CHAPTER 2

COMPUTER ORGANIZATION AND ARCHITECTURE

Syllabus: Machine instructions and addressing modes, ALU and data path, CPU control design, memory interface, I/O interface (interrupt and DMA mode), instruction pipelining, cache and main memory, secondary storage

2.1 INTRODUCTION

Computer architecture and organization is the science of interconnecting hardware components, designing and configuring the hardware/software interface to fulfill functional and performance goals of a computer. This chapter outlines the basic hardware structure of a modern digital programmable computer, the basic laws for performance evaluation, designing the control and data path hardware for a processor, concept of pipelining for executing machine instructions simultaneously and designing fast memory and storage systems.

Computer architecture deals with the structure and behaviour of computer system as viewed by the user. It encompasses instruction formats, the instruction set architecture (ISA) and addressing modes.

Computer organization deals with the operation and interconnection of the various hardware components.

2.2 COMPUTER ARCHITECTURE

2.2.1 Register Set

The computer needs registers for processing and manipulating data and for holding memory addresses that are available to the machine-code programmer. Some registers for a basic computer are given in Table 2.1.

Table 2.1 Types of registers and their functions

Register Symbol	Register Name	Function
DR	Data register	Holds memory operand
ACC	Accumulator	Special purpose processor register
AR	Address register	Holds address for memory

(*Continued*)

Table 2.1 | Continued

Register Symbol	Register Name	Function
IR	Instruction register	Holds an instruction that is to be executed
PC	Program counter	Holds address of instruction to be executed next
TR	Temporary register	Holds temporary data if required
INPR	Input register	Holds input character
OUTR	Output register	Holds output character

2.2.2 Quantitative Principles to Design High-Performance Processor

Amdahl's law focused on performance gain after enhancing the system. The performance gain is denoted by S_{overall} and ET stands for execution time.

$$S_{\text{overall}} = \frac{\text{Performance of the system with enhancement}}{\text{Performance of the system without enhancement}}$$

$$S_{\text{overall}} = \frac{1/\text{ET}_{\text{new}}}{1/\text{ET}_{\text{old}}} \tag{2.1}$$

$$S_{\text{overall}} = \frac{\text{ET}_{\text{old}}}{\text{ET}_{\text{new}}}$$

After enhancement, the system consists of two portions: unenhanced and enhanced portion.

$$\text{ET}_{\text{new}} = \text{ET of the unenhanced portion} + \text{ET of enhanced portion}$$

To calculate ET_{new}, the following two factors are needed:

1. **Fraction$_{\text{enhance}}$ (F):** It indicates how much portion of the old system undergoes enhancement.
2. **Speed$_{\text{enhance}}$ (S):** It indicates how many times the new portion is running faster than the old portion.

$$S = \frac{\text{Performance}_{\text{new}} F}{\text{Performance}_{\text{old}} F} = \frac{1/\text{ET}_{\text{new}} F}{1/\text{ET}_{\text{old}} F} = \frac{\text{ET}_{\text{old}} F}{\text{ET}_{\text{new}} F}$$

So, $$\text{ET}_{\text{new}} F = \frac{\text{ET}_{\text{old}} F}{S}$$

On the basis of the above factor,

$$\text{ET}_{\text{new}} F = \text{ET}_{\text{old}}(1-F) + \frac{\text{ET}_{\text{old}} F}{S}$$

Substitute the value of ET in Eq. (2.1):

$$\text{ET}_{\text{new}} F = \text{ET}_{\text{old}}(1-F) + \frac{\text{ET}_{\text{old}} F}{S}$$

Let $\text{ET}_{\text{old}} = 1$,

$$S_{\text{overall}} = \frac{1}{(1-F) + (F/S)} = \left[(1-F) + \frac{F}{S}\right]^{-1}$$

When the system consists of multiple frequent cases, where i is the number of frequent cases:

$$S_{\text{overall}} = \left[\left(1 - \sum F_i\right) + \sum \frac{F_i}{S}\right]^{-1}$$

Problem 2.1: Consider a hypothetical processor used in mathematical model simulation. It consists of two functional units, floating point and integer. The floating point is enhanced then it runs two times faster, but only 10% of the instructions are floating point. What is the speed up?

Solution: Here $S = 2$, $F = 0.1$

$$S_{\text{overall}} = \left[(1-0.1) + \frac{0.1}{2}\right]^{-1} = 1.052$$

2.3 MACHINE INSTRUCTIONS AND ADDRESSING MODES

Machine instruction is an individual machine code. The complete set of all machine codes recognized by a particular processor makes its Instruction Set. Instructions can be grouped according to the function they perform. The number of ways by which arguments for these machine instructions can be specified constitutes the addressing modes for a processor.

2.3.1 Machine Instructions

An instruction is a command to the microprocessor to perform a given task. Most computer instructions are classified as follows:

1. **Data transfer instructions:** These instructions move data from one place to another in the computer without changing the data content. Example: LOAD, MOVE, IN, OUT, PUSH, STORE.
2. **Data manipulation instructions:** These instructions perform arithmetic, logical and shift operations on data. Example: ADD, SUB, MUL, DIV, INC, AND, XOR, OR, SHR, SHL, ROR, ROL.
3. **Program control instructions:** These instructions may change the address value in program counter and cause the normal sequential flow to change.

On the basis of the number of address fields in an instruction, they are classified as follows:

1. **Three-address instruction:** Computer with three-address instruction format can use each address field to specify two sources and a destination, which can be either a processor register or a memory operand. It results in short program but requires too many bits to specify three addresses.

   Example: ADD R_1, A, B $\quad (R_1 \leftarrow M[A] + M[B])$

2. **Two-address instruction:** Each address field can specify either a processor register or a memory word.

   Example: MOV R_1, A $(R_1 \leftarrow M[A])$;
   MUL R_1, R_2 $(R_1 \leftarrow R_1{*}R_2)$

3. **One-address instruction:** It used an implied accumulator (AC) register for all data manipulation. The other operand is in register or memory.

   Example: LOAD A $(AC \leftarrow M[A])$;
   ADD B $(AC \leftarrow AC + M[B])$

4. **Zero-address instruction:** A stack organized computer does not use an address field for the instruction ADD and MUL.

Problem 2.2:

(a) ISA of a processor consists of 64 registers, 125 instructions and 8 bits for immediate mode. In a given program, 30% of the instructions take one input register and have one output register, 30% have two input registers and one output register, 20% have one immediate input, and one output register, and remaining have two immediate input, 1 register input and one output register. Calculate the number of bits required for each instruction type. Assume that the ISA requires that all instructions be a multiple of 8 bits in length.

(b) Compare the memory space required with that of variable length instruction set.

Solution:

(a) Since there are 125 instructions so we need 7 bits to differentiate them as $64 < 125 < 128$. For 64 registers, we need 6 bits and 8 bits for immediate mode.
For Type 1, 1 reg in, 1 reg out: $7 + 6 + 6 =$ 19 bits ~ 32 bits
For Type 2, 2 reg in, 1 reg out: $7+ 6 + 6 + 6 =$ 26 bits ~ 32 bits
For Type 3, 1 imm in, 1 reg out: $7 + 6 + 8 =$ 21 bits ~ 32 bits
For Type 4, reg in, 2 imm in, 1 reg out: $7 + 6 + 8 + 8 + 6 = 35$ bits ~48 bits

(b) As the largest instruction type requires 48 bit instructions, the fixed-length instruction format uses 48 bits per instruction. Variable length instruction format uses $0.3 \times 32 + 0.3 \times 32 + 0.2 \times 32 + 0.2 \times 48 = 36 =$ bits on average, that is, 25% less space.

2.3.2 Addressing Modes

The addressing mode specifies how effective address of an operand is calculated from an instruction. Computers use various addressing mode techniques to:

1. provide programming flexibility to users through use of pointers to memory, counter for loop control, data indexing and program relocation.
2. reduce the size of the addressing field of the instruction.

Let us suppose $[x]$ means contents at location x for all the addressing modes.

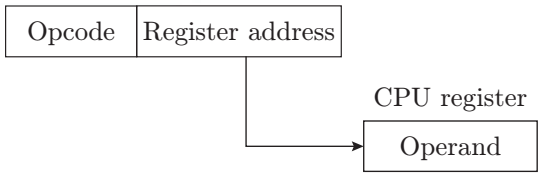
2.3.2.1 Types of Addressing Modes

1. **Implied mode:** In this mode, the operands are implicitly stated in the instruction. For example, register reference instructions such as CMA (complement accumulator), CLA (clear accumulator) and zero-address instructions that use stack organization.
2. **Immediate mode:** In this mode, the operand is specified in the instruction itself, that is, address field is replaced by an actual operand. Immediate mode instructions are useful for initializing registers to a constant value. For example, used for initializing CPU registers to some constant value such as MOV R_1, #34.

