
电子科技大学
格拉斯哥学院
Glasgow College, UESTC

# UESTC4019: Real-Time Computer Systems and Architecture

**Lecture 15**
**Processor Structure and Function**

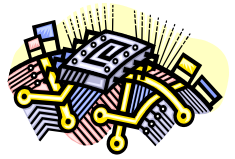
# Processor Organization (1 of 2)

**Processor Requirements:**

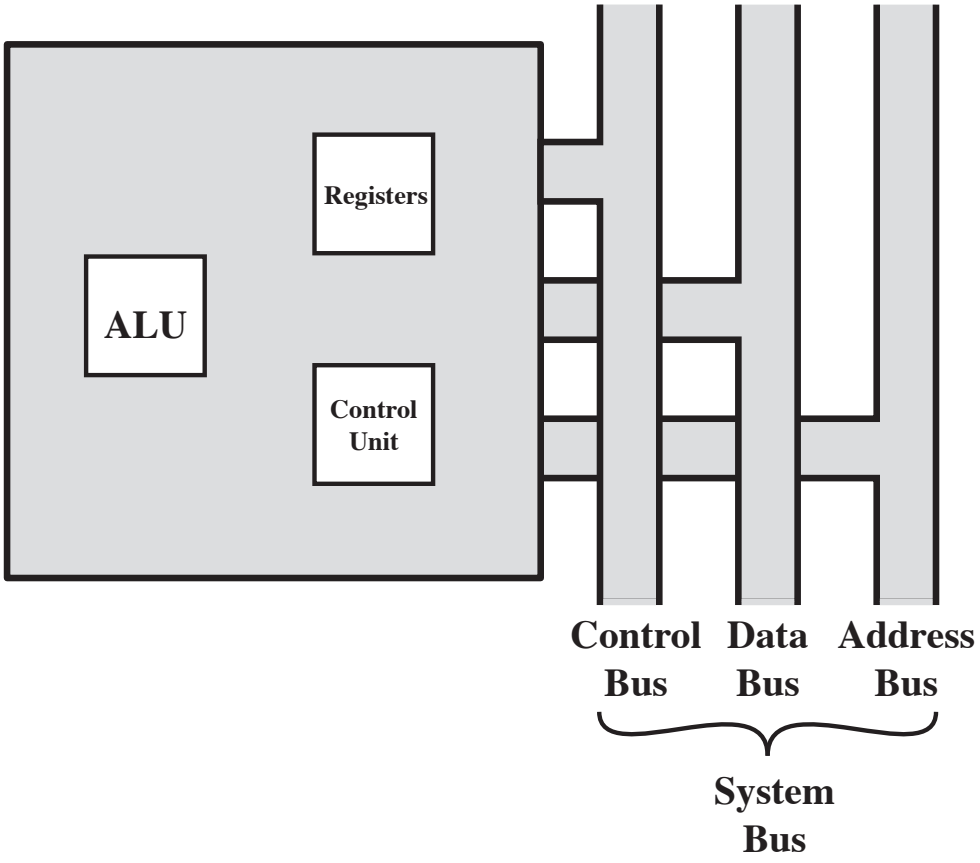
- Fetch instruction
  - The processor reads an instruction from memory (register, cache, main memory)
- Interpret instruction
  - The instruction is decoded to determine what action is required
- Fetch data
  - The execution of an instruction may require reading data from memory or an I/O module

# Processor Organization (2 of 2)

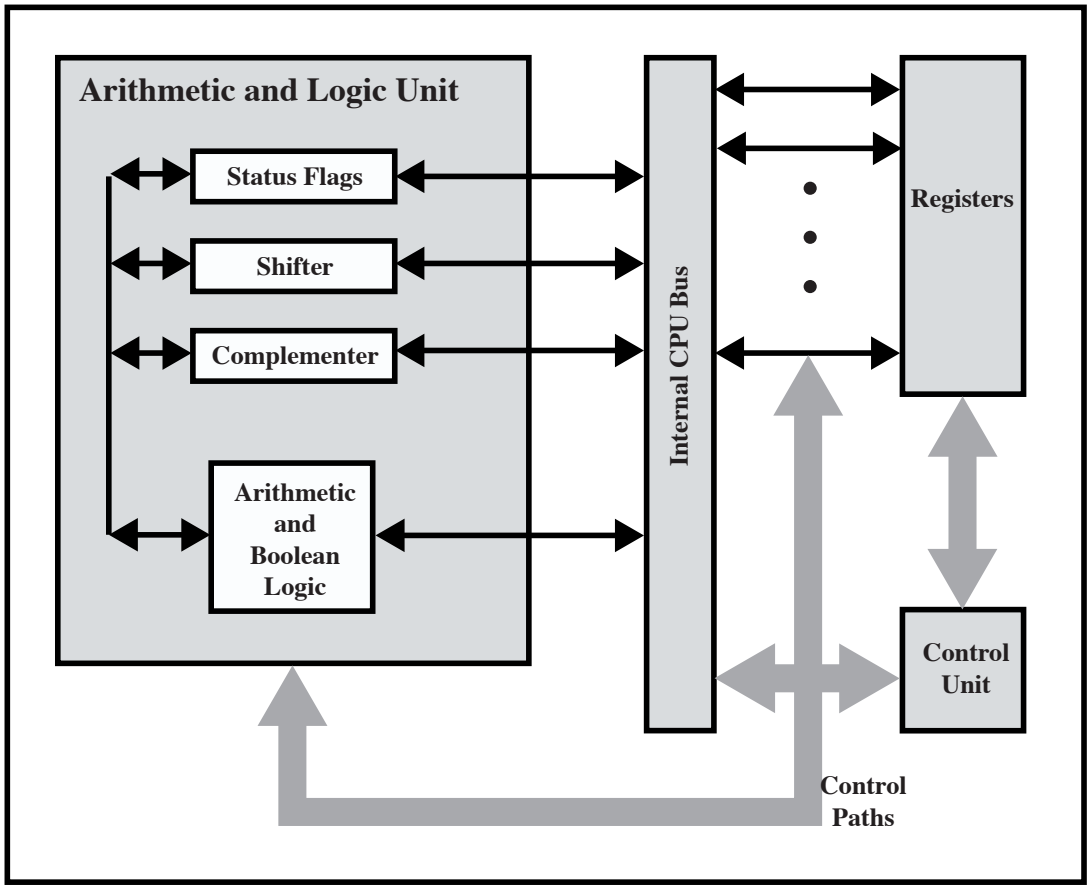
- Process data
  - The execution of an instruction may require performing some arithmetic or logical operation on data
- Write data
  - The results of an execution may require writing data to memory or an I/O module
- In order to do these things the processor needs to store some data temporarily and therefore needs a small internal memory



# The CPU with the System Bus



# Internal Structure of the CPU




Internal processor bus 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.

