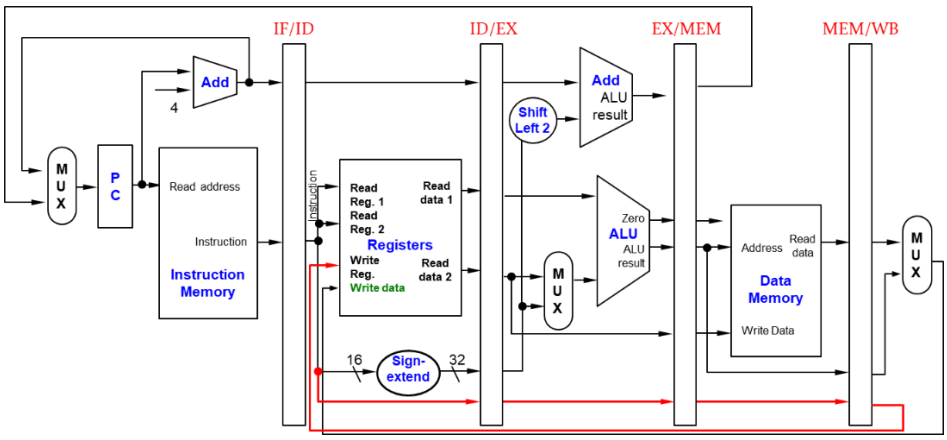


Fig. 11-19. Corrected Datapath for Load

Fig. 11-19 shows the corrected datapath for Load instruction. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in Red color in the following figure:

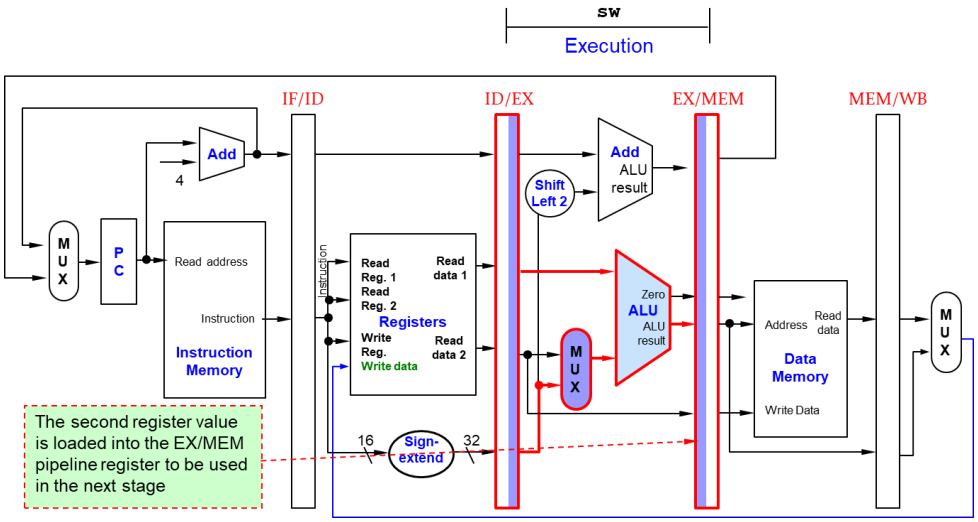


Fig. 11-20. Execute Stage for Store

Fig. 11-20 shows the execute stage of Store instruction. Unlike the third stage of the load instruction, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

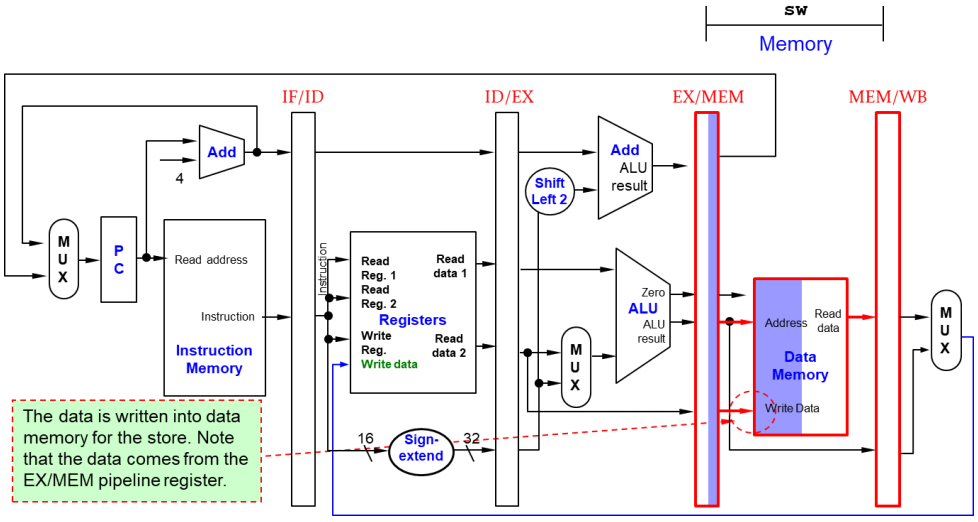


Fig. 11-21. Memory Access Stage for Store

Fig. 11-21 shows the memory access stage of Store instruction, where the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register.

Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in the last (WB) stage.

Multi-Cycle Pipeline Diagram

Fig. 11-22 shows the multiple-clock-cycle pipeline diagram for five instructions. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom.

A representation of the pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles. These stylized datapaths represent the five stages of our pipeline graphically. In the figure, IM represents the instruction memory and PC in the instruction fetch stage and DM represents data memory.

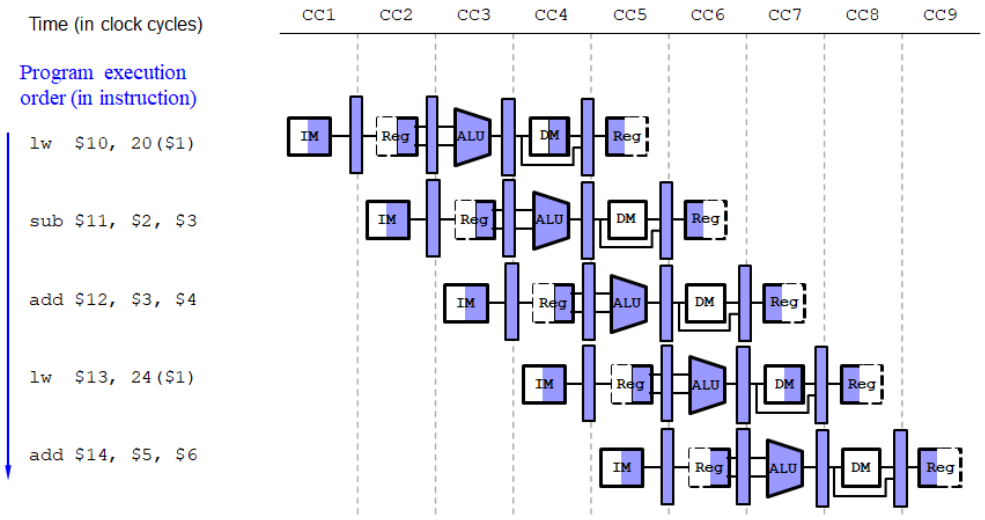


Fig. 11-22. Multi-Cycle Pipeline Resource Usage

Fig. 11-23 shows the more traditional version of the multiple-clock-cycle pipeline diagram. The previous figure shows the physical resources used at each stage, while This figure uses the name of each stage.

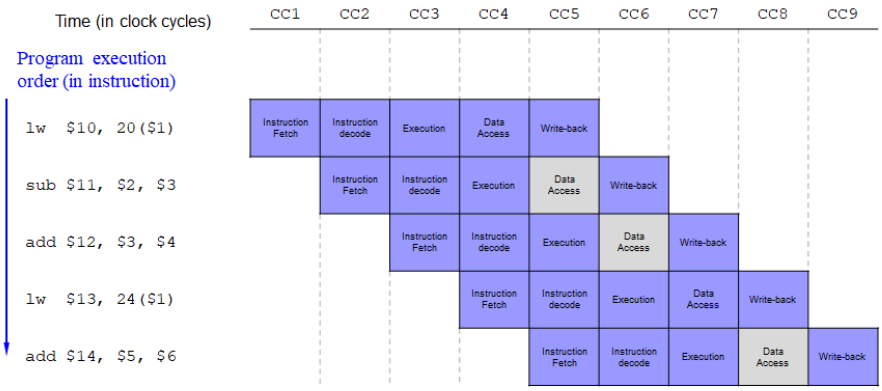


Fig. 11-23. Multi-Cycle Pipeline Resource Usage

Exercises

Assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
260 ps	360 ps	170 ps	310 ps	220 ps

1) What is the clock cycle time in a pipelined and non-pipelined processor?
   - Pipelined processor: Clock cycle time = 360 ps
   - Non-pipelined processor: Clock cycle time = 260 + 360 + 170 + 310 + 220 = 1320 ps

2) What is the total latency of seven LW instructions in a pipelined and non-pipelined processor (assume no stalls or hazards)
   - Pipelined processor: Total latency = 360 × 5 + 360 × 6 = 3960 ps
   - Non-pipelined processor: Total latency = 1320 ps × 7= 9240 ps

11.3. Pipelined Controls

In the pipeline, there are lots of control signals. Depend on the control signals enabled or disabled, the components of the pipeline are executed to complete each stage. The following figure shows that what control signals are used for each stage:

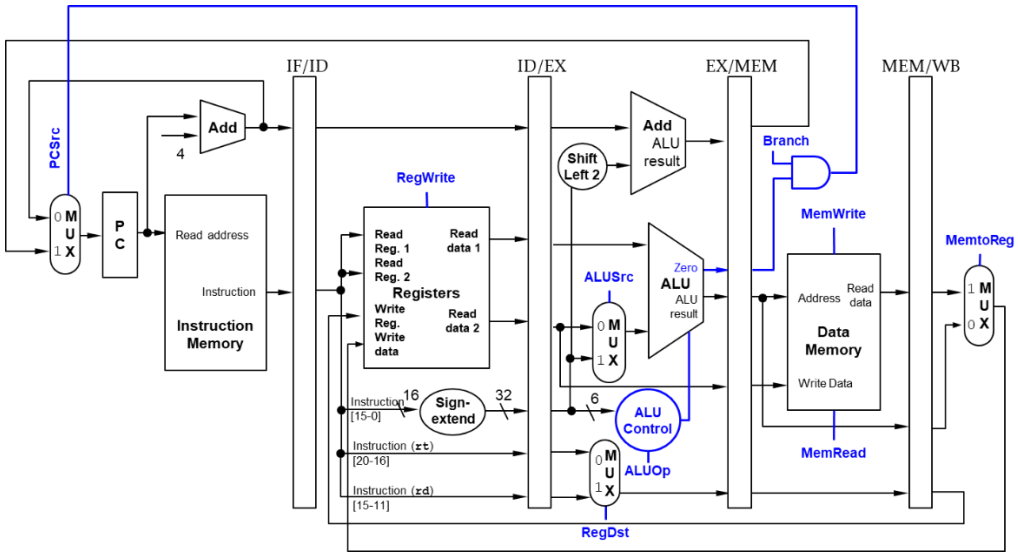


Fig. 11-24. Simplified Pipelined Control

- IF: If PCSrc set to 0, the PC value increased by 4; otherwise, a specific address forwarded from a branch instruction.
- ID/register file read: the same thing happens at every clock cycle. No optional control lines.
- Execution/address calculation: the signals, i.e. RegDst, ALUOp, and ALUSrc, are set. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.
- Memory access: the control lines, i.e. Branch, MemRead, and MemWrite are set.
- Write Back: two control lines, MemtoReg and RegWrite.

