

CHAPTER 2: CPU

1

Computer Architecture and Organization

LEARNING OBJECTIVES

- Present an overview of essential characteristics of machine instructions.
- Describe the types of operands used in typical machine instruction sets.
- Describe the various types of addressing modes common in instruction sets.

WHAT IS AN INSTRUCTION SET?

- The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions*.
- The collection of different instructions that the processor can execute is referred to as the processor's *instruction set*.
- Elements of a Machine Instruction
 - **Operation code**: Specifies the operation to be performed (e.g., ADD, I/O).
 - **Source operand reference**: operands that are inputs for the operation.
 - **Result operand reference**: The operation may produce a result.
 - **Next instruction reference**: address of the next instruction.
- Source and result operands can be in one of four areas: Main or virtual memory, Processor register, immediate and I/O Device

INSTRUCTION REPRESENTATION

- Within the computer, each instruction is represented by a **sequence of bits**.
- The instruction is divided into fields

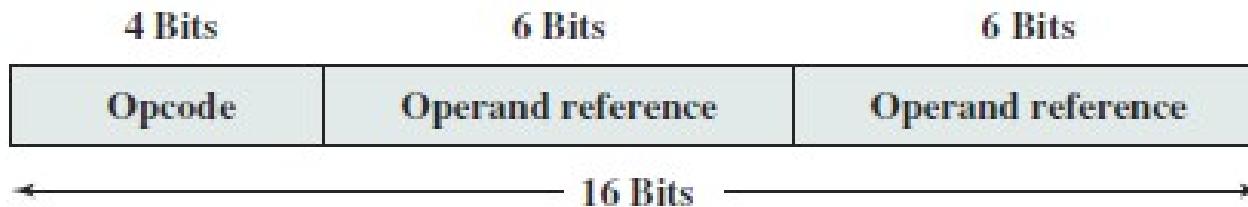


Figure 12.2 A Simple Instruction Format

- It's difficult to deal with binary representations of machine instructions.
 - Common practice: use a **symbolic representation** of machine instructions. E.g. ADD, SUB, MUL...etc.

INSTRUCTION TYPES

- Any program written in a **high-level language** must be translated into **machine language** to be executed.
- Thus, the set of machine instructions **must be sufficient to express any of the instructions** from a high-level language.
- With this in mind we can categorize instruction types as follows:
 - **Data processing:** Arithmetic and logic instructions.
 - **Data storage:** Movement of data into or out of register and or memory locations.
 - **Data movement:** I/O instructions.
 - **Control:** Test and branch instructions.

NUMBER OF ADDRESSES

- What is the maximum number of addresses one might need in an instruction?
- In most architectures, many instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the PC).
- With 3 addresses, 2 source operand locations and a destination operand location.
 - ADD A, B, C $//A = B + C$
 - Not common:- require a relatively long instruction format to hold the three address references.

NUMBER OF ADDRESSES

- With two-address instructions, and for binary operations, one address must do **double duty** as both an operand and a result.
 - `SUB Y, B` // $Y = Y - B$
 - The two-address format reduces the space requirement but also alters the value of an operand,
 - **MOVE** instruction is used to move one of the values to a result or temporary location before performing the operation.
 - `MOVE A, Y, SUB A, B`, answer in A
 - Reduces length of instruction but **requires some extra work**
 - Temporary storage to hold some results
- Simpler yet is the one-address instruction. For this to work, a second address must be implicit (**AC**).
 - $C = A - B \Rightarrow$ `LOAD A, SUB B, STOR C`
 - Common on early machines

NUMBER OF ADDRESSES

- Write a three, two and one address instructions that could be used to compute

$$Y = (A - B) / [C + (D * E)]$$

- Note that there are four instructions and that the original expression had five operands.
- Try not to alter the value of any of the operand locations.
- Try to compute the expression with smallest no of instructions possible.

<u>Instruction</u>		<u>Comment</u>
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

<u>Instruction</u>		<u>Comment</u>
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

<u>Instruction</u>		<u>Comment</u>
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Figure 12.3 Programs to Execute $Y = \frac{A - B}{C + (D \times E)}$

CON...

- It is, in fact, possible to make do with **zero addresses** for some instructions.
- Zero-address instructions are applicable to a special memory organization called a **stack**.
- The stack is in a known location and, often, at least the top two elements are in processor registers.
 - zero-address instructions would reference the top two stack elements.

Table 12.1 Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	$A \leftarrow B \text{ OP } C$
2	OP A, B	$A \leftarrow A \text{ OP } B$
1	OP A	$AC \leftarrow AC \text{ OP } A$
0	OP	$T \leftarrow (T - 1) \text{ OP } T$

AC = accumulator

T = top of stack

(T - 1) = second element of stack

A, B, C = memory or register locations

SO HOW MANY ADDRESSES?

- The number of addresses per instruction is a basic design decision.
- More addresses
 - More complex instructions
 - Fewer instructions per program
- Fewer addresses
 - Less complex instructions
 - More instructions per program
 - Require less complex processor
 - It also results in instructions of shorter length.
 - Results in longer execution times

INSTRUCTION SET DESIGN

The design of an instruction set is very complex because it affects so many aspects of the computer system.

The most important of these **fundamental design issues** include the following:

- **Operation list:** How many and which operations to provide, and how complex operations should be.
- **Data types:** The various types of data upon which operations are performed.
- **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on.
- **Registers:** Number of processor registers that can be referenced by instructions, and their use.
- **Addressing:** The mode or modes by which the address of an operand is specified.

TYPES OF OPERAND

- Addresses
- Numbers
 - Integer/floating point
- Characters
 - ASCII etc.
- Logical Data
 - Bits or flags

TYPES OF OPERATION

- Data Transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System Control
- Transfer of Control

DATA TRANSFER

- Specify
 - Source
 - Destination
 - Amount of data
 - Mode of addressing
- May be different instructions for different movements
 - e.g. IBM 390.....(L, LH, LD,ST,STH.....etc)
- Or one instruction and different addresses
 - e.g. VAX.....(MOV)

ARITHMETIC

- Add, Subtract, Multiply, Divide
- Signed Integer
- Floating point
- May include
 - Increment ($a++$)
 - Decrement ($a--$)
 - Negate ($-a$)
 - Absolute

LOGICAL

- Bitwise operations based upon Boolean operations
- AND, OR, NOT, EQUAL

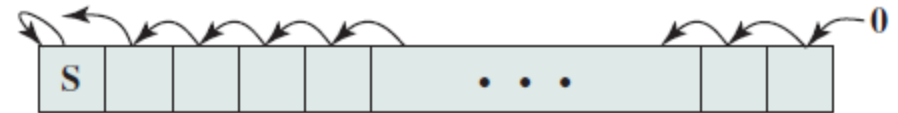
Table 12.6 Basic Logical Operations

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P = Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

SHIFT AND ROTATE OPERATIONS



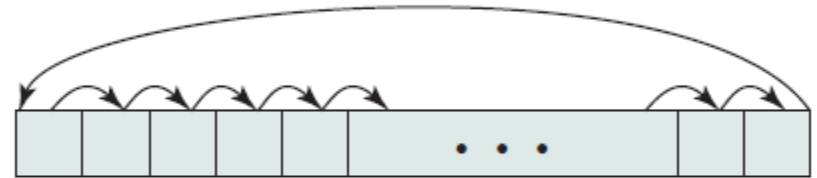
(a) Logical right shift



(d) Arithmetic left shift



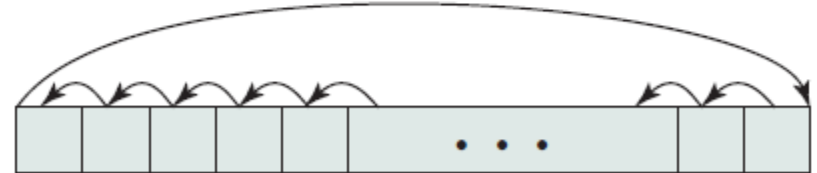
(b) Logical left shift



(e) Right rotate



(c) Arithmetic right shift



(f) Left rotate

Examples of Shift and Rotate Operations
10100110

CONVERSION

- Change the format or operate on the format of data.
- E.g. Binary to Decimal

INPUT/OUTPUT

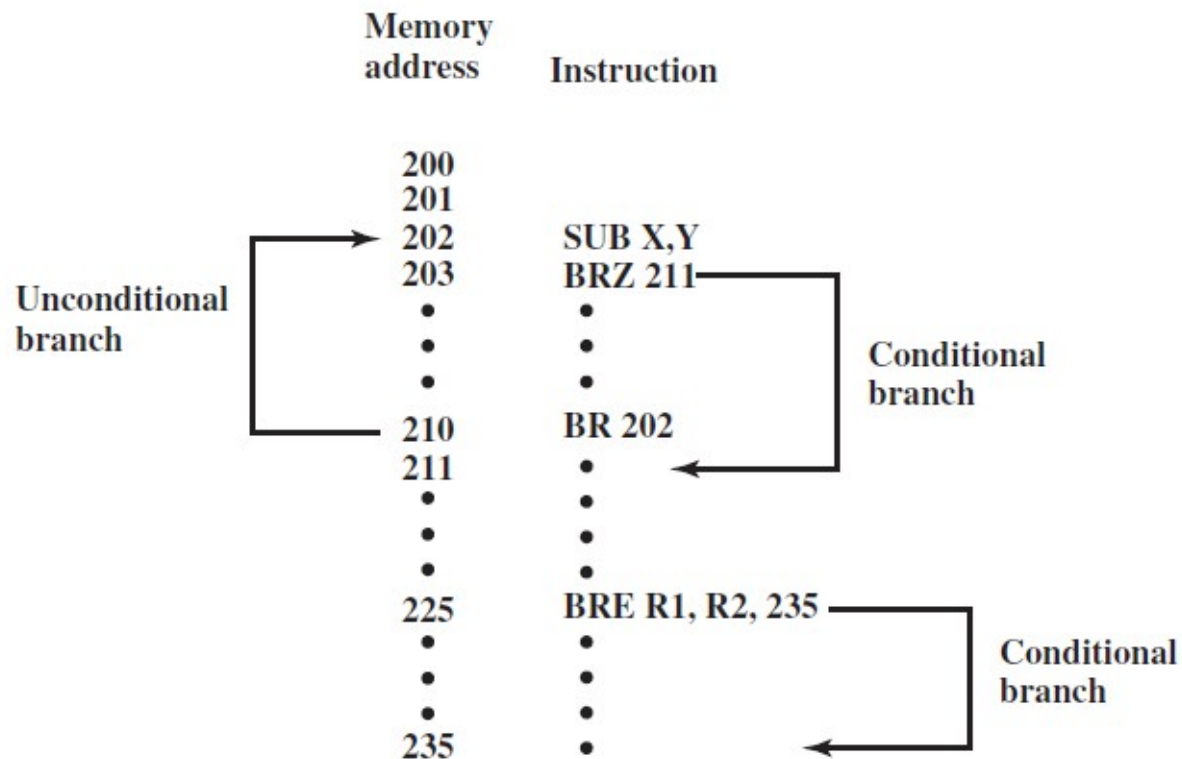
- For i/o operation

SYSTEM CONTROL

- Executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory.
- Typically, these instructions are reserved for the use of the operating system.

TRANSFER OF CONTROL

- Branch AKA Jump
 - Can be conditional or unconditional
 - e.g. BRZ,BRP,BRN,BRO



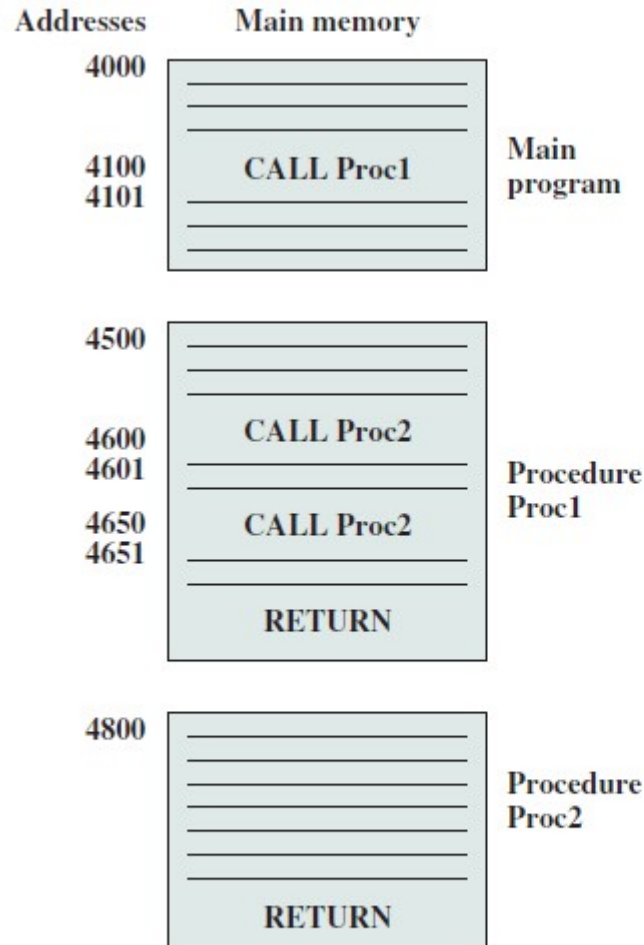
TRANSFER OF CONTROL

- Skip
 - Implies that one instruction be skipped
 - e.g. increment and skip if zero

