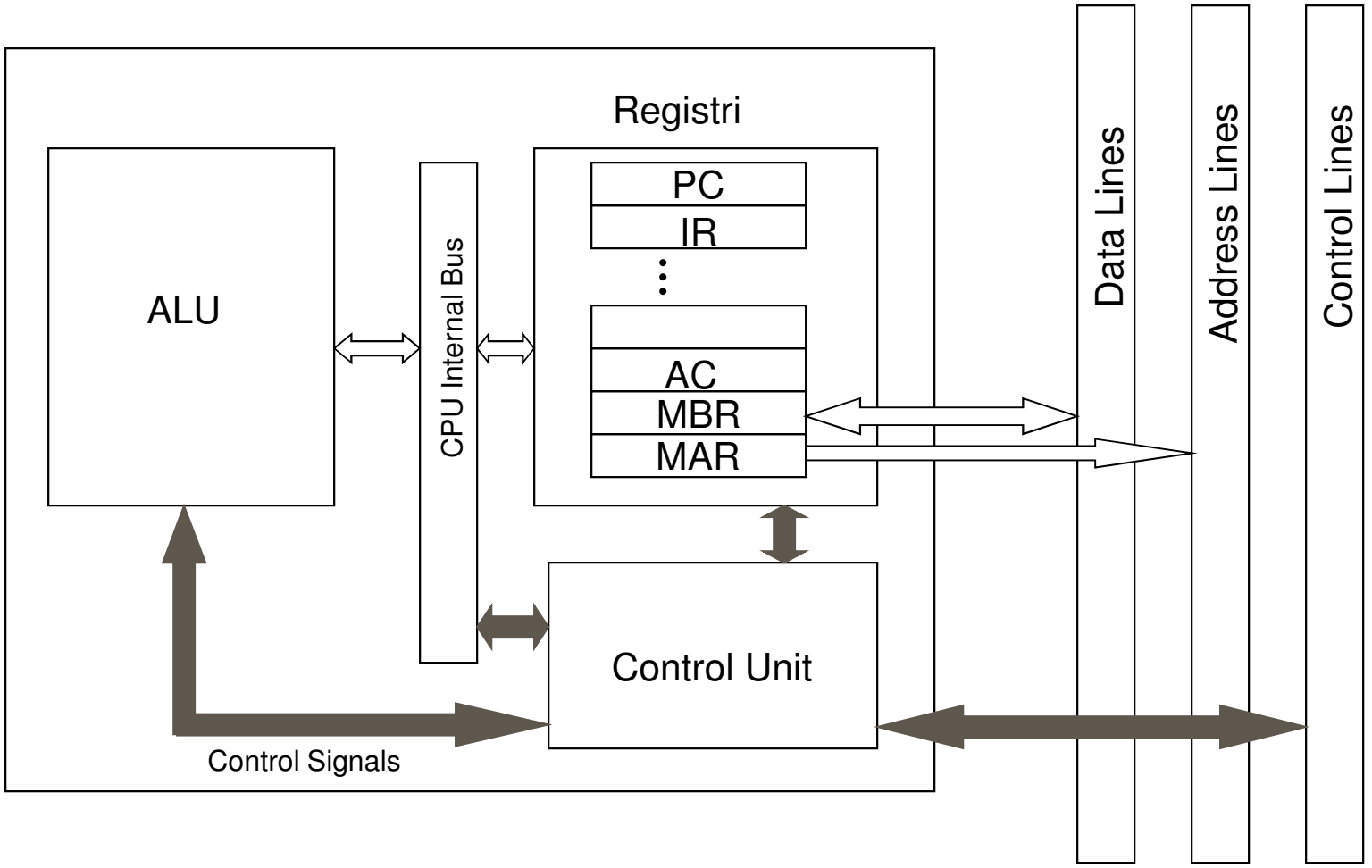
William Stallings
Computer Organization and Architecture

Chapter 12
CPU Structure and Function

CPU Functions

- CPU must:
  - Fetch instructions
  - Decode instructions
  - Fetch operands
  - Execute instructions / Process data
  - Store data
  - Check (and possibly serve) interrupts

CPU Components



Kind of Registers

- User visible and modifiable
  - General Purpose
  - Data (e.g. accumulator)
  - Address (e.g. base addressing, index addressing)
- Control registers (not visible to user)
  - Program Counter (PC)
  - Instruction Decoding Register (IR)
  - Memory Address Register (MAR)
  - Memory Buffer Register (MBR)
- State register (visible to user but not directly modifiable)
  - Program Status Word (PSW)

Kind of General Purpose Registers

- May be used in a general way or be restricted to contains only data or only addresses
- Advantages of general purpose registers
  - Increase flexibility and programmer options
  - Increase instruction size & complexity
- Advantages of specialized (data/address) registers
  - Smaller (faster) instructions
  - Less flexibility

How Many General Purposes Registers?

- Between 8 - 32
- Fewer = more memory references
- More does not reduce memory references and takes up processor real estate

How many bits per register?

- Large enough to hold full address value
- Large enough to hold full data value
- Often possible to combine two data registers to obtain a single register with a double length

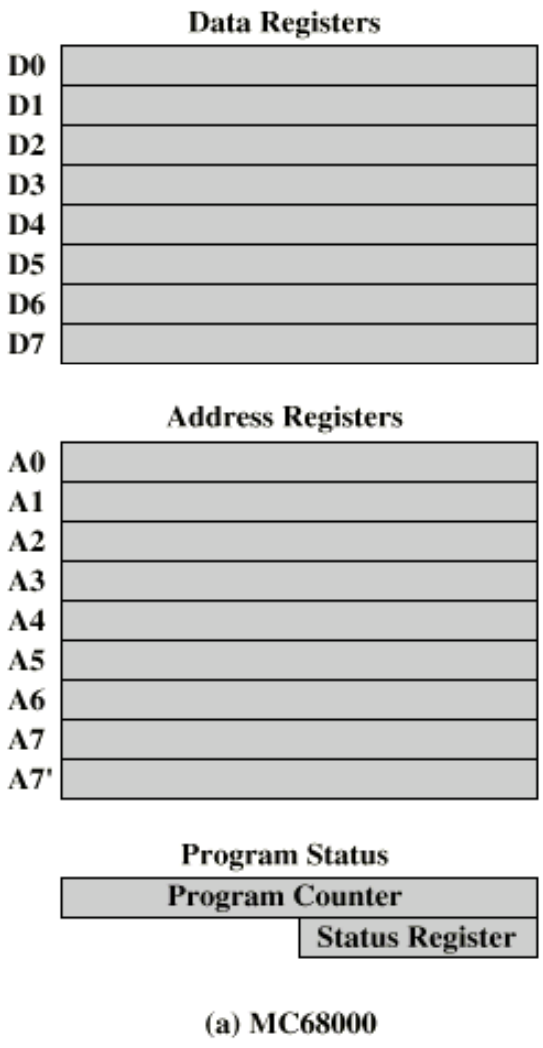
State Registers

- Sets of individual bits
  - e.g. store if result of last operation was zero or not
- Can be read (implicitly) by programs
  - e.g. Jump if zero
- Can not (usually) be set by programs
- There is always a Program Status Word (see later)
- Possibly (for operating system purposes):
  - Interrupt vectors
  - Memory page table (virtual memory)
  - Process control blocks (multitasking)

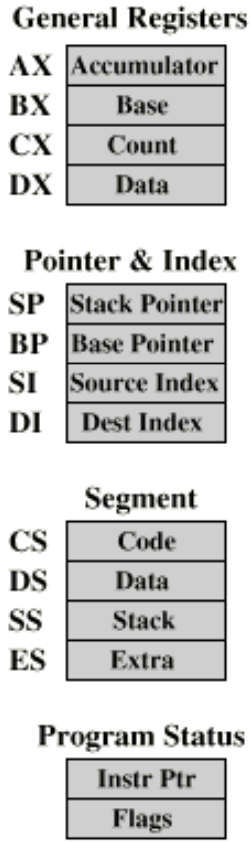
Program Status Word

- A set of bits, including condition code bits, giving the status of the program
  - Sign of last result
  - Zero
  - Carry
  - Equal
  - Overflow
  - Interrupt enable/disable
  - Supervisor mode (allow to execute privileged instructions)
    - Used by operating system (not available to user programs)

