21CSS201T
COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-4

Contents

- *Basic processing unit*
- *ALU operations*
- *Instruction execution*
- *Branch instruction*
- *Multiple bus organization*
- *Hardwired control*
- *Generation of control signals,*
- *Micro-programmed control*
- *Pipelining: Basic concepts of pipelining*
- *Performance*
- *Hazards-Data, Instruction and Control*
- *Influence on instruction sets.*

Overview

- Instruction Set Processor (ISP)
- Central Processing Unit (CPU)
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- An instruction is executedby carrying out a sequence of more rudimentary operations.

Fundamental Concepts

- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

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

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 **(Fetch phase)**.

$$PC \leftarrow [PC] + 4$$

- Carry out the actions specified by the instruction in the IR **(Execution phase)**.

Processor Organization

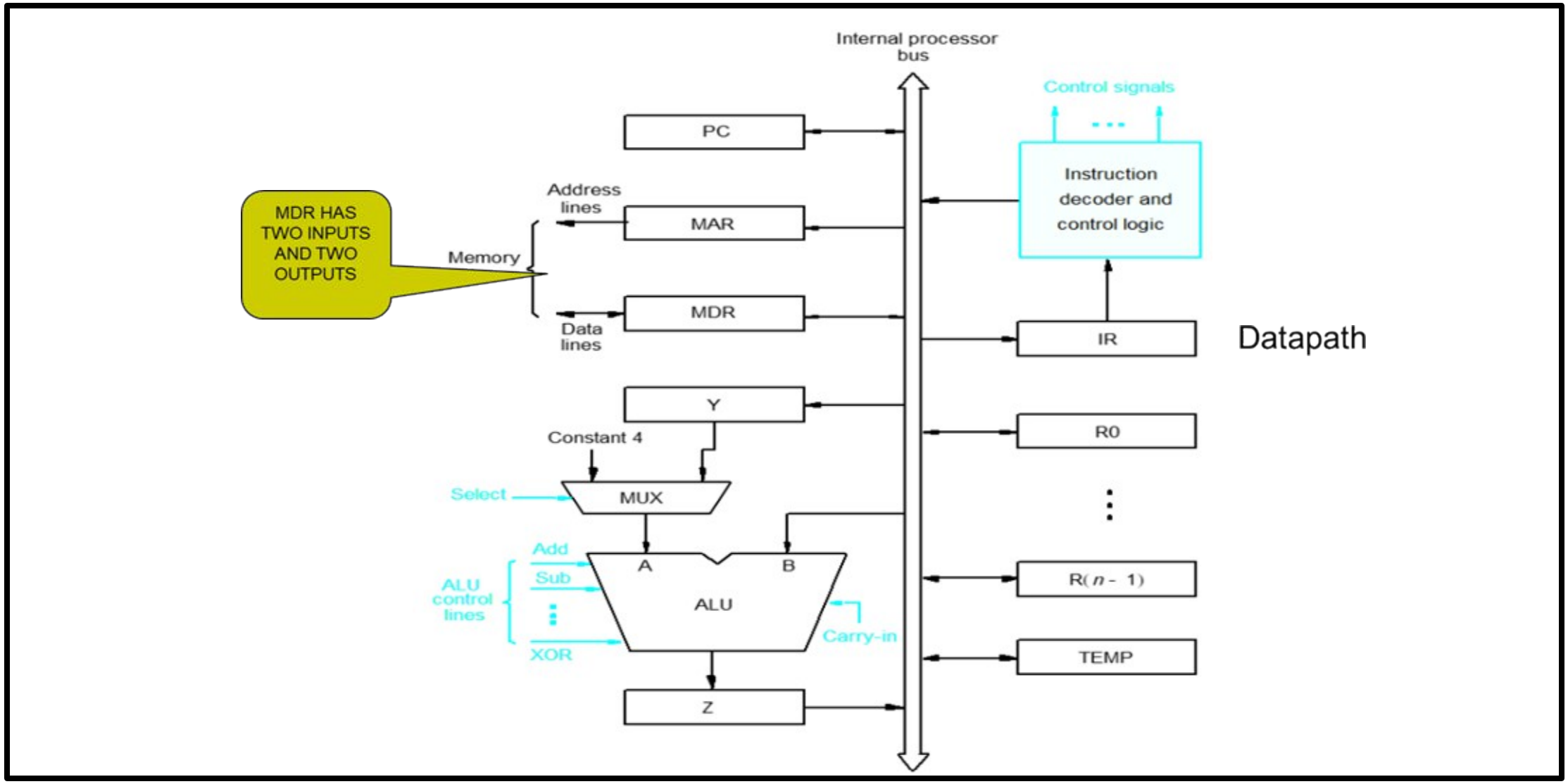


Figure. Single Bus Organization of the Datapath Inside a Processor

Executing an Instruction

An instruction can be executed by performing one or more of the following operations:

- ➢ Transfer a word of data from one processor register to another or to the ALU.
- ➢ Perform an arithmetic or a logic operation and store the result in a processor register.
- ➢ Fetch the contents of a given memory location and load them into a processor register.
- ➢ Store a word of data from a processor register into a given memory location

Single Bus Organization

- ALU
- Registers for temporary storage
- Various digital circuits for executing different micro operations.(gates, MUX, decoders, counters).
- Internal path for movement of data between ALU and registers.
- Driver circuits for transmitting signals to external units.
- Receiver circuits for incoming signals from external units.

Single Bus Organization - contd.

Program Counter (PC)

- Keeps track of execution of a program
- Contains the memory address of the next instruction to be fetched and executed.

Memory Address Register (MAR)

- Holds the address of the location to be accessed.
- I/P of MAR is connected to Internal bus and an O/P to external bus.

Memory Data Register (MDR)

- It contains data to be written into or read out of the addressed location.
- It has 2 inputs and 2 outputs.
- Data can be loaded into MDR either from memory bus or from internal processor bus.
- The data and address lines are connected to the internal bus via MDR and MAR

Single Bus Organization - contd.

Registers

- The processor registers R_0 to R_{n-1} vary considerably from one processor to another.
- Registers are provided for general purpose used by programmer.
- Special purpose registers-index & stack registers.
- Registers Y, Z & TEMP are temporary registers used by processor during the execution of some instruction.

Multiplexer

- Select either the output of the register Y or a constant value 4 to be provided as input A of the ALU.
- Constant 4 is used by the processor to increment the contents of PC.

Single Bus Organization - contd.

ALU

- B input of ALU is obtained directly from processor-bus.
- As instruction execution progresses, data are transferred from one register to another, often passing through ALU to perform arithmetic or logic operation.

Data Path

- The registers, ALU and interconnecting bus are collectively referred to as the data path.

1. Register Transfers

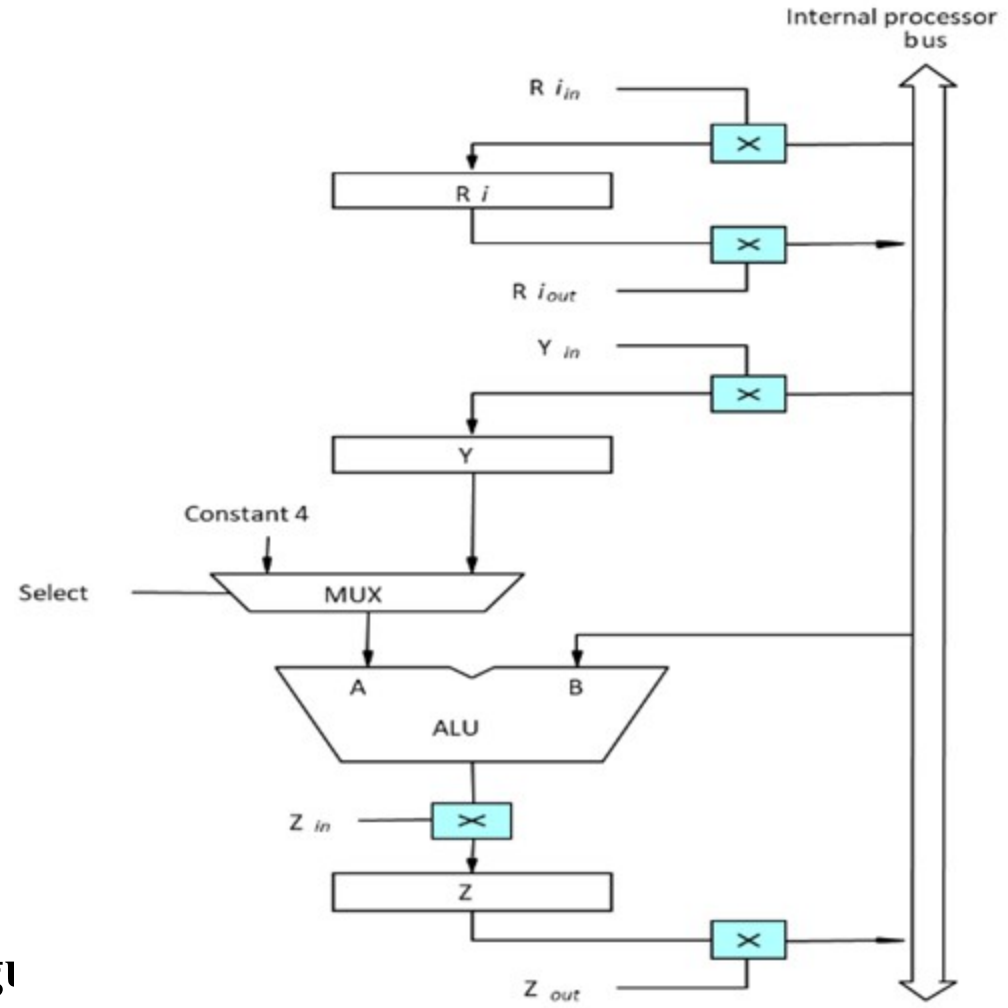
- The input and output gates for register Ri are controlled by signals is Ri_{in} and Ri_{out}
  - Ri_{in} is set to 1 – data available on common bus are loaded into Ri.
  - Ri_{out} is set to 1 – the contents of register are placed on the bus.
  - Ri_{out} is set to 0 – the bus can be used for transferring data from other registers .
- All operations and data transfers within the processortake place within time-periods defined by the processor clock.
- When edge-triggered flip-flops are not used, 2 or more clock-signals may be needed to guarantee proper transfer of data. This is known as multiphase clocking.

Data Transfer Between Two Registers

Example:

- Transfer the contents of R1 to R4.
- Enable output of register R1 by setting $R1_{out}$=1. This places the contents of R1 on the processor bus.
- Enable input of register R4 by setting $R4_{in}$=1. This loads the data from the processor bus into register R4.

Input and Output Gating for Registers



Figu

Input and Output Gating for One Register Bit

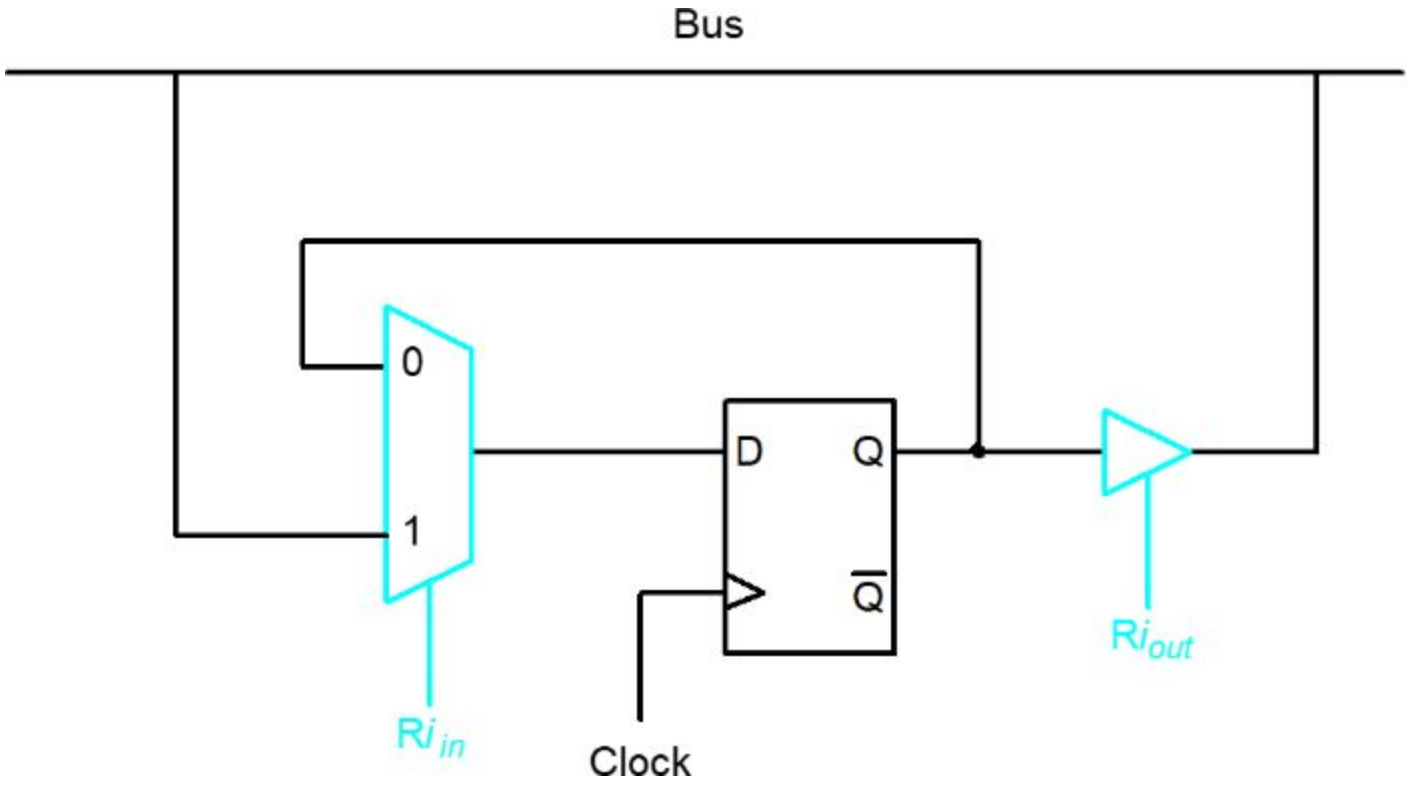


Figure. Input and Output Gating for One Register Bit

Input and Output Gating for One Register Bit - contd

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
  - When Ri_{in}=1, mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock.
  - When Ri_{in} = 0, mux feeds back the value currently stored in flip-flop.
- Q output of flip-flop is connected to bus via a tri-state gate.
  - When Ri_{out}= 0, gate's output is in the high-impedance state. (This corresponds to the open circuit state of a switch).
  - When Ri_{out}=1, the gate drives the bus to 0 or 1, depending on the value of Q.

2. Performing an ALU Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.

- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?
  - $R1_{out}$, Y_{in}
  - $R2_{out}$, SelectY, Add, Z_{in}
  - Z_{out}, $R3_{in}$

ALU Operation - contd

The sequence of operations for [R3] ⇦ [R1]+[R2] is as follows

- Step 1: Output of the register R1 and input of the register Y are enabled, causing the contents of R1 to be transferred to Y.
- Step 2: The multiplexer select signal is set to select Y causing the multiplexer to gate the contents of register Y to input A of the ALU.
- Step 3: The contents of Z are transferred to the destination register R3.

3.Fetching a Word from Memory

- The response time of each memory access varies (cache miss, memory-mapped I/O,…)
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed **(Memory-Function-Completed, MFC)**.

- Move (R1), R2
  - MAR ← [R1]
  - Start a Read operation on the memory bus
  - Wait for the MFC response from the memory
  - Load MDR from the memory bus
  - R2 ← [MDR]

Fetching a Word - contd.

