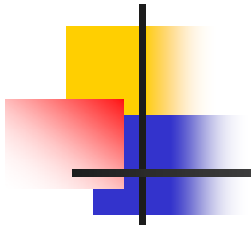
Computer Organization & Assembly Languages

Computer Organization (I)

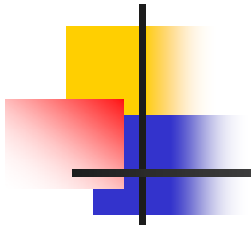
Fundamentals

Pu-Jen Cheng



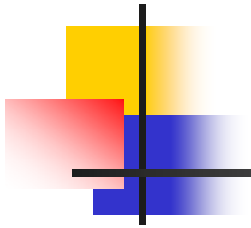
Materials

- Some materials used in this course are adapted from
  - The slides prepared by Kip Irvine for the book, Assembly Language for Intel-Based Computers, 5th Ed.
  - The slides prepared by S. Dandamudi for the book, Fundamentals of Computer Organization and Designs.
  - The slides prepared by S. Dandamudi for the book, Introduction to Assembly Language Programming, 2nd Ed.
  - Introduction to Computer Systems, CMU
    (http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f05/www/)
  - Assembly Language & Computer Organization, NTU
    (http://www.csie.ntu.edu.tw/~cyy/courses/assembly/05fall/news/)
    (http://www.csie.ntu.edu.tw/~acpang/course/asm_2004)



Outline

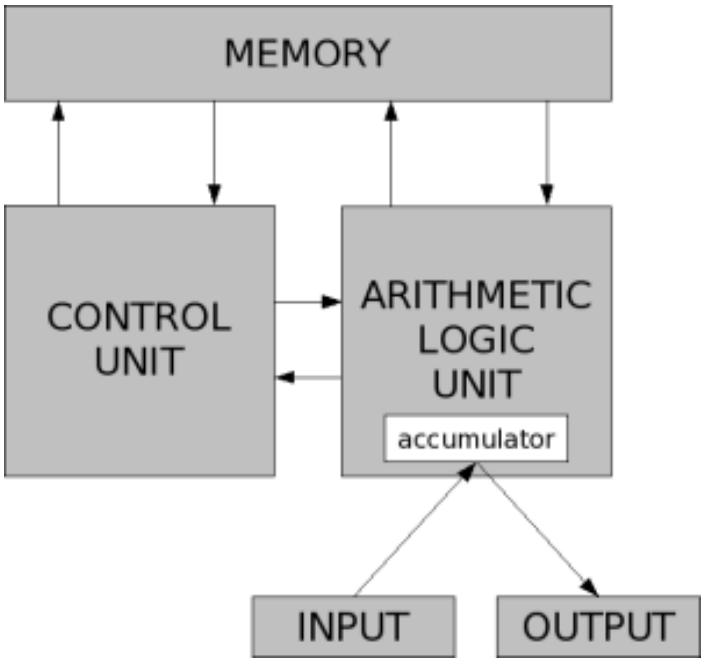
- General Concepts of Computer Organization
  - Overview of Microcomputer
    - CPU, Memory, I/O
    - Instruction Execution Cycle
  - Central Processing Unit (CPU)
    - CISC vs. RISC
    - 6 Instruction Set Design Issues
  - How Hardwares Execute Processor's Instructions
    - Digital Logic Design (Combinational & Sequential Circuits)
    - Microprogrammed Control
  - Pipelining
    - 3 Hazards
    - 3 technologies for performance improvement
  - Memory
    - Data Alignment
    - 2 Design Issues (Cache, Virtual Memory)
  - I/O Devices



Overview of Microcomputer

Von Neumann Machine, 1945

- Memory, Input/Output, Arithmetic/Logic Unit, Control Unit
- Stored-program Model
  - Both data and programs are stored in the *same* main memory
- Sequential Execution

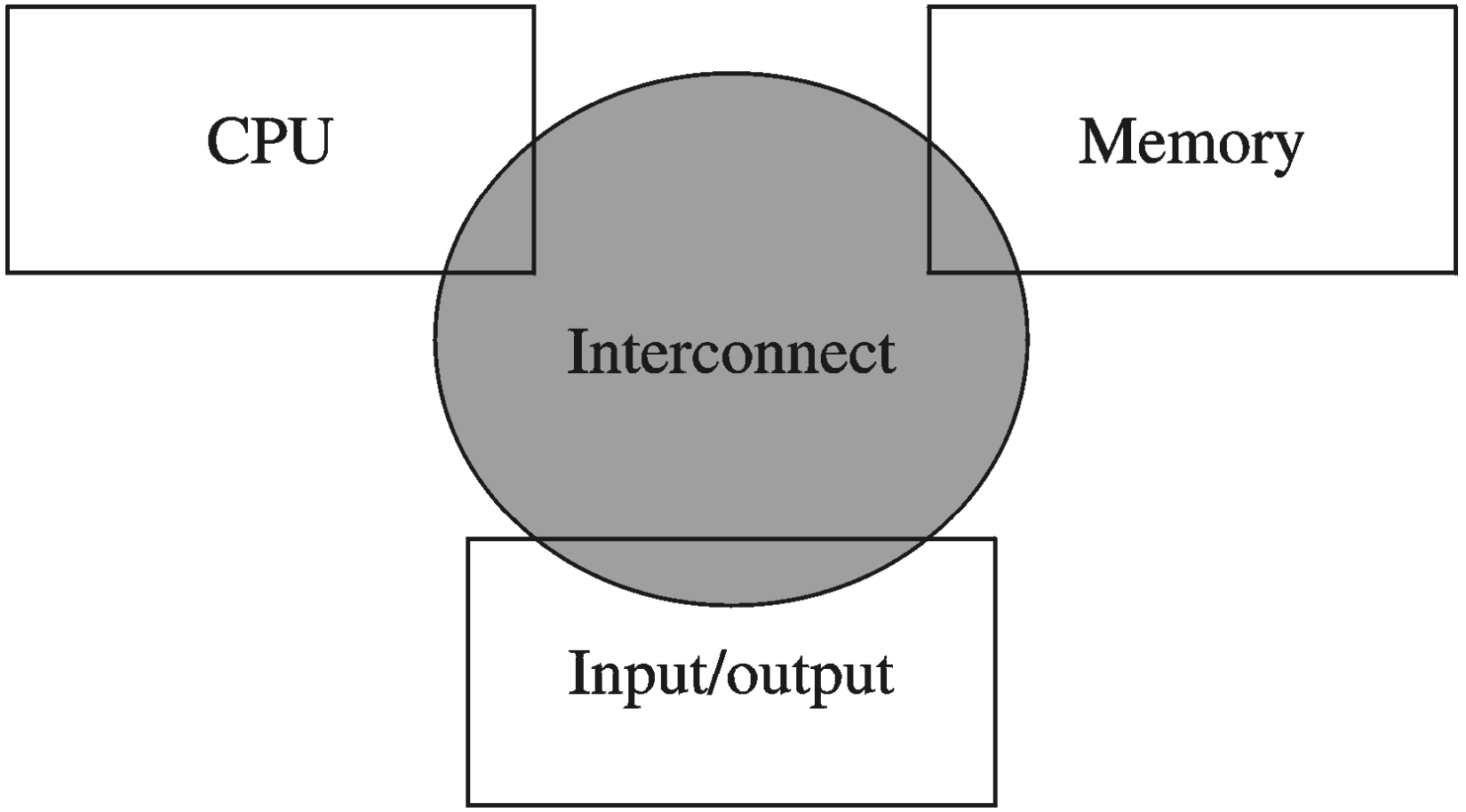


http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf

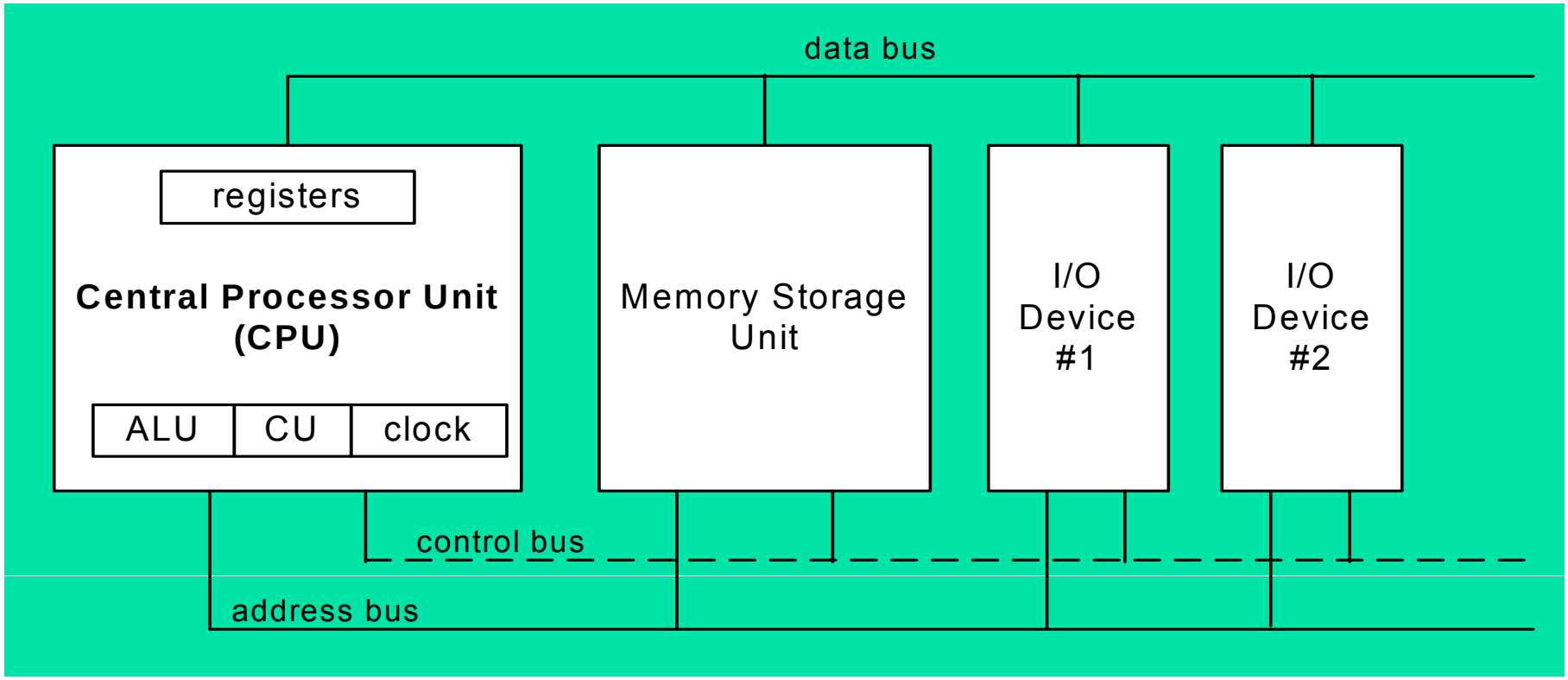
What is Microcomputer

- Microcomputer
  - A computer with a microprocessor (μP) as its central processing unit (CPU)
- Microprocessor (μP)
  - A digital electronic component with transistors on a single semiconductor integrated circuit (IC)
  - One or more microprocessors typically serve as a central processing unit (CPU) in a computer system or handheld device.

