+

Bahria University
Discovering Knowledge

Computer Architecture and Logic Design (CALD)

Lecture 08

Dr. Sorath Hansrajani

Assistant Professor

Department of Software Engineering

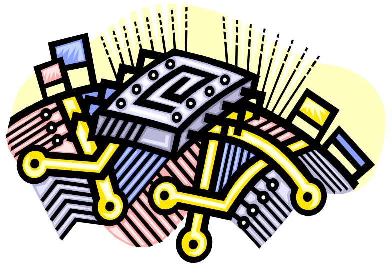
Bahria University Karachi Campus

Email: sorathhansrajani.bukc@bahria.edu.pk

Processor Structure and Function

Processor Organization

Processor Requirements:



- Fetch instruction
  - The processor reads an instruction from memory (register, cache, main memory)
- Interpret instruction
  - The instruction is decoded to determine what action is required
- Fetch data
  - The execution of an instruction may require reading data from memory or an I/O module
- Process data
  - The execution of an instruction may require performing some arithmetic or logical operation on data
- Write data
  - The results of an execution may require writing data to memory or an I/O module
- In order to do these things the processor needs to store some data temporarily and therefore needs a small internal memory

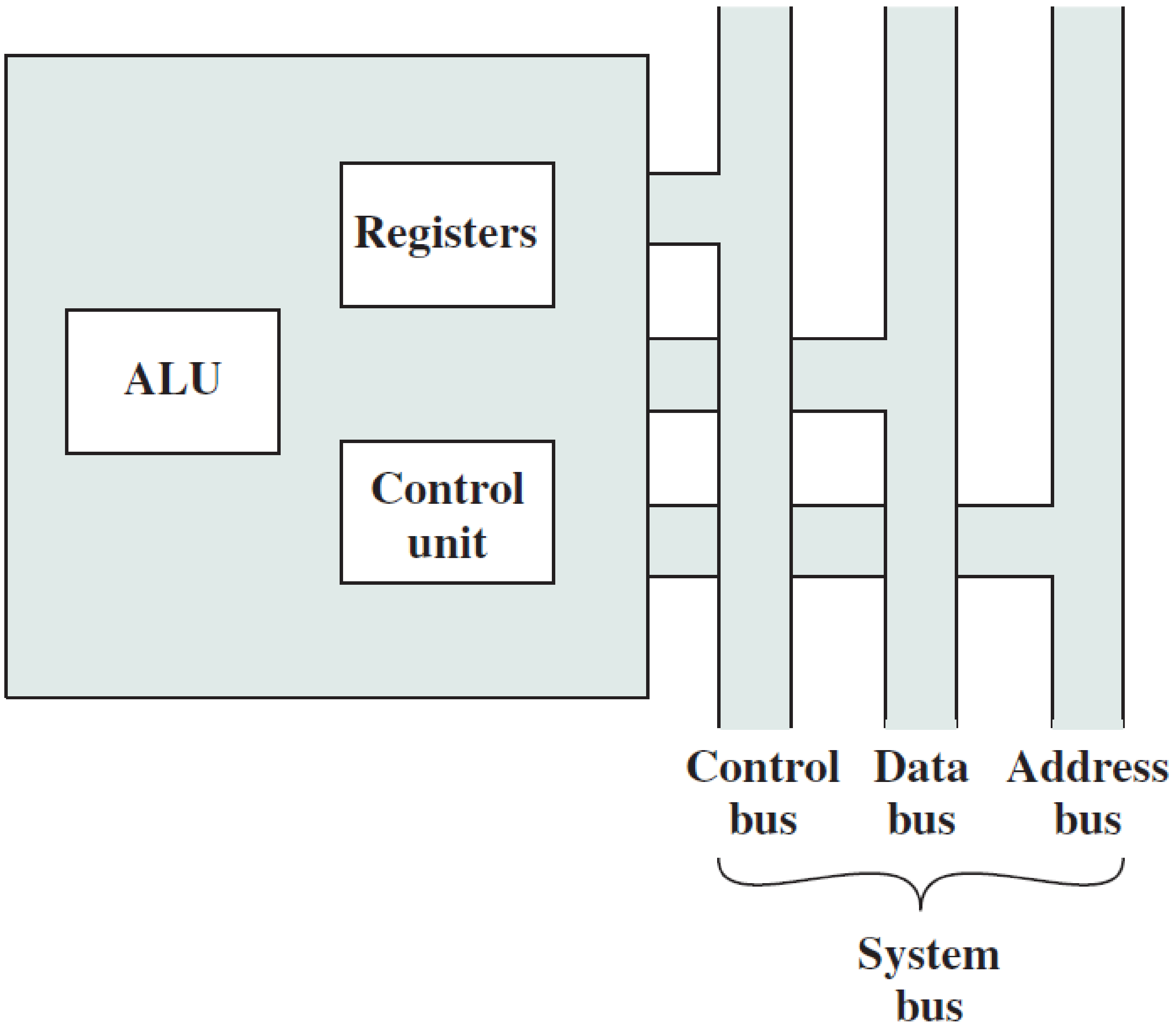


Figure 14.1 The CPU with the System Bus

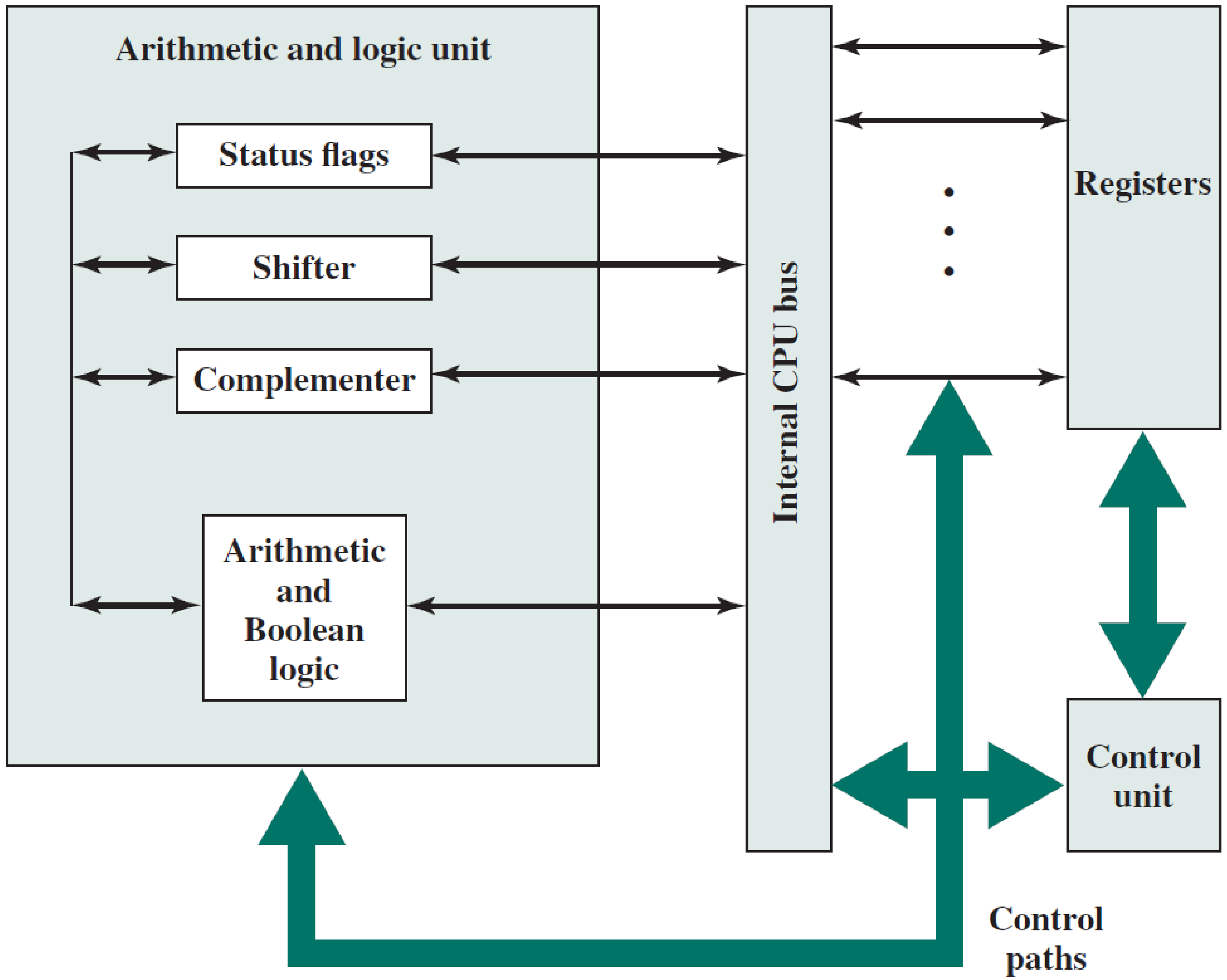


Figure 14.2 Internal Structure of the CPU

+

Register Organization

- Within the processor there is a set of registers that function as a level of memory above main memory and cache in the hierarchy
- The registers in the processor perform two roles:

User-Visible Registers	Control and Status Registers
Enable the machine or assembly language programmer to minimize main memory references by optimizing use of 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

User-Visible Registers

Referenced by means of the machine language that the processor executes

Categories:

- **General purpose**
  - Can be assigned to a variety of functions by the programmer
- **Data**
  - May be used only to hold data and cannot be employed in the calculation of an operand address
- **Address**
  - May be somewhat general purpose or may be devoted to a particular addressing mode
  - Examples: segment pointers, index registers, stack pointer
- **Condition codes**
  - Also referred to as *flags*
  - Bits set by the processor hardware as the result of operations

+
Condition Codes

Table 14.1 Condition Codes

Advantages	Disadvantages
1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed.	**1.** Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer.
2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST and BRANCH.	**2.** Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections.
3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.	**3.** Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations.
4. Condition codes can be saved on the stack during subroutine calls along with other register information.	**4.** In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

+

Control and Status Registers

Four registers are essential to instruction execution:

- Program counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Memory address register (MAR)
  - Contains the address of a location in memory
- Memory buffer register (MBR)
  - Contains a word of data to be written to memory or the word most recently read

+
Program Status Word (PSW)

Register or set of registers that contain status information

Common fields or flags include:

- Sign
- Zero
- Carry
- Equal
- Overflow
- Interrupt Enable/Disable
- Supervisor

+

Common Flags in PSW Register

- 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 (sub- traction) out of a high-order bit. Used for multiword arithmetic operations.
- 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. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