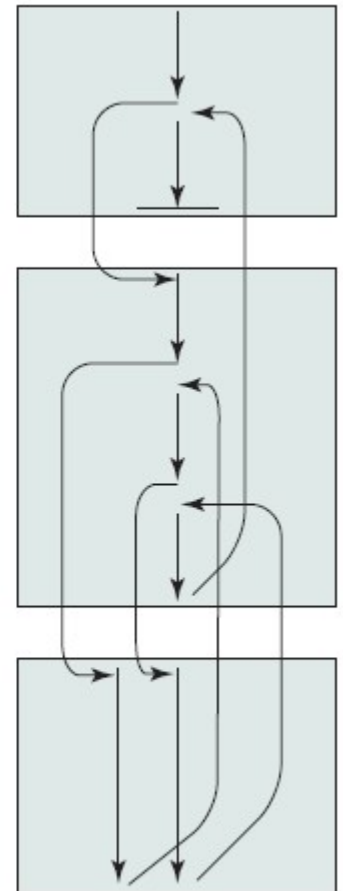
```
301  
:  
:  
309  ISZ  R1  
310  BR   301  
311
```

TRANSFER OF CONTROL

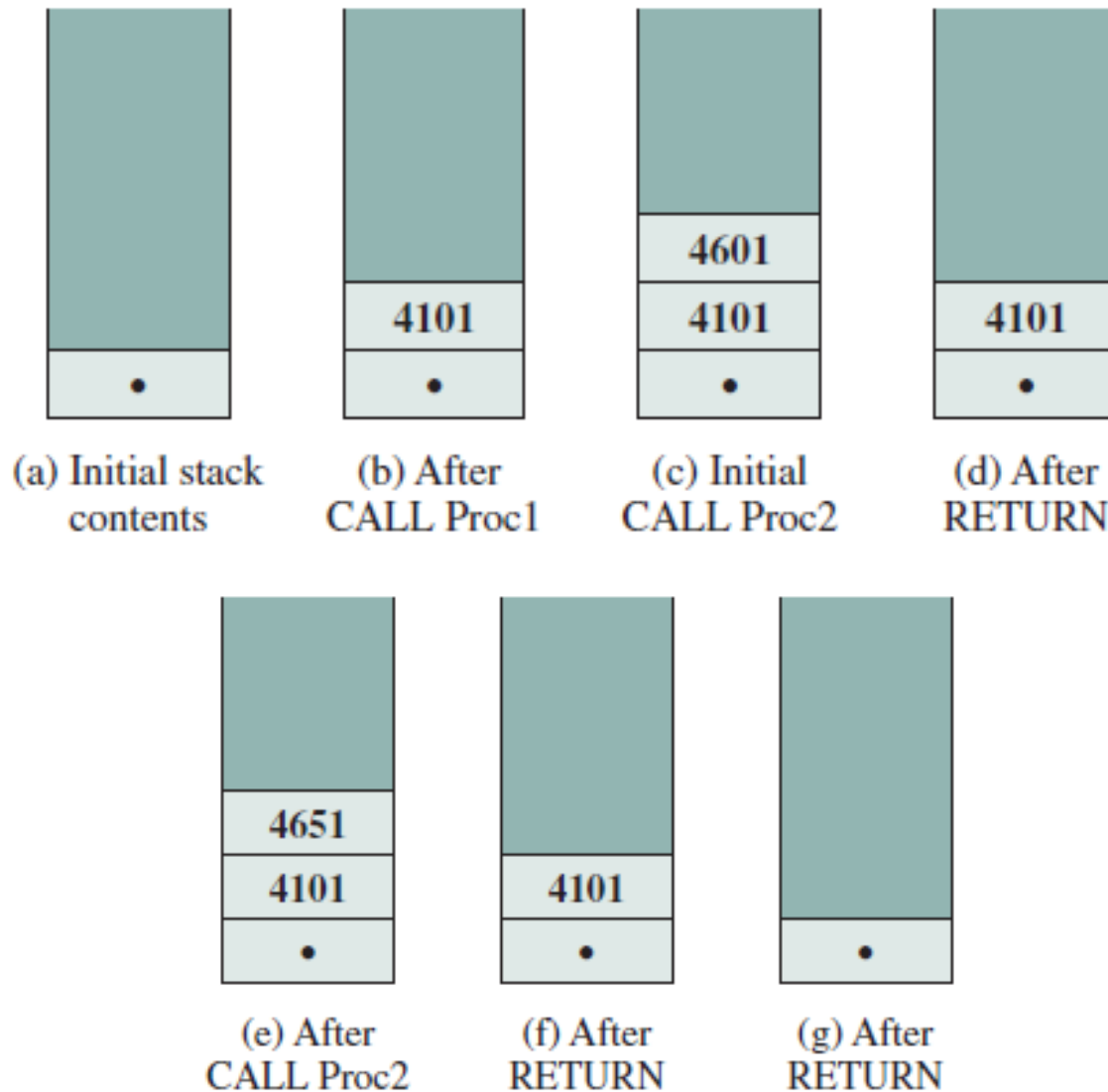
- Subroutine call
 - Procedure call



(a) Calls and returns



(b) Execution sequence



Use of Stack to Implement Nested Subroutines of Figure 12.8

ADDRESSING MODES

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement (Indexed)
- Stack

how the processor can determine which address mode is being used in a particular instruction?

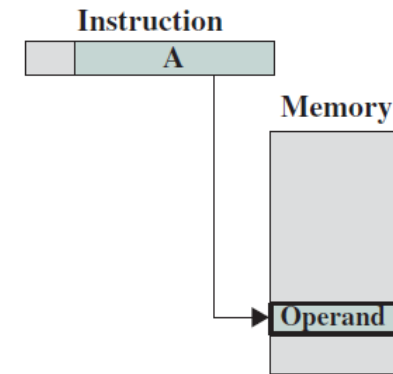
IMMEDIATE ADDRESSING

Instruction

Opcode	Operand
--------	---------

- The simplest form of addressing
- Operand is part of instruction
- Used to **define & use constants** or **set initial values** of variables
- e.g. ADD 0101
 - Add 5 to contents of accumulator
 - 5 is operand
- Pros
 - Fast:- No memory reference to fetch data
- Cons
 - Limited range: - the size of the number is restricted to the size of the address field

DIRECT ADDRESSING

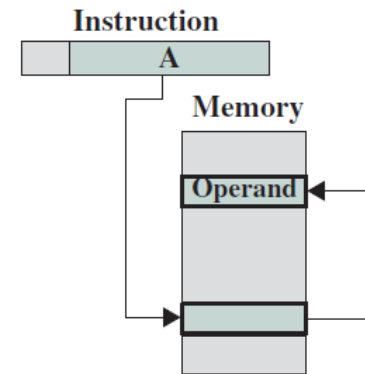


- The address field contains the effective address of the operand

$$EA = A$$

- e.g. `ADD A`
 - Add contents of memory address A to accumulator
 - Look in memory at address A for operand
- Was common in earlier generations of computers
- Single memory reference to access data
 - No additional calculations to work out effective address
- Limited address range
 - restricted to the size of the address field

INDIRECT ADDRESS

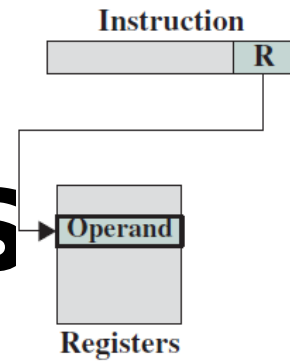


- The address field refer to the address of a word in memory, which in turn contains a full-length address of the operand.

$$EA = (A)$$

- Look in A, find address (A) and look there for operand
- e.g. ADD (A)
 - Add contents of cell pointed to by contents of A to accumulator
- Pros
 - Increased address space
- Cons
 - Instruction execution requires **two memory references** to fetch the operand:
 - one to get its address and a second to get its value.

REGISTER ADDRESS



- Address field refers to a register rather than a main memory address

$$EA = R$$

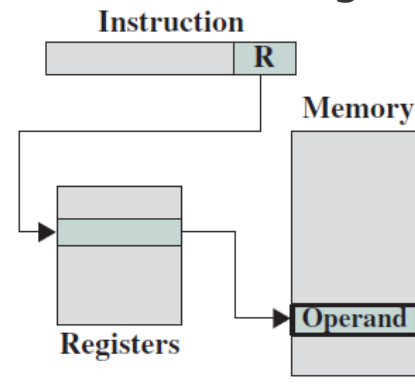
- Pros
 - Only a small address field is needed in the instruction
 - Very fast: - No time-consuming memory references are required.
- Cons
 - Address space is very limited
 - Limited number of registers

REGISTER INDIRECT ADDRESSING

- C.f. indirect addressing

$$EA = (R)$$

- Operand is in memory cell pointed to by contents of register R
- Pros
 - Large address space (2^n)
 - One fewer memory access than indirect addressing
- Cons
 - Extra memory access

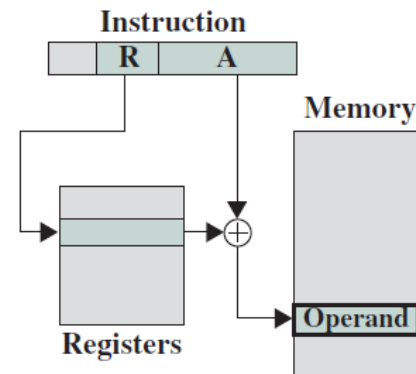


DISPLACEMENT ADDRESSING

- Powerful mode of addressing combines the capabilities of **direct addressing** and **register indirect addressing**.

$$EA = A + (R)$$

- Address field hold two address fields
 - A = base value
 - R = register that holds displacement
 - or vice versa



RELATIVE ADDRESSING

- A version of displacement addressing
- $R = \text{Program counter, PC}$
- $EA = A + (PC)$
- i.e. the next instruction address is added to the address field to produce the EA.
- locality of reference

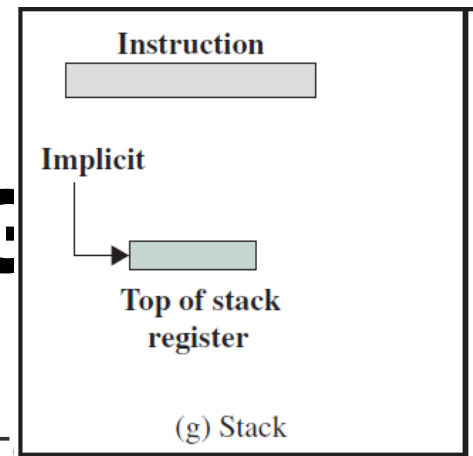
BASE-REGISTER ADDRESSING

- A version of displacement addressing
- $EA = A + (R)$
- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g. segment registers in 80x86

INDEXED ADDRESSING

- A version of displacement addressing
- $A = \text{base}$
- $R = \text{displacement}$
- $EA = A + (R)$
- Good for accessing arrays
 - $EA = A + (R)$
 - $(R)++$

STACK ADDRESSING



- Top of stack pointer is maintained in a register
 - In some processors, the top two elements stored in the processor's registers
- Operand is (implicitly) on top of stack
- e.g.
 - ADD Pop top two items from stack and add
- Pros
 - No memory reference
- Cons
 - Limited applicability

SUMMARY OF ADDRESSING

Table 13.1 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability



PART 2

Chapter 2: CPU

Computer Architecture and Organization

LEARNING OBJECTIVES

- Distinguish between **user-visible** and **control/status registers**, and discuss the purposes of registers in each category.
- Summarize the **instruction cycle**.
- Discuss the principle behind instruction pipelining and how it works in practice.
- Compare and contrast the various forms of pipeline hazards.

INSTRUCTION FORMAT

Reading Assignments

Page 469

PROCESSOR ORGANIZATION

What is the requirements placed on the processor, (the things that it must do) ?

- **Fetch instruction**
- **Interpret instruction**
- **Fetch data**
- **Process data**
- **Write data.**

To do these things, it should be clear that the processor **needs to store** some data temporarily.

In other words, the processor needs a small internal memory.

