COA

Unit - 3

Fundamental Concept of Basic processing Unit :-

The processor fetch one instruction at a time and performs the operation specified.
The processor keep track of address of the memory location containing the next instruction to be ~~used~~ fetched using Program Counter.

* Instruction Register :-

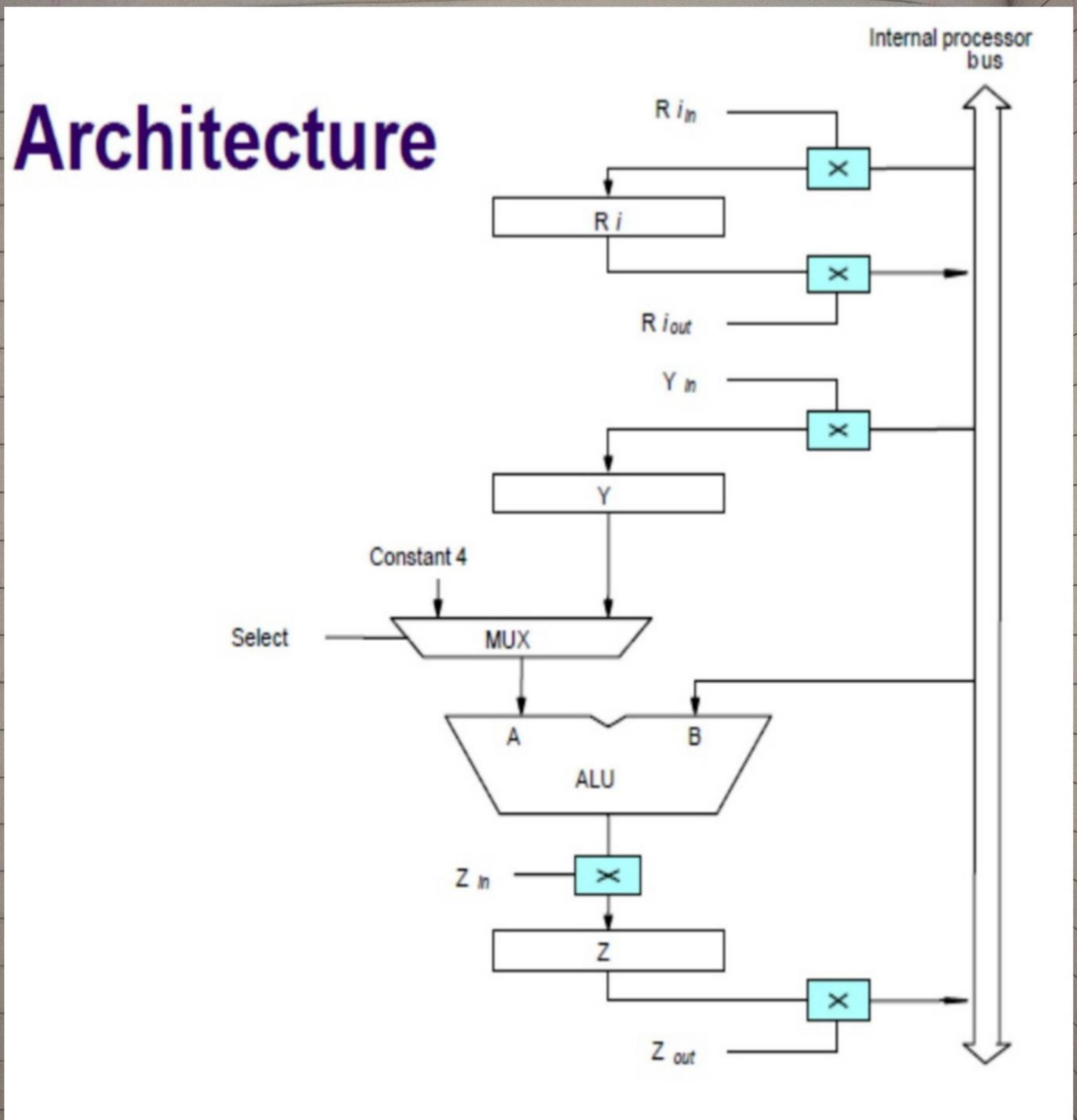
Executing the instruction and fetch the content of memory location pointed to by the PC.
The content of this location is ~~to~~ loaded into the IR.

$$IR \leftarrow [[PC]]$$

* Executing an Instruction :-

→ Transfer data from one processor register to another or to the ALU.
→ Perform arithmetic or logic operation and store result in ~~progra~~ processor register.
→ Fetch content of given memory location and load them into processor register.
→ Store data from processor register into a given memory location.

Architecture of Basic Processing Unit :-



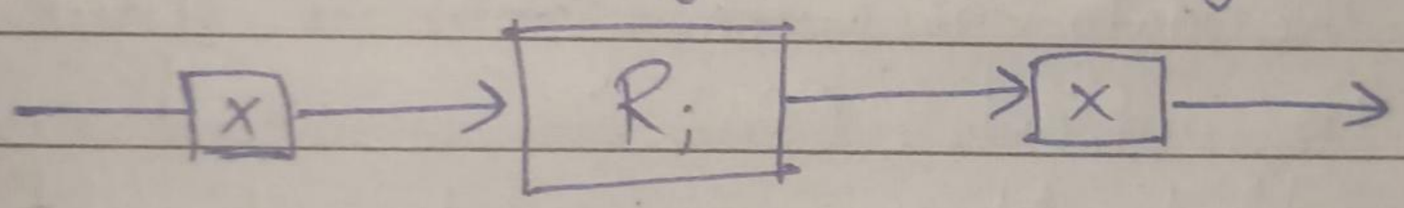
* All operation & data transfer are controlled by processor clock.

Performing ALU operations :-

* ALU is a combinational circuit that has no internal storage.
* ALU gets its two operands from MUX and bus. Result is stored temporily in Z.

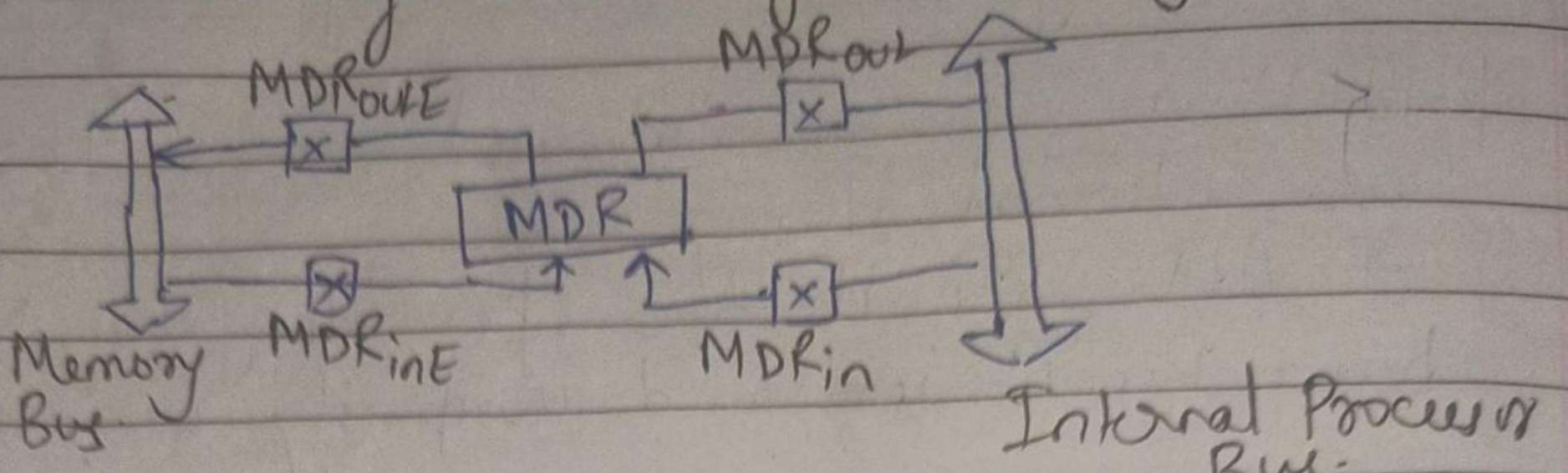
Input & output gating for the registers :-

Certain signals are used to control the input & output mechanism of various registers like



To store data in register R_i, R_{in} signal must be set; similarly to get output from register R_i, R_{out} must be set.