➢ Address into MAR; issue Read operation; data into MDR

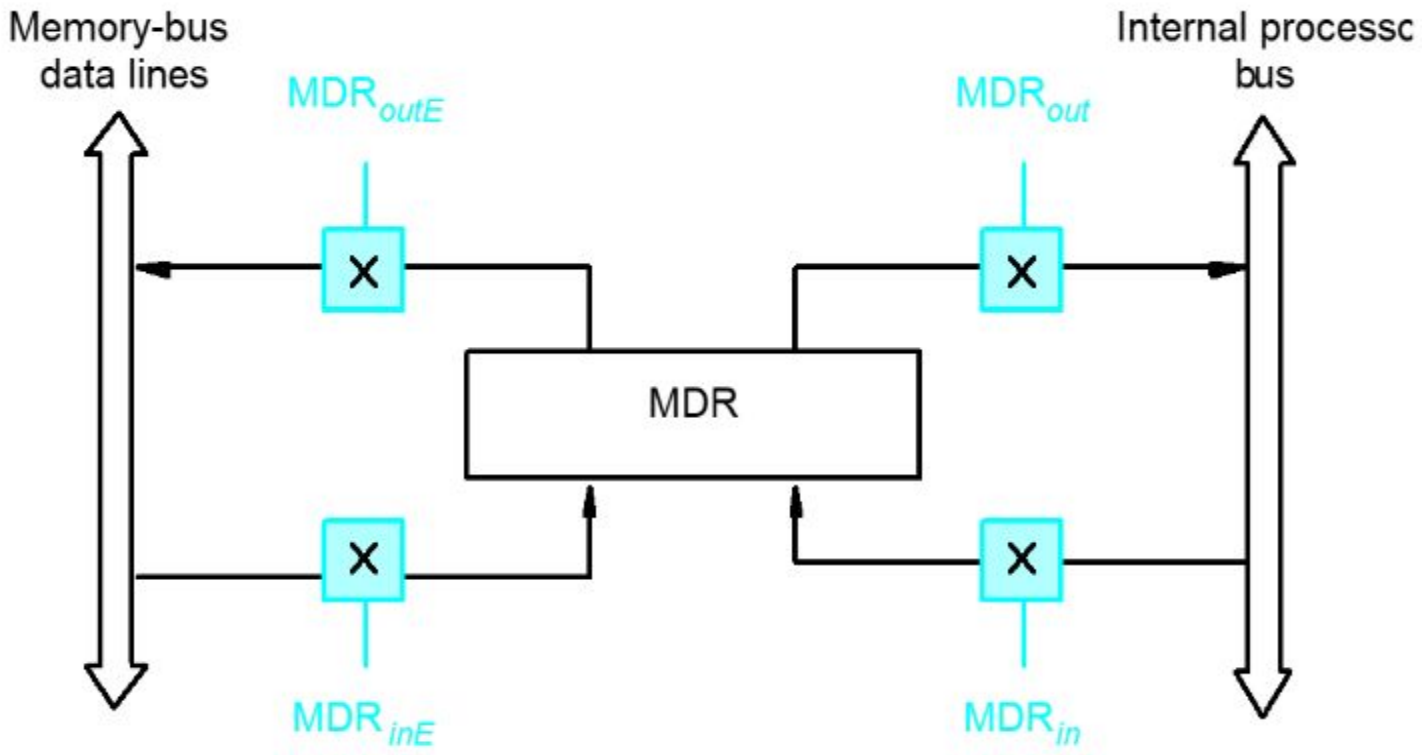


Figure. Connection and Control Signals for Register MDR

Timing

➢ Assume MAR is always available on the address lines of the memory bus

Move (R1), R2

1.$R1_{out}$, MARin, Read

2.$MDR_{in}E$, WMFC

3.MDR_{out}, $R2_{in}$

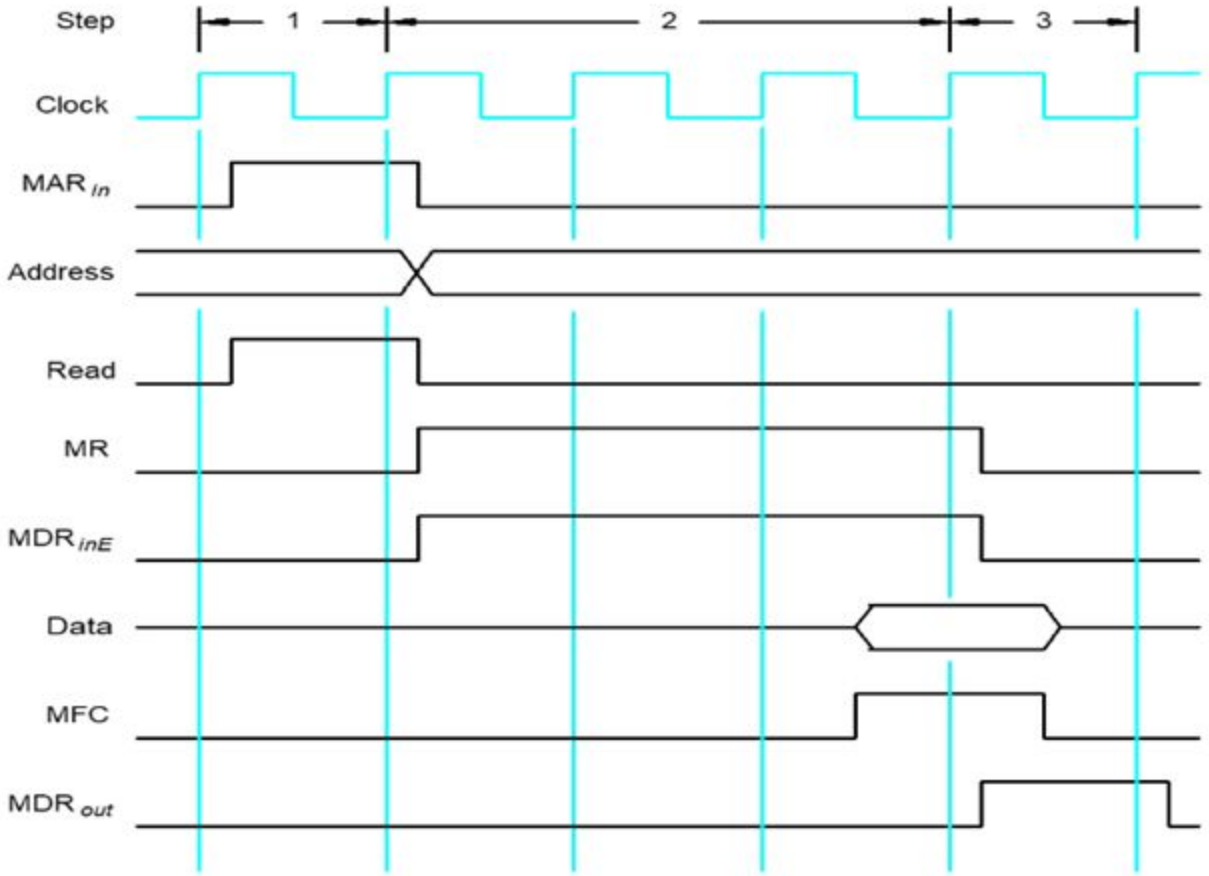


Figure. Timing of a Memory Read Operation

4.Storing a Word from Memory

- Address is loaded into MAR
- Data to be written loaded into MDR.
- Write command is issued.
- Example: Move R2,(R1)
  - $R1_{out}$,MAR_{in}
  - $R2_{out}$,MDR_{in},Write
  - MDR_{out}E, WMFC

Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Load the result into R1

Execution of a Complete Instruction

➢ Add (R3), R1

Step	Action
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

Figure. Control Sequence for Execution

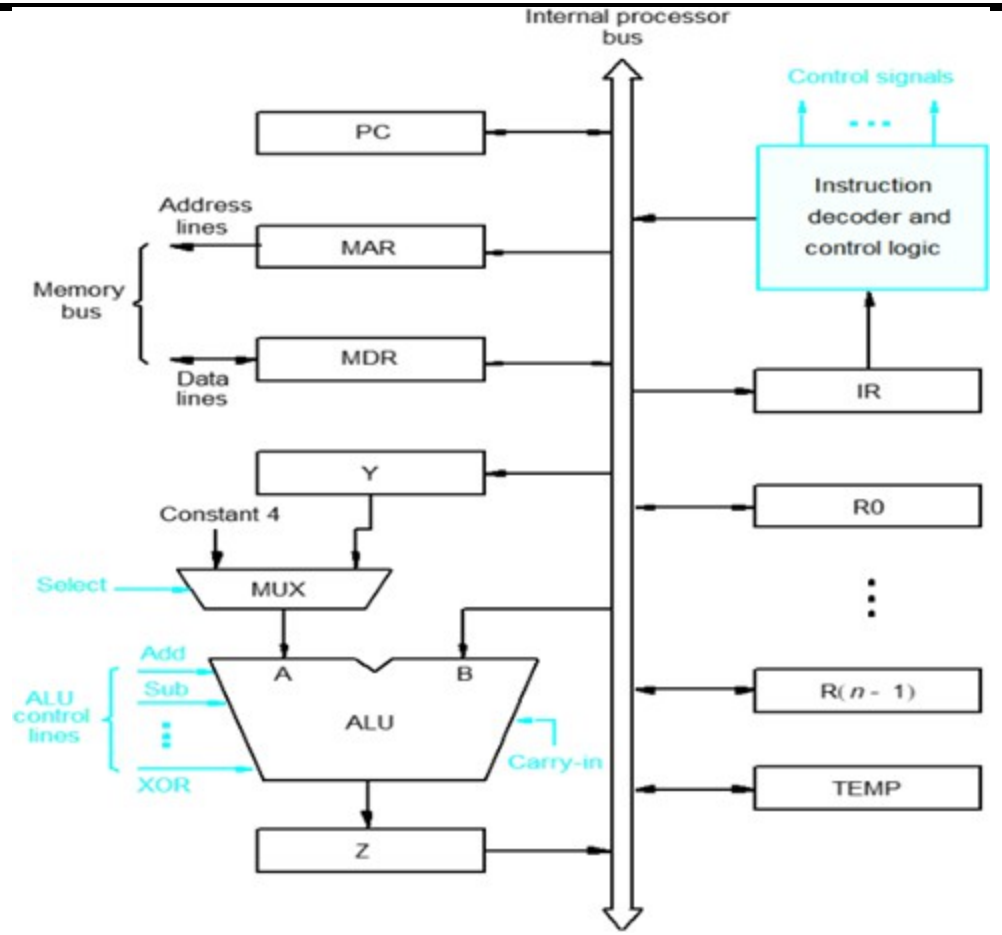


Figure. Single Bus Organization of the Datapath inside a processor

Execution of Branch Instruction

- A branch instruction replaces the contents of PC with the branch target address,

  which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address

  immediately following the branch instruction.

- UnConditional branch

Execution of Branch Instruction

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMF C
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure. Control Sequence for Unconditional Branch Instructions

Simple single-bus structure

- Has one common bus for data transfer.
- Interconnected circuits/devices which has varying speed of execution like CPU and memory.
- Results in long control sequences, because only one data item can be transferred over the bus in a clock cycle.

Multi-bus structure

- Most commercial processors provide multiple internal paths to enable several transfers to take place in parallel.
- Data transfer requires less control sequences.
- Mutiple data transfer can be done in a single clock cycle

Multibus Architecture

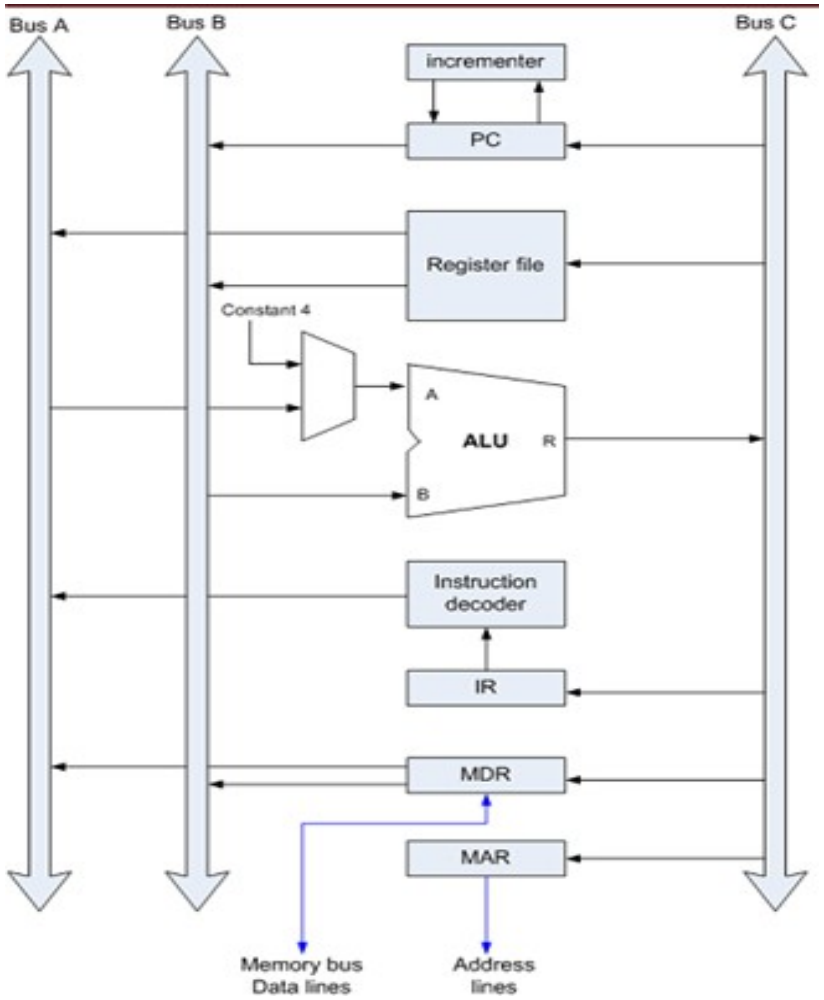
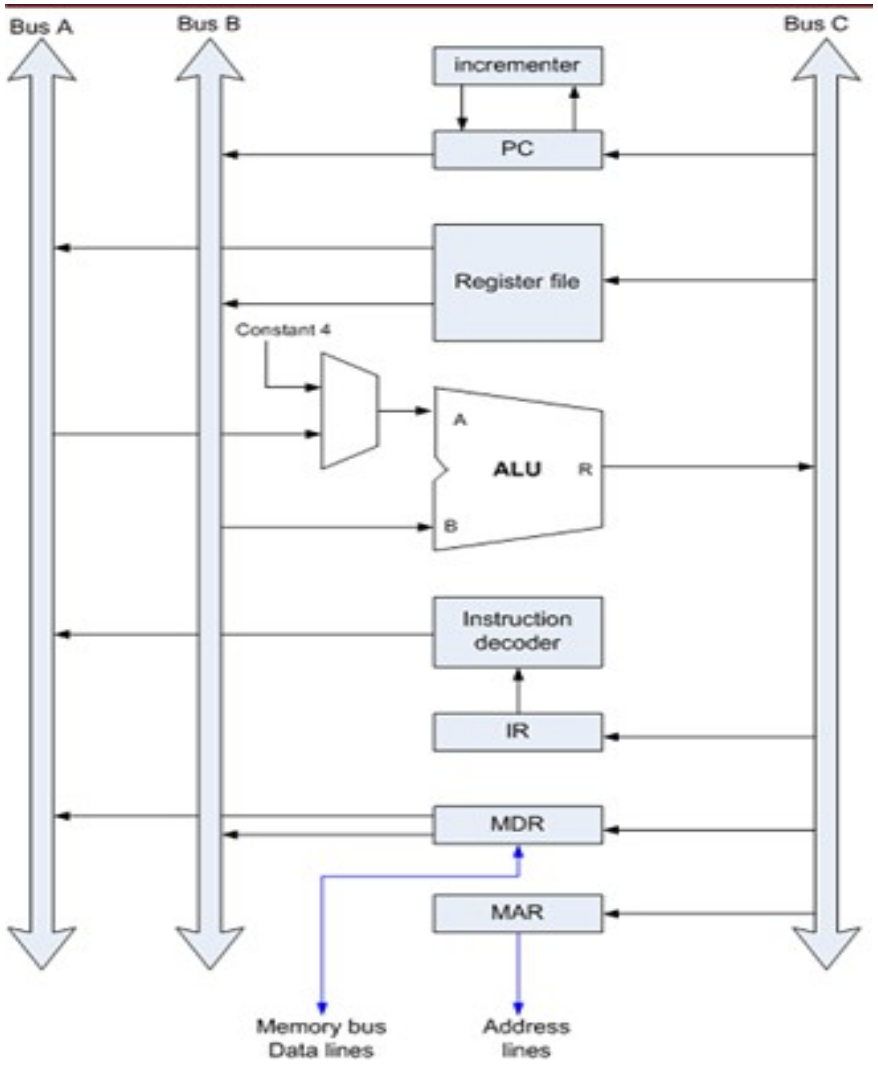