Registers

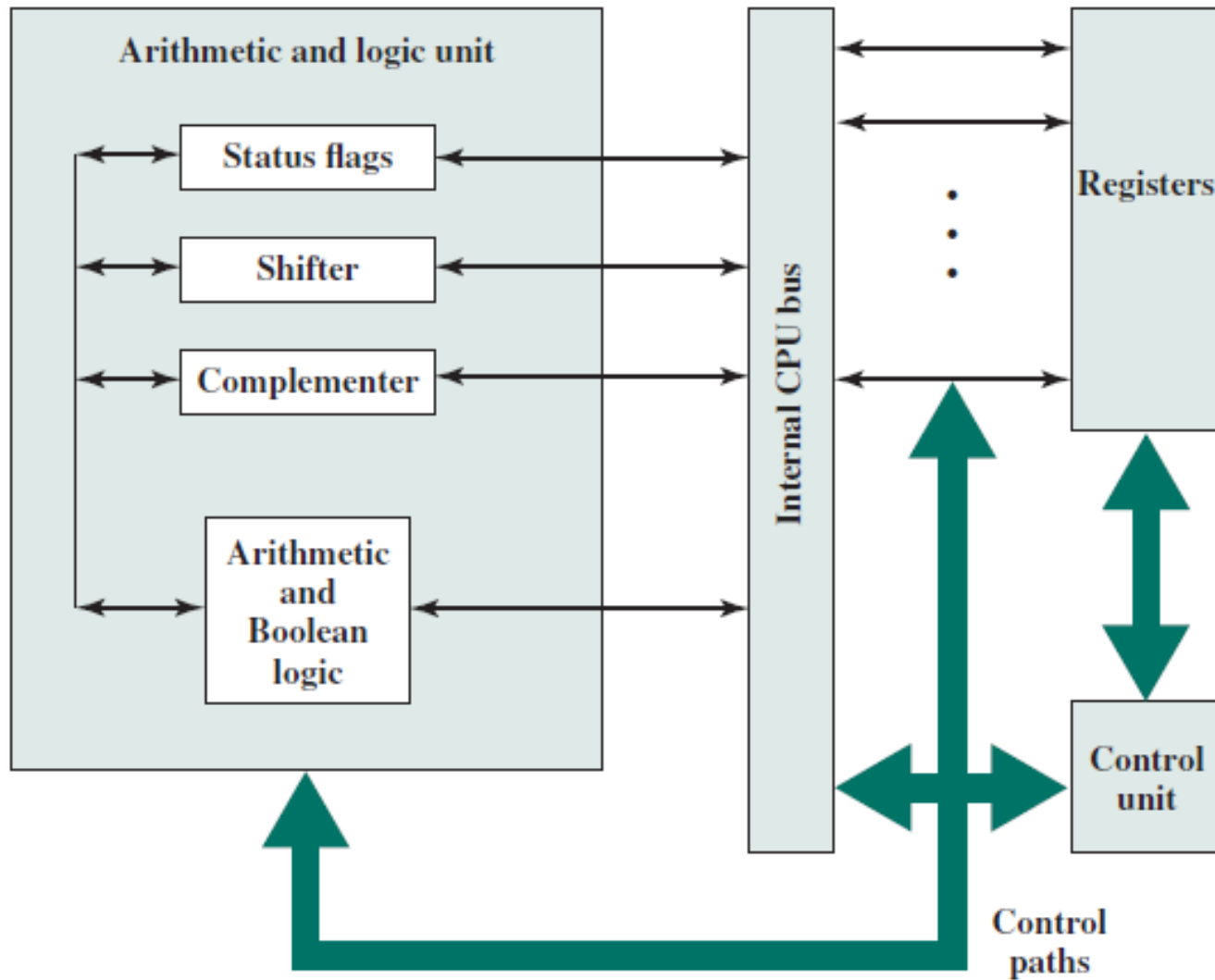


Figure 14.2 Internal Structure of the CPU

REGISTER ORGANIZATION

Registers: - are a very fast, small, and most expensive storage inside the CPU.

The registers in the processor perform **two roles**:

- **User-visible registers**: Enable the machine- or assembly language programmer to **minimize main memory references** by optimizing use of registers.
- **Control and status registers**:
 - Used by the control unit to **control the operation of the processor** and by privileged, operating system programs to **control the execution of programs**.

There is **no clean separation** of registers into these two categories.

USER-VISIBLE REGISTERS

Is one that may be **referenced** by means of the **machine language** that the processor executes. (4 categories)

- **General purpose**: can be assigned to a variety of functions by the programmer (with small restrictions)
- **Data**: used only to hold data
- **Address**: somewhat general purpose, or they may be devoted to a particular addressing mode (**Segment pointers, Index registers, Stack pointer**).
- **Condition codes**: are bits set by the processor hardware as the result of operations.
 - partially visible to the user
 - read by implicit reference, but the programmer cannot alter them.
 - Condition Codes pros and cons ? **Check your reference book.**

REGISTERS DESIGN ISSUES....

1. Use completely **general-purpose registers** or to **specialize their use?**

- Make them general purpose
 - Increase flexibility and programmer options
 - Increase instruction size & complexity
- Make them specialized
 - Smaller (faster) instructions format (**register is given implicitly**)
 - limits the programmer's flexibility

2. How Many Registers?

- Between 8 – 32 registers appears optimum
- Fewer = more memory references
- More does not reduce memory references and takes up processor real estate
- Some RISC systems use 100 or more registers.

REGISTERS DESIGN ISSUES....

3. How big? (register length)

- Large enough to hold full address (for addresses registers)
- Large enough to hold full word (for data registers)
- Some machines allow two contiguous registers to be used as one for holding double-length values
- C programming
 - double int a;
 - long int a;

CONTROL AND STATUS REGISTERS

- Employed to **control the operation of the processor.**
- Most of these, on most machines, **are not visible to the user.**
- Some of them **may be visible** to machine instructions executed in a **control or operating system mode.**

- **Program Counter (PC)**

- **Instruction Register (IR)**

what do these all do?

- **Memory Address Register (MAR)**

- **Memory Buffer Register (MBR)**

PROGRAM STATUS WORD (PSW),

PSW : a register or set of registers, which contains **condition codes plus other status information**.

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a **carry** (addition) into or **borrow** (subtraction) out of a high-order bit.
- **Equal:** Set if a **logical compare** result is equality.
- **Overflow:** Used to indicate **arithmetic overflow**.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode.

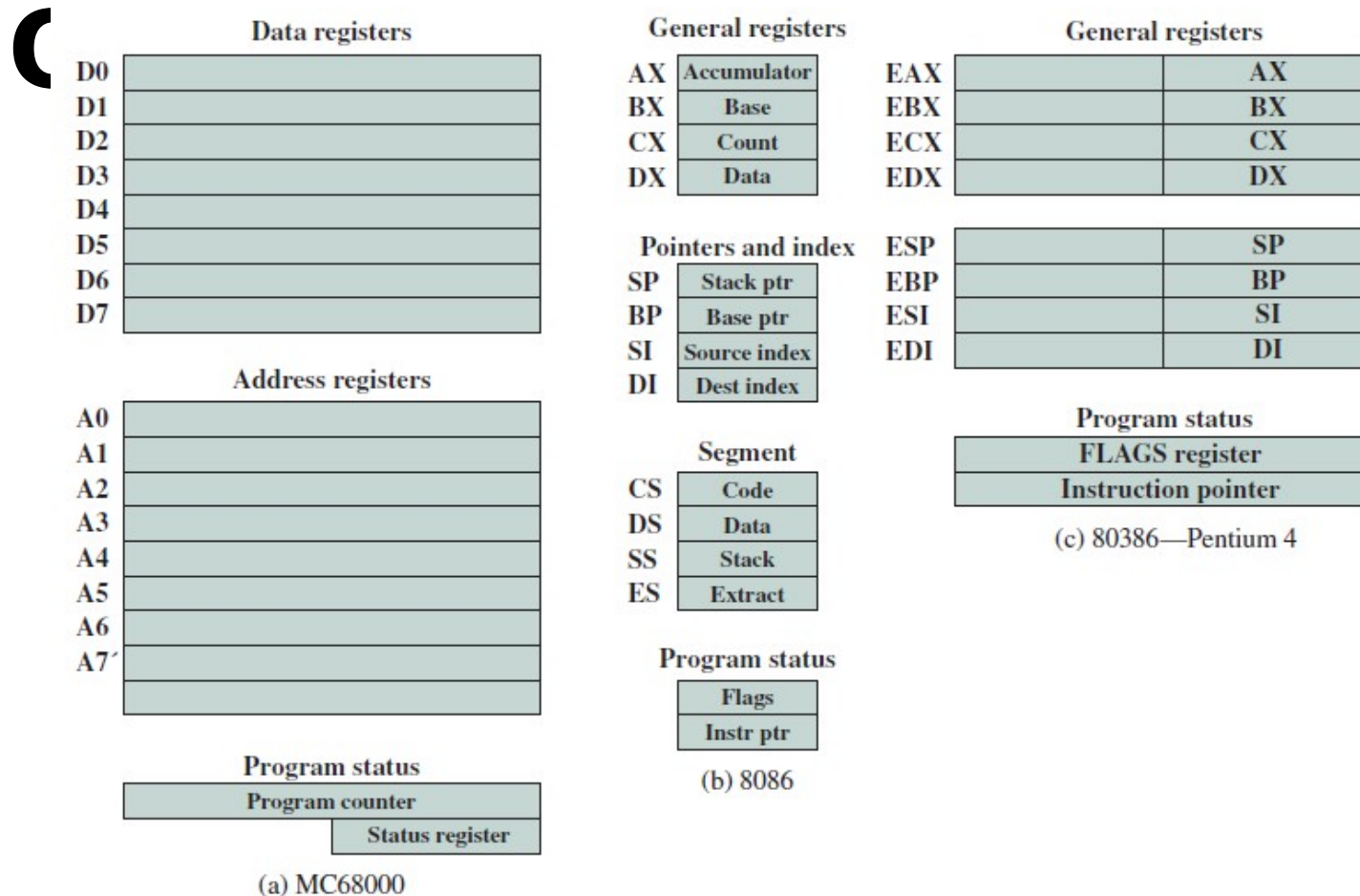
CONT...

- Other registers (related to status and control)
 - Process control blocks
 - Interrupt vector register
 - System stack pointer
 - Page table pointer
- The allocation of control information between registers and memory.
 - must decide how much control information should be in registers and how much in memory.

Control and status register organization design issues

- OS support
 - N.B. CPU design and operating system design are closely linked

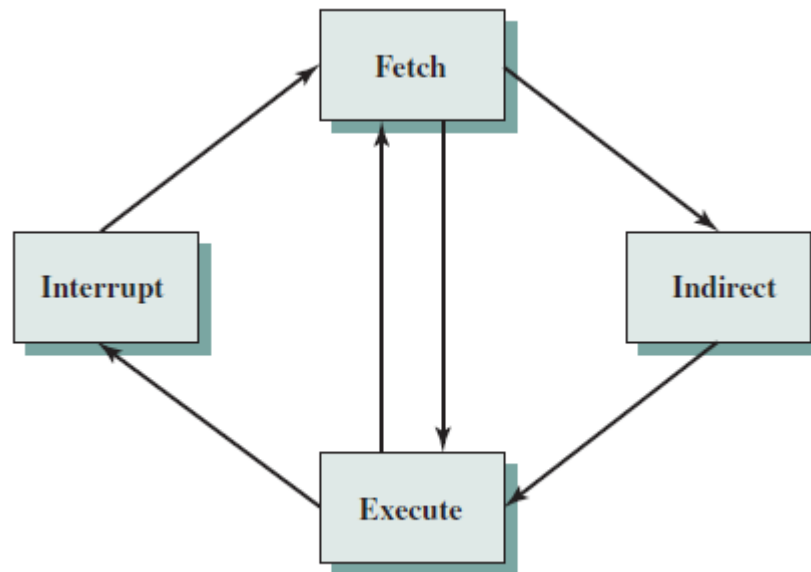
EXAMPLE MICROPROCESSOR REGISTER



INDIRECT CYCLE

- Processor's instruction cycle, **Fetch - Execute - Interrupt**
- If indirect addressing is used, then **additional memory accesses** are required.
- We can think of the more instruction sta

