

18CSC203J – COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-III

Course Outcome

CLR-3: Understand the concepts of Pipelining and basic processing units

CLO-3 : Analyze the detailed operation of Basic Processing units and the performance of Pipelining

Topics Covered

- Fundamental concepts of basic processing unit
- Performing ALU operation
- Execution of complete instruction, Branch instruction
- Multiple bus organization
- Hardwired control,
- Generation of control signals
- Micro-programmed control, Microinstruction
- Micro-program Sequencing
- Micro instruction with Next address field
- Basic concepts of pipelining
- Pipeline Performance
- Pipeline Hazards-Data hazards, Methods to overcome Data hazards
- Instruction Hazards
- Hazards on conditional and Unconditional Branching
- Control hazards
- Influence of hazards on instruction sets

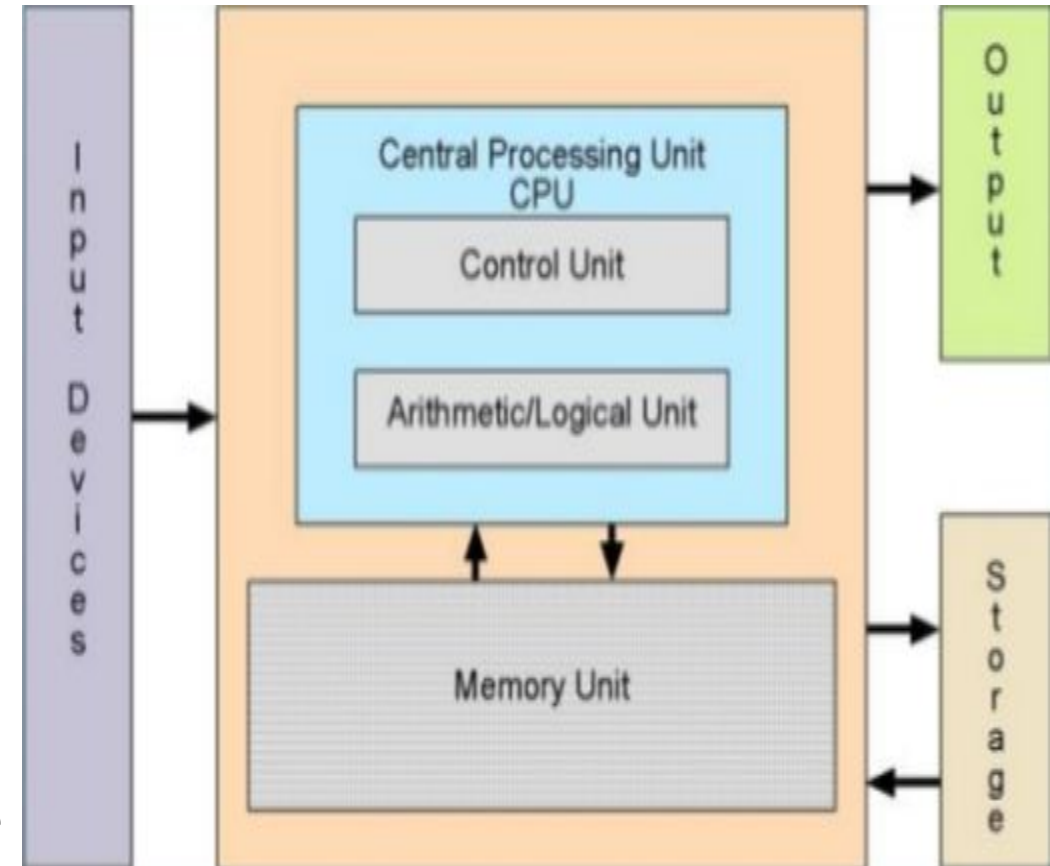
PROCESSING UNIT

FUNCTIONS OF CPU:

- CPU carries out all forms of data processing tasks.
- It saves information, intermediate results and instructions.
- CPU monitors the functionality of all computer components.

COMPONENTS OF CPU:

- Memory or storage unit:** The memory unit stores all the instructions and data. This unit provides data to other units of the computer if necessary. Also known as main memory, or **RAM** (Random Access Memory).
- Control unit:** This unit monitors all computing processes but does not execute actual data processing.
- Arithmetic Logic Unit (ALU):** This does all the calculations and makes the decisions.



FUNDAMENTAL CONCEPTS OF BASIC PROCESSING UNIT

- Processor fetches one instruction at a time and perform the specified operation.
- Instructions are fetches from successive memory locations except for branch/ jump instruction.
- The address of the next instruction to be executed is tracked by the Program Counter (PC) register.
- Instruction Register (IR) contains instruction that is currently executed.
- Instruction execution happens in three phases:
 - ✓ Fetch: Fetch the instruction from the specified memory
 - ✓ Decode: Determined the opcode and the operands
 - ✓ Execute: Run the instruction

EXECUTING AN INSTRUCTION

- Fetch the contents of memory location pointed by the PC. The contents of this memory location is loaded to the IR-**Fetch phase**

$IR \leftarrow [PC]$

- Increment the PC by 4 (assume the word size as 4)

$PC \leftarrow PC + 4$

- Carry out the actions specified by the instruction in the IR-**Execution phase**
- Datapath:** collection of functional units such as ALU or multipliers that perform data processing operations.
- MDR:** Two inputs and two outputs since data can be loaded from memory or processor bus.
- MAR:** Input line is connected to internal bus and output line to external bus
- Control lines:** connected to instruction decoder and control logic block to issue control signals
- R0-R(n-1):** General Purpose registers whose numbers vary between processors.
- TEMP, Y and Z:** temporary registers used by the processor during instruction execution

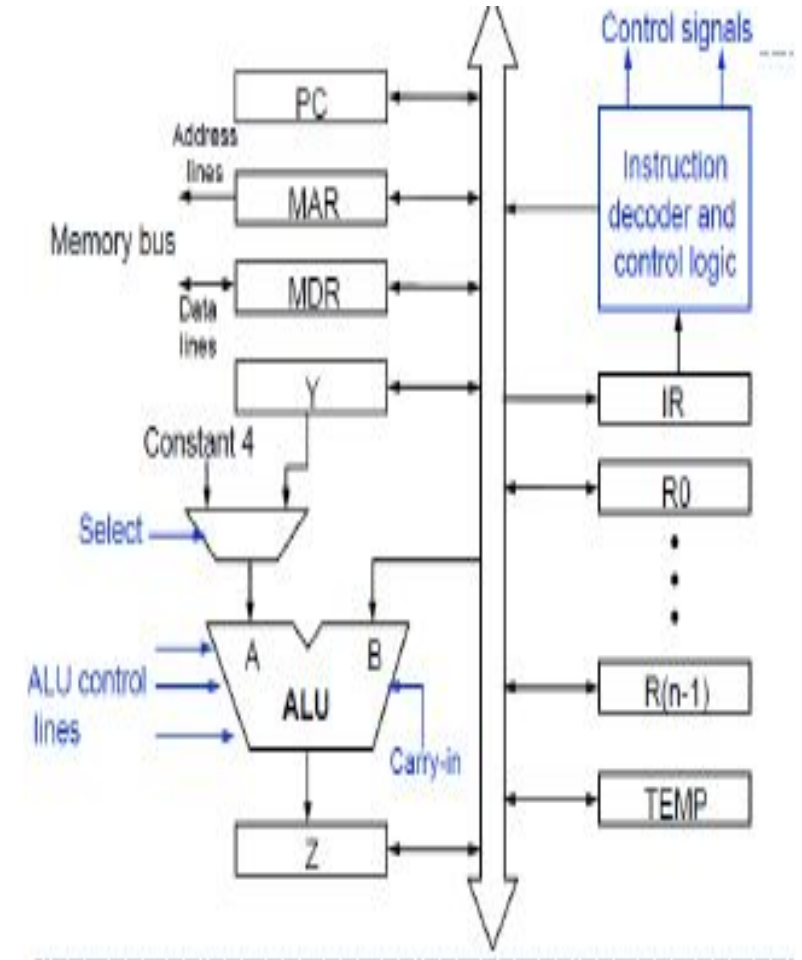


Fig : Single bus organization of datapath

- Transfer a word of data from one processor register to another or to the ALU
- Perform Arithmetic and Logic operation and store the result in processor register
- Fetch the contents of a given memory location and load the, into processor register.
- Store a word of data from a processor register into a given memory location

REGISTER TRANSFERS

- For each register two control signals are used :
 - ✓ To place the contents of that register on the bus
 - ✓ To load the data on the bus into register
- The input and output of register are connected to the bus through switches controlled by the signals Rin and Rout.

PERFORMING AN ARITHMETIC OR LOGIC 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.
- The sequence of operations to add contents of R1 and R2 and store it in R3:

1. R1out, Yin
2. R2out, SelectY, Add, Zin
3. Zout, R3in

FETCHING WORD FROM MEMORY

- Address will be loaded into MAR from PC
- Read operation will be issued in the bus and the data will be loaded into MDR.
- Wait for Memory Function Completed (WMFC):** This signal is issued to make the processor to wait for the reply from the memory (MFC).

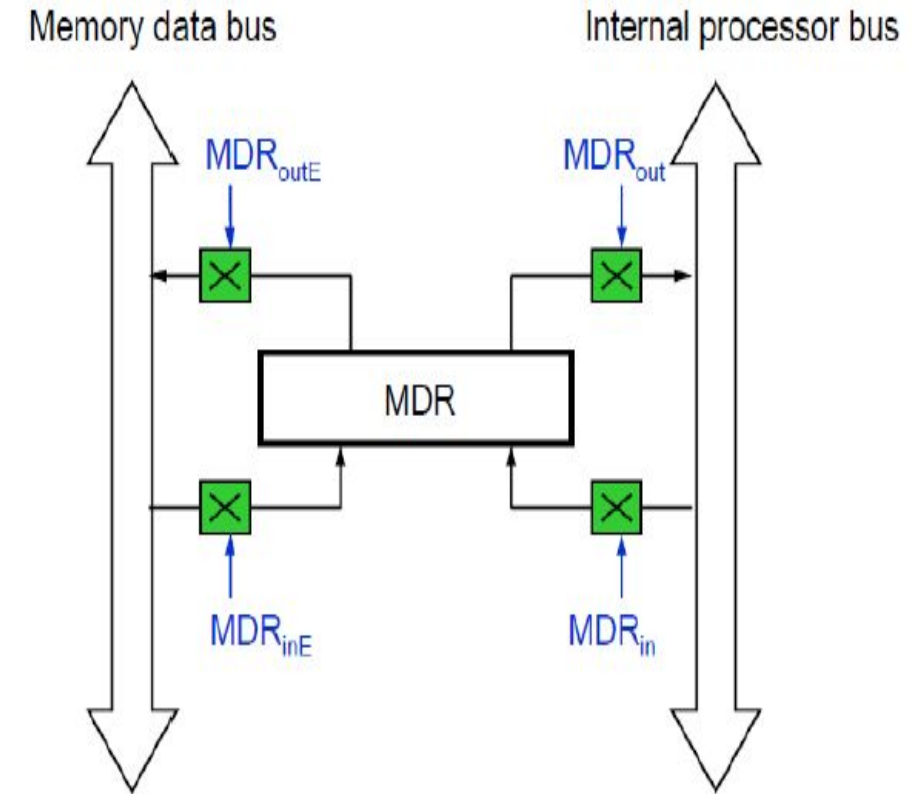


Fig: Fetching word form Memory

Move (R1), R2

1. R1_{out}, MARin, Read
2. MDRinE, WMFC
3. MDRout, R2in

EXECUTION OF COMPLETE INSTRUCTION

- Initiate instruction fetch by loading contents of PC to MAR.
- Send Read request to memory.


Incrementing PC

- Contents of PC is loaded to register B.
- Set select signal to 4, which make multiplexer to select const 4, which is then added to B. The result is available in Z.
- The content of Z is loaded to PC.

Load IR

- The word is fetched from the memory and loaded into IR
- Decoding: Instruction decoding circuit interprets contents of IR.
- Execution: Control circuitry is activated to issue relevant control signals.
- Load MAR with contents of R3. Initiate memory read. Transfer R1 to Y.

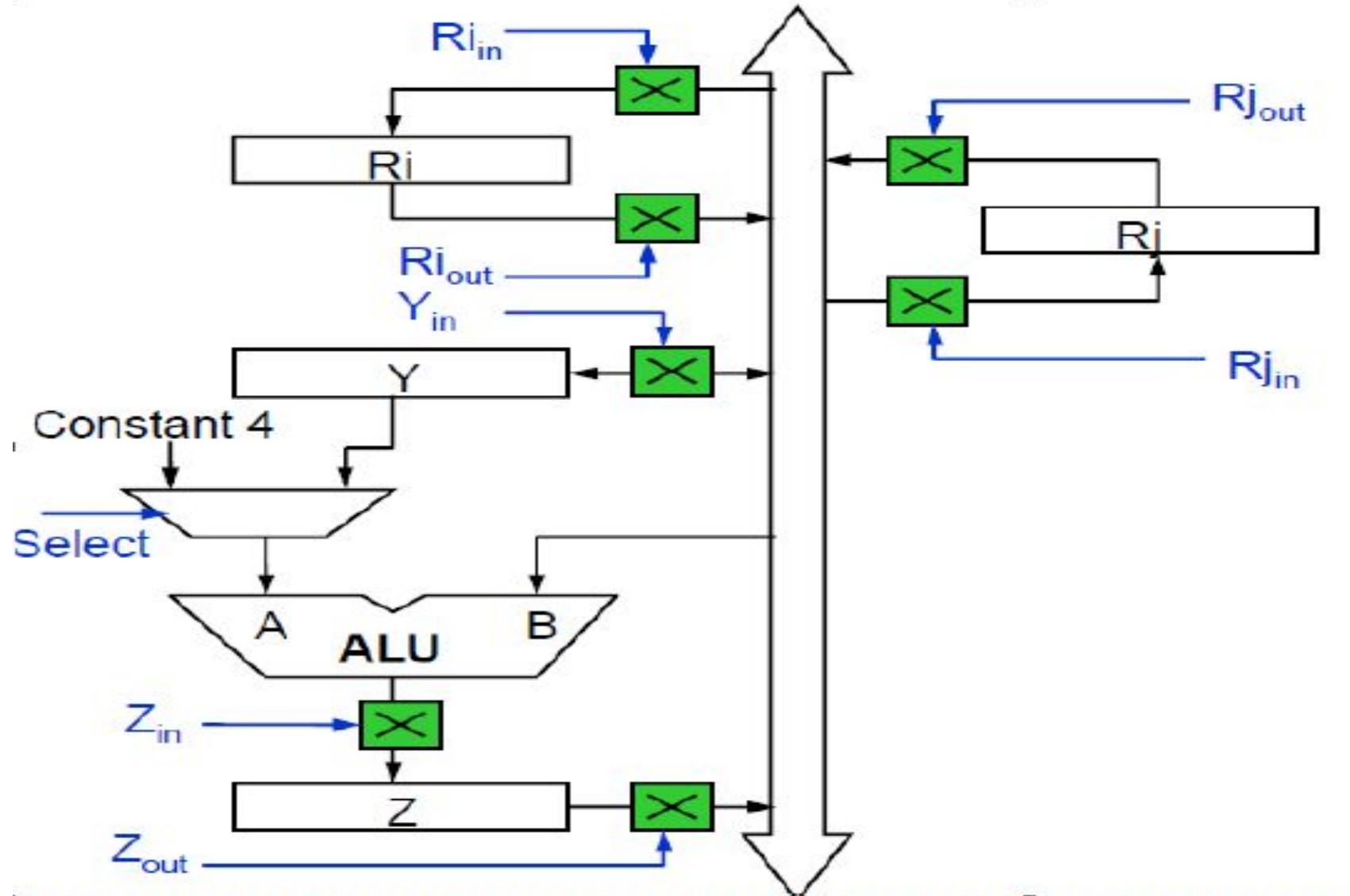
ADD (R3), R1



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

- Perform addition on Y and MDR after selecting Y.
- Load the result from Z to R1.

SIGNAL DIAGRAM



EXECUTION OF BRANCH INSTRUCTIONS

- 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.
- Two types of branch: conditional and unconditional branch

UnConditional Branch:

- **Offset value** : branch target address –address immediately following the branch instruction.
- Offset value is obtained from IR and is available in X.
- Y contains the updated PC value. Add the offset address along with PC to find the branch target.
- Now load the calculated branch target from Z to PC.

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	$Offset\text{-}field\text{-}of\text{-}IR_{out}, Add, Z_{in}$
5	Z_{out}, PC_{in}, End

- Check status of condition codes before loading branch target into PC

Condition Codes

- The CPU maintains a set of single-bit *condition code* registers.
- These registers are set by arithmetic and logical operations and can be tested to perform conditional branches.