# Register Organization

- Within the processor there is a set of registers that function as a level of memory above main memory and cache in the hierarchy
- The registers in the processor perform two roles:
  - User-Visible Registers
    - Enable the machine or assembly language programmer to minimize main memory references by optimizing use of registers
  - Control and Status Registers
    - Used by the control unit to control the operation of the processor and by privileged operating system programs to control the execution of programs

# User-Visible Registers (1 of 2)

- Referenced by means of the machine language that the processor executes
- Categories:
  - General purpose
    - Can be assigned to a variety of functions by the programmer
  - Data
    - May be used only to hold data and cannot be employed in the calculation of an operand address

# User-Visible Registers (2 of 2)

- Address
  - May be somewhat general purpose or may be devoted to a particular addressing mode
  - Examples:  segment pointers, index registers, stack pointer
- Condition codes
  - Also referred to as flags
  - Bits set by the processor hardware as the result of operations

# Condition Codes

| **Advantages** | **Disadvantages** |
|---|---|
| Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed. | 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 micro programmer and compiler writer. |
| Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST and BRANCH. | Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. |
| 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. | 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. |
| Condition codes can be saved on the stack during subroutine calls along with other register information. | In a pipelined implementation, condition codes require special synchronization to avoid conflicts. |

# Control and Status Registers

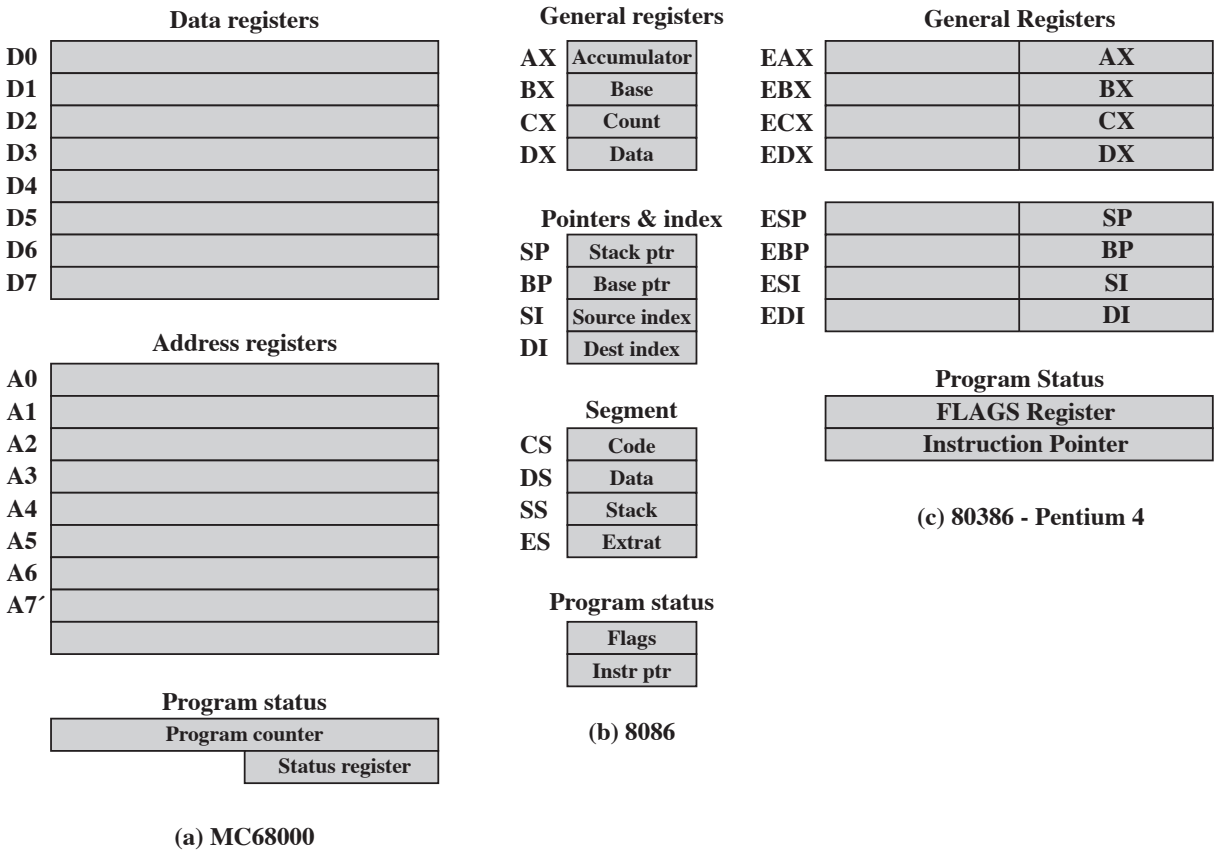
**Four registers are essential to instruction execution:**

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

# Program Status Word (PSW)

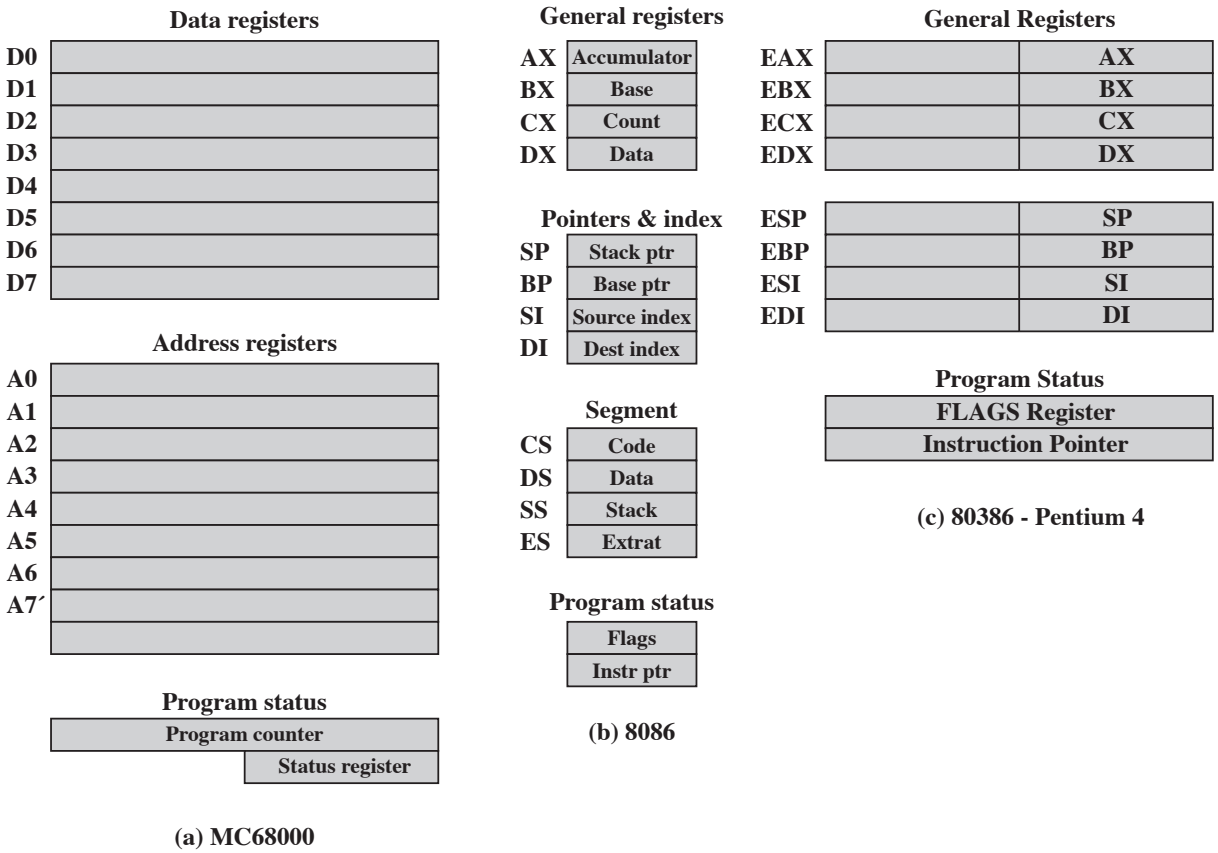
- Register or set of registers that contain status information
- Common fields or flags include:
  - Sign
  - Zero
  - Carry
  - Equal
  - Overflow
  - Interrupt Enable/Disable
  - Supervisor

