UNIT 5
Processor Organization

Agenda

1. Processor Organization
2. Register Organization : Case Study:: 8086
3. Instruction Cycle
4. Instruction Pipelining
5. Pipeline Hazards
6. Dealing with Branches
7. Superscalar Systems
8. Instruction Level Pipelining
9. Superpipelined Systems
10. Design Issues

Processor Organization

- Basic Functions of Processor

1. **Fetch Instruction:** The processor reads an instruction from memory (register, cache, main memory).
2. **Interpret Instruction:** The instruction is decoded to determine what action is required.
3. **Fetch Data:** The execution of an instruction may require reading data from memory or an I/O module.
4. **Process Data:** The execution of an instruction may require performing some arithmetic or logical operation on data.
5. **Write Data:** The results of an execution may require writing data to memory or an I/O module.

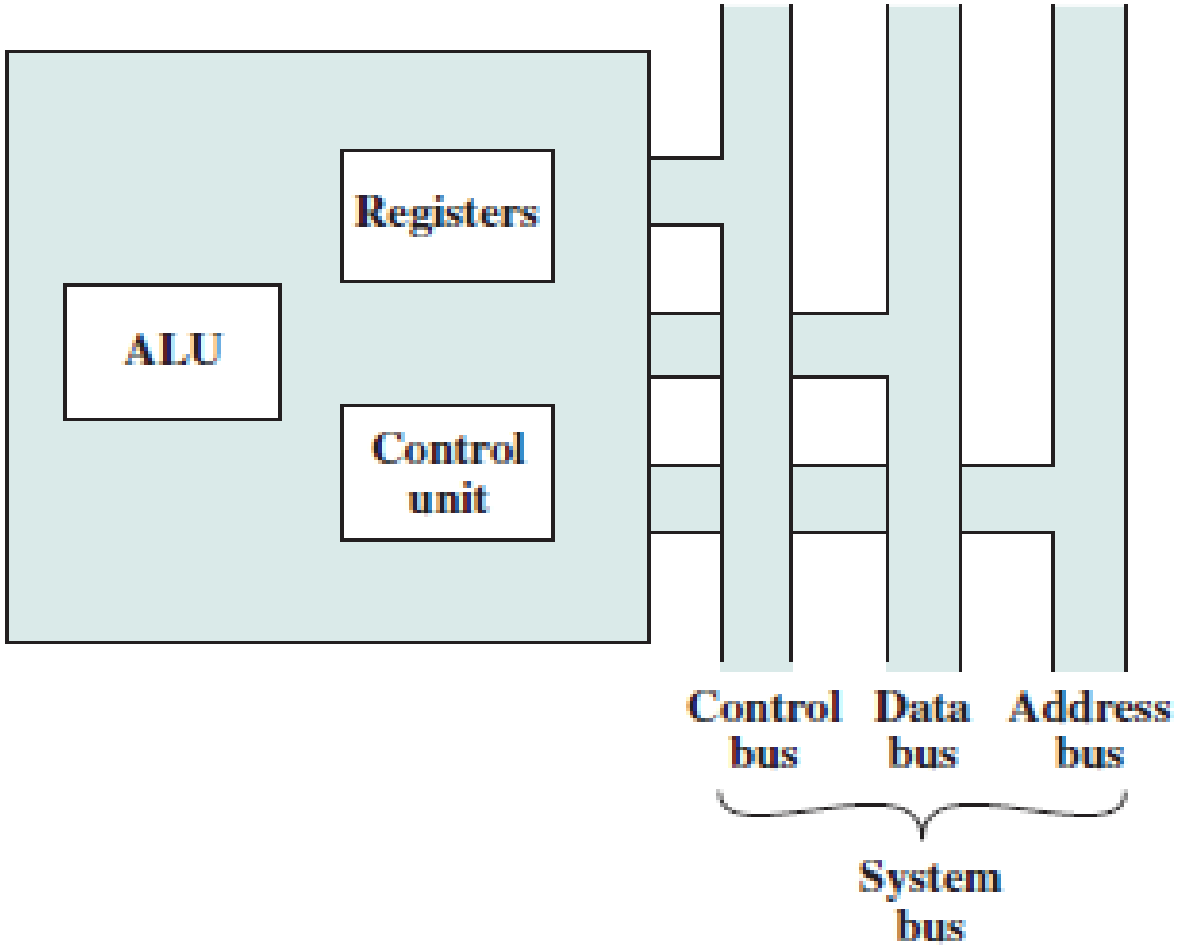
- To do these things, the processor needs to store some data temporarily.

- It must remember the location of the last or next instruction so that it can know where to get the next instruction.

- It needs to store instructions and data temporarily while an instruction is being executed.

- So, the **processor needs a small internal memory to do all these operations.**

The Basic Processor Organization



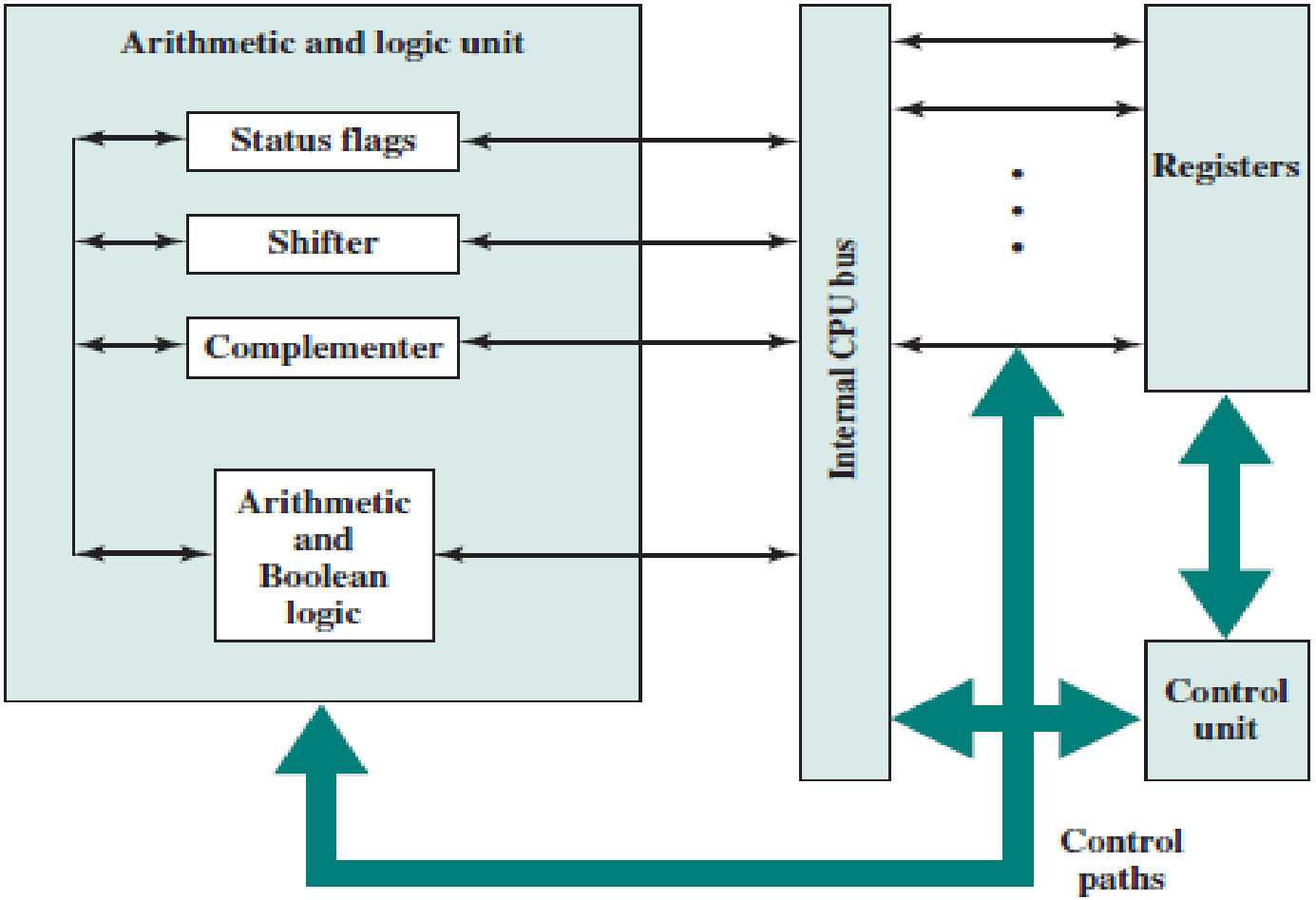
- The major components of the processor are an **Arithmetic Logic Unit (ALU), a Control Unit (CU)** and **Registers.**

- The **ALU** does the actual computation or processing of data.

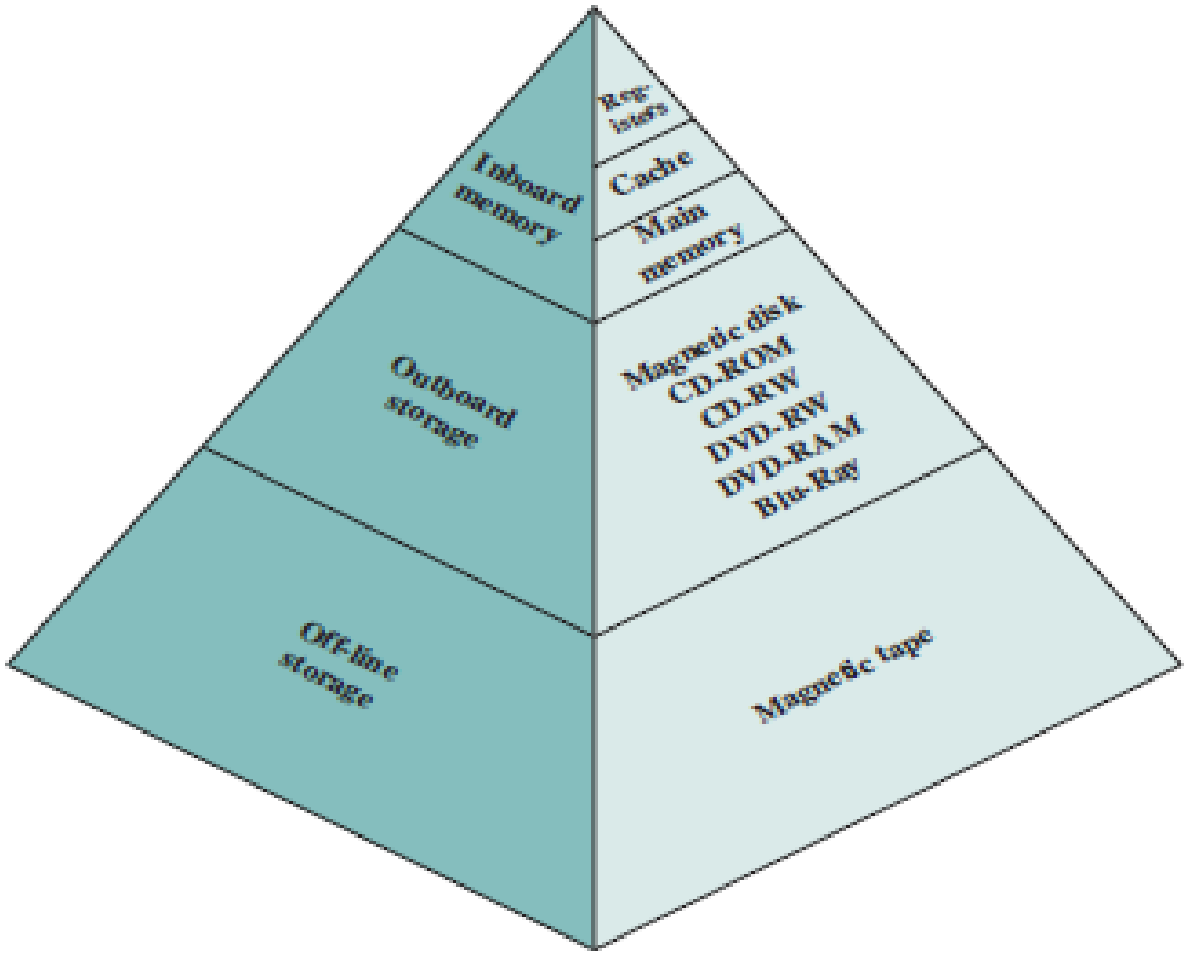
- The **CU** controls the movement of data and instructions into and out of the processor and controls the operation of the ALU.

- **Registers** consisting of a set of small storage locations.

Internal Structure of the Processor



Memory Hierarchy



Register Organization

- A computer system employs a **Memory Hierarchy.**

- At higher levels of the hierarchy, memory is faster, smaller, and more expensive (per bit).

- 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:

1. **User-visible Registers:** Enable the machine or assembly language programmer to minimize main memory references by optimizing use of registers.

2. **Control and Status Registers:** Used by the control unit to control the operation of the processor and gives status of results after execution of operations or programs.

1. User-Visible Registers

- A user-visible register is one that is directly used in machine language program, which is directly accessed by processor.

- We can characterize these in the following categories:
  - **General Purpose Registers**
    - **Data Registers**
    - **Address Registers**
  - **Condition codes**

- **General-purpose registers** can be assigned to a variety of functions by the programmer.

- Any general-purpose register can contain the operand for any opcode.

- Also they are used as per different cases in program.
  - There may be dedicated registers for **Floating-point** (Used by FPU) **and Stack operations** (SP, BP).

  - In some cases, general-purpose registers can be used for **Addressing Functions** (e.g. Register Indirect Addressing, Index or Displacement).

- In some processors, there is separate set of Data Registers and Address Registers.

- **Data registers** may be used only to hold data and cannot be employed in the calculation of an operand address. (eg. CX is default Count Register in 8086)

- **Address Registers** may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. (eg. BX can hold Base Address in 8086 addressing modes)

- Other examples of Address Registers include the following:
  - **Segment Pointers:** In a machine with segmented addressing, a Segment Register holds the address of the base of the segment. (CS, DS, SS, ES are segment registers for 8086)
  - **Index Registers:** These are used for Indexed Addressing and may be auto indexed. (SI, DI in8086)
  - **Stack Pointer:** If there is user-visible Stack Addressing, then typically there is a dedicated register that points to the top of the stack. (SP, BP are used to point to stack in 8086)

- A final category of registers, which is at least partially visible to the user, holds **Condition Codes** (also referred to as **Flags**).

- Condition codes are bits set by the processor hardware as the result of operations.

- Condition code bits are collected into one or more registers. Usually, they form part of a **Control Register**.

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. 2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH. 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. 4. Condition codes can be saved on the stack during subroutine calls along with other register information.	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. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. 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. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

2. Control and Status Registers

- **Program Counter (PC):** Contains the address of an next 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.