Examples:

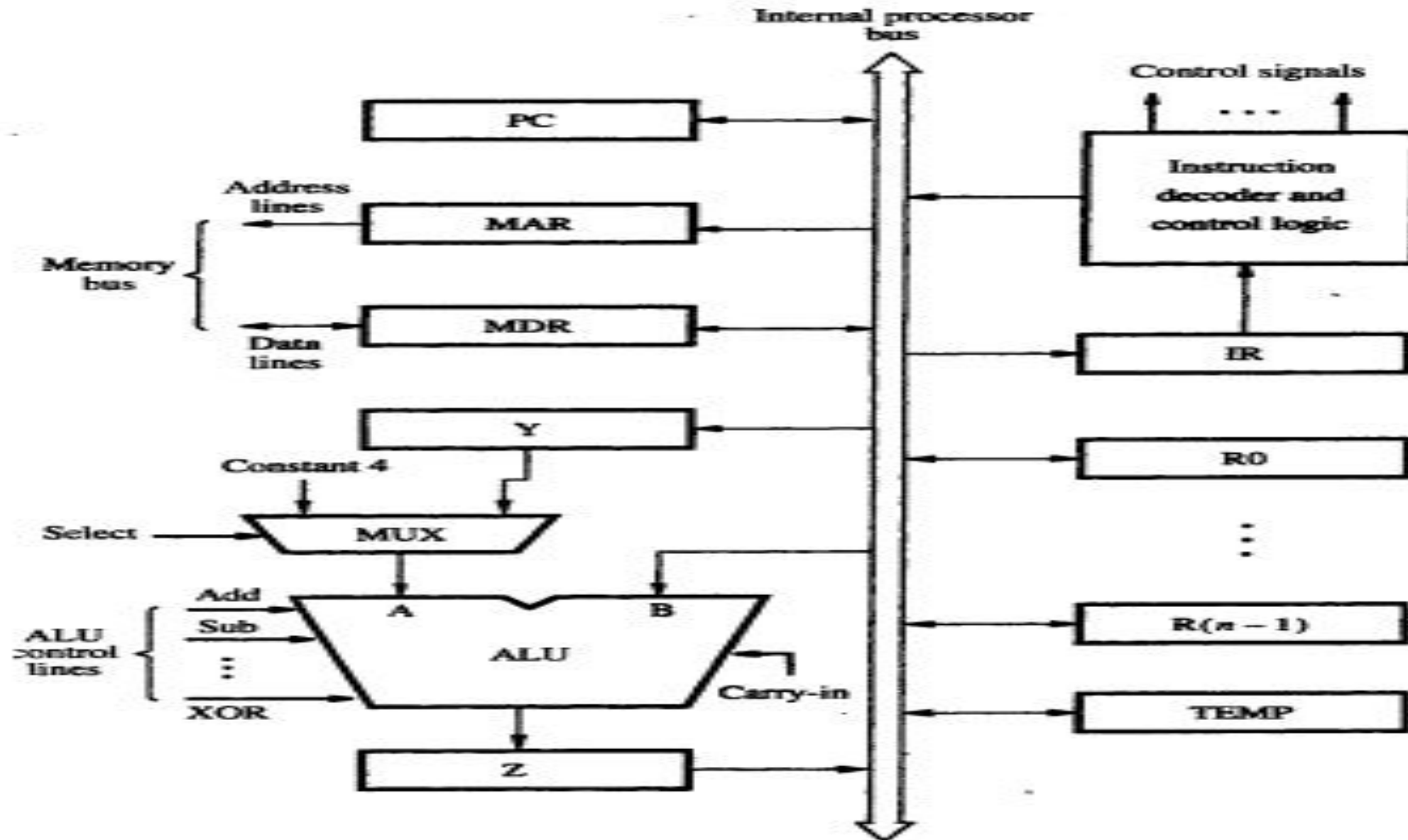
- ✓ **Carry flag.** If set, the most recent operation generated a carry out of the most significant bit. Use to detect overflow for unsigned operations.
 - ✓ **Zero flag.** The most recent operation yielded zero.
 - ✓ **Sign flag.** The most recent operation yielded a negative value.
 - ✓ **Overflow flag.** The most recent operation caused a two's complement overflow – either negative or positive.
- Offset-field-of- IR_{out} , Add, Z_{in} , If $N = 0$ then End

Branch on negative

- If $N=0$ then the processor return to step 1 after step 4.
- If $N=1$ then load a new value into PC to perform branching.

Multiple bus organization

SINGLE BUS ORGANISATION OF A DATAPATH



MULTIPLE BUS ORGANIZATION

- Single bus structure takes long time for transfer of data in a clock cycle
- To avoid the above said process ,multiple internal paths are enabled in commercial processors .
- This allows to have several transfers to take place in parallel

Multi bus

- The next slide has the three bus structure which is connecting registers and the ALU of processors
- Register file: is a single block of all general purpose registers
- Register file has three ports.
- First Two output ports are connected to different registers placed on the Bus A and Bus B.
- One more third port connected to Bus C during the same clock cycle.

Cont...

- Buses A and B used to Transfer the source operands to A and B inputs of ALU.(Arithmetic and logic operations are performed)
- The result is transferred over the Bus C.
- Three bus arrangement avoids the usage of registers Y and Z.
- Incrementer Unit: is used to increment the PC by 4 which eliminates the usage of ALU.

Cont...

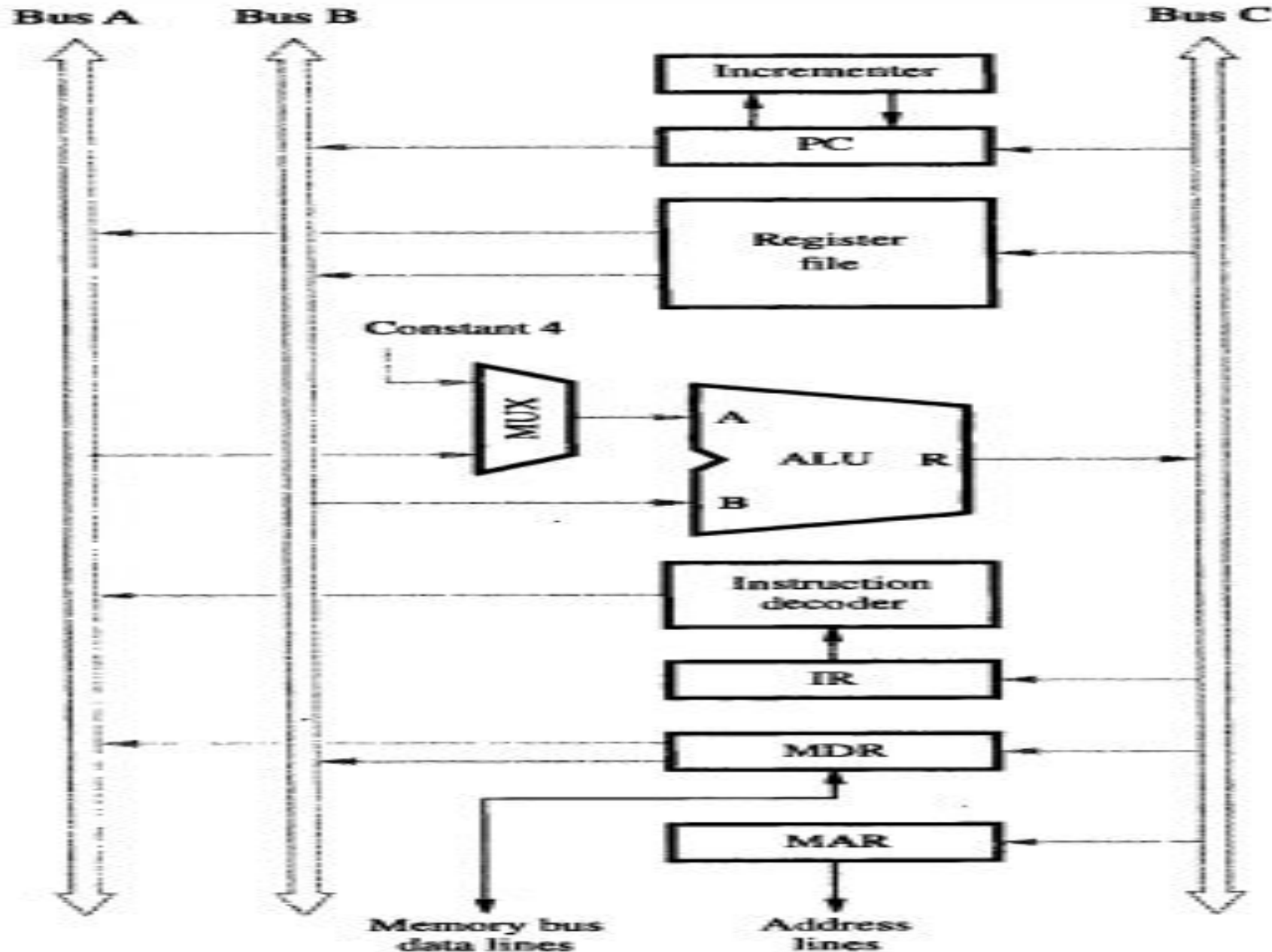
- It increments the other addresses such as the memory addresses in LoadMultiple and StoreMultiple instructions.
- Three operand instruction
- Add R4,R5,R6
- STEP1: The contents of the PC are passed through the ALU using R=B control signal and loaded into MAR for(Memory Read operation) and the PC is incremented by 4

Cont..

- Step2: The processor waits for MFC (WMFC-Wait for Memory Function completed)
- STEP3: MFC loads the data received into MDR. And again data is transferred into IR
- STEP4: Final execution phase requires only one control step

Providing more paths for data transfer results in reduction of clock cycles needed to execute an instruction.

MULTIPLE BUS ORGANISATION



Multiple-Bus Organization

- Add R4, R5, R6

Step	Action
1	PC _{out} , R=B, MAR _{in} , Read, IncPC
2	WMF C
3	MDR _{outB} , R=B, IR _{in}
4	R4 _{outA} , R5 _{outB} , SelectA, Add, R6 _{in} , End

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6,
for the three-bus organization in Figure 7.8.

Hardwired Control

Overview

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Two categories: hardwired control and microprogrammed control
- Hardwired system can operate at high speed; but with little flexibility.

7 Steps:

Cont...

- The before slide depicts the sequence of control signals
 - Every step is completed in one clock period
 - Counter is used to keep track of the control steps.
 - Control signals are determined by following information
1. Contents of the control step counter
 2. Contents of instruction register
 3. Contents of condition code flags
 4. External input signals, such as MFC and interrupt requests

Decoder/encoder block

- Is a combinational circuit that generates the required control outputs depending on the state of all its input.
- The control unit organization of the decoder and encoder block is shown in the next slide

Control Unit Organization

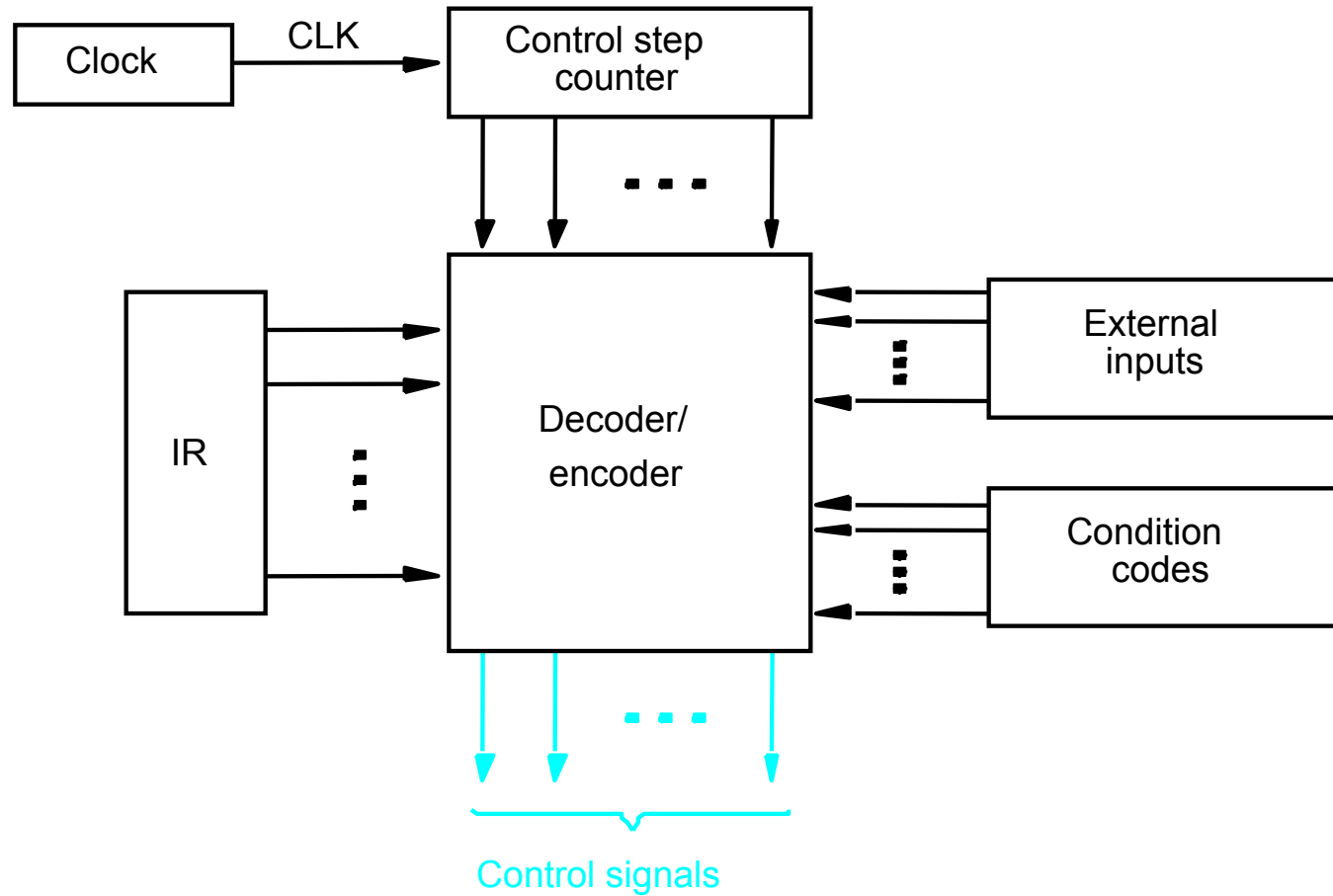


Figure 7.10. Control unit organization.

Detailed Block Description

Explanation of decoding and encoding

- The step decoder provides a separate signal line for each step or time slot in the control sequence
- Output of the instruction decoder consists of separate line for each machine instruction.
- Any instruction loaded into IR, one of the output lines INS_1 through INS_m is set to 1 and all others are set to 0.
- The input signals to the encoder are combined to generate individual control signals Y_{in} , PC_{out}

Generating Z_{in}

- $Z_{in} = T_1 + T_6 \cdot \text{ADD} + T_4 \cdot \text{BR} + \dots$
- The signal is asserted during time slot T1 for all instruction, T6 for an ADD instruction, T4 for unconditional branch instruction

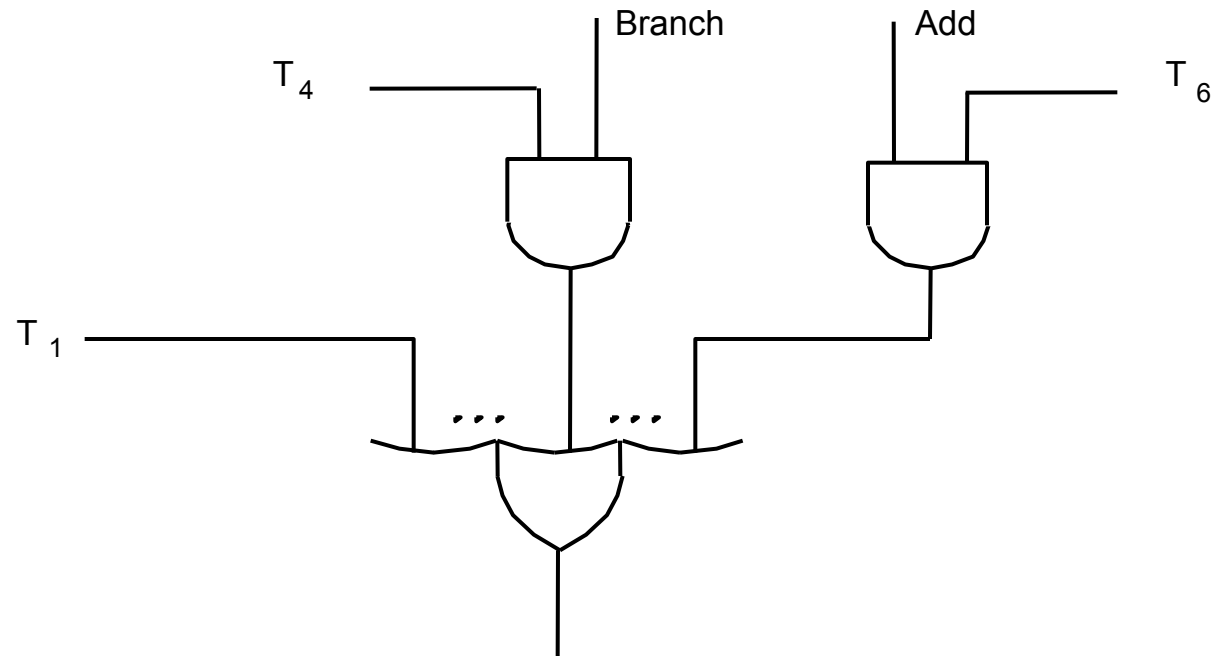


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- $\text{End} = T_7 \cdot \text{ADD} + T_5 \cdot \text{BR} + (T_5 \cdot N + T_4 \cdot N) \cdot \text{BRN} + \dots$

End signal

- End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value.
- Control signal called RUN.
- RUN set as 1 ,causes the counter to be incremented by one at the end of every clock cycle.
- RUN equal to 0 , the counter stops counting
- WMFC signal is issued to cause the processor to wait for the reply from memory

