# COM/BLM 376 Computer Architecture

# Chapter 14 Processor Structure and Function

Asst. Prof. Dr. Gazi Erkan BOSTANCI

ebostanci@ankara.edu.tr

Slides are mainly based on

Computer Organization and Architecture: Designing for Performance by William Stallings, 9th Edition, Prentice Hall

## Outline

1. Processor Organization
2. Register Organization
3. Instruction Cycle
4. Instruction Pipelining

We begin with a summary of processor organization. Registers, which form the internal memory of the processor, are then analyzed. We are then in a position to return to the discussion of the instruction cycle.

A description of the instruction cycle and a common technique known as instruction pipelining completes our description.

## PROCESSOR ORGANIZATION

- To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:
    - **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.

To do these things, it should be clear that the processor needs to store some data temporarily. It must remember the location of the last 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.

In other words, the processor needs a small internal memory.

Figure is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. A similar interface would be needed for any of the interconnection structures.

You will recall that the major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data.

The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers*.

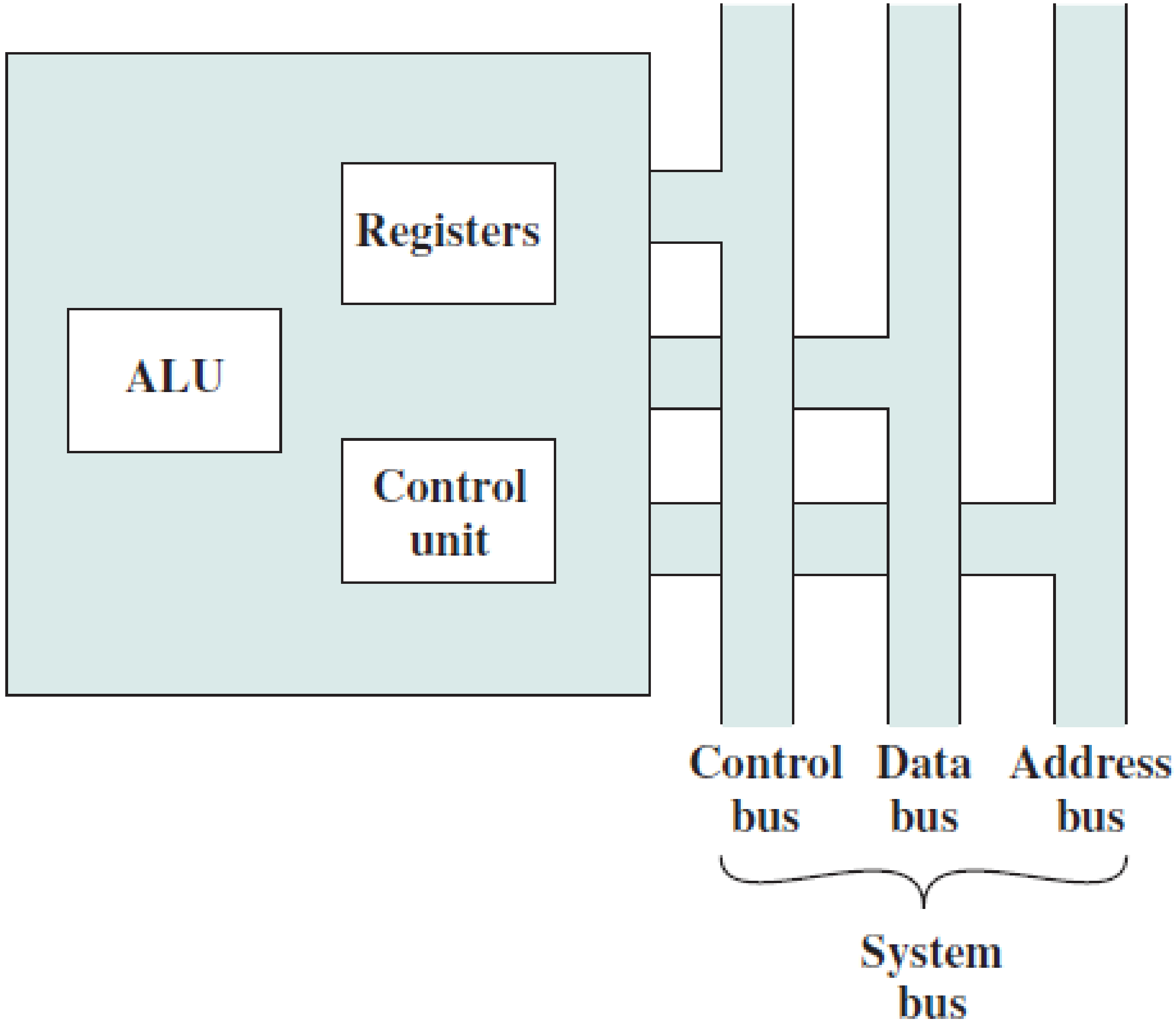
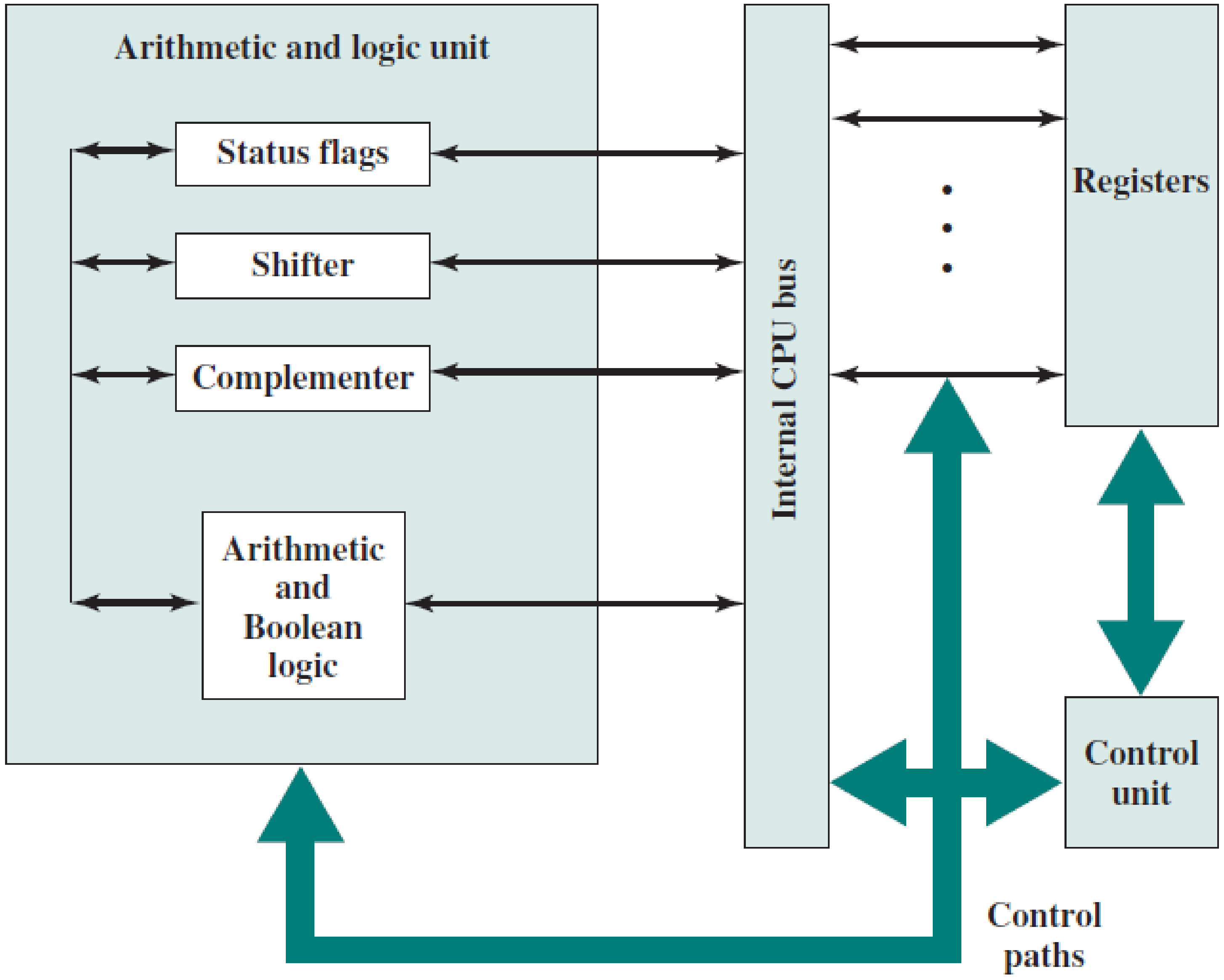


Figure gives a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled *internal processor bus*. This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory.

The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.



## REGISTER ORGANIZATION

As we discussed earlier, 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:

- **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.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible (e.g., x86), but on many it is not. For purposes of the following discussion, however, we will use these categories.

## User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes. We can characterize these in the following categories:

- General purpose
- Data
- Address
- Condition codes

**General-purpose registers** can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions.

- For example, there may be dedicated registers for floating-point and stack operations.

In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between 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.

**Address registers** may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers:** In a machine with segmented addressing, a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
- **Index registers:** These are used for indexed addressing and may be auto-indexed.
- **Stack pointer:** If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack. This allows implicit addressing; that is, push, pop, and other stack instructions need not contain an explicit stack operand.