   Instruction with immediate mode

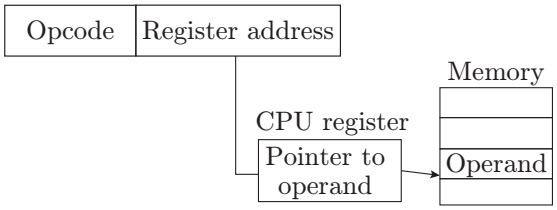
3. **Register (direct) mode:** In this mode, the operands are in CPU registers. An n-bit register field can specify any one of 2^n registers. Example: ADD R_1 will add the contents of an accumulator and contents of R_1, that is, ACC = [ACC] + [R_1].

   Instruction with register direct mode



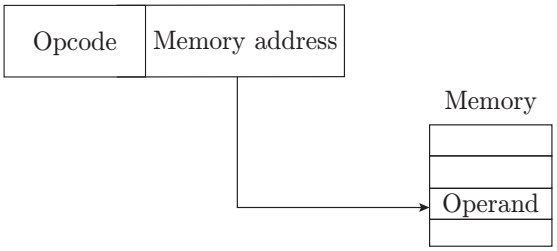
4. **Register (indirect) mode:** In this mode, the instruction format specifies a CPU register which contains an effective address of the operand residing in memory. This mode ensures less number of bits to specify a register value than to specify a memory location. Example: ADD @ R_1 will add the contents of an accumulator with contents of the register R_1, that is, AC = [ACC] + [[R_1]].

   Instruction with register indirect mode



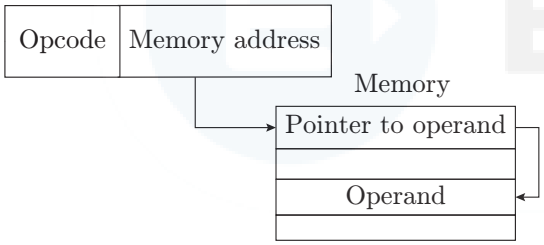
5. **Auto-increment or Auto-decrement mode:** This is similar to register indirect mode except the register containing effective address is incremented or decremented after (or before) its value is used to access memory.
6. **Direct address mode:** In this mode, the effective address of an operand is equal to the address part of the instruction. Example: ADD A instruction adds content of memory cell A to accumulator, that is, ACC = [ACC] + $M[A]$.

   Instruction with direct address mode



7. **Indirect address mode:** In this mode, memory address specified by address field contains the address of (pointer to) the operand. Example: ADD @ A will add the contents of the memory cell A, that is, ACC = [ACC] + $M[M[A]]$.

   Instruction with indirect address mode



8. **Relative address mode:** In this mode, the effective address of an operand is obtained by adding the content of a program counter to the address part of the instruction. The address part of the instruction can be either positive or negative represented in 2's complement. The result obtained after adding the content of the program counter to the address field produces an effective address whose position in memory is relative to the address of the next instruction.
9. **Index address mode:** In this mode, the effective address of an operand is obtained by adding the content of an index register to the address part of the instruction. The index register is a special CPU register that stores an index value and the address field of the instruction stores the base address of a data array in the memory. The distance between the base address and the address of the operand is the index value that is stored in the index register. The index register can be incremented to facilitate access to consecutive operands stored in arrays using the same instruction.
10. **Base register addressing mode:** In this mode, the effective address of an operand is obtained by adding the content of a base register to the address part of the instruction. This is somewhat similar to the indexed addressing mode except that the base register stores base or beginning address instead of an index register. It is used for program relocation.

Problem 2.3: A two-word instruction LOAD is stored at location 300 with its address field in the next location. The address field has value 600 and value stored at 600 is 500 and at 500 is 650. The words stored at 900, 901 and 902 are 400, 401 and 402, respectively. A processor register R contains the number 800 and index register has value 100. Evaluate the effective address and operand if addressing mode of the instruction is as follows:

1. Direct
2. Indirect
3. Relative
4. Immediate
5. Register indirect
6. Index

Solution: Memory layout is as follows

300	LOAD
301	600
500	650
600	500
700	900
800	700
900	400
901	401
902	402

Addressing Mode	Effective Address	Operand
Direct	600	500
Indirect	500	650
Relative	902	402
Immediate	301	600
Register indirect	800	700
Index	700	900

Problem 2.4: A relative mode branch type instruction is stored in memory at an address equivalent to decimal 600 and the branch is made to an address equivalent to decimal 400. What is the value of the relative address field of the instruction (in decimal)?

Solution: Relative address = 400 − 601 = −201

Table 2.2 | Arithmetic circuit function table

Select			Input to Adder Y	Output of Binary Adder $D = A + Y + C_{in}$	Micro-Operation
S_1	S_0	C_{in}			
0	0	0	0	$D = A$	Transfer A
0	0	1	0	$D = A + 1$	Increment A
0	1	0	B	$D = A + B$	Add
0	1	1	B	$D = A + B + 1$	Add with carry
1	0	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
1	0	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

2.4 ARITHMETIC LOGIC UNIT

Arithmetic logic unit (ALU) is a combinational circuit that performs all arithmetic and logic operations so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

2.4.1 Arithmetic Micro-Operations

The basic arithmetic micro-operations such as addition, subtraction, increment, decrement and shift are performed on numeric data stored in registers. The basic component of arithmetic is parallel binary adder, and by controlling the input to adder, different micro-operations can be realized. Figure 2.1 depicts a 2-bit arithmetic circuit which includes two full-adder circuits and two multiplexers for choosing different arithmetic micro-operations. There are two 2-bit input numbers A and B and 2-bit output D. The two inputs from A go directly to X inputs of full adder. The output of multiplexer goes to input Y of full adder.

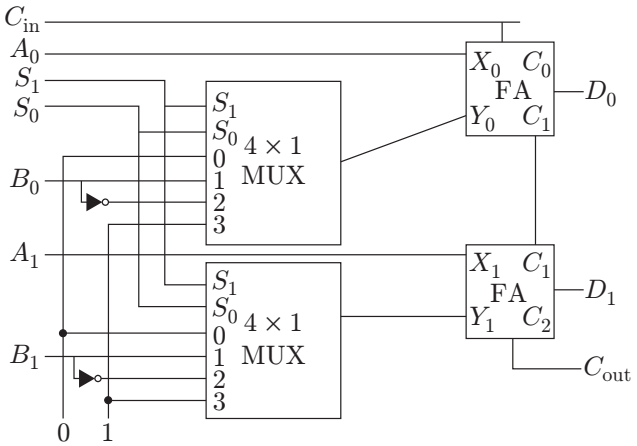


Figure 2.1 | A 2-bit arithmetic circuit.

By controlling the output Y of multiplexers with two selection inputs S_1 and S_0 and C_{in} either 0 or 1, we can generate the eight arithmetic micro-operations (Table 2.2).

2.4.2 Logic Micro-Operations

Logic micro-operations such as AND, OR, Exclusive OR, etc., consider each bit of register separately and specify binary operations for strings of bits (Table 2.3).

Table 2.3 | Types of micro-operations

Micro-operation	Name
$F \leftarrow 0$	Clear
$F \leftarrow A \wedge B$	AND
$F \leftarrow A \wedge \overline{B}$	
$F \leftarrow A$	Transfer A
$F \leftarrow \overline{A} \wedge B$	
$F \leftarrow B$	Transfer B
$F \leftarrow A \oplus B$	Exclusive OR
$F \leftarrow A \vee B$	OR
$F \leftarrow \overline{A \vee B}$	NOR
$F \leftarrow \overline{A \oplus B}$	Exclusive NOR
$F \leftarrow \overline{B}$	Complement B
$F \leftarrow A \vee \overline{B}$	
$F \leftarrow \overline{A}$	Complement A
$F \leftarrow \overline{A} \vee B$	
$F \leftarrow \overline{A \wedge B}$	NAND
$F \leftarrow$ all 1's	Set to all 1's

Logical micro-operations are capable of manipulating individual bits or a portion of word stored in CPU registers. Let us consider the data in a register A. In another register, B is the operand that will be used to modify the contents of A using logic micro-operations. Some of the applications are as follows:

1. **Selective set operation:** In this, If a bit in B is set to 1, that same position in A sets to 1, otherwise that bit in A retains its previous value.

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 0 \end{array} \quad \begin{array}{l} A_t \\ B \text{(To set some bits in } A) \\ A_{t+1}(A \leftarrow A + B) \end{array}$$

2. **Selective complement operation:** If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it remains unchanged.

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 \end{array} \quad \begin{array}{l} A_t \\ B \text{(To complement some bits in } A) \\ A_{t+1}(A \leftarrow A \oplus B) \end{array}$$

3. **Selective clear operation:** If a bit in B is set to 1, that same position in A sets to 0, otherwise it remains unchanged.

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \end{array} \quad \begin{array}{l} A_t \\ B \text{(To clear some bits in } A) \\ A_{t+1}(A \leftarrow A \cdot B') \end{array}$$

4. **Mask operation:** If a bit in B is set to 0, that same position in A sets to 0, otherwise it remains unchanged.

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array} \quad \begin{array}{l} A_t \\ B \text{(To clear some bits in } A) \\ A_{t+1}(A \leftarrow A \cdot B) \end{array}$$

5. **Clear operation:** If the bits in the same position in A and B are the same, they are cleared in A, else they are set in A.

