

UNIT 3- PROCESSOR AND PARALLELISM



By Jacquelin Anushya.P

SYLLABUS



Unit - III

Fundamental concepts – Execution of a complete instruction – Multiple bus organization – Hardwired control – Micro programmed control – Pipelining: Basic concepts – Data hazards – Instruction hazards – Influence on Instruction sets – Data path and control consideration

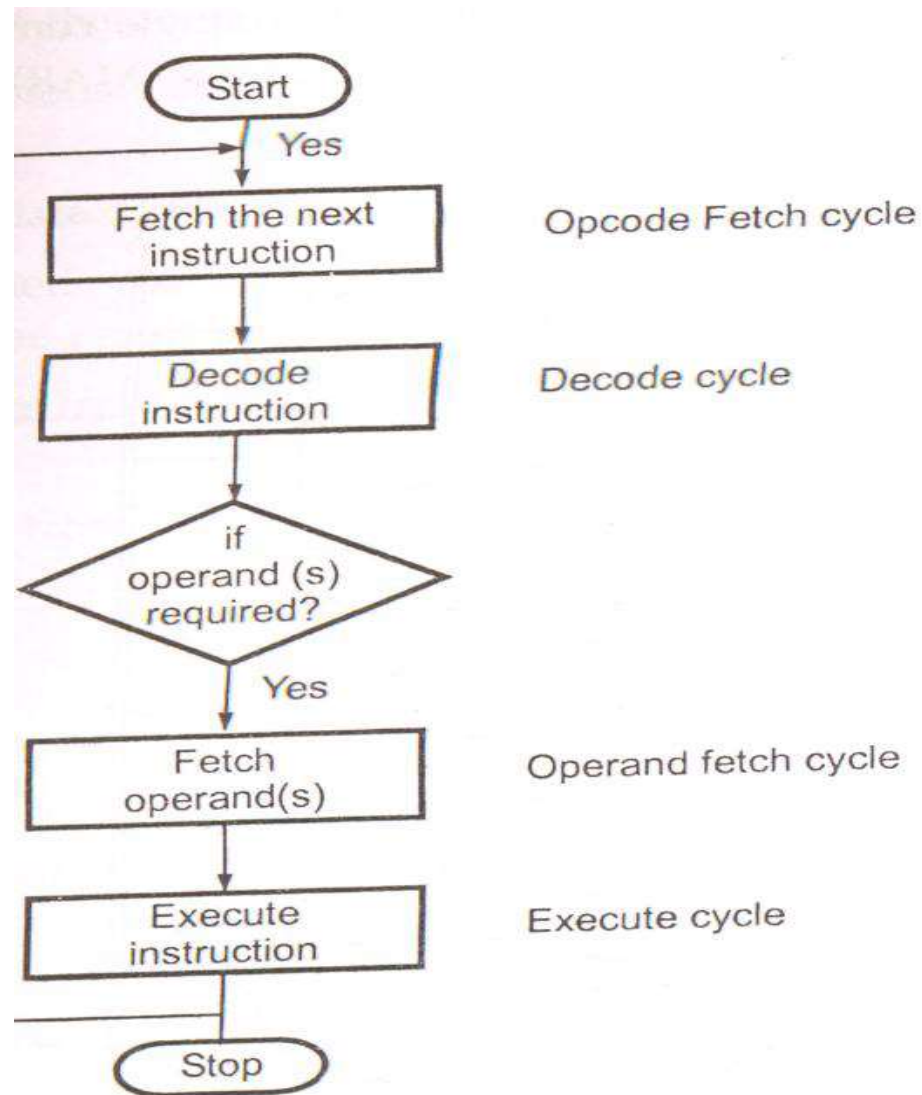
Basic fundamental concepts

Some fundamental concepts

- The **primary function of a processor unit is to execute sequence of instructions stored in a memory.**
- The sequence of operations involved in **processing an instruction constitutes an instruction cycle**, which can be subdivided into 3 major phases:-

1. Fetch cycle
2. Decode cycle
3. Execute cycle

Basic instruction cycle



g. 3.1 Basic instruction cycle

Basic instruction cycle

- To perform fetch, decode and execute cycles the processor unit has to **perform set of operations called micro-operations.**
- **Single bus organization** of processor unit shows how the building blocks of processor unit are organized and how they are interconnected.
- They can be organized in a variety of ways, in which the arithmetic and logic unit and all processor registers are connected through a single common bus.
- It also shows the external memory bus connected to memory address(MAR) and data register(MDR).

Single Bus Organization of processor

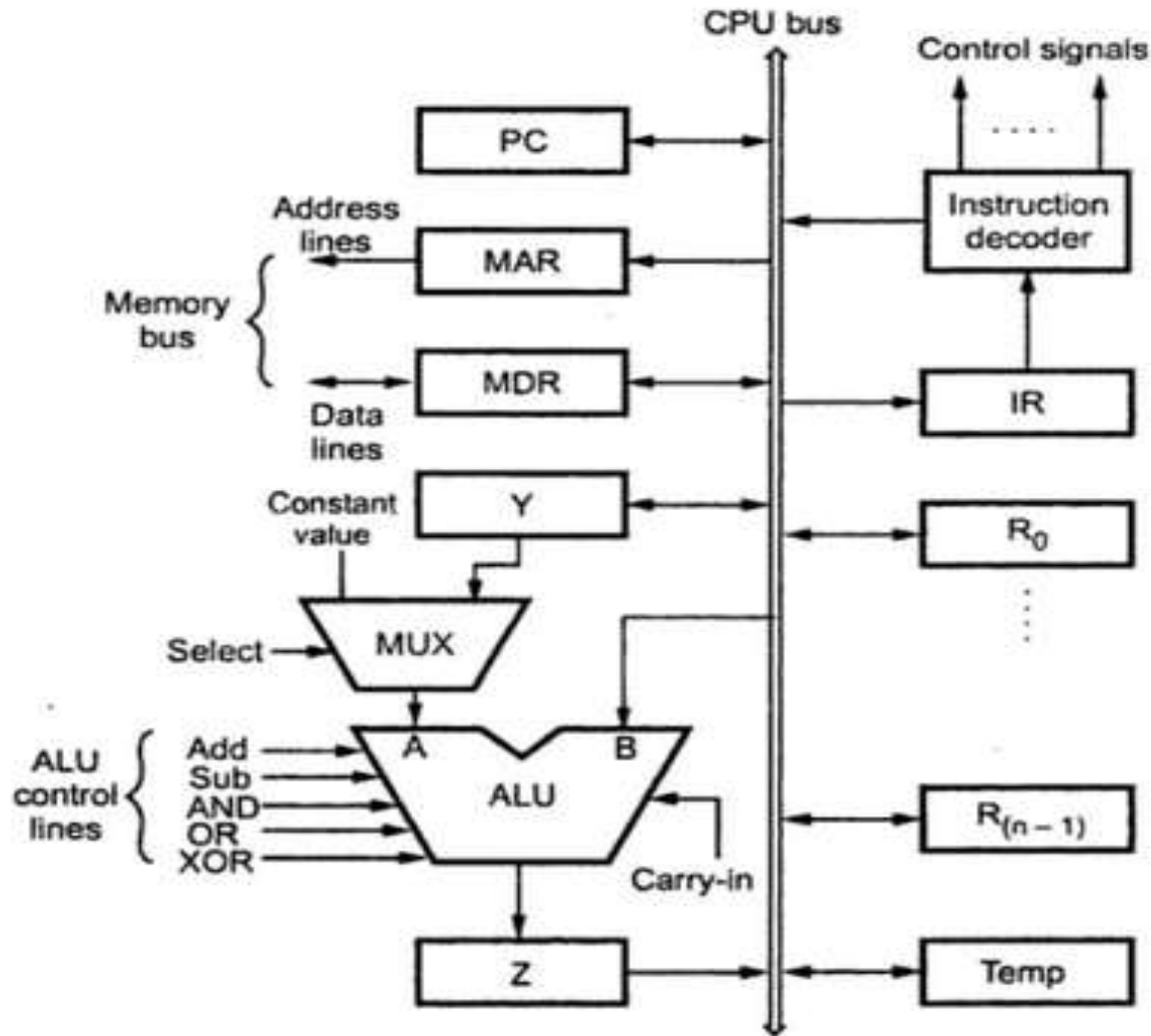


Fig. 3.30 Single bus organisation of processor

Single Bus Organization of processor

- ❑ The registers Y,Z and Temp are used only by the processor unit for temporary storage during the execution of some instructions.
- ❑ These registers are never used for storing data generated by one instruction for later use by another instruction.
- ❑ The programmer cannot access these registers.
- ❑ The IR and the instruction decoder are integral parts of the control circuitry in the processing unit.
- ❑ All other registers and the ALU are used for storing and manipulating data.
- ❑ The data registers, ALU and the interconnecting bus is referred to as data path.
- ❑ Register R0 through R(n-1) are the processor registers.
- ❑ The number and use of these register vary from processor to processor.

Single Bus Organization of processor

- ❑ These registers include general purpose registers and special purpose registers such as stack pointer, index registers and pointers.
- ❑ These are 2 options provided for A input of the ALU.
- ❑ The multiplexer(MUX) is used to select one of the two inputs.
- ❑ It selects either **output of Y register** or a **constant number as an A input for the ALU** according to the status of the select input.
- ❑ It **selects output of Y** when select input is **1** (select Y) and it **selects a constant number** when select input is **0**(select C) **as an input A for the multiplier.**
- ❑ The constant number is used to increment the contents of program counter.

Single Bus Organization of processor

- ❑ For the execution of various instructions processor has to perform one or more of the following basic operations:
 - a) Transfer a word of data from one processor register to the another or to the ALU.
 - b) Perform the arithmetic or logic operations on the data from the processor registers and store the result in a processor register.
 - c) Fetch a word of data from specified memory location and load them into a processor register.
 - d) Store a word of data from a processor register into a specified memory location.

Register Transfers

- How the data is transferred between register and common bus is indicated with the arrow heads.
- Each register has input and output gating and these gates are controlled by corresponding control signals.
- R_i in and R_i out \rightarrow controls the input and output gating of register R_i
- When R_i in \rightarrow $1 \rightarrow$ data is available on the common bus and loaded into R_i register.
- The signals R_i in and R_i out are commonly known as input enable and output enable signals of registers.

Register Transfers

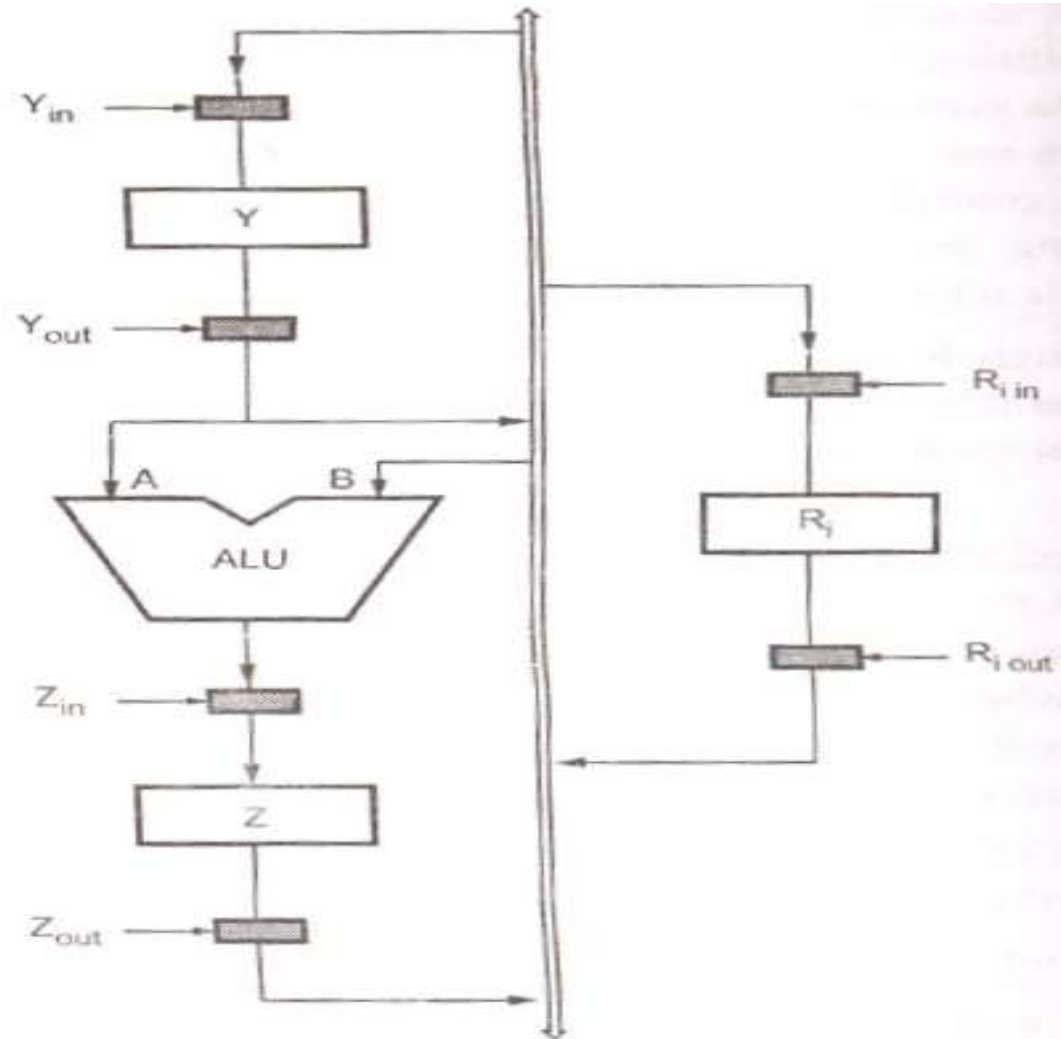
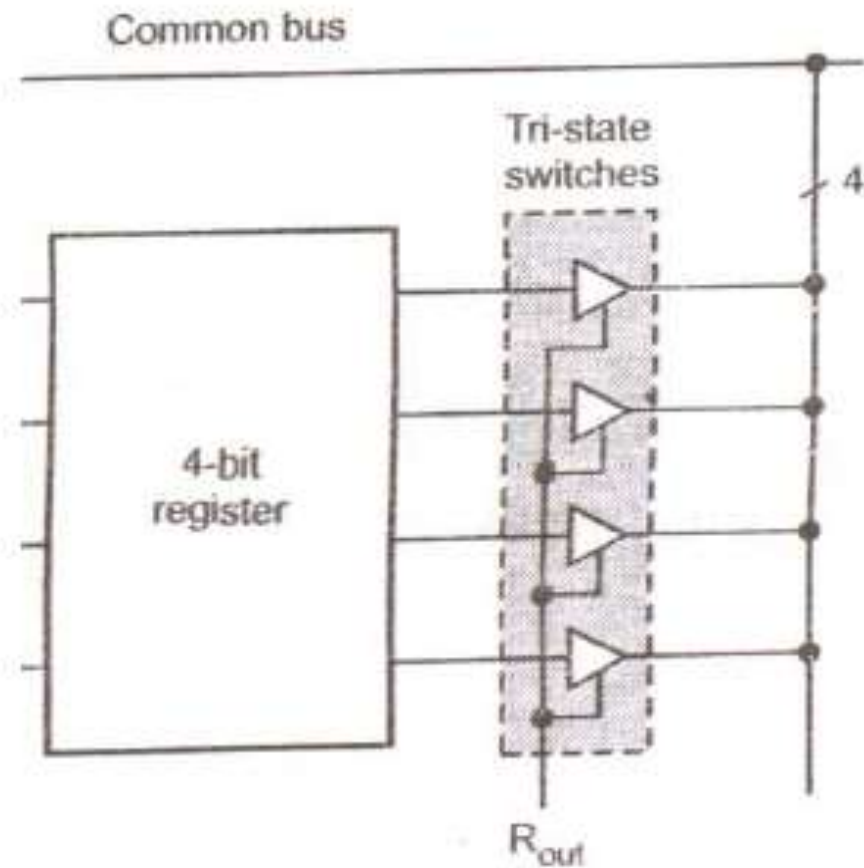


Fig. 3.3 Input and output gating for the register

Register Transfers

- ❑ The input and output gates are nothing but the **electronic switches which can be controlled by the control signals.**
- ❑ When **signal is 1, the switch is ON and when the signal is 0, the switch is OFF.**

Implementation of input and output gates of a 4 bit register



Implementation of input and output gates of a 4 bit register

Consider that we have transfer data from register R1 to R2

It can be done by,

- a. Activate the output enable signal of R1, $R1_{out}=1$. It places the contents of R1 on the common bus.
- b. Activate the input enable signal of R2, $R2_{in}=1$. It loads data from the common bus into the register R2.