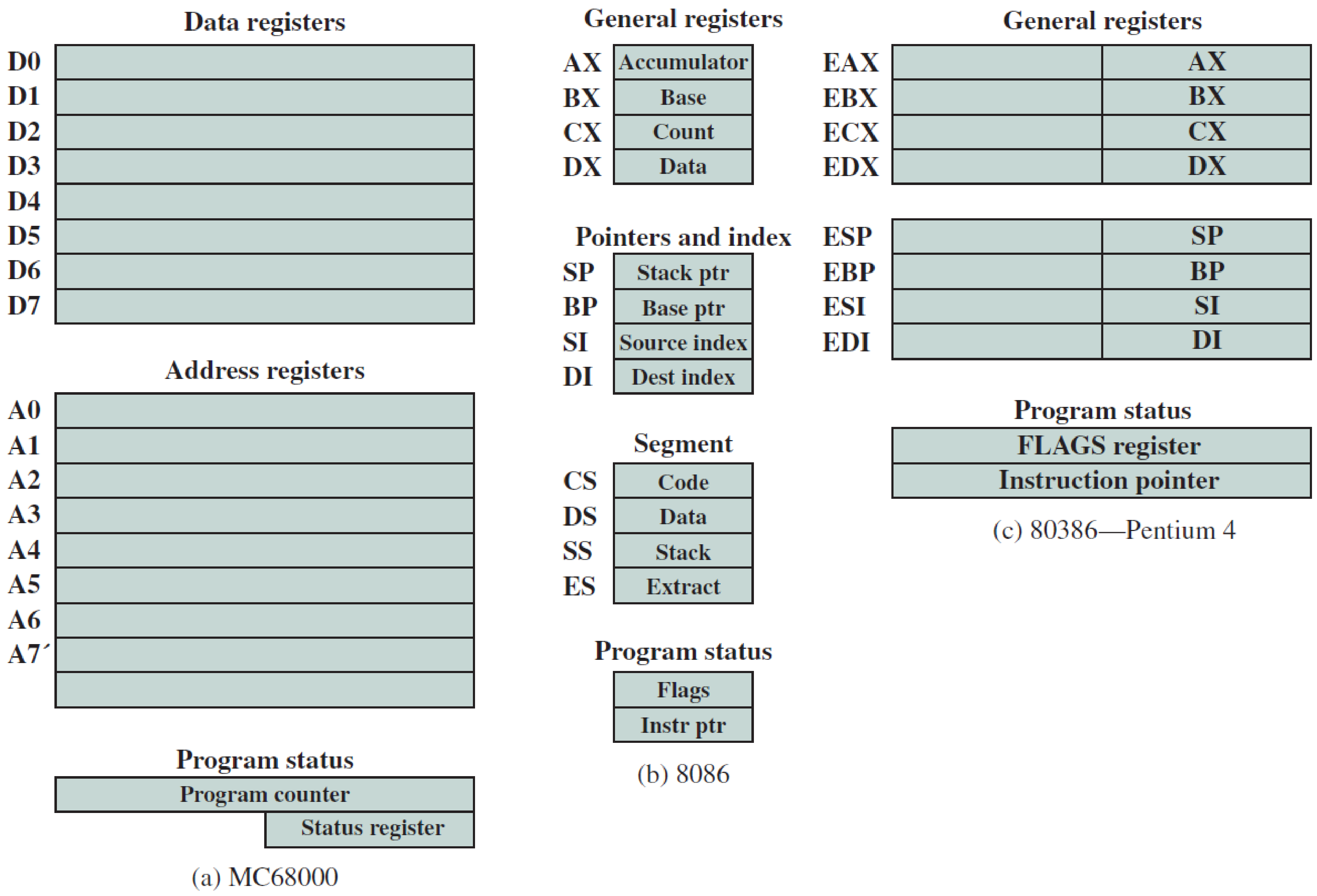


Figure 14.3 Example Microprocessor Register Organizations

Instruction Cycle

Includes the following stages:

- Fetch
  - Read the next instruction from memory into the processor
- Execute
  - Interpret the opcode and perform the indicated operation
- Interrupt
  - If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt

+

Instruction Cycle

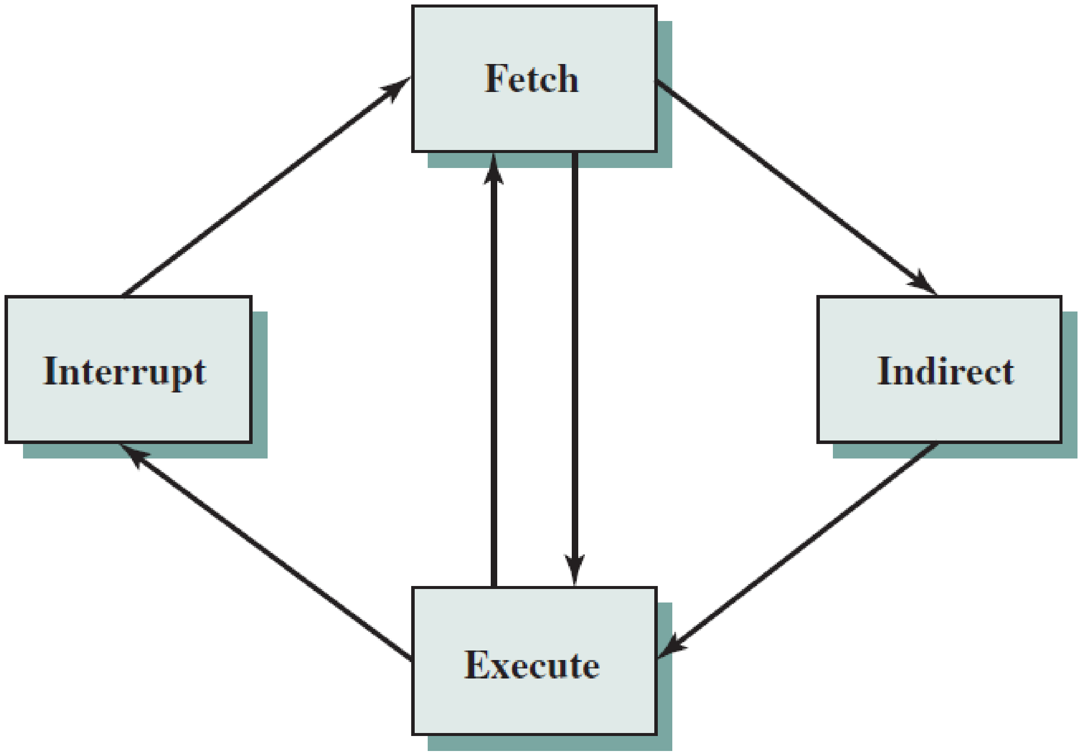


Figure 14.4 The Instruction Cycle

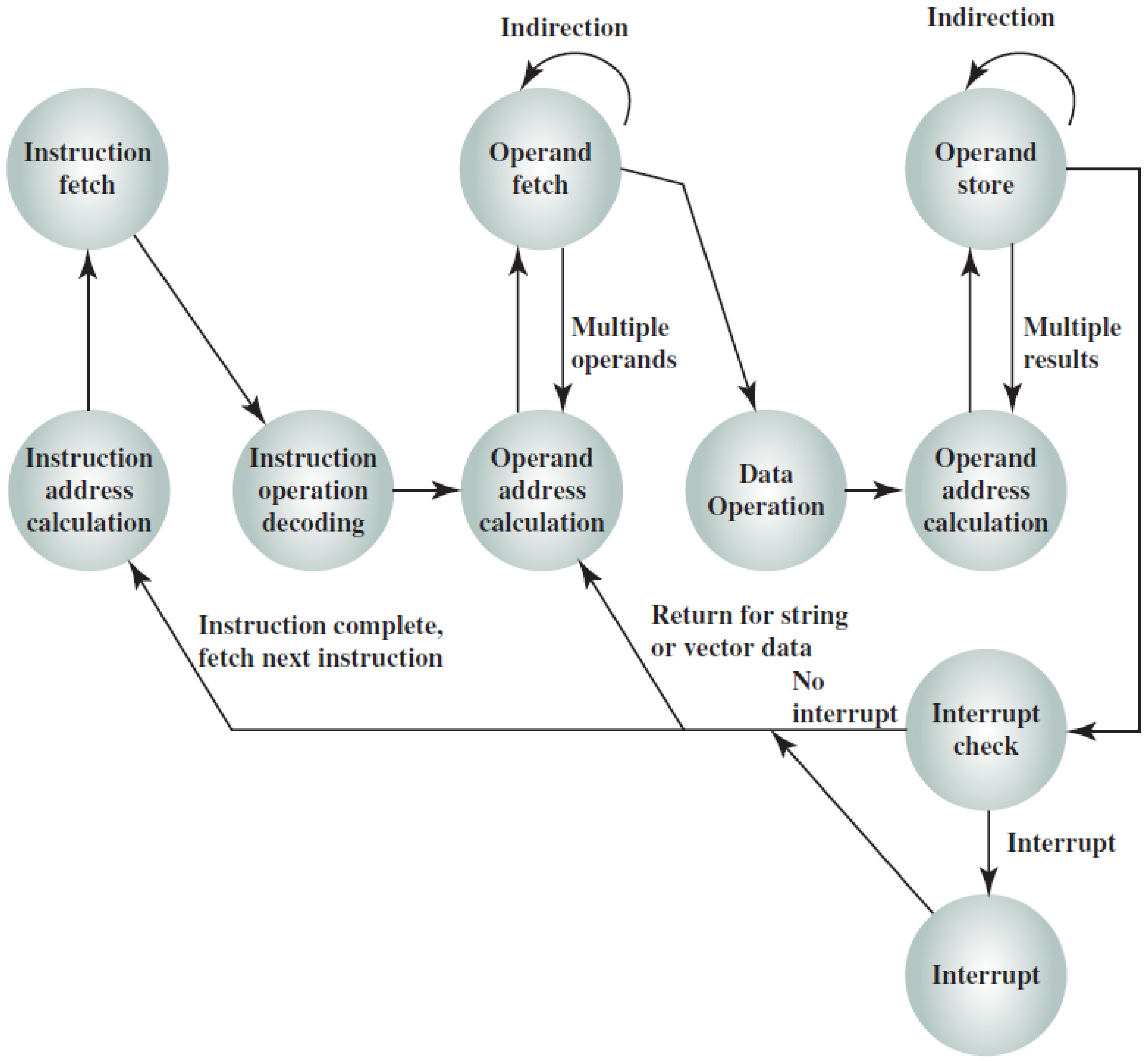
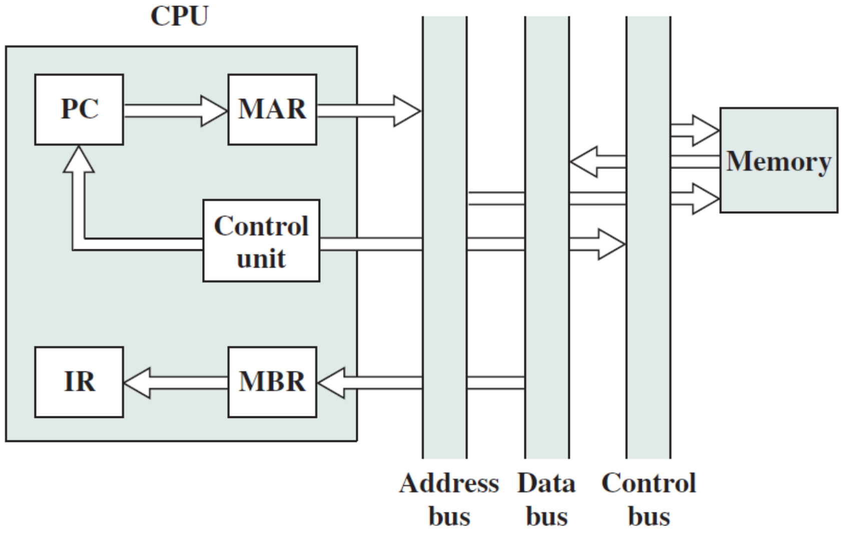

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt
Interrupt check
Interrupt
Interrupt

Figure 14.5 Instruction Cycle State Diagram

+

Fetch Cycle

- During the fetch cycle, an instruction is read from memory.
- The PC contains the address of the next instruction to be fetched.
- This address is moved to the MAR and placed on the address bus.
- The control unit requests a memory read, and the result is placed on the data bus and copied into the MBR and then moved to the IR.
- Meanwhile, the PC is incremented by 1, preparatory for the next fetch.