Components of Microcomputer

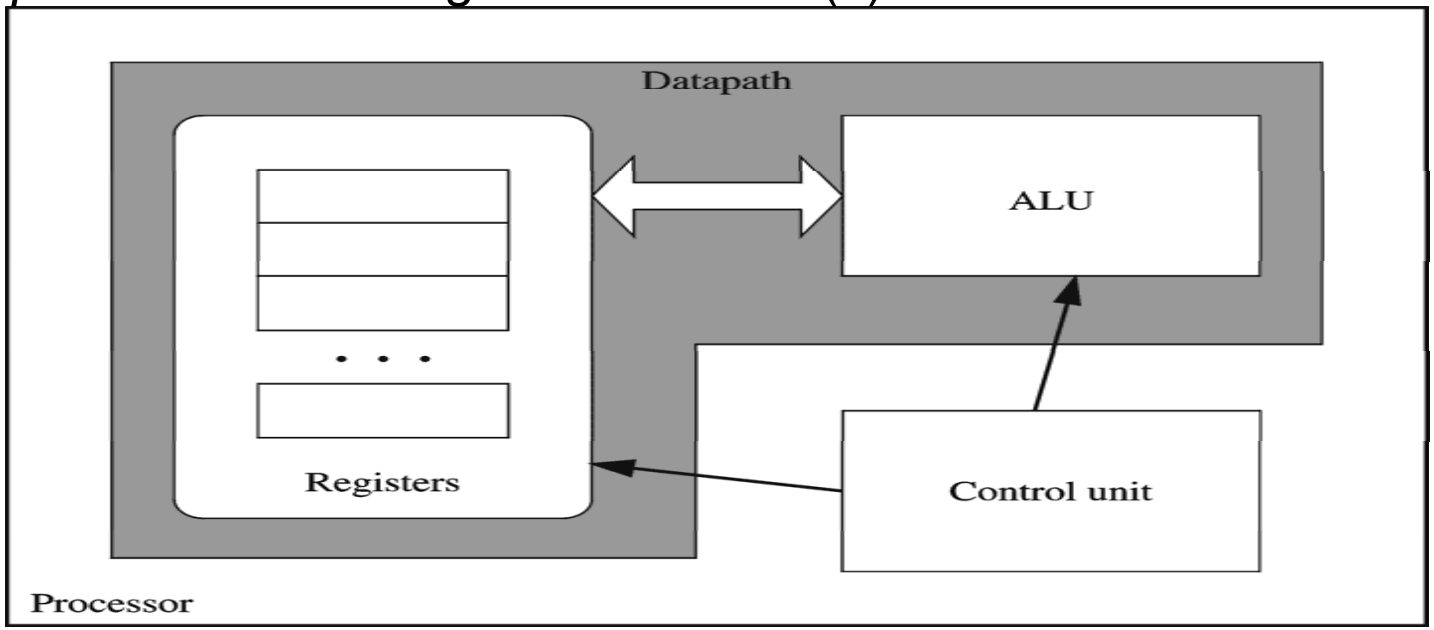


Basic Microcomputer Design



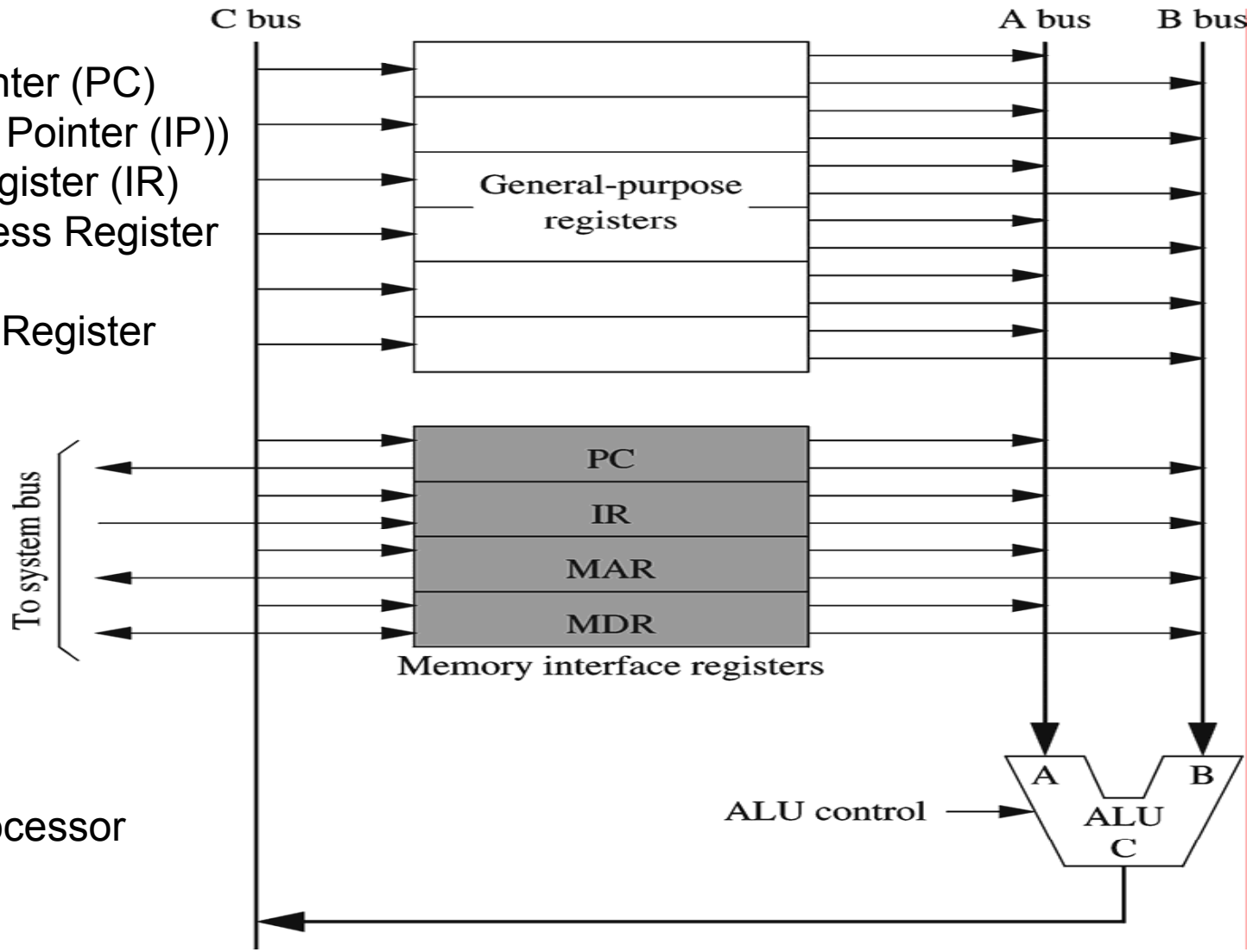
CPU

- *Arithmetic and logic unit* (ALU) performs arithmetic (add, subtract) and logical (AND, OR, NOT) operations
- *Registers* store data and instructions used by the processor
- *Control unit* (CU) coordinates sequence of execution steps
  - Fetch instructions from memory, decode them to find their types
- *Clock*
- *Datapath* consists of registers and ALU(s)



Datapath

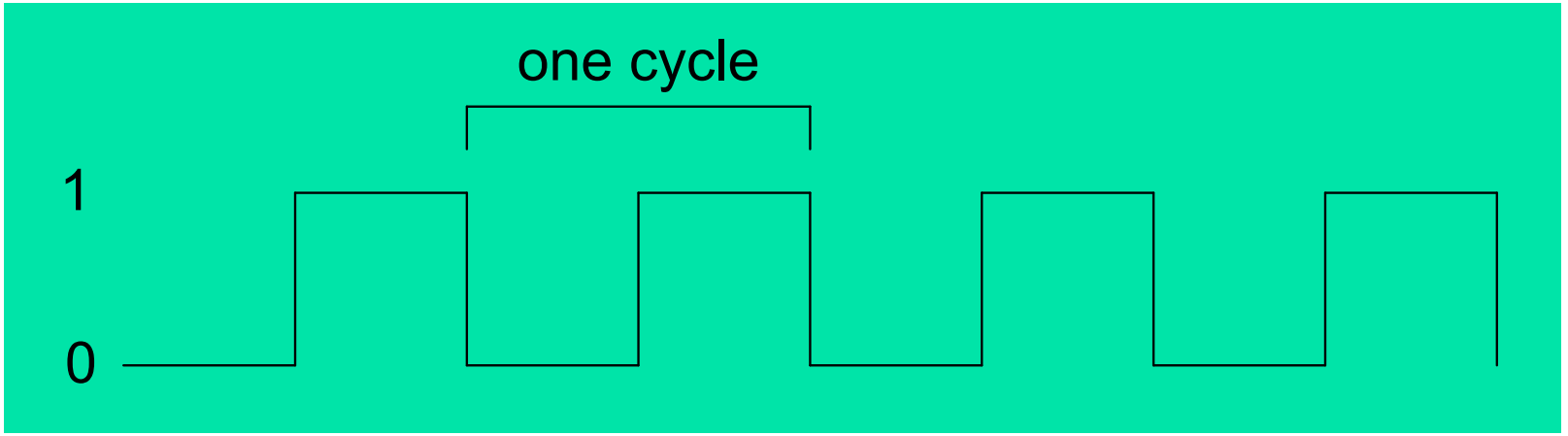
Program Counter (PC)
(or Instruction Pointer (IP))
Instruction Register (IR)
Memory Address Register (MAR)
Memory Data Register (MDR)



RISC processor

Clock

- Provide timing signal and the basic unit of time
- Synchronize all CPU and BUS operations
- Machine (clock) cycle measures time of a single operation
- Clock is used to trigger events
- Clock period = $\frac{1}{\text{Clock frequency}}$ 1GHz→clock cycle=1ns
- A instruction could take multiple cycles to complete, e.g. multiply in 8088 takes 50 cycles



Memory, I/O, System Bus

- Main/primary memory (random access memory, RAM) stores both program instructions and data
- I/O devices
  - Interface: *I/O controller*
  - User interface: keyboard, display screen, printer, modem, …
  - Secondary storage: disk
  - Communication network
- System Bus
  - A bunch of parallel wires
  - Transfer data among the components
  - *Address bus* (determine the amount of physical memory addressable)
  - *Data bus* (indicate the size of the data transferred)
  - *Control bus* (consists of control signals:
    memory/IO read/write, interrupt, bus request/grand)

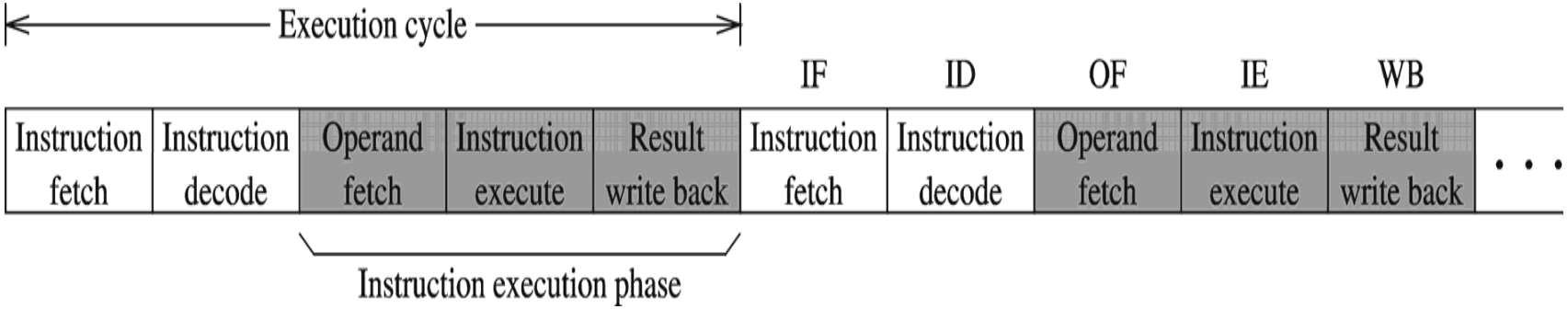
Instruction Execution Cycle

- Execution Cycle
  - Fetch (IF): CU fetches next instruction, advance PC/IP
  - Decode (ID): CU determines what the instruction will do
  - Execute

    Fetch operands (OF): (memory operand needed) read value from memory

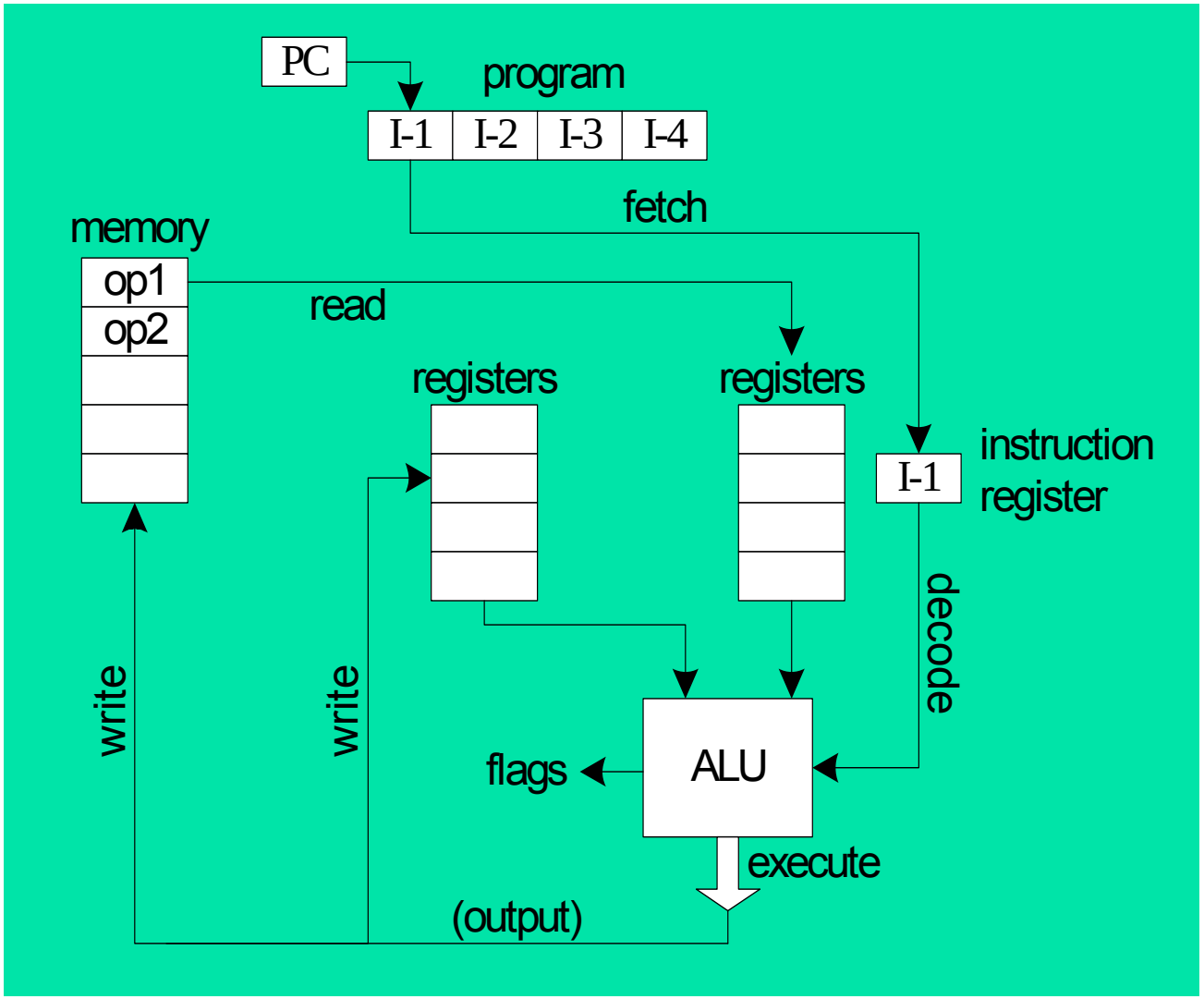
    Execute the instruction (IE)

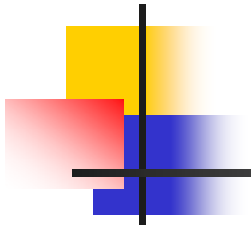
    Store output operand (WB): (memory operand needed) write result to memory



Instruction Execution Cycle (cont.)

