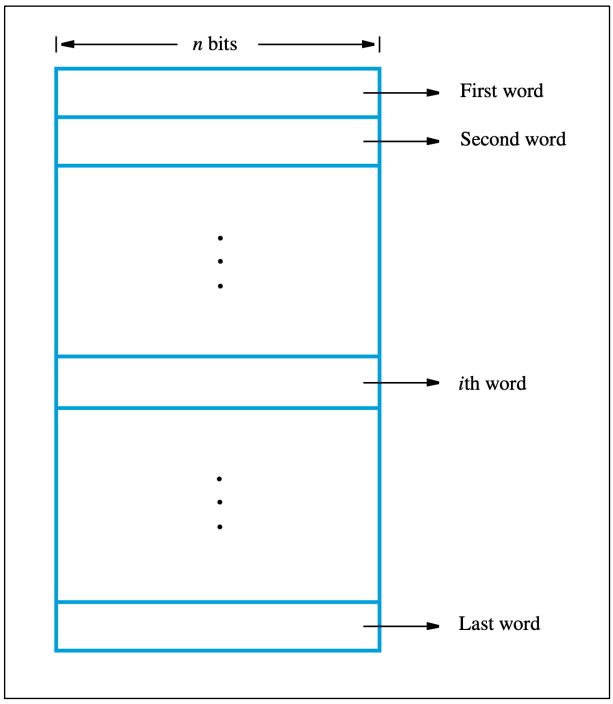
UNIT - 2 **[Contact Hours - 10]**

Contents

- Memory Location and Addresses
- Memory Operations
- Instruction Notations
- Instruction Sequencing
- Instruction Branching
- Addressing Modes
- Execution of a Complete Instruction and Single Bus Organization
- Control Unit Operations: Instruction sequencing, Micro operations and Register Transfer. Hardwired Control,
- Micro-programmed Control: Basic concepts, Microinstructions and micro- program sequencing
- Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement

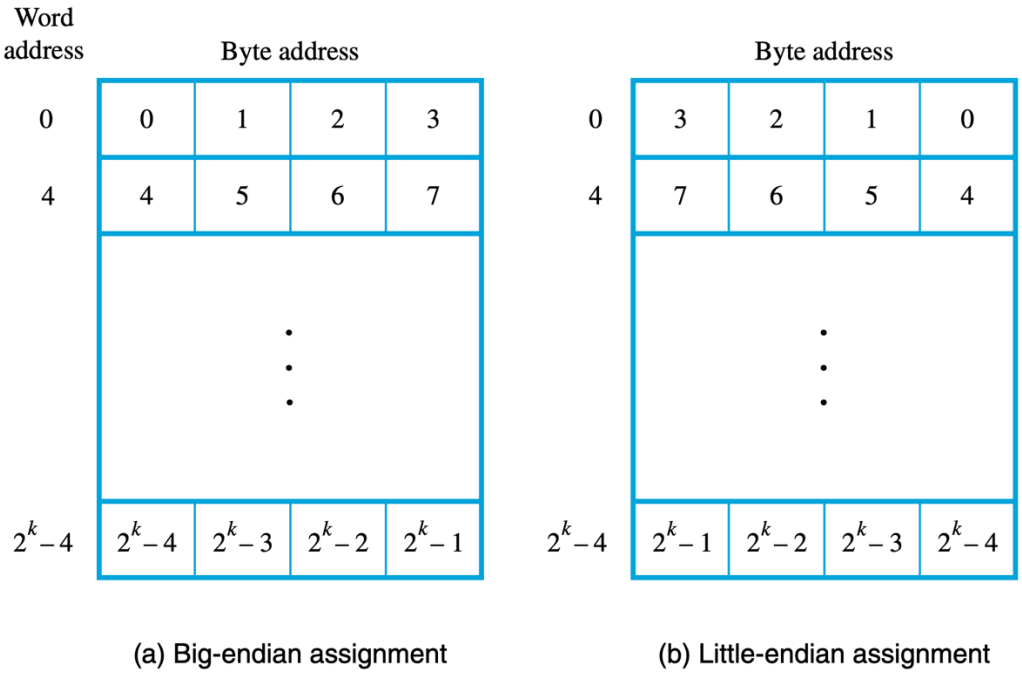
TOPIC 1: Memory Locations and Addresses

- The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1.
- The usual approach is to deal with them in groups of fixed size.
- Each group of n bits is referred to as a word of information, and n is called the word length.
- memory of a computer can be schematically represented as a collection of words
- Modern computers have word lengths that typically range from 16 to 64 bits.
- Machine instructions may require one or more words for their representation.
- Every word, requires a distinct *addresses* for each location such as 0 to $2^k - 1$.
- Thus, the memory can have up to 2^k addressable locations.