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

Figure 14.6 Data Flow, Fetch Cycle

+
Indirect Cycle

- Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing.
- If so, an indirect cycle is performed.
- The right- most N bits of the MBR, which contain the address reference, are transferred to the MAR.
- Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

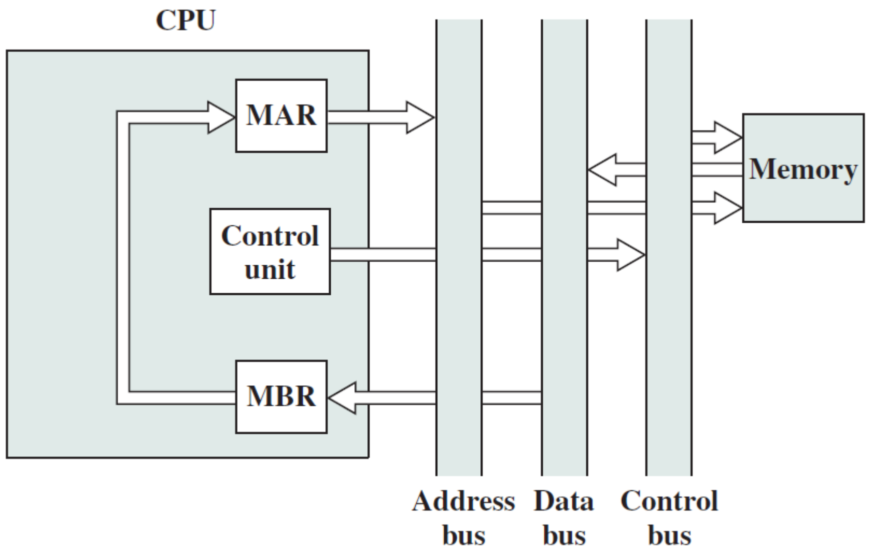


Figure 14.7 Data Flow, Indirect Cycle

+
Interrupt Cycle

- The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt.
- Thus, the contents of the PC are transferred to the MBR to be written into memory.
- The special memory location reserved for this purpose is loaded into the MAR from the control unit.
- It might, for example, be a stack pointer.
- The PC is loaded with the address of the interrupt routine.
- As a result, the next instruction cycle will begin by fetching the appropriate instruction.