The effect of each control signal is summarized in the following table:

Table 11-2. Effect of Each Control Signal

Signal name	Effect when reasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16)	The register destination number for the Write register comes from the rd field (bits 15:11)
RegWrite	None	The register on the Write input is written with the value of the Write data input
ALUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write date input comes from the data memory

The control signals are derived from the instruction, as shown in the following figure:

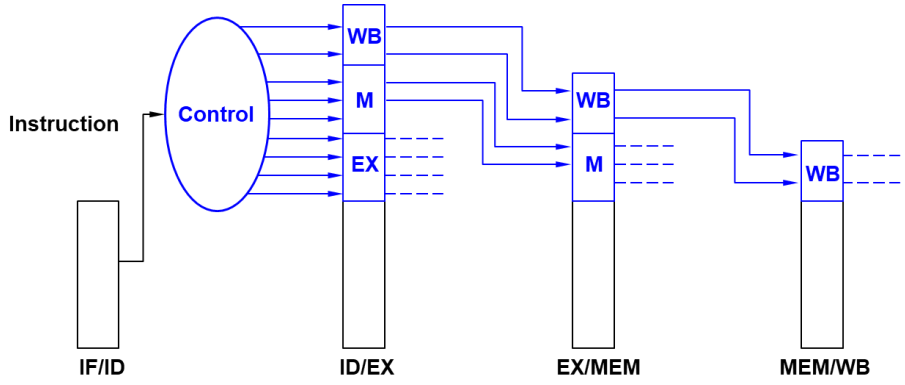


Fig. 11-25. Pipelined Control Signal

Since the control lines start with the EX stage, the control information, i.e. total nine control signals, can be created during ID stage. Four of the nine control lines are used in the EX stage, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines. Three are used during the MEM stage, and the last two are passed to MEM/WB pipeline register for use in the WB stage.

Example – Pipeline Control

Let's look at some example what control signals are created in a given instruction and how these signals are used for each pipeline stage with the following instructions, where we assume that there are no hazard illustrations:

```
lw    $10, 20($1)
sub   $11, $2, $3
and   $12, $4, $5
or    $13, $6, $7
add   $14, $8, $9
```

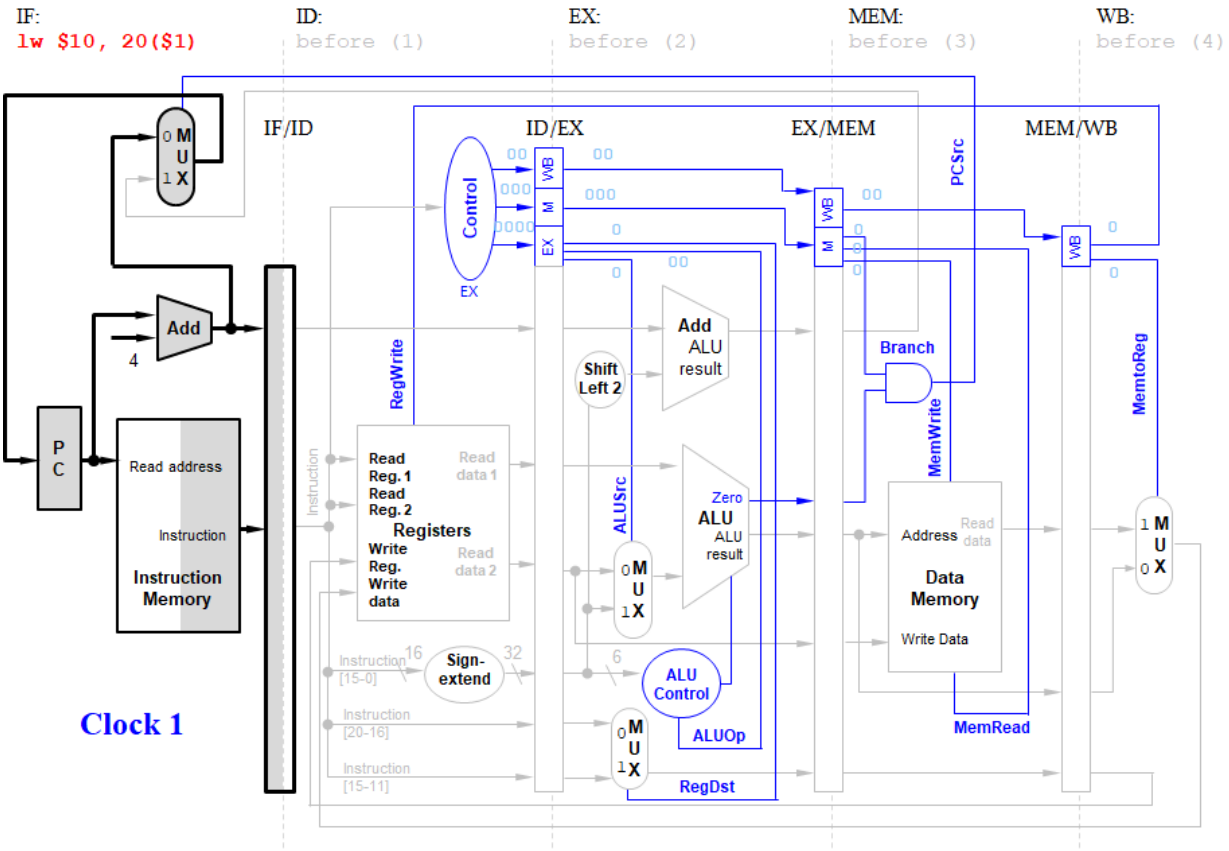


Fig. 11-26. Pipeline Control – Click 1

Fig. 11-26 shows that the LW instruction is fetched in the instruction memory of IF stage. At the end of the clock cycle, the LW instruction is in the IF/ID pipeline registers. Note that since there is no control signal created in this stage, all the control signals are set to 0.

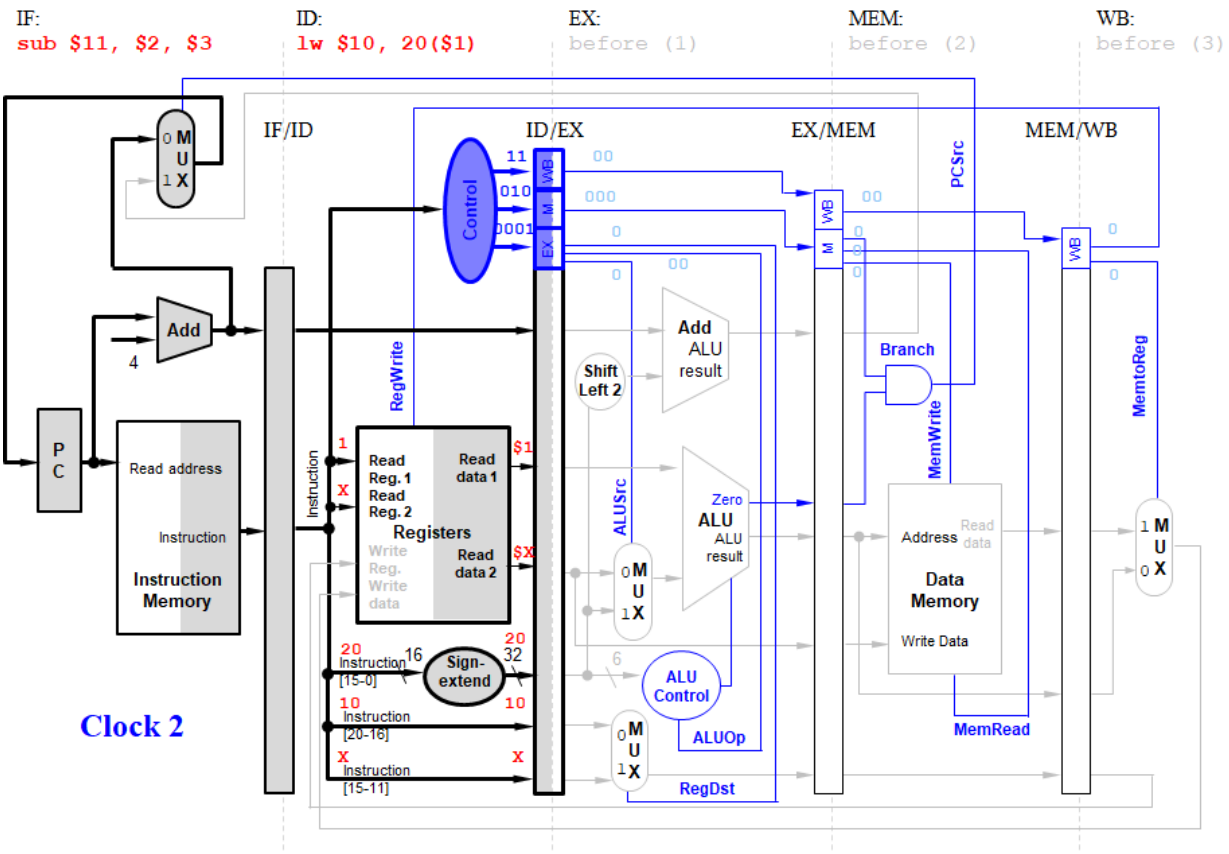


Fig. 11-27. Pipeline Control – Click 2

Fig. 11-27 shows the second clock cycle, where the LW instruction moves to the ID stage, and sub instruction enters in the IF stage.

Note that the values of the instruction fields and the selected source registers are shown in the ID stage. Hence register `$1` and the constant 20, the operands of LW, are written into the ID/EX pipeline register. The number 10, representing the destination register number of LW, is also placed in ID/EX. Bits 15–11 are 0, but we use the symbol X to show that a field plays no role in a given instruction.

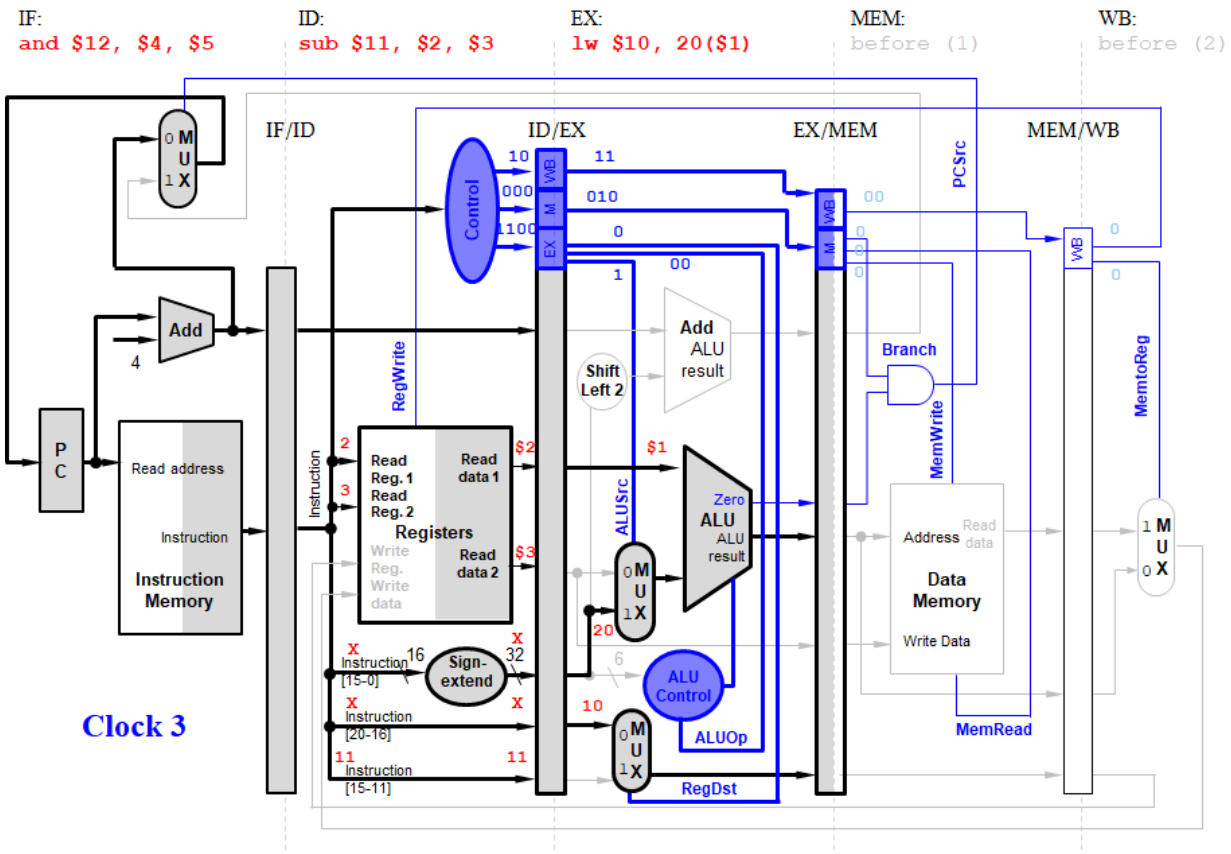


Fig. 11-28. Pipeline Control – Click 3

LW instruction enters the EX stage in the third clock cycle, adding `$1` and 20 to form the address in the EX/MEM pipeline register.