Implementation of One-bit register

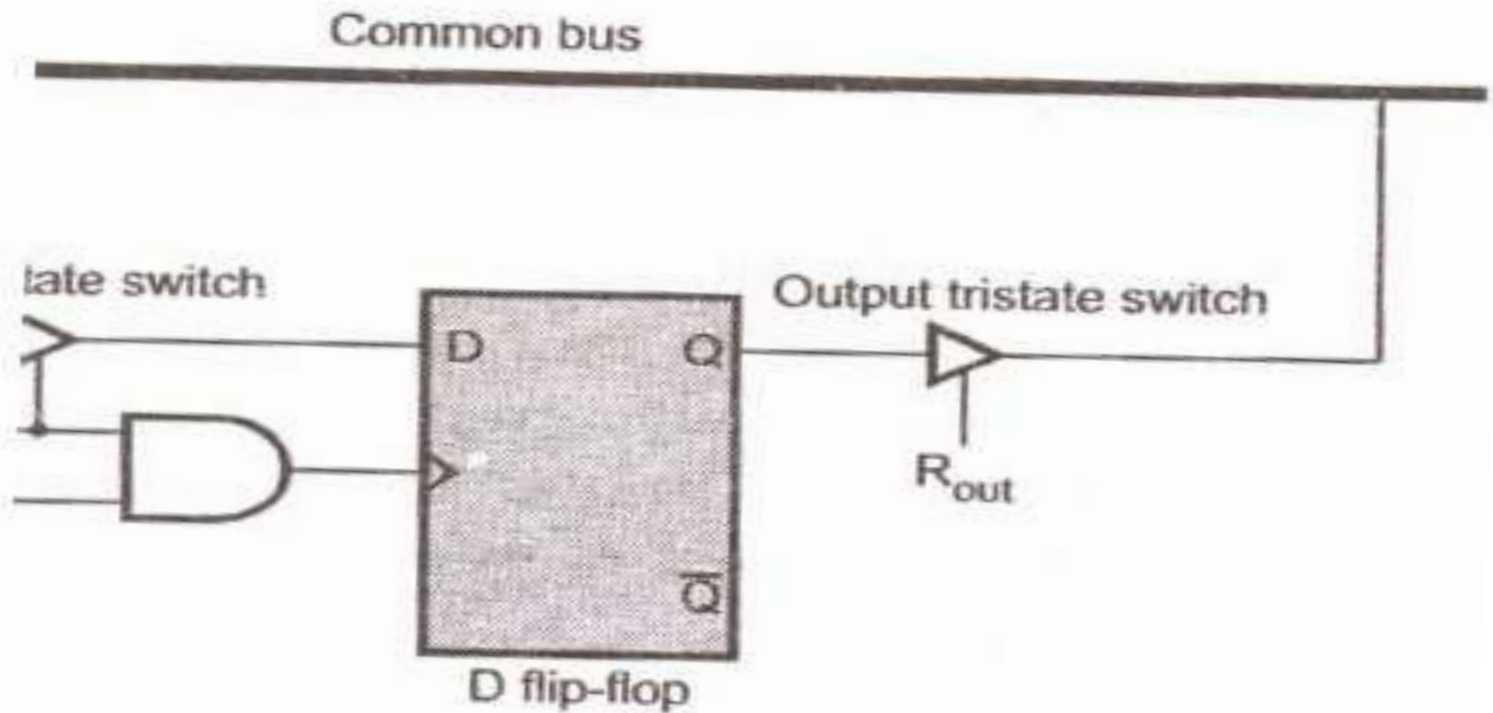


Fig. 3.5 One-bit register

One-bit register

- The edge triggered D flip-flop which stores the one-bit data is connected to the common bus through tri-state switches.
- Input D is connected through input tri-state switch and output Q is connected through output tri-state switch.
- The control signal R_{in} enables the input tri-state switch and the data from common bus is loaded into the D flip-flop in synchronization with clock input when R_{in} is active.
- It is implemented using AND gate .
- The control signal R_{out} is activated to load data from Q output of the D flip-flop on to the common bus by enabling the output tri-state switch.

2. Performing an arithmetic or logic operation

- ALU performs arithmetic and logic operations.
- It is a combinational circuit that has no internal memory.
- It has 2 inputs A and B and one output.
- It's A input gets the operand from the output of the multiplexer and its B input gets the operand directly from the bus.
- The result produced by the ALU is stored temporarily in register Z.

Example:-

- Let us find the sequence of operations required to subtract the contents of register R2 from register R1 and store the result in register R3.

This sequence can be followed as:

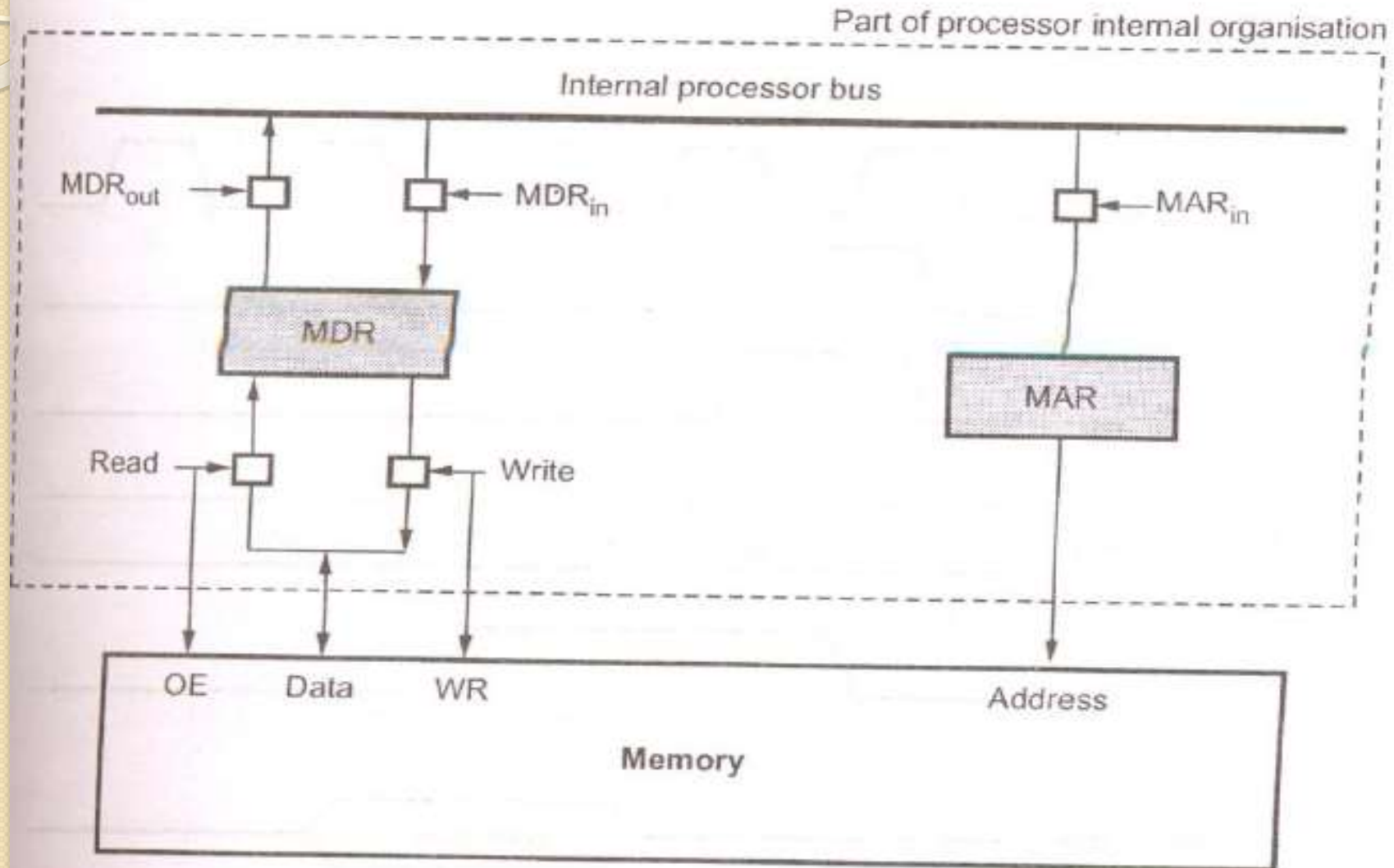
- a) R1 out, Yin
 - b) R2 out, Select Y, sub, Zin
 - c) Zout, R3 in
- **Step 1:** contents from register R1 are loaded into register Y.
 - **Step 2:** contents from Y and from register R2 are applied to the A and B inputs of ALU, subtraction is performed and result is stored in the Z register.
 - **Step 3:** The contents of Z register is stored in the R3 register.

3. Fetching a word from memory

- To fetch a word from memory the processor gives the address of the memory location where the data is stored on the address bus and activates the Read operation.
- The processor loads the required address in MAR, whose output is connected to the address lines of the memory bus.
- At the same time processor sends the Read signal of memory control bus to indicate the Read operation.
- When the requested data is received from the memory its stored into the MDR, then it can be transferred to other processor registers.

3. Fetching a word from memory

- Data ,address and control signals for data transfer between memory and processor



4. Storing a word in memory

- To write a word in memory location processor has to load the address of the desired memory location in the MAR, load the data to be written in memory, in MDR and activate write operation.
- Assume that we have to execute instruction Move(R2), R1.
- This instruction copies the contents of register R1 into the memory whose location is specified by the contents of register R2.

- The actions needed to execute this instruction are as follows:
 - a) $MAR \leftarrow [R2]$
 - b) $MDR \leftarrow [R1]$
 - c) Activate the control signal to perform the write operation. If memory is slow, wait for memory function complete(WMFC).

- The various control signals which are necessary to activate to perform given actions in each step.
 - a) $R2_{out}, MAR_{in}$
 - b) $R1_{out}, MDR_{in}$
 - c) $MAR_{out}, MDR_{out}, Write$

EXECUTION OF A COMPLETE INSTRUCTION

- Let us find the complete control sequence for execution of the instruction **Add (R3),R1** for the single bus processor.
- This instruction adds the contents of register R1 and the contents of memory location specified by register R3 and stores results in the register R1.
- To execute bus instruction it is necessary to perform following actions:
 1. Fetch the instruction
 2. Fetch the operand from memory location pointed by R3.
 3. Perform the addition
 4. Store the results in R1.

- The sequence of control steps required to perform these operations for the single bus architecture are as follows:

1. PCout, MARin , Read ,Add, Zin

2. Zout, PCin, Yin, WMFC

3. MDRout, IRin


4. R3out , MARin ,Read

5. R I out , Yin, WMFC


6. MDRout , select Y,Add, Zin

7. Zout, R I in,End

- **Step 1**, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory.
- The select signal is set to select4, which causes the multiplexer MUX to select the constant4.
- This value is added at the operand at input B, which is the content of the PC, and result is stored in the Z register.
- The updated value is moved from the register Z back into the PC during **step 2**, while waiting for the memory to respond.
- In **step 3** the word is fetched from the memory is loaded into the IR.

- 
- Step 4: The instruction decoding circuit interprets the contents of the IR.
 - This enables the control circuitry to activate the control signals for step 4 through step 7, which constitutes the execution phase.
 - The content of register R3 are transferred to the MAR in step 4 and a memory read operation is initiated.
 - Then the content of R1 are transferred to register Y.


- When the Read operation is completed ,the memory operand is available in register MDR and the addition operation is performed in step 6.
- The content of the MDR are gated to the bus, and thus also to the B input of the ALU, register Y is selected as the second input to the ALU by choosing select Y.
- The sum is stored in register Z then transferred to RI.
- The end signal causes a new instruction fetch cycle to begin by returning to step 1.

- 
- There is no need to copy the updated contents of PC into register Y when executing the Add instruction.
 - But in branch instructions the updated value of the PC is needed to compute the branch target address.
 - To speed up the execution of branch instruction this value is copied into register Y
 - This does not cause any harm because register Y is not used for any other purpose at that time.

Branch Instruction

- The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address.
- The branch target address is usually obtained by adding the offset in the contents of PC. The offset is specified within the instruction.

- The control sequence for unconditional branch instruction is as follows:
- 1. PCout, MARin, Read, Select4, Add, Zin
- 2. Zout, PCin, Yin, WMFC
- 3. MDRout, IRin
- 4. Offset_field_Of_IRout, SelectY, Add, Zin
- 5. Zout, PCin

- 
- First 3 steps are same as in the previous example.
 - Step 4: The contents of PC are transferred to register Y by activating PCout and Yin signals.
 - Step 5: The contents of PC and the offset field of IR register are added and result is saved in register Z by activating corresponding signals.
 - Step 6: The contents of register Z are transferred to PC by activating Zout and PCin signals.

Multiple Bus Organization

- Single bus only one data word can be transferred over the bus in a clock cycle.
- This increases the steps required to complete the execution of the instruction.
- To reduce the number of steps needed to execute instructions, most commercial process provide multiple internal paths that enable several transfer to take place in parallel.
- This reduces the number of steps needed to execute instructions reducing the execution time.

Multiple Bus Organization

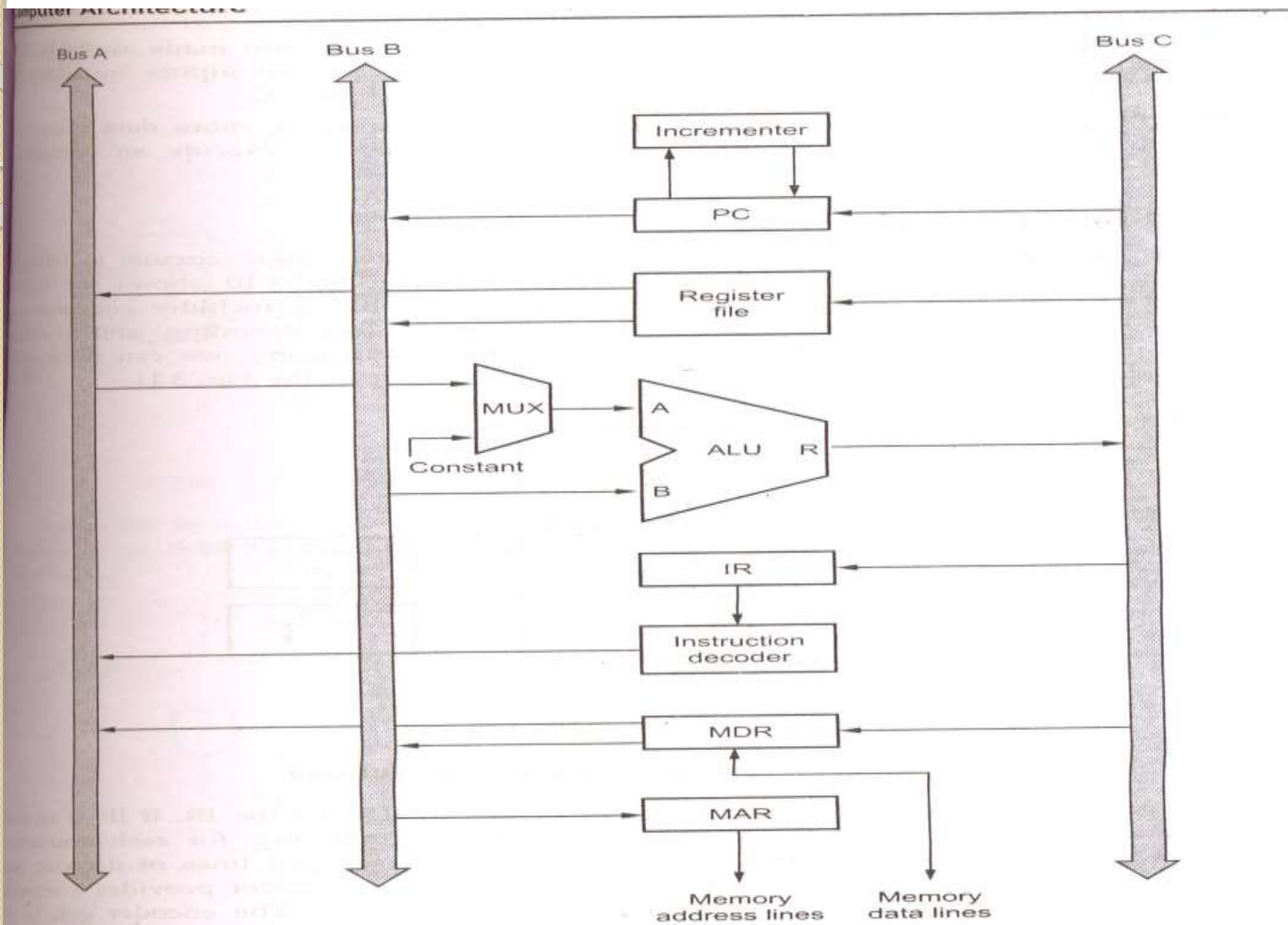




Fig. 3.9 Three-bus organisation of processor

Structure

- 3 buses are used to connect registers and the ALU of the processor.
- All general purpose registers are shown by a single block called register file.
- Register file has 3 ports, one input port and two output ports.
- So it is possible to access data of three register in one clock cycle, the value can be loaded in one register from bus C and data from two register can be accessed to bus A and bus B.
- Buses A and B are used to transfer the source operands to the A and B inputs of the ALU.
- After performing arithmetic or logic operation result is transferred to the destination operand over bus C.