## There are several design issues to be addressed here.

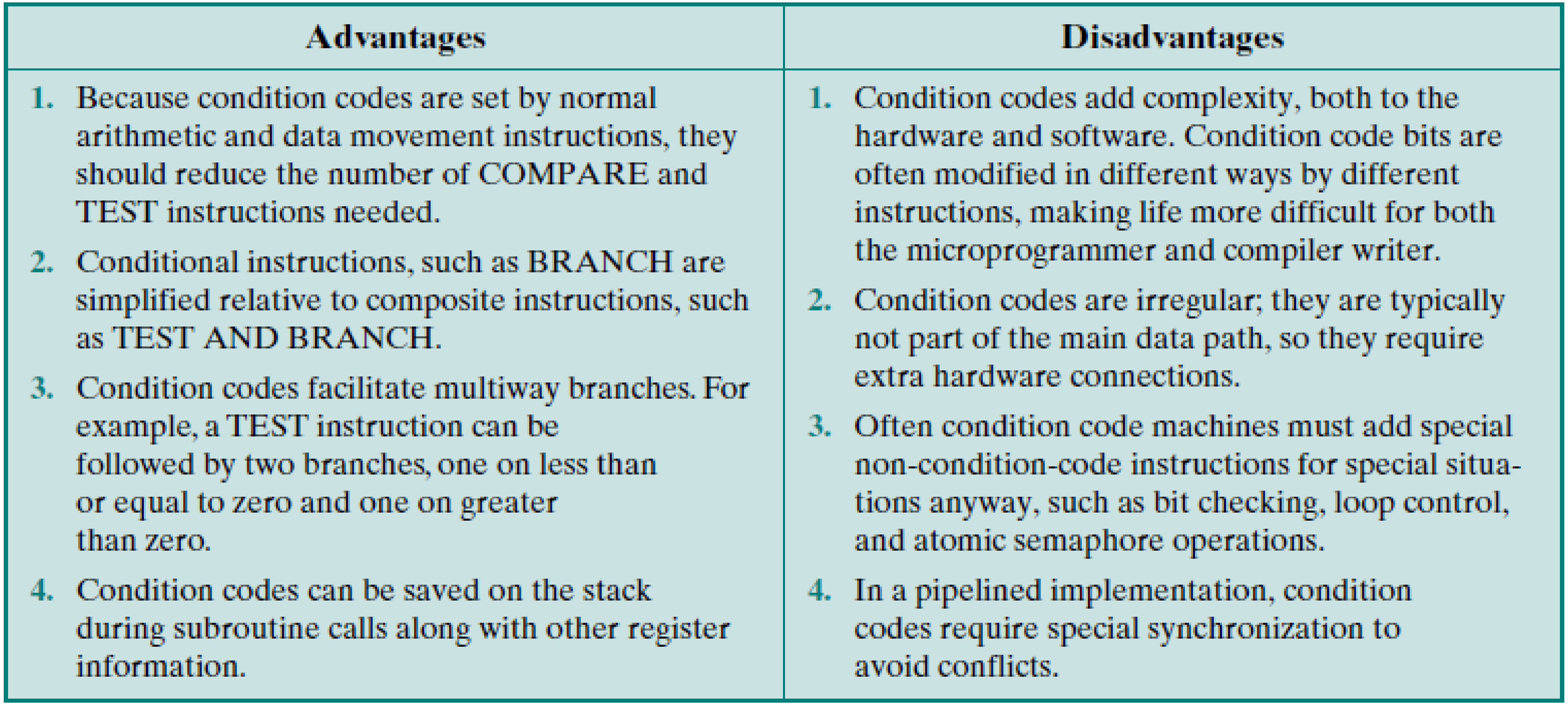
- An important issue is whether to use completely general-purpose registers or to specialize their use.
    - This affects instruction set design. With the use of specialized registers, it can generally be implicit in the opcode which type of register a certain operand specifier refers to. The operand specifier must only identify one of a set of specialized registers rather than one out of all the registers, thus saving bits. On the other hand, this specialization limits the programmer's flexibility.
- Another design issue is the number of registers, either general purpose or data plus address, to be provided.
    - Again, this affects instruction set design because more registers require more operand specifier bits. As we previously discussed, somewhere between 8 and 32 registers appears optimum. Fewer registers result in more memory references; more registers do not noticeably reduce memory references. However, a new approach, which finds advantage in the use of hundreds of registers.
- Finally, there is the issue of register length.
    - Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

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. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result.

In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

# Table lists the advantages and disadvantages of 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. |

In some machines, a subroutine call will result in the automatic saving of all user-visible registers, to be restored on return. The processor performs the saving and restoring as part of the execution of call and return instructions. This allows each subroutine to use the user-visible registers independently.

On other machines, it is the responsibility of the programmer to save the contents of the relevant user-visible registers prior to a subroutine call, by including instructions for this purpose in the program.

## Control and Status Registers

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Of course, different machines will have different register organizations and use different terminology. We list here a reasonably complete list of register types, with a brief description.

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.

Not all processors have internal registers designated as MAR and MBR, but some equivalent buffering mechanism is needed whereby the bits to be transferred to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Typically, the processor updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC.

The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed. Data are exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between the processor and memory. Within the processor, data must be presented to the ALU for processing. The ALU may have direct access to the MBR and user-visible registers. Alternatively, there may be additional buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user-visible registers.

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.

A number of other registers related to status and control might be found in a particular processor design.

- There may be a pointer to a block of memory containing additional status information (e.g., process control blocks).
- In machines using vectored interrupts, an interrupt vector register may be provided.
- If a stack is used to implement certain functions (e.g., subroutine call), then a system stack pointer is needed.
- A page table pointer is used with a virtual memory system.
- Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization:

- One key issue is operating system support. Certain types of control information are of specific utility to the operating system. If the processor designer has a functional understanding of the operating system to be used, then the register organization can to some extent be tailored to the operating system.
- Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much in memory. The usual trade-off of cost versus speed arises.

## INSTRUCTION CYCLE

Earlier, we described the processor’s instruction cycle . To recall, 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.

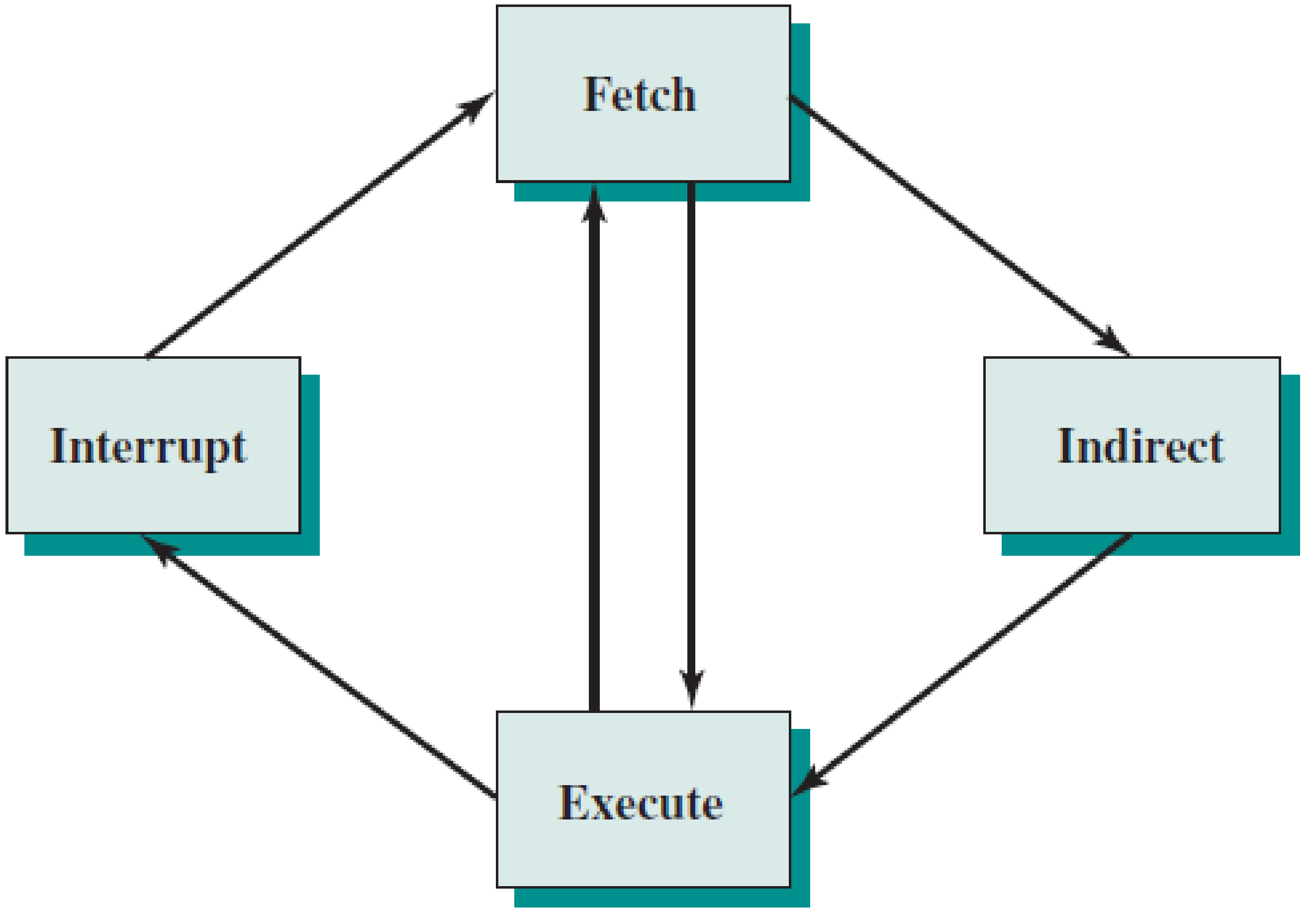
We are now in a position to elaborate somewhat on the instruction cycle. First, we must introduce one additional stage, known as the indirect cycle.