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 \end{array} \quad \begin{array}{l} A_t \\ B \\ A_{t+1}(A \leftarrow A \oplus B) \end{array}$$

6. **Insert operation:** It is used to insert a specific bit pattern into A register, leaving the other bit positions unchanged. This is accomplished by two sub-operations: masking operation to clear the desired bit positions, followed by OR operation to introduce the new bits into the desired positions. Suppose you wanted to introduce 10 into the low order two bits of A:

1101 A (Original) and 1110 A (Desired)

$$\begin{array}{cccc} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 0 \end{array} \quad \begin{array}{l} A \text{ (Original)} \\ \text{Mask} \\ A \text{(Intermediate)} \\ \text{Added bits} \\ A \text{ (Desired)} \end{array}$$

2.5 CPU CONTROL DESIGN

Central processing unit (CPU), or the brain of a computer, performs the data processing operations. It consists of three major parts: register set that stores intermediate data during instruction execution, ALU performs the required micro-operations and control unit that supervises all other elements for the transfer of information from one register to the other. The main function of a CPU is to fetch an instruction from the memory and execute it. CPU is divided into three types of organizations:

1. **Single accumulator organization:** In this, one operand is implied in the accumulator, a special purpose register, and the other operand is a register or the memory. Example: ADD R_1 ($R_1 \leftarrow$ AC + R_1), LOAD A (AC $\leftarrow A$), STORE T ($M[T] \leftarrow$ AC).
2. **General register organization:** In this, the CPU will have several general purpose registers which lead to shorter and efficient programs because registers are faster. Example: ADD R_1, R_2 ($R_1 \leftarrow R_1 + R_2$). Figure 2.2 shows bus organization for three registers R_1, R_2 and R_3. The output of these registers and one from the external input is connected to two multiplexers A and B. The two

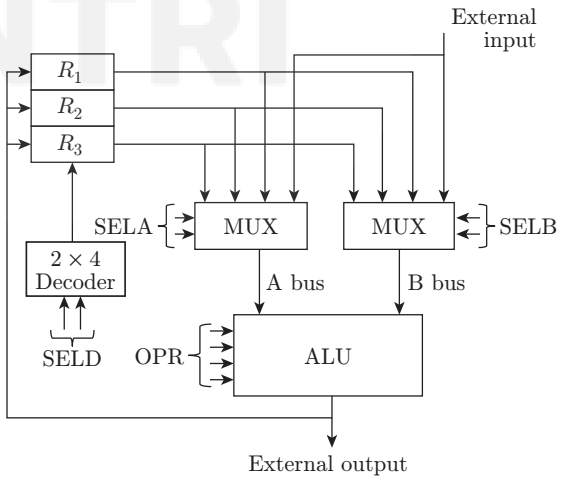


Figure 2.2 | General register organization.

select lines SELA and SELB from multiplexers A and B select one of the input and feed to ALU. OPR specifies one of the possible operation codes that ALU will perform on the data inputs and the output is transferred either to one of the registers using 2 × 4 decoder or to the external output say memory. The control word (Fig. 2.3) for the two-operand instruction is as follows:

2 bits	2 bits	2 bits	4 bits
SELA	SELB	SELD	OPR

Figure 2.3 | A control word.

3. **Stack organization:** Stack may consist of number of registers or a part of main memory in which data items are stored in consecutive locations that are accessed by LIFO (last in, first out) mechanism. As there is limited number of registers, a part of memory is implemented as stack for storage and retrieval of intermediate data. Stack pointer (SP) keeps a track of the top item of a stack. The process of inserting a new item onto a stack is known as push accomplished by first incrementing stack pointer and then inserting an item from the data register.

$$\text{SP} \leftarrow \text{SP} + 1$$

$$M[\text{SP}] \leftarrow \text{DR}$$

The process of removing an item from the top of a stack is known as pop performed by first transferring data into DR and then decrementing SP.

$$\text{DR} \leftarrow M[\text{SP}]$$

$$\text{SP} \leftarrow \text{SP} - 1$$

Problem 2.5: A system has CPU organized in the form of general register organization consisting of 16 registers, each storing 32-bit data. Assume the ALU has 35 operations.

(a) How many multiplexers are there in A bus and B bus, and what is the size of each multiplexer?
(b) How many selection inputs are needed for MUX A and MUX B?
(c) How many inputs and outputs are there in a decoder?
(d) How many inputs and outputs are there in ALU for data, including input and output carries?
(e) Formulate a control word for the system.

Solution:

(a) 32 Multiplexers, each of size 16×1.
(b) 4 Inputs each, to select one of 16 registers.
(c) 4 to 16 – Line decoder
(d) $32 + 32 + 1 = 65$ data input lines
(e) $32 + 1 = 33$ data output lines

4 bits	4 bits	4 bits	6 bits
SELA	SELB	SELD	OPR

2.5.1 Instruction Execution

A CPU generally executes one instruction at a time sequentially and a sequence of such instructions is known as a program. The CPU executes the instructions that reside in the main memory. In order to execute an instruction, the CPU has to fetch the instruction first from the main memory into one of its registers. It then **decodes** the instruction, that is, it decides what the instruction intended to do, fetch operands required and finally **executes** the instruction. This process is repeated continuously for a complete program and is known as the **fetch–execute** cycle (Fig. 2.4). The following steps are performed for executing an instruction:

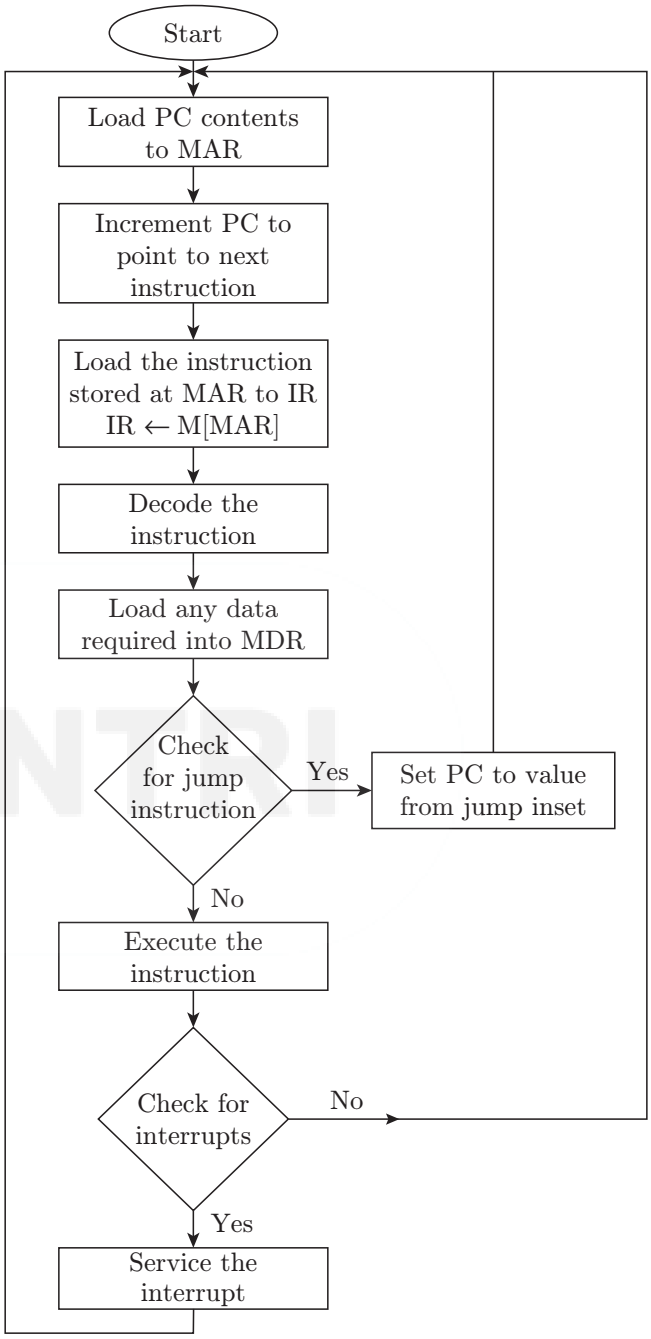


Figure 2.4 | Instruction cycle.

1. **Fetching the instruction:** The next instruction is fetched from the memory address that is saved in the program counter, and memory content fetched is stored in instruction register (IR). The program counter then points to the next instruction that will be read in the next cycle.
2. **Decode the instruction:** During this cycle, the instruction inside the IR gets interpreted by the decoder.

3. **Operand fetch:** In case of a direct or indirect memory instruction, the execution begins in the next clock cycle. If the instruction has an indirect address, the effective address of the operand is read from the main memory, and the required data is fetched from the memory into memory data registers. If the instruction has direct address, nothing is done at this clock cycle.
4. **Execute the instruction:** The control unit of the CPU passes the instruction decoded by decoder as a sequence of control signals to the different functional units of the CPU to execute the tasks required by the instruction such as reading values from registers or input devices, performing mathematical or logic micro-operations by ALU, and writing the result back to a register or main memory.