Addressability:

- Word length typically ranges from 16 to 64 bits.
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations.
- The term byte-addressable memory is used for this assignment.
- There are two methods for storing the data in the memory one is little endian method and big end method.
- In big endian method the first byte data is stored from MSByte location where as in little end method the first byte data is stored from LSByte location.
- The Addresses for word remains same.
- it is also necessary to specify the labelling of bits within a byte.
- The common convention for a byte storage remains same in both method for byte addressability b7, b6,...,b0, from left to right .



(a) Big-endian assignment (b) Little-endian assignment

Word Alignment:

- Natural word boundaries occur at addresses 0, 4, 8, . . .
- The word locations have aligned addresses if they begin at a byte address that is a multiple of the number of bytes in a word.
- There is no fundamental reason why words cannot begin at an arbitrary byte address.
- The most common case is to use aligned addresses, which makes accessing of memory operands more efficient

Question: Consider a data which needs to be stored in the byte addressability format for a memory with word length of 32-bit.

0X1A2B3C4D5E6F0784
Represent the given number in byte addressability with both big end and little end method.

Answer:

- A 32-bit word length memory will be able to manage 4 bytes of data in a single word
- The current data needs to be stored in the memory is in hexadecimal representation
- hence the bytes will be [1A], [2B], [3C], [4D], [5E], [6F], [07], [84]

Big-End method Byte addressability

Representation will be as follows

In byte addressability 8-bit data go together
In Hexadecimal:

Word 1:	[1A]	[2B]	[3C]	[4D]
Word 2:	[5E]	[6F]	[07]	[84]

In Binary:

Word 1:	[0001 1010]	[0010 1011]	[0011 1100]	[0100 1101]
Word 2:	[0101 1110]	[0110 1111]	[0000 0111]	[1000 0100]

Little-End method Byte addressability

Representation will be as follows

In byte addressability 8-bit data go together
In Hexadecimal:

Word 1:	[4D]	[3C]	[2B]	[1A]
Word 2:	[84]	[07]	[6F]	[5E]

In Binary:

Word 1:	[0100 1101]	[0011 1100]	[0010 1011]	[0001 1010]
Word 2:	[1000 0100]	[0000 0111]	[0110 1111]	[0101 1110]

TOPIC 2: Memory Operations

- Both program instructions and data operands are stored in the memory.
- To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor.
- Operands and results must also be moved between the memory and the processor.
- Thus, two basic operations involving the memory are needed, namely, Read and Write.

  Read operation: transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its

contents be read. The memory reads the data stored at that address and sends them to the processor.

Write operation: it transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

TOPIC 3: Instruction Notations

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

- Used to describe the transfer of information from one location in a computer to another.
- Possible transfers are memory locations, processor registers, or registers in the I/O subsystem
- It has some sources and destination.

a. Memory to Register Transfer

In this the data is transferred from some place in memory/location/local variable to a processor register. The notation is described

$R2 \leftarrow [LOC]$

b. Memory to Register Transfer

$R4 \leftarrow [R2] + [R3]$

c. Memory to I/O Device

$R4 \leftarrow [\text{Input Device}]$

$[\text{OUT STATUS}] \leftarrow R4$

Assembly Language Notation

Let us take an example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Add R4, R2, R3 //This is a three address instruction format

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

- An instruction specifies an operation to be performed and the operands involved.
- Operations are defined by using mnemonics, which are typically abbreviations of the words describing the operations.
- Address format may be Three-Address, Two-Address, One-Address, and Zero Address as given below.

Three-Address Instructions

Evaluate X = (A + B) * (C + D) using Three-Address instructions

- ADD R1, A, B // R1 ← M[A] + M[B]
- ADD R2, C, D // R2 ← M[C] + M[D]
- MUL X, R1, R2 // M[X] ← R1 * R2

Two-Address Instructions

Evaluate X = (A + B) * (C + D) using Two-Address instructions

- MOV R1, A // R1 ← M[A]
- ADD R1, B // R1 ← R1 + M[B]
- MOV R2, C // R2 ← M[C]
- ADD R2, D // R2 ← R2 + M[D]
- MUL R1, R2 // R1 ← R1 * R2
- MOV X, R1 // M[X] ← R1

One-Address Instructions

Use an implied AC register for all data manipulation

Evaluate X = (A + B) * (C + D) using One-Address instructions

- LOAD A AC ← M[A]
- ADD B AC ← AC + M[B]
- STORE T M[T] ← AC
- LOAD C AC ← M[C]
- ADD D AC ← AC + M[D]
- MUL T AC ← AC * M[T]
- STORE X M[X] ← AC

Here AC is accumulator

Zero-Address Instructions

Evaluate X = (A + B) * (C + D) using One-Address instructions

- PUSH A TOS ← A
- PUSH B TOS ← B
- ADD TOS ← (A + B)
- PUSH C TOS ← C
- PUSH D TOS ← D
- ADD TOS ← (C + D)
- MUL TOS ← (C + D) * (A + B)
- POP X M[X] ← TOS

The best way to write zero address program is using **reverse polish notation**.