- 
- If needed, the ALU may simply pass one of its two inputs operands unmodified to bus C.
 - We will call the ALU control signals for such operations $R=A$ or $R=B$.
 - The three bus arrangement obviates the need for registers Y and Z.
 - Second feature is the introduction of the incrementer unit, which is used to increment the PC by 4

- 
- To increment the contents of PC after execution of each instruction to fetch the next instruction, separate unit is provided. This unit is known as **incrementer**.
 - Incrementer increments the contents of PC accordingly to the length of the instruction so that it can point to next instruction in the sequence.
 - The incrementer eliminates the need of multiplexer connected at the A input of ALU.
 - It can be used to increment other addresses, such as the memory addresses in LoadMultiple and StoreMultiple instructions.

- Let us consider the execution of 3 operand instruction, Add, R1, R2, R3.
- This instruction adds the contents of registers R2 and the contents of register R3 and stores the result in R1.
- With 3 bus organization control steps required for execution of instruction Add R1, R2, R3 are as follows:
 1. PCout, R=B, MARin, Read, IncPC
 2. MARout, MDRin, Read
 3. WMFC
 3. MDRoutP, R=B, IRin
 4. R2outA, R3outB, Add, R1in, end

- Step 1: The contents of PC are transferred to MAR through bus B using $R=B$ control signal to start the Read operation and simultaneously PC is incremented to point the next instruction.
- At the same time PC is incremented by 4.
- Note that the value loaded into MAR is the original contents of the PC.
- The incremented value is loaded into PC at the end of each clock cycle and value will not affect the contents of MAR.
- Step 2: The processor waits for WMFC and loads the data into MDR and then transfers to IR in step 3
- In step 4, two operands from register R2 and register R3 are made available at A and B inputs of ALU through BUS A and BUS B.
- These two inputs are added by activation of Add signal and result is stored in RI through BUS C.

Hardwired control

- To execute the instructions, the processor must generate the control signals that is needed in the proper sequence.
- Computer designers use a wide variety of technique to solve this problem
- There are two categories
 - Hardwired control
 - Microprogrammed control

Hardwired control

- The required control signals are determined by the following information
 - Contents of the control step counter
 - Contents of the instruction register
 - Contents of the condition code flags
 - External input signals, such as MFC and interrupt request

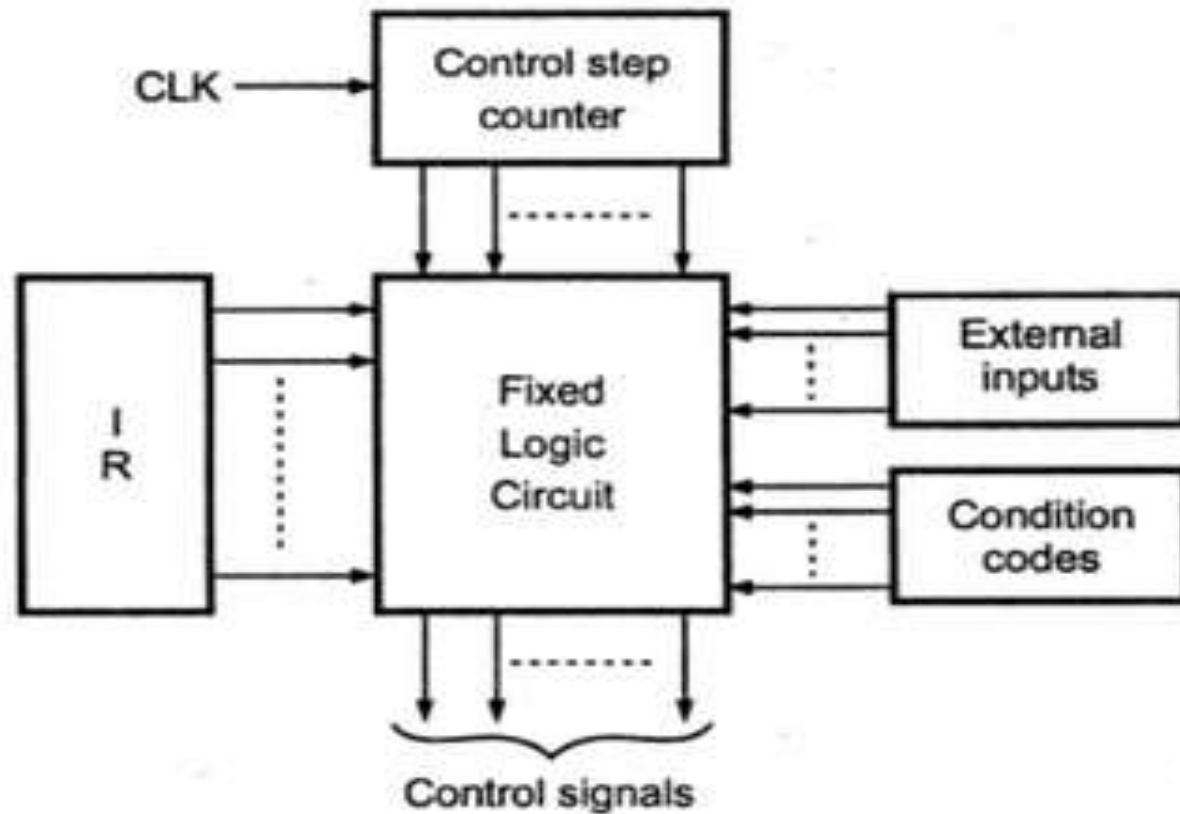


Fig. 3.10 Typical hardwired control unit

- The control units use fixed logic circuits to interpret instructions and generate control signals from them.
- The fixed logic circuit block includes combinational circuit that generates the required control outputs by decoding and encoding functions.

Separation of decoding and encoding function

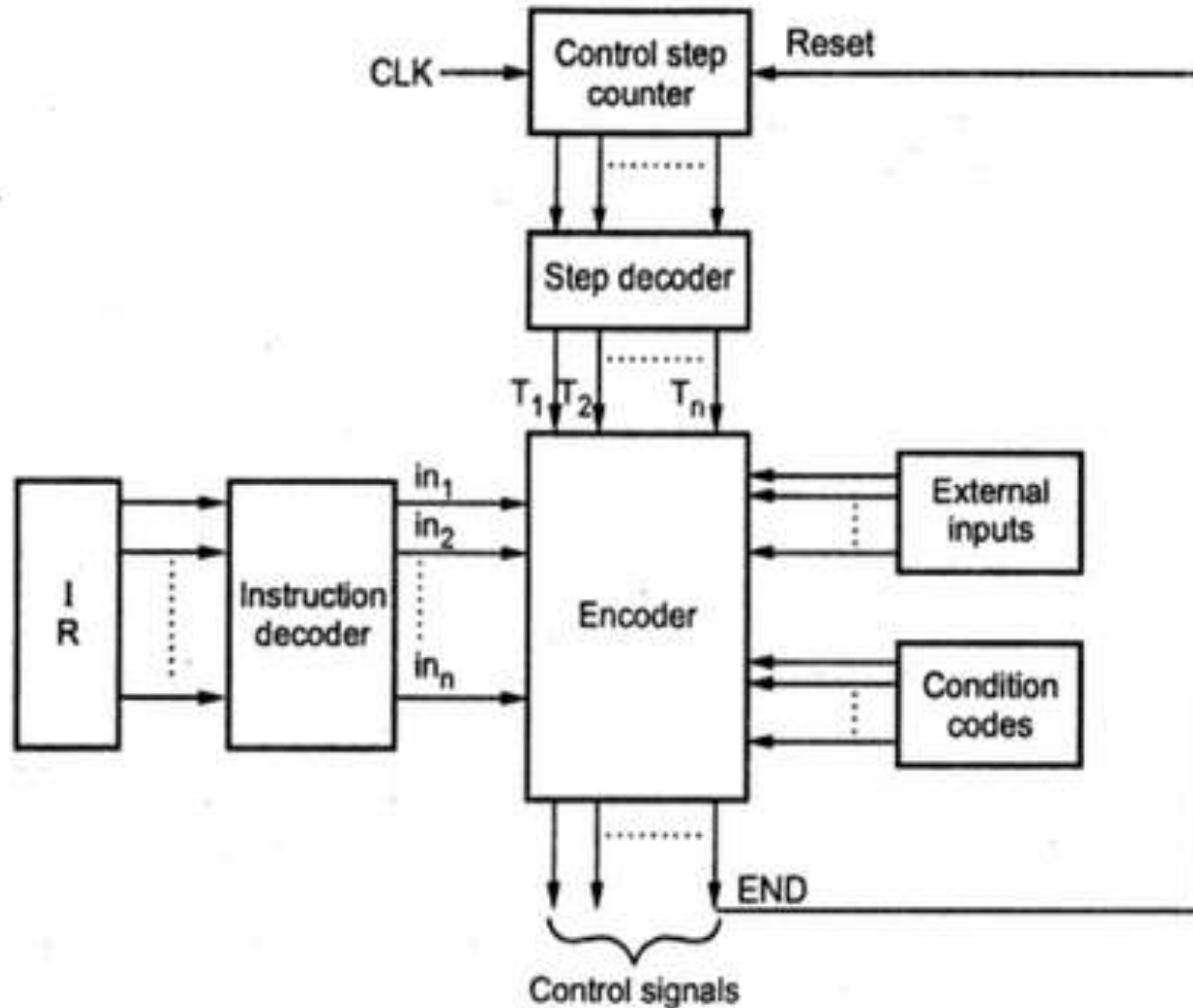


Fig. 3.11 Detail block diagram for hardwired control unit

Separation of decoding and encoding function

Instruction Decoder

- The output of each instruction decoder consists of a **separate line for** each machine instruction.
- For any instruction loaded in IR, one of the **output lines IRS_1 through IRS_m is set to 1 (one)** and all the **other lines are 0 (zero)**.
- The **input signals to the encoder block** are combined to generate individual control signals **Y_{in} , P_{cout} , Add, End** and so on.

Step decoder

- It provides a **separate signal line for each step, or time slot**, in a control sequence.

Encoder

- It gets in the input from instruction decoder, step decoder, external inputs and condition codes.
- It uses all these inputs to generate the individual control signals.
- After execution of each instruction **END signal is generated** this resets control step counter and make it ready for generation of control signal for next instruction.

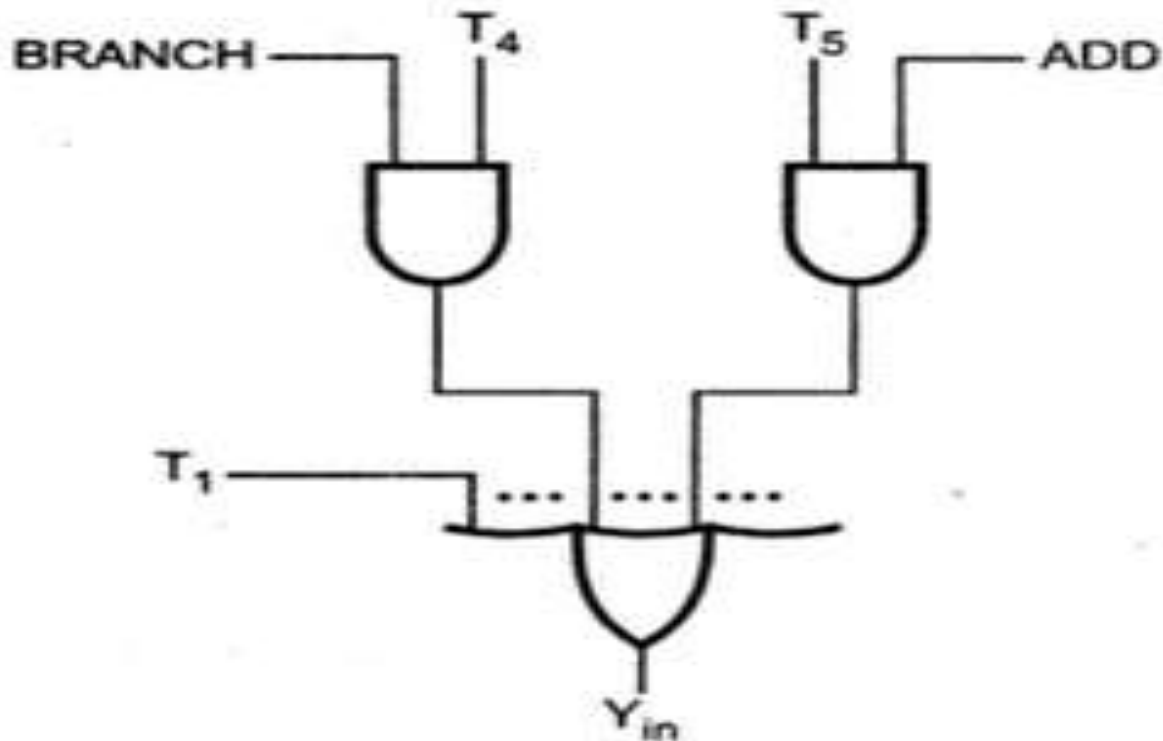
- Let us see the how the encoder generates the Y_{in} control signal for the processor organization.

- The encoder circuit implements the following logic function to generate Y_{in}

$$Y_{in} = T1 + T5.ADD + T4.BRANCH + \dots$$

- The Y_{in} signal is asserted during time interval T1 for all instructions, during T5 for an ADD instruction, during T4 for an unconditional branch instruction, and so on.

Generation of the Y_{in} control signal for the processor



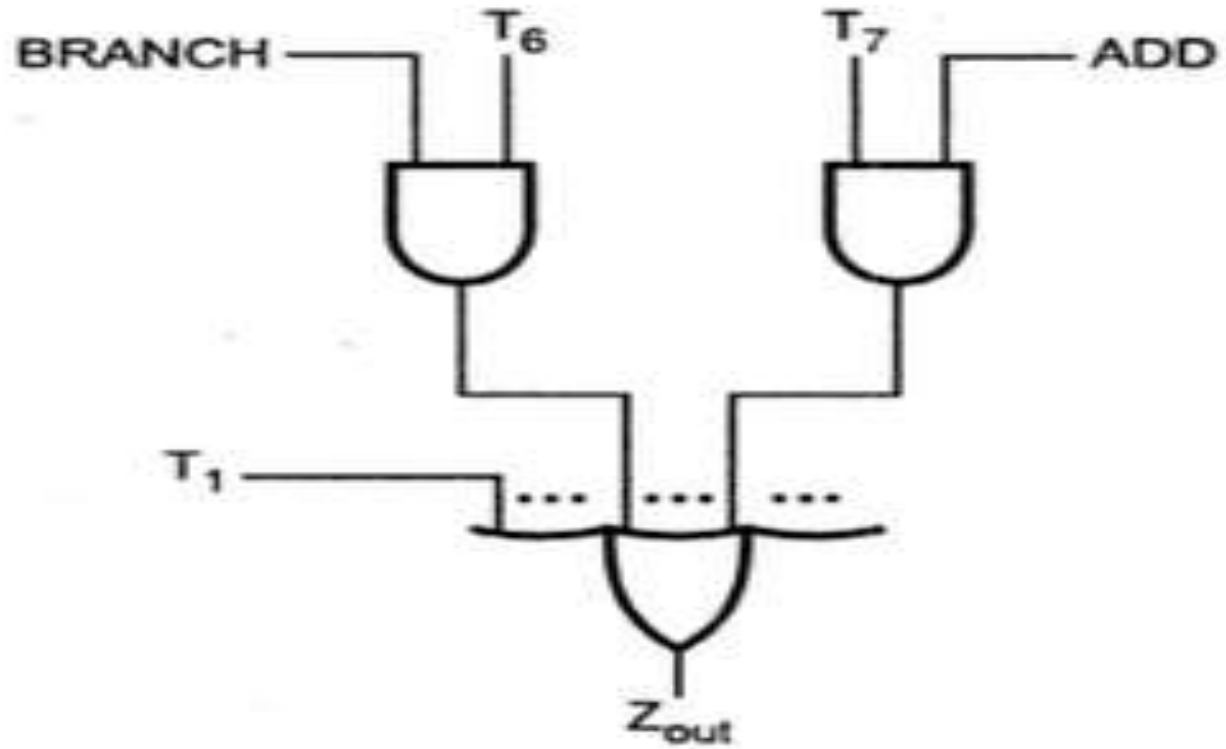
Generation of the Y_{in} control signal

Fig. 3.12

Contd.,


- As another example, the logic function to generate Z_{out} signal can given by
- **$Z_{out} = T2 + T7 \cdot ADD + T6 \cdot BRANCH + \dots$**
- The Z_{out} signal is asserted during time interval T2 of all instructions, during T7 for an ADD instruction, during T6 for an unconditional branch instruction, and so on.