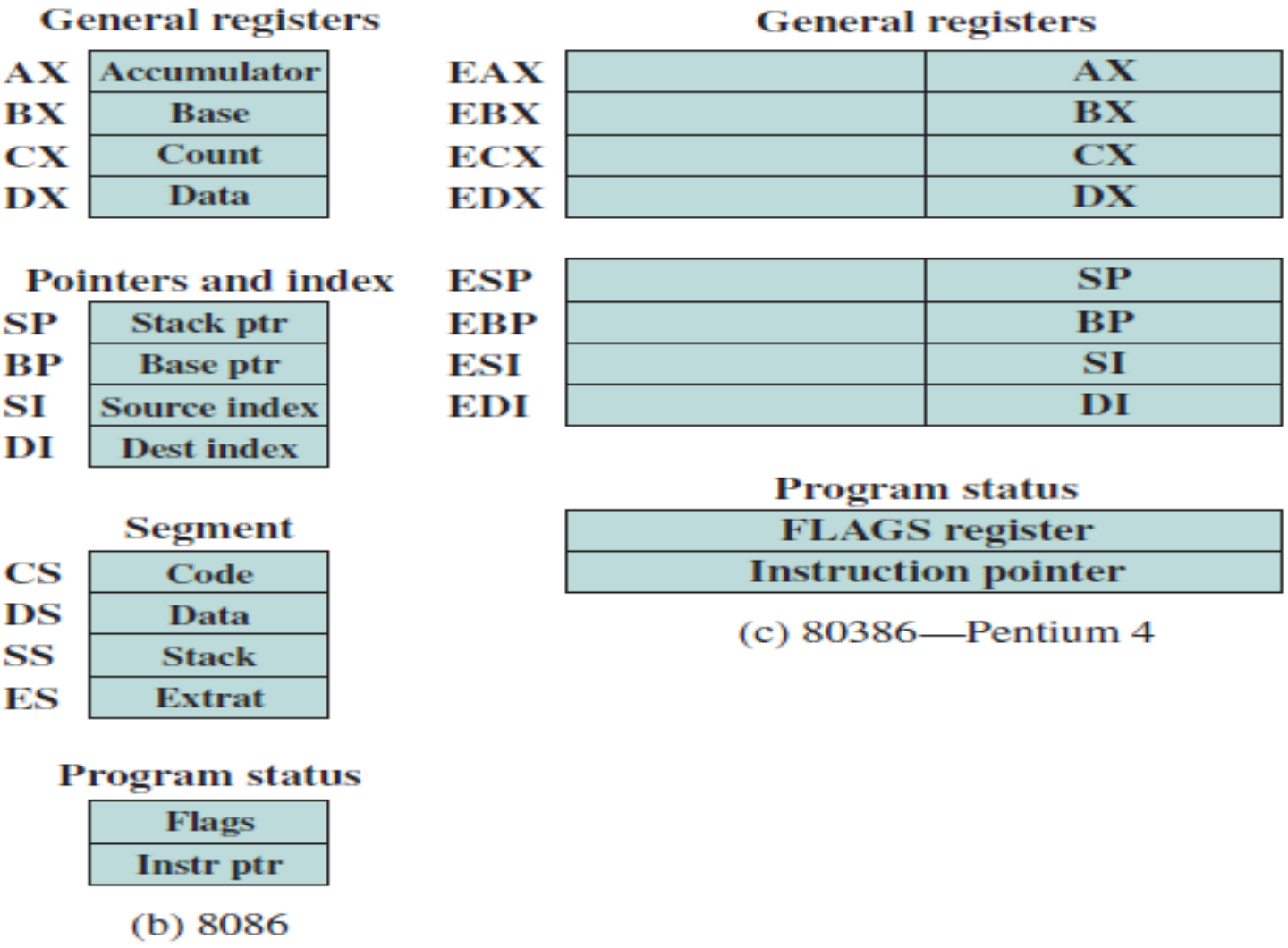
- Many processor designs include a register or set of registers, often known as the **Program Status Word (PSW),** that contain status information.

- The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

  - **Sign:** Contains the sign bit of the result of the last arithmetic operation.
  - **Zero:** Set when the result is 0.
  - **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.

- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

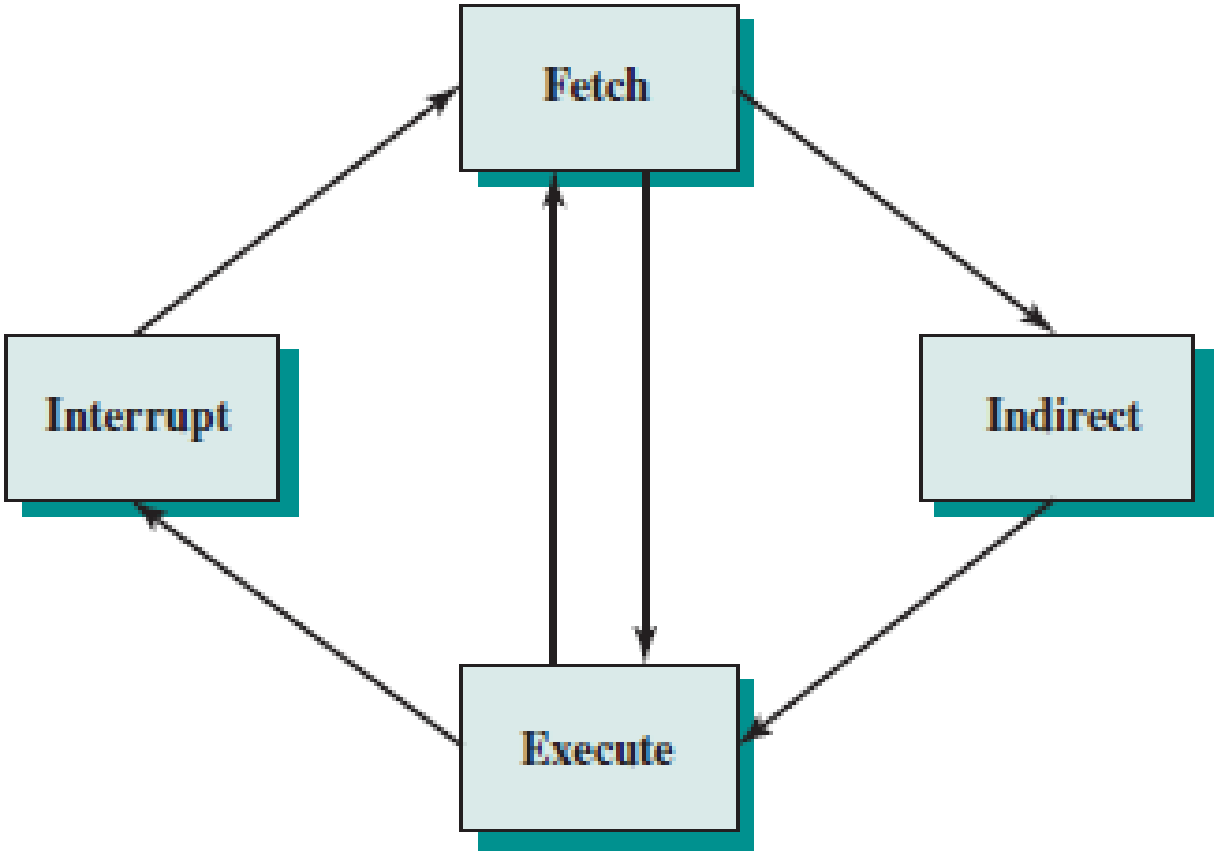
Ex. Register Organization



Instruction Cycle

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

Instruction Cycle



- **The Indirect Cycle**
  - The execution of an instruction may involve one or more operands in memory, each of which requires a memory access.
  - Usually each instruction is fetched first then executed.
  - 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.
  - Further, if indirect addressing is used, then additional memory accesses are required.

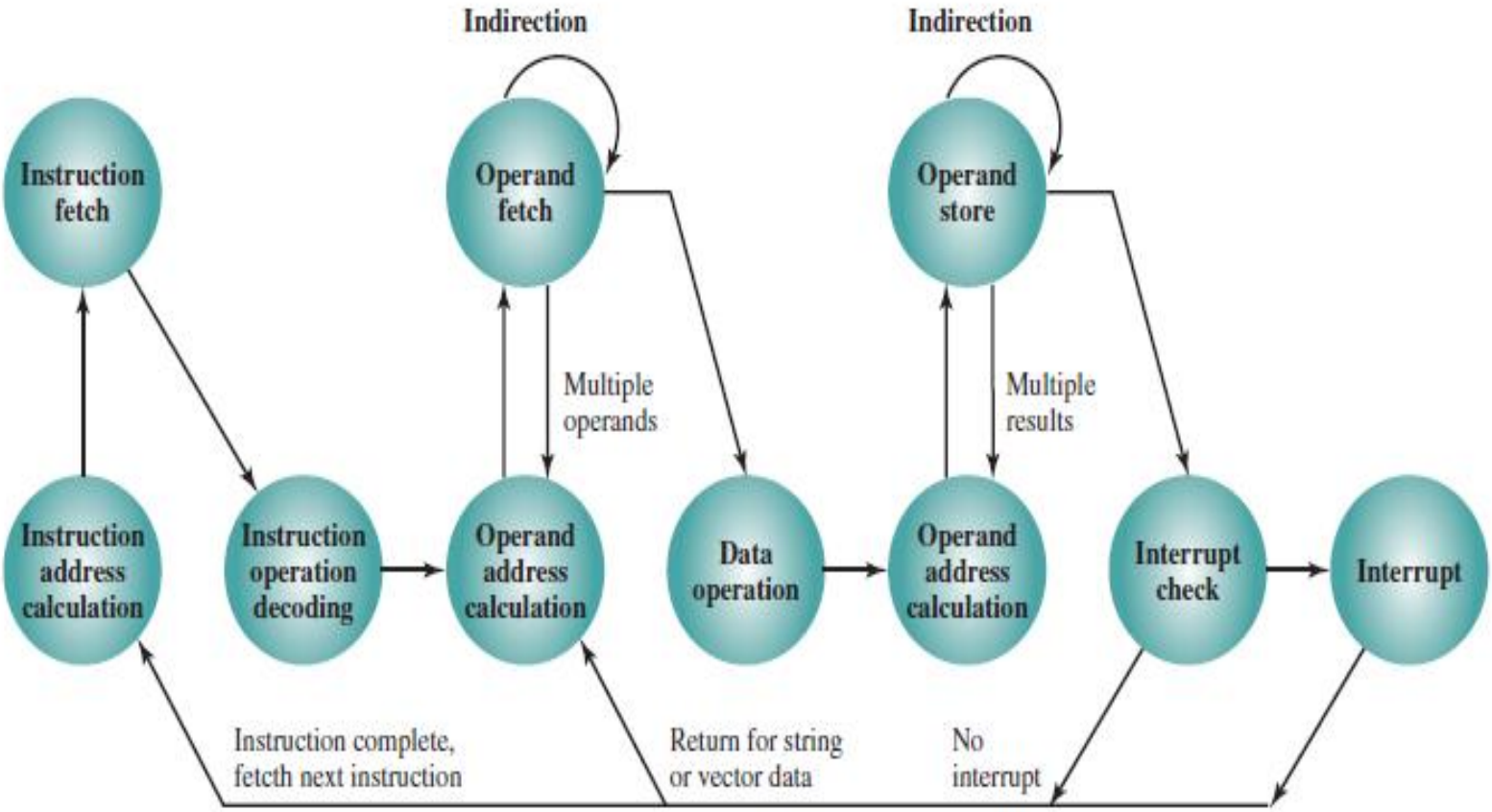


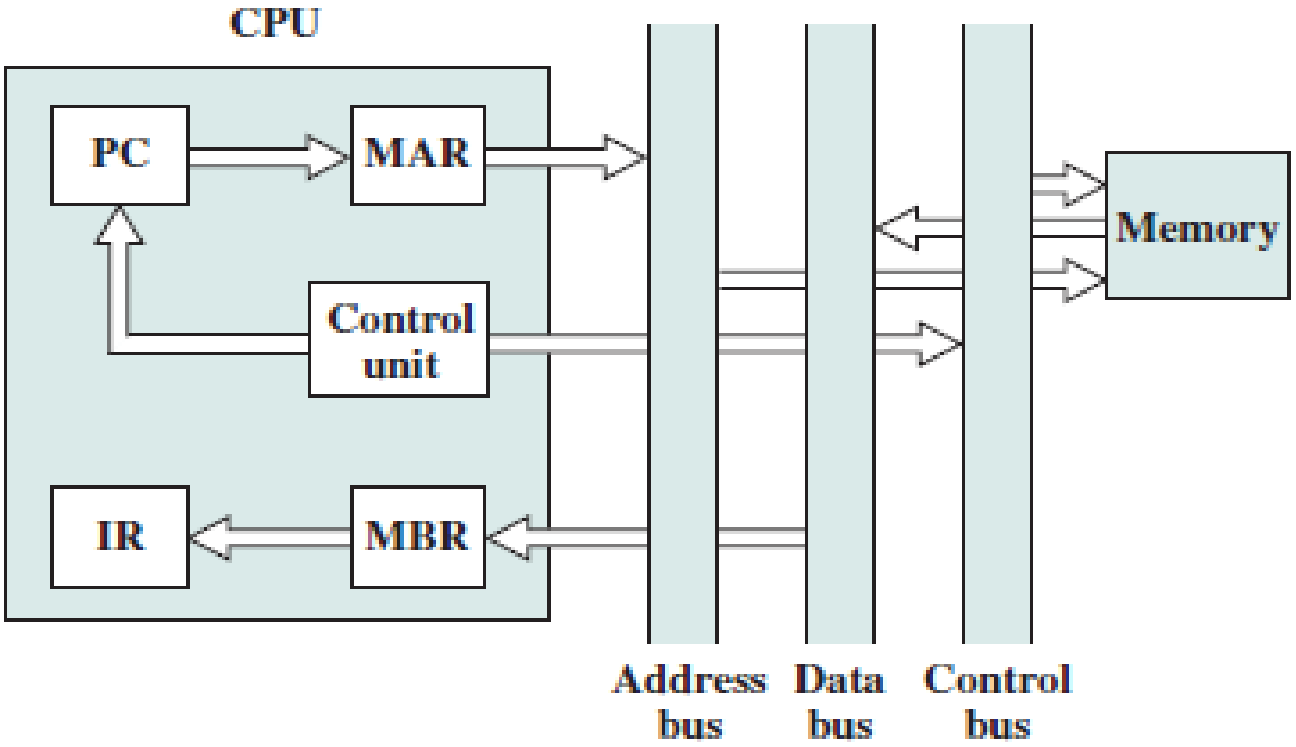
Figure 14.5 Instruction Cycle State Diagram