For arithmetic expressions: A + B

- A + B is a Infix notation
- + A B is a Prefix or Polish notation
- A B + is a Postfix or reverse Polish notation
- The reverse Polish notation is very suitable for stack manipulation
- e.g. A * B + C * D can be written as = AB * CD * +

Questions on reverse polish notation for writing program in zero address format.

Give Zero, one, two and three address programs for given equations:

Q1: A * B + C * D + E * F

Ans: AB*CD*EF*+ +

Q2: A * B + A * (B * D + C * E)

Ans: AB*ABD*CE*+*+

Q3: [A * [B + C * (D + E)]] / [F * (G + H)]

Ans: ABCDE+*+*FGH+*/

Do the following question by yourself

Q4. (A + B * C) / (D – E * F + G * H)

Q5. A + B * [C * D + E * (F + G)]

Q6. (A + B – C) / (D * (5 * F – G))

Q7. (A – B + C * (D * E – F)) / (G + H * K)

Some questions with their zero, one, two, three address format programs

A * B + C * D + E * F			
A B * C D * E F * + +			
0 Address	1 Address	2 Address	3 Address (DEST, SRC, SRC)

PUSH A
PUSH B
MUL
PUSH C
PUSH D
MUL
PUSH E
PUSH F
MUL
ADD
ADD
POP X
	LOAD A
	MUL B
	STORE T
	LOAD C
	MUL D
	ADD T
	STORE T
	LOAD E
	MUL F
	ADD T
	STORE X
		MOV R1, A
		MUL R1, B
		MOV R2, C
		MUL R2, D
		ADD R1,R2
		MOV R2, E
		MUL R2,F
		ADD R1,R2
		MOV X,R1
			MUL R1,A,B
			MUL R2,C,D
			ADD R1,R1,R2
			MUL R2,E,F
			ADD X,R1,R2

A * B + A * (B * D + C * E)			
A B * A B D * C E * + * +			
0 Address	1 Address	2 Address	3 Address (DEST, SRC, SRC)

PUSH A
PUSH B
MUL
PUSH A
PUSH B
PUSH D
MUL
PUSH C
PUSH E
MUL
ADD
MUL
ADD
POP X
	LOAD C
	MUL E
	STORE T
	LOAD B
	MUL D
	ADD T
	MUL A
	STORE T
	LOAD A
	MUL B
	ADD T
	STORE X
		MOV R1,C
		MUL R1,E
		MOV R2,B
		MUL R2,D
		ADD R1,R2
		MUL R1,A
		MOV R2,A
		MUL R2,B
		ADD R1,R2
		MOV X,R1
			MUL R1,C,E
			MUL R2,B,D
			ADD R1,R1,R2
			MUL R1,R1,A
			MUL R2,A,B
			ADD X,R1,R2

[A * [B + C * (D + E)]] / [F * (G + H)]			
A B C D E + * + * F G H + * /			
0 Address	1 Address	2 Address	3 Address (DEST, SRC, SRC)

PUSH A
PUSH B
PUSH C
PUSH D
PUSH E
ADD
MUL
ADD
MUL
PUSH F
PUSH G
PUSH H
ADD
MUL
DIV
POP X
	LOAD H
	ADD G
	MUL F
	STORE T
	LOAD E
	ADD D
	MUL C
	ADD B
	MUL A
	DIV T
	STORE X
		MOV R1,E
		ADD R1,D
		MUL R1,C
		ADD R1,B
		MUL R1,A
		MOV R2,H
		ADD R2,G
		MUL R2,F
		DIV R1,R2
		MOV X,R1
			ADD R1,D,E
			MUL R1,R1,C
			ADD R1,R1,B
			MUL R1,R1,A
			ADD R2,G,H
			MUL R2,R2,F
			DIV X,R1,R2

(A + B * C) / (D – E * F + G * H)			
A B C * + D E F * – G H * + /			
0 Address	1 Address	2 Address	3 Address (DEST, SRC, SRC)

