Processor Structure and Function

Lecture Content

- Processor Organization
- Pipelining & Branch Prediction
- Intel Pipelining & Processor Structure

Processor Structure and Function

Learning Objectives

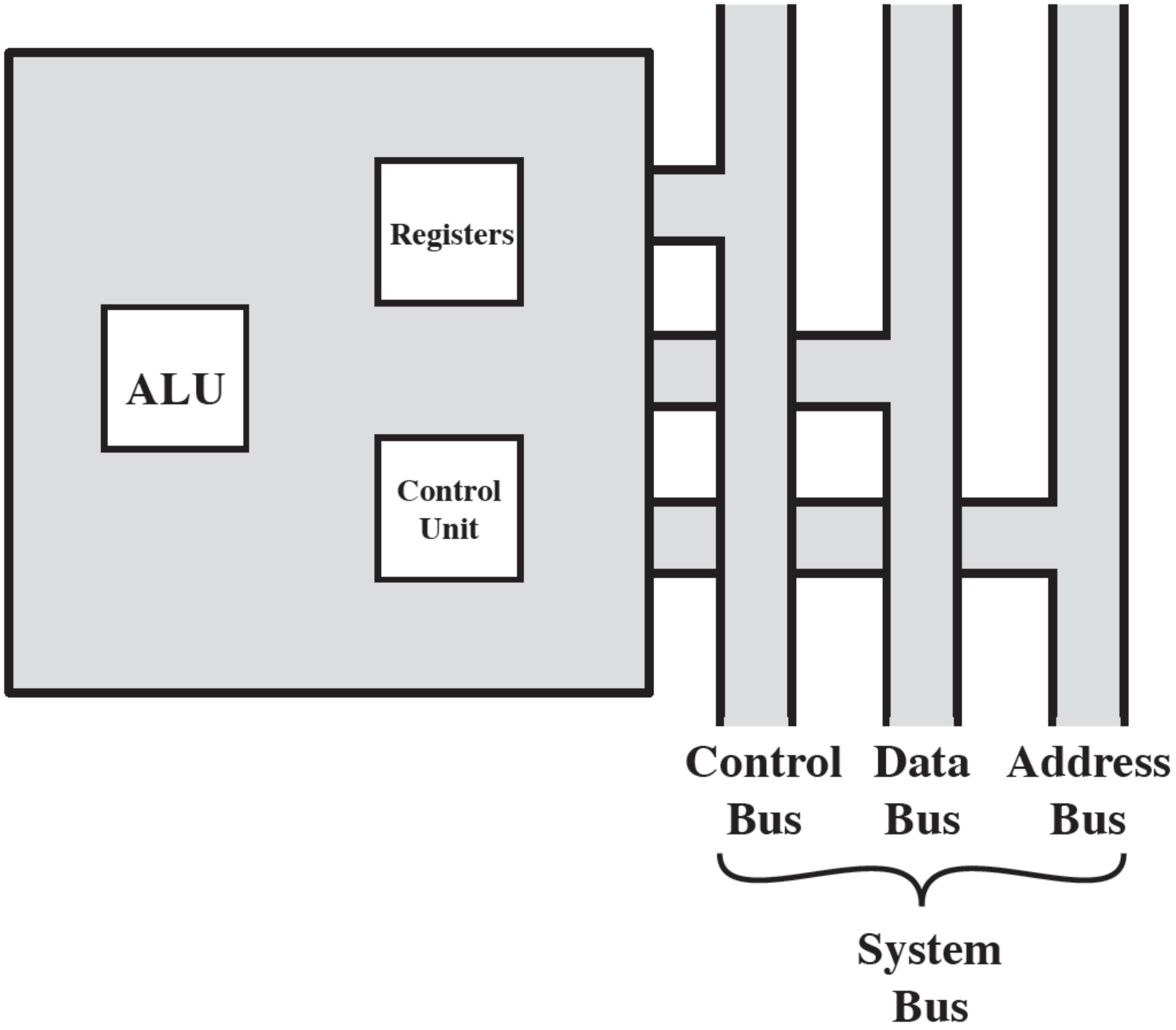
- Recap of the instruction cycle
- Distinguish between user-visible and control/status register
- Understand the principles of pipelining
- Understand pipeline hazards
- Understand branch prediction

Requirements for a Processor

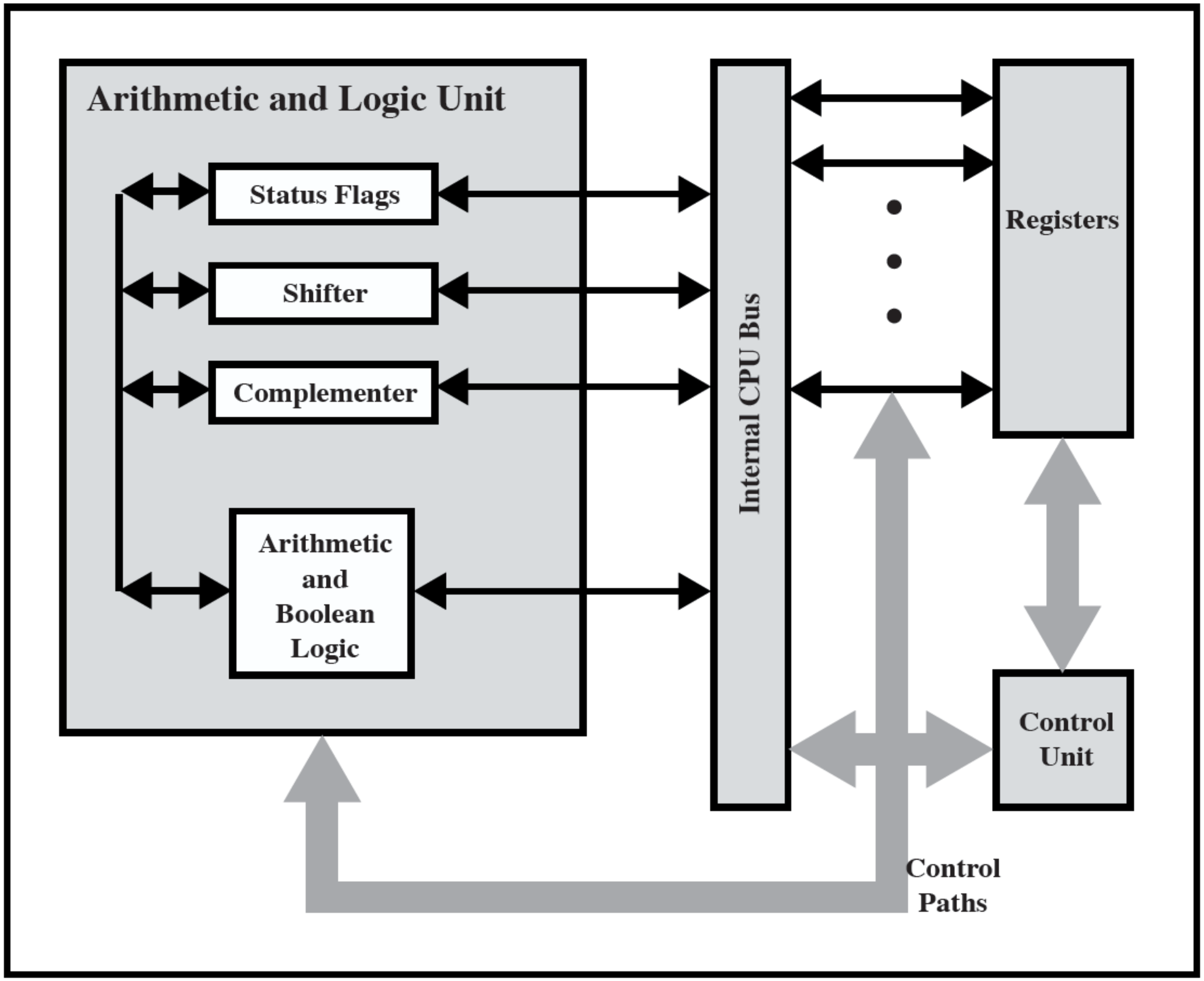
- **Fetch instruction**
  - The processor reads an instruction from register, cache, main memory
- **Interpret instruction**
  - The instruction is decoded to determine what action is required
- **Fetch data**
  - The instruction may require reading data from memory or an I/O module
- **Process data**
  - The instruction may be performing some arithmetic or logical operation
- **Write data**
  - The results 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

Schematic View of a Processor



Schematic Internal Structure



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**
  - 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

User-Visible Registers

- **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

Design Issues

- **General-Purpose vs. Specialized**
  - Affects instruction set design
  - Specialized registers are less complicated and slim the ISA
  - Specialization limits the programmer’s flexibility.
- **Number of Registers**
  - Somewhere between 8 and 32 registers appears optimum
  - Fewer registers result in more memory references
  - More registers do not noticeably reduce memory references
- **Register Length**
  - Address registers must be long enough to hold the largest address
  - Data registers should be able to hold values of most data types

Condition Codes

- Many processors do not use condition codes at all
- A conditional branch instructions specify a comparison to be made and act on the result of the comparison
- **Advantages**
- **Disadvantages**

