Computer Organization

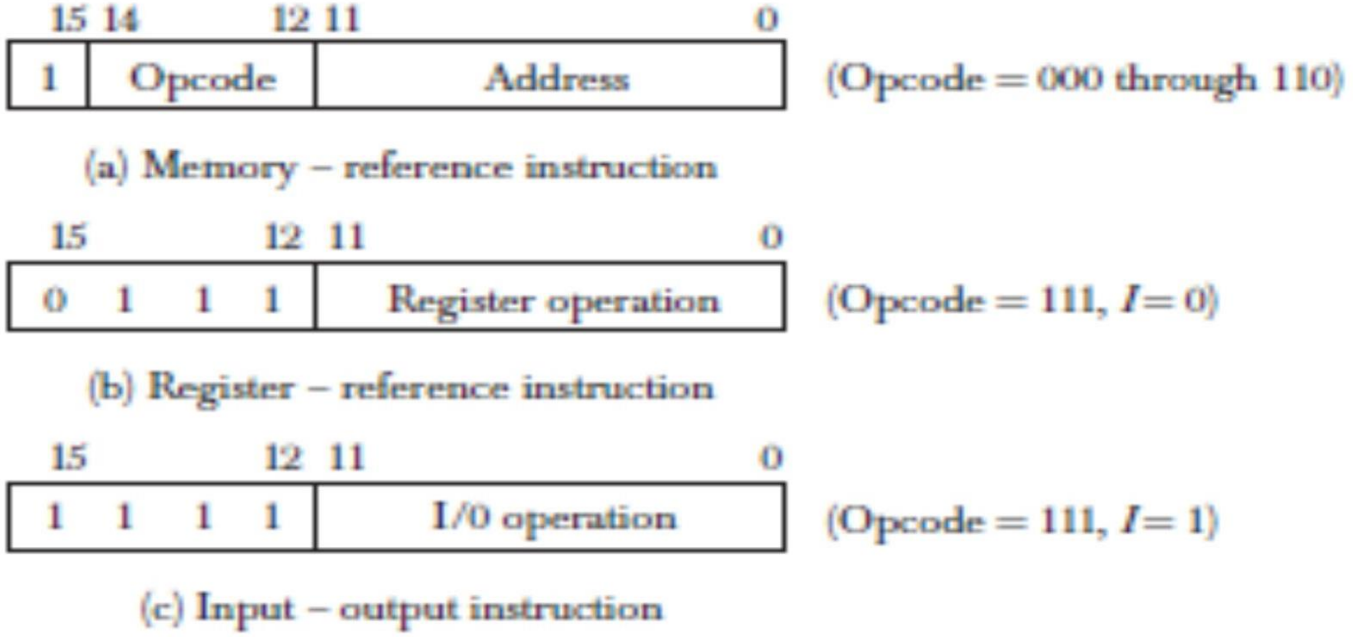
Instruction

An instruction is an order given to a computer processor by a computer program. At the lowest level, each instruction is a sequence of 0s and 1s that describesaphysical operation the computer is to perform

- In a computer's assembler language, each language statement generallycorresponds to a single processor instruction.
- In high-level languages, a language statement generally results (after program compilation) in multiple processor instructions.
- For **example**, consider the **instruction**, Add (R1), R0. To execute the Add **instruction**, the processor uses the value in register R1 as the effective address of the operand

Instructions

- The basic computer has three instruction code formats, Each format has 16 bits.
- The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode *I*.
- *I* is equal to 0 for direct address and to 1 for indirect address
- The register reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction.



Memory address

Address(12 bits)	Bits(16 bits)
111111111111	
111111111110	
----------	----------
000000000011	0001
000000000010	7f3a
000000000001	67ff
000000000000	0002

Memory size: 4096x16

4Kx16

Instructions

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
HLT	7001		Halt computer
INP	F800		Input character to *AC*
OUT	F400		Output character from *AC*
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

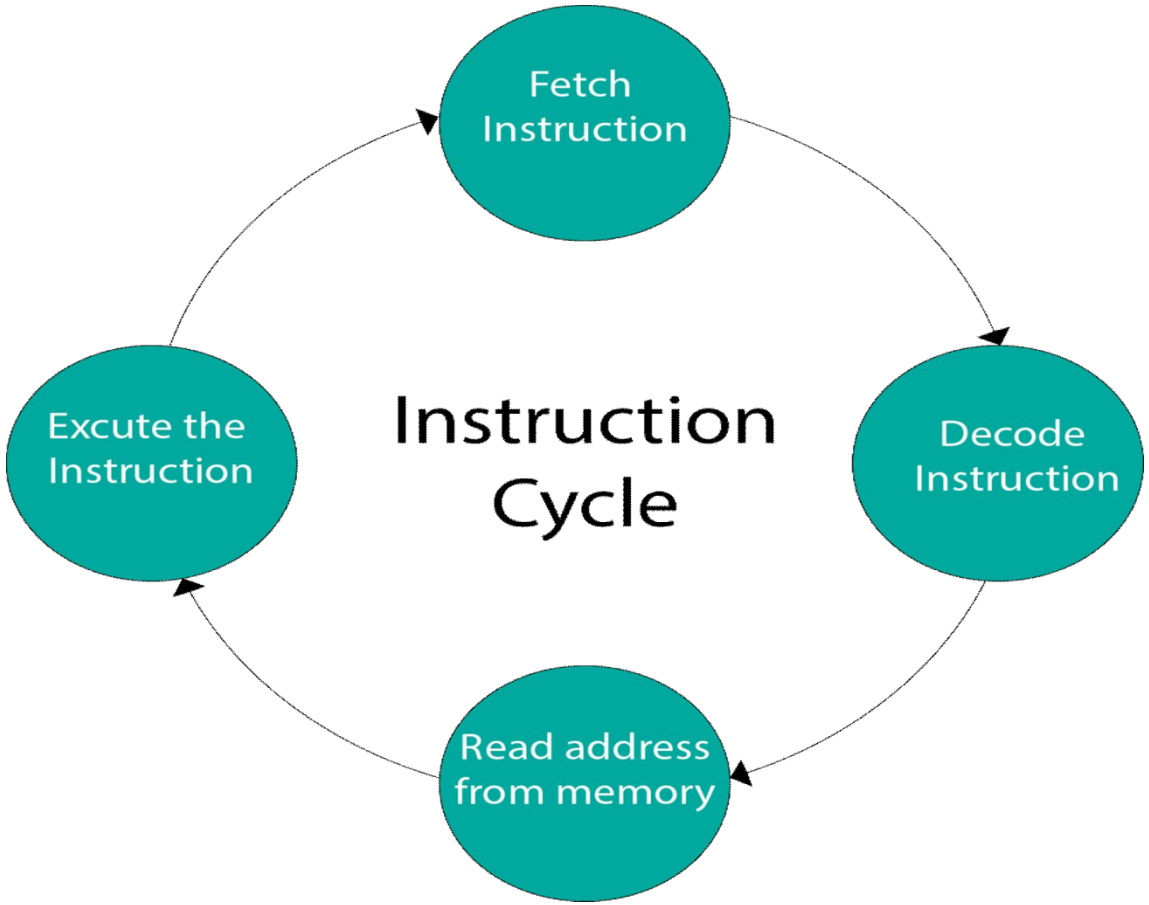
Instruction Cycle

- A program residing in the memory unit of a computer consists of a sequence of instructions.
- These instructions are executed by the processor by going through a cycle for each instruction.

In a basic computer, each instruction cycle consists of the following phases:

- Fetch instruction from memory.
- Decode the instruction.
- Read the effective address from memory.
- Execute the instruction.