Fetching a word from Memory :-



As we know there is huge difference between the speede of CPU and a peripheral device. CPU has to wait until the data received from a slow device. To indicate CPU, a special control signal called Memory Function Completed (MFC) is used which tells to the CPU that data is available. ~~for instruction~~.

For instruction MOV R1,R2, needed action are :-

1. MAR ← [R1]
2. Start read operation on memory bus
3. Wait for MFC response from memory.
4. Load MDR from memory bus.
5. $R_2 \leftarrow [MDR]$.

Signals activated :-
1. $R1_{out}$, MAR_{in}, Read
2. MDR_{inE}, WMFC
3. MDR_{out}, $R2_{in}$.

A Complete Execution of Instruction :-

ADD R3, R1

1. PC_{out} , MAR_{in} , Read; Select 4, Add, Z_{in}.
2. Z_{out} , PC_{in} , Y_{in} , WMFC
3. MDR_{out} , IR_{in}
4. $R3_{out}$, MAR_{in} , Read
5. $R1_{out}$, Y_{in} , WMFC
6. MDR_{out} , Select Y, Add , Z_{in}
7. Z_{out} , $R1_{in}$. End.

The instruction ADD (R3), R1 denotes the addition of content of register R1 with the content of the memory whose address is stored in register R3.

The very first step in execution of any instruction is INSTRUCTION FETCH (IF). In IF, the instruction which is to be executed is transfered from memory into INSTRUCTION REGISTER (IR). In IR the meaning of instruction is found then required data is fetched from memory then the instruction is applied on data and result is stored in specific location

In the first three steps of the sequence Fetching of instruction is done. initially the address of current instruction is in PC. then transferred to MAR and read signal is issued. then the current value of PC is updated (by 4) to point to the next instruction/data. After that the data whose address is given in R3 is fetched from memory. the data in R1 is transferred into Y register. than addition is taken place. and result is stored in R1.

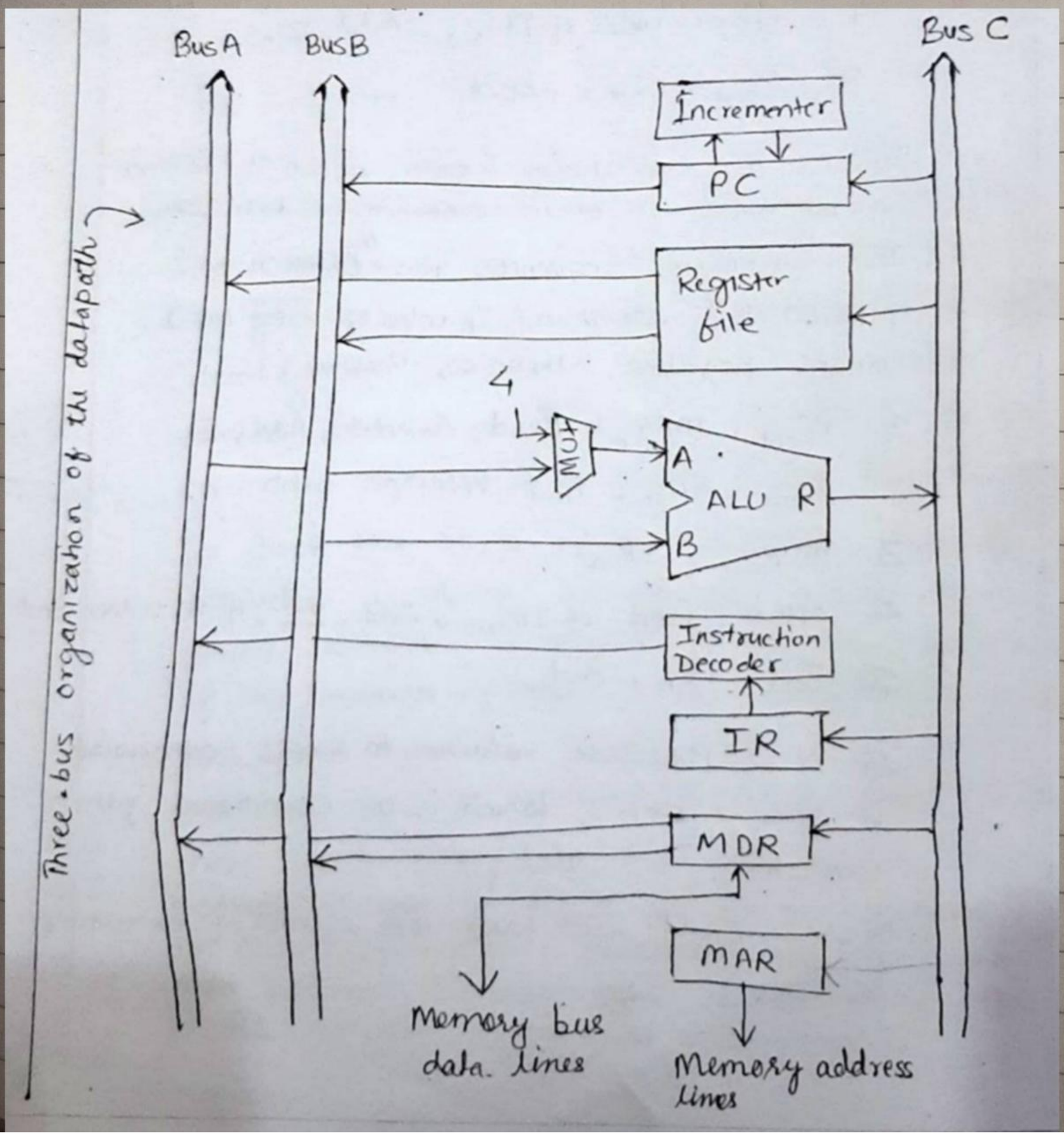
Branch Instruction :-

A branch instruction replace the contents of PC with the branch target address, which is usually obtained by adding an offset X given in branch instruction.

The offset X is usually the difference between branch target address and the address immediately following branch instruction.

Multiple Bus Organisation

Buses are used to carry data among various components of a computer system. If a single bus is available then only a single data can be carried away within one cycle. If more than one bus are available then more than one data can be transfered within one cycle. thus, speed of instruction execution will be increased.



Differences between Single Bus and Double Bus Structure :

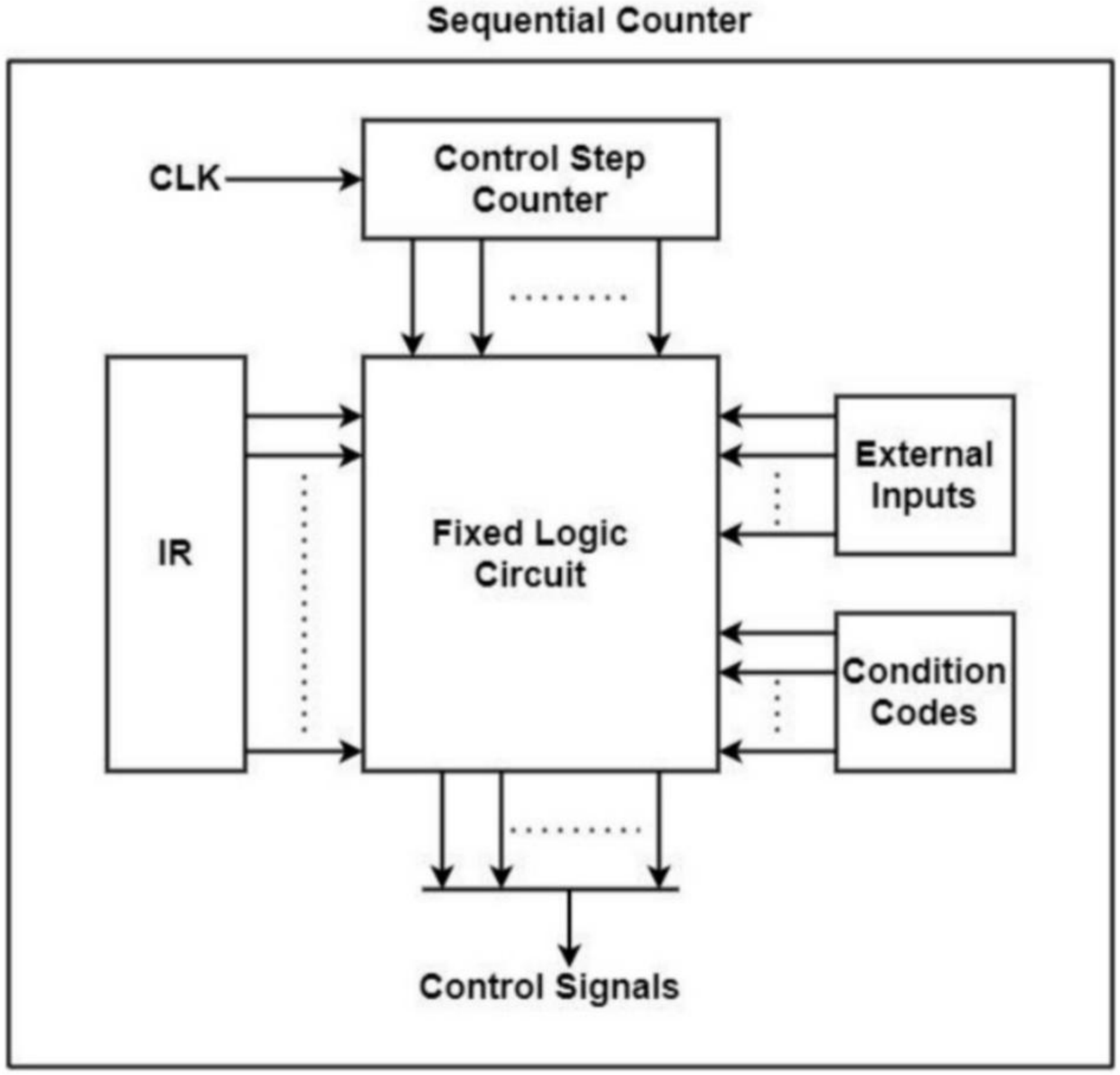
Single Bus Structure	Double Bus Structure
One common bus is used for communication between peripherals and processor.	Two buses are used, one for communication from peripherals and other for processor.