2.5.2 CPU Data Path

CPU contains data paths that are responsible for routing data between the functional units of a computer. The following are the different data path structures available for routing:

1. **Single bus structure:** In this architecture, all CPU registers are connected to the same bus. Data can be transferred either between CPU registers or between CPU register and ALU at a given clock pulse. The speed of operation is slow as only one operand can be transferred in one clock cycle and addition operation ($R_1 \leftarrow R_2 + R_3$) occurs in three clock cycles.
2. **Two bus structure:** All general purpose CPU registers are connected to both buses say bus A and bus B; but special purpose registers are divided into two groups, say group 1 connecting bus A to program counter and one input of ALU and group 2 connecting bus B to MDR (Memory Data Register) and other input of ALU. The two operands are transferred to ALU in 1 clock cycle and the addition operation ($R_1 \leftarrow R_2 + R_3$) occurs in 2 clock cycles.
3. **Three bus structure:** The performance can be further be improved by using three buses such that addition operation ($R_1 \leftarrow R_2 + R_3$) can occur in one clock cycle.

2.5.3 Control Unit Design

Control unit is considered as brain of a CPU that controls various units in the data path. The performance of control unit is important as it determines the clock cycle of the processor. Control unit can be designed either by hardwired or by microprogram.

1. **Hardwired control:** Control unit is made up of sequential and combinatorial circuits to generate the control signals and interpret instructions (Fig. 2.5). The instruction decoder decodes the instruction loaded in instruction register. The step decoder generates a separate control line for each step in the control sequence. The encoder gets its input signal from the decoder, step decoder, external input and condition codes and generates individual control signals. It is faster and more efficient but less flexible and is difficult to add new feature or correct mistakes in original design.

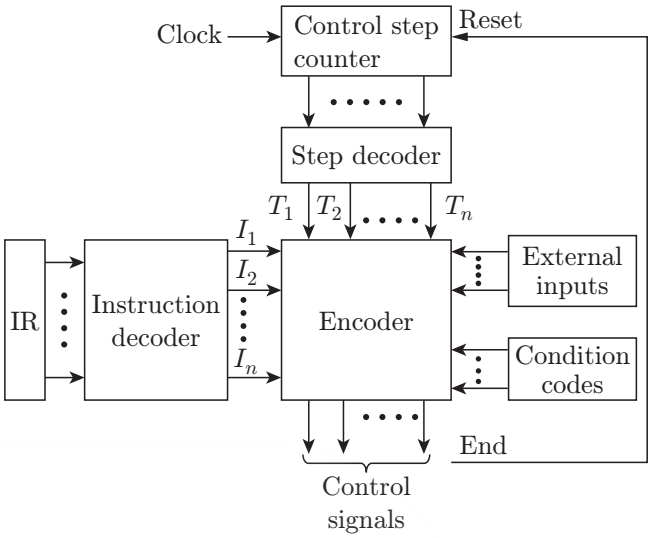


Figure 2.5 | Block diagram of hardwired control unit.

2. **Micro-programmed control:** Control signals are generated by using programming known as micro-programs that constitutes micro-instructions (control word) (Fig. 2.6). Memory that is part

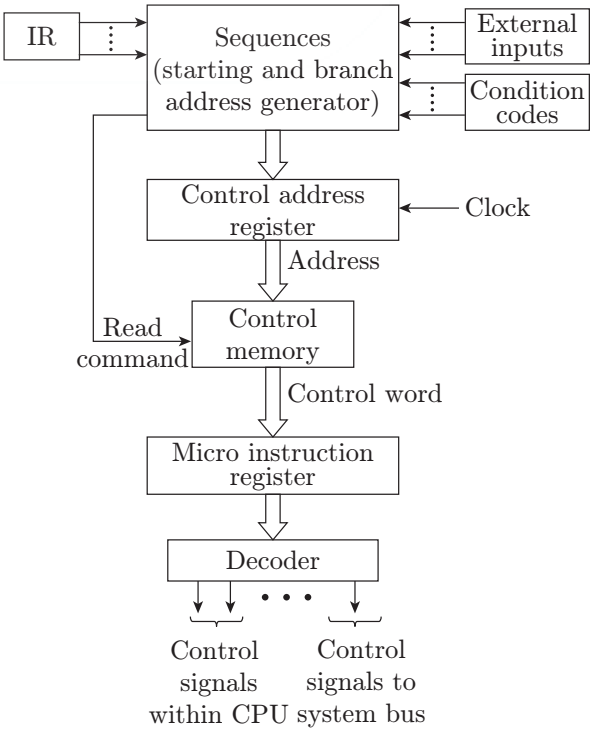


Figure 2.6 | Block diagram of micro-programmed control unit.

of CPU is known as control memory and stores micro-instructions. The micro-program sequencer generates the address of micro-instruction according to instruction stored in instruction register. The address of micro-instruction to be executed is available in content addressable register. Micro-program sequencer issues read command to read micro-instruction from control memory into micro-instruction register which on execution generates control signals for various parts of a processor. This control unit design is more flexible to accommodate new features and less error prone but quite slower than the hardwired unit.

The format of the control word is

Branch condition	Flag	Control signal	Control memory address

On the basis of the type of control word supported, it is divided into two types:

1. **Horizontal micro-programmed control unit:** In this design, the control signals are represented in the form of 1 bit per control signal and it supports longer control word.
2. **Vertical micro-programmed control unit:** In this design, the control signal is represented by using encoding format.

Problem 2.6: Consider a control unit which has 1024 control word memory; it supports 48 control signals and 8 flag conditions. What is the size of the control word in bits and control memory in bytes?

Solution:

(a) Using horizontal programmed control unit

0 bits	3 bits	48 bits	10 bits
Branch condition	Flag	Control signal	Control memory

Size of control word = 61 bits
Control memory = $(1024 \times 61)/8 = 128 \times 61$ bytes

(b) Using vertical programmed control unit

0 bits	3 bits	48 bits	10 bits
Branch condition	Flag	Control signal	Control memory
		log 48 ~ 6 bits	

Size of control word = 19 bits
Control memory = $(1024 \times 19)/8 = 128 \times 19$ bytes

2.5.4 RISC versus CISC Processors

The differences between reduced and complex instruction set computers is given in Table 2.4

Table 2.4 RISC versus CISC

RISC (Reduced Instruction Set Computers)	CISC (Complex Instruction Set Computers)
Rich register set	Less number of registers
Supports less addressing modes	Supports more number of addressing modes
Supports fixed length instruction	Supports variable length instruction
Successful pipeline with one instruction per cycle	Unsuccessful pipeline
Example: ARM, Motorola	Example: Pentium processors

2.6 I/O INTERFACE (INTERRUPT AND DMA MODE)

I/O interface bridges the differences between CPU and peripheral devices and provides a method for transferring information between internal storage and external I/O devices. There are the following three modes of I/O transfer:

1. **Programmed I/O:** The I/O device does not have direct access to memory. It requires execution of several instructions by the CPU and the CPU has to wait for the I/O device to be ready for either reception or transmission of data.
2. **Interrupt initiated I/O:** In this, instead of waiting, the control is transferred from a currently running program to another service program as a result of an external/internal generated request.
   - **Hardware interrupts:** These interrupts are present in the hardware pins.
   - **Software interrupts:** These are the instructions used in the program whenever the required functionality is needed.
   - **Maskable interrupts:** These interrupts may be enabled or disabled explicitly.
   - **Non-maskable interrupts:** These interrupts are always there in the enable state. We cannot disable them by explicit conditions (flags).
   - **Vectored interrupts:** These interrupts are associated with the static vector address.
   - **Non-vectored interrupts:** These interrupts are associated with dynamic vector address.
   - **External interrupts:** These interrupts are generated by external devices such as I/O.
   - **Internal interrupts:** These devices are generated by the internal components of the processor such as temperature sensor, power failure, error instruction, etc.

- **Synchronous interrupts:** These interrupts are controlled by the fixed time interval. All the interval interrupts are called as synchronous interrupt.
- **Asynchronous interrupts:** These interrupts are initiated based on the feedback of previous instructions. All the external interrupts are called as asynchronous interrupt.

3. **Direct memory access (DMA):** It is one of several methods for coordinating the data transfers between an I/O device and the core processing unit or memory in a computer. It refers to transfer of data directly between a fast storage device and memory bypassing CPU because of its limited speed. DMA provides a significant improvement in terms of latency and throughput as it allows the I/O device to access the memory directly, without using the processor. There are certain advantages of using DMA for data transfer:
   - DMA saves processor's MIPS as the core can operate in parallel.
   - DMA saves power because it requires less circuitry than the processor to transfer data.
   - DMA has no modulo block size restrictions.

   Direct memory access (DMA) controller takes over the control of buses to manage the transfer directly between the I/O device and memory. Bus request (BR) and Bus grant (BG) signals are used by the DMA controller to request the CPU to relinquish control of the buses and get the control of system buses (Fig. 2.7). The DMA controller consists of 3 different registers: an address register, a control register and a word counter register. To transfer a block of data between an I/O device and memory, the controller stores initial values in the address register. The DMA channel then transfers the block of information from or to memory according to the control register. The starting address of the block in memory is given by the address register, and the length of the bytes to transfer is given by the word count register. The controller decrements a word counter each time it moves a data byte.