PUSH A
PUSH B
PUSH C
MUL
ADD
PUSH D
PUSH E
PUSH F
MUL
PUSH G
PUSH H
MUL
ADD
DIV
POP X
	LOAD E
	MUL F
	STORE T
	LOAD D
	SUB T
	STORE T
	LOAD G
	MUL H
	ADD T
	STORE T
	LOAD B
	MUL C
	ADD A
	DIV T
	STORE X
		MOV R2,E
		MUL R2,F
		MOV R1,D
		SUB R1,R2
		MOV R2, G
		MUL R2,H
		ADD R1,R2
		LOAD R2,B
		MUL R2,C
		ADD R2,A
		DIV R2,R1
		MOV R2,X
			MUL R1,E,F
			SUB R2, D,R1
			MUL R1,G,H
			ADD R2,R1,R2
			MUL R1,B,C
			ADD R1,R1,A
			DIV X,R1,R2

(A + B – C) / (D * (5 * F – G))			
A B + C – D 5 F * G – * /			
0 Address	1 Address	2 Address	3 Address (DEST, SRC, SRC)

PUSH A
PUSH B
ADD
PUSH C
SUB
PUSH D
PUSH 5
PUSH F
MUL
PUSH G
SUB
MUL
DIV
POP X
	LOAD 5
	MUL F
	SUB G
	MUL D
	STORE T
	LOAD A
	ADD B
	SUB C
	DIV T
	STORE X
		MOV R1,A
		ADD R1,B
		SUB R1,C
		MOV R2,F
		MUL R2,5
		SUB R2,G
		MUL R2,D
		DIV R1,R2
		MOV X,R2
			MUL R2,5,F
			SUB R1,R1,G
			MUL R2,R2,D
			ADD R1,A,B
			SUB R1,R1,C
			DIV X,R1,R2

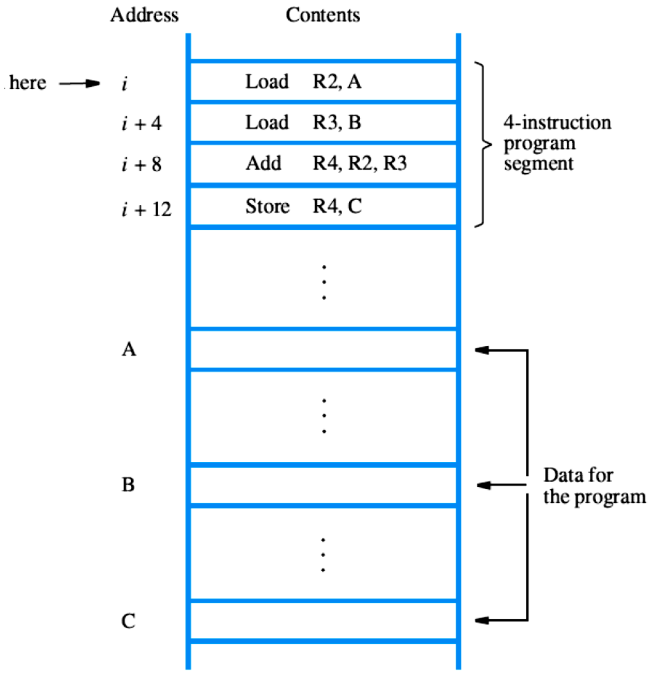
(A – B + C * (D * E – F)) / (G + H * K)			
A B – C D E * F – * + G H K * + /			
0 Address	1 Address	2 Address	3 Address (DEST, SRC, SRC)

PUSH A
PUSH B
SUB
PUSH C
PUSH D
PUSH E
MUL
PUSH F
SUB
MUL
ADD
PUSH G
PUSH H
PUSH K
MUL
ADD
DIV
POP X
	LOAD H
	MUL K
	ADD G
	STORE T
	LOAD D
	MUL E
	SUB F
	MUL C
	ADD A
	SUB B
	DIV T
	STORE X
		MOV R2,H
		MUL R2,K
		ADD R2,G
		MOV R1,D
		MUL R1,E
		SUB R1,F
		MUL R1,C
		ADD R1,A
		SUB R1,B
		DIV R1,R2
		MOV X,R1
			MUL R2,H,K
			ADD R2,R2,G
			MUL R1,D,E
			SUB R1,R1,F
			MUL R1,R1,C
			SUB R3,A,B
			ADD R1,R1,R3
			DIV X,R1,R2

TOPIC 4: Instruction Sequencing

Straight Line Instruction Sequencing

- the task C = A + B, implemented as C ← [A] + [B], as an example.
- Figure shows a possible program segment for this task as it appears in the memory of a computer.
- We assume that the word length is 32 bits and the memory is byte-addressable.
- The four instructions of the program are in successive word locations, starting at location *i*.
- Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses *i* + 4, *i* + 8, and *i* + 12.
- Let us consider how this program is executed. The processor contains a register called the program counter (PC), which holds the address of the next instruction to be executed.
- To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC.
- Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing.
- During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location i + 12 is executed, the PC contains the value i + 16, which is the address of the first instruction of the next program segment.