At the same time, the SUB instruction (`sub $11, $2, $3`) enters ID stage, reading registers `$2` and `$3`, and the AND instruction (`and $12, $4, $5`) starts IF stage.

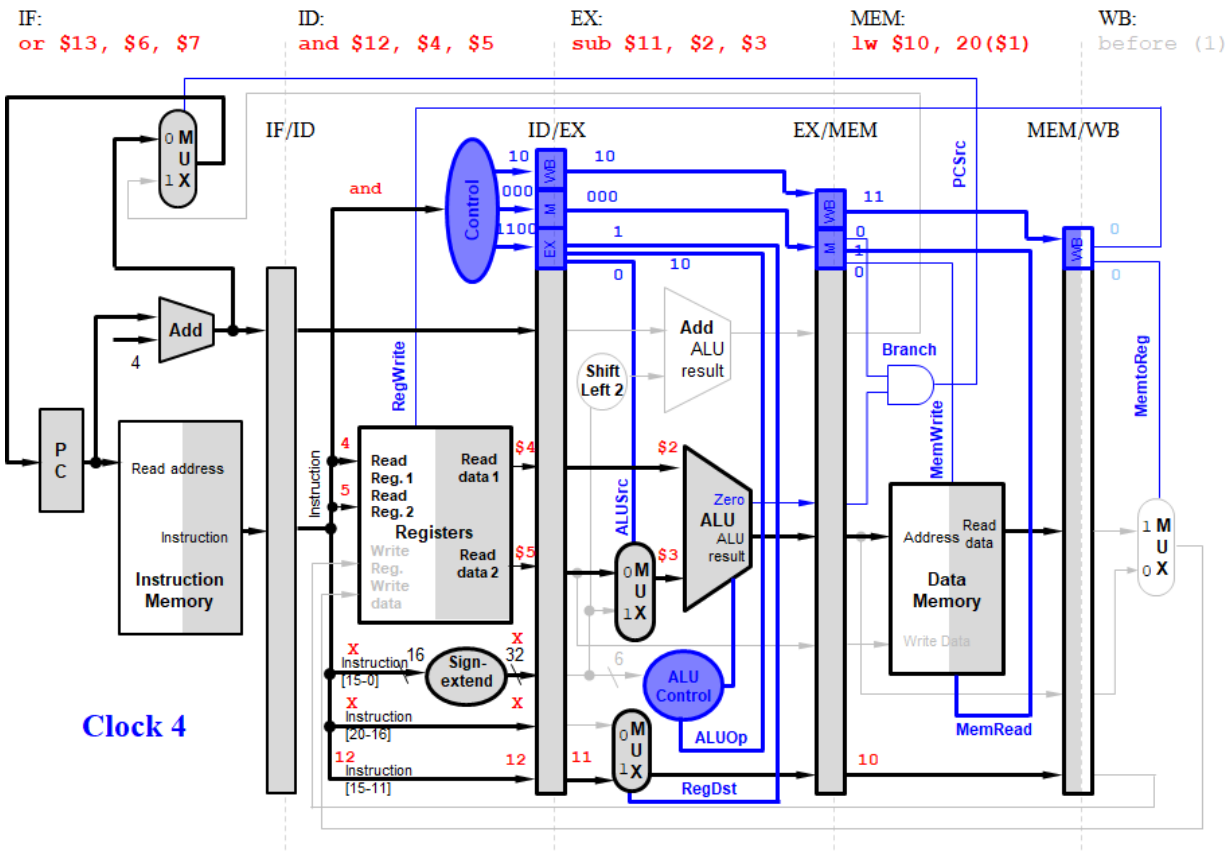


Fig. 11-29. Pipeline Control – Click 4

In the fourth clock cycle, LW instruction moves into MEM stage, reading memory using the value in EX/MEM as the address.

In the same clock cycle, the ALU subtracts `$3` from `$2` and places the difference into EX/MEM pipeline registers, reads registers `$4` and `$5` during ID stage, and the OR instruction (`or $13, $6, $7`) enters IF stage.

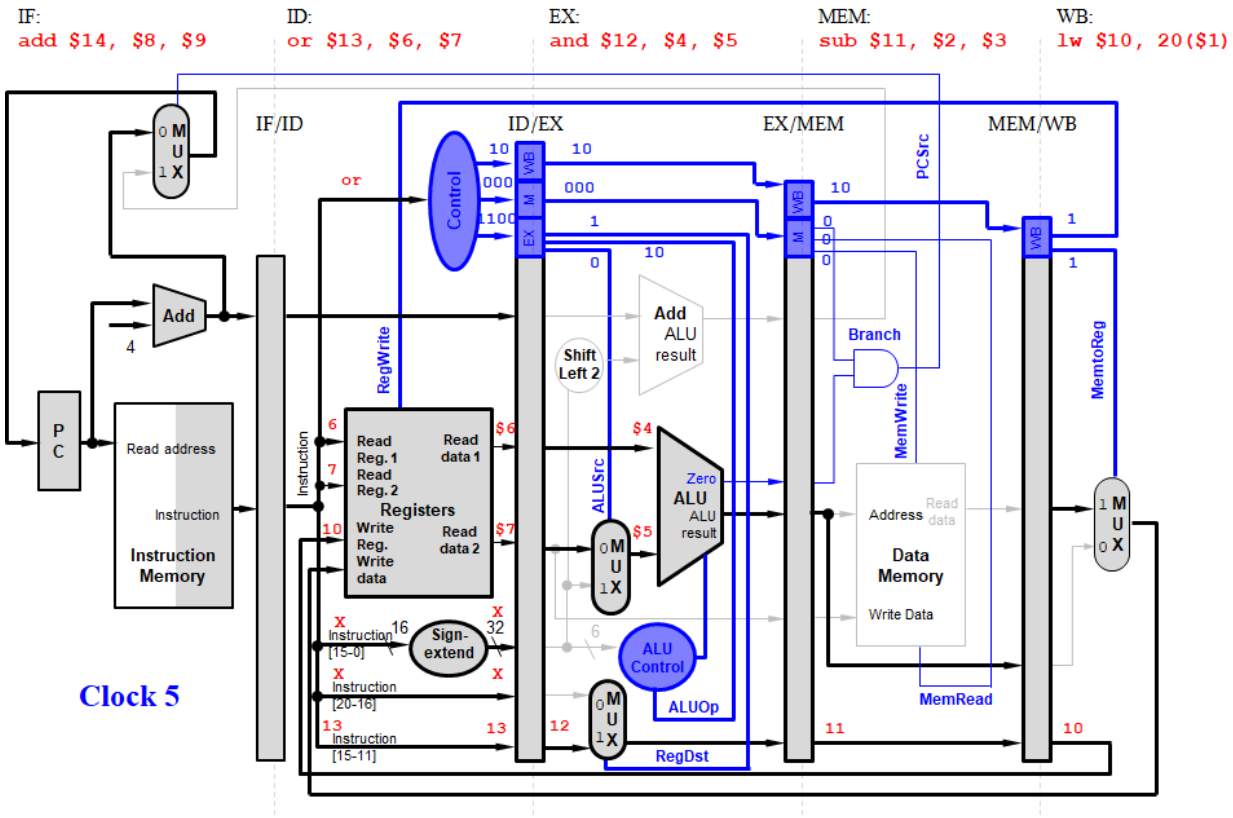


Fig. 11-30. Pipeline Control – Click 5

The final instruction, an ADD instruction in this example, enters IF stage in the datapath. All instructions are engaged in the fifth clock cycle. By writing the data in MEM/WB into the write register 10, LW instruction completes and both the data and the register number are in MEM/WB.

In the same clock cycle, SUB instruction sends the difference in EX/MEM to MEM/WB, and the rest of the instructions move forward.

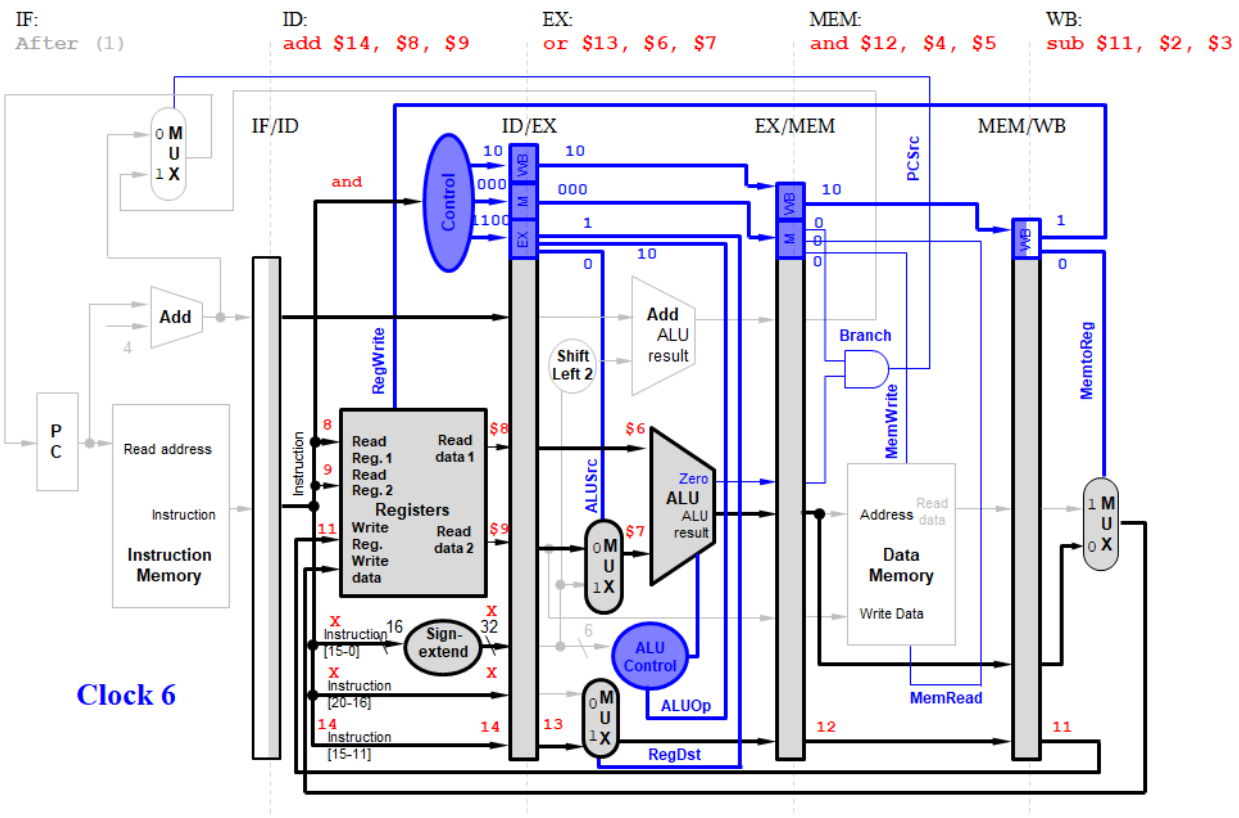


Fig. 11-31. Pipeline Control – Click 6

In the sixth clock cycle, SUB instruction selects the value in MEM/WB to write to the write register number 11, again found in MEM/WB.