- Fetch
- Decode
- Fetch operands
- Execute
- Store output

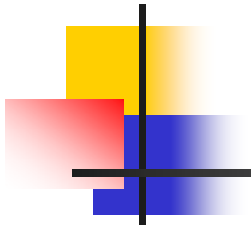




Introduction to Digital Logic Design

- See asm_ch2_dl.ppt

CPU



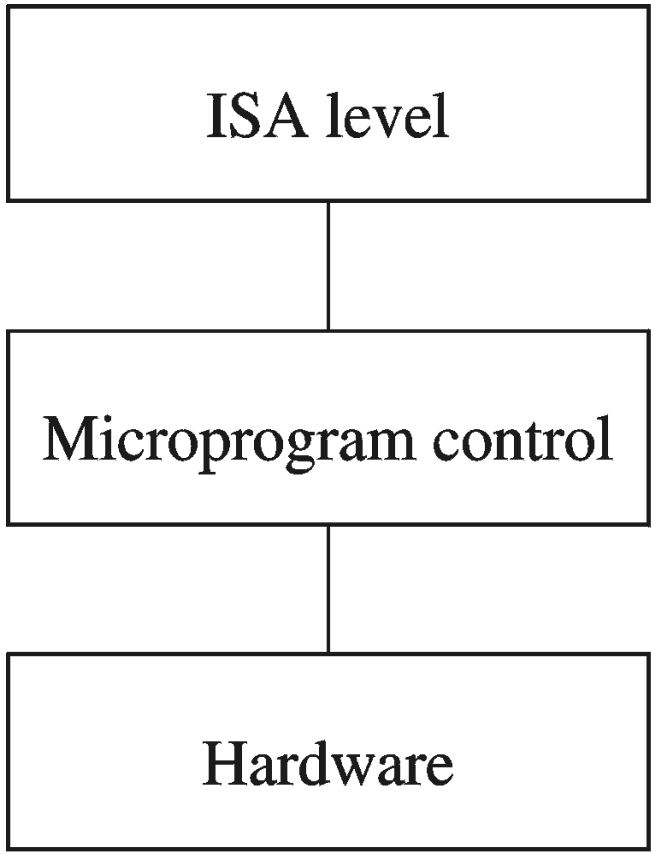
CPU

- CISC vs. RISC
- 6 Instruction Set Design Issues
  - Number of Addresses
  - Flow of Control
  - Operand Types
  - Addressing Modes
  - Instruction Types
  - Instruction Formats

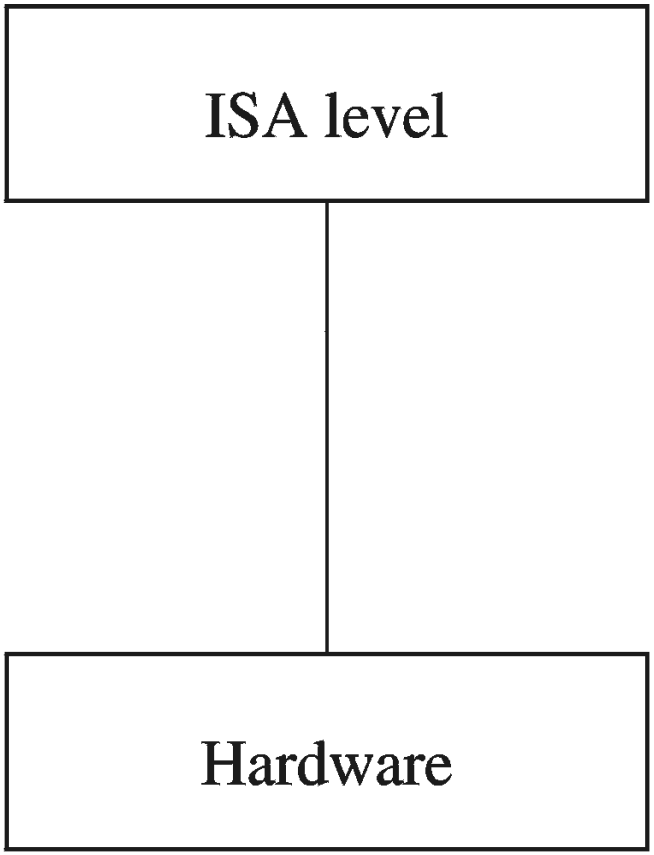
Processor

- RISC and CISC designs
  - Reduced Instruction Set Computer (RISC)
    - Simple instructions, small instruction set
    - Operands are assumed to be in processor registers
      - Not in memory
      - Simplify design (e.g., fixed instruction size)
    - Examples: ARM (Advanced RISC Machines), DEC Alpha (now Compaq)
  - Complex Instruction Set Computer (CISC)
    - Complex instructions, large instruction set
    - Operands can be in registers or memory
      - Instruction size varies
    - Typically use a microprogram
    - Example: Intel 80x86 family

Processor (cont.)



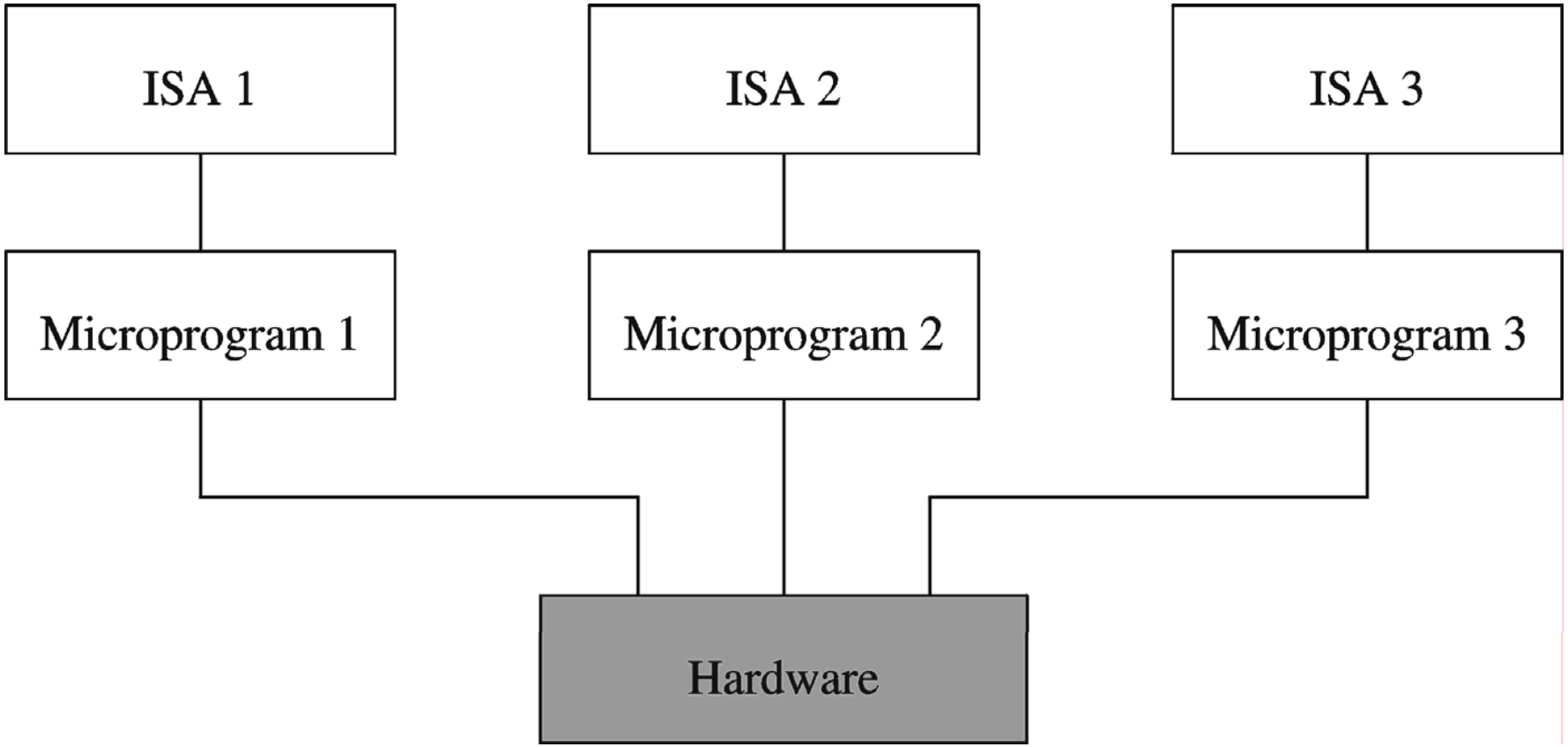
(a) CISC implementation

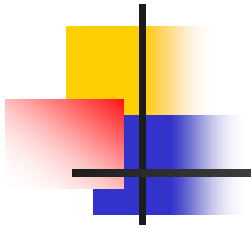


(b) RISC implementation

Processor (cont.)

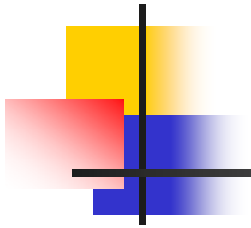
- Variations of the ISA-level can be implemented by changing the microprogram





Instruction Set Design Issues

- Number of Addresses
- Flow of Control
- Operand Types
- Addressing Modes
- Instruction Types
- Instruction Formats



Number of Addresses

- Four categories
  - 3-address machines
    - 2 for the source operands and one for the result
  - 2-address machines
    - One address doubles as source and result
  - 1-address machine
    - Accumulator machines
    - Accumulator is used for one source and result
  - 0-address machines
    - Stack machines
    - Operands are taken from the stack
    - Result goes onto the stack

Number of Addresses (cont.)

- Three-address machines
  - Two for the source operands, one for the result
  - RISC processors use three addresses
  - Sample instructions

```
add     dest,src1,src2
        ; M(dest)=[src1]+[src2]
sub     dest,src1,src2
        ; M(dest)=[src1]-[src2]
mult    dest,src1,src2
        ; M(dest)=[src1]*[src2]
```

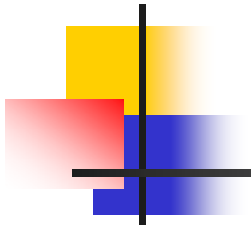
Number of Addresses (cont.)

- Example
  - C statement

    A = B + C * D – E + F + A
  - Equivalent code:

```
mult    T,C,D   ;T = C*D
add     T,T,B   ;T = B+C*D
sub     T,T,E   ;T = B+C*D-E
add     T,T,F   ;T = B+C*D-E+F
add     A,T,A   ;A = B+C*D-E+F+A
```

Number of Addresses (cont.)



- Two-address machines
  - One address doubles (for source operand & result)
  - Last example makes a case for it
    - Address T is used twice
  - Sample instructions

```
load    dest,src ; M(dest)=[src]
add     dest,src ; M(dest)=[dest]+[src]
sub     dest,src ; M(dest)=[dest]-[src]
mult    dest,src ; M(dest)=[dest]*[src]
```

Number of Addresses (cont.)

- Example
  - C statement

    A = B + C * D – E + F + A
  - Equivalent code:

```
load    T,C   ;T = C
mult    T,D   ;T = C*D
add     T,B   ;T = B+C*D
sub     T,E   ;T = B+C*D-E
add     T,F   ;T = B+C*D-E+F
add     A,T   ;A = B+C*D-E+F+A
```

Number of Addresses (cont.)

- One-address machines
  - Use special set of registers called accumulators
    - Specify one source operand & receive the result
  - Called accumulator machines
  - Sample instructions

```
load    addr ; accum = [addr]
store   addr ; M[addr] = accum
add     addr ; accum = accum + [addr]
sub     addr ; accum = accum - [addr]
mult    addr ; accum = accum * [addr]
```

Number of Addresses (cont.)

- Example
  - C statement

    A = B + C * D – E + F + A

  - Equivalent code:

```
load    C   ;load C into accum
mult    D   ;accum = C*D
add     B   ;accum = C*D+B
sub     E   ;accum = B+C*D-E
add     F   ;accum = B+C*D-E+F
add     A   ;accum = B+C*D-E+F+A
store   A   ;store accum contents in A
```

Number of Addresses (cont.)