Condition Codes

- Many processors do not use condition codes at all
- A conditional branch instructions specify a comparison to be made and act on the result of the comparison
- **Advantages**
  - Set by normal arithmetic and data movement instructions
  - Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH
  - Condition codes facilitate multi-way branches.
  - Condition codes can be saved on the stack during subroutine calls along with other register information
- **Disadvantages**

Condition Codes

- Many processors do not use condition codes at all
- A conditional branch instructions specify a comparison to be made and act on the result of the comparison
- **Advantages**
- **Disadvantages**
  - Condition codes add complexity
  - Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections
  - 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
  - In a pipelined implementation, condition codes require special synchronization to avoid conflicts

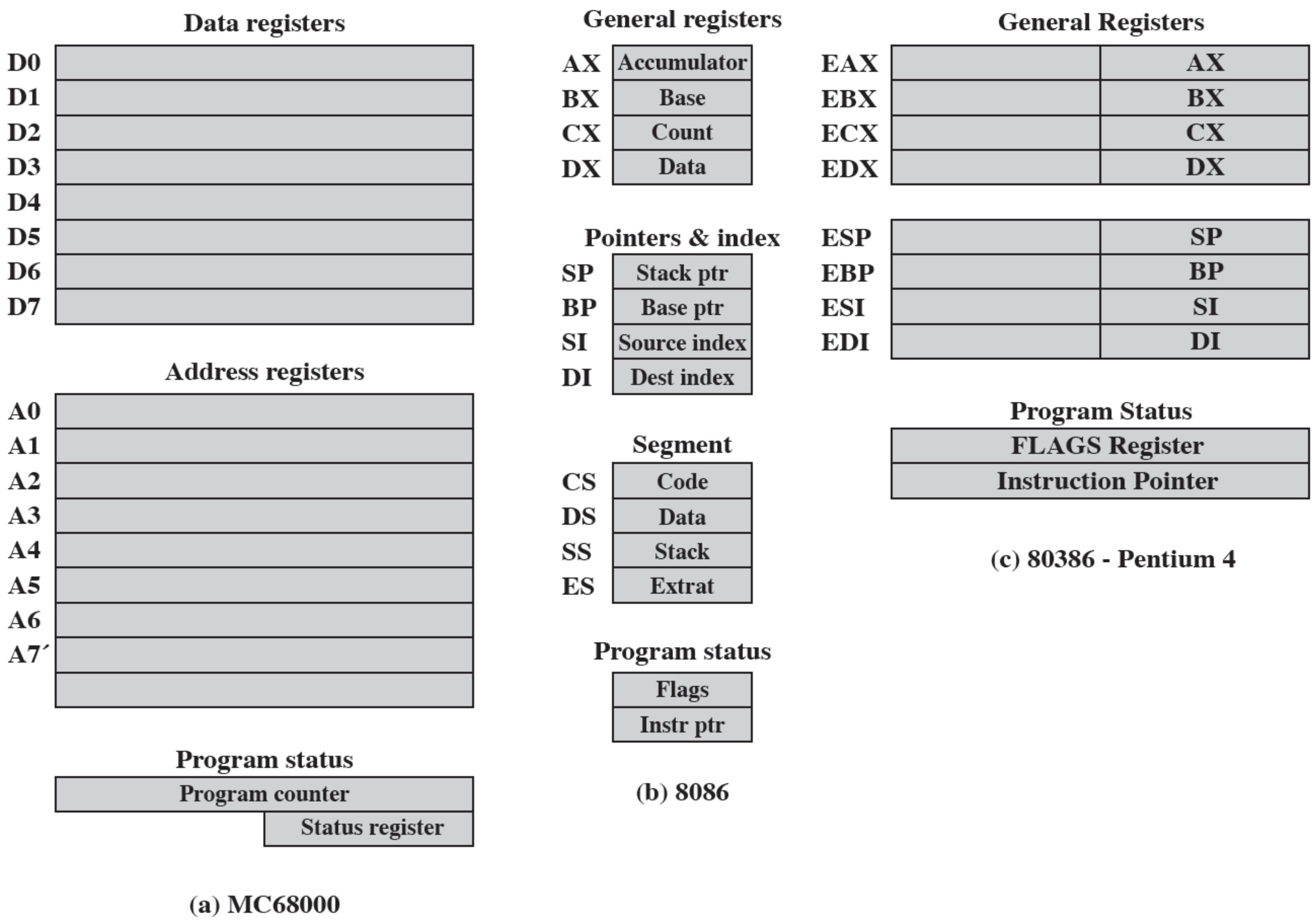
Control and Status Registers

- **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)

- **Sign**
  - Contains the sign bit of the result of the last arithmetic operation
- **Zero**
  - Set when the result is 0
- **Equal**
  - Set if a logical compare result is equality
- **Overflow**
  - Used to indicate arithmetic overflow
- **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
- **Interrupt Enable/Disable**
  - Used to enable or disable interrupts
- **Supervisor**
  - Indicates whether the processor is executing in supervisor or user mode

Register Organization Examples



Processor Structure and Function

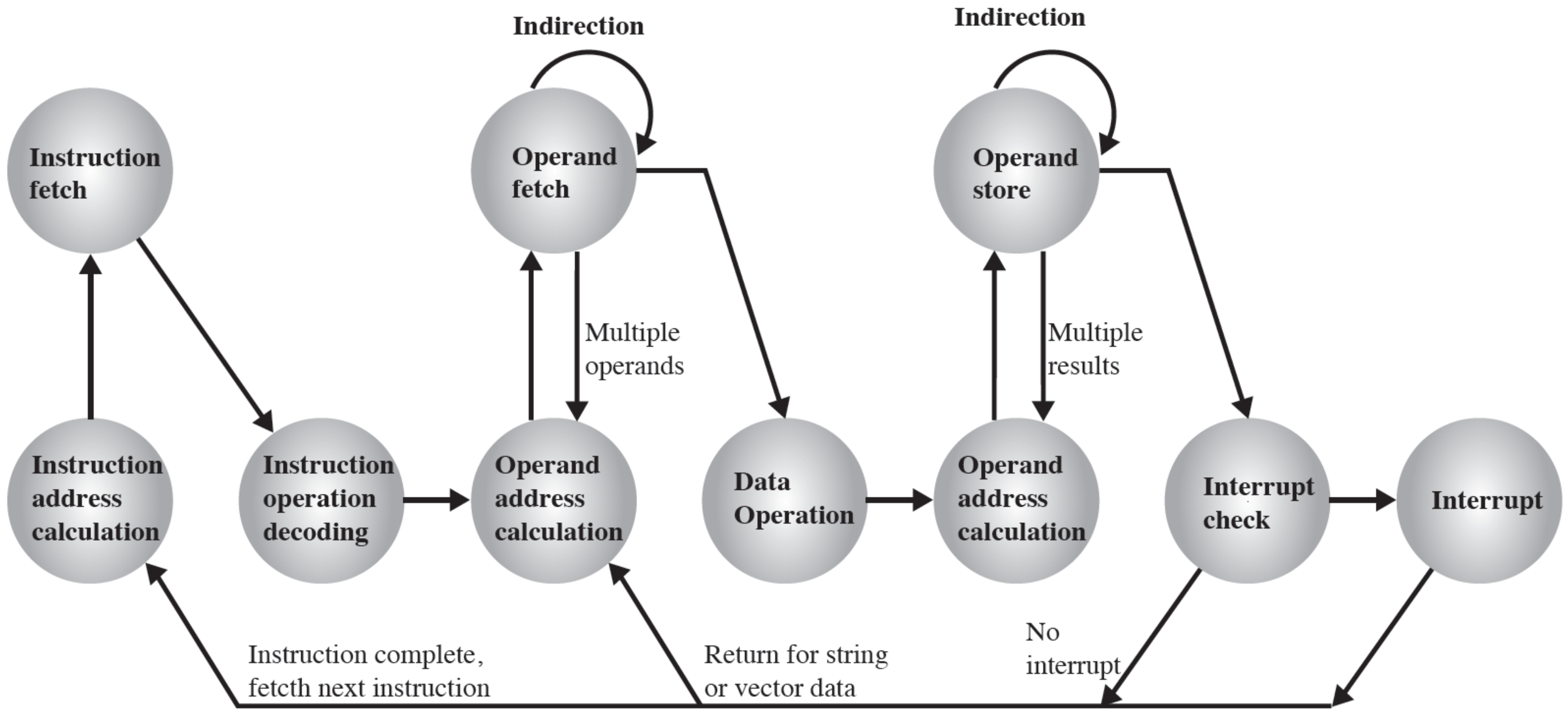
Lecture Content

- Processor Organization
- Pipelining & Branch Prediction
- Intel Pipelining & Processor Structure