## The Indirect Cycle

We have seen that the execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory accesses are required.

We can think of the fetching of indirect addresses as one more instruction stages. The result is shown in the figure. 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.

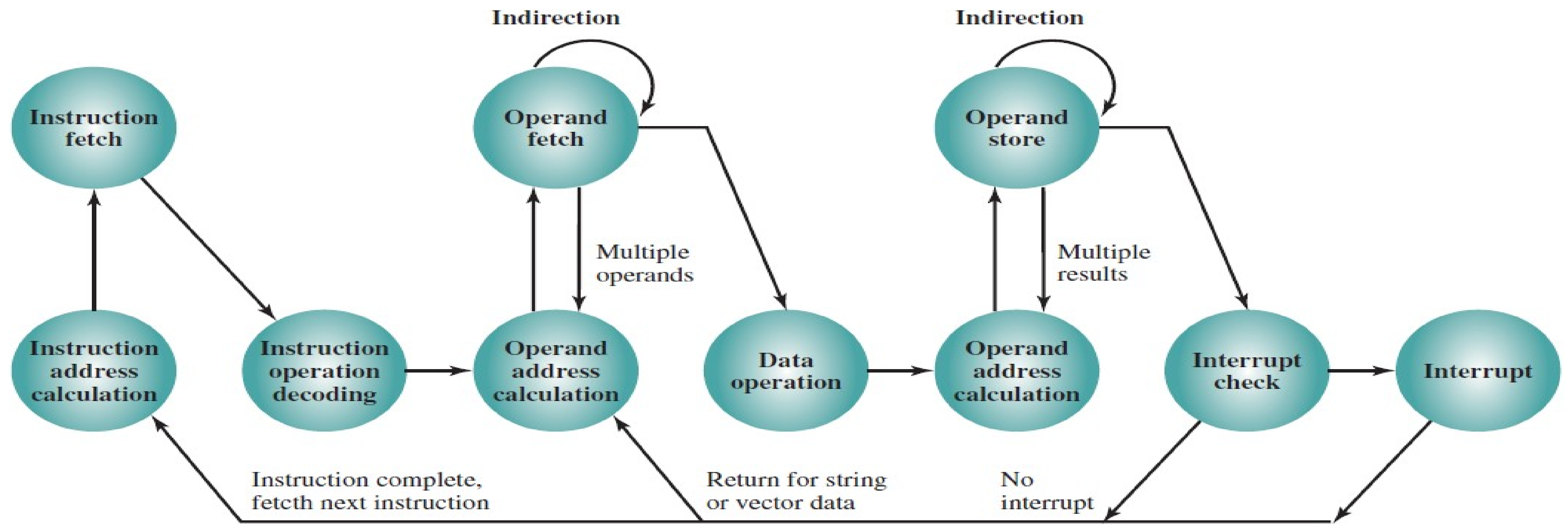
Following execution, an interrupt may be processed before the next instruction fetch.



Another way to view this process is shown in the figure. This illustrates more correctly the nature of the instruction cycle. Once an instruction is fetched, its operand specifiers must be identified.

Each input operand in memory is then fetched, and this process may require indirect addressing. Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory.

Question in Final


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

## Data Flow

The exact sequence of events during an instruction cycle depends on the design of the processor. We can, however, indicate in general terms what must happen.

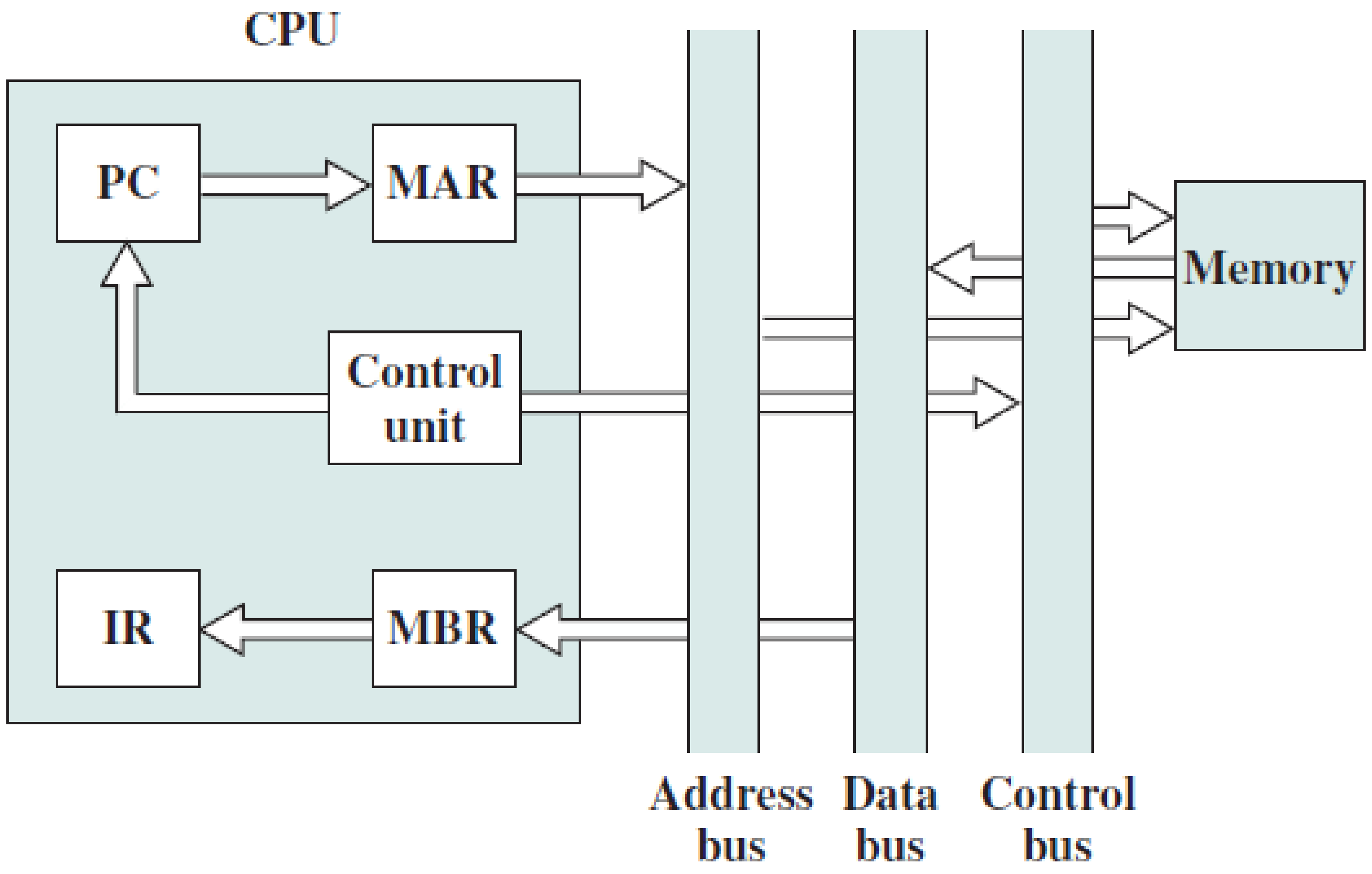
Let us assume that a processor that employs a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), and an instruction register (IR).

During the *fetch cycle*, an instruction is read from memory.

Figure 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.