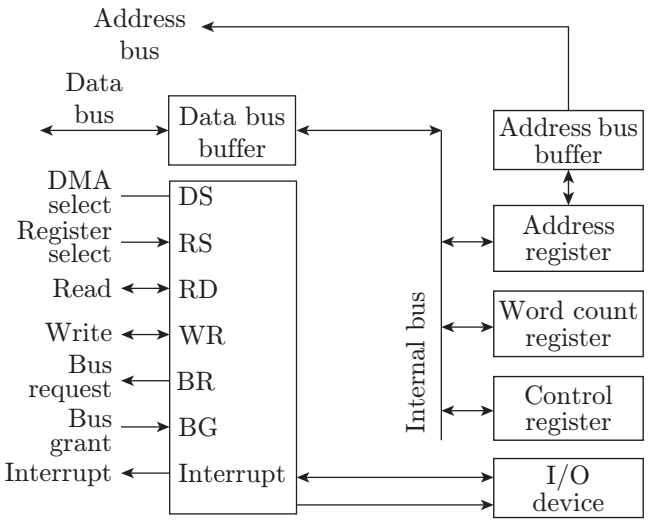


Figure 2.7 Block diagram of the DMA controller.

There are several modes of operation of DMA:

- **Burst or block transfer mode:** In this mode, the entire block of data is transferred once the DMA controller is granted access to the system bus by the CPU. The bytes of data in the block are transferred before releasing control of the system buses back to the CPU. The only disadvantage of this mode is that it renders the CPU inactive for some long periods of time.
- **Cycle stealing mode:** In this mode, the DMA controller obtains access to the system buses like burst mode; but after one byte of data transfer, the control of the system bus is released back to the CPU via BG. It is then continually requested again via BR, transferring one byte of data per request, until the entire block of data has been transferred. This mode is suitable for the systems in which the CPU cannot be disabled for the considerable length of time as in burst transfer modes such as for controllers monitoring the data in real time. The advantage is that CPU is not idled for as long as in burst mode, but the data block is not transferred as quickly.
- **Transparent mode:** It is the slowest yet more efficient data transfer mode in terms of overall system performance. The DMA controller transfers data only when the CPU is busy in performing operations that do not use the system buses. So, the CPU never stops executing its programs but the biggest disadvantage is complex hardware circuitry that needs to determine when the CPU is not using the system buses.

  A DMA read transfers data from the memory to the I/O device, while DMA write transfers data from an I/O device to memory. The functional behaviour of a DMA transfer outlined in Fig. 2.8:

- The CPU transmits the following information to a DMA controller:
  (a) beginning address in memory which is stored in address register in DMA controller.
  (b) Number of words to transfer which is stored in word count register in DMA Controller.
  (c) direction (memory-to-I/O device or I/O device-to-memory), port ID, DMA mode of transfer and end of block transfer either through interrupt request or no interrupt request which is stored in control register as command word.
- The processor them relinquishes control of address, data and control buses to DMA Controller and returns to other processing activities while the DMA controller starts the data transfer between I/O device and memory.

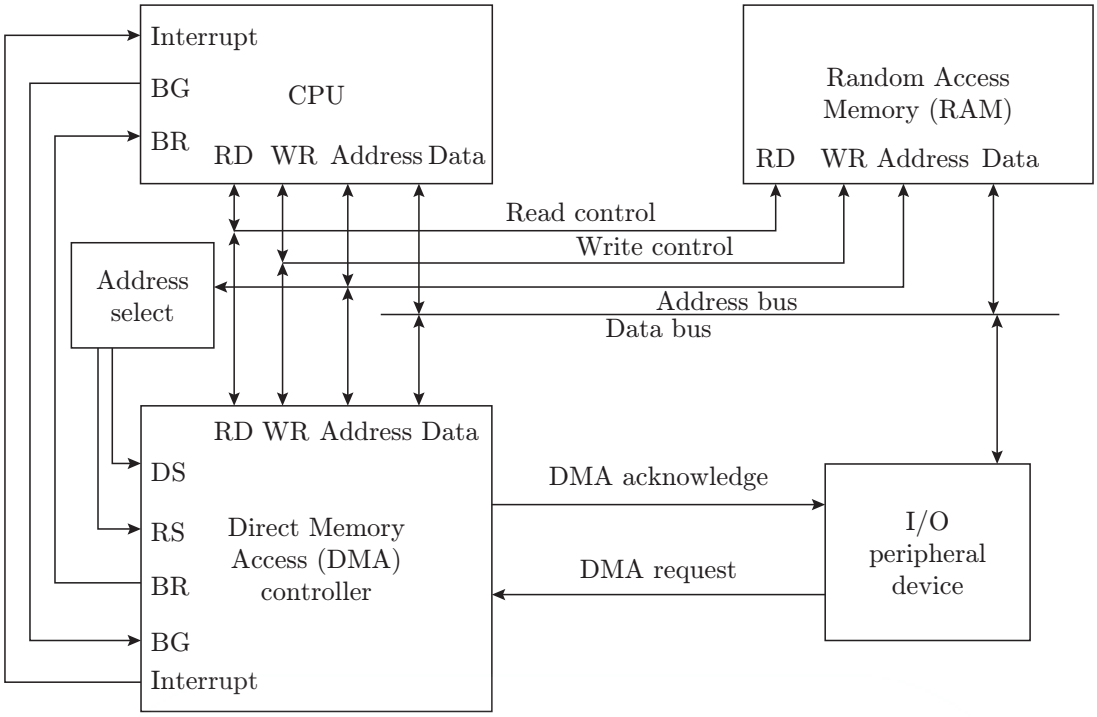


Figure 2.8 DMA controller interconnection with memory, CPU and I/O devices.

- When the DMA controller accesses memory, it synchronizes this memory request with an idle period of the processor, thus disabling the processor, or requesting a halt of the processor, and awaits an acknowledgement.
- After the completion of the block transfer, the DMA controller either raises an interrupt request if the interrupts are enabled or indicates the completion in its status register and the processor recognizes I/O completion (either by interrupt signal or by reading the status register) and gets its system buses back and normal processing starts. The device has to initiate a new data transfer through DMA request signal which is again acknowledged by CPU through DMA acknowledge signal via DMA controller.

2.7 INSTRUCTION PIPELINING

In early computers, each instruction completely finished before the execution of the next one began. The hardware circuits needed to perform different operations of an instruction cycle are different and most part of these processor circuits are idle at a given moment of time. These processor circuits wait for the other parts of the processor to complete its part of execution first. Instruction pipelining is a technique for overlapping the execution of several instructions to reduce the overall execution time of a set of instructions and there is no need to wait of the most part of the processor circuits for the other parts of the processor to complete their part of execution. Pipeline speed is limited by the slowest pipeline stage.

Throughput of a processor is the rate at which operations get executed. Latency is the amount of time that a single operation takes to execute. In an unpipelined computer, throughput = 1/latency, as each operation executes by itself and for pipelined computer, throughput > 1/latency, since execution of instruction is overlapped.

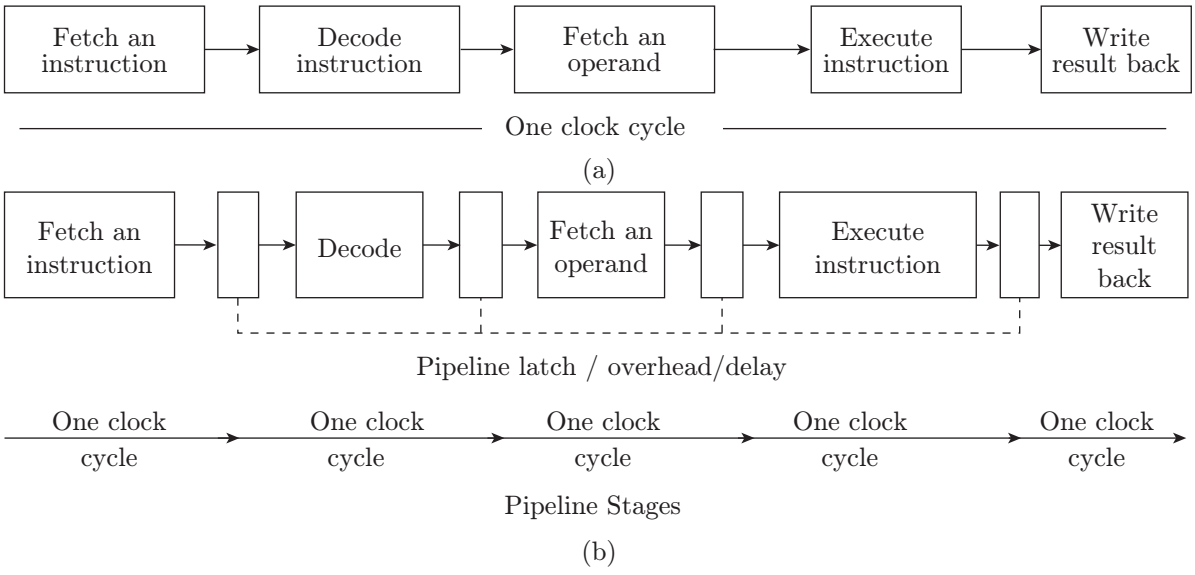
Consider a k-segment pipeline with a clock cycle time T_p used to execute n tasks (Fig. 2.9). An equivalent non-pipelined system takes T_n time to complete each task. The speed up of a pipelined system over a non-pipelined system is given by the following relation:

