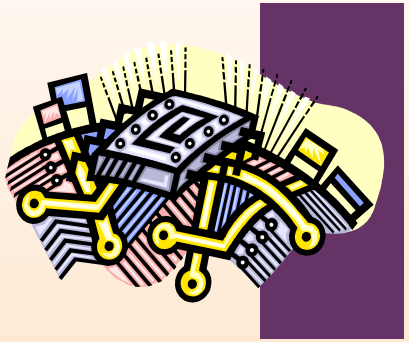
William Stallings Computer Organization and Architecture 10th Edition

Chapter 14

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

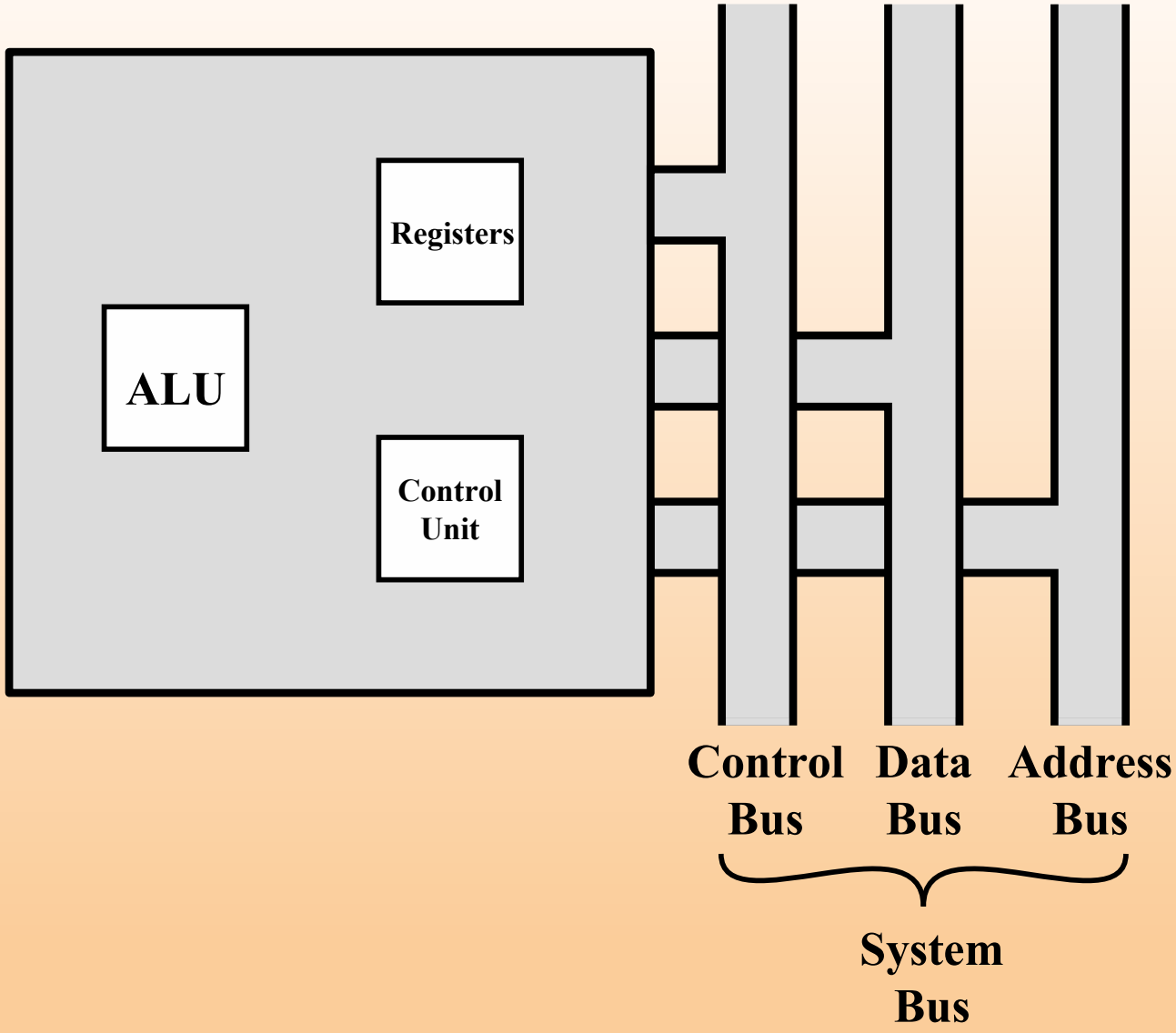


Figure 14.1 The CPU with the System Bus

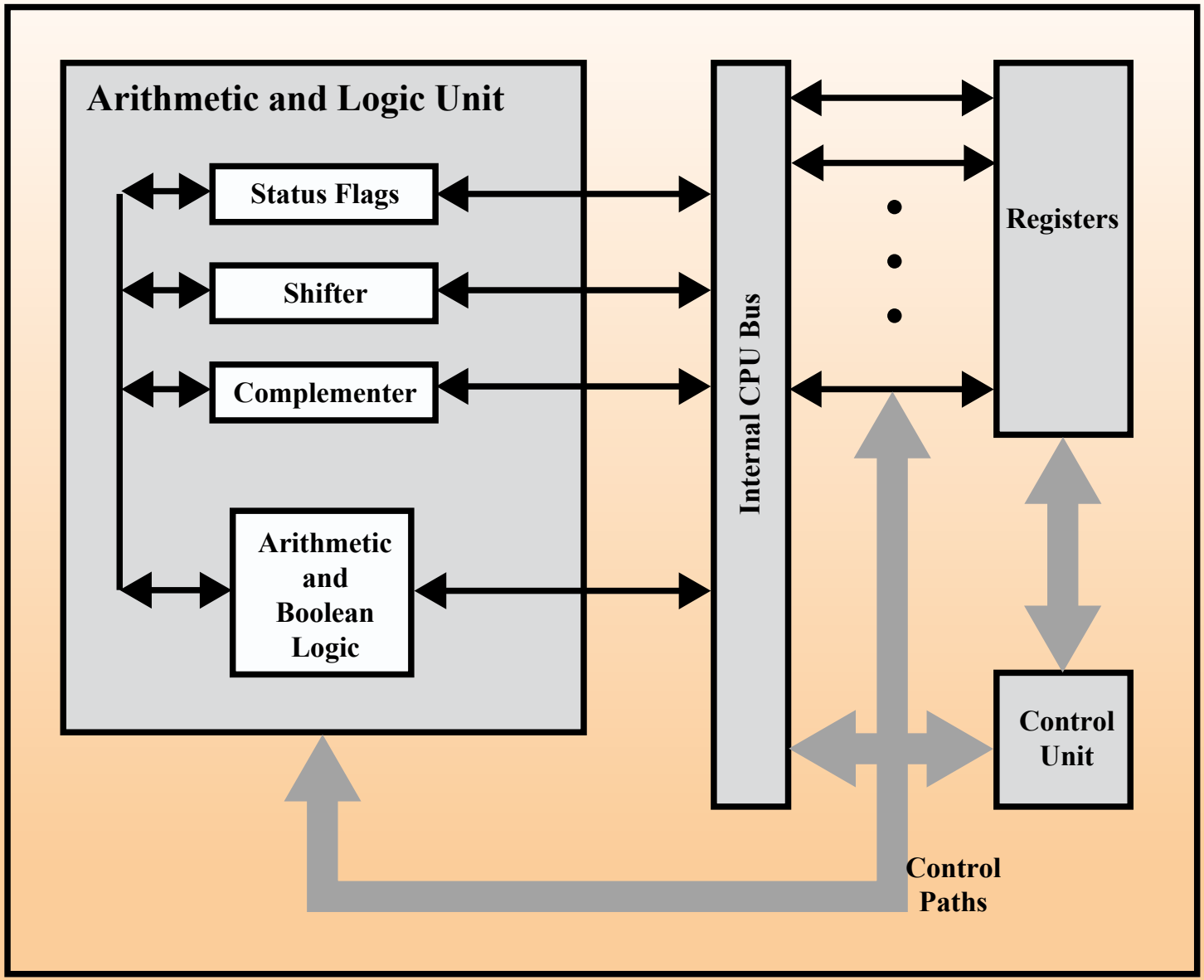


Figure 14.2 Internal Structure of the CPU

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

- 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

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

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:

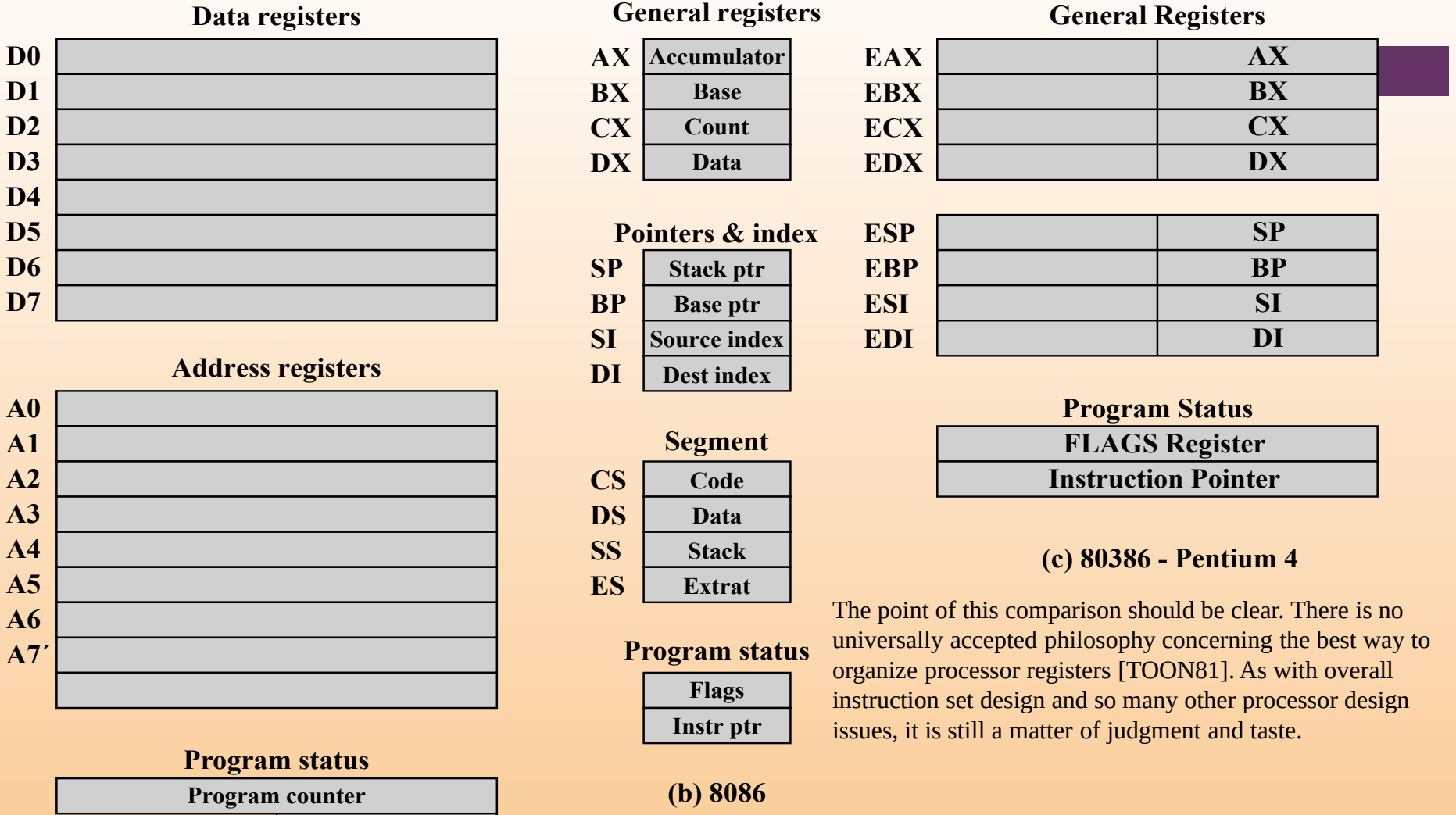
- Program counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Memory address register (MAR)
  - Contains the address of a location in memory
- Memory buffer register (MBR)
  - Contains a word of data to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

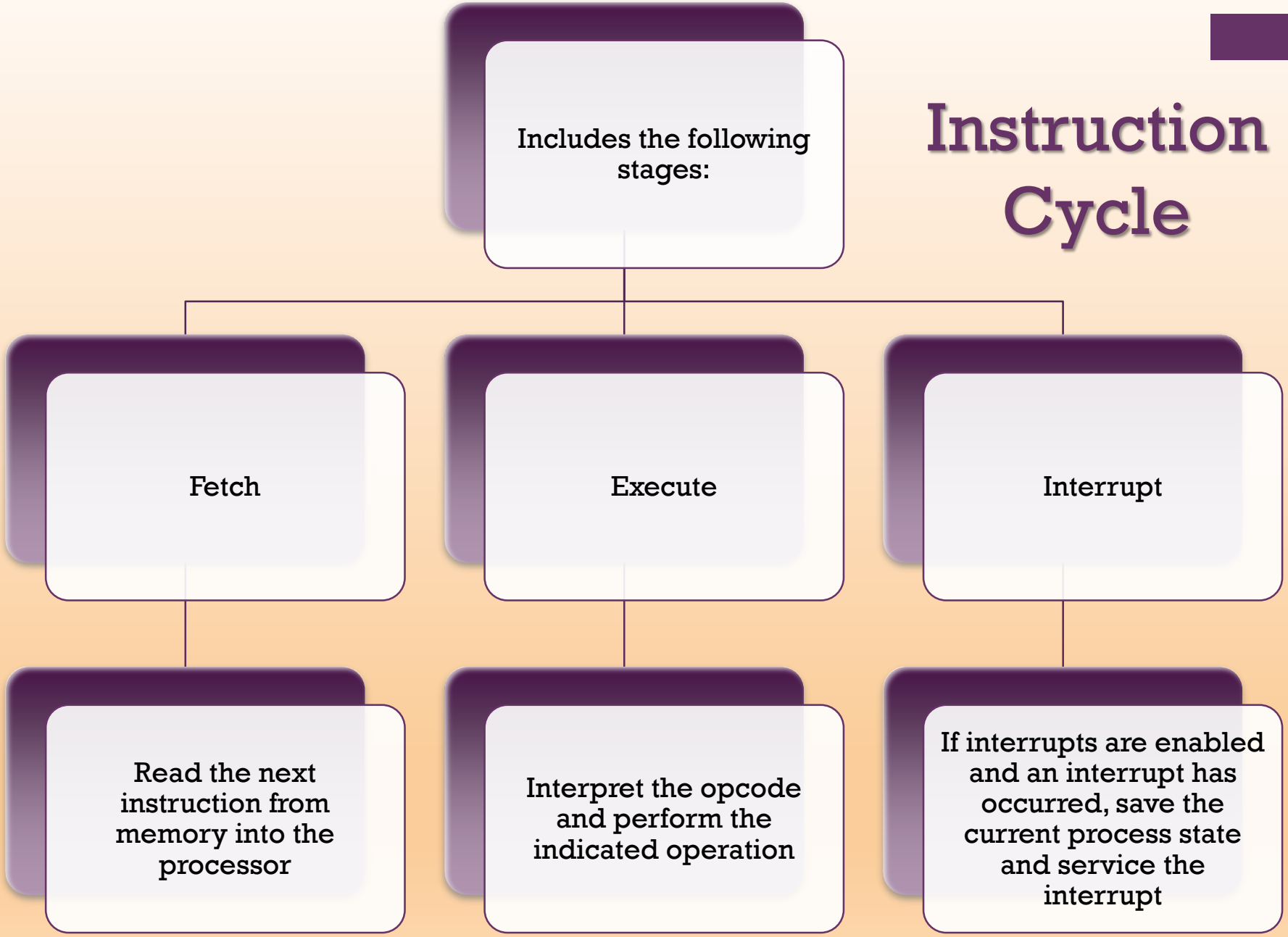


The point of this comparison should be clear. There is no universally accepted philosophy concerning the best way to organize processor registers [TOON81]. As with overall instruction set design and so many other processor design issues, it is still a matter of judgment and taste.