+
Interrupt Cycle

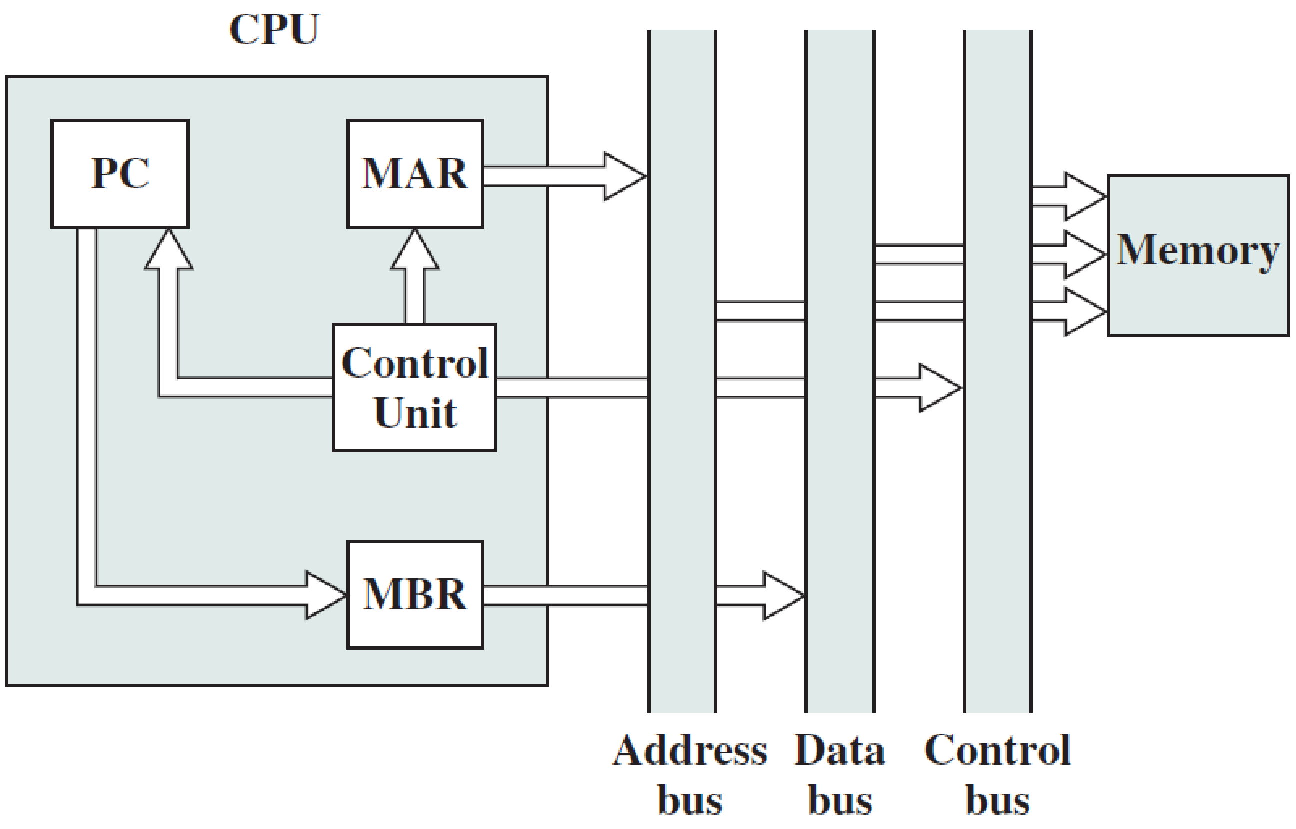


Figure 14.8 Data Flow, Interrupt Cycle

Pipelining Strategy

Similar to the use of an assembly line in a manufacturing plant

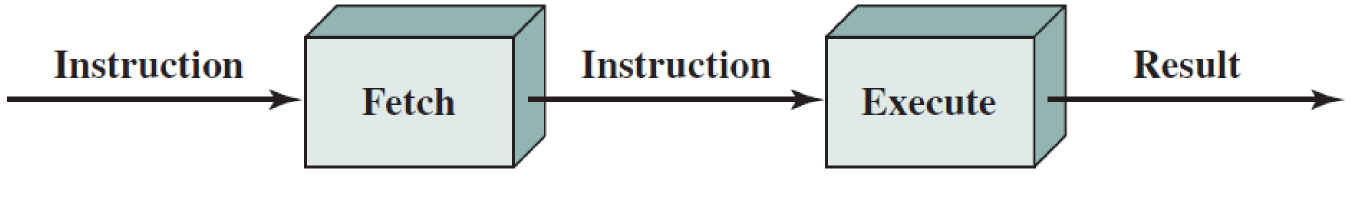
New inputs are accepted at one end before previously accepted inputs appear as outputs at the other end

To apply this concept to instruction execution we must recognize that an instruction has a number of stages

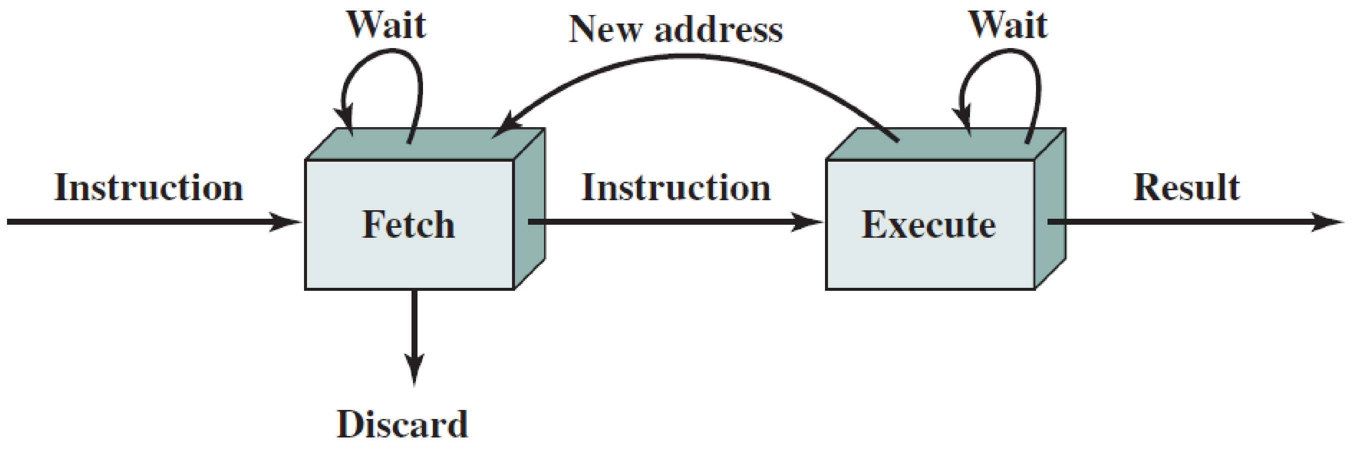
Pipelining

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution
- Today, pipelining is the key implementation technique used to make fast CPUs
- A pipeline is like an assembly line
  - Each step operates in parallel with the other steps
- In a computer pipeline, each step in the pipeline completes a part of an instruction

Instruction Pipeline



(a) Simplified view



(b) Expanded view

Figure 14.9 Two-Stage Instruction Pipeline

+ Additional Stages

- Fetch instruction (FI)
  - Read the next expected instruction into a buffer
- Decode instruction (DI)
  - Determine the opcode and the operand specifiers
- Calculate operands (CO)
  - Calculate the effective address of each source operand
  - This may involve displacement, register indirect, indirect, or other forms of address calculation
- Fetch operands (FO)
  - Fetch each operand from memory
  - Operands in registers need not be fetched
- Execute instruction (EI)
  - Perform the indicated operation and store the result, if any, in the specified destination operand location
- Write operand (WO)
  - Store the result in memory

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure 14.10 Timing Diagram for Instruction Pipeline Operation

Time $\longrightarrow$

Branch penalty $\longleftrightarrow$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