# Example Microprocessor Register Organizations (1 of 3)



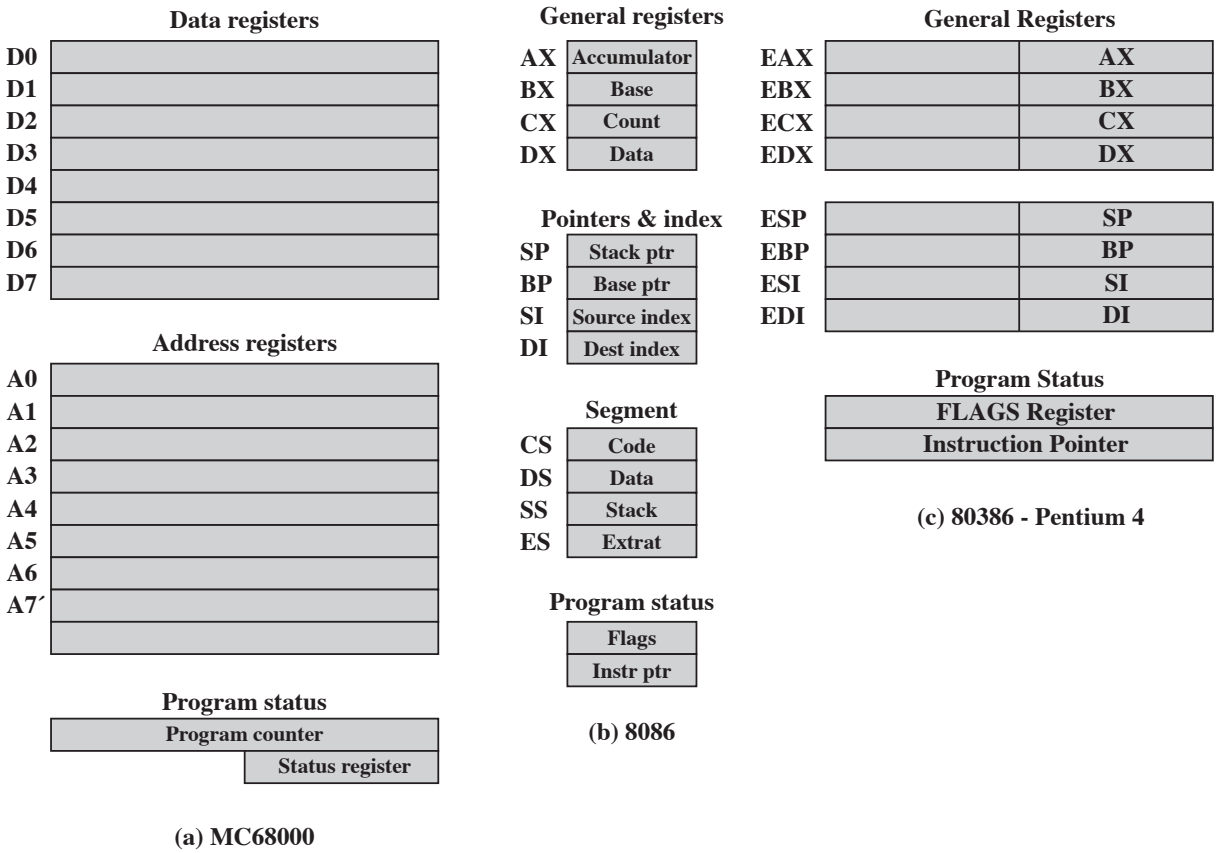

The MC68000 partitions its 32-bit registers into eight data registers and nine address registers. The eight data registers are used primarily for data manipulation and are also used in addressing as index registers. The width of the registers allows 8-, 16-, and 32-bit data operations, determined by opcode

# Example Microprocessor Register Organizations (2 of 3)




The Intel 8086 takes a different approach to register organization. Every register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers. The data registers can be used as general purpose in some instructions. In others, the registers are used implicitly.

# Example Microprocessor Register Organizations (3 of 3)

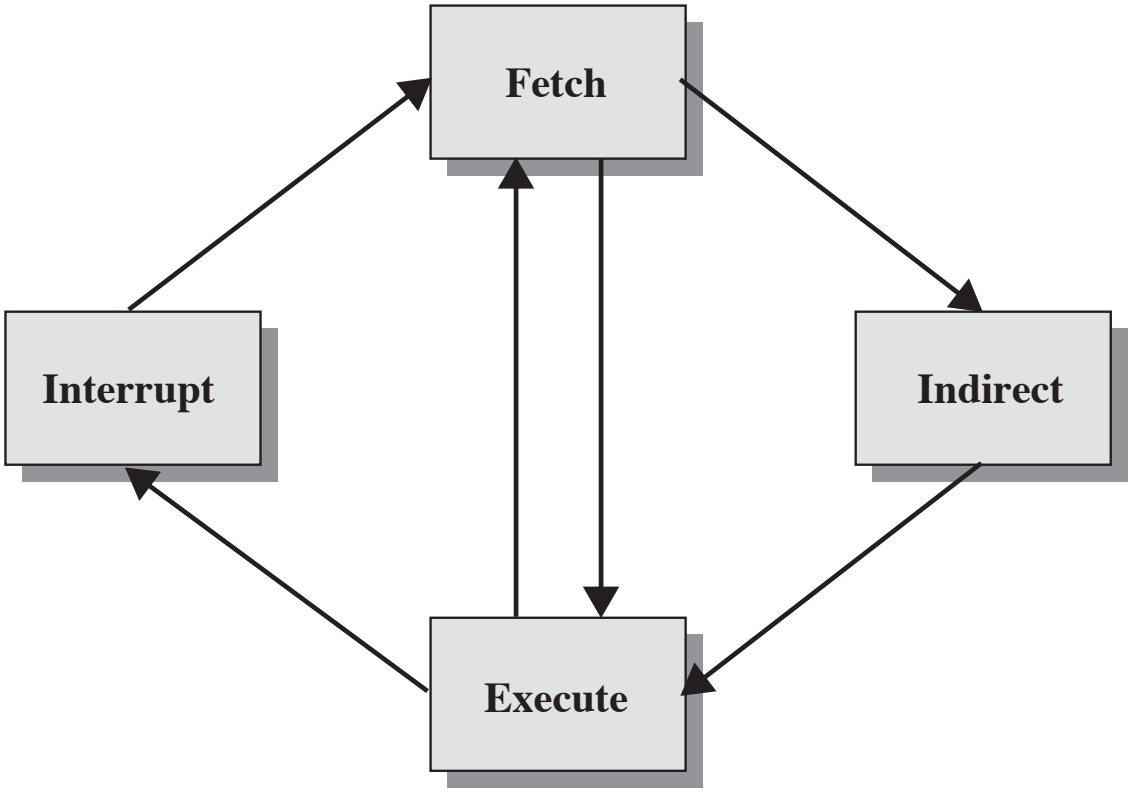



The Intel 80386 (P4) is a 32-bit microprocessor designed as an extension of the 8086. The 80386 uses 32-bit registers. However, to provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization. Given this design constraint, the architects of the 32-bit processors had limited flexibility in designing the register organization.