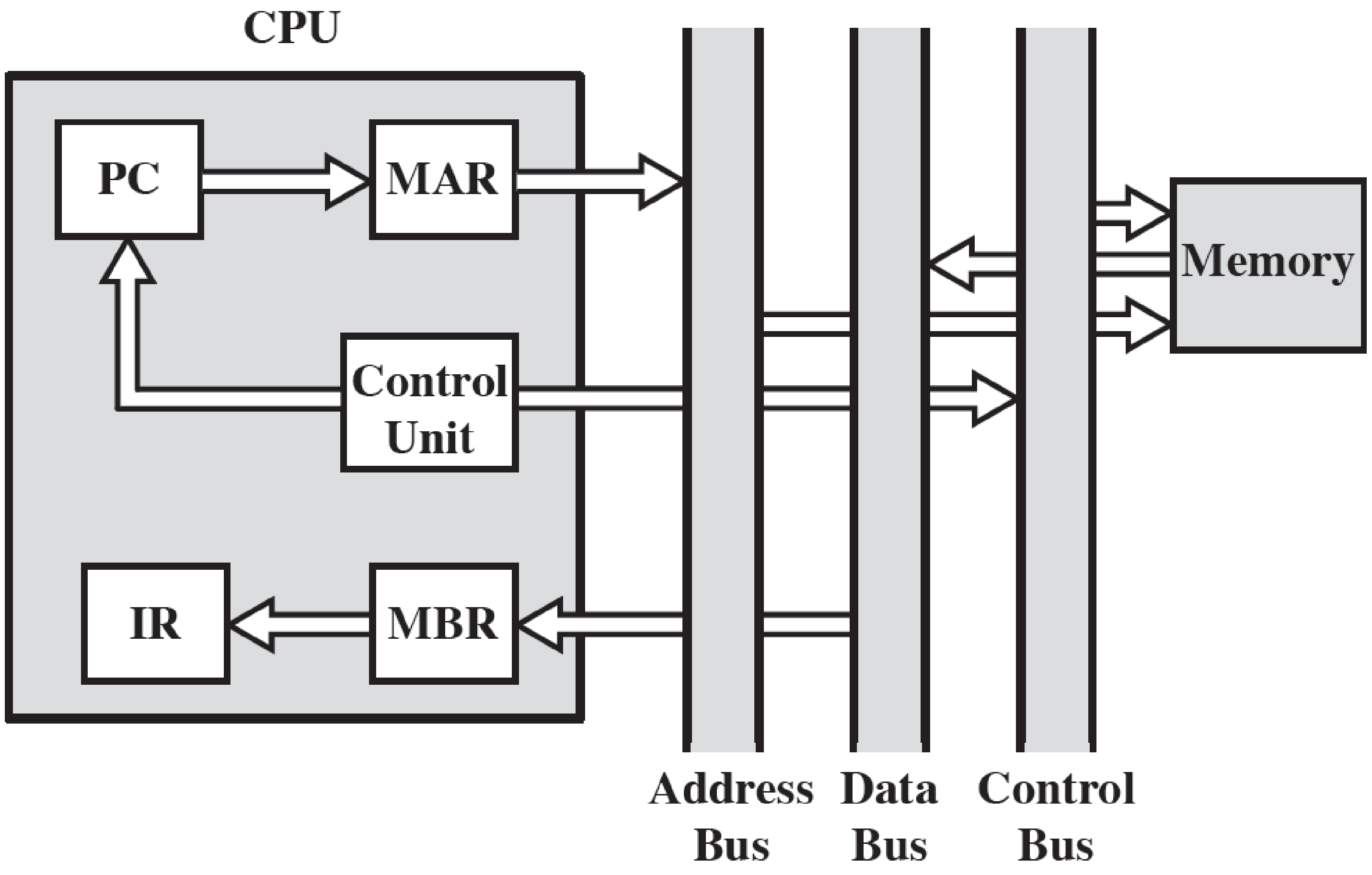
Recall: The Instruction Cycle

- **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

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
Interrupt check
Interrupt
No interrupt
Instruction complete, fetcth next instruction
Return for string or vector data

Example Data Flow: Fetch Cycle

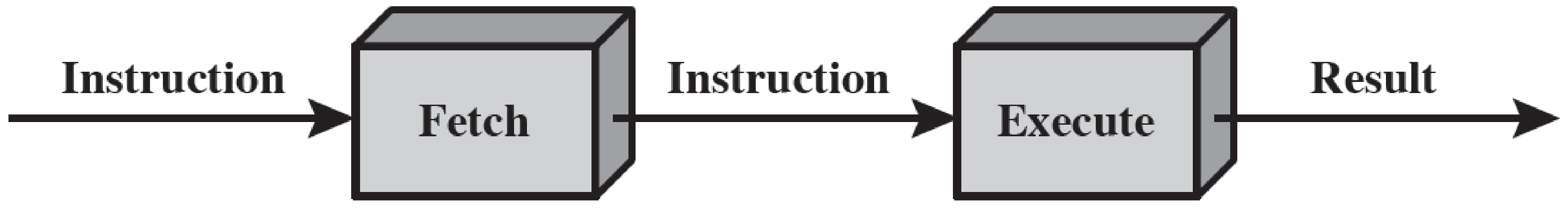


MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

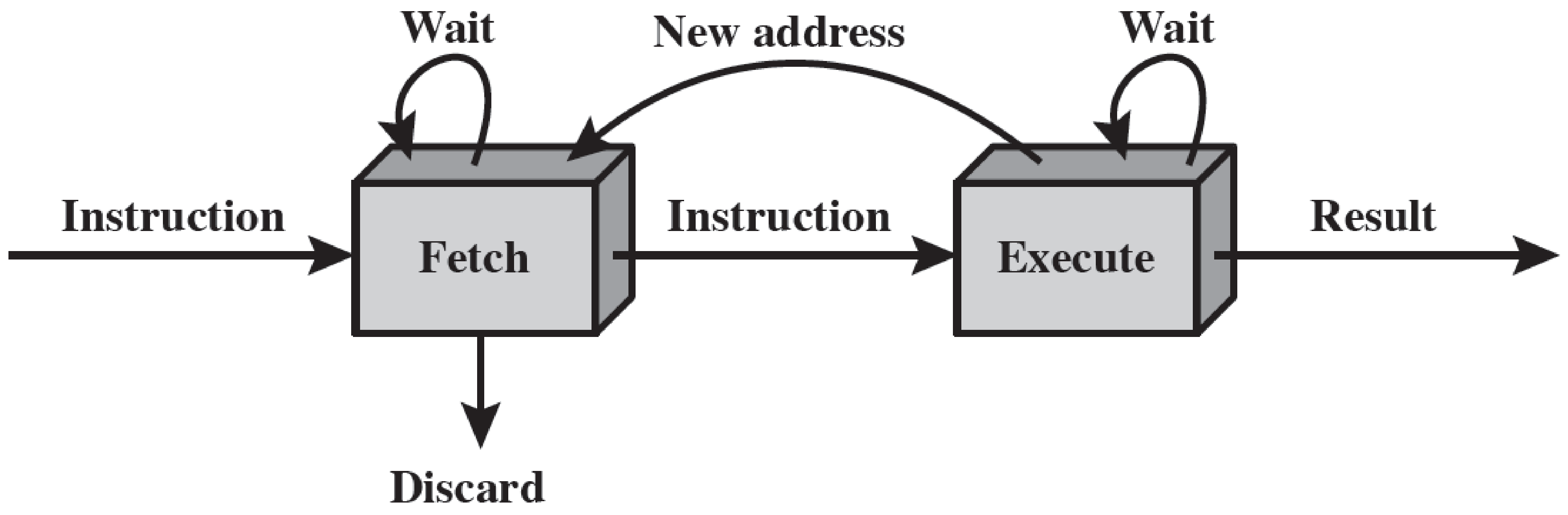
Pipelining: General Idea

- Each step of the instruction cycle only requires parts of the CPU's infrastructure
- 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