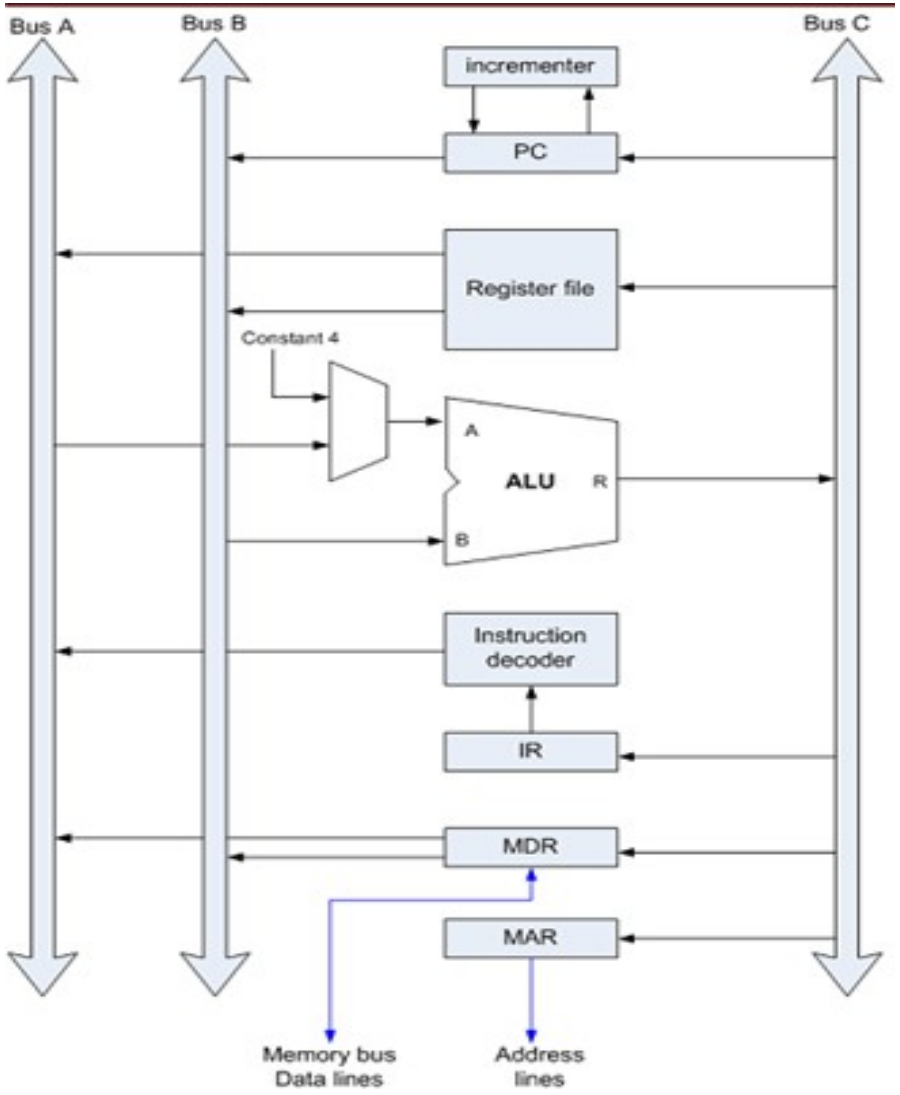
Image from Computer Organization By Carl Hamacher

Multi-Bus Organization

- Three-bus organization to connect the registers and the ALU of a processor.
- All general-purpose registers are combined into a single block called register file.
- Register file has three ports.
- Two outputs ports connected to buses A and B, allowing the contents of two different registers to be accessed simultaneously, and placed on buses A and B.
- Third input port allows the data on bus C to be loaded into a third register during the same clock cycle.
- Inputs to the ALU and outputs from the ALU:
- Buses A and B are used to transfer the source operands to the A and B inputs of the ALU.
- Result is transferred to the destination over Bus C



- ALU can also pass one of its two input operands unmodified if needed:
- Control signals for such an operation are R=A or R=B.
- Three bus arrangement obviates the need for Registers Y and Z in the single bus organization.
- Incrementer unit: Used to increment the PC by 4.
- Source for the constant 4 at the ALU multiplexer can be used to increment other addresses such as the memory addresses in multiple load/store instructions.



Input Output Specifications

Component	Input	Output
Program Counter	PC_{IN}	PC_{OUT}
Register File	$R1_{IN}$	$R1_{OUT}$
ALU	A ,B 4	R=B for selecting B Select A for selecting A Select 4 – for fetching consecutive memory locations
Instruction Register	IR_{IN}	IR_{OUT}
Memory Data Register	MDR_{IN}	$MDR_{OUTA/OUTB}$
Memory Address Register	MAR_{IN}	MAR_{OUT}

“Add R4 R5 R6” in multibus environment

Step	Action	Remarks
1	PC_{out} R=B MAR_{in} Read , IncPC	Pass the contents of the PC through ALU and load it into MAR. Increment PC.
2	WMFC	Wait for Memory Function Complete.
3	MDR_{outB} R=B IR_{in}	Load the data received into MDR and transfer to IR through ALU.
4	$R4_{outA}$ $R5_{outB}$, SELECTA ADD $R6_{IN}$ END	Output the Register R4 and R5 contents to Bus A and Bus B respectively. Add R4 and R5 and input the result from ALU in register R6

Control Unit Signals

To execute instructions the processor must generate the necessary control signals in proper sequence.

Hardwired control

- Control signals are produced by appropriate circuitries.
- Control unit is designed as a finite state machine.
- Inflexible but fast.
- Appropriate for simpler machines (e.g. RISC machines)

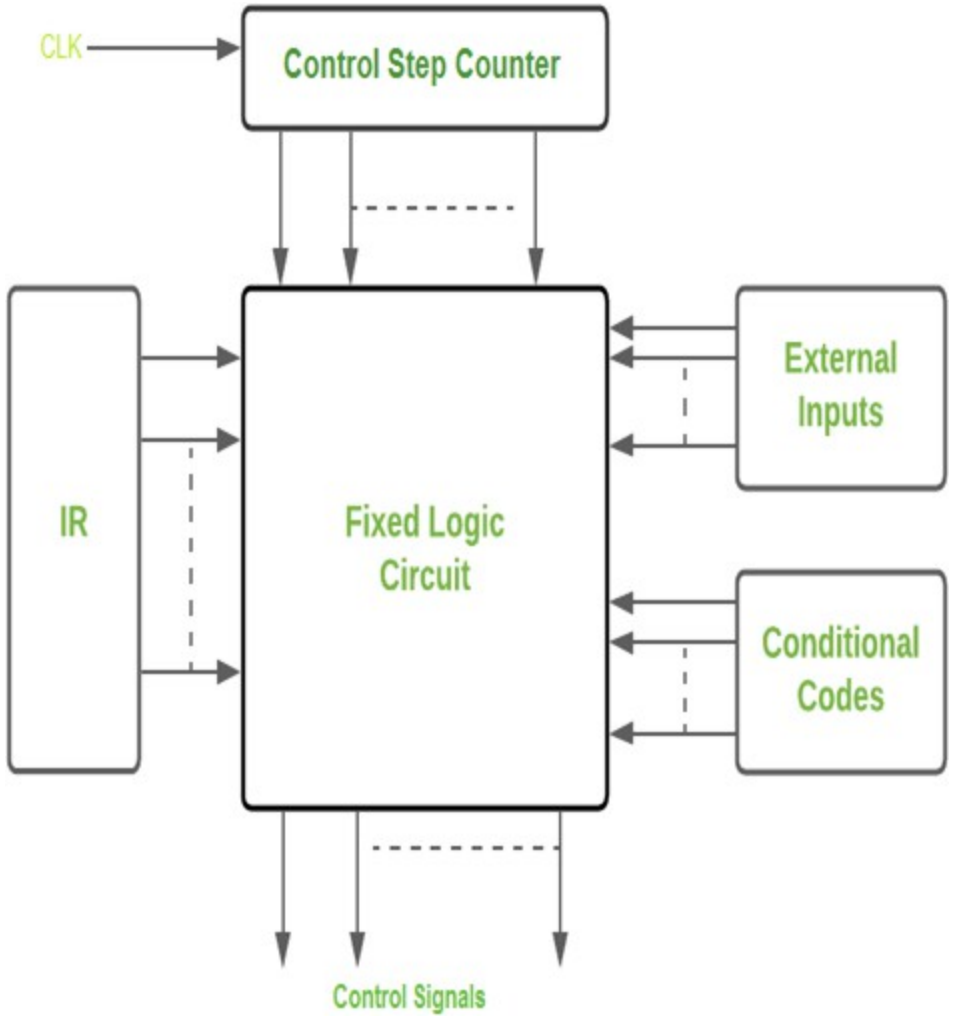
Microprogrammed control

- Control signals are generated as a micro program consisting of 0s and 1s.
- Control path is designed hierarchically using principles identical to the CPU design.
- Flexible, but slow.
- Appropriate for complex machines (e.g. CISC machines)

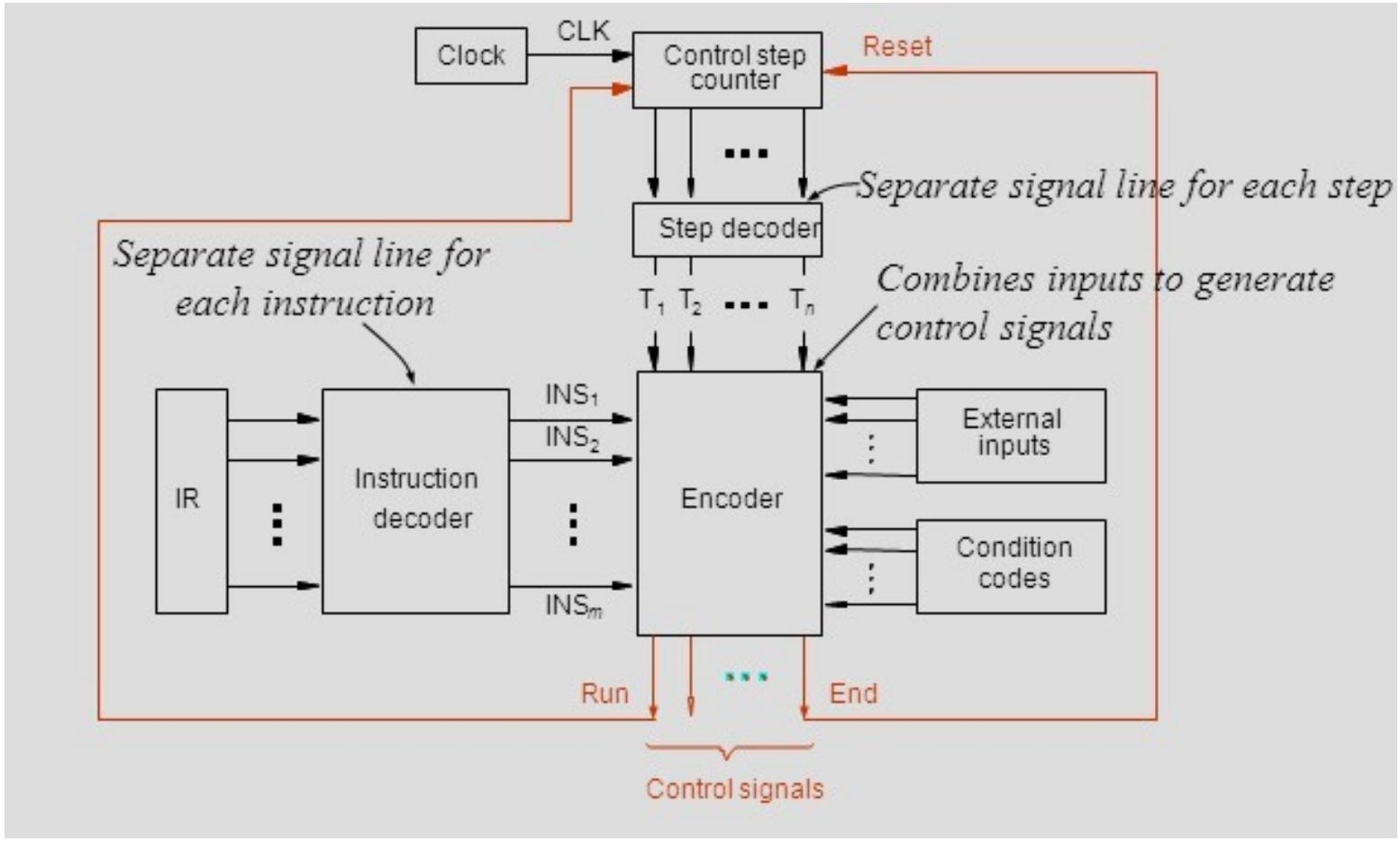
Hardwired Control

- Required control signals are determined by the following information:

- Contents of the control step counter- Determines which step in the sequence.
- Contents of the instruction register - Determines the actual instruction.
- Contents of the condition code flags - Used for example in a BRANCH instruction.
- External input signals such as MFC.
- The decoder/encoder block in the Figure is a combinational circuit that generates the required control outputs, depending on the state of all above inputs.



Hardwired Control

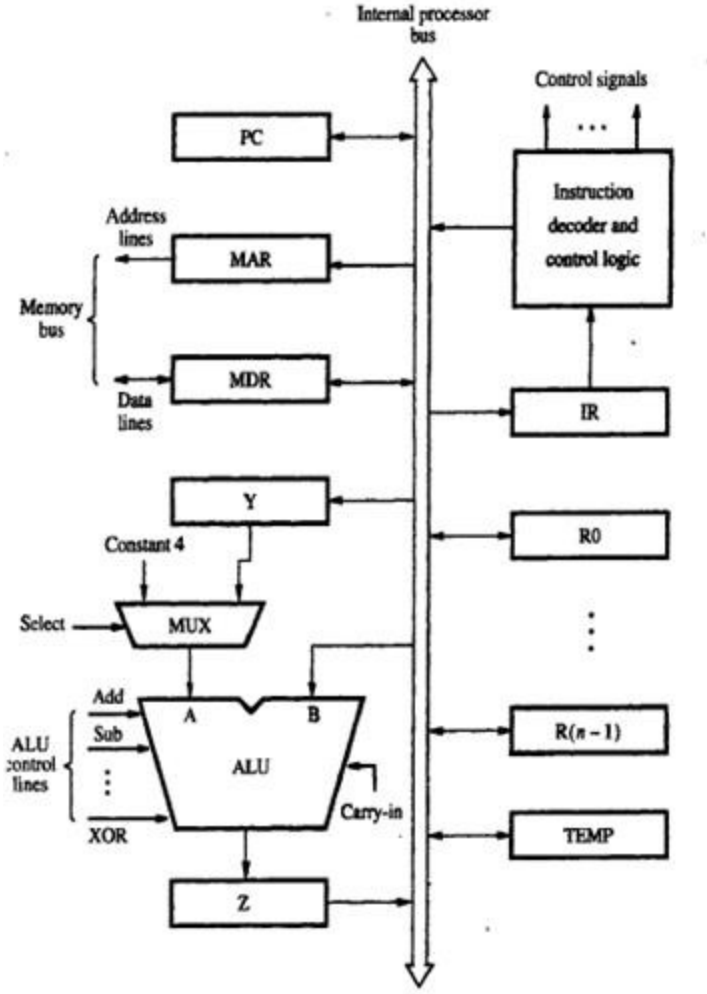


- Each step in this sequence is completed in one clock cycle.
- A counter is used to keep track of the control steps.
- Each state or count, of this counter corresponds to one control step.