Single Bus Structure	Double Bus Structure
Instructions and data both are transferred in same bus.	Instructions and data both are transferred in different buses.
Its performance is low.	Its performance is high.
Cost of single bus structure is low.	Cost of double bus structure is high.
Number of cycles for execution is more.	Number of cycles for execution is less.
Execution of process is slow.	Execution of process is fast.
Number of registers associated are less.	Number of registers associated are more.
At a time single operand can be read from bus.	At a time two operands can be read.

Hardwired Control :-

Hardwired Control is an approach to generate control signal in a proper sequence in order to execute an instruction. Since the sequence of operation are to carried out is determined by the wiring of logic elements, it is called Hardwired control.

It can operate at high speed but less flexible and cannot handle complex instruction set. To avoid those problems MICROPROGRAMMED CONTROL is used.



A hardwired control consists of two decoders a sequence counter and number of logic gates. An instruction fetched from memory unit is placed in the instruction register (IR).

Instruction register :-
↳ Consist of 1 bits, the operation code, and bit 0. ~~through~~

* Hardwired control produces signals using Finite State Machines (FSM).

The major goal of implementing Hardwired control is to minimize cost of circuit and to achieve grater efficiency in operation speed.

Some of the method that come up for designing the hardwired control logic are as follows :-

Sequence Counter Method :- Most efficient use of clocked delay elements.

Delay Element Method :- depend on

State Table Method :- Involves traditional algorithmic approach.

Microprogrammed Control Unit

A control unit whose binary control values are saved as words in memory is called a microprogrammed control unit.

In this approach each instruction has a MICROROUTINE, this microroutine is further divided into multiple Micro-Instructions (or Control Words).

Control sequence of instruction ADD (R3), R1

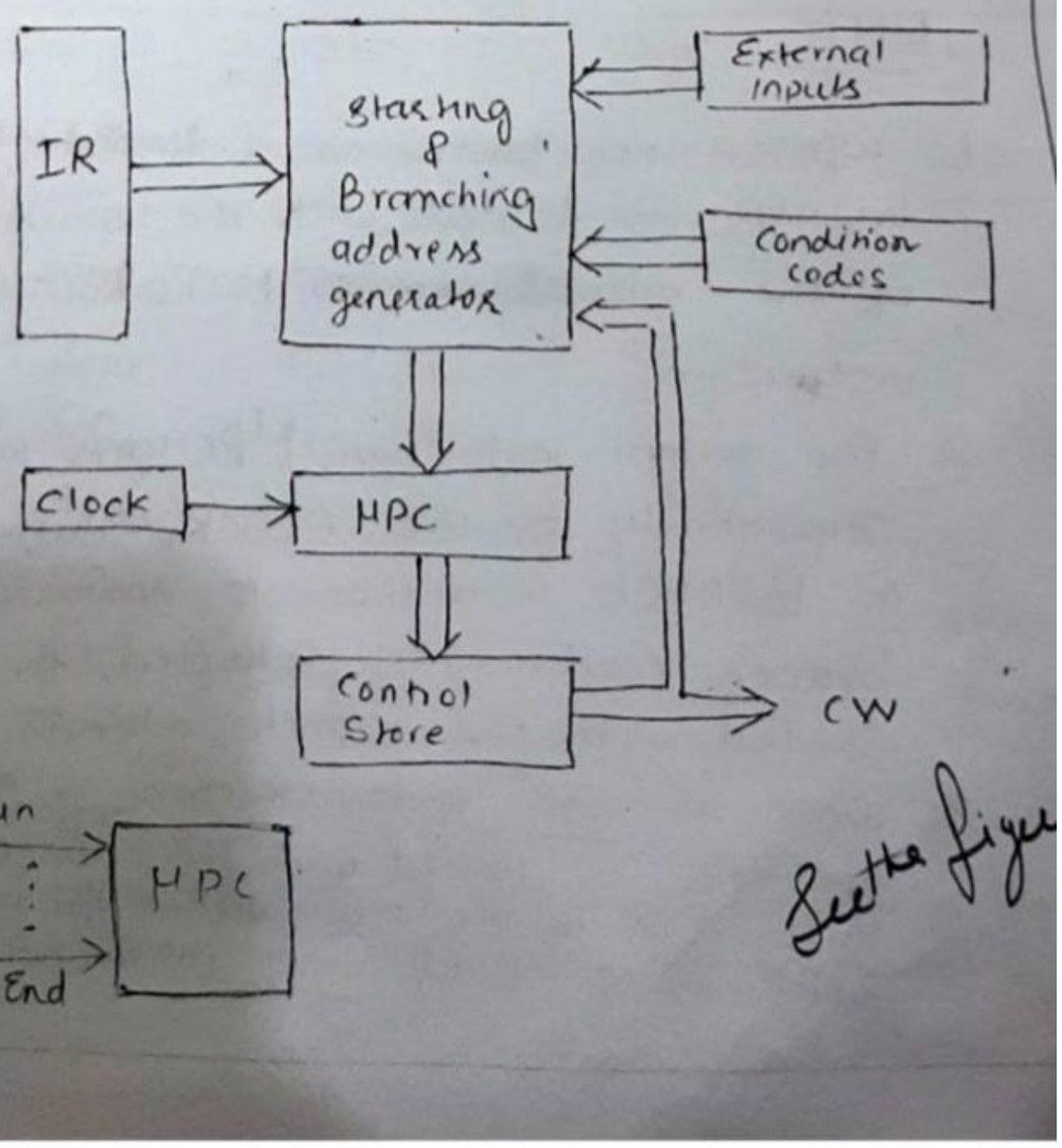
1. PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2. Z_{out}, PC_{in}, Y_{in} WMFC
3. MDR_{out}, IR_{in}
4. $R3_{out}$, MAR_{in}, Read
5. $R1_{out}$, Y_{in}, WMFC
6. MDR_{out}, SelectY, Add, Z_{in}
7. Z_{out} $R1_{in}$, End.

Example of micro-Instruction of above sequence is given below :-

Micro instruction	PCout	PCin	Read	Select4	Add	Zin	Zout	Yin	WMFC	MDRout	IRin	R3out	MARin	R1out	Select Y	R1in	End
1	1	0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0
2	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
5	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0
6	0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0
7	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1

* Note :- It is also an example of HORIZONTAL ORGANIZATION.

ORGANIZATION OF A CONTROL UNIT WITH CONDITIONAL BRANCHING



See the figu

Terminology

CONTROL STORE - In it, all the microroutines for all instructions in the instruction set are stored. Control unit generates the microinstructions (control words) for the corresponding microroutine by reading control store, sequentially (or not, in case of BRANCH INSTRUCTIONS).