A Simple Two Stage Pipeline



(a) Simplified view



More Stages of a Pipeline

- **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

Principle of a Pipeline

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

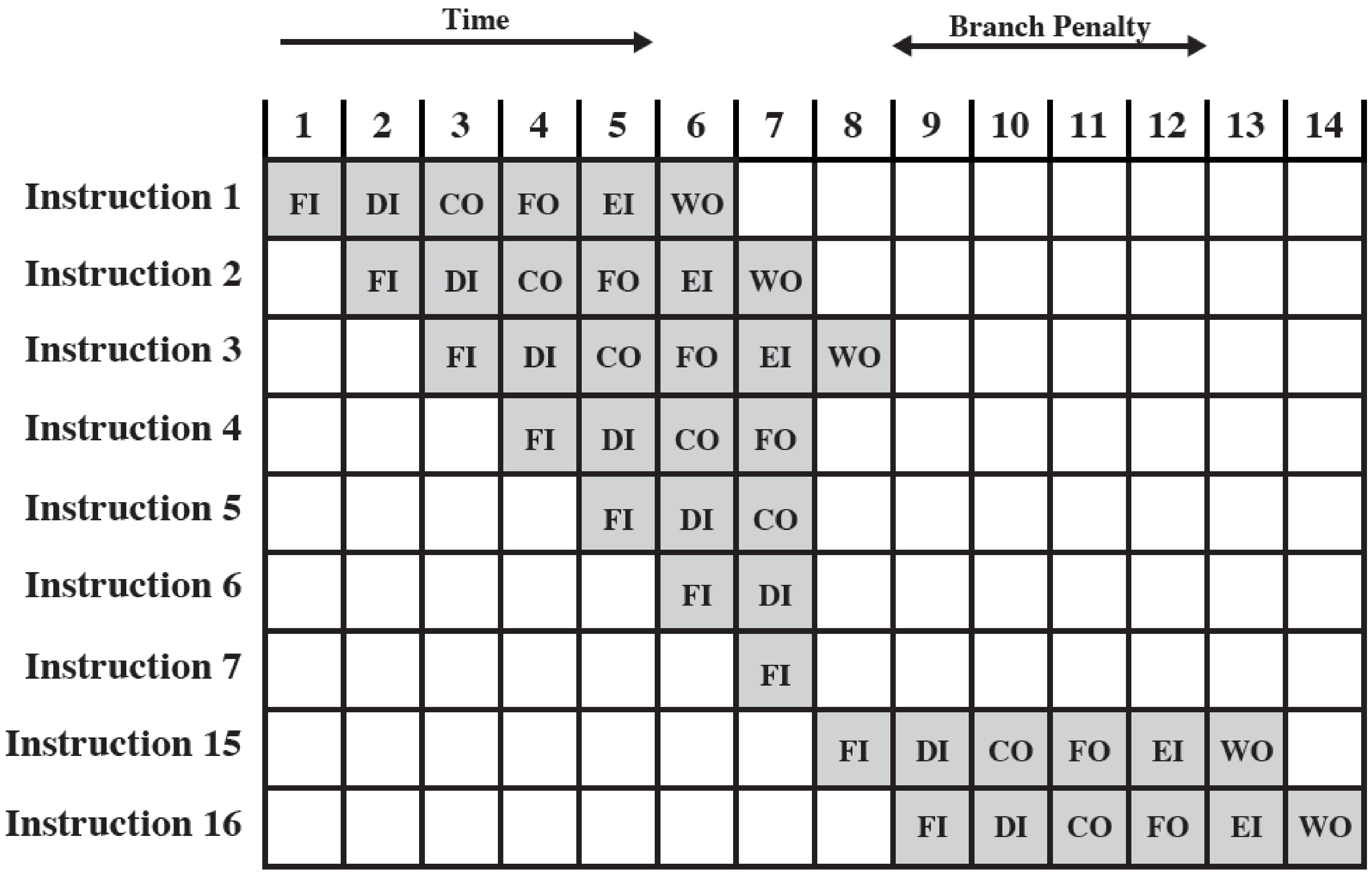
Speed-Up of a Pipeline

- Assuming we have a pipeline with k stages
- Thus, each instruction requires k cycles to finish
- Once, the pipeline is full, we finish one instruction every cycle

- Does this mean we have a speed-up of k?

- When does this principle not apply? What can happen?