- **Data Flow**

1. **Fetch Cycle**

2. **Indirect Cycle**

3. **Interrupt Cycle**



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

Figure 14.6 Data Flow, Fetch Cycle

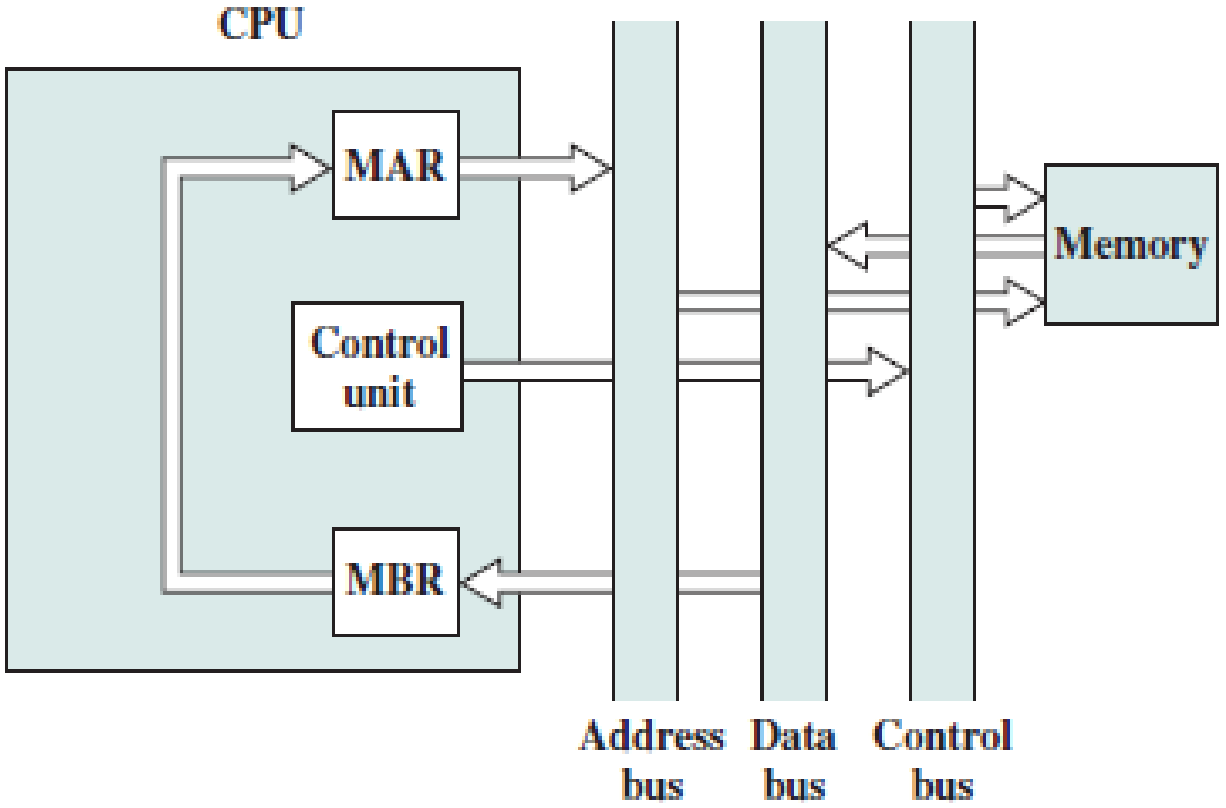


Figure 14.7 Data Flow, Indirect Cycle

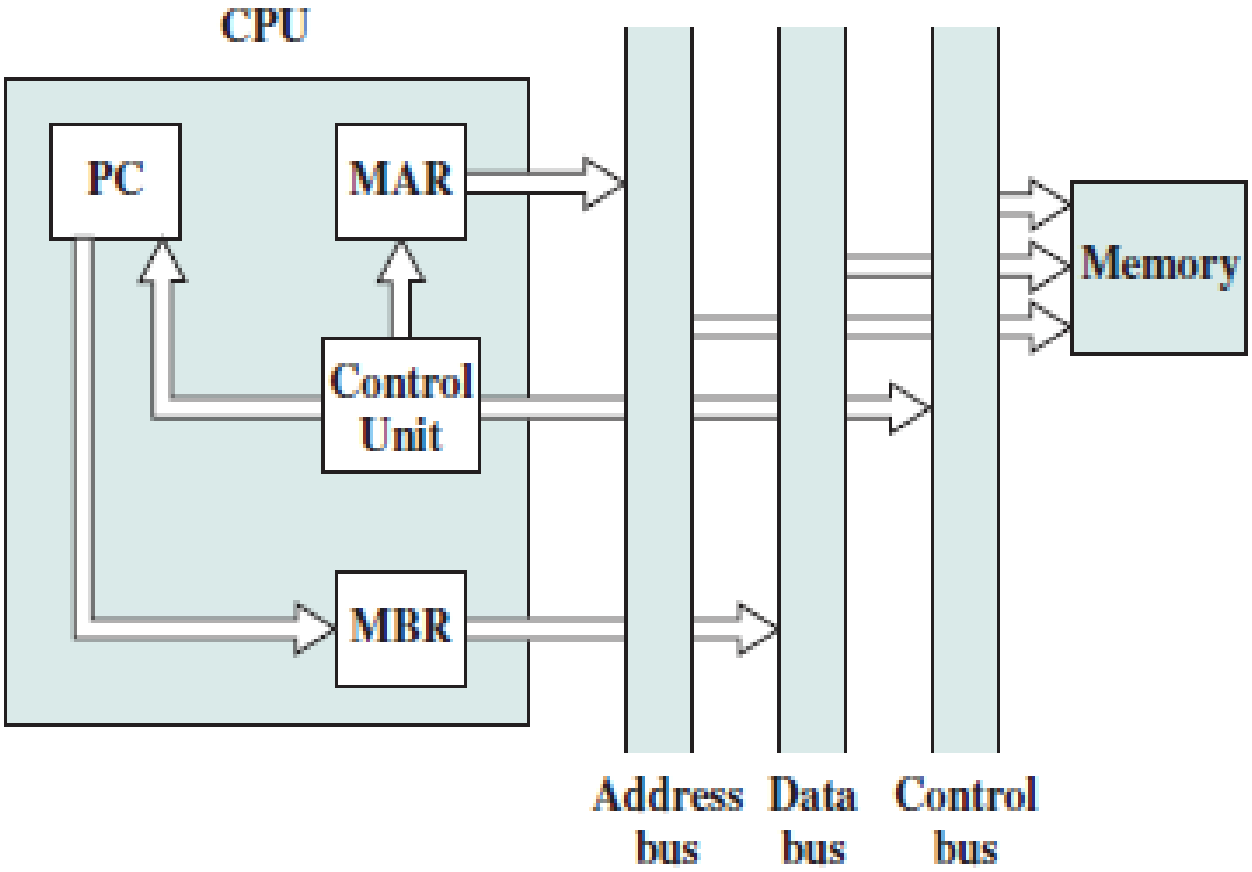
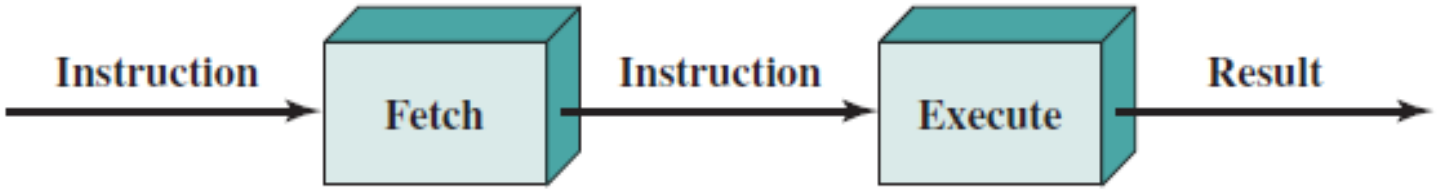
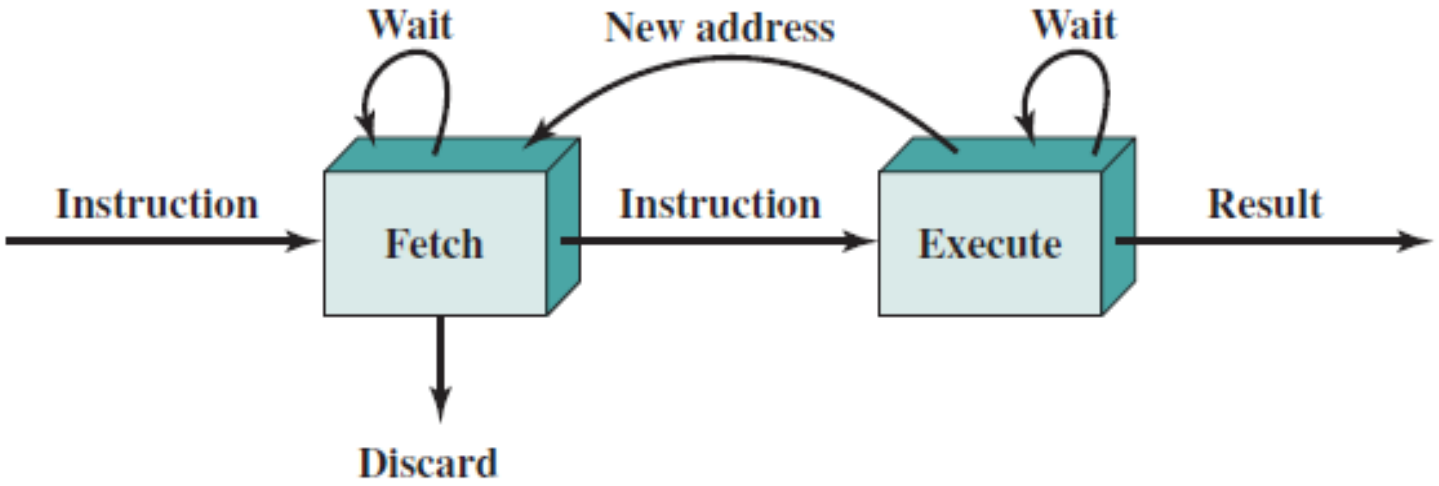


Figure 14.8 Data Flow, Interrupt Cycle

Instruction Pipelining: Pipelining Strategy



(a) Simplified view



(b) Expanded view

- The pipeline has two independent stages.
- The **first stage fetches an instruction and buffers it** (eg. Instruction stream byte queue in 8086). Where on same time the second stage is free, the first stage passes it the buffered instruction.
- The **second stage is executing the instruction**.
- During execution time 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**.

- **This process will speed up instruction execution, If the fetch and execute stages were of equal duration**, the instruction cycle time would be halved. But this do not happen. Because:

1. 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 execute cycle can empty its buffer**.

2. In case of conditional branch instruction, processor don't knows the address of the next instruction to be fetched. 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.**

- So new decomposition of the instruction processing is
  - **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.

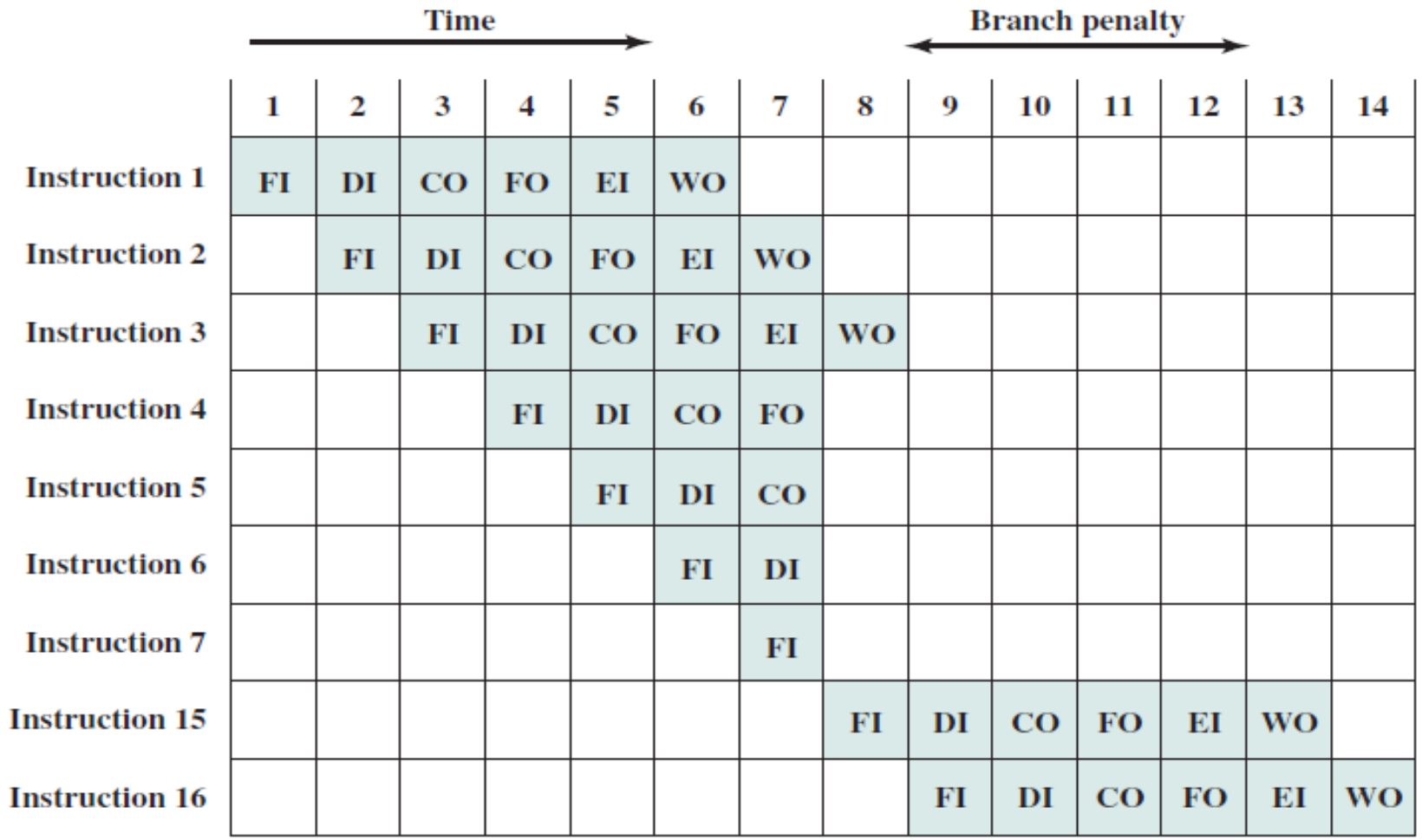
Timing Diagram for Instruction Pipeline Operation: Ideal

(Assume each step requires equal 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

- Several other factors still limit the performance enhancement.
  - If the **six stages are not of equal duration**, there will be some waiting involved at various pipeline stages.
  - **Conditional Branch Instruction** can invalidate several instruction fetches.
  - If **Interrupt** occurs in a execution process.