- Zero-address machines
  - Stack supplies operands and receives the result
    - Special instructions to load and store use an address
  - Called stack machines (Ex: HP3000, Burroughs B5500)
  - Sample instructions

```
push    addr ; push([addr])
pop     addr ; pop([addr])
add          ; push(pop + pop)
sub          ; push(pop - pop)
mult         ; push(pop * pop)
```

Number of Addresses (cont.)

- Example
  - C statement

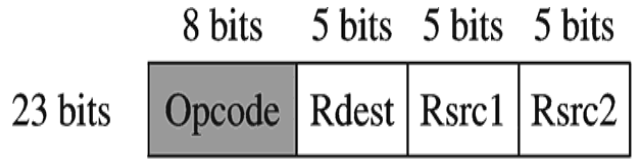
    A = B + C * D – E + F + A

  - Equivalent code:

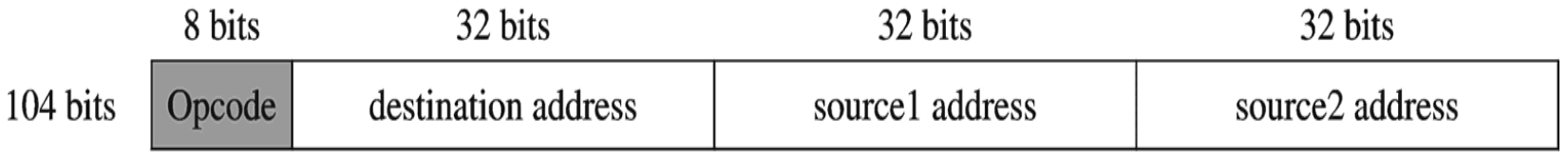
```
push    E          |    sub
push    C          |    push    F
push    D          |    add
Mult               |    push    A
push    B          |    add
add                |    pop     A
```

Load/Store Architecture

- Instructions expect operands in internal processor registers
  - Special LOAD and STORE instructions move data between registers and memory
  - RISC uses this architecture
  - Reduces instruction length



Register format



Memory format

Load/Store Architecture (cont.)

- Sample instructions

```
load    Rd,addr     ;Rd = [addr]
store   addr,Rs     ;(addr) = Rs
add     Rd,Rs1,Rs2  ;Rd = Rs1 + Rs2
sub     Rd,Rs1,Rs2  ;Rd = Rs1 - Rs2
mult    Rd,Rs1,Rs2  ;Rd = Rs1 * Rs2
```

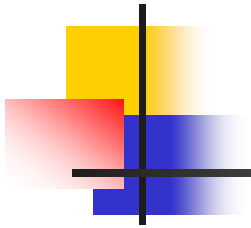
Number of Addresses (cont.)

- Example
  - C statement

```
A = B + C * D – E + F + A
```

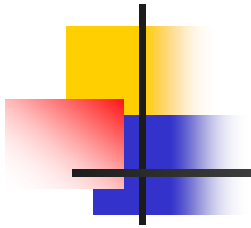
  - Equivalent code:

```
load   R1,B  |  mult   R2,R2,R3
load   R2,C  |  add    R2,R2,R1
load   R3,D  |  sub    R2,R2,R4
load   R4,E  |  add    R2,R2,R5
load   R5,F  |  add    R2,R2,R6
load   R6,A  |  store  A,R2
```



Flow of Control

- Default is sequential flow
- Several instructions alter this default execution
  - Branches
    - Unconditional
    - Conditional
    - Delayed branches
  - Procedure calls
    - Delayed procedure calls



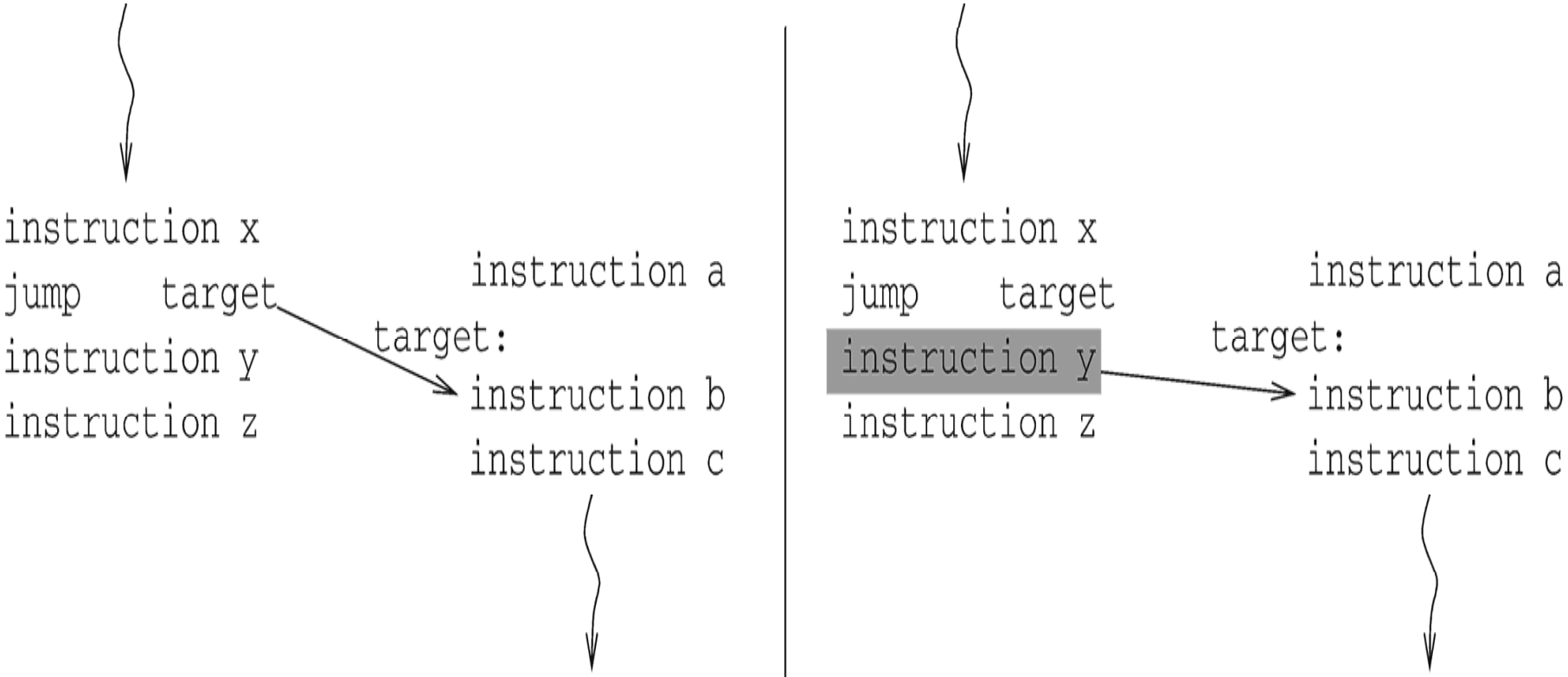
Flow of Control (cont.)

- Branches
  - Unconditional
    - Absolute address
    - PC-relative
      - Target address is specified relative to PC contents
      - Relocatable code
  - Example: MIPS
    - Absolute address

      j target
    - PC-relative

      b target

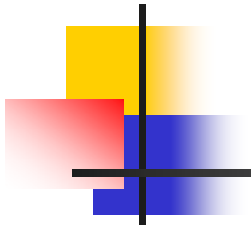
Flow of Control (cont.)



(a) Normal branch execution
e.g., Pentium

(b) Delayed branch execution
e.g., SPARC

Flow of Control (cont.)



- Branches
  - Conditional
    - Jump is taken only if the condition is met
  - Two types
    - Set-Then-Jump
      - Condition testing is separated from branching
      - Condition code registers are used to convey the condition test result
      - Condition code registers keep a record of the status of the last ALU operation such as overflow condition
    - Example: Pentium code

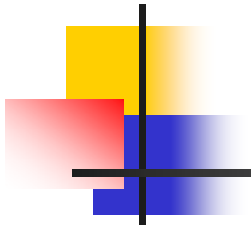
```
cmp     AX,BX       ; compare AX and BX
je      target      ; jump if equal
```

Flow of Control (cont.)

- Test-and-Jump
  - Single instruction performs condition testing and branching
  - Example: MIPS instruction

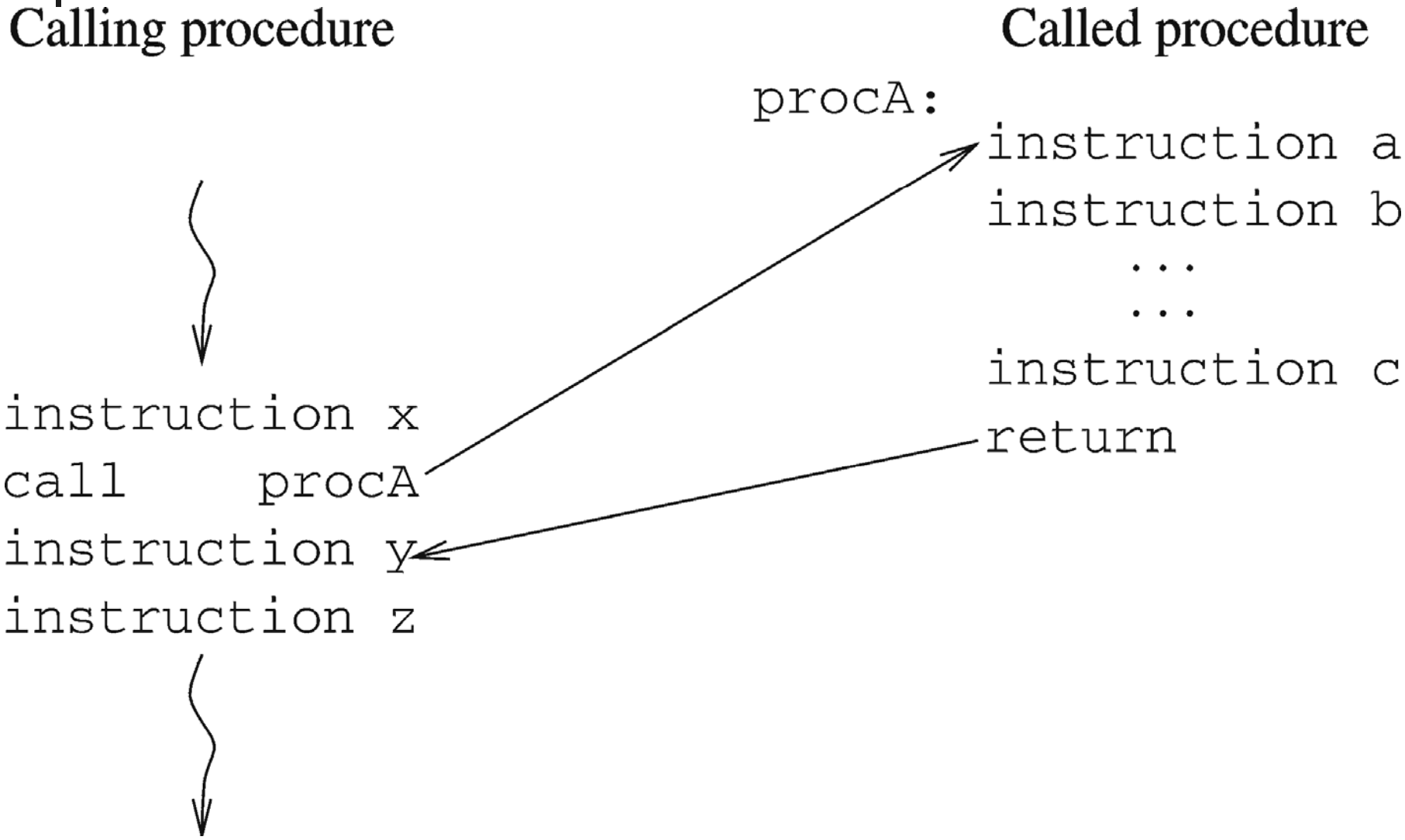
    `beq  Rsrc1,Rsrc2,target`
    - Jumps to target if Rsrc1 = Rsrc2
- Delayed branching
  - Control is transferred after executing the instruction that follows the branch instruction
    - This instruction slot is called delay slot
  - Improves efficiency
  - Highly pipelined RISC processors support

Flow of Control (cont.)