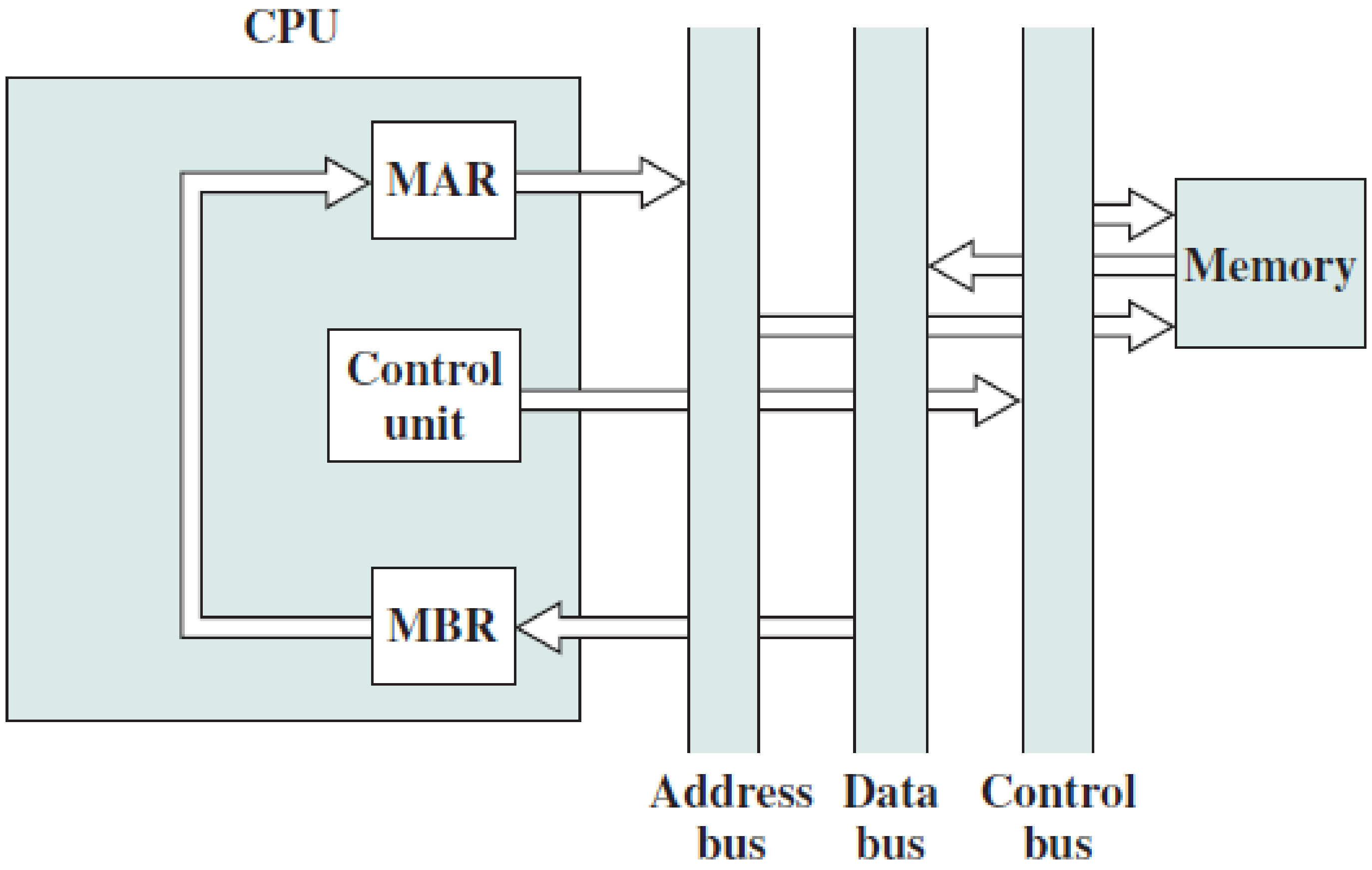
MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

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 the figure, this is a simple cycle. The rightmost *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.

The fetch and indirect cycles are simple and predictable.

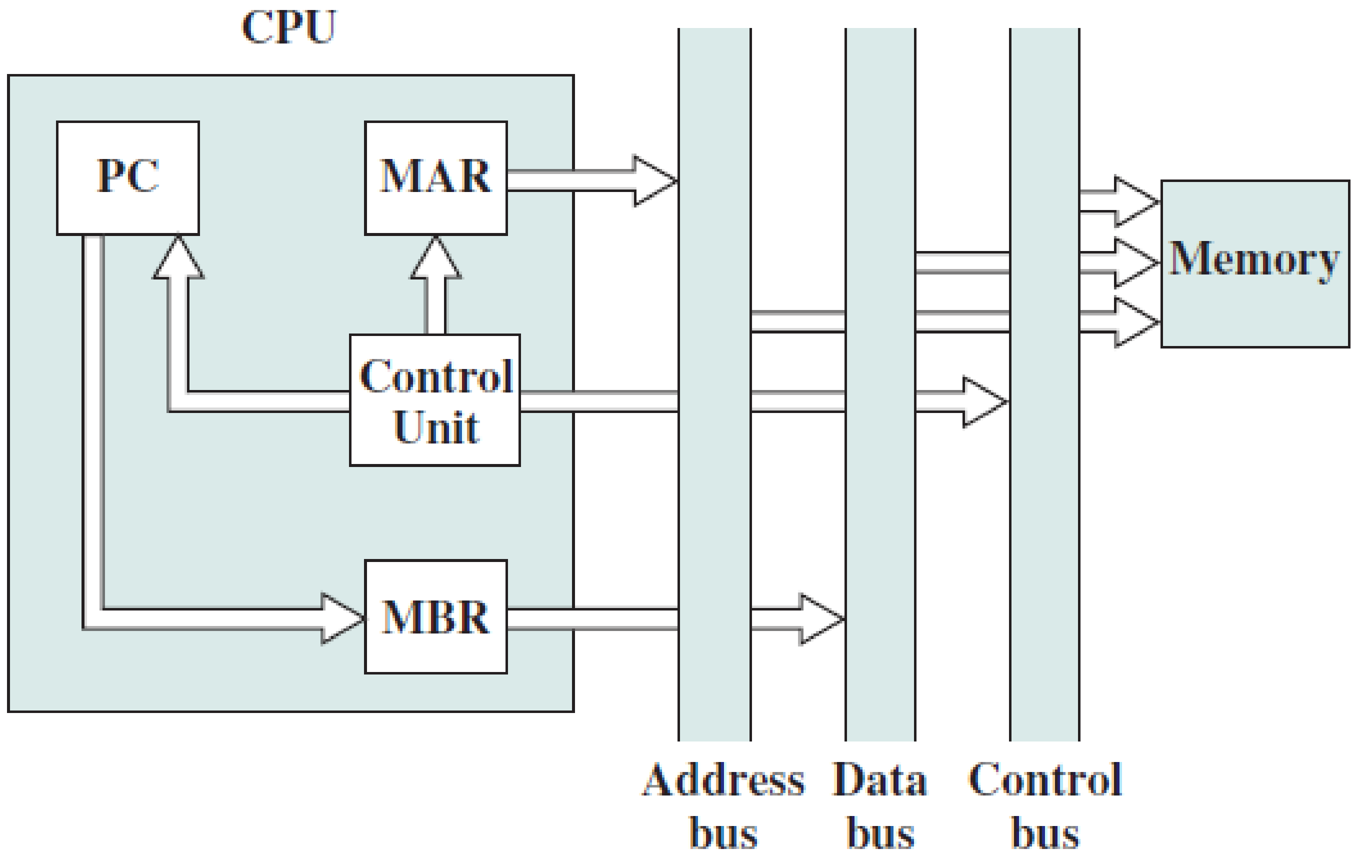
The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU.



Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable .

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.



## INSTRUCTION PIPELINING

As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry. In addition, organizational enhancements to the processor can improve performance.

We have already seen some examples of this, such as the use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach, which is quite common, is instruction pipelining.

## Pipelining Strategy

Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously.

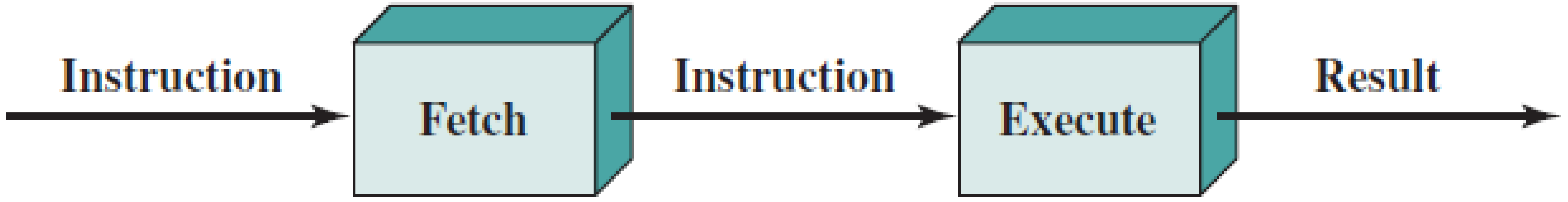
This process is also referred to as *pipelining*, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. The instruction state diagram, for example, breaks the instruction cycle up into 10 tasks, which occur in sequence. Clearly, there should be some opportunity for pipelining.

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one.

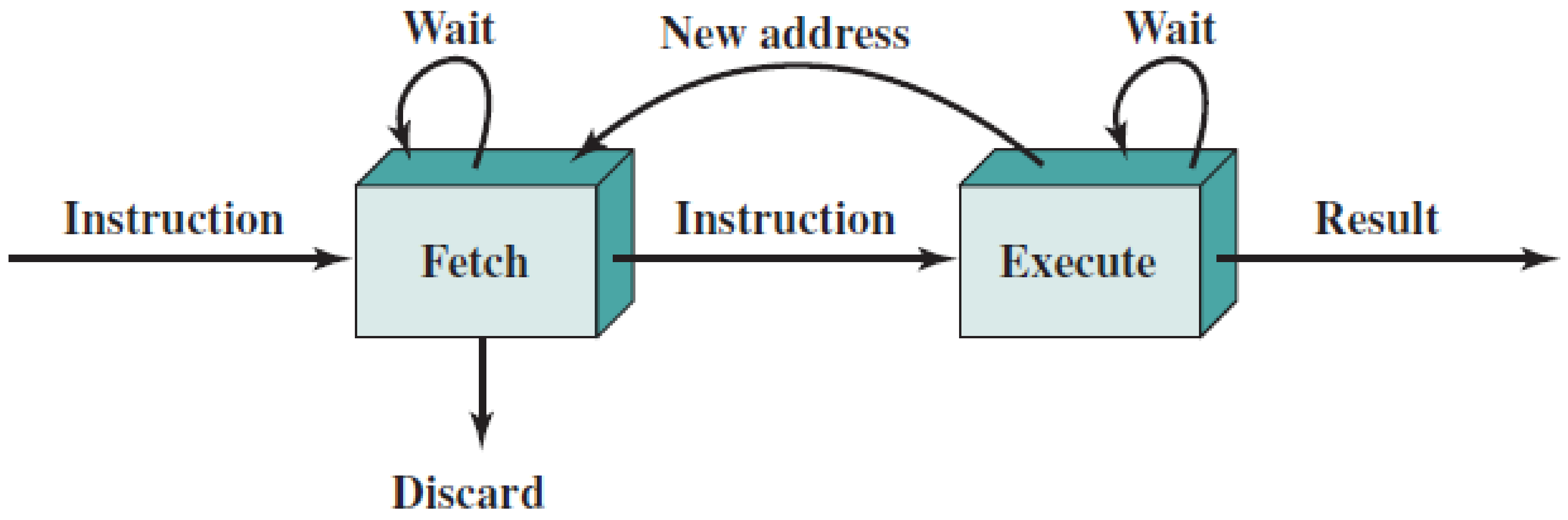
Figure depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any 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 below), we will see that this doubling of execution rate is unlikely for two reasons:

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 it can empty its buffer.
2. 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.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages.