Generation of the Zout control signal for the processor



Generation of the Z_{out} control signal

Fig. 3.13

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

- 
- The **control hardware** can be viewed as a **state machine** that changes from **one state to the another in every clock cycle**, depending on the contents of the instruction register, the conditional codes, and the external inputs.
 - The **output of the state machine are the control signals**.
 - The sequence of the operations carried out by this machine is determined by the wiring of the logical elements, hence the name **“Hardwired”**. (All your logical components are connected by wires)
 - A controller that uses this approach can operate at high speed.
 - However , it has **little flexibility** and the complexity of the instruction set it can implemented is limited.

Advantages of hardwired control

- It is fast because control signals are generated by combinational circuits.
- The delay in generation of control signals depends upon the number of gates.
- It has greater chip area efficiency since it uses less area on-chip.

Disadvantages

- More the control signals required by CPU, more complex will be the design of control unit.
- Modifications in control signal are very difficult. That means it requires rearranging of wires in the hardware circuit.
- It is difficult to correct mistake in original design or adding new feature in existing design of control unit.

A Complete processor

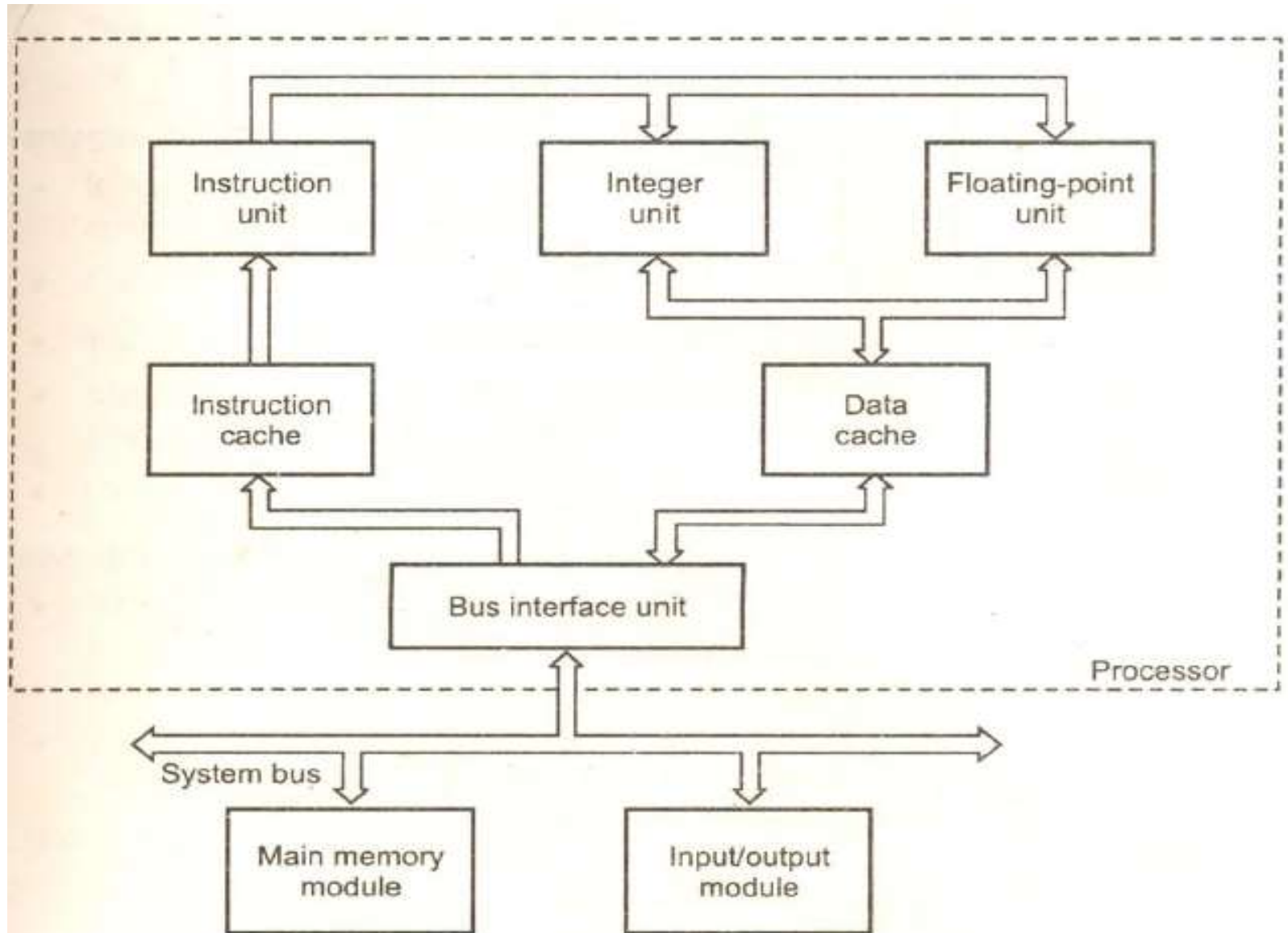


Fig. 3.14 Block diagram of a complete processor

A Complete processor

- It consists of
 - Instruction unit
 - Integer unit
 - Floating-point unit
 - Instruction cache
 - Data cache
 - Bus interface unit
 - Main memory module
 - Input/Output module.

A Complete processor

- **Instruction unit-** It fetches instructions from an instruction cache or from the main memory when the desired instructions are not available in the cache.
- **Integer unit –** To process integer data
- **Floating unit –** To process floating –point data
- **Data cache –** The integer and floating unit gets data from data cache
- The **80486 processor has 8-kbytes single cache** for both instruction and data **where as the Pentium processor has two separate 8-kbytes caches** for instruction and data.
- The processor provides bus interface unit to control the interface of processor to system bus, main memory module and input/output module.

Microprogrammed control

- The control signals are generated by a *program similar to machine language program*.
- *Control word(CW)*-It is word whose individual bits represent the various control signals, which defines a unique combinations of 1's and 0's.

Microprogrammed control

- A sequence of CWs corresponding to the control sequence-**microroutine**
- Individual control words in this microroutine are referred to as **microinstructions**
- The micro instruction for all instructions in the **instruction set of a computer are stored in a special memory called *control store*.**
- The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding microroutine from the control store.



Organization of a microprogrammed control unit

Organization of a microprogrammed control unit

- To read the control words sequentially from the control store a *microprogram counter*(μpc) is used.
- Every time a new instruction is loaded into IR, the output of block labeled “*starting address generator*” is loaded into the μpc .
- The μpc read the microinstruction from the control store is then automatically incremented by the clock.
- Hence the control signals are delivered to the various parts of the processor in correct sequence.

Organization of a microprogrammed control unit

- There is a situation arises when the control unit is required to check the status of the conditional codes or external inputs.
- In case of Hardwired control, this situation is handled by including appropriate logic function.
- In **microprogrammed control**, an alternative approach is to use **conditional branch microinstructions**.
- In addition to the branch address ,these microinstructions specify which of the external inputs, conditional codes should be checked.

Organization of a microprogrammed control unit

Address	Microinstruction
0	PC_{out} , MAR_{in} , Read, Y_{in} , SelectC, Add, Z_{in}
1	Z_{out} , PC_{in} , WMFC
2	MDR_{out} , IR_{in}
3	Branch to starting address of appropriate microroutine
45	If $N=0$, then branch to microinstruction 0
46	offset_field_of_ IR_{out} , SelectY, Add, Z_{in}
47	Z_{out} , PC_{in} , End

Fig. 3.28 Microroutine for the instruction branch < 0



Organization of control unit to allow conditional branching in the microprogram

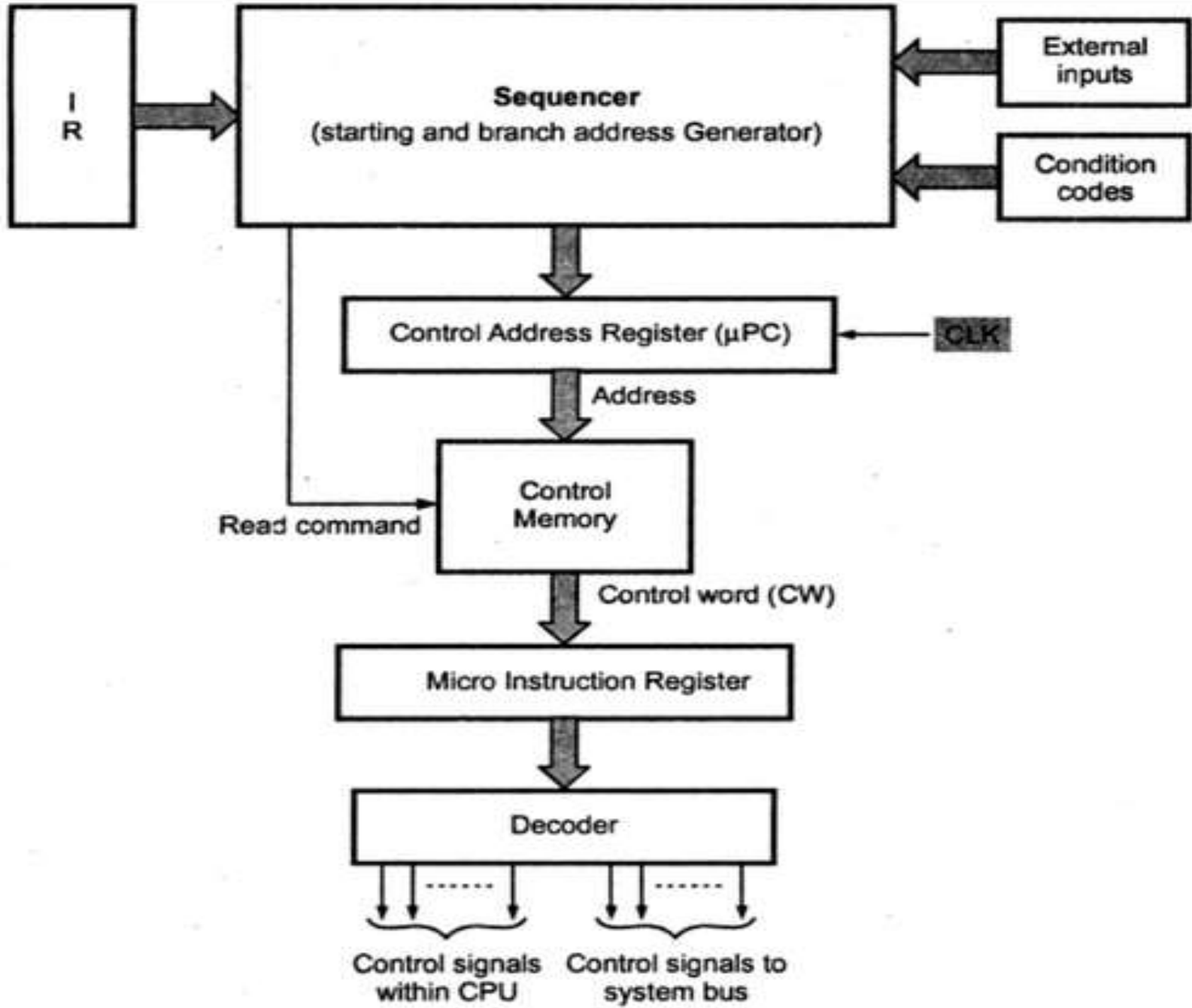


Fig. 3.27 Microprogrammed control unit

Organization of control unit to allow conditional branching in the microprogram

- The starting address generator block becomes the *starting and branch address generator*.
- This block loads a new address into μPC when microinstruction instructs it to do so.
- To *allow the implementation of a conditional branch*, inputs to this block consist of the *external inputs and conditional codes as well as the contents of the IR*.

- In this control unit ,the *μpc is incremented every time a new microinstruction* is fetched from the microprogram memory, expect in the following situations:
 - When a new instruction is loaded into IR, the *μpc is loaded with the starting address of the microroutine for that instruction.*
 - When a branch microinstruction is encountered and the branch condition is satisfied ,the μpc is loaded with the branch address.
 - When an END microinstruction is encountered, the μpc is loaded with the address of the first CW in the microroutine for the next instruction fetch cycle.

Advantages

- It simplifies the design of control unit. Thus it is both, cheaper and less error prone to implement
- Control functions are implemented in software rather than hardware.
- The design process is orderly and systematic.
- More flexible and can accommodate in new system and correct the design errors quickly and cheaply.
- Complex function such as floating point arithmetic can be realized easily.

Disadvantages

- It is somewhat slower than the hardwired control unit, because time is required to access the micro instructions from CM
- The design duration of microprogram control unit is more than hardwired control unit for smaller CPU.

Microinstruction

- A simple way to structure microinstructions is to assign one bit position to each control signal required in the CPU.
- But it is drawback –assigning individual bits to each control signal results in long micro instructions because the number of required signals is usually large.
- Moreover ,only a few bits are set to 1 in any given micro instruction, which means the available bit space is poorly used.

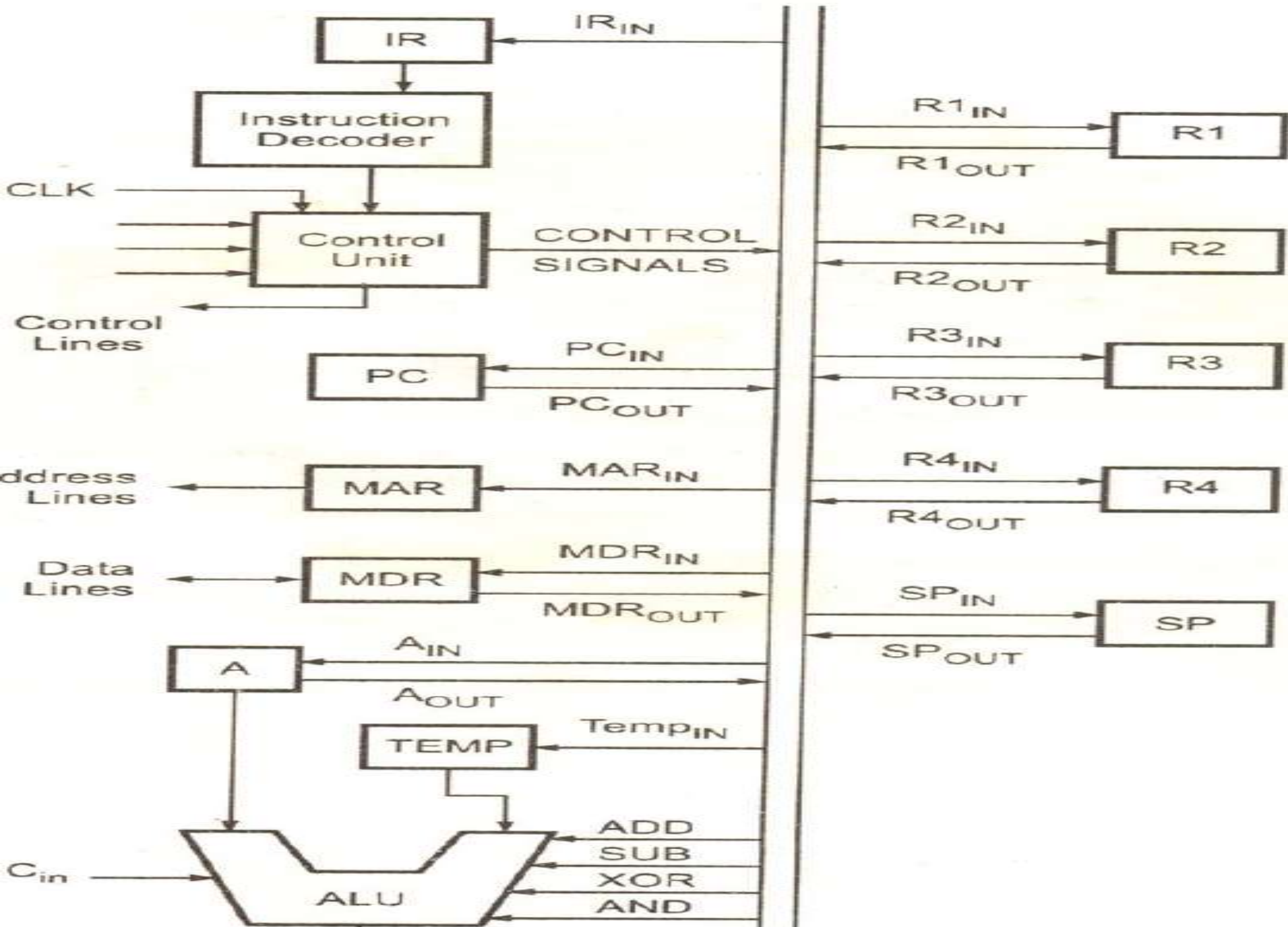
Grouping of control signals

- Grouping technique is used to *reduce the number of bits in the microinstruction.*
- **Gating signals:** IN and OUT signals
- **Control signals:** Read, Write, clear A, Set carry in, continue operation, end, etc.
- **ALU signals:** Add, Sub, etc.,
- There are 46 signals and hence each microinstruction will have 46 bits.
- It is not at all necessary to use all 46 bits for every microinstruction because by using grouping of control signals we minimize number of bits for microinstruction.