MICRO PROGRAM COUNTER (μPC) -
To read microinstructions/CW sequentially, MPC is used. Every time a new instruction loaded into IR, the output of the "Starting address generator" is loaded into the MPC, after that MPC gets incremented automatically by the clock.

Difference between Hardwired Control Unit & Microprogrammed Control Unit

Hardwired Control Unit	Microprogrammed Control Unit
Hardwired control unit generates the control signals needed for the processor using logic circuits	Micrprogrammed control unit generates the control signals with the help of micro instructions stored in control memory
Hardwired control unit is faster when compared to microprogrammed control unit as the required control signals are generated with the help of hardwares	This is slower than the other as micro instructions are used for generating signals here
Difficult to modify as the control signals that need to be generated are hard wired	Easy to modify as the modification need to be done only at the instruction level
More costlier as everything has to be realized in terms of logic gates	Less costlier than hardwired control as only micro instructions are used for generating control signals
It cannot handle complex instructions as the circuit design for it becomes complex	It can handle complex instructions
Only limited number of instructions are used due to the hardware implementation	Control signals for many instructions can be generated
Used in computer that makes use of Reduced Instruction Set Computers(RISC)	Used in computer that makes use of Complex Instruction Set Computers(CISC)

Working of Microprogrammed Control Unit :-

1. When a new instruction is loaded into IR, the PPC is loaded with the starting address of MICROROUTINE for that instruction.

2. For normal instruction PPC gets incremented sequentially by the CLOCK but when a BRANCH instruction is encountered, and the branch condition is satisfied the PPC is loaded with branch address.

3. When the "End" microinstruction is encountered, the PPC is loaded with the address of first CW in the microroutine for the instruction fetch cycle of next instruction.

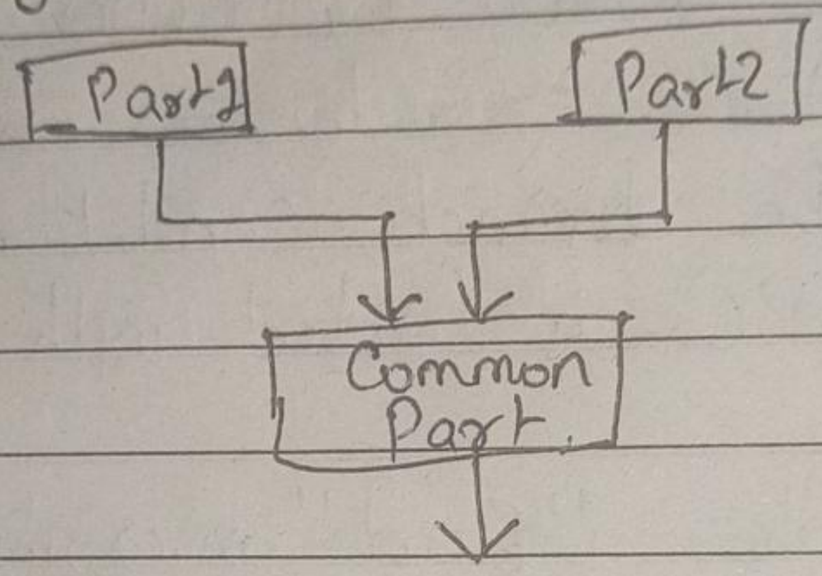
Microprogram Sequencing

Some branching capabilities within a microprogram can be introduced through special branch microinstructions that specify the branch address. This technique is simple but has two disadvantage :-

(a) Having a separate microroutine for each instruction results in large total number of microinstruction & large control store.

(b) Generally, a machine has several addressing mode so the combination of these would result in duplication.

The solution is one can provide switching among various common parts of microroutine.

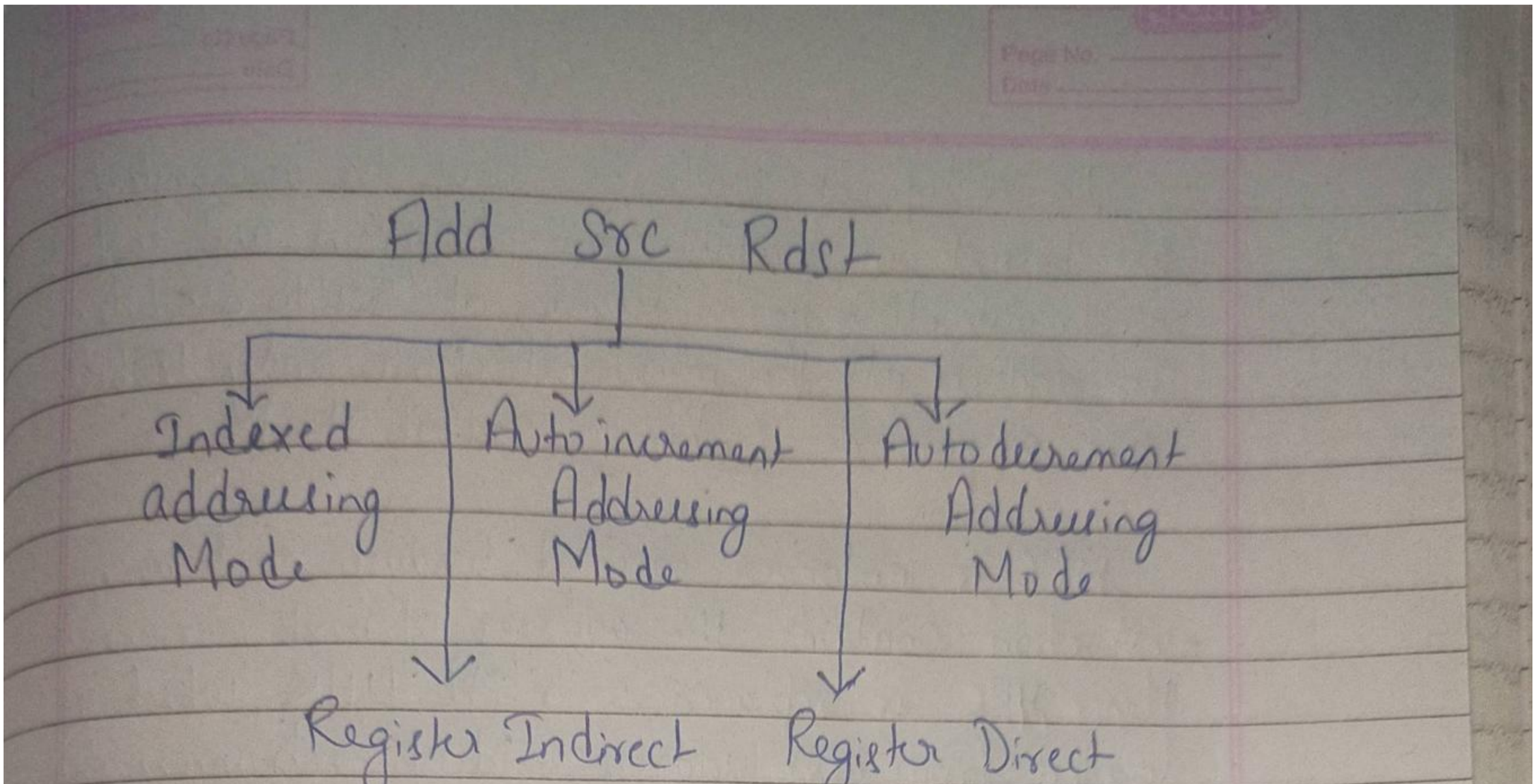


Wide Branch Addressing :-

When two or more options are available to calculate the effective address of an operand it is called wide-branch addressing.

For example, In instruction Add Src, Rdst, the effective address of "Src" can be calculated using indexed autoincrement, autodecrement, register direct, register indirect addressing mode.

The Add Scr, Rdst instruction can not be fully executed until the effective address of "Src" is calculated using one of addressing mode.



This type of addressing is known as Wide - ~~Addressing~~ Branch Addressing.

Microinstructions With Next-Address Field :-

In case of wide-branch addressing the number of branch microinstruction become too large, and these microinstruction performs no useful operation in the datapath, they are only needed to determine the address of next microinstruction.

This problem becomes worse when ~~one~~ other micro routines are considered. Another problem is sequential address are not available for the microinstruction that are generated executed in consecutive order.