Step	Action
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

Control Sequences for "**Add (R3), R1**" in a ~~single bus architecture~~

Step	Action
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



Control Sequences for **Unconditional Branch** in a single bus architecture

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

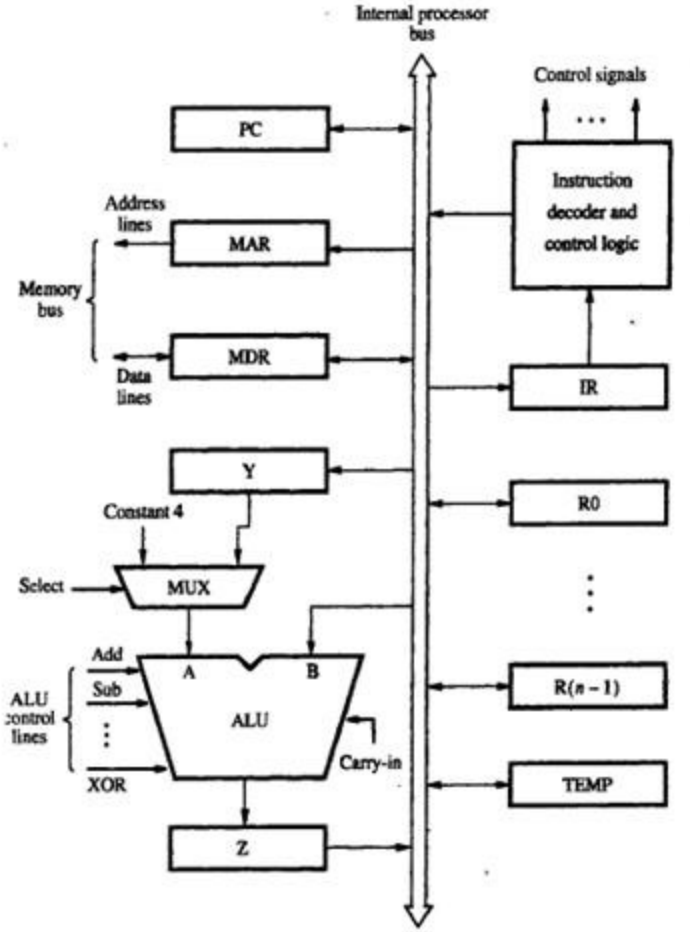
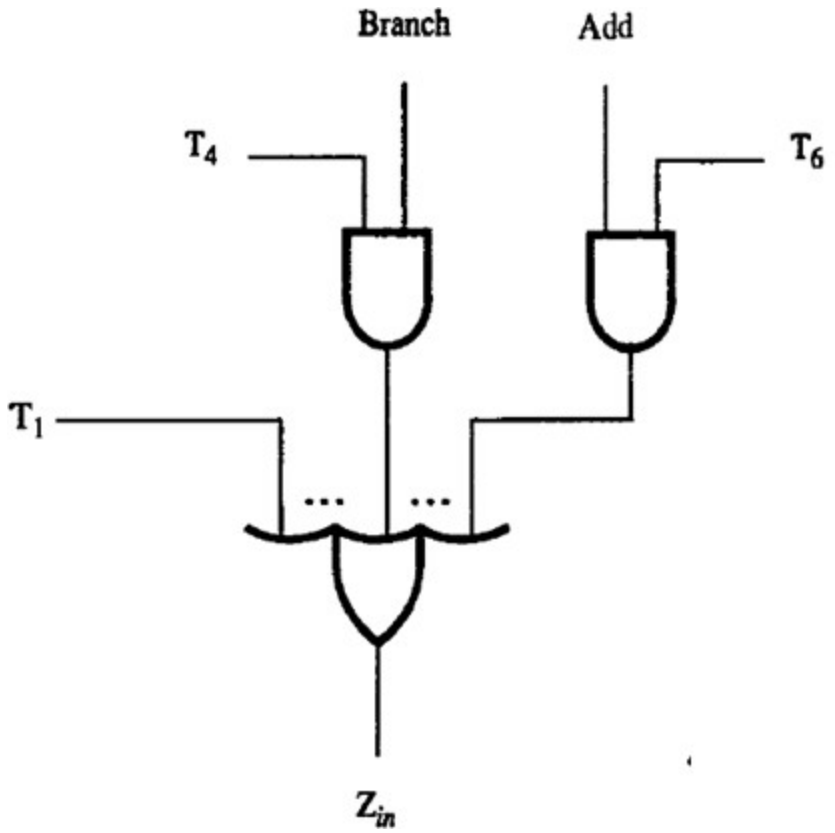


Figure 7.1 Single-bus organization of the datapath inside a processor.

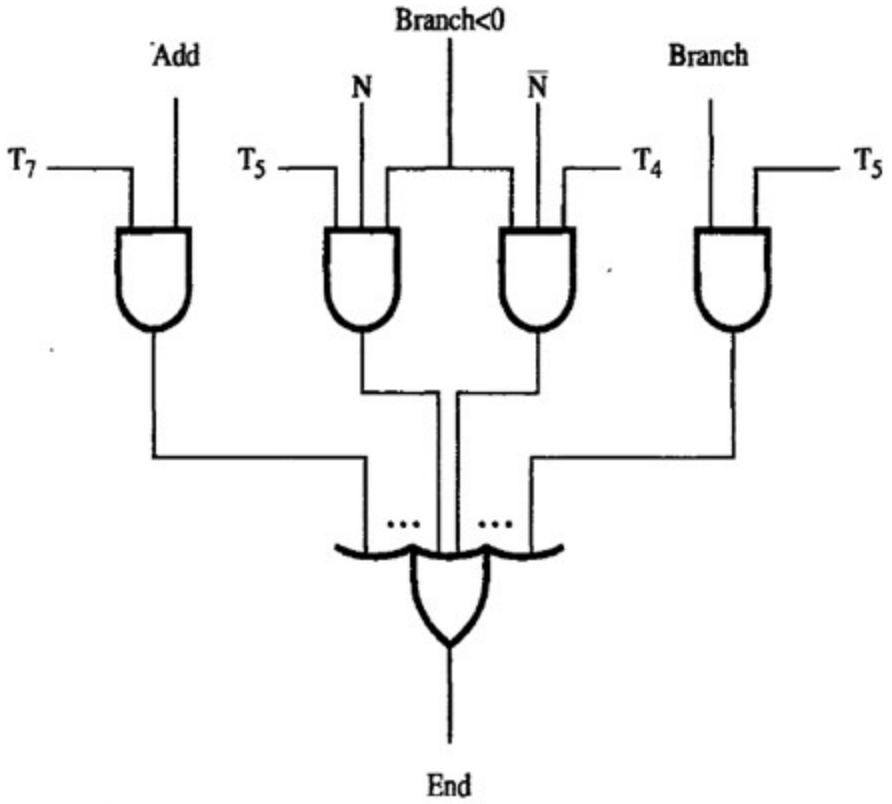
Logic Function for signal Z_{IN}

- **Z_{IN} = T1 + T6.ADD +T4.Br+…..**
- The above function is arrived by considering both ADD and BRANCH statements.
- The equation can be extended if more instructions are there to be executed.
- Here **Z_{IN}** occurs in the following steps.
  - first step of both the statements.
  - in step 6 along with add instruction.
  - In step 4 along with branching in the unconditional branching.



Logic Function for signal END

- END = T7.ADD+T5.BR+(T5.N+T4Ň).BRN+…..
- The above function is arrived by considering both ADD and BRANCH statements.
- The equation can be extended if more instructions are there to be executed.
- The end signal starts a new instruction fetch cycle by resetting the control step counter to its starting value.
- Here **END** occurs in the following steps.
  - in step 7 along with add instruction.
  - In step 5 along with branching
  - Either in Step 4 or step 5 in the unconditional branching along with the conditional inputs.



END and RUN signals

- The end signal starts a new instruction fetch cycle by resetting the control step counter to its starting value.
- The run signal will make the step counter count up every time but when set to 0 this signal will make the counter to hold irrespective of the clock pulse. This is done WMFC signal is issued.

Microprogrammed Control

- The Control signals are generated through a program similar to machine language programs.
- Here we use a sequence of bits to notify, the signals that are to be set for a particular action.
- For example if PC_{out} is used in a particular step then the bit allocated for PC_{out} will be set to 1.

Control Word

Step	Action
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

Control Signals:

PC_{out}
PC_{in}
MAR_{in}
Read
MDR_{out}
IR_{in}
Y_{in}
SelectY
Select4
Add
Z_{in}
Z_{out}
$R1_{out}$
$R1_{in}$
$R3_{out}$
WMFC
End .....

Sample micro instructions for the listing

Micro - instruction	:	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	;
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

- At every step, a Control W ord needs to be generated.
- Every instruction will need a sequence of CWs for its execution.
- Sequence of CWs for an instruction is the micro routine for the instruction.
- Each CW in this microroutine is referred to as a microinstruction.

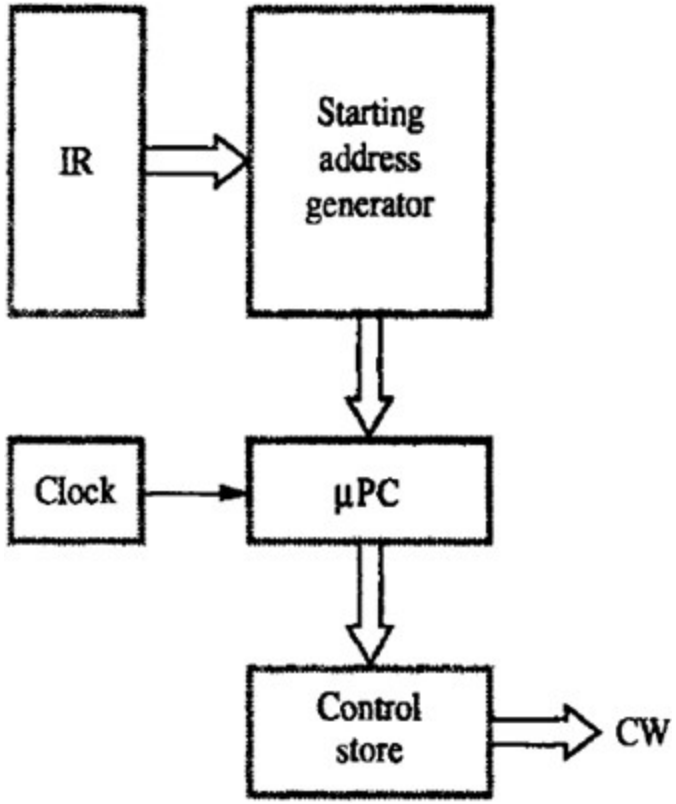
Control words/Microroutines/Control store

- At every step, a Control W ord needs to be generated.
- Every instruction will need a sequence of CWs for its execution.
- Sequence of CWs for an instruction is the micro routine for the instruction.
- Each CW in this microroutine is referred to as a microinstruction

- Every instruction will have its own microroutine which is made up of microinstructions.
- Microroutines for all instructions in the instruction set of a computer are stored in a special memory called Control Store.
- The Control Unit generates the control signals: by sequentially reading the CWs of the corresponding microroutine from the control store.

Basic organization of microprogrammed control unit

- Microprogram counter (μPC) is used to read CWs from control store sequentially.
- When a new instruction is loaded into IR, Starting address generator generates the starting address of the microroutine.
- This address is loaded into the μPC.
- μPC is automatically incremented by the clock, so successive microinstructions are read from the control store.



Status Code and External Inputs

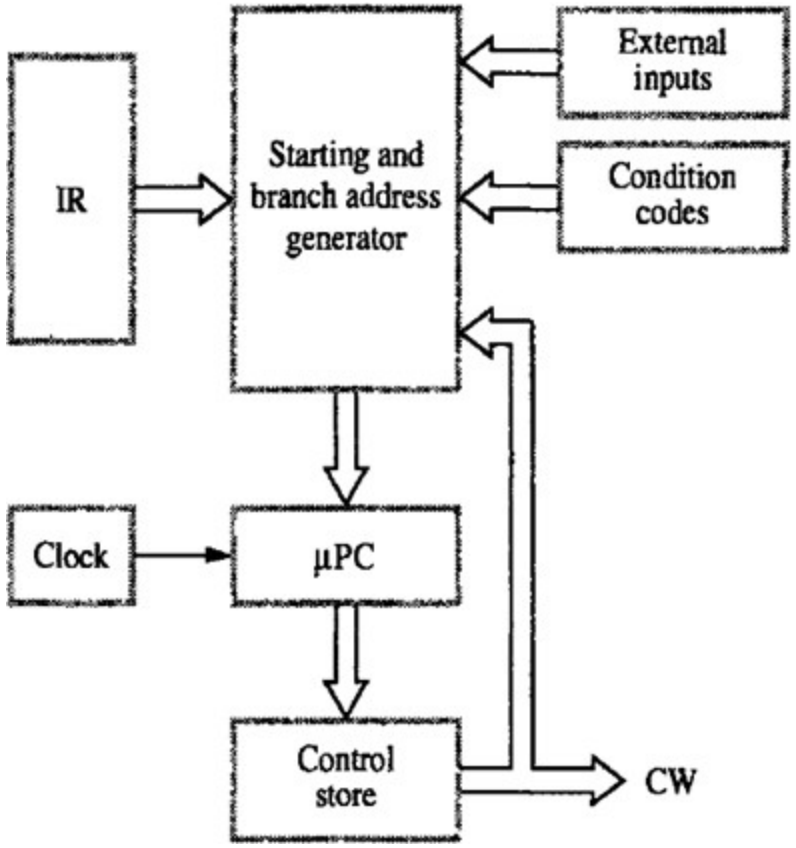
- Basic organization of the microprogrammed control unit cannot check the status of condition codes or external inputs to determine what should be the next microinstruction.
- In the hardwired control, this was handled by an appropriate logic function.
- In microprogrammed control this is handled as:
- Use conditional branch microinstructions.
- These microinstructions, in addition to the branch address also specify which of the external inputs, condition codes or possibly registers should be checked as a condition for branching.

Branching in Microinstructions

- The listing shows that a conditional jump is required to the location 25 from 3.
- This target address should be generated from the starting address generator.
- Some improvements will be done to the control units to accommodate external inputs and condition codes.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
.....	.....
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Changes in starting address generator



Micrprogammed Control Unit

Starting and branch address generator accepts as inputs

- Contents of the Instruction Register IR (as before).
- External inputs
- Condition codes
- Generates a new address and loads it into microprogram counter (μPC) when a microinstruction instructs it do so.

μPC is incremented every time a microinstruction is fetched except:

- New instruction is loaded into IR, μPC is loaded with the starting address of the microroutine for that instruction.
- Branch instruction is encountered and branch condition is satisfied, μPC is loaded with the branch address.
- End instruction is encountered, μPC is loaded with the address of the first CW in the microroutine for the instruction fetch cycle.

Reducing the size of the microinstructions

- Simple approach is to allocate one bit for each control signal- Results in long microinstructions, since the number of control signals is usually very large.
- Few bits are set to 1 in any microinstruction, resulting in a poor use of bit space.
- Reduce the length of the microinstruction by taking advantage of the fact that most signals are not needed simultaneously,
- Many signals are mutually exclusive. For example:- Only one ALU function is active at a time.- Source for a data transfer must be unique.- Read and Write memory signals cannot be active simultaneously.
- Group mutually exclusive signals in the same group. At most one microperation can be specified per group.
- Use binary coding scheme to represent signals within a group.

Examples:

- If ALU has 16 operations, then 4 bits can be sufficient.
- Group register output signals into the same group, since only one of these signals will be active at any given time.
- If the CPU has 4 general purpose registers, then PCout, MDRout, Zout,Offsetout, R0out, R1out, R2out, R3out and Tempout can be placed in a single group, and 4 bits will be needed to represent these.

- Inserts smallest number of bits that is large enough to fit.
- Most fields include one inactive code specifying no action needed.

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$	1111: XOR	
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	(16 ALU functions)	
0101: $R1_{out}$	101: $R1_{in}$			
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	...

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY	0: No action	0: Continue
1: Select4	1: WMFC	1: End

Basic Concepts of pipelining

How to improve the performance of the processor?

1. By introducing faster circuit technology
2. Arrange the hardware in such a way that, more than one operation can be performed at the same time.

What is Pipeining?

It is the process of arrangement of hardware elements in such way that, simultaneous execution of more than one instruction takes place in a pipelined processor so as to increase the overall performance.

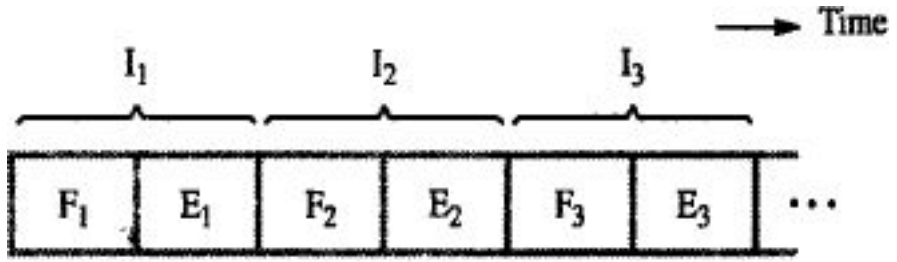
What is Instruction Pipeining?

- The number of instruction are pipelined and the execution of current instruction is overlapped by the execution of the subsequent instruction.
- It is a instruction level parallelism where execution of current instruction does not wait until the previous instruction has executed completely.

Basic idea of Instruction Pipelining

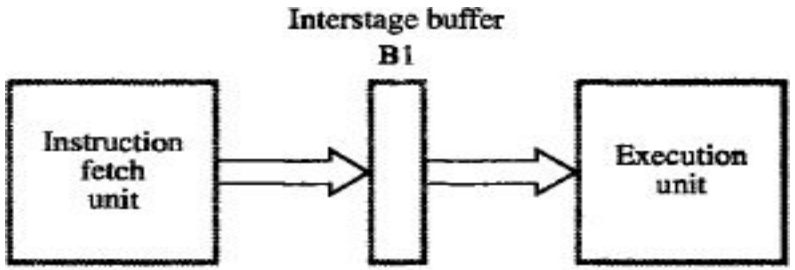
Sequential Execution of a program

- The processor executes a program by fetching(Fi) and executing(Ei) instructions one by one.