The Effect of a Conditional Branch on Instruction Pipeline Operation



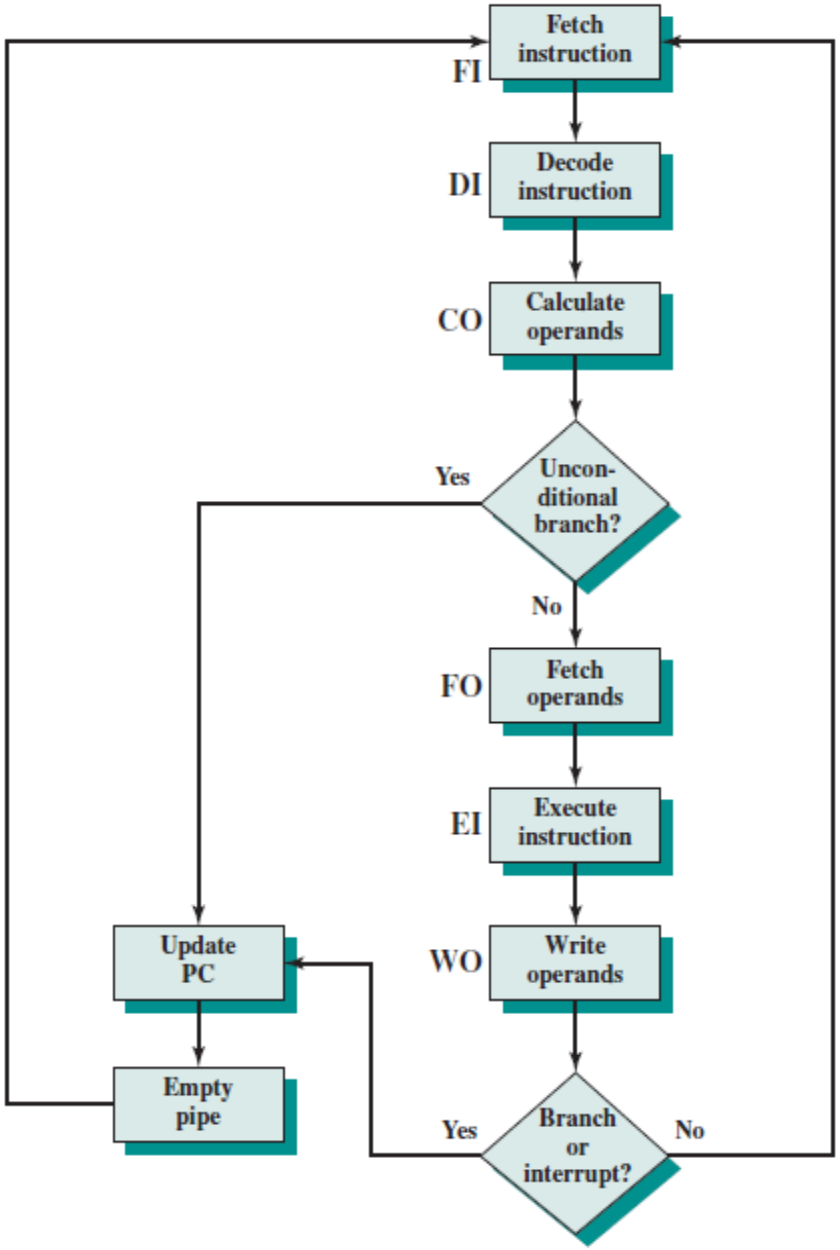
- Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline simply loads the next instruction in sequence (instruction 4) and proceeds.

- The branch is taken at the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful.

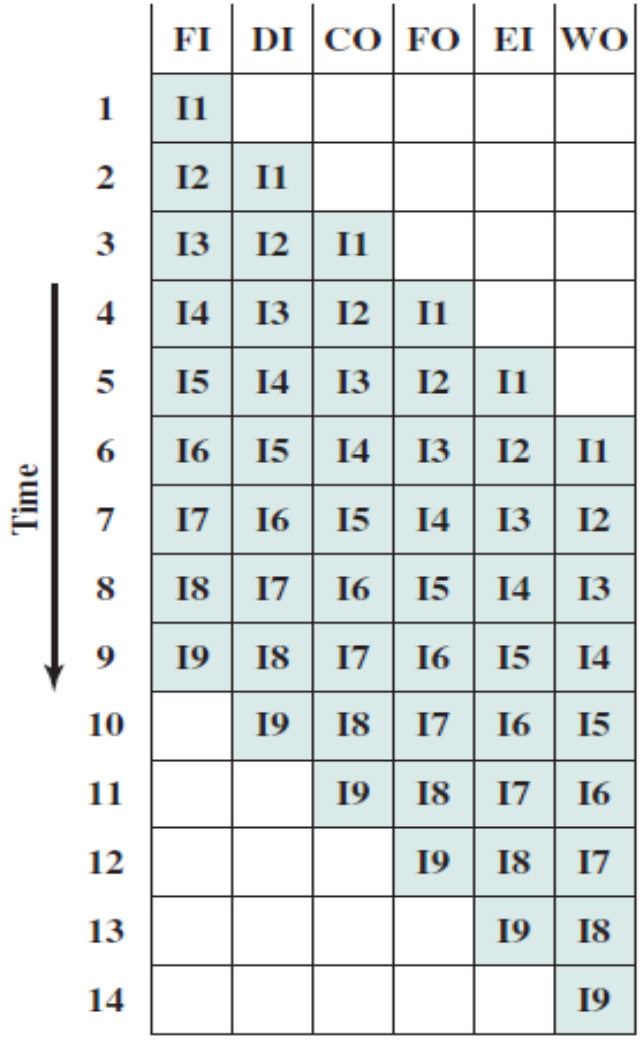
- During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch.

- So the **logic is needed for pipelining to account for branches and interrupts**, which is given next.

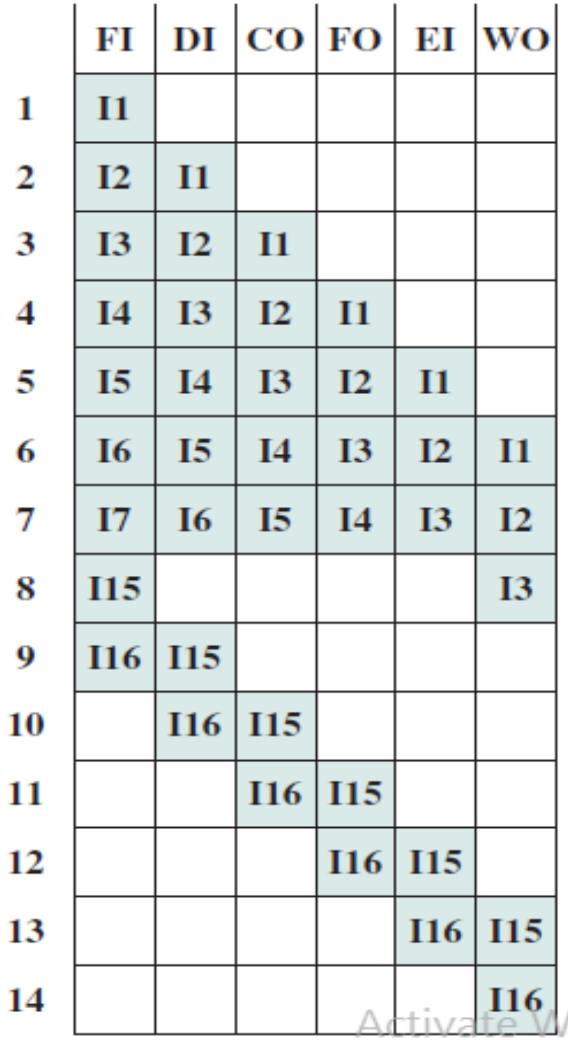
Six-Stage CPU Instruction Pipeline Ideal



An Alternative Pipeline Depiction



(a) No branches



(b) With conditional branch

- From the preceding discussion, it might appear that **the greater the number of stages in the pipeline, the faster the execution rate.**

- But, designer must still consider:
  - At each stage of the pipeline, there is some overhead involved **(1)** Moving data from buffer to buffer **(2)** In performing various preparation **(3)** Delivery functions. **This overhead can lengthen the total execution time of a single instruction.**

  - The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where **the logic controlling the gating between stages is more complex than the stages being controlled.**

Pipeline Hazard

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

- Such a pipeline stall is also referred to as a **Pipeline Bubble**. There are three types of hazards:

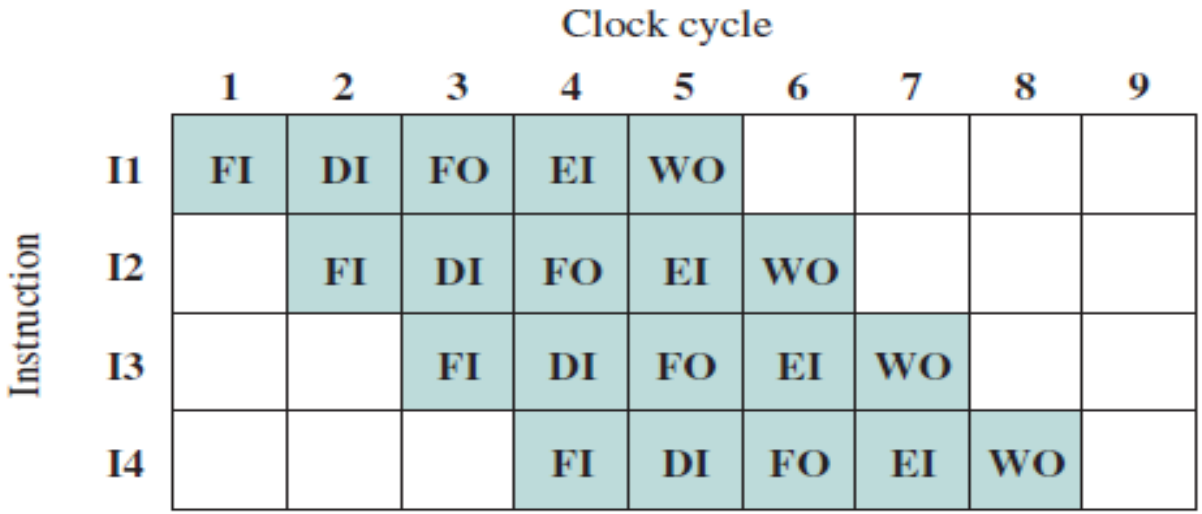
  - **Resource Hazard**
  - **Data Hazard**
  - **Control Hazard**

Resource Hazards

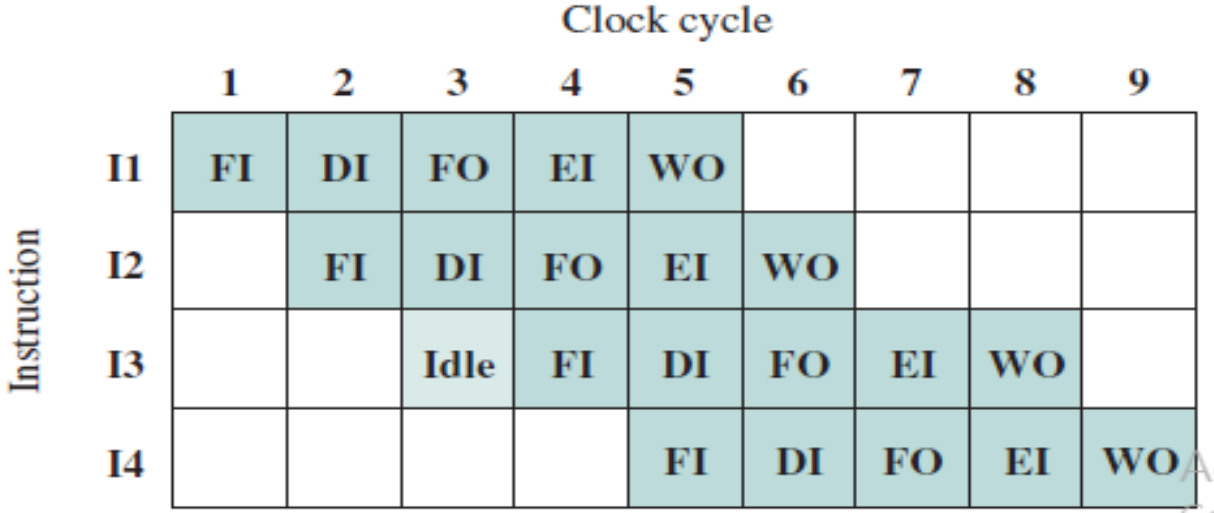
- A resource hazard occurs when **two (or more) instructions that are already in the pipeline need the same resource.**

- The result is that the **instructions must be executed in serial rather than parallel for a portion of the pipeline.**

- A resource hazard is sometime referred to as a **Structural Hazard**.



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

- Another example of a resource conflict is a situation in which **multiple instructions are ready to enter the execute instruction phase and there is a single ALU.**

- One solutions to such resource hazards is to increase available resources, such as **having multiple ports into main memory and multiple ALU units.**

Data Hazards

- A data hazard occurs when **there is a conflict in the access of an operand location.**

- In general terms, we can state the hazard in this form: **Two instructions in a program are to be executed in sequence and both access a particular memory or register operand.**

- If the two instructions are executed in strict sequence, no problem occurs.

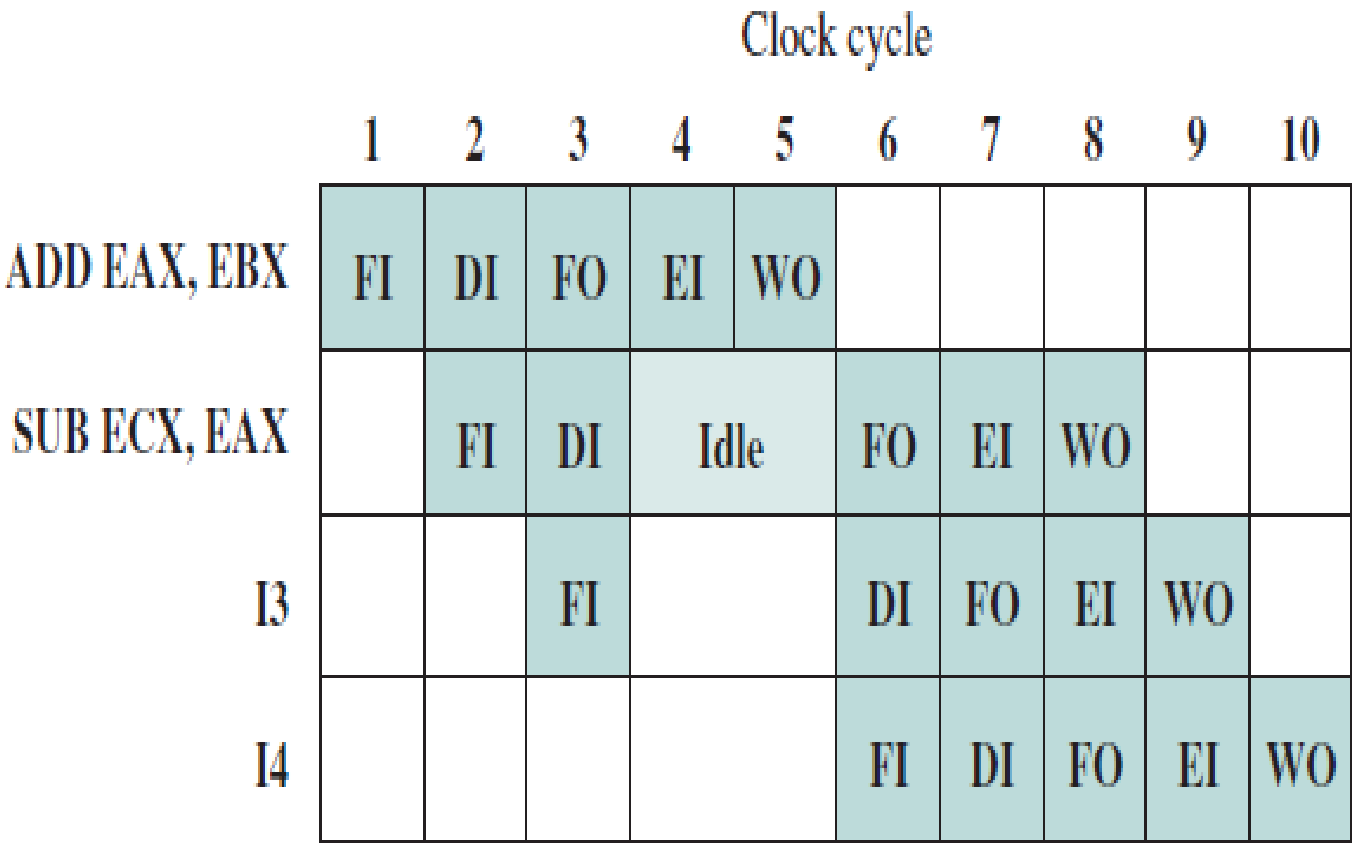
- However, **if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result** than would occur with strict sequential execution.

- In other words, the **program produces an incorrect result because of the use of pipelining.**

- As an example, consider the following x86 machine instruction sequence:

ADD EAX, EBX / EAX = EAX + EBX

SUB ECX, EAX / ECX = ECX - EAX


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

- There are three types of Data Hazards;
  - **Read after Write (RAW), or True Dependency:** An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.
  - **Write after Read (WAR), or Antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
  - **Write after Write (WAW), or Output Dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

Control Hazards

- A control hazard, also known as a **Branch Hazard.**

- Occurs when the **pipeline makes the wrong decision on a branch prediction** and therefore brings instructions into the pipeline that must subsequently be discarded.

Dealing with Branches

- One of the major problems in designing an instruction pipeline is **assuring a steady and correct flow of instructions to the initial stages of the pipeline.**

- The primary hurdle is the **conditional branch instruction**. Since, Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

- Following approaches have been taken for dealing with conditional branches:

1. **Multiple Streams**
2. **Prefetch Branch Target**
3. **Loop Buffer**
4. **Branch Prediction**
5. **Delayed Branch**

1. 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.**

- There are two problems with this approach:
  - With multiple pipelines there are contention **delays for access to the registers and to memory**.
  - **Additional branch instructions may enter the pipeline** (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.
- Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

2. Prefetch Branch Target

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

3. Loop Buffer

- A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline
- It **contains the "n" most recently fetched instructions, in sequence.**

- The external hardware first checks whether the branch target is within the buffer.

- If so, the next instruction is fetched from the buffer.

- The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. **Thus, instructions fetched in sequence will be available without the usual memory access time.**

2. If target of a branch is near to current executing instruction, then the target will already be in the buffer. **This is useful for the rather common occurrence of IF–THEN and IF–THEN–ELSE sequences.**

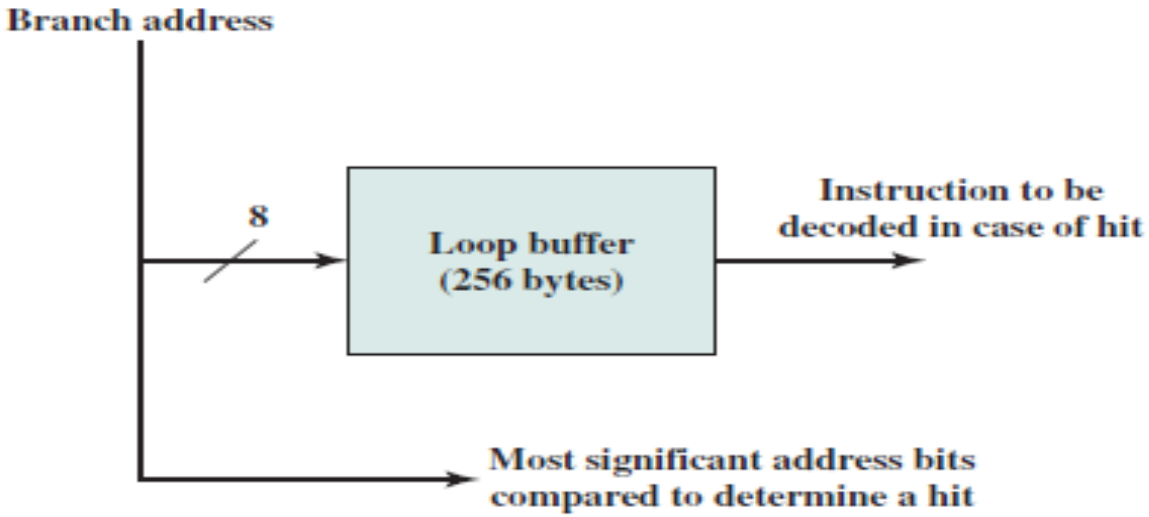
3. This strategy is particularly **well suited to dealing with loops, or iterations**; hence the name **Loop Buffer**.

4. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

- The loop buffer is similar in principle to a cache dedicated to instructions.

- The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

- Eg. 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.



4. Branch Prediction

- Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

  - **Predict never taken**
  - **Predict always taken**
  - **Predict by Opcode**
  - **Taken/not taken Switch**
  - **Branch History Table**

- **The first three approaches are static:** They do not depend on the execution history up to the time of the conditional branch instruction.

- **The latter two approaches are dynamic:** They depend on the execution history.

- **Predict never taken approach:** Processor assume that the branch will not be taken and continue to fetch instructions in sequence.
  - It is the most popular of all the branch prediction methods

- **Predict always taken approach:** Processor assume that the branch will be taken and always fetch for the branch target.

- **Predict by Opcode approach:** The processor assumes that the branch will be taken for certain branch opcodes and not for others.

- Dynamic branch strategies attempt to improve the accuracy of prediction **by recording the history of conditional branch instructions in a program.**

- One or more bits can be associated with each conditional branch instruction.

- Those bits are stored in the recent history of the instructions.

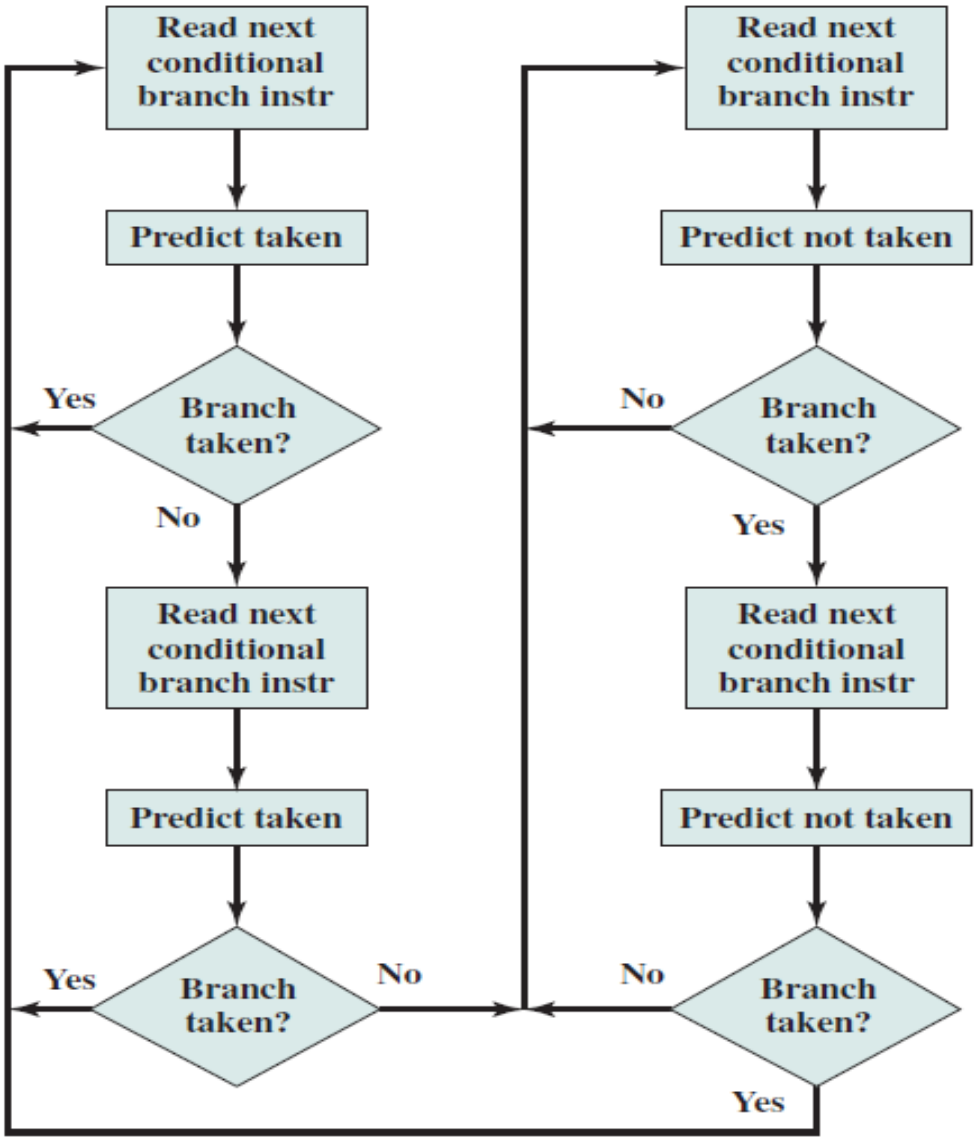
- These bits are referred by a **taken/ not taken switch** that directs the processor to make a particular decision when the next time instruction is encountered.

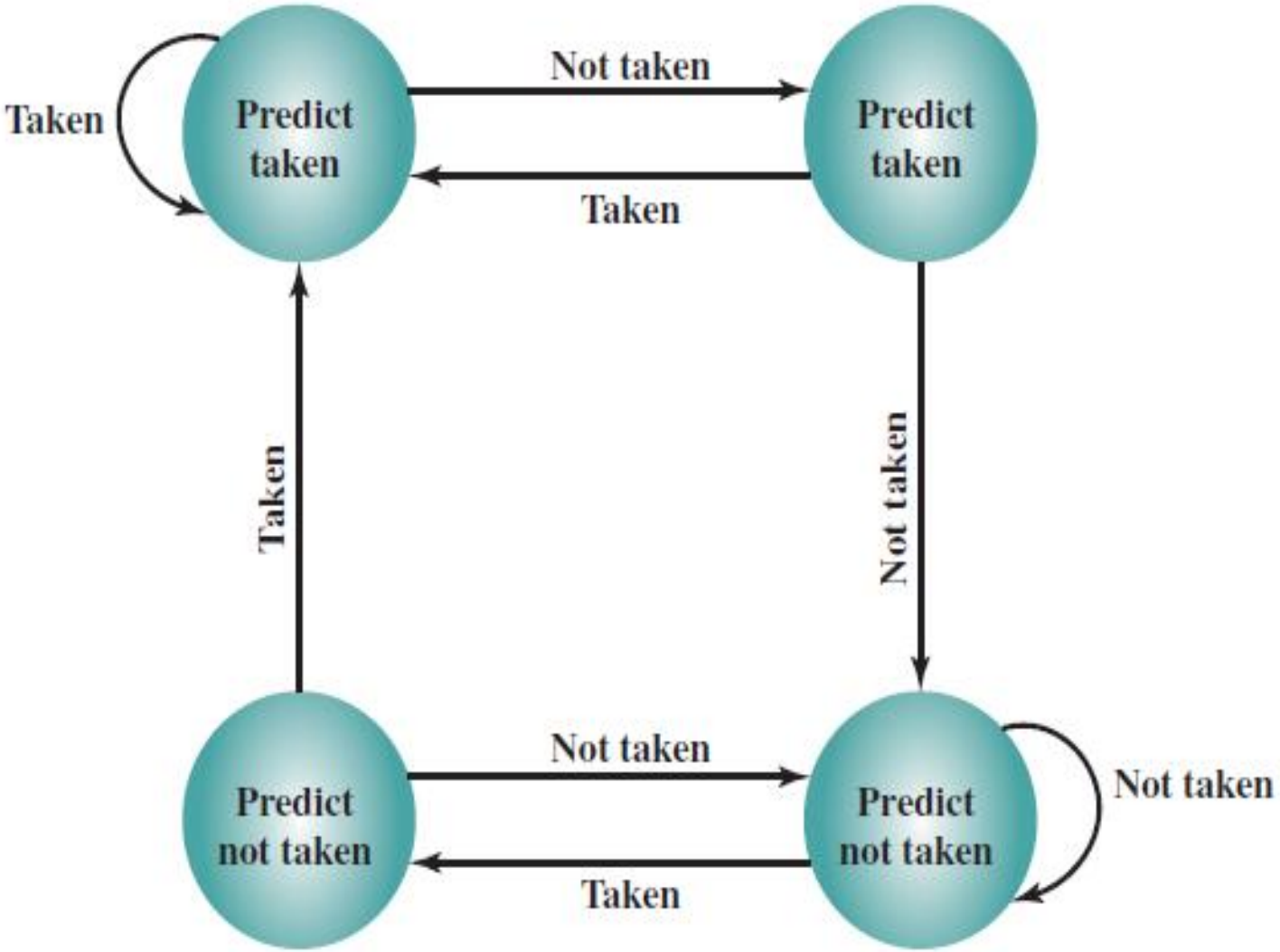
- These history bits are not stored in main memory. Rather, they are kept in temporary highspeed storage.

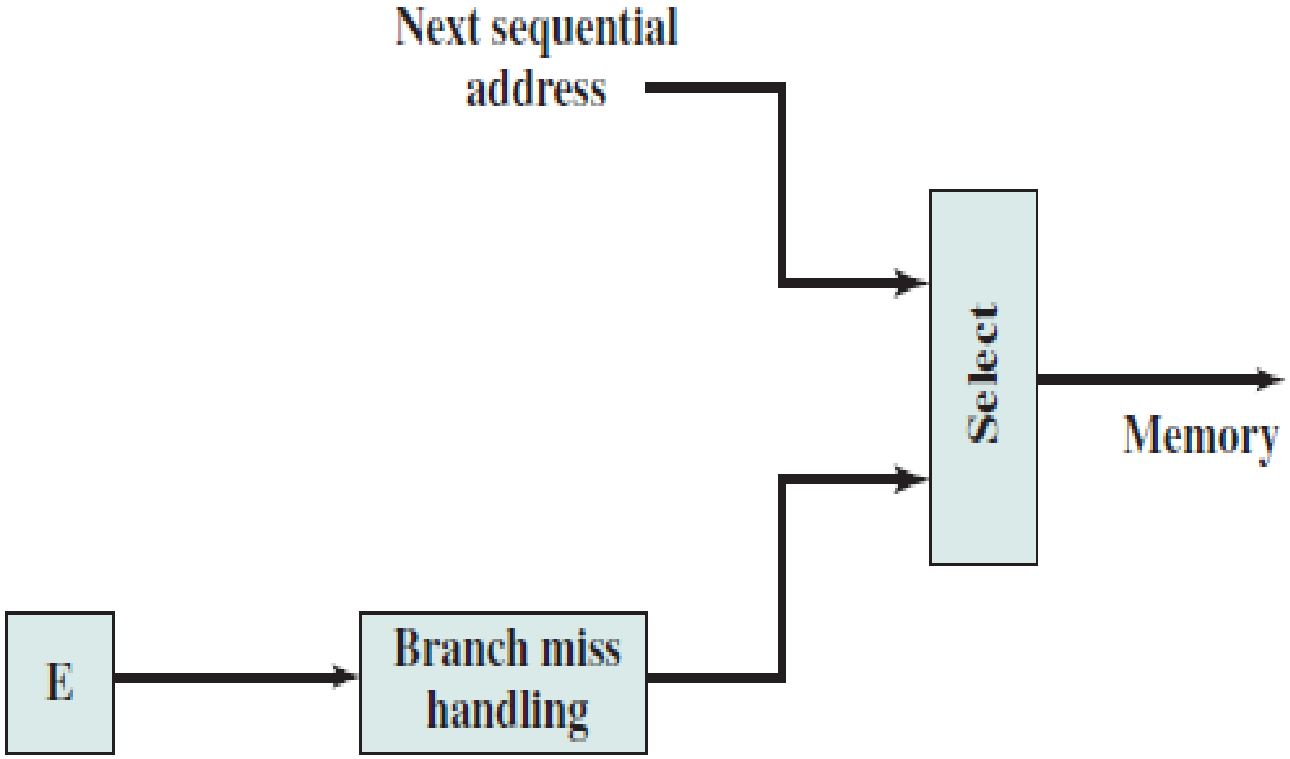
- Highspeed storages may be
  - **Cache Memory:** But when the instruction is replaced in the cache, its history is lost.

  - **History table:** One or more bits of condition branch instruction are stored in table and The processor could access the table associatively, like a cache.

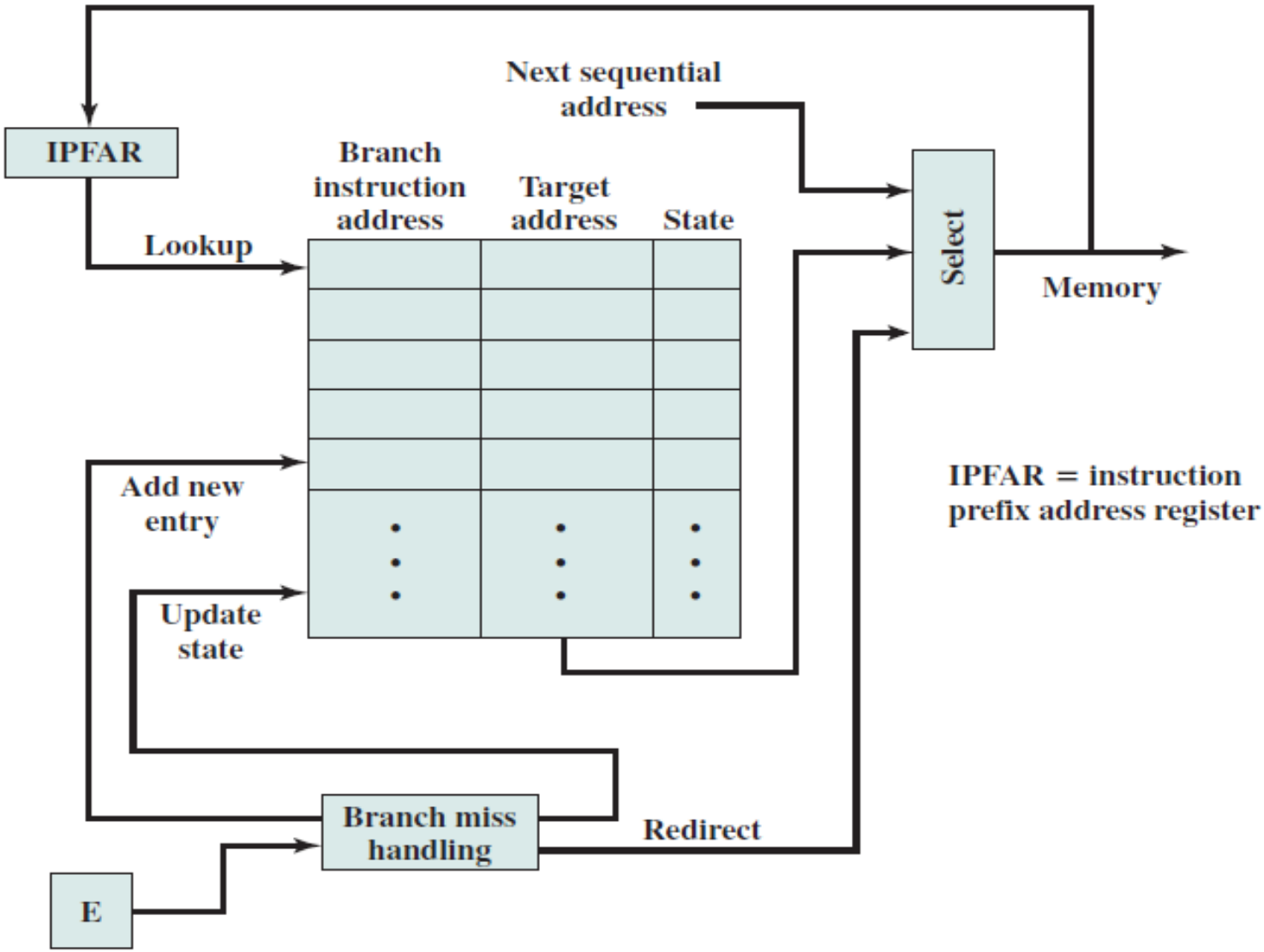
- **When Single bit is used:**
  - All that can be recorded is whether the last execution of this instruction resulted in a branch or not.
- **When two bits are used:**
  - They can be used to record the result of the last two instances of the execution of the associated instruction.


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


Taken
Predict taken
Not taken
Predict taken
Taken
Taken
Not taken
Predict not taken
Not taken
Predict not taken
Not taken
Taken



(a) Predict never taken strategy



(b) Branch history table strategy

5. Delayed Branch

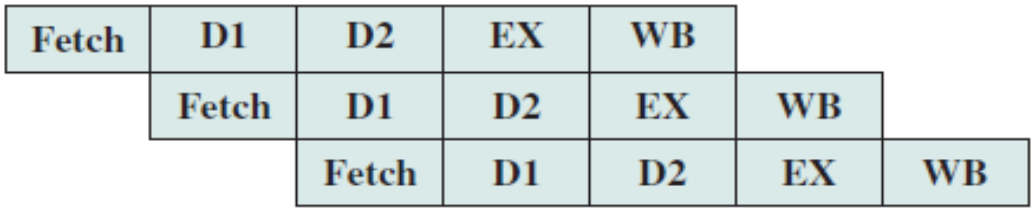
- It is possible to improve pipeline performance by **automatically rearranging instructions within a program**.

- So that **branch instructions occur later than actually desired.**

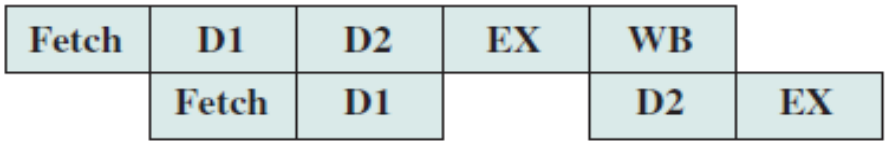
Intel 80486 Pipelining

- 80486 implements a five-stage pipeline:
  - **Fetch:** Instruction is fetched and data is filled in Buffer.
  - **Decode stage 1:** All opcode and addressing-mode information is decoded in the D1 stage.
  - **Decode stage 2:** The D2 stage expands each opcode into control signals for the ALU and calculates oprands.
  - **Execute:**
  - **Write back:**

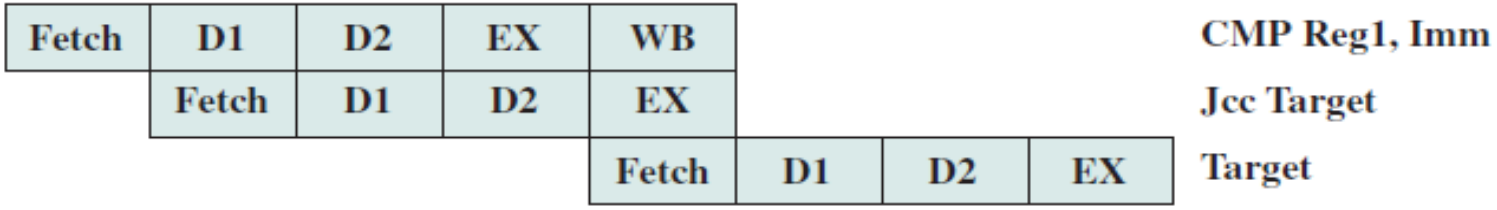
80486 Instruction Pipeline Examples



(a) No data load delay in the pipeline



(b) Pointer load delay



(c) Branch instruction timing

RISC [Reduced Instruction Set Computing]

- Small set of instructions.
- Simplified and reduced instruction set.
- The reduced instruction set means that **the processor can execute the instructions more quickly**, potentially allowing for greater speeds.
- However, only allowing such simple instructions means a **greater burden is placed upon the software itself (Compiler).**

- As less instructions are available in the instruction set, it is **difficult for programmer to write a software with the instructions that are available.**

- Because of less instructions, **Addressing modes are simplified and less**, and the length of the codes is fixed.

- **Instruction pipelining can be implemented easily.**

- Only **LOAD/STORE instructions can access memory.**

- **Each instruction requires single cycle time** to execute the program.
- So, **RISC systems shorten execution time as they have reduced the clock cycles per instruction.**
- Mainly used for **real time applications**.
- Examples of RISC Processors: **Atmel AVR, PIC, ARM.**

CISC [Complex Instruction Set Computing]

- Very large instruction sets.
- As the range of more advanced instructions available, **it becomes easy for compilers to execute programs and helps to improve performance.**
- **More specialized addressing modes** and registers also being implemented.

- **Instruction pipelining can not be implemented easily.**

- Mainly used in normal PC's, Workstations and servers.

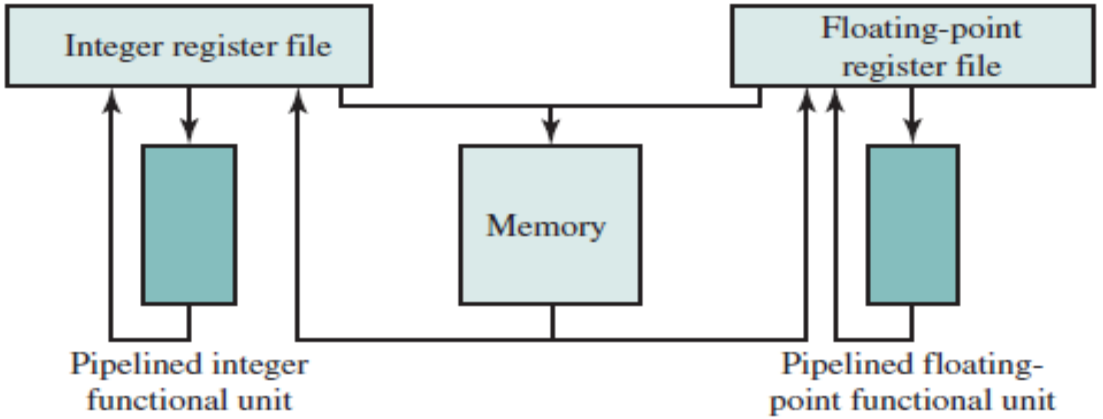
- **CISC systems shorten execution time by reducing the number of instructions per program**, as direct instructions are available for many operations.

- Examples of CISC Processors: **Intel 8086 and others.**

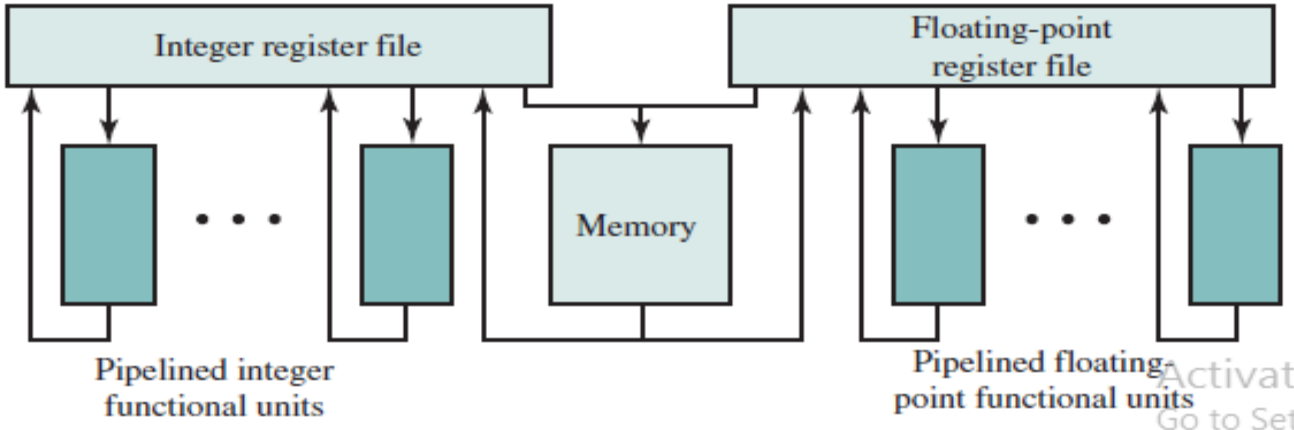
RISC	CISC
Multiple register sets, often consisting of more than 256 registers	Single register set, typically 6 to 16 registers total
Three register operands allowed per instruction (e.g., `add R1, R2, R3`)	One or two register operands allowed per instruction (e.g., `add R1, R2`)
Highly pipelined	Less pipelined
Fixed length instructions	Variable length instructions
Complexity in compiler	Complexity in microcode
Only load and store instructions can access memory	Many instructions can access memory
Few addressing modes	Many addressing modes

TABLE 9.1 The Characteristics of RISC Machines versus CISC Mach

Scalar Vs. Superscalar



(a) Scalar organization



(b) Superscalar organization

Superscalar Systems

- The term superscalar first coined in 1987.
- **Superscalar** refers to a machine that is **designed to improve the performance of the execution of scalar instructions.**
- The main characteristic of the superscalar approach is the **ability to execute instructions independently and concurrently in different pipelines.**

- This **allows instructions to be executed in an order different from the program order.**

- In the superscalar organization, **main system is divided into multiple functional units, each of which is implemented as a pipeline**.