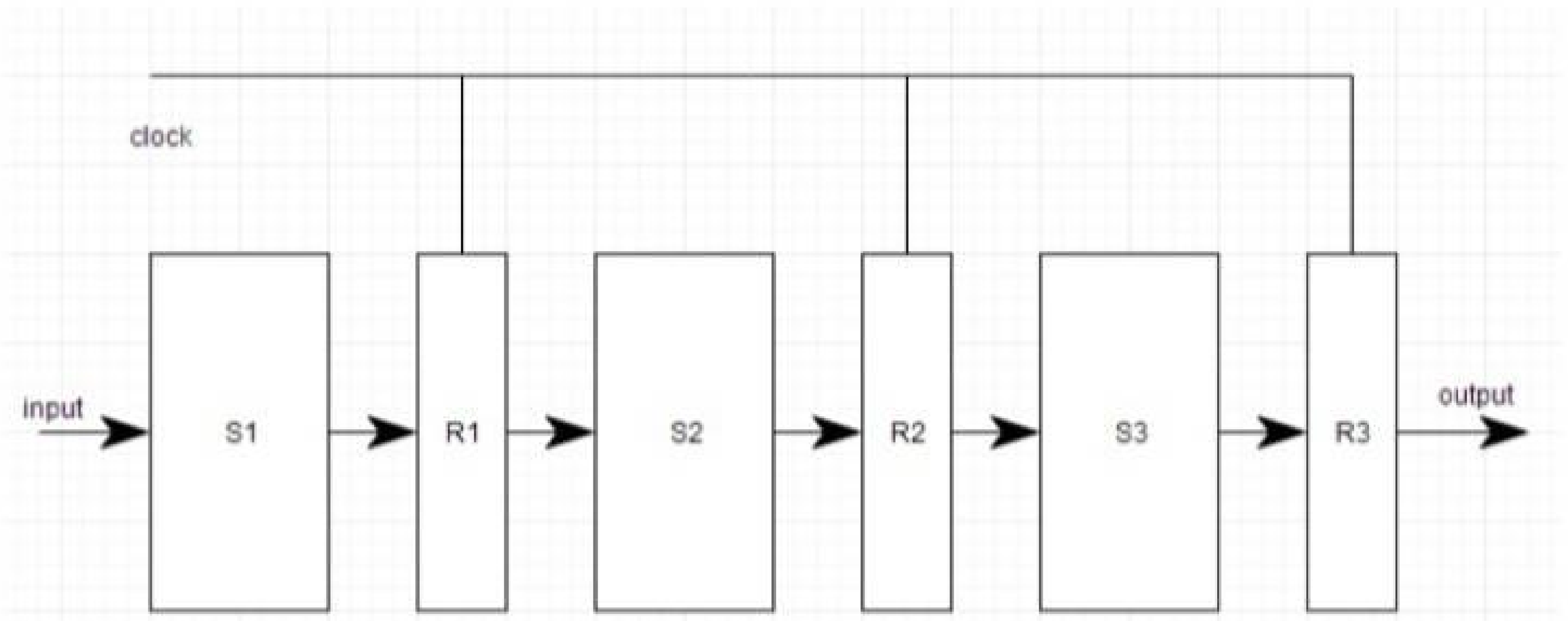
An alternate solution of this problem is to include a next address field in each microinstruction. Now each microinstruction become a branch microinstruction.

In a typical computer about 4k microinstruction are present in complete microprogram and each microinstruction has a lenght of 50 to 80 bits. To address all the microinstruction only 12 bits are required (1/6 of Control Store).

MPC is nolonger needed because each micro-instruction Contains the address of next microinstruction. Hence, MPC is replaced with MAR which is loaded from next address field of each microinstruction.

What is Pipelining?

- Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as pipeline processing.
- Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.
- Pipelining increases the overall instruction throughput.
- In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



- Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline

It is divided into 2 categories:

- Arithmetic Pipeline
- Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:

Here A and B are mantissas (significant digit of floating point numbers), while a and b are exponents.

The floating point addition and subtraction is done in 4 parts:

- Compare the exponents.
- Align the mantissas.
- Add or subtract mantissas
- Produce the result.
- Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

In this a stream of instructions can be executed by overlapping *fetch*, *decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Pipeline Conflicts

There are some factors that cause the pipeline to deviate its normal performance. Some of these factors are given below:

1. Timing Variations

All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

2. Data Hazards

When several instructions are in partial execution, and if they reference same data then the problem arises. We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

3. Branching

In order to fetch and execute the next instruction, we must know what that instruction is. If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

4. Interrupts

Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

5. Data Dependency

It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Advantages of Pipelining

- The cycle time of the processor is reduced.
- It increases the throughput of the system
- It makes the system reliable.

Disadvantages of Pipelining

- The design of pipelined processor is complex and costly to manufacture.
- The instruction latency is more.
- In instruction pipelining,
- A form of parallelism called as instruction level parallelism is implemented.
- Multiple instructions execute simultaneously.
- The efficiency of pipelined execution is more than that of non-pipelined execution.

Performance of Pipelined Execution-

The following parameters serve as criterion to estimate the performance of pipelined execution-

- Speed Up
- Efficiency
- Throughput

1. Speed Up-

It gives an idea of "**how much faster**" the pipelined execution is as compared to non-pipelined execution.

It is calculated as-

$$\text{Speed Up (S)} = \frac{\text{Non-pipelined execution time}}{\text{Pipelined execution time}}$$

2. Efficiency:

The efficiency of pipelined execution is calculated as-

$$\text{Efficiency } (\eta) = \frac{\text{Speed Up}}{\text{Number of stages in Pipelined Architecture}}$$

OR

$$\text{Efficiency } (\eta) = \frac{\text{Number of boxes utilized in phase-time diagram}}{\text{Total number of boxes in phase-time diagram}}$$

3. Throughput:

Throughput is defined as number of instructions executed per unit time.

It is calculated as-

$$\text{Throughput} = \frac{\text{Number of instructions executed}}{\text{Total time taken}}$$

There are three kinds of hazards:

- Structural Hazards
- Data Hazards
- Control Hazards

There are many specific solutions to dependencies. The simplest is introducing a ***bubble*** which stalls the pipeline and reduces the throughput. The bubble makes the next instruction wait until the earlier instruction is done with.

Structural Hazards

Structural hazards arise due to hardware resource conflict amongst the instructions in the pipeline. A resource here could be the Memory, a Register in GPR or ALU. This resource conflict is said to occur when more than one instruction in the pipe is requiring access to the same resource in the same clock cycle. This is a situation that the hardware cannot handle all possible combinations in an overlapped pipelined execution.