Sequential execution

- Consists of 2 hardware units one for fetching and another one for execution as follows.
- Also has intermediate buffer to store the fetched instruction

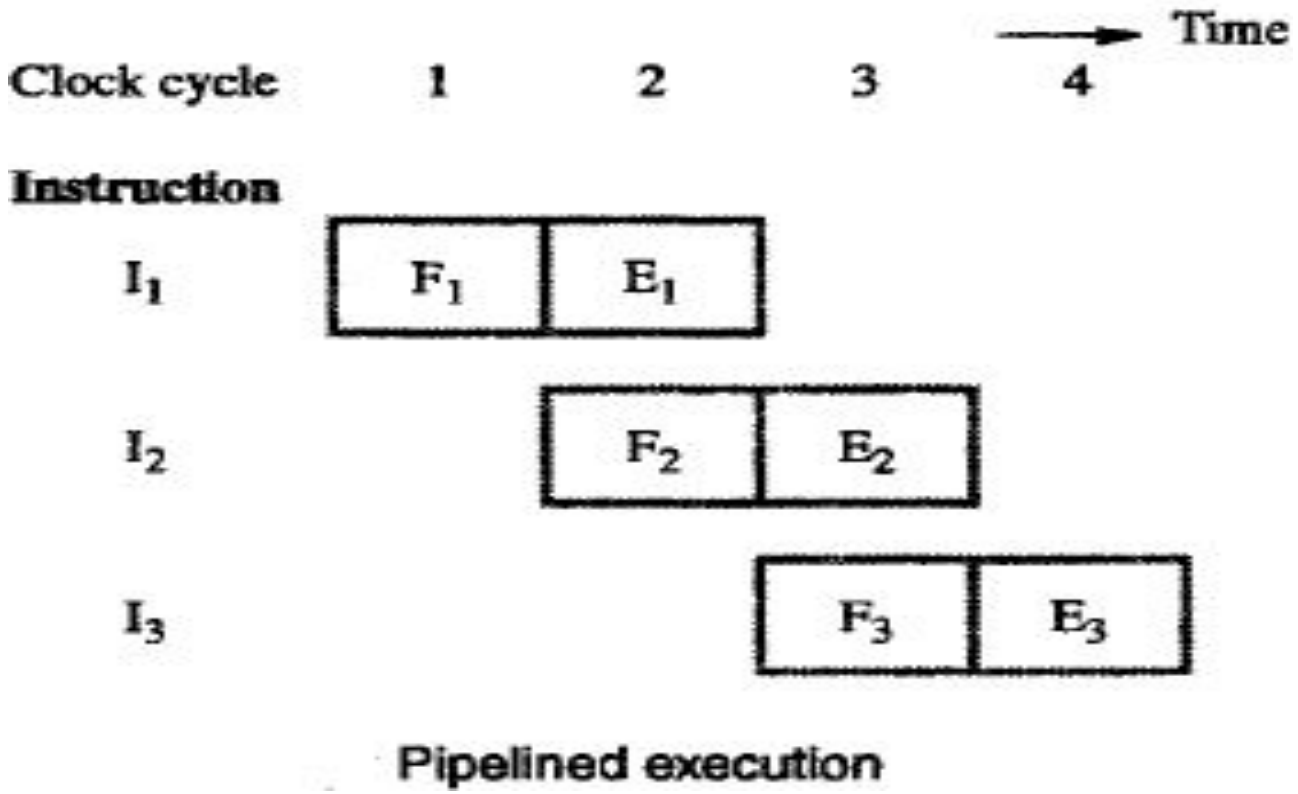


Hardware organization

2 stage pipeline

- Execution of instruction in pipeline manner is controlled by a clock.
- In the first clock cycle, fetch unit fetches the instruction I1 and store it in buffer B1.
- Inthe second clock cycle, fetch unit fetches the instruction I2 , and execution unit executes the instruction I1 which is available in buffer B1.
- By the end of the second clock cycle, execution of I1 gets completed and the instruction I2 is available in buffer B1.
- In the third clock cycle, fetch unit fetches the instruction I3 , and execution unit executes the instruction I2 which is available in buffer B1.
- In this way both fetch and execute units are kept busy always.

2 stage pipeline



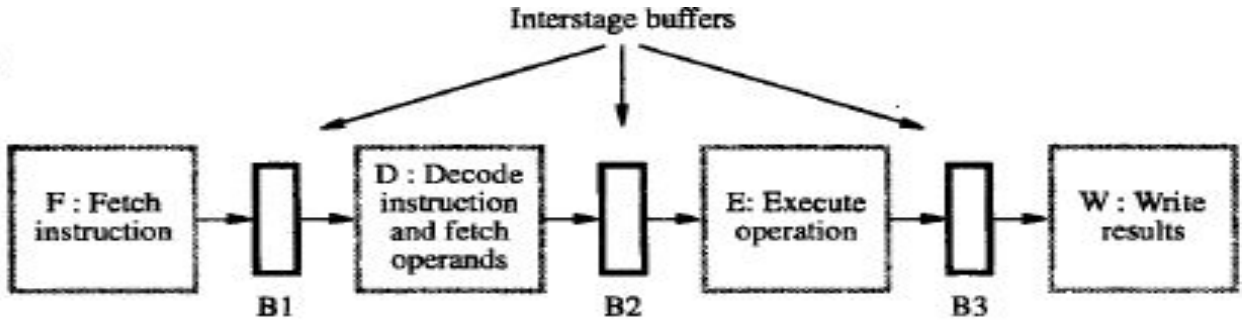
Pipelined execution

Hardware organization for 4 stage pipeline

- Pipelined processor may process each instruction in 4 steps.

1. Fetch(F): Fetch the Instruction
2. Decode(D): Decode the Instruction
3. Execute (E) : Execute the Instruction
4. Write (W) : Write the result in the destination location

- 4 distinct hardware units are needed as shown below.



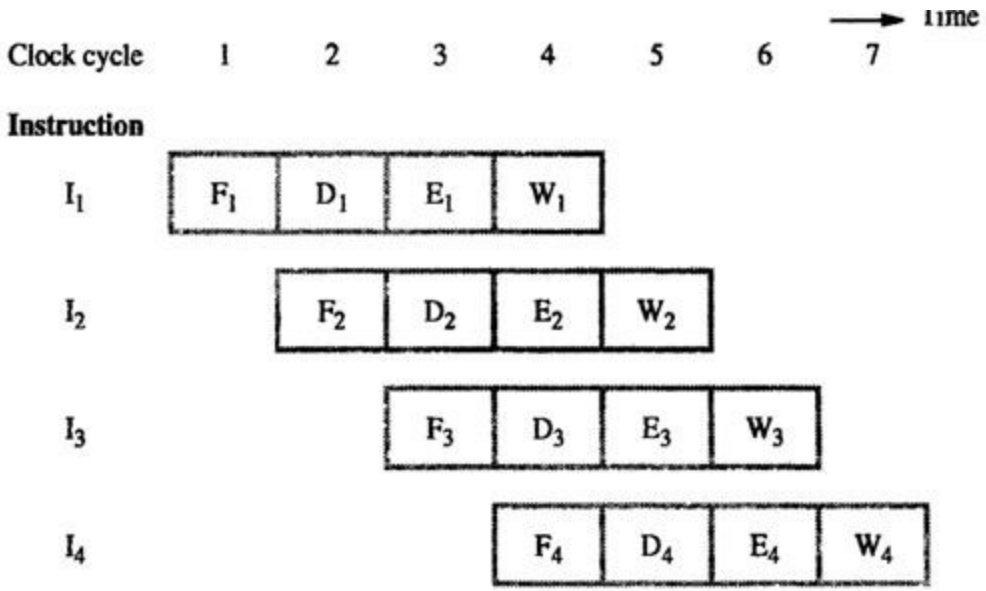
Hardware organization

A 4-stage pipeline.

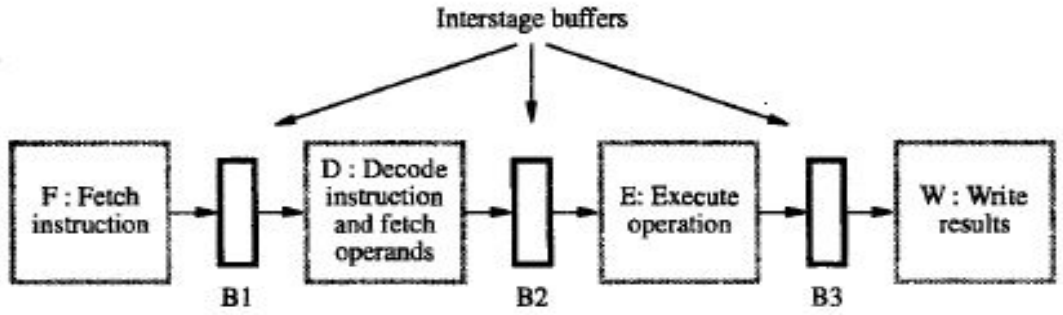
Execution of instruction in 4 stage pipeline

- In the first clock cycle, fetch unit fetches the instruction I1 and store it in buffer B1.
- In the second clock cycle, fetch unit fetches the instruction I2 , and decode unit decodes instruction I1 which is available in buffer B1.
- In the third clock cycle fetch unit fetches the instruction I3 , and decode unit decodes instruction I2 which is available in buffer B1 and execution unit executes the instruction I1 which is available in buffer B2.
- In the fourth clock cycle fetch unit fetches the instruction I4 , and decode unit decodes instruction I3 which is available in buffer B1, execution unit executes the instruction I2 which is available in buffer B2 and write unit write the result of I1.

4- stage pipeline



(a) Instruction execution divided into four steps



Hardware organization

A 4-stage pipeline.

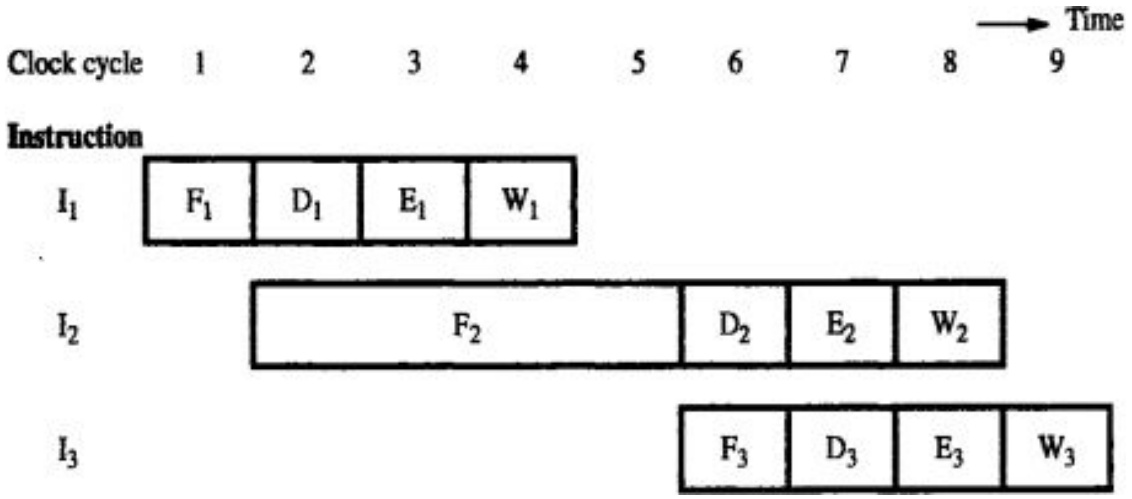
Role of cache memory in Pipelining

- Each stage of the pipeline is controlled by a clock cycle whose period is that the fetch, decode, execute and write steps of any instruction can each be completed in one clock cycle.
- However the access time of the main memory may be much greater than the time required to perform basic pipeline stage operations inside the processor.
- The use of cache memories solve this issue.
- If cache is included on the same chip as the processor, access time to cache is equal to the time required to perform basic pipeline stage operations .

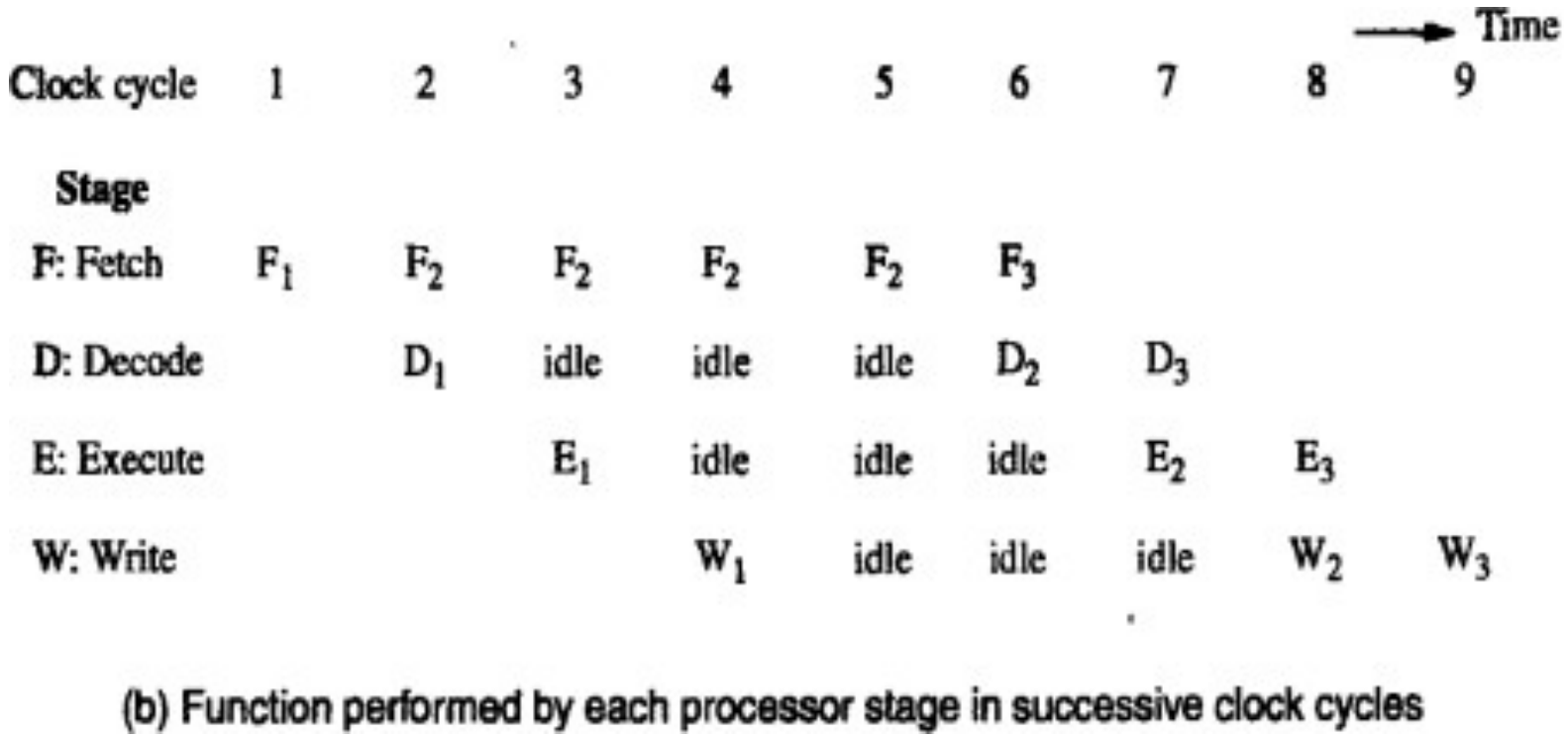
Pipeline Performance

- Pipelining increases the CPU instruction throughput - the number of instructions completed per unit time.
- The increase in instruction throughput means that a program runs faster and has lower total execution time.
- Example in 4 stage pipeline, the rate of instruction processing is 4 times that of sequential processing.
- Increase in performance is proportional to no. of stages used.
- However, this increase in performance is achieved only if the pipelined operation is continued without any interruption.
- But this is not the case always.