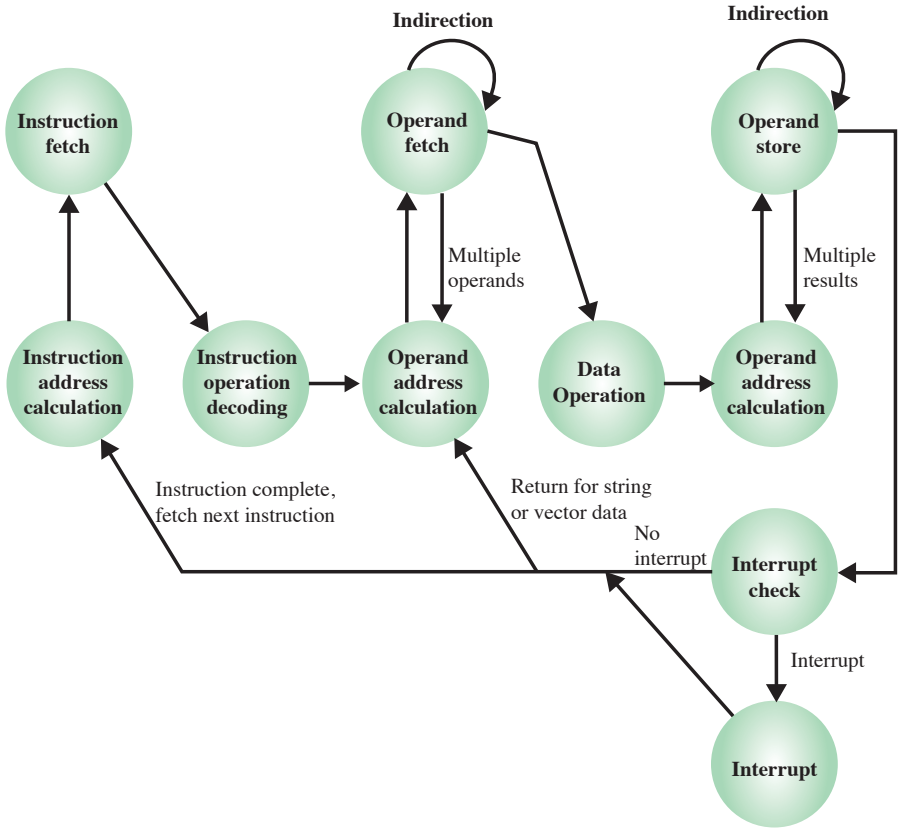
# Instruction Cycle (1 of 2)

- 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

# The Instruction Cycle (1 of 2)

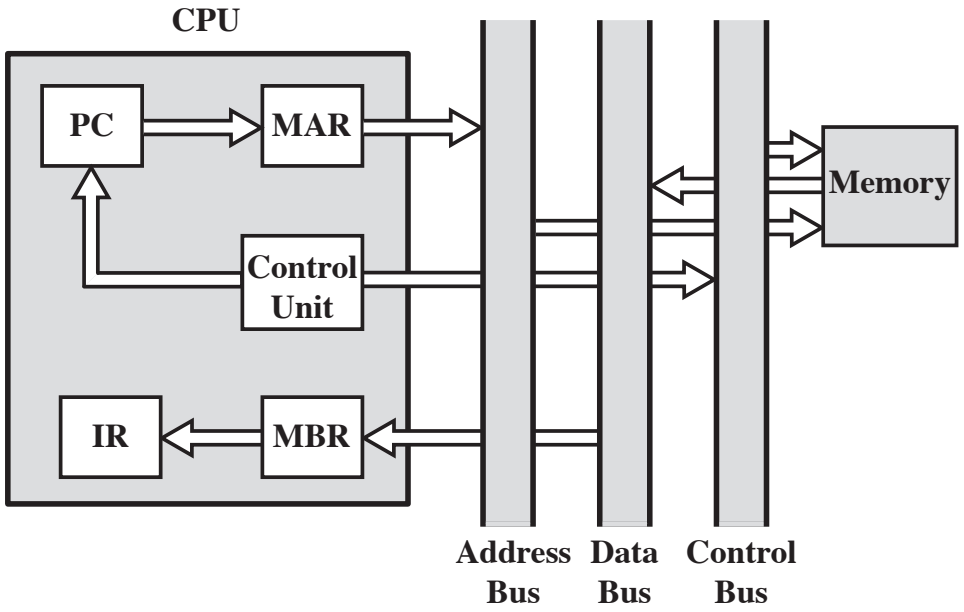


# Instruction Cycle State Diagram




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.