as one



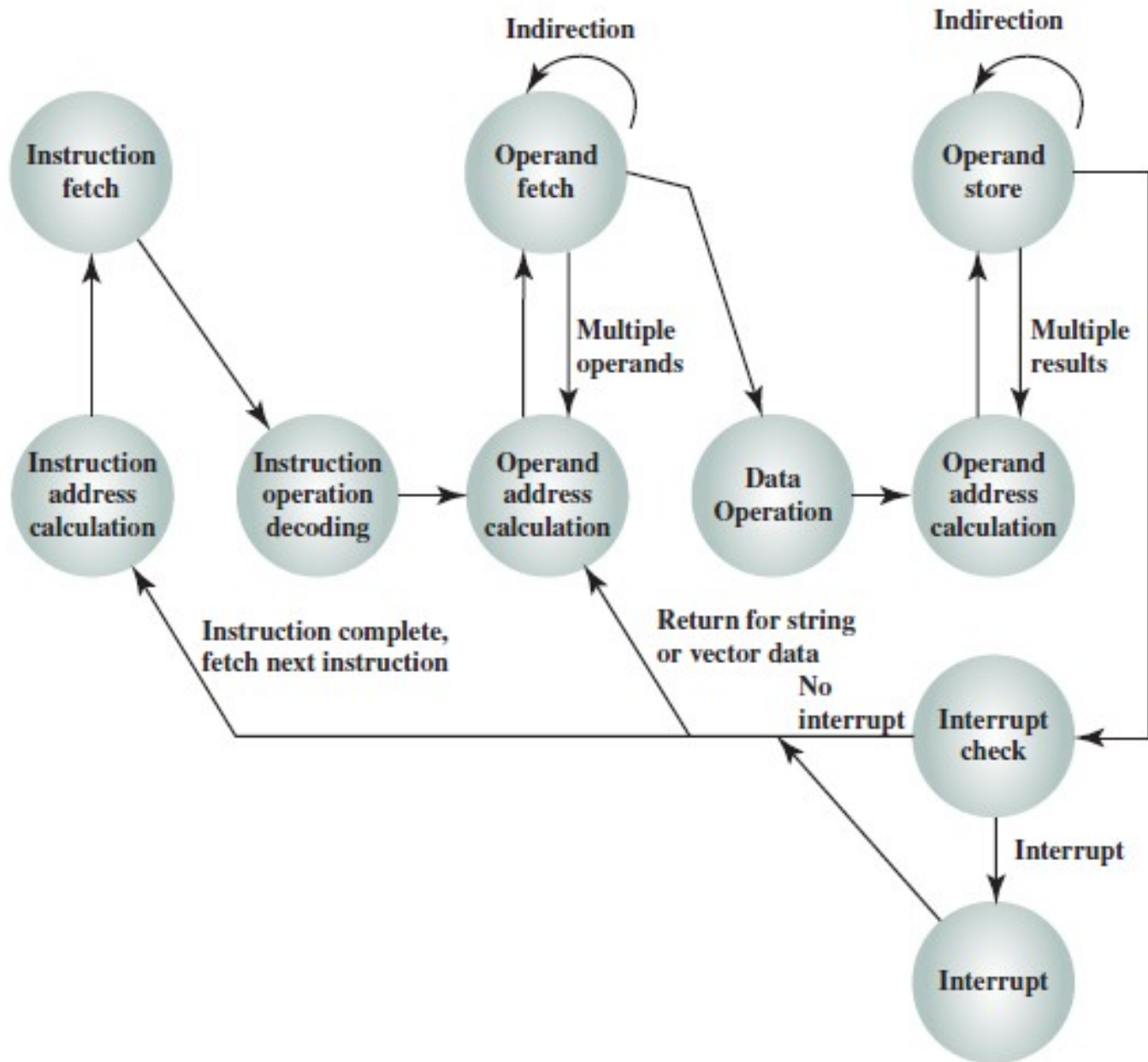
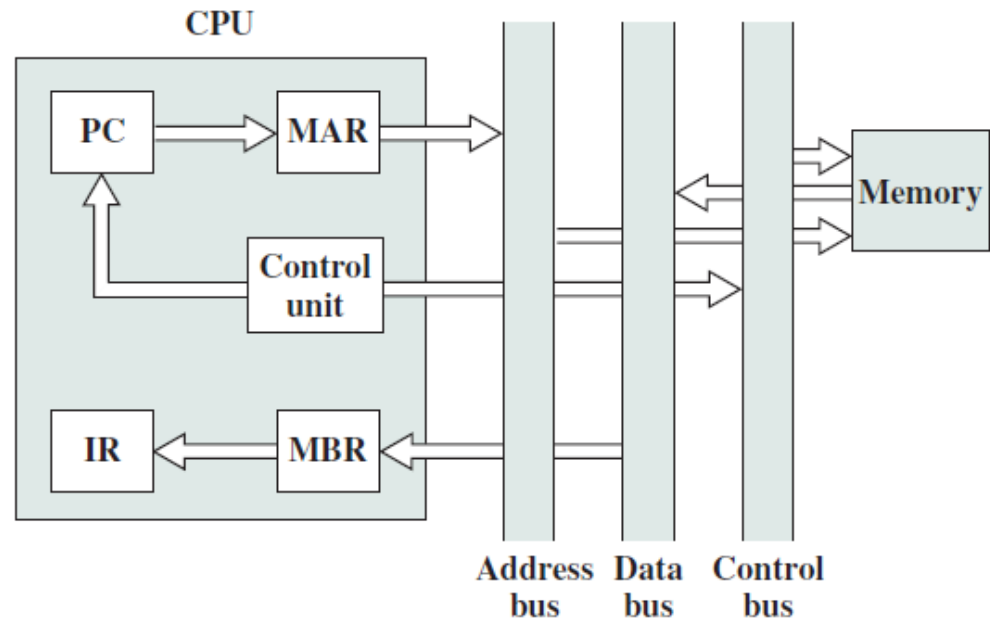


Figure 14.5 Instruction Cycle State Diagram

DATA FLOW

- The exact sequence of events during an instruction cycle depends on the design of the processor.
- But, what happens to MAR, MBR, PC, IR
- ??????



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Figure 14.6 Data Flow, Fetch Cycle

CONT...

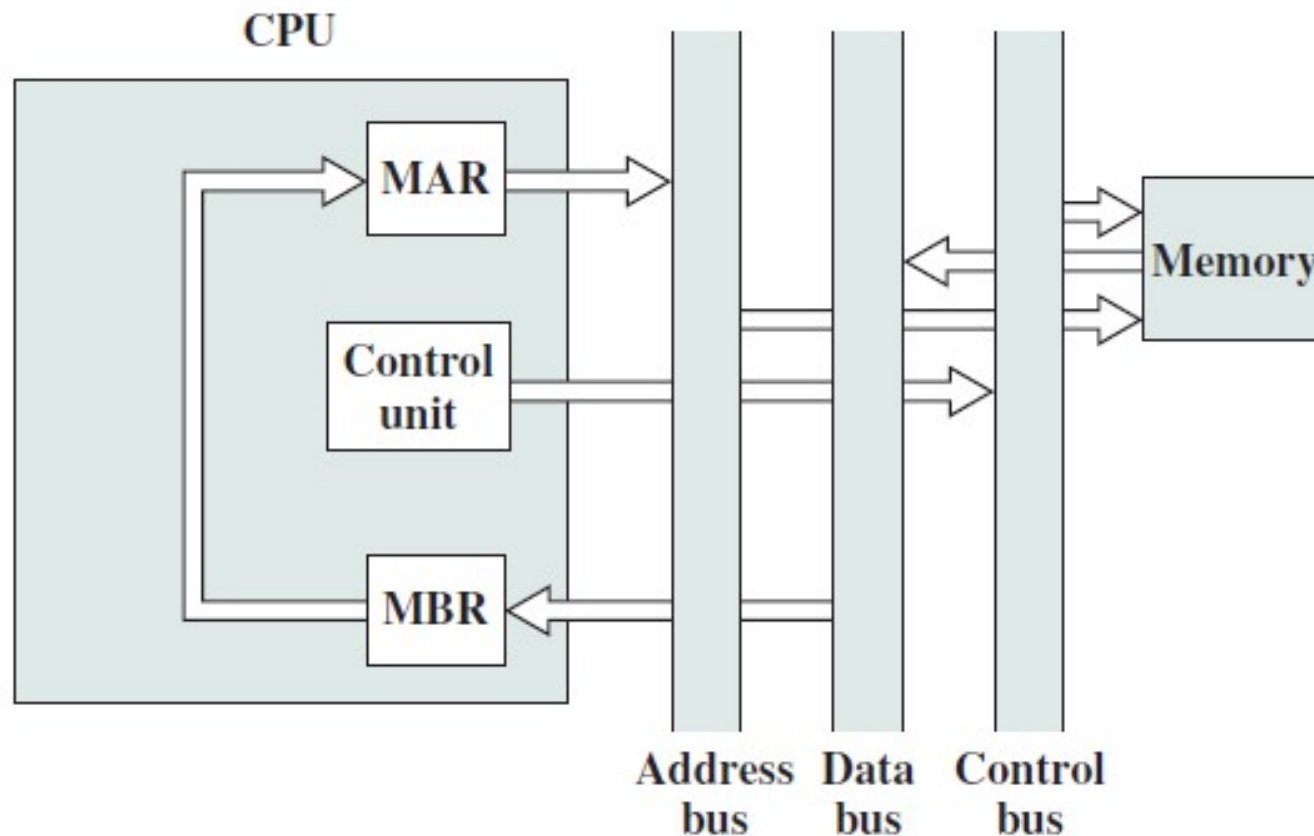


Figure 14.7 Data Flow, Indirect Cycle

CONT...

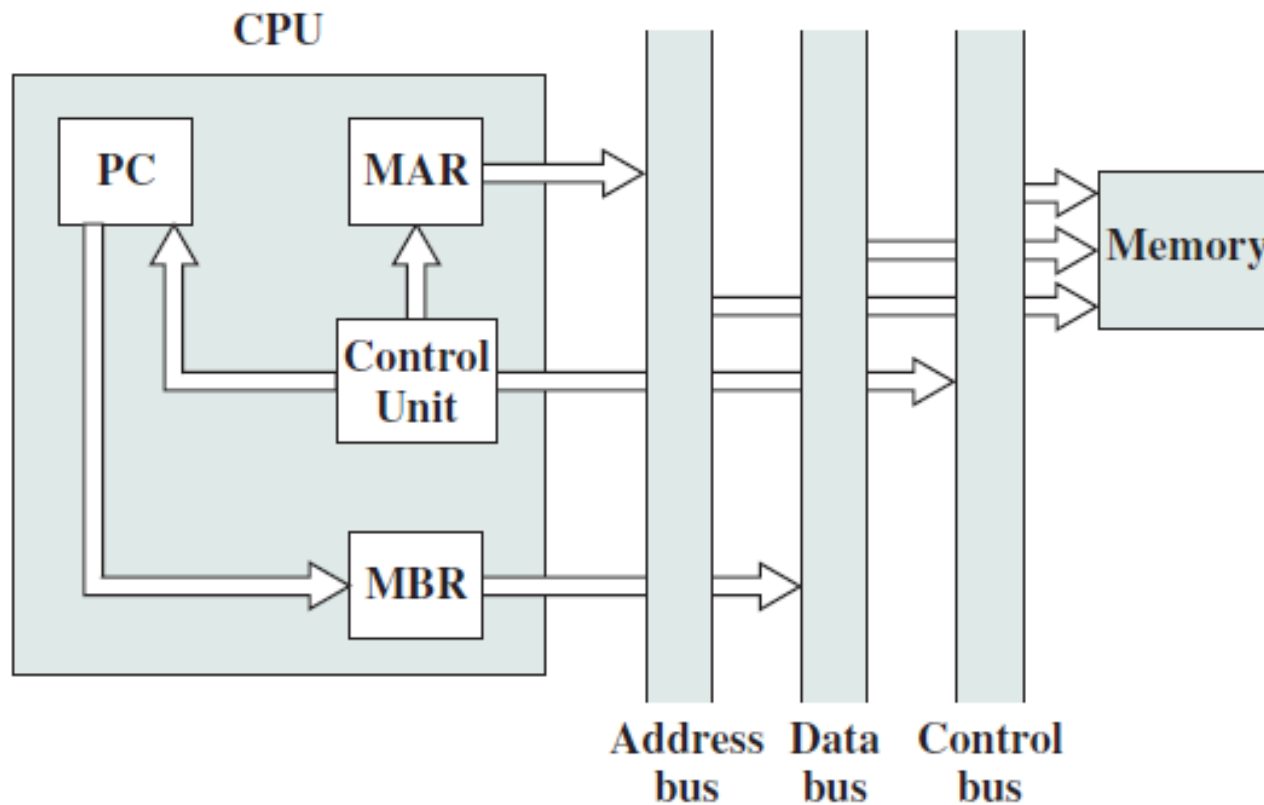
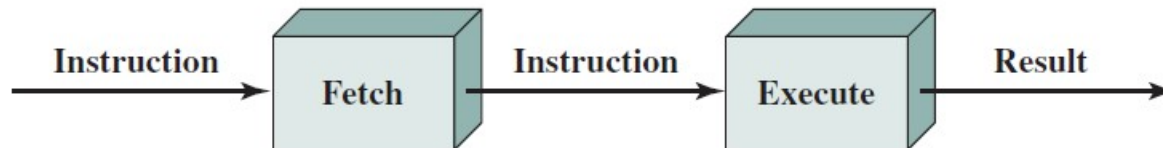


Figure 14.8 Data Flow, Interrupt Cycle

INSTRUCTION PIPELINING

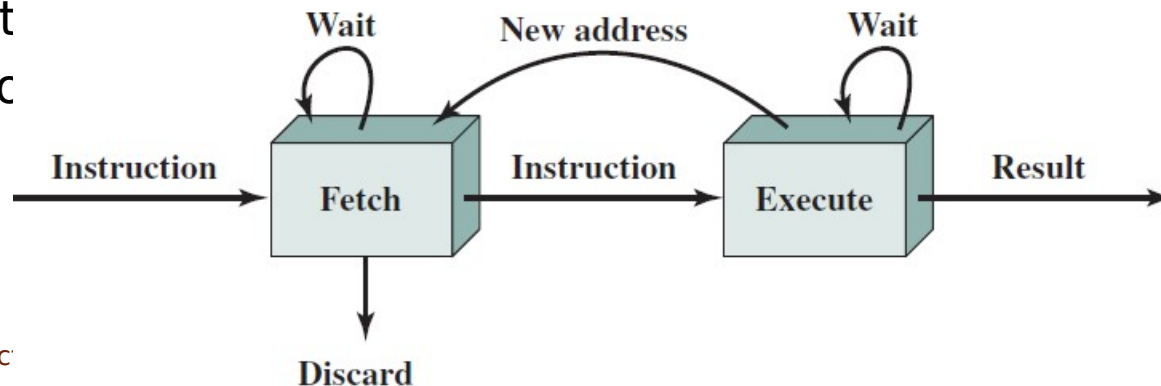
- Is similar to an **assembly line** in a manufacturing plant.
 - products at various stages can be worked on simultaneously.
 - new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.
- **Instruction pre-fetch / fetch overlap** :- to fetch the next instruction in parallel with the execution of the current one.
- Pipelining **requires registers** to store data between stages. (involves instructions buffering)
- This process will **speed up instruction execution**. (if equal duration stages)



CON...

In reality, doubling of execution rate is **unlikely** for two reasons

- The execution time will generally be **longer** than the fetch time.
 - wait for some time before it can empty its buffer
- A conditional **branch** instruction makes the **address of the next instruction** to be fetched **unknown**.
 - wait until it receives the next instruction address from the execut
- Any Sc



CON...