Mess Things Up: A Conditional Branch


Time
Branch Penalty

	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

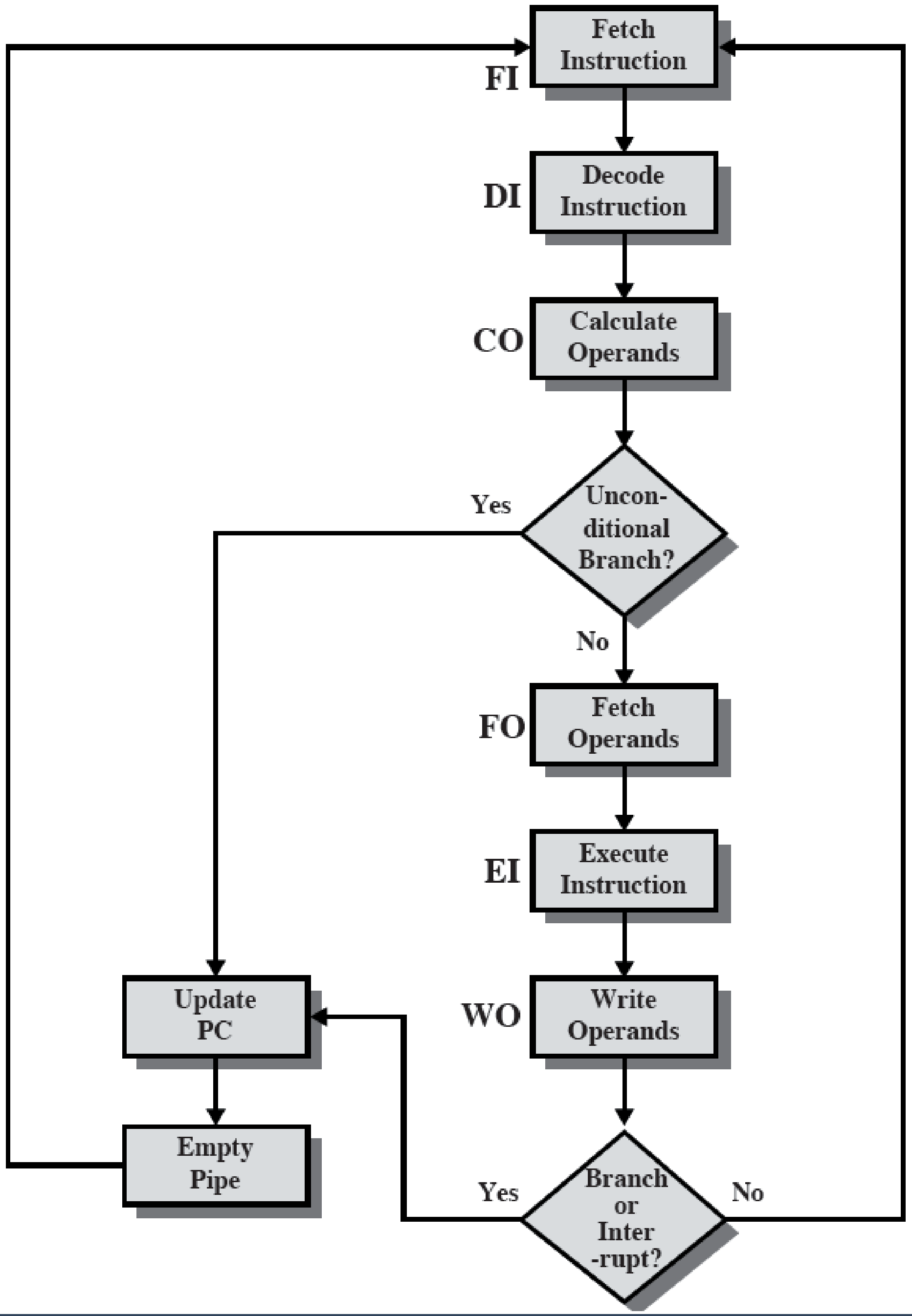
Alternate Display

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

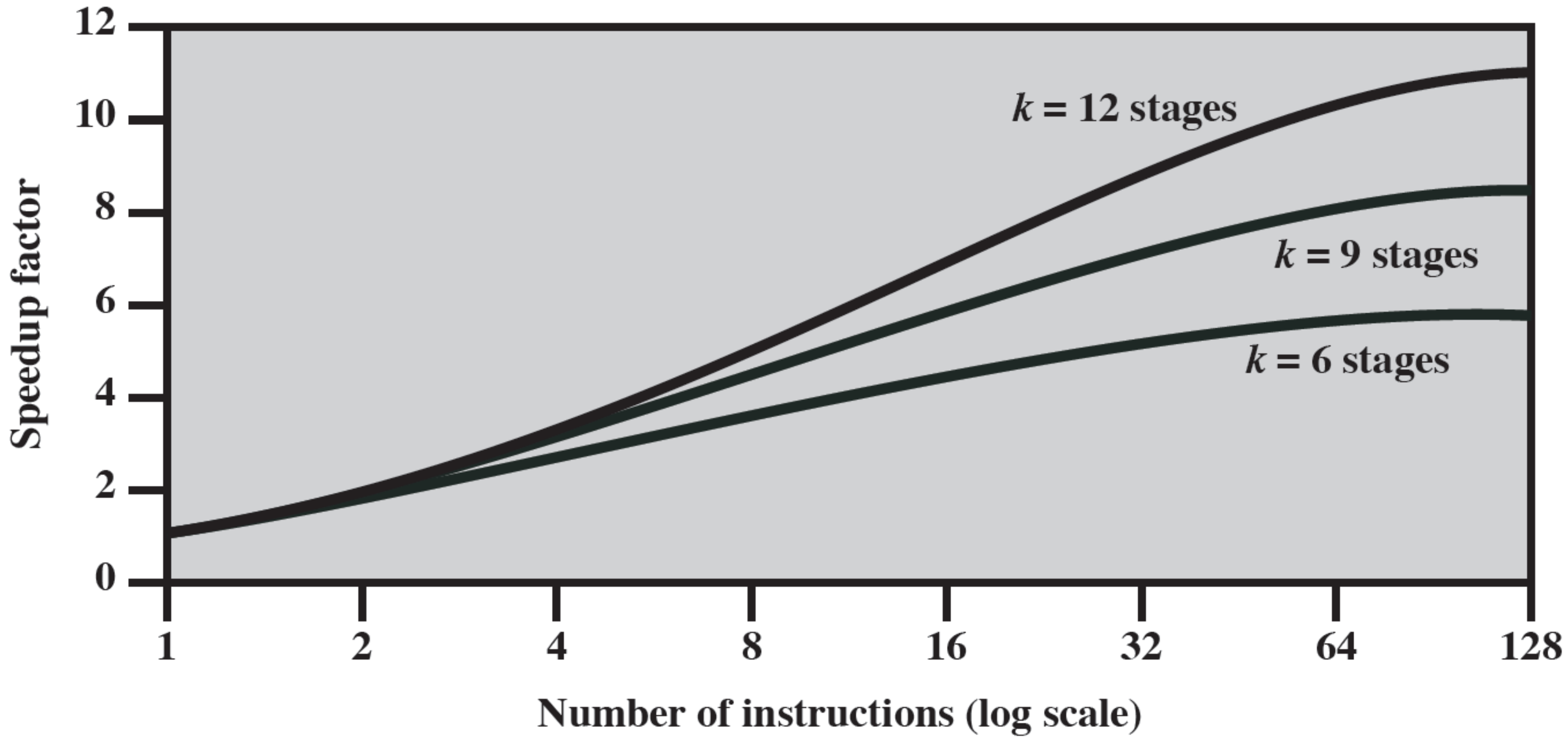
(a) No branches

	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

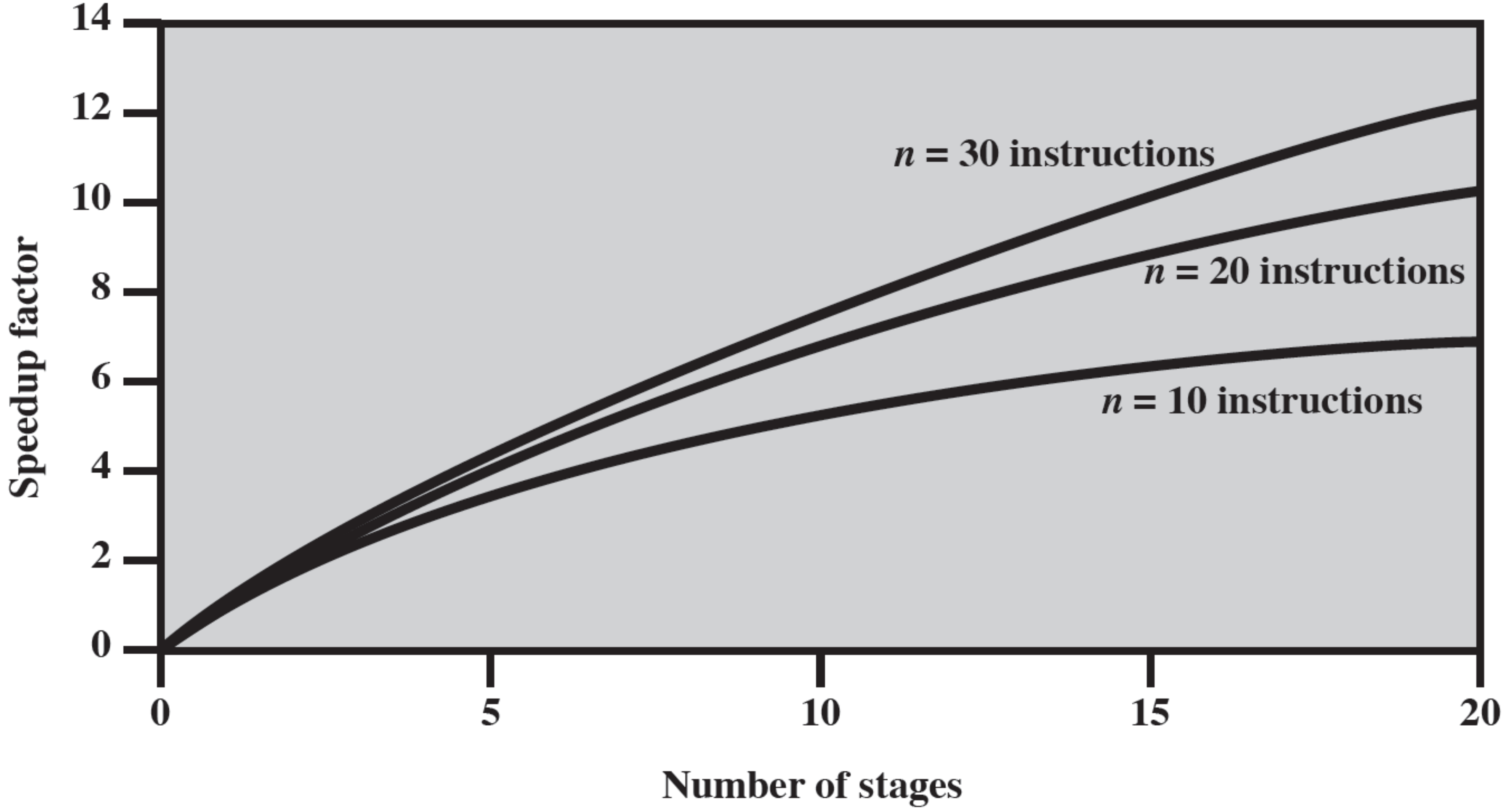

Fetch Instruction
FI
Decode Instruction
DI
Calculate Operands
CO
Uncon-ditional Branch?
Yes
No
Fetch Operands
FO
Execute Instruction
EI
Write Operands
WO
Update PC
Empty Pipe
Branch or Inter-rupt?
Yes
No

Speed-Up



Speedup factor
12
10
8
6
4
2
0
k = 12 stages
k = 9 stages
k = 6 stages
1
2
4
8
16
32
64
128
Number of instructions (log scale)

Speed-Up



14
12
10
8
6
4
2
0
Speedup factor
n = 30 instructions
n = 20 instructions
n = 10 instructions
0
5
10
15
20
Number of stages

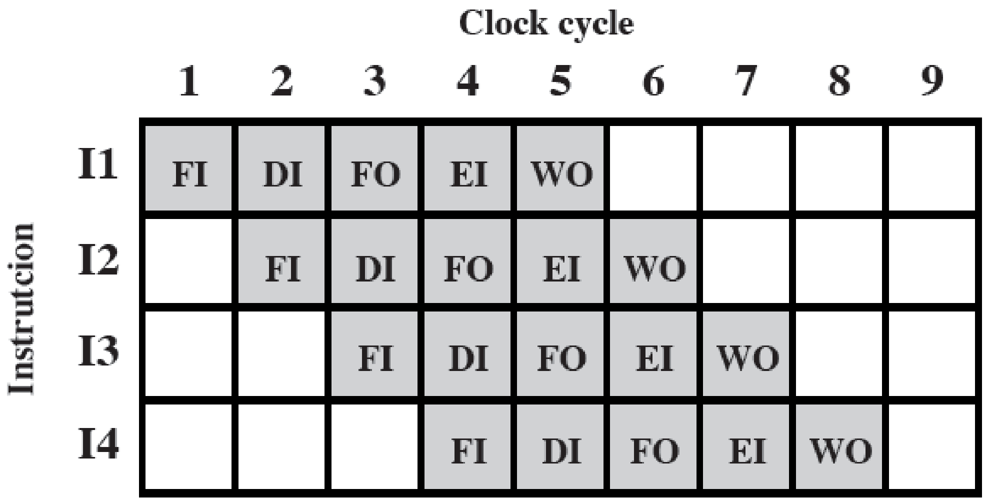
Pipeline Hazards

- 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 Hazard

- 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 sometimes referred to as a structural hazard

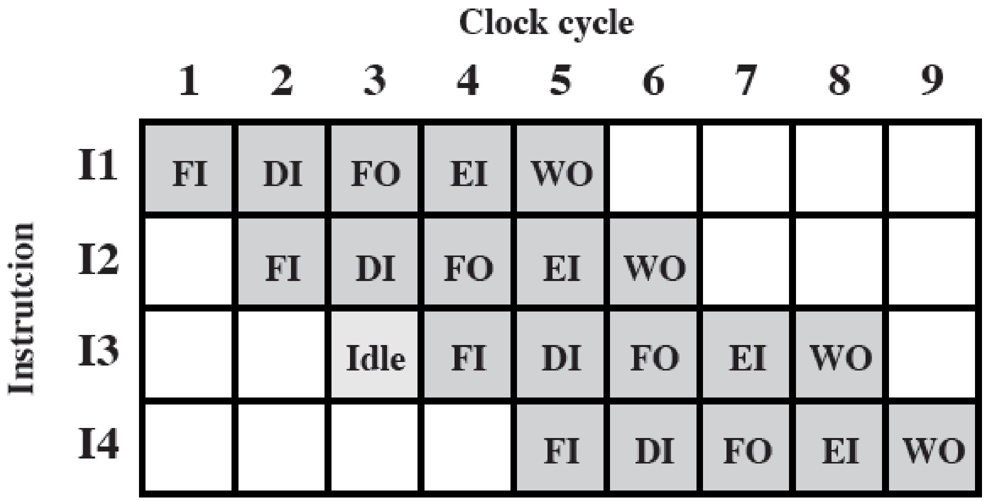
Resource Hazard



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

Data Hazards

- **Read After Write (RAW)**, or true dependency
- **Write After Read (WAR)**, or antidependency
- **Write After Write (WAW)**, or output dependency

Data Hazards

- **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
- **Write After Write (WAW)**, or output dependency

Data Hazards

- **Read After Write (RAW)**, or true dependency
- **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

Data Hazards

- **Read After Write (RAW)**, or true dependency
- **Write After Read (WAR)**, or antidependency
- **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

Example For a Data Hazard

Clock cycle

	1	2	3	4	5	6	7	8	9	10
`add %rbx,%rax`	FI	DI	FO	EI	WO					
`sub %rax,%rcx`		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

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:

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

- At each branch, a pipeline has to choose the next instruction to fetch
- A wrong decision is costly
- A brute-force approach:
  - Replicate the initial portions of the pipeline
  - Pipeline can fetch two 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