A program for C ← [A] + [B].

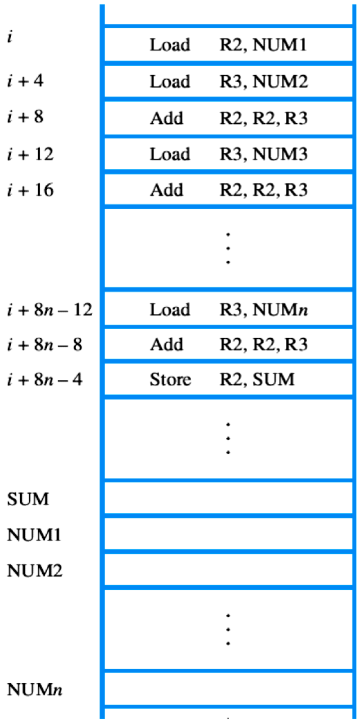


Figure 2.5 A program for adding *n* numbers.

TOPIC 5: Instruction Branching

- Consider the task of adding a list of *n* numbers to understand instruction branching.
- The addresses of the memory locations containing the *n* numbers are symbolically given as NUM1, NUM2,..., NUM*n*
- All the numbers have been added, the result is placed in memory location SUM.
- Instead of using a long list of Load and Add instructions, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum.
- To add all numbers, the loop has to be executed as many times as there are numbers in list.
- We now introduce *branch* instructions, this instructions loads a new address into the program counter.
- As a result, the processor fetches and executes the instruction at this new address, called the *branch target*.
- instead of sequential address order.
- A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.
- This method of looping some instruction based on a specified condition is known as instruction branching.

Instruction Sequencing

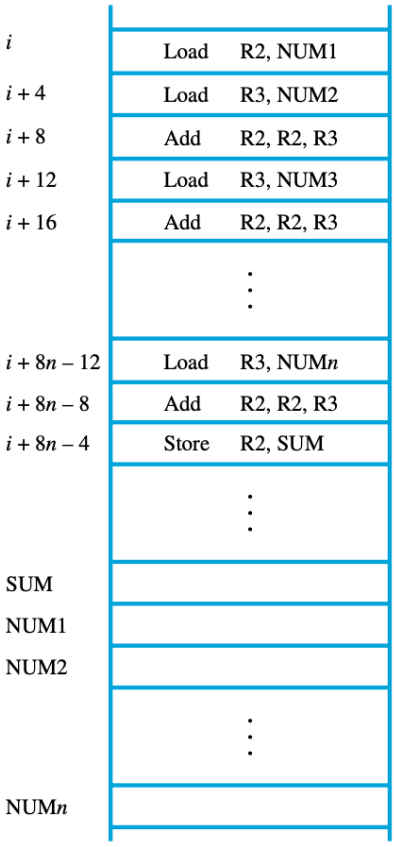


Figure 2.5 A program for adding *n* numbers.

Instruction Sequencing

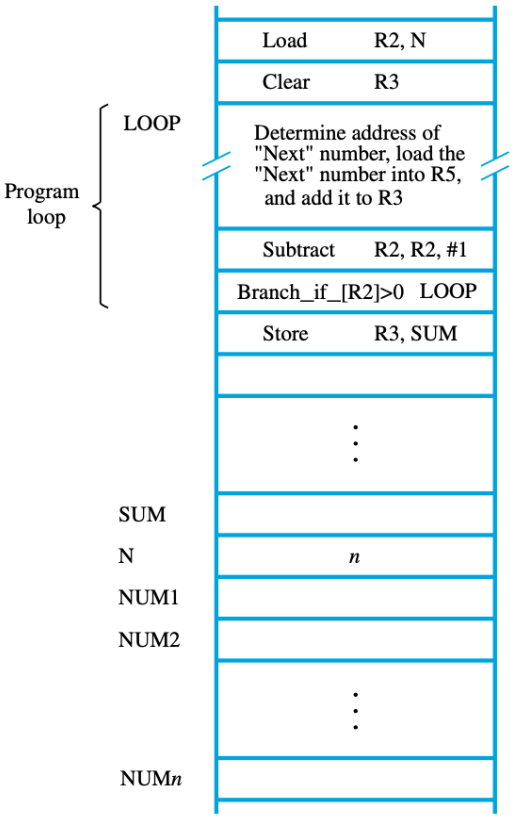


Figure 2.6 Using a loop to add *n* numbers.