- To gain further speedup, the **pipeline must have more stages.**
- Consider the following decomposition of the instruction processing.
 - Fetch instruction (FI)
 - Decode instruction (DI)
 - Calculate operands (CO)
 - Fetch operands (FO)
 - Execute instruction (EI)
 - Write operand (WO)
- let us assume equal duration for each stages

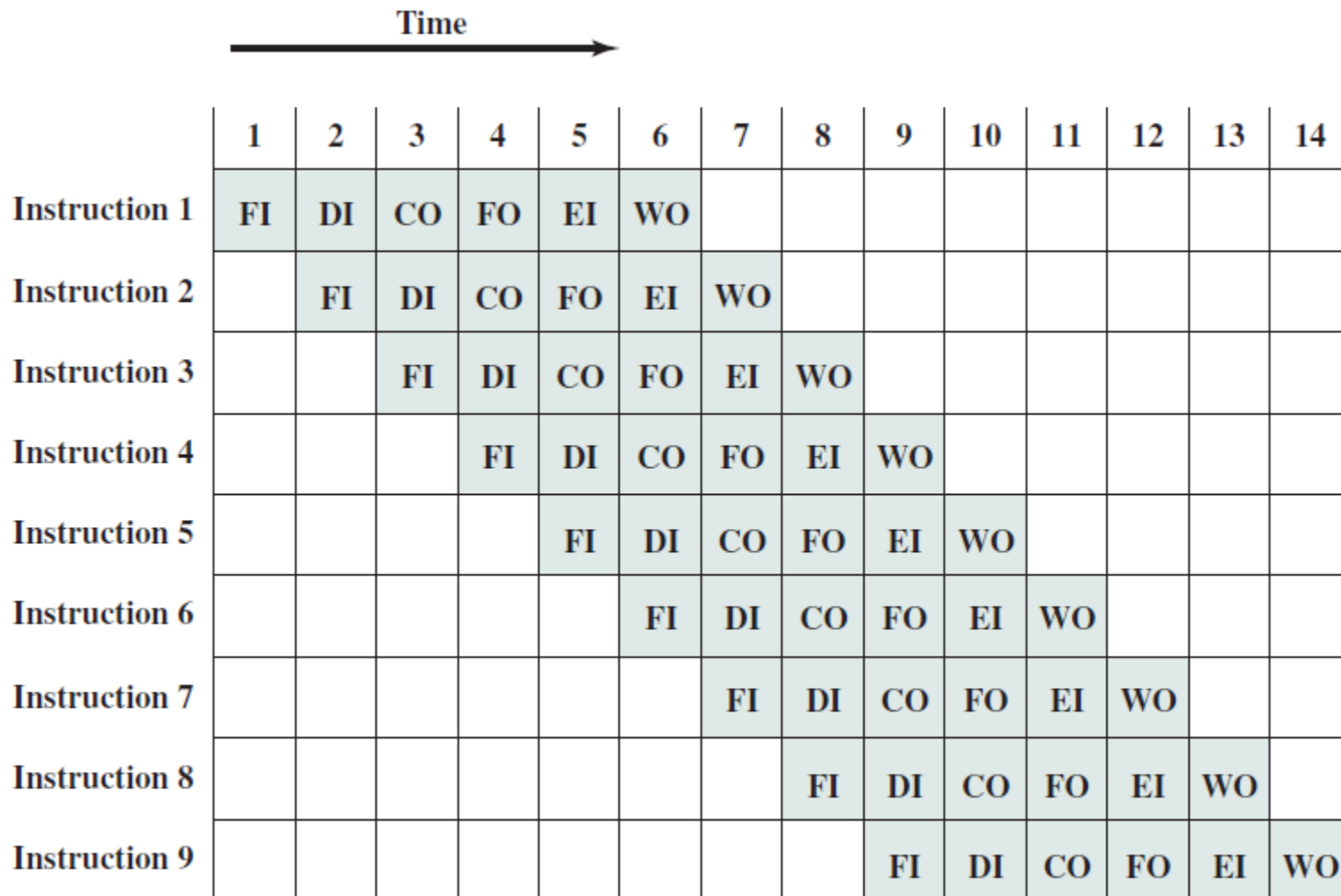


Figure 14.10 Timing Diagram for Instruction Pipeline Operation

This six- stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

CONT...

Keep in mind that in the pervious diagram we assume that

- Each instruction **goes through all six stages** of the pipeline.
- All of the stages **can be performed in parallel**.
- There are **no memory conflicts**. (FI, FO, and WO can occur simultaneously)
- Another difficulty is **the conditional branch** instruction, which can invalidate several instruction fetches.

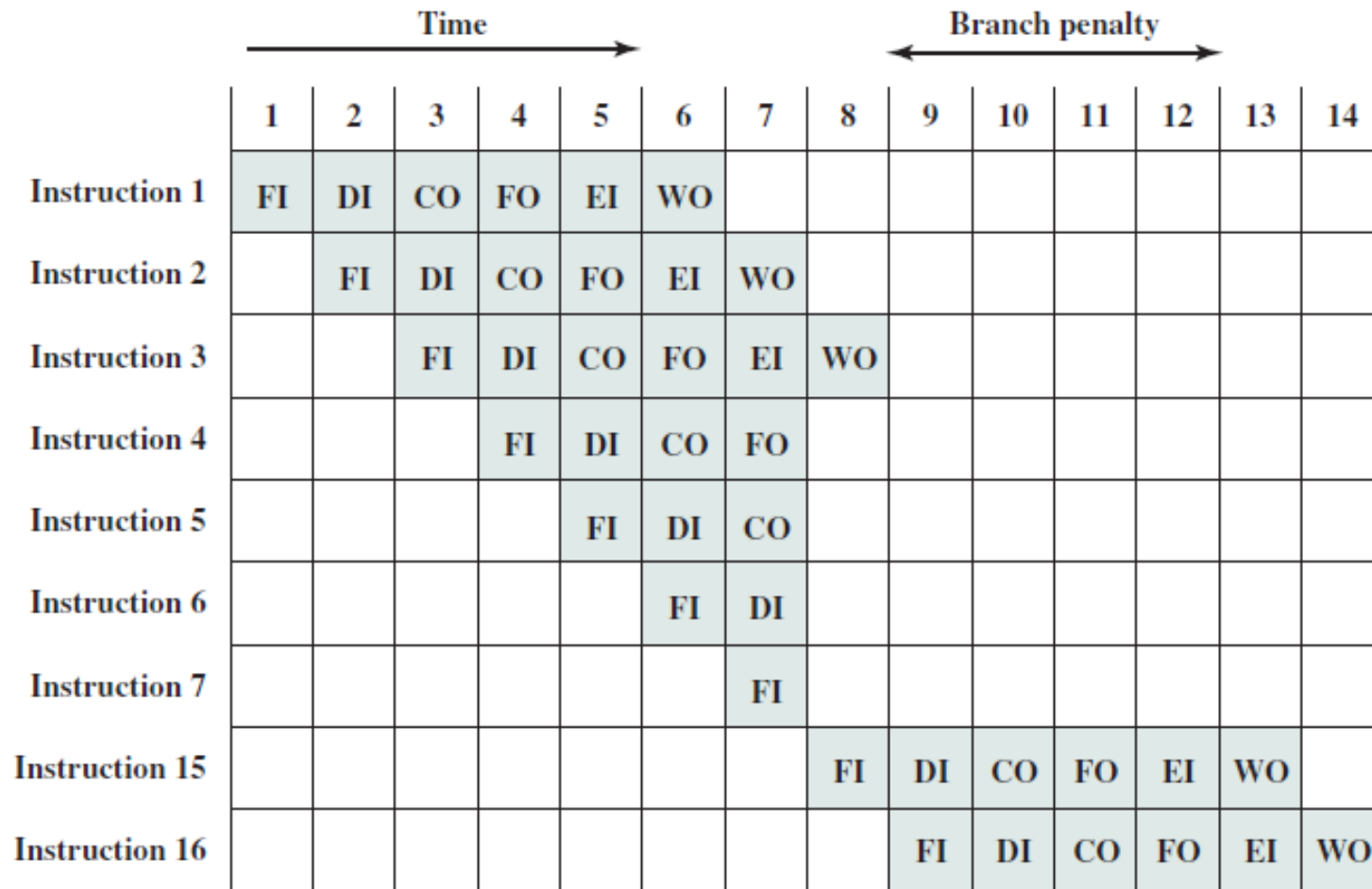


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Time
↓

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

(a) No branches

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

(b) With conditional branch

Figure 14.13 An Alternative Pipeline Depiction

CONT..

- The greater the number of stages in the pipeline, the faster the execution rate. But what is the burden
- Movement overhead
 - There is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions.
 - This **can lengthen the total execution time** of a single instruction. This is significant when sequential instructions **are logically dependent**
- Very complex:
 - The amount of control logic required increases enormously with the number of stages.

PIPELINE HAZARDS

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, **must stall** because conditions **do not permit continued execution**.

- **Resource hazards/structural hazard:-** occurs hardware resource can't support parallel execution of instructions.
 - So instructions must be executed in serial
 - Eg. main memory has a single read/write port, source operand for instruction I1 is in memory

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			FI	DI	FO	EI	WO		
	I4				FI	DI	FO	EI	WO	

(a) Five-stage pipeline, ideal case

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			Idle	FI	DI	FO	EI	WO	
	I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

CONT...

Data hazards: occurs when there is a conflict in the access of an operand location.

- Two instructions in a program are to be executed in sequence and both access a particular memory or register operand

ADD R1, R2 $\Rightarrow$ $R1 = R1 + R2$

SUB R3, R1 $\Rightarrow$ $R3 = R3 - R1$

		Clock cycle									
		1	2	3	4	5	6	7	8	9	10
ADD	R1, R2	FI	DI	FO	EI	WO					
SUB	R3, R1		FI	DI	Idle		FO	EI	WO		
I3				FI			DI	FO	EI	WO	
I4							FI	DI	FO	EI	WO

DATA HAZARDS TYPES: RAW

I1	ADD R1, R2
I2	SUB R3, R1
I3	MOV R4, R1
I4	MUL R7, R8

There are three types of data hazards: **RAW**, **WAR** and **WAW**

- **Read after write (RAW)**, or **true dependency**: An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location.
 - A hazard occurs if the read takes place before the write operation is complete.
 - This is the **most common** type of hazard and the kind that we use **forwarding** to overcome

	Clock cycle								
	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	Idle		FO	EI	WO	
I3			FI	DI	Idle	FO	EI	WO	
I4				FI	DI	FO	EI	WO	

without forwarding

	Clock cycle								
	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			FI	DI	FO	EI	WO		
I4				FI	DI	FO	EI	WO	

with forwarding

DATA HAZARDS TYPES: WAR

- **Write after read (WAR), or anti-dependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location.
 - A hazard occurs if the write operation completes before the read operation takes place.
 - This can not happen in our example pipeline because all reads are early (in FO) and all writes are late (in WO).
 - This hazard occurs when there are some instructions that **write results early** in the instruction pipeline, and other instructions that **read a source late** in the pipeline.
 - WAR hazards are **rare**

DATA HAZARDS TYPES: WAW

- **Write after write (WAW), or output dependency:**
Two instructions both write to the same location.
 - A hazard occurs if the write operations take place in the reverse order of the intended sequence.
 - This data hazard only occurs if there are more than one write stages in the pipeline.

CONT...

Control hazards

- A control hazard, also known as a *branch hazard*
- Occurs when the pipeline makes the wrong decision on a branch prediction
 - therefore brings instructions into the pipeline that must subsequently be discarded.

DEALING WITH BRANCHES

Multiple streams

- A brute-force approach
- Allow the pipeline to fetch both instructions, making use of two streams
- Disadvantage:- **contention delays** for access to the registers and to memory & **additional branch instructions**
- IBM 370/168 and the IBM 3033

Memory Address	Instruction
1	ADD R1, R2
2	BRZ 7
3	SUB R4, R1
⋮	⋮
7	MOV R4, 10010
8	MUL R3, R2

	Clock cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
I1	FI	DI	FO	EI	WO							
I2		FI	DI	FO	EI	WO						
I3			FI	DI	FO	EI	WO					
I4				FI	DI	FO	EI	WO				
I5					FI	DI	FO	EI	WO			
I6						FI	DI	FO	EI	WO		
I7							FI	DI	FO	EI	WO	
I8								FI	DI	FO	EI	WO

Pipeline stream 1 (by assuming the branch not taken)

	Clock cycle								
	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			FI	DI	FO	EI	WO		
I7				FI	DI	FO	EI	WO	
I8					FI	DI	FO	EI	WO

Pipeline stream 2
(by assuming the branch is taken)

DEALING WITH BRANCHES

- **Pre-fetch branch target**

- The target of the branch is **pre-fetched** in addition to the instruction following the branch
 - then saved **until the branch instruction is executed**.
- If the branch is taken, the target has already been pre-fetched.
- Used in IBM 360/91

CONT...

- **Loop buffer**

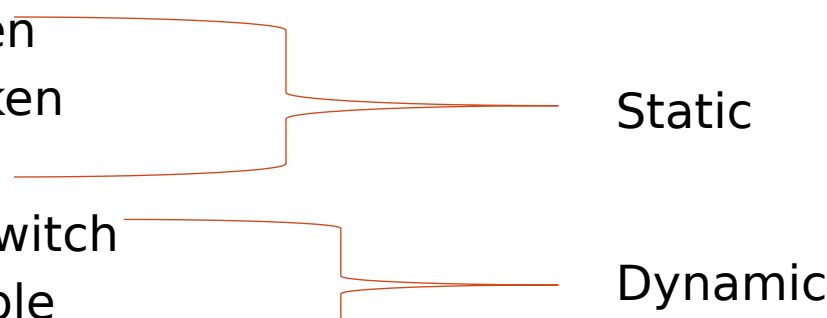
- A loop buffer is a small, very-high-speed memory
 - maintained by the instruction fetch stage of the pipeline and
 - containing the *n* most recently fetched instructions, in sequence.
 - If a branch is to be taken, the hardware first checks whether the branch target is within the buffer.
 - If so, the next instruction is fetched from the buffer.
-
- The loop buffer has three benefits:
 - No memory reference required
 - Useful for IF-THEN and IF-THEN-ELSE sequences
 - This strategy is particularly well suited to dealing with loops, or iterations

CONT...