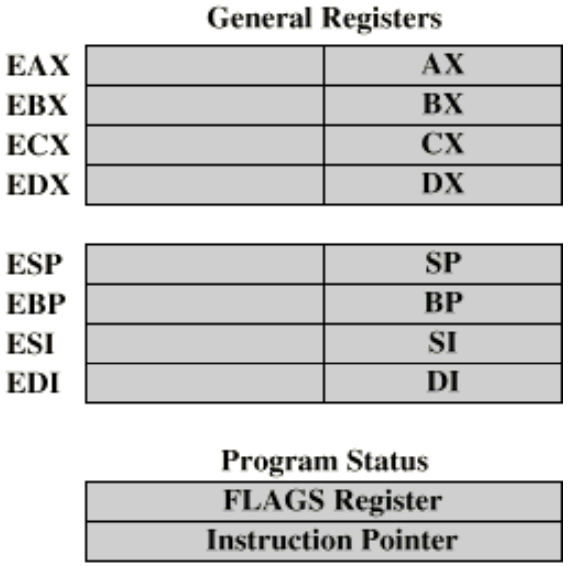
Example Register Organizations



(a) MC68000

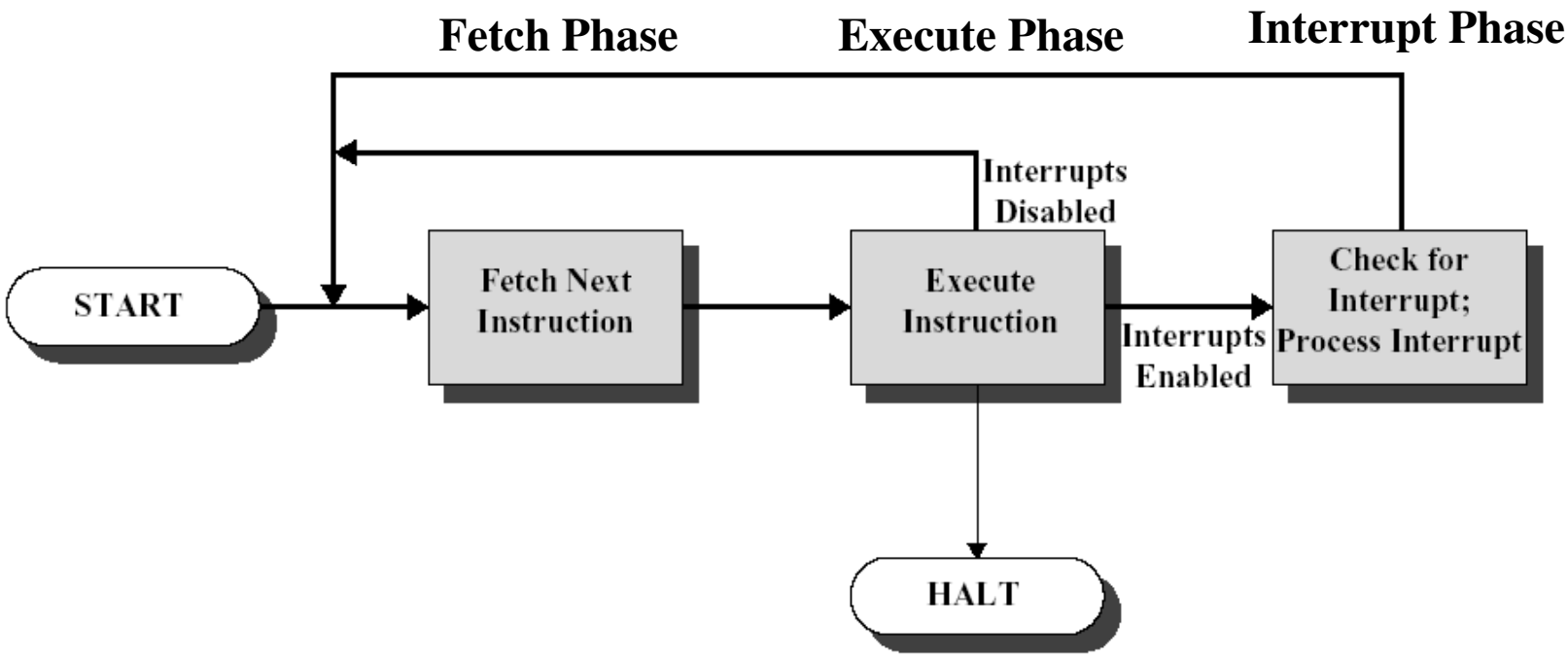


(b) 8086

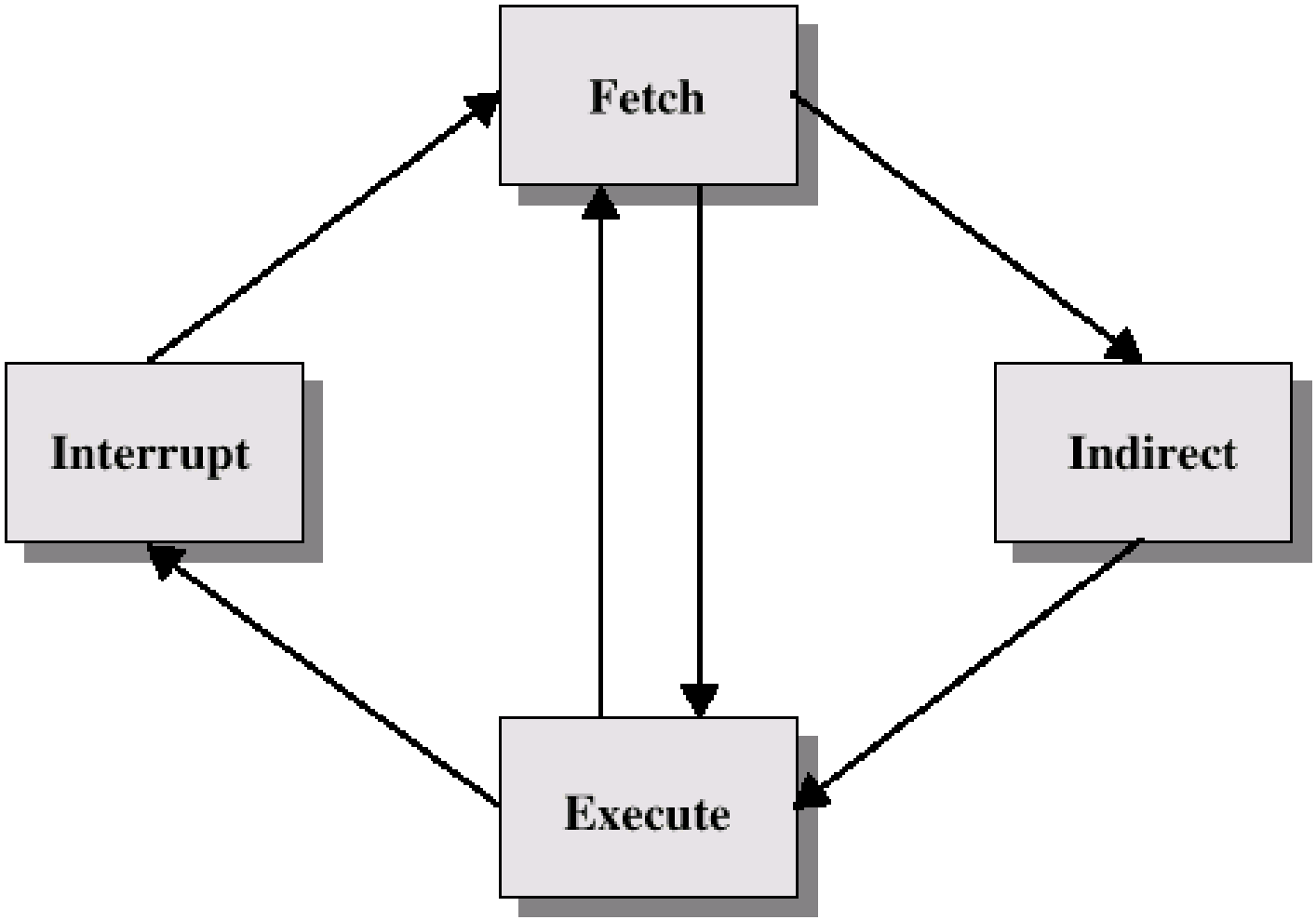


(c) 80386 - Pentium II

Instruction Cycle (with Interrupt)



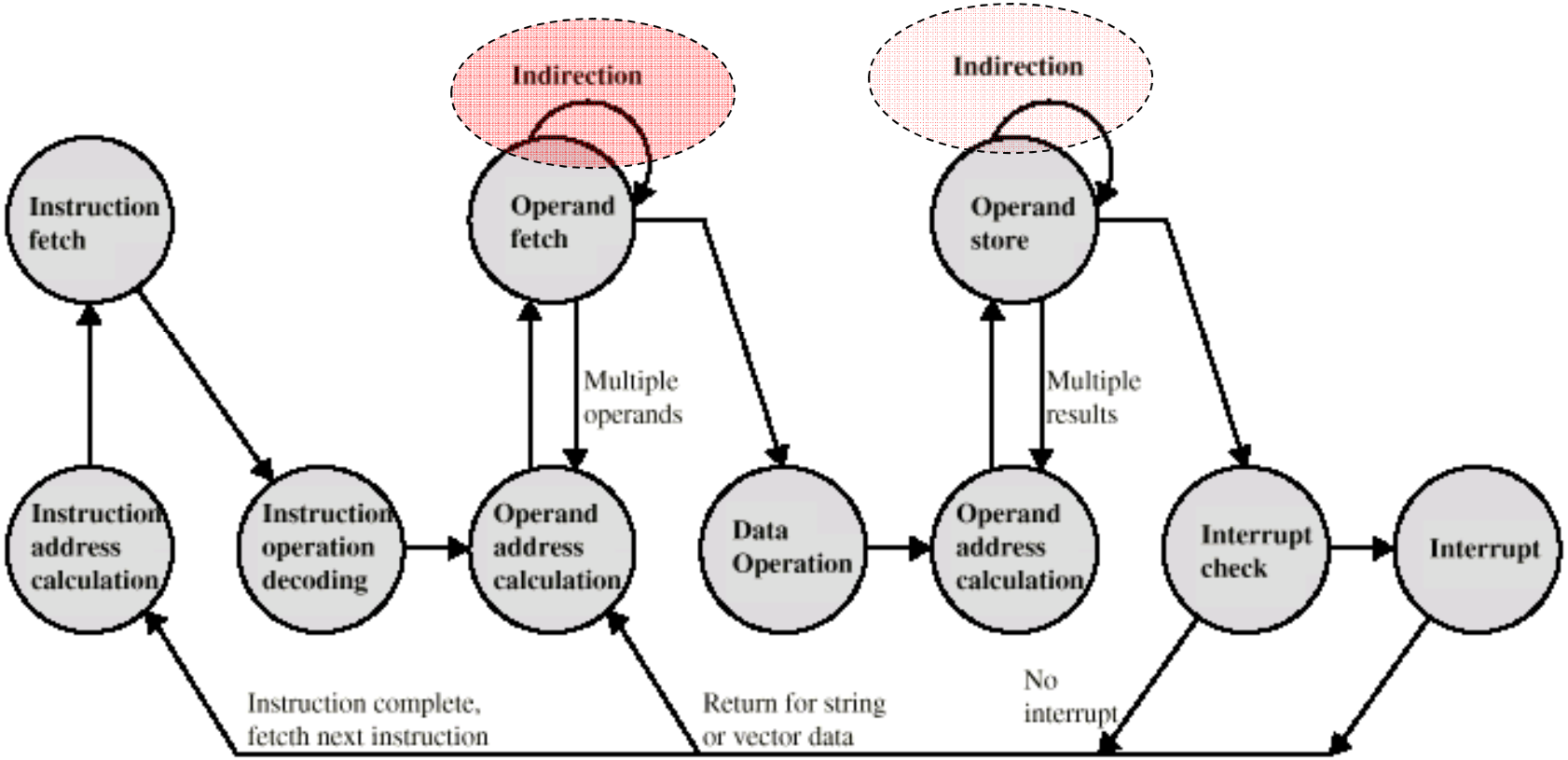
Instruction Cycle (with Indirect Addressing)



A closer look at the execution phase

- Execute
- Decode – Execute
- Decode – Fetch Operand – Execute
- Decode – Calculate Address – Fetch Operand – Execute
- Decode – Calculate Address – Fetch Address – Fetch Operand – Execute
- Decode – Calculate … – … – … Operand – Execute – Write Result

Instruction Cycle State Diagram (with Indirection)