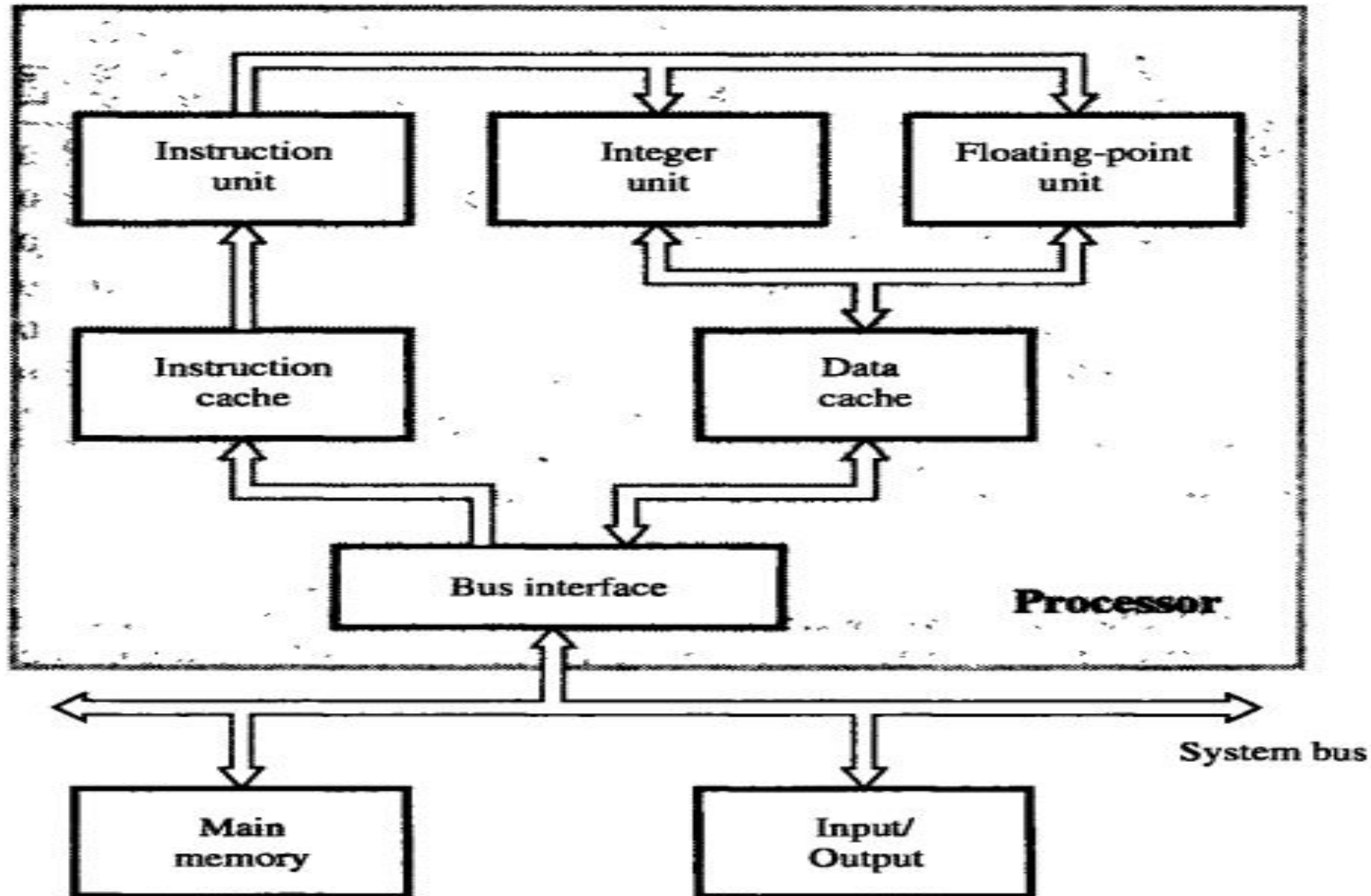
Cont...

- These control unit organization viewed as state machine(one state to another in every clock cycle)
- Output of the state machines are control signals.
- Sequence of operations carried out by machine is determined by the wiring of logic elements,hence the name “hardwired”.
- Controller operates at high speed

Complete Processor

- This structure has instruction unit that fetches from instruction cache or from the main memory.
- Processing unit to handle integer and floating point data.
- Data cache is introduced between these units and main memory
- Separate cache for instruction and data
- Processor is connected to system bus.
- Bus interface is used to connect to rest of memory and input output unit.

Complete Processor



Microprogrammed Control

A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Microprogrammed Control Unit

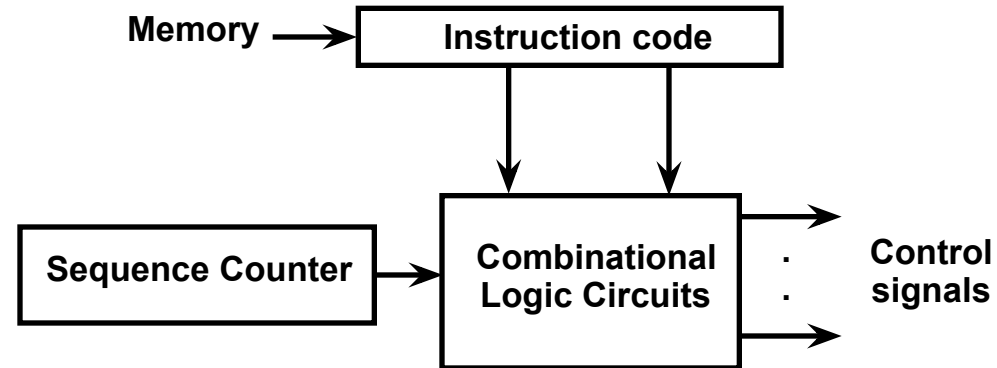
- Control signals
 - Group of bits used to select paths in multiplexers, decoders, arithmetic logic units
- Control variables
 - Binary variables specify microoperations
 - Certain microoperations initiated while others idle
- Control word
 - String of 1's and 0's represent control variables

Microprogrammed Control Unit

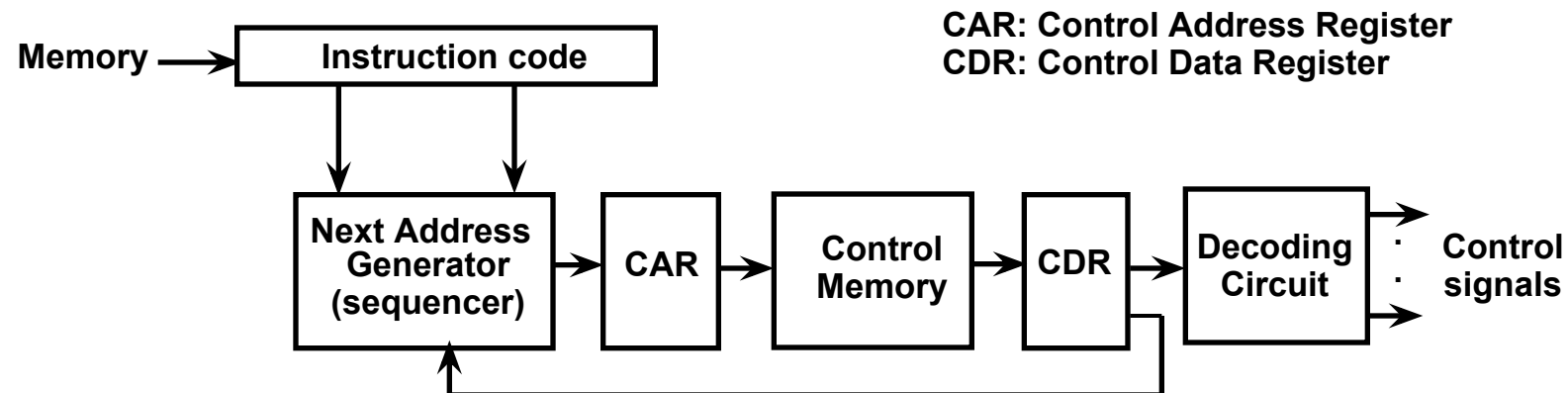
- Control memory
 - Memory contains control words
- Microinstructions
 - Control words stored in control memory
 - Specify control signals for execution of microoperations
- Microprogram
 - Sequence of microinstructions

Control Unit Implementation

- Hardwired

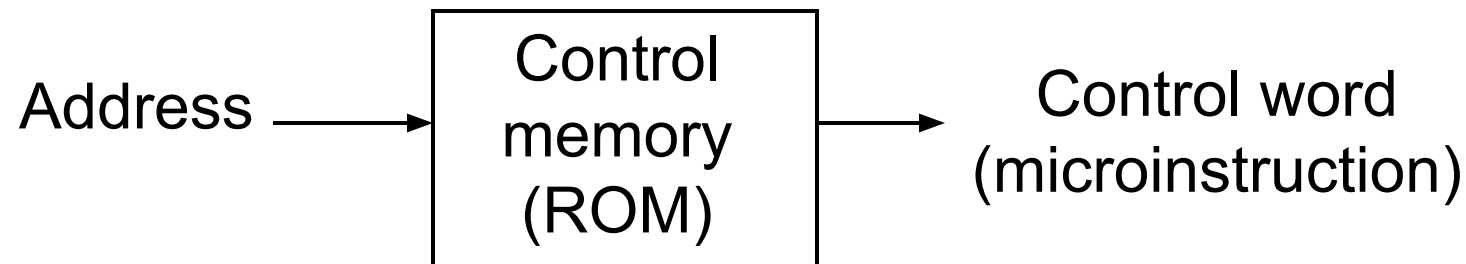


- Microprogrammed

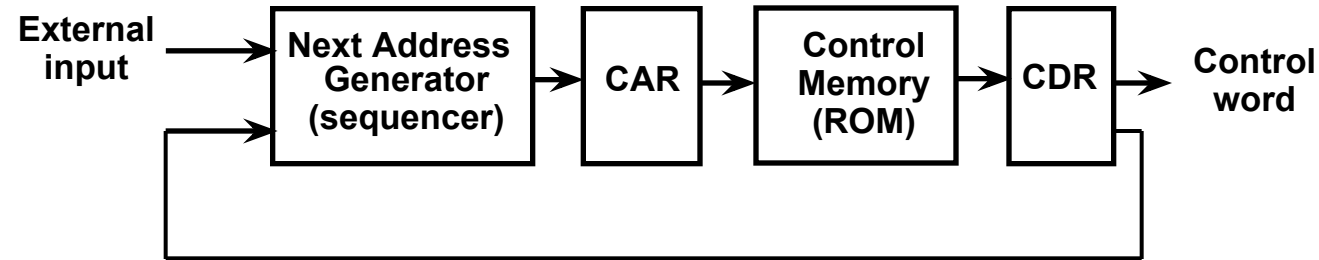


Control Memory

- Read-only memory (ROM)
- Content of word in ROM at given address specifies microinstruction
- Each computer instruction initiates series of microinstructions (microprogram) in control memory
- These microinstructions generate microoperations to
 - Fetch instruction from main memory
 - Evaluate effective address
 - Execute operation specified by instruction
 - Return control to fetch phase for next instruction



Microprogrammed Control Organization



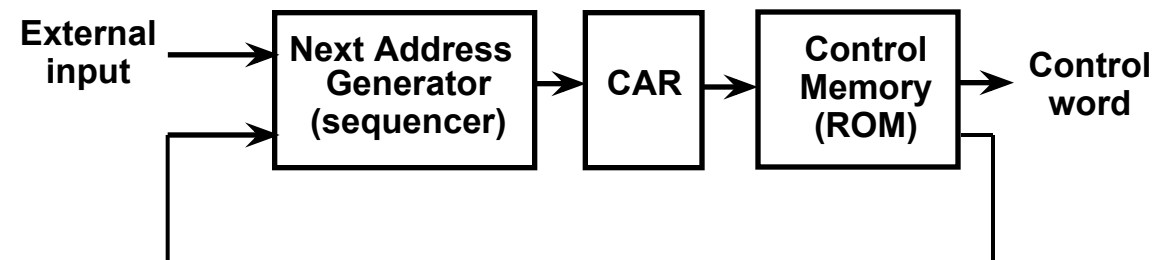
- Control memory
 - Contains microprograms (set of microinstructions)
 - Microinstruction contains
 - Bits initiate microoperations
 - Bits determine address of next microinstruction
- Control address register (CAR)
 - Specifies address of next microinstruction

Microprogrammed Control Organization

- Next address generator (microprogram sequencer)
 - Determines address sequence for control memory
- Microprogram sequencer functions
 - Increment CAR by one
 - Transfer external address into CAR
 - Load initial address into CAR to start control operations

Microprogrammed Control Organization

- Control data register (CDR)- or pipeline register
 - Holds microinstruction read from control memory
 - Allows execution of microoperations specified by control word simultaneously with generation of next microinstruction
- Control unit can operate without CDR



Microinstruction Sequencing:

A micro-program control unit can be viewed as consisting of two parts:

The control memory that stores the microinstructions.

Sequencing circuit that controls the generation of the next address.

Microinstruction Sequencing:

A micro-program sequencer attached to a control memory inputs certain bits of the microinstruction, from which it determines the next address for control memory. A typical sequencer provides the following address-sequencing capabilities:

Increment the present address for control memory.

Branches to an address as specified by the address field of the micro instruction.

Branches to a given address if a specified status bit is equal to 1.

Transfer control to a new address as specified by an external source (Instruction Register).

Has a facility for subroutine calls and returns.

Microinstruction Sequencing:

Depending on the current microinstruction condition flags, and the contents of the instruction register, a control memory address must be generated for the next micro instruction.

There are three general techniques based on the format of the address information in the microinstruction:

Two Address Field.

Single Address Field.

Variable Format

Two address field

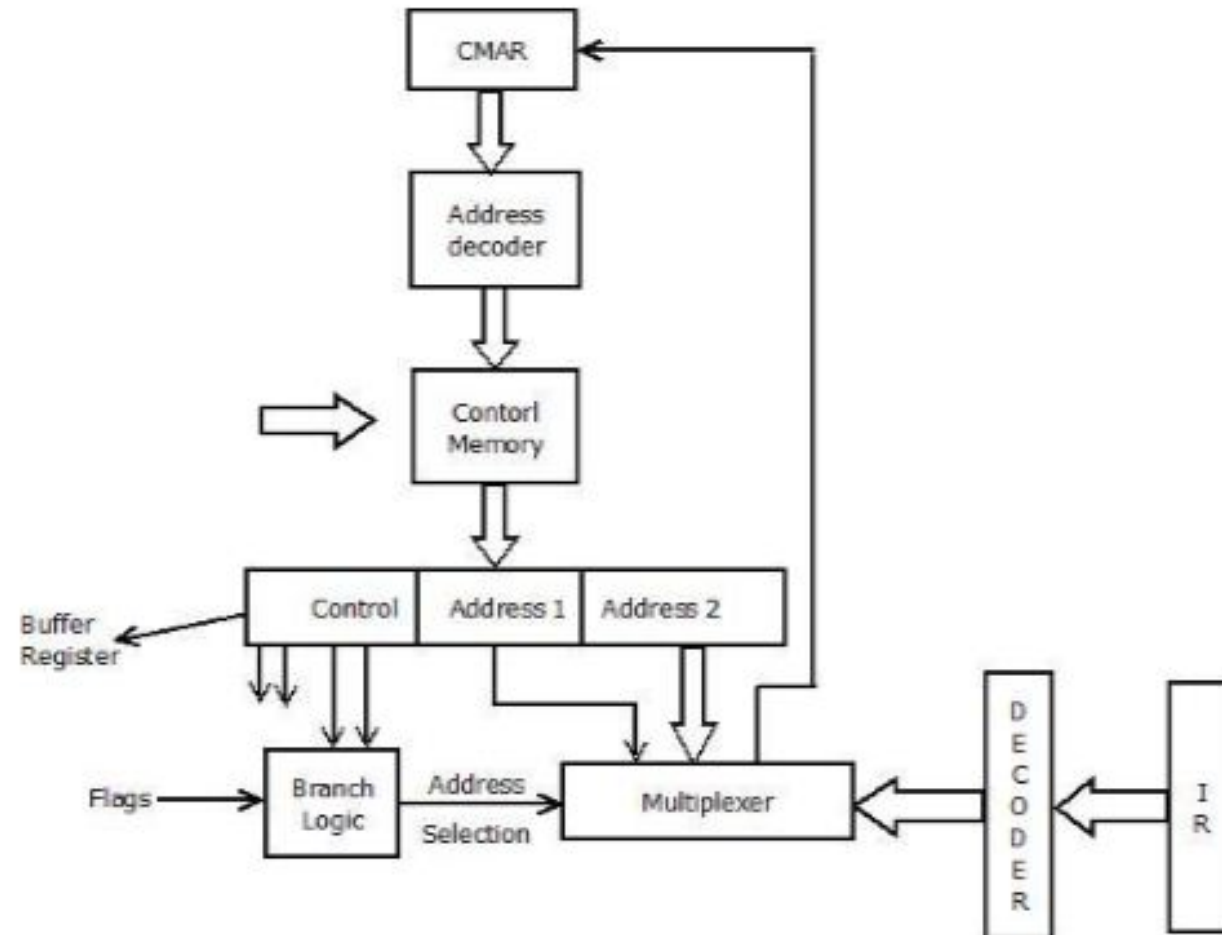
The simplest approach is to provide two address field in each microinstruction and multiplexer is provided to select:

Address from the second address field.

Starting address based on the OPcode field in the current instruction.

The address selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control partition of the micro instruction.

Two address field



Single address field

Two-address approach is simple but it requires more bits in the microinstruction. With a simpler approach, we can have a single address field in the micro instruction with the following options for the next address.

Address Field.