Way to reduce number of bits microinstruction:

- Most signals are not needed simultaneously.
- Many signals are mutually exclusive
 - For Eg:- only one function of the ALU can be activated at a time.
- Data transfer must be unique because it is not possible to gate the contents of two registers onto the bus at the same time.
- Read and write signals cannot be active simultaneously.
 - This suggests that signals can be grouped so that all mutually exclusive signals are placed in a same group.

- For Eg:- Register output control signals can be placed in a group consisting of Pc_{out} , MDR_{out} , Z_{out} , $R0_{out}$, $R1_{out}$, $R2_{out}$, $R3_{out}$ and $Temp_{out}$ any of these can be selected by a unique 4-bit code.



- 46 control signals can be grouped in 7 different groups.

G_1 (4 Bits) : IN grouping		G_2 (4 Bits) : OUT grouping	
0000	No Transfer	0000	No Transfer
0001	IR_{IN}	0001	PC_{OUT}
0010	PC_{IN}	0010	MDR_{OUT}
0011	MDR_{IN}	0011	R_{1OUT}
0100	MAR_{IN}	0100	R_{2OUT}
0101	A_{IN}	0101	R_{3OUT}
0110	$Temp_{IN}$	0110	R_{4OUT}
0111	R_{1IN}	0111	SP_{OUT}
1000	R_{2IN}		
1001	R_{3IN}		
1010	R_{4IN}		
1011	SP_{IN}		

G₃ (4 bits) : ALU Functions	G₄ (2 Bits) : RD/WR Control Signals
0000 ADD 0001 SUB ⋮ ⋮ 1111 XOR	00 No Action 01 Read 10 Write
} 16 ALU Functions	

G₅ (1 Bit) : A Register	G₆ (1 Bit) : Carry
0 - No action	0 - Carry in to ALU = 0
1 - Clear A	1 - Carry in to ALU = 1
G₇ (1 bit) :	G₈ (1 bit) : Operation
0 : No action	0 : Continue Operation
1 : WMFC	1 : End

- 
- The total number of grouping bits are 17. Therefore, we minimized 46 bits microinstruction to 17 bit microinstruction.

Techniques for grouping of control signals

- The grouping of control signal can be done either by using technique called vertical organization or by using technique called horizontal organization.

Vertical organization

- Highly encoded scheme that can be compact codes to *specify only a small number of control functions* in each microinstruction are referred to as a vertical organization.

Horizontal organization

- The minimally encoded scheme, in which *resources can be controlled with a single instruction* is called a horizontal organization.

Comparison

S.No	Horizontal	Vertical
1.	Long formats	Short formats
2.	Ability to express a high degree of parallelism	Limited ability to express parallel microoperations
3.	Little encoding of the control information	Considerable encoding of the control information
4.	Useful when higher operating speed is desired	Useful when slower operating speeds is desired

Advantages of vertical and horizontal organization

- Vertical approach is the significant factor, it is used to reduce the requirement for the parallel hardware required to handle the execution of microinstructions.
- Less bits are required in the microinstruction.
- The horizontal organization approach is suitable when operating speed of computer is a critical factor and where the machine structure allows parallel usage of a number of resources.

Disadvantages

- Vertical approach results in slower operations speed.

Comparison

Attribute	Hardwired Control	Microprogrammed Control
Speed	Fast	Slow
Control functions	Implemented in hardware	Implemented in software
Flexibility	Not flexible to accommodate new system specifications or new instructions	More flexible, to accommodate new system specification or new instructions redesign is required
Ability to handle large/complex instruction sets	Difficult	Easier
Ability to support operating systems and diagnostic features	Very difficult	Easy


Comparison

Attribute	Hardwired Control	Microprogrammed Control
Design process	Complicated	Orderly and systematic
Applications	Mostly RISC microprocessors	Mainframes, some microprocessors
Instruction set size	Usually under 100 instructions	Usually over 100 instructions
ROM size	-	2K to 10K by 20-400 bit microinstructions
Chip area efficiency	Uses least area	Uses more area

Microprogram sequencing

The task of microprogram sequencing is done by *microprogram sequencer*.

- 2 important factors must be considered while designing the microprogram sequencer:
 - a) The size of the microinstruction
 - b) The address generation time.
- The *size of the microinstruction should be minimum* so that the size of *control memory required to store microinstructions is also less*.
- This reduces the *cost of control memory*.
- With *less address generation time*, microinstruction can be executed in *less time resulting better throughput*.

- 
- During execution of a microprogram the address of the next microinstruction to be executed and it has 3 sources:
 - i. Determined by instruction register
 - ii. Next sequential address
 - iii. Branch
 - Microinstructions can be shared using microinstruction branching.

Consider instruction **ADD src, Rdst.**

- The instruction adds the source operand to the contents of register Rdst and places the sum in Rdst, the destination register.
- Let us assume that the source operand can be specified in the following addressing modes:
 - a) Indexed
 - b) Autoincrement
 - c) Autodecrement
 - d) Register indirect
 - e) Register direct

We now use this instruction in conjunction with the processor structure.

Techniques for modification or generation of branch addresses

- **Bit-ORing**

- The branch address is determined by ORing particular bit or bits with the current address of microinstruction.

Eg: If the current address is 170 and branch address is 172 then the branch address can be generated by ORing 02(bit 1), with the current address. This is known as the bit-Oring technique for modifying branch addresses.

- **Using condition variables**

- It is used to modify the contents CM address register directly, thus eliminating whole or in part the need for branch addresses in microinstructions.

Eg: Let the condition variable CY indicate occurrence of $CY = 1$, and no carry when $CY = 0$.

- Suppose that we want to execute a SKIP_ON_CARRY microinstruction.
- It can be done by logically connecting CY to the count enable input of μpc at an appropriate point in the microinstruction cycle.
- It allows the overflow condition increment μpc an extra time, thus performing the desired skip operation.

Wide branch addressing

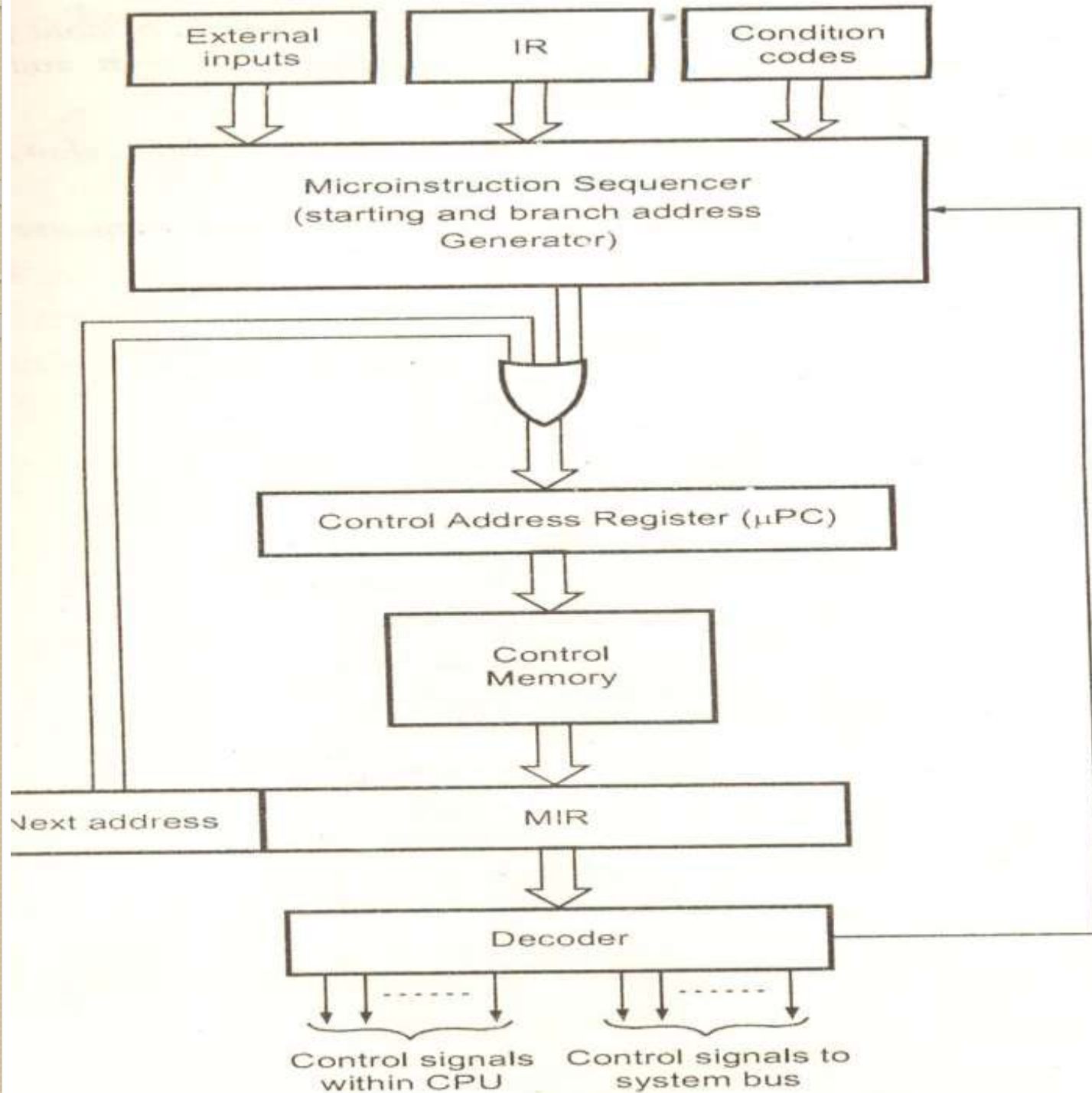
- Generating branch addresses becomes more difficult as the *number of branches increases*.
- In such situations *programmable logic array* can be used to generate the required branch addresses.
- The simple and inexpensive way of generating branch addresses is known as wide-branch addressing.


Wide branch addressing

- The opcode of a machine instruction is translated into the starting address of the corresponding micro-routine.
- This is achieved by connecting the opcode bits of the instruction register as inputs to the PLA , which acts as a decoder.
- The output of the PLA is the address of the desired microroutine

Microinstructions with next address field


- These micro instructions perform no useful operation in the data path; they are needed only to determine the address of the next instruction.
- Thus they reduce the operating speed of the processor.
- This situation can become significantly worse when other micro routines are considered.
- This problem is solved by providing special address field for branch address.



- 
- Here the branch address is stored in the special address field within the microinstruction.
 - The branch address is loaded into CM address register when a branch condition is satisfied.
 - The special address field within the microinstruction increases the size of the microinstruction. This can be reduced by storing part of the address (low order bits) in the microinstruction.
 - This restricts the range of branch instructions to a small region of the CM and may therefore increase the difficulty of writing some micro program.

Nano Programming

- In most microprogrammed processors ,an instruction fetched from memory is interpreted by a microprogram stored in a single control memory CM.
- However, in few microprogrammed processors ,the micro instruction do not directly used by the decoder to generate control signals.
- Instead they are used to access second control memory called a *nano control memory(nCM)*.

- 
- So there are two levels of control memories, a higher-level one known as microcontrol memory(μ CM) and lower level one known as nano control memory(nCM).
 - μ CM-stores the microinstructions
 - nCM-stores the nanoinstructions
 - Here, decoder uses the nanoinstrucions from μ CM to generate control signals.

From sequencer



Micro program counter(μ PC)



Micro control memory(μ CM)



Micro instruction Register(μ IR)



Nano program counter(nPC)



Nano control memory(nCM)



Nano Instruction Register(nIR)

Advantages

- Reduces total size of required control memory.
 - The reduced size of control memory reduces the total chip area.
- Greater design flexibility
 - Because of two level memory organization more design flexibility exists between instructions and hardware.

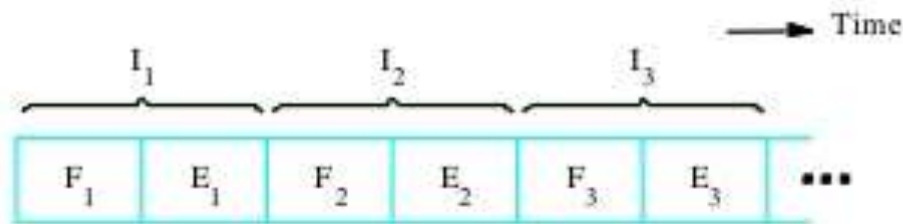
Disadvantages

- The main disadvantage of the two-level memory approach is the loss of speed due to the extra memory access required for nano control memory.

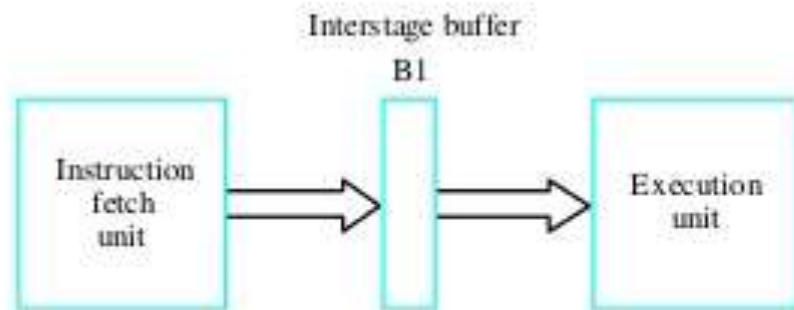
Use the Idea of Pipelining in a Computer



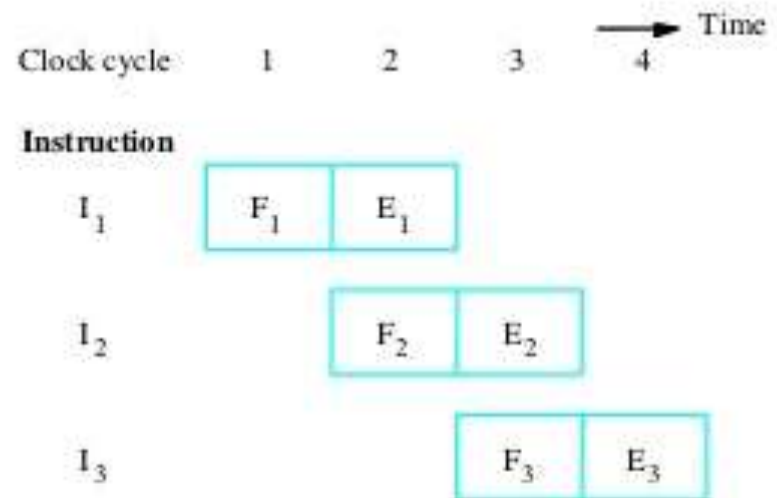
Fetch + Execution



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

Figure 8.1. Basic idea of instruction pipelining.



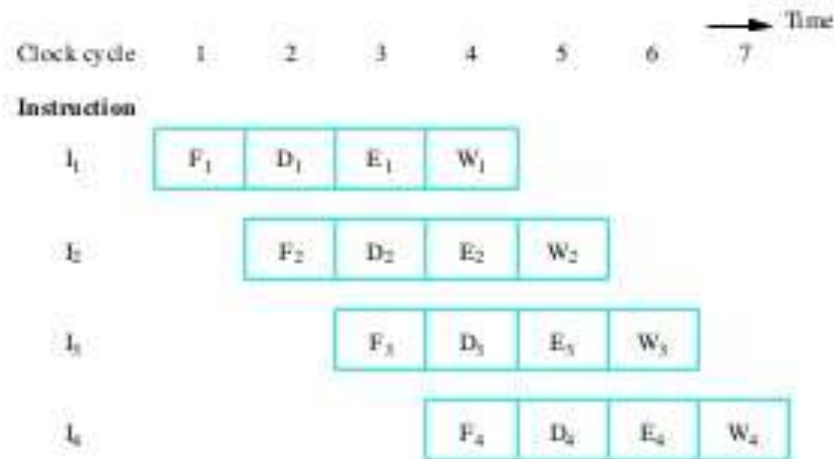
Contd.,

- Computer that has two separate hardware units, one for fetching and another for executing them.
- the instruction fetched by the fetch unit is deposited in an intermediate buffer **B1**.
- This buffer needed to enable the execution unit while fetch unit fetching the next instruction.

