

# Chapter 12

## CPU Structure and Function

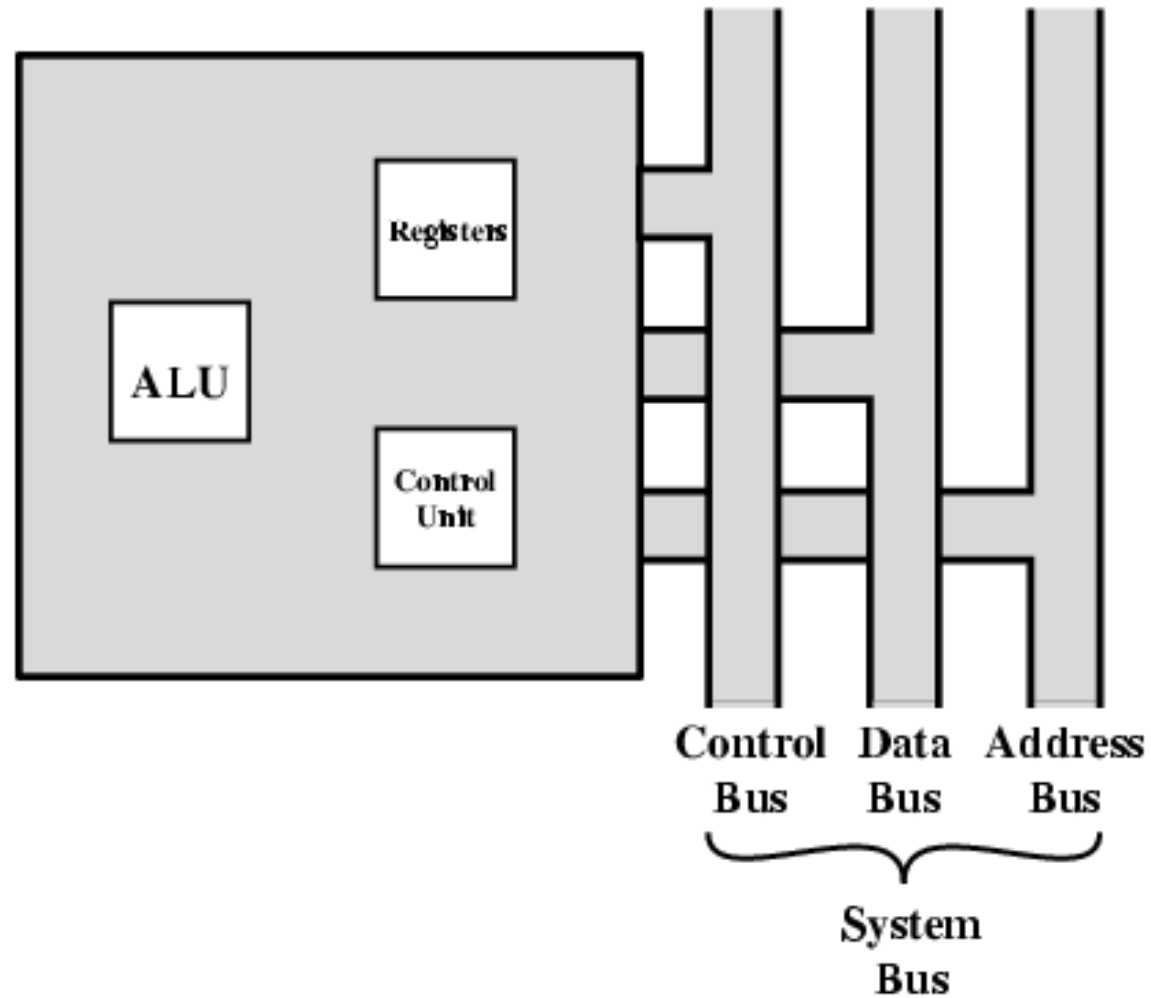
# Objectives

- To know the processor organization
- To know the register organization
- To know the instruction cycle
- To know the instruction pipelining

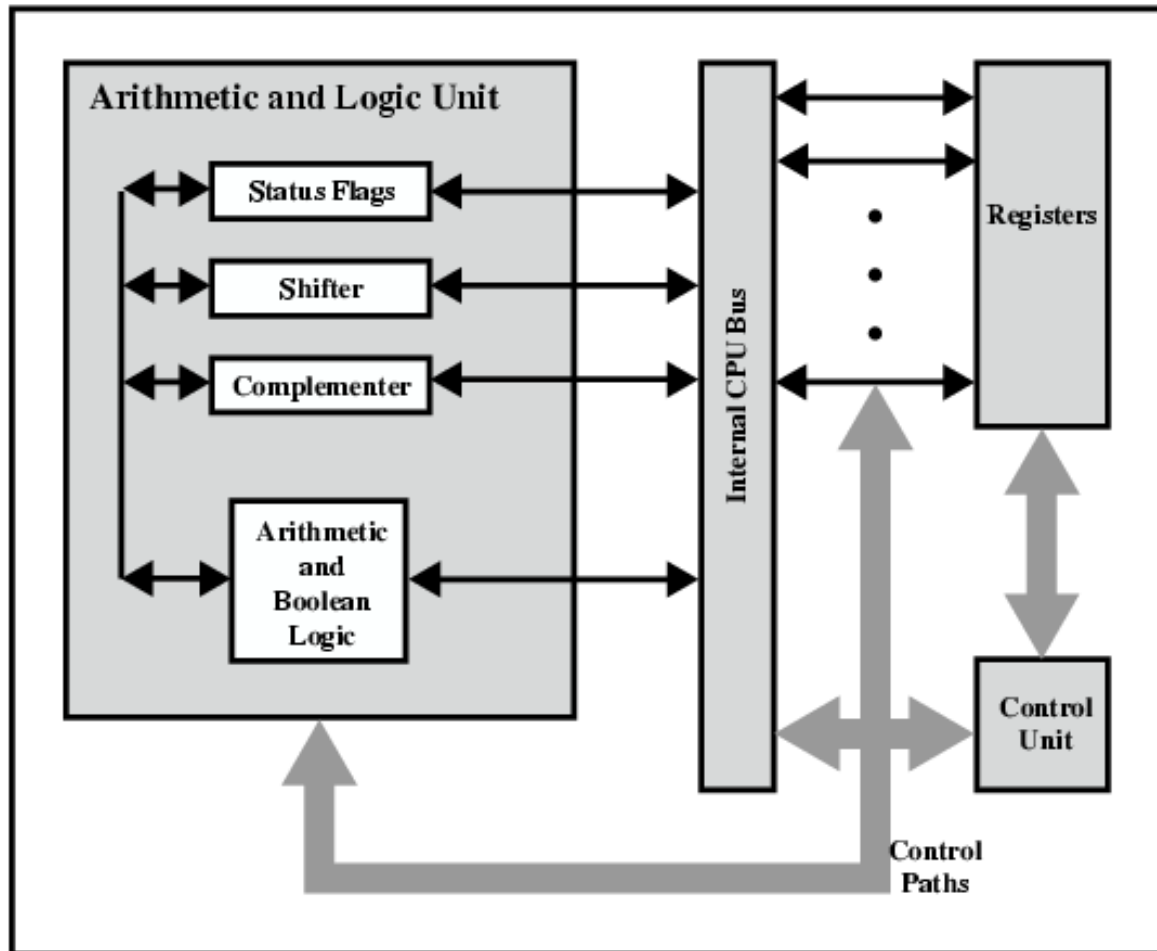
# Processor Organization

- CPU must perform the following tasks step-by-step:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data

## Register Addressing



## CPU Internal Structure



# Register Organization

- CPU must have some working space (temporary storage) which is called **registers**.
- Register is at the top level of memory hierarchy.
- Number and function of register vary between processor designs.
- This is one of the major CPU design decisions.

# User-Visible Registers

- A **user visible register** is one that may be referenced by means of the machine language that the CPU executes.
- **User-visible registers can be categorized into:**
  - General Purpose
    - Data
    - Address
  - Condition Codes

# General-Purpose Registers

- **General purpose registers:**
  - may be true general purpose
  - may be restricted for floating-point and stack operations
  - may be used for data or addressing
    - Data register may be used only to hold data.
    - Address register may be general purpose or devoted to a particular address mode.