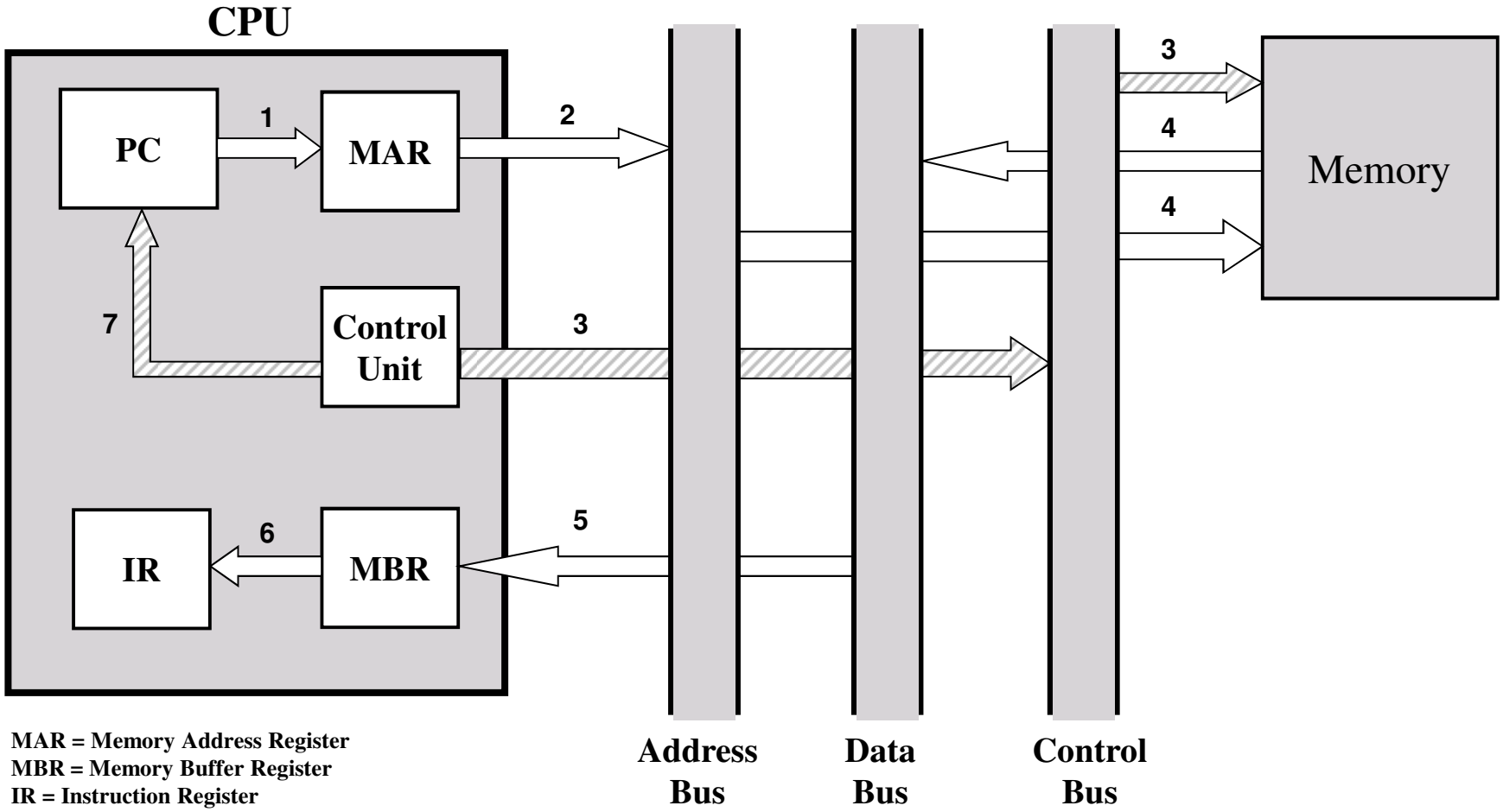
Data Flow for Instruction Fetch

- PC contains address of next instruction
- Sequence of actions needed to execute instruction fetch:
  1. PC is moved to MAR
  2. MAR content is placed on address bus
  3. Control unit requests memory read
  4. Memory read address bus and places result on data bus
  5. Data bus is copied to MBR
  6. MBR is copied to IR
  7. Meanwhile, PC is incremented by 1
- Action 7 can be executed in parallel with any other action after the first

Diagrams representing Data Flows

- The previous example shows 7 distinct actions, each corresponding to a DF (= data flow)
- Distinct DFs are not necessarily executed at distinct time steps (i.e.: DF_n and DF_{n+1} might be executed during the same time step – see chapter 14)
- Large arrows in white represents DFs with a true flow of data
- Large hatched arrows represents DFs where flow of data acts as a control: only the more relevant controls are shown

Data Flow Diagram for Instruction Fetch

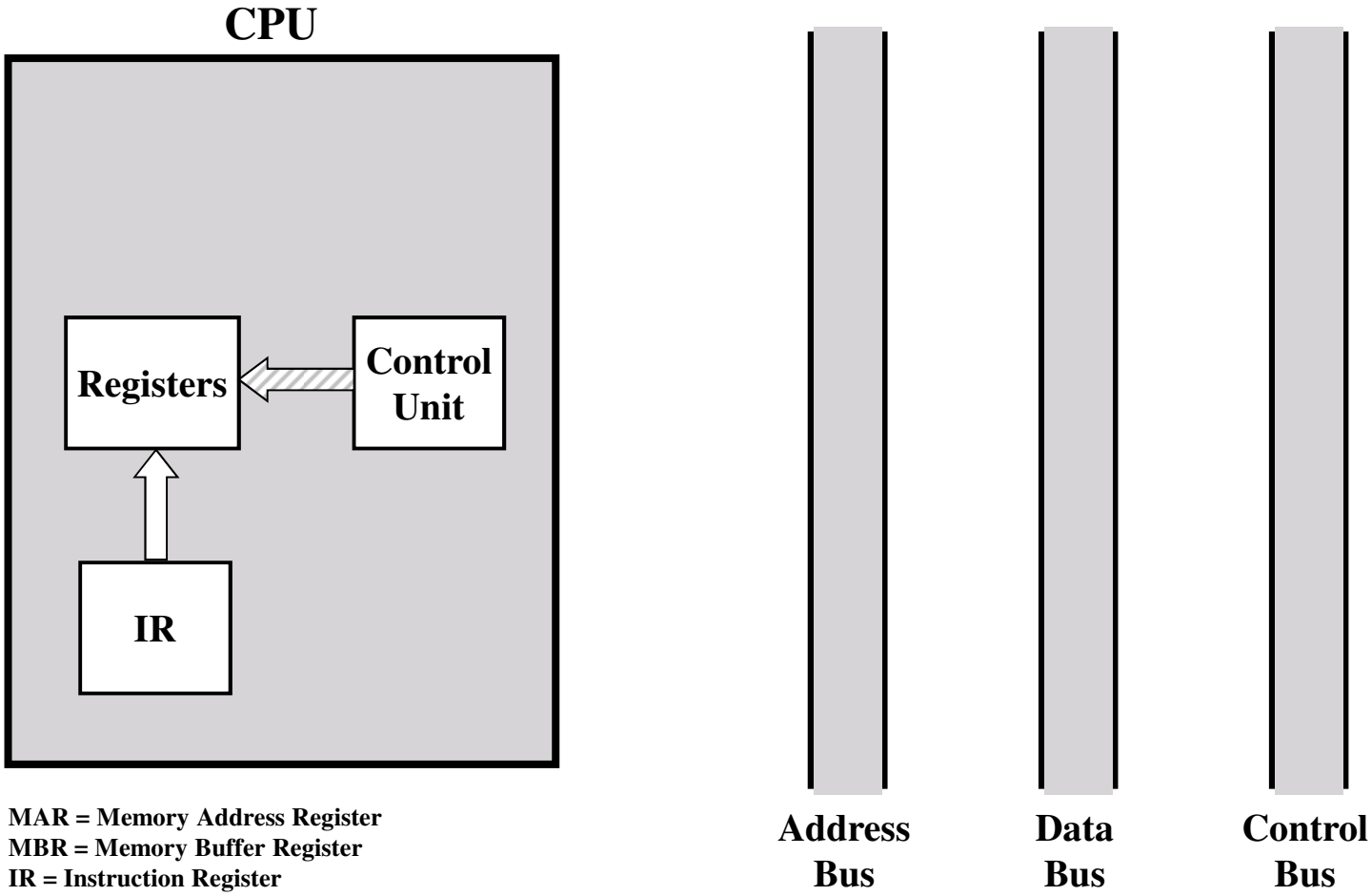


MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter

Data Flow for Data Fetch: Immediate and Register Addressing

- ALWAYS:
  - IR is examined to determine addressing mode
- Immediate addressing:
  - The operand is already in IR
- Register addressing:
  - Control unit requests read from register selected according to value in IR

Data Flow Diagram for Data Fetch with Register Addressing

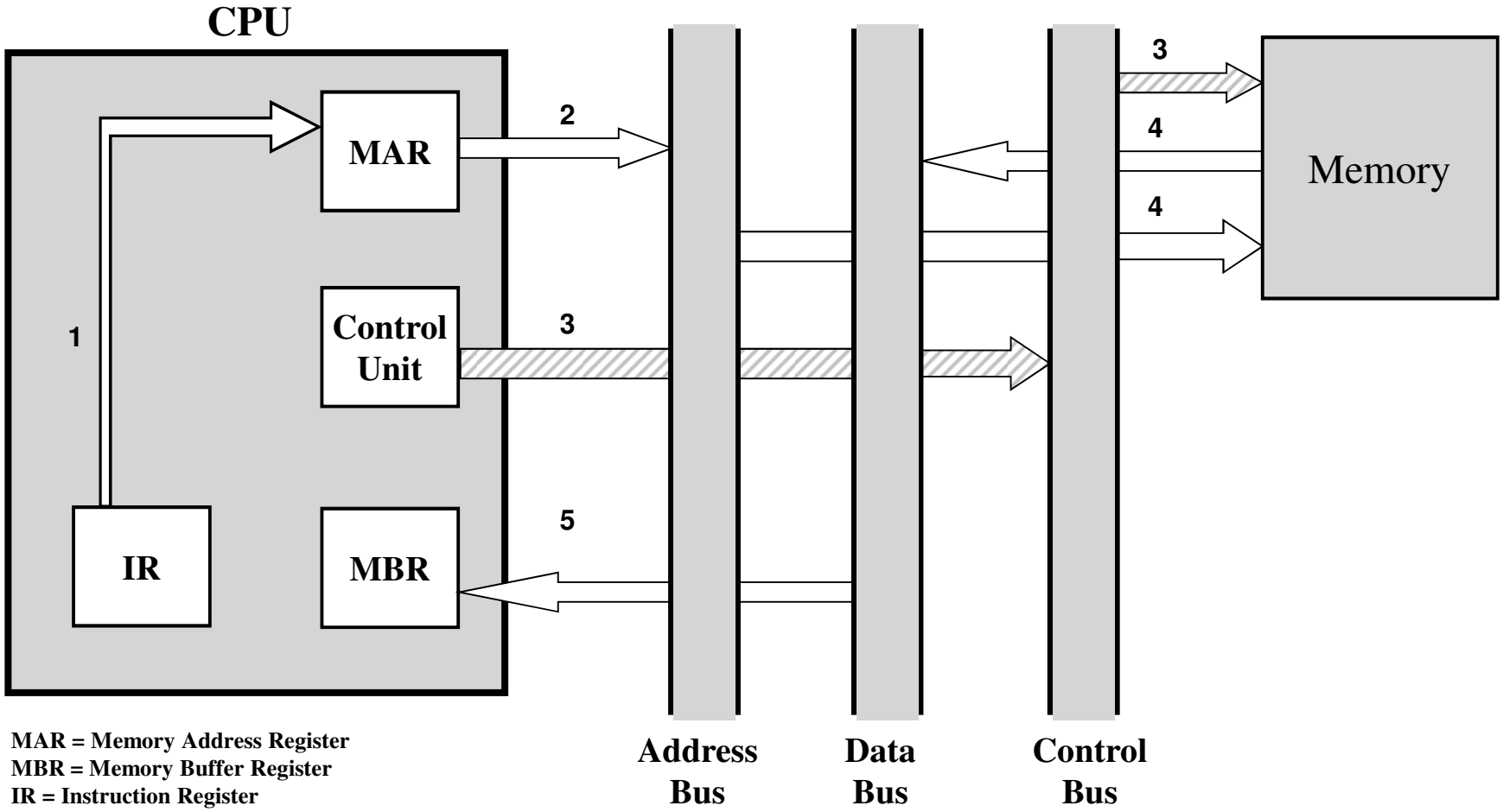


MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter

Data Flow for Data Fetch: Direct Addressing

- Direct addressing:
  1. Address field is moved to MAR
  2. MAR content is placed on address bus
  3. Control unit requests memory read
  4. Memory reads address bus and places result on data bus
  5. Data bus (= operand) is copied to MBR