The remaining instructions play follow-the-leader: the ALU calculates the OR of `$6` and `$7` for the OR instruction in the EX stage, and registers `$8` and `$9` are read in the ID stage for the ADD instruction.

The instructions after ADD are shown as inactive just to emphasize what occurs for the five instructions in the example.

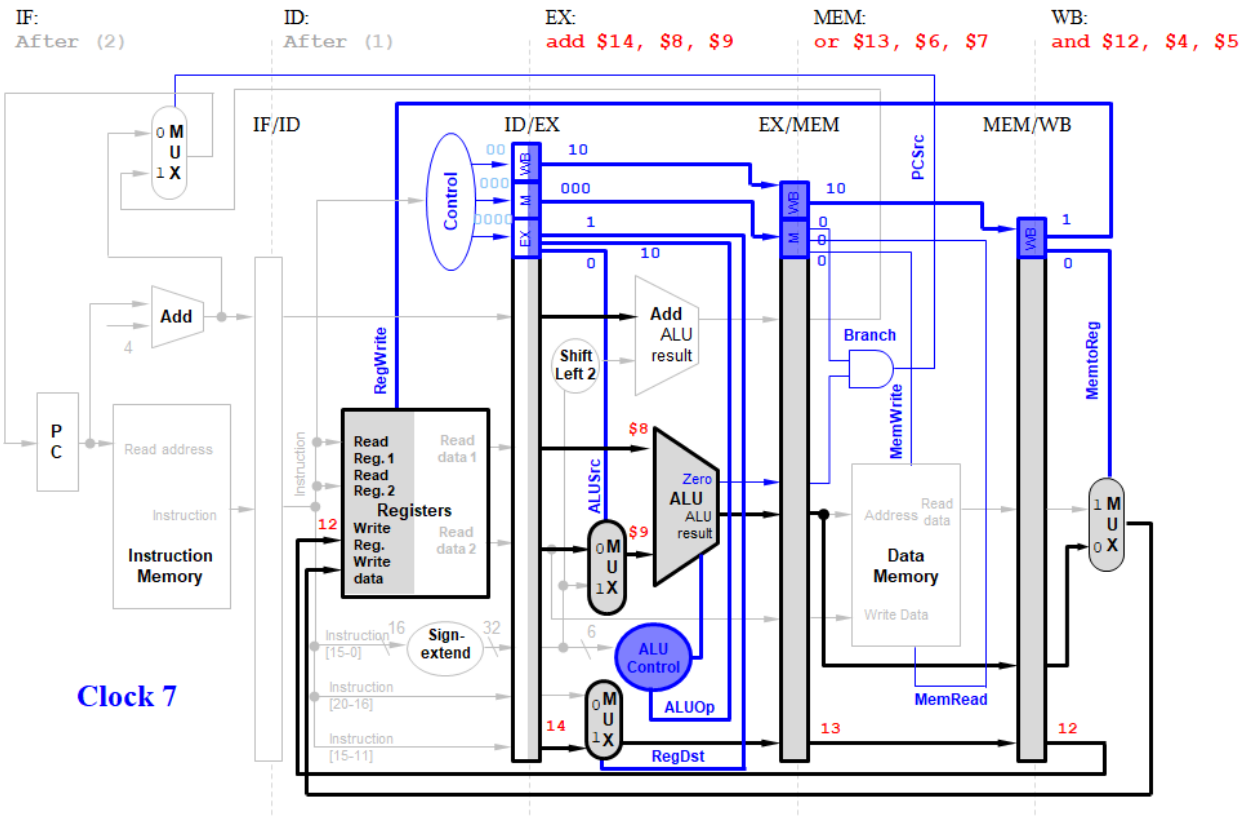


Fig. 11-32. Pipeline Control – Click 7

In the seventh clock cycle, the ADD instruction brings up the rear, adding the values corresponding to registers `$8` and `$9` during the EX stage.

The result of the OR instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result of the AND instruction in MEM/WB to the write register `$12`.

Note that the control signals are deasserted (set to 0) in the ID stage, since no instruction is being executed.

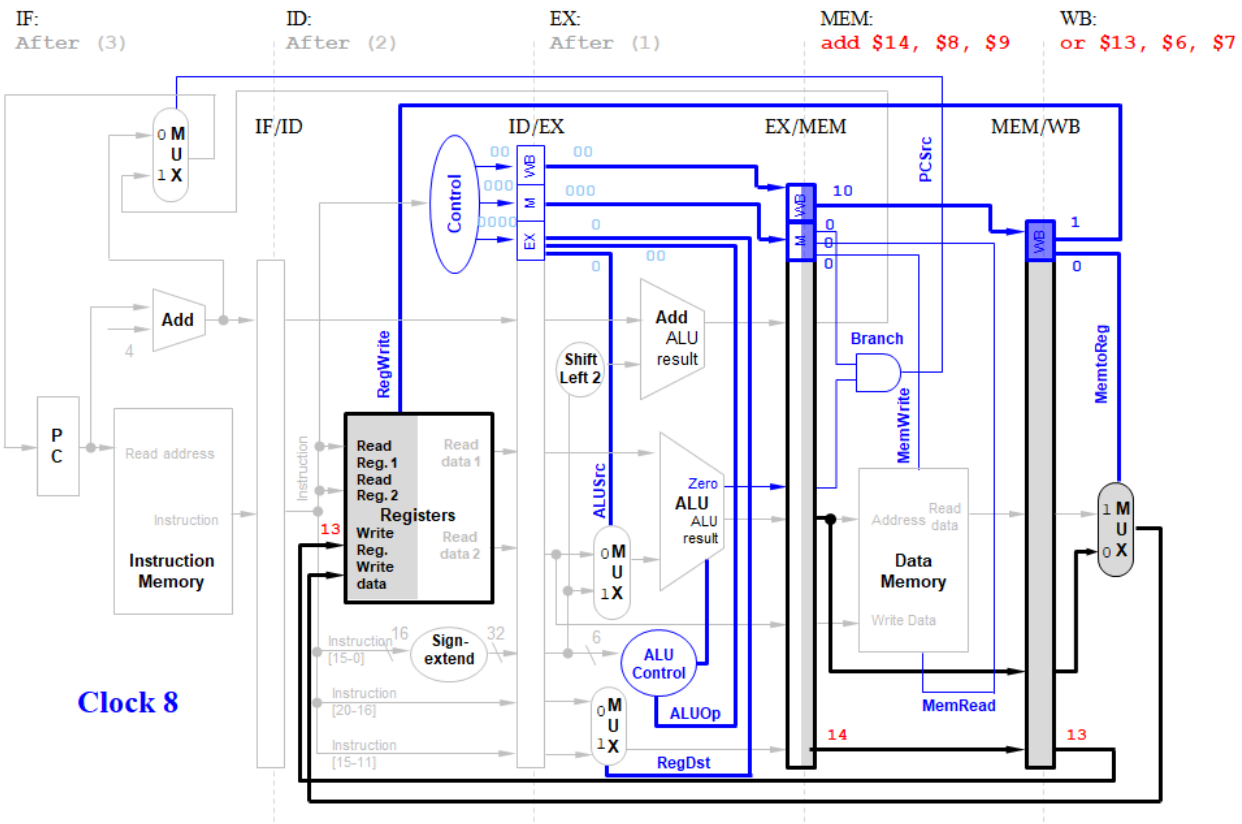


Fig. 11-33. Pipeline Control – Click 8

In the eighth clock cycle, the WB stage writes the result to the write register `$13`, thereby completing OR instruction, and the MEM stage passes the sum of the ADD instruction from EX/MEM to MEM/WB.

The instructions after ADD instruction are shown as inactive for pedagogical reasons.

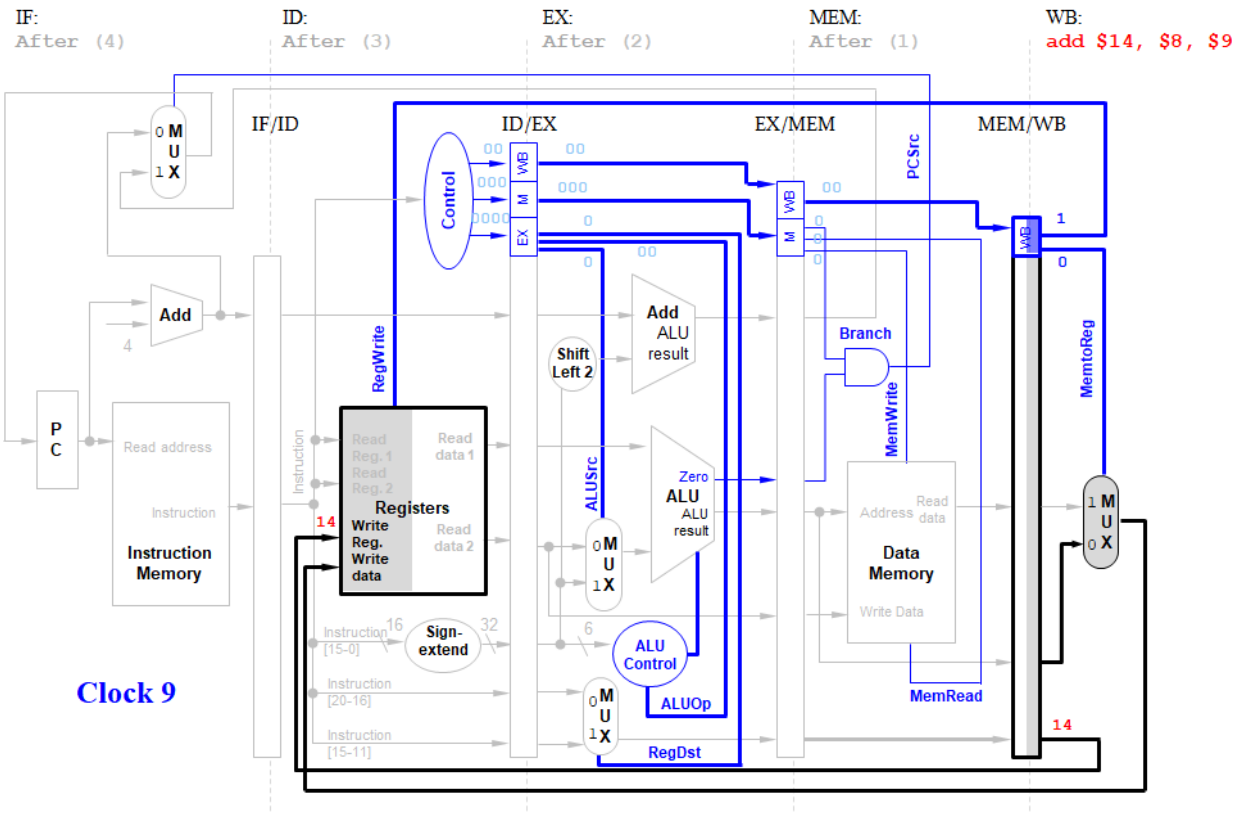


Fig. 11-34. Pipeline Control – Click 9

The WB stage writes the sum in MEM/WB into the write register `$14`, completing all five-instruction sequences including ADD instruction. The instructions after ADD instruction are shown as inactive for pedagogical reasons.