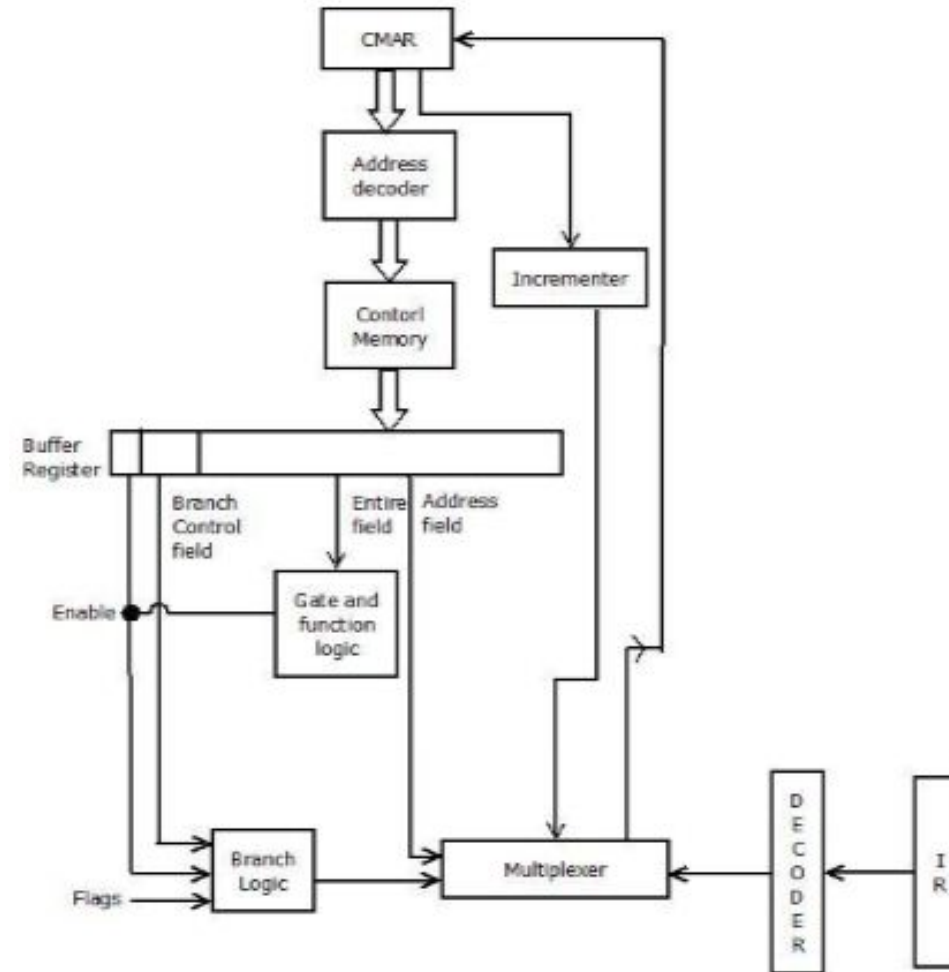
Based on OPcode in instruction register.

Next Sequential Address.

enter image description here

The address selection signals determine which option is selected. This approach reduces the number of address field to one. In most cases (in case of sequential execution) the address field will not be used. Thus the microinstruction encoding does not efficiently utilize the entire microinstruction.

Single address field

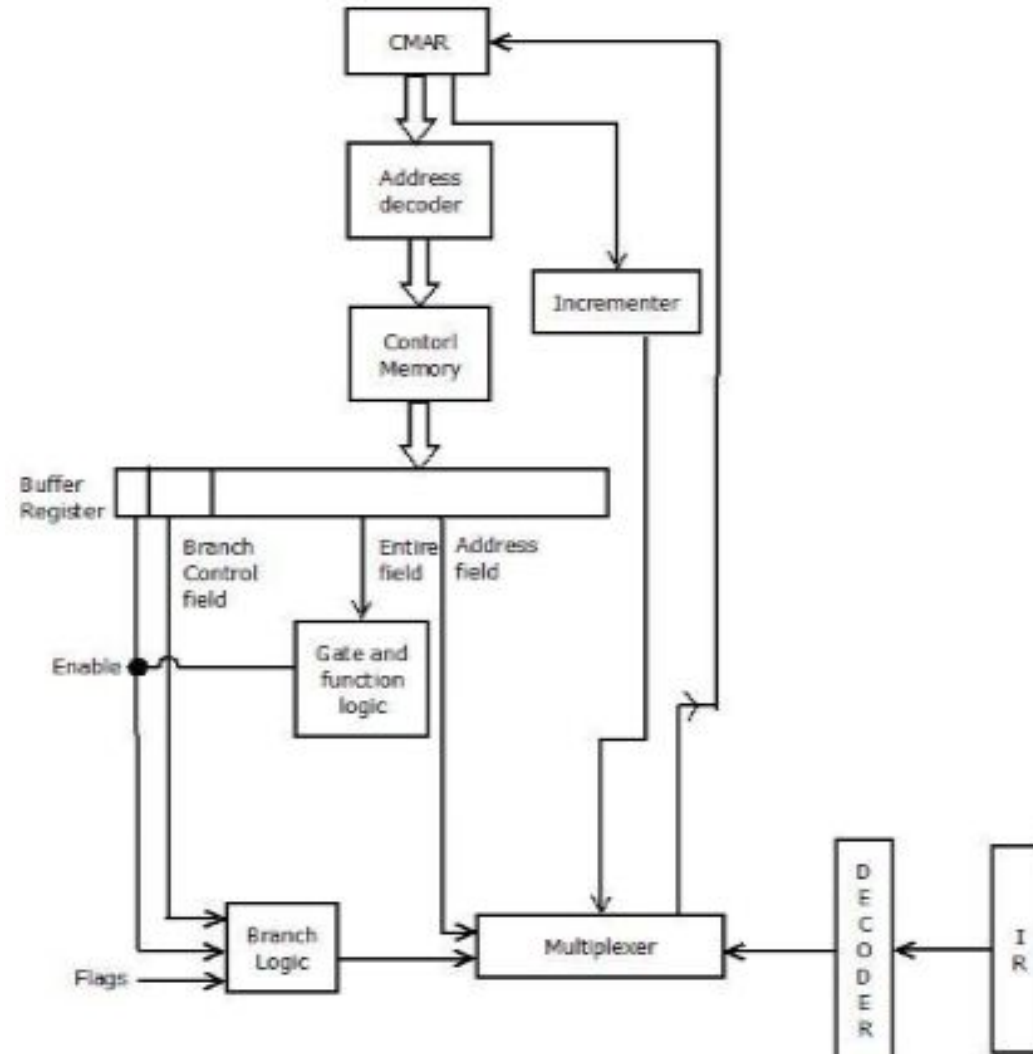


Variable Format

In this approach, there are two entirely different microinstruction formats. One bit designates which format is being used. In this first format, the remaining bits are used to activate control signals.

In the second format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is specified.

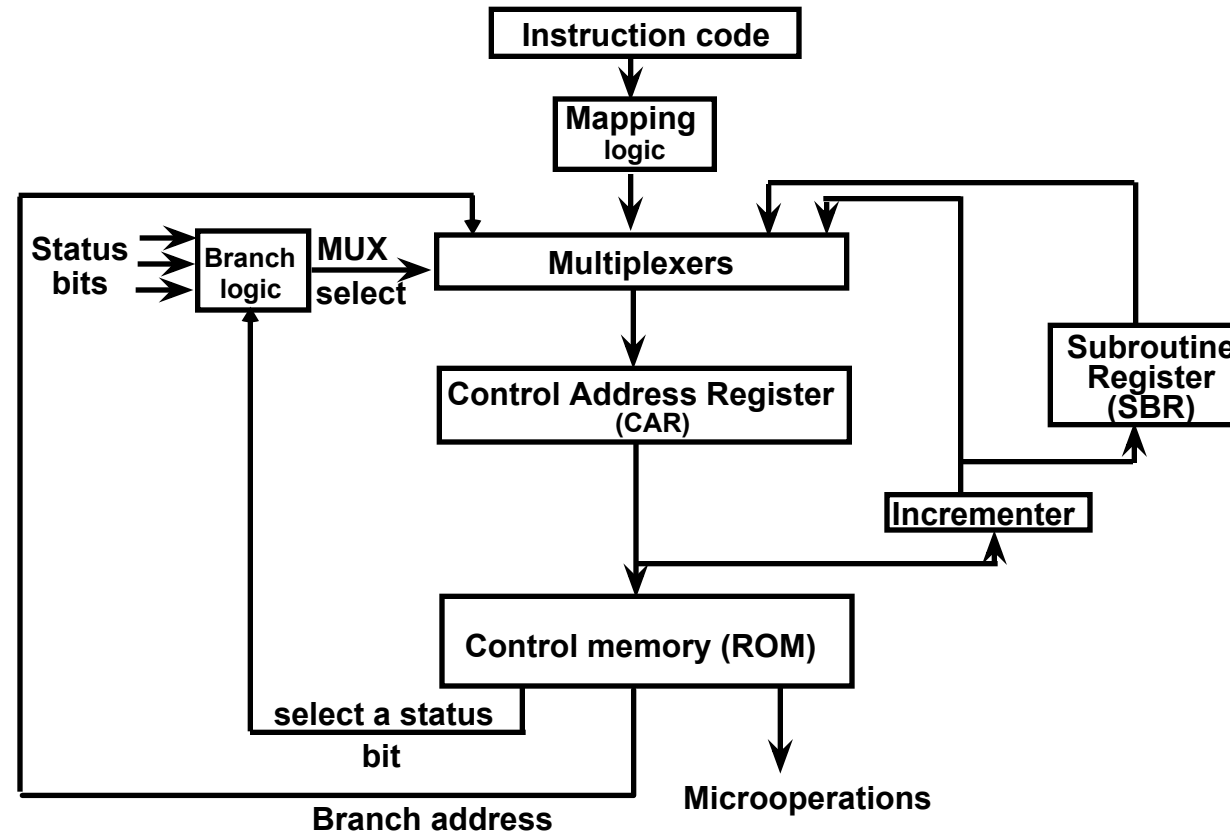
Variable Format



Address Sequencing

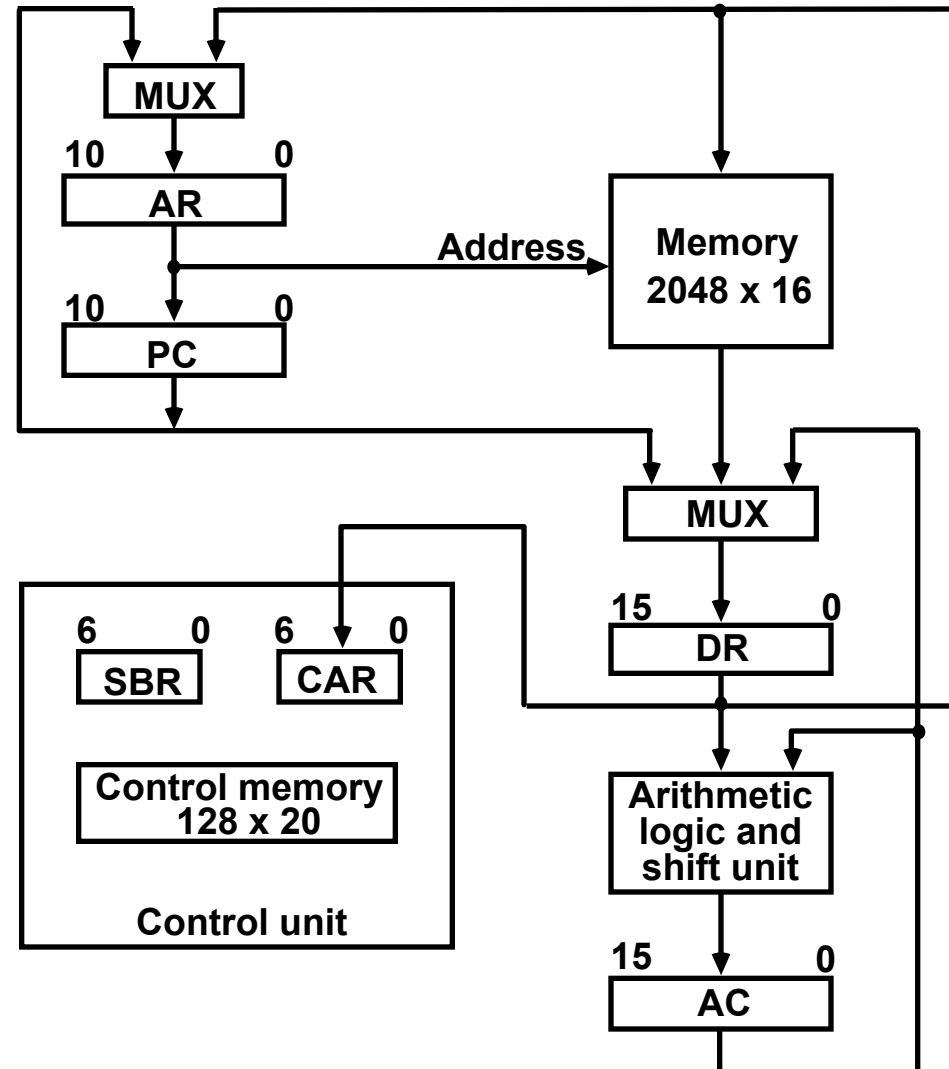
- Address sequencing capabilities required in control unit
 - Incrementing CAR
 - Unconditional or conditional branch, depending on status bit conditions
 - Mapping from bits of instruction to address for control memory
 - Facility for subroutine call and return

Address Sequencing



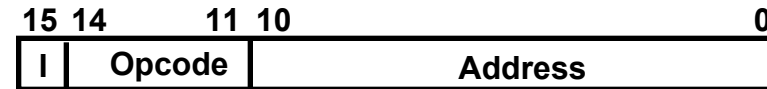
Microprogram Example

**Computer
Configuration**



Microprogram Example

Computer instruction format

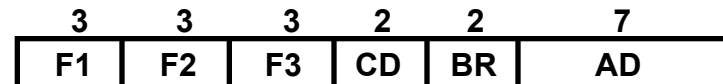


Four computer instructions

Symbol	OP-code	Description
ADD 0000	AC \leftarrow AC + M[EA]	
BRANCH 0001	if (AC < 0) then (PC \leftarrow EA)	
STORE 0010	M[EA] \leftarrow AC	
EXCHANGE 0011	AC \leftarrow M[EA], M[EA] \leftarrow AC	

EA is the effective address

Microinstruction Format



F1, F2, F3: Microoperation fields
 CD: Condition for branching
 BR: Branch field
 AD: Address field

Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0-10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0-10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow AC'$	COM
011	$AC \leftarrow shl AC$	SHL
100	$AC \leftarrow shr AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

Microinstruction Fields

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR \leftarrow AD if condition = 1 CAR \leftarrow CAR + 1 if condition = 0
01 = 1	CALL	CAR \leftarrow AD, SBR \leftarrow CAR + 1 if condition = 1 CAR \leftarrow CAR + 1 if condition = 0
10	RET	CAR \leftarrow SBR (Return from subroutine)
11	MAP	CAR(2-5) \leftarrow DR(11-14), CAR(0,1,6) \leftarrow 0

Symbolic Microinstruction

- **Sample Format**

Label:	Micro-ops	CD	BR	AD
---------------	------------------	-----------	-----------	-----------

- **Label** may be empty or may specify symbolic address terminated with colon
- **Micro-ops** consists of 1, 2, or 3 symbols separated by commas
- **CD** one of {U, I, S, Z}
 U: Unconditional Branch
 I: Indirect address bit
 S: Sign of AC
 Z: Zero value in AC
- **BR** one of {JMP, CALL, RET, MAP}
- **AD** one of {Symbolic address, NEXT, empty}

Fetch Routine

- Fetch routine
 - Read instruction from memory
 - Decode instruction and update PC

Microinstructions for fetch routine:

```

AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

```

Symbolic microprogram for fetch routine:

```

          ORG 64
FETCH:    PCTAR      U  JMP  NEXT
          READ, INCPC U  JMP  NEXT
          DRTAR      U  MAP

```

Binary microporgram for fetch routine:

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Symbolic Microprogram

- **Control memory:** 128 20-bit words
- **First 64 words:** Routines for 16 machine instructions
- **Last 64 words:** Used for other purpose (e.g., fetch routine and other subroutines)
- **Mapping:** OP-code XXXX into 0XXXX00, first address for 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