TOPIC 6: Addressing Modes

The different ways for specifying the locations of instruction operands are known as *addressing modes*.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = effective address
Value = a signed number
X = index value

Immediate mode—The operand is given explicitly in the instruction.

For example, the instruction

Add R4, R6, $200_{immediate}$

Add R4, R6, #200

Register mode—The operand is the contents of a processor register; the name of the register is given in the instruction.

Add R4, R2, R3

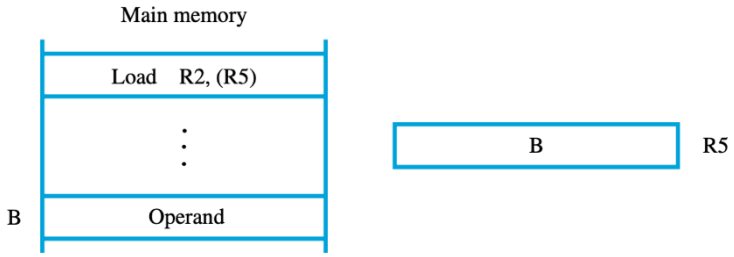
Absolute mode—The operand is in a memory location; the address of this location is given explicitly in the instruction.

Load R2, NUM1

Implied Mode—When the effective address of the operand is at a default address.

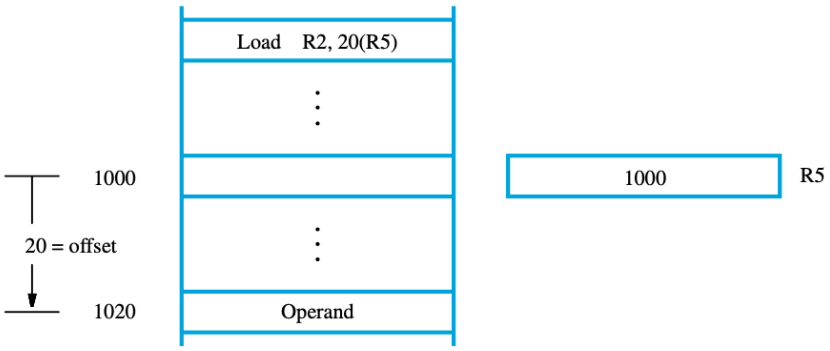
CMA //Complement the content of accumulator

Indirect mode—The effective address of the operand is the contents of a register that is specified in the instruction.

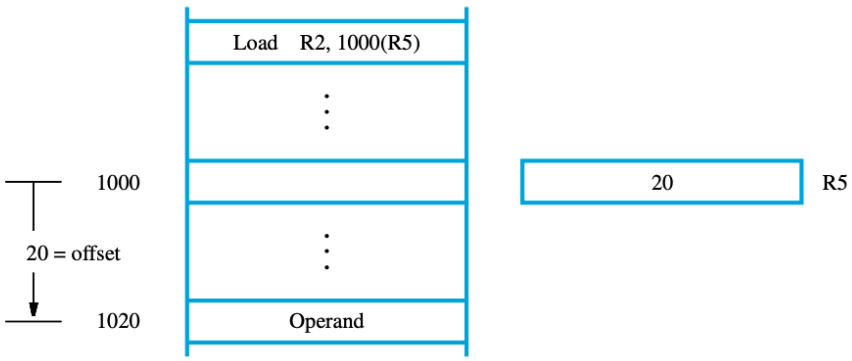


Index mode—The effective address of the operand is generated by adding a constant value to the contents of a register.

X(R*i*)
EA = X + [R*i*]