Data Flow Diagram for Data Fetch with Direct Addressing

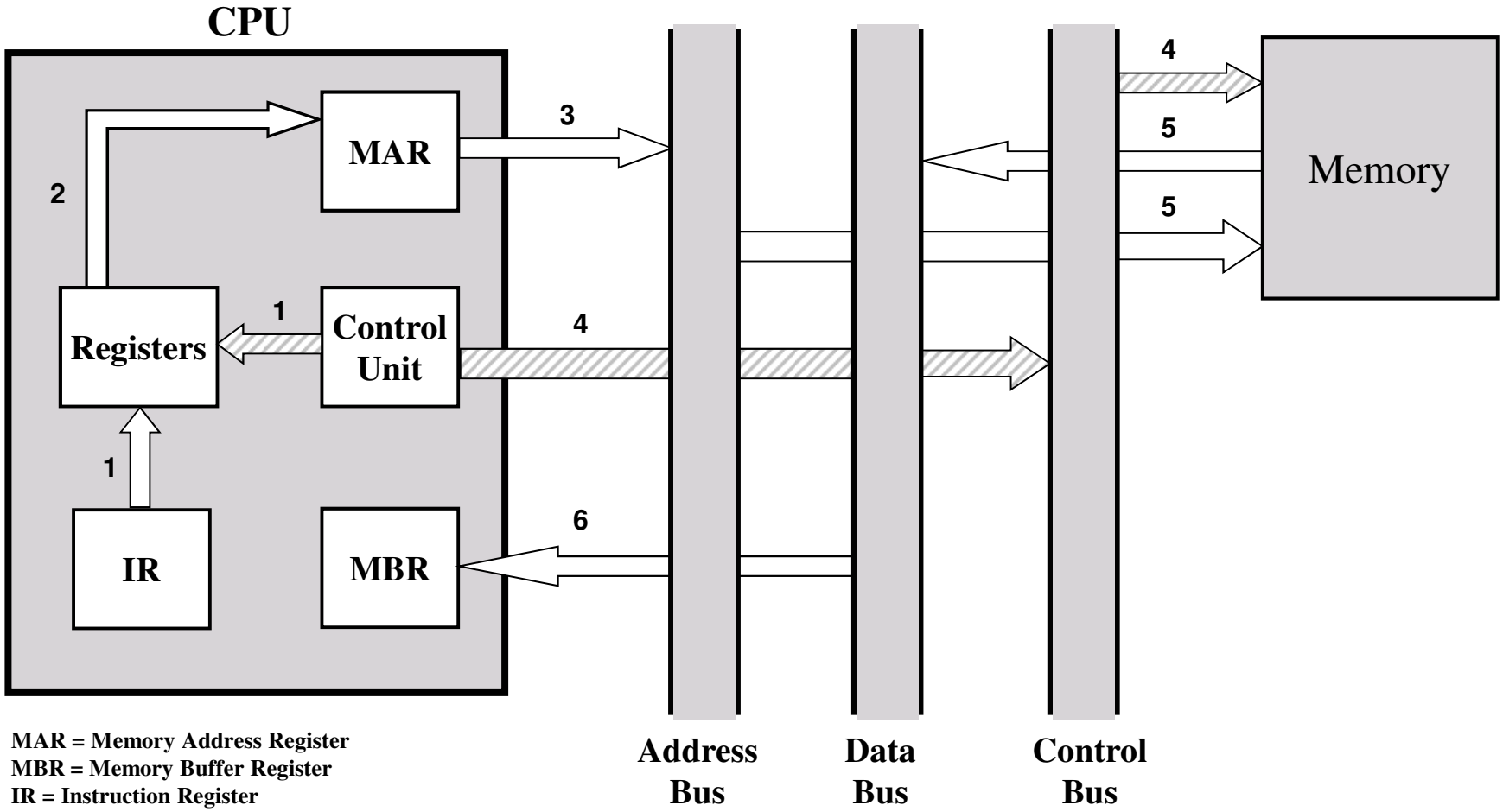


MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter

Data Flow for Data Fetch: Register Indirect Addressing

- Register indirect addressing:
  1. Control unit requests read from register selected according to value in IR
  2. Selected register value is moved to MAR
  3. MAR content is placed on address bus
  4. Control unit requests memory read
  5. Memory reads address bus and places result on data bus
  6. Data bus (= operand) is moved to MBR

Data Flow Diagram for Data Fetch with Register Indirect Addressing

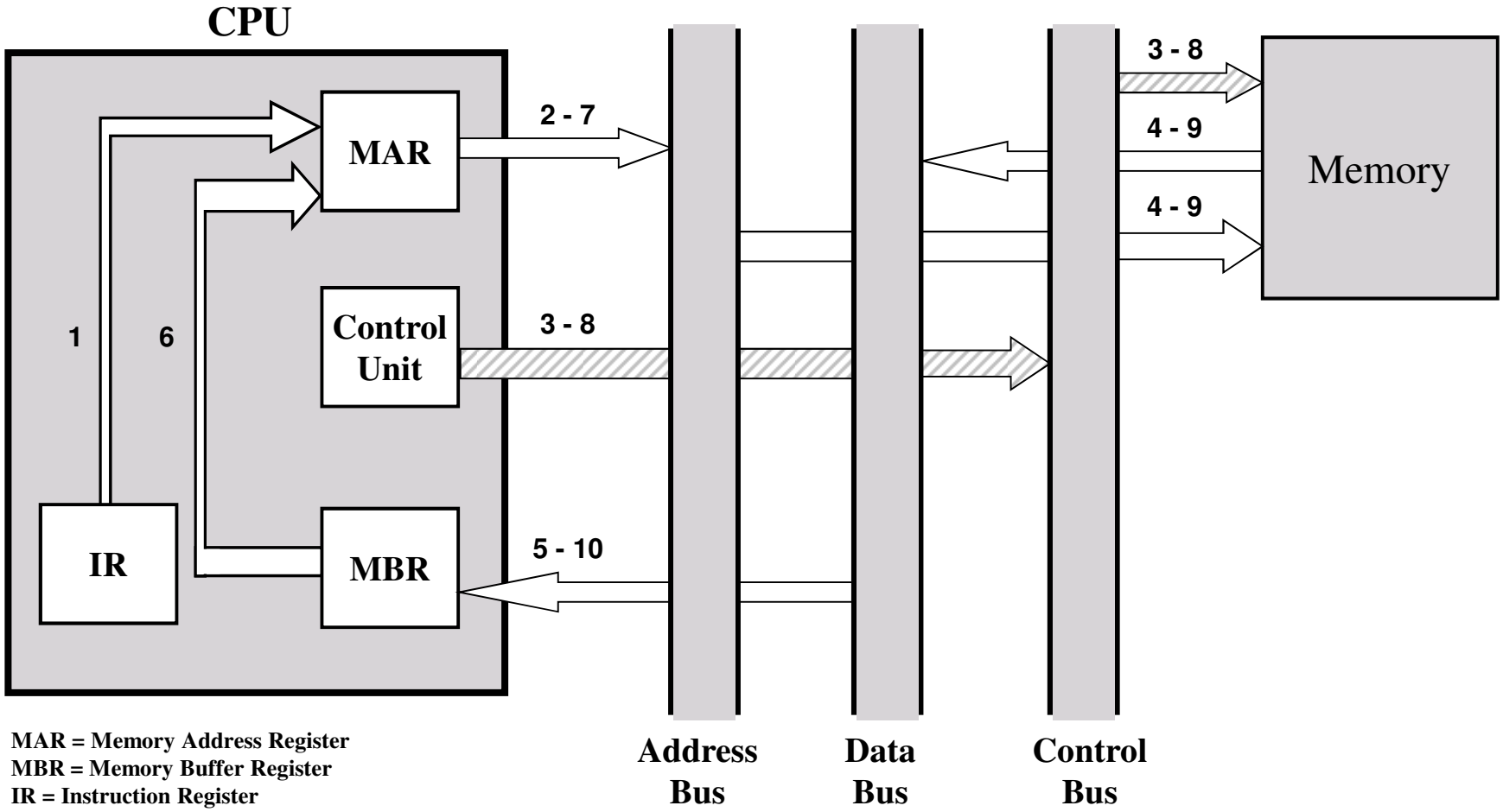


MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter

Data Flow for Data Fetch: Indirect Addressing

- Indirect addressing:
  1. Address field is moved to MAR
  2. MAR content is placed on address bus
  3. Control unit requests memory read
  4. Memory reads address bus and places result on data bus
  5. Data bus (= address of operand) is moved to MBR
  6. MBR is transferred to MAR
  7. MAR content is placed on address bus
  8. Control unit requests memory read
  9. Memory reads address bus and places result on data bus
  10. Data bus (= operand) is copied to MBR

Data Flow Diagram for Data Fetch with Indirect Addressing

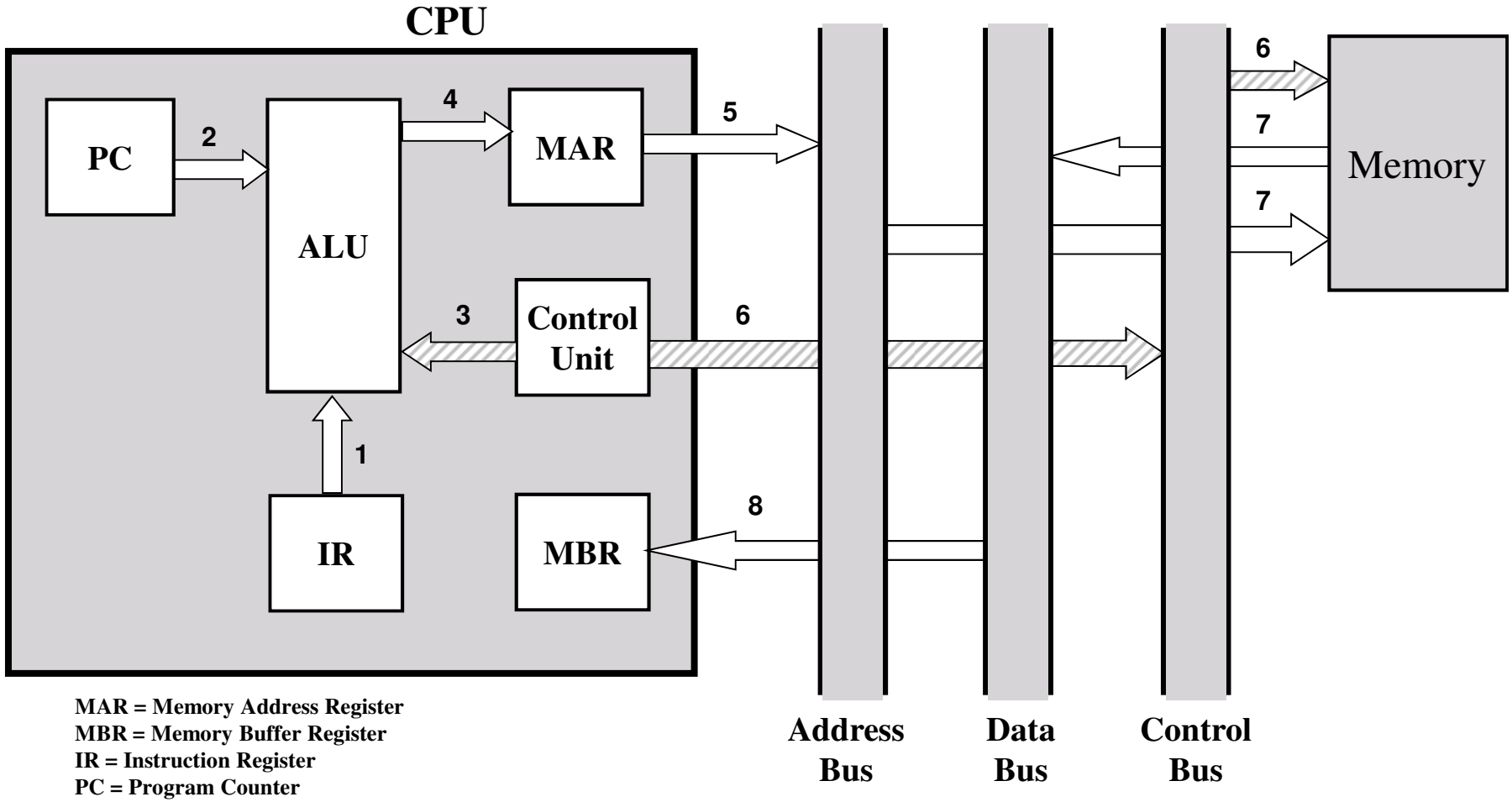


MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter

Data Flow for Data Fetch: Relative Addressing

- Relative addressing (a form of displacement):
  1. Address field is moved to ALU
  2. PC is moved to ALU
  3. Control unit requests sum to ALU
  4. Result from ALU is moved to MAR
  5. MAR content is placed on address bus
  6. Control unit requests memory read
  7. Memory reads address bus and places result on data bus
  8. Data bus (= operand) is copied to MBR