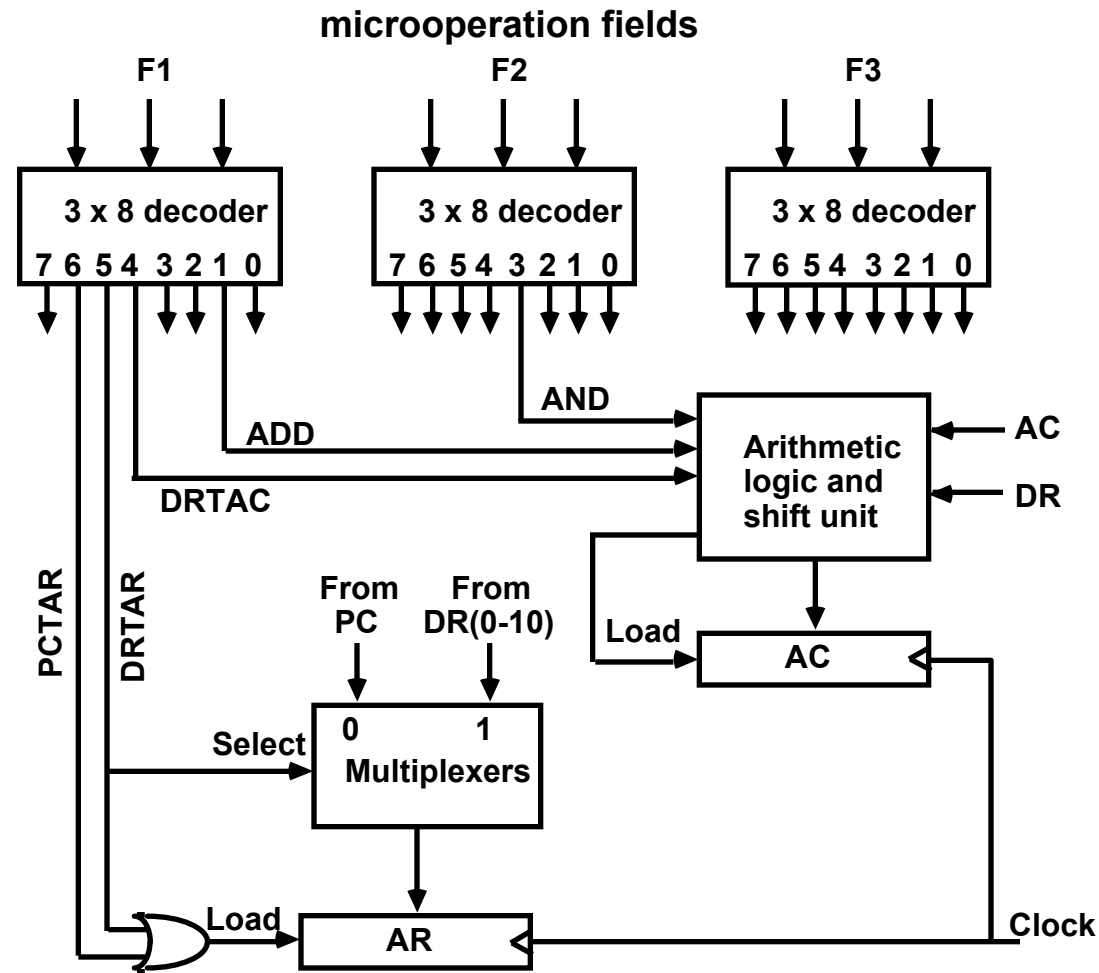
Partial Symbolic Microprogram

Label	Microops	CD	BR	AD
ADD:	ORG 0			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
BRANCH:	ORG 4			
	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
	OVER:	I	CALL	INDRCT
STORE:	ARTPC	U	JMP	FETCH
	ORG 8			
	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
EXCHANGE:	WRITE	U	JMP	FETCH
	ORG 12			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
FETCH:	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
	PCTAR	U	JMP	NEXT
INDRCT:	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
	READ	U	JMP	NEXT
	DRTAR	U	RET	

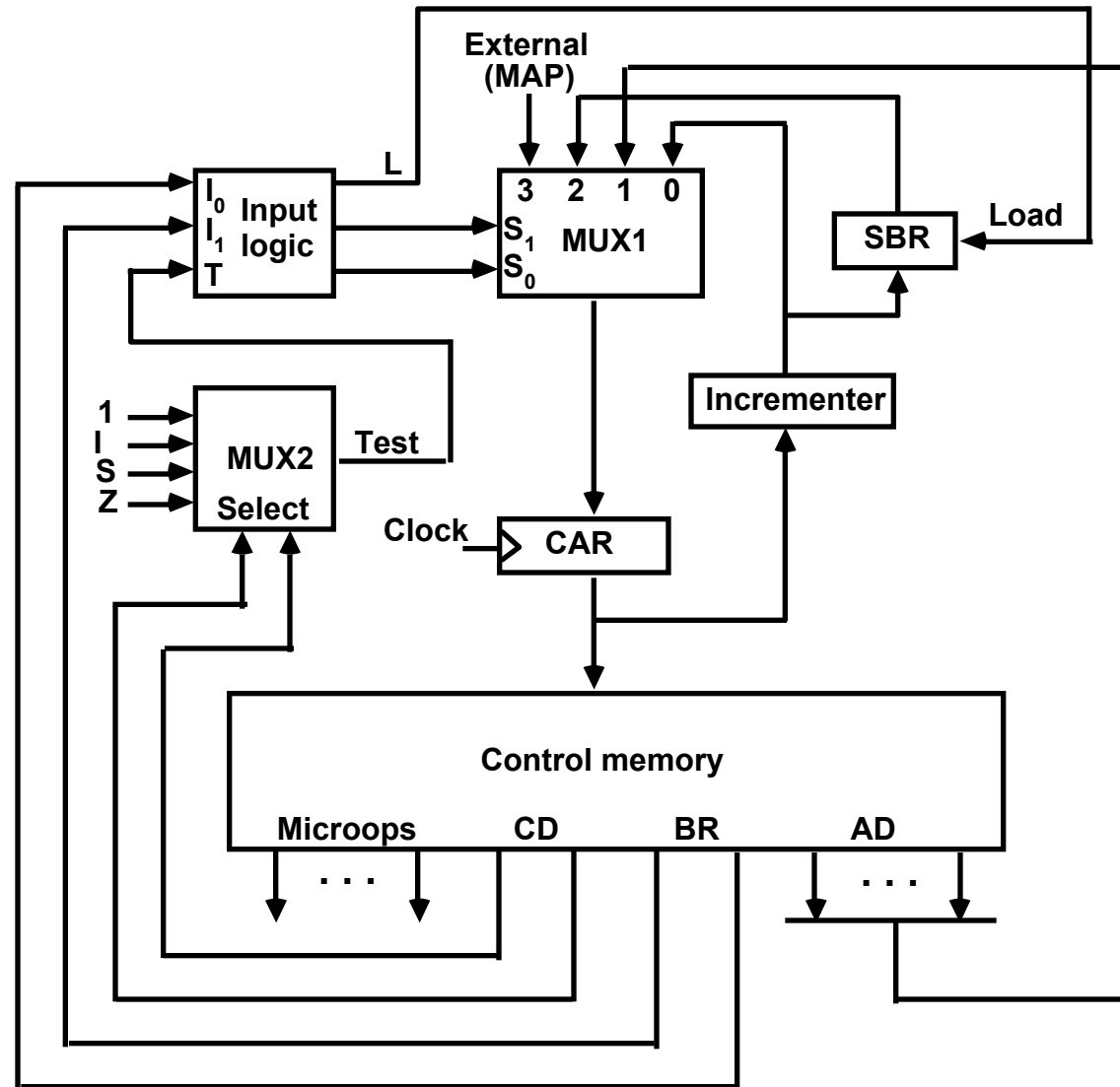
Binary Microprogram

	Micro Routine	Address		Binary Microinstruction							
		Decimal	Binary	F1	F2	F3	CD	BR			
0000110	AD										
	ADD	0	0000000	000	000	000	01	01	1000011		
		1	0000001		000	100		000	00	00	
	0000010										
		2	0000010		001	000		000	00	00	
	1000000										
		3	0000011		000	000		000	00	00	
	1000000										
	BRANCH	4	0000100		000	000		000	10	00	
	0000110										
		5	0000101		000	000		000	00	00	1000000
		6	0000110		000	000		000	01	01	1000011
		7	0000111		000	000		110	00	00	1000000
	STORE	8	0001000		000	000		000	01	01	1000011
		9	0001001		000	101		000	00	00	0001010
		10	0001010		111		000		000	00	00
1000000											
		11	0001011		000		000		000	00	00
1000000											
EXCHANGE	12	0001100		000		000		000	01	01	1000011
		13	0001101		001		000		000	00	00
0001110											
		14	0001110		100		101		000	00	00
0001111											
		15	0001111		111		000		000	00	00
1000000											

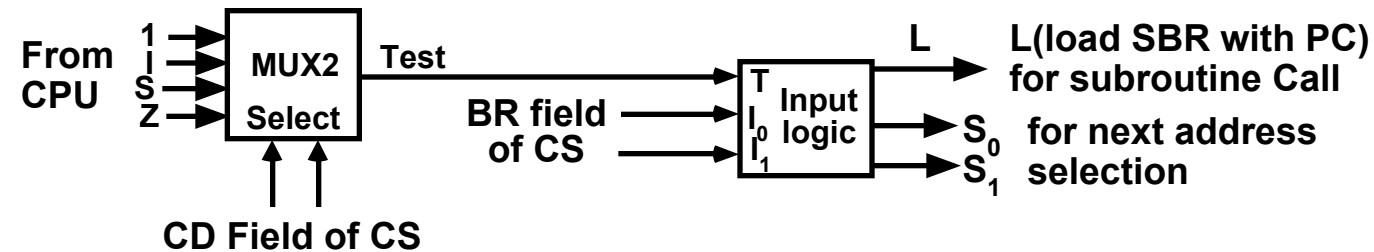
Design of Control Unit



Microprogram Sequencer



Input Logic for Microprogram Sequencer



Input Logic

I ₁ I ₀ T	Meaning	Source of Address	S ₁ S ₀	
L				
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	01	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR ← CAR+1	01	1
10x	RET	SBR	10	0
11x	MAP	BR(11-14)	11	0

$$S_1 = I_1$$

$$S_0 = I_0 I_1 + I_1' T$$

$$L = I_1' I_0 T$$

Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a routine.

To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3

- The next step is to generate the microoperations that execute the instruction fetched from memory.
- The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

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 Pipelining?

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.

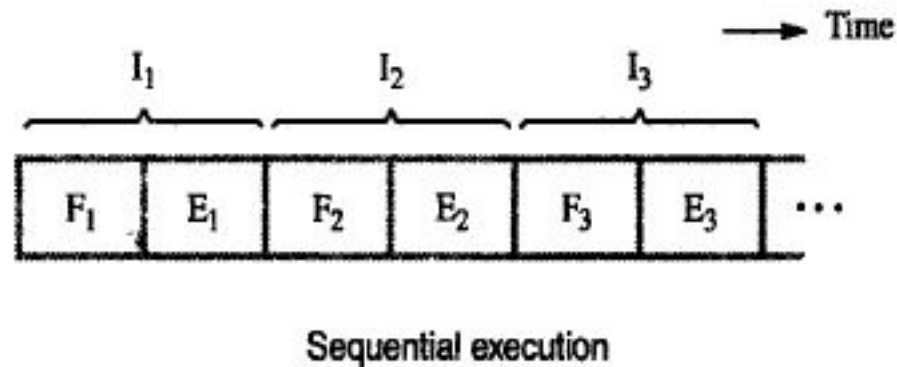
What is Instruction Pipelining?

- 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

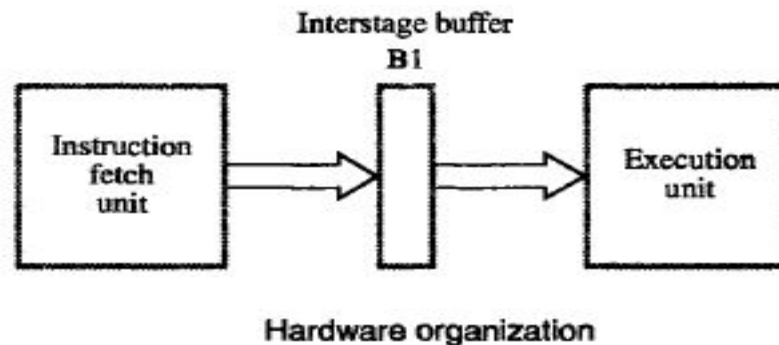
Sequential Execution of a program

- The processor executes a program by fetching(F_i) and executing(E_i) instructions one by one.



Hardware organization and instruction pipeline

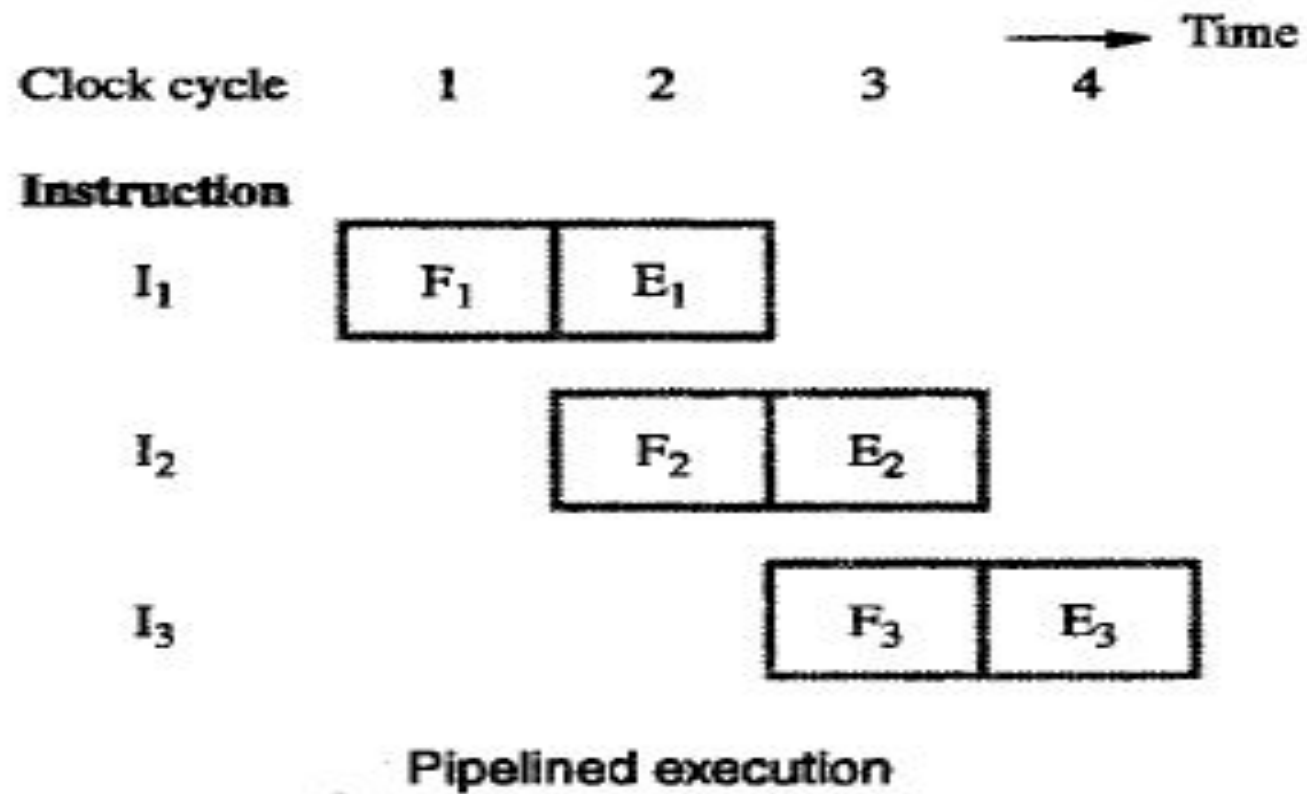
- Consists of 2 hardware units one for fetching and another one for execution as follows.
- Also has intermediate buffer to store the fetched instruction



2 stage pipeline

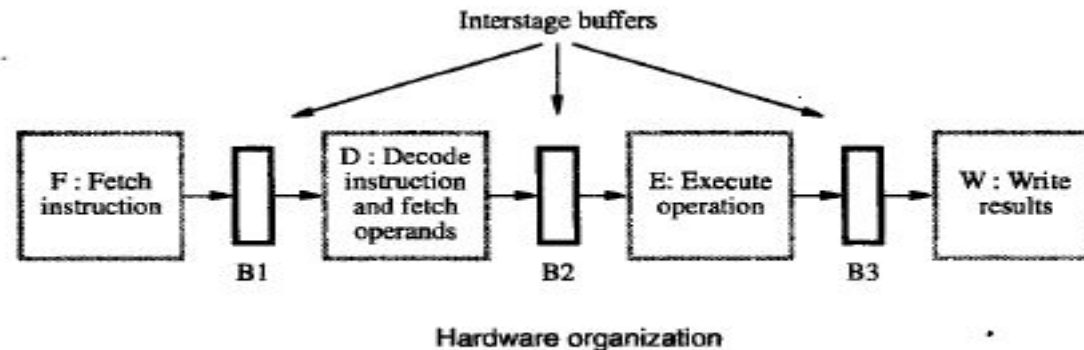
- Execution of instruction in pipeline manner is controlled by a clock.
- In the first clock cycle, fetch unit fetches the instruction I1 and store it in buffer B1.
- In the second clock cycle, fetch unit fetches the instruction I2 , and execution unit executes the instruction I1 which is available in 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.

Contd...



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.

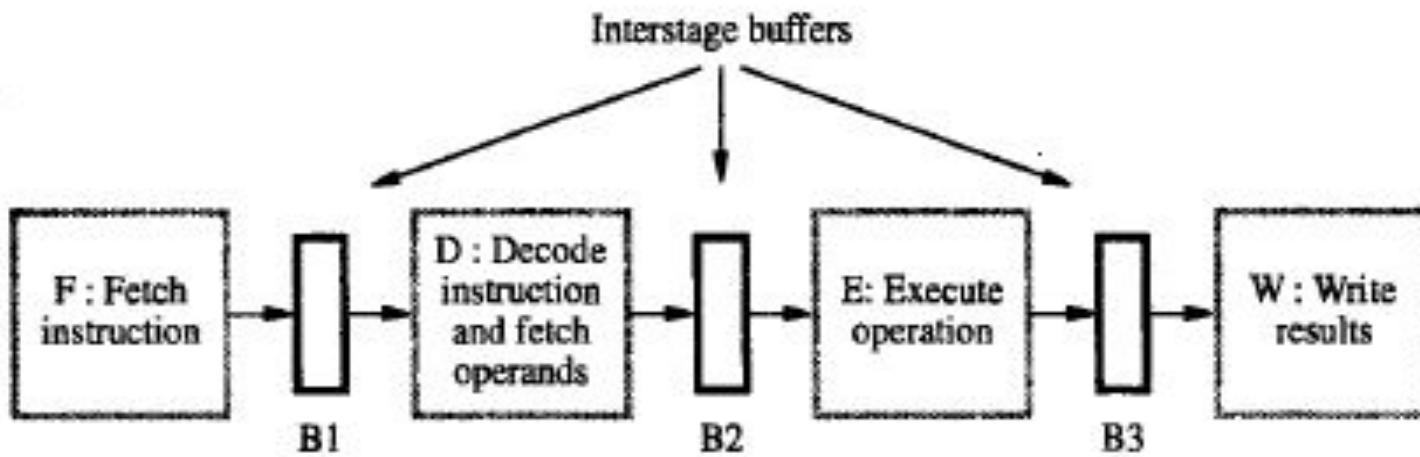


A 4-stage pipeline.