(a) Offset is given as a constant

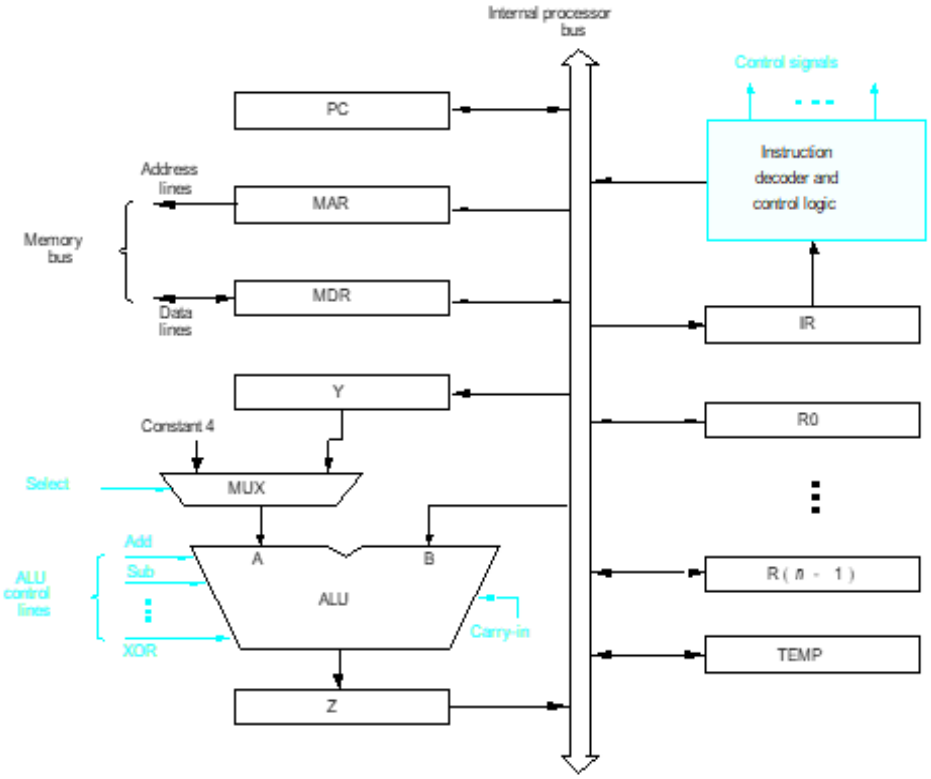


(b) Offset is in the index register

TOPIC 7: Execution of a Complete Instruction with Single Bus

Ex: Add (R3), R1

Step 1- Fetch the instruction from memory to instruction register (IR)
Step 2 - Fetch first operand from memory
Step 3 - Perform arithmetic addition operation
Step 4 - Store/Load results in register R1



Single-bus organization of the datapath inside a processor.

Control Bit Sequence:

Step No.	Control Bit Sequence	Step Detail
1	PCout, MARin, Read, Select-4, Add, Zin	The first instruction of the program in memory has been initiated by the program counter and next address is calculated by adding 4 bytes.
2	Zout, PCin, WMFC	Next address is pointed by program counter, and memory function is completed
3	MDRout, IRin	The machine code for first opcode is transferred from MDR to Instruction register (IR)
4	R3out, MARin, Read	Operand is present at address contained in R3, So operand will be read from MARin with the help of R3out
5	R1out, Y, WMFC	All operands are read here and memory function is completed

6	MDRout, Select Y, Add, Zin	Addition is performed on operands, Selection Y is used to take R1out data from bus to ALU via multiplexer, and output is available in Z
7	Zout, R1in, End	Output is stored in R1 and then program ends

Note: Also refer the diagram taught in class where all processor components were discussed.

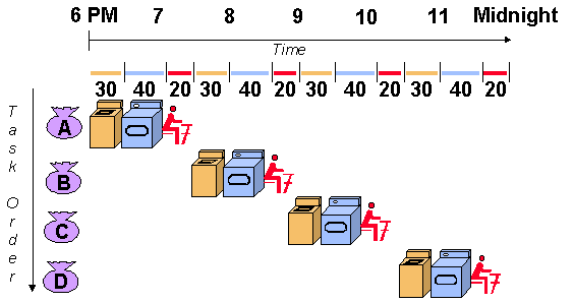
Concept of Pipelining: How Pipelining Works?

Pipelining, a standard feature in RISC processors, in this processor works on different steps of the instruction at the same time, such that more instructions can be executed in a shorter period of time.

Example: Let's say that there are four loads of dirty laundry that need to be washed, dried, and folded. We could put the first load in the washer for 30 minutes, dry it for 40 minutes, and then take 20 minutes to fold the clothes. Then pick up the second load and wash, dry, and fold, and repeat for the third and fourth loads. Supposing we started at 6 PM and worked as efficiently as possible, we would still be doing laundry until midnight.

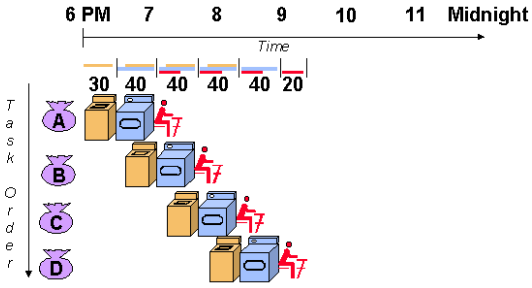
However, a smarter approach to the problem would be to put the second load of dirty laundry into the washer after the first was already clean and whirling happily in the dryer. Then, while the first load was being folded, the second load would dry, and a third load could be added to the pipeline of laundry. Using this method, the laundry would be finished by 9:30.