- Each **individual functional unit provides a degree of parallelism** by considering of its pipelined structure.

- The use of **multiple functional units enables the processor to execute streams of instructions in parallel**, one stream for each pipeline.

- It is the **responsibility of the hardware, in conjunction with the compiler, to assure that the parallel execution does not violate the execution of the program.**

Instruction-level Parallelism

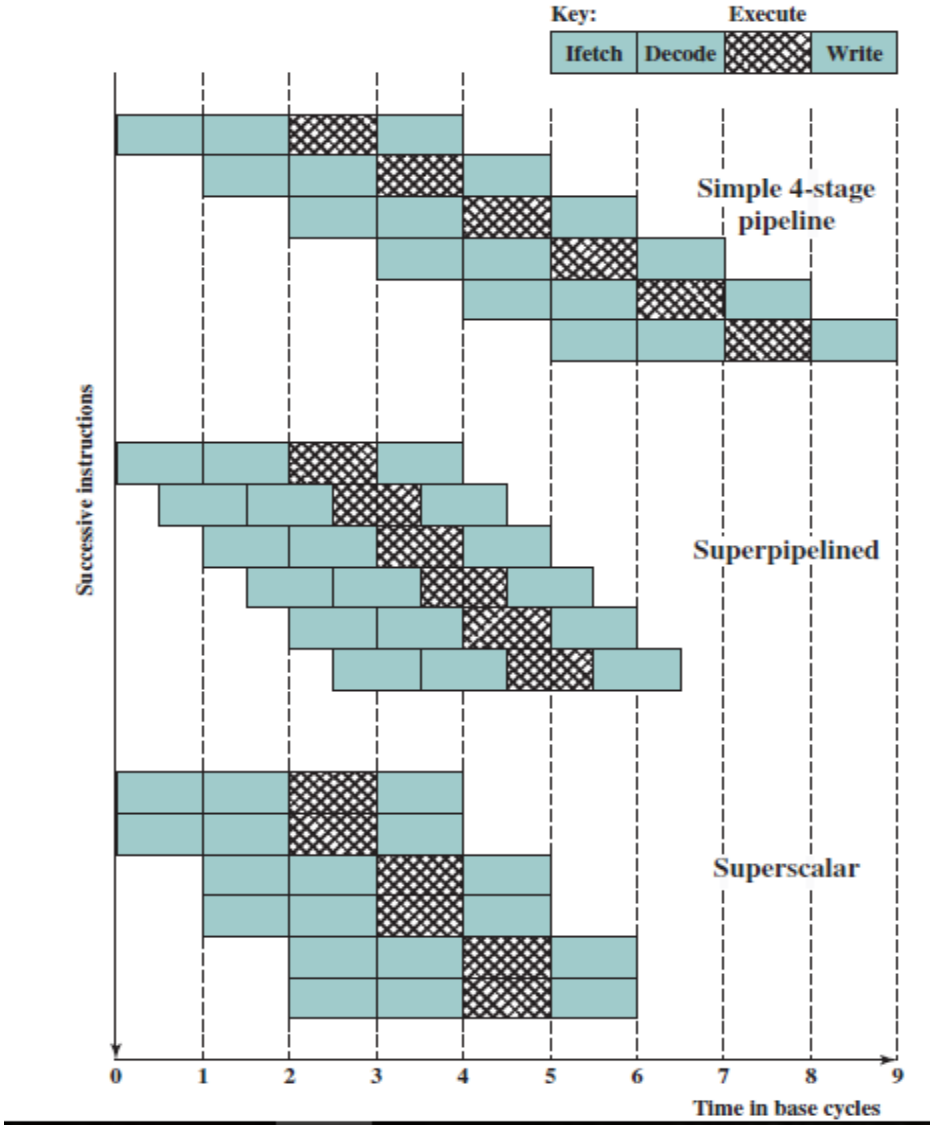
- Instruction-level parallelism (ILP) is a measure of **how many of the instructions in a computer program can be executed simultaneously.**

- Can be achieved using
  - **Hardware**
  - **Software (Compiler)**

Superpipelined Systems

- Term Superpipelining first coined in 1988.
- It is **capable of performing two pipeline stages per clock cycle.**
- An alternative way of looking at this is that **the functions performed in each stage can be split into two nonoverlapping parts and each can execute in half a clock cycle.**
- A superpipeline implementation that behaves in this fashion is said to be of **degree 2**.

Superscalar Vs. Superpipelined



Constraints

- If user want to implement instruction-level parallelism in superscalar machine, or want to increase its degree, then following are the limitations:

  - **True data dependency (Read after Write)**
  - **Procedural dependency**
  - **Resource conflicts**
  - **Output dependency (Write after Write)**
  - **Antidependency (Write after Read)**

True Data Dependency

```
ADD     EAX, ECX
MOV     EBX, EAX
```

- The second instruction can be fetched and decoded but cannot execute until the first instruction executes.
- The reason is that the second instruction needs data produced by the first instruction.

Procedural Dependencies

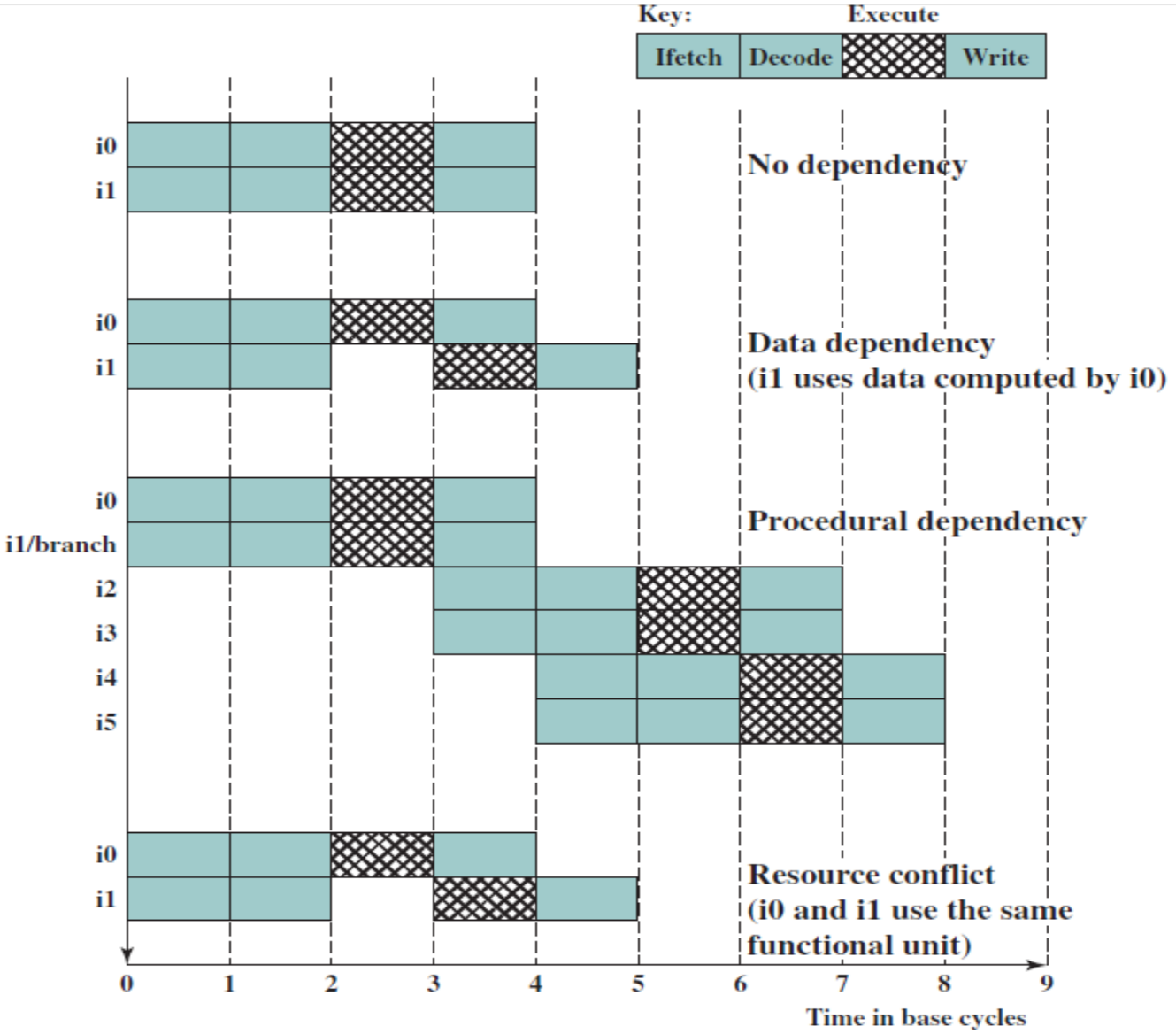
- The presence of branches in an instruction sequence complicates the pipeline operation.
- The instructions following a branch (taken or not taken) have a **Procedural Dependency** on the branch and cannot execute next instruction until the branch is executed.
- Since, **you can not execute next instruction until you execute branch instruction.**

Resource Conflict

- A resource conflict is a **competition of two or more instructions for the same resource at the same time.**

- Examples of resources include memories, caches, buses, register-file ports, and functional units (e.g., ALU adder).

Effect of Dependencies



Design Issues

1. Instruction Level and Machine Parallelism
2. Instruction Issue Policy
3. Register Renaming
4. Machine Parallelism
5. Branch Prediction
6. Superscalar Execution and Implementation.

1. Instruction Level and Machine Parallelism

- **Instruction-level Parallelism** exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping.

- As an example of the concept of instruction-level parallelism, consider the following two code fragments

```
Load R1 ← R2          | Add R3 ← R3, "1"
Add R3 ← R3, "1"      | Add R4 ← R3, R2
Add R4 ← R4, R2       | Store [R4] ← R0
```

- The three instructions on the left are independent, and in theory all three could be executed in parallel.

- In contrast, the three instructions on the right cannot be executed in parallel because the second instruction uses the result of the first, and the third instruction uses the result of the second.

- So we can conclude,
  - The degree of Instruction-level Parallelism is determined by

    1. **True Data Dependencies**
    2. **Procedural Dependencies** in the code.
    3. **Operational Latency:** The time until the result of an instruction is available for use as an operand in a subsequent instruction.

  - These factors are actually **dependent on next instructions in program and the instruction set architecture in the application.**

- **What is Machine Parallelism:**

  – Machine parallelism is a measure of the **ability of the processor to take advantage of instruction-level parallelism.**

- Machine parallelism is determined by

  1. The number of instructions that can be fetched and executed at the same time (the **number of parallel pipelines**).

  2. The speed by which processor finds independent instructions.

- So while designing the Superscalar architectures:
  - Both Instruction-level and Machine Parallelism are important factors in enhancing performance.
  - A program may not have enough instruction-level parallelism to take full advantage of machine parallelism.
  - The degree of instruction level parallelism is high in RISC systems compared to CISC systems.

2. Instruction Issue Policy

- In order to achieve Instruction level parallelism, the processor must be able to carefully organize the stages (fetching, decoding, and execution) of instructions in parallel, in order to get result correctly.

- The term **Instruction Issue** to refer to the process of initiating instruction execution in the processor's functional units.

- The term **Instruction Issue Policy** to refer to the protocol used to issue instructions.

- We can group superscalar instruction issue policies into the following categories:
  - In-order issue with In-order completion.
  - In-order issue with Out-of-order completion.
  - Out-of-order issue with Out-of-order completion.

In-order issue with in-order completion

- The simplest instruction issue policy is to issue instructions in the exact order that would be achieved by sequential execution (**In-order issue)** and to write results in that same order **(in-order completion).**

- Ex. Assume a superscalar pipeline system, which is
  - 4 stage pipelining
  - Fetch and Decode: 2 functional units
  - Execute: 3 functional units
  - Write Back: 2 functional units

- Also
  - **I1** requires two cycles to execute.
  - **I3** and **I4** conflict for the same functional unit.
  - **I5** depends on the value produced by **I4**.
  - **I5** and **I6** conflict for a same functional unit.

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

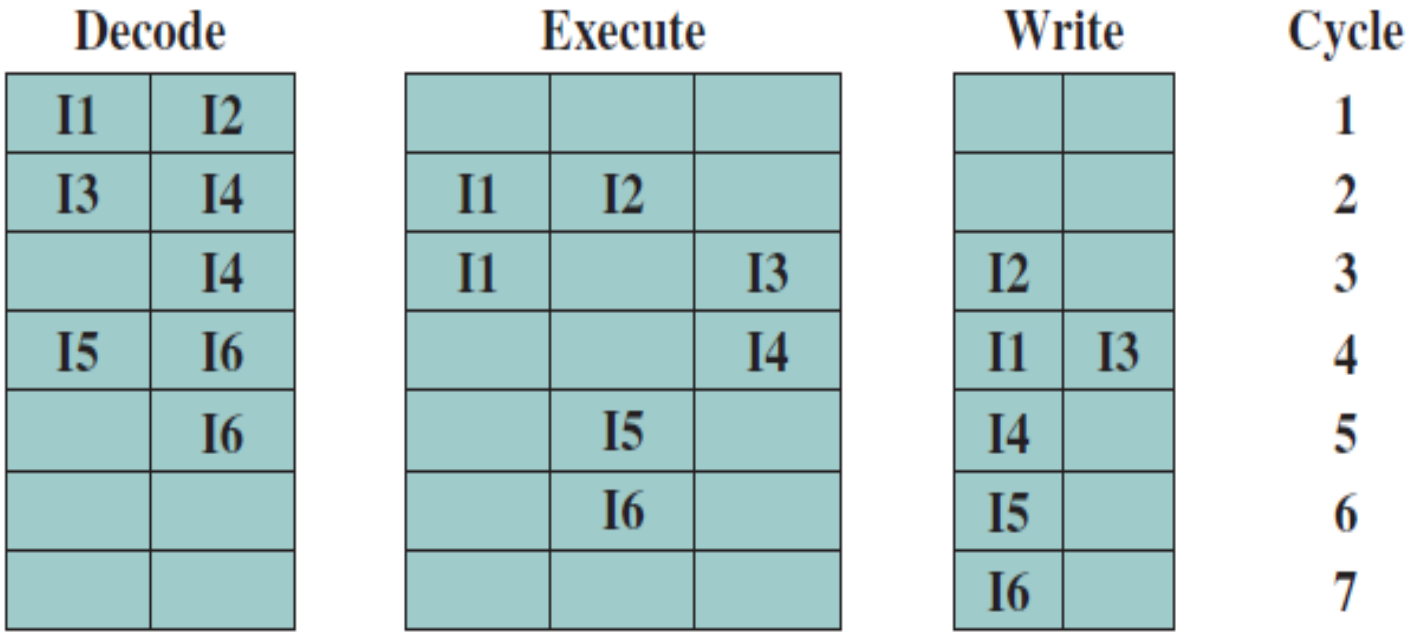
(a) In-order issue and in-order completion

In-order issue with out-of-order completion

- Out-of-order completion is **used in scalar RISC processors** to improve the performance of instructions that require multiple cycles.