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.

Contd...



Hardware organization

A 4-stage pipeline.

Role of cache memory in Pipelining

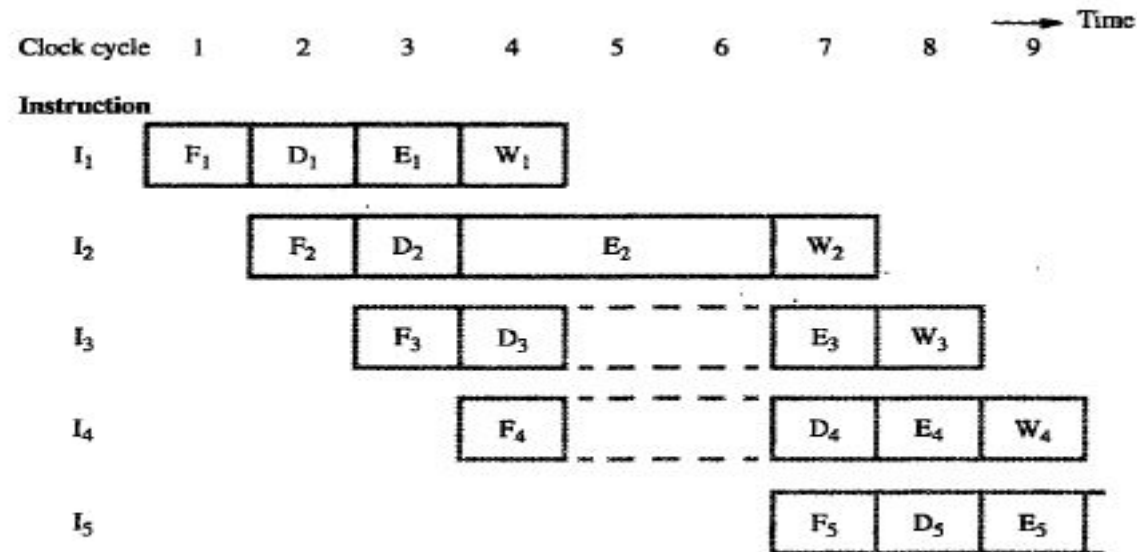
- Each stage of the pipeline is controlled by a clock cycle whose period is that the fetch, decode, execute and write steps of any instruction can each be completed in one clock cycle.
- However the access time of the main memory may be much greater than the time required to perform basic pipeline stage 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 of 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.

Contd...

- 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 complete its execution (cycle 4,5,6)
- In cycle 5,6 the write stage must be told to do nothing, because it has no data to work with.



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

What is Pipeline Hazard?

- Meanwhile, the information in buffer B2 must remain there until the execute stage has completed.
- It shows that stage 2 and stage 1 are blocked from accepting new instructions.
- Thus the steps D4 and F5 must be postponed which is shown in the previous figure.
- The previous figure shows that the pipelined operations are paused for 2 clock cycles and resume the pipelined operations in cycle 7.
- Any condition that causes the pipeline to pause/stall is called **hazard**.

- **Types of hazard:**

1. Data hazard

It is a condition in which either the source or the destination operands are not available at the expected time in the pipeline. The previous figure shows an example of data hazard.

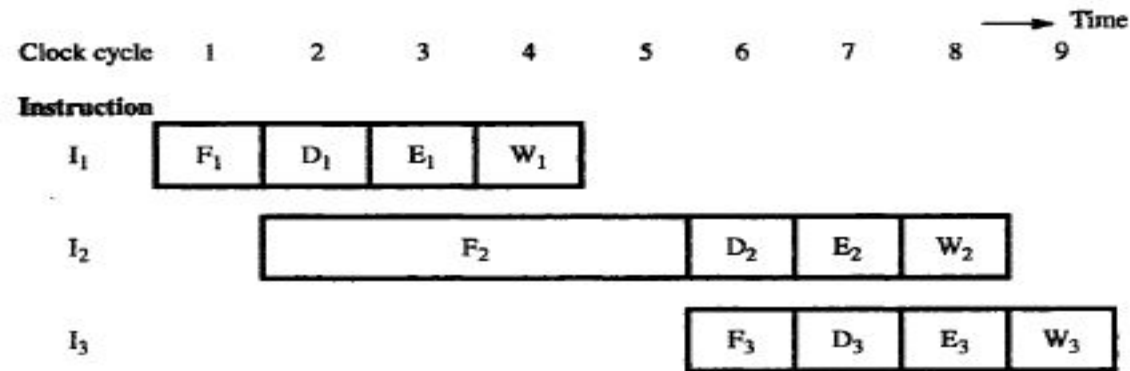
2. Control or Instruction hazard

It is a condition in which the pipelined operations are paused because of the delay in the availability of an instruction.

3. Structural hazard

Example for Instruction hazard

- The following figure illustrate that, Instruction I1 is fetched from cache in clock cyce1 and its execution proceeds normally.
- In clock cyce2, the fetch operation of I2 from cache miss and it suspends the further fetch requests and wait for the arrival of I2.
- The figure shows that, I2 arrived and loaded into B1 at the end of clock cyce5. Now the normal pipeline operation resumes.



Instruction execution steps in successive clock cycles

Contd...

- Note that the decode unit is suspended from clock cyce3 to clock cycle5,
- The execute unit is suspended from clock cycle4 to clock cycle6
- The write unit is suspended from clock cycle5 to clock cycle7
- These suspended period is called stalls/bubbles in the pipeline

What is Structural hazard?

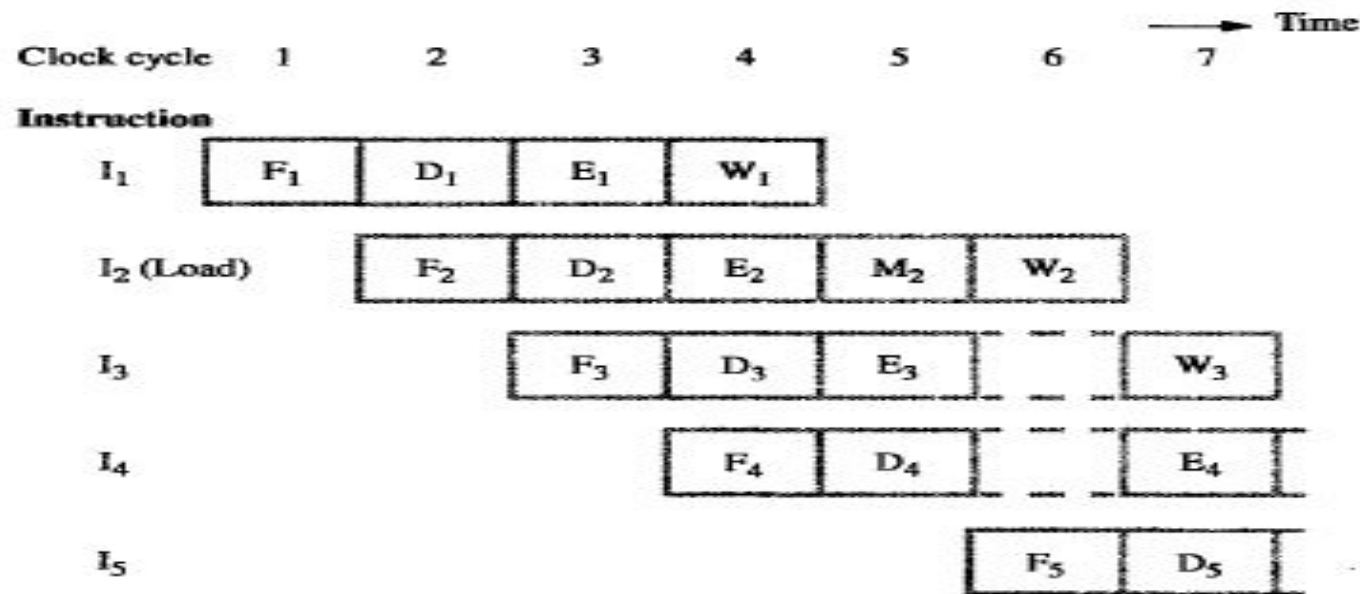
- This hazard arises in a situation when 2 instruction tries to use the same hardware resources at a time.
- Consider a situation, where one instruction may need to access memory either in execute or write stage whereas the other instruction may need to access memory in fetch stage .
- Also assumes that the instruction and data are available in cache.
- In this scenario, only one instruction can proceed while the other is suspended.
- Consider the instruction

I2 : Load X(R1),R2

- Here the memory address, $X + [R1]$ is computed at clock cycle 4, then the memory access takes place in clock cycle 5. (i.e) I2 execution takes 2 clock cycles.

Contd...

- It means that memory access at clock cycle 5 is written into register R2 in clock cycle 6.
- It causes the pipeline operation is suspended for clock cycle 6, since both I2 and I3 require access to the register file in clock cycle 6.



Effect of a Load instruction on pipeline timing.

Important point to consider

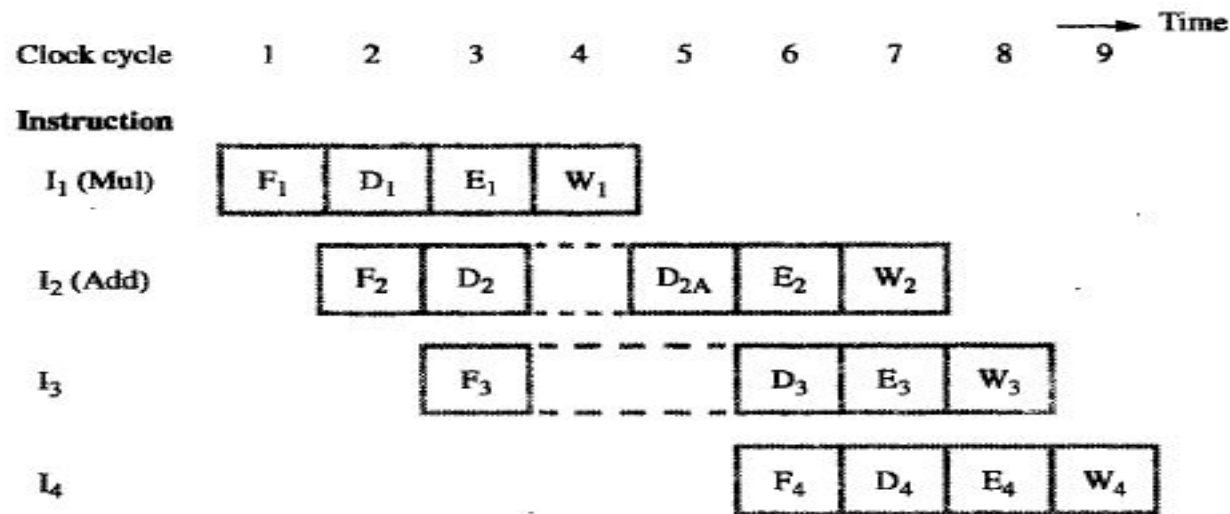
- It is very important that, pipeline operation does not execute the individual instruction faster, instead it increases the overall performance of program execution
- At any point of time, if any one of the stages in pipeline can not complete its operation in one clock cycle, then the pipeline is suspended and some performance degradation occurs.
- So the goal is to identify all hazards and find ways to minimize their impact.

Data Hazards

- It is a condition in which pipeline is stalled/suspended coz of either the source or the destination operands are not available at the expected time in the pipeline .
- Consider the following 2 instruction
I1: $A=3+A$
I2: $B=4*A$
- When these instructions are executed in pipeline, the result generated by I1 may not be available to I2 , since execution of I2 begins before the completion of I1. **As a result, it gives the incorrect result.**
- **In the above example the result of the I1 is taken as the Input to I2. So data dependency arises.**
- **The data dependency may also arises when the destination of one instruction is used as a source in the next instruction.**
- **Example:**
 - I1 : Mul R2,R3,R4 $R4= R2*R3$
 - I2 : Add R5,R4,R6 $R6=R5+R4$

Contd...

- The result of Mul instruction is R4, which in turn is used as one of the two source operands of Add instruction.
- Assume that the execution unit of Mul instruction takes one clock cycle.
- In clock cycle 3, when the decode unit of Add instruction realizes that R4 is used as the source operand, it cannot be completed the decoding until the write unit of Mul instruction has been completed.



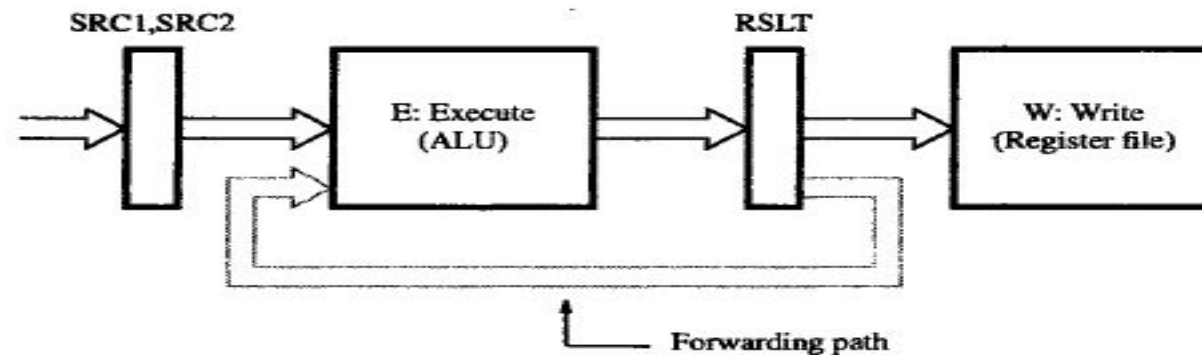
Pipeline stalled by data dependency between D₂ and W₁.

Contd...

- So the completion of D2 must be delayed to clock cycle 5 as shown in the above Figure as D2A.
- Instruction I3 is fetched in clock cycle 3, but its decoding is done in clock cycle 6. As shown in the above Figure the pipeline is stalled for two clock cycles.

Methods to overcome Data hazards (Operand Forwarding)

- Data hazards occur when instructions that exhibit data dependence
- In the previous example, instruction I2 is waiting for data to be written in the register by instruction I1.
- However, this data is available at the o/p of ALU once the execution of I1 gets over in the unit E1.
- Hence, the delay can be reduced or sometimes eliminated if we rearrange for the result of instruction I1 to be forwarded directly to the step E2.

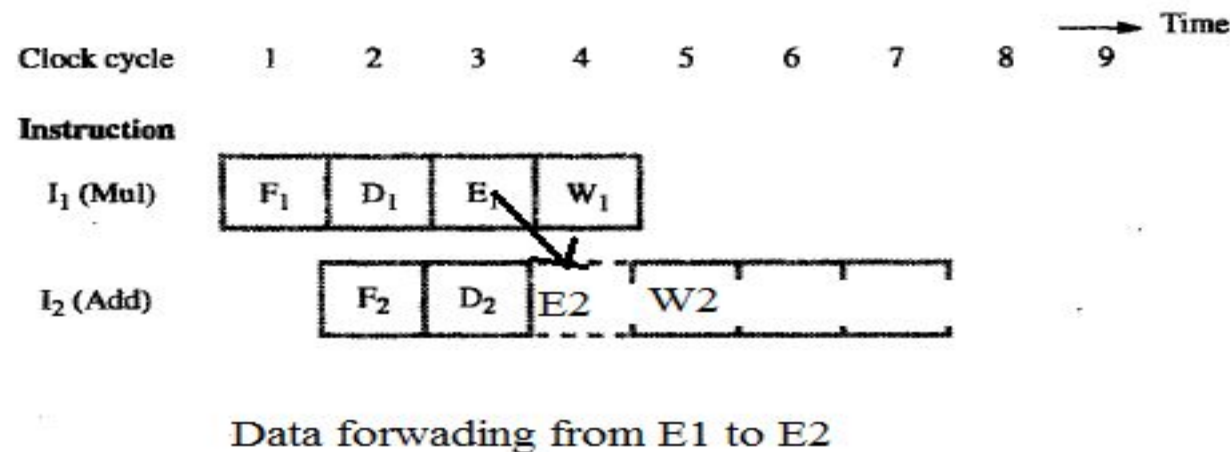


(b) Position of the source and result registers in the processor pipeline

Operand forwarding in a pipelined processor.

Contd...

- After decoding the instruction I2 in clock cycle 3 , the system identifies the data dependency and so a decision is made to use data forwarding.
- The operand not involved in the dependency is read and loaded in register SRC1 in clock cycle 3 .
- In the next clock cycle , the result produced by I1 is available in the register RSLT (because of data forwarding) can be used in step E2.
- Hence execution of I2 proceeds without interruption.



Handling Data hazards in Software

- Another approach of identifying data dependencies and dealing with them by using software.
- In this case, the compiler can introduce 2 clock cycle delay needed between I1 and I2 by inserting NOP (No OPeration) instructions as follows.

- I1 : Mul R2,R3,R4 $R4 = R2 * R3$
- NOP
- NOP
- I2 : Add R5,R4,R6 $R6 = R5 + R4$

What is implicit dependency and Side Effects of this?

- The data dependencies encountered in the previous example is explicit, easily detected and can be resolved.
- But this is not the case always, when a location other than one explicitly named in an instruction as a destination operand is affected, then the **instruction said to have side effect**.
- **Example1:**
 - Stack instructions such as push and pop produce side effects because they implicitly use auto increment and auto decrement addressing modes.
- **Example2:**
 - I1 : Add R1,R3
 - I2 : Add WithCarry R2,R4

Contd...