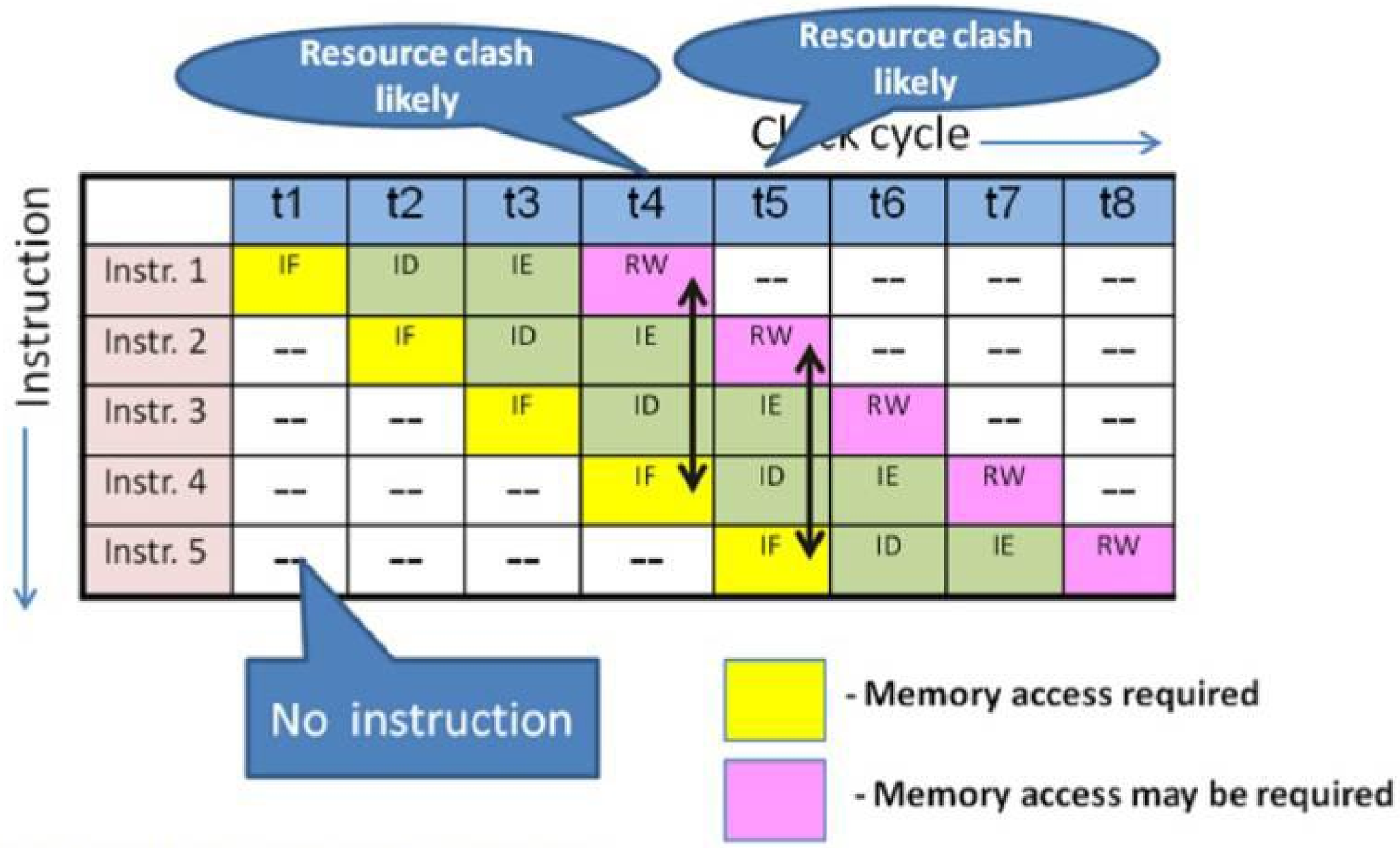


Figure 16.1 Structural dependency scenario

Observe the figure 16.1. In any system, instruction is fetched from memory in IF machine cycle. In our 4-stage pipeline Result Writing (RW) may access memory or one of the General Purpose Registers depending on the instruction. At t4, Instruction-1(I1) is at RW stage and Instruction-4(I4) at IF stage. Alas!. Both I1 and I4 are accessing the same resource i.e memory if I1 is a STORE instruction. How is it possible to access memory by 2 instructions from the same CPU in a timing state? Impossible. This is structural dependency. What is the solution?

Solution 1: Introduce bubble which stalls the pipeline as in figure 16.2. At t4, I4 is not allowed to proceed, rather delayed. It could have been allowed in t5, but again a clash with I2 RW. For

the same reason, I4 is not allowed in t6 too. Finally, I4 could be allowed to proceed (stalled) in the pipe only at t7.

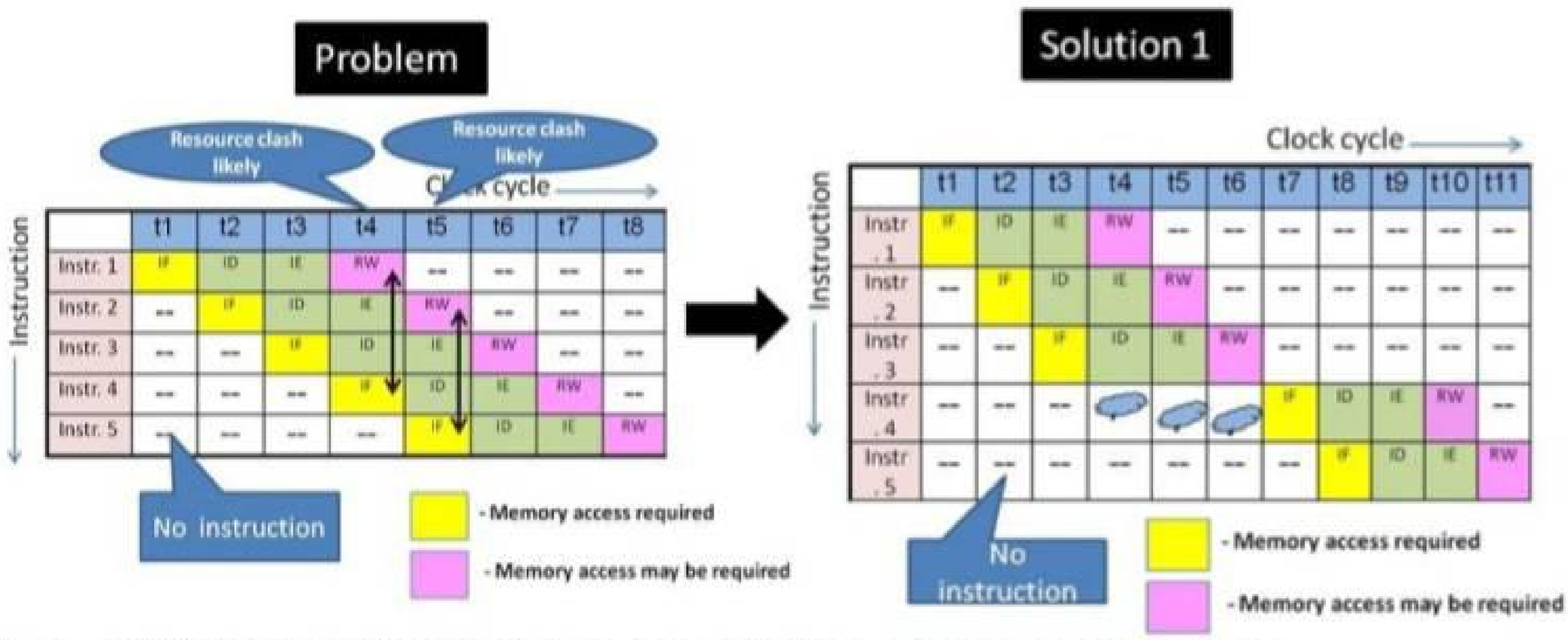


Figure 16.2 Structural Hazard solution using "stall" in a 4 stage pipeline design

This delay is percolated to all the subsequent instructions too. Thus, while the ideal 4-stage system would have taken 8 timing states to execute 5 instructions, now due to structural dependency it has taken 11 timing states. Just not this. By now you would have guessed that this hazard is likely to happen at every 4th instruction. Not at all a good solution for a heavy load on CPU. Is there a better way? Yes!

A better solution would be to increase the structural resources in the system using one of the few choices below:

The pipeline may be **increased** to 5 or more stages and suitably redefine the functionality of the stages and adjust the clock frequency. This eliminates the issue of the hazard at every 4th instruction in the 4-stage pipeline

The memory may physically be separated as ***Instruction memory*** and ***Data Memory***. A Better choice would be to design as Cache memory in CPU, rather than dealing with Main memory. IF uses Instruction memory and Result writing uses Data Memory. These become two separate resources avoiding dependency.

It is possible to have **Multiple levels of Cache** in CPU too.