$$S = \frac{n \times T_n}{(k + n - 1) \times T_p}$$

Theoretically, maximum speed up that a pipelined system can achieve is given by the following equation:

$$S = \frac{kT_p}{T_n} = k$$

Pipelining Hazards: These hazards reduce the ideal speed up gained by pipelining by preventing the next instruction in the sequence from being executing during its designated clock pulse. Hazards forces the pipeline to be stalled. There are three types of hazards:



step →	1	2	3	4	5	6	7	8
Segment ↓ Fetch	I_1	I_2	I_3	I_4				
Decode		I_1	I_2	I_3	I_4			
Fetch operand			I_1	I_2	I_3	I_4		
Execute				I_1	I_2	I_3	I_4	
Write back					I_1	I_2	I_3	I_4

(c)

Figure 2.9 (a) Unpipelined processor. (b) Pipelined five-stage processor. (c) Timing diagram of a five-stage instruction pipeline.

1. **Structural hazards:** These result from resource conflicts when the hardware cannot support instructions that need simultaneous execution in pipeling.
2. **Data hazards:** They arise when an instruction depends on the result of a previous instruction and that result is not yet calculated.

   There are three situations in which data hazards can occur:

   - Read after write (RAW), a true dependency
   - Write after read (WAR), an anti dependency
   - Write after write (WAW), an output dependency

   Consider two instructions `inst1` and `inst2` occurring, with `inst1` occurring before `inst2` in the program order.

   - **Read after write (RAW):** A read after write (RAW) data hazard is a situation in which an instruction **refers** to a result which is yet not been calculated, that is, in this `inst2` tries to read a source before `inst1` writes to it. This situation arises if the read operation by instruction takes place before write done by other instruction. For example,

     ```
     inst1: R3 <-R1 + R2
     inst2: R4 <-R3 + R2
     ```

     The first instruction calculates a value by adding values in registers R1 and R2 and saves the result in register R3, and the second instruction uses this saved value to calculate a result for register R4. However, in a pipeline, when operands for the second operation are fetched, the results from the first instruction will not have been saved yet, and so there arises a data dependency. It can be said that there is a data dependency with instruction `inst2`, as it is dependent on the completion of instruction `inst1`.

- **Write after read (WAR):** A write after read (WAR) data hazard refers to a situation in which there is a problem with concurrent execution, that is, `inst2` tries to write a destination before it is read by `inst1`. This situation arises if write operation completes first by instruction before the read operation takes place by other instruction. For example,

```
inst1: R4 <-R1 + R3
inst2: R3 <-R1 + R2
```

  If a situation arises in which there is a chance that `inst2` may get completed before `inst1` (i.e., with concurrent execution) we must note that we do not store the result of register `R3` before inst1 has had a chance to fetch the operands.
- **Write after write (WAW):** A write after write (WAW) data hazard refers to a situation in which there is a concurrent execution environment, that is, `inst2` tries to write an operand before it is written by `inst1`.This situation arises if write operation by an instruction occurs in the reverse order of the intended sequence. For example,

```
inst1: R2 <-R1 + R3
inst2: R2 <-R4 + R5
```

  The WB (write back) of `inst2` must be delayed until the execution of `inst1`.

3. **Control hazards:** They arise from the pipelining of branches and other instructions that change the value of PC.

Speed up from pipelining

$$= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

Speed up from pipelining

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle pipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$\text{Ideal CPI} = \frac{\text{CPI unpipelined}}{\text{Pipeline depth}}$$

Speed up from pipelining

$$= \frac{\text{Ideal CPI} \times \text{Pipeline depth} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

Speed up from pipelining

$$= \frac{\text{Ideal CPI} \times \text{Pipeline depth} \times \text{Clock cycle unpipelined}}{(\text{Ideal CPI} + \text{Pipeline stall}) \times \text{Clock cycle pipelined}}$$

Assuming ideal CPI as 1, speed up is:

Speed up from pipelining

$$= \frac{\text{Pipeline depth} \times \text{Clock cycle unpipelined}}{(1 + \text{Pipeline stall}) \times \text{Clock cycle pipelined}}$$

where CPI is cycles per instruction.

Problem 2.7: Consider a four-stage pipeline processor. The number of cycles needed by the four instructions I_1, I_2, I_3 and I_4 in stages instruction fetch, decode, operand fetch and execute are shown below. Assume I_2 is the branch instruction. Draw the timing space diagram.

	S_1	S_2	S_3	S_4
I_1	2	1	1	1
I_2	1	2	3	1
I_3	1	1	1	2
I_4	2	1	3	1

Solution:

STEP →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Fetch	I_1	I_1	I_2	I_3	–	–	–	–	–	I_3	I_4	I_4					
Decode			I_1	I_2	I_2	–	–	–	–	–	I_3	–	I_4				
Operand Fetch				I_1		I_2	I_2	I_2	–	–	–	I_3	–	I_4	I_4	I_4	
Execute					I_1	–	–	–	I_2	–	–	–	I_3	I_3	–	–	I_4

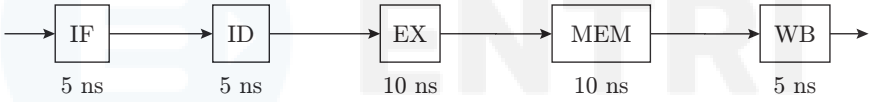
Problem 2.8: Assume a simple 5-stage pipeline (IF, ID, E, DF, W) each stage takes a single cycle. Assuming there are no cache misses. How many cycles would the following code take to execute if there is no special hardware to improve performance in the presence of hazards?

```
MOV   edx,[ecx+100]
MOV   ebx,[ecx+104]
ADD   edx,ebx
MOV   [ecx+108],ebx
MOV   eax,[ecx+100]
ADD   ebx,eax
```

Solution: The above code takes 14 cycles to execute, as shown below:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	DF	E	W									
	IF	ID	DF	E	W								
		IF	ID	DF	stall	E	W						
			IF	ID	stall	DF	stall	W					
				IF	ID	stall	DF	stall	stall	E	W		
					IF	ID	stall	DF	stall	stall	stall	E	W

Problem 2.9: In the below figure, calculate the total execution time after which the result of the fourth task entering the pipe above ready?



Solution:

	5	10	15	20	25	30	35	40	45	50	55	60	65
Inst1	IF	ID	EX	EX	MEM	MEM	WB						
Inst2		IF	ID		EX	EX	MEM	MEM	WB				
Inst3			IF	ID			EX	EX	MEM	MEM	WB		
Inst4				IF	ID				EX	EX	MEM	MEM	WB

Therefore, the total execution time is 65 ns.

Problem 2.10: What is the mean overhead of a pipeline with 8 stages and an execution time per stage of 2 ns?

Solution: The mean overhead = (Stages − 1) × Execution time per stage = $(8-1)\times 2 = 7\times 2 = 14$ ns

Problem 2.11: How many stages has a pipeline that achieves a speed of 9.9 for 100 operations?

Solution:

$$\text{Speed} = \frac{n\times k}{k-1+n} \Rightarrow 9.9 = \frac{n\times 90}{(90-1)+n} \Rightarrow n = 11$$

Problem 2.12: Calculate the time required to perform 1000 operations in a 6-staged pipeline with an execution time of 3 ns per stage?

Solution:

$$T_p = (k-1+n)\times T = (6-1+1000)\times 3 = 3.015\ \mu s$$

Problem 2.13: Calculate the mean overhead of a pipeline with 7 stages and an execution time per stage of 2 ns?