- In example 2, an implicit dependency exists between these instructions through carry flag.
- The carry flag set by the first instruction is used in the second instruction which performs $R4 = [R2] + [R4] + \text{carry}$.
- The instructions that have side effects give rise to multiple data dependencies which lead to the increase in both the hardware and software needed to resolve them.
- So, the instructions which are executed in pipeline manner should have few side effects.

Instruction Hazards

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

Unconditional Branches

- . 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 three 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

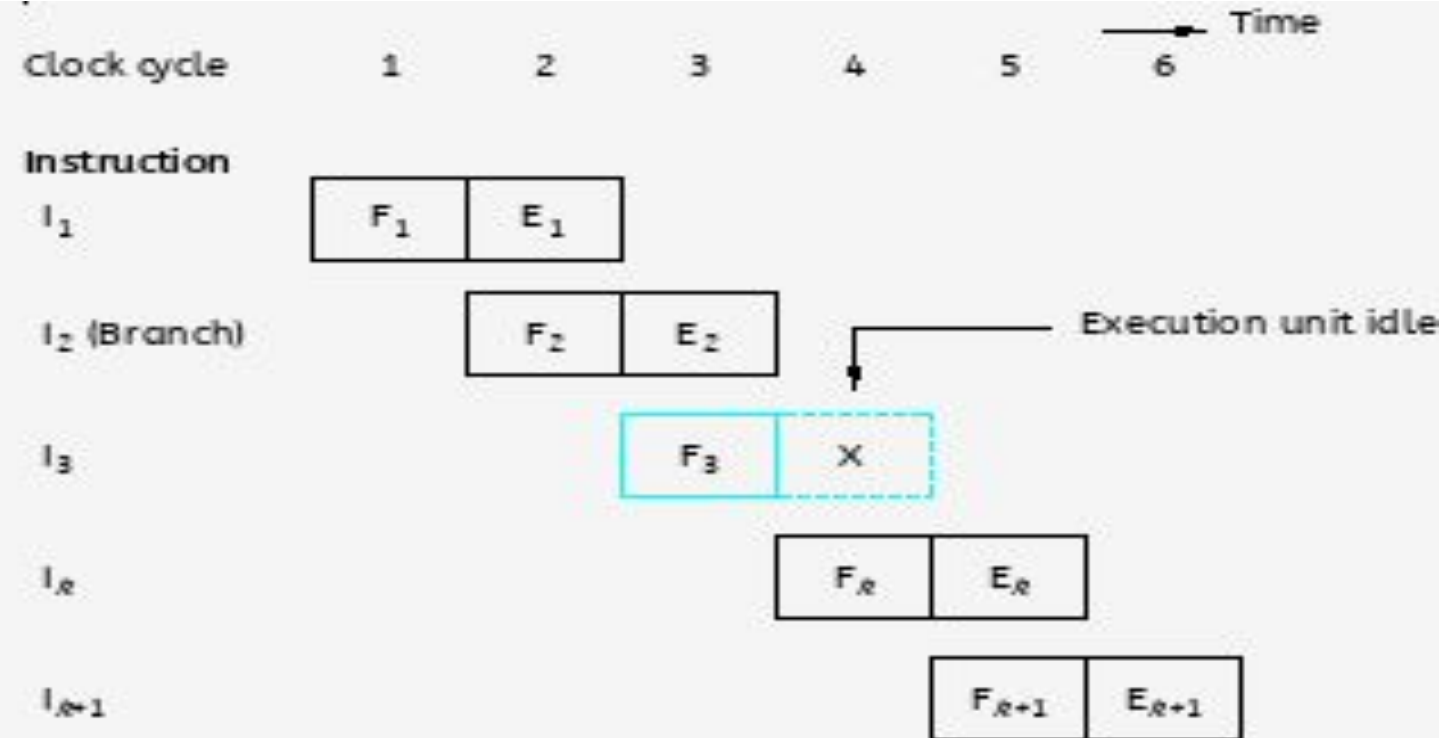


Figure 8.8. An idle cycle caused by a branch instruction.

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 Through compiler techniques

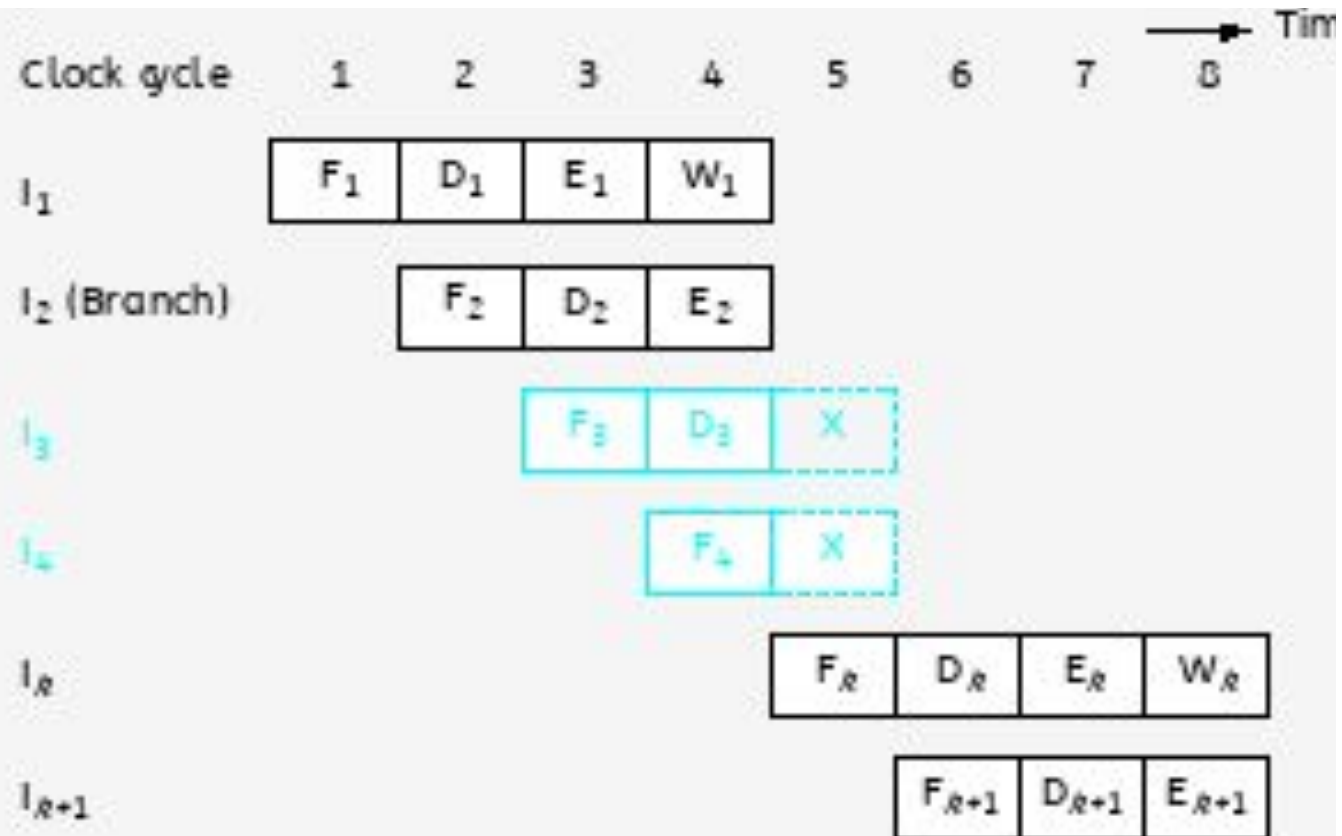
For longer pipeline, the branch penalty may be higher

Reducing the branch penalty requires branch address to be computed earlier in the pipeline

Instruction fetch unit has dedicated hardware

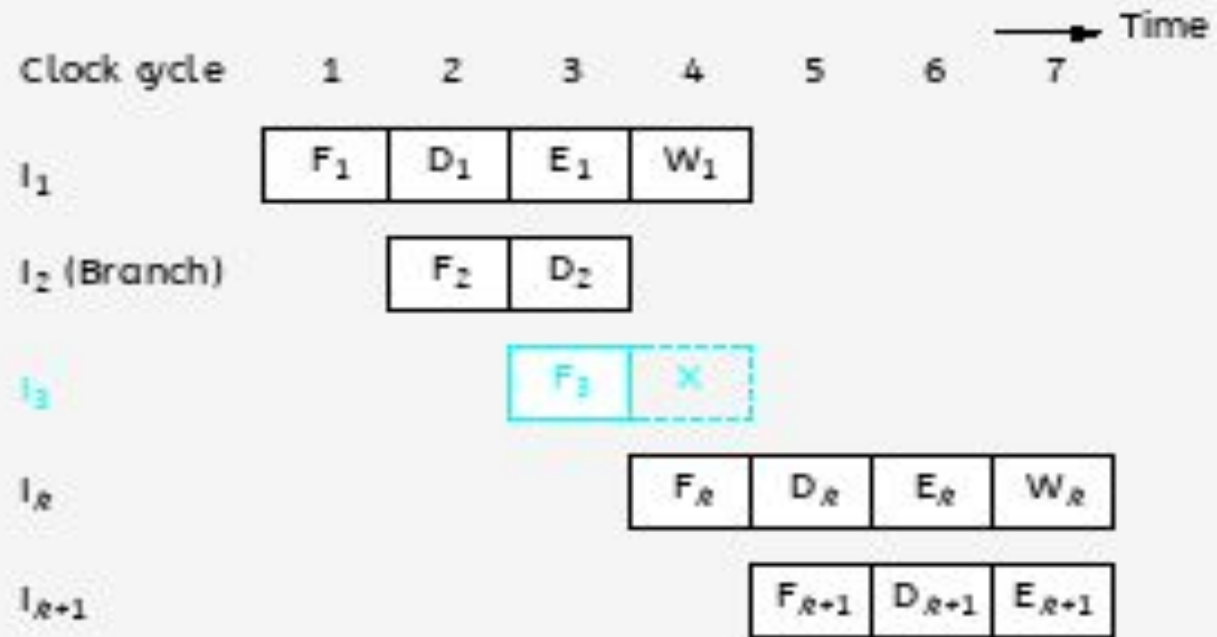
to identify a branch instruction and compute branch target address as quickly as possible after an instruction is fetched

Branch Timing



(a) Branch address computed in the stage

Branch Timing

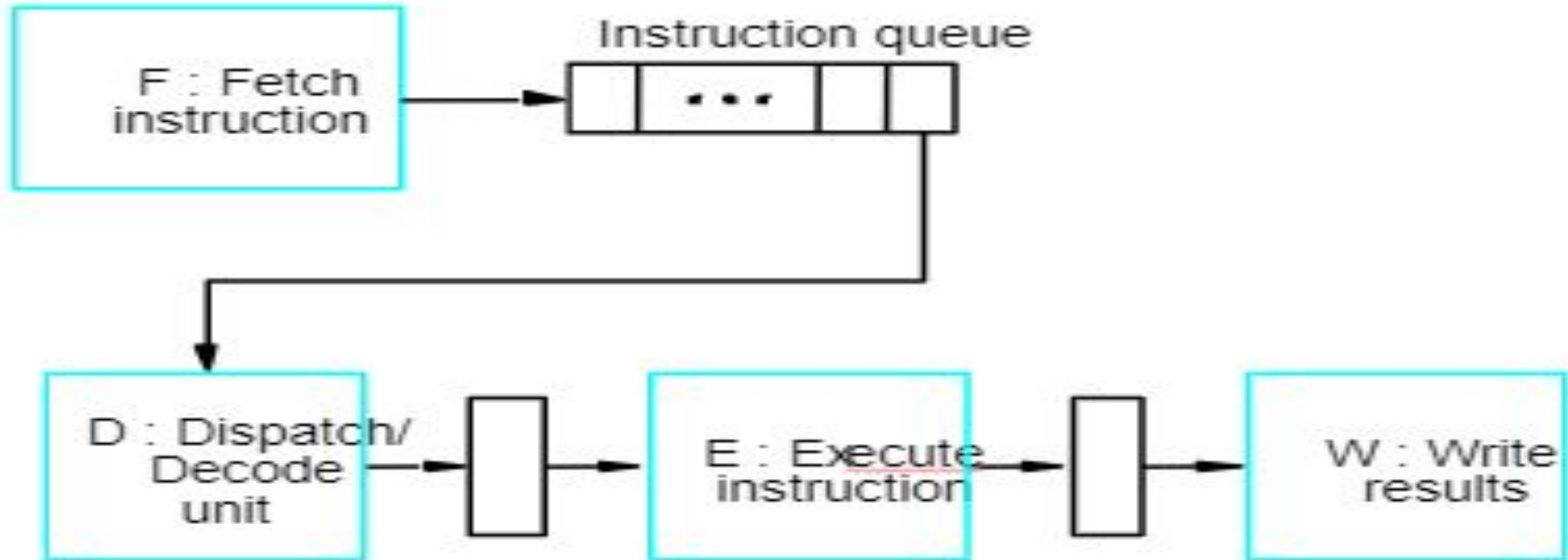


(b) Branch address computed in Decode stage

Figure 8.9. Branch timing.

Instruction Queue and Prefetching

Instruction fetch unit



Instruction Queue and Prefetching

- Fetch Unit-Contain instruction queue to store the instruction before the are needed to avoid interruption
- 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, fetch instruction and add them to the queue.
- If there is delay in fetching the instrution,the dispatch unit continues to issue the instruction from the instruction queue

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.

Branch instructions represent about 20% of the dynamic instruction count of most programs.

Delayed Branch

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

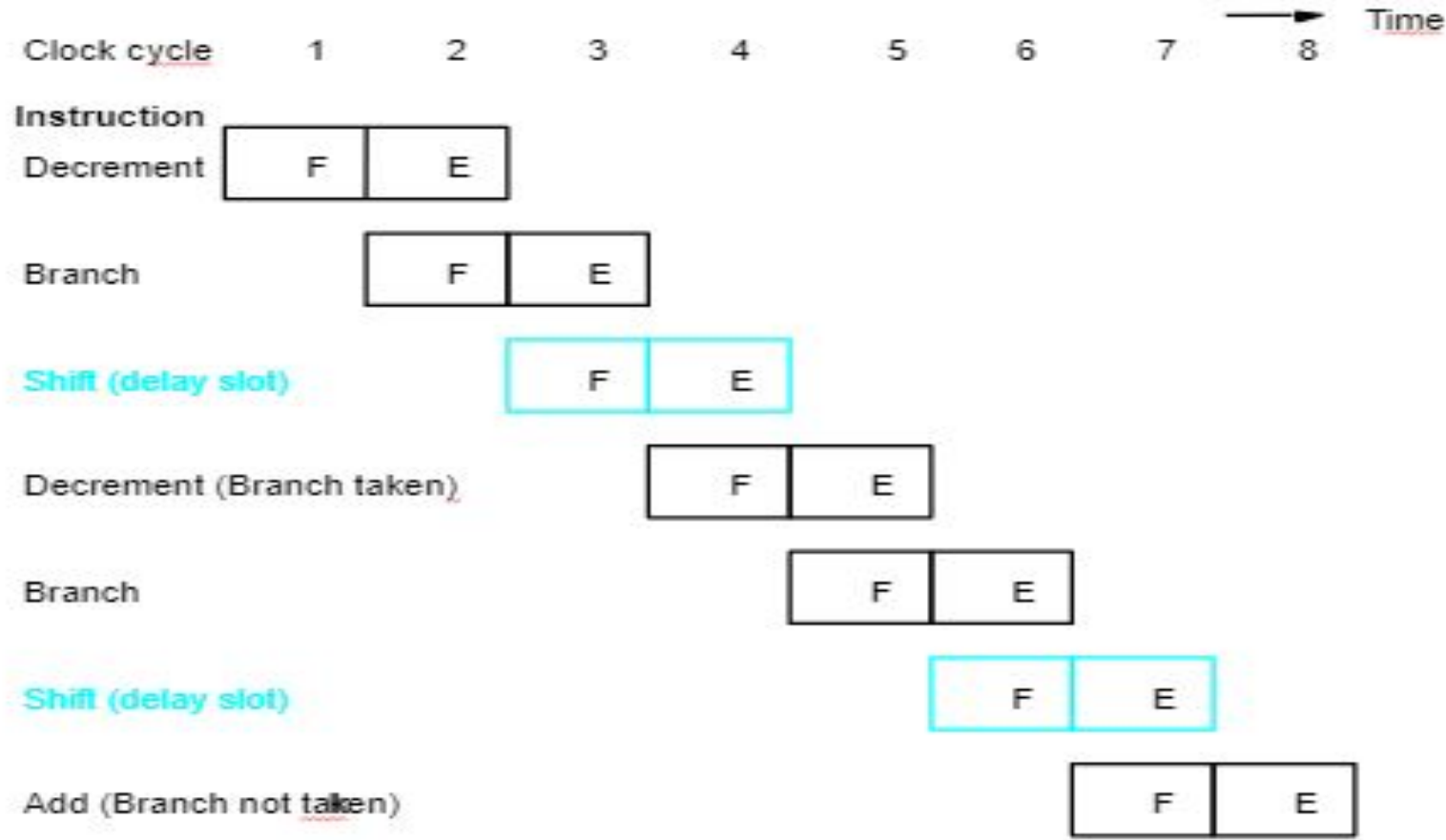
LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

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

(b) Reordered instructions

Delayed Branch



Branch Prediction

To predict whether or not a particular branch will be taken.