Data Flow Diagram for Data Fetch with Relative Addressing

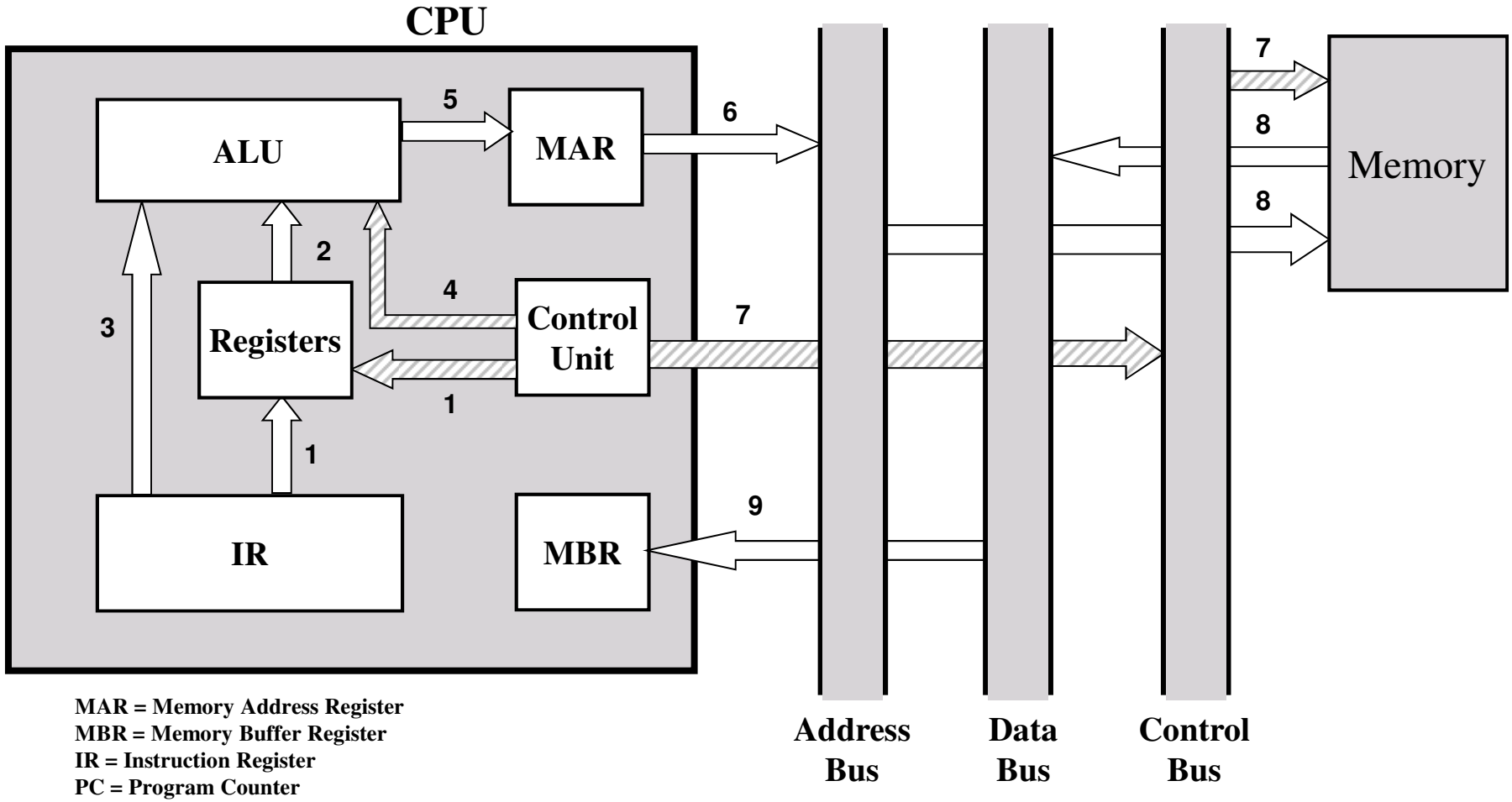


MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter
ALU = Arithmetic Logic Unit

Data Flow for Data Fetch: Base Addressing

- Base addressing (a form of displacement):
  1. Control unit requests read from register selected according to value in IR (explicit selection)
  2. Selected register value is moved to ALU
  3. Address field is moved to ALU
  4. Control unit requests sum to ALU
  5. Result from ALU is moved to MAR
  6. MAR content is placed on address bus
  7. Control unit requests memory read
  8. Memory reads address bus and places result on data bus
  9. Result (= operand) is moved to MBR

Data Flow Diagram for Data Fetch with Base Addressing



MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter
ALU = Arithmetic Logic Unit

Data Flow for Data Fetch: Indexed Addressing

- Same data flow as Base addressing
- Indexed addressing (a form of displacement):
  1. Control unit requests read from register selected according to value in IR (explicit selection)
  2. Selected register value is moved to ALU
  3. Address field is moved to ALU
  4. Control unit requests sum to ALU
  5. Result from ALU is moved to MAR
  6. MAR content is placed on address bus
  7. Control unit requests memory read
  8. Memory reads address bus and places result on data bus
  9. Result (= operand) is moved to MBR

Data Flow Diagram for Data Fetch with Indexed Addressing

- The diagram is the same as for Base addressing

Data Flow for Data Fetch with indirection and displacement

- Two different combinations of displacement and indirection (pre-index and post-index)
- See chapter 11 for the logical diagrams
- The data flow is a combination of what happens with the two techniques
- Try drawing the data flow diagrams yourself !

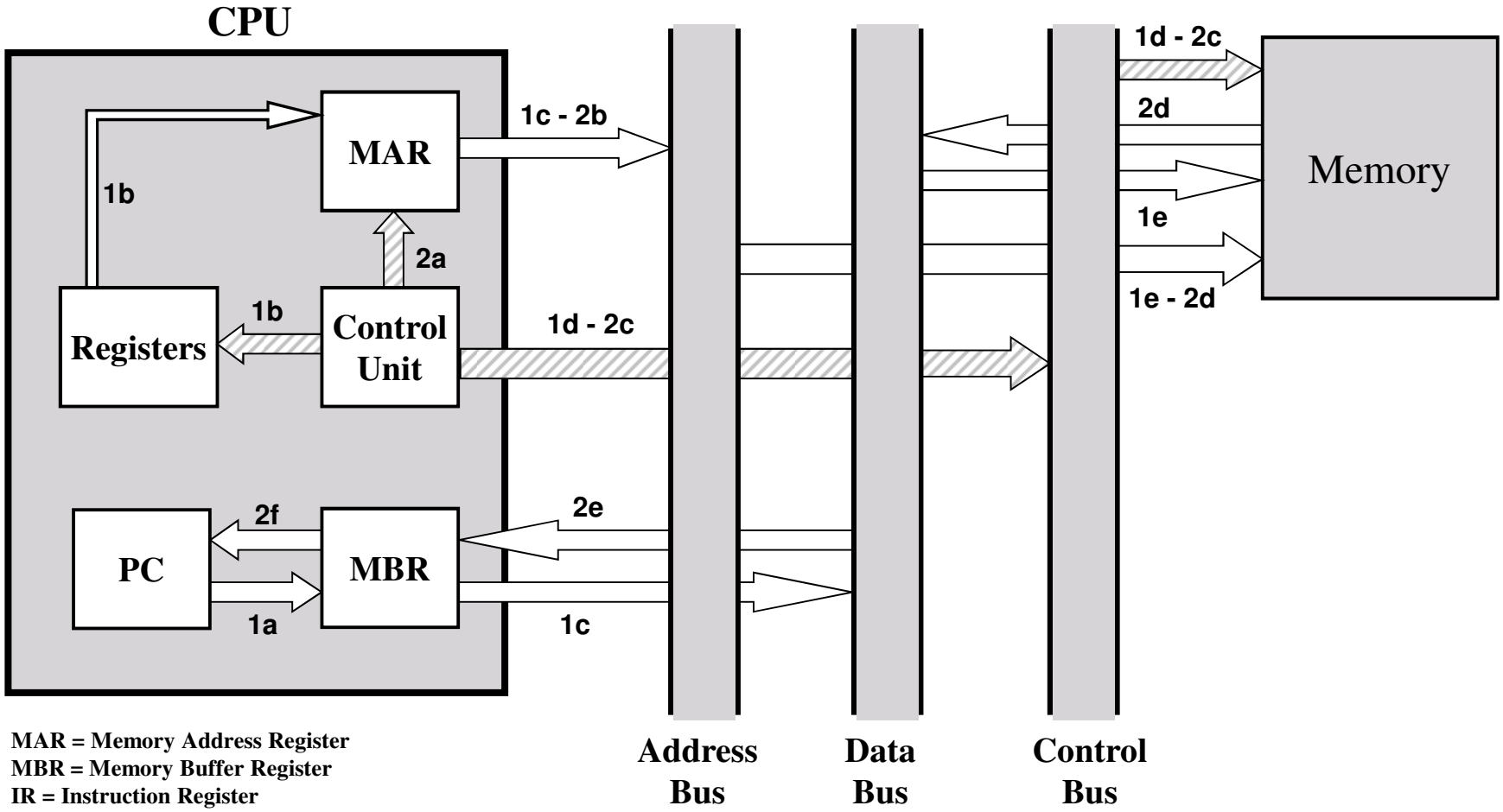
Data Flow for Execute

- May take many forms
- Depends on the actual instruction being executed
- May include
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations

Data Flow for Interrupt

- Current PC has to be saved (usually to stack) to allow resumption after interrupt and execution has to continue at the interrupt handler routine
  1. Save the content of PC
     a. Contents of PC is copied to MBR
     b. Special memory location (e.g. stack pointer register) is loaded to MAR
     c. Contents of MAR and MBR are placed, respectively, on address and data bus
     d. Control unit requests memory write
     e. Memory reads address and data bus and store to memory location
     f. Stack pointer register is updated (*data flow not shown*)
  2. PC is loaded with address of the handling routine for the specific interrupt (usually by means of indirect addressing through the **Interrupt Vector**)
     a. Move to MAR the address into the interrupt vector for the specific interrupt
     b. MAR content is placed on address bus
     c. Control unit requests memory read
     d. Memory reads address bus and places result on data bus
     e. Data bus is copied to MBR
     f. MBR is moved to PC
- Next instruction (first of the specific interrupt handler) can now be fetched