# General-Purpose Registers (2)

- **Making them general purpose:**
  - Increase flexibility and programmer options
  - Increase instruction size & complexity
- **Making them specialized:**
  - Smaller (faster) instructions
  - Less flexibility

## General-Purpose Registers (3)

- **How many general purpose registers are commonly implemented?**
  - Between 8 - 32
  - Fewer → More memory references
  - More → Does not reduce memory references and takes up processor real estate
- **How big should general purpose registers be?**
  - Large enough to hold full address
  - Large enough to hold full word
  - Often possible to combine two data registers (like C programming)
    - `double int a;`
    - `long int a;`

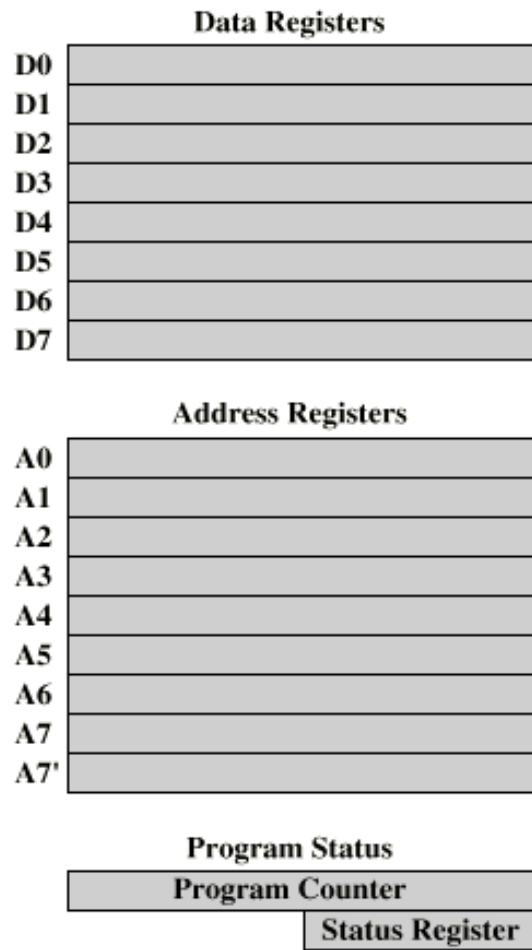
# Control and Status Registers

- Control and status registers are sets of individual bits.
  - e.g. result of last operation was zero.
- They can be read (implicitly) by programs.
  - e.g. Jump if zero
- They cannot (usually) be set by programs.
- **Four registers are essential to instruction execution:**
  - Program Counter
  - Instruction Register
  - Memory Address Register
  - Memory Buffer Register

## Control and Status Registers (2)

- All CPU designs include a register or set of registers, known as program status word (PSW), that contain status information.
- The PSW typically contains condition codes plus other status information.
  - **Sign**: contain 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 (+) or borrow (-)
  - **Equal**: set if a logical compare result is equality
  - **Overflow**: used to indicate arithmetic overflow
  - **Interrupt enable/disable**: used to enable or disable interrupt
  - **Supervisor**: indicate whether the CPU is executing in supervisor or user mode.