Solution: Mean overhead of pipeline =

$$\frac{(k\times T_p - T_n)}{k} = (k-1)\times T = (7-1)\times 2 = 12\ \text{ns}$$

Problem 2.14: Consider a pipeline with 5 stages: IF, ID, EX, M and W. Assume that each stage requires one clock cycle. Show how the following program segment for adding 2 arrays is processed and compare the clock cycles needed in non-pipelined system with pipelined system when result of the branch instruction i.e. content of is available after WB stage.

```
LOAD R4 #400
L1: LOAD R1, 0 (R4);
LOAD R2, 400 (R4);
ADD R3, R1, R2;
STORE R3, 0 (R4);
SUB R4, R4, #4;
BNEZ R4, L1;
```

Solution: Number of cycles = [Initial instruction + (Number of instructions in the loop L1) × Number of loop cycles] × Number of clock cycles/instruction (CPI)

$$= [1 + (6) \times 400/4] \times 5 = 3005$$

Timing diagram for one loop iteration in a pipelined system is as follows:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LOAD R4 #400	IF	ID	EX	M	W										
LOAD R1, 0 (R4)		IF	ID	EX	M	W									
LOAD R2, 400 (R4)			IF	ID	stall	stall	EX	M	W						
ADD R3, R1, R2				IF	ID	stall	stall	EX	stall	M	W				
STORE R3, 0 (R4)					IF	ID	stall	DF	stall	stall	E	W			
SUB R4,R4, #4							IF	ID	stall	Ex	M	W			
BNEZ R4, L1								IF	stall	ID	stall	stall	EX	M	W

Number of cycles in the loop = 15

Number of clock cycles for segment execution on pipelined processor

$$= 1 + (\text{Number of clock cycles in the loop L1}) \times \text{Number of loop cycles}$$

$$= 1 + 15 \times 400/4 = 1501$$

$$\text{Speedup} = \frac{\text{Number of Clock cycles for the program execution on non-pipelined processor}}{\text{Number of Clock cycles for the segment execution on pipelined processor}}$$

$$= \frac{3005}{1501} = 2 \text{ times}$$

Problem 2.15: Consider a 5-stage pipeline with stages: For all following questions we assume that: (a) Pipeline contains stages: IF (Instruction Fetch), IS (Issue), FO (Fetch operand), E (Execute) and W (Write). (b) Each stage except E requires one clock cycle and system has 4 Functional Units for floating point operations, FP load/store, FP addition/subtraction, FP multiplication and FP division, (c) Execution stage for Load/Store operations requires 1 clock cycle, for ADD or SUB operations requires 1 clock cycle, for MUL operation requires 3 clock cycles and for DIV operation requires 4 clock cycles. All memory references hit in cache. Pipeline has forwarding circuitry for all FUs, except FP-Load/Store where operand is ready after W-stage.

Timing diagram of is presented below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LOAD F6, 20(R5)	IF	IS	FO	E	W											
LOAD F2, 28(R5)		IF	ISD	FO	E	W										
MUL F0, F2, F4			IF	IS	stall	stall	FO	E	E	E	W					
SUB F8, F6, F3				IF	IS	FO	E	W								
DIV F10, F0, F6					IF	IS	stall	stall	stall	stall	FO	E	E	E	E	W
ADD F6, F8, F2						IF	IS	FO	E	W						
STORE F8, 50(R5)							IF	IS	FO	E	W					

Identify the hazards in the following instructions from the following list (Structural, Data, Control, RAW, WAR, WAW, None)

1. MULT F0, F2, F4 and STORE F8, 50(R5)
2. DIV F10, F0, F6 and ADD F6, F8, F2
3. MULT F0, F2, F4 and DIV F10, F0, F6
4. DIV F10, F0, F6 and ADD F6, F8, F2

Solution: 1. Structural; 2. Data; 3. RAW; 4. WAR.

2.8 MEMORY HIERARCHY

The storage media can be categorized in hierarchy according to their speed and cost (Fig. 2.10). As we move down the hierarchy, access time increases and cost per bit decreases.

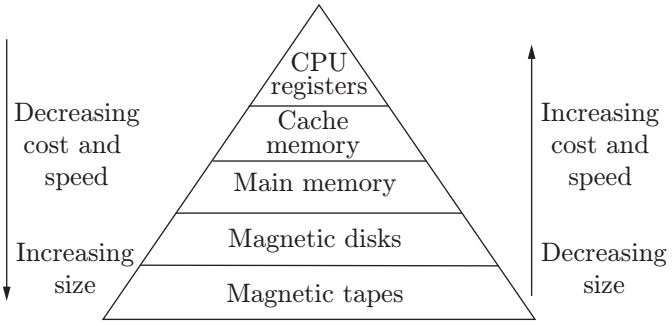


Figure 2.10 | Memory hierarchy.

2.8.1 Main Memory

It is the central storage unit that directly communicates with the CPU. It is designed using semiconductor-integrated circuits and needs constant power supply to maintain the information. It is expensive as compared to auxiliary storage so it has limited capacity. Example: R/W (read/write) memory or RAM (random access memory) and ROM (read only memory). Integrated RAM chips are available in two modes:

1. **Static RAM:** It stores the binary information in flip flops and information remains valid until power is supplied. It has faster access time and is used in implementing cache memory.
2. **Dynamic RAM:** It stores the binary information as a charge on the capacitor. It requires refreshing circuitry to maintain the charge on the capacitors after few milliseconds. It contains more memory cells per unit area as compared to SRAM.

2.8.1.1 Memory Interfacing

If the required memory for the computer is larger than the capacity of one chip, it is necessary to connect multiple RAM and ROM chips to a CPU through the data and address buses (Fig. 2.11). The low-order address bus lines select the word within a chip and other lines select a particular chip through its chip select inputs. Assume a computer system needs 256 bytes of RAM and 512 bytes of ROM. The configuration of RAM chip is 128×8 and ROM chip is 512×8. The RAM and ROM chips required are as follows:

$$\text{Number of RAM chips} = 256/128 = 2$$
$$\text{Number of ROM chips} = 512/512 = 1$$

The memory interconnection is depicted in the following diagram:

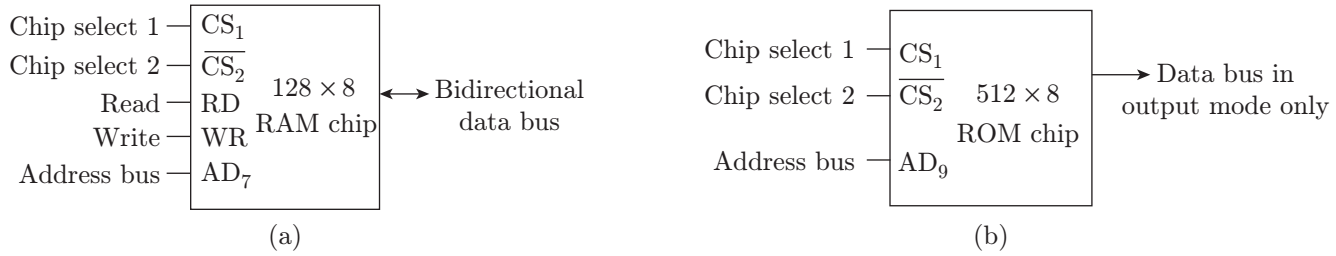


Figure 2.11 | (a) RAM chip. (b) ROM chip.

Problem 2.16: A computer employs RAM chips of 256 × 8 and ROM chips of 1024 × 16. The computer system needs 2K bytes of RAM and 4K bytes of ROM and four interface units each with four registers. Draw a memory address map for the system and give the address range in hexadecimal for RAM and ROM chips.

Solution: RAM 2048/256 = 8 chips; 2048 = 211; 256 = 28
ROM 4096/1024 = 4 chips; 4096 = 212; 1024 = 210
Interface 4 × 4 = 16 registers; 16 = 24

Component	Address	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
RAM	0000-O7FF	0	0	0	0	0	↔	x	x	x	x	x	x	x	x		
								3 × 8 decoder									
ROM	4000-4FFF	0	1	0	0	↔	x	x	x	x	x	x	x	x	x	x	
								2 × 4 decoder									
Interface	8000-800F	1	0	0	0	0	0	0	0	0	0	0	0	x	x	x	x

2.8.2 Secondary Memory

Secondary memory, also known as auxiliary memory or external memory, can store a large amount of data at lesser cost per byte than the main memory. They are non-volatile in nature, that is, data is not lost when the device is powered off. The most common auxiliary storage devices used in consumer systems are flash memory, optical disks and magnetic disks.

1. **Flash memory:** Flash memory is an electronic non-volatile fastest computer storage device that can be electrically erased and reprogrammed. Example: flash drives and solid state drive.
2. **Optical disk:** Optical disks are low-cost mass storage devices from which read and write operations are performed using laser technology. Optical disks can store huge amounts of data up to 6 GB (6 billion bytes). Different types of optical disks are CD-ROM (compact disk read-only), WORM (write-once read-many), EO (erasable optical disks) and DVD.
3. **Magnetic disk:** A magnetic disk is composed of a circular platter made of metal or plastic and coated with magnetized material on both sides. Multiple disks are stacked over one another on the spindle with read/write heads on each surface. Bits are stored as spots on magnetized surface along concentric circles called tracks. Tracks are further divided into wedge-shaped sectors.
4. **Magnetic tapes:** It consists of tape made up of plastic covered with magnetic oxide coating. Tapes are mounted on reels. Bits are recorded as magnetic spots on tape along several tracks. R/W heads are mounted in each track so that data can be recorded and read as a sequence of characters. Seven or nine bits are recorded to form a character together with a parity bit. Data is recorded in contiguous blocks separated by inter-record gaps.