Instruction Execution steps in successive clock cycles



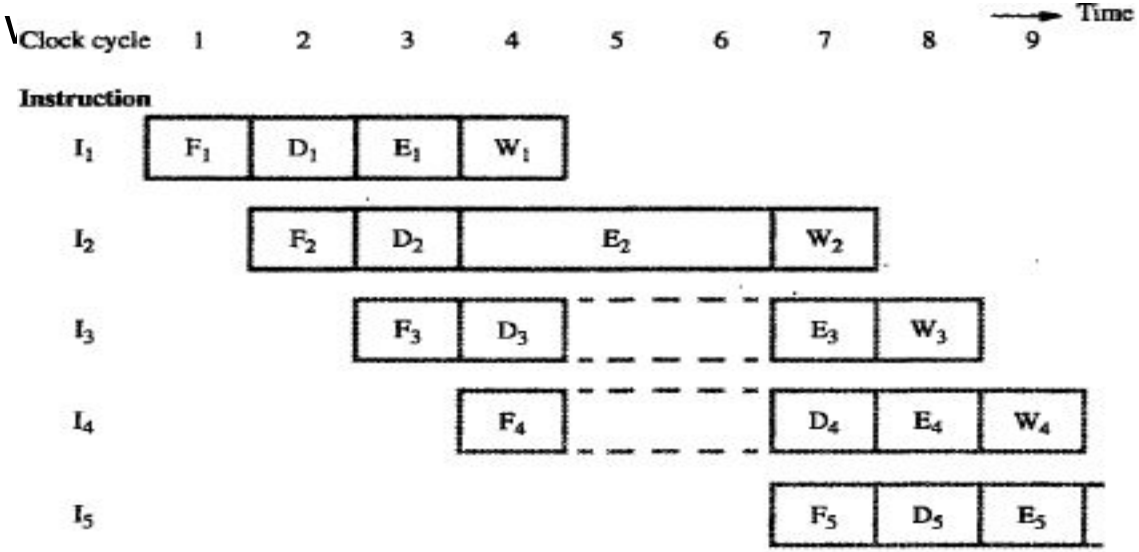
Pipeline stall caused by cache miss in F2



Clock cycle	1	2	3	4	5	6	7	8	9 → Time
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

- Consider the scenario, where one of the pipeline stage may require more clock cycle than the other.
- For example, consider the following figure where instruction I2 takes 3 cycles to completes its execution(cycle 4,5,6)
- In cycle 5,6 the W stage must wait until it has no data to work with.



Effect of an execution operation taking more than one clock cycle.

The Major Hurdle of Pipelining—Pipeline Hazards

- These situations are called *hazards,* that prevent the next instruction in the instruction stream from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

1. *Structural hazards*

- arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

2. *Data hazards*

- arise when an instruction depends on the results of a previous instruction

3.*Control/Instruction hazards*

- The pipeline may be stalled due to unavailability of the instructions due to cache miss and instruction need to be fetched from main memory.
- arise from the pipelining of branches and other instructions that change the PC.

Structural Hazards

- If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard.
- When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available. Such stalls will increase the CPI from its usual ideal value of 1.
- Some pipelined processors have shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data memory reference, it will conflict with the instruction reference for a later instruction
- To resolve this hazard, we stall the pipeline for 1 clock cycle when the data memory access occurs. A stall is commonly called a pipeline bubble or just bubble

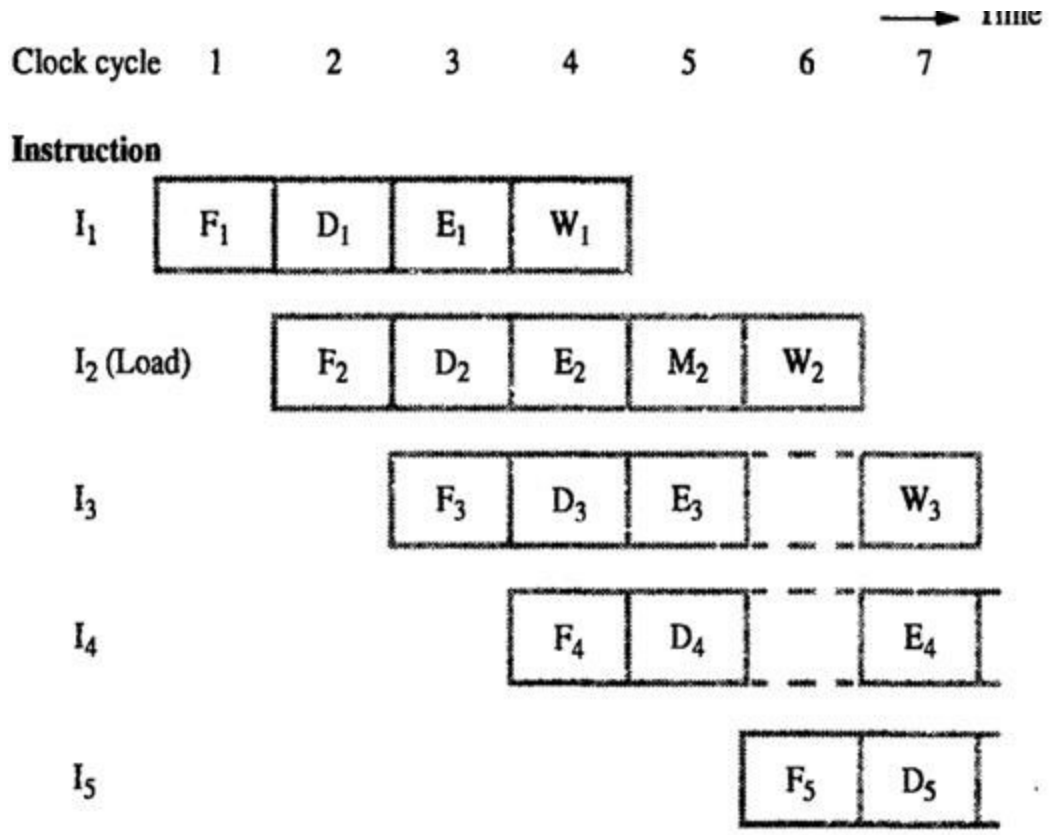


Figure 8.5 Effect of a Load instruction on pipeline timing.

Load x(r1),r2

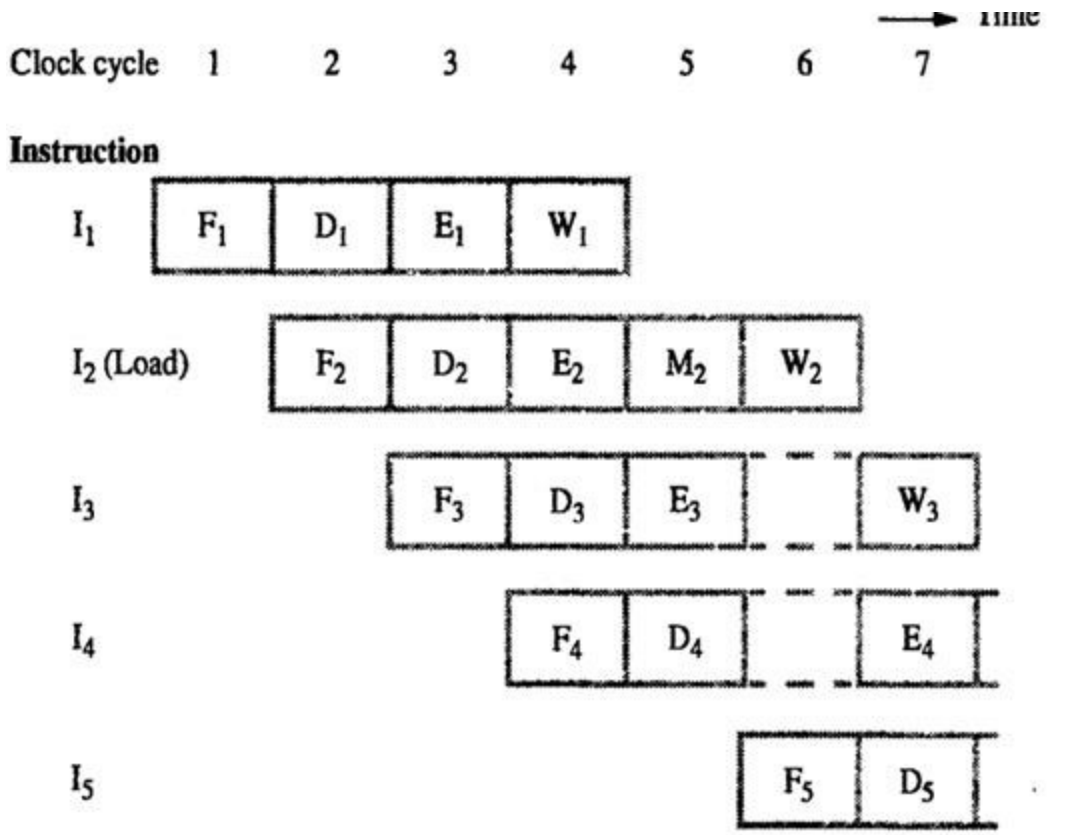


Figure 8.5 Effect of a Load instruction on pipeline timing.

Data Hazards

- *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- Consider the pipelined execution of these instructions:
- ADD R2,R3,R1
- SUB R4,R1,R5

- DADD instruction writes the value of R1 in the WB pipe stage, but the DSUB instruction reads the value during its ID stage. This problem is called a *data hazard*

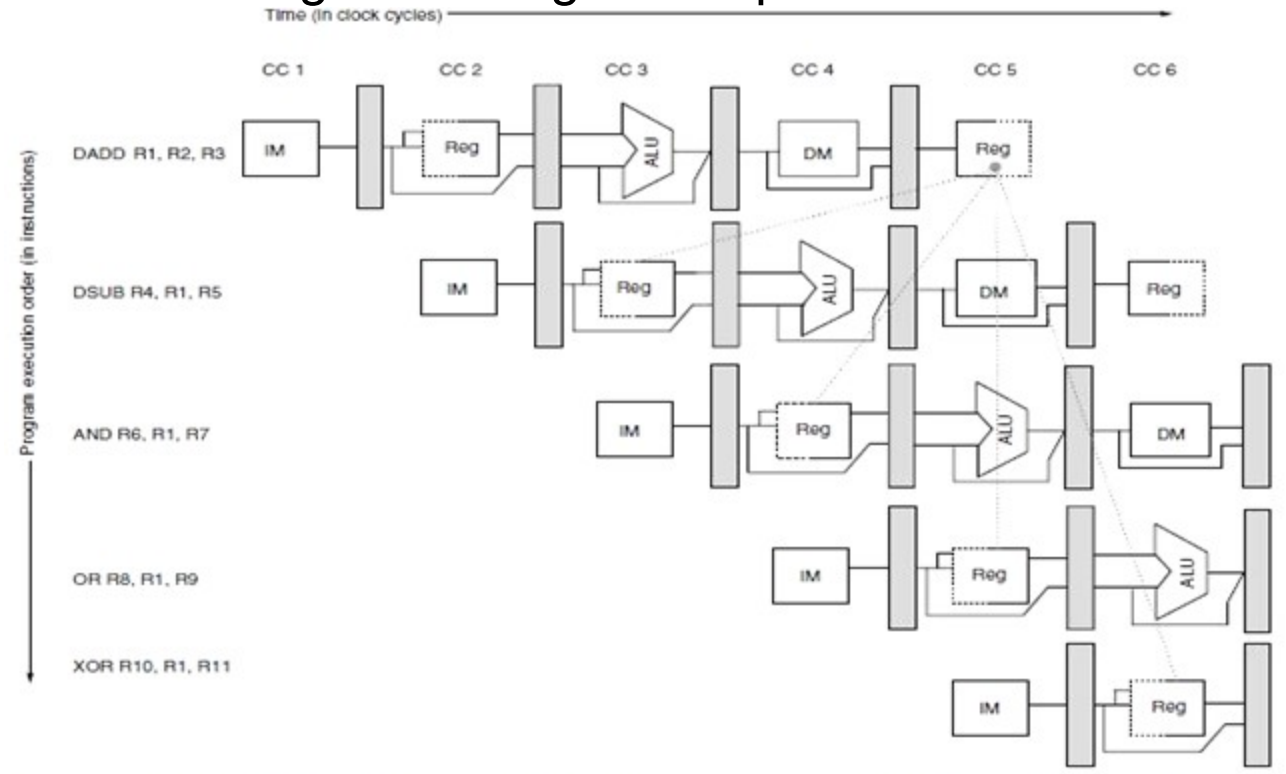
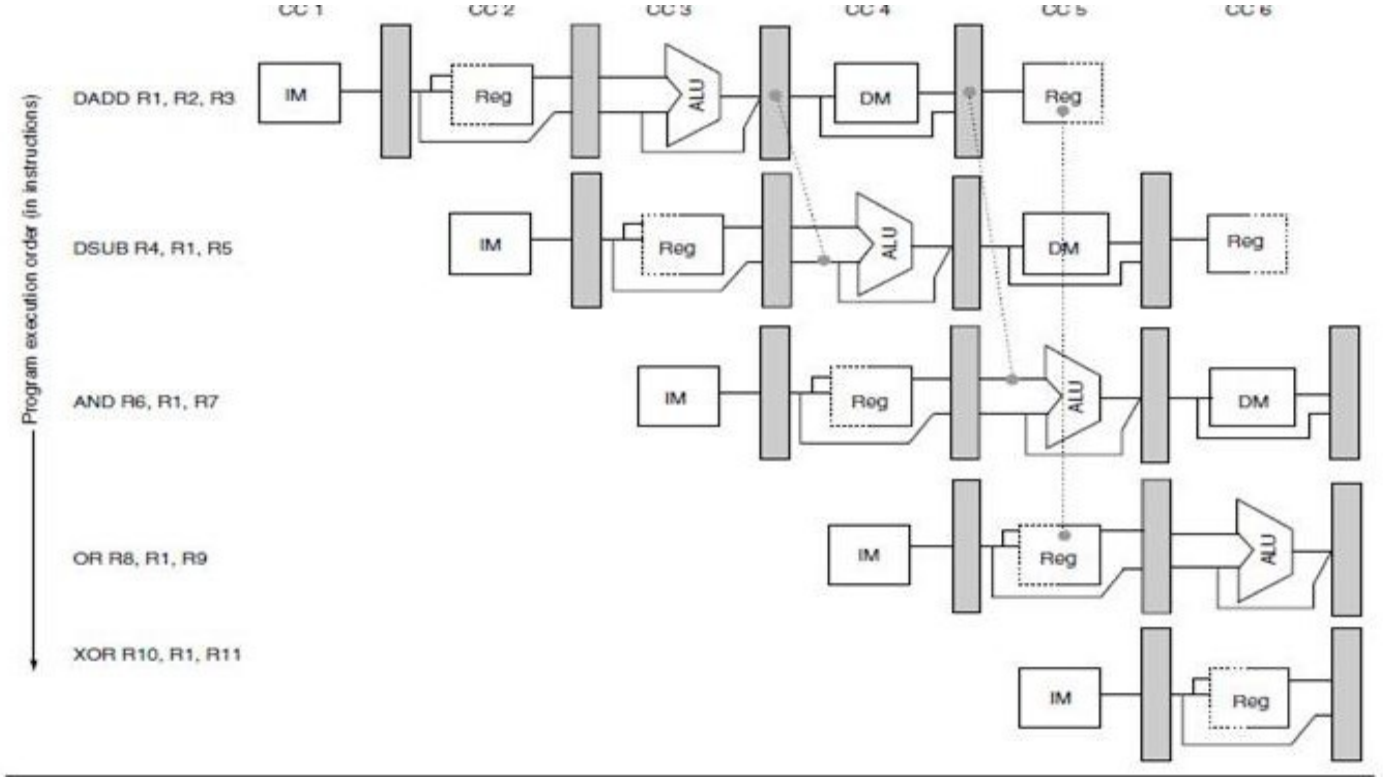


Figure A.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Minimizing Data Hazard Stalls by Forwarding

- *Forwarding* (also called *bypassing* and sometimes *short-circuiting*



Data Hazards Requiring Stalls

- Consider the following sequence of instructions:
- LD 0(R2),R1
- DSUB R4,R1,R5
- AND R6,R1,R7
- OR R8,R1,R9

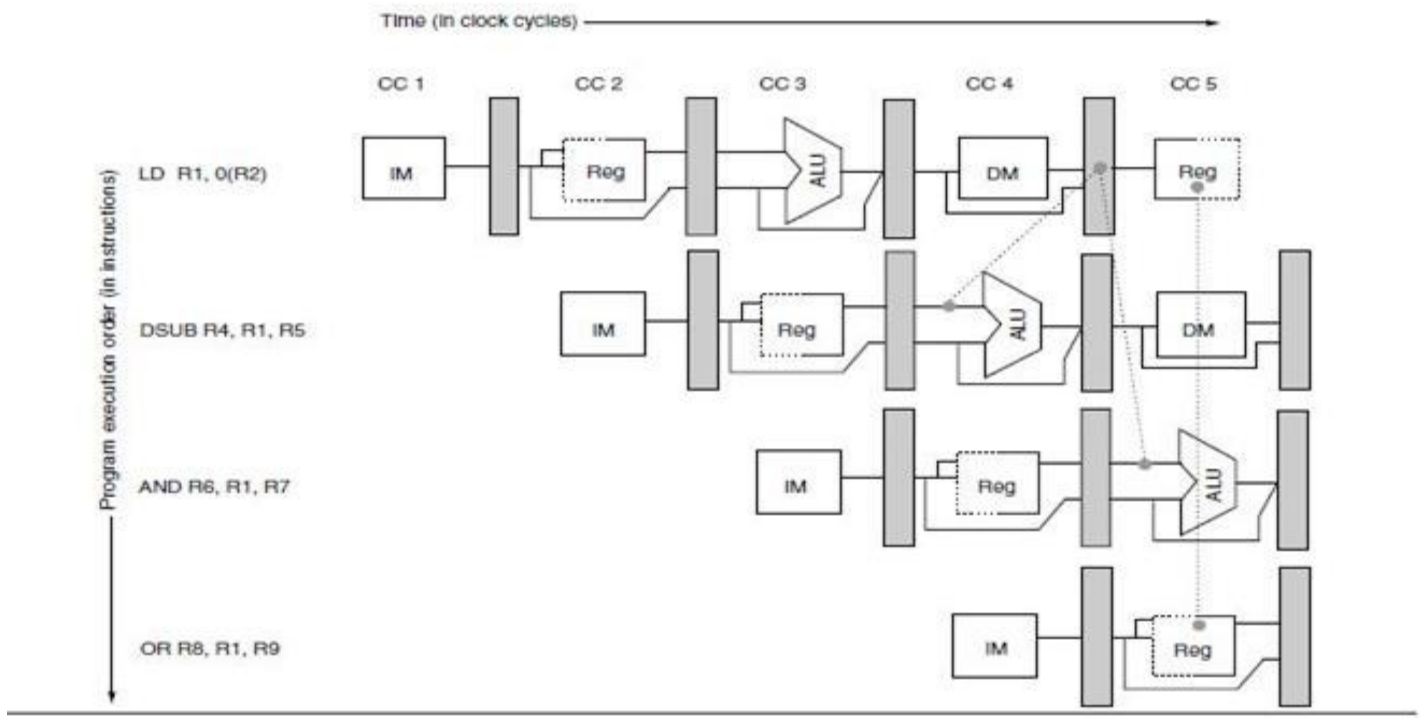


Figure A.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in "negative time."

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

Figure A.10 In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Instruction Hazards

- Whenever the stream of instructions supplied by the instruction fetch unit is interrupted, the pipeline stalls.

Unconditional Branches

- If Sequence of instruction being executed in two stages pipeline instruction I1 to I3 are stored at consecutive memory address and instruction I2 is a branch instruction.
- If the branch is taken then the PC value is not known till the end of I2.
- Next third instructions are fetched even though they are not required
- Hence they have to be flushed after branch is taken and new set of instruction have to be fetched from the branch address

Unconditional Branches

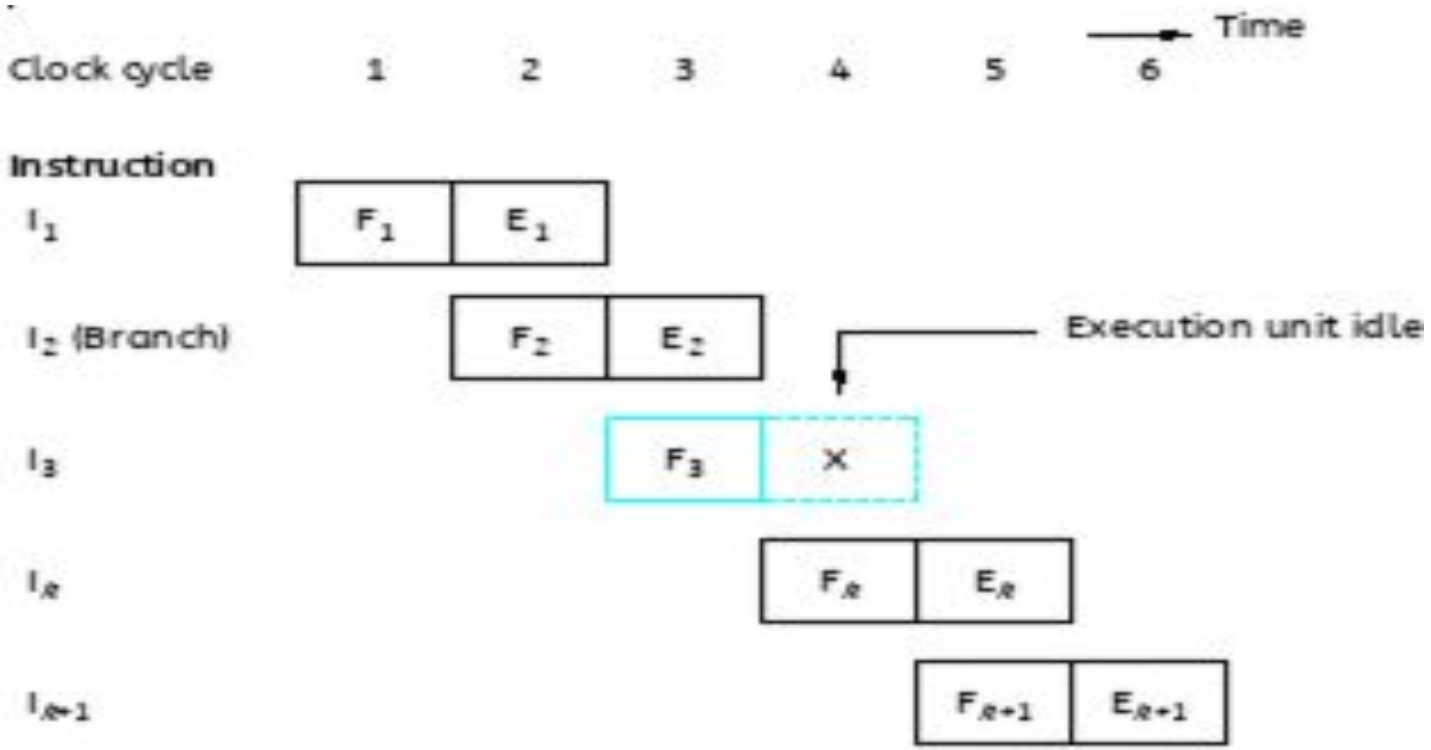


Figure 8.8. An idle cycle caused by a branch instruction.

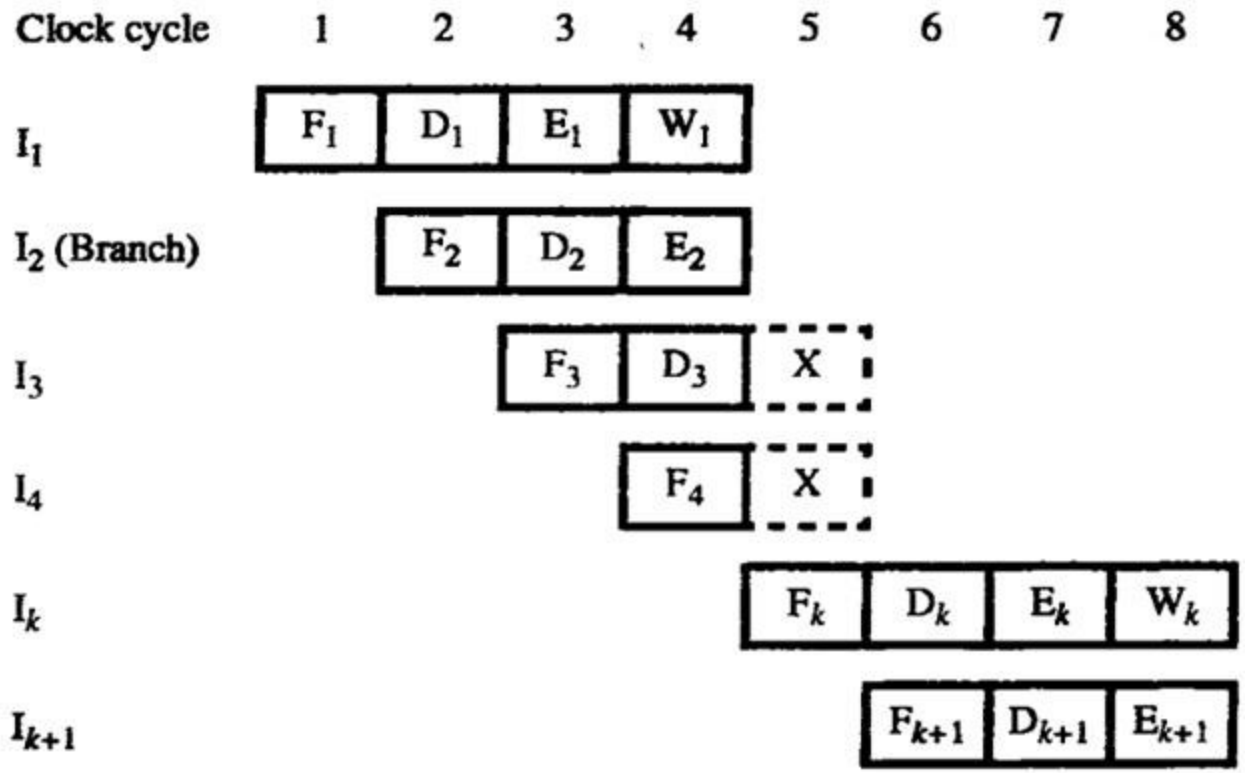
Branch Timing

Branch penalty

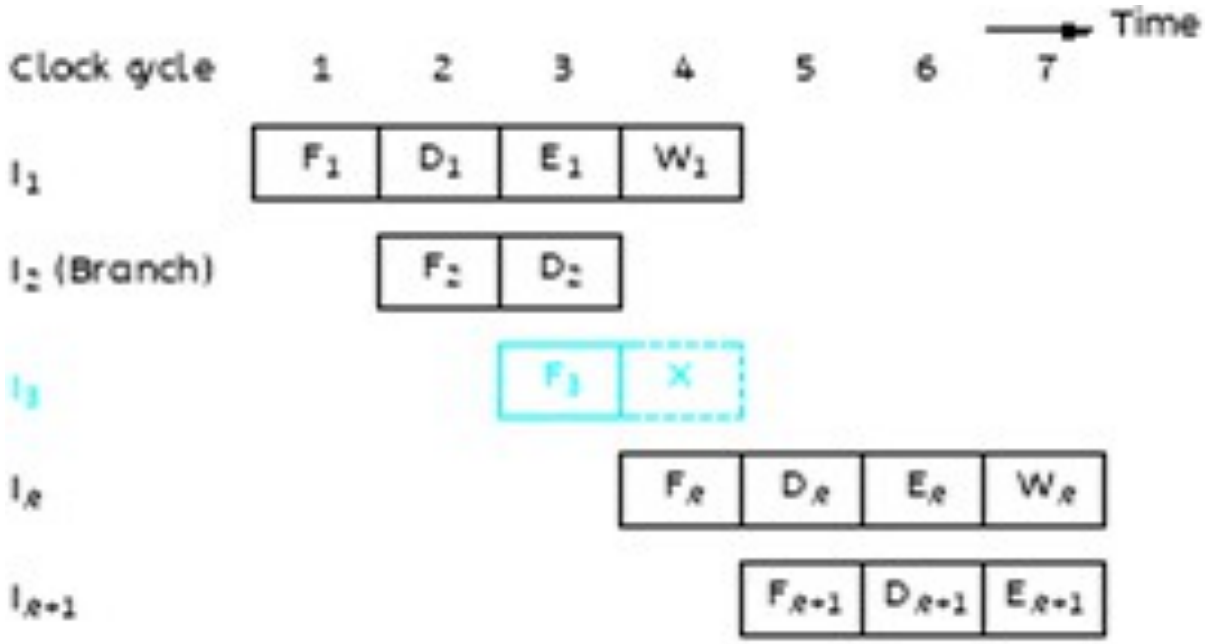
- The time lost as the result of branch instruction

Reducing the penalty

- The branch penalties can be reduced by proper scheduling using compiler techniques.
- For longer pipeline, the branch penalty may be much higher
- Reducing the branch penalty requires branch target address to be computed earlier in the pipeline
- Instruction fetch unit must have dedicated hardware to identify a branch instruction and compute branch target address as quickly as possible after an instruction is fetched



(a) Branch address computed in Execute stage

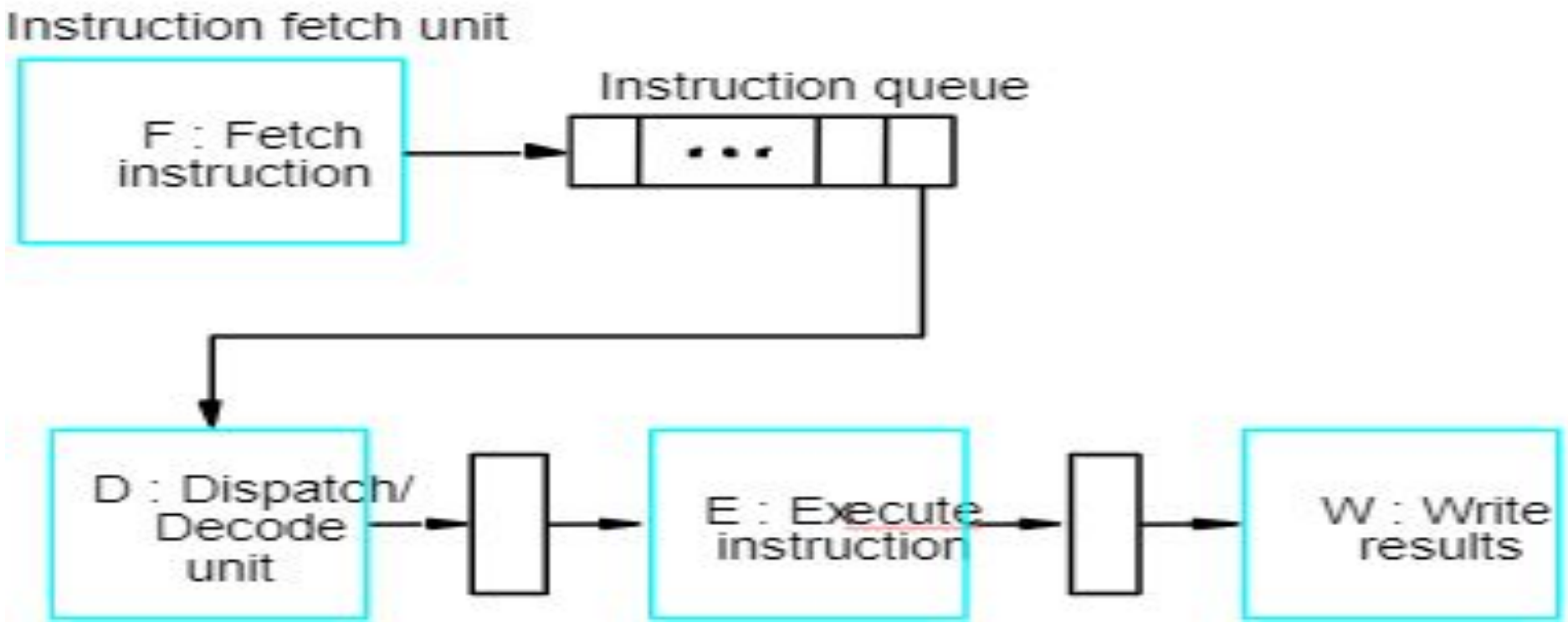


(b) Branch address computed in Decode stage

Instruction Queue and Prefetching

- Either a cache miss or a branch instruction may stall the pipeline for one or more clock cycle.
- To reduce the interruption many processor uses the instruction fetch unit which fetches instruction and put them in a queue before it is needed.
- Dispatch unit-Takes instruction from the front of the queue and sends them to the execution unit, it also perform the decoding operation
- Fetch unit keeps the instruction queue filled at all times.
- If there is delay in fetching the instruction, the dispatch unit continues to issue the instruction from the instruction queue

Instruction Queue and Prefetching



Conditional Branches

- A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
- The decision to branch cannot be made until the execution of that instruction has been completed.

Delayed Branch

- The location following the branch instruction is branch delay slot.
- The delayed branch technique can minimize the penalty arise due to conditional branch instruction
- The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken.
- The objective is to place useful instructions in these slots.
- The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions.