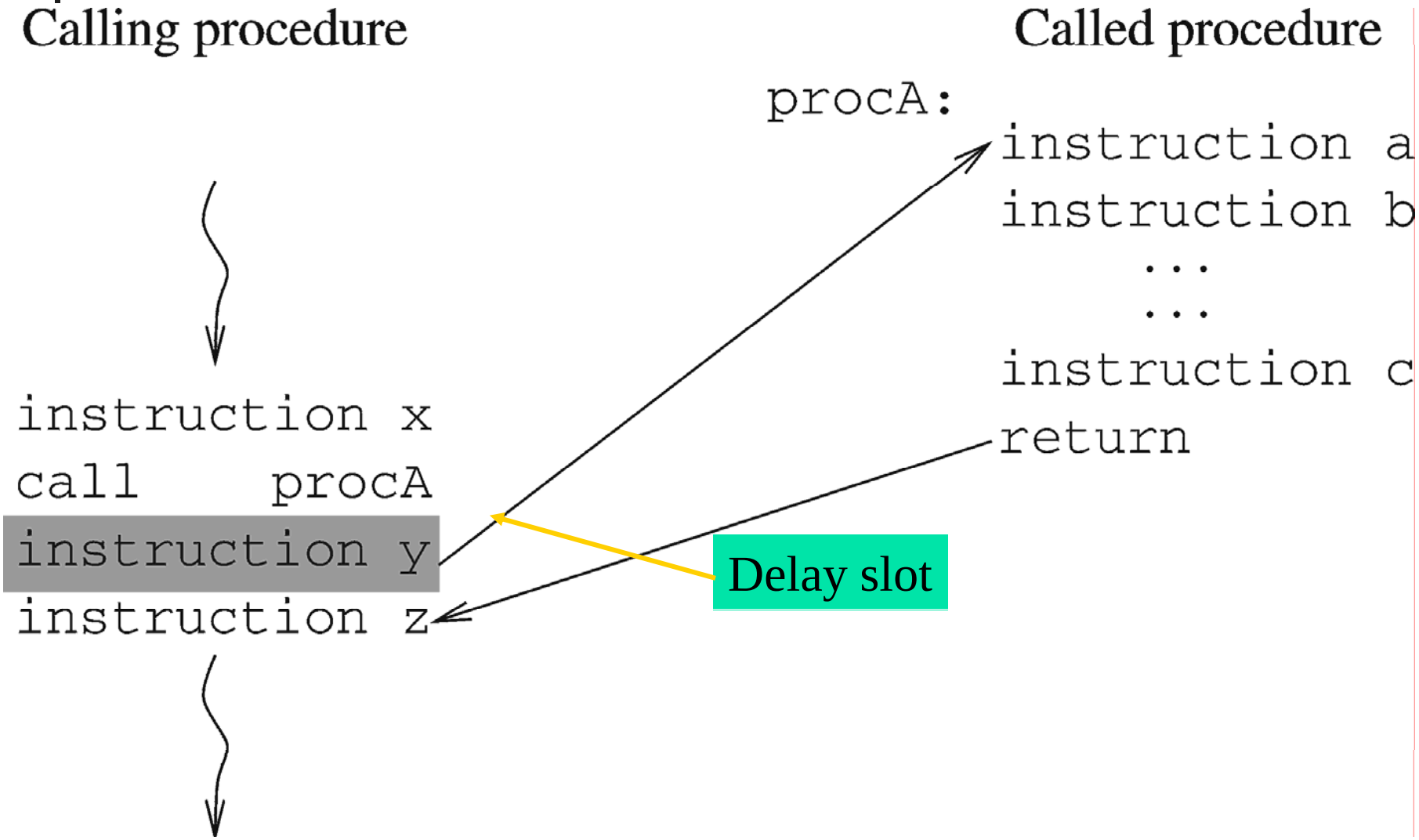


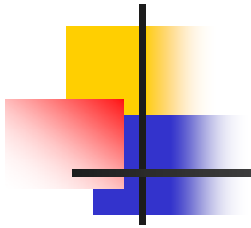
- Procedure calls
  - Facilitate modular programming
  - Require two pieces of information to return
    - End of procedure
      - Pentium
        - uses **ret** instruction
      - MIPS
        - uses **jr** instruction
    - Return address
      - In a (special) register
        - MIPS allows any general-purpose register
      - On the stack
        - Pentium

Flow of Control (cont.)



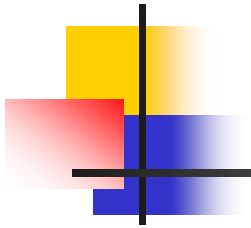
Flow of Control (cont.)





Parameter Passing

- Two basic techniques
  - Register-based (e.g., PowerPC, MIPS)
    - Internal registers are used
      - Faster
      - Limit the number of parameters
      - Recursive procedure
  - Stack-based (e.g., Pentium)
    - Stack is used
      - More general



Operand Types

- Instructions support basic data types
  - Characters
  - Integers
  - Floating-point
- Instruction overload
  - Same instruction for different data types
  - Example: Pentium

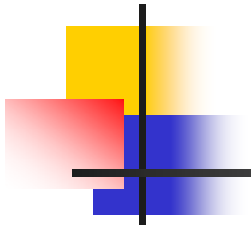
```
mov     AL,address    ;loads an 8-bit value
mov     AX,address    ;loads a 16-bit value
mov     EAX,address   ;loads a 32-bit value
```

Operand Types

- Separate instructions
  - Instructions specify the operand size
  - Example: MIPS

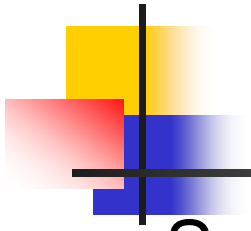
```
lb    Rdest,address    ;loads a byte
lh    Rdest,address    ;loads a halfword
                       ;(16 bits)
lw    Rdest,address    ;loads a word
                       ;(32 bits)
ld    Rdest,address    ;loads a doubleword
                       ;(64 bits)
```

Similar instruction: store



Addressing Modes

- How the operands are specified
  - Operands can be in three places
    - Registers
      - Register addressing mode
    - Part of instruction
      - Constant
      - Immediate addressing mode
      - All processors support these two addressing modes
    - Memory
      - Difference between RISC and CISC
      - CISC supports a large variety of addressing modes
      - RISC follows load/store architecture



Instruction Types

- Several types of instructions
  - Data movement
    - Pentium: **`mov dest,src`**
    - Some do not provide direct data movement instructions
    - Indirect data movement

      `add Rdest,Rsrc,0 ;Rdest = Rsrc+0`
  - Arithmetic and Logical
    - Arithmetic
      - Integer and floating-point, signed and unsigned
      - add, subtract, multiply, divide
    - Logical
      - **and, or, not, xor**

Instruction Types (cont.)

- Condition code bits
  - **S**: Sign bit (0 = +, 1= -)
  - **Z**: Zero bit (0 = nonzero, 1 = zero)
  - **O**: Overflow bit (0 = no overflow, 1 = overflow)
  - **C**: Carry bit (0 = no carry, 1 = carry)

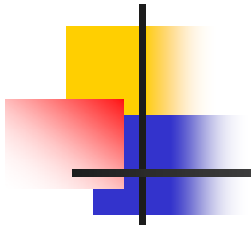
- Example: Pentium

```
cmp     count,25   ;compare count to 25
                   ;subtract 25 from count
je      target     ;jump if equal
```

Instruction Types (cont.)

- Flow control and I/O instructions
  - Branch
  - Procedure call
  - Interrupts
- I/O instructions
  - Memory-mapped I/O
    - Most processors support memory-mapped I/O
    - No separate instructions for I/O
  - Isolated I/O
    - Pentium supports isolated I/O
    - Separate I/O instructions

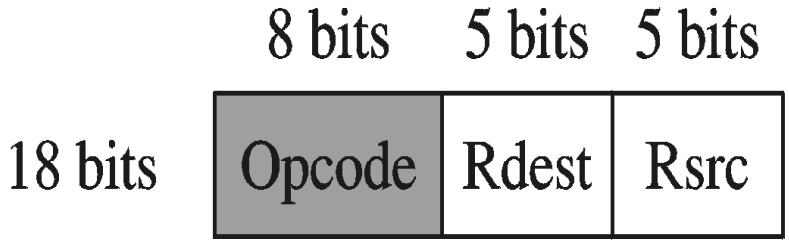
```
in    AX,io_port  ;read from an I/O port
out   io_port,AX  ;write to an I/O port
```



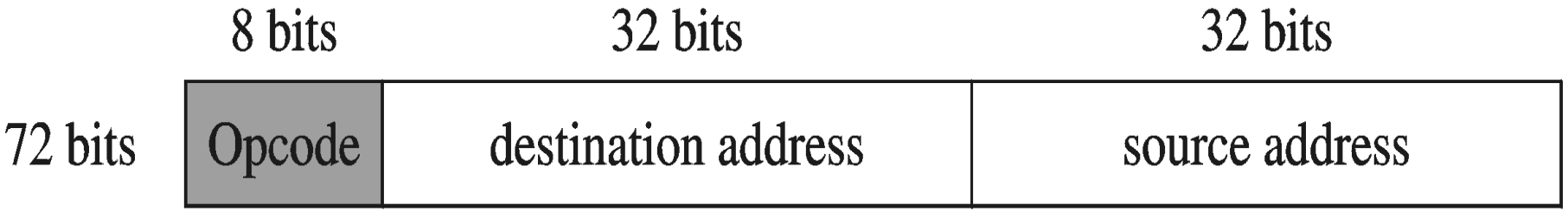
Instruction Formats

- Two types
  - Fixed-length
    - Used by RISC processors
    - 32-bit RISC processors use 32-bits wide instructions
      - Examples: SPARC, MIPS, PowerPC
  - Variable-length
    - Used by CISC processors
    - Memory operands need more bits to specify
- Opcode
  - Major and exact operation

Examples of Instruction Formats

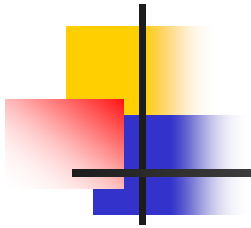


Register format



Memory format

How Hardware Executes Processor's Instructions

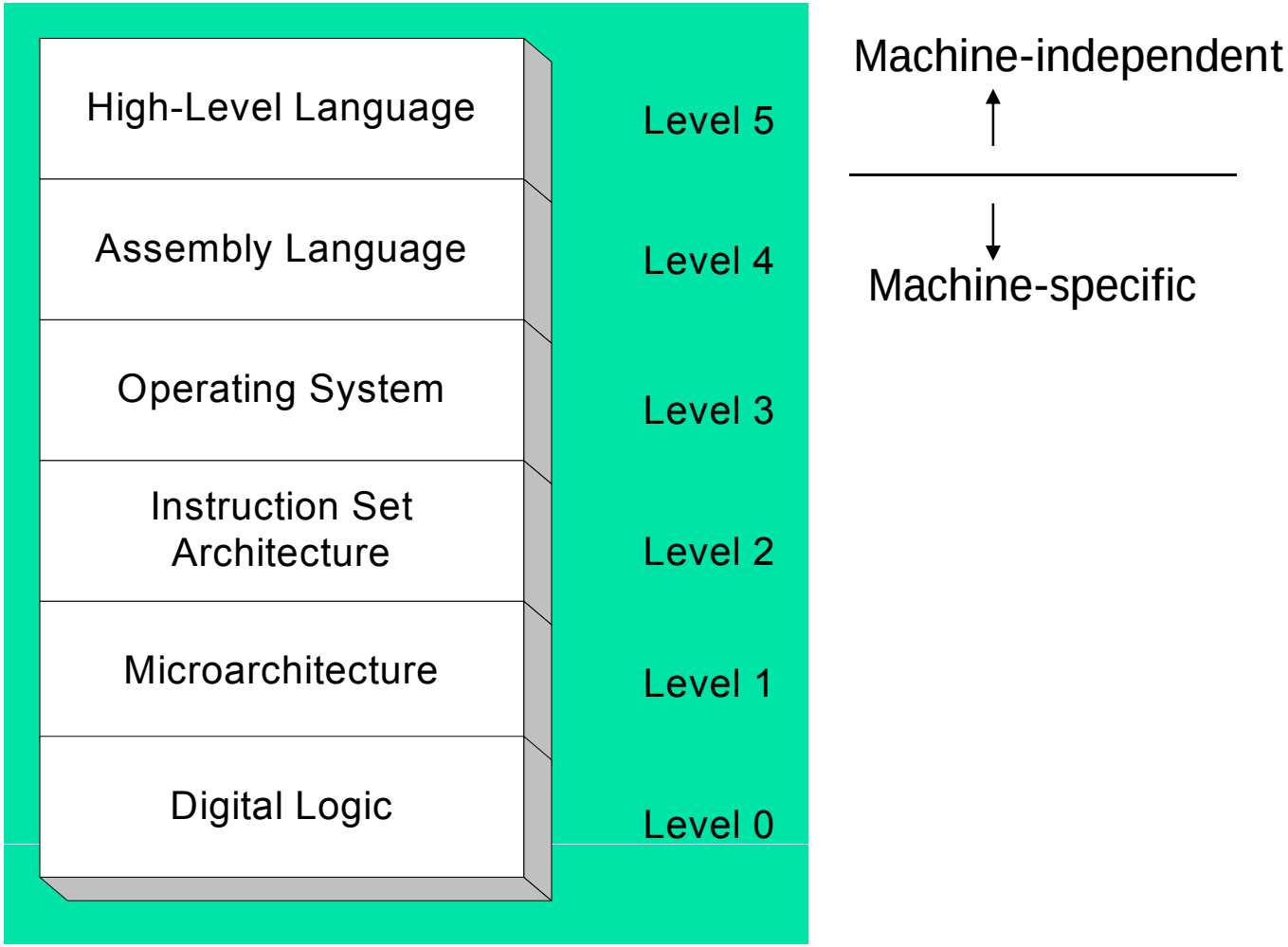


How Hardware Executes Processor's Instructions

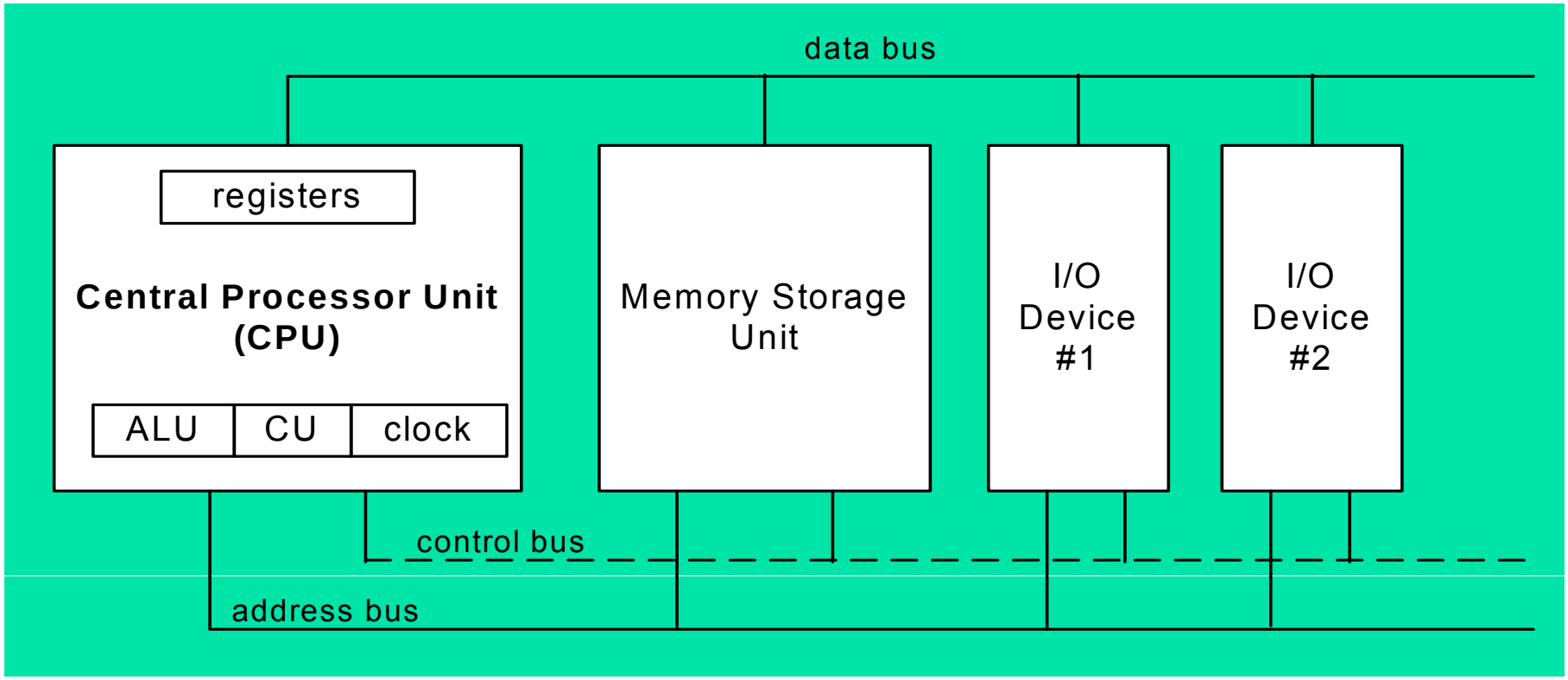
- Digital Logic Design
  - Combinational and Sequential Circuits
- Microprogrammed Control