Data Flow Diagram for Interrupt



MAR = Memory Address Register
MBR = Memory Buffer Register
IR = Instruction Register
PC = Program Counter

Prefetch

- Fetch accesses main memory
- Execution usually does not access main memory
- CPU could fetch next instruction during the execution of current instruction
- Requires two sub-parts in CPU able to operate independently
- Called *instruction prefetch*

Improved Performance

- But performance is not doubled:
  - Fetch usually shorter than execution (but for simple operations with many operands)
    - Prefetch more than one instruction?
  - Any conditional jump or branch means that prefetched instructions may be useless
- Performance can be improved by adding more stages in instruction processing ...
  - ... and more independent sub-parts in CPU

Instruction pipelining

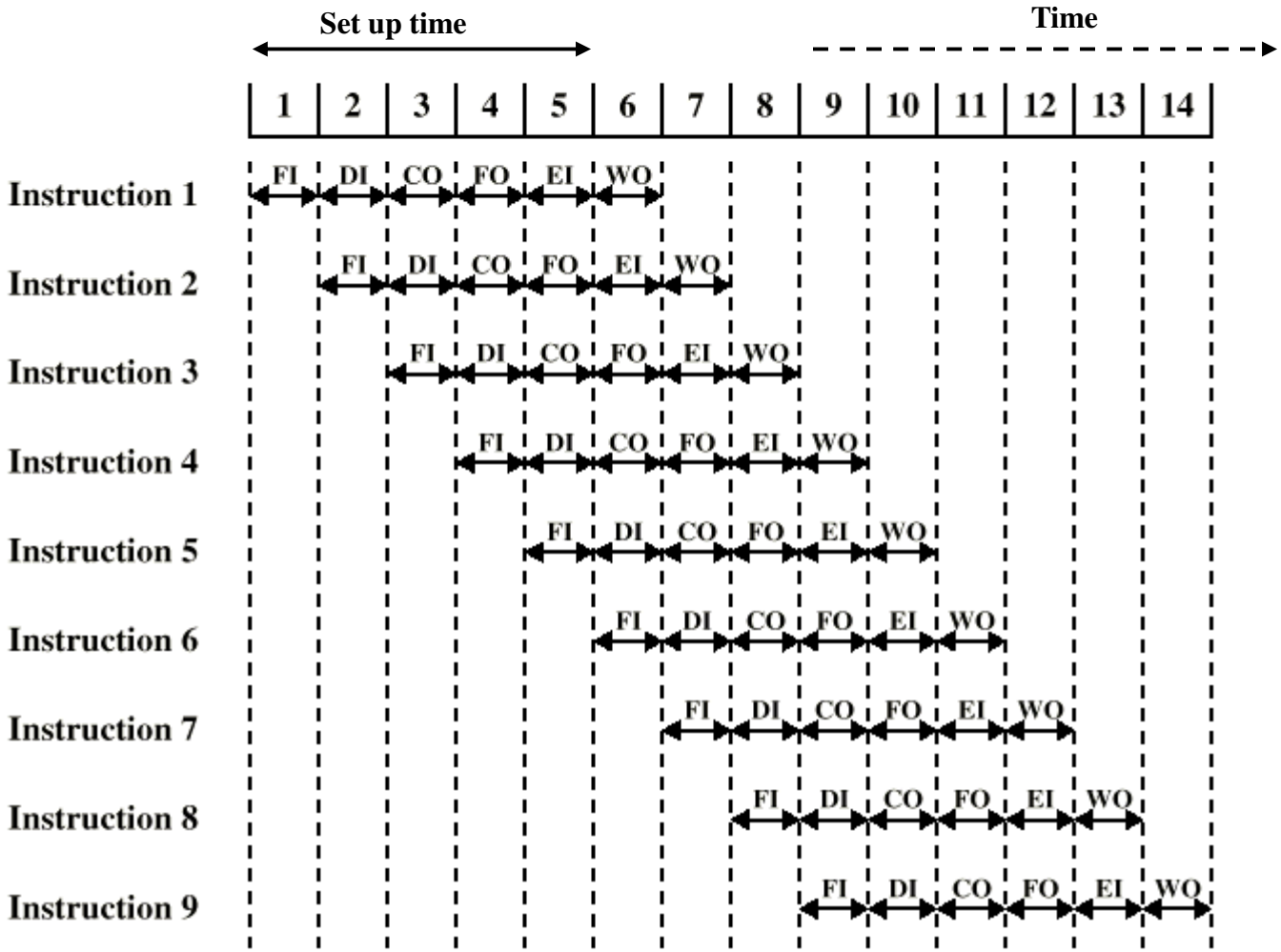
- Similar to the use of an assembly line in a manufacturing plant
  - product goes through various stages of production
  - products at various stages can be worked on simultaneously
- In a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end

Pipelining

- Instruction cycle can be decomposed in elementary phases, for example:
  - FI: Fetch instruction
  - DI: Decode instruction
  - CO: Calculate operands (i.e. calculate EAs)
  - FO: Fetch operands
  - EI: Execute instructions
  - WO: Write output

- **Pipelining** improves performance by overlapping these phases (ideally can all be overlapped)

Timing of Pipeline



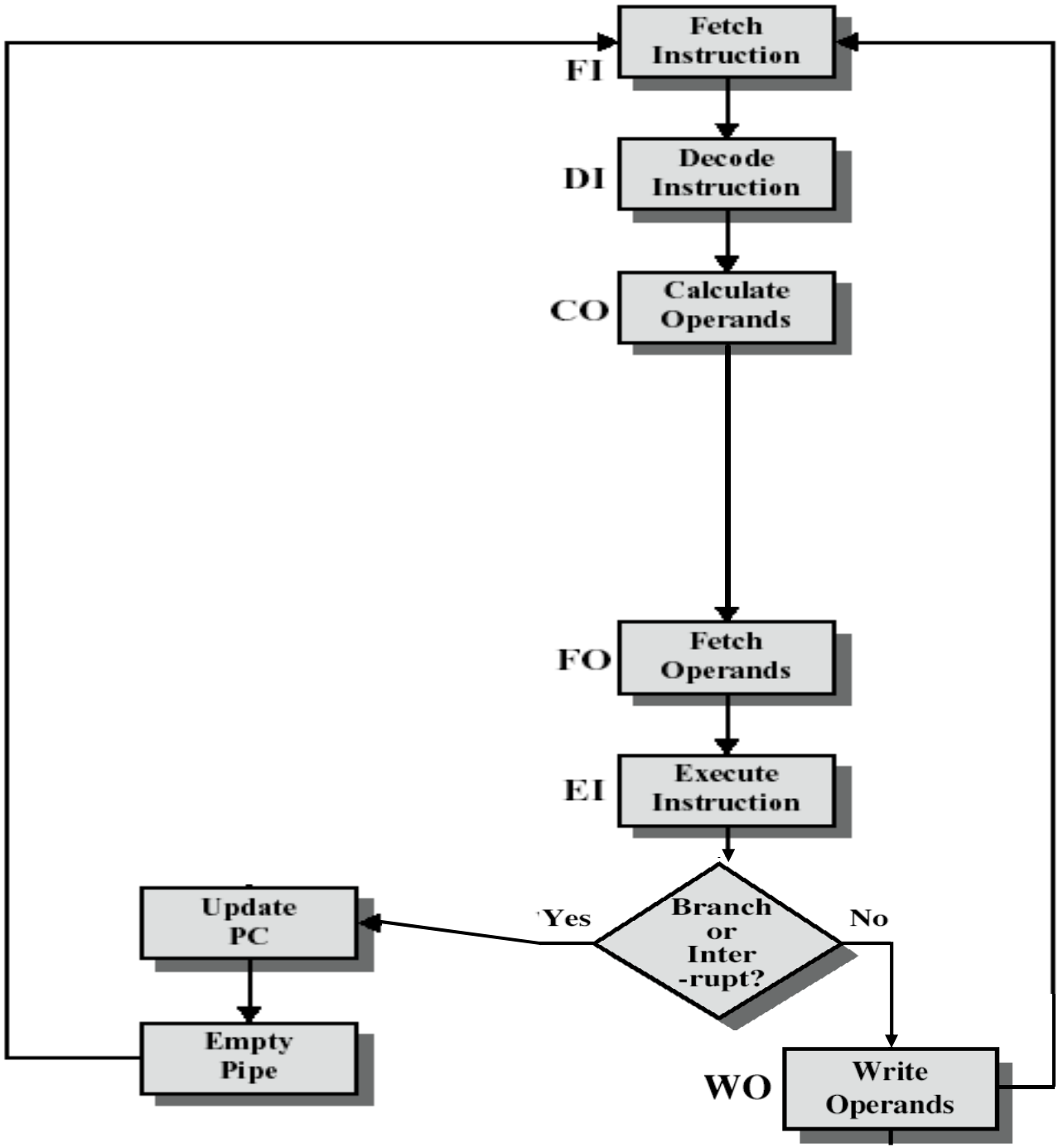
Some remarks

- An instruction can have only some of the six phases
- We are assuming that all the phases can be performed in parallel
  - e.g., no bus conflicts, no memory conflicts…
- The maximum improvement is obtained when the phases take more or less the same time