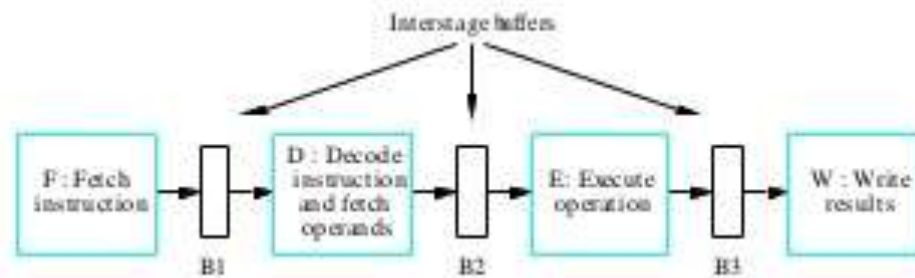
Use the Idea of Pipelining in a Computer



Fetch + Decode
+ Execution + Write



(a) Instruction execution divided into four steps



(b) Hardware organization

Textbook page: 457

Figure 8.2. A 4-stage pipeline.



- Fetch(F)- read the instruction from the memory
- Decode(D)- Decode the instruction and fetch the source operand
- Execute(E)- perform the operation specified by the instruction
- Write(W)- store the result in the destination location



Role of Cache Memory

- Each pipeline stage is expected to complete in **one clock cycle**.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless. [**ten times** greater than the time needed to perform pipeline stage]
- Fortunately, we have cache.



Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the **number of pipeline stages**.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not **true**.
- Floating point may involve many clock cycle

Pipeline Performance

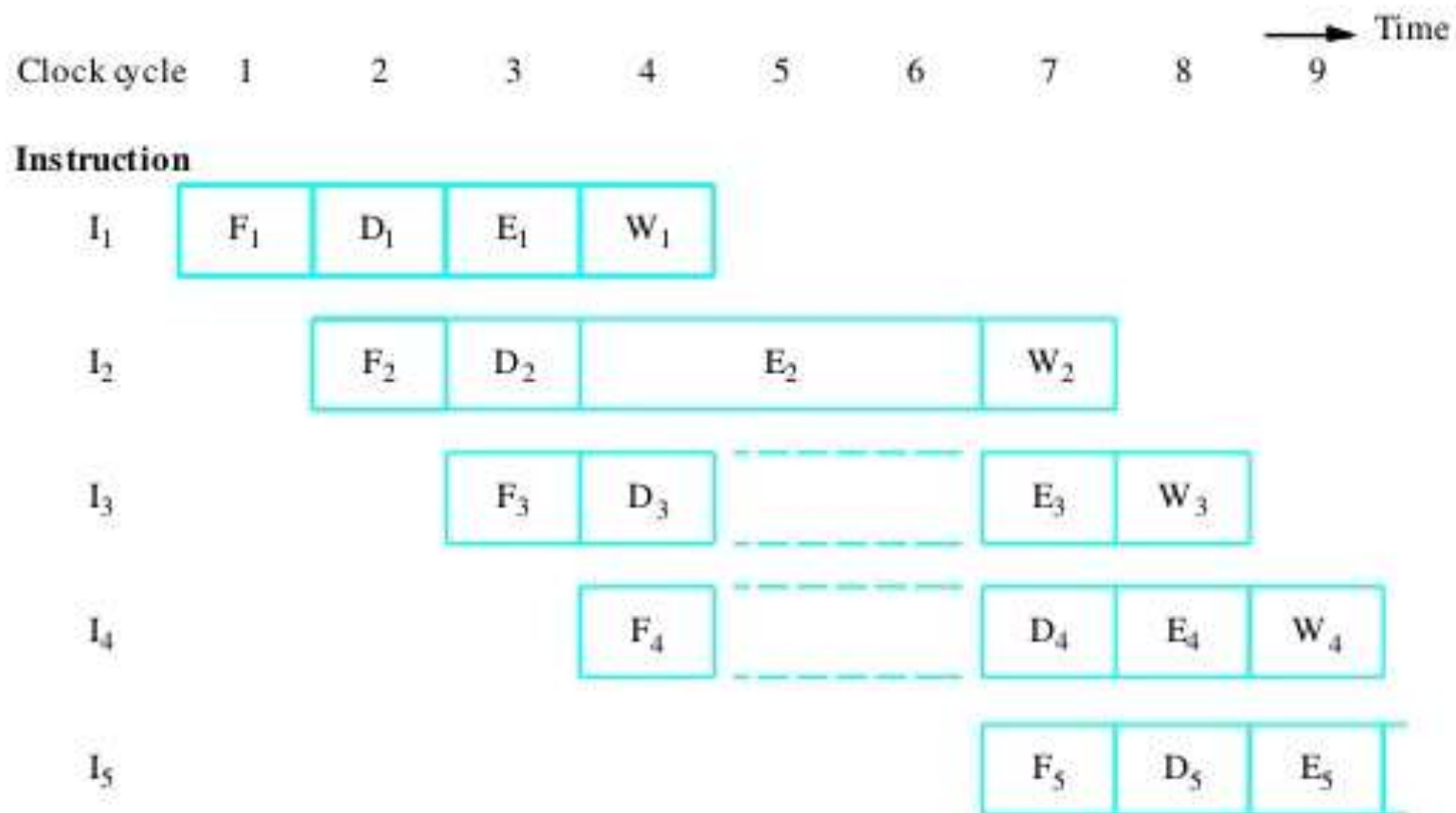


Figure 8.3. Effect of an execution operation taking more than one clock cycle.



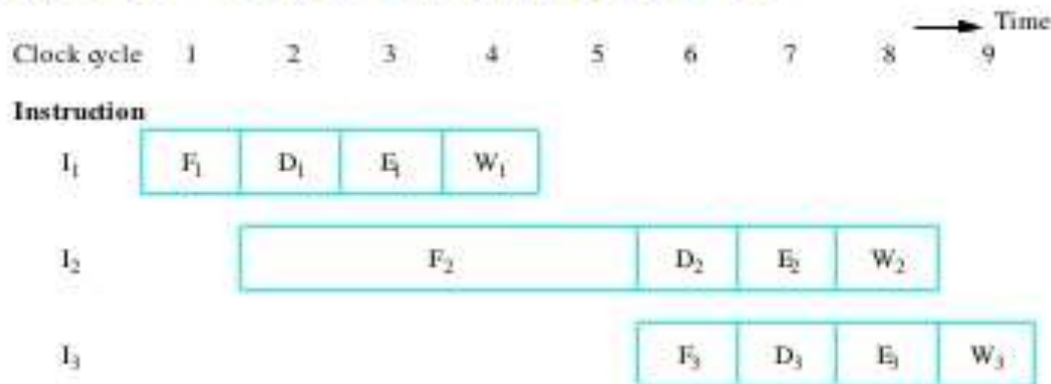
Pipeline Performance

- The previous pipeline is said to have been stalled for **two clock** cycles.
- Any condition that causes a pipeline to stall is called a **hazard**.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.[cache miss]
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.



Pipeline Performance

Instruction hazard (Cache miss)



(a) Instruction execution steps in successive clock cycles

Decode unit is idle in cycles 3 through 5, Execute unit idle in cycle 4 through 6 and write unit is idle in cycle 5 through 7 such idle period is called stalls.



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

Idle periods – stalls (bubbles)

Figure 8.4. Pipeline stall caused by a cache miss in F2.

Pipeline Performance

The memory address, $X+(R1)$ is computed in step E2 in cycle 4, then memory access takes place in cycle 5. the operand read from memory is written into register R2 in cycle 6 [Execution takes 2 cycles] it stalls pipeline to stall for one cycle. Bcoz both instruction I2 and I3 require access of register file in cycle 6.



Structural hazard

Load $X(R1), R2$

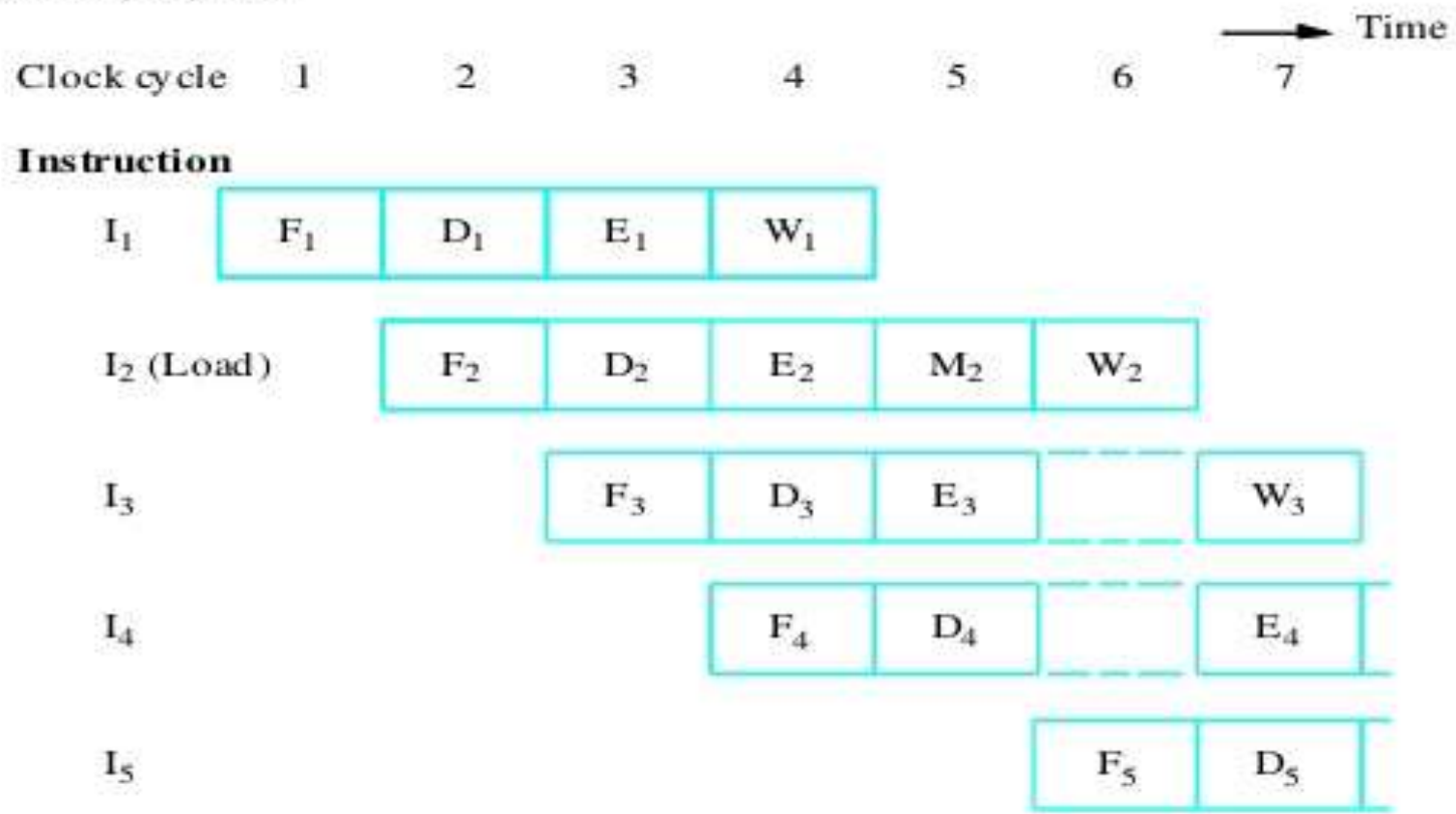


Figure 8.5. Effect of a Load instruction on pipeline timing.



Pipeline Performance

- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

Data Hazards





Data Hazards

- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.
- Hazard occurs
$$A \leftarrow 3 + A$$
$$B \leftarrow 4 \times A$$
- No hazard
$$A \leftarrow 5 \times C$$
$$B \leftarrow 20 + C$$
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:
$$\text{Mul } R2, R3, R4$$
$$\text{Add } R5, R4, R6$$

Data Hazards

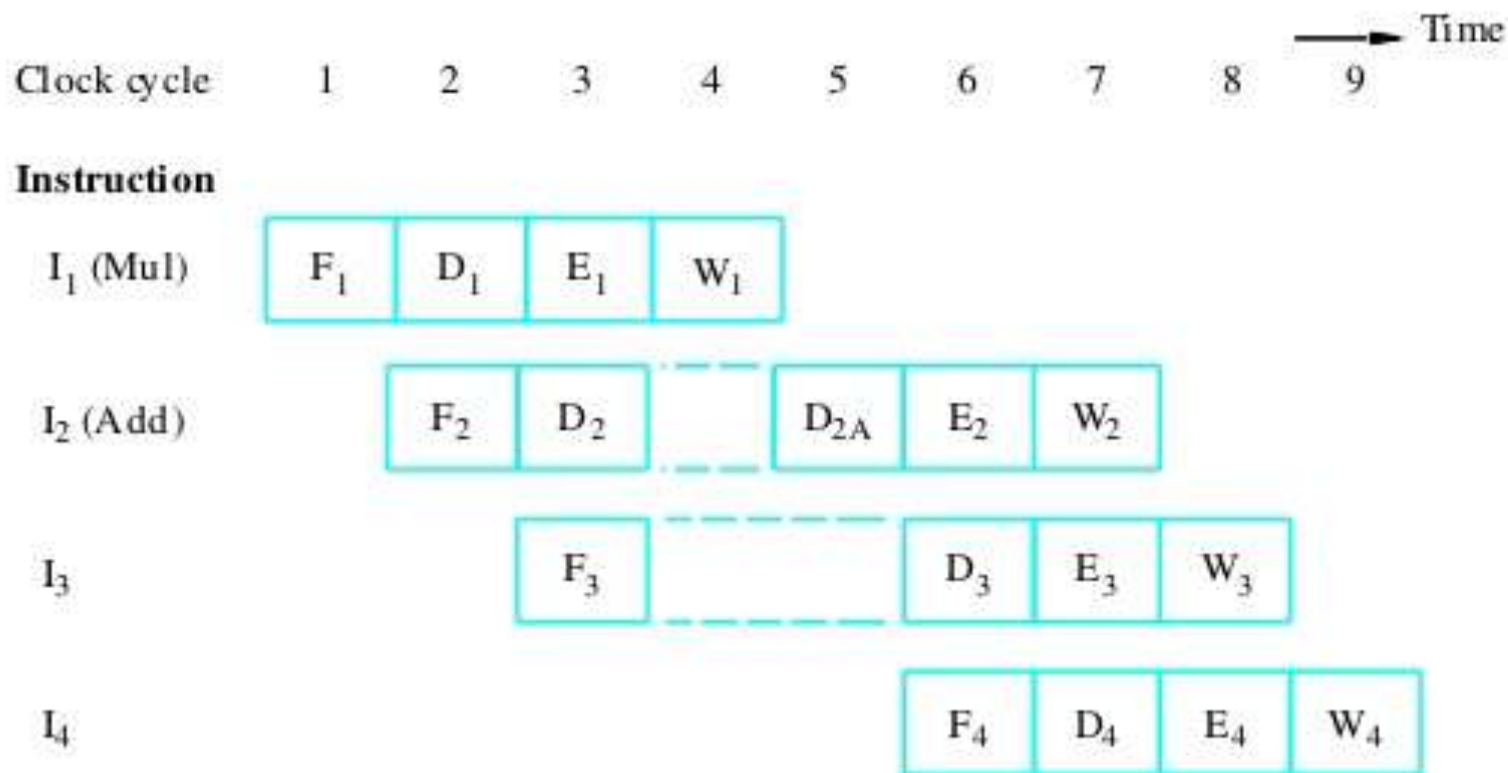
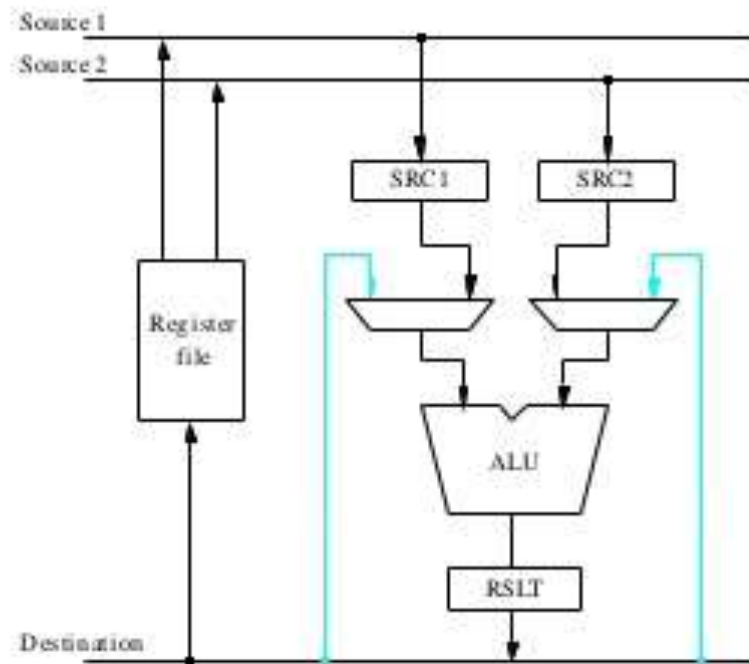


Figure 8.6. Pipeline stalled by data dependency between D_2 and W_1 .