## Example of Register Organization



(a) MC68000

**General Registers**

AX	Accumulator
BX	Base
CX	Count
DX	Data

**Pointer & Index**

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

**Segment**

CS	Code
DS	Data
SS	Stack
ES	Extra

**Program Status**

Instr Ptr
Flags

(b) 8086

**General Registers**

EAX		AX
EBX		BX
ECX		CX
EDX		DX

ESP		SP
EBP		BP
ESI		SI
EDI		DI

**Program Status**

FLAGS Register
Instruction Pointer

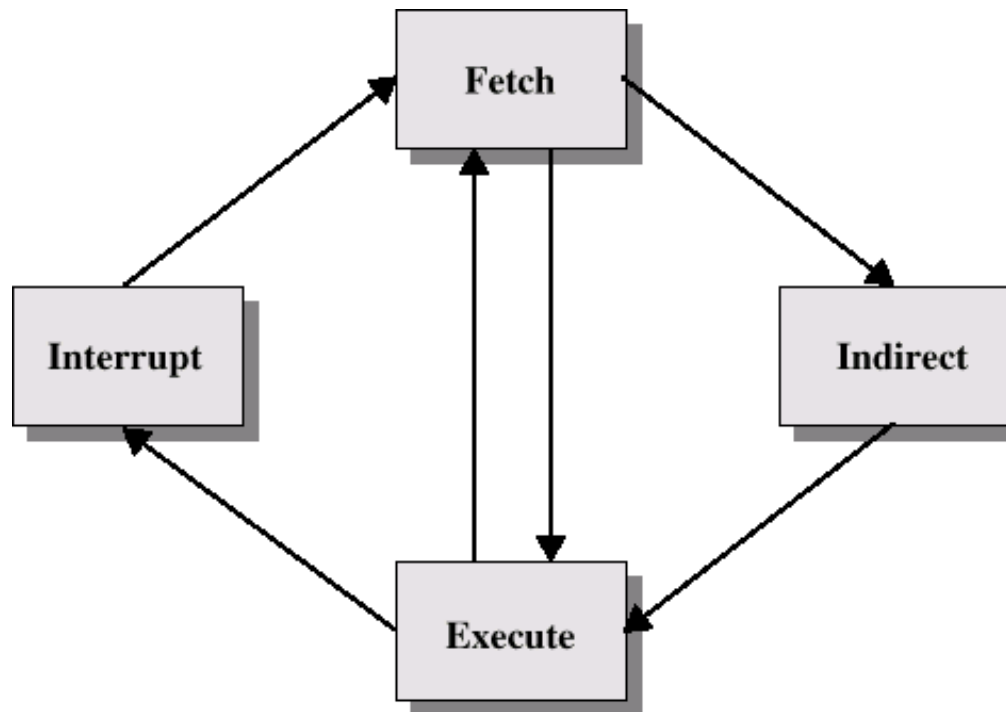
(c) 80386 - Pentium II

# Instruction Cycle

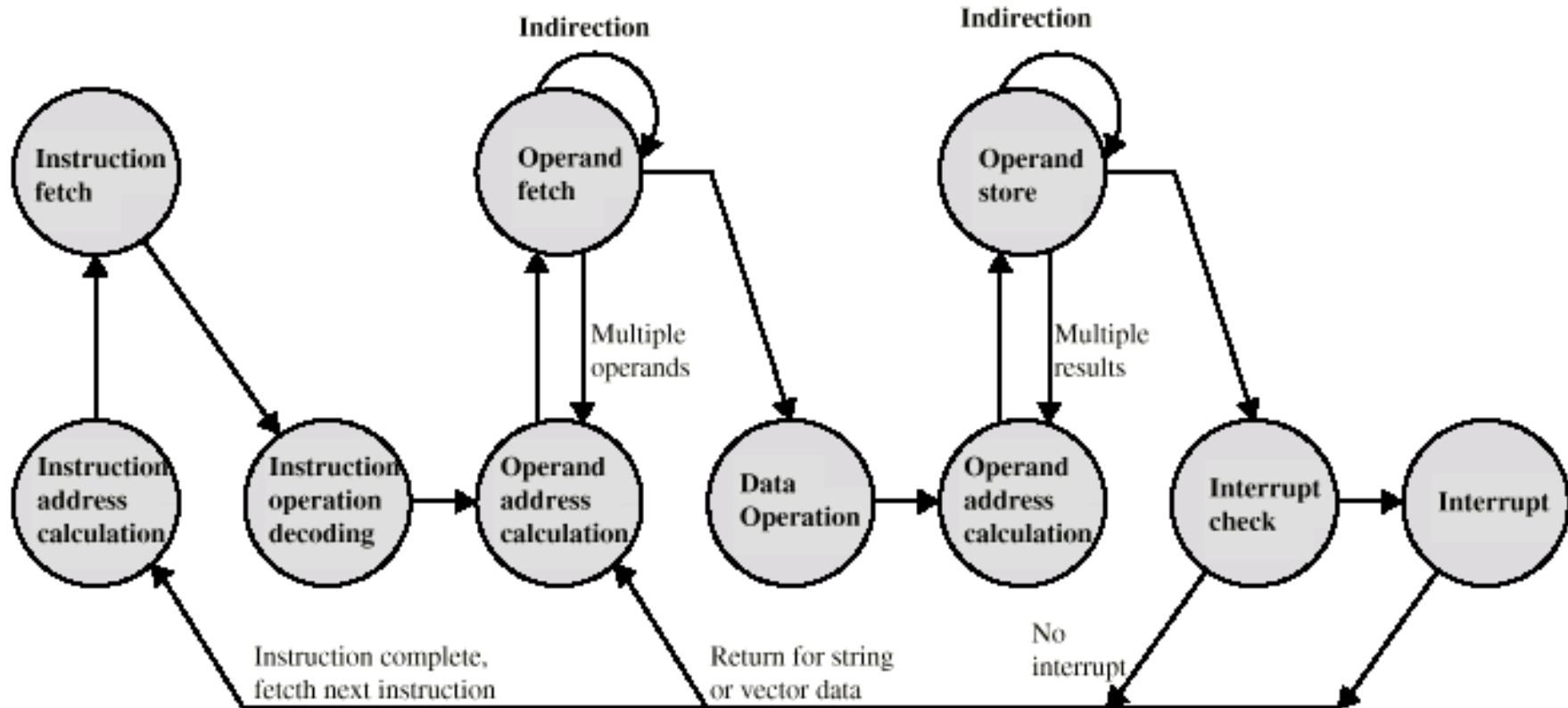
- Instruction cycle includes the following subcycles:
  - **Fetch**: read the next instruction from memory into the CPU
  - **Execute**: interpret the opcode and perform the indicated operation
  - **Interrupt**: if interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.
- There is one additional subcycle known as **indirect cycle**.

# Instruction Cycle

- **Indirect cycle** may require memory access to fetch operands.
  - Indirect addressing requires more memory accesses
  - It can be thought of as additional instruction subcycle.



## Instruction Cycle State Diagram

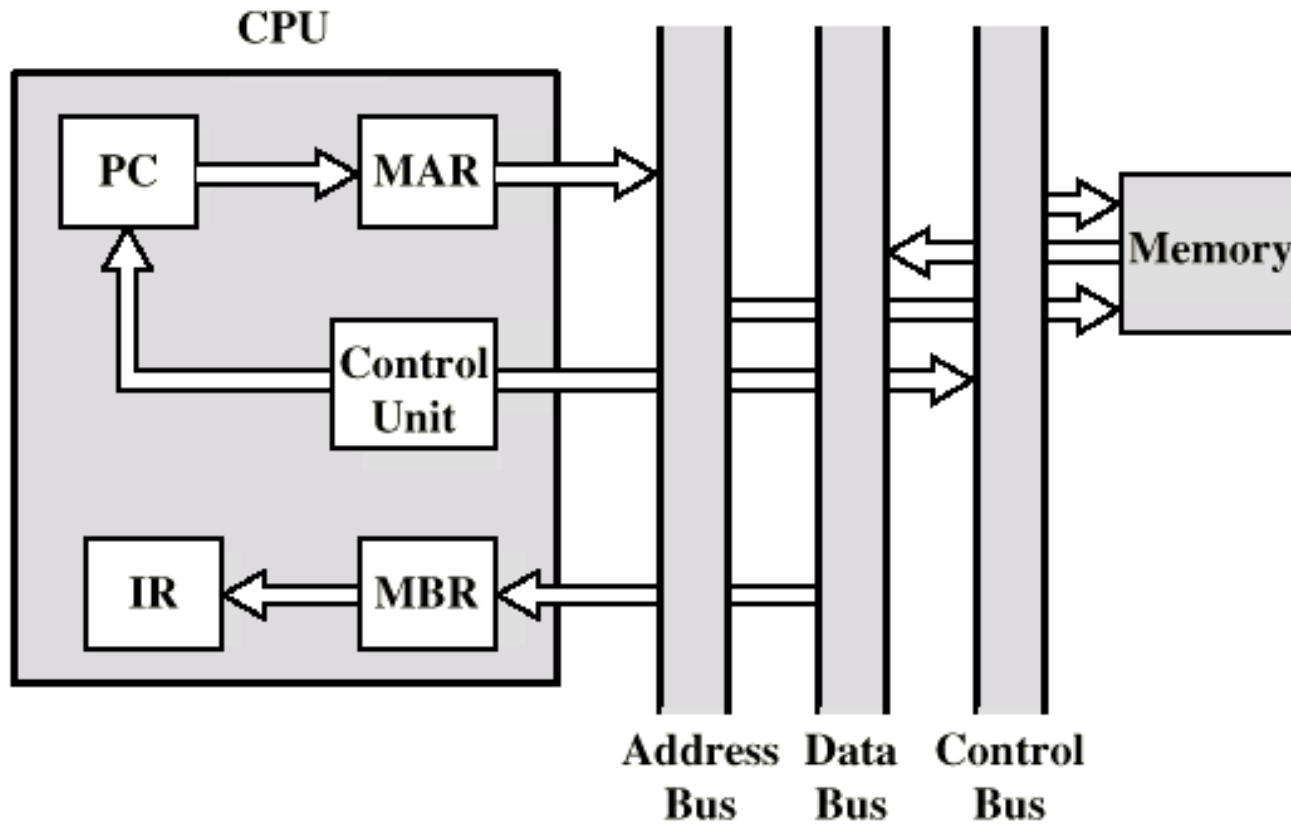


# Data Flow (Fetch Cycle)

- **Instruction Fetch**

- PC contains address of next instruction.
- Address is moved to MAR.
- Address is placed on address bus.
- Control unit requests memory read.
- Result is placed on data bus, copied to MBR, then to IR.
- Meanwhile, PC is incremented by 1.

## Data Flow (Fetch Cycle)

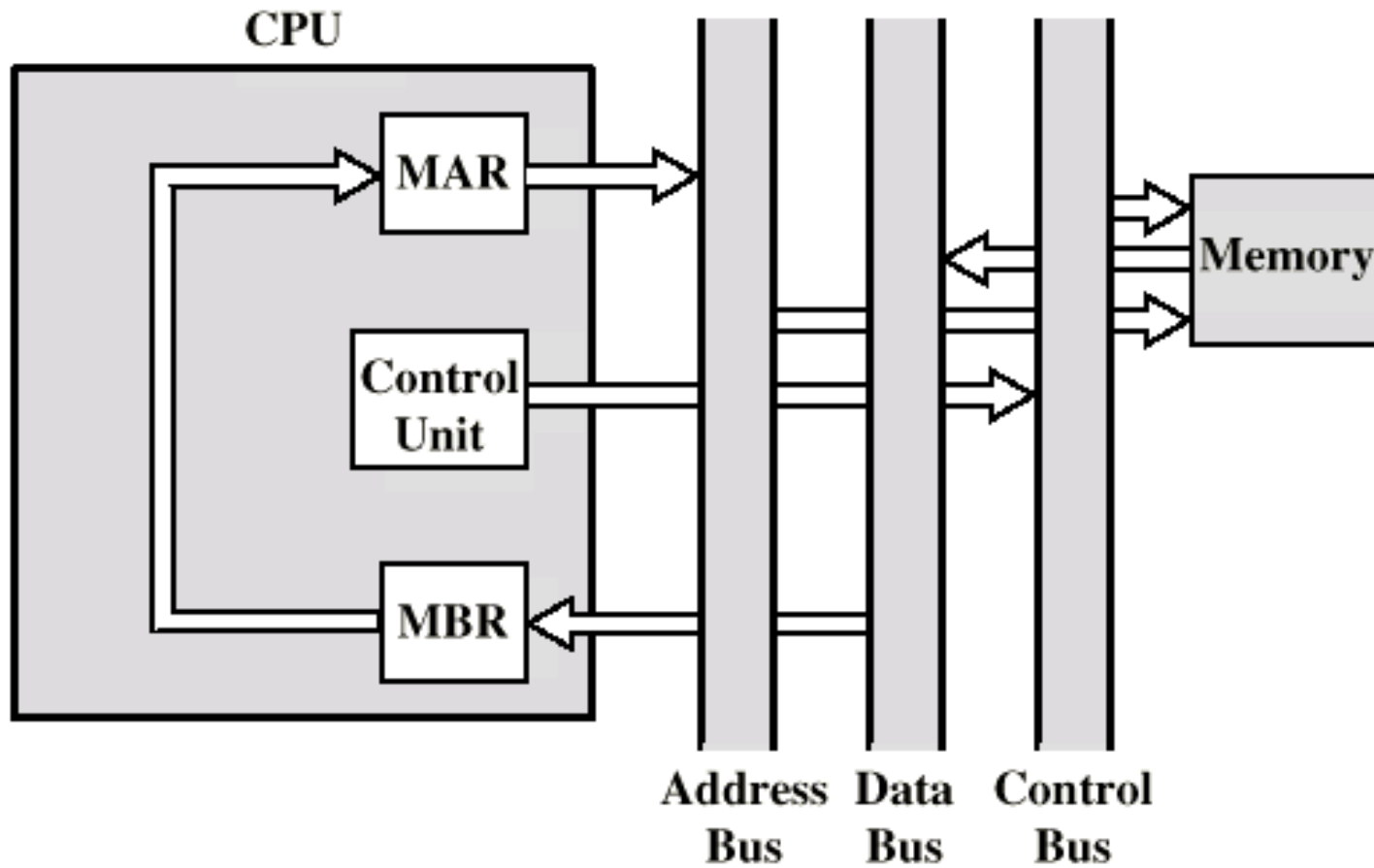


MBR = Memory buffer register  
MAR = Memory address register  
IR = Instruction register  
PC = Program counter

## Data Flow (Fetch Cycle) (2)

- **Data Fetch**
  - IR is examined.
  - If indirect addressing, indirect cycle is performed.
    - Right most N bits of MBR are transferred to MAR.
    - Control unit requests memory read.
    - Result (address of operand) is moved to MBR.

## Data Flow (Fetch Cycle)



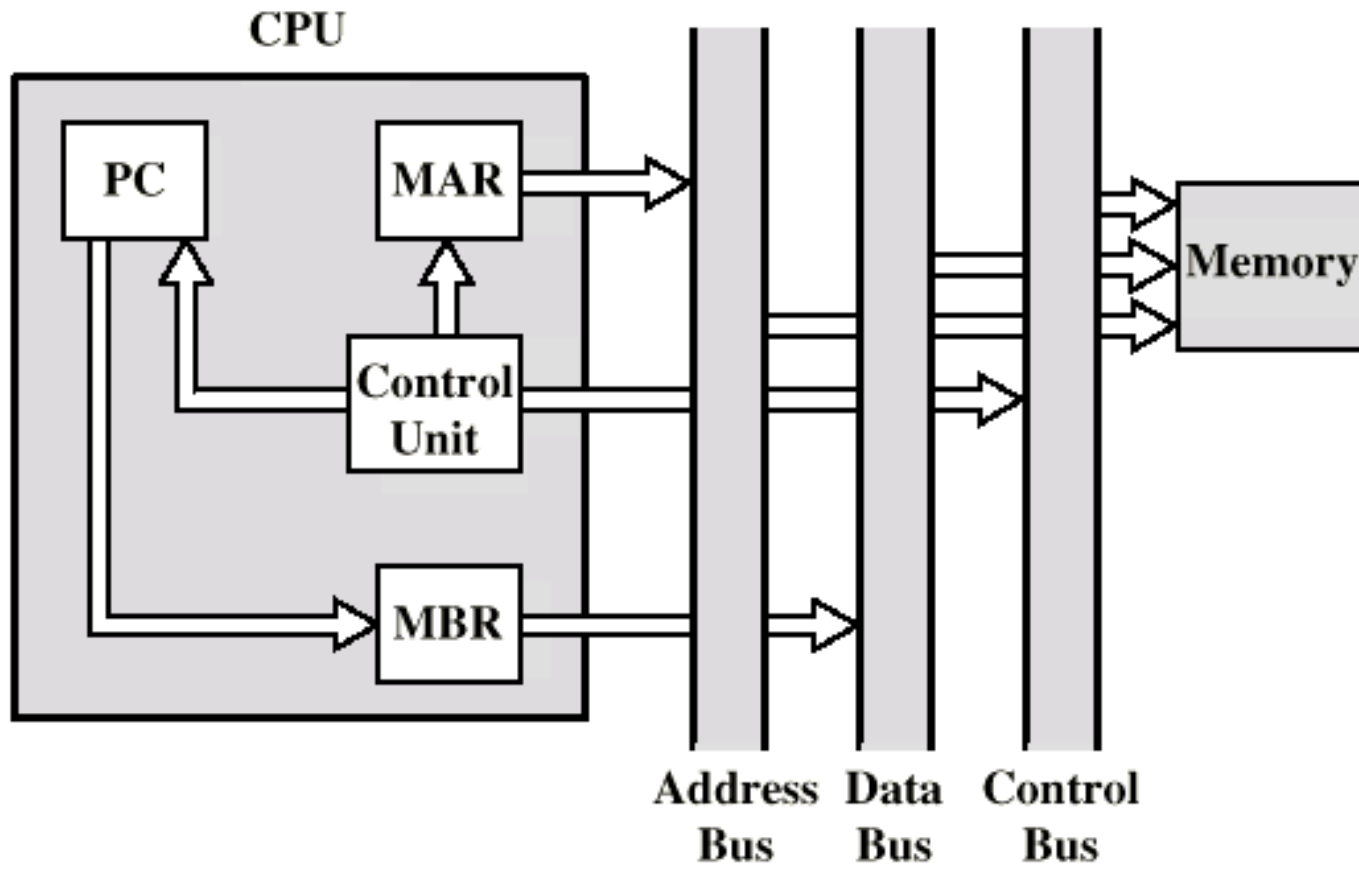
# Data Flow (Execute Cycle)

- **Execute Cycle** may take many forms depending on instruction being executed.
- It may include:
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations

## Data Flow (Interrupt Cycle)

- **Interrupt Cycle** is simple and predictable.
  - Current PC is saved to allow resumption after interrupt.
  - Contents of PC are copied to MBR.
  - Special memory location (e.g. stack pointer) is loaded to MAR.
  - MBR is written to memory.
  - PC is loaded with address of interrupt handling routine.
  - Next instruction (first of interrupt handler) can be fetched.

## Data Flow (Interrupt Cycle)



# Performance Improvement (Prefetch)

- Fetching accesses main memory.
- Execution usually does not access main memory.
- CPU can fetch next instruction during execution of current instruction.
- This is called instruction **prefetch**.
- **Improved Performance:**
  - Fetching is usually shorter than execution.
    - Prefetch more than one instruction?
  - Any jump or branch means that prefetched instructions are not the required instructions
- Adding more instruction cycle stages may improve performance.

# Instruction Pipelining

- An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).
- Instruction pipelining is similar to the use of an assembly line in a manufacturing plant.
- In general, more pipeline stages increase speedup.