- **Branch prediction:-** Various techniques can be used to predict whether a branch will be taken.
 - Predict never taken
 - Predict always taken
 - Predict by opcode

Static

 - Taken/not taken switch
 - Branch history table

Dynamic
- 

Static: they do not depend on on the execution history

dynamic: They depend on the execution history

- Delayed branch: automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.



PART 3

LEARNING OBJECTIVES

- To know the difference between **RISC** and **CISC** architecture.
- To understand the **micro-operations** involved in instruction cycle.
- Discuss the inputs and outputs of **Control Unit** and how it works.
- To understand the two implementations of control unit.

CPU IS ARCHITECTURE: CISC vs RISC

- Two Instruction set architectures
 - Complex Instruction Set Computers (CISC)
 - Reduced Instruction Set Computers (RISC)

Example: *Multiply the content of two memory locations and store the result to another memory location?*

- **The CISC Approach**

- The primary goal of CISC architecture is to complete a task **in as few lines of assembly as possible**.
- This is achieved by building **processor hardware** that is **capable of understanding and executing a complex instruction**.

MULTI [loc 3] , [loc 1], [loc 2]

- When executed, this instruction loads the two values from memory into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register (R1).

**A single instruction takes
more than 1 clock cycle to finish**

CONT...THE CISC APPROACH

- Instructions for CISC arc **closely resembles** a command in a higher level language.
 - Potential advantage ? the compiler **has to do very little work** to translate a high-level language statement into assembly.
- Because the length of the code is relatively short, **very little RAM is required** to store instructions.
- The emphasis is put on building complex instructions directly into the hardware.
- The CISC approach **attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.**

THE RISC APPROACH

- RISC processors only **use simple instructions** that can be executed within **one clock cycle**.
 - LOAD A, [loc 1]
 - LOAD B, [loc 2]
 - PROD A, B
 - STORE [loc 3] , A
- Because there are **more lines of code**, **more RAM** is needed to store the assembly level instructions.
- The compiler must also **perform more work** to convert a high-level language statement into code of this form.
- The RISC approach **reduces the cycles per instruction at the cost of the number of instructions per program**.

- Because each instruction requires only one clock cycle to execute, the entire program in RISC will execute in approximately the same amount of time as of CISC.
 - **Potential advantage** ? RISC "reduced instructions" **require less transistors of hardware space** than the complex instructions, leaving more room for general purpose registers.
 - Because all of the instructions execute in a uniform amount of time (i.e. one clock), **pipelining is possible**.
- **What happen to the registers after we finish the current program?**
 - In CISC:- processor automatically **erases the registers**. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register.
 - In RISC:- the operand **will remain** in the register **until another value is loaded in its place**.

CISC	RISC
The original ISA	Redesigned on the original ISA
Instruction can take several clock cycle	Instruction take one clock cycle
Hardware-centric design	Software-centric design
More efficient use of RAM than RISC	Heavy use of RAM
Complex and variable length instructions	Simple standardized instructions
Sequential instruction execution	Parallel instruction execution
Complexity in CPU (micro-programs)	Complexity in compiler
Small code size	Large code size
Pipeline is difficult	Pipeline is easy
More addressing modes	Fewer addressing mode
Have less registers	Have more registers

Micro-operation and Control Unit

- *What are micro-operations*
 - *Grouping rule*
 - *Examples*
- *Input and output of CU*
- *Implementation of CU*

MICRO-OPERATIONS

- The execution of a program consists of the **sequential execution of instructions**.
- Each instruction is executed during an instruction cycle made up of shorter **subcycles**
 - (fetch, indirect, execute, interrupt).
- The execution of each subcycle involves one or more shorter operations, that is, **micro-operations**.
- Micro-operations are the **functional, or atomic, operations** of a processor.
- The prefix **micro** refers to the fact that each step is **very simple** and **accomplishes very little**.

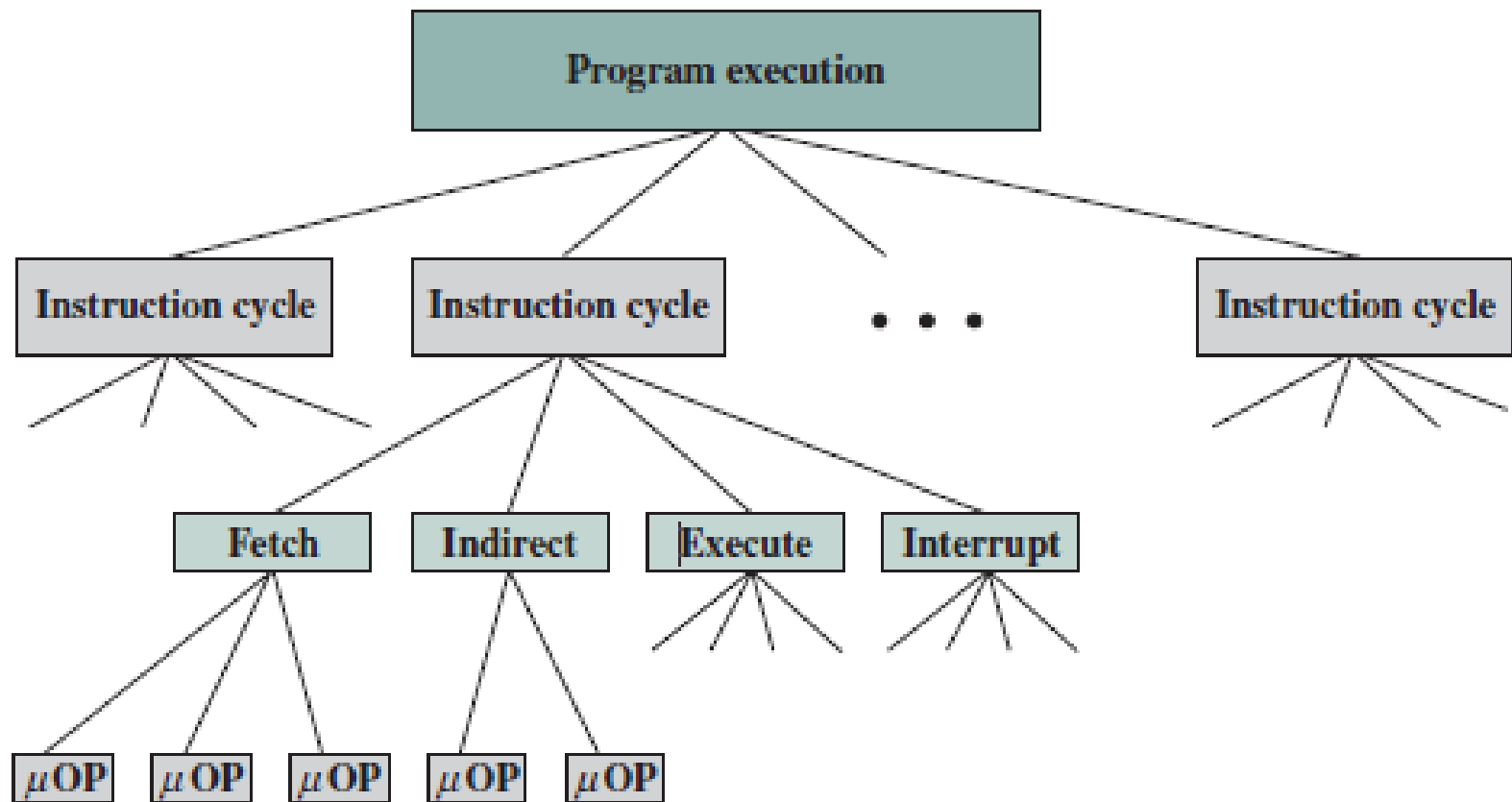
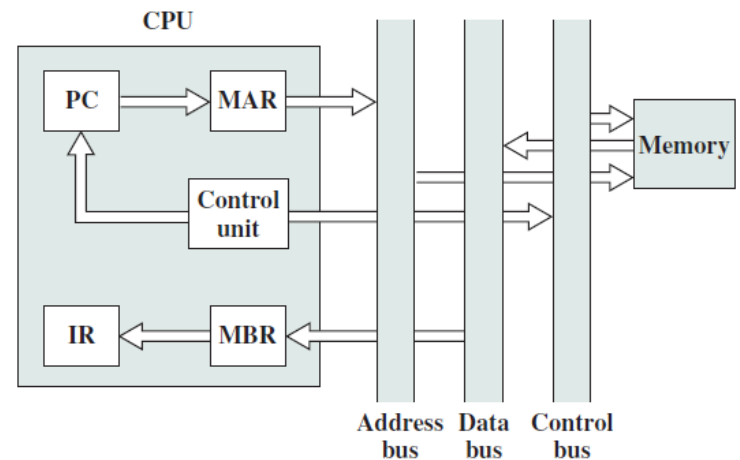


Figure 20.1 Constituent Elements of a Program Execution

OP'S IN FETCH CYCLE

- Four registers are involved: MAR, MBR, PC, IR
- Simple fetch cycle actually consists of **3 steps** and **4 OP**
- Each μ OP involves the movement of data into or out of a register.

t_1 : MAR $\leftarrow$ (PC)
 t_2 : MBR $\leftarrow$ Memory
PC $\leftarrow$ (PC) + I
 t_3 : IR $\leftarrow$ (MBR)



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Figure 14.6 Data Flow, Fetch Cycle

RULES OF μ OP GROUPING

1. The proper sequence of events must be followed.
 - Thus (MAR $\Leftarrow$ PC) must precede (MBR $\Leftarrow$ Memory) because the memory read operation makes use of the address in the MAR.
2. Conflicts must be avoided.
 - One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable.
 - For example, the micro- operations (MBR $\Leftarrow$ Memory) and (IR $\Leftarrow$ MBR) should not occur during the same time unit.

Note: As long as each μ OP follows the above rules and do not interfere with one another, several of them can take place during one step, to saving time.

OP'S IN INDIRECT CYCLE

- let us assume a one-address instruction format, with direct and indirect addressing allowed.
- The address field of the instruction is transferred to the MAR.
- This is then used to fetch the address of the operand.
- Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

$t_1: \text{MAR} \leftarrow (\text{IR}(\text{Address}))$

$t_2: \text{MBR} \leftarrow \text{Memory}$

$t_3: \text{IR}(\text{Address}) \leftarrow (\text{MBR}(\text{Address}))$

OP'S IN INTERRUPT CYCLE

- The nature of this cycle varies greatly from one machine to another.

t_1 : MBR $\leftarrow$ (PC)
 t_2 : MAR $\leftarrow$ Save_Address
PC $\leftarrow$ Routine_Address
 t_3 : Memory $\leftarrow$ (MBR)

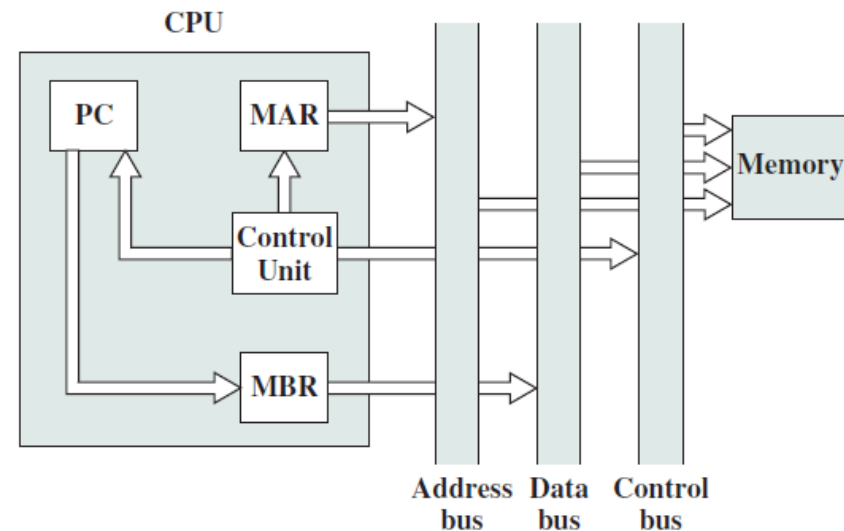


Figure 14.8 Data Flow, Interrupt Cycle

OP'S IN EXECUTE CYCLE

- This cycle is different, because of the variety of opcodes, there are a number of **different sequences of micro-operations**
- ADD R1, X
 - t1: $MAR \Leftarrow (IR(address))$
 - t2: $MBR \Leftarrow Memory$
 - t3: $R1 \Leftarrow (R1) + (MBR)$
- ISZ X
 - t1: $MAR \Leftarrow (IR(address))$
 - t2: $MBR \Leftarrow Memory$
 - t3: $MBR \Leftarrow (MBR) + 1$
 - t4: $Memory \Leftarrow (MBR)$
 - If $((MBR) = 0)$ then $(PC \Leftarrow (PC) + I)$

BSA X

- t₁: $MAR \Leftarrow (IR(address))$
 $MBR \Leftarrow (PC)$
- t₂: $PC \Leftarrow (IR(address))$
 $Memory \Leftarrow (MBR)$
- t₃: $PC \Leftarrow (PC) + I$

THE INSTRUCTION CYCLE

- Defines the complete sequence of micro-operations
- We assume a new 2-bit register called the *instruction cycle code* (ICC).
- The ICC designates the state of the processor in terms of which portion of the cycle it is in:
 - 00: Fetch
 - 01: Indirect
 - 10: Execute
 - 11: Interrupt
- At the end of each of the four cycles, the ICC is set appropriately.