Let us consider the following decomposition of the instruction processing.

**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.

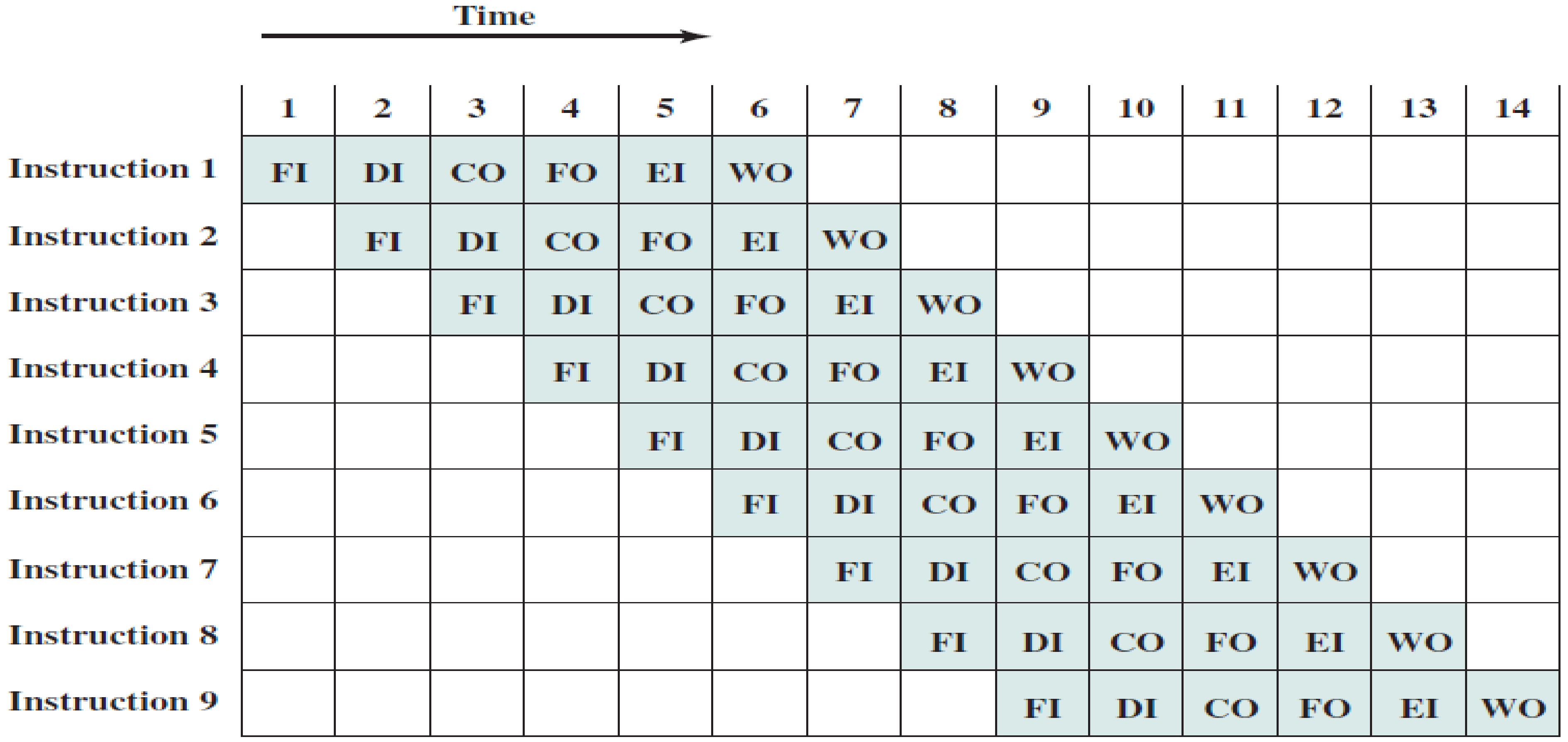
**Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.

**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.

With this decomposition, the various stages will be of more nearly equal duration.

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



| | 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 comments are in order:

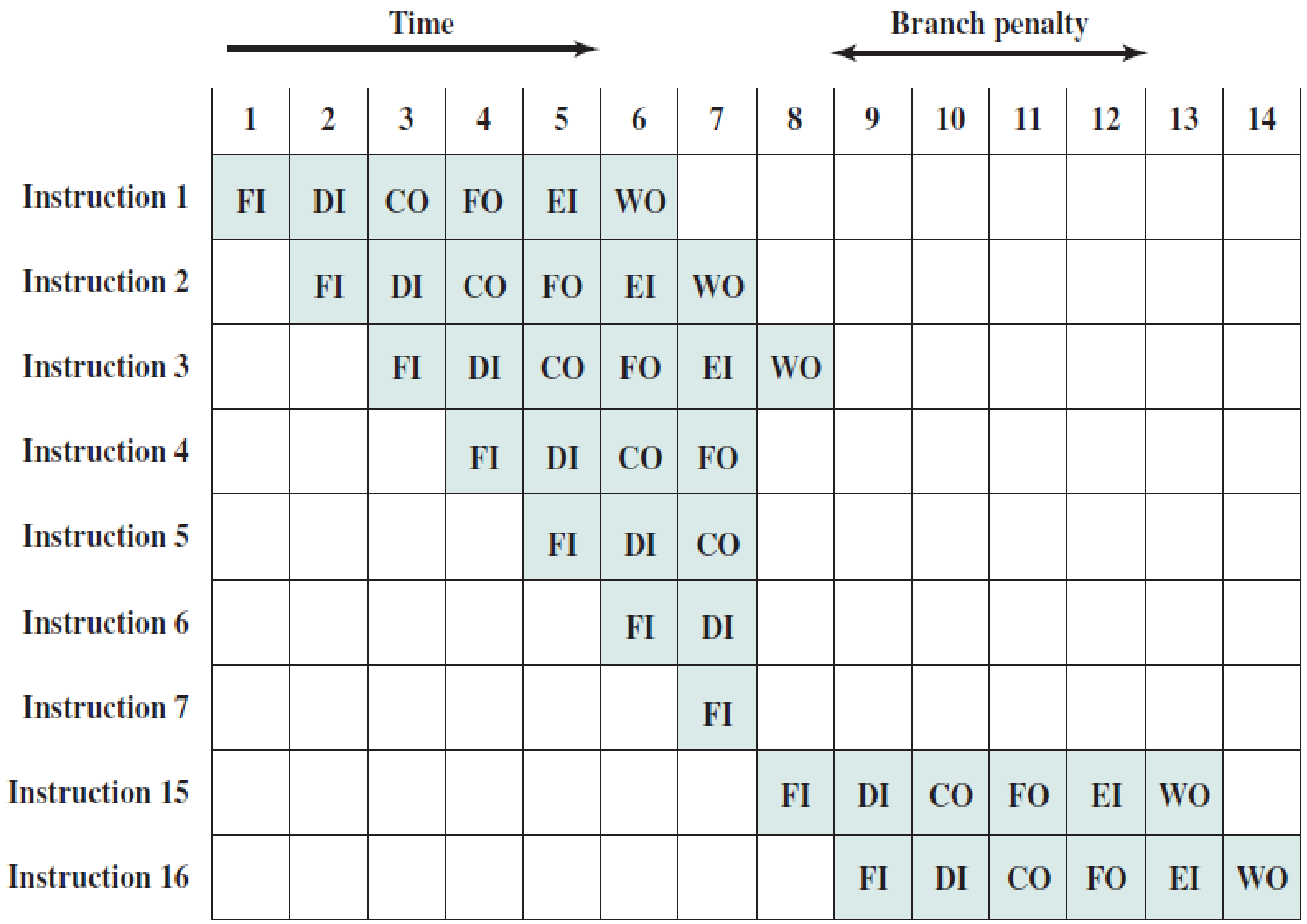
- The diagram assumes that each instruction goes through all six stages of the pipeline.
    - This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six 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.

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages, as discussed before for the two-stage pipeline.

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

Figure illustrates the effects of the conditional branch, using the same program as the figure above.

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.