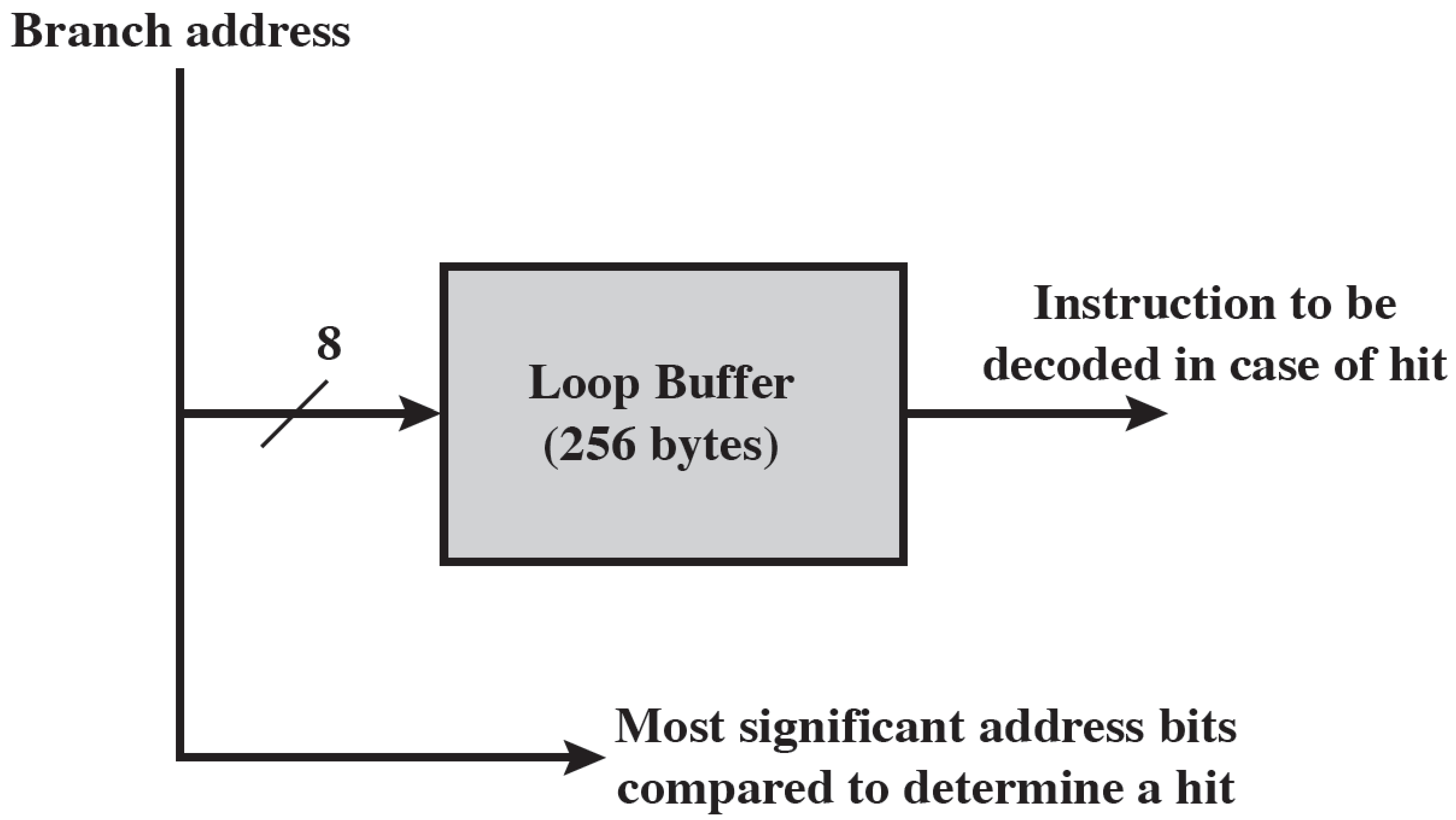
Prefetch Branch Target

- 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 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

Loop Buffer


Branch address
8
Loop Buffer
(256 bytes)
Instruction to be decoded in case of hit
Most significant address bits compared to determine a hit

Branch Prediction

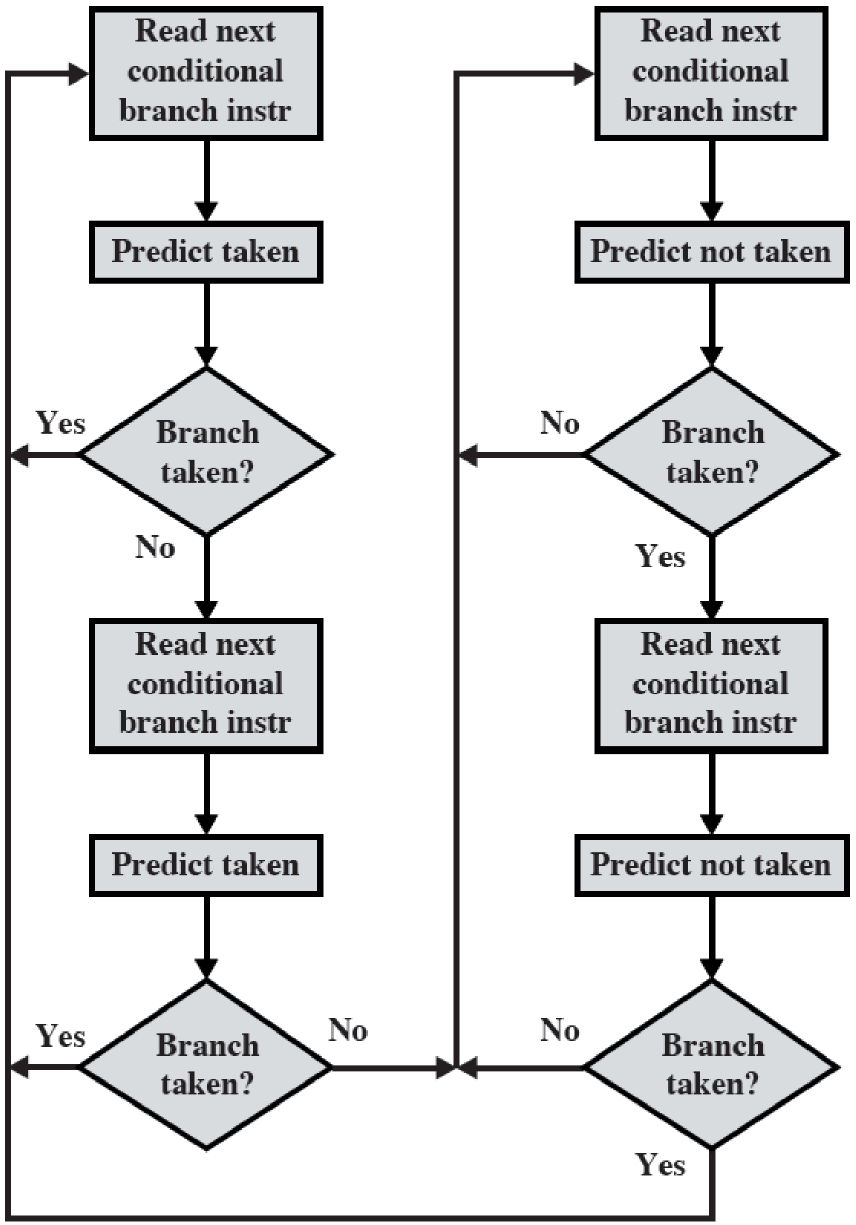
- **Static Approaches**
  - Predict never taken
  - Predict always taken
  - Predict by opcode
- **Dynamic Approaches**
  - Taken/not taken switch
  - Branch history table