A second instructive point concerning register organization design is illustrated in Figure 14.3c. This figure shows the user-visible register organization for the Intel 80386 [ELAY85], which is a 32-bit microprocessor designed as an extension of the 8086. The 80386 uses 32-bit registers. However, to provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization. Given this design constraint, the architects of the 32-bit processors had limited flexibility in designing the register organization.

Figure 14.3 Example Microprocessor Register Organizations

Instruction Cycle



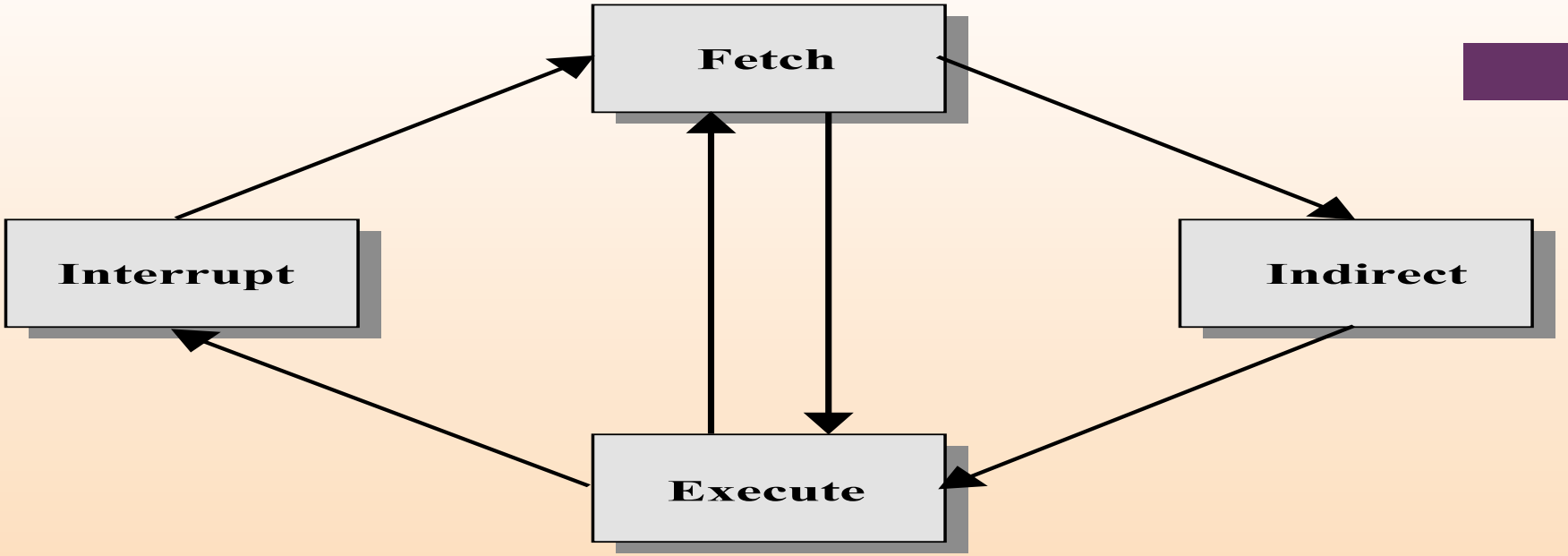


Figure 14.4 The Instruction Cycle

- The main line of activity consists of alternating instruction fetch and instruction execution activities.
- After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.
- Following execution, an interrupt may be processed before the next instruction fetch.

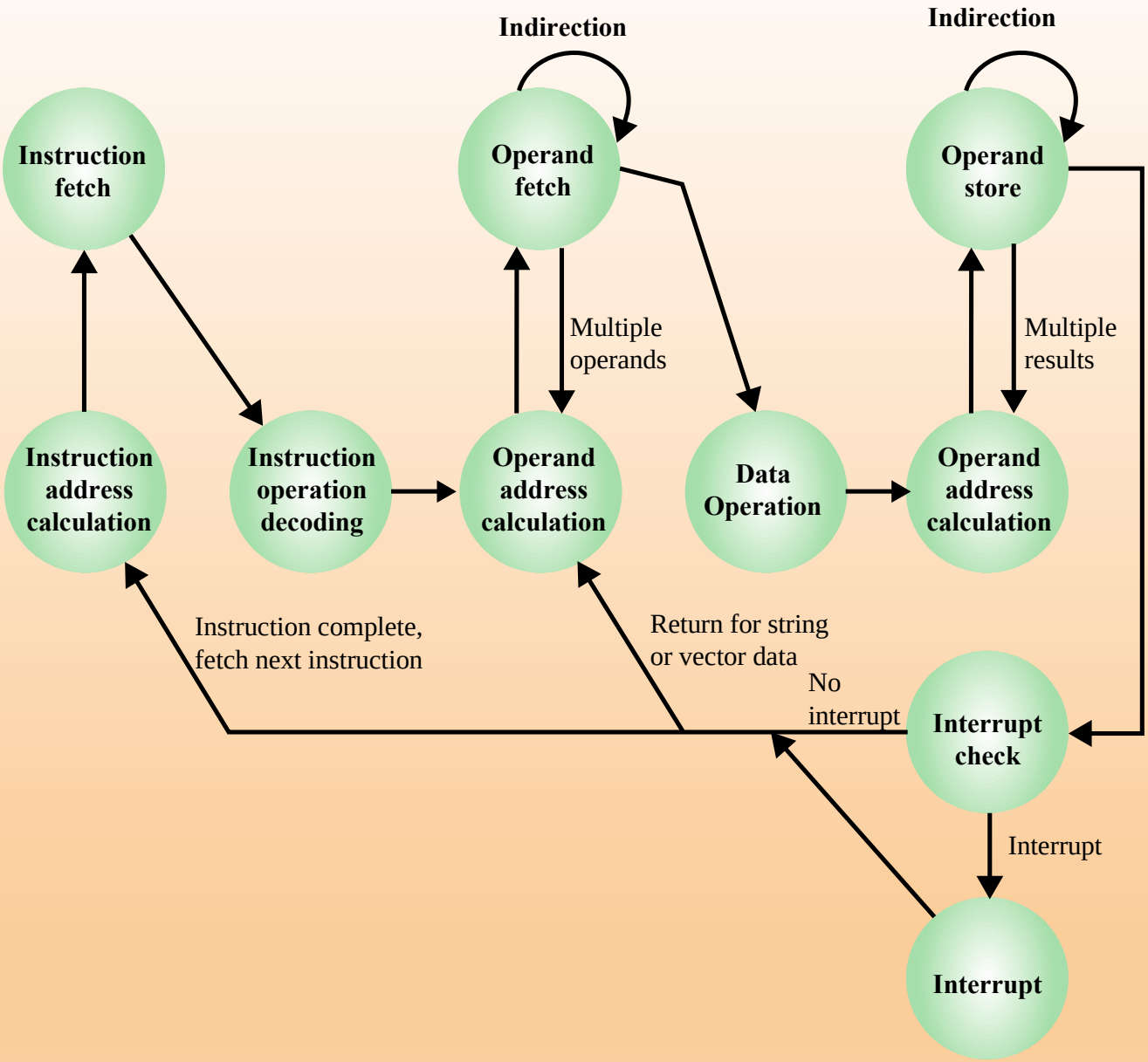


Figure 14.5 Instruction Cycle State Diagram

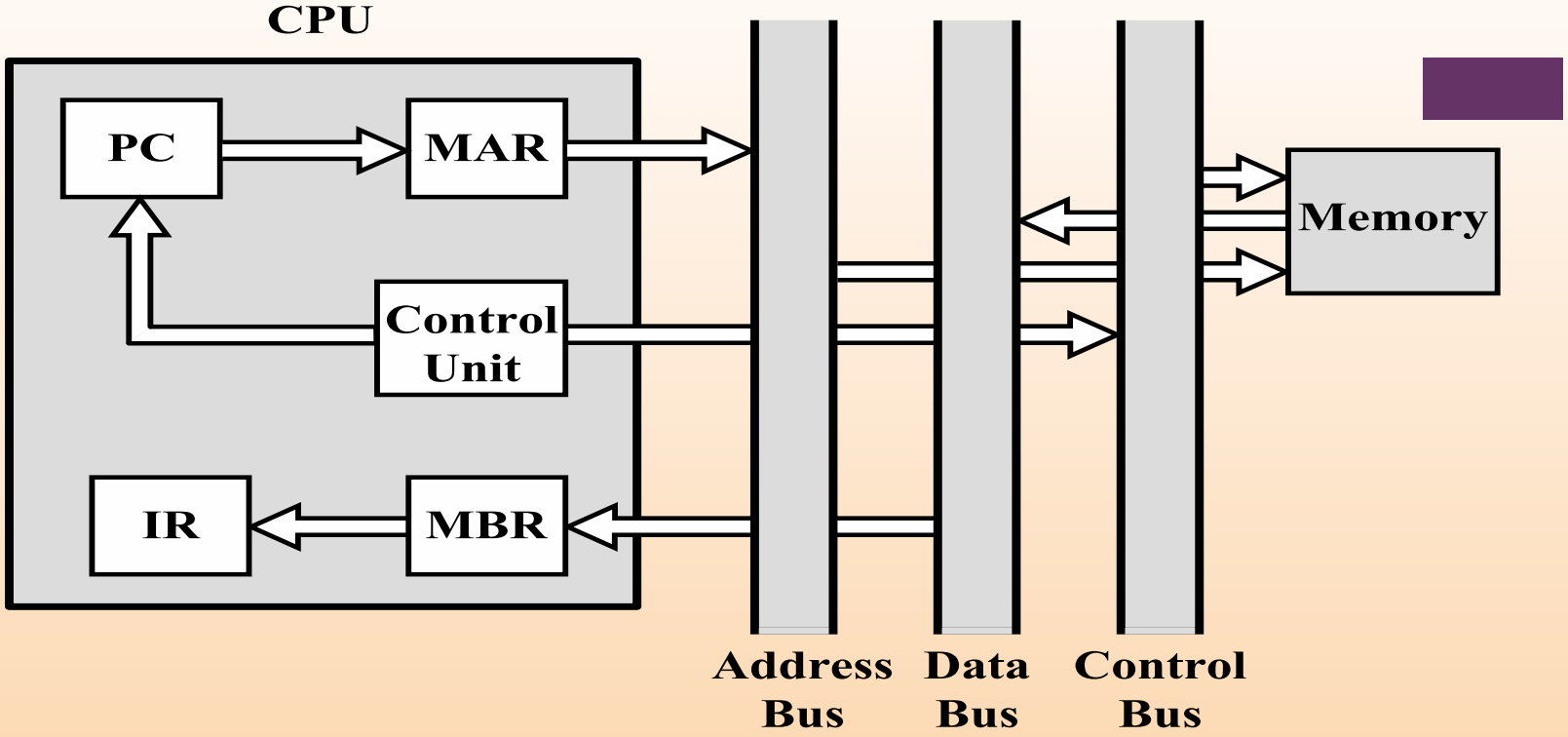


Figure 14.6 Data Flow, Fetch Cycle

During the *fetch cycle,* an instruction is read from memory. Figure 14.6 shows the flow of data during this cycle. 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.

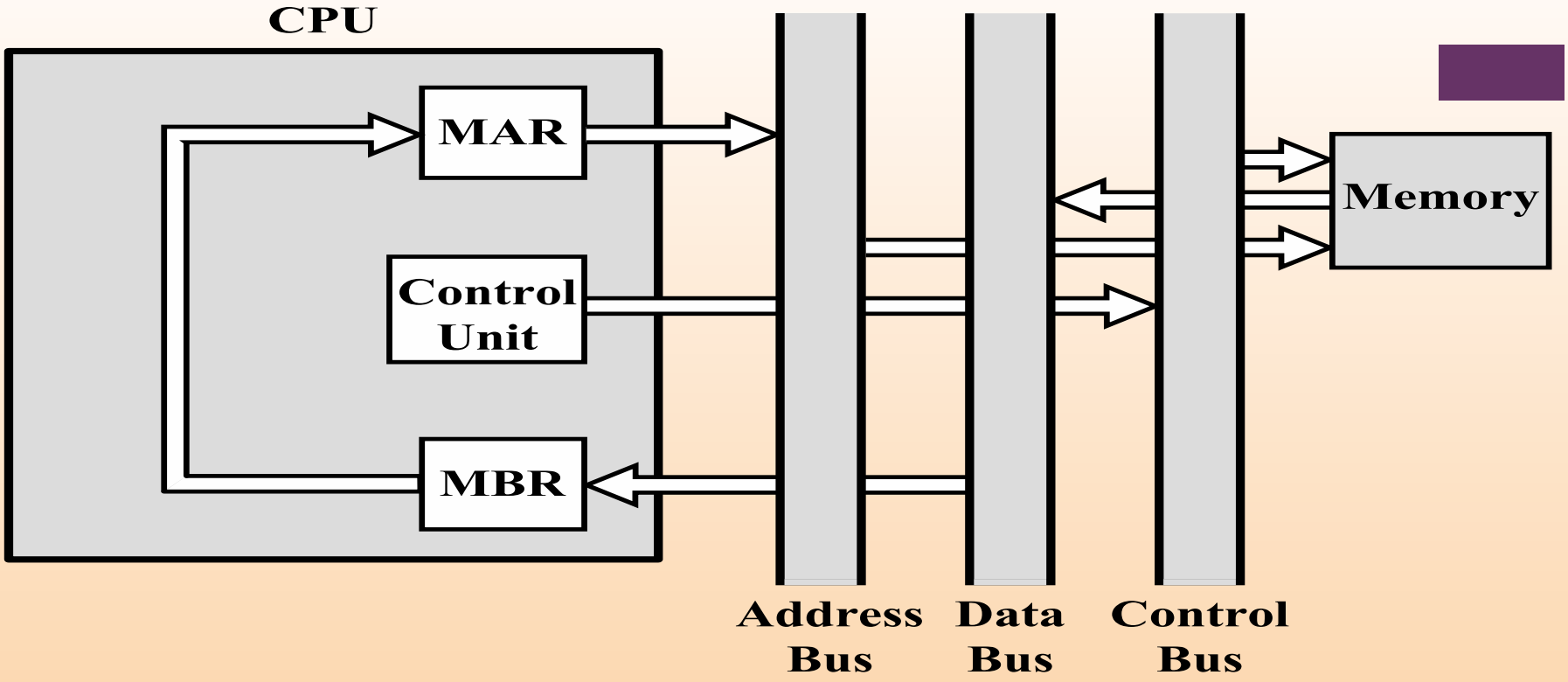


Figure 14.7 Data Flow, Indirect Cycle

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. As shown in Figure 14.7, this is a simple cycle. 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.

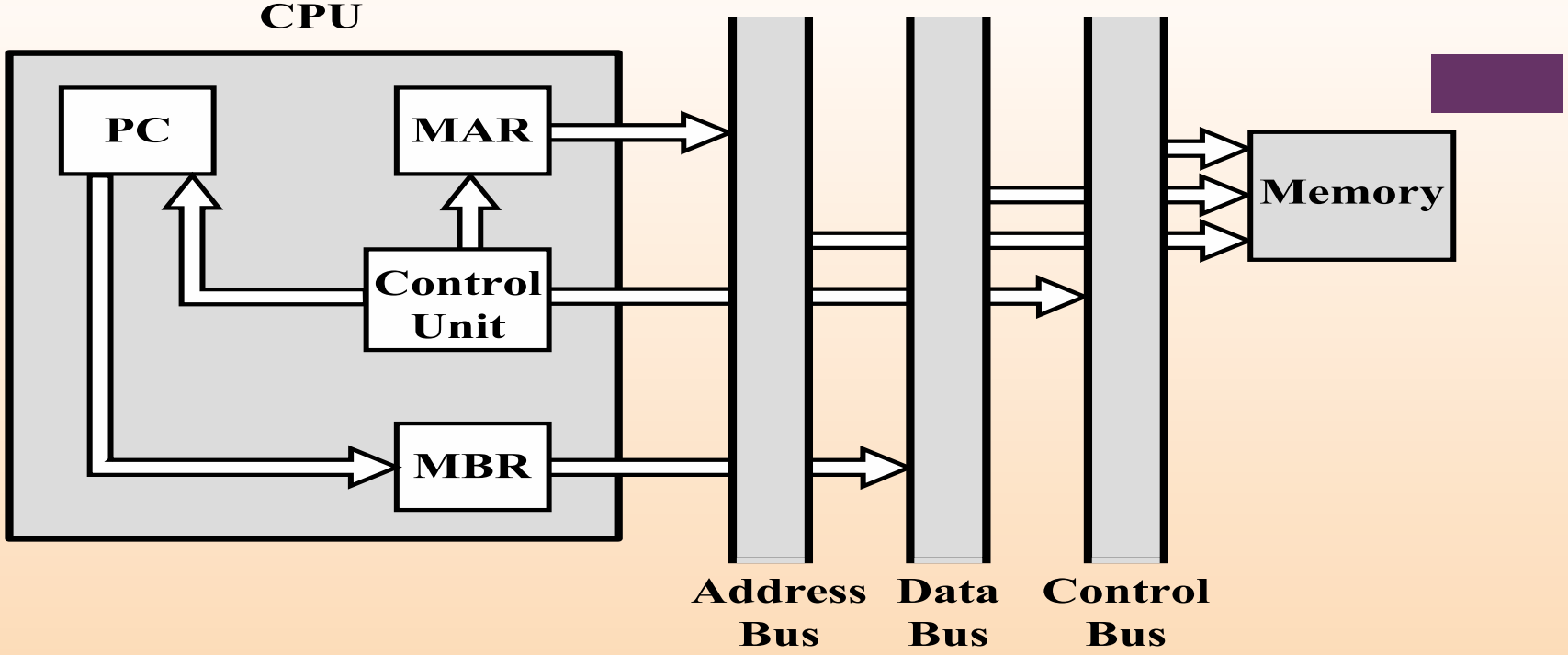


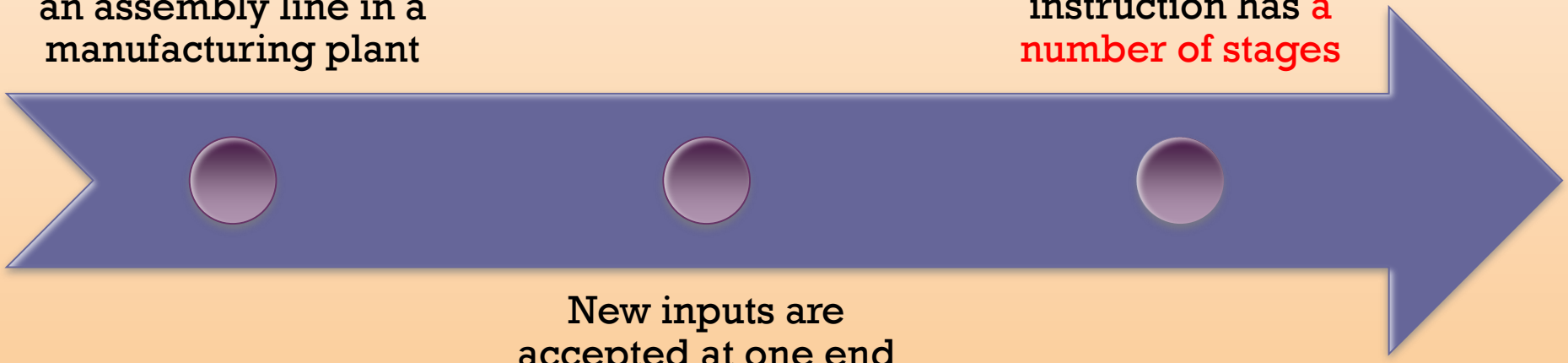
Figure 14.8 Data Flow, Interrupt Cycle

Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable (Figure 14.8). 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.

Pipelining Strategy

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

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



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

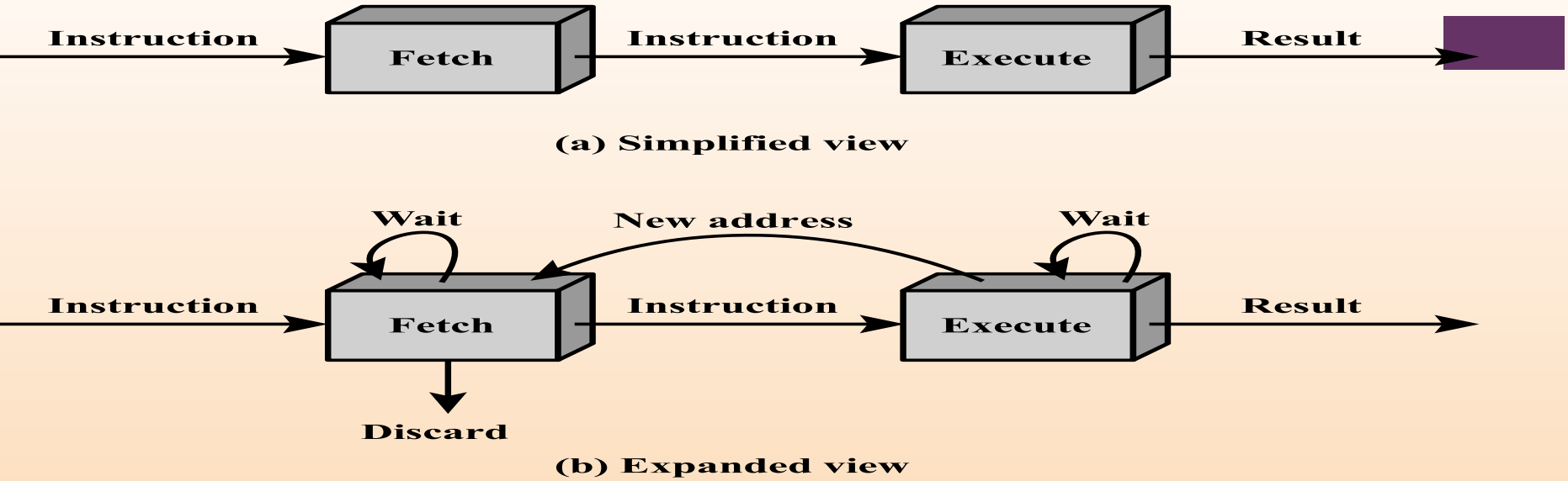


Figure 14.9 Two-Stage Instruction Pipeline

While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called instruction prefetch or *fetch overlap*. Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 14.9b), we will see that this doubling of execution rate is unlikely for two reasons:

-The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to 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. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

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

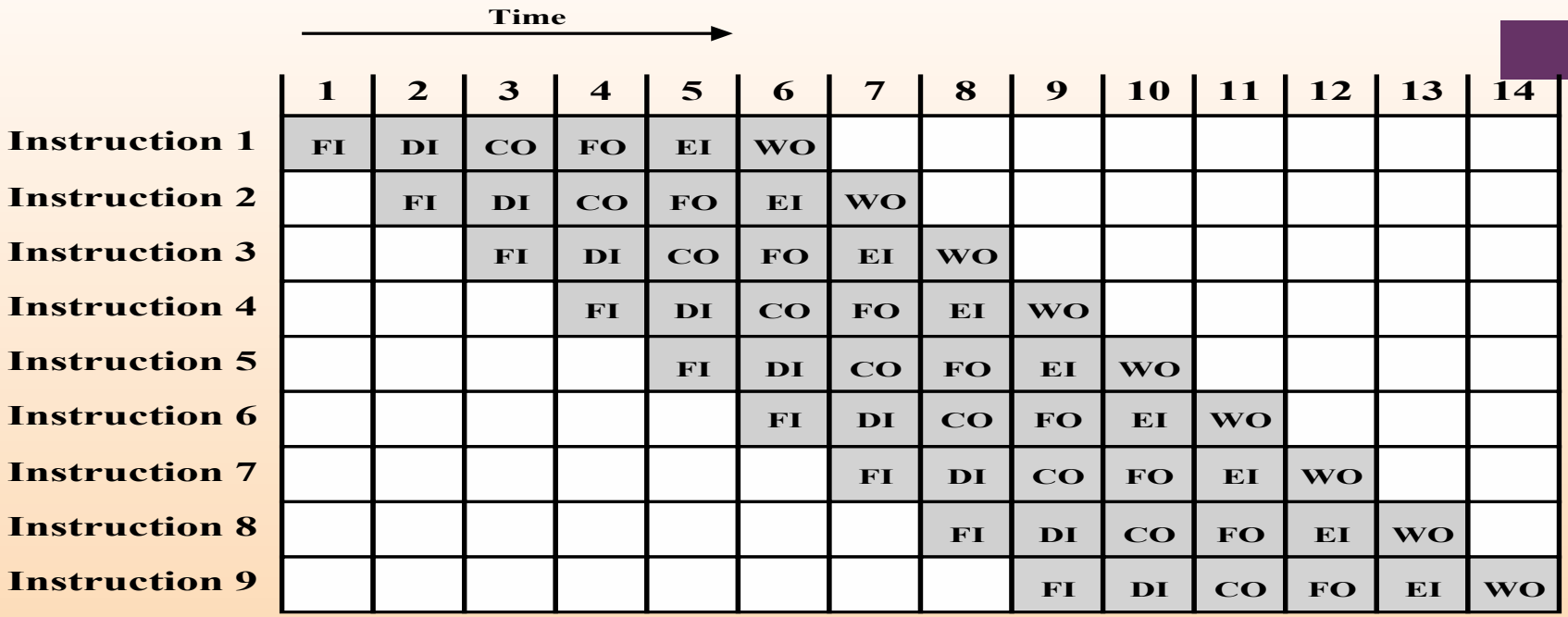


Figure 14.10 Timing Diagram for Instruction Pipeline Operation

For the sake of illustration, let us assume equal duration. Using this assumption, Figure 14.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously. Most memory systems will not permit that. However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

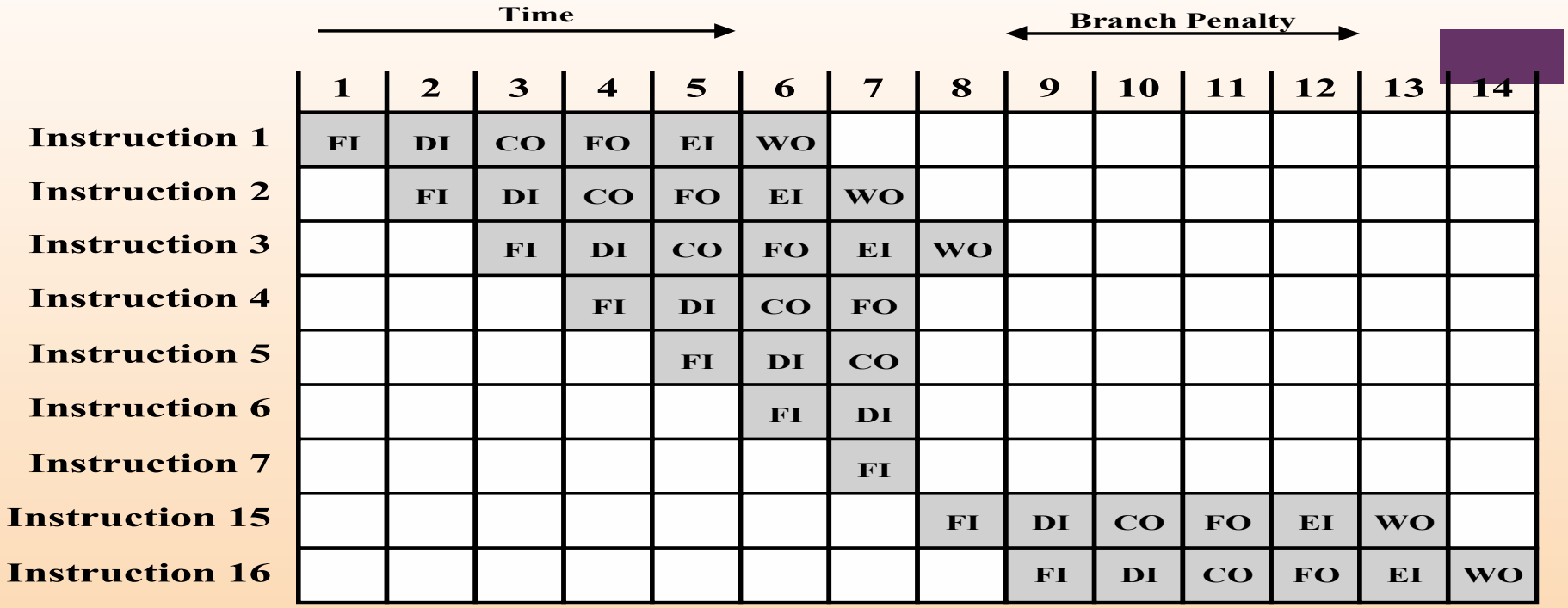


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt.

Figure 14.11 illustrates the effects of the conditional branch, using the same program as Figure 14.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. The branch is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch.

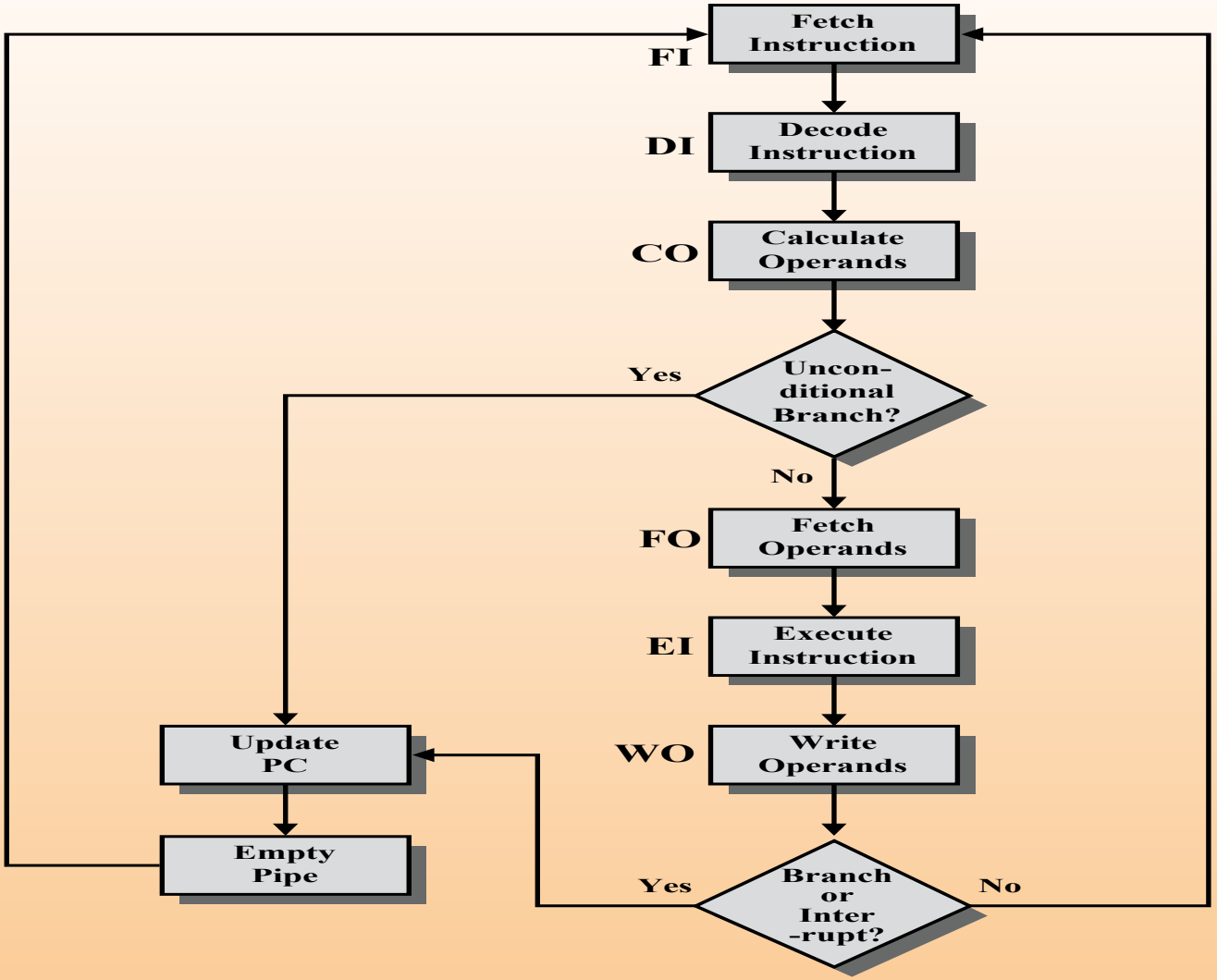


Figure 14.12 Six-Stage Instruction Pipeline

Figure 14.12 indicates the logic needed for pipelining to account for branches and interrupts.

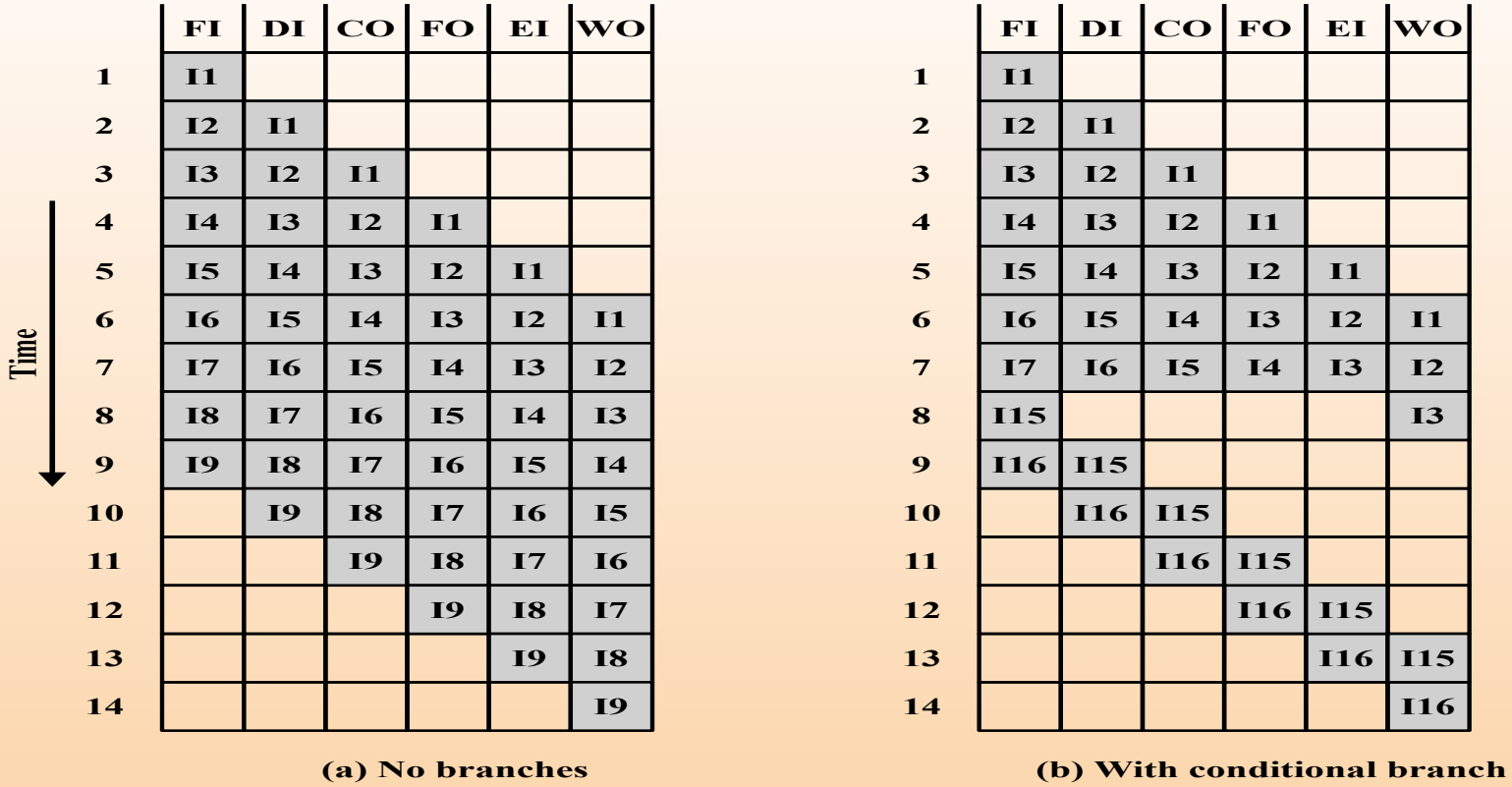


Figure 14.13 An Alternative Pipeline Depiction

In Figure 14.13a (which corresponds to Figure 14.10), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

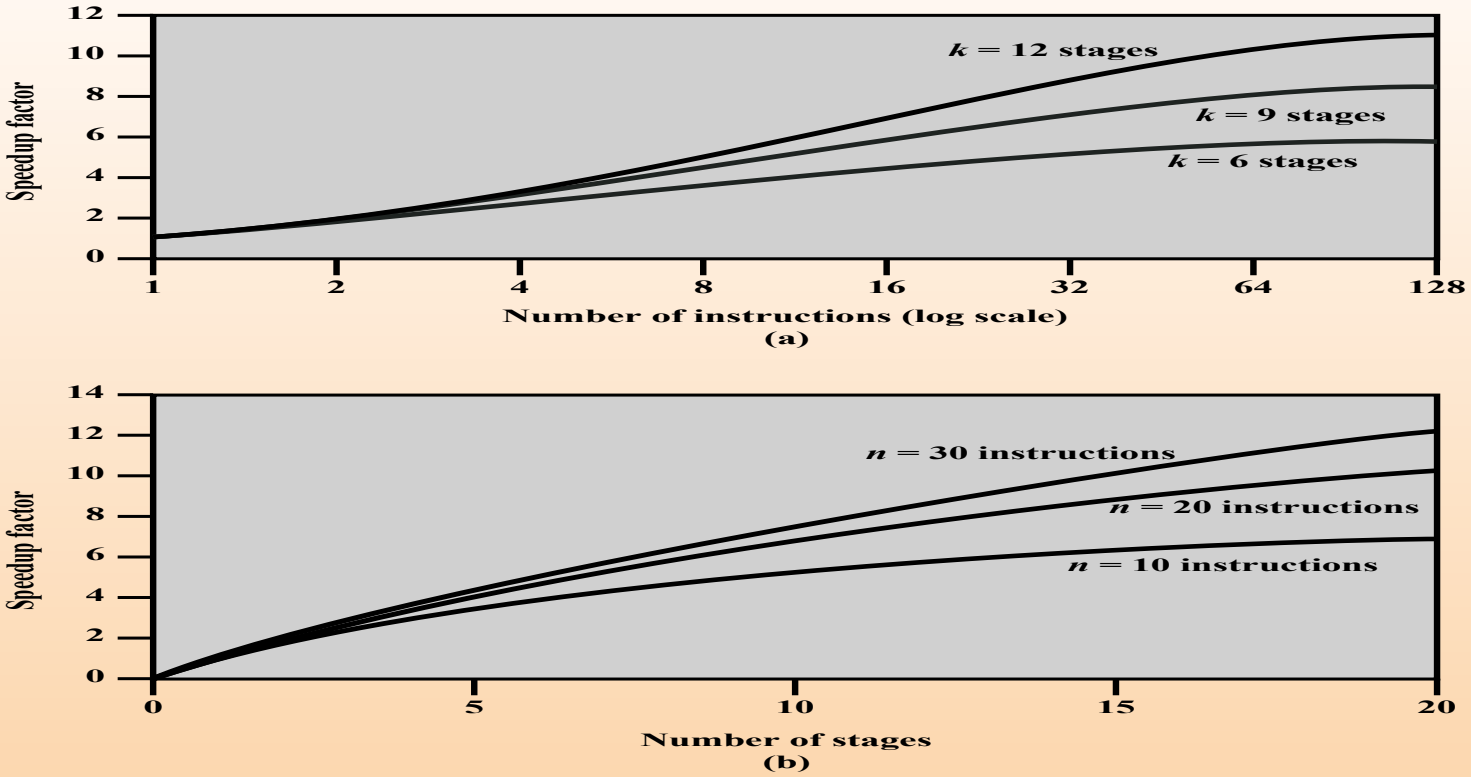


Figure 14.14 Speedup Factors with Instruction Pipelining

Figure 14.14a plots the speedup factor as a function of the number of instructions that are executed without a branch.

Figure 14.14b shows the speedup factor as a function of the number of stages in the instruction pipeline

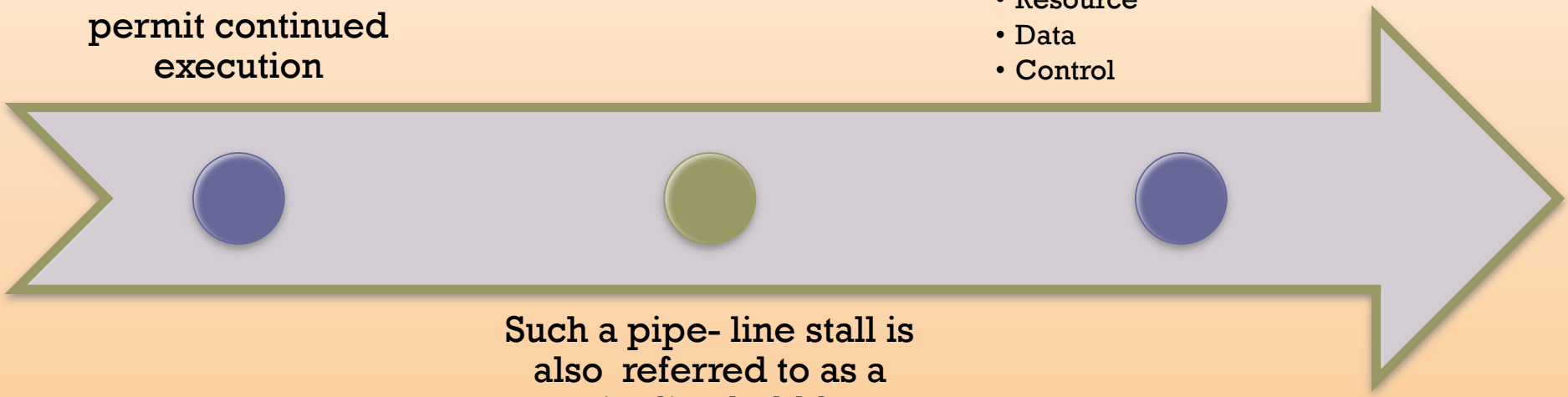
Thus, the larger the number of pipeline stages, the greater the potential for speedup. However, as a practical matter, the potential gains of additional pipe- line stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

Pipeline Hazards

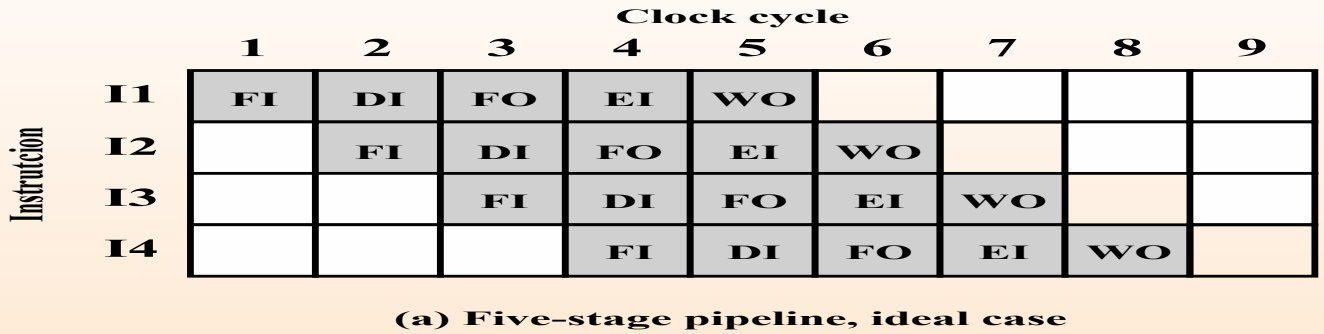
A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution

There are three types of hazards:

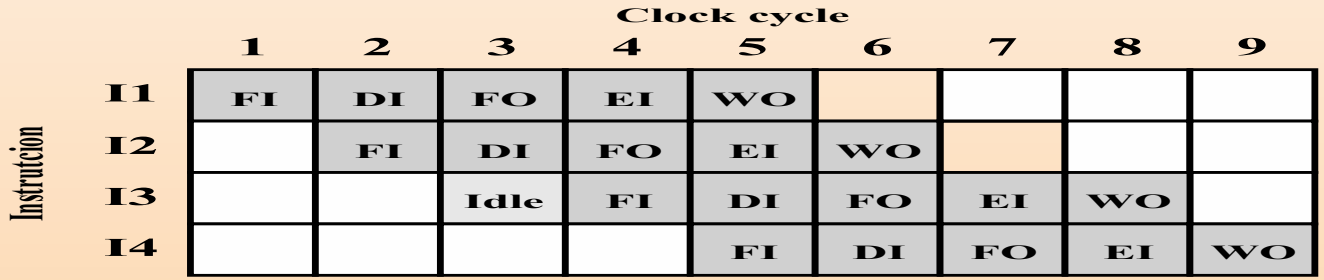
- Resource
- Data
- Control



Such a pipe- line stall is also referred to as a *pipeline bubble*



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Figure 14.15 Example of 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 sometime referred to as a *structural hazard.*

Now 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. Further, ignore the cache. In this case, 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.

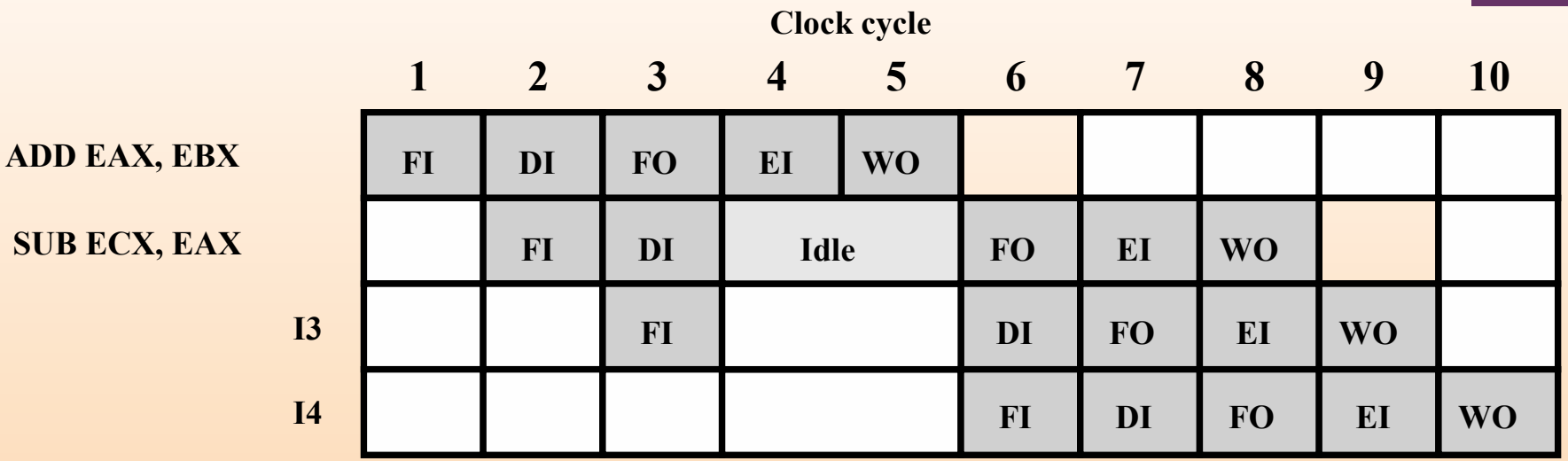


Figure 14.16 Example of Data Hazard

ADD EAX, EBX /* EAX = EAX + EBX

SUB ECX, EAX /* ECX = ECX – EAX

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.

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

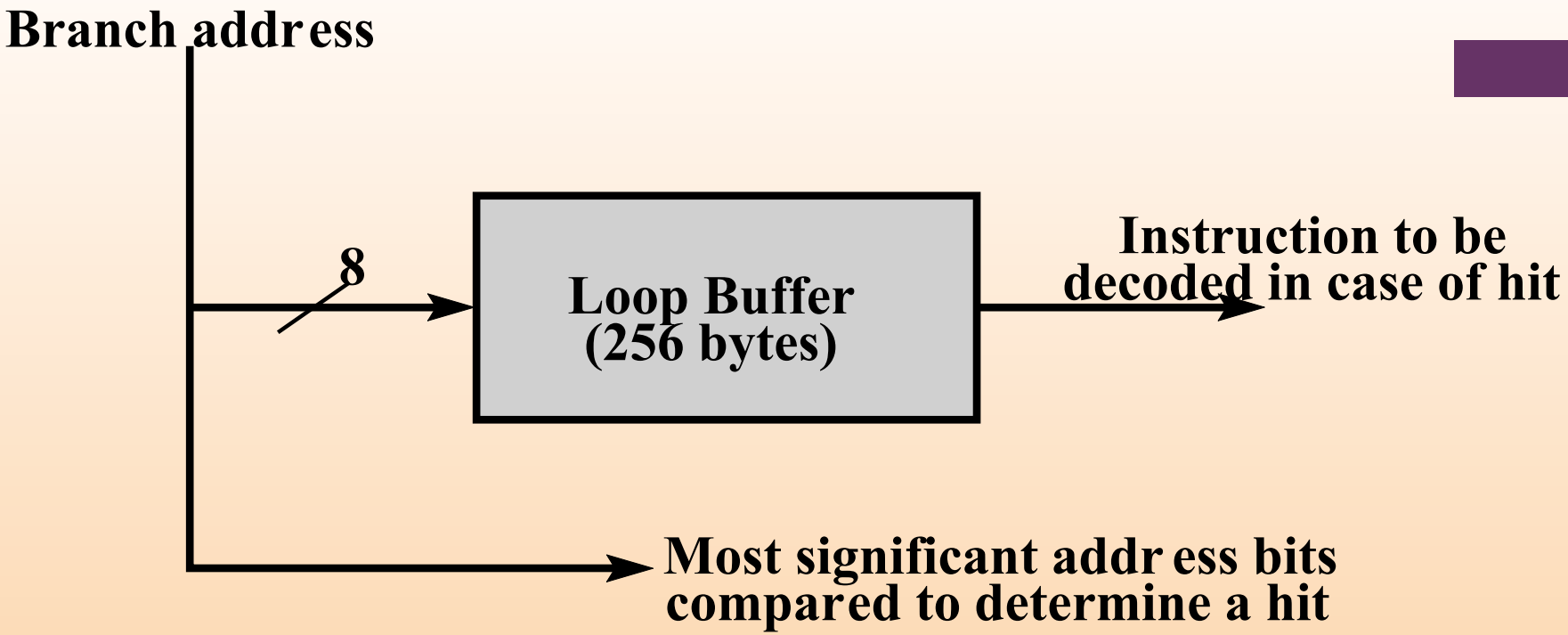


Figure 14.17 Loop Buffer

Figure 14.17 gives an example of a loop buffer. If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the buffer. The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.

Among the machines using a loop buffer are some of the CDC machines (Star- 100, 6600, 7600) and the CRAY-1. A specialized form of loop buffer is available on the Motorola 68010, for executing a three-instruction loop involving the DBcc (decrement and branch on condition) instruction (see Problem 14.14). A three-word buffer is maintained, and the processor executes these instructions repeatedly until the loop condition is satisfied.

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

The first two approaches are the simplest. These either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The predict-never-taken approach is the most popular of all the branch prediction methods. Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88], and so if the cost of prefetching from either path is the same, then always prefetching from the branch target address should give better performance than always prefetching from the sequential path.

The final static approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. [LILJ88] reports success rates of greater than 75% with this strategy.

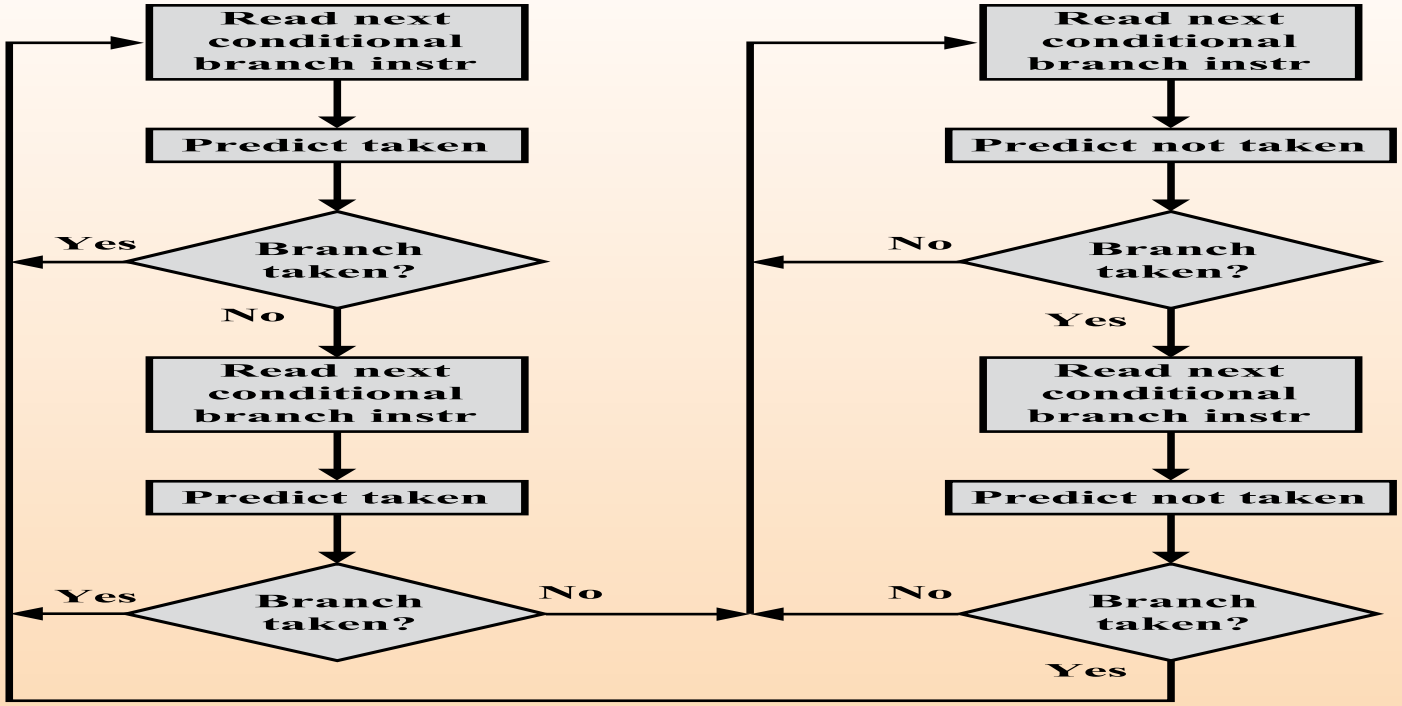


Figure 14.18 Branch Prediction Flow Chart

With a single bit, all that can be recorded is whether the last execution of this instruction resulted in a branch or not. A shortcoming of using a single bit appears in the case of a conditional branch instruction that is almost always taken, such as a loop instruction. With only one bit of history, an error in prediction will occur twice for each use of the loop: once on entering the loop, and once on exiting.

If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction, or to record a state in some other fashion. Figure 14.18 shows a typical approach (see Problem 14.13 for other possibilities). Assume that the algorithm starts at the upper-left-hand corner of the flowchart. As long as each succeeding conditional branch instruction that is encountered is taken, the decision process predicts that the next branch will be taken. If a single prediction is wrong, the algorithm continues to predict that the next branch is taken. Only if two successive branches are not taken does the algorithm shift to the right-hand side of the flowchart. Subsequently, the algorithm will predict that branches are not taken until two branches in a row are taken. Thus, the algorithm requires two consecutive wrong predictions to change the prediction decision.

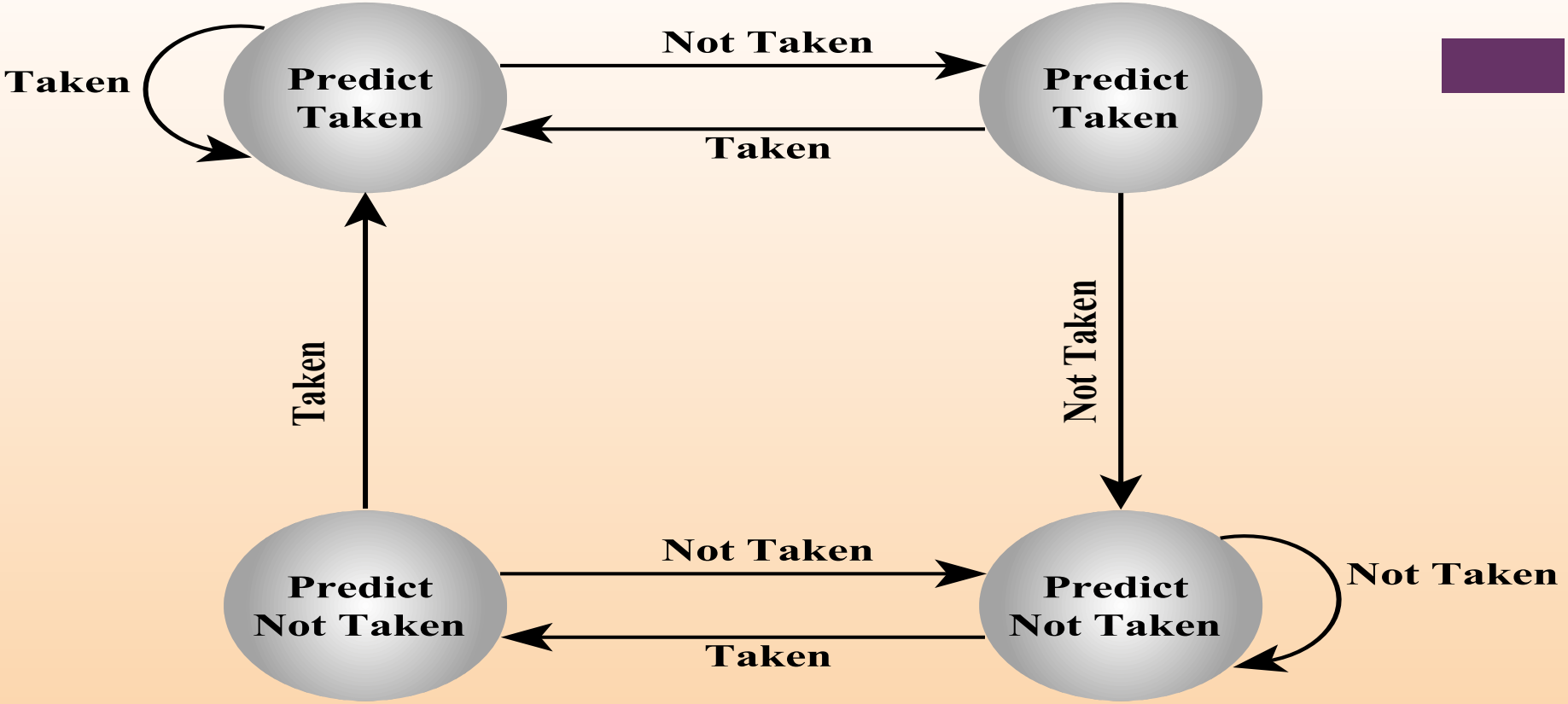
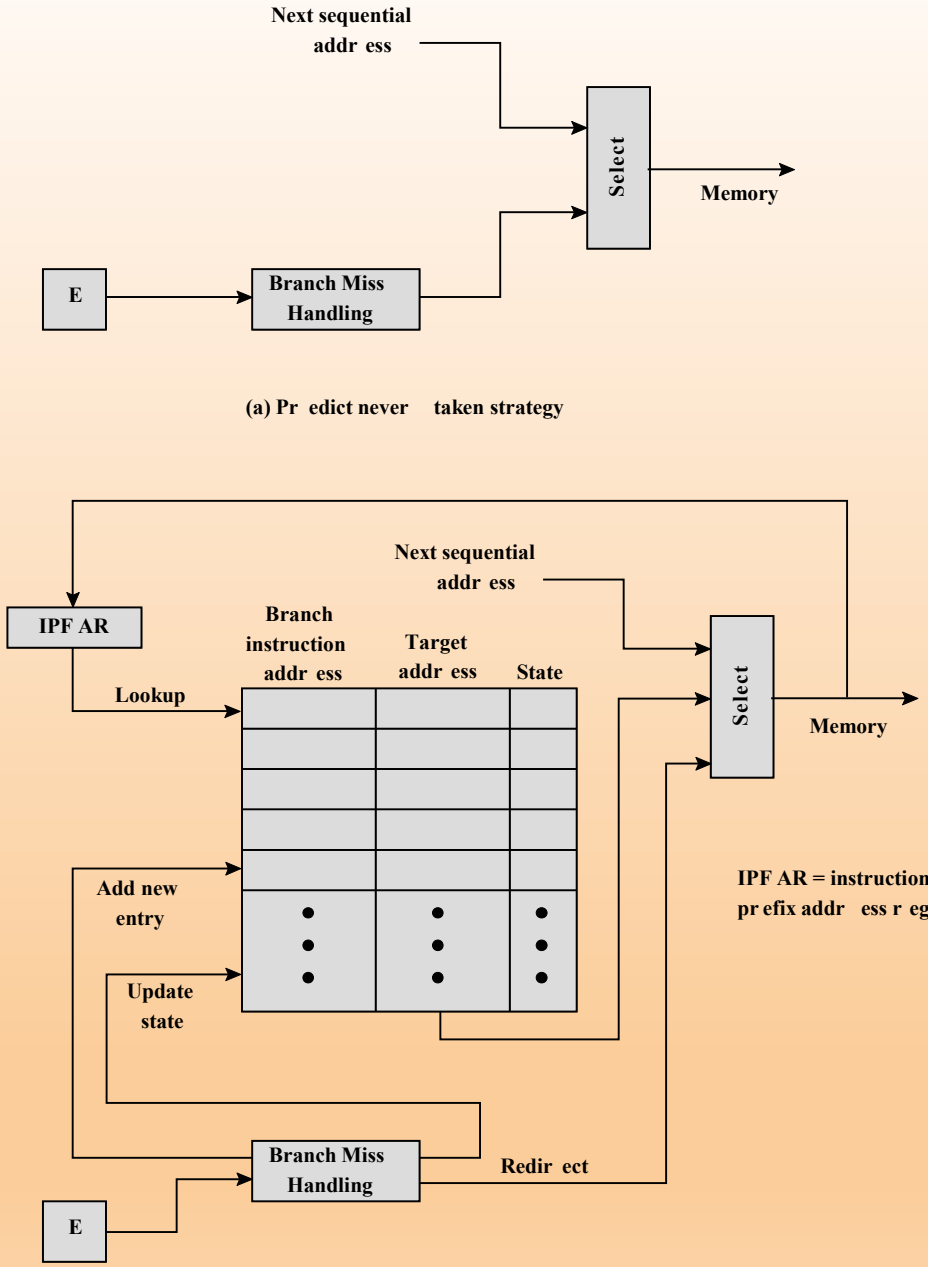


Figure 14.19 Branch Prediction State Diagram

-The decision process can be represented more compactly by a finite-state machine, shown in Figure 14.19.

-The use of history bits, as just described, has one drawback: If the decision is made to take the branch, the target instruction cannot be fetched until the tar- get address, which is an operand in the conditional branch instruction, is decoded. Greater efficiency could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, more information must be saved, in what is known as a branch target buffer, or a branch history table.



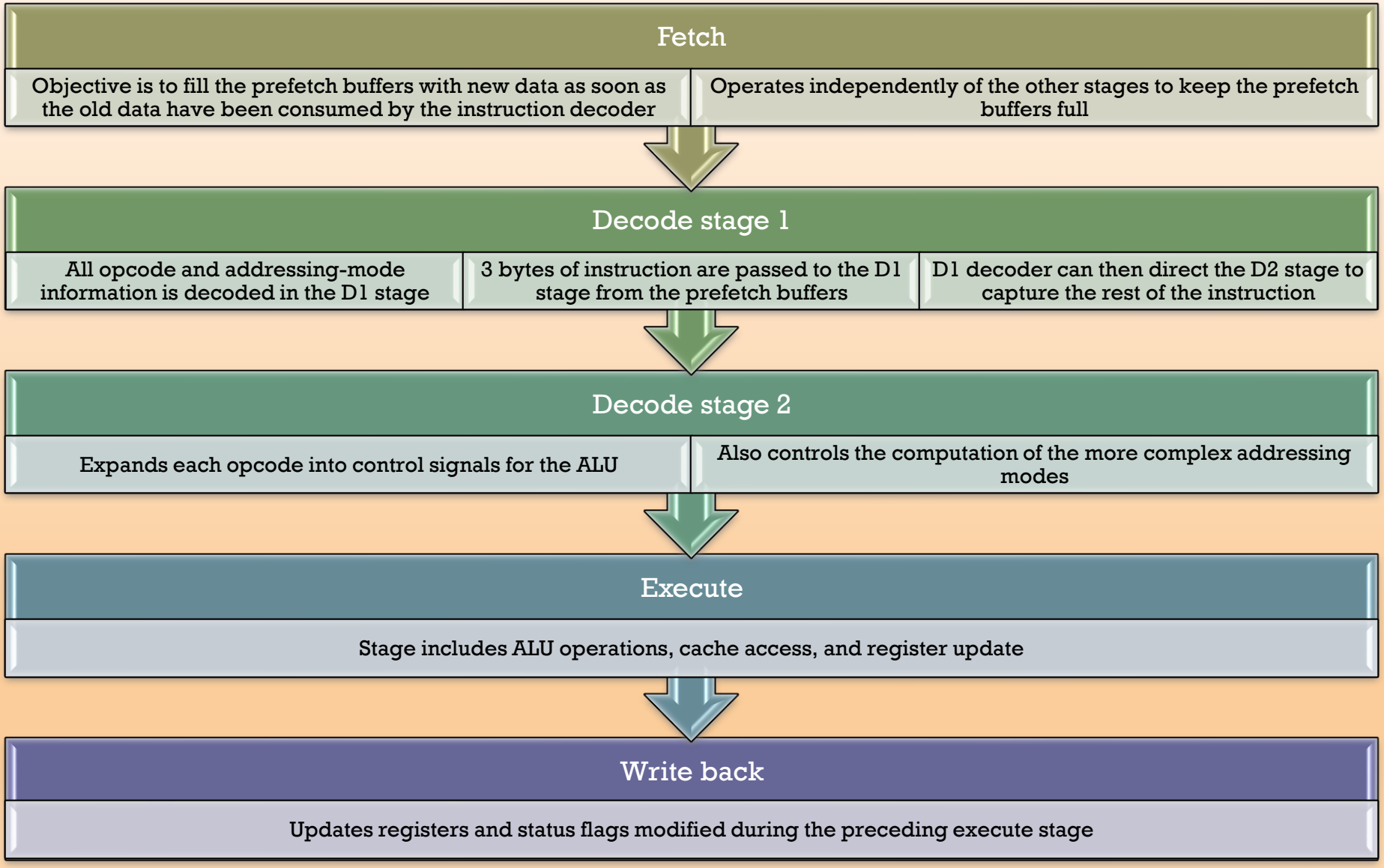
Figur e 14.20 Dealing with Branches

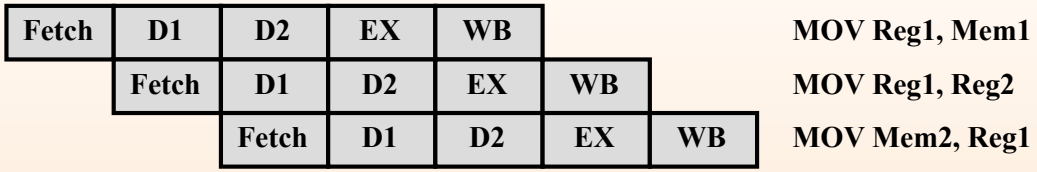
-Figure 14.20 contrasts this scheme with a predict-never-taken strategy. With the former strategy, the instruction fetch stage always fetches the next sequential address. If a branch is taken, some logic in the processor detects this and instructs that the next instruction be fetched from the target address (in addition to flushing the pipeline).

-The branch history table is treated as a cache. Each prefetch triggers a lookup in the branch history table. If no match is found, the next sequential address is used for the fetch. If a match is found, a prediction is made based on the state of the instruction: Either the next sequential address or the branch target address is fed to the select logic.

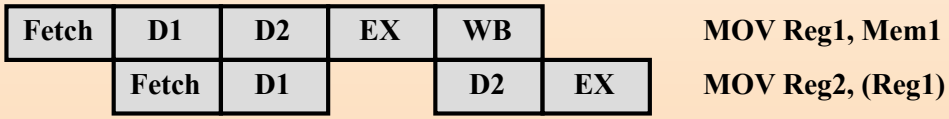
-When the branch instruction is executed, the execute stage signals the branch history table logic with the result. The state of the instruction is updated to reflect a correct or incorrect prediction. If the prediction is incorrect, the select logic is redirected to the correct address for the next fetch. When a conditional branch instruction is encountered that is not in the table, it is added to the table and one of the existing entries is discarded, using one of the cache replacement algorithms discussed in Chapter 4.

Intel 80486 Pipelining

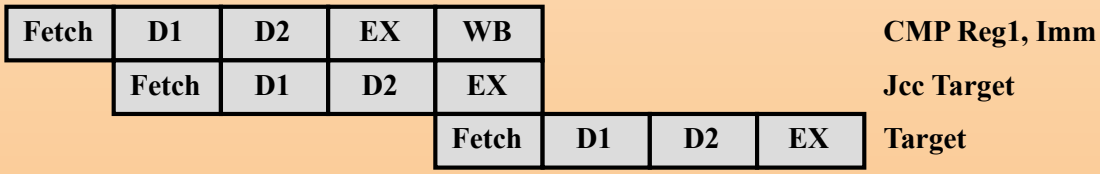




(a) No Data Load Delay in the Pipeline



(b) Pointer Load Delay



(c) Branch Instruction Timing

Figure 14.21 80486 Instruction Pipeline Examples

-Figure 14.21 shows examples of the operation of the pipeline. Figure 14.21a shows that there is no delay introduced into the pipeline when a memory access is required.

-However, as Figure 14.21b shows, there can be a delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register and that register is then used as a base register in the next instruction, the processor will stall for one cycle.

-Figure 14.21c illustrates the timing of a branch instruction, assuming that the branch is taken. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the jump instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump during the EX stage of the jump instruction. If the processor determines a false branch condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded).

The x86 Processor Family

- The x86 organization has evolved dramatically over the years. In this section we examine some of the details of the most recent processor organizations, concentrating on common elements in single processors

- **Register Organization:**

- **General:** There are eight 32-bit general-purpose registers (see Figure 14.3c). These may be used for all types of x86 instructions; they can also hold operands for address calculations. In addition, some of these registers also serve special purposes. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands without having to reference these registers explicitly in the instruction. As a result, a number of instructions can be encoded more compactly. In 64-bit mode, there are sixteen 64-bit general-purpose registers.

- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in Chapter 8. The code segment (CS) register references the segment containing the instruction being executed. The stack segment (SS) register references the segment containing a user-visible stack. The remaining segment registers (DS, ES, FS, GS) enable the user to reference up to four separate data segments at a time.
- **Flags:** The 32-bit EFLAGS register contains condition codes and various mode bits. In 64-bit mode, this register is extended to 64 bits and referred to as RFLAGS. In the current architecture definition, the upper 32 bits of RFLAGS are unused.
- **Instruction pointer:** Contains the address of the current instruction.

There are also registers specifically devoted to the floating-point unit:

- **Numeric:** Each register holds an extended-precision 80-bit floating-point number. There are eight registers that function as a stack, with push and pop operations available in the instruction set.

- **Control:** The 16-bit control register contains bits that control the operation of the floating-point unit, including the type of rounding control; single, double, or extended precision; and bits to enable or disable various exception conditions.
- **Status:** The 16-bit status register contains bits that reflect the current state of the floating-point unit, including a 3-bit pointer to the top of the stack; condition codes reporting the outcome of the last operation; and exception flags.
- **Tag word:** This 16-bit register contains a 2-bit tag for each floating-point numeric register, which indicates the nature of the contents of the corresponding register. The four possible values are valid, zero, special (NaN, infinity, denormalized), and empty. These tags enable programs to check the contents of a numeric register without performing complex decoding of the actual data in the register. For example, when a context switch is made, the processor need not save any floating-point registers that are empty.

Table 14.2

x86 Processor Registers

(a) Integer Unit in 32-bit Mode

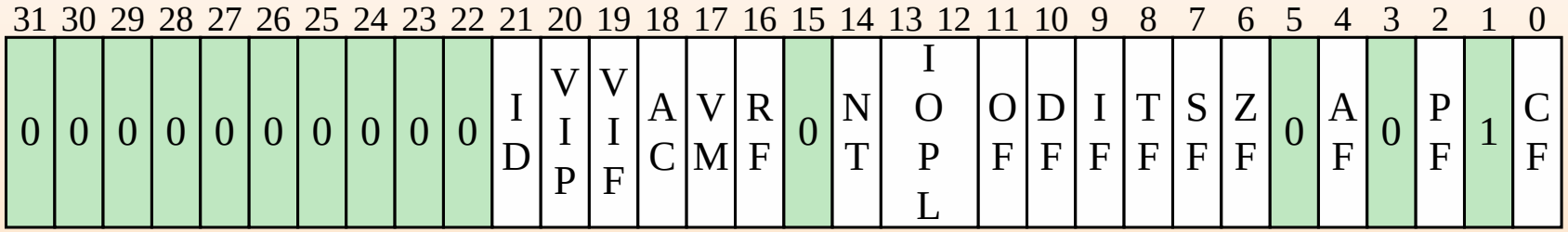
Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(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

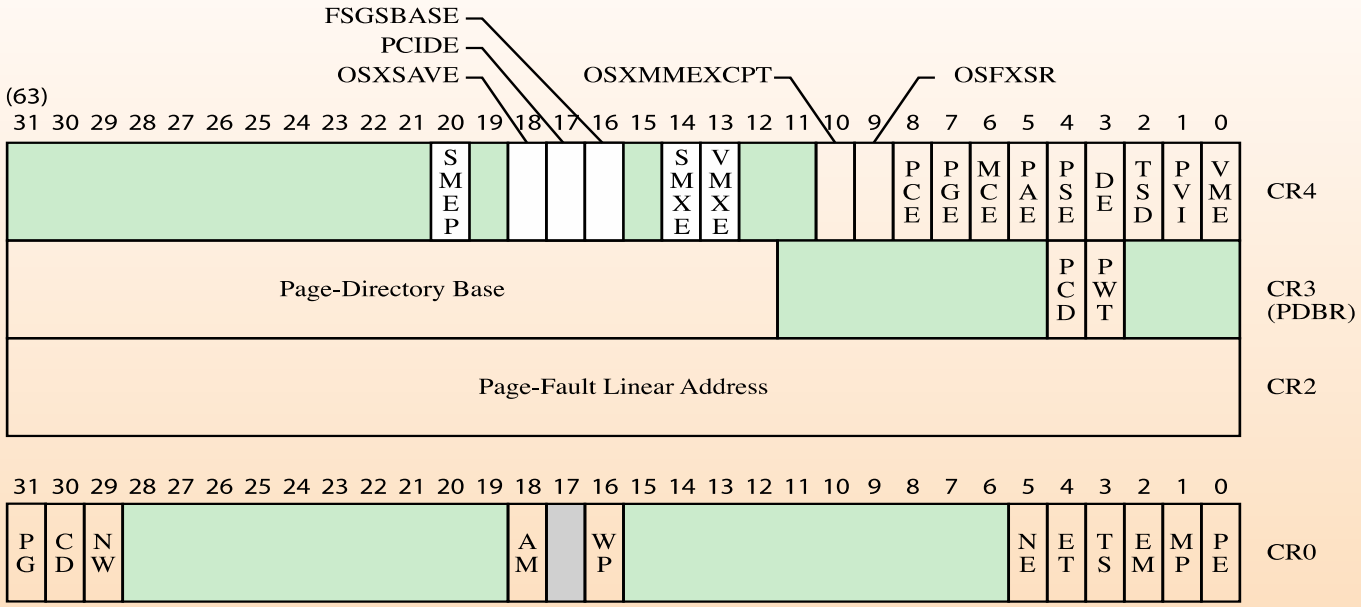


X ID = Identification flag
X VIP = Virtual interrupt pending
X VIF = Virtual interrupt flag
X AC = Alignment check
X VM = Virtual 8086 mode
X RF = Resume flag
X NT = Nested task flag
X IOPL = I/O privilege level
S OF = Overflow flag

C DF = Direction flag
X IF = Interrupt enable flag
X TF = Trap flag
S SF = Sign flag
S ZF = Zero flag
S AF = Auxiliary carry flag
S PF = Parity flag
S CF = Carry flag

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Shaded bits are reserved

Figure 14.22 x86 EFLAGS Register



shaded area indicates reserved bits

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

Figure 14.23 x86 Control Registers

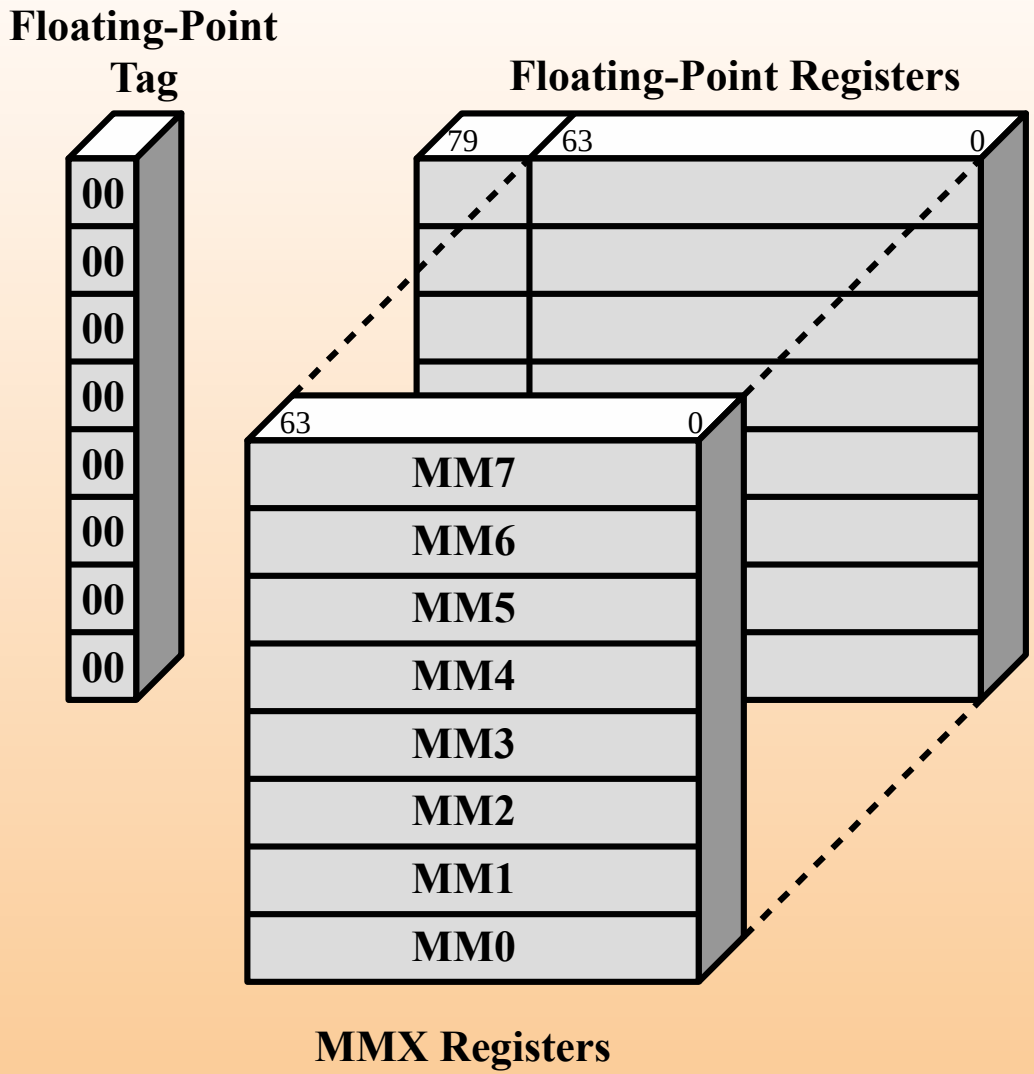


Figure 14.24 Mapping of MMX Registers to Floating-Point Registers

Interrupt Processing

Interrupts and Exceptions

- Interrupts
  - Generated by a signal from hardware and it may occur at random times during the execution of a program
  - Maskable (. The processor does not recognize a maskable interrupt unless the interrupt enable flag (IF) is set.)
  - Nonmaskable (Recognition of such interrupts cannot be prevented.)
- Exceptions
  - Generated from software and is provoked by the execution of an instruction
  - Processor detected
  - Programmed
- Interrupt vector table
  - Every type of interrupt is assigned a number
  - Number is used to index into the interrupt vector table

Vector Number	Description
0	Divide error; division overflow or division by zero
1	Debug exception; includes various faults and traps related to debugging
2	NMI pin interrupt; signal on NMI pin
3	Breakpoint; caused by INT 3 instruction, which is a 1-byte instruction useful for debugging
4	INTO-detected overflow; occurs when the processor executes INTO with the OF flag set
5	BOUND range exceeded; the BOUND instruction compares a register with boundaries stored in memory and generates an interrupt if the contents of the register is out of bounds.
6	Undefined opcode
7	Device not available; attempt to use ESC or WAIT instruction fails due to lack of external device
8	Double fault; two interrupts occur during the same instruction and cannot be handled serially
9	Reserved
10	Invalid task state segment; segment describing a requested task is not initialized or not valid
11	Segment not present; required segment not present
12	Stack fault; limit of stack segment exceeded or stack segment not present
13	General protection; protection violation that does not cause another exception (e.g., writing to a read-only segment)
14	Page fault
15	Reserved
16	Floating-point error; generated by a floating-point arithmetic instruction
17	Alignment check; access to a word stored at an odd byte address or a doubleword stored at an address not a multiple of 4
18	Machine check; model specific
19-31	Reserved
32-255	User interrupt vectors; provided when INTR signal is activated

Table 14.3

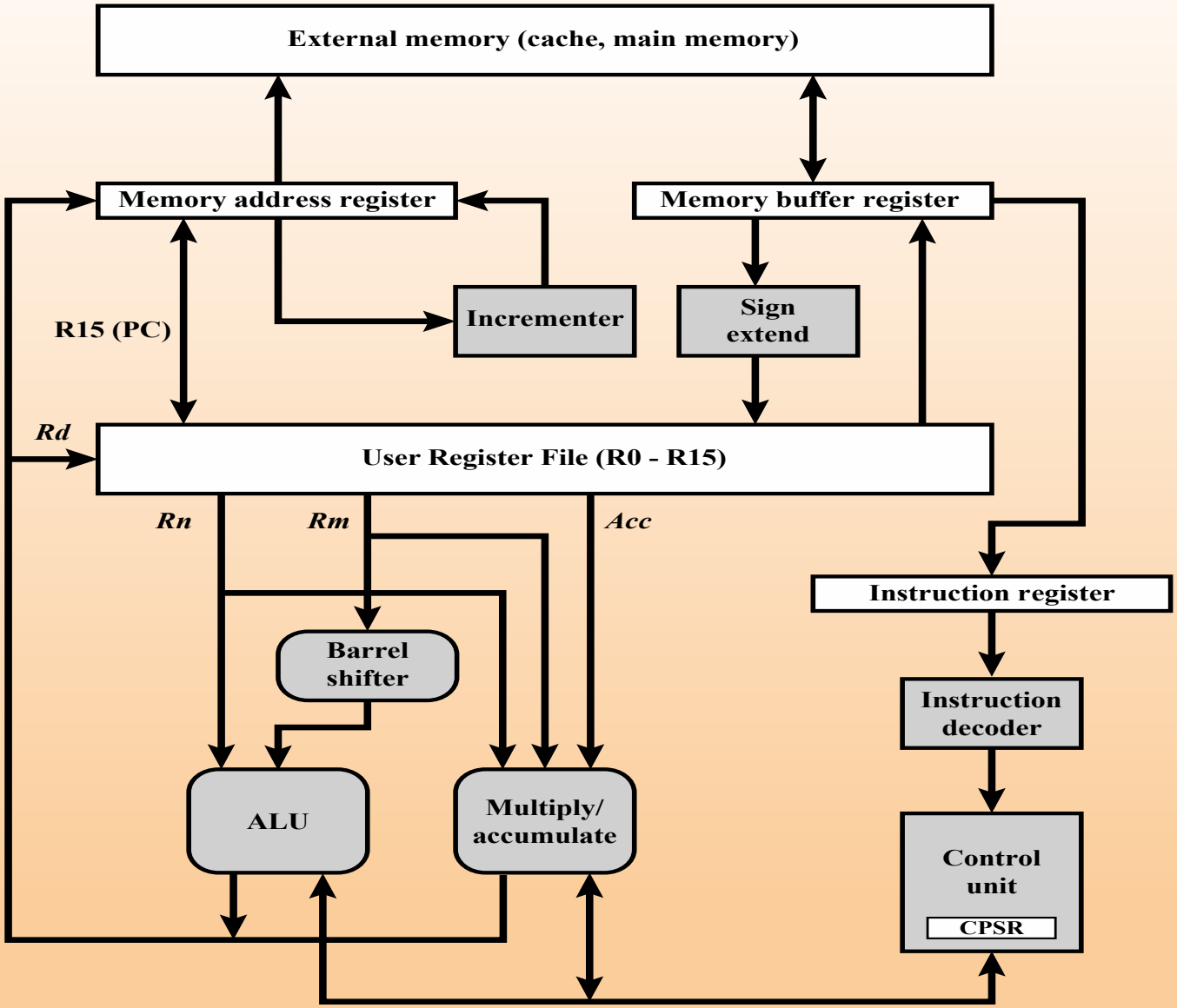
x86 Exception and Interrupt Vector Table

Unshaded: exceptions
Shaded: interrupts

The ARM Processor

ARM is primarily a RISC system with the following attributes:

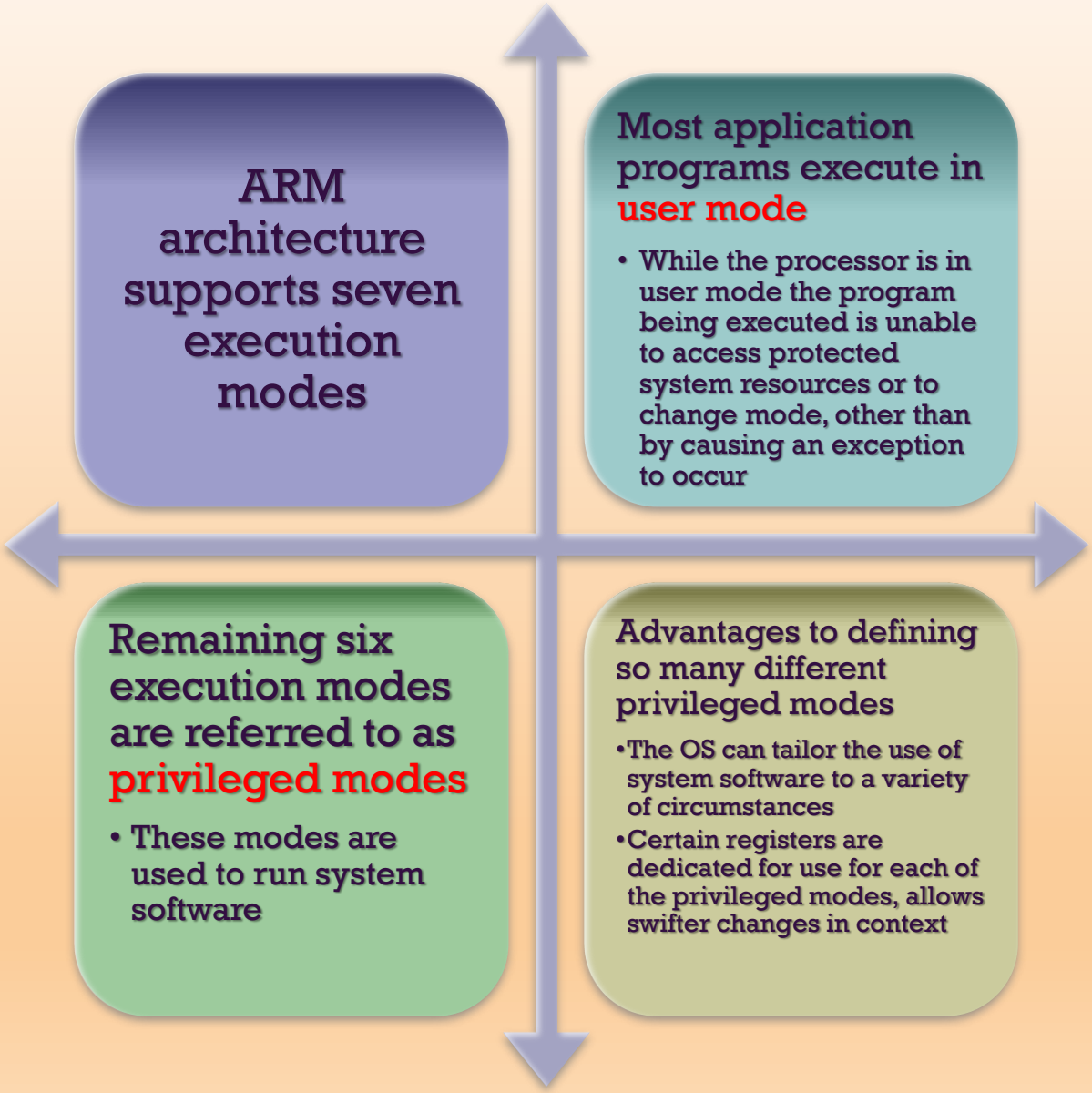
- Moderate array of uniform registers
- A load/store model of data processing in which operations only perform on operands in registers and not directly in memory
- A uniform fixed-length instruction of 32 bits for the standard set and 16 bits for the Thumb instruction set
- Separate arithmetic logic unit (ALU) and shifter units
- A small number of addressing modes with all load/store addresses determined from registers and instruction fields
- Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops
- Conditional execution of instructions minimizes the need for conditional branch instructions, thereby improving pipeline efficiency, because pipeline flushing is reduced



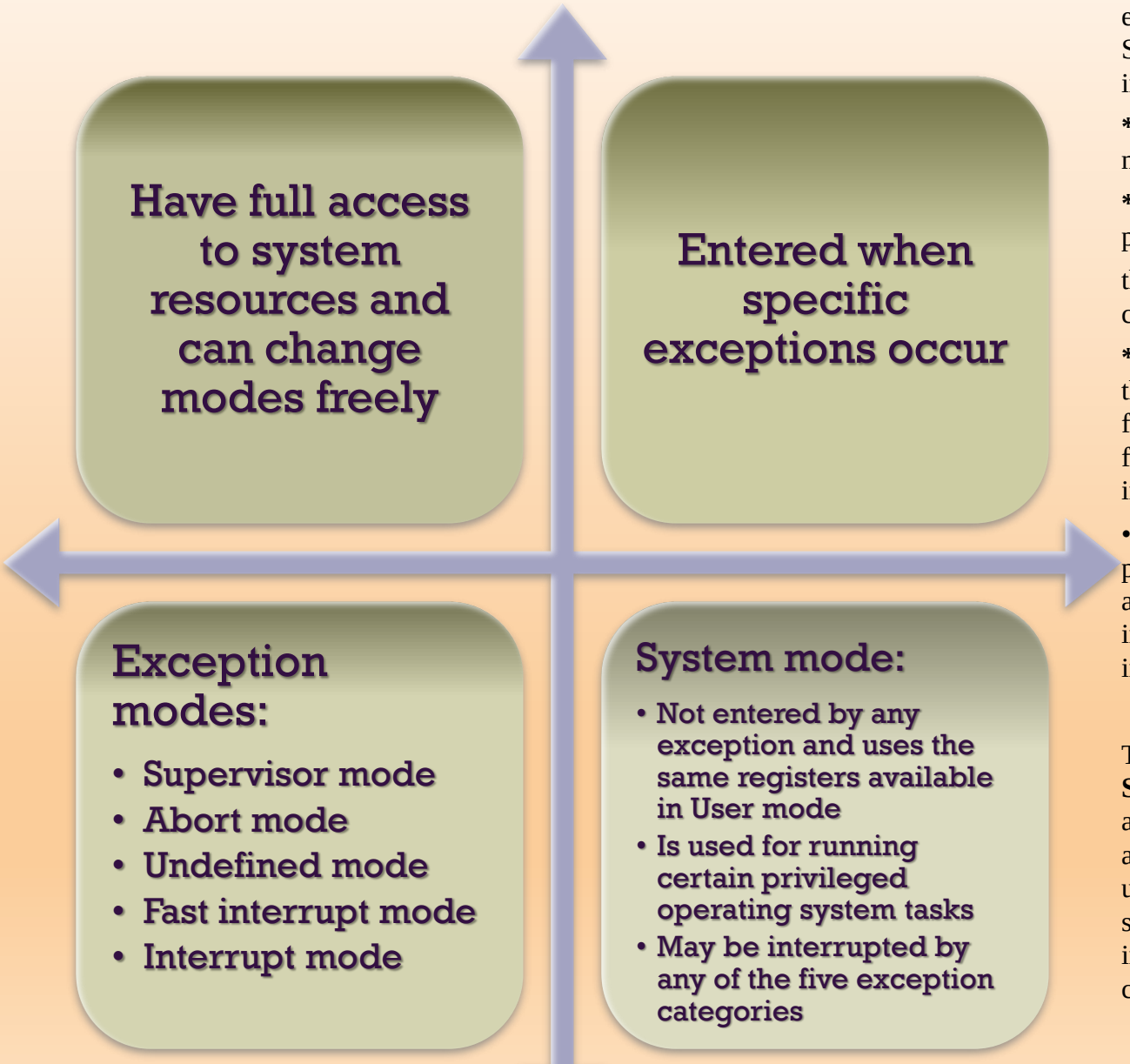
ARM data processing instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. The source register values feed into the ALU or a separate multiply unit that makes use of an additional register to accumulate partial results. The ARM processor also includes a hardware unit that can shift or rotate the *Rm* value before it enters the ALU. This shift or rotate occurs within the cycle time of the instruction and increases the power and flexibility of many data processing operations.

Figure 14.25 Simplified ARM Organization

Processor Modes



Exception Modes



*** Supervisor mode:** Usually what the OS runs in. It is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.

*** Abort mode:** Entered in response to memory faults.

*** Undefined mode:** Entered when the processor attempts to execute an instruction that is supported neither by the main integer core nor by one of the coprocessors.

*** Fast interrupt mode:** Entered whenever the processor receives an interrupt signal from the designated fast interrupt source. A fast interrupt cannot be interrupted, but a fast interrupt may interrupt a normal interrupt.

• **Interrupt mode:** Entered whenever the processor receives an interrupt signal from any other interrupt source (other than fast interrupt). An interrupt may only be interrupted by a fast interrupt.

The remaining privileged mode is the **System mode.** This mode is not entered by any exception and uses the same registers available in User mode. The System mode is used for running certain privileged operating system tasks. System mode tasks may be interrupted by any of the five exception categories.

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13 (SP)	R13 (SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14 (LR)	R14 (LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

SP = stack pointer
LR = link register
PC = program counter

CPSR = current program status register
SPSR = saved program status register

Figure 14.26 ARM Register Organization

The ARM processor has a total of 37 32-bit registers, classified as follows:

• Thirty-one registers referred to in the ARM manual as general-purpose registers. In fact, some of these, such as the program counters, have special purposes.

• Six program status registers.

Registers are arranged in partially overlapping banks, with the current processor mode determining which bank is available. At any time, sixteen numbered registers and one or two program status registers are visible, for a total of 17 or 18 software-visible registers

Registers R0 through R7, register R15 (the program counter) and the current program status register (CPSR) are visible in and shared by all modes.

Registers R8 through R12 are shared by all modes except fast interrupt,which has its own dedicated registers R8_fiq through R12_fiq.

All the exception modes have their own versions of registers R13 and R14.

All the exception modes have a dedicated saved program status register (SPSR)

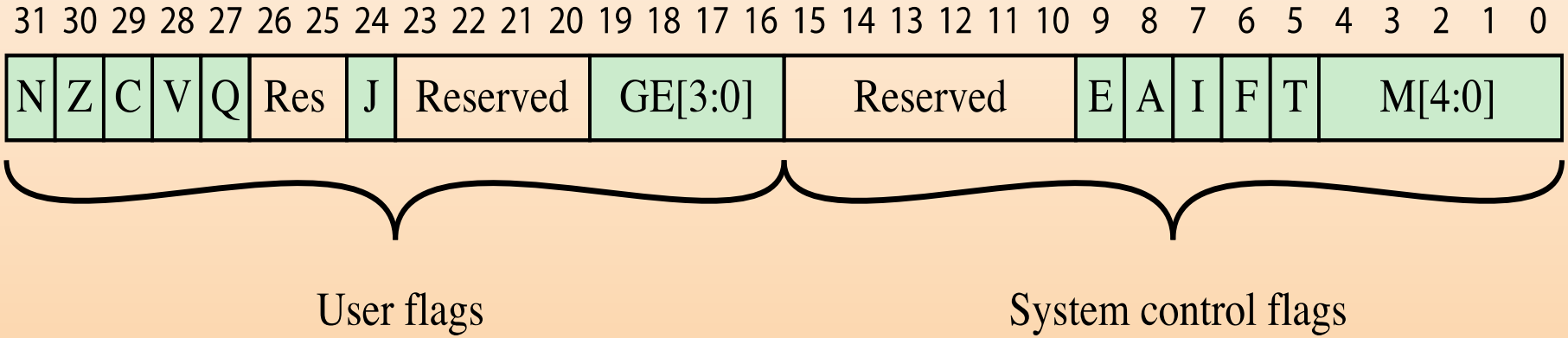


Figure 14.27 Format of ARM CPSR AND SPSR

Exception type	Mode	Normal entry address	Description
Reset	Supervisor	0x00000000	Occurs when the system is initialized.
Data abort	Abort	0x00000010	Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.
FIQ (fast interrupt)	FIQ	0x0000001C	Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.
IRQ (interrupt)	IRQ	0x00000018	Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.
Prefetch abort	Abort	0x0000000C	Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.
Undefined instructions	Undefined	0x00000004	Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.
Software interrupt	Supervisor	0x00000008	Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.

Table 14.4

ARM Interrupt Vector

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
- The Arm processor
  - Processor organization
  - Processor modes
  - Register organization
  - Interrupt processing