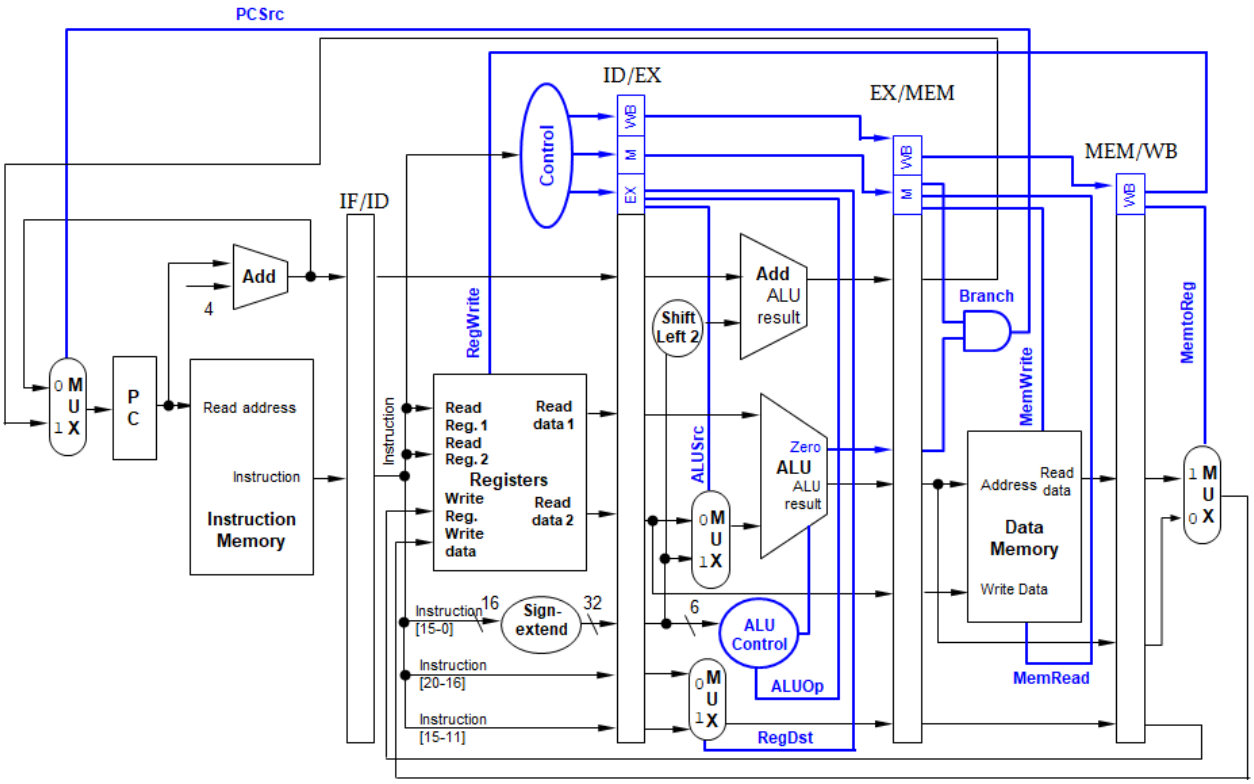


Fig. 11-35. Summary of Pipelined Control

Fig. 11-35 summarized the pipeline control. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. All the control values are as follows for each stage:

- EX stage: ALUSrc, ALUOp, and RegDst
- MEM stage: Branch, MemWrite, PCSrc, and MemRead
- WB stage: MEMtoReg and RegWrite

The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Chapter 12: Memory

In this chapter, we introduce memory hierarchy and cache memory. Computer system performance depends on the processor microarchitecture as well as the memory system. The current memory systems are slower than processors. The increasing gap between processor and memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor. We introduce memory hierarchy to mitigate the increasing gap between them. We also introduce SRAM and DRAM technologies and flash and disk storages.

Objectives

By the end of this chapter you should be able to:

- Demonstrate knowledge of memory hierarchy
- Recall how the (temporal and spatial) locality to make memory access fast
- Clarify knowledge of the terms, i.e. hit, hit rate, miss, miss rate in the memory hierarchy
- Differentiate memory techniques; SRAM and DRAM technologies, and flash and disk storages
- Evaluate efficiency of direct-mapped cache

12.1. Memory Hierarchy

Computer performance depends on processor performance and memory system performance.

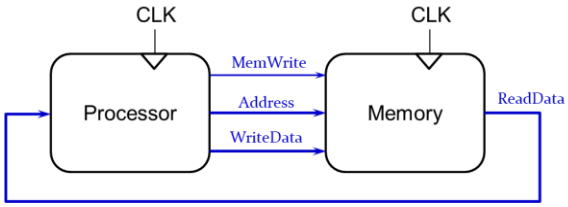


Fig. 12-1. Memory Interface

As shown in the above figure, the processor frequently needs to access and read the data in the memory. If the memory system performance to read the data is not fast as much as the processor performance, the overall performance will be degraded due to the memory performance.

The process performance was similar to the memory performance in 1980, as shown below. But it hasn't been true since the 1980's. A technological improvement raises the performance of the processor. On the other hand, the memory performance was not improved as much as the processor performance. The memory performance is not good as much as the processor performance and We call this gap as the processor-memory performance gap.

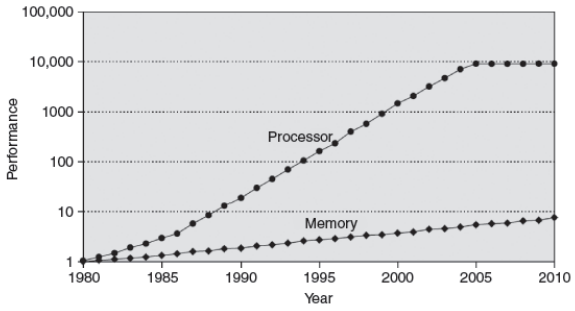
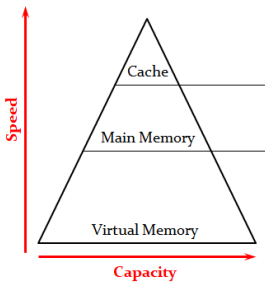


Fig. 12-2. Gap between Processor and Memory Speeds

Since the memory system is not fast as much as the processor performance, we should make memory system appear as fast as processor. We can use memory hierarchy to make the memory system fast as much as the processor speed.

The ideal memory has the following characteristics: 1) it should be fast and cheap (inexpensive); and 2) it should have a large capacity to store data in term of volume. But in reality, we cannot choose the one that meets all the requirement. If we choose one of the fastest one, it will be expensive and have limited capacity. If we choose one with the largest capacity, it will be slow.

There are three memory types in the memory hierarchy, i.e. cache, main memory, virtual memory, as shown below:



Technology	Price / *GB*	Access time (*ns*)	Bandwidth (*GB/s*)
SRAM	$10,000	1	25 +
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

Fig. 12-3. Memory Hierarchy Pyramid

The cache memory (SRAM) is fast but it can only keep small amount of data. SRAM (Static RAM) needs a lot more transistors in order to store a certain amount of memory. That's why it is very expensive. On the other hand, the virtual memory (SSD: solid state drive, and HDD: hard disk drive) is very slow but it can store unlimited data in the storage. The main memory (DRAM) is located in between the cache memory and the virtual memory. In term of cost, it is cheaper than the cache memory. But in term of speed, it is faster than the virtual memory. DRAM (Dynamic RAM) requires the data to be refreshed periodically in order to retain the data.

Locality

The memory hierarchy uses the locality to speed up the performance of the memory. There are two types of locality; one is temporal locality, and the other one is spatial locality.

Temporal Locality uses the locality in time. If the data is used recently, the processor may use it again soon. By keeping recently accessed data in higher levels of memory hierarchy, the processor can access the data immediately.

Spatial locality uses the locality in space. If the data is used recently, the processor is most likely to use neighboring data soon. When the processor access data, the memory system brings nearby data into higher levels of memory hierarchy too. The process can access the neighboring data immediately.

We take advantage of principle of locality by implementing the memory of a computer as a memory hierarchy.

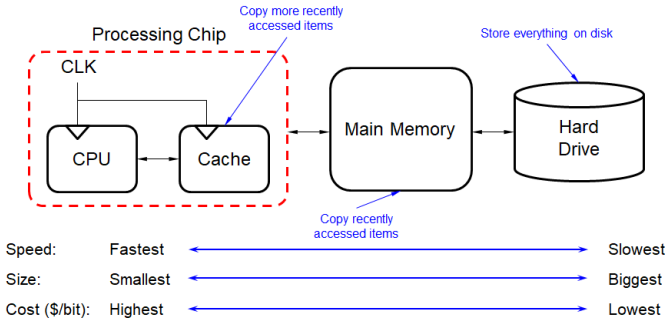


Fig. 12-4. Principle of Locality

As shown in the above figure, the faster memories are more expensive per bit than the slower memories and thus are smaller. The computer systems store everything on hard disk drive (virtual memory). The memory systems copy recently accessed items in the main memory and copy more recently accessed items in the cache memory.

When the memory systems make a copy from low levels to high levels of memory hierarchy, it copies chunk of data, not a single line of data. When the data is found in that level of memory hierarchy, it is called as *hit*. The fraction or percentage of accesses that result in a hit is called the hit rate, expressed as the following equation:

$$Hit\_rate = \frac{\#\ hits}{\#\ memory\_accesses}$$

When the data is not found in that level of memory hierarchy, it is called as *miss*. It may take time to go to the next level, called as *miss penalty*. The fraction or percentage of accesses that result in a miss is called the miss rate, expressed as the following equation:

$$Miss\_rate = \frac{\#\ Misses}{\#\ memory\_accesses} = 1 - hit\_rate$$

It follows that the sum of the hit rate and the miss rate is equal to 1.0 (100%).

The system makes a copy from the virtual memory to the main memory with a unit of data, called as a *page*. The system makes a copy from the main memory to the cache memory with a unit of data, called as a *block*.

SRAM and DRAM Technologies

SRAMs are Integrated circuits that are memory arrays with (usually) a single access port. SRAM has a fixed access time to any datum. It doesn't need to refresh and so the access time is very close to the cycle time. Typically, it uses six to eight transistors per bit to prevent the information from being disturbed when reading the data. It costs a lot.

DRAMs store data as a charge in a capacitor, where a single transistor is used to access the charge. That's why it is much denser and cheaper per bit than SRAM. However, the data cannot be kept indefinitely and must periodically be refreshed, call 'dynamic'.

Table 12-1. DRAM Generations

Year introduced	Chip size	$ per *GB*	Total access time to a new row/column	Average column access time to existing row
1980	64 *Kbit*	$1,500,000	250 *ns*	150 *ns*
1983	256 *Kbit*	$500,000	185 *ns*	100 *ns*
1985	1 *Mbit*	$200,000	135 *ns*	40 *ns*
1989	4 *Mbit*	$50,000	110 *ns*	40 *ns*
1992	16 *Mbit*	$15,000	90 *ns*	30 *ns*
1996	64 *Mbit*	$10,000	60 *ns*	12 *ns*
1998	128 *Mbit*	$4,000	60 *ns*	10 *ns*
2000	256 *Mbit*	$1,000	55 *ns*	7 *ns*
2004	512 *Mbit*	$250	50 *ns*	5 *ns*
2007	1 *Gbit*	$50	45 *ns*	1.25 *ns*
2010	2 *Gbit*	$30	40 *ns*	1 *ns*
2012	4 *Gbit*	$1	35 *ns*	0.8 *ns*

[source] Computer Organization and Design, Fifth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)

The above table shows how DRAM generations gradually developed from 1980s.

Disk Storage

Disk storage (also sometimes called drive storage) is a general category of storage mechanisms where data is recorded by various electronic, magnetic, optical, or mechanical changes to a surface layer of one or more rotating disks.

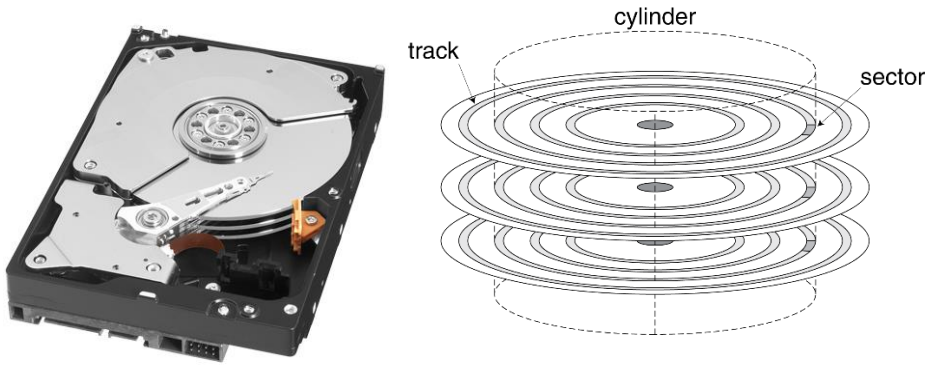


Fig. 12-5. Disk Storage

In the disk storage, a sector is a subdivision of a track on a magnetic disk. Each sector stores a fixed amount of user-accessible data, traditionally 512 bytes for hard disk drives (HDDs) and 2048 bytes for CD-ROMs and DVD-ROMs. The data area contains the sync bytes, user data and an error-correcting code (ECC) that is used to check and possibly correct errors that may have been introduced into the data.