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

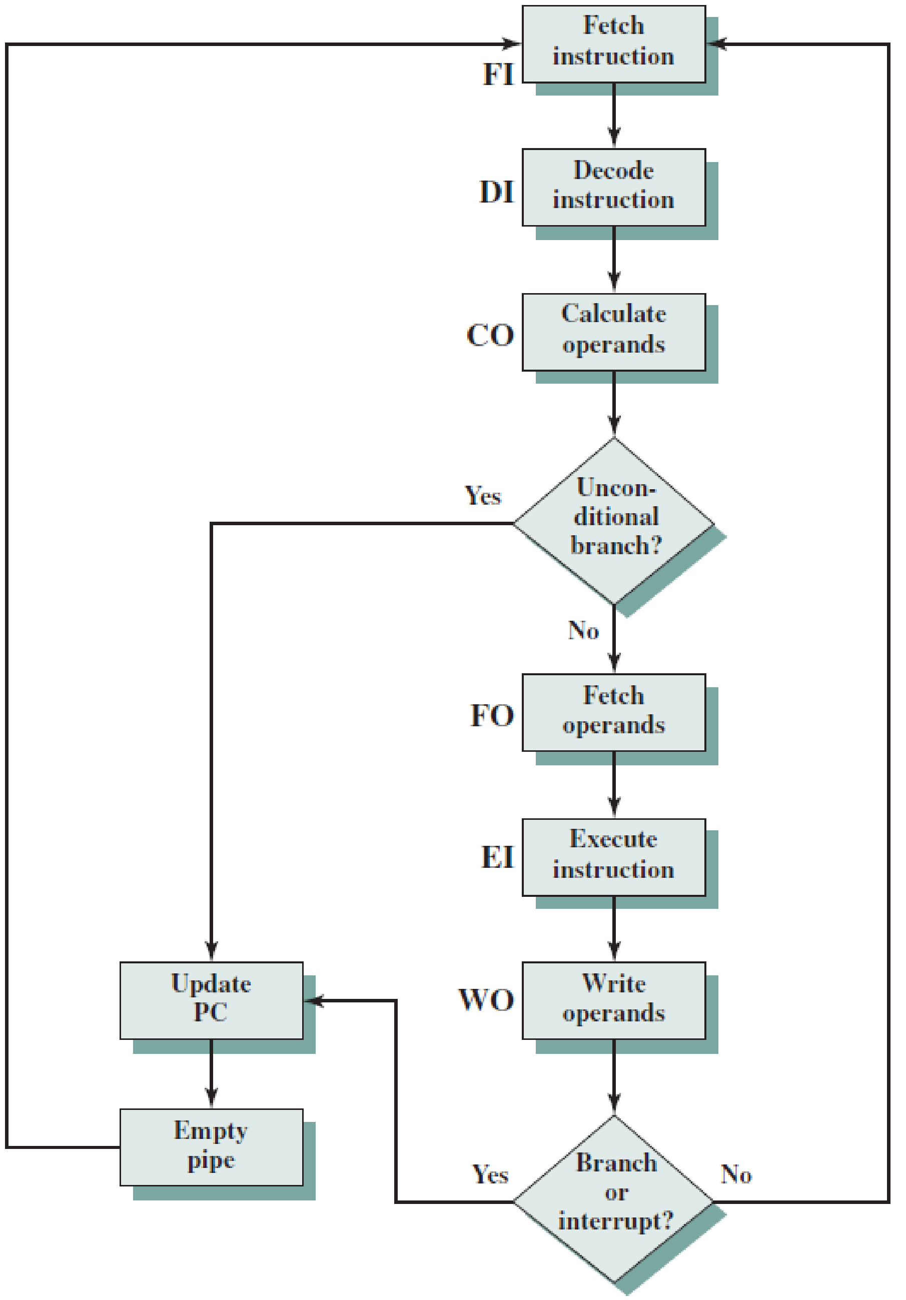


Figure 14.12 Six-Stage CPU Instruction Pipeline

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

Pipeline Hazards

Occur when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution

Also referred to as a *pipeline bubble*

There are three types of hazards:

- Resource
- Data
- Control

Resource Hazards

- A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource.
- The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline.
- A resource hazard is sometime referred to as a structural hazard.

Clock cycle

Instrutcion	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	

(a) Five-stage pipeline, ideal case

Clock cycle

Instrutcion	1	2	3	4	5	6	7	8	9
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

Figure 14.15 Example of Resource Hazard

+
Resource Hazards

- Assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time.
- An operand read to or write from memory cannot be performed in parallel with an instruction fetch.
- This is illustrated in Figure 14.15b, which assumes that the source operand for instruction I1 is in memory, rather than a register.
- Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3.
- The figure assumes that all other operands are in registers.

Resource Hazards

- Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU.
- One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.

+

Data Hazards

- A data hazard occurs when there is a conflict in the access of an operand location.
- In general terms, we can state the hazard in this form:
  - Two instructions in a program are to be executed in sequence and both access a particular memory or register operand.
  - If the two instructions are executed in strict sequence, no problem occurs.
  - However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution.
  - In other words, the program produces an incorrect result because of the use of pipelining.

	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure 14.16 Example of Data Hazard

+

Types of Data Hazard

- Read after write (RAW), or true dependency
  - An instruction modifies a register or memory location
  - Succeeding instruction reads data in memory or register location
  - Hazard occurs if the read takes place before write operation is complete
- Write after read (WAR), or antidependency
  - An instruction reads a register or memory location
  - Succeeding instruction writes to the location
  - Hazard occurs if the write operation completes before the read operation takes place
- Write after write (WAW), or output dependency
  - Two instructions both write to the same location
  - Hazard occurs if the write operations take place in the reverse order of the intended sequence

+

Control Hazard

- Also known as a *branch hazard*
- Occurs when the pipeline makes the wrong decision on a branch prediction
- Brings instructions into the pipeline that must subsequently be discarded
- Dealing with Branches:
  - Multiple streams
  - Prefetch branch target
  - Loop buffer
  - Branch prediction
  - Delayed branch

Multiple Streams

A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice

A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams

Drawbacks:

- With multiple pipelines there are contention delays for access to the registers and to memory
- Additional branch instructions may enter the pipeline before the original branch decision is resolved

Prefetch Branch Target

- When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch
- Target is then saved until the branch instruction is executed
- If the branch is taken, the target has already been prefetched
- IBM 360/91 uses this approach