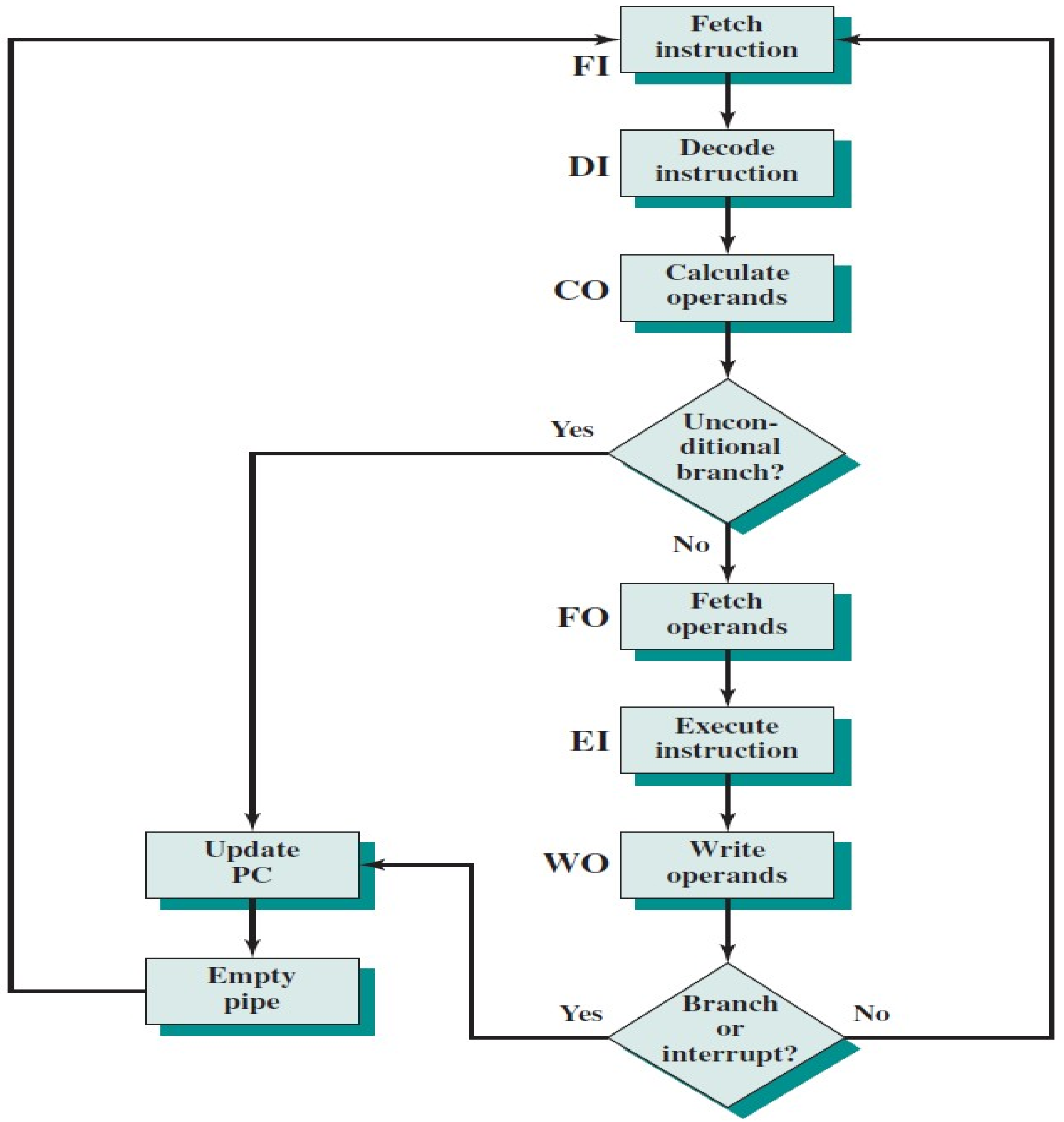

Time
Branch penalty

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

In the previous figure, the branch was not taken, and we get the full performance benefit of the enhancement.

In the figure above, the branch is taken. This 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 indicates the logic needed for pipelining to account for branches and interrupts.



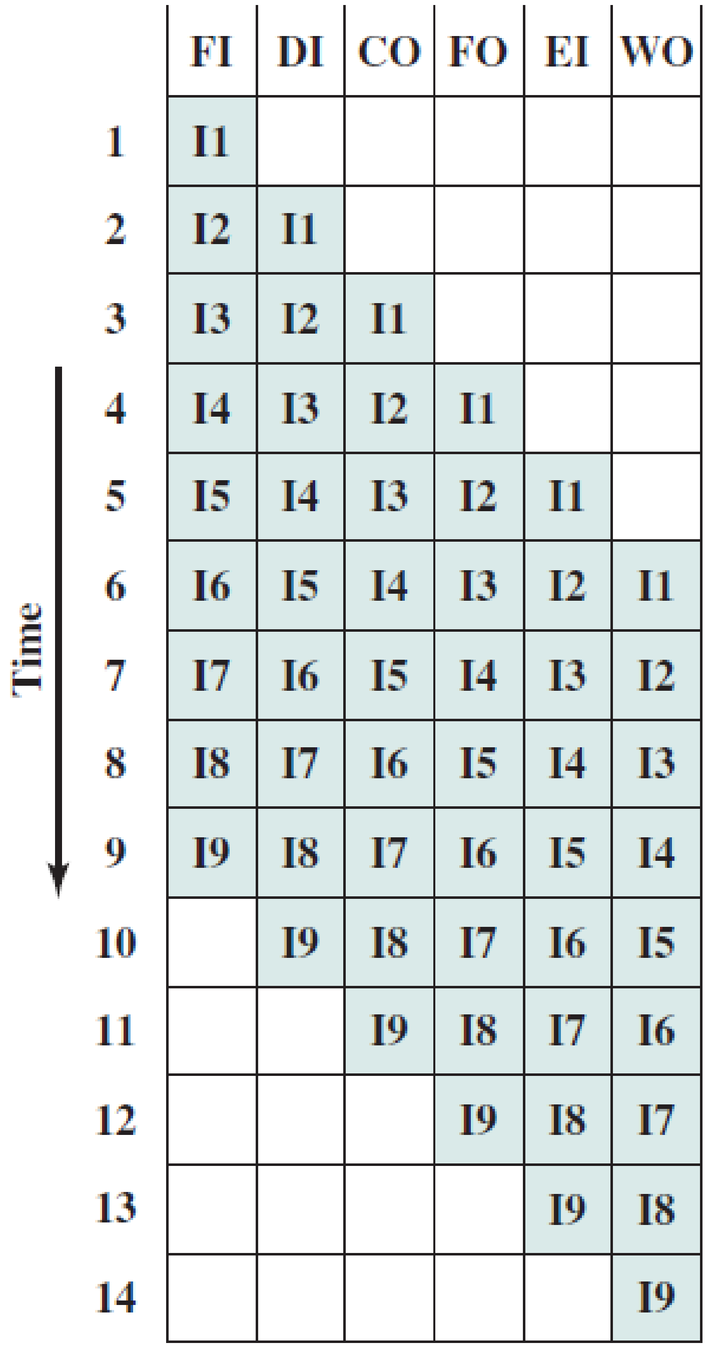
Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

To clarify pipeline operation, it might be useful to look at an alternative depiction. Previous figures show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction.

Figure below shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time.

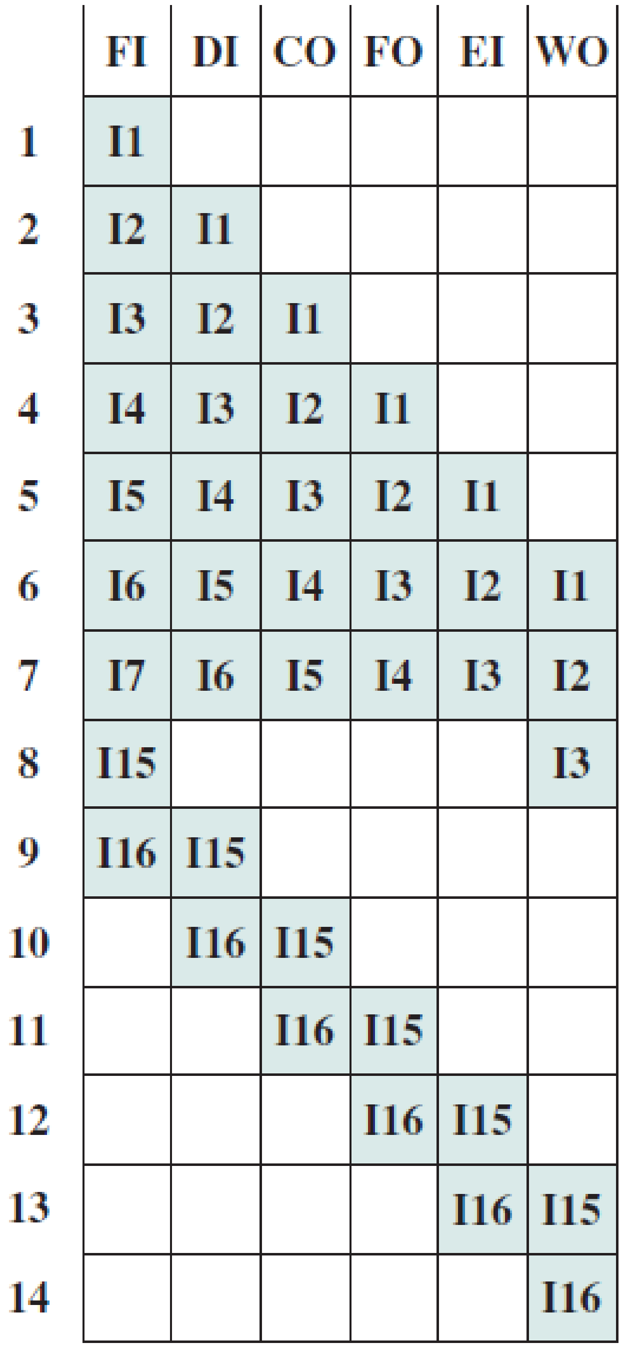
In figure (a), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed.