Access to a sector involves

- Queuing delay if other accesses are pending
- Seek time: move the heads
- Rotational latency
- Data transfer
- Controller overhead

For example, disks rotate at 5400 RPM to 15,000 RPM. What is the average rotational latency at 5400 RPM?

The average rotational latency is calculated as follows: 0.5 rotation/5400 RPM = 0.5 rotation/(5400 rotation/60 seconds) = 30/5400 seconds = 5.6 *ms*

Exercises

1) A program has 2,000 load and store instructions. There exists 1,250 of these data values in cache and the rest of them supplied by other levels of memory hierarchy. What are the hit and miss rates for the cache?

Answer)

- Hit_rate = 1250/2000 = 0.625
- Miss_rate = 750/2000 = 0.375 = 1 – Hit_rate

2) Disk Access
   Given: 512B sector, 15,000 RPM, 1 ms average seek time, 100 MB/s transfer rate,
   0.2 ms controller overhead, idle disk.
   What is the average read time?

Answer)

- No queuing delay because of idle disk
- Seek time: 1 *ms*
- Rotational latency: 0.5/(15,000/60) = 2 *ms*
- Data transfer time: 512 B / 100 MB/s = 0.005 *ms*
- Controller overhead: 0.2 *ms*

The sum of the above items is 3.205 *ms*.

12.2. Cache Memory

Cache memory is located in the highest level of memory hierarchy. It is fast, and typically takes 1 clock cycle to access the data in the cache memory. Ideally it supplies most data to a processor. It usually holds most recently accessed data.

Caches first appeared in research computers in the early 1960s and in production computers later in that same decade. Every general-purpose computer built today, from servers to low-power embedded processors, includes caches.

When designing cache, the following questions are considered:

- What data is held in the cache?
- How is data found?
- What data is replaced?

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache.

Ideally, cache anticipates needed data and puts it in the cache memory, but it is impossible to predict the future demanding with perfect accuracy. Instead, the cache uses the past pattern to predict future demanding with temporal and spatial localities:

- Temporal locality: copy newly accessed data into cache
- Spatial locality: copy neighboring data into cache too

Before diving into the detail description, let's look at the cache terminology.

- Capacity (C): number of data bytes in cache
- Block size (b): bytes of data brought into cache at once
- Number of blocks (B = C/b): number of blocks in cache: B = C/b
- Degree of associativity (N): number of blocks in a set
- Number of sets (S = B/N): each memory address maps to exactly one cache set

The cache is organized into S sets. Each memory address maps to exactly one set. The caches are categorized by # of blocks in a set:

- Direct mapped: 1 block per set
- N-way set associative: N blocks per set
- Fully associative: all cache blocks in 1 set

We exemplified the cache parameters as follows:

- C = 8 words (capacity)
- b = 1 word (block size)
- So, B = C/b = 8 (# of blocks)

It is ridiculously small, but will illustrate organizations with these simple parameters in the next subsection.

Direct Mapped Cache

In the direct mapped cache, the cache memory assigns the location of the cache for each work based on the address of the word (block) in the main memory. Since there is only one choice to put the data of memory into the blocks of the cache memory, it is called as '*Directed mapped*', and the block number is calculated with the following modulo operation:

- (block address) modulo (# blocks in cache)

Let's find out where the data at addresses 0x00000004, 0x00000024,…, 0xFFFFFFE4 map to. The following figure illustrates a direct mapped cache with a capacity of eight words (C = 8 words) and a block size of one word (b = 1 word). The number of blocks in cache is a power of 2 (B = C/b = 8).

The cache has eight sets, each of which contains a one-word block. The two rightmost bits of the address are always 00, because they are word aligned. The next three rightmost bits ($\log_2 8 = 3$ bits) indicate the set (cache index) onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, . . . , 0xFFFFFFE4 map to the set number 1. Likewise, data at addresses 0x00000010 and 0xFFFFFFF0 map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

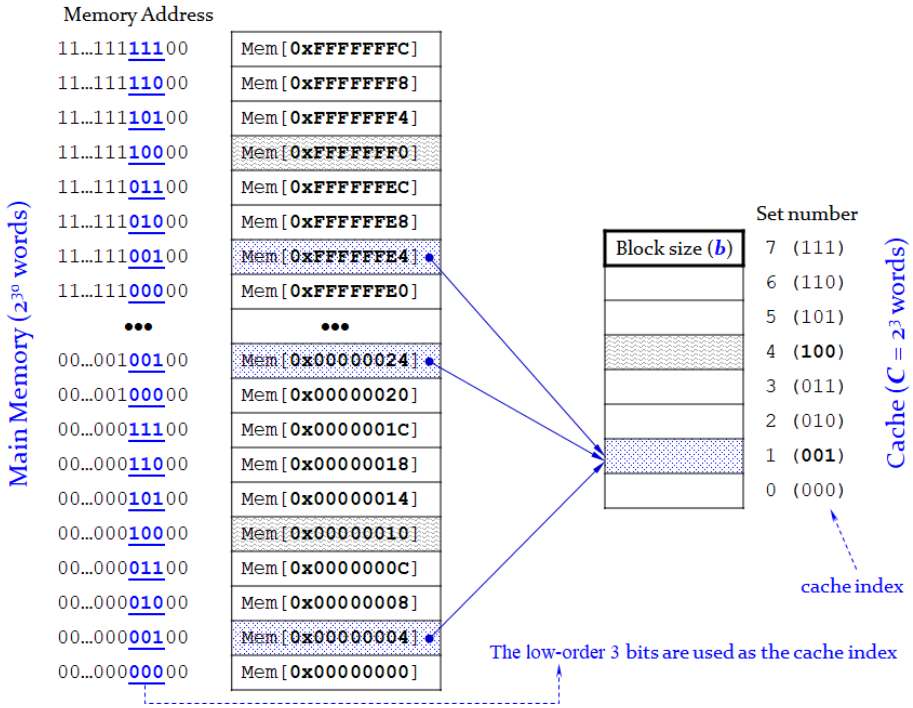


Fig. 12-6. Mapping of Main Memory to a Direct Mapped Cache

The following figure shows the direct mapped cache hardware. The cache is constructed with an eight-entry SRAM. Each entry, or set, contains one line consisting of 1 valid bit, 27 bits of tag, and 32 bits of data, as shown in the right side of Fig. 12-7. The cache is accessible using the 32-bit (memory) address that consists of the tag field (27 bits), the set bits (3 bits) and the byte offset (2 bits), as indicated in the top left of Fig. 12-7.

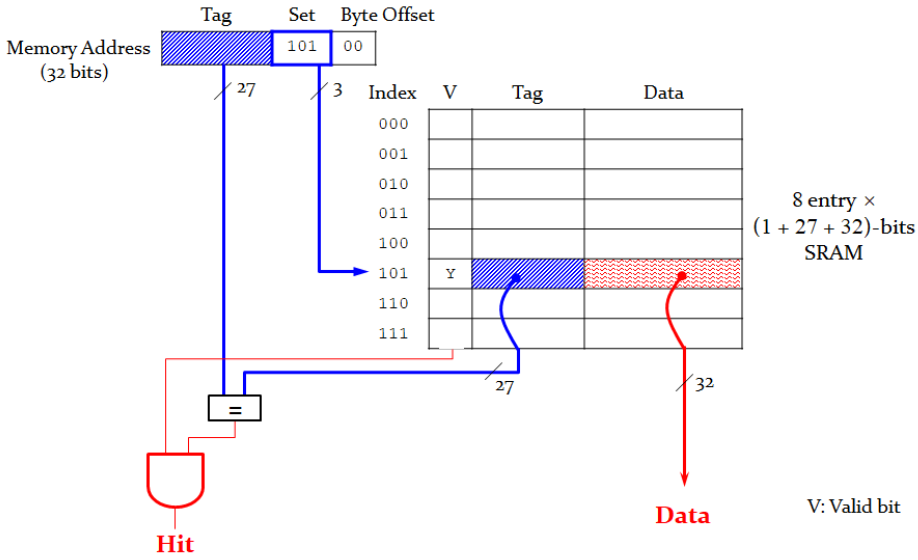


Fig. 12-7. Direct Mapped Cache Hardware

The set bits specify the entry or set in the cache. Using the set value, the system finds out the cache index number. The system compares two values, the tag value of the memory address and the tag value in the cache. If the two values are identical and the valid bit is set to '1', the memory system will get 'hit', and the data in the cache can be returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

The system knows whether a requested block is in the cache or not through tag values. The tags contain the address information required to identify whether a block (a word) in the cache corresponds to the requested block (word). The tag needs only to contain the upper portion of the address. Then what if there is no data in a location? The system indicates whether an entry contains a valid address through a *valid bit*. If the valid bit is one, there exists a valid address; otherwise there is no valid address in that entry. It is initially set to 0.

Let's look at how the cache memory is utilized when executing the following MIPS assembly codes:

```
 # MIPS assembly code

       addi $t0, $0, 5
loop:  beq  $t0, $0, done
       lw   $t1, 0x4($0)
       lw   $t2, 0xC($0)
       lw   $t3, 0x8($0)
       addi $t0, $t0, -1
       j    loop
done:
```

The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. We assume that the cache is initially empty. The first two instructions (`addi` and `beq`) require no memory access. Since the cache is initially empty, there is no data in the memory. The third instruction, `lw  $t1, 0x4($0)`, got missed, where the memory address consists of the tag (00…00), the set value (001) and the byte offset (00). The system makes a copy the data from memory and the entry of index number 1 is filled with the data including the valid bit (1) and the tag value (00…00), as shown below:

Table 12-2. Temporal Locality with a Direct Mapped Cache with `lw $t1, 0x4($0)`

V	Tag	Data	
0			Set 0 (**000**)
1	**00…00**	**Mem(0x00…04)**	Set 1 (**001**)
0			Set 2 (**010**)
0			Set 3 (**011**)
0			Set 4 (**100**)
0			Set 5 (**101**)
0			Set 6 (**110**)
0			Set 7 (**111**)

The fourth instruction, `lw $t2, 0xc($0)`, got missed because there is no data in the memory. The memory address of the instruction consists of the tag (00…00), the set value (011) and the byte offset (00). The system makes a copy the data from memory in the same way. The entry of index number 3 is filled with the data including the valid bit (1) and the tag value (00…00), as shown below:

Table 12-3. Temporal Locality with a Direct Mapped Cache with `lw $t2, 0xc($0)`

V	Tag	Data	
0			Set 0 (**000**)
1	00…00	Mem(0x00…04)	Set 1 (**001**)
0			Set 2 (**010**)
1	**00…00**	**Mem(0x00…0C)**	Set 3 (**011**)
0			Set 4 (**100**)
0			Set 5 (**101**)
0			Set 6 (**110**)
0			Set 7 (**111**)

The fifth instruction, `lw $t3, 0x8($0)`, got missed again because there is no data in the memory. The memory address of the instruction consists of the tag (00…00), the set value (010) and the byte offset (00). The system makes a copy the data from memory in the same way before. The entry of index number 2 is filled with the data including the valid bit (1) and the tag value (00…00), as shown below:

Table 12-4. Temporal Locality with a Direct Mapped Cache with `lw $t3, 0x8($0)`

V	Tag	Data	
0			Set 0 (**000**)
1	00…00	Mem(0x00…04)	Set 1 (**001**)
1	**00…00**	**Mem(0x00…08)**	Set 2 (**010**)
1	00…00	Mem(0x00…0C)	Set 3 (**011**)
0			Set 4 (**100**)
0			Set 5 (**101**)
0			Set 6 (**110**)
0			Set 7 (**111**)

No memory access is required for the last two instructions, `addi` and `j`. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. The processor jumps to the instruction line `loop`. However, the next four times the loop executes, the data is found in the cache. We can calculate the miss rate, 3/15 = 20%.