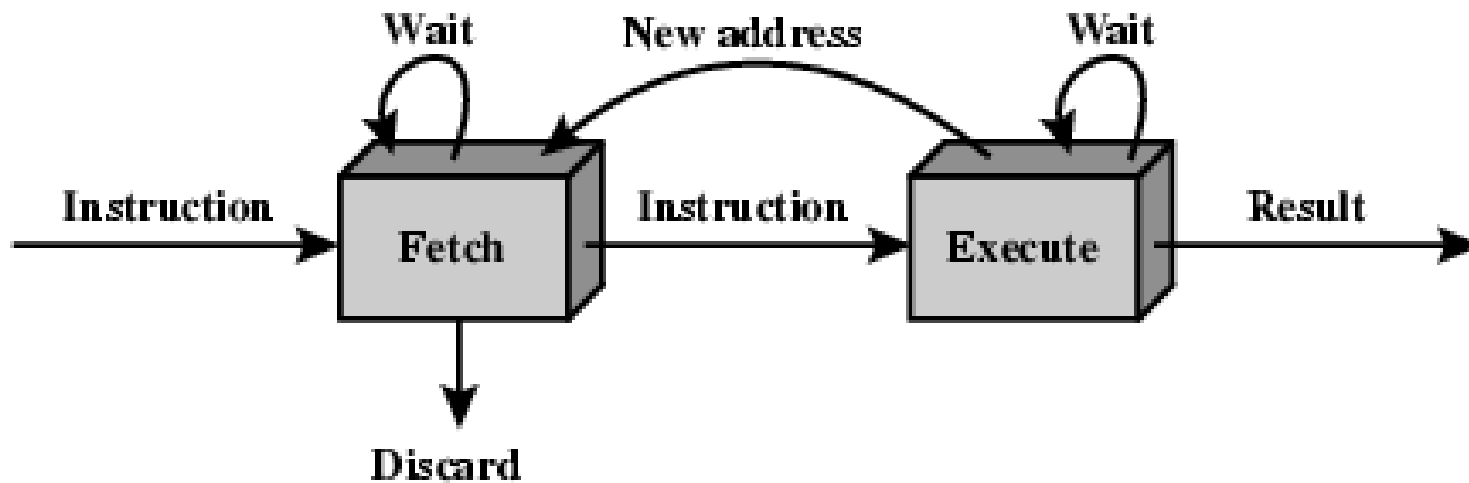
## Instruction Pipelining (2)

- 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 specifier
  - **Calculate operands (CO):** calculate the effective address of each source operand
  - **Fetch operands (FO):** fetch each operand from memory
  - **Execute instructions (EI):** perform the indicated operation and store the result, if any, in the specified destination operand location
  - **Write result (WO):** store the result in memory
- Instruction pipelining overlaps these operations.

## Two-Stage Instruction Pipeline

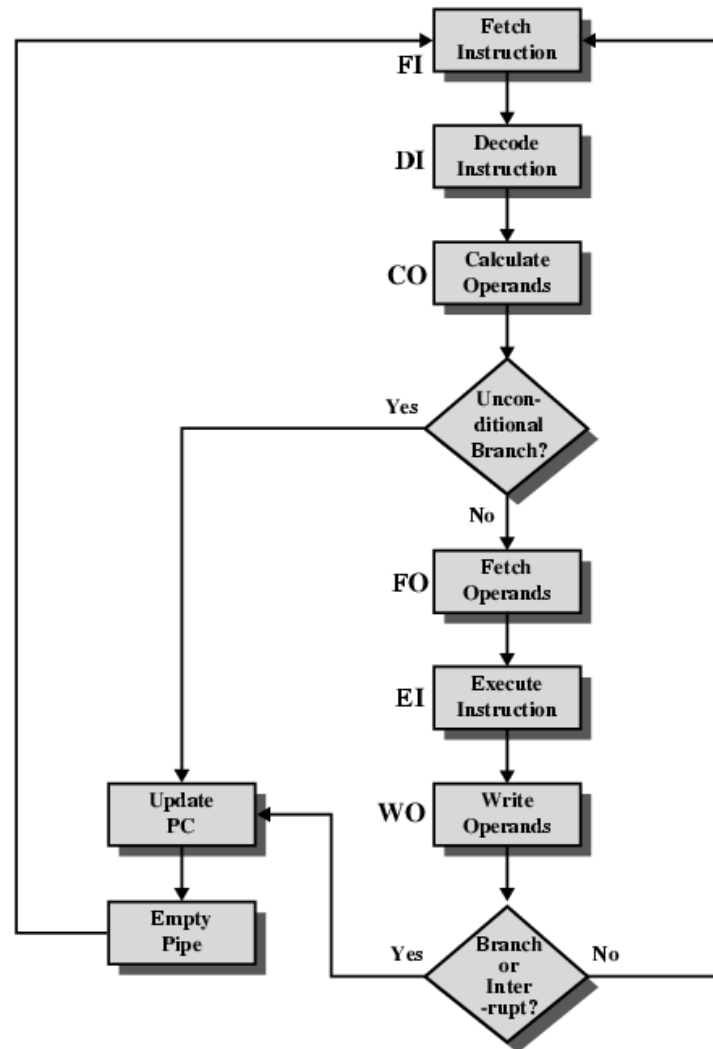


(a) Simplified view

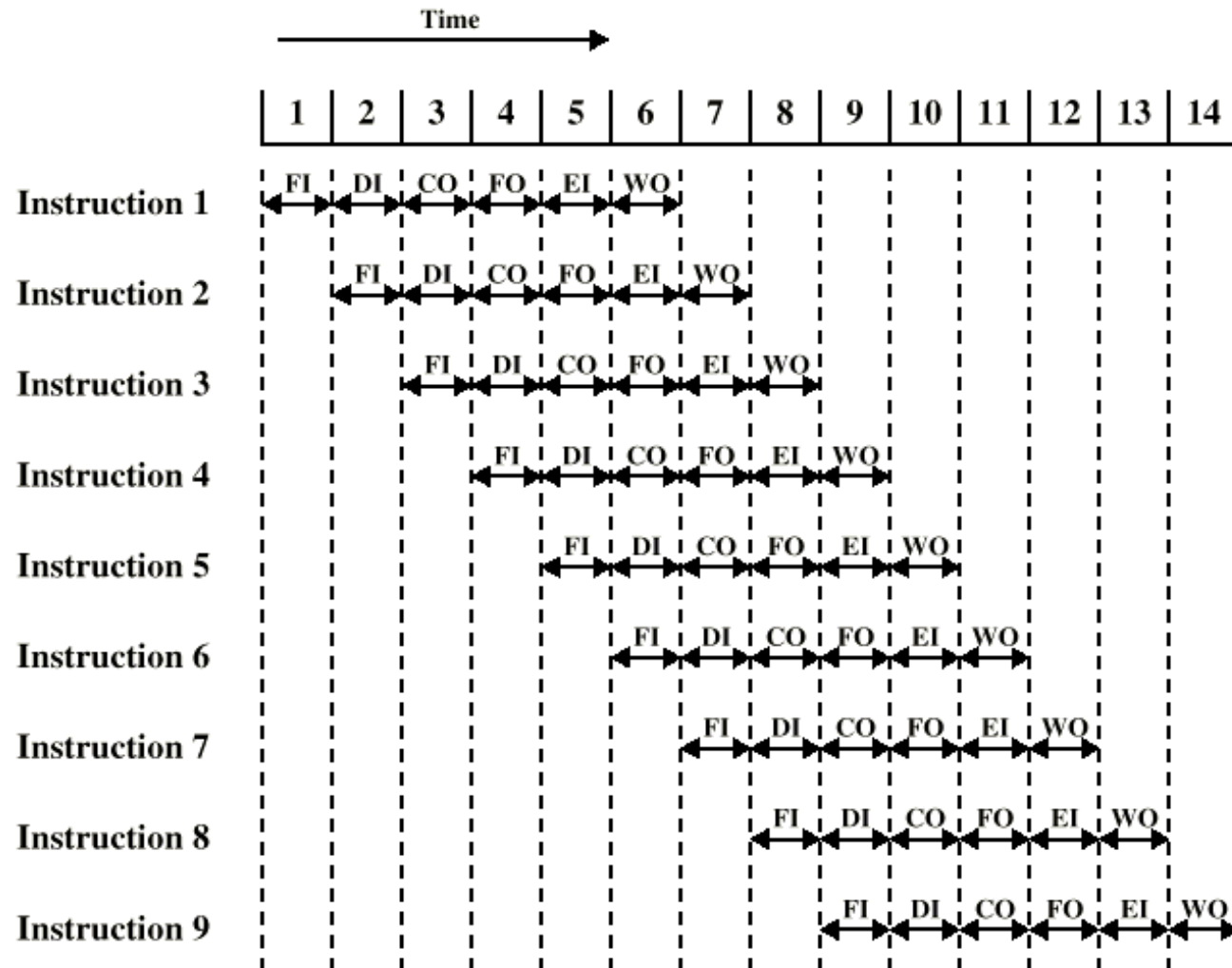


(b) Expanded view

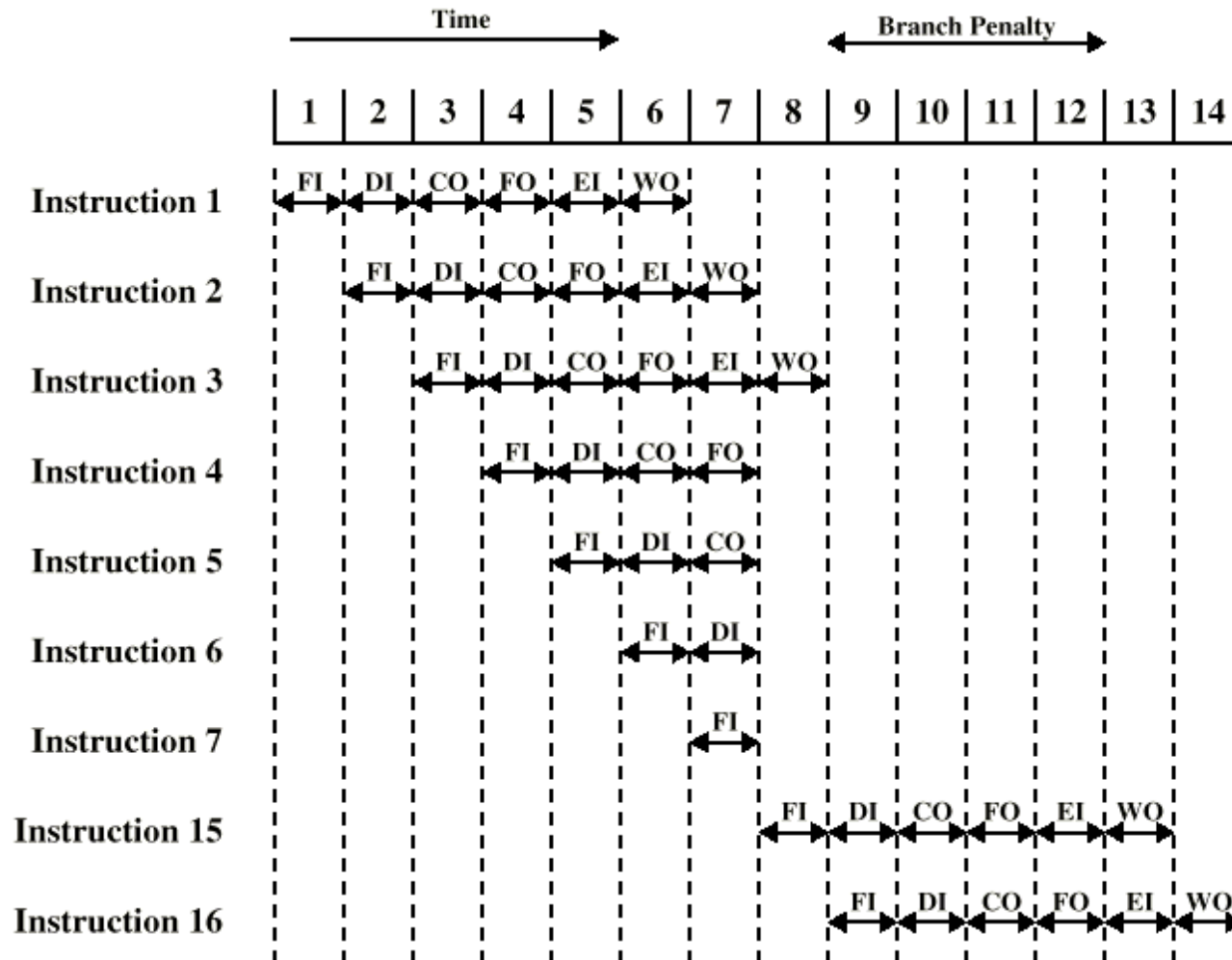
## Six-Stage Instruction Pipeline



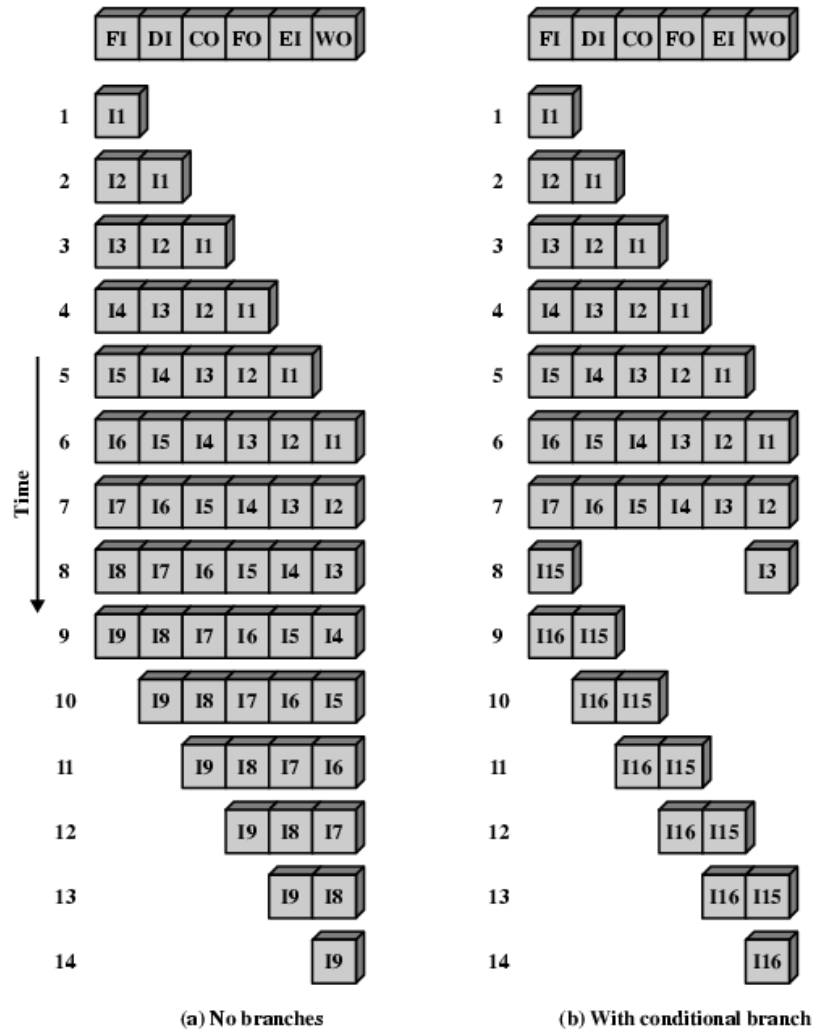
## Timing of Pipeline



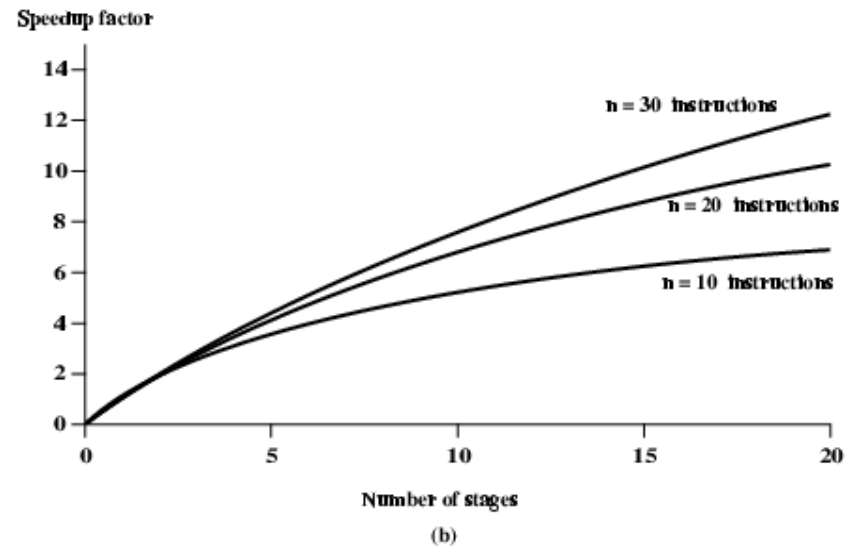
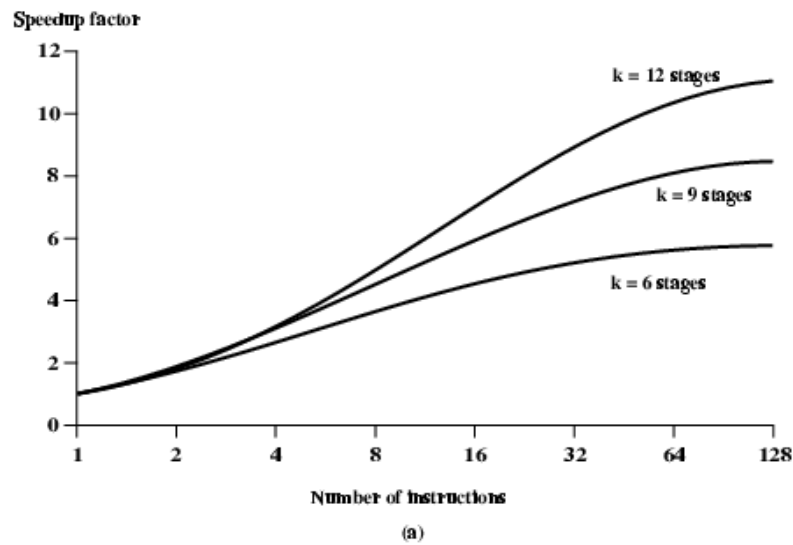
## Branch in a Pipeline



## Alternative Pipeline Depiction



# Speedup Factors with Instruction Pipelining



# Dealing with Branch

- Multiple Streams
- Prefetch Branch Target
- Loop buffer
- Branch prediction
- Delayed branching

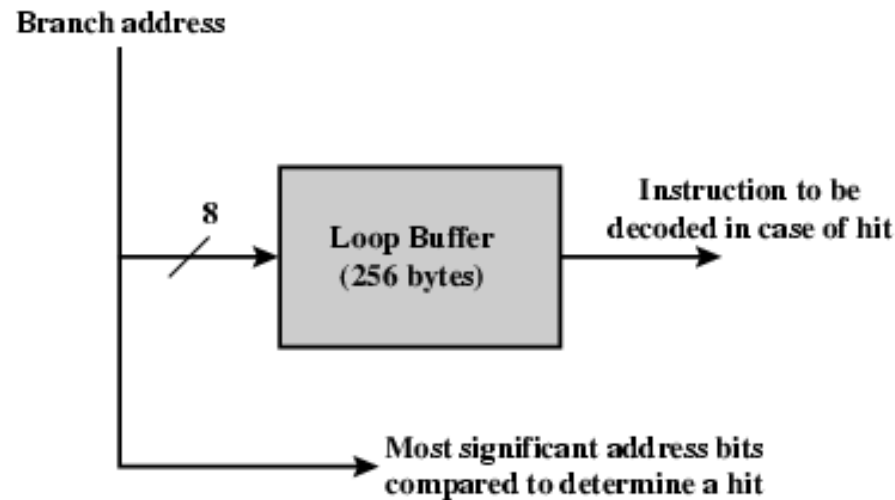
# Dealing with Branch

- **Multiple Streams**
  - There are two pipelines.
  - CPU prefetches each branch into a separate pipeline.
  - CPU uses appropriate pipeline.
  - This leads to bus and register contention.
  - Multiple branches lead to further pipelines being needed.
- **Prefetch Branch Target**
  - Target of branch is prefetched in addition to instructions following branch.
  - CPU keeps target until branch is executed.

## Dealing with Branch (2)

- **Loop Buffer**

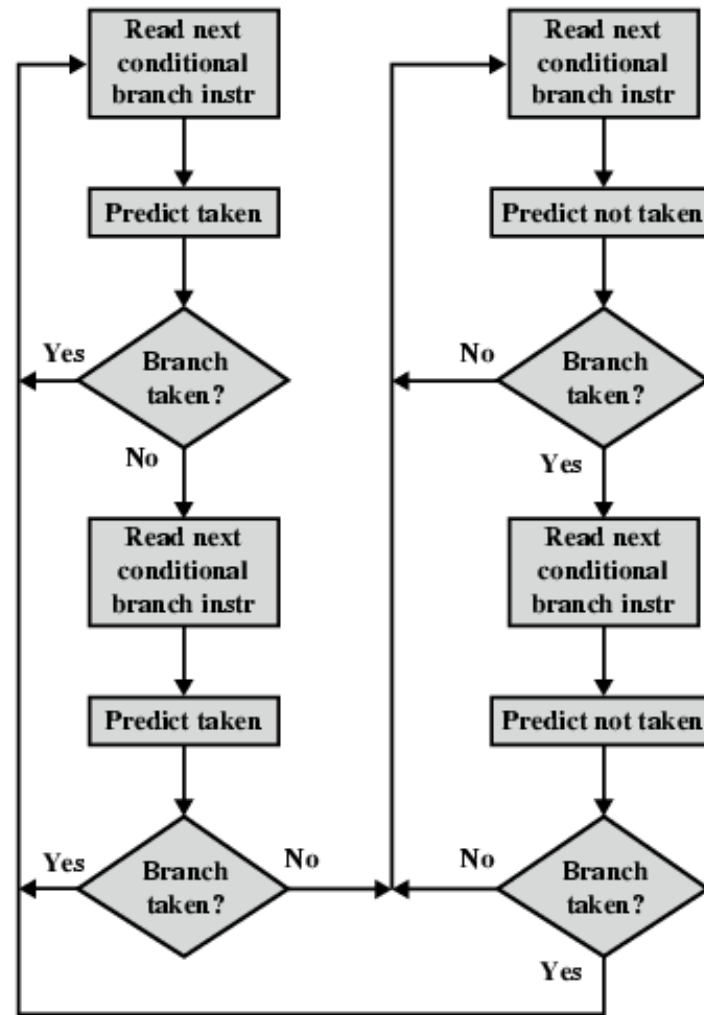
- Loop buffer is a small and very high speed memory.
- It is maintained by fetch stage of pipeline.
- CPU checks loop buffer before fetching from memory.
- This implementation is very good for small loops or jumps.
- It is similar in principle to an instruction cache.
- It is used by CRAY-1.