- With out-of-order completion, any number of instructions may be in the execution stage at any one time, up to the maximum degree of machine parallelism across all functional units.

- Instruction issuing is troubled by a Resource Conflict, a Data Dependency, or a Procedural Dependency.



(b) In-order issue and out-of-order completion

- One more dependency is **Output Dependency (write after write [WAW] dependency),** arises.

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

- Out-of-order completion requires more complex instruction issue logic than in-order completion. In addition, it is more difficult to deal with instruction interrupts and exceptions.

Out-of-order issue with out-of-order completion

- With in-order issue, the processor will only decode instructions up to the point of a dependency or conflict.
- No additional instructions are decoded until the conflict is resolved.
- As a result, the processor cannot look ahead of the point of conflict to subsequent instructions that may be independent of those already in the pipeline and that may be usefully introduced into the pipeline.

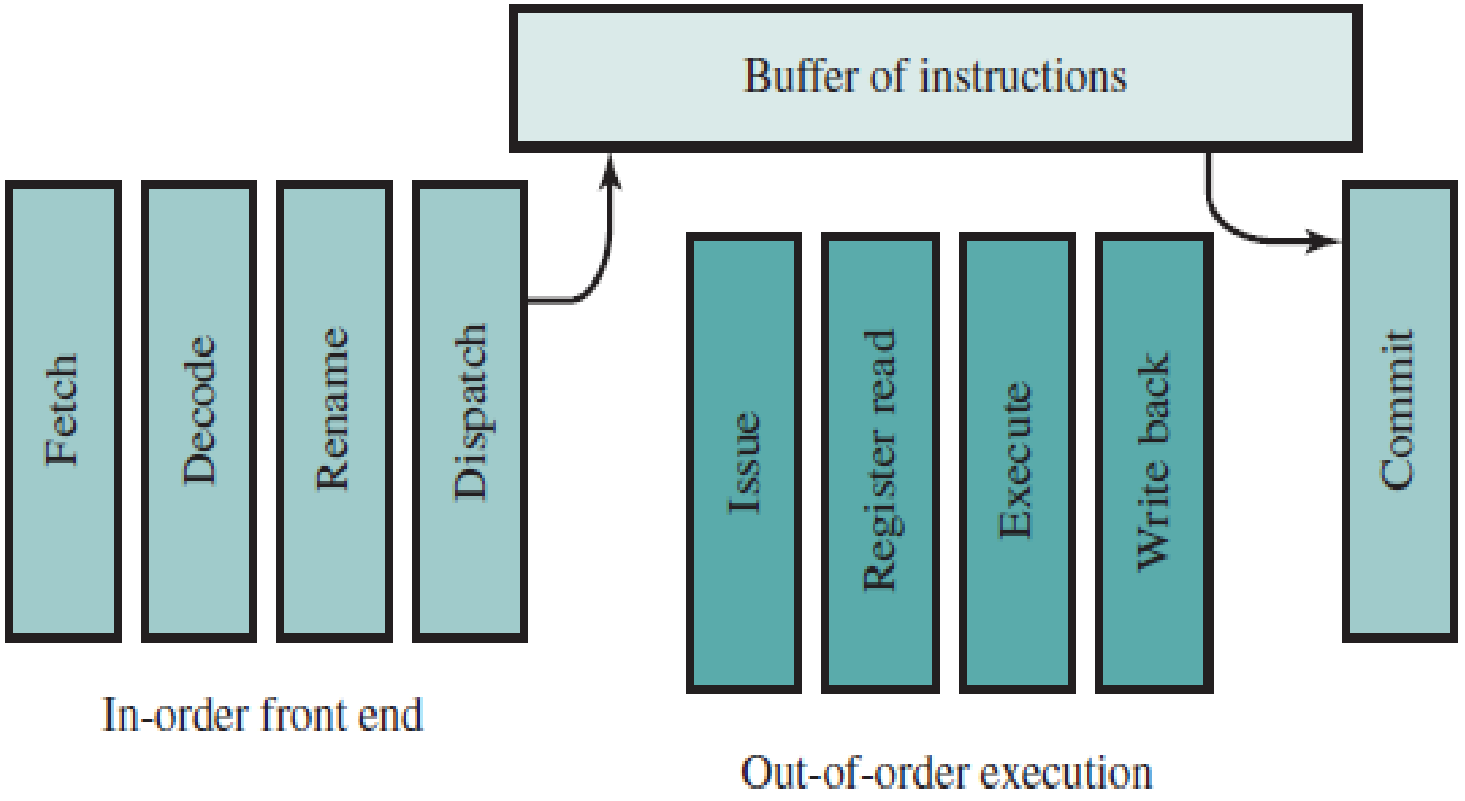
- To allow **out-of-order issue, it is necessary to decouple the decode and execute** stages of the pipeline. This is done with a buffer referred to as an **Instruction Window.**

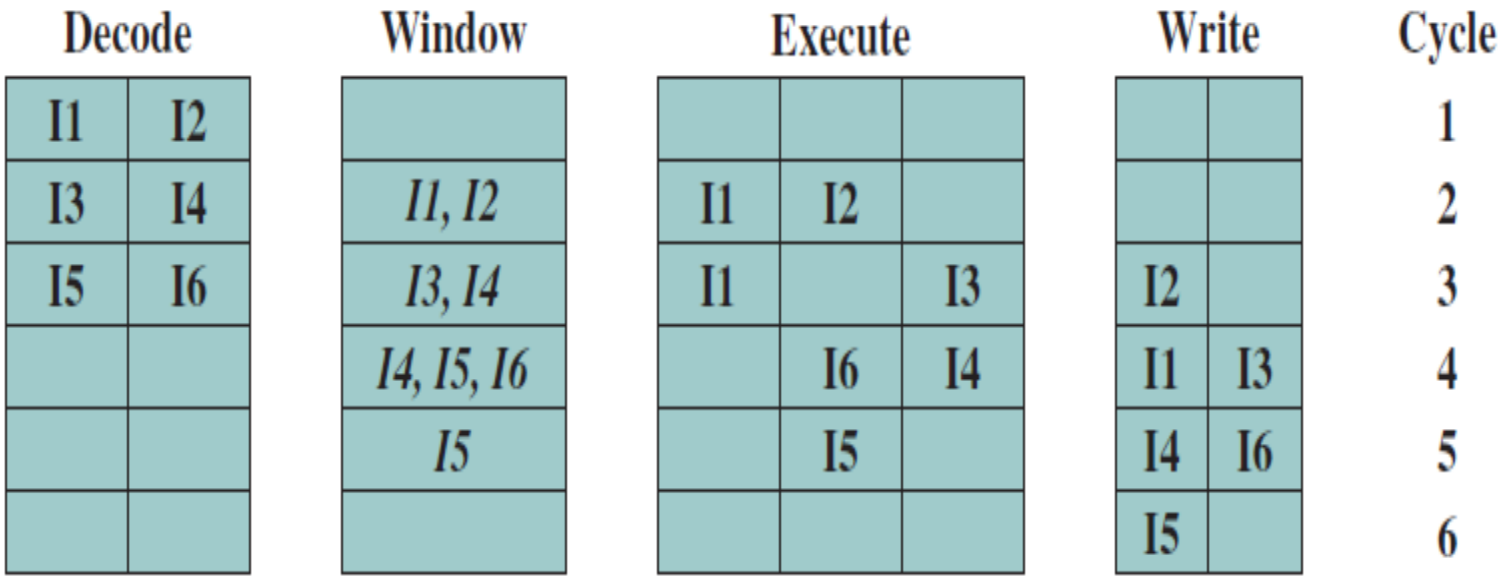
- With this organization, **after a processor has finished decoding an instruction, it is placed in the instruction window.** As long as this buffer is not full, the processor can continue to fetch and decode new instructions.

- When a functional unit becomes available in the execute stage, an instruction from the instruction window may be issued to the execute stage.

- Any instruction may be issued, provided that **(1)** it needs the particular functional unit that is available, and **(2)** no conflicts or dependencies block this instruction.

Organization for Out-of-Order Issue with Out-of-Order Completion





(c) Out-of-order issue and out-of-order completion

- The problem in this approach is **Antidependency (Write After Read [WAR] dependency),** arises.

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

- Disadvantage of Instruction Issue Policy:
  - May occur **WAW, WAR** or **RAW** dependencies.
  - Which may lead to **Storage Conflicts**.
  - [**Storage Conflict:** Multiple instructions are competing for the use of the same register locations. Which reduces performance of pipeline.]

3. Register Renaming

- One of the **best technique to resolve Storage Conflicts**.
- **Registers are allocated dynamically by the processor hardware**, and they are associated with the values needed by instructions at various points in time.
- When an instruction executes, for a destination operand, a new register is allocated for that value.
- Subsequent instructions that access that value as a source operand in that register must go through a renaming process.

Example

I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4

After Register Renaming

$I1: R3_b \leftarrow R3_a \text{ op } R5_a$
$I2: R4_b \leftarrow R3_b + 1$
$I3: R3_c \leftarrow R5_a + 1$
$I4: R7_b \leftarrow R3_c \text{ op } R4_b$

4. Machine Parallelism

- To improve the machine parallelism, it is probably not worthwhile to add functional units without **Register Renaming**.

5. Branch Prediction

- Any high-performance pipelined machine must address the issue of **“Dealing with Branches”.**

6. Superscalar Implementation

- We can make some general comments about the processor hardware required for the superscalar approach.

1. Instruction fetch and Decode function require the use of **multiple pipeline fetch and decode stages,** and **branch prediction logic**.

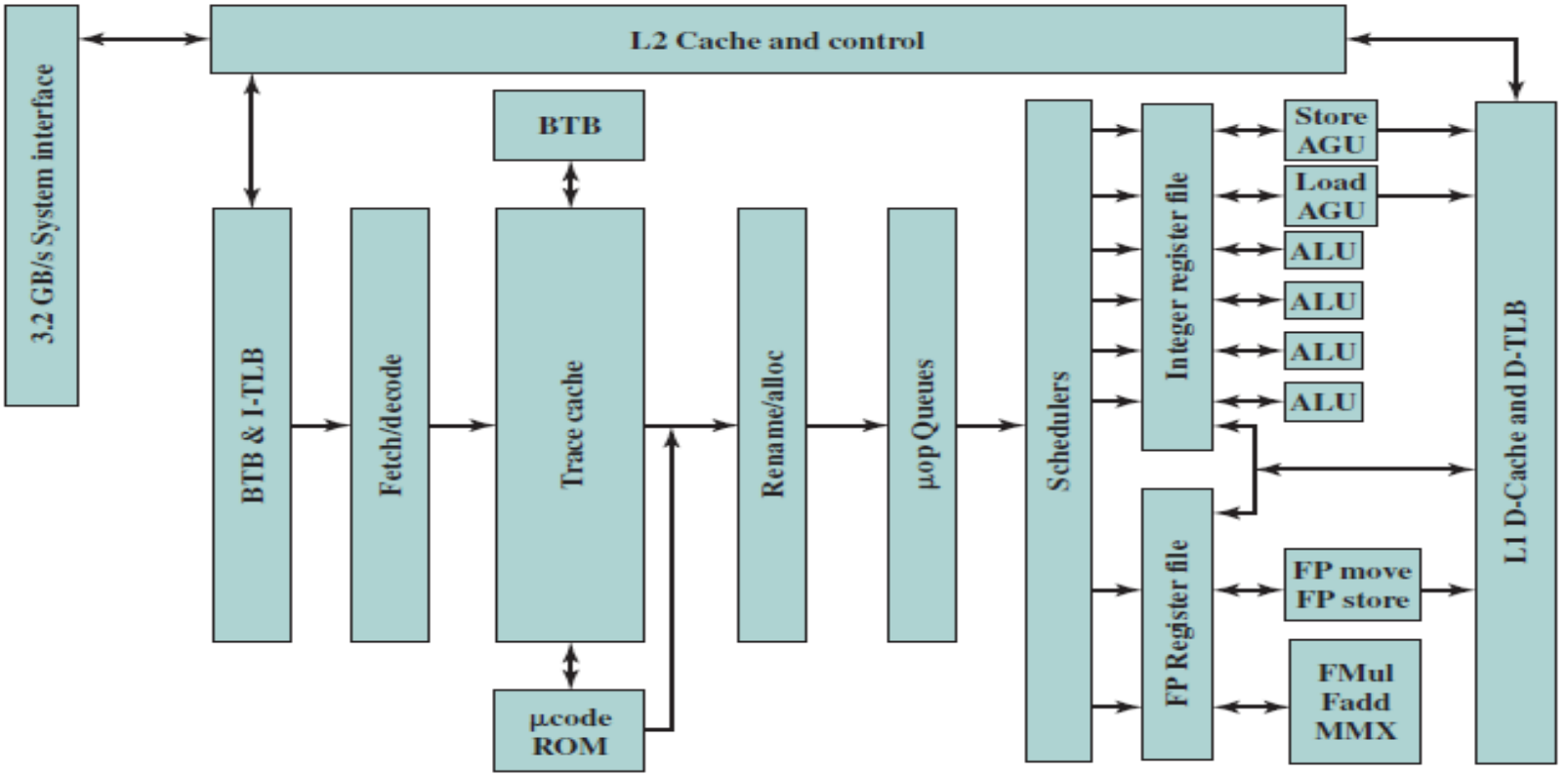
2. Logic is needed to determine **True Dependencies**.

3. Mechanisms for **issuing multiple instructions in parallel.**

4. **Resources for parallel execution of multiple instructions are needed**. Which includes multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references.

5. Mechanisms for **committing the process state in correct order.**

Case Study: Pentium 4



AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer

- Although the concept of superscalar design is generally associated with the RISC architecture, the same **superscalar principles can be applied to a CISC machine. Eg. Pentium.**

- The 80486 introduced the first pipelined x86 processor but with no superscalar elements.

- The original Pentium had a modest superscalar component, **consisting of the use of two separate integer execution units.**

- The Pentium Pro introduced a full-blown **superscalar design with out-of-order execution.** Subsequent x86 models have refined and enhanced the superscalar design.

- The operation of the Pentium 4 can be summarized as follows:

1. The **processor fetches instructions from memory** in the order of the static program.
2. Each instruction is translated into one or more fixed-length RISC instructions, known as **micro-operations,** or **micro-ops.**
3. The processor executes the micro-ops on a superscalar pipeline organization, so that the micro-ops may be **executed out of order**.
4. The **processor commits the results** of each micro-op execution to the processor's register set in the order of the original program flow.