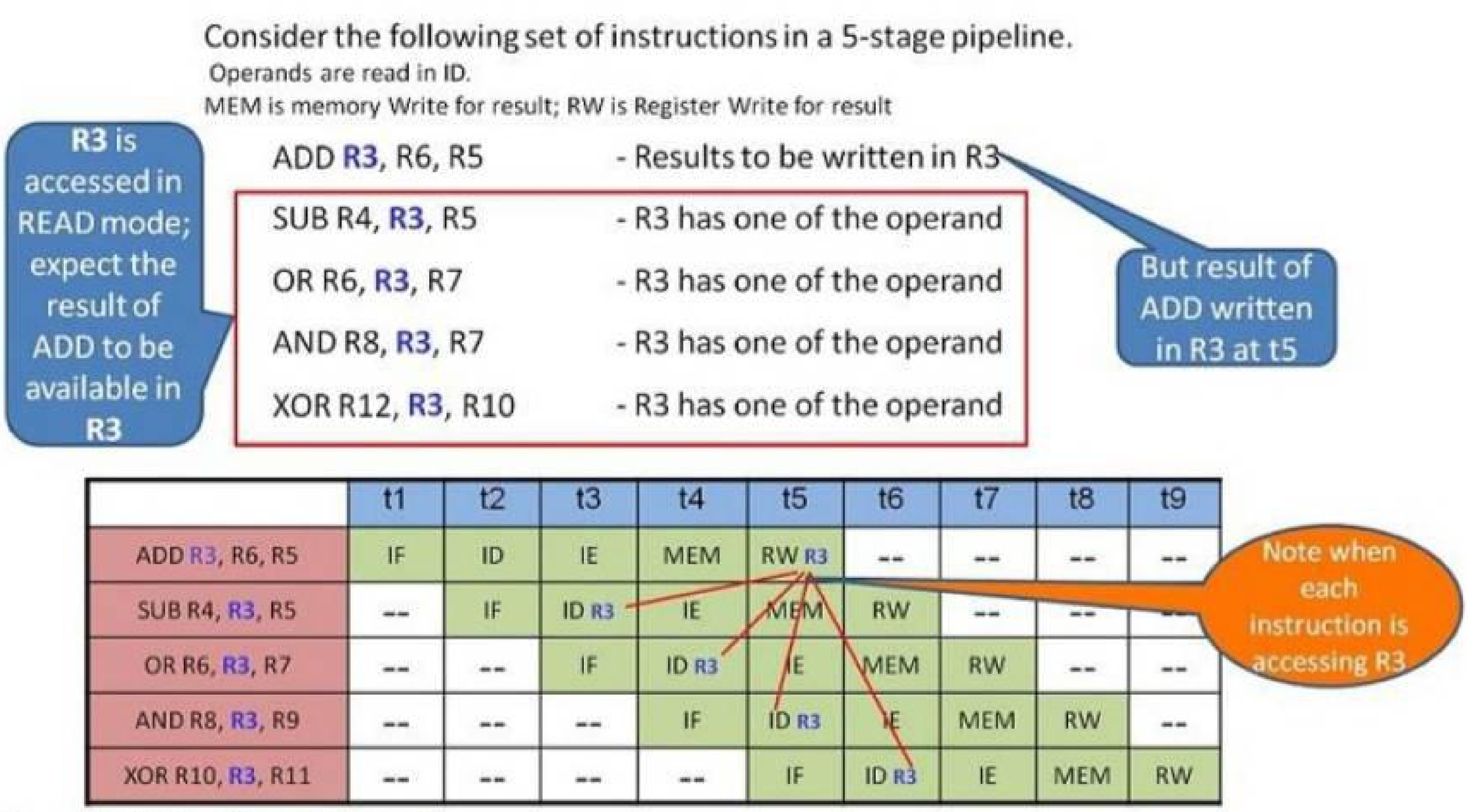
There is a possibility of ALU in resource dependency. ALU may be required in IE machine cycle by an instruction while another instruction may require ALU in IF stage to calculate Effective Address based on addressing mode. The solution would be either stalling or have an exclusive ALU for address calculation.

Register files are used in place of GPRs. Register files have multiport access with exclusive read and write ports. This enables simultaneous access on one write register and read register.

The last two methods are implemented in modern CPUs. Beyond these, if dependency arises, Stalling is the only option. Keep in mind that increasing resources involves increased cost. So the trade-off is a designer's choice.

Data Hazards

Data hazards occur when an instruction's execution depends on the results of some previous instruction that is still being processed in the pipeline. Consider the example below.



	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD R3, R6, R5	IF	ID	IE	MEM	RW R3	--	--	--	--
SUB R4, R3, R5	--	IF	ID R3	IE	MEM	RW	--	--	--
OR R6, R3, R7	--	--	IF	ID R3	IE	MEM	RW	--	--
AND R8, R3, R9	--	--	--	IF	ID R3	IE	MEM	RW	--
XOR R10, R3, R11	--	--	--	--	IF	ID R3	IE	MEM	RW

Figure 16.3 Data Hazard scenario

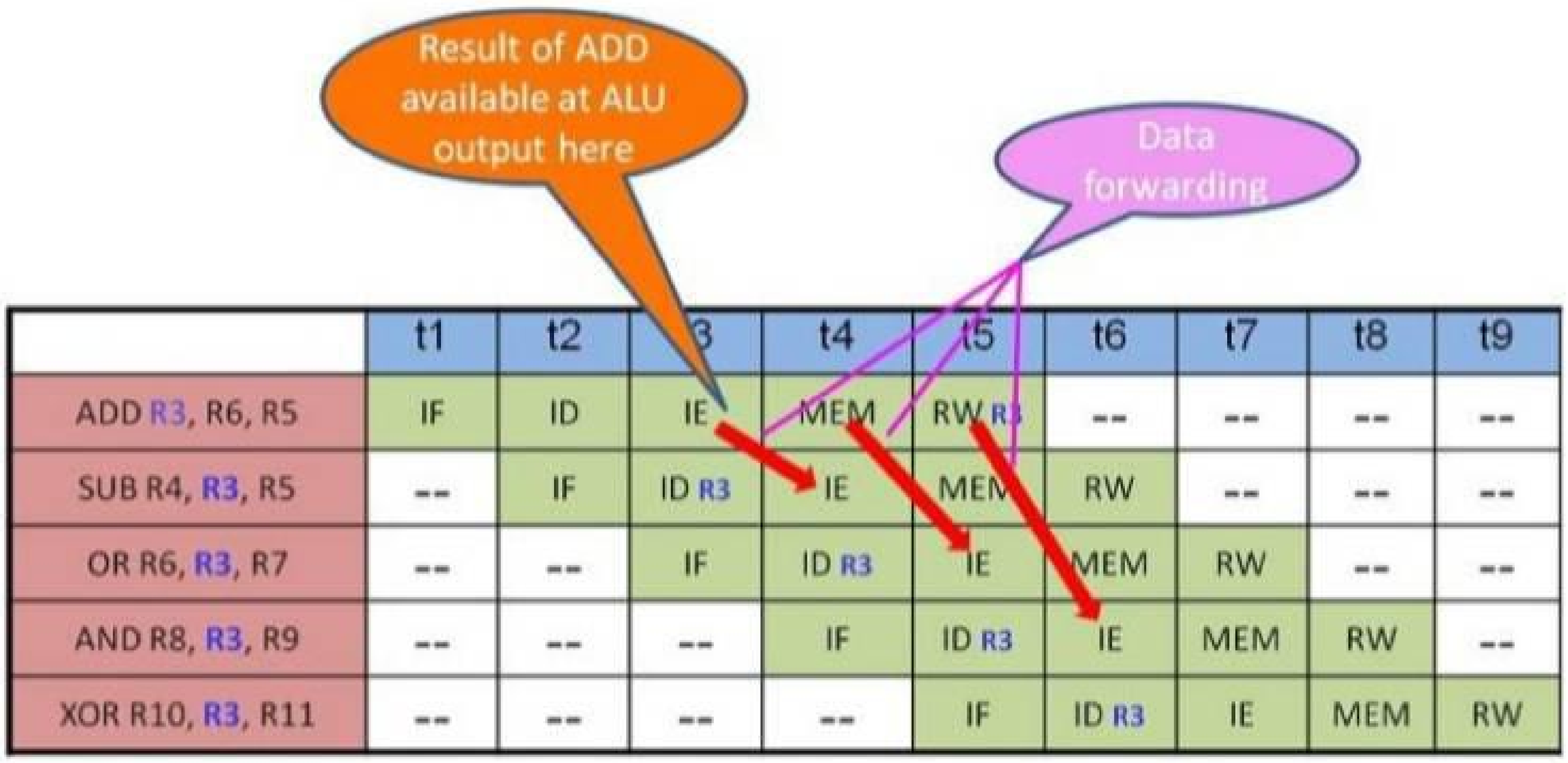
In the above case, ADD instruction writes the result into the register R3 in t5. If bubbles are not introduced to stall the next SUB instruction, all three instructions would be using the wrong data from R3, which is earlier to ADD result. The program goes wrong! The possible solutions before us are:

Solution 1: Introduce three bubbles at SUB instruction IF stage. This will facilitate SUB – ID to function at t6. Subsequently, all the following instructions are also delayed in the pipe.

Solution 2: Data forwarding - Forwarding is passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another. The purpose is to make available the solution early to the next instruction.

In this case, ADD result is available at the output of ALU in ADD –IE i.e t3 end. If this can be controlled and forwarded by the control unit to SUB-IE stage at t4, before writing on to output

register R3, then the pipeline will go ahead without any stalling. This requires extra logic to identify this data hazard and act upon it. It is to be noted that although normally Operand Fetch happens in the ID stage, it is used only in IE stage. Hence forwarding is given to IE stage as input. Similar forwarding can be done with OR and AND instruction too.