Conventional Approach



Total Time - 6 Hrs

Smarter Pipelined Approach

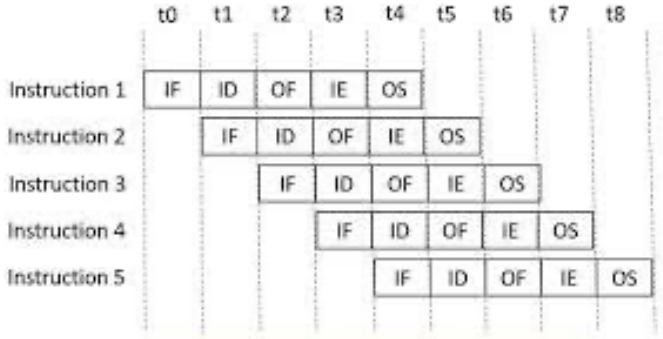


Total Time - 3 Hrs 30 Minutes

In the above example we can replace the wash, dry, and fold stapes as the processor steps given below. In processor there are following steps which are basically done when a instruction is executed.

1. Instruction Fetch - IF
2. Instruction Decode - ID
3. Instruction Execute - IE
4. Operand Fetch - IF
5. Output Store - OS

- Total loads can be represented as total instruction.
- Stage delay time can be represented as t0, t1,.., t8



Pipelining of 5 Instructions

- The above instruction steps are executed and results are stored in temporary registers known as buffer registers.
- Each step has its own time required to complete the step which is known as stage delay.
- The maximum stage delay of a step is an important parameter.
- For example in the laundry system question, if the maximum stage delay which is 40 in present case (for drying) is increased than the overall execution time will be increased and hence the throughput will decrease
- Throughput is a total no. of instruction which can be executed in a specified time.
- When the maximum stage delay increases the total throughput decreases.

Question: Stage Delays in a 4-stage pipeline are 800, 300, 200, and 100 picoseconds. The first stage (with a delay of 800 picoseconds) is replaced with a functionally equivalent design involving two stages with respective delays of 600 and 100 picoseconds. Find the throughput increase of the pipeline. [GATE: CSE 2016]

Answer:

In the given question the maximum stage delay in conventional steps is 800 ps. We know that as the maximum stage delay is inversely proportional to the through put. Hence throughput in the case when stage delays are [800, 300, 200, and 100].

T_1 = 1/800; // The maximum stage delay value will be taken

Now the first stage (with a delay of 800 picoseconds) is replaced with a functionally equivalent design involving two stages with respective delays of 600 and 100 picoseconds. so now stage delays will be [600, 100, 300, 200, 100].

Hence throughput in this case will be.

T_2 = 1/600; // The maximum stage delay value will be taken

Increase in throughput $= \left[\frac{(T_2 - T_1)}{T_1}\right] \times 100$

$= \left[\frac{\left(\frac{1}{600} - \frac{1}{800}\right)}{\frac{1}{800}}\right] \times 100$

$= 33.33\%$

So the total throughput will get increased by 33.33%.

Question: Stage Delays in a 4-stage pipeline are 400, 300, 200, and 100 picoseconds. The first stage (with a delay of 400 picoseconds) is replaced with a functionally equivalent design involving two stages with respective delays of 500 and 100 picoseconds. Find the throughput decrease of the pipeline.

In the given question the maximum stage delay in conventional steps is 400 ps. We know that as the maximum stage delay is inversely proportional to the through put. Hence throughput in the case when stage delays are [400, 300, 200, and 100].

T_1 = 1/400; // The maximum stage delay value will be taken

Now the first stage (with a delay of 400 picoseconds) is replaced with a functionally equivalent design involving two stages with respective delays of 500 and 100 picoseconds. so now stage delays will be [500, 100, 300, 200, 100].

Hence throughput in this case will be.

T_2 = 1/500; // The maximum stage delay value will be taken

decrease in throughput $= \left[\frac{(T_1 - T_2)}{T_1}\right] \times 100$

$= \left[\frac{\left(\frac{1}{400} - \frac{1}{500}\right)}{\frac{1}{400}}\right] \times 100$

= 20%

Question: Define instruction level, arithmetic level pipelining [Self-study]

Refer Book For Remaining Topics Below:

- Control Unit Operations: Instruction sequencing, Micro operations and Register Transfer. Hardwired Control,
- Micro-programmed Control: Basic concepts, Microinstructions and micro- program sequencing
- Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement