CSE 4305
Computer Organization and Architecture

Processor Structure & Function

Course Teacher: Md. Hamjajul Ashmafee

Assistant Professor, CSE, IUT

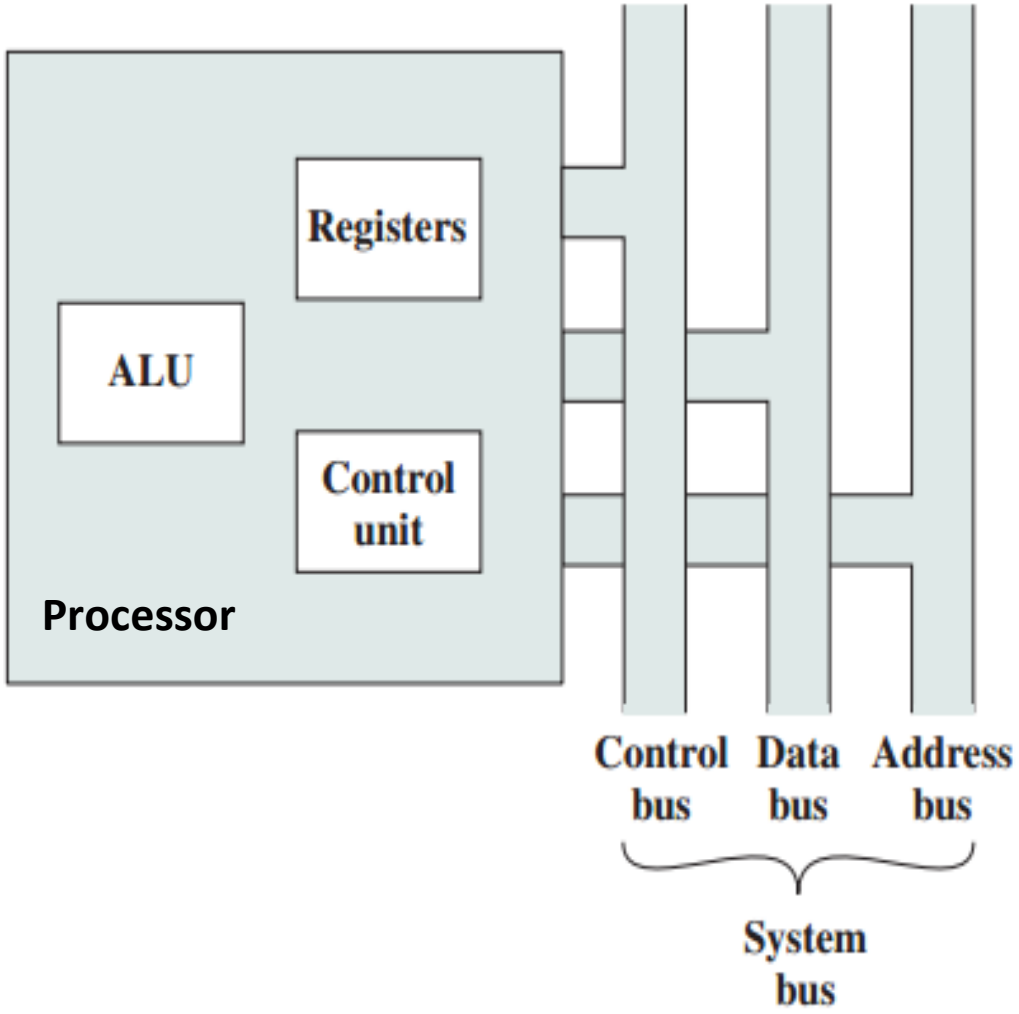
Email: ashmafee@iut-dhaka.edu

Processor's Functionalities

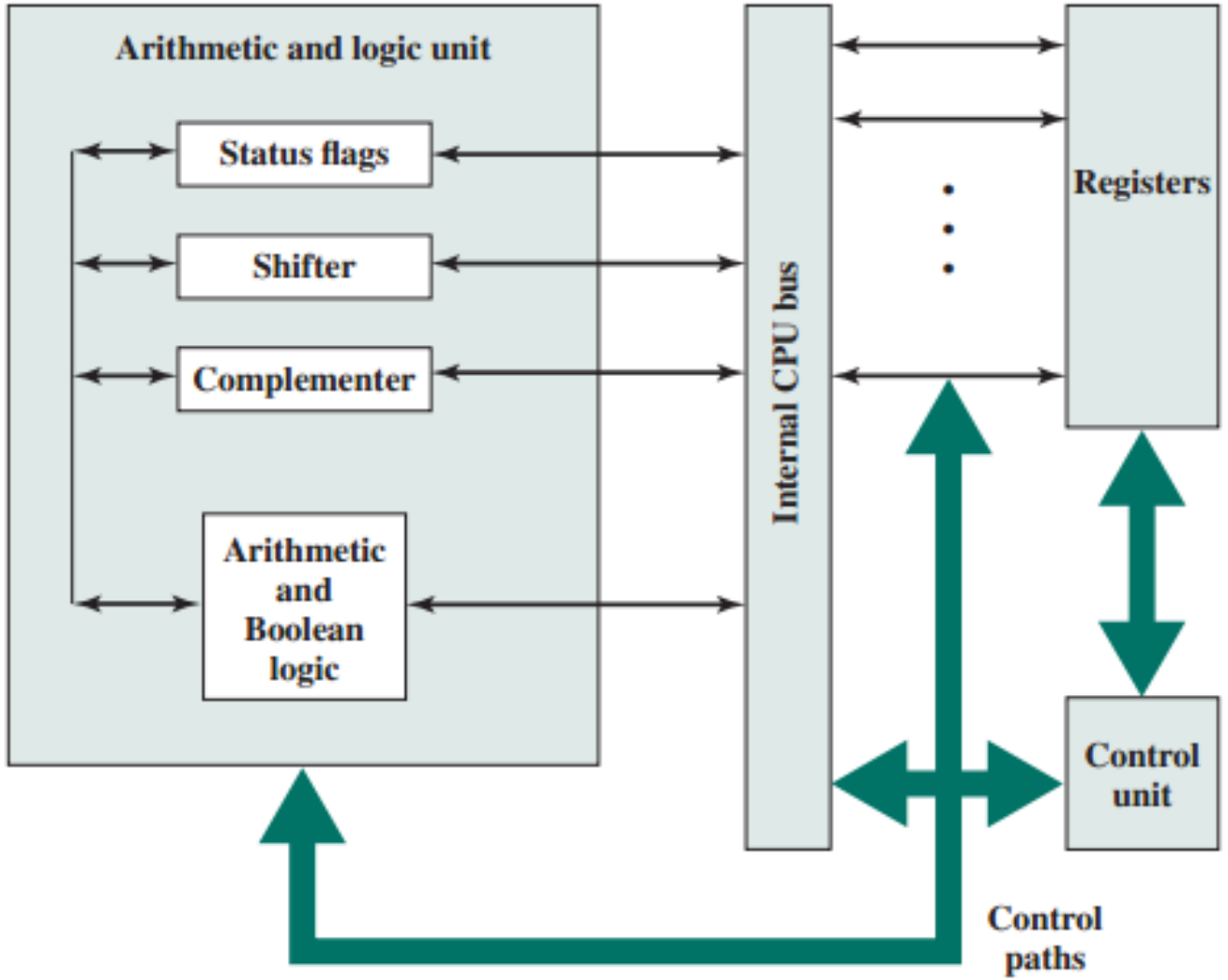
- **Fetch Instruction**
  - Processor reads an instruction from the memory like register, cache or main memory
- **Interpret Instruction**
  - Instruction is decoded to determine the required task
- **Fetch Data**
  - Execution of an instruction may require to read data from memory or I/O devices
- **Process Data**
  - Execution of an instruction may require arithmetic or logical operation on data
- **Write Data**
  - Result of an execution may require to write data to memory or I/O devices

Here it is obvious that processor needs a small **internal memory** to store data, instruction & other information temporarily.

Processor with rest of the system



Detailed View of a Processor



Registers in Processor

- At the top level of the memory hierarchy – **Registers** in processor
  - Faster, smaller, and most expensive per bit
- Two roles of registers:
  1. **User Visible Register:**
     - Programmer can use them to reduce memory references during programming and make the optimized use of the registers
  2. **Control and Status Register**:
     - Used by **control unit** to manage the operations of the processor
     - Also used by **OS** to supervise the execution of programs

  Note: there is no clean separation of registers into these two categories – machine to machine they may vary
     - Sometimes same register may act as both type (e.g., **PC**)

User Visible Register

- User visible registers are those which may be **referenced** by the means of machine instructions
    - **Programmer** can address those registers in their codes
- User visible registers can be categorized as:
    - General Purpose
    - Data
    - Address
    - Condition Codes

Issues of User Visible Register

- There is a **dilemma** between **specializing** (narrowing down) the use of those user visible registers and keeping them **generalized**
  - If we specialize their use, mode of the registers implicit depends on the operation – saves mode bits
    - Also, set of a specific registers will become smaller – also saves bits in register reference
  - But left less flexibility for the programmer
- **Number of registers** – [8~32] registers are optimum
  - But if we use more and more registers, more bits are required in register addressing field of any instruction
  - Another **dilemma** is that less number of registers result in more memory references but more number of registers doesn't reduce memory references noticeably
- **Length of register** – at least long enough to store the **largest address**
  - Sometimes two contagious registers could be used to make the length double

More about User Visible Register

- During subroutine call, all user visible registers are automatically stored on the stack (***context switching***)
    - They are saved and restored by the processor automatically as a part of execution of ***call*** and ***return*** instructions
        - Sometimes it is the responsibility of the programmer to mention specific instructions in this purpose
    - So each subroutine can now use these user visible registers independently

Control and Status Register

Essential control registers for Instruction Execution

- **Program Counter (PC)**
  - Contains the address of an instruction to be fetched very next
  - A ***BRANCH, CALL*** or ***SKIP*** instruction may modify the content of PC
- **Instruction Register (IR)**
  - Contains the instruction most recently fetched where opcode and operand specifiers are analyzed
- **Memory Address Register (MAR)**
  - Contains the address of a location in memory to exchange data with it
  - In bus organized system, it connects directly to the address bus
- **Memory Buffer Register (MBR)**
  - Contains a word of data to be written to memory or read from it
  - In bus organized system, it connects directly to the data bus

Registers for ALU

- For data processing, ALU must access **its required data** in different registers
- ALU may directly access to **MBR** and other user visible registers
  - **Alternatively,** there may be some other **buffer registers** at the boundary of ALU to serve as input and output for ALU and to exchange data with those registers

Status Registers or Flags

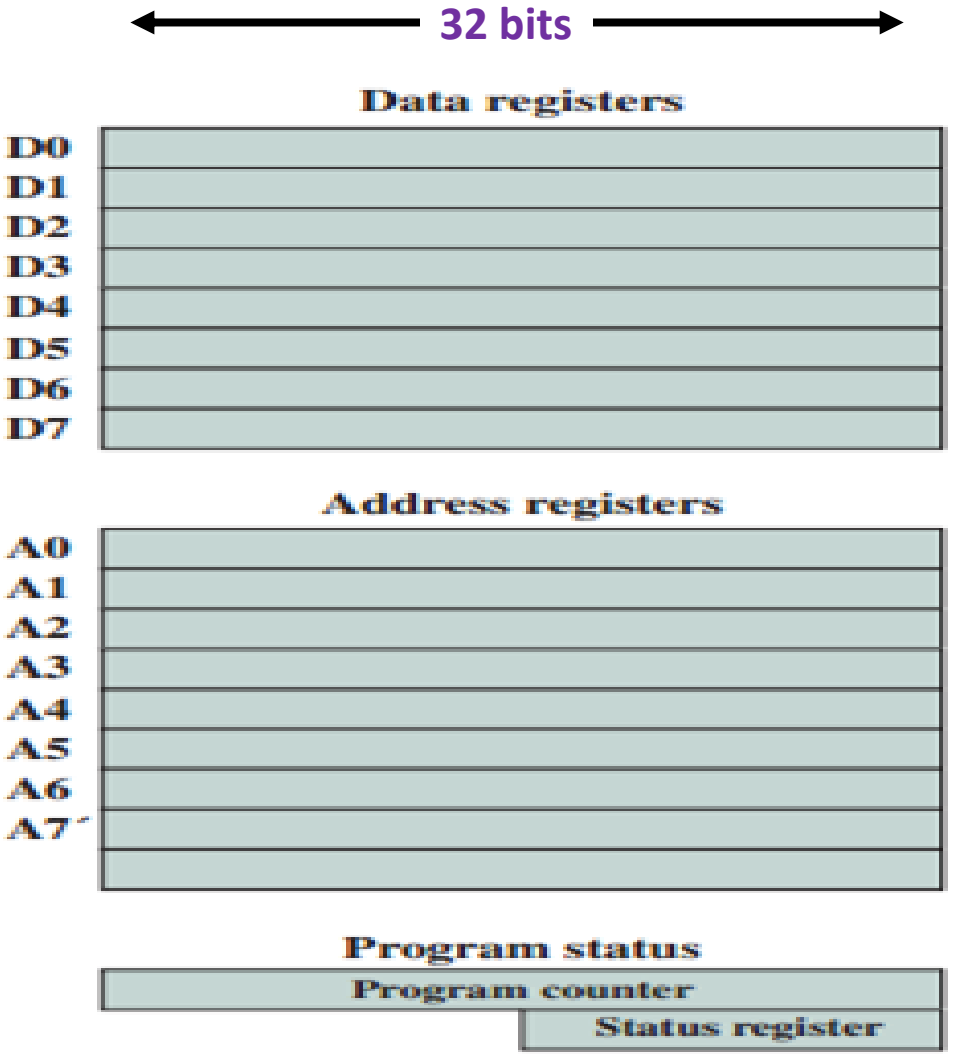
- A register or a set of registers that contains the **status information** of the processor
  - Also known as ***program status word (PSW)*** **or** ***flags***
- Common flags:
    - Sign
    - Zero
    - Carry
    - Equal
    - Overflow
    - Interrupt enable/disable
    - Supervisor – status of a processor itself which is in execution mode currently

Other Control and Status Register

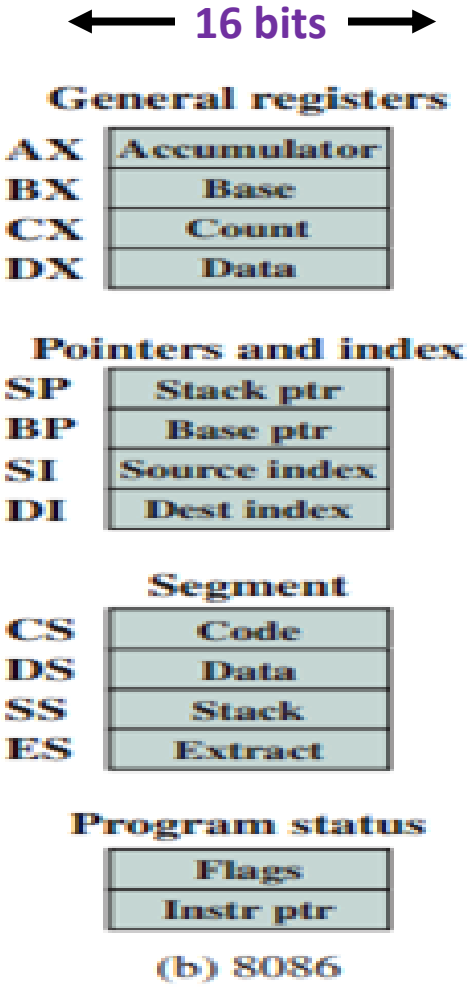
- Pointer to the PCB
  - PCB contains other status information regarding the process in memory
- Interrupt Vector Register
  - To save the location of the system's interrupt vector (address table of all interrupt handlers)
- Stack Pointer
  - For system stack in memory
- Page table pointer
  - Used in virtual memory system to point the addresses of the frames where data is stored physically
- I/OBR and I/OAR
  - For I/O operations

Examples of Register Organizations

Comparison Between Two 16-bit Microprocessors



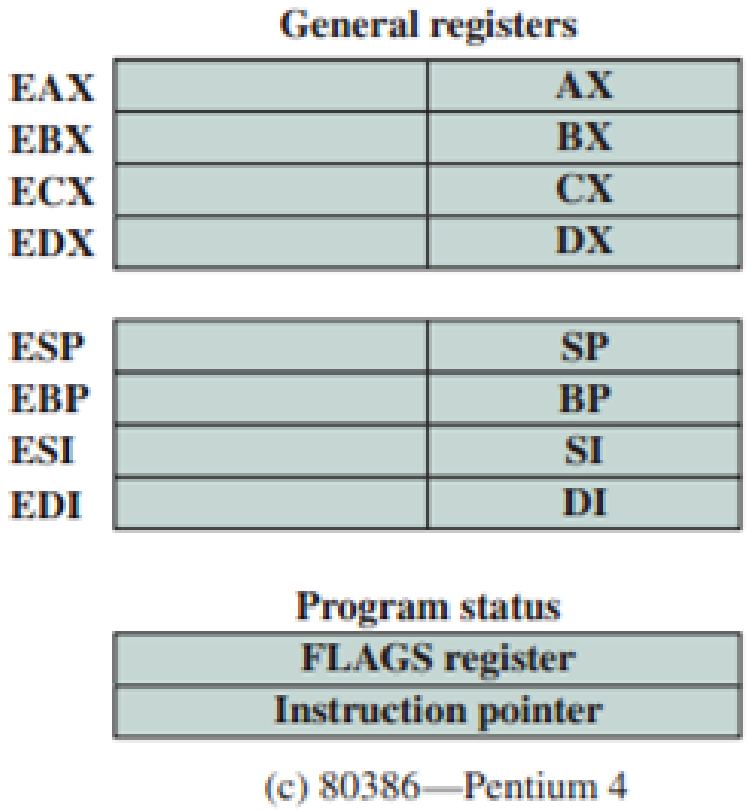
(a) MC68000



(b) 8086

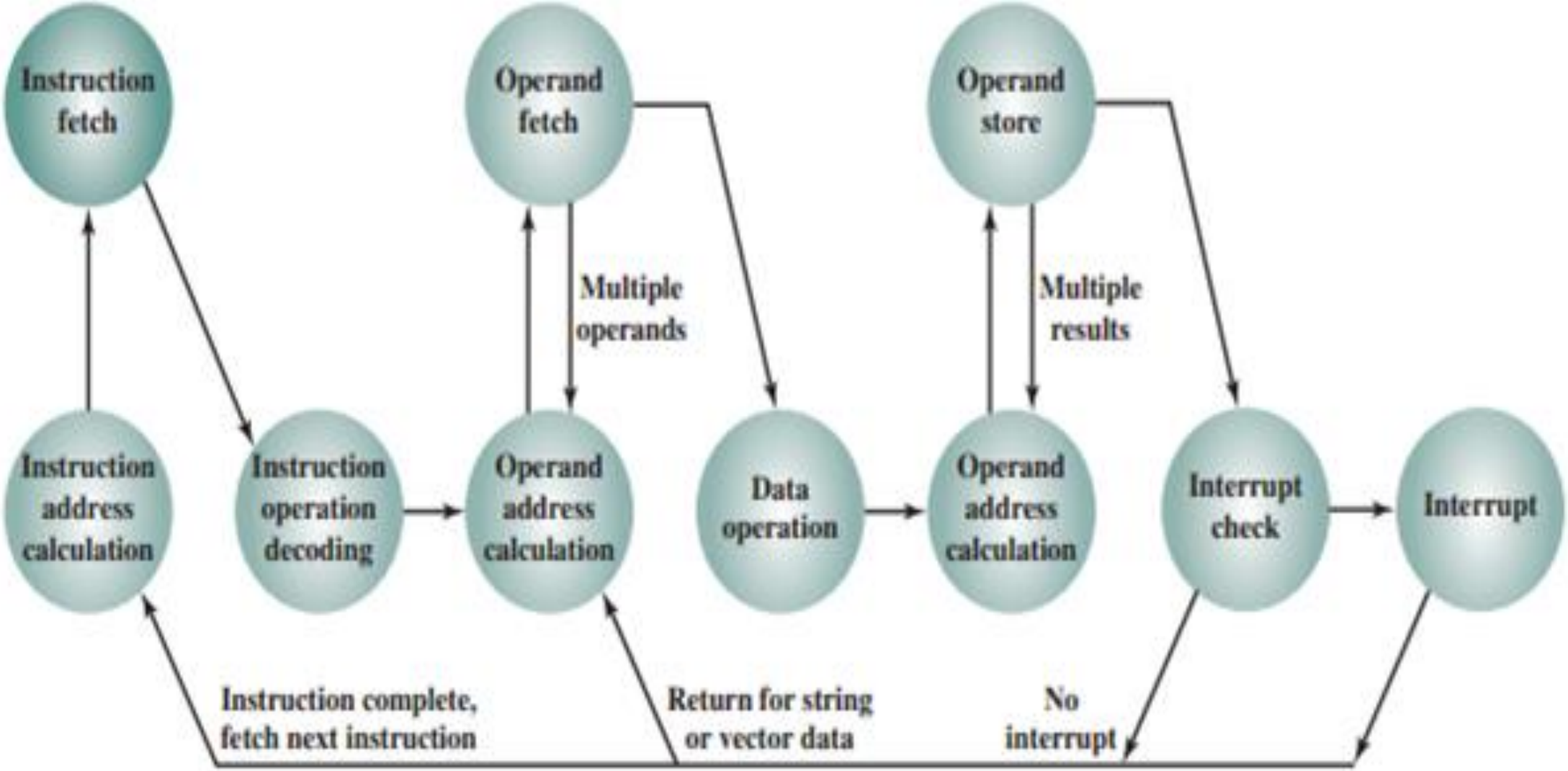
Examples of Register Organizations

Intel 80386 – A 32-bit Microprocessor



(c) 80386—Pentium 4

Instruction Cycle



Instruction Cycle: Explained

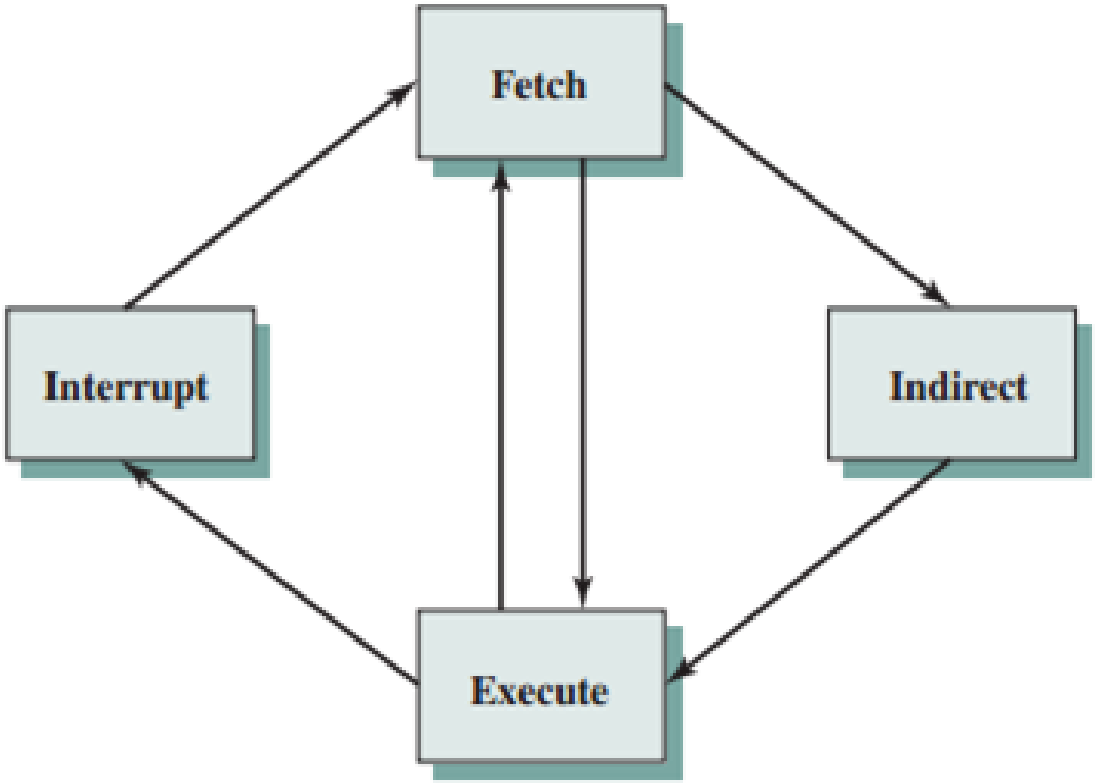
- An instruction cycle includes following stages:
  - **Fetch** – Read the next instruction from memory into the processor
  - **Execute** – Interpret the opcode and perform the indicated operation
  - **Interrupt** – If an interrupt has occurred, serve that interrupt saving current process state
- **Another intermediate stage** during the instruction cycle – **Indirect Cycle**
  - To access memory **once more** to **fetch actual operands** during indirect addressing mode

Indirect Cycle: Explained

- Execution of an instruction may involves one or more operands **in memory** (e.g. ADD REG1, ADDR1, ADDR2…)
  - Each of those operands requires a memory access
  - If any of them is in indirect addressing mode, an additional memory access is required
- Fetching these indirect addresses regarded as **one more instruction-stage** or **sub-cycle** – **Indirect cycle**
- **Revised activity line (roadmap):**
  1. At first instruction is fetched from memory into the processor
  2. It is examined to determine if any **indirect addressing** involved. If so, the required operands are fetched using indirect addressing
  3. Interpret the opcode and perform the execution
  4. If there is any interrupt, it should be served by the processor

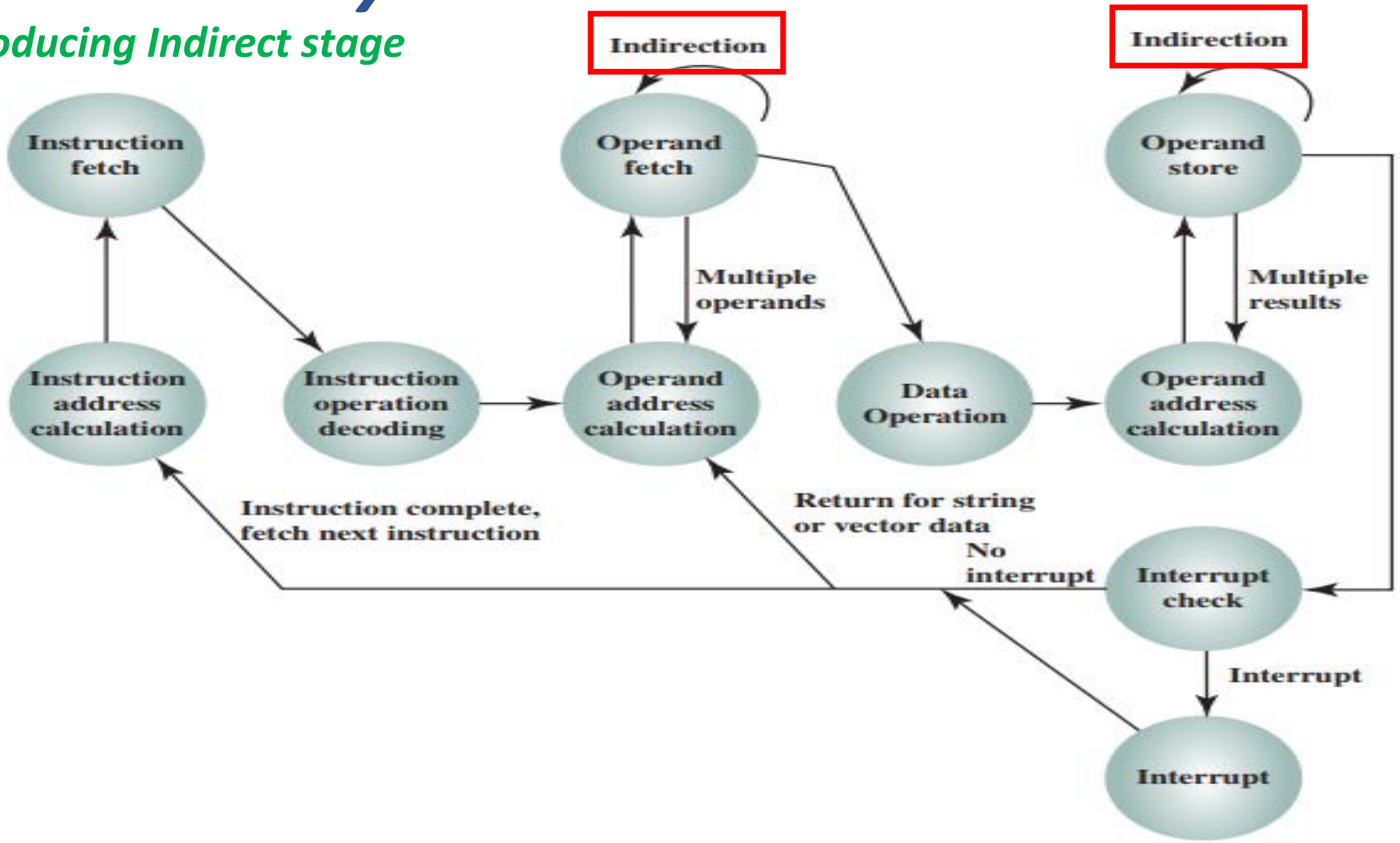
Instruction Cycle: Revised

Introducing Indirect stage



Instruction Cycle: Revised

Introducing Indirect stage



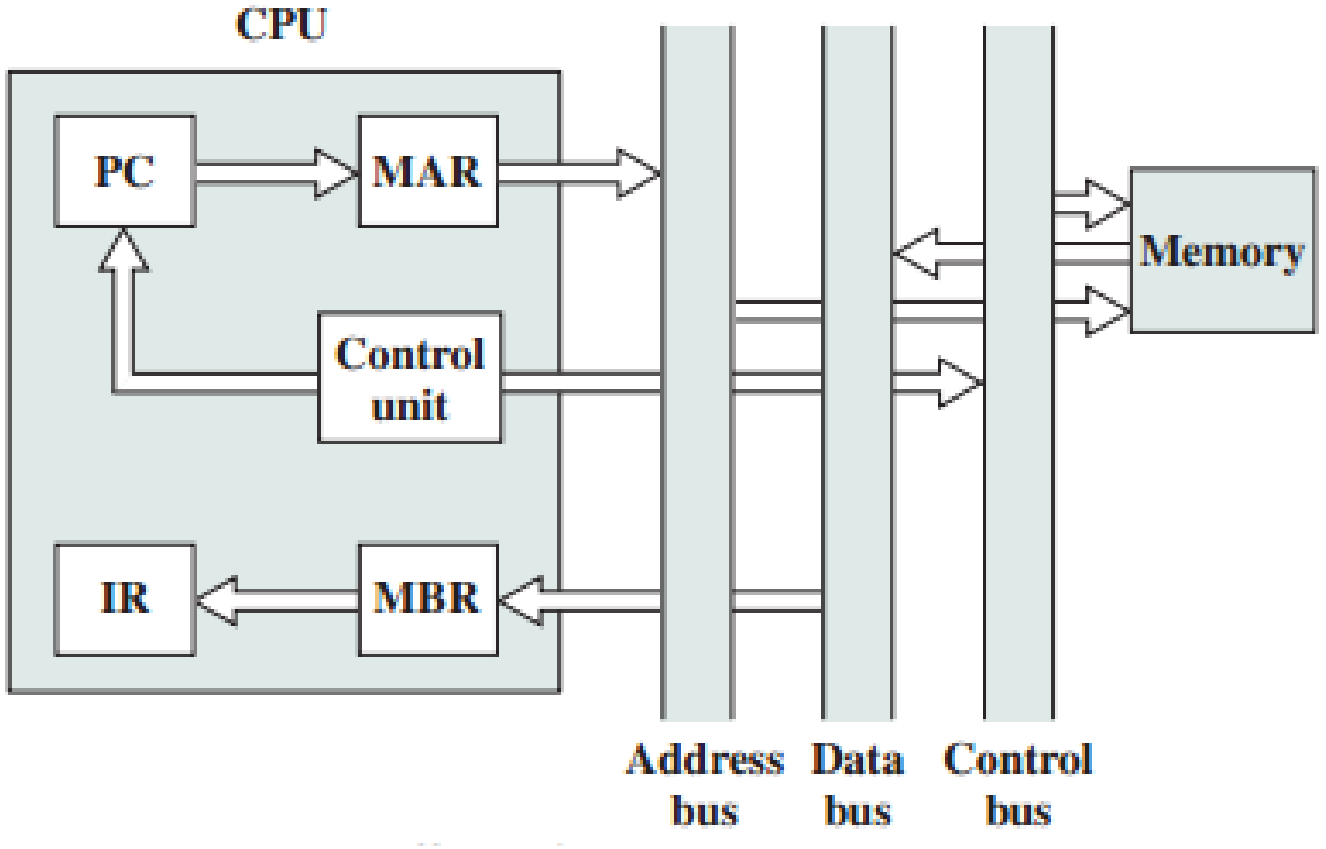
Data Flow

- **What is data-flow in instruction execution?**
  - The **EXACT SEQUENCE** of the stages (or events) during an instruction cycle depends on specific design/architecture of the processor utilizing following registers:
    - **MAR**
    - **MBR**
    - **PC**
    - **IR**

Fetch Cycle: Details in Sequence

- **Fetch Cycle** involves **reading an instruction** from memory according to the following steps:
    1. The **PC** contains the address of the next instruction to be fetched.
    2. This address from **PC** is moved to the **MAR** and placed on the **address bus**.
    3. The control unit requests a **memory read** signal, and the result is placed on the **data bus**
    4. That data from data bus copied into the **MBR** and then moved to the **IR**.
    5. Meanwhile, the PC is incremented by 1, preparatory for the next fetch cycle.
- Once the fetch cycle is completed, the control unit examines the ***IR*** to determine if there is any **operand specifier** using indirect addressing or not.
    - If so, **indirect cycle** is performed.

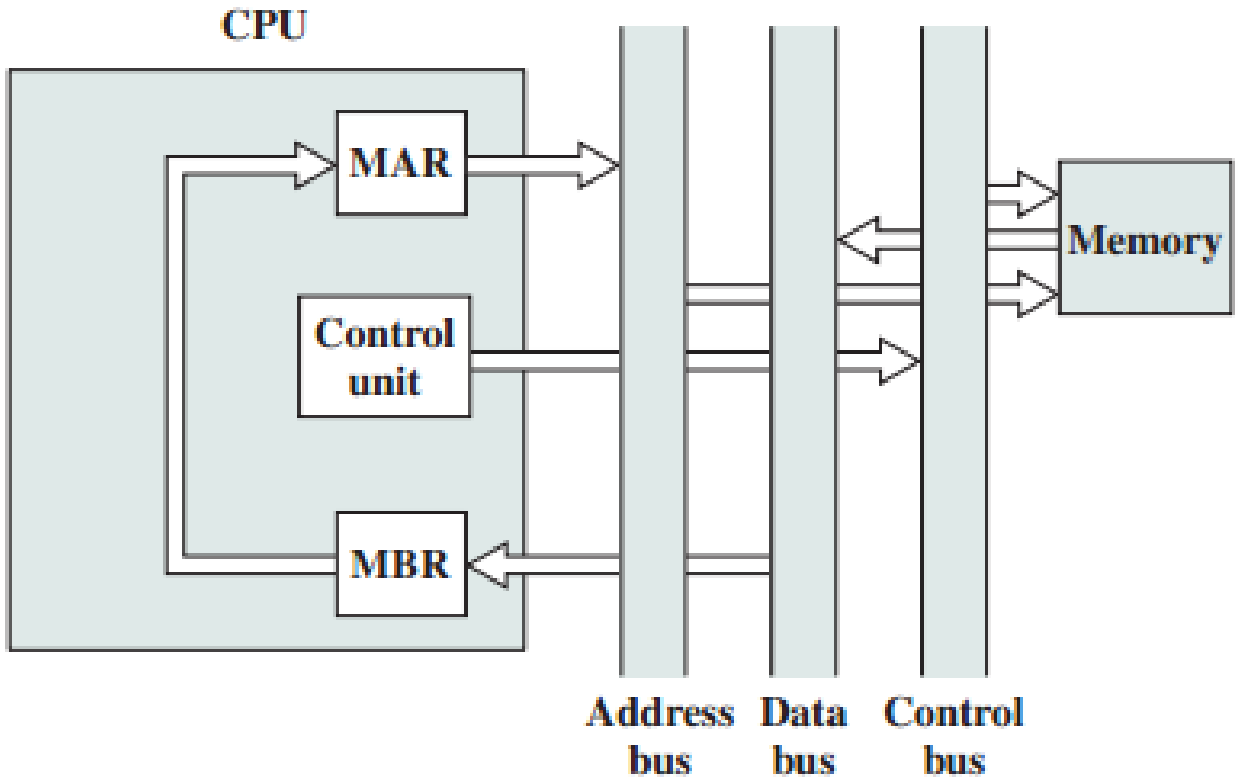
Fetch Cycle: Data Flow



Indirect Cycle: Details in Sequence

- It involves following steps to fetch actual operands:
  - Usually, the **rightmost N bits** of the **MBR** (very immediate fetched instruction from memory), which contain the address reference of the operand, are transferred to the **MAR**.
  - Then the control unit requests **a memory read**, to get the desired (effective) address of the operand into the **MBR**.
    - Based on cascading indirection, MBR content could be transferred **to MAR** for next indirection or **to AC** for data processing

Indirect Cycle: Data Flow



Execute Cycle: Details in Sequence

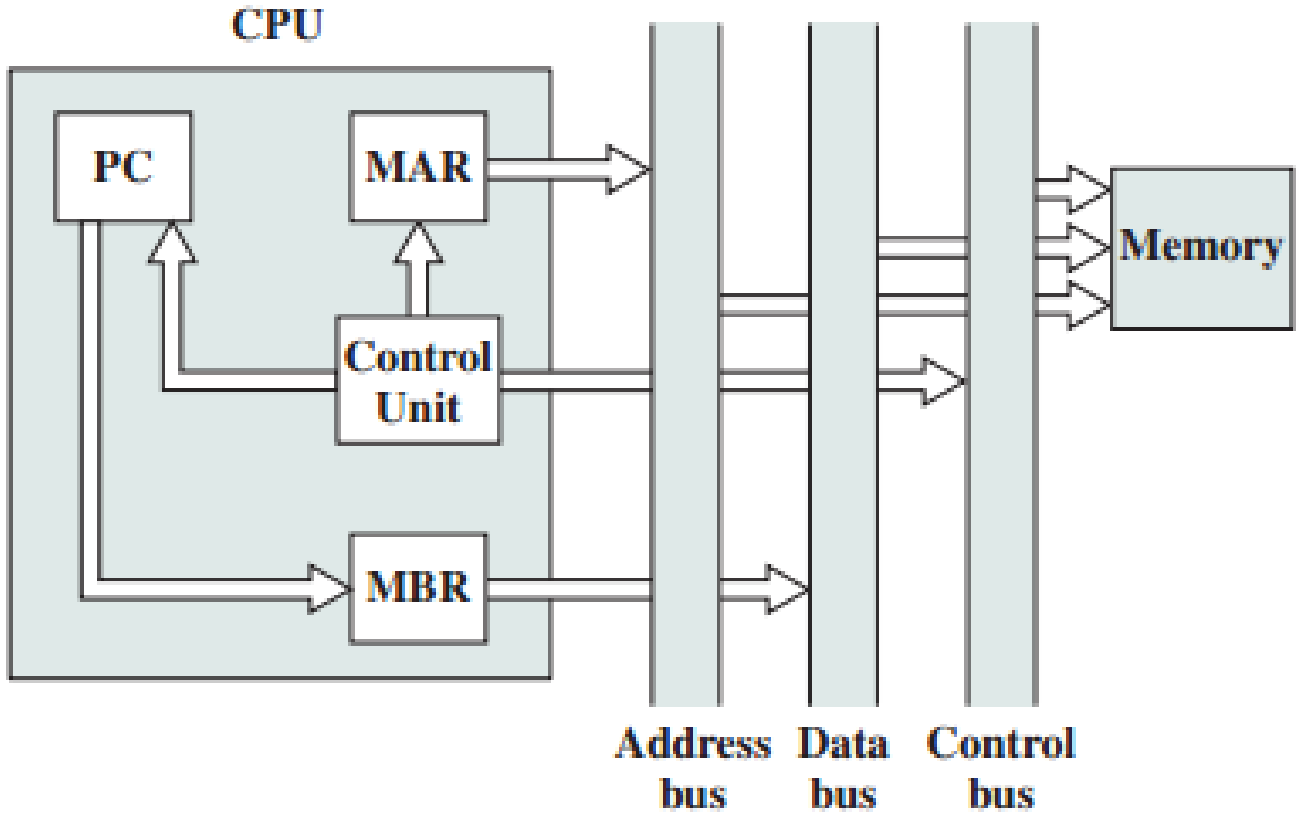
- It may follow **different ways based on machine instructions or operations (opcodes)** in IR
    - Difficult to make any genialized outline
- May involve following operations:
    - **Transfer data** among registers
    - **Read or write** from memory or I/O
    - Command to the **ALU**

.

Interrupt Cycle: Details in Sequence

- It involves:
  - Contents of the **PC** are transferred to the **MBR** to be written into memory **(e.g., in a stack reserved by system)**
    - So that the current program can be resumed after serving the interrupt safely
    - The special **memory location (*e.g.,* stack pointer)** should be **reserved** for this purpose is loaded into the **MAR** from the control unit.
  - The control unit requests a **memory write** signal to save the contents of PC
  - Now, the **PC** is loaded with the **address** of the interrupt service routine (**ISR**) from interrupt vector table
    - As a result, the next instruction cycle can begin by fetching the appropriate instruction from the expected interrupt handler.

Interrupt Cycle: Data Flow



Pipeline Strategy

- To achieve **greater** performance, different improvements in technology along with underlying organization are applied
  - ***e.g.,*** faster circuitry, multiple registers rather than one, cache memory
  - ***Instruction pipelining*** also among those improvements

Pipeline Strategy: In Instruction Execution

- In instruction execution, there are a number of stages – ***e.g.,*** IAC, IF, IOD, OAC, OF, DO, OS ….
- In **two-stage pipeline**, an instruction execution can be subdivided of equal durations into:
    1. **Fetch instruction**
    2. **Execute instruction**

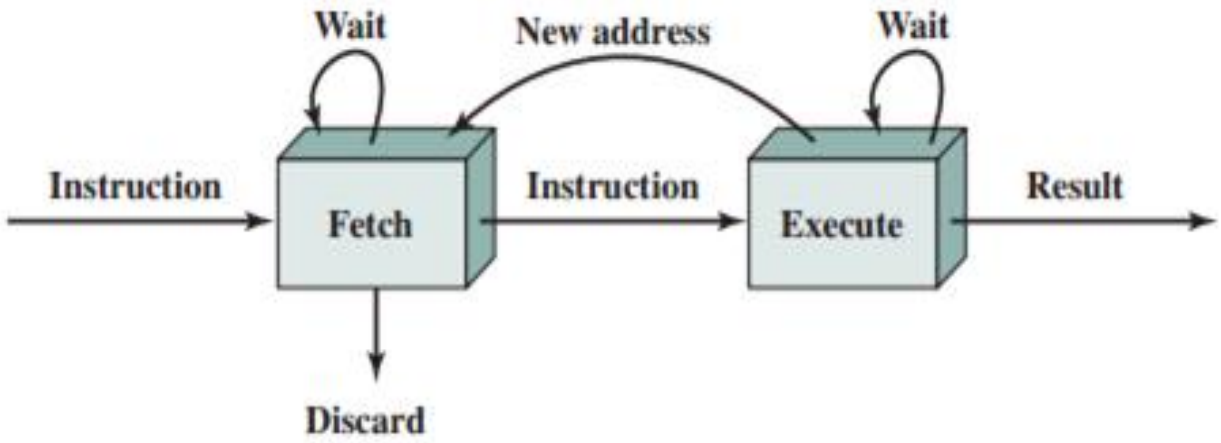
(a) Simplified view

Pipeline Strategy: Speed Up

- Using two-stage pipeline approach, the instruction cycle time (***T***) will be halved, execution rate (***R***) will be doubled (ultimate speed up)
    - Through fetching next instruction during executing the current one in parallel
        - Also known as instruction **prefetch** or **fetch overlap**
        - Always **more registers** are required for **buffering** in between two stages

But Things Are Not Always Ideal!!!

- There is a **little chance** to achieve the best speed up because:
  - The execution time will take longer than fetch time as execution stage involves *reading and storing operands*, *operation performance*, *storing back the results*
  - A conditional branch instruction makes the address of the next instruction **unknown**



More Amendments to Speed up

- To increase the **speedup** (increase the execution rate), pipeline must have more stages like followings:
    1. **Fetch Instruction** (FI) – read next instruction into buffer form memory
    2. **Decode Instruction** (DI) – determine the opcode and the operand specifiers
    3. **Calculate Operands** (CO) – effective address calculation
    4. **Fetch Operands** (FO) – fetch operands from memory
    5. **Execute Instruction** (EI) – perform the operation
    6. **Write Operand** (WO) – store the result into memory

  Note: Greater the number of the stages in the pipeline, the better the execution rate

- As the number of stages are increased, these stages will be probably of **equal duration**

6-stage Instruction 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

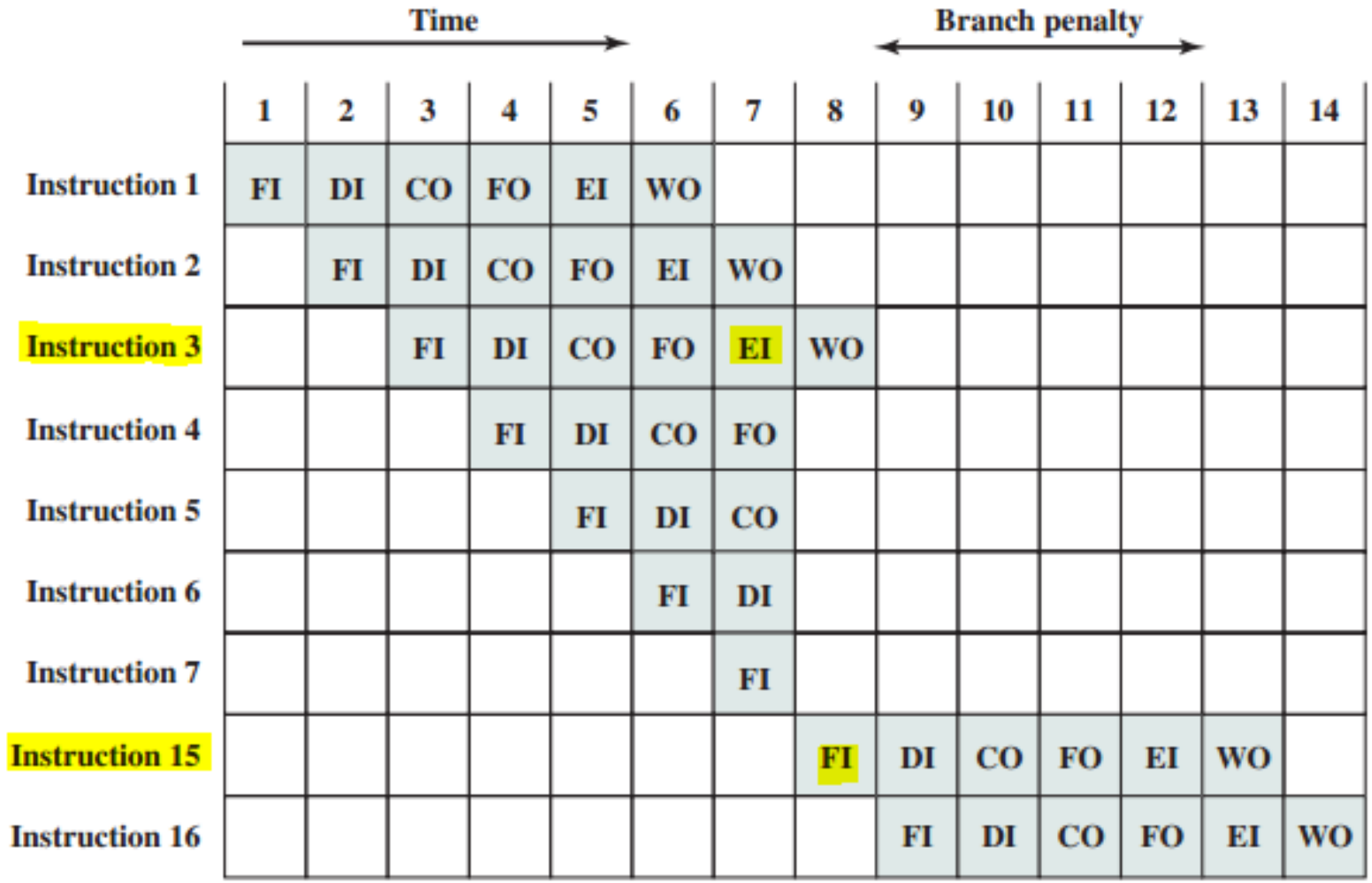
a) Total time for 9 instructions?
   i. Before pipeline (54)
   ii. After pipeline (14)
b) Speed up?

Limitations Again!!!

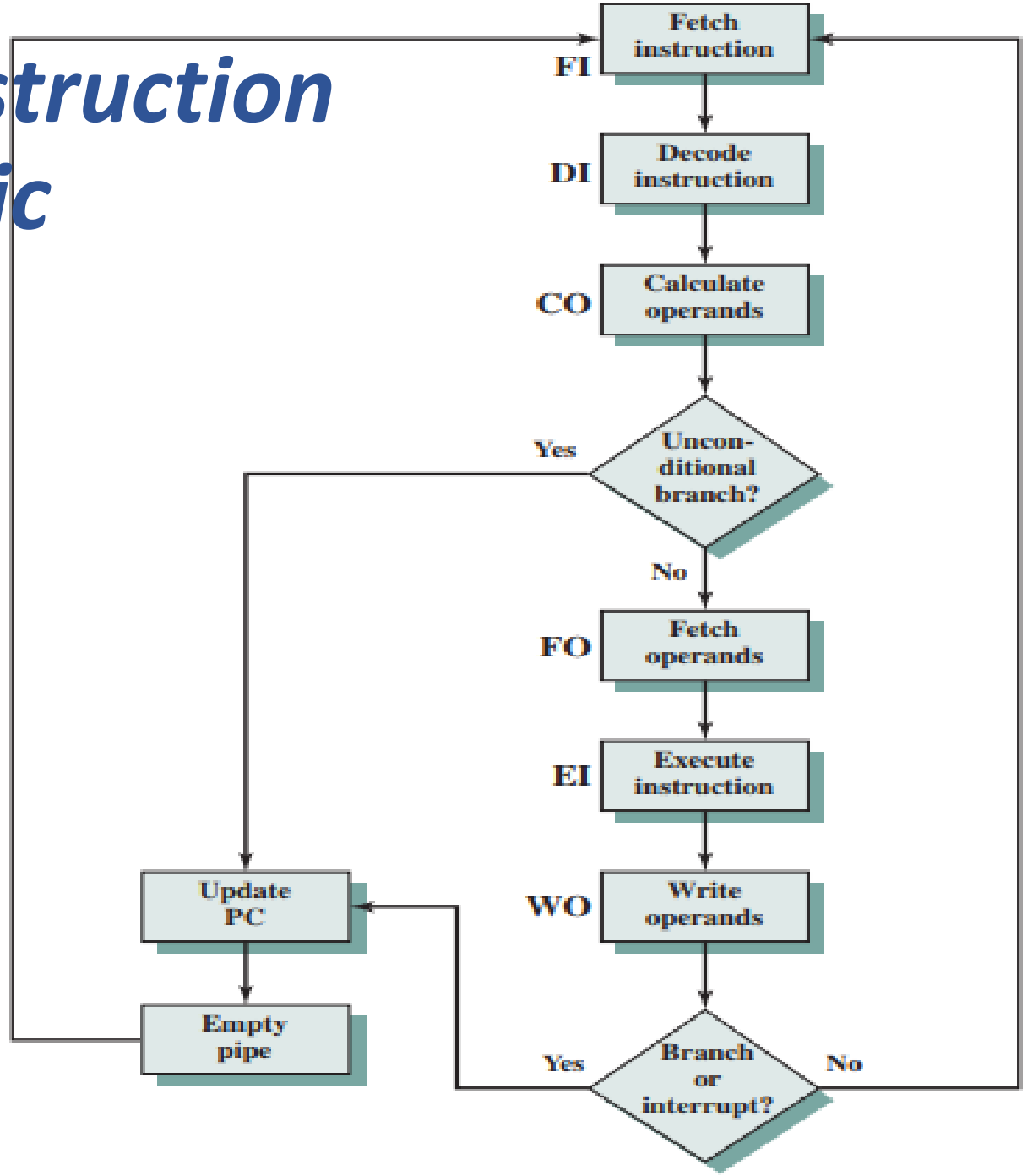
1. Some instructions will not cover all stages of the pipeline
   - ***e.g.,*** **Load instruction** no need to have the WO stage
2. All stages in pipeline are **assumed** to be performed in parallel
3. If the duration of the stages are not equal, it will keep some stages waiting
4. ***Conditional branch*** instruction and ***interrupt*** may invalidate several instruction prefetches
5. **Conflicts** due to **operand dependency** are there during calculating the operands in CO stage

Effect of Conditional Branch in Pipeline

Note: Instruction 3 is a conditional branch to instruction 15

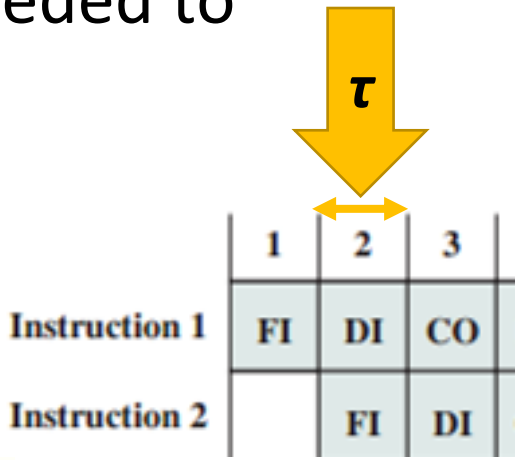


Six-Stage Instruction Pipeline Logic



Pipeline Performance

- The **cycle time (*τ*)** of an instruction pipeline is the time needed to advance a set of instructions **one stage** ahead:



$$\tau = \max_i[\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

$$\tau_m \gg d.$$

τ_i = time delay of the circuitry in the ith stage of the pipeline
τ_m = maximum stage delay (delay through stage which experiences the largest delay)
k = number of stages in the instruction pipeline
d = time delay of a latch, needed to advance signals and data from one stage to the next

Pipeline Performance…

- $T_{k,n}$ is the ***total time*** for a pipeline with ***k stages*** to execute ***n instructions*** considering no conditional branches:

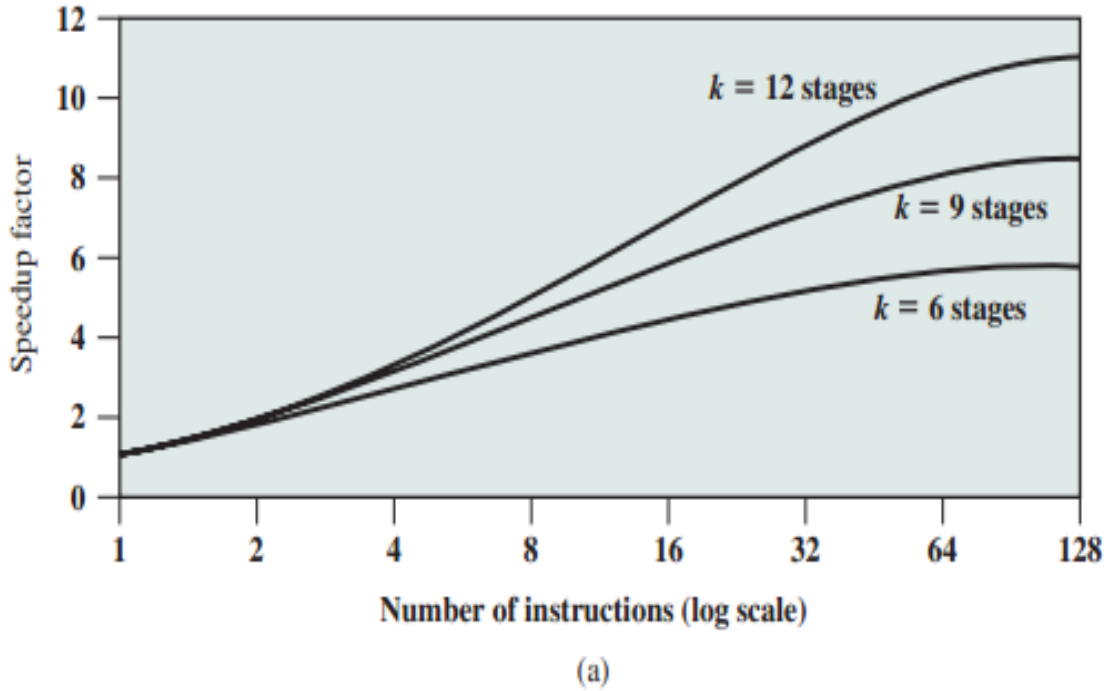
$$T_{k,n} = [k + (n - 1)]\tau$$

- Example: From the previous 6 stage pipeline example, $T_{6,9}$ = 14.
- **Speedup** ***with*** and ***without*** pipeline:

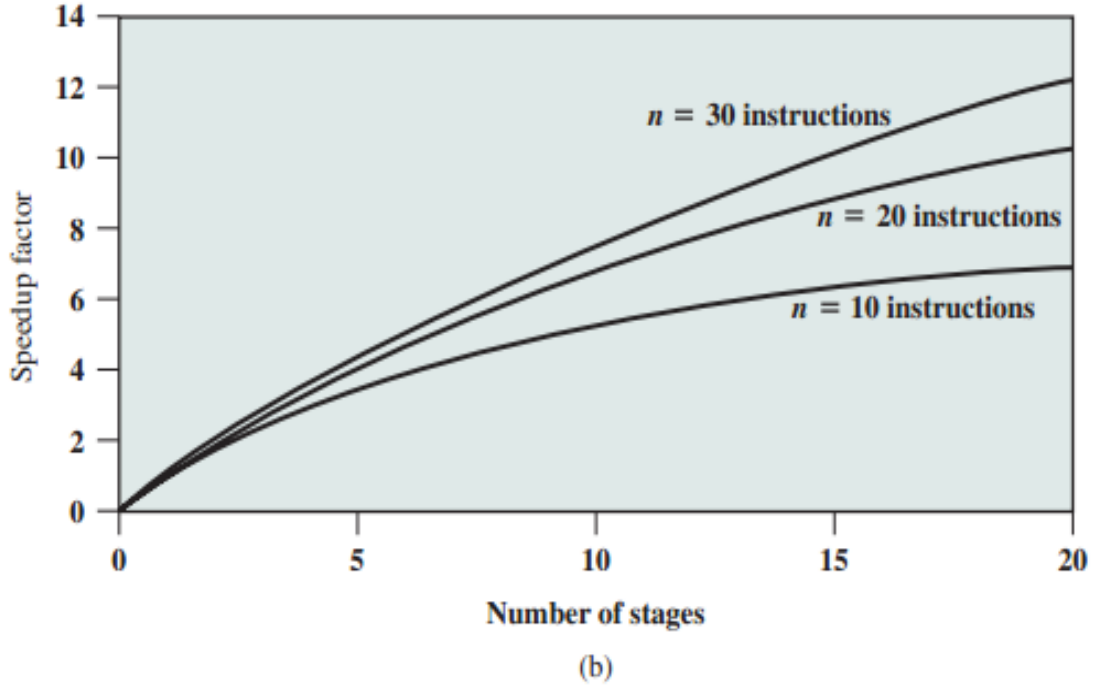
$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

k = number of stages
n = number of instructions
kτ = instruction cycle time

Pipeline Speedup



(a)



(b)

*** With certain constraints

Pipeline Hazards

- **When** the pipeline entirely or some portion of the pipeline must **stall** **(postpone)** because conditions don't permit to continue execution more
  - Also known as ***pipeline bubble***
- **Three** types of hazards:
  1. Resource Hazards
  2. Data Hazards
  3. Control Hazards

Resource Hazards

- When two or more instructions in the pipeline need the same resource
  - As a result, those instructions must be executed in serial rather than in parallel for a portion of the pipeline
- Also known as ***structural hazards***
- Example:
  1. A **main memory** having **a single port** to fetch and store data and instruction
     - Data and instruction read and data write one at a time; **not in parallel**
  2. Multiple instructions are ready for the execute stage but there is a **single ALU**
- **Solution:**
  1. Increase available resources – multiple ports in main memory, multiple ALU units
  2. Reservation Table

Resource Hazards: Solution

- ***Reservation table:*** A way of representing the **task flow pattern** of a pipelined system.

index	time-1	time-2	time-3	time-4	time-5	time-6
resource 1	X		X			X
resource 2		X				
resource 3		X		X		
resource 4					X	

Data Hazards

- When there is a **conflict** to access an operand location
  - Two sequential instructions of a program in a pipeline access to the **same** memory or register reference
    - If they are executed **in strict sequence**, no problem occur. But **in parallel**, they produce different result
    - Producing **incorrect** result because of use of pipelining
- Example of instructions from x86 machine:

```
ADD EAX, EBX /* EAX = EAX + EBX
SUB ECX, EAX /* ECX = ECX - EAX
```

Types of Data Hazards

1. **Read after write (RAW)** or **true dependency**:
2. **Write after read (WAR)** or **anti-dependency**:
3. **Write after write (WAW)** or **output dependency**:

Control Hazards

- **When** the pipeline makes a wrong decision on a branch prediction
  - So based on misprediction, it brings useless instructions into the pipeline that must be discarded (flushed)
- Its **impact** depends on frequency of branch instruction, accuracy of branch prediction, stall penalty of each misprediction
- Also known as ***branch hazard***
- There are several **approaches** to deal with **control hazards** or **branches**:
  1. Multiple streams
  2. Prefetch branch target***
  3. Loop buffer
  4. Branch prediction
  5. Delayed branch

  - **These approaches guarantee a steady flow of instructions to the initial stages of the pipeline**

1. Multiple Streams

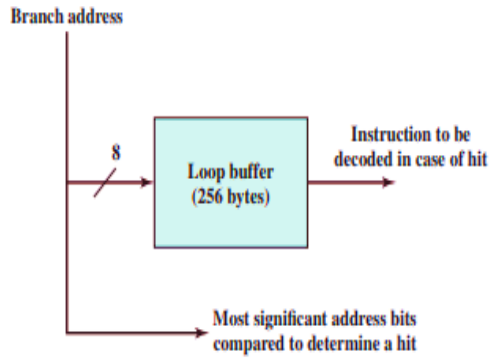
- Instead of single (simple) pipeline, a **brute-force** approach will allow the pipeline to fetch **both** of the instructions replicating the initial portion of the pipeline
    - Allow two different **streams** for both of the choices of a conditional branch to proceed

2. Prefetch Branch Target

- When a conditional branch is recognized, target of the branch is prefetched **along with the instruction** immediately following the branch instruction
  - This target instruction is saved (not executed) until the branch instruction is executed
  - If the branch is taken later on, the target instruction is already available
    - Later instructions will be fetched sequentially

3. Loop Buffer

- A small very high speed **memory**, controlled by **instruction fetch stage** of the pipeline, that **contains** the ***n*** most recently fetched instructions in sequence
    - Similar to a cache dedicated to recently fetched instructions but stores those instructions in sequence and smaller in size and lower in cost
    - Conforming the "**locality of reference**" (basically ***spatial locality***)
- If a branch instruction is to be taken earlier to the current instruction, loop buffer is first checked if the **branch target** is available or not
    - If so, the next instruction is fetched from the buffer (similar to "**cache hit**")



4. Branch Prediction

- Various techniques **to predict** whether a branch will be taken or not:
    i. **Predict (that branch) never taken**
    ii. **Predict (that branch) always taken**
    iii. **Predict (branch) by opcode**
    iv. **Taken – Not taken switch [prediction bit(s)]**
    v. **Branch history table**
  - First three approaches are **static**
    - Not dependent on the **execution history**
  - The latter two are **dynamic**
    - Dependent of the **execution history**

i. Predict "Never Taken"

- Simplest static approach
- Always assume that branch will not be taken
  - Continue to fetch instruction in sequence
  - No update for PC
- Most popular approach

ii. Predict "Always Taken"

- Another simplest static approach
- Always assume that branch will be taken
  - Fetch instruction from the **branch target**
  - Update the **PC** register according to the address of the branch target

iii. Predict by Opcode

- Final static approach
- Decision based on the **opcode** of the branch instruction
  - Processors assumes that branches are takes usually for **certain branch opcodes** and not for others
  - Reports says that **success rate** is more than 75%

Dynamic Branch Prediction Approaches

- Attempt to **improve the accuracy** of prediction by **recording the history** of the conditional branch instructions in a program
- **One or more bits** can be associated with *each conditional branch instruction* that reflect the **history** of these instructions

iv (a). Taken/Not-Taken Single Switch Bit

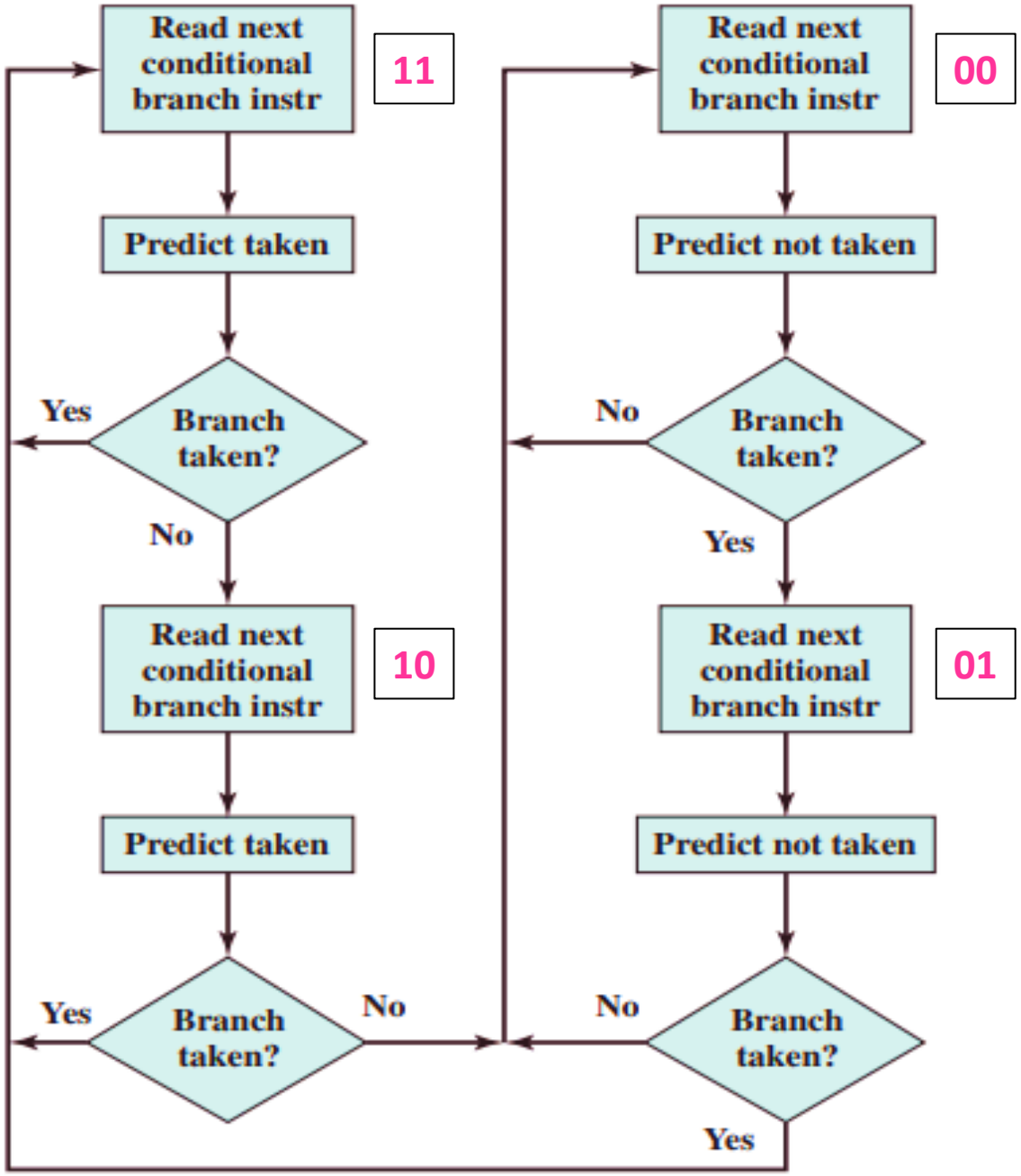
- Records whether the **very last** execution of this type instruction resulted in a branch or not
- **Disadvantage**: Depending solely on this **single** bit history, **an error** during prediction will occur **twice** for each use of loop instruction
    - Once on **entering** the loop and once on **exiting**

iv (b). Taken/Not-Taken Double Switch Bits

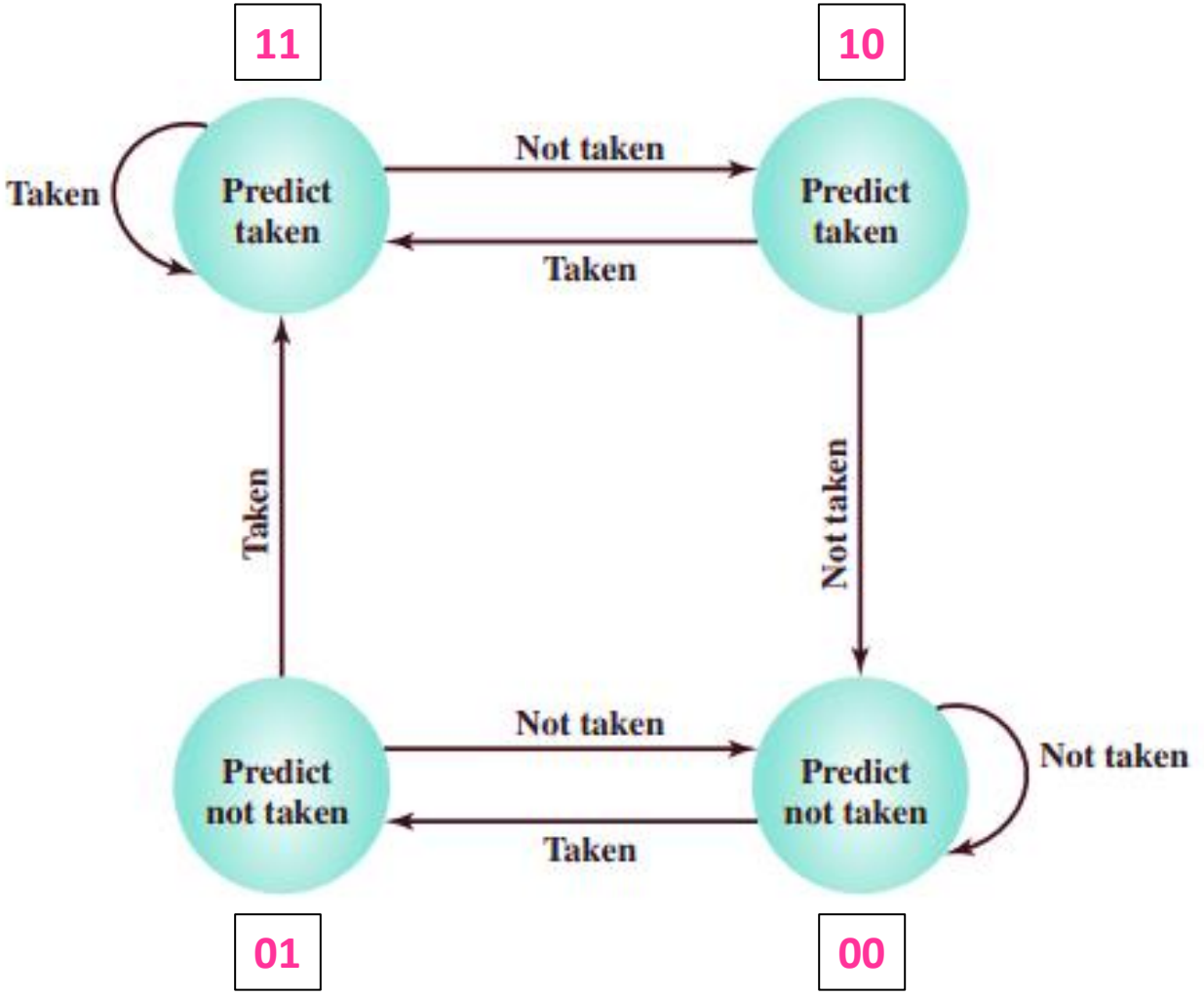
- **Recording** the result of the **last two instances** of the execution of the associated branch instruction
  - Same as recording a particular state in a finite state machine (state diagram)
  - Exemplary algorithm:
    1. Starting with the state "**to be taken**" at the very beginning
    2. As long as the succeeding branch instructions encountered as "**taken**", the decision is **"to be taken**"
    3. If single prediction is wrong ("**not taken**"), still it predicts "**to be taken**" in next state
    4. Only if two successive branches encountered as "**not taken**" will change the decision as "**not to be taken**"
    5. Subsequently, the algorithm will predict "**not to be taken**" until two branches are "**taken**" in a row