	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD R3, R6, R5	IF	ID	IE	MEM	RW R3	--	--	--	--
SUB R4, R3, R5	--	IF	ID R3	IE	MEM	RW	--	--	--
OR R6, R3, R7	--	--	IF	ID R3	IE	MEM	RW	--	--
AND R8, R3, R9	--	--	--	IF	ID R3	IE	MEM	RW	--
XOR R10, R3, R11	--	--	--	--	IF	ID R3	IE	MEM	RW

Figure 16.4 Data forwarding solution for Data Hazard

Solution 3: Compiler can play a role in detecting the data dependency and reorder (resequence) the instructions suitably while generating executable code. This way the hardware can be eased.

Solution 4: In the event, the above reordering is infeasible, the compiler may detect and introduce NOP (no operation) instruction(s). NOP is a dummy instruction equivalent bubble, introduced by the software.

The compiler looks into data dependencies in code optimisation stage of the compilation process.

Data Hazards classification

Data hazards are classified into three categories based on the order of READ or WRITE operation on the register and as follows:

RAW (Read after Write) [Flow/True data dependency]

This is a case where an instruction uses data produced by a previous one. Example

ADD R0, R1, R2

SUB R4, R3, R0

WAR (Write after Read) [Anti-Data dependency]

This is a case where the second instruction writes onto register before the first instruction reads. This is rare in a simple pipeline structure. However, in some machines with complex and special instructions case, WAR can happen.

ADD R2, R1, R0

SUB R0, R3, R4

WAW (Write after Write) [Output data dependency]

This is a case where two parallel instructions write the same register and must do it in the order in which they were issued.

ADD R0, R1, R2

SUB R0, R4, R5

WAW and WAR hazards can only occur when instructions are executed in parallel or out of order. These occur because the same register numbers have been allotted by the compiler although avoidable. This situation is fixed by renaming one of the registers by the compiler or by delaying the updating of a register until the appropriate value has been produced. Modern CPUs not only have incorporated Parallel execution with multiple ALUs but also Out of order issue and execution of instructions along with many stages of pipelines.

Control Hazards

Control hazards are called Branch hazards and caused by Branch Instructions. Branch instructions control the flow of program/ instructions execution. Recall that we use conditional statements in the higher-level language either for iterative loops or with conditions checking (correlate with for, while, if, case statements). These are transformed into one of the variants of BRANCH instructions. It is necessary to know the value of the condition being checked to get the program flow. Life is complicating you! So it is for the CPU!

Thus a Conditional hazard occurs when the decision to execute an instruction is based on the result of another instruction like a conditional branch, which checks the condition's resultant value.

The branch and jump instructions decide the program flow by loading the appropriate location in the Program Counter(PC). The PC has the value of the next instruction to be fetched and executed by CPU. Consider the following sequence of instructions.

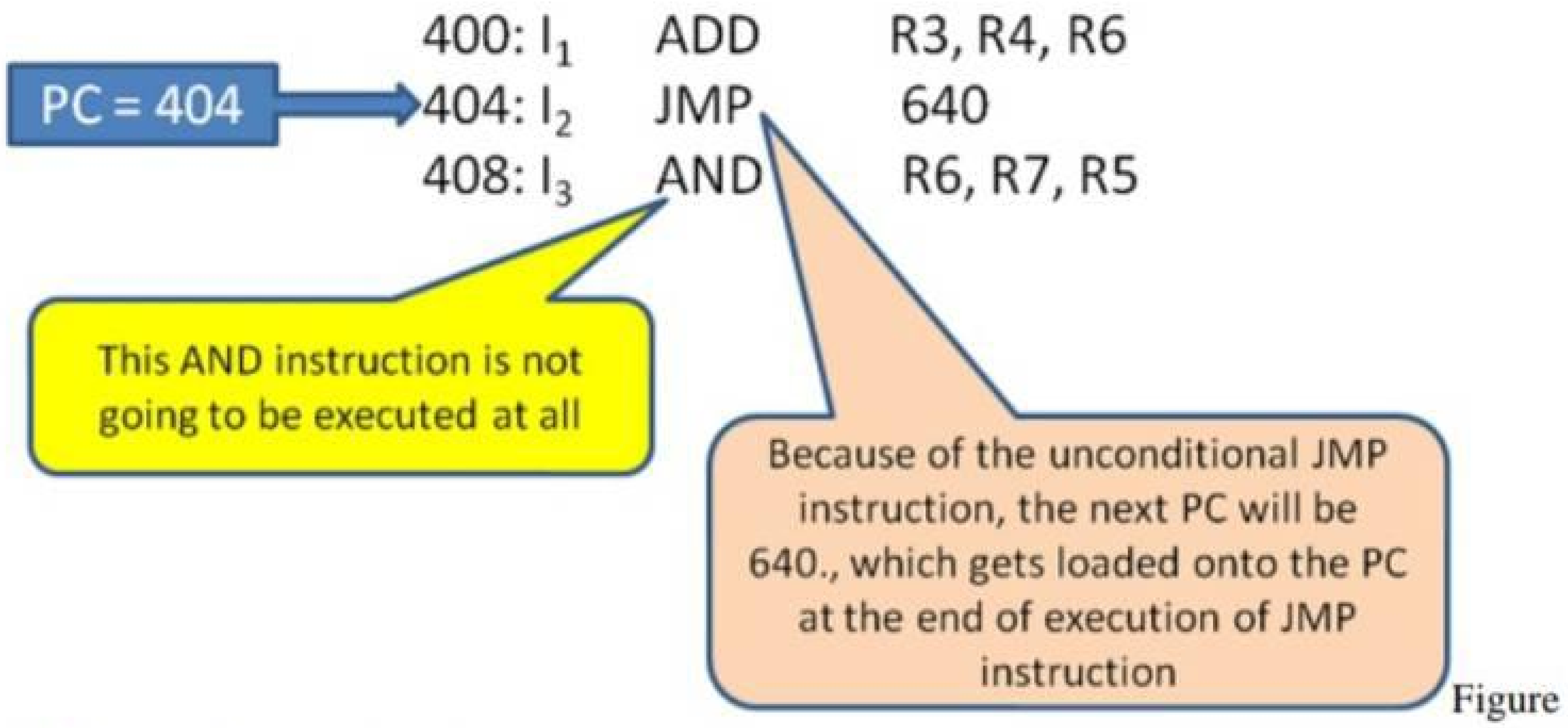


Figure 16.5 Control Hazard scenario

In this case, there is no point in fetching the I3. What happens to the pipeline? While in I2, the I3 fetch needs to be stopped. This can be known only after I2 is decoded as JMP and not until then. So the pipeline cannot proceed at its speed and hence this is a Control Dependency (hazard). In case I3 is fetched in the meantime, it is not only a redundant work but possibly some data in registers might have got altered and needs to be undone.

Similar scenarios arise with conditional JMP or BRANCH.

Solutions for Conditional Hazards

Stall the Pipeline as soon as decoding any kind of branch instructions. Just not allow anymore IF. As always, stalling reduces throughput. The statistics say that in a program, at least 30% of the instructions are BRANCH. Essentially the pipeline operates at 50% capacity with Stalling.

Prediction – Imagine a for or while loop getting executed for 100 times. We know for sure 100 times the program flows without the branch condition being met. Only in the 101st time, the program comes out of the loop. So, it is wiser to allow the pipeline to proceed and undo/flush when the branch condition is met. This does not affect the throttle of the pipeline as much stalling.

Dynamic Branch Prediction - A history record is maintained with the help of Branch Table Buffer (BTB). The BTB is a kind of cache, which has a set of entries, with the PC address of the Branch Instruction and the corresponding effective branch address. This is maintained for every

branch instruction encountered. SO whenever a conditional branch instruction is encountered, a lookup for the matching branch instruction address from the BTB is done. If hit, then the corresponding target branch address is used for fetching the next instruction. This is called dynamic branch prediction.

Branch Instruction Address	Target Branch address taken

Figure 16.6 Branch Table Buffer

This method is successful to the extent of the temporal locality of reference in the programs. When the prediction fails flushing needs to take place.

Reordering instructions - Delayed branch i.e. reordering the instructions to position the branch instruction later in the order, such that safe and useful instructions which are not affected by the result of a branch are brought-in earlier in the sequence thus delaying the branch instruction fetch. If no such instructions are available then NOP is introduced. This delayed branch is applied with the help of Compiler.

Control unit is expected to handle the following scenarios:

- No Dependence
- Dependence requiring Stall
- Dependence solution by Forwarding
- Dependence with access in order
- Out of Order Execution
- Branch Prediction Table and more