Instruction Cycle



Instruction Cycle

Fetch and Decode:

- Initially, the program counter *PC* is loaded with the address of the first instruction in the program.
- The sequence counter *SC* is cleared to 0, providing a decoded timing signal *T*0.
- After each clock pulse, *SC* is incremented by one, so that the timing signals go through a sequence *T*0, *T*1, *T*2, and so on.
- The micro operations for the fetch and decode phases can be specified by the following register transfer statements.

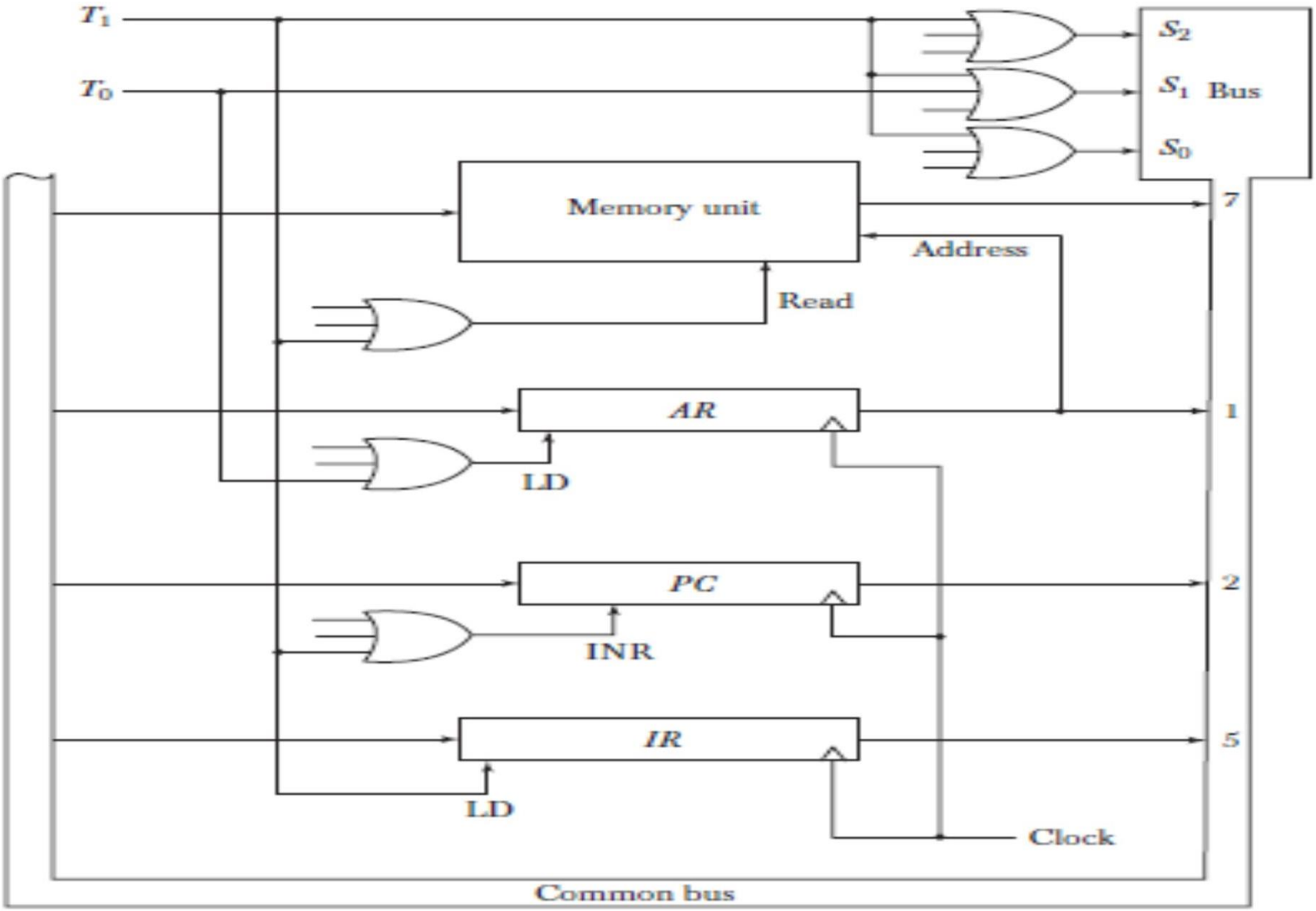
$$T_0: \quad AR \leftarrow PC$$

$$T_1: \quad IR \leftarrow M[AR],\ PC \leftarrow PC + 1$$

$$T_2: \quad D_0, \ldots, D_7 \leftarrow \text{Decode } IR(12\text{–}14),\ AR \leftarrow IR(0\text{–}11),\ I \leftarrow IR(15)$$

Instruction Cycle

Figure 5-8 Register transfers for the fetch phase.



Instruction Cycle

1. Place the content of *PC* onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
2. Transfer the content of the bus to *AR* by enabling the LD input of *AR*.

The next clock transition initiates the transfer from *PC* to *AR* since $T_0 = 1$. In order to implement the second statement

$$T_1: \qquad IR \leftarrow M[AR], \qquad PC \leftarrow PC + 1$$

it is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
3. Transfer the content of the bus to *IR* by enabling the LD input of *IR*.
4. Increment *PC* by enabling the INR input of *PC*.

Determine the Type of Instruction

- if *D7* = 1, the instruction must be a register-reference or input–output type.
- If *D*7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.
- Control then inspects the value of the first bit of the instruction, which is now available in flip-flop *I*.
- If *D*7= 0 and *I* =1, we have a memory reference instruction with an indirect address.
- It is then necessary to read the effective address from memory
- The micro operation for the indirect address condition can be symbolized by the register transfer statement $AR \leftarrow M[AR]$

Instruction Cycle

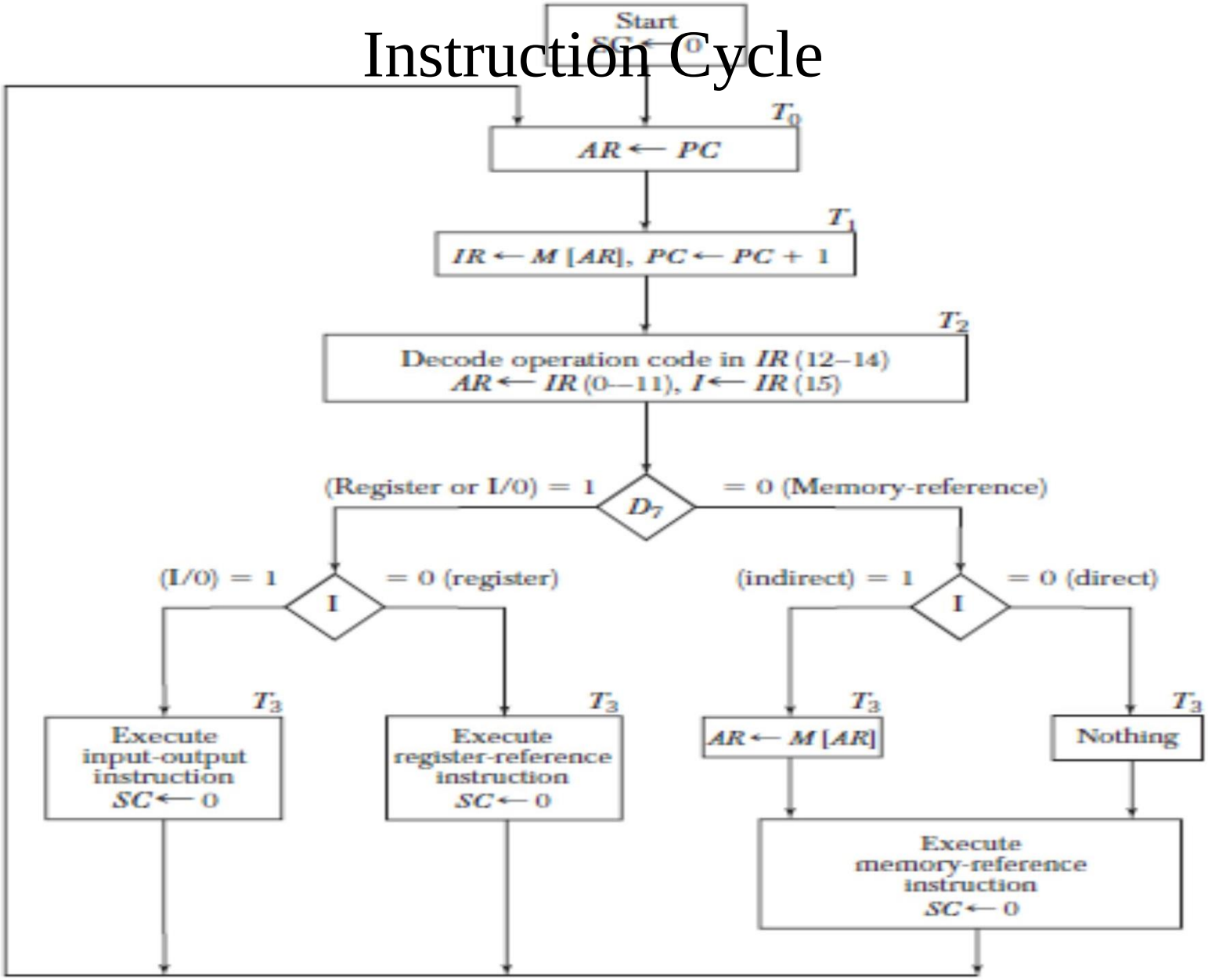


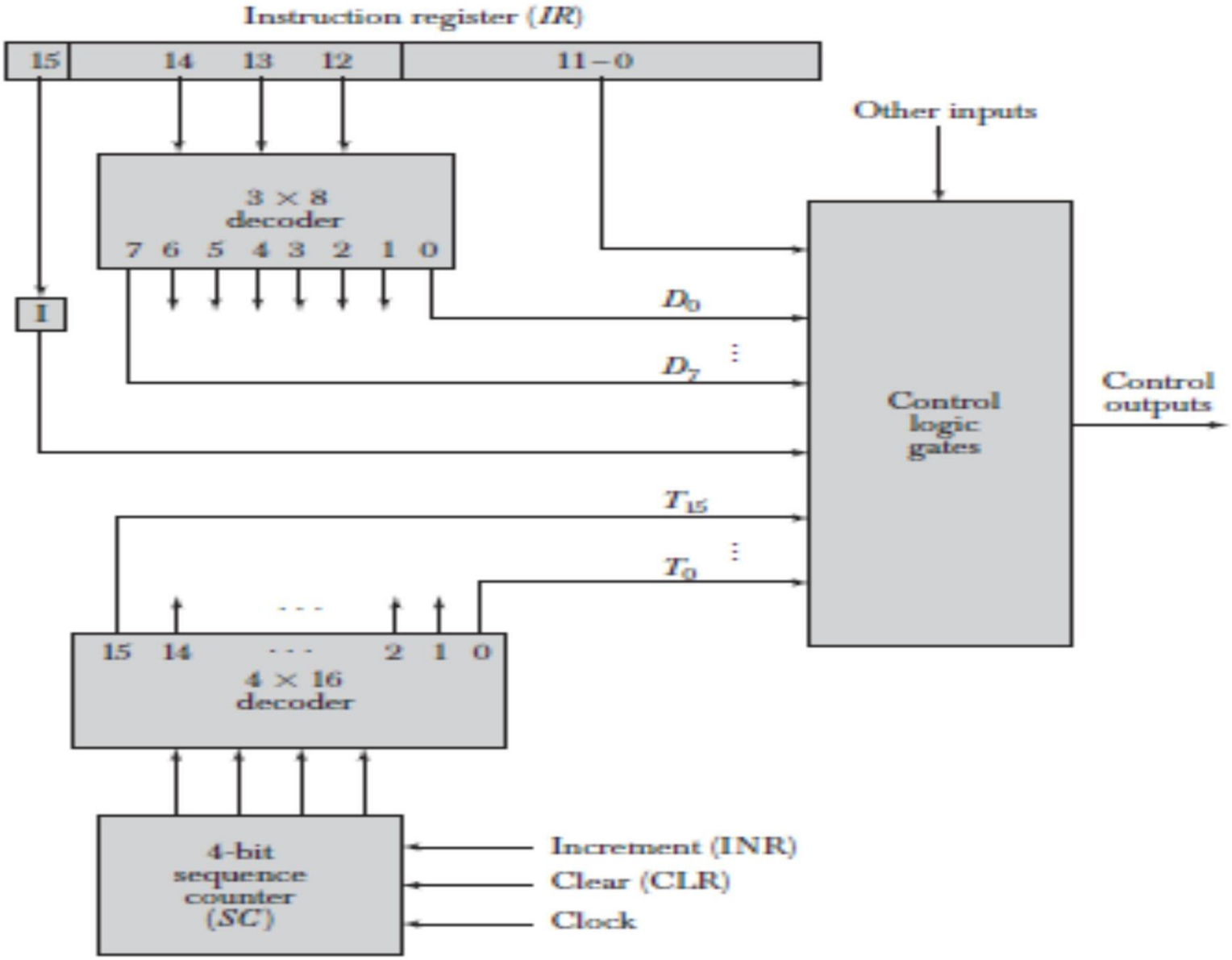
Figure 5-9 Flowchart for instruction cycle (initial configuration).

Instruction Set Completeness

The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

- Arithmetic, logical, and shift instructions
- Instructions for moving information to and from memory and processor Registers
- Program control instructions together with instructions that check status conditions
- Input and output instructions

Control unit



Instruction Formats

- The physical and logical structure of computers is normally described in reference manuals provided with the system.
- Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities.
- They list all hardware-implemented instructions, specify their binary code format, and provide a precise definition of each instruction.
- A computer will usually have a variety of instruction code formats.
- It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.
- The bits of the instruction are divided into groups called fields.
- The most common fields found in instruction formats are:
  - ✓ **1.** An operation code field that specifies the operation to be performed.
  - ✓ **2.** An address field that designates a memory address or a processor register.
  - ✓ **3.** A mode field that specifies the way the operand or the effective address is determined.

Instruction Formats

- Operations specified by computer instructions are executed on some data stored in memory or processor registers.
- Operands residing in memory are specified by their memory address.
- Operands residing in processor registers are specified with a register address.
- A register address is a binary number of k bits that defines one of 2^k registers in the CPU.
- Thus a CPU with 16 processor registers *R*0 through *R*15 will have a register address field of four bits.
- The binary number 0101, for example, will designate register *R*5.
- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

Instruction Formats

Most computers fall into one of three types of CPU organizations:

- Single accumulator organization.
- General register organization.
- Stack organization.

Single Accumulator organization

- All operations are performed with an implied accumulator register.
- The instruction format in this type of computer uses one address field.
- For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD X
- The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$.
- *AC* is the accumulator register and $M[X]$ symbolizes the memory word located at address *X*.

Opcode	Address field

Instruction Formats

General register organization

- The instruction format in this type of computer needs three register address fields.
- Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 to denote the operation $R1 \leftarrow R2 + R3$.
- The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction
- ADD R1, R2 would denote the operation $R1 \leftarrow R1 + R2$.
- Only register addresses for $R1$ and $R2$ need be specified in this instruction.
- Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction.
- Thus the instruction MOV R1, R2 denotes the transfer $R1 \leftarrow R2$ (or $R2 \leftarrow R1$, depending on the particular computer).
- Thus transfer-type instructions need two address fields to specify the source and the destination.
- General register-type computers employ two or three address fields in their instruction format.
- Each address field may specify a processor register or a memory word.
- An instruction symbolized by ADD R1, X would specify the operation $R1 \leftarrow R1 + M[X]$.

Instruction Formats

Stack organization

- Computers with stack organization would have PUSH and POP instructions which require an address field.
- Thus the instruction PUSH X will push the word at address *X* to the top of the stack.
- The stack pointer is updated automatically.
- Operation-type instructions do not need an address field in stack-organized computers.
- This is because the operation is performed on the two items that are on top of the stack.
- The instruction ADD in a stack computer consists of an operation code only with no address field.
- This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- There is no need to specify operands with an address field since all operands are implied to be in the stack.
- Most computers fall into one of the three types of organizations that have just been described.
- Some computers combine features from more than one organizational structure.

Three-Address Instructions

- Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- The program in assembly language that evaluates X = (A + B) * (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

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

It is assumed that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

opcode	Address of Destination Op	Address of operand2	Address of Operand1

Two-Address Instructions

- Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word.
- The program to evaluate $X = (A + B) * (C + D)$ is as follows:

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

- The MOV instruction moves or transfers the operands to and from memory and processor registers.
- The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

Opcode	Address of operand2	Address of operand1

One-Address Instructions

- One-address instructions use an implied accumulator (*AC*) register for all data manipulation.
- For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the *AC* contains the result of all operations.
- The program to evaluate $X = (A + B) * (C + D)$ is

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

- All operations are done between the *AC* register and a memory operand.
- *T* is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- The following program shows how $X = (A + B) * (C + D)$ will be written for a stack-organized computer. (*TOS* stands for top of stack).

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

- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

- The operation code specifies the operation to be performed.
- The mode field is used to locate the operands needed for the operation.
- There may or may not be an address field in the instruction.
- If there is an address field, it may designate a memory address or a processor register.
- The instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Addressing Modes

Implied Mode:

- In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- In fact, all register reference instructions that use an accumulator are implied-mode instructions.
- Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode:

- In this mode the operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand field rather than an address field.
- The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.
- Immediate-mode instructions are useful for initializing registers to a constant value.

Addressing Modes

Register Mode:

- In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.
- A k-bit field can specify any one of 2^k registers.

Register Indirect Mode:

- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.
- In other words, the selected register contains the address of the operand rather than the operand itself.
- Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.
- A reference to the register is then equivalent to specifying a memory address.
- The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Addressing Modes

Autoincrement or Autodecrement Mode:

- This is similar to the register indirect mode except that the register is Incremented or decremented after (or before) its value is used to access memory.

Effective address:

- The *effective address* is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction.

Direct Addressing Mode:

- In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction.

Indirect Address Mode:

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

Addressing Modes

- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

- The CPU register used in the computation may be the program counter, an index register, or a base register.

Addressing Modes

Relative Address Mode:

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.
- The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative.
- When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.
- To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24.
- The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826.
- The effective address computation for the relative address mode is 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction.

Addressing Modes

Indexed Addressing Mode:

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- The index register is a special CPU register that contains an index value.
- The address field of the instruction defines the beginning address of a data array in memory.
- Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register.
- Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value.
- The index register can be incremented to facilitate access to consecutive operands.
- Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.
- Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when, the index-mode instruction is used.
- In computers with many processor registers, any one of the CPU registers can contain the index number.
- In such a case the register must be specified explicitly in a register field within the instruction format.

Addressing Modes

Base Register Addressing Mode:

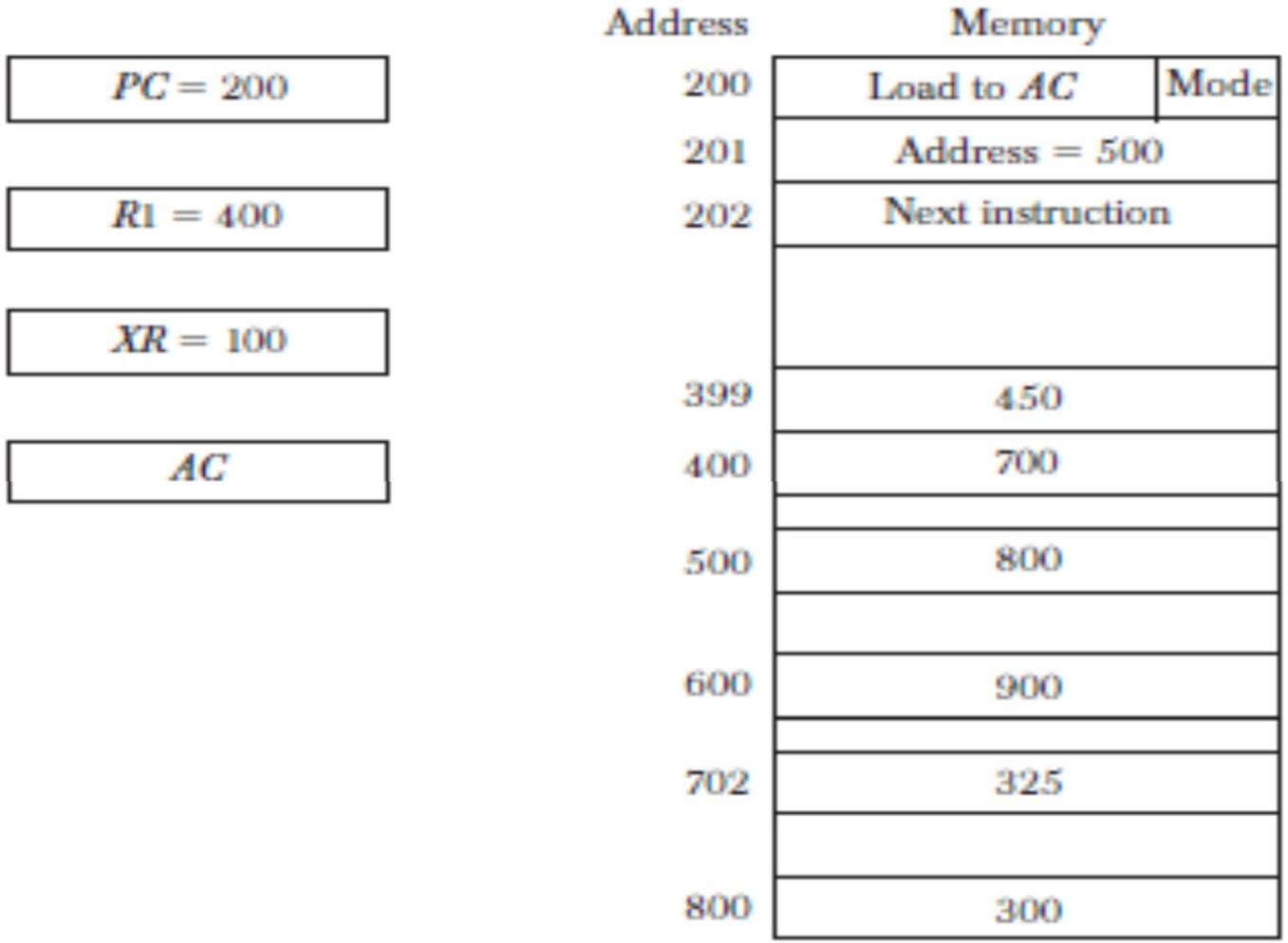
- In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.
- The difference between the two modes is in the way they are used rather than in the way that they are computed.
- An index register is assumed to hold an index number that is relative to the address part of the instruction.
- A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.

Addressing Modes

Numerical Example

- To show the differences between the various modes, we will show the effect of the addressing modes on the instruction.
- The two-word instruction at address 200 and 201 is a “load to *AC*” instruction with an address field equal to 500.
- The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.
- *PC* has the value 200 for fetching this instruction.
- The content of processor register *R*1 is 400, and the content of an index register *XR* is 100.
- *AC* receives the operand after the instruction is executed.
- The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

Addressing Modes



200		
202		

Addressing Modes

Numerical Example

- The mode field of the instruction can specify any one of a number of modes.
- For each possible mode we calculate the effective address and the operand that must be loaded into *AC*.
- In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into *AC* is 800.
- In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into *AC*.
- In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the relative mode the effective address is 500 + 202 = 702 and the operand is 325. (Note that the value in *PC* after the fetch phase and during the execute phase is 202.)
- In the index mode the effective address is *XR* + 500 = 100 + 500 = 600 and the operand is 900.
- In the register mode the operand is in *R*1 and 400 is loaded into *AC*. (There is no effective address in this case.)
- In the register indirect mode the effective address is 400, equal to the content of *R*1 and the operand loaded into *AC* is 700.
- The auto increment mode is the same as the register indirect mode except that *R*1 is incremented to 401 after the execution of the instruction.
- The auto decrement mode decrements *R*1 to 399 prior to the execution of the instruction. The operand loaded into *AC* is now 450.

Addressing Modes

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	–	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

UNIT-II

Pipelining and Vector Processing

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline Vector Processing, Array Processors

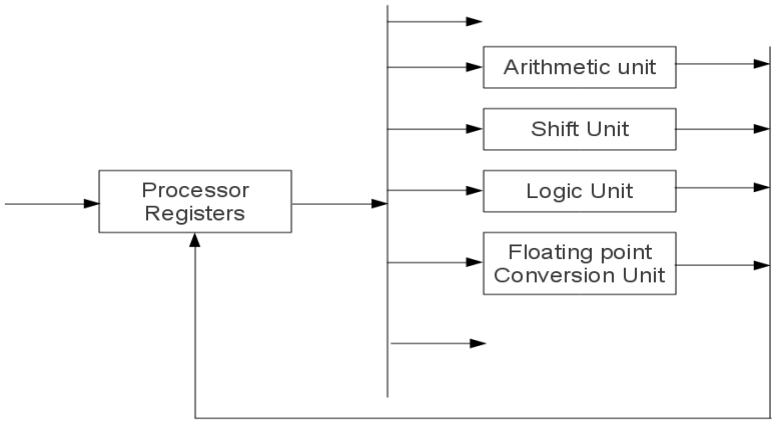
Parallel Processing:

The term parallel processing indicates that the system is able to perform several operations in a single time. Now we will elaborate the scenario, in a CPU we will be having only one Accumulator which will be storing the results obtained from the current operation. Now if we are giving only one command such that "a+b" then the CPU performs the operation and stores the result in the accumulator. Now we are talking about parallel processing, therefore we will be issuing two instructions "a+b" and "c-d" in the same time, now if the result of "a+b" operation is stored in the accumulator, then "c-d" result cannot be stored in the accumulator in the same time. Therefore, the term parallel processing in not only based on the Arithmetic, logic or shift operations. The above problem can be solved in the following manner. Consider the registers R1 and R2 which will be storing the operands before operation and R3 is the register which will be storing the results after the operations. Now the above two instructions "a+b" and "c-d" will be done in parallel as follows.

- Values of "a" and "b" are fetched in to the registers R1 and R2
- The values of R1 and R2 will be sent into the ALU unit to perform the addition
- The result will be stored in the Accumulator
- When the ALU unit is performing the calculation, the next data "c" and "d" are brought into R1 and R2.
- Finally the value of Accumulator obtained from "a+b" will be transferred into the R3
- Next the values of C and D from R1 and R2 will be brought into the ALU to perform the "c-d" operation.
- Since the accumulator value of the previous operation is present in R3, the result of "c-d" can be safely stored in the Accumulator.

This is the process of parallel processing of only one CPU. Consider several such CPU performing the calculations separately. This is the concept of parallel processing.

Concept of Parallel Processing



In the above figure we can see that the data stored in the processor registers is being sent to separate devices basing on the operation needed on the data. If the data inside the processor registers is requesting for an arithmetic operation, then the data will be sent to the arithmetic unit and if in the same time another data is requested in the logic unit, then the data will be sent to logic unit for logical operations. Now in the same time both arithmetic operations and logical operations are executing in parallel. This is called as parallel processing.

Instruction Stream: The sequence of instructions read from the memory is called as an Instruction Stream

Data Stream: The operations performed on the data in the processor is called as a Data Stream.

The computers are classified into 4 types based on the Instruction Stream and Data Stream. They are called as the Flynn's Classification of computers.

Flynn's Classification of Computers:

- Single Instruction Stream and Single Data Stream (SISD)
- Single Instruction Stream and Multiple Data Stream (SIMD)
- Multiple Instruction Stream and Single Data Stream (MISD)
- Multiple Instruction Stream and Multiple Data Stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

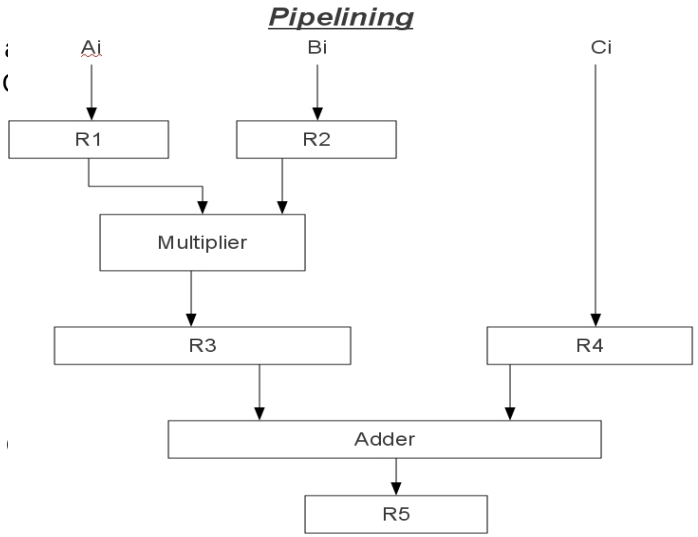
MISD structure is only of theoretical interest since no practical system has been constructed using this organization because Multiple instruction streams means more no of instructions, therefore we have to perform multiple instructions on same data at a time. This is practically impossible.

MIMD structure refers to a computer system capable of processing several programs at the same time operating on different data.

Pipelining: Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. We can consider the pipelining concept as a collection of several segments of data processing programs which will be processing the data and sending the results to the next segment until the end of the processing is reached. We can visualize the concept of pipelining in the example below.

Consider the following operation: Result=(A+B)*C

- First the A and B values are Fetched which is nothing but a “Fetch Operation”.
- The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.
- The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed.
- Finally the Result is again stored in the “Result” variable.



In this process we a

→ Fetch (| Fetch Operation(C) |

Multiplication of ((A+B), C)

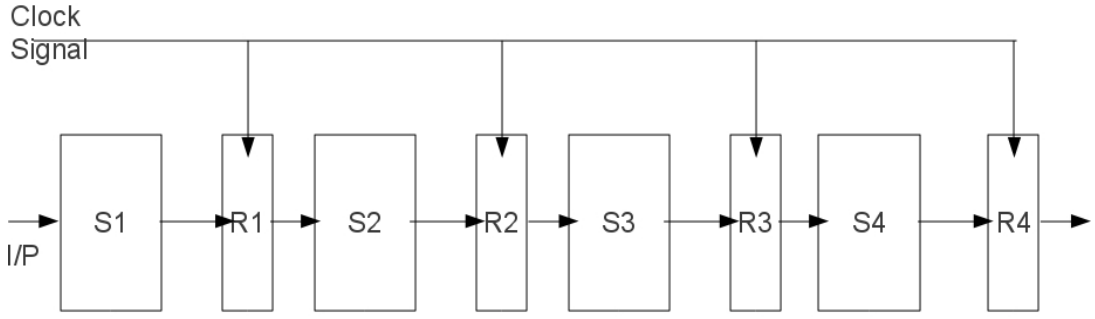
The contents of the Registers in the above pipeline concept are given below. We are considering the implementation of A[7] array with B[7] array.