Static Branch Prediction

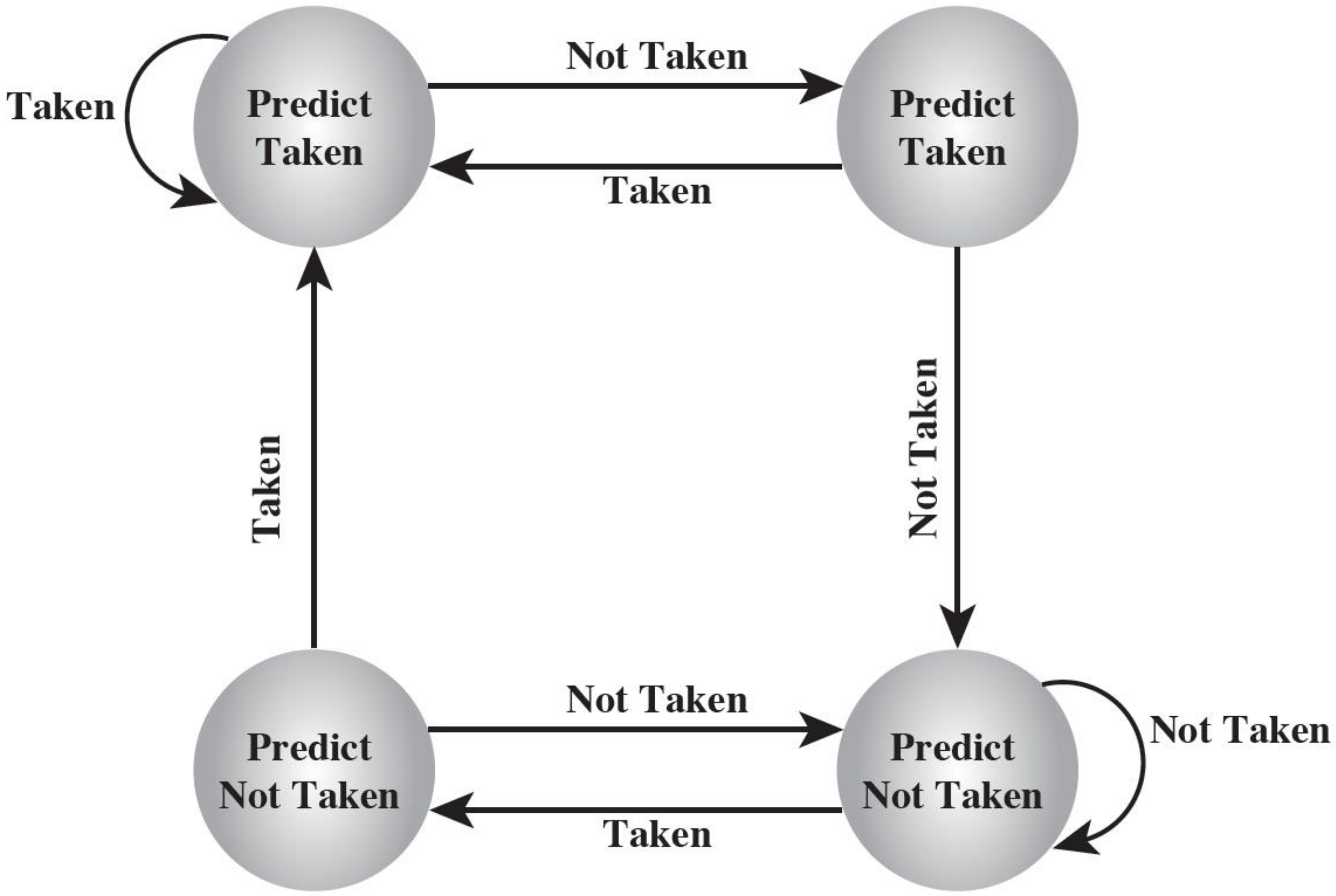
- **Predict Always/Never Taken**
  - Continue to fetch instructions in sequence or fetch from the branch target
  - The predict-never-taken approach is the most popular of all the branch prediction methods
  - Analyzing program behavior shows that conditional branches are taken more than 50% of the time
  - However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in sequence

- **Opcode Dependent**
  - The processor assumes that the branch will be taken for certain branch opcodes and not for others
  - Success rates of greater than 75% with this strategy.

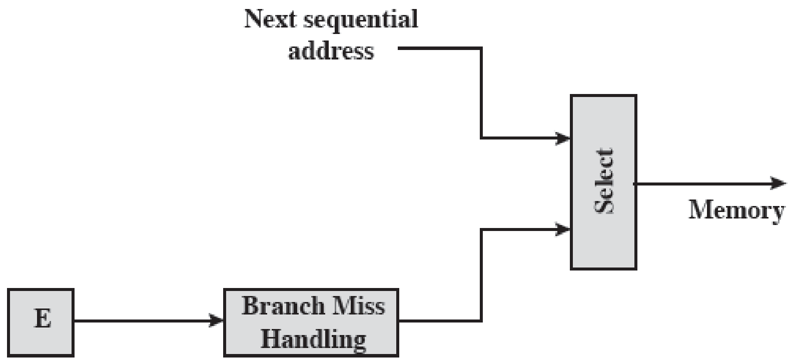
Branch Prediction Flow Chart


Read next conditional branch instr
Predict taken
Branch taken?
Yes
No
Read next conditional branch instr
Predict taken
Branch taken?
Yes
No
Read next conditional branch instr
Predict not taken
Branch taken?
No
Yes
Read next conditional branch instr
Predict not taken
Branch taken?
No
Yes

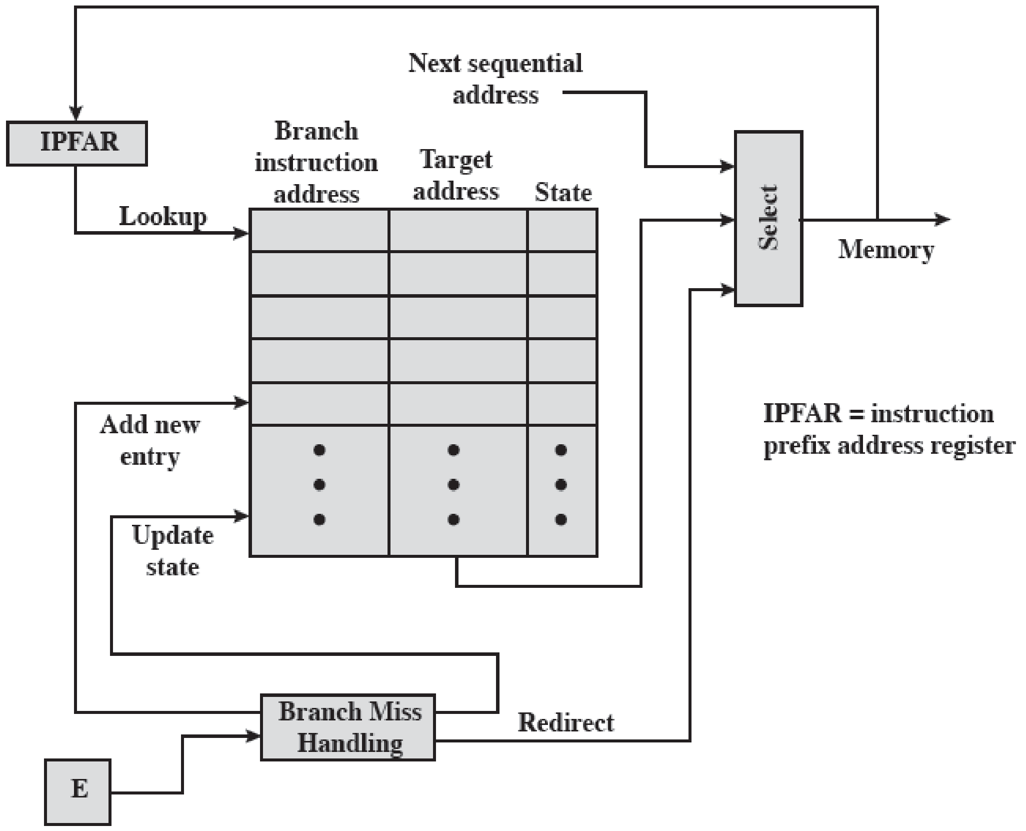
Branch Prediction State Diagram



Static vs. Dynamic Branch Prediction



(a) Predict never taken strategy



Processor Structure and Function

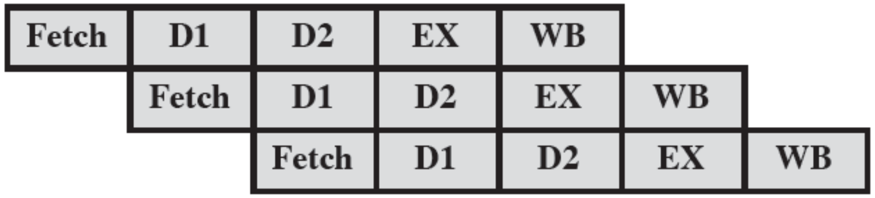
Lecture Content

- Processor Organization
- Pipelining & Branch Prediction
- Intel Pipelining & Processor Structure

Intel 80486 Pipeline

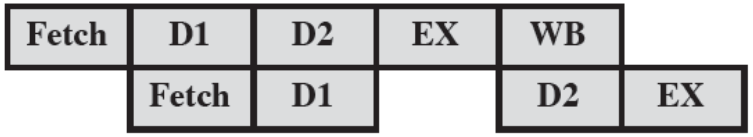
- **Fetch**
  - Operates independently of the other stages to keep the prefetch buffers full
- **Decode Stage 1**
  - All opcode and addressing-mode information is decoded in the D1 stage
  - 3 bytes of instruction are passed to the D1 stage from the prefetch buffers
- **Decode Stage 2**
  - Expands 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

80486 Pipeline Examples



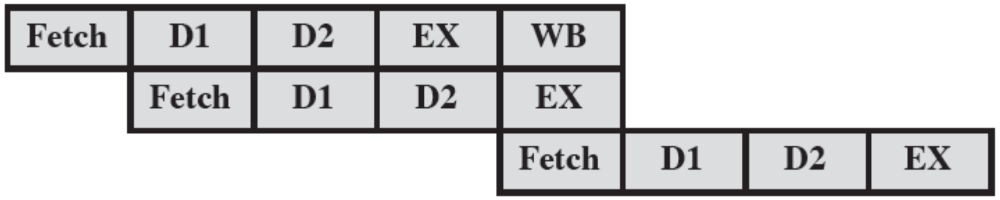
```
mov $mem1,%ax
mov %bx,%ax
mov %ax,$mem2
```

(a) No Data Load Delay in the Pipeline



```
mov $mem1,%ax
mov (%ax),%bx
```

(b) Pointer Load Delay



```
         cmp $Imm,%ax
         jcc $target
target:  something
```