Now let's assume that we have the memory addresses `0x4` and `0x24` in a loop, as shown below:

- `0x4` : tag (00…00), the set value (001) and the byte offset (00).
- `0x24` : tag (00…01), the set value (001) and the byte offset (00).

Both memory addresses map to the set number 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict, and the miss rate is 100% in this case.

N-Way Set Associative Cache

An N-way set associative cache reduces conflicts by providing N blocks in each set, where data mapping to that set might be found. The following figure shows the hardware for a C = 8-word, N = 2-way set associative cache. The cache now has only S = 4 sets rather than 8. Thus, only $\log_2 4 = 2$ set bits are used to select the set.

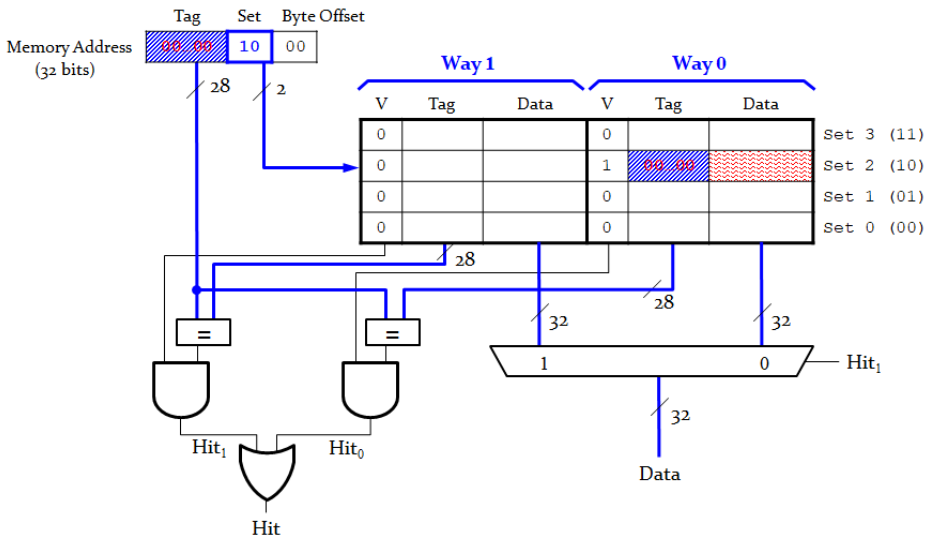


Fig. 12-8. 2-way Set Associative Cache

The number of tag bits increases from 27 to 28 bits. 2-way set associative cache has two options to store the tag value and the data. Each way consists of a data block and the valid and tag bits. When the memory address is searched as noticed in the top left of the above figure, the tag value in memory address is compared with the tag value in the cache. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Let's look at how the 2-way set associative cache is utilized when executing the following MIPS assembly codes:

```
 # MIPS assembly code

       addi $t0, $0, 5
loop:  beq  $t0, $0, done
       lw   $t1, 0x4($0)
       lw   $t2, 0x24($0)
       addi $t0, $t0, -1
       j    loop
done:
```

The program contains a loop that repeats for five iterations. Each iteration involves two memory accesses (loads), resulting in 10 total memory accesses. We assume that the cache is initially empty.

The first load instruction, `lw $t1, 0x4($0)`, got missed because there is no data in the cache memory. The memory address of the instruction consists of the tag (00…00), the set value (01) and the byte offset (00). The system makes a copy the data from memory to cache and the way 0 entry of the set number 1 (`01`) is filled with the data including the valid bit (1) and the tag value (00…00), as shown below:

Table 12-5. 2-way Set Associative Cache with `lw $t1, 0x4($0)`

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3 (11)
0			0			Set 2 (10)
0			**1**	**00…00**	**Mem[0x00…04]**	Set 1 (01)
0			0			Set 0 (00)

The second load instruction, `lw $t2, 0x24($0)`, got missed again because there is no data in the memory. The memory address of the instruction consists of the tag (00…10), the set value (01) and the byte offset (00). The system makes a copy the data from memory to cache in the same way. In this time, the way 1 entry of the set number 1 is filled with the data including the valid bit (1) and the tag value (00…10), as shown below:

Table 12-6. 2-way Set Associative Cache with `lw $t2, 0x24($0)`

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3 (11)
0			0			Set 2 (10)
1	**00…10**	**Mem[0x00…24]**	1	00…00	Mem[0x00…04]	Set 1 (01)
0			0			Set 0 (00)

Both memory accesses, to addresses 0x4 and 0x24, map to the set number 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of the set number 1. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%.

Full Associative Cache

A fully associative cache allows a given block to go in any cache entry. The cache is expensive to build because it requires all entries to be searched at once. But it can reduce conflict misses.

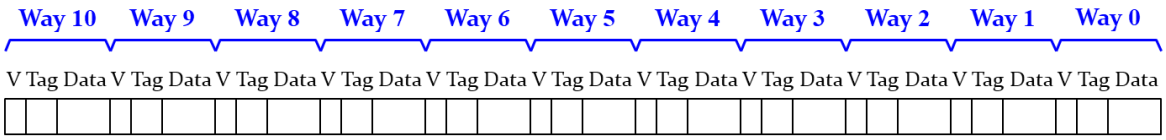


Fig. 12-9. Fully Associative Cache

Exercises

1) The modulo operation finds the remainder after division of one number by another. For example, 5 modulo 2, where 5 is the dividend and 2 is the divisor, would evaluate to 1 because 5 divided by 2 leaves a quotient of 2 and a remainder of 1. What are the results of the following modulo operations?
   - 9 modulo 3 =
   - 000012 modulo 23 =
   - 100012 modulo 23 =

Chapter 13: Virtual Memory

In this chapter, we introduce virtual memory. Most modern computer systems use a hard driver made of magnetic or solid-state storage as the lowest level in the memory hierarchy. The virtual memory is located in the lowest level of the memory hierarchy while still provide the speed of faster memory for most accesses. Processors can access data anywhere using virtual addresses that specify the location in virtual memory. We also introduce virtual memory definitions and show how to translate the virtual address into the physical address.

Objectives

By the end of this chapter you should be able to:

- Differentiate virtual and physical addresses
- Identify the difference between virtual memory analogue and cache memory analogue
- Recognize the address translation in virtual address
- Carry out the address translation from virtual address to physical address
- Demonstrate knowledge of page table
- Identify features of translation lookaside buffer

13.1. Virtual Memory Address

There are three memory types in the memory hierarchy, i.e. cache, main memory, virtual memory, as shown in the following figure. As we discuss in the cache memory, the cache memory (SRAM) is fast but it can only keep small amount of data because it is very expensive. The virtual memory gives the illusion of bigger memory. Ideally, we have no limitation to store data in the virtual memory, where the main memory (DRAM) acts as cache for hard disk. That means we can make a copy a chunk of data from virtual memory to main memory.

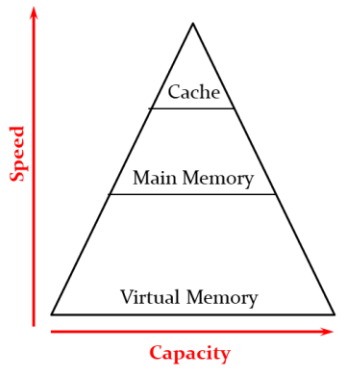


Fig. 13-1. Memory Hierarchy Pyramid

The following table describes the cache and virtual memory analogues.

Table 13-1. Cache/Virtual Memory Analogues

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

When the system makes a copy data from main memory to cache memory, it does in a unit of ***block***, where cache memory is exactly a memory unit. When the system makes a copy data from virtual memory to main memory, it does in a unit of ***page***. Virtual memory is not a memory unit, it is a technique. The page size is the amount of memory transferred from hard disk (Virtual memory) to DRAM at once. The typical page size is between 1 KB and 8 KB and is generally 4 KB for 32-bit systems. The page number is the number of bits required to represent the pages in Virtual Address Space, where the page offset is the number of bits required to represent particular word in a page or page size of Virtual Address Space or word number of a page.

The data is stored in virtual memory and processors use virtual addresses when they execute. The entire virtual address space is stored on a hard drive and only subset of virtual address data moves in physical memory (DRAM). Accordingly, CPU translates virtual addresses into physical addresses (DRAM addresses) so that it can find the physical location of data in DRAM. If data is not in DRAM fetched from hard drive, it is called "***page fault***", which is a similar concept of "miss". As shown in the following figure, the ***address translation*** determines physical address from virtual address, where the ***page table*** is used as a lookup table to translate virtual addresses to physical addresses.

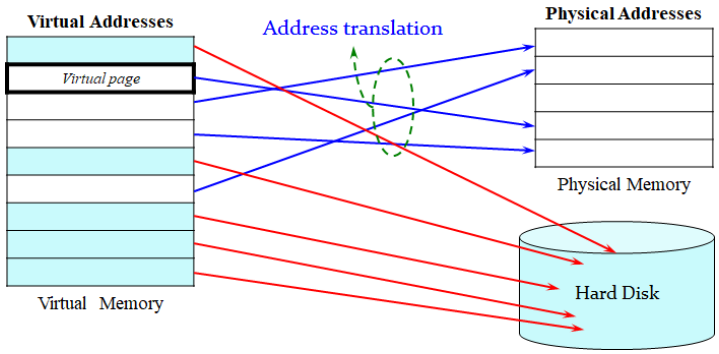


Fig. 13-2. Virtual and Physical Addresses

As shown in the above figure, Virtual memory is divided into virtual pages, typically 4 KB in size. Physical memory is likewise divided into physical pages of the same size (4 KB). A virtual page may be located in

physical memory (DRAM) or on the disk. Some virtual pages are present in physical memory, and some are located on the disk. The process of determining the physical address from the virtual address is called ***address translation***. When we execute a program, we expect that most memory accesses got hit in physical memory. But what if the program size is bigger than DRAM size? In this case, we cannot move all the programs to the physical memory. The programs can have the large capacity in virtual memory. The programmer no longer needs to worry about the amount of physical memory available. All the programs will be stored in the virtual memory with virtual memory address. The system only makes a copy required data from virtual memory to physical memory.

Address Translation

The following figure illustrates how to translate a virtual address to a physical address. In this example, we assume that the system has the following specification:

- Virtual memory size: 2 GB = 2^{31} bytes
- Physical memory size: 128 MB = 2^{27} bytes
- Page size: 4 KB = 2^{12} bytes

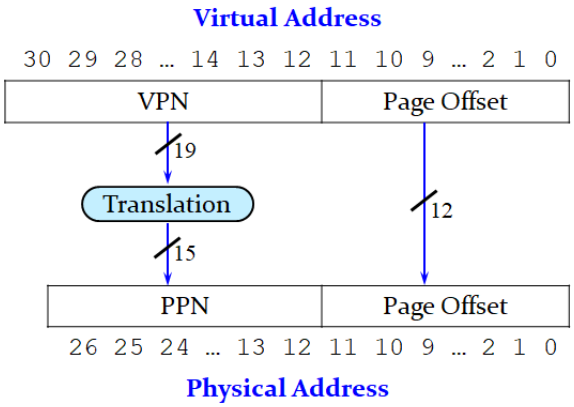


Fig. 13-3. Address Translation

In the figure above, the least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the virtual page number (VPN) and are translated to a 15-bit physical page number (PPN).