Clock Pulse Number	Segment1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	-	-	-
2	A2	B2	A1*B1	C1	-
3	A3	B3	A2*B2	C2	A1*B1+C1
4	A4	B4	A3*B3	C3	A2*B2+C2
5	A5	B5	A4*B4	C4	A3*B3+C3
6	A6	B6	A5*B5	C5	A4*B4+C4
7	A7	B7	A6*B6	C6	A5*B5+C5
8			A7*B7	C7	A6*B6+C6
9					A7*B7+C7

If the above concept is executed with out the pipelining, then each data operation will be taking 5 cycles, totally they are 35 cycles of CPU are needed to perform the operation. But if are using the concept of pipeline, we will be cutting off many cycles. Like given in the table below when the values of A1 and B1 are coming into the registers R1 and R2, the registers R3, R4 and R5 are empty. Now in the second cycle the multiplication of A1 and B1 is transferred to register R3, now in this point the contents of the register R1 and R2 are empty. Therefore the next two values A2 and B2 can be brought into the registers. Again in the third cycle after fetching the C1 value the operation (A1*B1)+C1 will be performed. So in this way we can achieve the total concept in only 9 cycles. Here we are assuming that the clock cycle timing is fixed. This is the concept of pipelining.

Below is the diagram of 4 segment pipeline.

Segment Representation



4 Segment pipeline

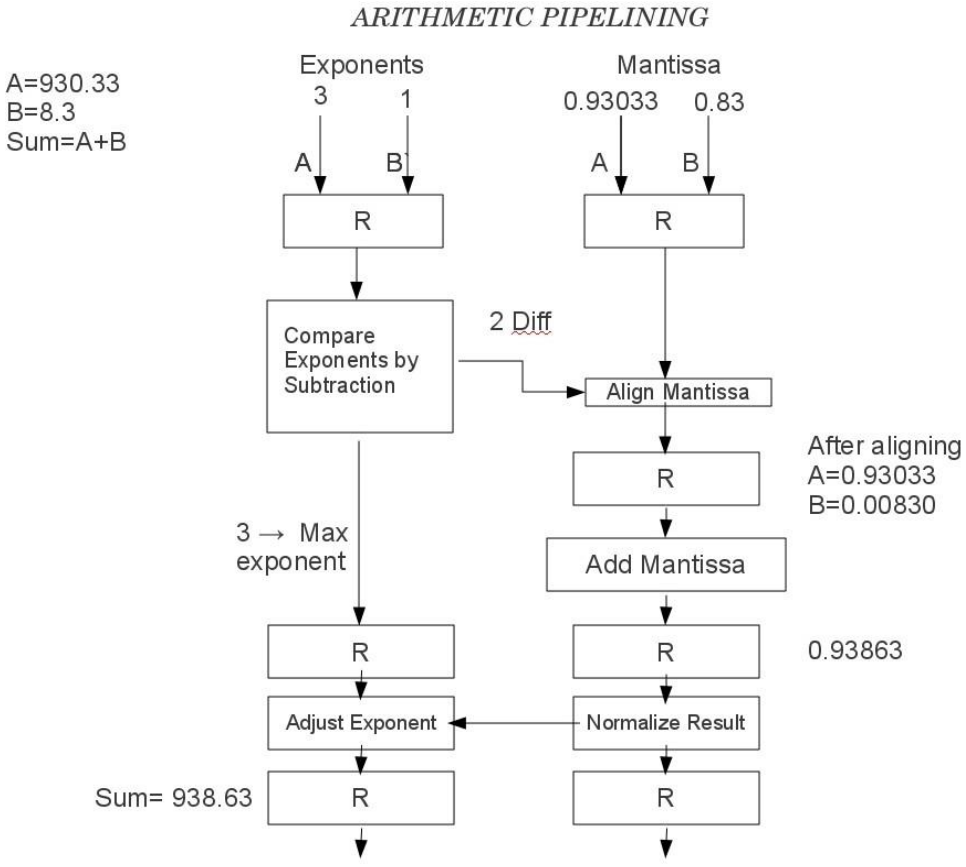
The below table is the space time diagram for the execution of 6 tasks in the 4 segment pipeline.

Space and Time Diagram

Seg/ clock	C1	C2	C3	C4	C5	C6	C7	C8	C9
S1	T1	T2	T3	T4	T5	T6			
S2		T1	T2	T3	T4	T5	T6		
S3			T1	T2	T3	T4	T5	T6	
S4				T1	T2	T3	T4	T5	T6

$$S = nTn/(K+n-1)*tp$$

Arithmetic pipeline:



The above diagram represents the implementation of arithmetic pipeline in the area of floating point arithmetic operations. In the diagram, we can see that two numbers A and B are added together. Now the values of A and B are not normalized, therefore we must normalize them before start to do any operations. The first thing is we have to fetch the values of A and B into the registers. Here R denote a set of registers. After that the values of A and B are normalized, therefore the values of the exponents will be compared in the comparator. After that the alignment of mantissa will be taking place. Finally, we will be performing addition, since an addition is happening in the adder circuit. The source registers will be free and the second set of values can be brought. Like wise when the normalizing of the result is taking place, addition of the new values will be added in the adder

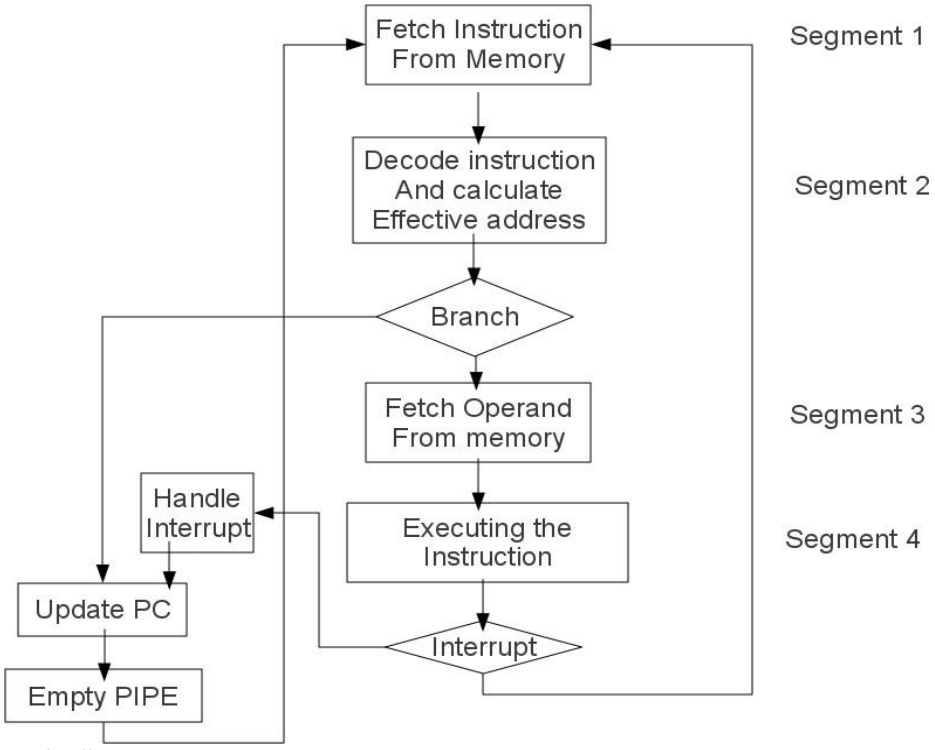
circuit and when addition is going on, the new data values will be brought into the registers in the start of the implementation. We can see how the addition is being performed in the diagram.

Instruction Pipeline: Pipelining concept is not only limited to the data stream, but can also be applied on the instruction stream. The instruction pipeline execution will be like the queue execution. In the queue the data that is entered first, will be the data first retrieved. Therefore when an instruction is first coming, the instruction will be placed in the queue and will be executed in the system. Finally the results will be passing on to the next instruction in the queue. This scenario is called as Instruction pipelining. The instruction cycle is given below

- Fetch the instruction from the memory
- Decode the instruction
- calculate the effective address
- Fetch the operands from the memory
- Execute the instruction
- Store the result in the proper place.

In a computer system each and every instruction need not necessary to execute all the above phases. In a Register addressing mode, there is no need of the effective address calculation. Below is the example of the four

Instruction pipelining



segment instruction pipeline.

In the above diagram we can see that the instruction which is first executing has to be fetched from the memory, there after we are decoding the instruction and we are calculating the effective address. Now we have two ways to execute the instruction. Suppose we are using a normal instruction like ADD, then the operands for that instruction will be fetched and the instruction will be executed. Suppose we are executing an instruction such as Fetch command. The fetch command itself has internally three more commands which are like ACTDR, ARTDR etc.., therefore we have to jump to that particular location to execute the command, so we are using the branch operation. So in a branch operation, again other instructions will be executed. That means we will be updating the PC value such that the instruction can be executed. Suppose we are fetching the operands to perform the original operation such as ADD, we need to fetch the data. The data can be fetched in two ways, either from the main memory or else from an input output devices. Therefore in order to use the input output devices, the

devices must generate the interrupts which should be handled by the CPU. Therefore the handling of interrupts is also a kind of program execution. Therefore we again have to start from the starting of the program and execute the interrupt cycle.

The different instruction cycles are given below:

- FI → FI is a segment that fetches an instruction
- DA → DA is a segment that decodes the instruction and identifies the effective address.
- FO → FO is a segment that fetches the operand.
- EX → EX is a segment that executes the instruction with the operand.

Timing of Instruction Pipeline

FI → Fetch Instruction
FO → Fetch Operand
DA → Decode instruction and Fetch Effective Address
EX → Execute the Instruction

Step	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Pipelining Conflicts: There are different conflicts that are caused by using the pipeline concept. They are

- Resource Conflicts: These are caused by access to memory by two or more segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories
- Data Dependency: These conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- Branch difficulties: These difficulties arise from branch and other instructions that change the value of PC.

Data Dependency Conflict: The data dependency conflict can be solved by using the following methods.

- Hardware Interlocks: The most straight forward method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destination of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delay.
- Operand Forwarding: Another technique called operand forwarding uses special hardware to detect a conflict and avoid the conflict path by using a special path to forward the values between the pipeline segments.
- Delayed Load: The delayed load operation is nothing but when executing an instruction in the pipeline, simply delay the execution starting of the instruction such that all the data that is needed for the instruction can be successfully updated before execution.

Branch Conflicts:

The following are the solutions for solving the branch conflicts that are obtained in the pipelining concept.

- Pre-fetch Target Instruction: In this the branch instructions which are to be executed are pre-fetched to detect if any errors are present in the branch before execution.
- Branch Target Buffer: BTB is the associative memory implementation of the branch conditions.
- Loop buffer: The loop buffer is a very high speed memory device. Whenever a loop is to be executed in the computer. The complete loop will be transferred in to the loop buffer memory and will be executed as in the cache memory.
- Branch Prediction: The use of branch prediction is such that, before a branch is to be executed, the instructions along with the error checking conditions are checked. Therefore we will not be going into any unnecessary branch loops.
- Delayed Branch: The delayed branch concept is same as the delayed load process in which we are delaying the execution of a branch process, before all the data is fetched by the system for beginning the CPU.

RISC Pipeline:

The ability to use the instruction pipelining concept in the RISC architecture is very efficient. The simplicity of the instruction set can be utilized to implement an instruction pipeline using a small number of sub operations, with each being executed in one clock cycle. Due to fixed length instruction format, the decoding of the operation can occur at the same time as the register selection. Since the arithmetic, logic and shift operations are done on register basis, there is no need for extra fetching or effective address decoding steps to perform the operation. So pipelining concept can be effectively used in this scenario. Therefore the total operations can be categorized as one segment will be fetching the instruction from program memory, the other segment executes the instruction in the ALU and the third segment may be used to store the result of the ALU operation in a destination register. The data transfer instructions in RISC are limited to only Load and Store instructions. To prevent conflicts in data transfer, we will be using two separate buses one for storing the instructions and other for storing the data.

Example of three segment instruction pipeline:

We want to perform a operation in which there is some arithmetic, logic or shift operations. Therefore as per the instruction cycle, we will be having the following steps:

- I: Instruction Fetch
- A: ALU Operation
- E: Execute Instruction.

The I segment will be fetching the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. In the A segment the ALU operation instruction will be fetched and the effective address will be retrieved and finally in the E segment the instruction will be executed.

Delayed Load:

Consider the following instructions:

1. LOAD: R1 ← M[address 1]
2. LOAD: R2 ← M[address 2]
3. ADD: R3 ← R1 + R2
4. STORE: M[address 3] ← R3

The below tables will be showing the pipelining concept with the data conflict and without data conflict.

Pipeline timing with data conflict

Clock Cycles	1	2	3	4	5	6
1.Load R1	I	A	E			
2.Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

Pipeline timing with delayed load

Clock Cycles	1	2	3	4	5	6	7
1.Load R1	I	A	E				
2.Load R2		I	A	E			
3. No Operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

Vector Processing:

Normal computational systems are not enough in some special processing requirements. Such as, in special processing systems like artificial intelligence systems and some weather forecasting systems, terrain analysis, the normal systems are not sufficient. In such systems the data processing will be involving on very high amount of data, we can classify the large data as a very big array. Now if we want to process this data, naturally we will need new methods of data processing. The vectors are considered as the large one-dimensional array of data. The term vector processing involves the data processing on the vectors of such large data.

The vector processing system can be understand by the example below.

Consider a program which is adding two arrays A and B of length 100;

Machine level program

```
        Initialize I=0
20      Read A(I)
        Read B(I)
        Store C(I)=A(I)+B(I)
        Increment I=I+1
        If I<=100 go to 20
        continue
```

so in this above program we can see that the two arrays are being added in a loop format. First we are starting from the value of 0 and then we are continuing the loop with the addition operation until the I value has reached to 100. In the above program there are 5 loop statements which will be executing 100 times. Therefore the total cycles of the CPU taken is 500 cycles. But if we use the concept of vector processing then we can reduce the unnecessary fetch cycles, since the fetch cycles are used in the creation of the vector. The same program written in the vector processing statement is given below.

C(1:100)=A(1:100)+B(1:100)

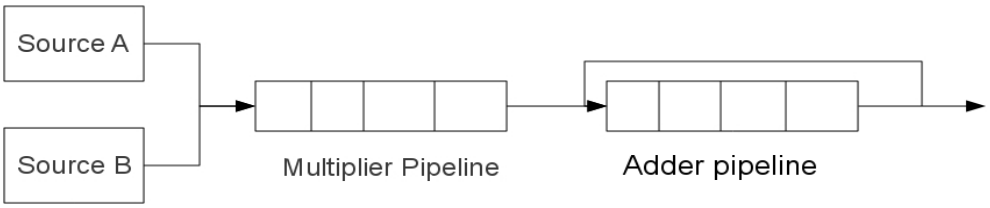
In the above statement, when the system is creating a vector like these the original source values are fetched from the memory into the vector, therefore the data is readily available in the vector. So when a operation is initiated on the data, naturally the operation will be performed directly on the data and will not wait for the fetch

Operation Code	Base Address SRC 1	Base Address SRC 2	Base Address DST	Vector length

cycle. So the total no of CPU Cycles taken by the above instruction is only 100.