Virtual Machines

Abstractions for computers

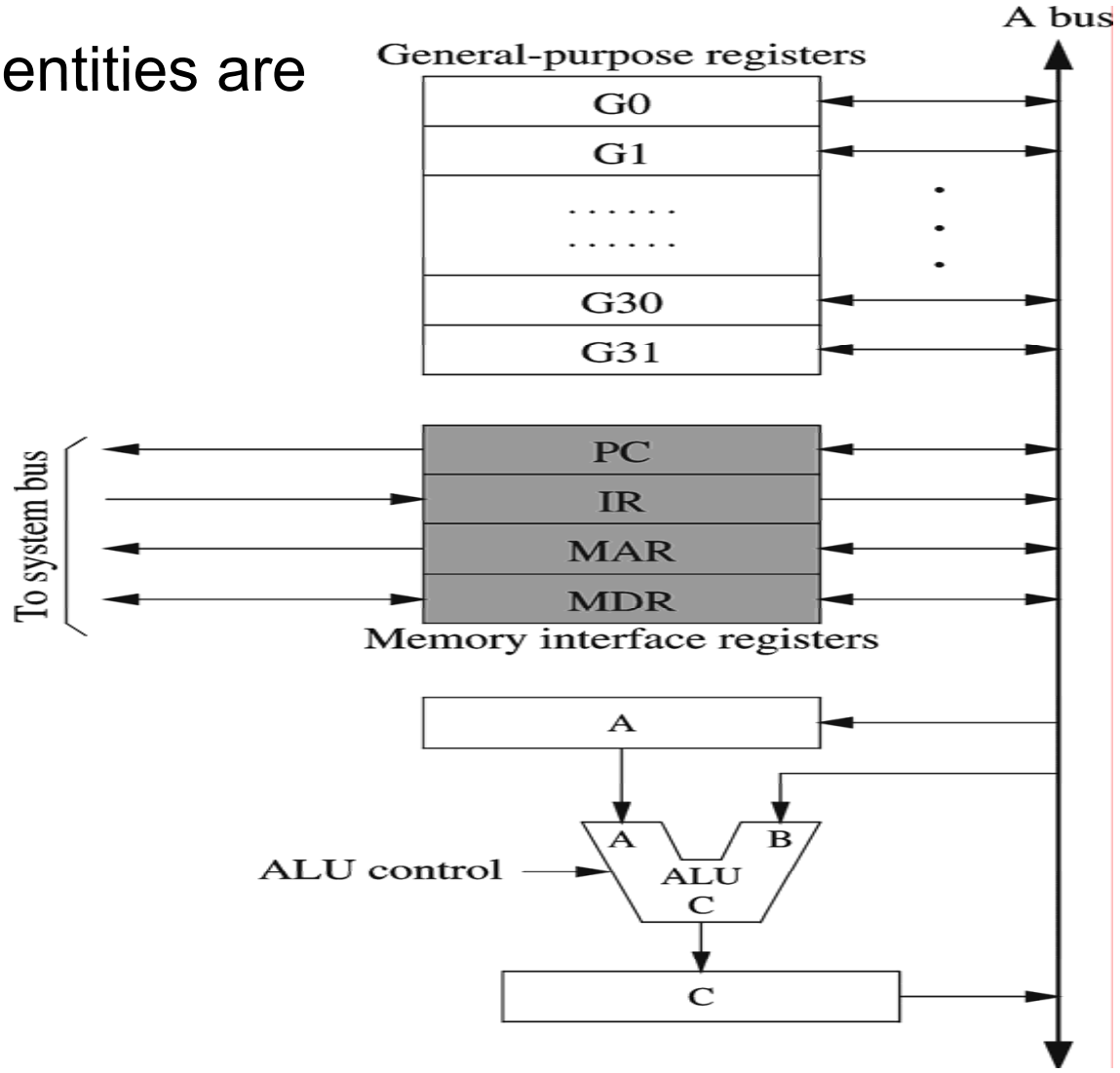


Basic Microcomputer Design

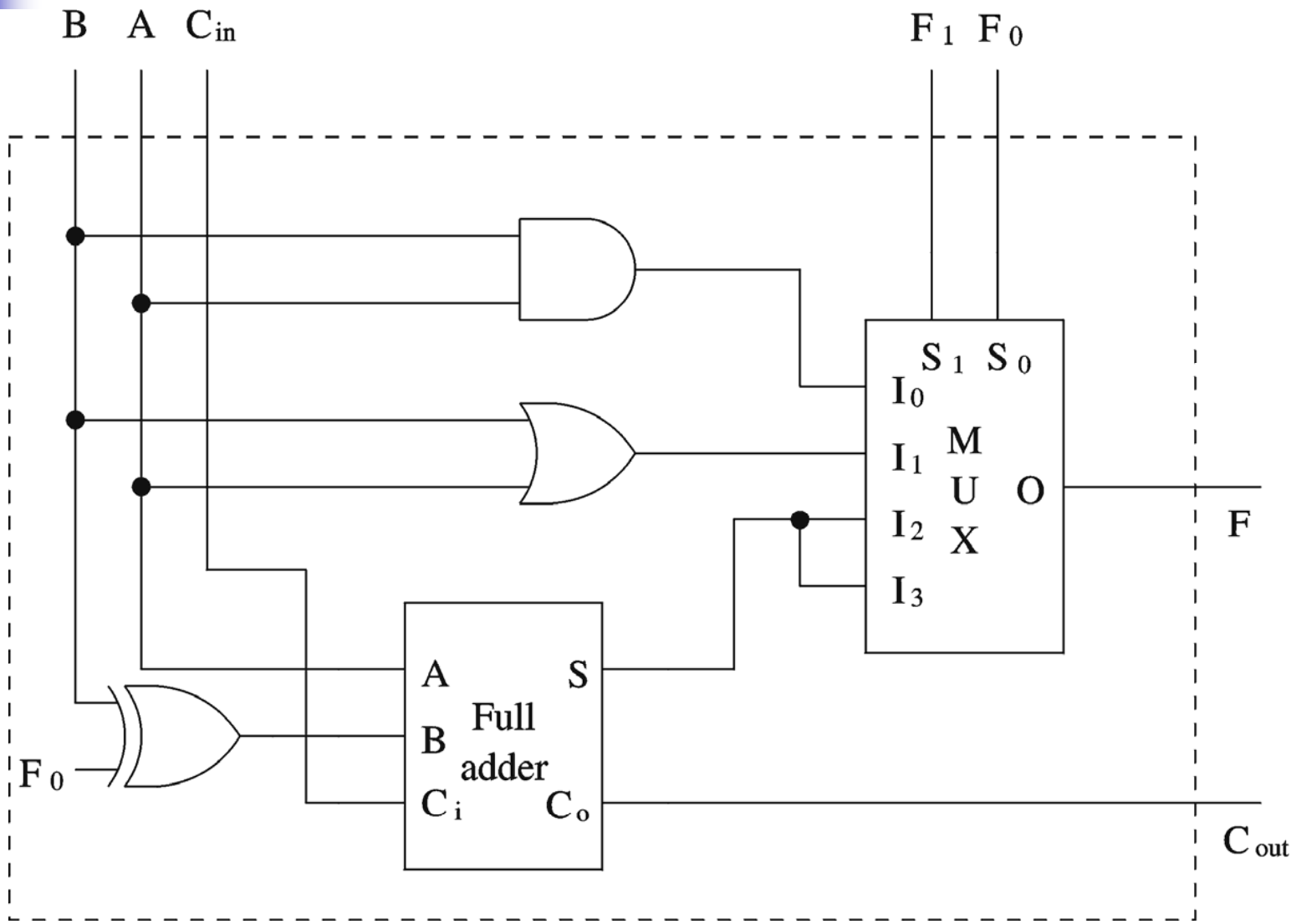


Consider 1-bus Datapath

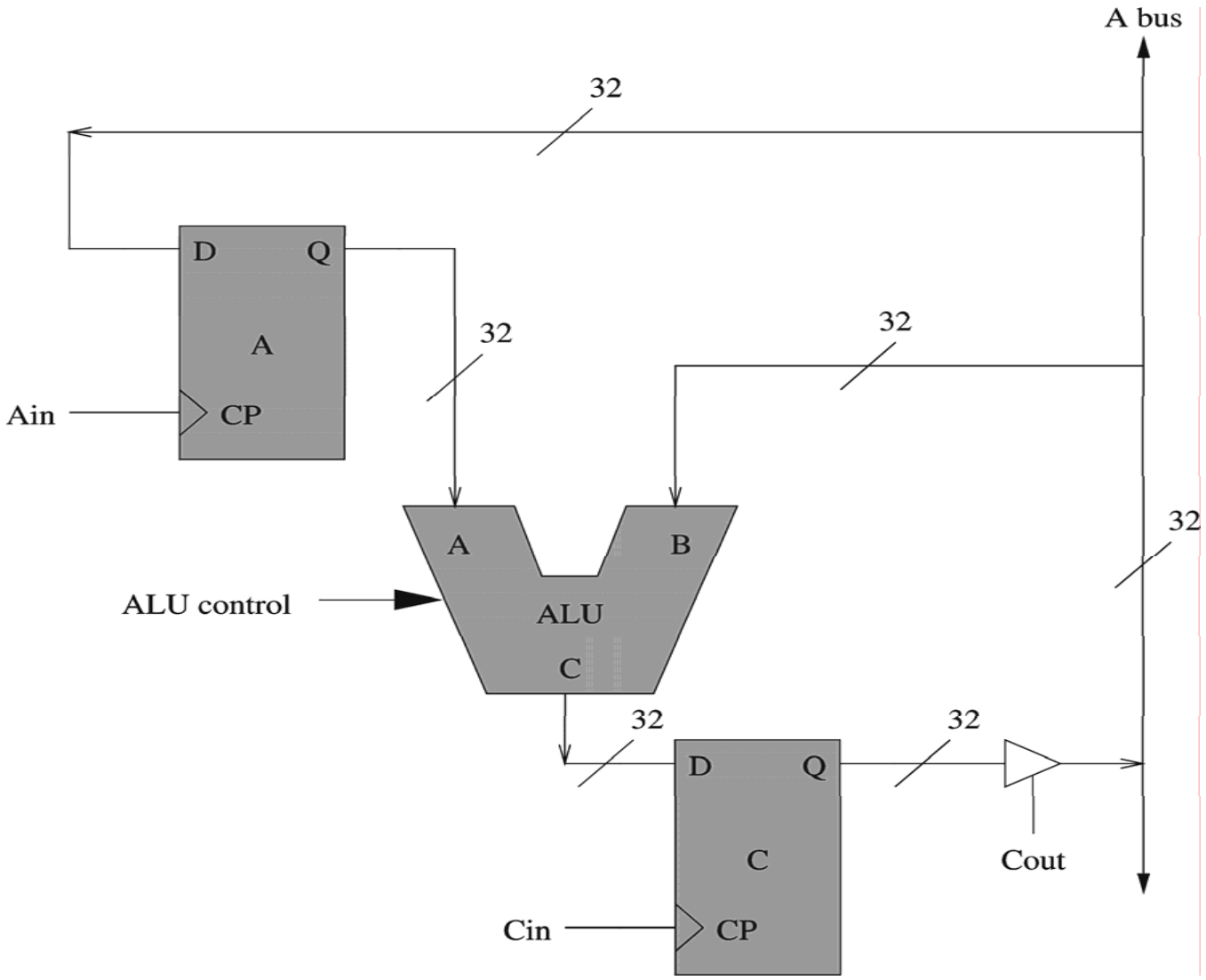
Assume all entities are 32-bit wide



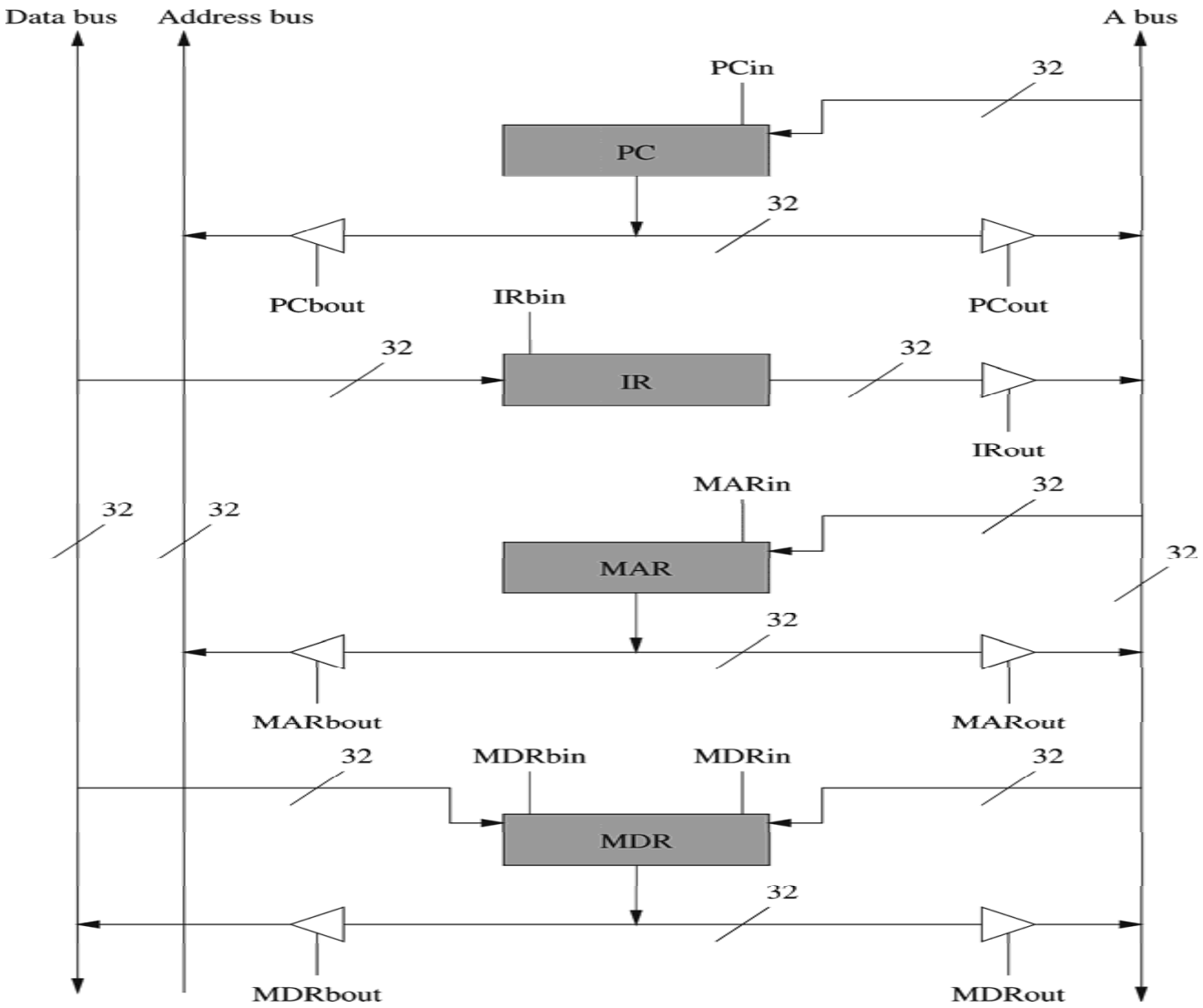
1-bit ALU



ALU Circuit in 1-bus Datapath



Memory Interface Implementation



Microprogrammed Control

- 32 32-bit general-purpose registers
  - Interface only with the A-bus
  - Each register has two control signals
    - **Gxin** and **Gxout**
- Control signals used by the other registers
  - PC register:
    - **PCin**, **PCout**, and **PCbout**
  - IR register:
    - **IRout** and **IRbin**
  - MAR register:
    - **MARin**, **MARout**, and **MARbout**
  - MDR register:
    - **MDRin**, **MDRout**, **MDRbin** and **MDRbout**

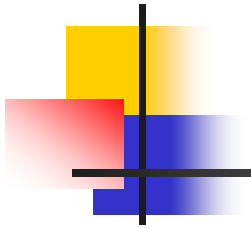
Microprogrammed Control (cont.)

```
add     %G9,%G5,%G7
```

Implemented as

- Transfer G5 contents to A register
  - Assert **G5out** and **Ain**
- Place G7 contents on the A bus
  - Assert **G7out**
- Instruct ALU to perform addition
  - Appropriate ALU function control signals
- Latch the result in the C register
  - Assert **Cin**
- Transfer contents of the C register to G9
  - Assert **Cout** and **G9in**

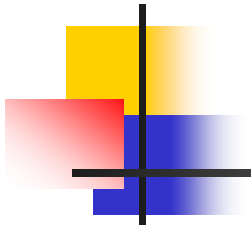
Microprogrammed Control (cont.)



Instruction Fetch

Implemented as

- `PCbout: read: PCout: ALU=add4: Cin;`
- `read: Cout: PCin;`
- `Read: IRbin;`
- `Decodes the instruction and jumps to the appropriate execution rountine`

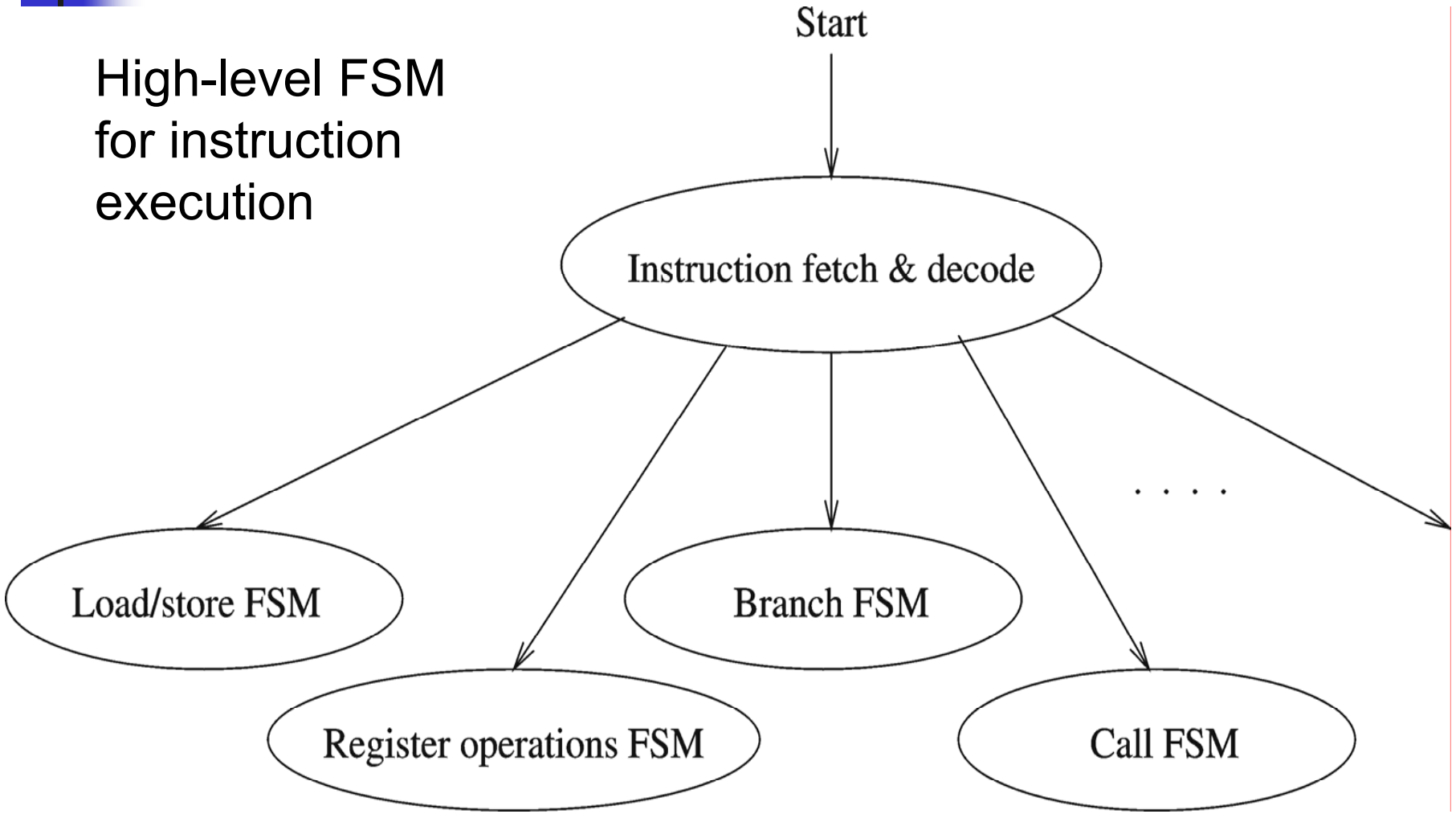


Microprogrammed Control (cont.)

- Example instruction groups
  - Load/store
    - Moves data between registers and memory
  - Register
    - Arithmetic and logic instructions
  - Branch
    - Jump direct/indirect
  - Call
    - Procedures invocation mechanisms
  - More…

Microprogrammed Control (cont.)

High-level FSM for instruction execution



FSM: finite state machine

Microprogrammed Control (cont.)