In Figure (b), 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.



|  | FI | DI | CO | FO | EI | WO |
|---|---|---|---|---|---|---|
| 1 | I1 |  |  |  |  |  |
| 2 | I2 | I1 |  |  |  |  |
| 3 | I3 | I2 | I1 |  |  |  |
| 4 | I4 | I3 | I2 | I1 |  |  |
| 5 | I5 | I4 | I3 | I2 | I1 |  |
| 6 | I6 | I5 | I4 | I3 | I2 | I1 |
| 7 | I7 | I6 | I5 | I4 | I3 | I2 |
| 8 | I8 | I7 | I6 | I5 | I4 | I3 |
| 9 | I9 | I8 | I7 | I6 | I5 | I4 |
| 10 |  | I9 | I8 | I7 | I6 | I5 |
| 11 |  |  | I9 | I8 | I7 | I6 |
| 12 |  |  |  | I9 | I8 | I7 |
| 13 |  |  |  |  | I9 | I8 |
| 14 |  |  |  |  |  | I9 |

(a) No branches



|  | FI | DI | CO | FO | EI | WO |
|---|---|---|---|---|---|---|
| 1 | I1 |  |  |  |  |  |
| 2 | I2 | I1 |  |  |  |  |
| 3 | I3 | I2 | I1 |  |  |  |
| 4 | I4 | I3 | I2 | I1 |  |  |
| 5 | I5 | I4 | I3 | I2 | I1 |  |
| 6 | I6 | I5 | I4 | I3 | I2 | I1 |
| 7 | I7 | I6 | I5 | I4 | I3 | I2 |
| 8 | I15 |  |  |  |  | I3 |
| 9 | I16 | I15 |  |  |  |  |
| 10 |  | I16 | I15 |  |  |  |
| 11 |  |  | I16 | I15 |  |  |
| 12 |  |  |  | I16 | I15 |  |
| 13 |  |  |  |  | I16 | I15 |
| 14 |  |  |  |  |  | I16 |

(b) With conditional branch

From the preceding discussion, it might appear that the greater the number of stages in the pipeline, the faster the execution rate. Some of the IBM S/360 designers pointed out two factors that frustrate this seemingly simple pattern for high-performance design, and they remain elements that designer must still consider:

1. At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction. This is significant when sequential instructions are logically dependent, either through heavy use of branching or through memory access dependencies.
2. 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.

Another consideration is latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.

Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

## Pipeline Performance

Now, we develop some simple measures of pipeline performance and relative speedup. The cycle time *t* of an **instruction pipeline** is the time needed to advance a set of instructions one stage through the pipeline; each column in the initial set of figures above represents one cycle time. The cycle time can be determined as

$$\tau = \max_i [\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

where

- $t_i$ = time delay of the circuitry in the *i*th stage of the pipeline
- $t_m$ = 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

In general, the time delay *d* is equivalent to a clock pulse and $t_m \gg d$. Now suppose that *n* instructions are processed, with no branches. Let $T_{k,\,n}$ be the total time required for a pipeline with *k* stages to execute *n* instructions. Then

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

A total of *k* cycles are required to complete the execution of the first instruction, and the remaining *n* - 1 instructions require *n* - 1 cycles. This equation is easily verified from figure without the branch condition. The 9th instruction completes at time cycle 14:

$$14 = [6 + (9 - 1)]$$

Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is *k*t. The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

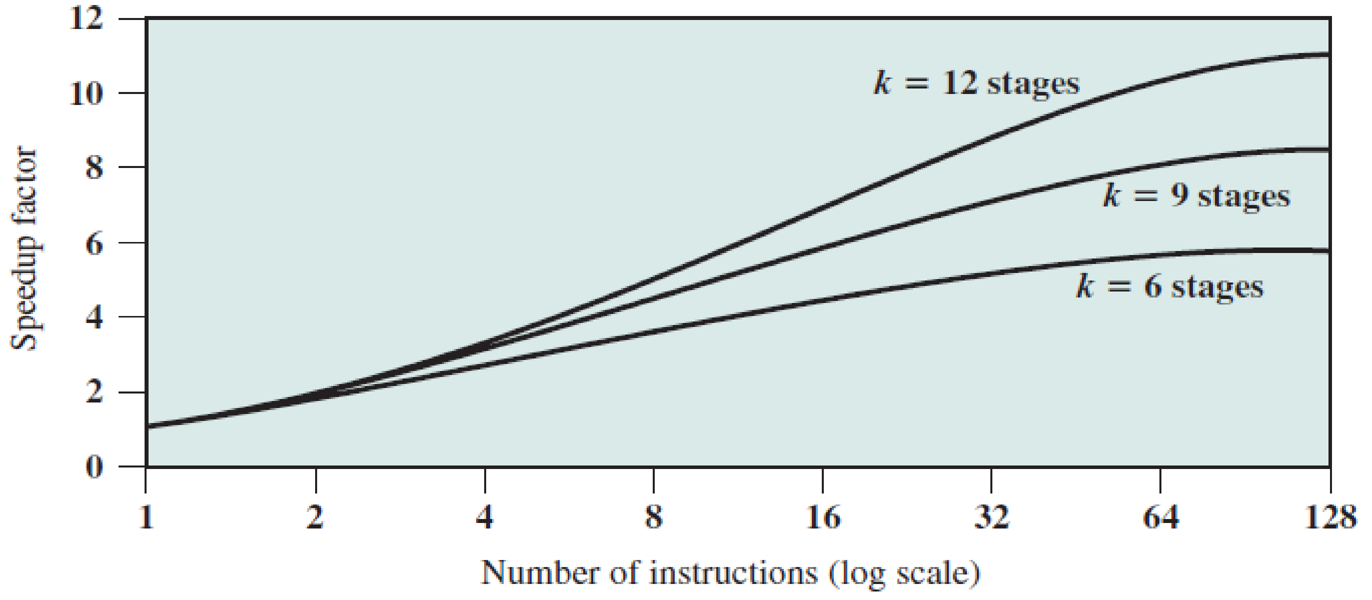
$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

Figure (a) plots the speedup factor as a function of the number of instructions that are executed without a branch. As might be expected, at the limit (*n* ->∞), we have a *k*-fold speedup.

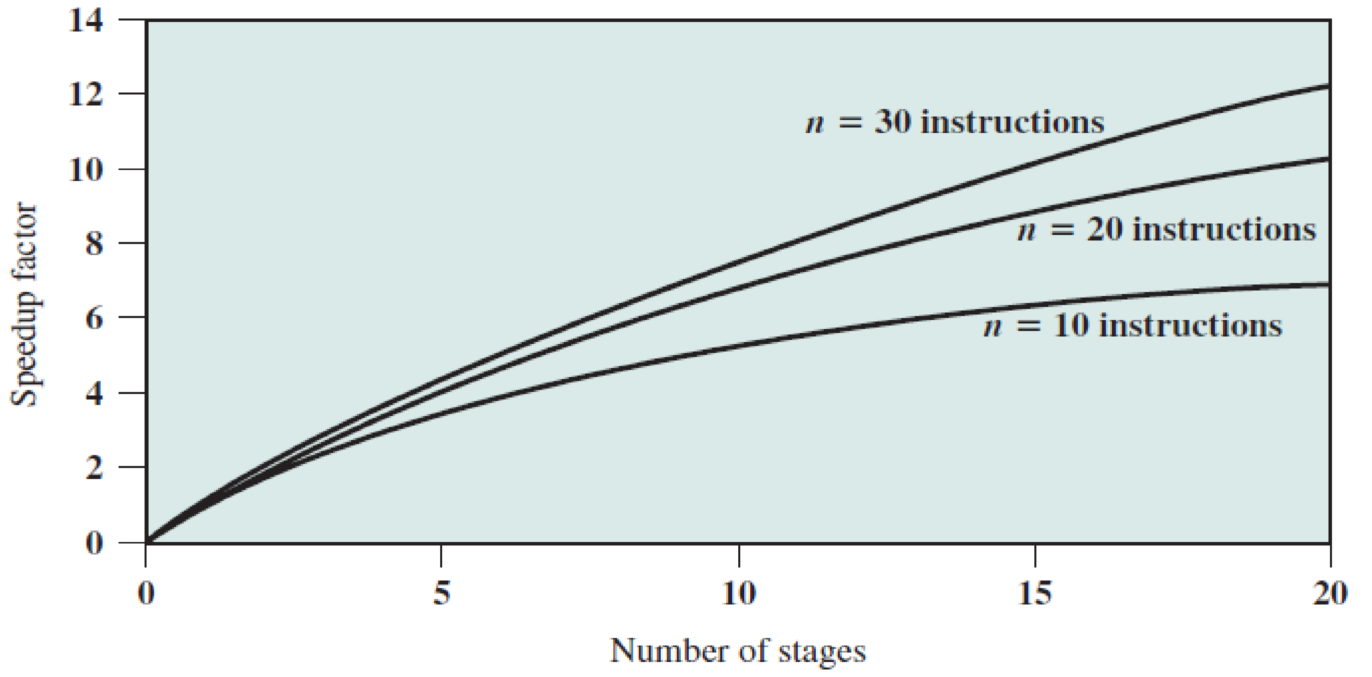
Figure (b) shows the speedup factor as a function of the number of stages in the instruction pipeline.