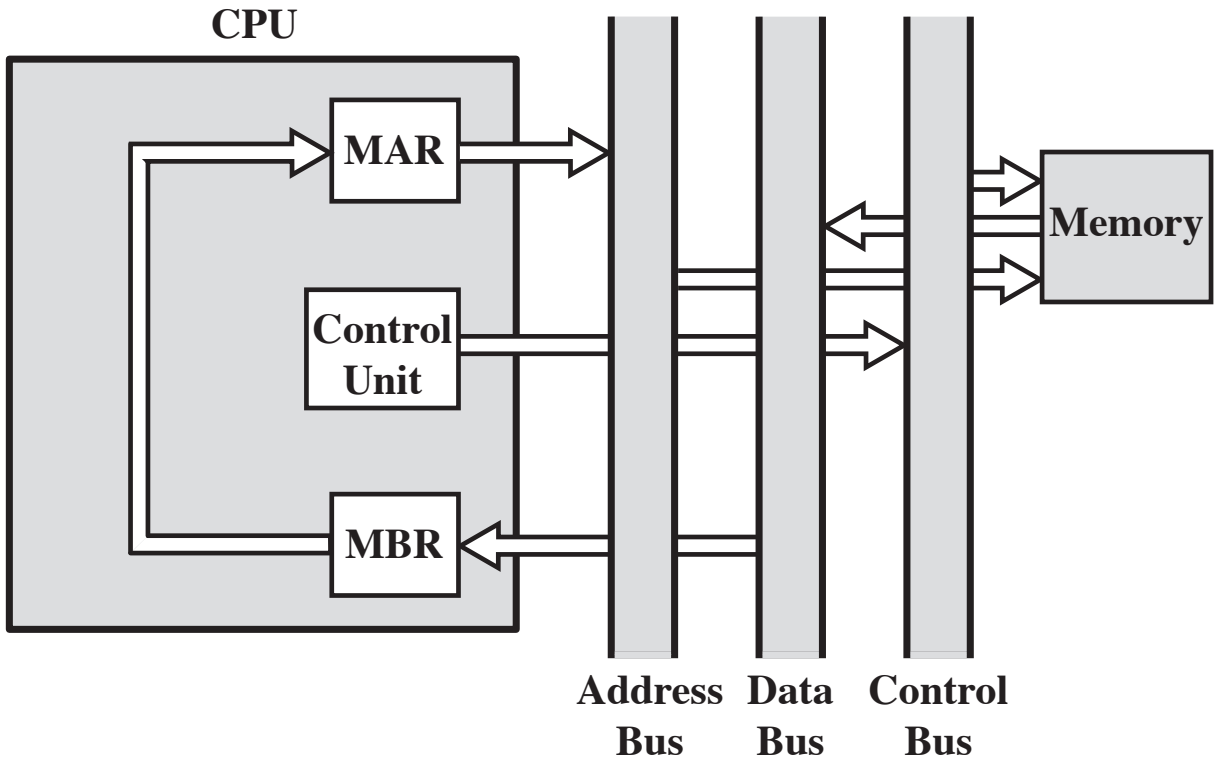
# Data Flow, Fetch Cycle




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

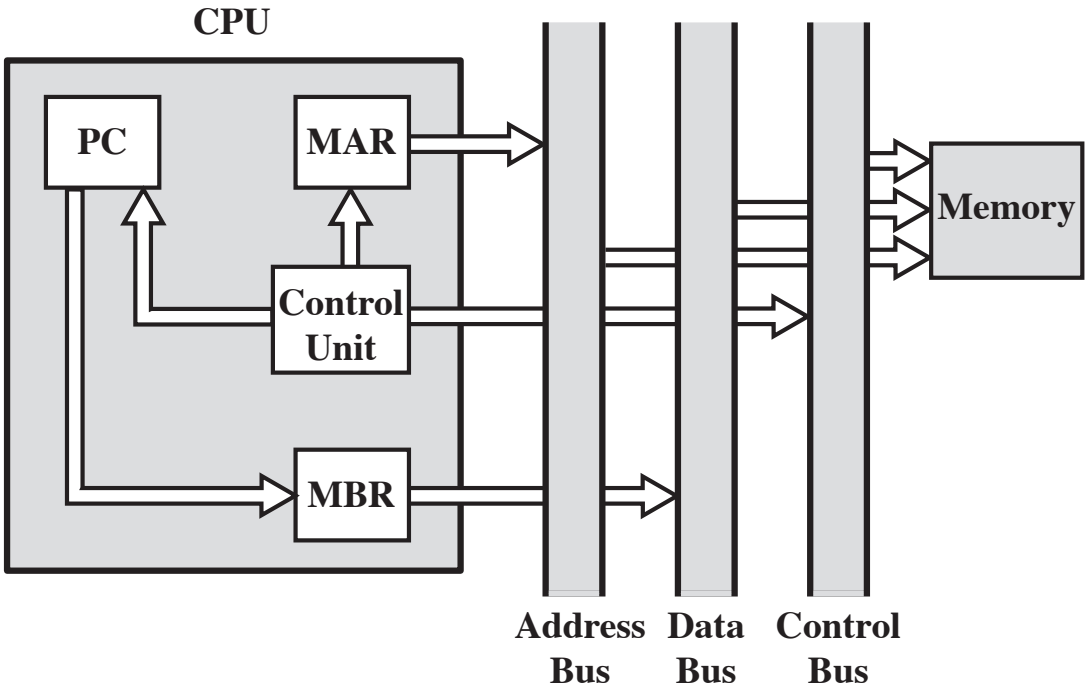
The exact sequence of events during an instruction cycle depends on the design of the processor. During the fetch cycle, an instruction is read from memory. Figure shows the flow of data during this cycle. The PC contains the address of the next instruction to be fetched. This address is moved to the 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.

# Data Flow, Indirect Cycle




Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing. If so, an indirect cycle is performed. As shown in Figure, this is a simple cycle. The right- most N bits of the MBR, which contain the address reference, are transferred to the MAR. Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

# Data Flow, Interrupt Cycle



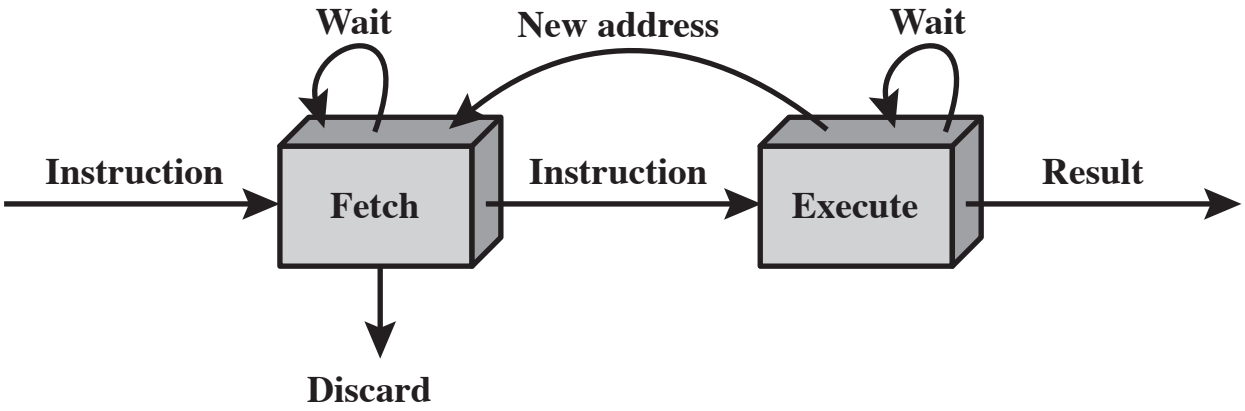

The interrupt cycle is simple and predictable. The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is 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.

# Pipelining Strategy

- Similar to the use of an assembly line in a manufacturing plant
- New inputs are accepted at one end before previously accepted inputs appear as outputs at the other end
- To apply this concept to instruction execution we must recognize that an instruction has a number of stages

# Two-Stage Instruction Pipeline (1 of 2)


(a) Simplified view

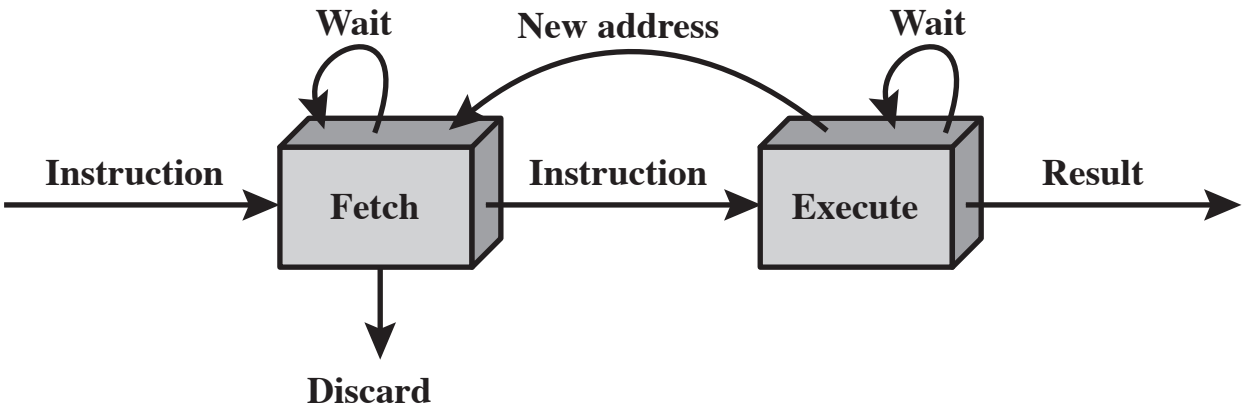



(b) Expanded view

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.