Instruction format of Vector Instruction

Below we can see the implementation of the vector processing concept on the following matrix multiplication. In the matrix multiplication, we will be multiplying the row of A matrix with the column of the B matrix elements

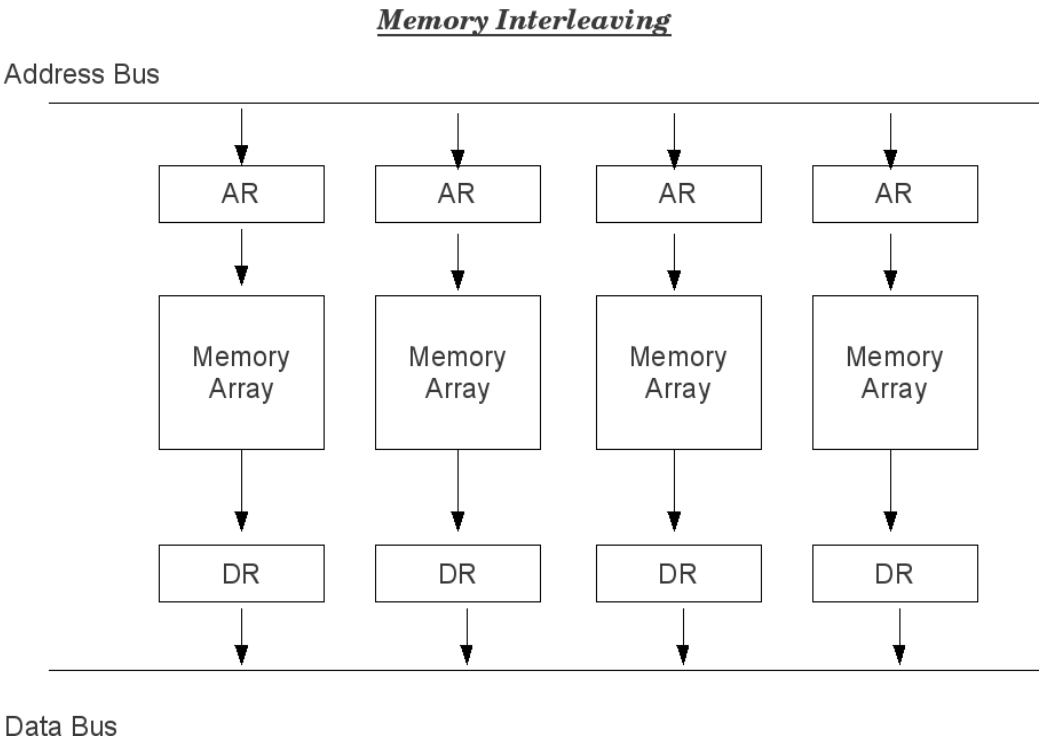


individually finally we will be adding the results.

In the above diagram we can see that how the values of A vector and B Vector which represents the matrix are being multiplied. Here we will be considering a 4x4 matrix A and B. Now the from the source A vector we will be taking the first 4 values and will be sending to the multiplier pipeline along with the 4 values from the vector B. The resultant 1 value is stored in the adder pipeline. Like wise remaining values from a row and column multiplication will be brought into the adder pipeline, which will be performing the addition of all the things finally

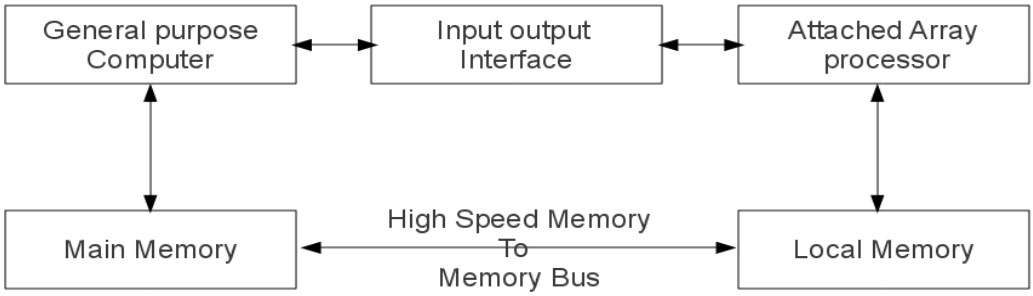
we will have the result of one row to column multiplication. When addition operation is taking place in the adder pipeline the next set of values will be brought into the multiplier pipeline, so that all the operations can be performed simultaneously using the parallel processing concepts by the implementation of pipeline.

Memory Interleaving:



Pipelining and vector processing naturally requires the several data elements for processing. So instead of using the same memory and selecting one at a time, we will be using several modules of the memory such that we can have separate data for each processing unit. As we can see in the above in the diagram each memory array is designed independently of the next memory array. Such that when the data needed for a operation is stored in the first memory array, another data for another operation can be safely stored in the next memory array, so that the operations can be performed concurrently. This process is called as memory interleaving.

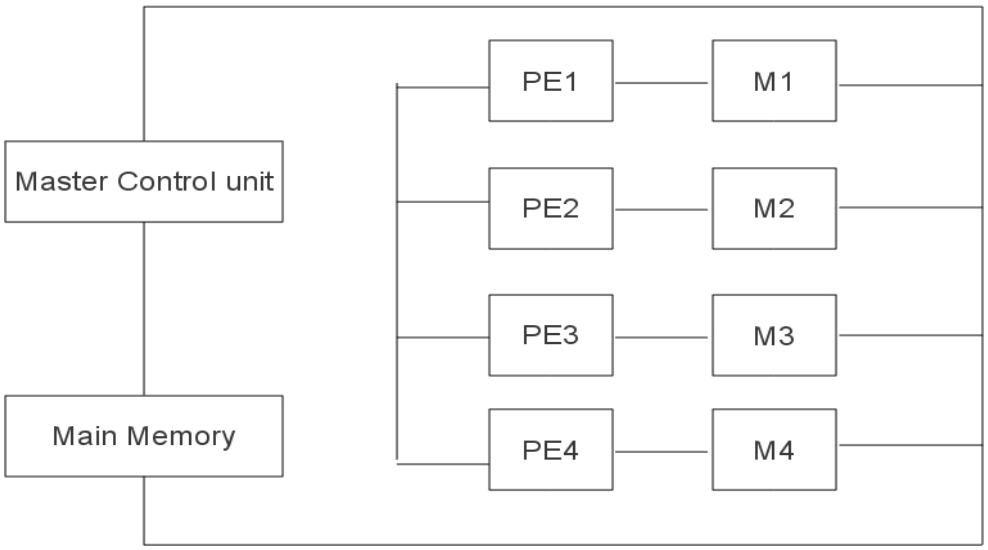
Array Processors: In a distributed computing we will be having several computers working on the same task such that their processing power will be shared among all the systems so that they can perform the task fast. But the disadvantage of the distributed computing is that we have to give separate resources for each system and every system need to be controlled by a task initiating system or can be called as a central control unit. The management of this kind of systems is very hard. In order to perform a specific operation involving a large processing there is no need of distributed computing. The alternate for this kind of scenarios is array processors



or attached array processors. The simplest is the SIMD Attached array processor.

Attached Array processor

The above diagram shows that the system is attached a separate processor which will be used for operation specific purpose. If the array processor is designed for solving floating point arithmetic, then it will only perform that operations. The detailed figure of the attached array processor is given in the diagram below. This will be having the SIMD architecture. In this we will be having a master control unit which will be coordinating all the process in the array processor. Each processing unit in the array processor is having a local memory unit as in the memory interleaving concept on which it performs the operations. Finally we will be having a main memory in which the original source data and the results that are obtained from the array processor will be stored. This is the working principle of the SIMD array processor technology.



SIMD Array Processor Technology