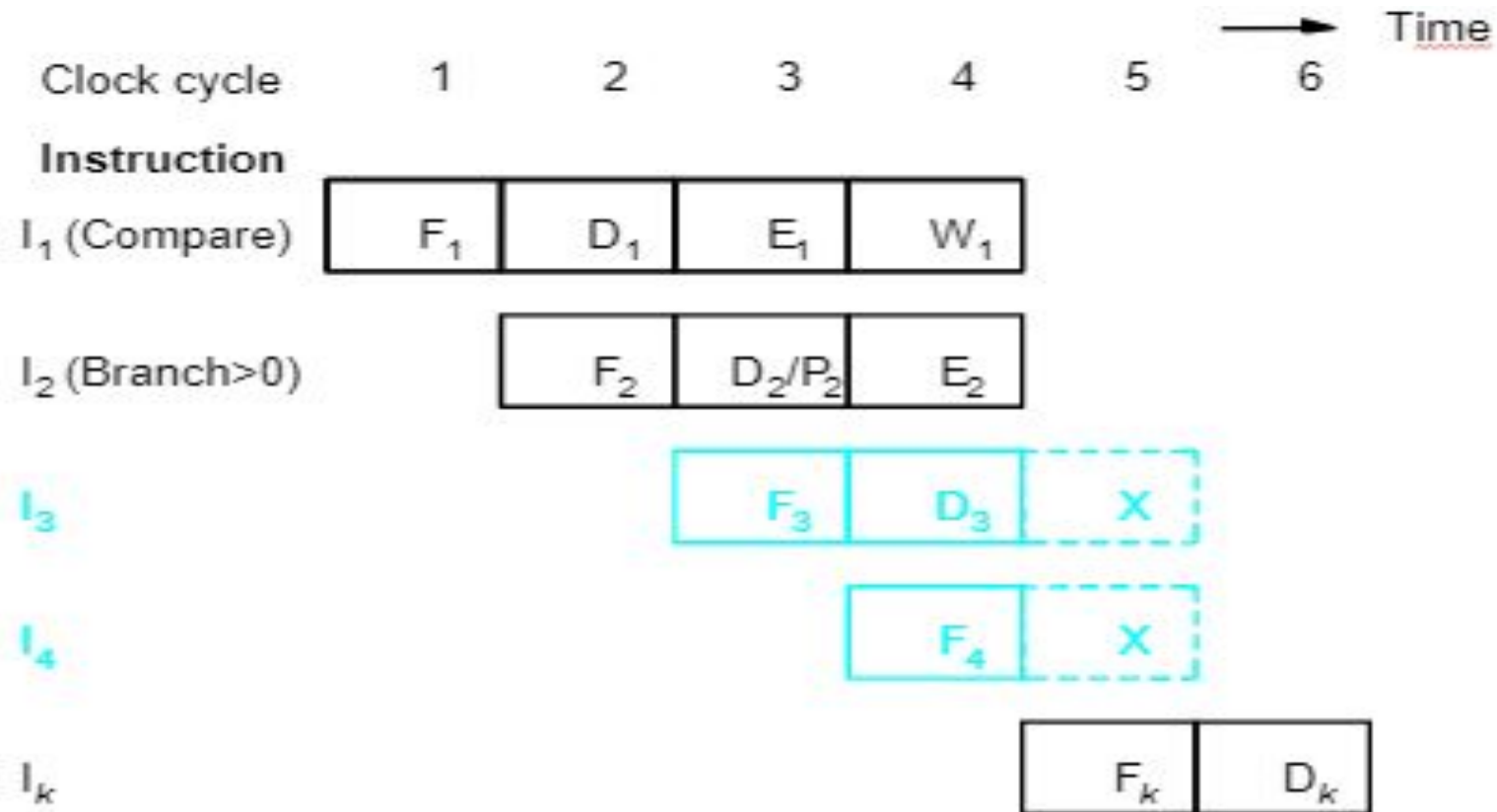
Simplest form: assume branch will not take place and continue to fetch instructions in sequential address order.

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.

Incorrectly Predicted Branch



Branch Prediction

Better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.

Use hardware to observe whether the target address is lower or higher than that of the branch instruction.

Let compiler include a branch prediction bit.

So far the branch prediction decision is always the same every time a given instruction is executed – static branch prediction.

Branch Prediction

- . Static Prediction
- . Dynamic branch Prediction

Static Prediction

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

Dynamic Branch Prediction

- 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

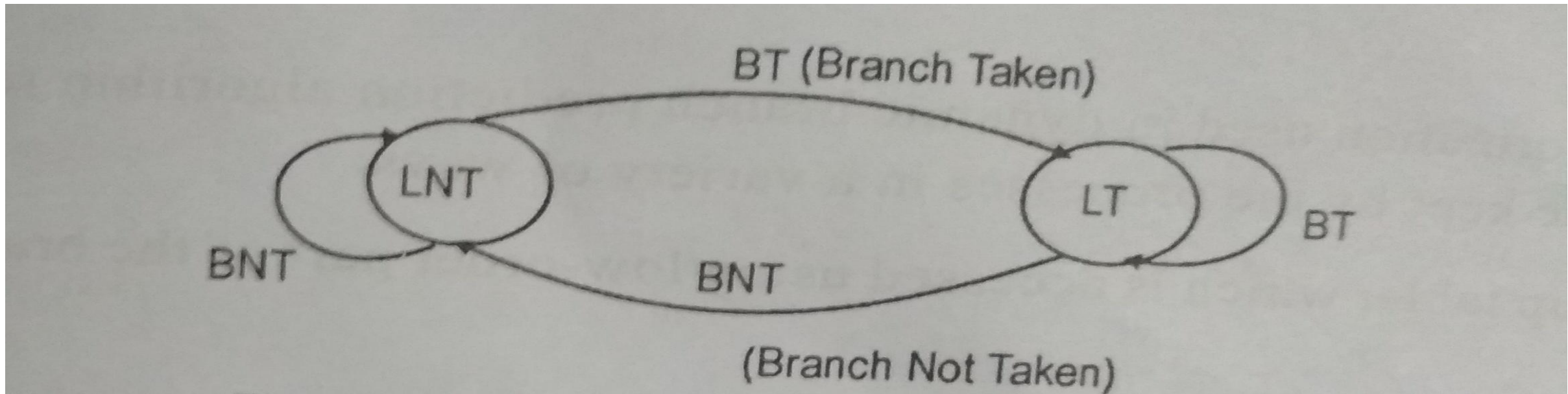
State 1: LT : Branch is likely to be taken

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

Branch Prediction Algorithm



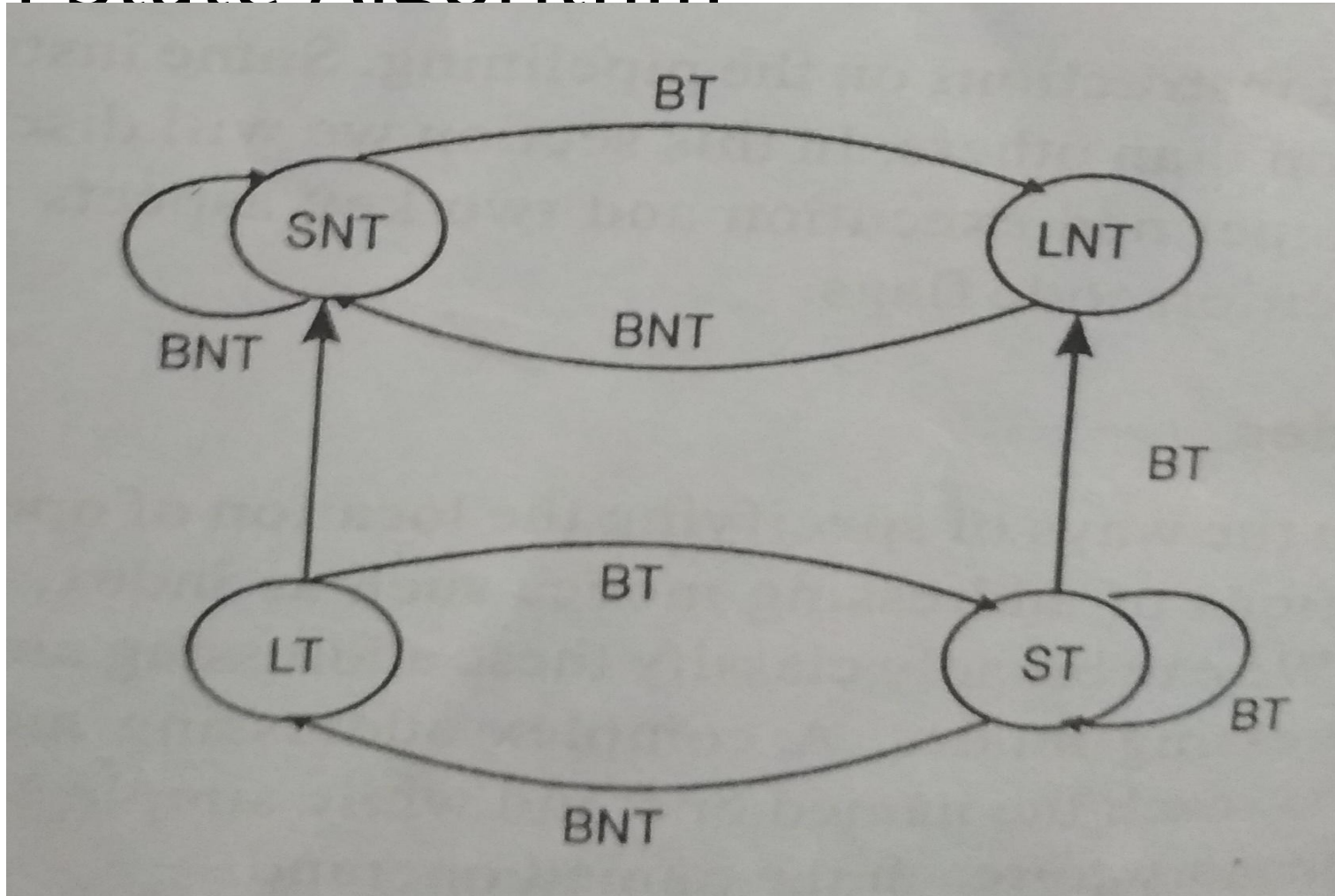
4 State Algorithm

- . 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

4 State Algorithm





CONTROL HAZARDS

CONTROL HAZARDS

- A Control Hazard occurs if there is a control instruction(e.g.BEQ) because the program counter(PC) following the control instruction is not known until the control instruction is computes if the branch should be taken or not.
- If branch taken,then need to zap/flush instructions.
- There is a performance penalty for branches:
- Need to stall,then may ned to zap(flush)
- Subsequent instructions that have already been fetched.

Contd...

Control Hazards

- Instructions are fetched in stage 1(IF)
- Branch and jump decisions occur in stage 3(EX)
i.e. Next PC is not known until 2 cycles after
branch/jump
- Can optimize and move branch and jump decision to
stage 2
i.e. Next PC is not known until 1 cycles after
branch/jump

Contd...

Stall(+Zap)

- Prevent PC update
- Clear IF/ID pipeline register
 - Instruction just fetched might be wrong one,so convert to nop
- Allow branch to continue into EX stage
- We can reduce cost of a control hazard by moving branch decision and calculation from EX stage to ID stage.This reduces the cost from flushing two instructions to only flushing one.

Contd...

- ISA says N instructions after branch/jump always executed
 - MIPS has 1 branch delay slot
 - i.e. Whether branch taken or not, instruction following branch is always executed
- Delay Slots can potentially increase performance due to control hazards by putting a useful instruction in the delay slot since the instruction in the delay slot will always be executed.
- Require software (compiler) to make use of delay slot.
- Put NOP in delay slot if not able to put useful instruction in delay slot.

Contd...

- Pipelining and branching don't get along
- Transfer of control (jumps, procedure call/returns, successful branches) cause control hazards
- When a branch is known to succeed, at the Mem stage (but could be done one stage earlier), there are instructions in the pipeline in stages before Mem that
 - need to be converted into "no-op"
 - and we need to start fetching the correct instructions by using the right PC

instructions

Contd...

- Branches(conditional, unconditional, call-return)
- Interrupts : asynchronous event(e.g.,I/O)
 - Occurrence of an event checked at every
 - If an interrupt has been raised, don't fetchflush the pipe, handle the interrupt
- Exceptions(e.g., arithmetic overflow, page fault etc.,)
 - Program and data dependent(repeatable),
hence"synchronous"

cycle
next instruction,

RESOLVING CONTROL HAZARDS

- Detecting a potential control hazard is easy
 - Look at the opcode
- We must insure that the state of the program is not changed until the outcome of the branch is known.

Possibilities are:

- Stall as soon as opcode is detected(cost 3 bubbles; same type of logic as for the load stall but for 3 cycles instead of 1)
- Assume that branch won't be taken(cost only if branch is taken)
- Use some predictive techniques

INFLUENCE ON INSTRUCTION SETS

OVERVIEW

- 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

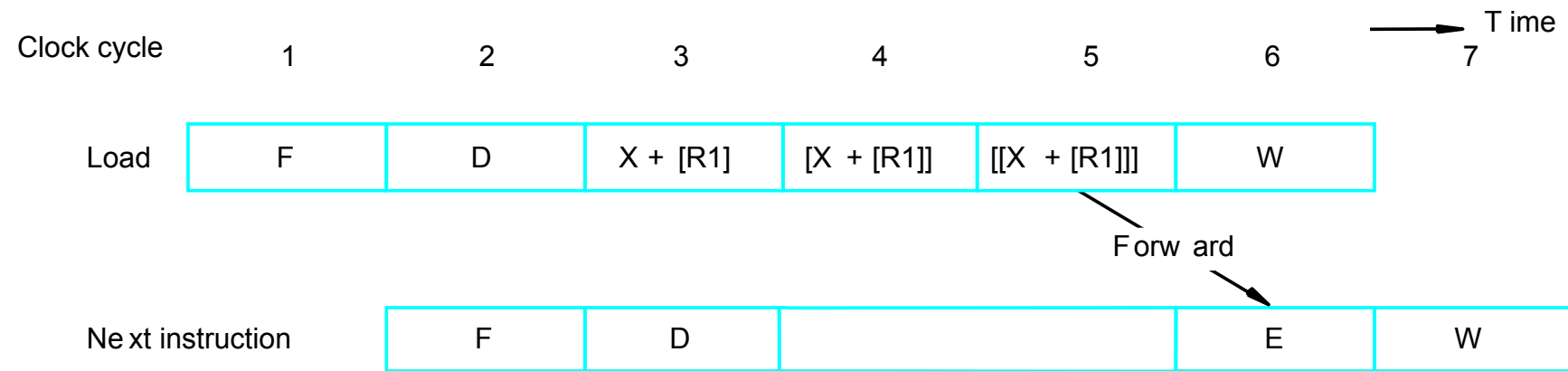
RECALL

Load $X(R1), R2$

Load $(R1), R2$

COMPLEX ADDRESSING MODE

Load (X(R1)), R2



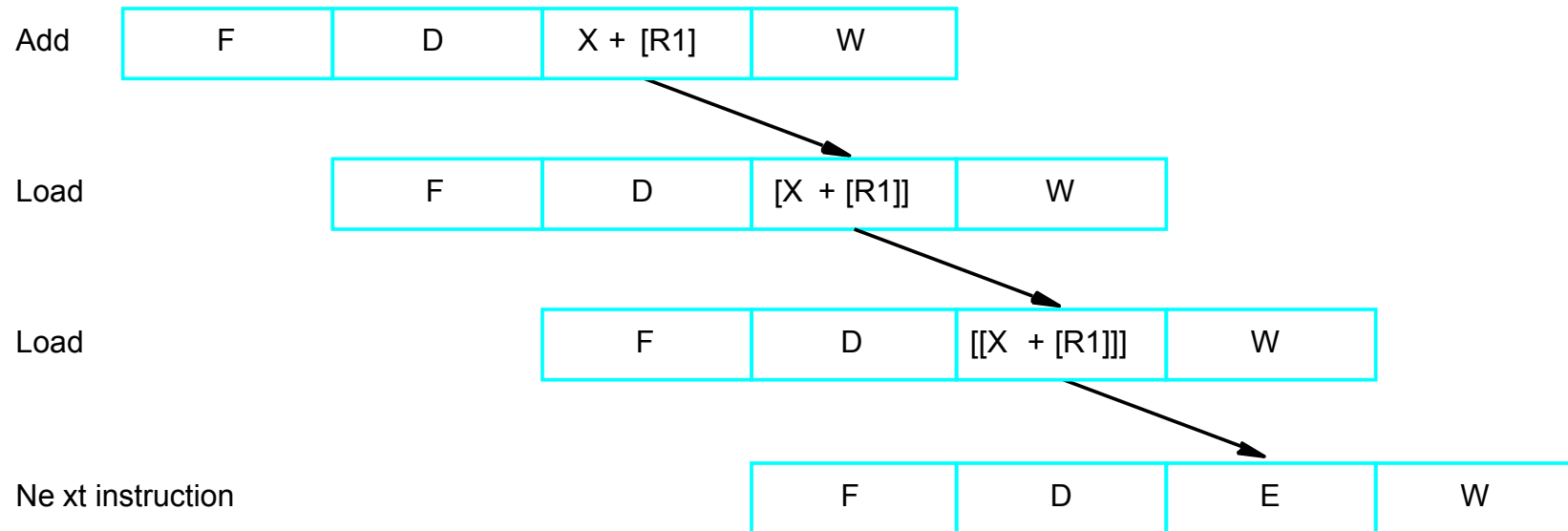
(a) Complex addressing mode

SIMPLE ADDRESSING MODE

Add #X, R1, R2

Load (R2), R2

Load (R2), R2



(b) Simple addressing mode

ADDRESSING MODES

- 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.

ADDRESSING MODES

- 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.

CONDITIONAL CODES

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.