In this case, the speedup factor approaches the number of instructions that can be fed into the pipeline without branches. Thus, the larger the number of pipeline stages, the greater the potential for speedup.

However, as a practical matter, the potential gains of additional pipeline stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.



(a)



(b)

## Pipeline Hazards

In the previous subsection, we mentioned some of the situations that can result in less than optimal pipeline performance. In this subsection, we examine this issue in a more systematic way.

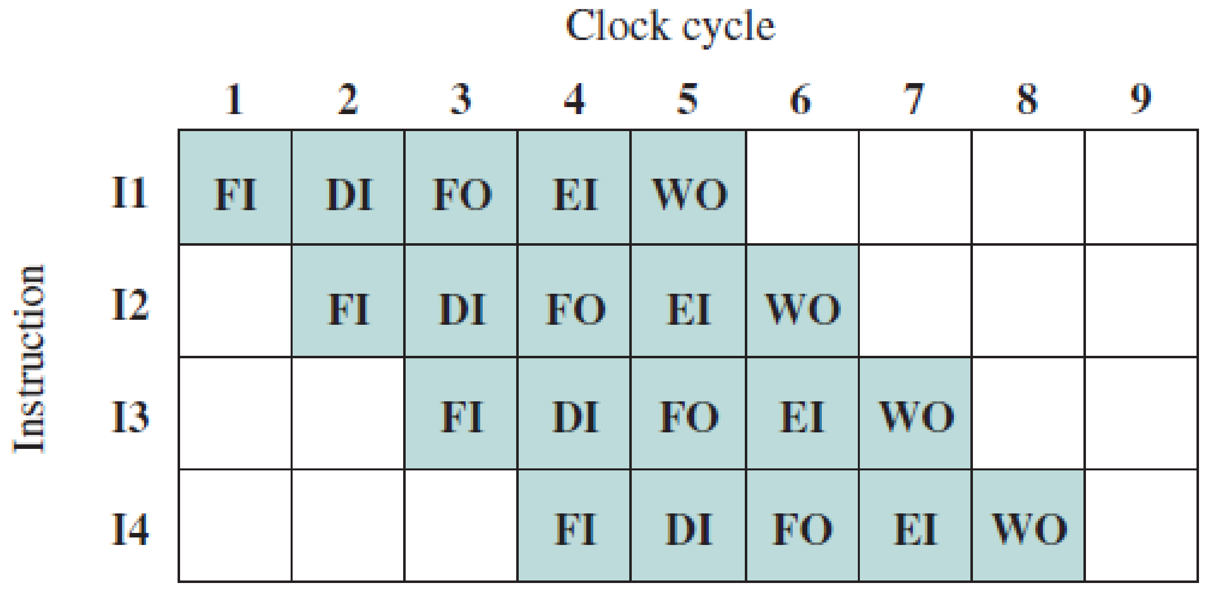
A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

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

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle.

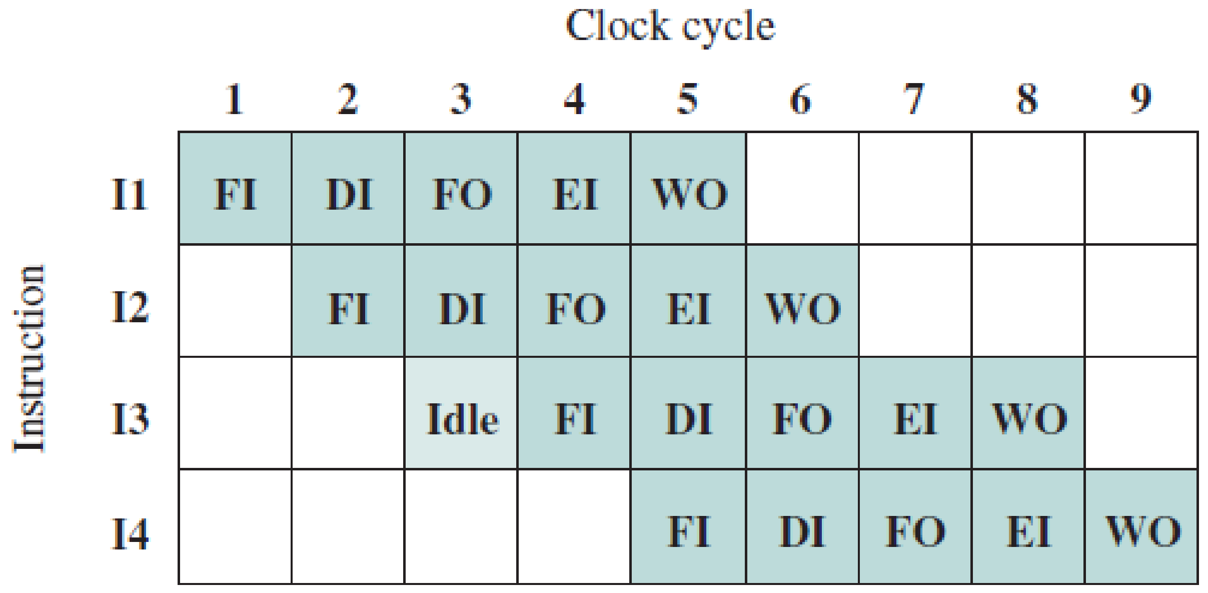
Figure (a) shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch.

This is illustrated in figure (b), 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.



| Instruction \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| I1 | FI | DI | FO | EI | WO | | | | |
| I2 | | FI | DI | FO | EI | WO | | | |
| I3 | | | FI | DI | FO | EI | WO | | |
| I4 | | | | FI | DI | FO | EI | WO | |

(a) Five-stage pipeline, ideal case



| Instruction \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| I1 | FI | DI | FO | EI | WO | | | | |
| I2 | | FI | DI | FO | EI | WO | | | |
| I3 | | | Idle | FI | DI | FO | EI | WO | |
| I4 | | | | | FI | DI | FO | EI | WO |

(b) I1 source operand in memory

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 solution 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 */

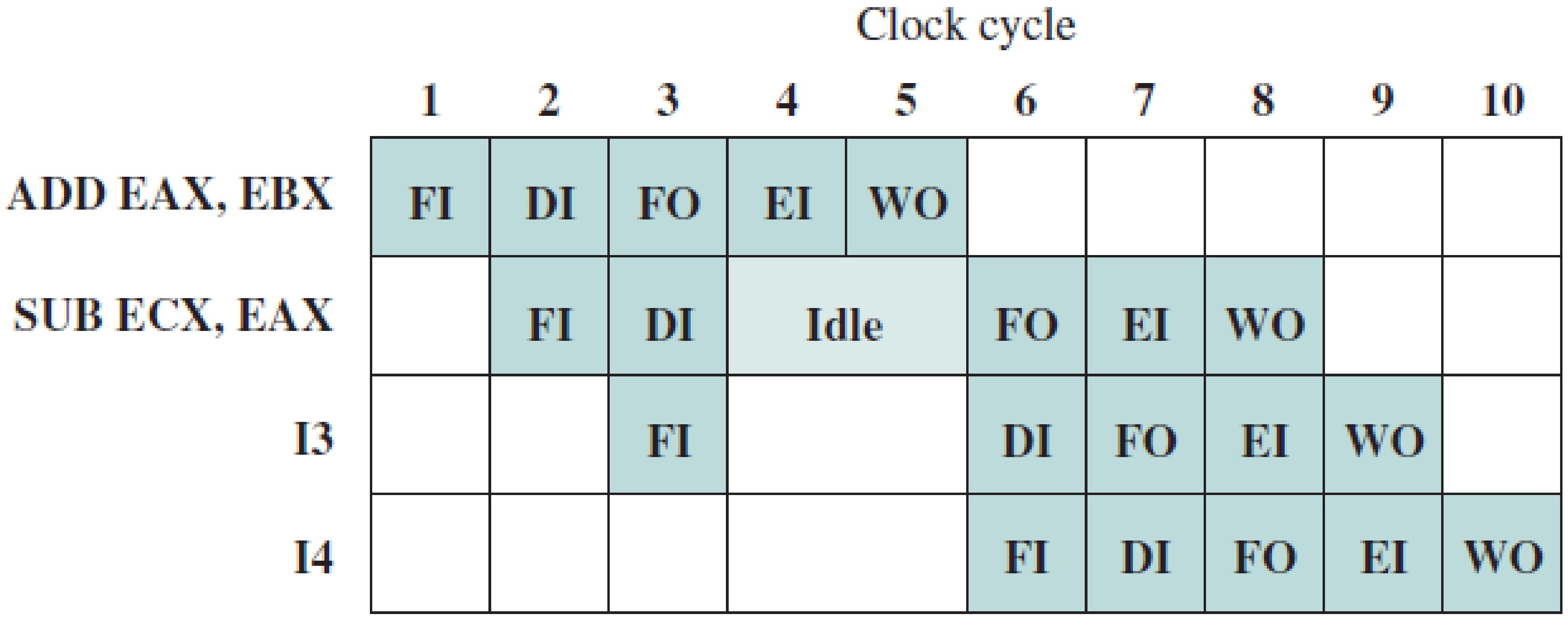
The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX. Figure below shows the pipeline behavior.

The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clocks cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage.

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.

The example above (shown in the figure) is a RAW hazard.



| | 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 |

***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.