Note: This algorithm requires two consecutive wrong predictions to change the decision

Branch Prediction Flowchart



Branch Prediction State Diagram



Drawback of History Bit(s)

- Self Study

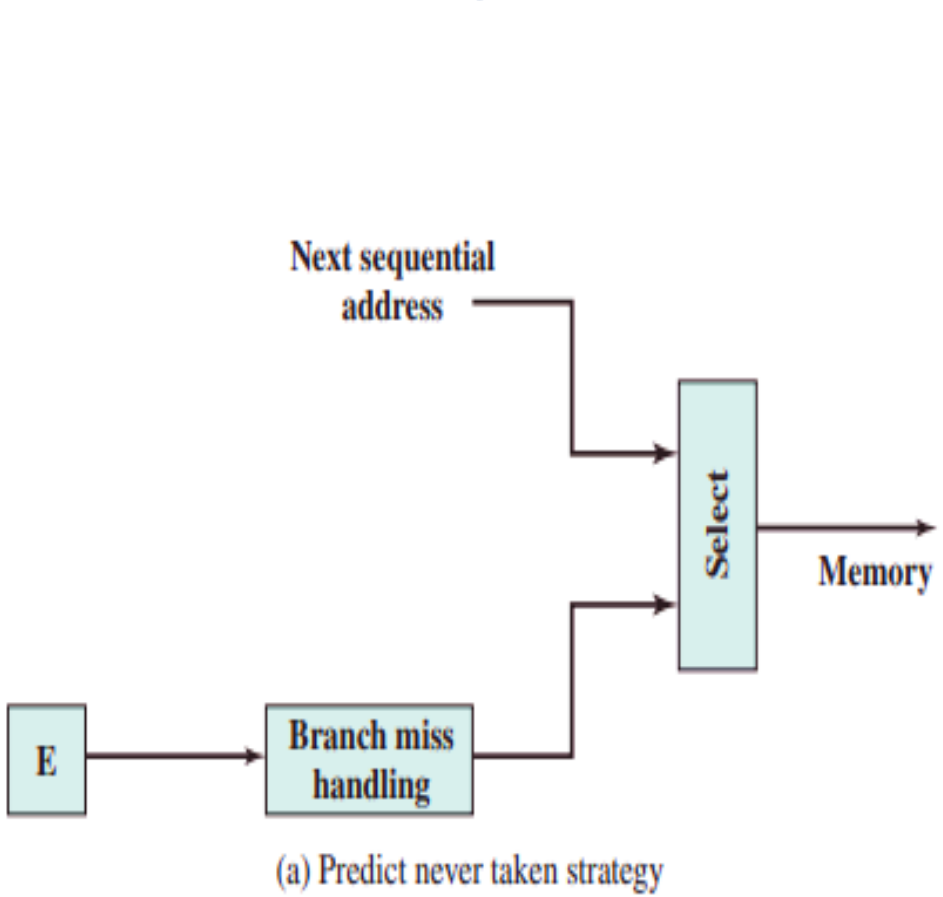
v. Branch History Table

- A small **cache memory** associated with the **instruction fetch stage** of the pipeline
- Each **entry** of the table consists of **three** fields:
    1. **Address** of the branch instruction
    2. **History bit(s)** that signifies the **state** of the branch prediction
    3. **Information** of the branch target instruction
- In most implementations, the third field contains the **address** of the branch target instruction
    - **Alternatively,** the third field contains the **branch target instruction** itself
    - **Tradeoff:** Storing the target address yields **smaller history table** but a **greater** instruction **fetch time** compared with storing the target instruction itself

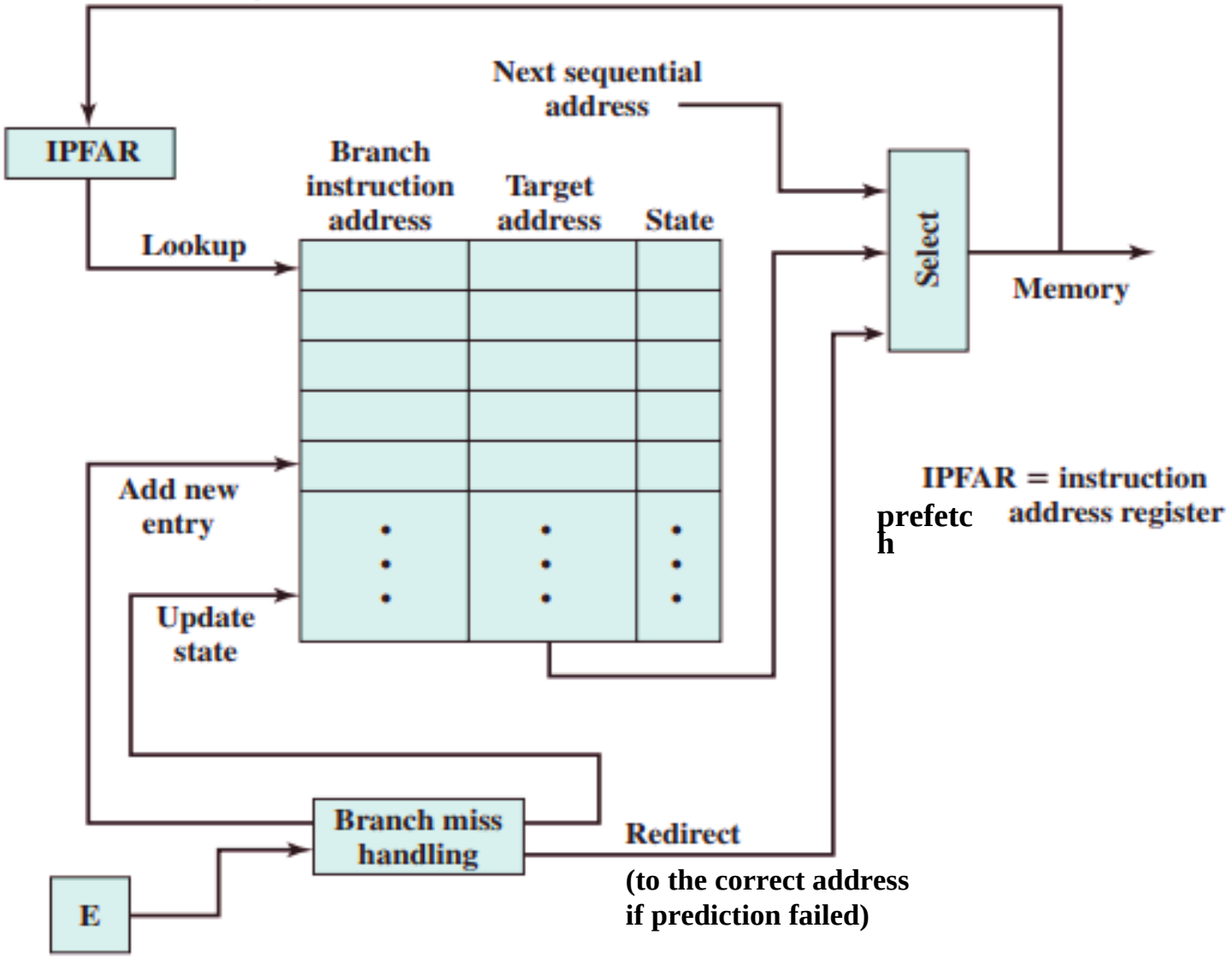
Branch History Table VS Predict Never Taken

- **Predict never taken (simplest approach):**
  - Always fetches the next sequential instruction (according to PC content)
    - If the branch is "**taken**" (prediction failed), certain logic in processor detects it and instructs that the next instruction be fetched from the **branch target address** and pipeline be **flushed**
- **Branch history table (most complex approach):**
  - This table is treated as **cache memory**
  - Each prefetch in **instruction fetch stage** triggers a **lookup (consult)** in this table
    - If **no match** is found, next to current prefetched instruction **in sequence** is going to be fetched
    - If **a match** is found, this instruction is treated as a branch instruction that already observed and a **prediction** is made based on the state of that entry in that table.
      - Either the next sequential address or branch target address is fed (placed) to the ***select logic*** for prefetching.

Comparing Their Logic Diagrams



(a) Predict never taken strategy



(b) Branch history table strategy

5. Delayed Branch

Deliberate approach

- Automatically **rearranging** the instructions within the program so that branch instructions occur **later** than actually desired
  - Execute one or more instructions **following** the conditional branch before the branch is taken to avoid stalling the pipeline while the branch instruction itself is being evaluated
    - These following executed instructions are **safe** to execute whether the branch is taken or not based on compiler's decision
    - No more time is wasted in search of optimum prediction
- It improves the pipeline performance maximizing its use

Intel 80486 Pipelining

Self Study