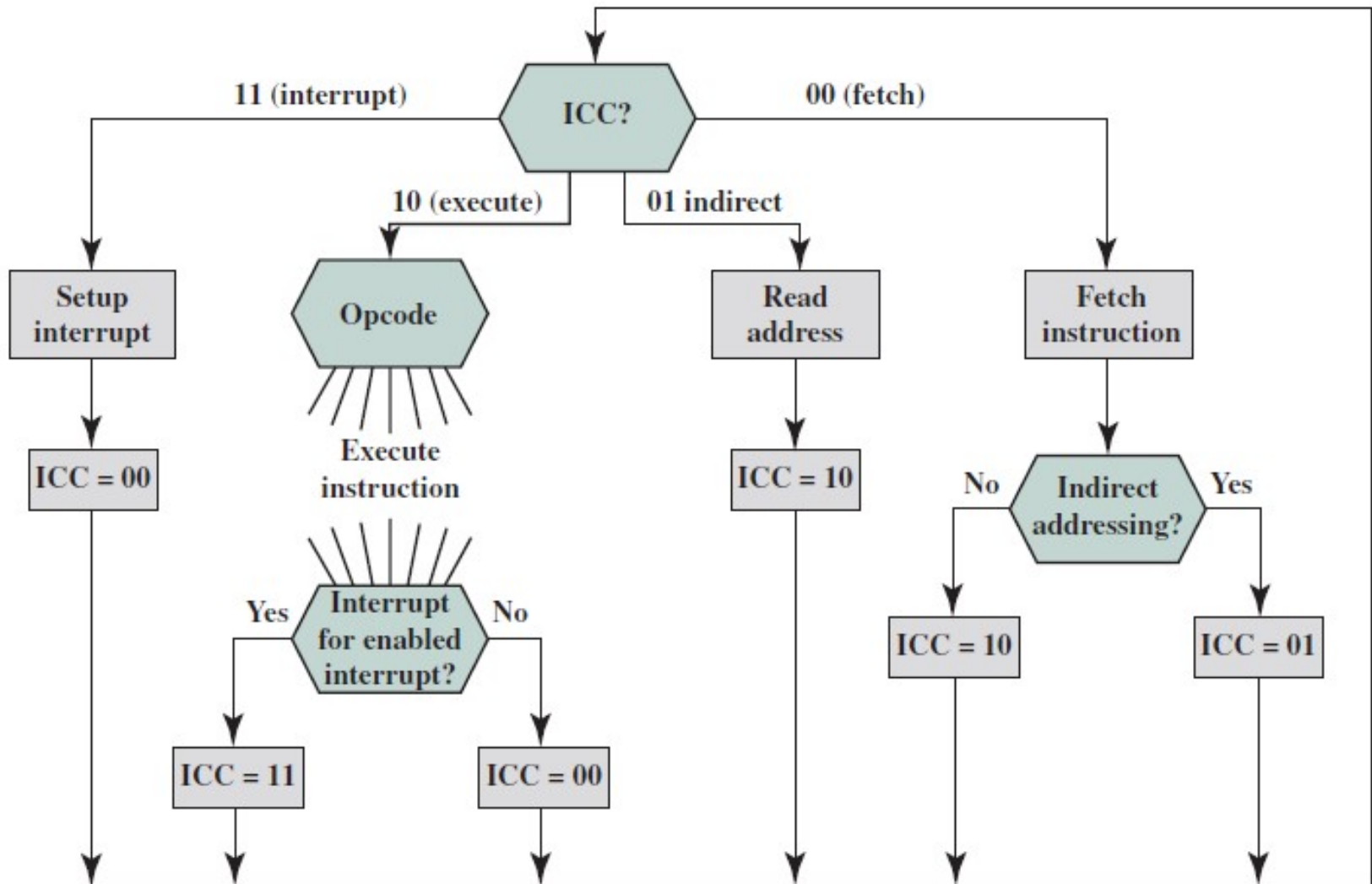


Figure 20.3 Flowchart for Instruction Cycle

CONTROL OF THE PROCESSOR

Define basic elements of processor ?

Describe micro-operations processor performs ?

Determine functions control unit must perform ?

- The functional requirements of the control unit: (those functions that the control unit must perform)
 - **Sequencing**: The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
 - **Execution**: The control unit causes each micro-operation to be performed with the use of control signals

EXTERNAL SPECIFICATIONS OF THE CONTROL UNIT

IN
PU
TS

- **Clock** ---This is how the control unit “keeps time.”
 - One micro-instruction (or set of parallel micro-instructions) per clock cycle
- **Instruction register**
 - Op-code and addressing mode for current instruction
 - Determines which micro-instructions to perform
- **Flags**
 - State of processor
 - Results of previous ALU operations
- **Control signals from control bus**
 - Interrupts
 - Acknowledgements

CONT...

The outputs are as follows:

- **Control signals within the processor:**
 - These are two types: Cause data movement & Activate specific ALU functions
- **Control signals to control bus:**
 - These are also of two types: control signals to memory, and control signals to the I/O modules.

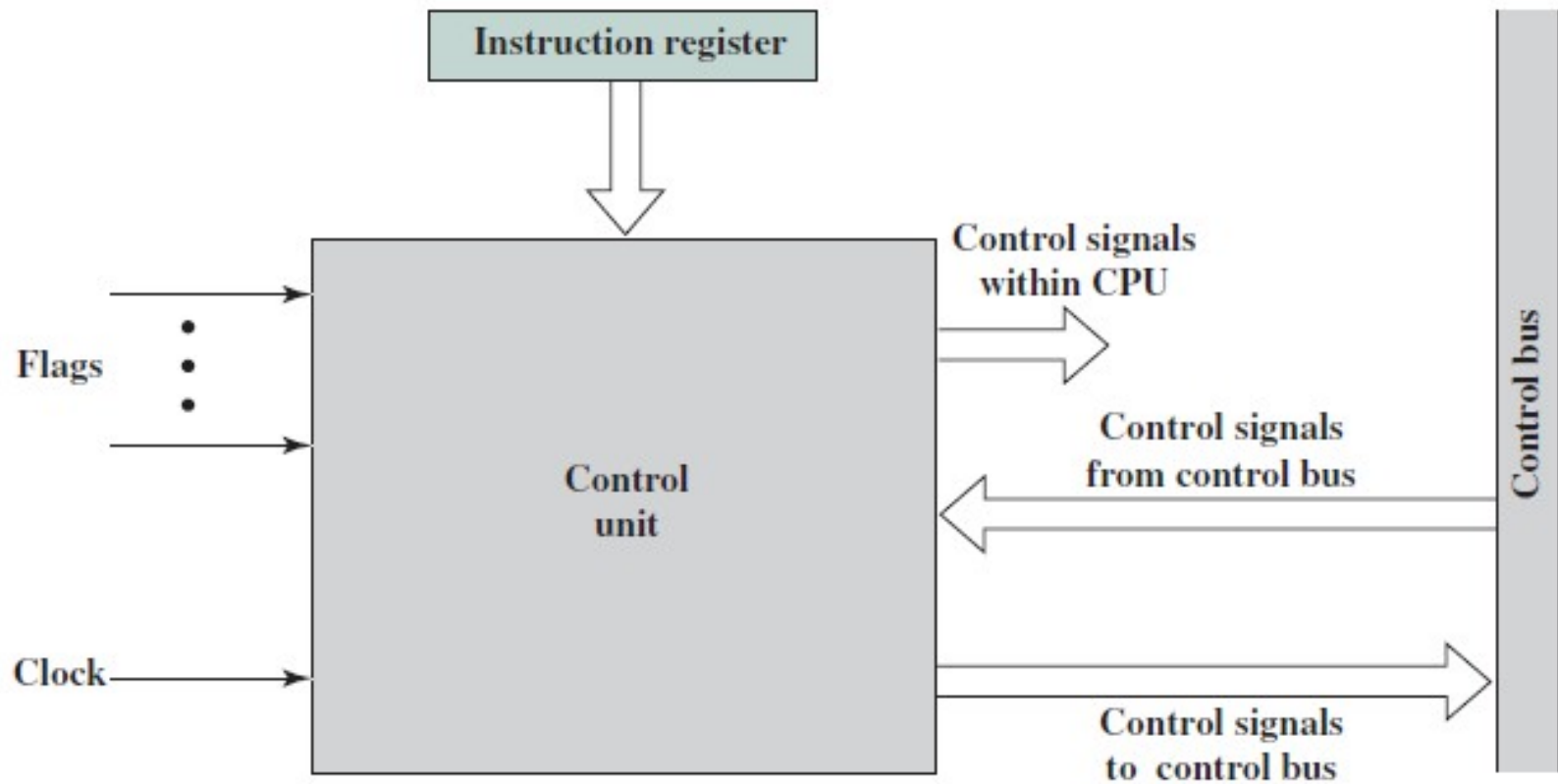


Figure 20.4 Block Diagram of the Control Unit

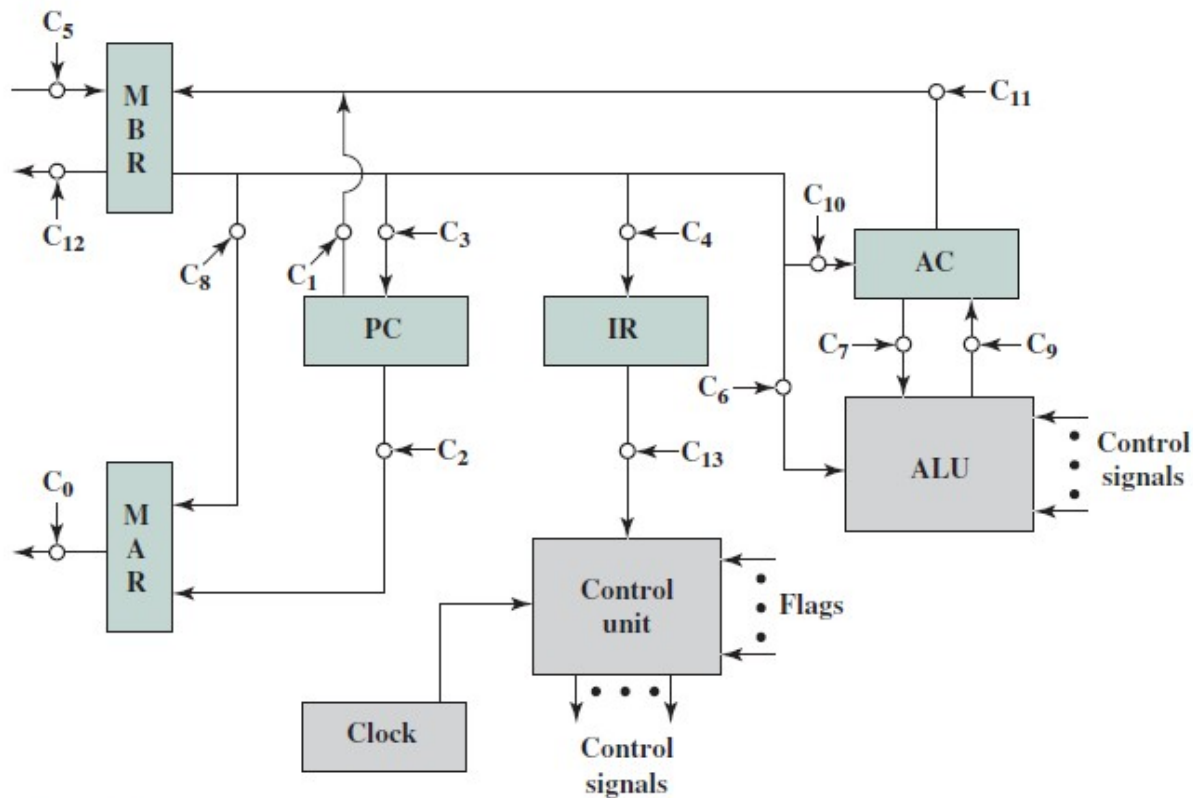
CONTROL SIGNALS

- Three types of control signals are used:
 - Those that **activate an ALU function**;
 - Those that **activate a data path**; and
 - Those that are **signals on the external system** bus or other external interface.
- All of these signals are ultimately applied directly **as binary inputs to individual logic gates**
- Let us consider again the fetch cycle to see how the control unit maintains control.