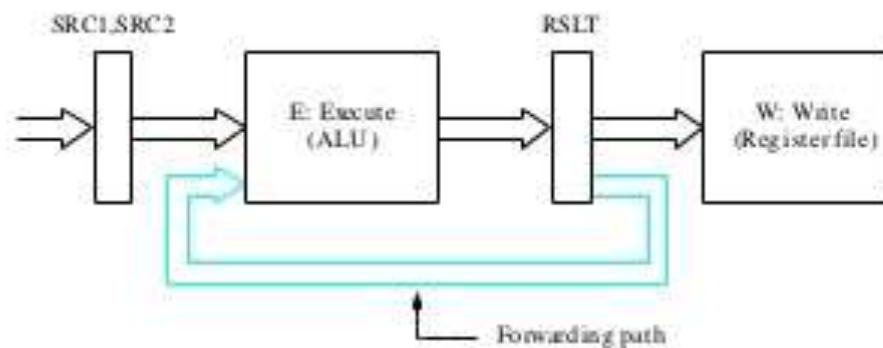


Operand Forwarding

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.
- A special arrangement needs to be made to “forward” the output of ALU to the input of ALU.



(a) Datapath



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

Figure 8.7. Operand forwarding in a pipelined processor.

Handling Data Hazards in Software



- Let the compiler detect and handle the hazard:

I1: Mul R2, R3, R4

NOP

NOP

I2: Add R5, R4, R6

- The compiler can reorder the instructions to perform some useful work during the NOP slots.



Side Effects

- The previous example is explicit and easily detected.
- Sometimes an instruction changes the contents of a register other than the one named as the destination.
- When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. (Example?)
- Example: conditional code flags:
 - Add R1, R3
 - AddWithCarry R2, R4
- Instructions designed for execution on pipelined hardware should have few side effects.

Data Hazard Classification

Given two instructions I , J , with I occurring before J in an instruction stream:

- RAW (read after write): *A true data dependence*
 J tried to read a source before I writes to it, so J incorrectly gets the old value.
- WAW (write after write): *A name dependence*
 J tries to write an operand before it is written by I
The writes end up being performed in the wrong order.
- WAR (write after read): *A name dependence*
 J tries to write to a destination before it is read by I , so I incorrectly gets the new value.
- RAR (read after read): Not a hazard.

Instruction Hazards





Overview

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



Unconditional Branches

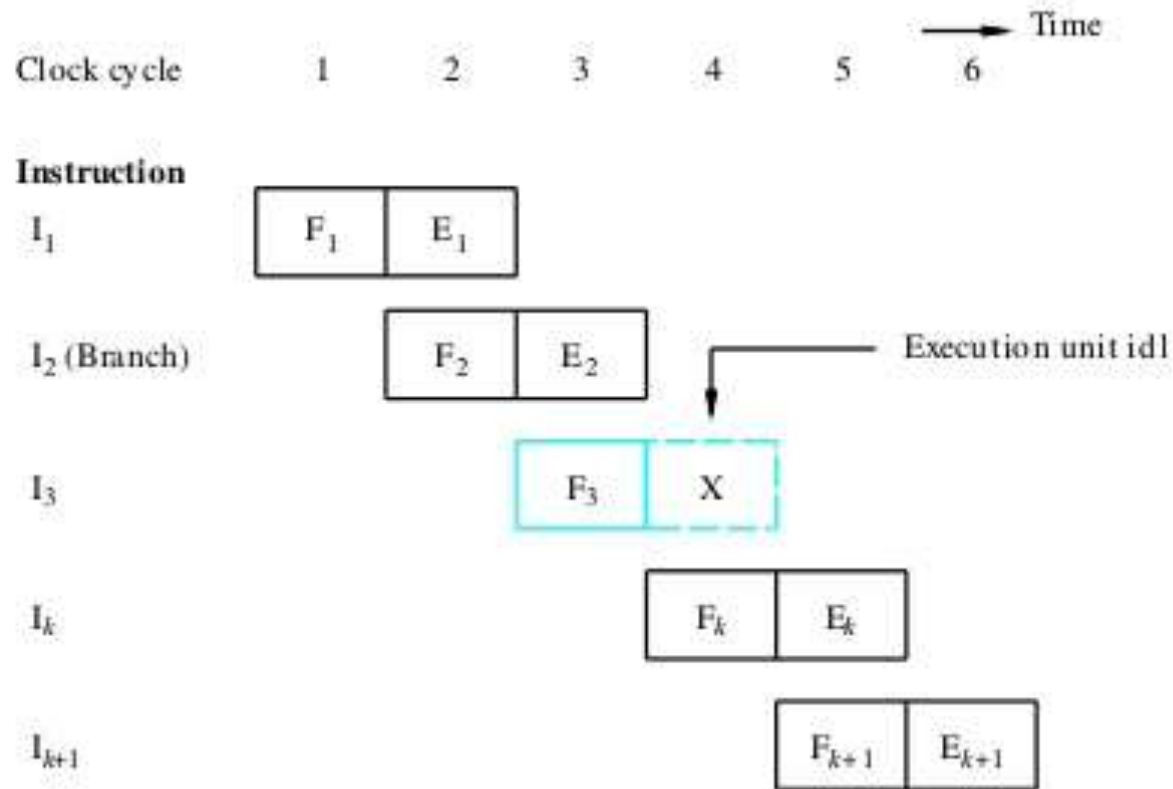


Figure 8.8. An idle cycle caused by a branch instruction.

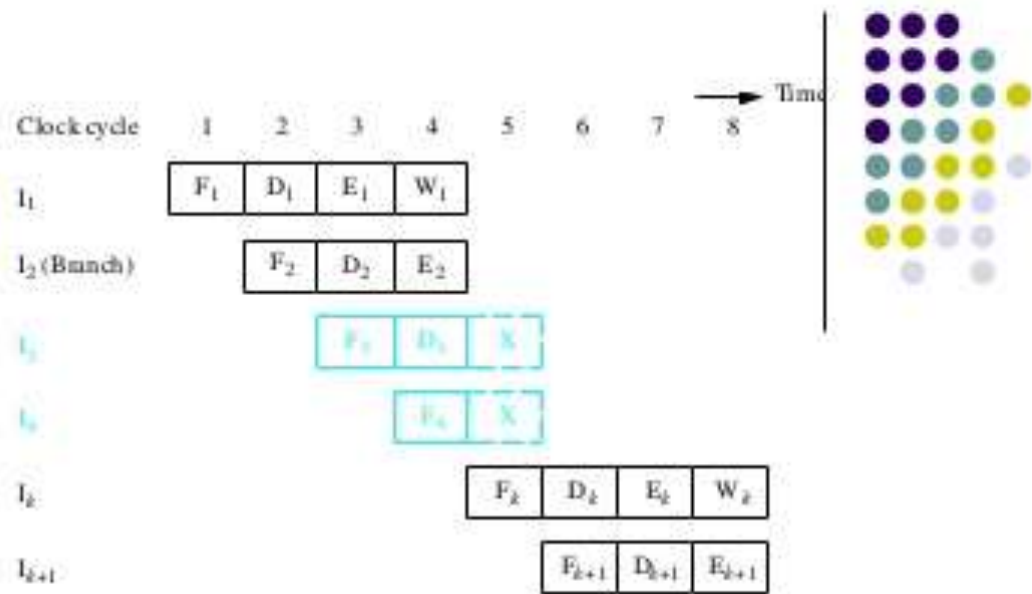
Unconditional Branches



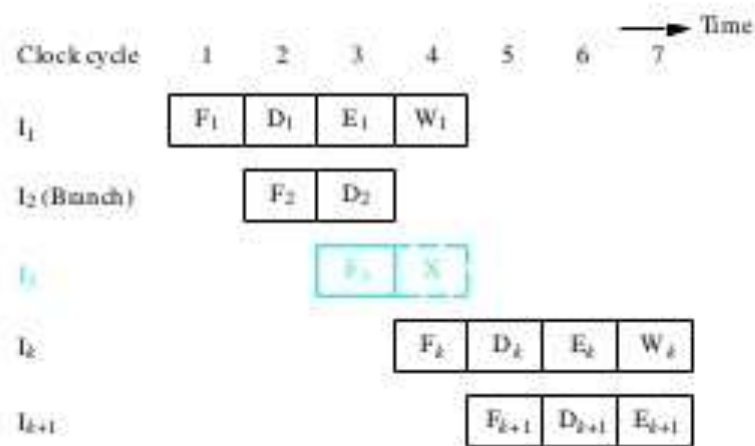
- The time lost as a result of a branch instruction is referred to as the **branch penalty**.
- The previous example instruction I3 is wrongly fetched and branch target address k will discard the i3.
- Reducing the branch penalty requires the branch address to be **computed earlier** in the pipeline.
- Typically the Fetch unit has dedicated h/w which will identify the branch target address as quick as possible after an instruction is fetched.

Branch Timing

- Branch penalty
- Reducing the penalty



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Figure 8.9. Branch timing.

Instruction Queue and Prefetching



- Either cache (or) branch instruction stalls the pipeline.
- Many processor employs dedicated fetch unit which will fetch the instruction and put them into a **queue**.
- It can store several instruction at a time.
- A separate unit called **dispatch unit**, takes instructions from the front of the queue and send them to the execution unit.

Instruction Queue and Prefetching

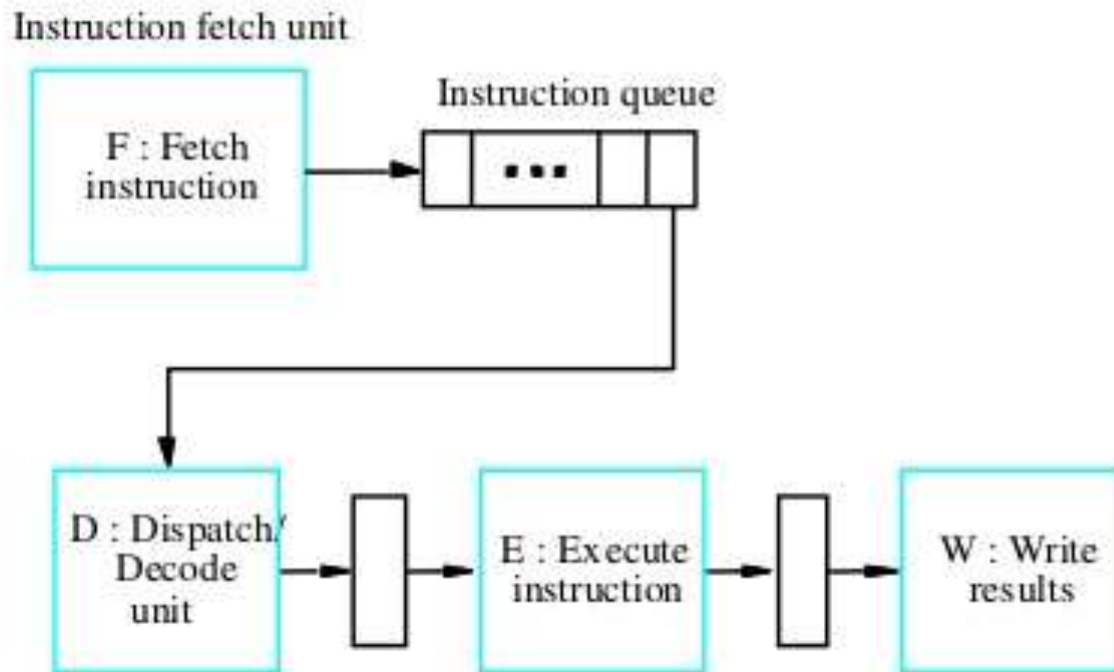


Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2b.



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

Figure 8.12. Reordering of instructions for a delayed branch.

Delayed Branch

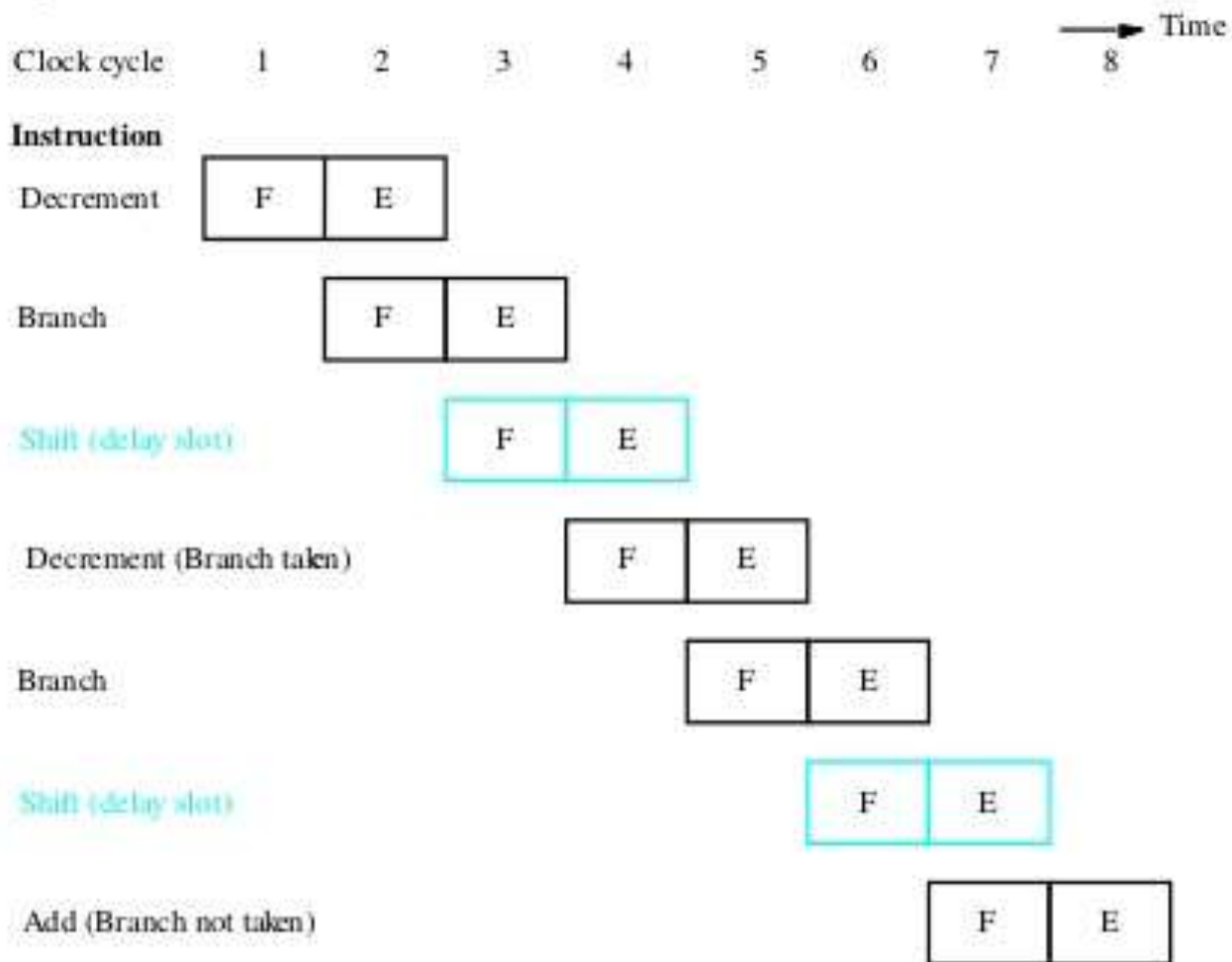


Figure 8.13. Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12.