Delayed Branch

```
LOOP    Shift_left    R1
        Decrement     R2
        Branch=0      LOOP
NEXT    Add           R1,R3
```

(a) Original program loop

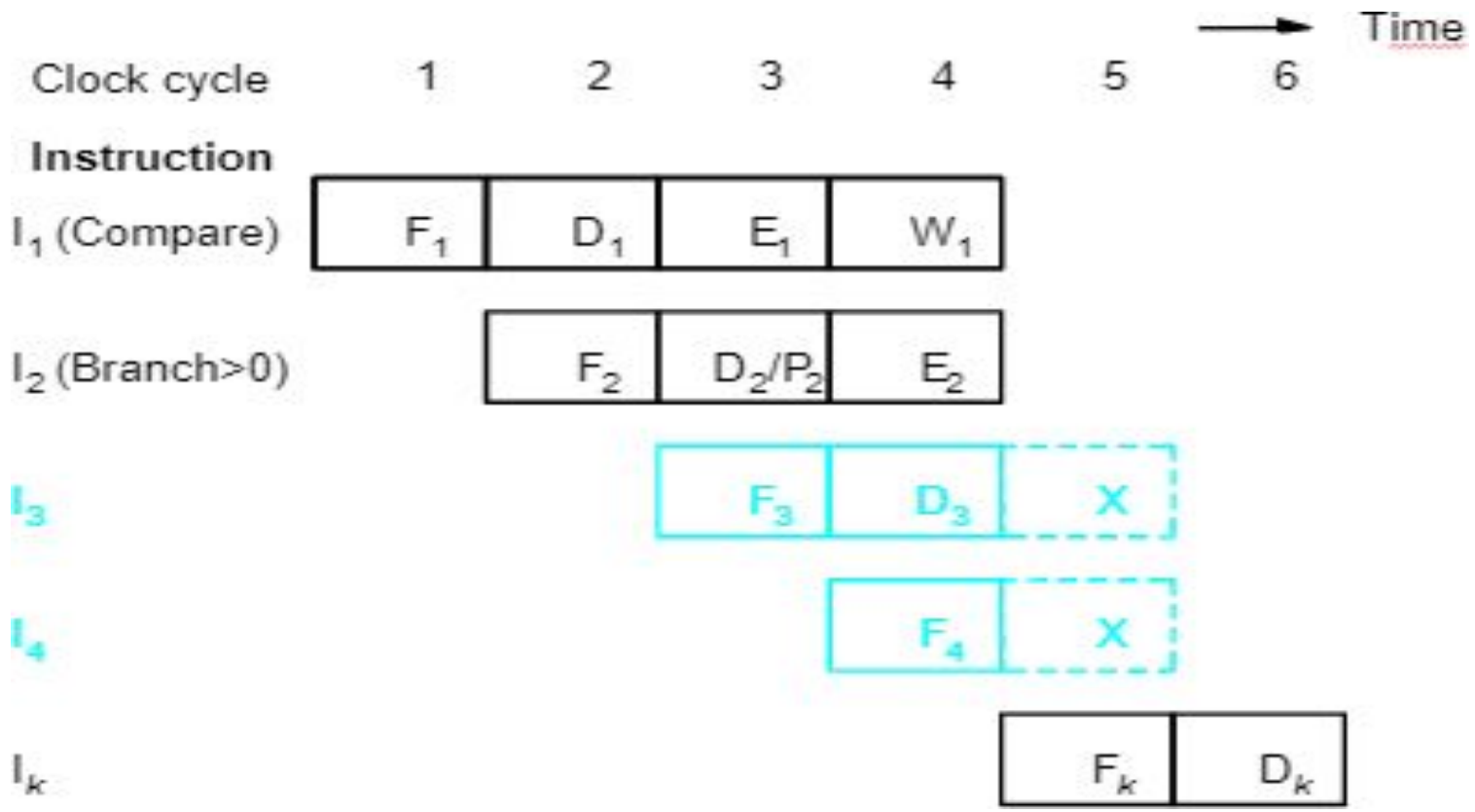
```
LOOP    Decrement     R2
        Branch=0      LOOP
        Shift_left    R1
NEXT    Add           R1,R3
```

(b) Reordered instructions

Branch Prediction

- To predict whether or not a particular branch will be taken.
- Simplest form: assume branch will not take place and continue to fetch instructions in sequential address order.
- Until the branch is evaluated, instruction execution along the predicted path must be done on a speculative basis.
- Speculative execution: instructions are executed before the processor is certain that they are in the correct execution sequence.
- Need to be careful so that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.

Incorrect Predicted Branch



Types of branch prediction

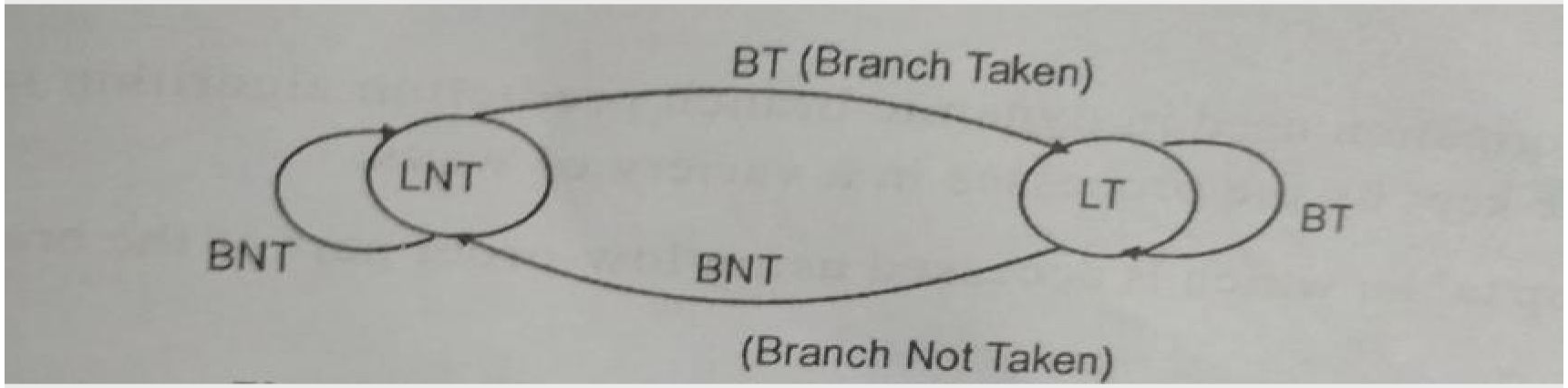
- Static Prediction
- Dynamic branch Prediction

Prediction is carried out by compiler and it is static because the prediction is already known before the program is executed.

Dynamic prediction in which the prediction decision may change depending on the execution history.

Branch Prediction Algorithm

- If the branch taken recently,the next time if the same branch is executed,it is likely that the branch is taken
- State 1: LT : Branch is likely to be taken
- State 2: LNT : Branch is likely not to be taken
- 1.If the branch is taken,the machine moves to LT. otherwise it remains in state LNT.
- 2.The branch is predicted as taken if the corresponding state machine is in state LT, otherwise it is predicted as not taken

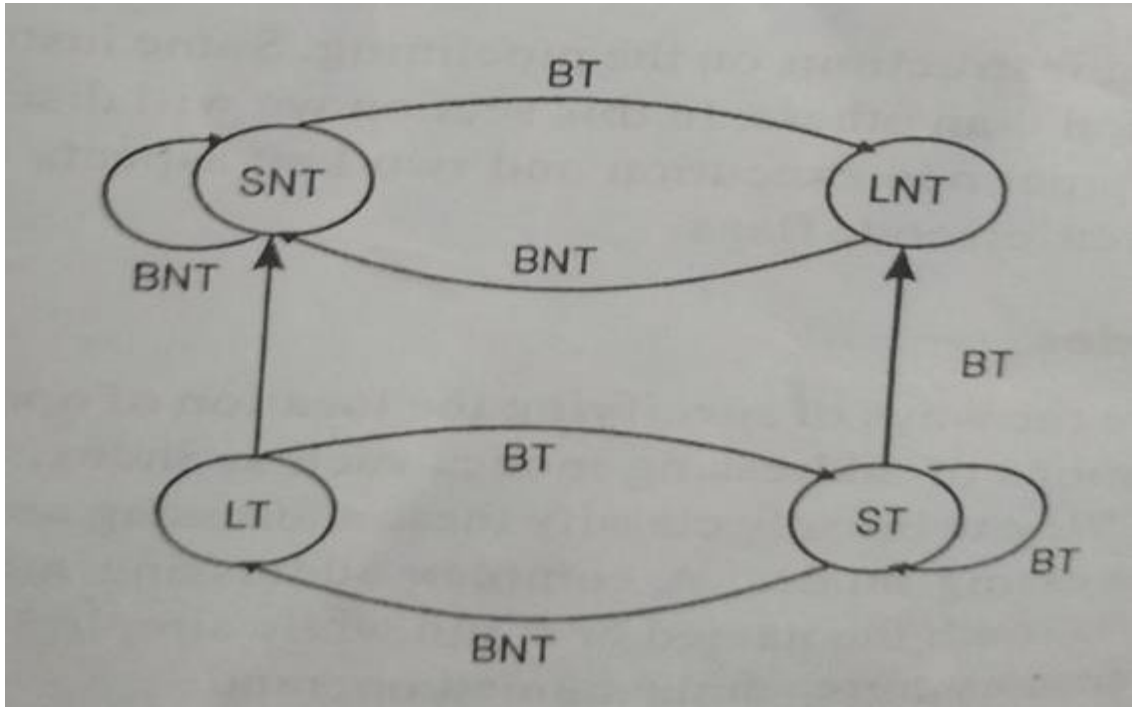


4 State Algorithm

- ST-Strongly likely to be taken
  - LT-Likely to be taken
  - LNT-Likely not to be taken
  - SNT-Strongly not to be taken
- Step 1: Assume that the algorithm is initially set to LNT
- Step 2: If the branch is actually taken changes to ST, otherwise it is changed to SNT.
- Step 3: when the branch instruction is encountered, the branch will taken if the state is either LT or ST and begins to fetch instruction at branch target address, otherwise it continues to fetch the instruction in sequential manner

4 State Algorithm

- When in state SNT, the instruction fetch unit predicts that the branch will not be taken
- If the branch is actually taken, that is if the prediction is incorrect, the state changes to LNT



Influence on Instruction Sets

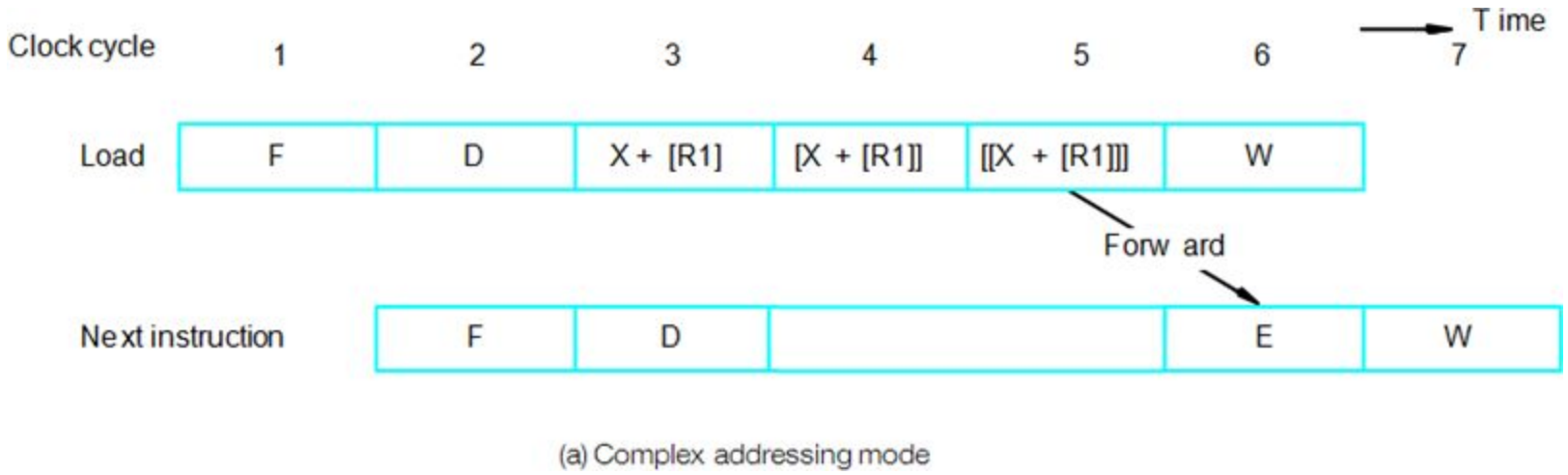
- Some instructions are much better suited to pipeline execution than others.
- Addressing modes
- Conditional code flags

Addressing Modes

- Addressing modes include simple ones and complex ones.
- In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline:
  - Side effects
  - The extent to which complex addressing modes cause the pipeline to stall
  - Whether a given mode is likely to be used by compilers

Complex Addressing Mode

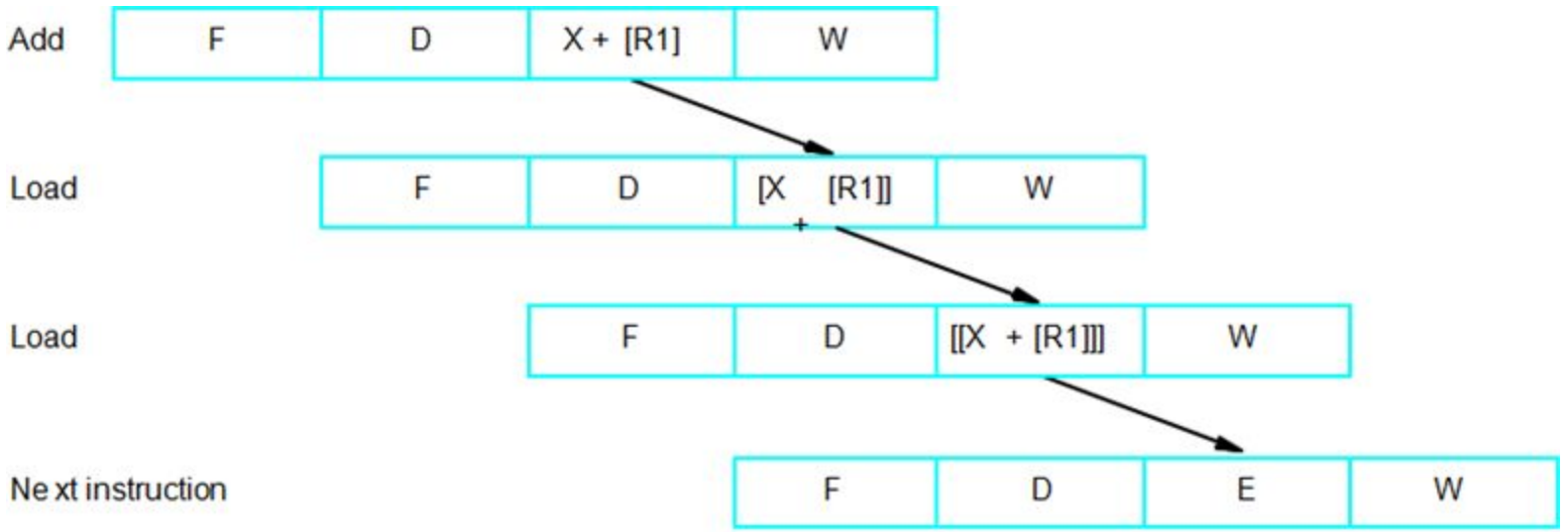
Load (X(R1)), R2



(a) Complex addressing mode

Simple Addressing Mode

Add #X, R1,
R2 Load
(R2), R2 Load
(R2), R2



(b) Simple addressing mode

- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- **Advantage:** reducing the number of instructions / program space
- **Disadvantage:** cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- **Conclusion**: complex addressing modes are not suitable for pipelined execution.

- Good addressing modes should have:
  - Access to an operand does not require more than one access to the memory
  - Only load and store instruction access memory operands
  - The addressing modes used do not have side effects
- Register, register indirect, index

Conditional Codes

- If an optimizing compiler attempts to reorder instruction to avoid stalling the pipeline when branches or data dependencies between successive instructions occur, it must ensure that reordering does not cause a change in the outcome of a computation.
- The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

```
Add          R1,R2
Compare      R3,R4
Branch=0     . . .
```

a) A program fragment

```
Compare      R3,R4
Add          R1,R2
Branch=0     . . .
```

b) Instructions reordered

Instruction reordering

Conditional Codes

Two conclusion:

- To provide flexibility in reordering instructions, the condition-code flags should be affected by as few instruction as possible.

- The compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not.

Thank You