2.8.3 Cache Memory

It is a special memory that compensates the speed mismatch between processor and main memory access time. It temporarily stores frequently used instructions and data for faster processing by the CPU. Cache hit ratio is calculated to measure its performance. If a data

item requested by the CPU is found in cache it is called hit otherwise it is a miss. Hit ratio is defined as ratio of number of hits divided by total CPU references to memory.

$$\text{Hit ratio } (h) = \frac{\text{Number of hits}}{\text{Number of hits} + \text{Number of misses}}$$

$$\text{Average access time} = \text{Hit ratio} \times T_c + (1 - \text{Hit ratio})(T_c + T_m)$$

where T_c is cache access time and T_m is the main memory access time.

2.8.3.1 Elements of Cache Design

The various elements of cache design are as follows:

1. **Cache size:** It should be optimum, small enough to keep average cost per bit close to the main memory and large enough to keep overall average access time close to the cache memory.
2. **Mapping function:** It describes the mapping of main memory block to cache block. There are three different mapping techniques: fully associative, direct mapped and set associative cache organization.
3. **Replacement algorithm:** When a new memory block is required in cache, one of the existing blocks must be replaced by a new block. Example: FIFO (first in, first out), LRU (least recently used).
4. **Write policy:** Cache memory follows write-through and write-back updating policies. In write-through policy, cache controller copies data immediately to main memory as data is written in cache. The data in main memory is always valid, but this approach reduces system performance. In write back, update to memory block is delayed until the updated cache block is replaced by a new block.

2.8.4 Cache Mapping Techniques

The cache memory can store a reasonable number of blocks, but this number is always small as compared to blocks in the main memory to keep average cost per bit low. The correspondence between memory blocks and cache block is specified by the following mapping techniques. Consider a cache memory consisting of 2K words with 128 blocks of 16 words each. Number of bits required to address a cache block is 11 bits. Main memory has 64K words and bits required to address is 16 (Fig. 2.12).

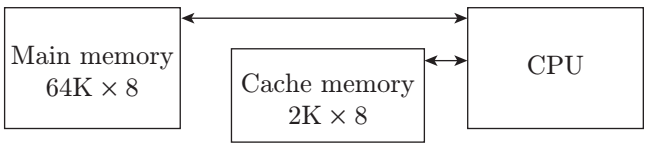


Figure 2.12 | Cache mapping example.

2.8.4.1 Direct Mapping

In this technique, each block from the main memory has only one possible location in the cache memory. In this example, say a block from main memory maps onto a block (i mod 128) of the cache. If there are 2^n words in the cache memory and 2^m words in the main memory, then m-bit main memory address is divided into two fields: n bits for index field to access the cache and $(m - n)$ bits for the tag field. Each word in cache consists of the data and the associated tag. Whenever a new block is brought into cache, tag is stored along with data bits. Index field is further divided into block and word if there are multiple words (say k) in a block. The lower k bits select one of the k words in a block known as word field. The block field is used to distinguish a block from other blocks.

Tag $(m - n)$ bits	Index (n bits)

Tag $(m - n)$ bits	Block $(n - k)$ bits	Word (k bits)

When CPU generates a memory request, the block field points to a particular block location in the cache. The high-order tag field is compared with tag bits associated with that cache location. If they match, then the desired word is in that block of cache. If there is no match, then the block containing the required word must be loaded to cache first (Fig. 2.13).

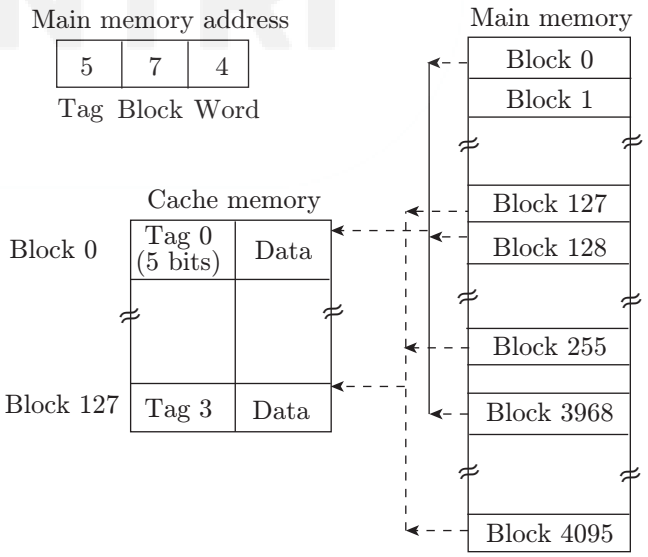


Figure 2.13 | Direct mapped cache organization.

The demerit of direct mapping is that hit ratio drops considerably if two or more words having same index and different tags are accessed consecutively one after the other.

2.8.4.2 Fully Associative Mapping

In this technique, a main memory block can be placed into any cache block location. It is the most flexible cache organization. The main memory address is divided into

two fields: word and tag. The associative memory stores both the address (tag) and data of the main memory. Figure 2.14 shows the mapping of different blocks into cache. High-order 12 bits of CPU address is placed in the argument register of the associative memory and compared to tag bits of each block of the cache to see if the desired block is present. Once the desired block is present, 4-bit word is used to extract necessary word from the cache.

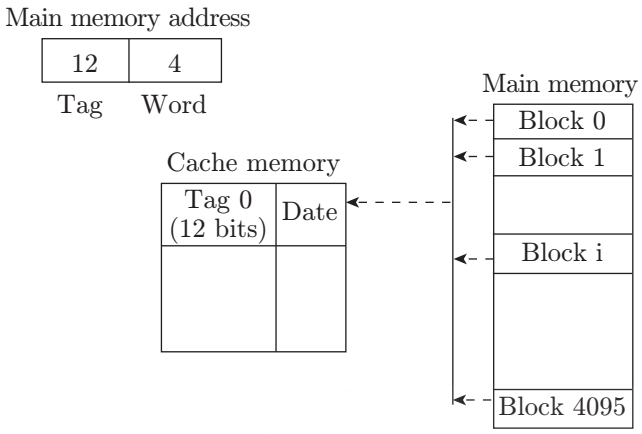


Figure 2.14 | Associative mapped cache organization.

It is necessary to compare high-order bits of main memory with all tag bits corresponding to each block to find whether a given block is present in cache, so it is the most expensive.

2.8.4.3 Set-Associative Mapping

As fully associative mapping is an expensive solution and direct mapping does not allow words with same index but different tag to exist in cache, set associative mapping is a combination of both. It is an improvement over direct mapping where contention problem is solved by having several choices for block placement. The figure below shows two-way set associative cache because each block of main memory has two choices for block placement in cache. A block i in the main memory can be in any block belonging to set i mod S of cache, where S is the number of sets. The block 0, 64, 128, … and so on of main memory can map into any of the two blocks in set 0.

The main memory address is divided into three fields: low-order bits for word field, set field to determine the desired block from all possible sets and high-order bits for the tag field. Each word in cache consists of data and the associated tag.

Tag	Set	Word

When the CPU generates a memory request, the set field points to a particular set of the cache which might contain the desired block. The high-order tag field is then compared associatively to the tags corresponding to the matched set. If a match occurs, the corresponding word is read from cache else main memory is referred and block containing that word is brought into cache for future reference (Fig. 2.15).

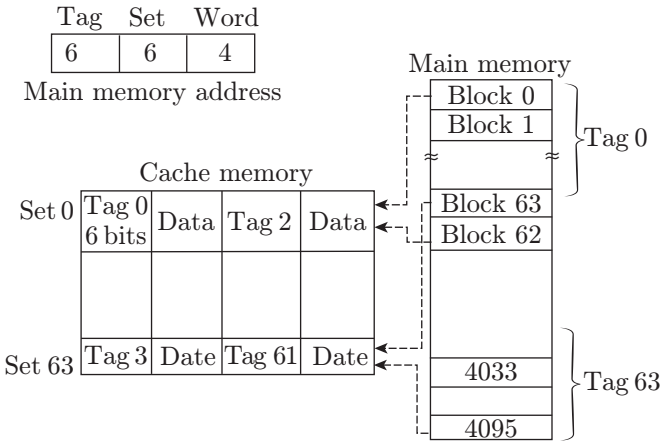


Figure 2.15 | Set-associative mapped cache organization.

Problem 2.17: Consider a memory hierarchy system containing a cache, a main memory and a virtual memory. Assuming, cache access time of 5 ns, and 80% hit ratio . The access time of the main memory is 100 ns, and it has a 99.5% hit rate. The access time of the virtual memory is 10 ms. Calculate the average access time of the memory hierarchy.

Solution: As we know, the hit rate of virtual memory is 100%, the average access time for requests that reach the main memory as (l00 ns × 0.995) + (10 ns × 0.005) = 50,099.5 ns. Given this, the average access time for requests that reach the cache is (5 ns × 0.80) + (50,099.5 ns × 0.20) = 10,024 ns.

Problem 2.18: A computer uses RAM chips of 1024 × 1 capacity.

(a) How many chips are needed to provide a memory capacity of 16K bytes?
(b) How many of these lines will be common to all chips?

Solution:

(a) Chips are needed to provide a memory capacity of 16K bytes = 16 × 8 = 128 chips
(b) Using 14 address lines (16K = 2^{14}), we have 10 lines specifying the chip address which is common to all chips.