A general principle

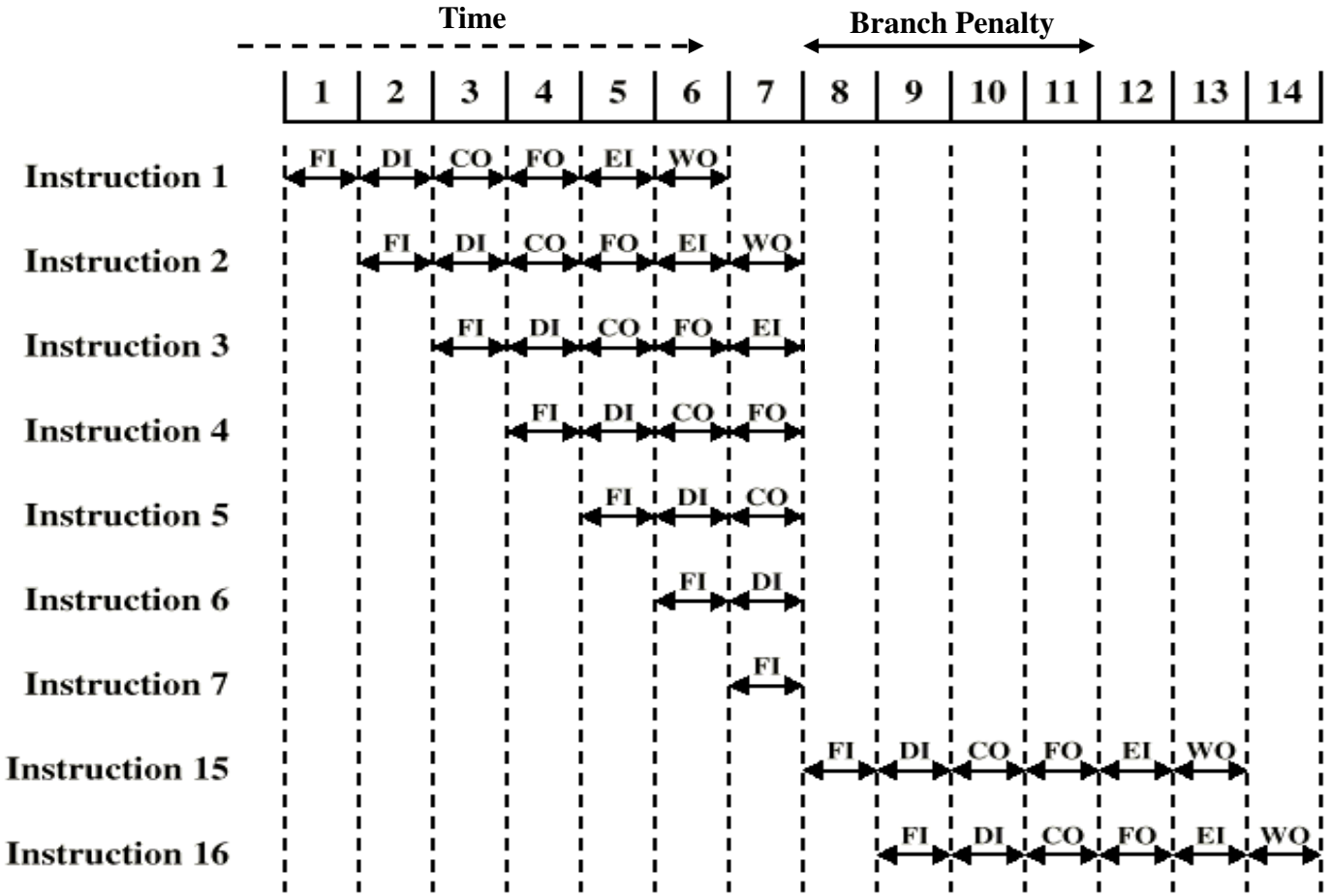
- The more overlapping phases are in a pipeline the more additional processing is needed to manage each phase and synchronization among phases
  - Logical dependencies between phases
- There is a trade-off between number of phases and speed-up of instruction execution

Control flow (1)



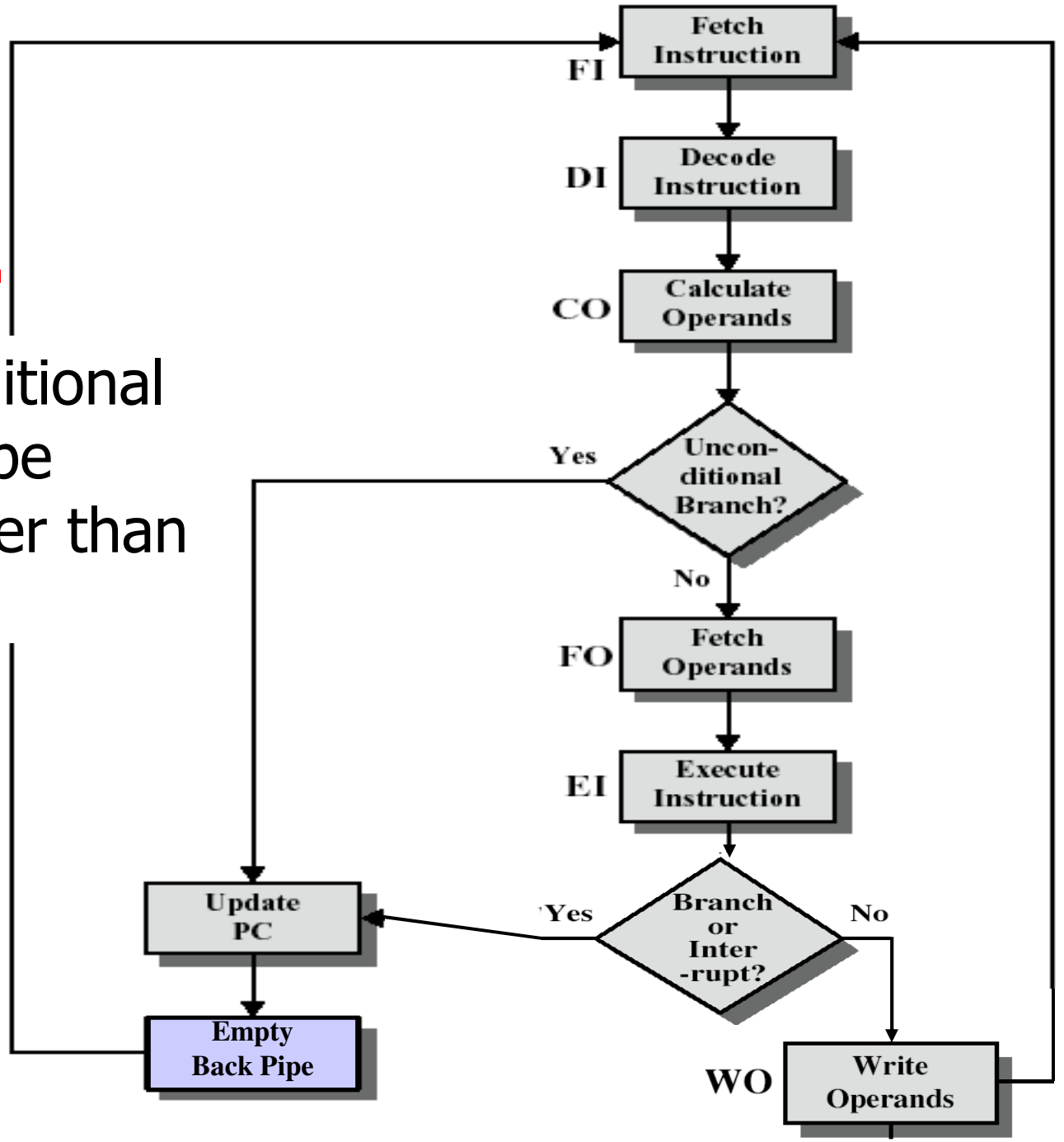
Branch in a Pipeline (1)

- Instruction 3 is an conditional branch to instruction 15



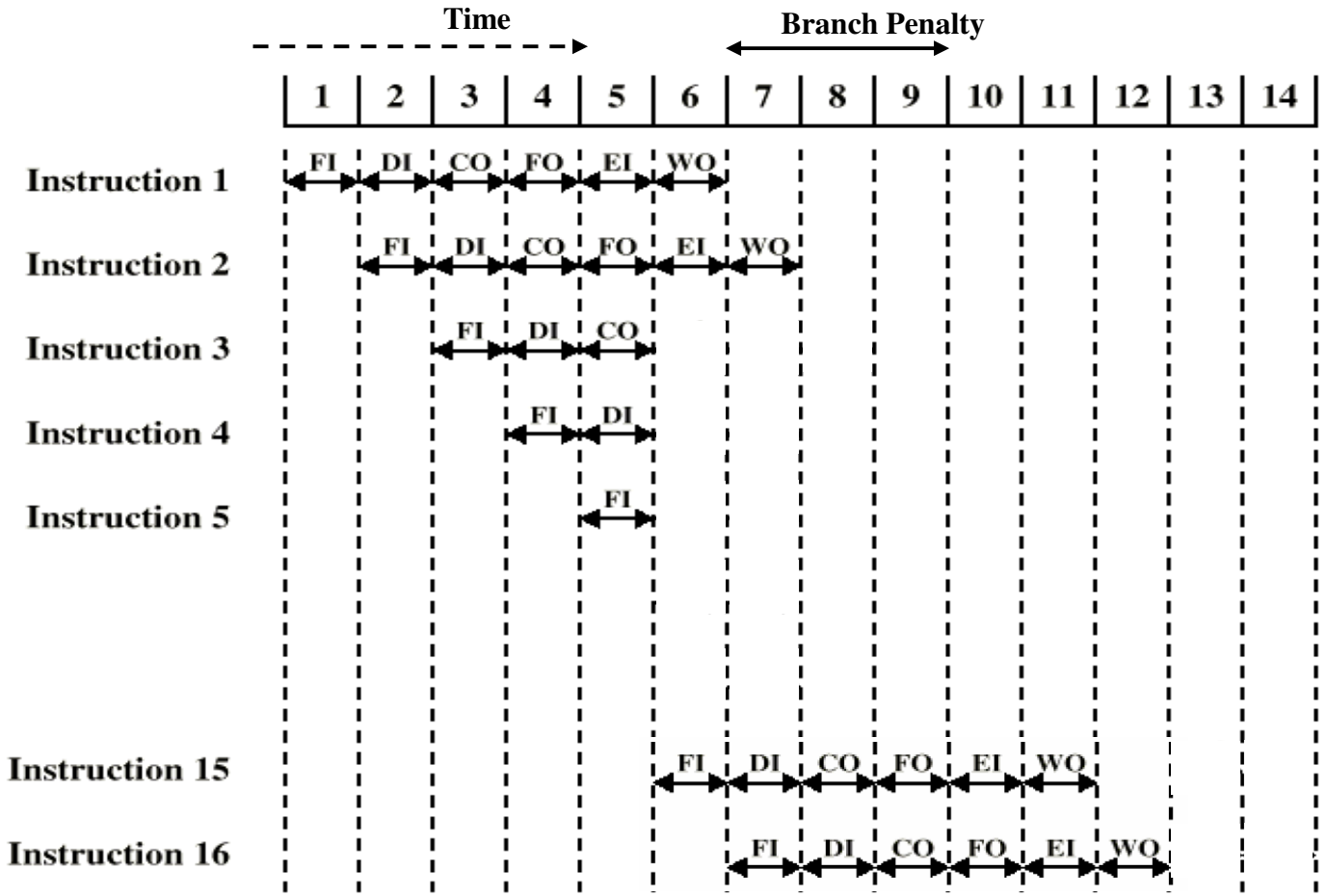
Control flow (2)