CONTROL SIGNALS IN FETCH CYCLE

- The first step is to transfer the contents of the PC to the MAR.
 - CU activate the control signal that opens the gates between the bits of the PC and the bits of the MAR.
- The next step is to read a word from memory into the MBR and increment the PC.
 - A control signal that opens gates, allowing the contents of the MAR onto the address bus;
 - A memory read control signal on the control bus;
 - A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR;
 - Control signals to logic that add 1 to the contents of the PC and store the result back to the PC.
- Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

CONTROL SIGNALS IN FETCH CYCLE



t_1 : MAR $\leftarrow$ (PC)
 t_2 : MBR $\leftarrow$ Memory
 PC $\leftarrow$ (PC) + I
 t_3 : IR $\leftarrow$ (MBR)

Figure 20.5 Data Paths and Control Signals

A CONTROL SIGNALS EXAMPLE

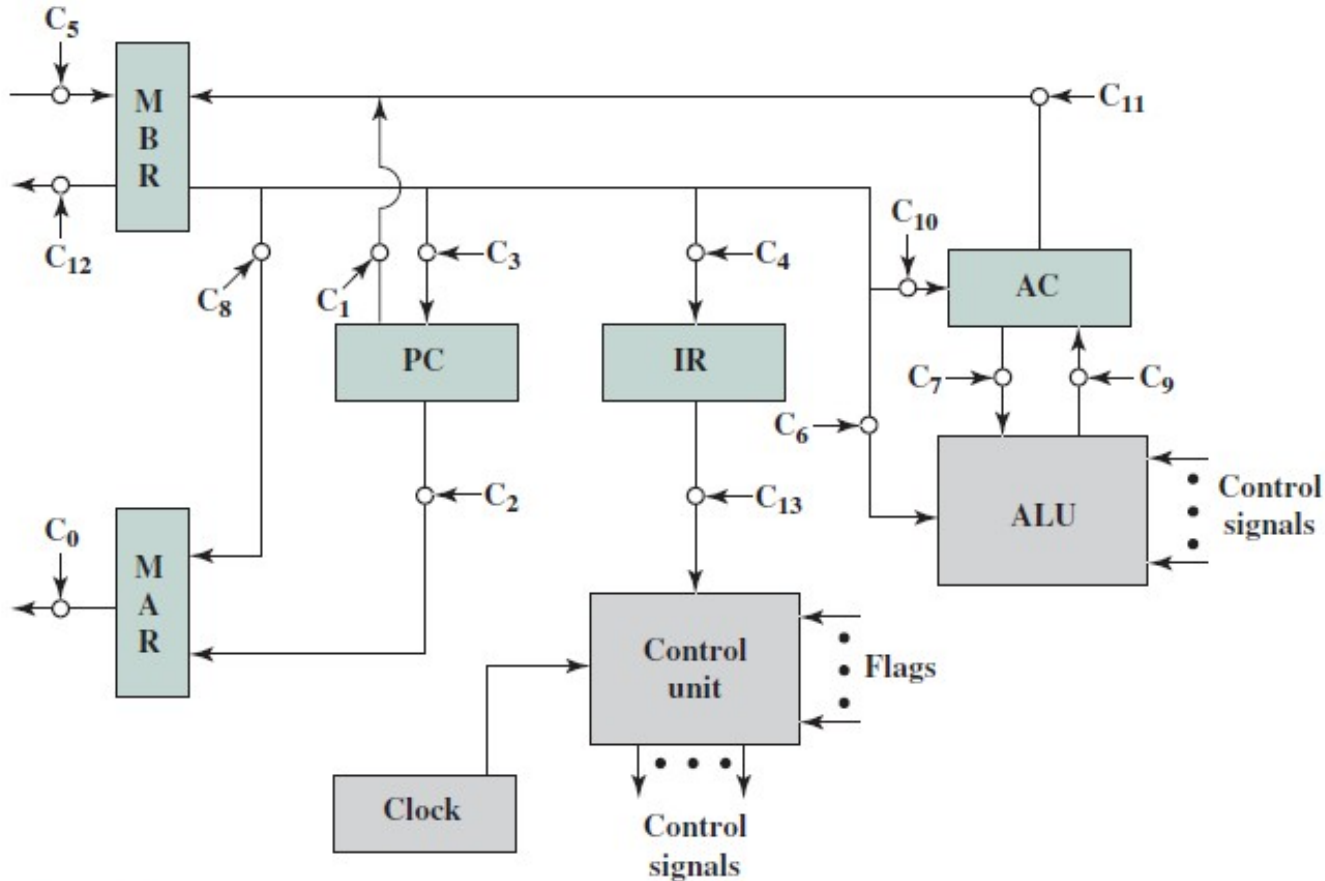


Figure 20.5 Data Paths and Control Signals

Table 20.1 Micro-operations and Control Signals

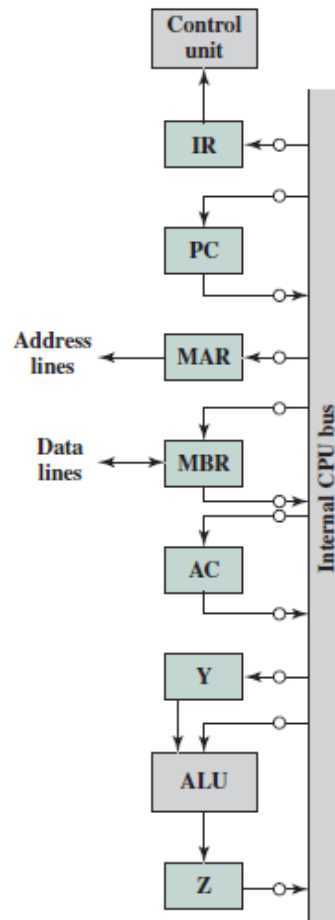
	Micro-operations	Active Control Signals
Fetch:	$t_1: \text{MAR} \leftarrow (\text{PC})$	C_2
	$t_2: \text{MBR} \leftarrow \text{Memory}$ $\text{PC} \leftarrow (\text{PC}) + 1$	C_5, C_R
	$t_3: \text{IR} \leftarrow (\text{MBR})$	C_4
Indirect:	$t_1: \text{MAR} \leftarrow (\text{IR}(\text{Address}))$	C_8
	$t_2: \text{MBR} \leftarrow \text{Memory}$	C_5, C_R
	$t_3: \text{IR}(\text{Address}) \leftarrow (\text{MBR}(\text{Address}))$	C_4
Interrupt:	$t_1: \text{MBR} \leftarrow (\text{PC})$	C_1
	$t_2: \text{MAR} \leftarrow \text{Save-address}$ $\text{PC} \leftarrow \text{Routine-address}$	
	$t_3: \text{Memory} \leftarrow (\text{MBR})$	C_{12}, C_W

C_R = Read control signal to system bus.

C_W = Write control signal to system bus.

For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.

INTERNAL PROCESSOR ORGANIZATION



CU IMPLEMENTATION

Control unit can be implemented using

- Hardwired implementation
- Microprogrammed implementation

Hardwired Control Unit

- When the control signals **are generated by hardware** using **conventional logic design** techniques, the CU is said to be hardwired.
- Can be viewed as a **state machine** that changes from one state to another in every clock cycle, depending on the inputs of CU.
- The **outputs** of the state machine are the control signals.
- The sequence of the operation carried out by this machine is determined by the **wiring of the logic elements** and hence named as “hardwired”.

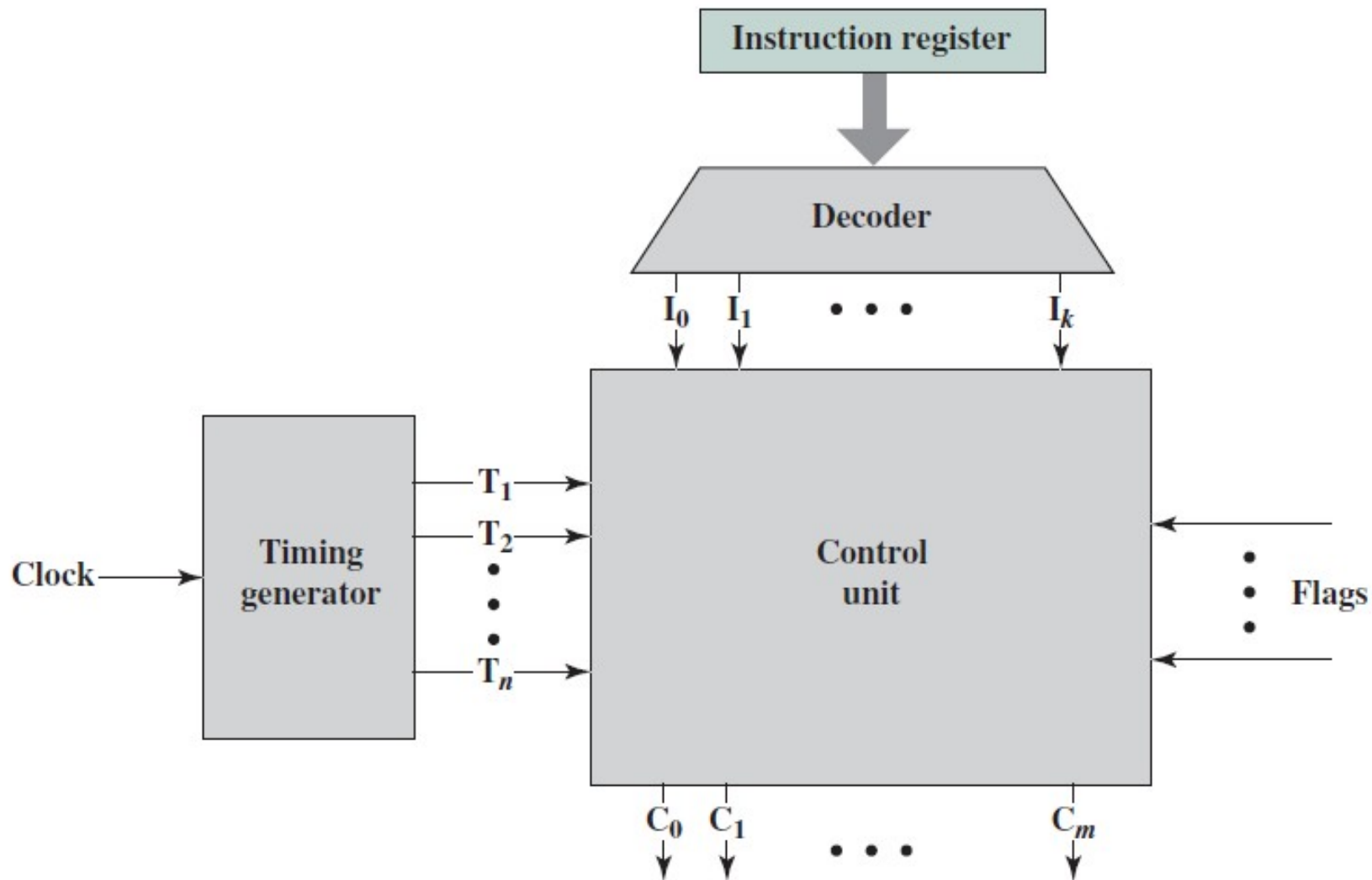


Figure 20.10 Control Unit with Decoded Inputs

SUMMARY OF HARDWIRED CU

- Fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals.
- Hardwired control is **faster than micro-programmed control**.
- A controller that uses this approach can **operate at high speed**.
- **RISC architecture** is based on hardwired control unit

Limitation of this implementation

- Complex sequencing & micro-operation logic
- Difficult to design and test
- Inflexible design
- Difficult to add new instructions

MICRO-PROGRAMMED IMPLEMENTATION OF CU

- The **control signals** associated with operations **are stored in special memory** units **inaccessible by the programmer** as **Control Words**.
- Control signals are **generated by a program** are similar to machine language programs.
- Micro-programmed CU is slower in speed because of **the time it takes to fetch microinstructions** from the control memory.
- Is the dominant technique for implementing control units in pure **CISC architectures processors**

SOME IMPORTANT TERMS IN MICRO-PROGRAMMED CU

- **Control Word** : is a word whose individual bits represent various control signals.
- **Micro-instruction** : Individual control words in this micro-routine are referred to as microinstructions.
- **Micro-program** : A **sequence of micro-instructions** is called a micro-program, which is stored in a ROM.
- **Micro-routine** : Group of micro-instructions each stored in control word that belongs to same routine.
- **Control Memory** : the micro-routines for all instructions in the instruction set of a computer are stored in a special memory called the Control Memory.

$$\begin{array}{l}
 \text{assembly} \\
 \text{program :}
 \end{array}
 \left\{
 \begin{array}{l}
 \text{instruction 1;} \\
 \text{instruction 2;} \\
 \dots \\
 \text{instruction } i; \\
 \dots \\
 \dots \\
 \text{instruction } j; \\
 \dots \\
 \dots \\
 \text{instruction } n;
 \end{array}
 \right.
 \begin{array}{l}
 \mu\text{-program for } I_i : \\
 \left\{
 \begin{array}{ll}
 \mu\text{-instruction 1} & (CW \ 1_i) \\
 \mu\text{-instruction 2} & (CW \ 2_i) \\
 \dots & \\
 \mu\text{-instruction } n_i & (CW \ n_i)
 \end{array}
 \right. \\
 \\
 \mu\text{-program for } I_j : \\
 \left\{
 \begin{array}{ll}
 \mu\text{-instruction } 1_j & (CW \ 1_j) \\
 \mu\text{-instruction } 2_j & (CW \ 2_j) \\
 \dots & \\
 \mu\text{-instruction } m_j & (CW \ m_j)
 \end{array}
 \right.
 \end{array}$$

ADVANTAGES AND DISADVANTAGES OF Micro-programmed implementation of CU

- Advantage
 - **Cheaper** and **less error prone** to implement.
 - Simple to design and test.
 - Flexible design..... easy to add new features and

- Disadvantage
 - Somewhat **slower** than a hardwired implementation

*To know more about this implementation refer to **Chapter 16 Micro-programmed Control** (page 586) in your textbook.*