# Two-Stage Instruction Pipeline (2 of 2)


(a) Simplified view




(b) Expanded view

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, requires more registers.

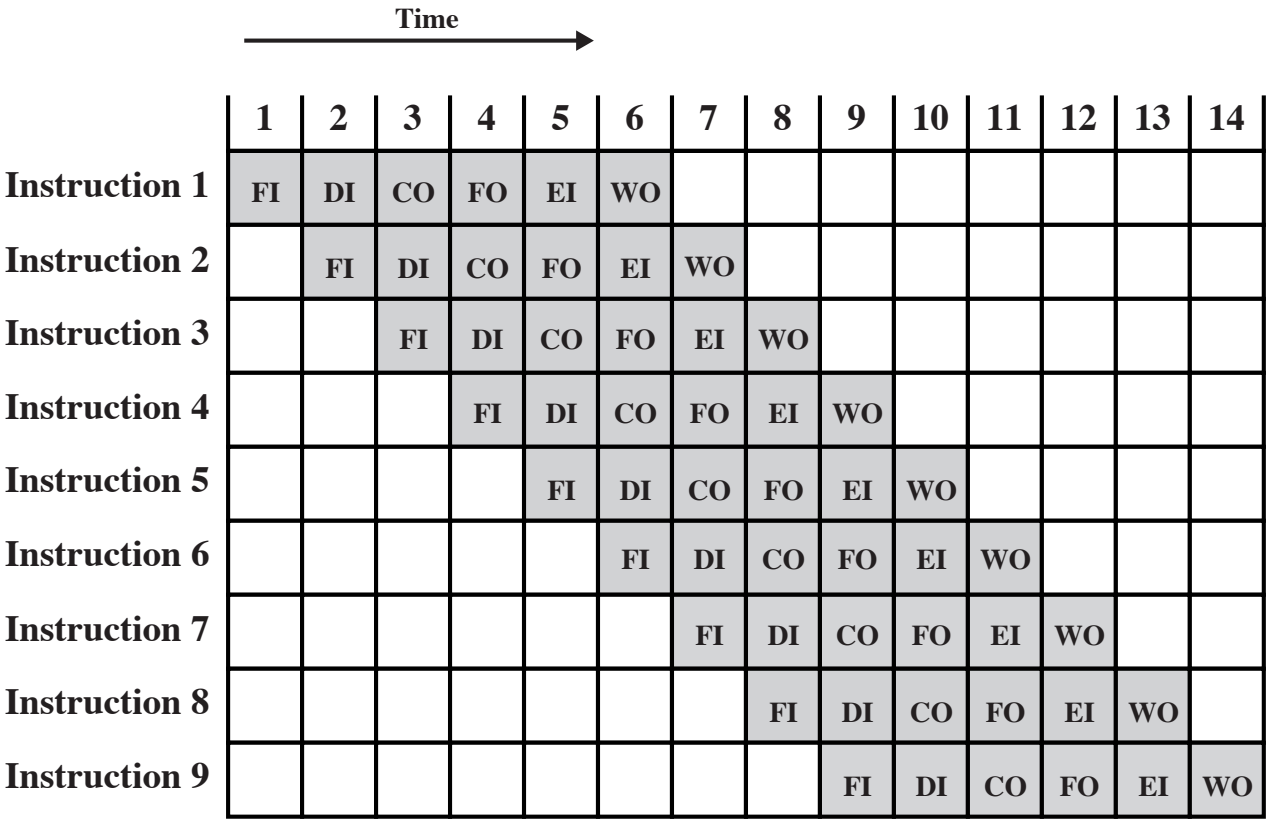
# Additional Stages (1 of 2)

- 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

# Additional Stages (2 of 2)

- 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




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

# Pipeline Operation

- 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 pipe-line 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.

# The Effect of a Conditional Branch on Instruction Pipeline Operation

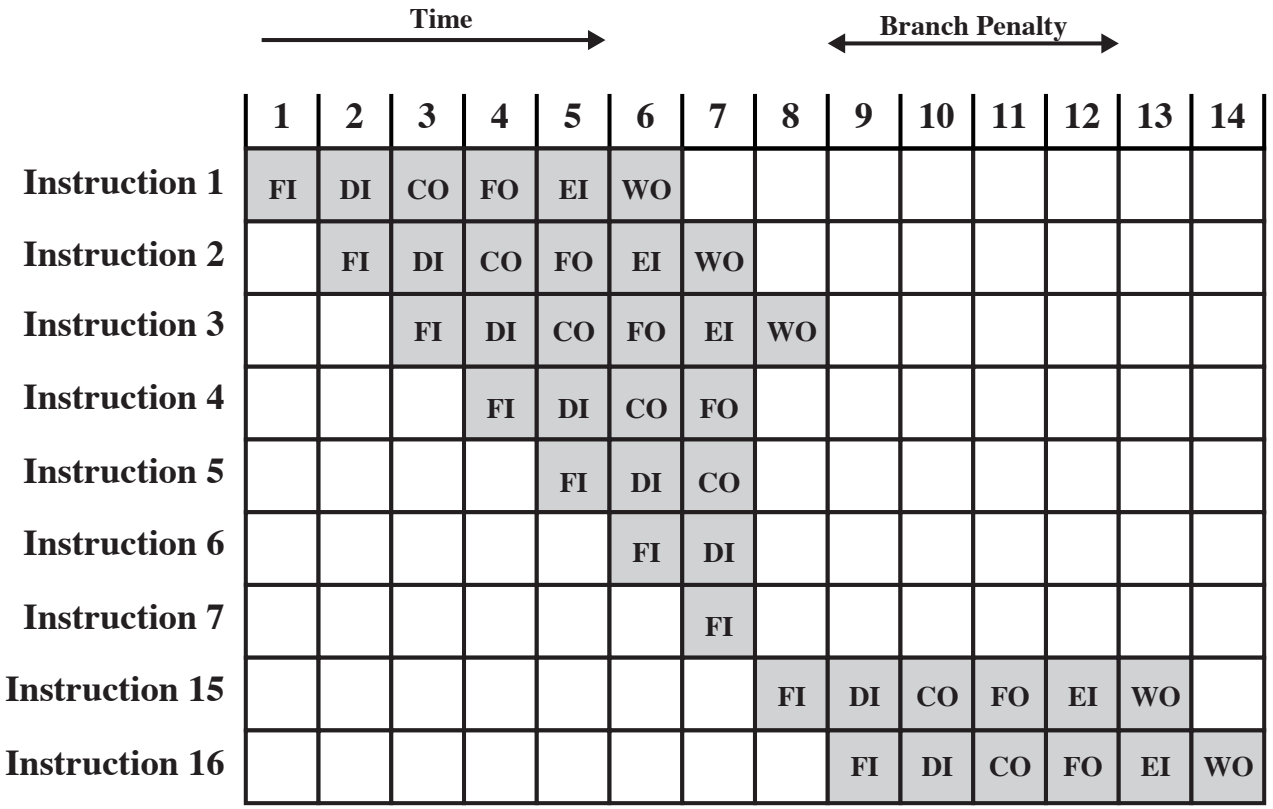



Figure above illustrates the effects of the conditional branch, using the same program as given in the previous Figure. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds.

# Effect of a Conditional Branch

- In the previous example, the branch is not taken, and we get the full performance benefit of the enhancement.
- In this Figure, 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.

# Six- Stage CPU Instruction Pipeline

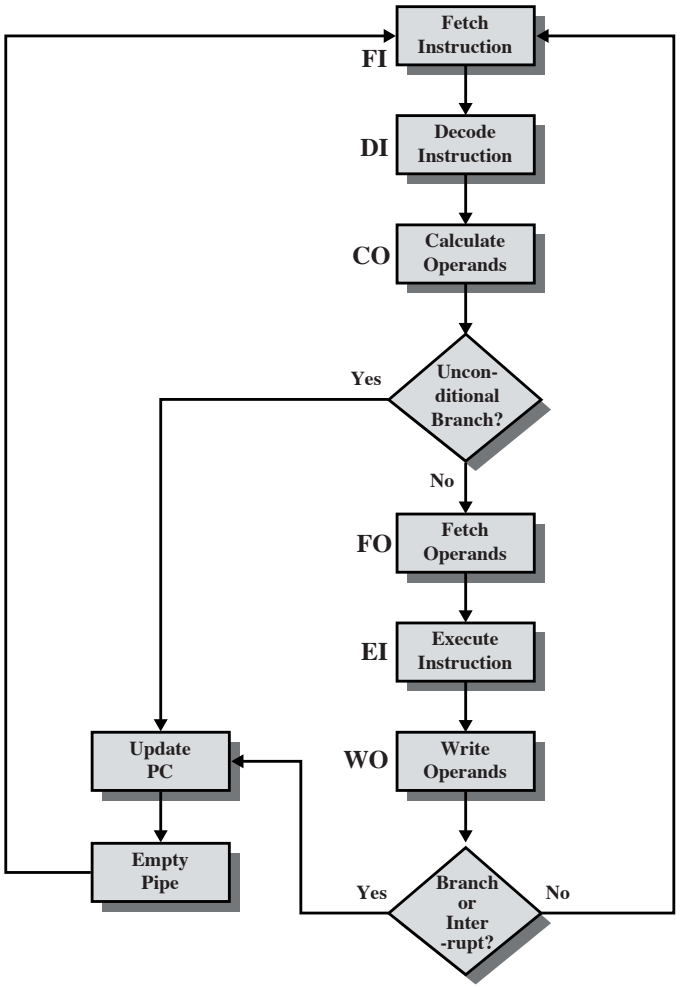
Figure indicates the logic needed for pipelining to account for branches and interrupts.

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.

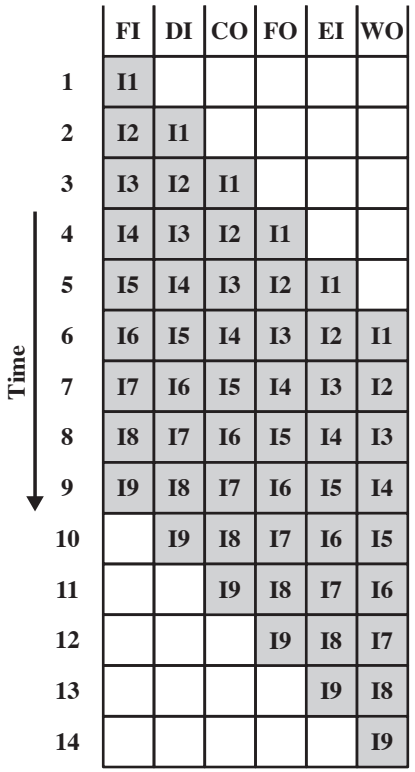


# An Alternative Pipeline Depiction

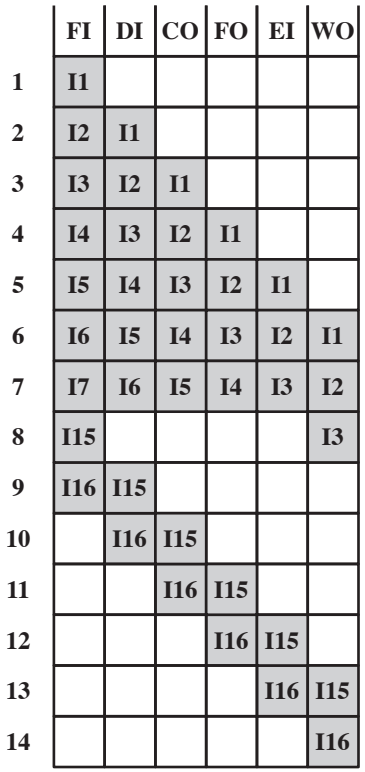
Figure shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time

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




(a) No branches




(b) With conditional branch

# Pipeline Hazards

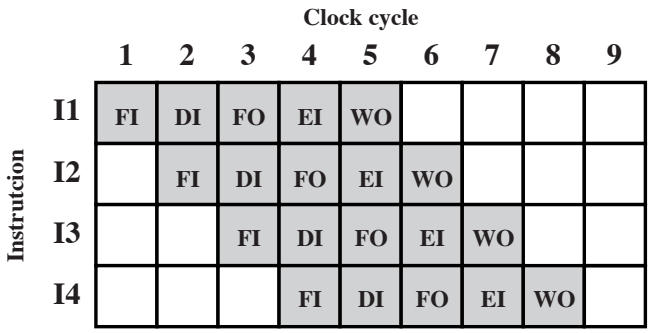
- Occur when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution
- Also referred to as a pipeline bubble
- There are three types of hazards:
  - Resource
  - Data
  - Control

# Example of Resource Hazard

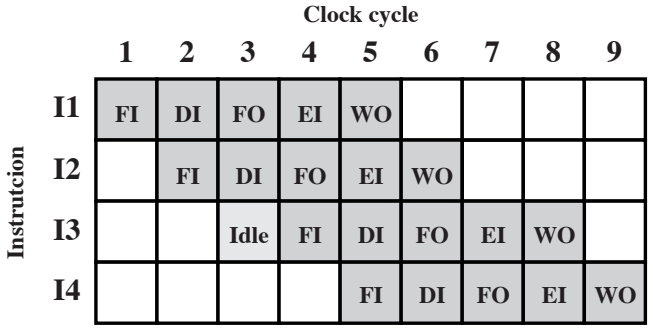
A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource.

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

A resource hazard is sometime referred to as a *structural hazard*.