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

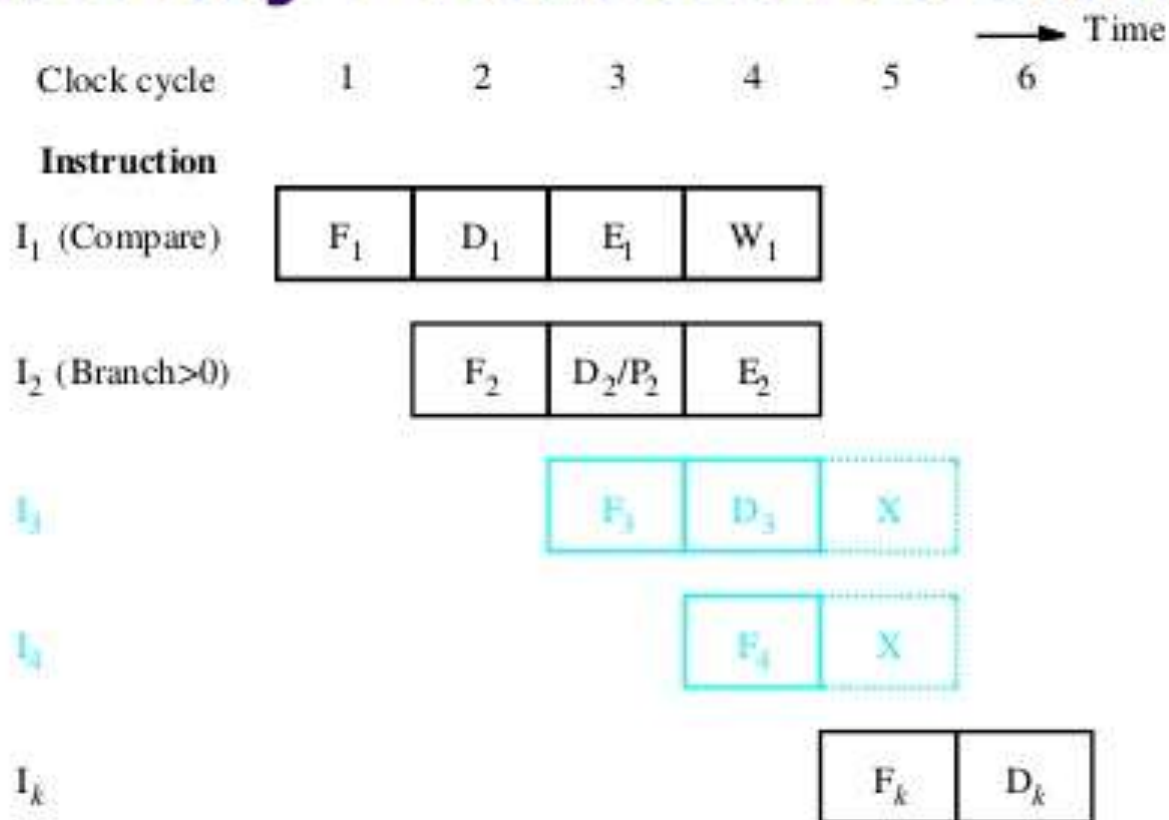


Figure 8.14. Timing when a branch decision has been incorrectly predicted as not taken.



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.

Influence on Instruction Sets





Overview

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



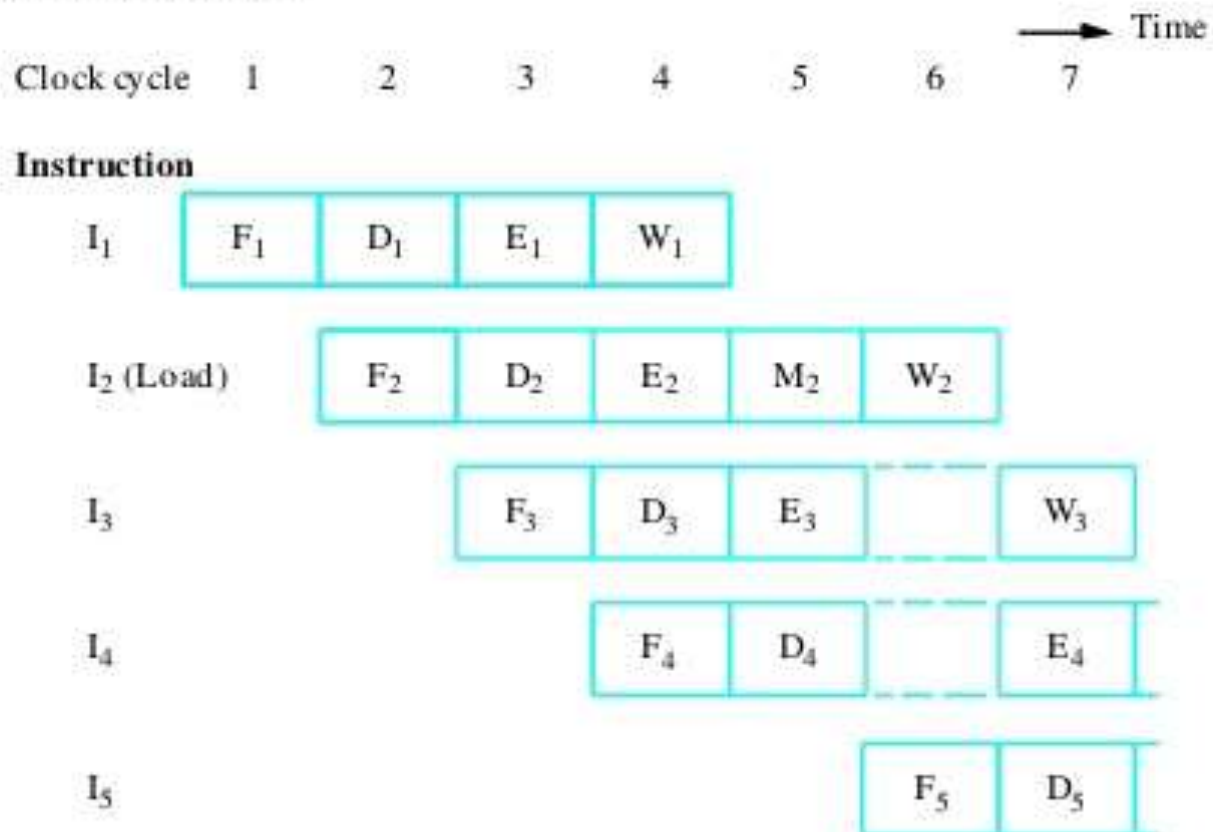
Addressing Modes

- 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



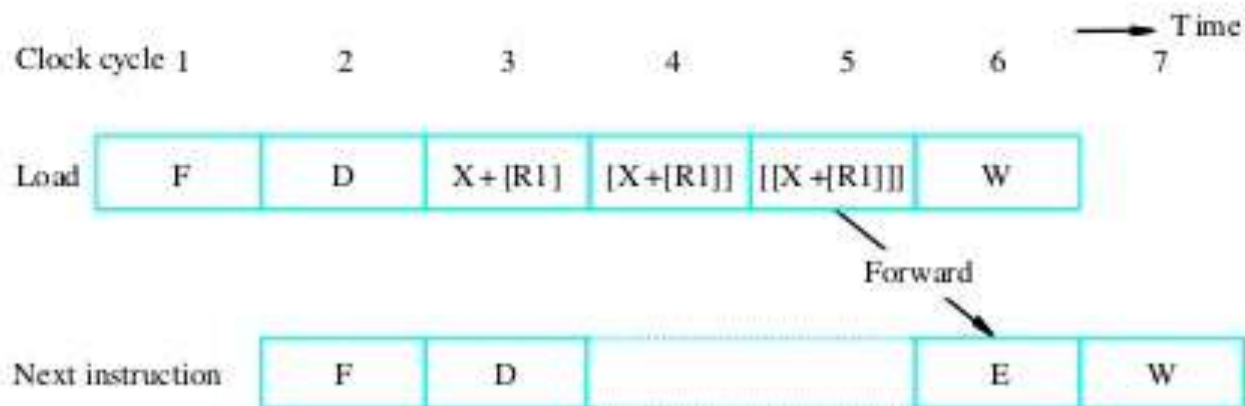
Load (R1), R2

Figure 8.5. Effect of a Load instruction on pipeline timing.



Complex Addressing Mode

Load (X(R1)), R2



(a) Complex addressing mode

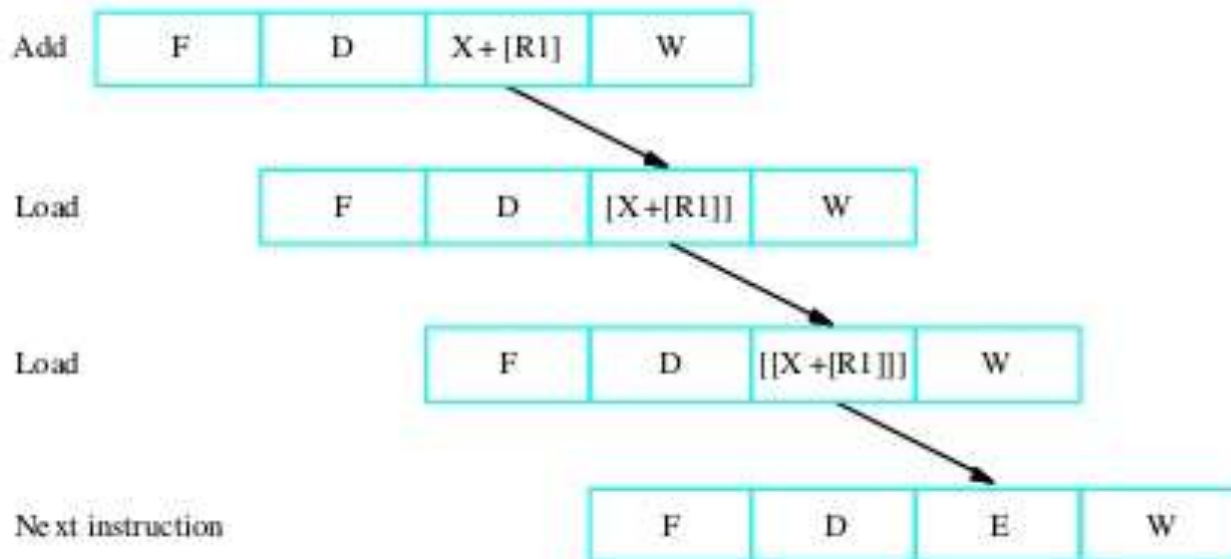


Simple Addressing Mode

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

Figure 8.17. 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.

Datapath and Control Considerations



Original Design

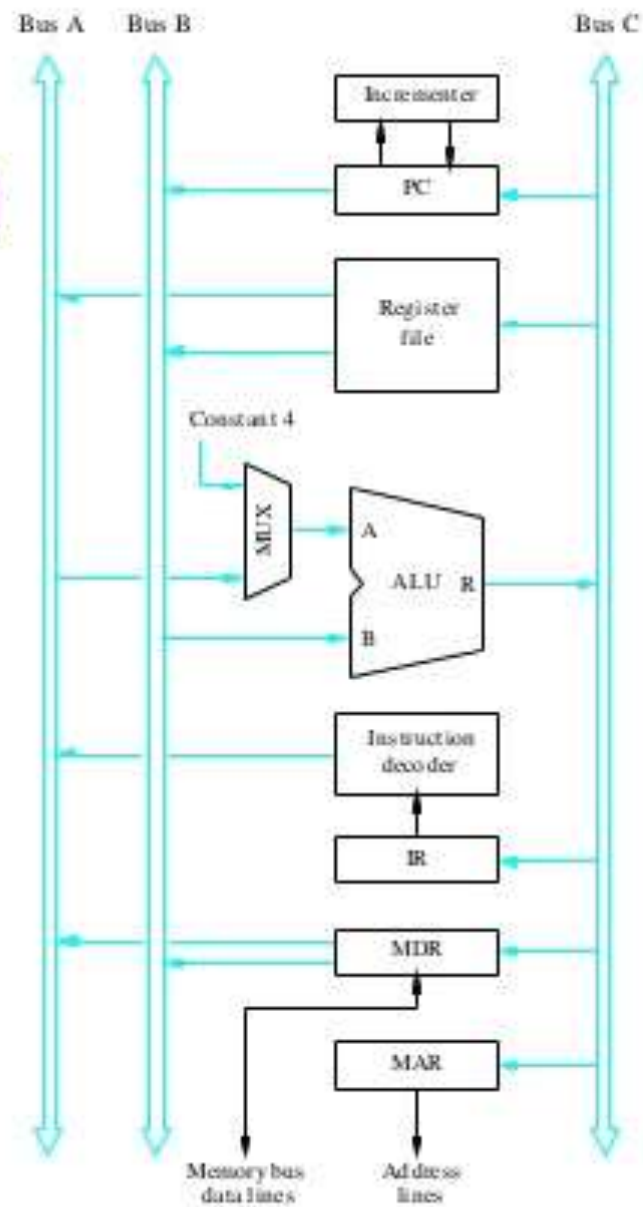
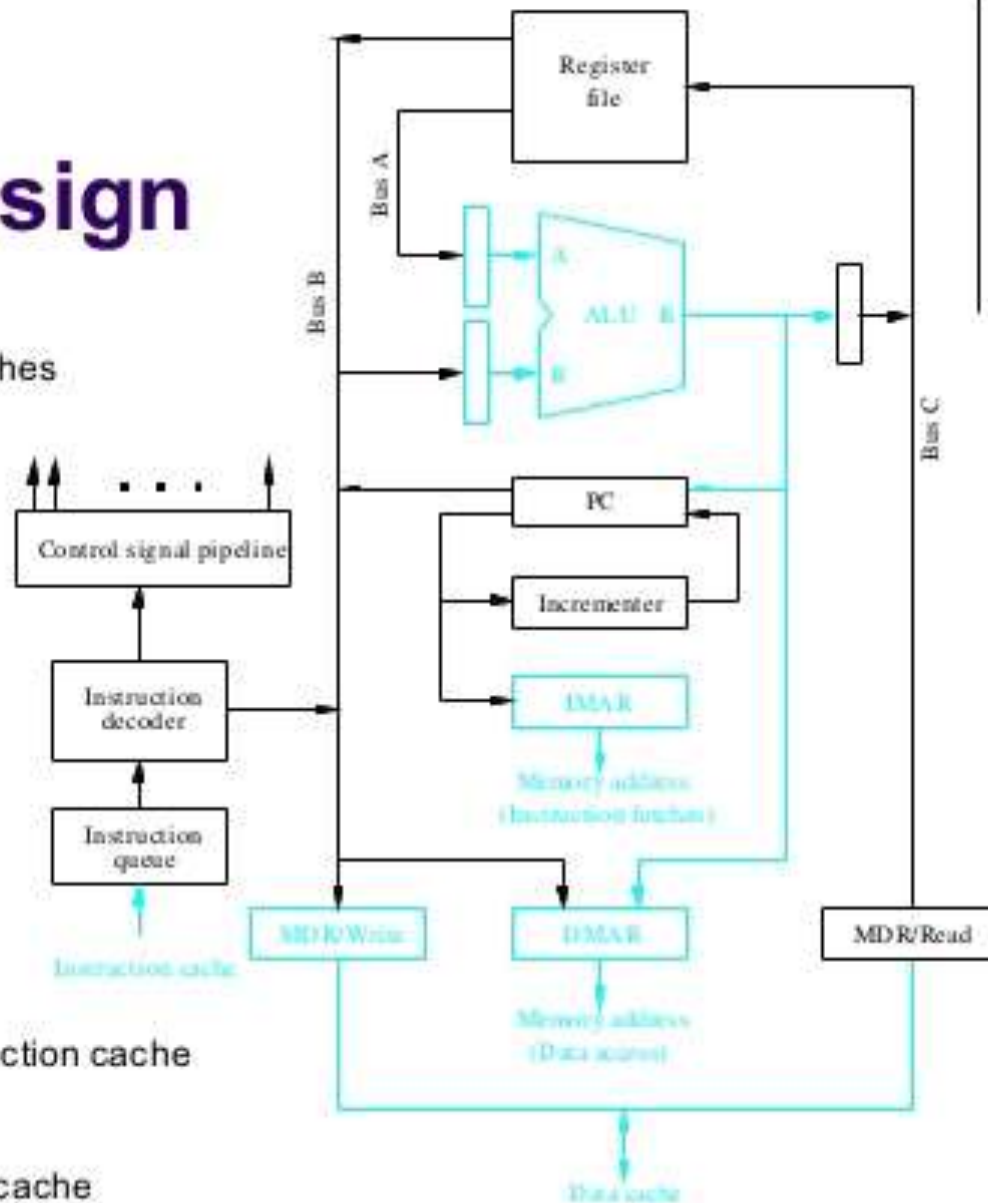


Figure 7.8. Three-bus organization of the datapath.

Pipelined Design

- Separate instruction and data caches
- PC is connected to IMAR
- DMAR
- Separate MDR
- Buffers for ALU
- Instruction queue
- Instruction decoder output



- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two regs
- Writing into one register in the reg file
- Performing an ALU operation

Figure 8.18. Datapath modified for pipelined execution, with interstage buffers at the input and output of the ALU.

Performance Considerations





Overview

- The execution time T of a program that has a dynamic instruction count N is given by:

$$T = \frac{N \times S}{R}$$

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate.

- Instruction throughput is defined as the number of instructions executed per second.

$$P_s = \frac{R}{S}$$