We can extract the following values for the give system:

- Virtual address: 31 bits
- Physical address: 27 bits
- Page offset: 12 bits = 3 hexes
- # Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = 19 bits)
- # Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = 15 bits)

The following figure shows the virtual page number 5 mapping to the physical page number 1, virtual page number 0x7FFFC mapping to physical page number 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

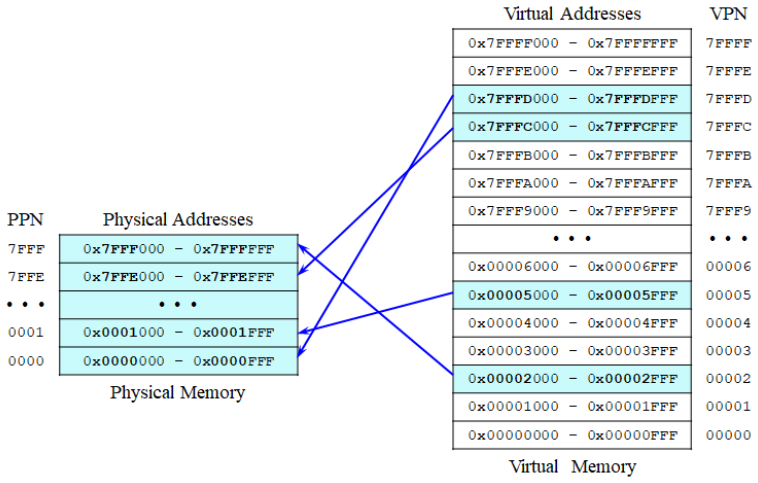


Fig. 13-4. VPN Mapping to PPN

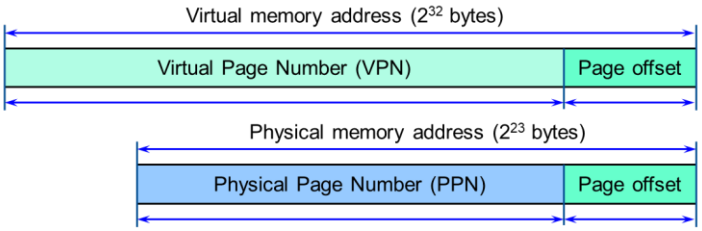
Exercises

1) Let's assume we have the virtual memory system with the given Fig. 13-4. What is the physical address of the virtual address `0x0000247C`?

Answer)

- VPN = `0x00002`
- VPN `0x00002` maps to PPN `7FFF`
- 12-bit page offset = `0x47C`
- Physical address = `0x7FFF47C`

2) Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space, but are limited to only 8 MB of semiconductor (Physical) memory. Assume that virtual and physical pages are each 4 KB in size. Configuration of the virtual and physical memory addresses, as follows:



The virtual memory address consists of virtual page number (VPN) and page offset. The physical memory address consists of physical page number (PPN) and page offset.

The lengths of page offset are same.

The total number of the virtual page: $2^{32} / 2^{12} = 2^{20}$. That means a total of 20 bits are used for the virtual page number.

The total number physical page: 8 MB / 4 KB = $2^{23} / 2^{12} = 2^{11}$. That means a total of 11 bits are used for the physical page number.

13.2. Page Table

Let's look at how to perform translation with the page table. The page table has the entry for each virtual page, where entry fields have the following information:

- Valid bit (V): set to 1 if the page is in physical memory
- Physical page number (PPN): where the page is located in the main memory

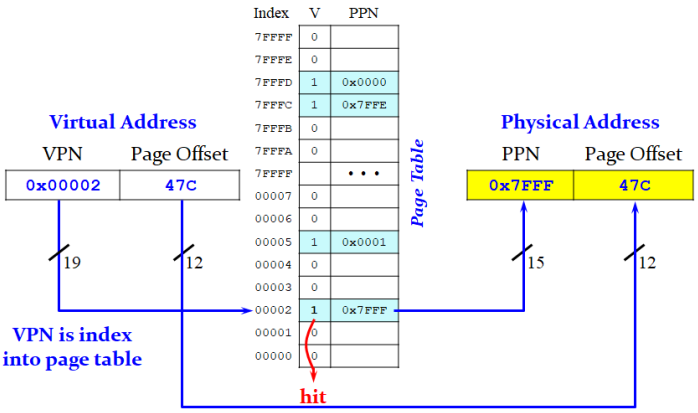


Fig. 13-5. Page Table Translation

As shown in left side of Fig. 13-5, the virtual address consists of virtual page number (VPN) and page offset. The VPN `0x00002` is translated into physical page number (PPN) `0x7FFF` using the given page table. If the PPN `0x7FFF` is already in the page table and the valid bit set to 1, then the system will get "hit". The PPN `0x7FFF` in page table is mapped to the PPN in physical address. The page offset of virtual address is directly translated into the page offset of physical address.

Translation Lookaside Buffer (TLB)

The page table is large and is usually located in physical memory. If a processor executes load or store instruction, the system requires two accesses of the main memory:

- one for translation (page table read)
- another to access data (after translation)

These accesses eventually degrade the memory performance in half, unless we get clever way to access the memory.

Translation Lookaside Buffer (TLB) is small cache of most recent translations, and reduces the number of memory accesses for most loads/stores from 2 to 1.

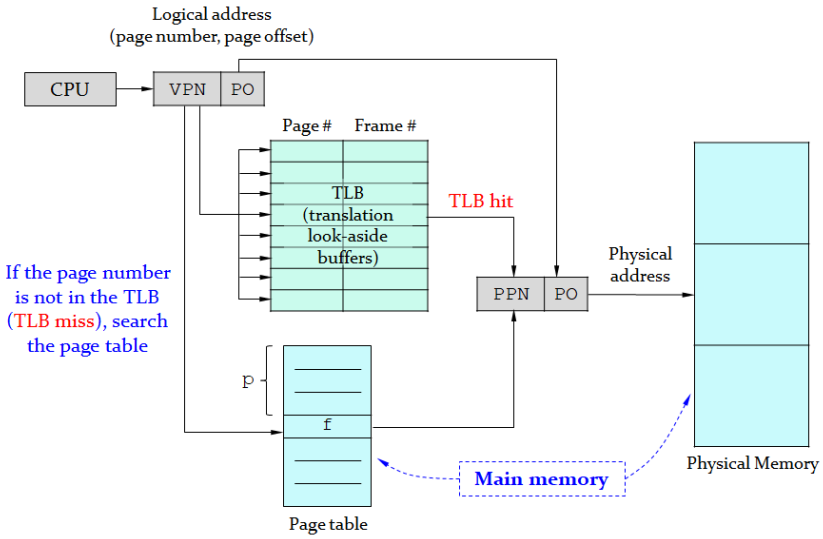


Fig. 13-6. Paging Hardware With TLB

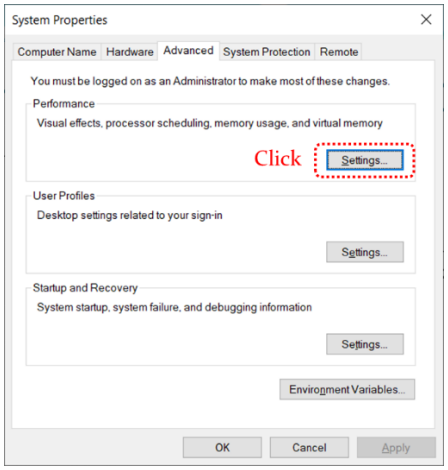
The CPU only looks at the virtual (logical) address which consists of VPN and PO. If the corresponding PPN is already in Translation look-aside buffers, the system will get TLB hit. VPN is directly translated into PPN using TLB located within CPU. In this case, only one memory access is required.

If the page number is not in the TLB (TLB miss), the system searches the page table which is located in the main memory. After getting PPN in the main memory, it can translate VPN into PPN, and then access the data located in the memory. In this case, two memory accesses are required.

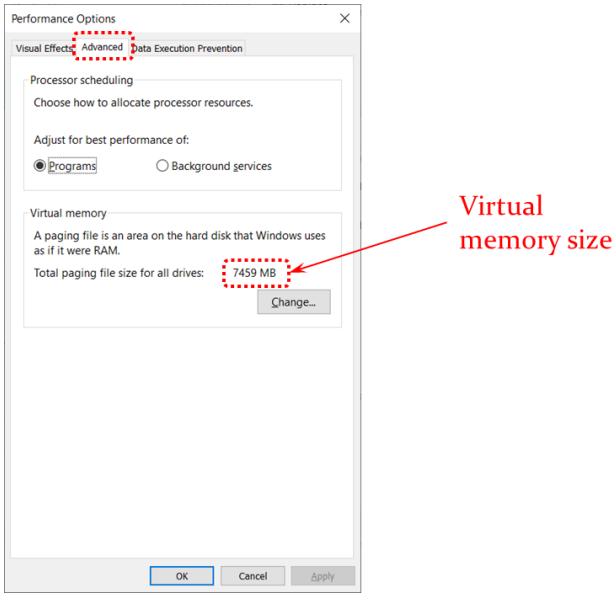
When we run multiple processes (programs) at once, each process has its own page table. Each process can use entire virtual address space. A process can only access physical pages mapped in its own page table.

Virtual Memory Settings

In the Window Search window, type “Advanced System Settings” and click it. You can see the following figure. Then Click “Setting”.



In the System Properties window, click the Advanced tab. You can check the virtual memory size.

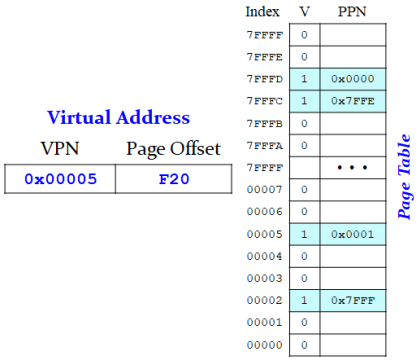


Virtual Memory for 64-bit versions of Windows

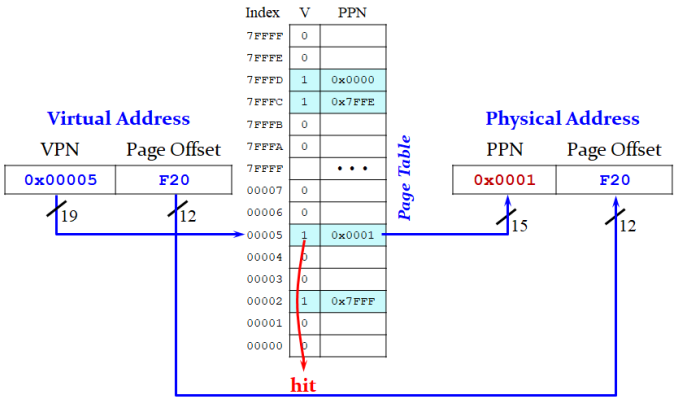
- How to determine the appropriate page file size for 64-bit versions of Windows

Exercises

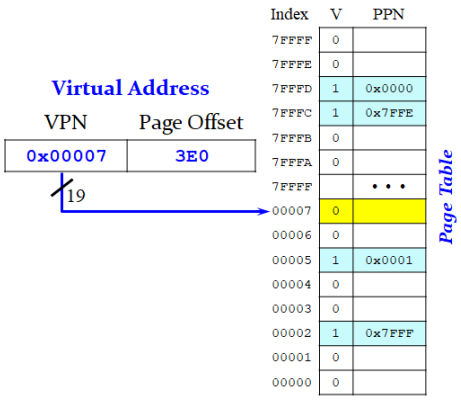
1) What is the physical address of virtual address `0x00005F20` with the given page table? (assume 12-bit page offset)



Answer) The least significant 12 bits of the virtual and physical addresses are the same (`0xF20`) and specify the page offset both the virtual and physical pages. According, the virtual page number 5 is mapping to the physical page number 1 in the page table. The virtual address `0x00005F20` is translated into the physical address `0x0001F20`, as shown below:



2) What is the physical address of virtual address `0x000073E0` with the given page table? (assume 12-bit page offset)



Answer) The least significant 12 bits of the virtual specify the page offset. According, the virtual page number (VPN) is 7. The corresponding physical page number is not valid (blank) in the page table. If the processor attempts to access a virtual address that is not in physical memory, a ***page fault*** occurs, and the operating system loads the page from the hard disk into physical memory.

3) What is the physical address of virtual address `0x7FFFCA20` with the given page table? (assume 12-bit page offset)