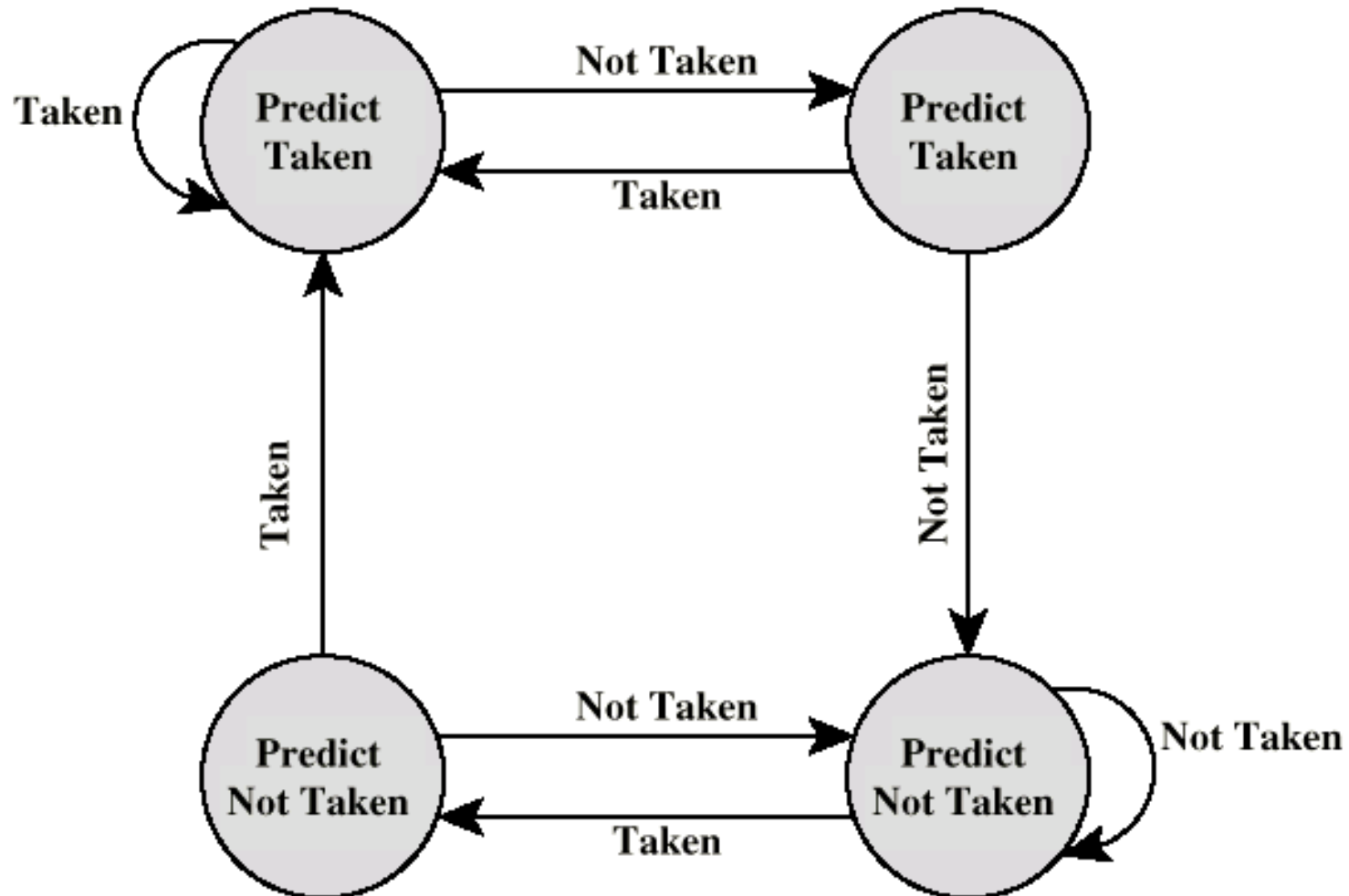
# Dealing with Branch (2)

- **Branch Prediction**
  - **Predict never taken**
    - It is assumed that jump will not happen.
    - CPU always fetches next instruction.
  - **Predict always taken**
    - It is assumed that jump will happen.
    - CPU always fetches target instruction.
  - **Predict by Opcode**
    - Some instructions are more likely to result in a jump than others.
    - It can get up to 75% success.
  - **Taken/Not Taken Switch**
    - It is based on previous history.
    - It is good for loops.
  - **Branch History Table (cache)**

## Branch Prediction Flowchart



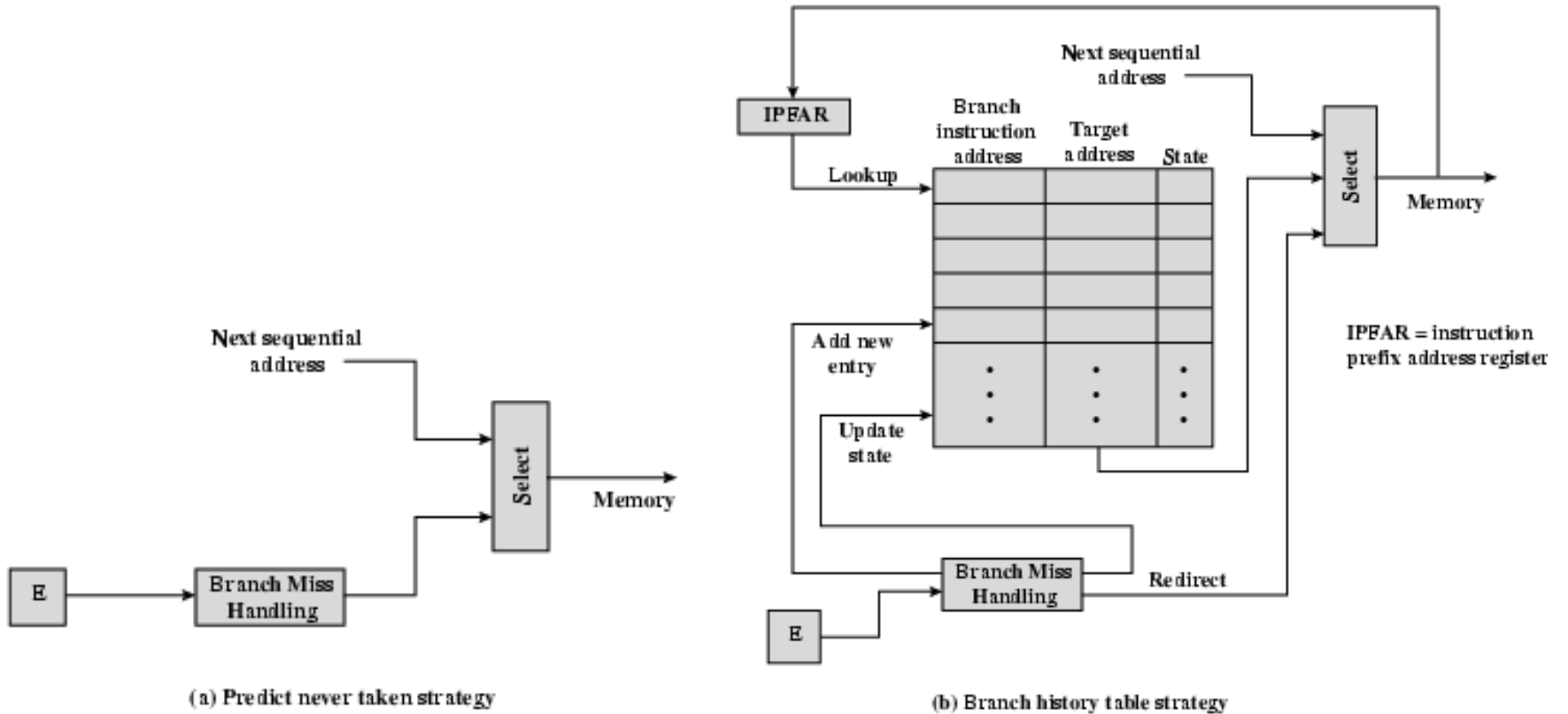
## Branch Prediction State Diagram



## Dealing with Branch (3)

- **Delayed Branch**
  - CPU does not take jump until it has to.
  - Compiler rearranges instruction sequence.

# Dealing with Branches



# Intel 486 Pipelining

- **Fetch**

- From cache or external memory
- Put in one of two 16-byte prefetch buffers
- Fill buffer with new data as soon as old data consumed
- Average 5 instructions fetched per load
- Independent of other stages to keep buffers full

- **Decode stage 1**

- Opcode & address-mode info
- At most first 3 bytes of instruction
- Can direct D2 stage to get rest of instruction

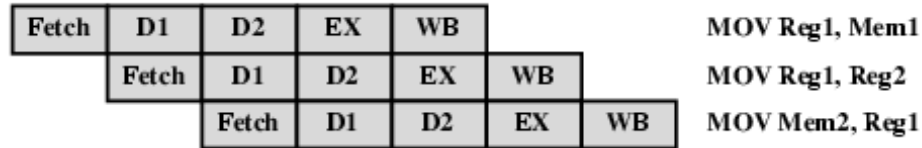
- **Decode stage 2**

- Expand opcode into control signals
- Computation of complex address modes

- **Execute**

- ALU operations, cache access, register update
- Writeback
- Update registers & flags
- Results sent to cache & bus interface write buffers

## 80486 Instruction Pipeline Examples



(a) No Data Load Delay in the Pipeline



(b) Pointer Load Delay



(c) Branch Instruction Timing