+
Loop Buffer

- Small, very-high speed memory maintained by the instruction fetch stage of the pipeline and containing the *n* most recently fetched instructions, in sequence
- Benefits:
  - Instructions fetched in sequence will be available without the usual memory access time
  - If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer
  - This strategy is particularly well suited to dealing with loops
- Similar in principle to a cache dedicated to instructions
  - Differences:
    - The loop buffer only retains instructions in sequence
    - Is much smaller in size and hence lower in cost

+

Branch Prediction

- Various techniques can be used to predict whether a branch will be taken:

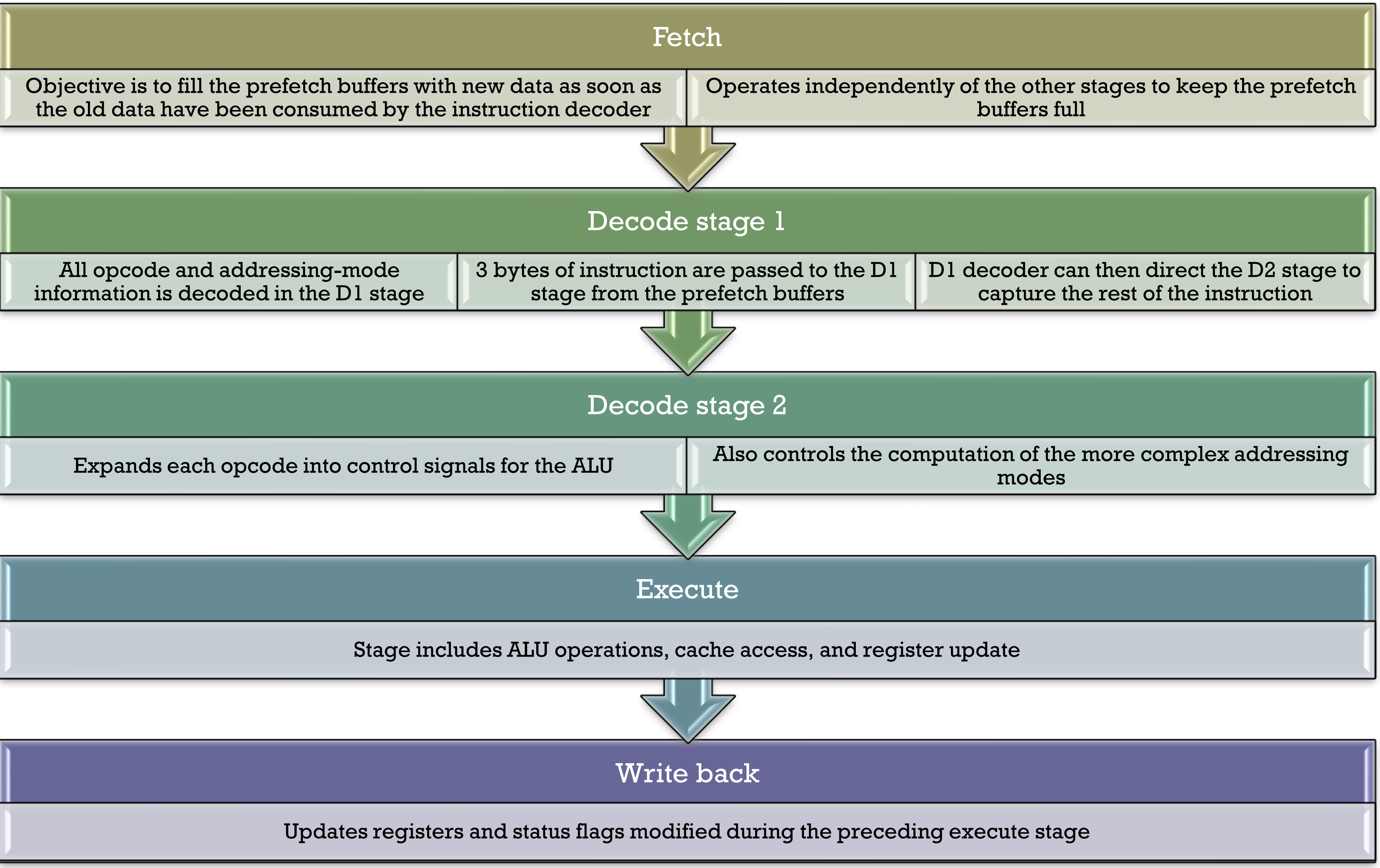
1. Predict never taken
2. Predict always taken
3. Predict by opcode

- These approaches are static
- They do not depend on the execution history up to the time of the conditional branch instruction

1. Taken/not taken switch
2. Branch history table

- These approaches are dynamic
- They depend on the execution history

Intel 80486 Pipelining



Fetch

Objective is to fill the prefetch buffers with new data as soon as the old data have been consumed by the instruction decoder

Operates independently of the other stages to keep the prefetch buffers full

Decode stage 1

All opcode and addressing-mode information is decoded in the D1 stage

3 bytes of instruction are passed to the D1 stage from the prefetch buffers

D1 decoder can then direct the D2 stage to capture the rest of the instruction

Decode stage 2

Expands each opcode into control signals for the ALU

Also controls the computation of the more complex addressing modes

Execute

Stage includes ALU operations, cache access, and register update

Write back

Updates registers and status flags modified during the preceding execute stage

Table 14.2 x86 Processor Registers

(a) Integer Unit in 32-bit Mode

Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Integer Unit in 64-bit Mode

Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

(c) Floating-Point Unit

Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

+

Summary

Chapter 14

Processor Structure and Function

- Processor organization
- Register organization
  - User-visible registers
  - Control and status registers
- Instruction cycle
  - The indirect cycle
  - Data flow
- The x86 processor family
  - Register organization
  - Interrupt processing
- Instruction pipelining
  - Pipelining strategy
  - Pipeline performance
  - Pipeline hazards
  - Dealing with branches
  - Intel 80486 pipelining