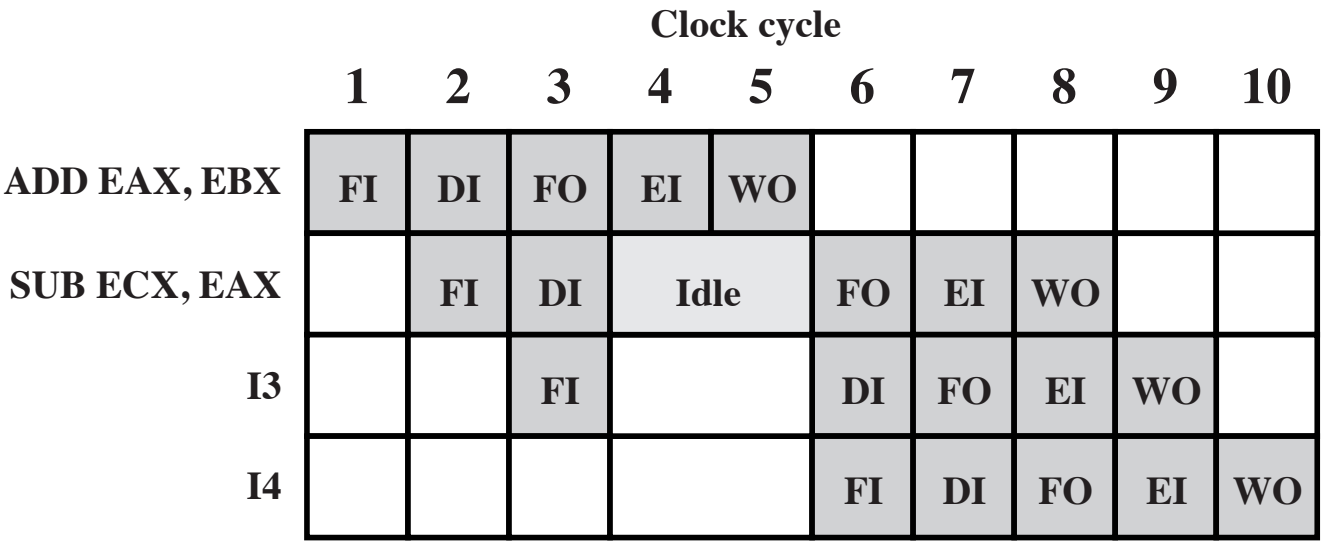



(a) Five-stage pipeline, ideal case




(b) I1 source operand in memory

# Example of Data Hazard




A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

# Types of Data Hazard (1 of 2)

- Read after write (RAW), or true dependency
  - An instruction modifies a register or memory location
  - Succeeding instruction reads data in memory or register location
  - Hazard occurs if the read takes place before write operation is complete
- Write after read (WAR), or antidependency
  - An instruction reads a register or memory location
  - Succeeding instruction writes to the location

# Types of Data Hazard (2 of 2)

  - Hazard occurs if the write operation completes before the read operation takes place

- Write after write (WAW), or output dependency
  - Two instructions both write to the same location
  - Hazard occurs if the write operations take place in the reverse order of the intended sequence

# Control Hazard

- Also known as a branch hazard
- Occurs when the pipeline makes the wrong decision on a branch prediction
- Brings instructions into the pipeline that must subsequently be discarded
- Dealing with Branches:
  - Multiple streams
  - Prefetch branch target
  - Loop buffer
  - Branch prediction
  - Delayed branch

# Multiple Streams

- A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice
- A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams
- Drawbacks:
  - With multiple pipelines there are contention delays for access to the registers and to memory
  - Additional branch instructions may enter the pipeline before the original branch decision is resolved

# Prefetch Branch Target

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

# Loop Buffer (1 of 2)

- Small, very-high speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence
- Benefits:
  - Instructions fetched in sequence will be available without the usual memory access time
  - If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer
  - This strategy is particularly well suited to dealing with loops

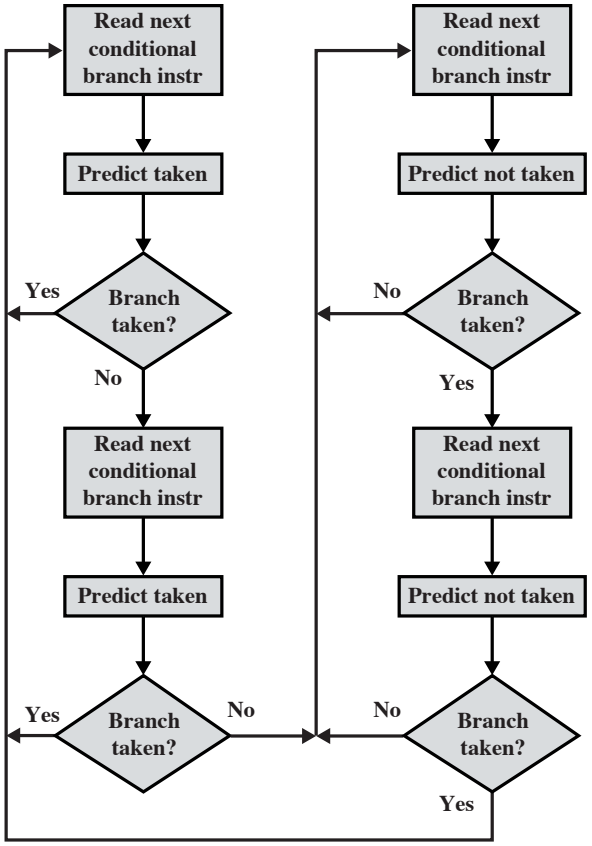
# Loop Buffer (2 of 2)

- Similar in principle to a cache dedicated to instructions
- Differences:
  - The loop buffer only retains instructions in sequence
  - Is much smaller in size and hence lower in cost

# Branch Prediction

- Various techniques can be used to predict whether a branch will be taken:
  - Predict never taken
  - Predict always taken
  - Predict by opcode
    - These approaches are static
    - They do not depend on the execution history up to the time of the conditional branch instruction
  - Taken/not taken switch
  - Branch history table
    - These approaches are dynamic
    - They depend on the execution history

# Branch Prediction Flowchart



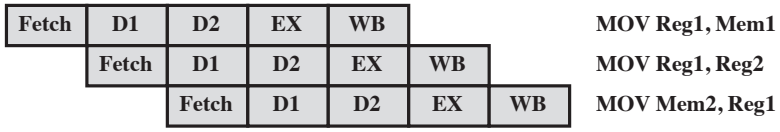
# Intel 80486 Pipelining (1 of 2)

- Fetch
  - Objective is to fill the prefetch buffers with new data as soon as the old data have been consumed by the instruction decoder
  - Operates independently of the other stages to keep the prefetch buffers full
- Decode stage 1
  - All opcode and addressing-mode information is decoded in the D1 stage
  - 3 bytes of instruction are passed to the D1 stage from the prefetch buffers
  - D1 decoder can then direct the D2 stage to capture the rest of the instruction

# Intel 80486 Pipelining (2 of 2)

- Decode stage 2
  - Expands each opcode into control signals for the ALU
  - Also controls the computation of the more complex addressing modes
- Execute
  - Stage includes ALU operations, cache access, and register update
- Write back
  - Updates registers and status flags modified during the preceding execute stage

# 80486 Instruction Pipeline Examples




(a) No Data Load Delay in the Pipeline


(b) Pointer Load Delay


(c) Branch Instruction Timing

# Interrupt Processing (1 of 2)

**Interrupts and Exceptions**

- Interrupts
  - Generated by a signal from hardware and it may occur at random times during the execution of a program
  - Maskable
  - Nonmaskable
- Exceptions
  - Generated from software and is provoked by the execution of an instruction

# Interrupt Processing (2 of 2)

  - Processor detected
  - Programmed
- Interrupt vector table
  - Every type of interrupt is assigned a number
  - Number is used to index into the interrupt vector table