- But an unconditional branch might be managed earlier than EI phase



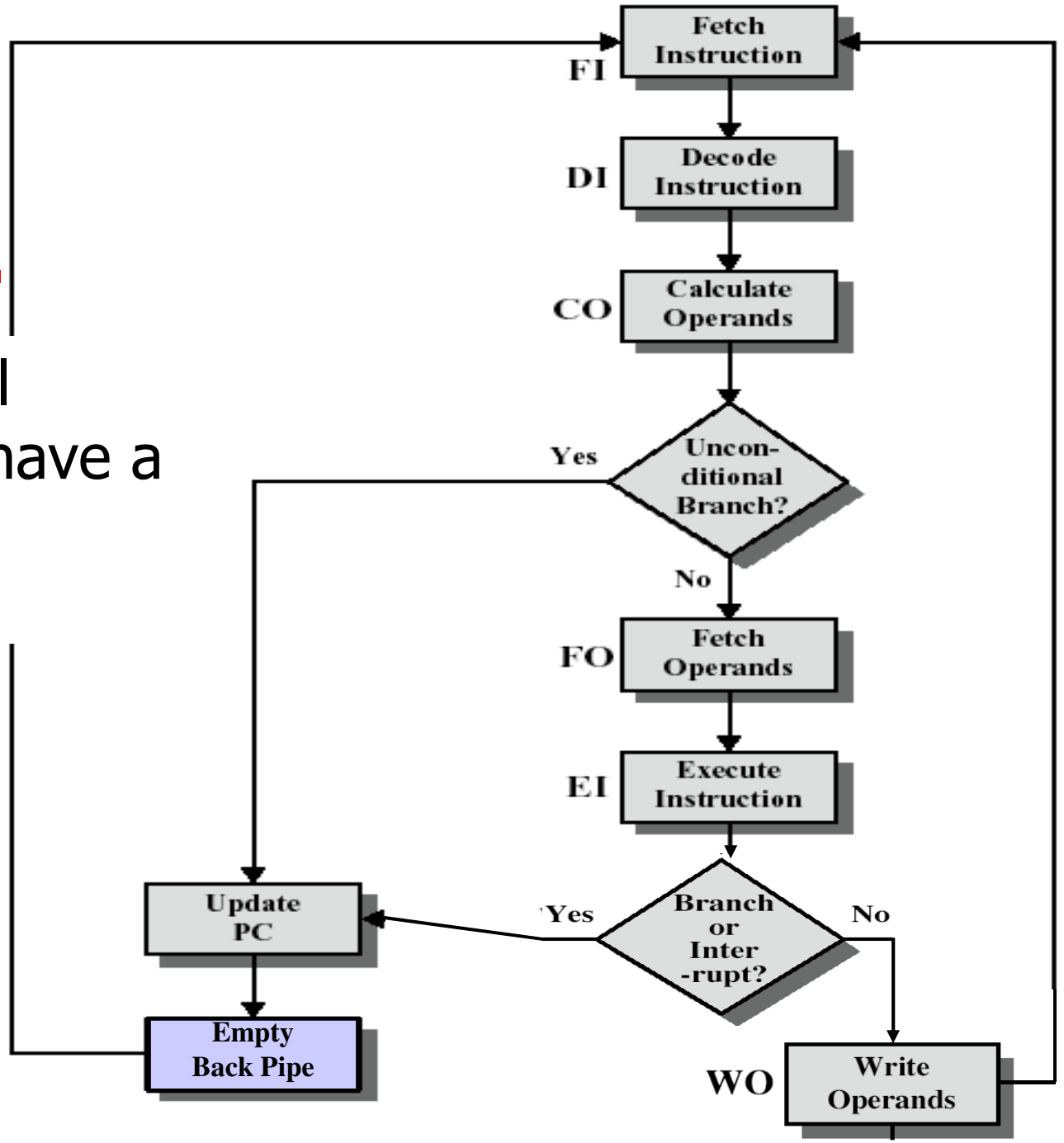
Branch in a Pipeline (2)

- The unconditional branch is managed after CO phase



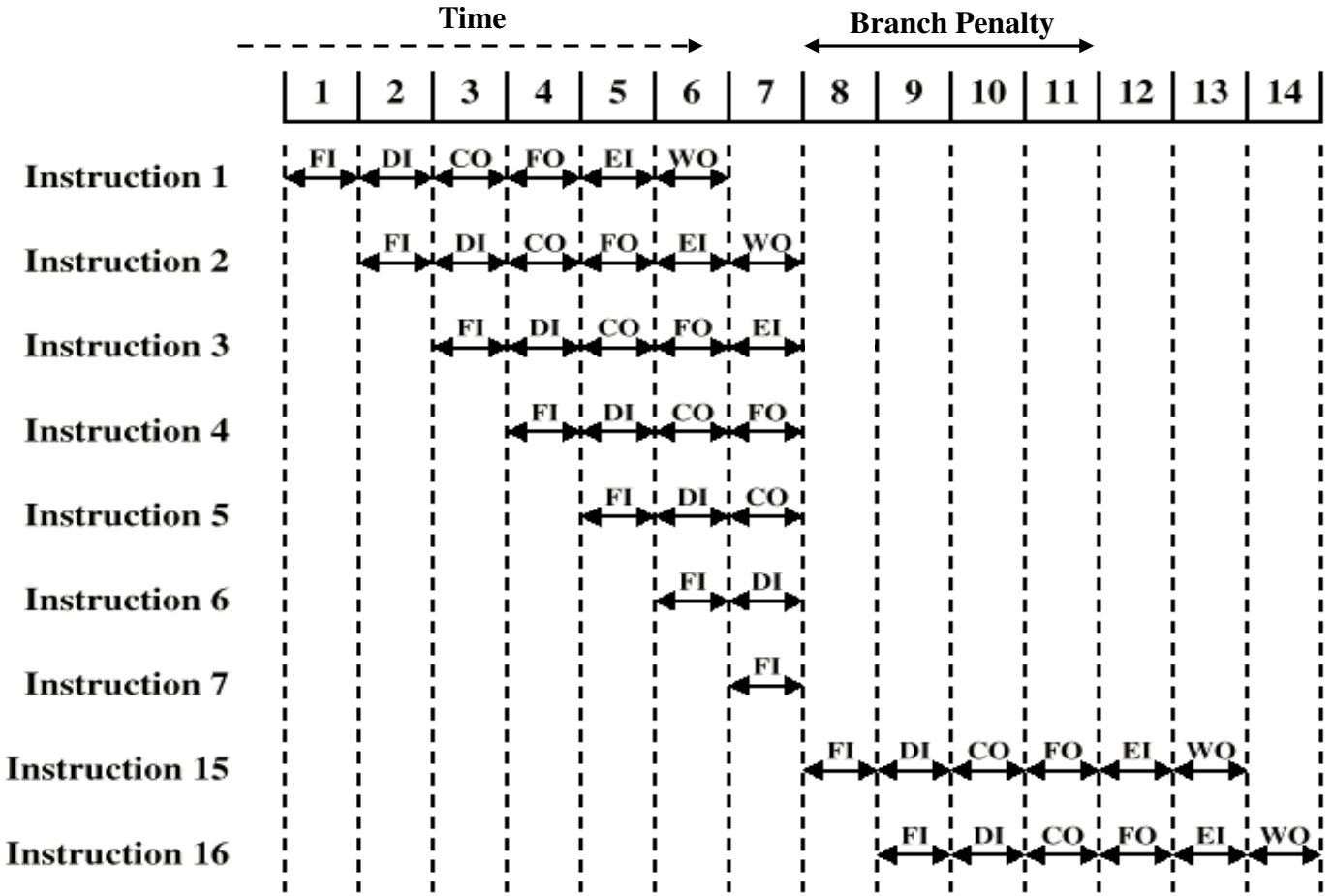
Control flow (3)

- But conditional branches still have a large penalty



Branch in a Pipeline (3)

- Here instruction 3 is a conditional branch to instruction 15



Dealing with Branches

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

Prefetch Branch Target

- Target of branch is prefetched, in addition to instruction following branch, and stored in an additional dedicated register
- Keep target until branch is executed
- Used by IBM 360/91

Multiple Streams

- Have two pipelines
- Prefetch each branch into a separate pipeline
- Use appropriate pipeline

- Leads to bus & register contention (only the sub-parts making up the pipeline are doubled)
- Additional branches entering the pipeline lead to further pipelines being needed

Loop Buffer

- Very fast memory internal to CPU
- Record the last *n* fetched instructions
- Maintained by fetch stage of pipeline
- Check loop buffer before fetching from memory
- Very good for small loops or close jumps
- The same concept as cache memory
- Used by CRAY-1

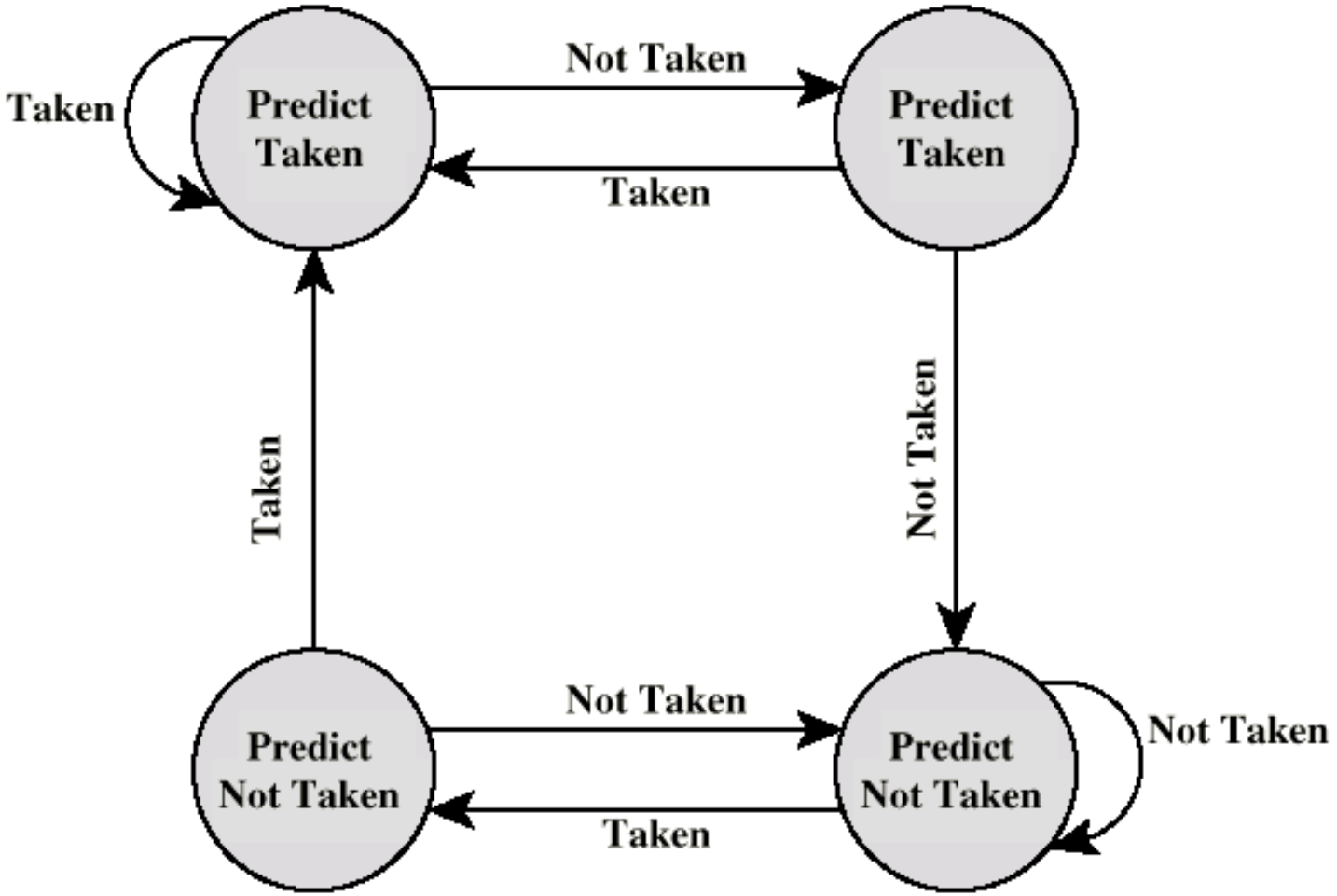
Branch Prediction (1)

- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
  - VAX will not prefetch after branch if a page fault would result (O/S v CPU design)
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction

Branch Prediction (2)

- Predict by Opcode
  - Some instructions are more likely to result in a jump than others
  - Can get up to 75% success
- Use a *Taken/Not taken* switch
  - Based on previous history of the instruction
  - Good for loops

Branch Prediction State Diagram



Delayed Branch

- Do not take jump until you have to
- Rearrange instructions

- Used for RISC (Reduced Instructions Set Computer) architectures