(c) Branch Instruction Timing

Integer Registers

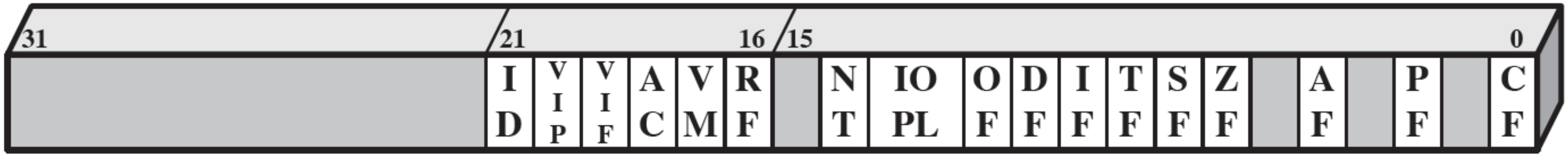
32 Bit			
Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

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

Floating Point & Additional Registers

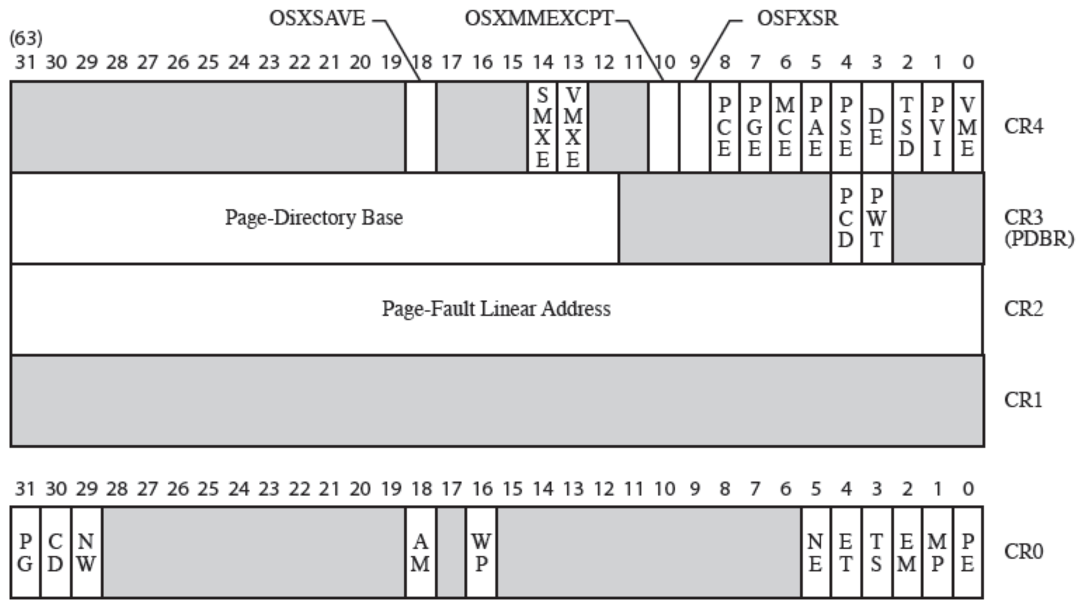
Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by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

EFLAGS Register



ID	=	Identification flag	DF	=	Direction flag
VIP	=	Virtual interrupt pending	IF	=	Interrupt enable flag
VIF	=	Virtual interrupt flag	TF	=	Trap flag
AC	=	Alignment check	SF	=	Sign flag
VM	=	Virtual 8086 mode	ZF	=	Zero flag
RF	=	Resume flag	AF	=	Auxiliary carry flag
NT	=	Nested task flag	PF	=	Parity flag
IOPL	=	I/O privilege level	CF	=	Carry flag
OF	=	Overflow flag			

Status Registers



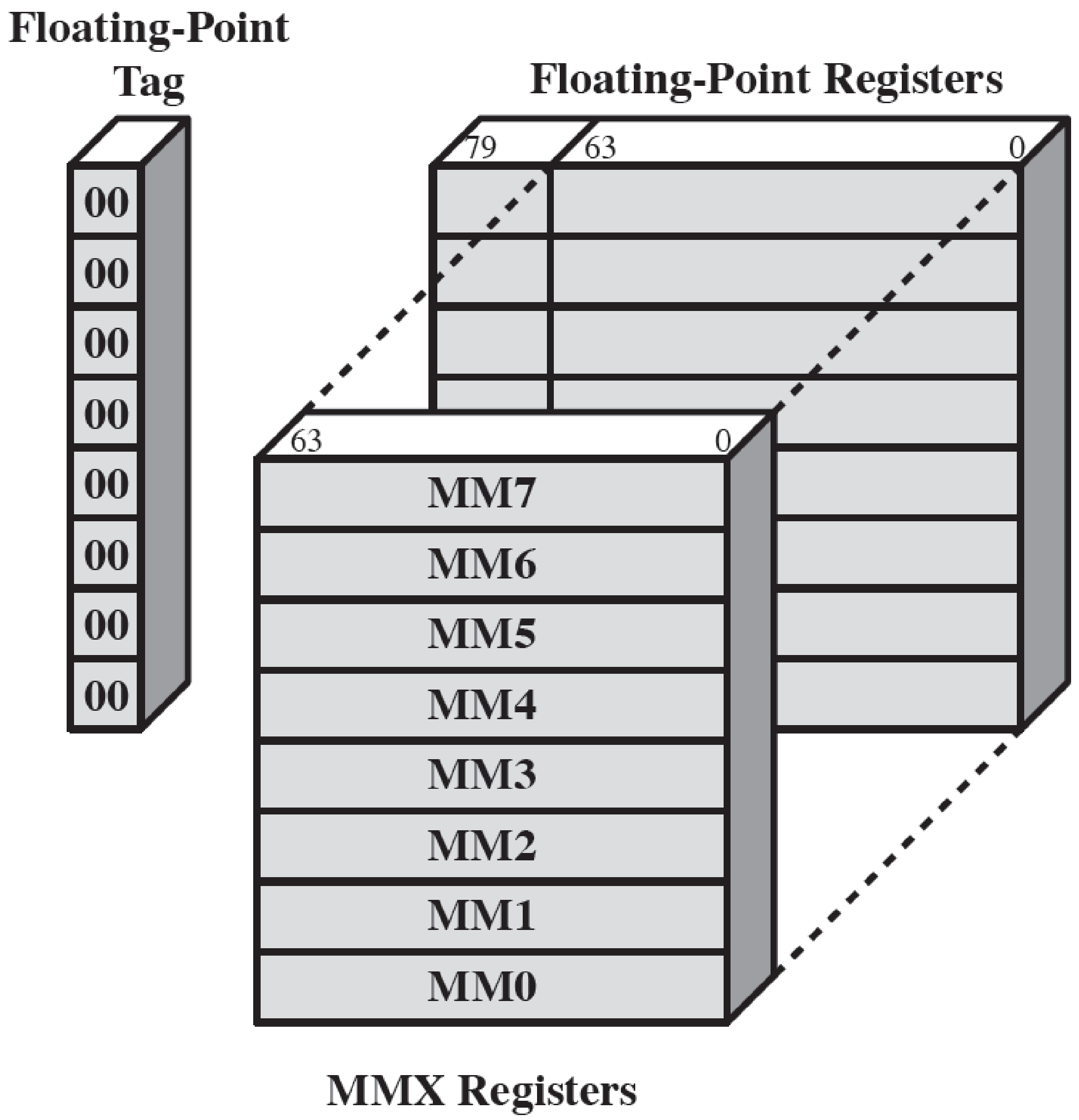
shaded area indicates reserved bits

OSXSAVE	=	XSAVE enable bit	PCD	=	Page-level Cache Disable
SMXE	=	Enable Safer mode extensions	PWT	=	Page-level Writes Transparent
VMXE	=	Enable virtual machine extensions	PG	=	Paging
OSXMMEXCPT	=	Support unmasked SIMD FP exceptions	CD	=	Cache Disable
OSFXSR	=	Support FXSAVE, FXSTOR	NW	=	Not Write Through
PCE	=	Performance Counter Enable	AM	=	Alignment Mask
PGE	=	Page Global Enable	WP	=	Write Protect
MCE	=	Machine Check Enable	NE	=	Numeric Error
PAE	=	Physical Address Extension	ET	=	Extension Type
PSE	=	Page Size Extensions	TS	=	Task Switched
DE	=	Debug Extensions	EM	=	Emulation
TSD	=	Time Stamp Disable	MP	=	Monitor Coprocessor
PVI	=	Protected Mode Virtual Interrupt	PE	=	Protection Enable
VME	=	Virtual 8086 Mode Extensions			

x86 Extension Registers

- **Multi Media Extensions (MMX)**
  - 8 Integer Registers, 64 Bit (mapped registers)
  - 24 new instruction
- **Streaming SIMD Extensions (SSE)**
  - 8 new floating point registers (128 Bit)
  - With amd64 architecture 8 additional 128 Bit registers
  - 70 new Instructions
- **SSE2-4**
  - Additional instructions

Mapping of the MMX Registers


Floating-Point Tag
Floating-Point Registers
79
63
0
00
00
00
00
00
00
00
00
63
0
MM7
MM6
MM5
MM4
MM3
MM2
MM1
MM0
MMX Registers