- Software implementation
  - Typically used in CISC
    - Hardware implementation (PLA) is complex and expensive
- Example

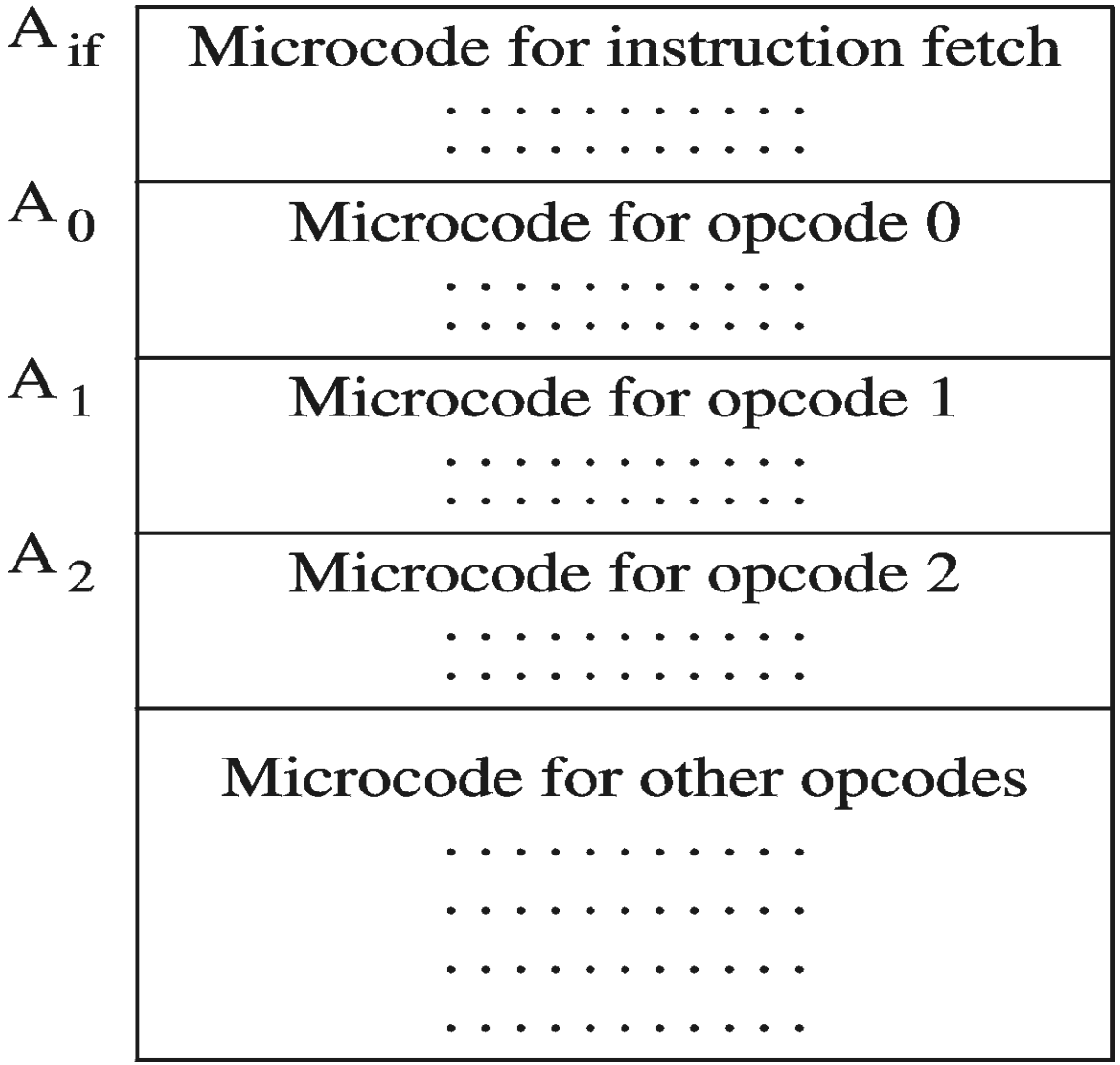
```
add     %G9,%G5,%G7
```

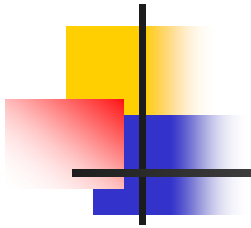
  - Three steps

```
S1    G5out: Ain;
S2    G7out: ALU=add: Cin;
S3    Cout: G9in: end;
```

Microprogrammed Control (cont.)

Simple microcode organization

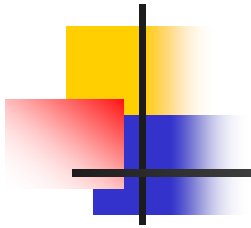




Microprogrammed Control (cont.)

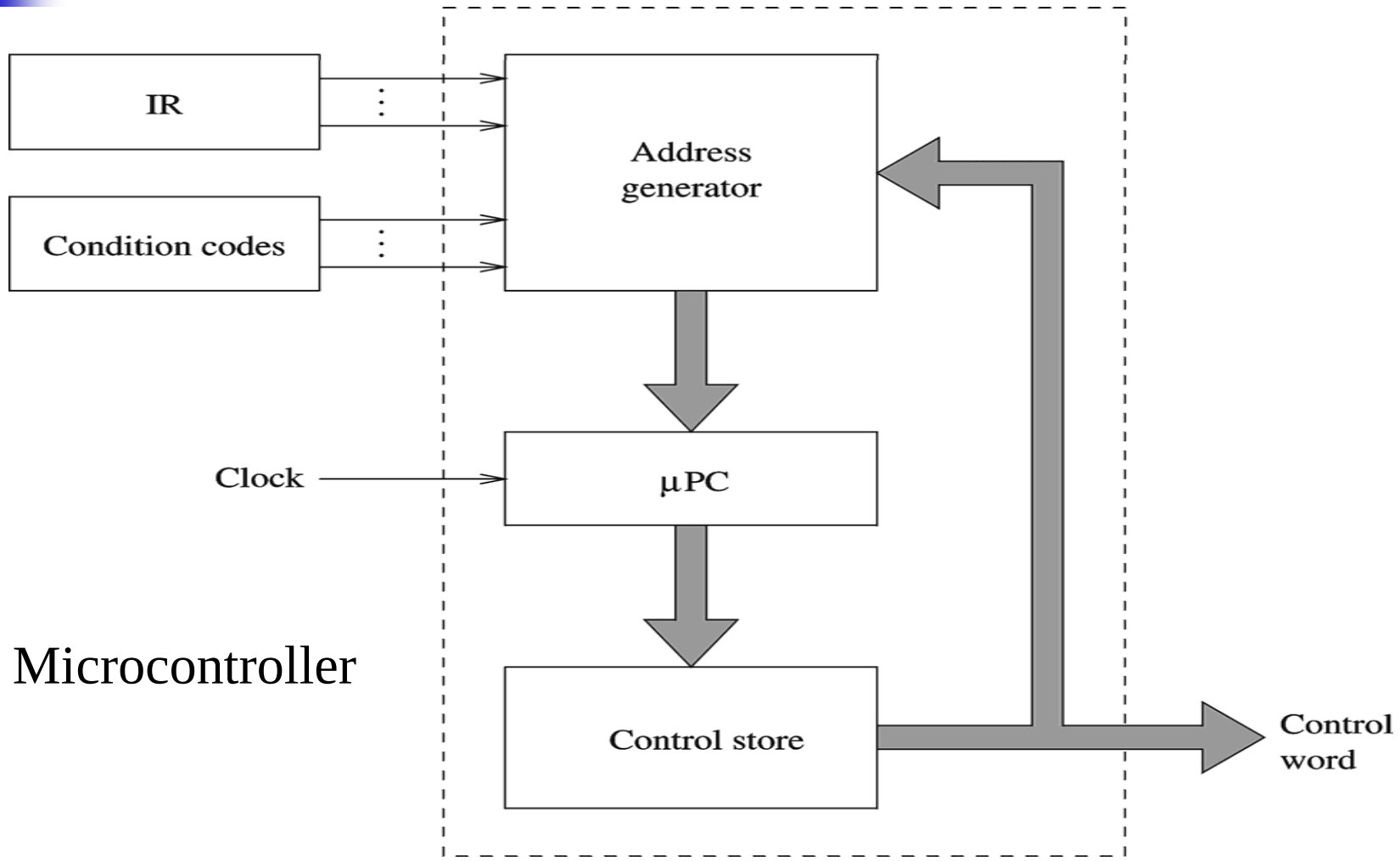
- Uses a microprogram to generate the control signals
  - Encode the signals of each step as a codeword
    - Called *microinstruction*
  - A instruction is expressed by a sequence of codewords
    - Called *microroutine*
- Microprogram essentially implements the FSM discussed before

Microprogrammed Control (cont.)

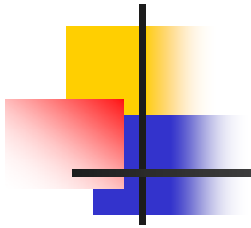


- A simple microcontroller can execute a microprogram to generate the control signals
  - Control store
    - Store microprogram
  - Use μ**PC**
    - Similar to PC
  - Address generator
    - Generates appropriate address depending on the
      - Opcode, and
      - Condition code inputs

Microprogrammed Control (cont.)



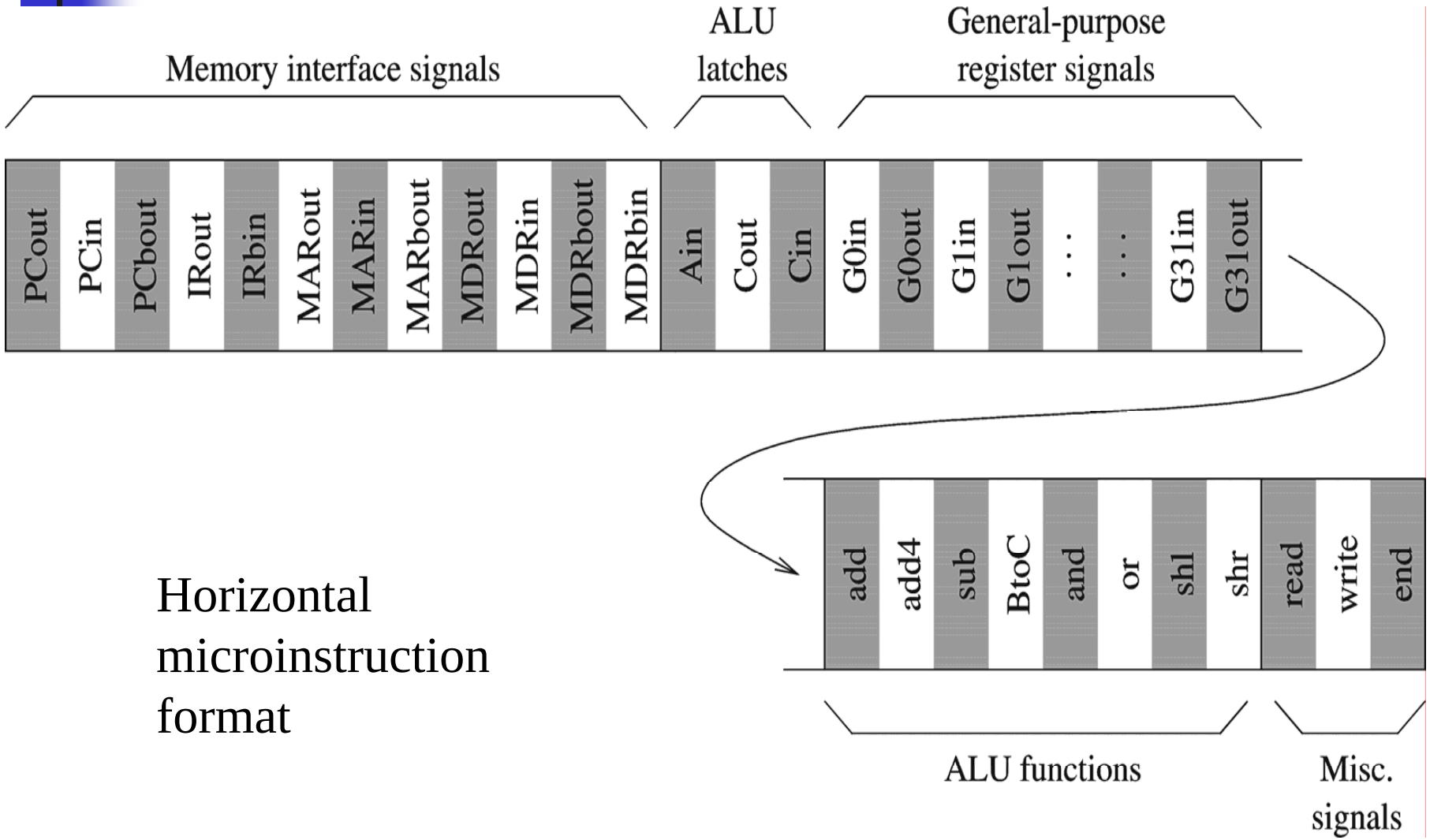
Microcodes reside in control store, which might be read-only memory (ROM)



Microprogrammed Control (cont.)

- Microinstruction format
  - Two basic ways
    - Horizontal organization
    - Vertical organization
  - Horizontal organization
    - One bit for each signal
    - Very flexible
    - Long microinstructions
    - Example: 1-bus datapath
      - Needs 90 bits for each microinstruction

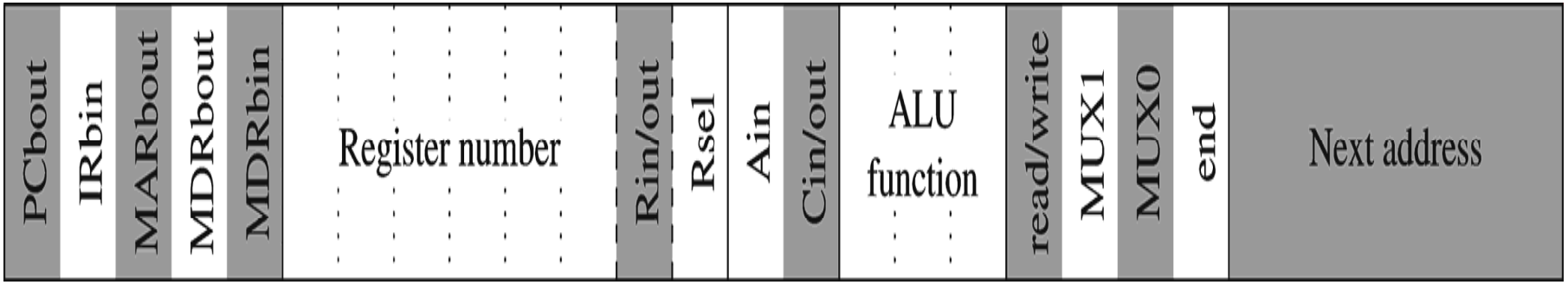
Microprogrammed Control (cont.)



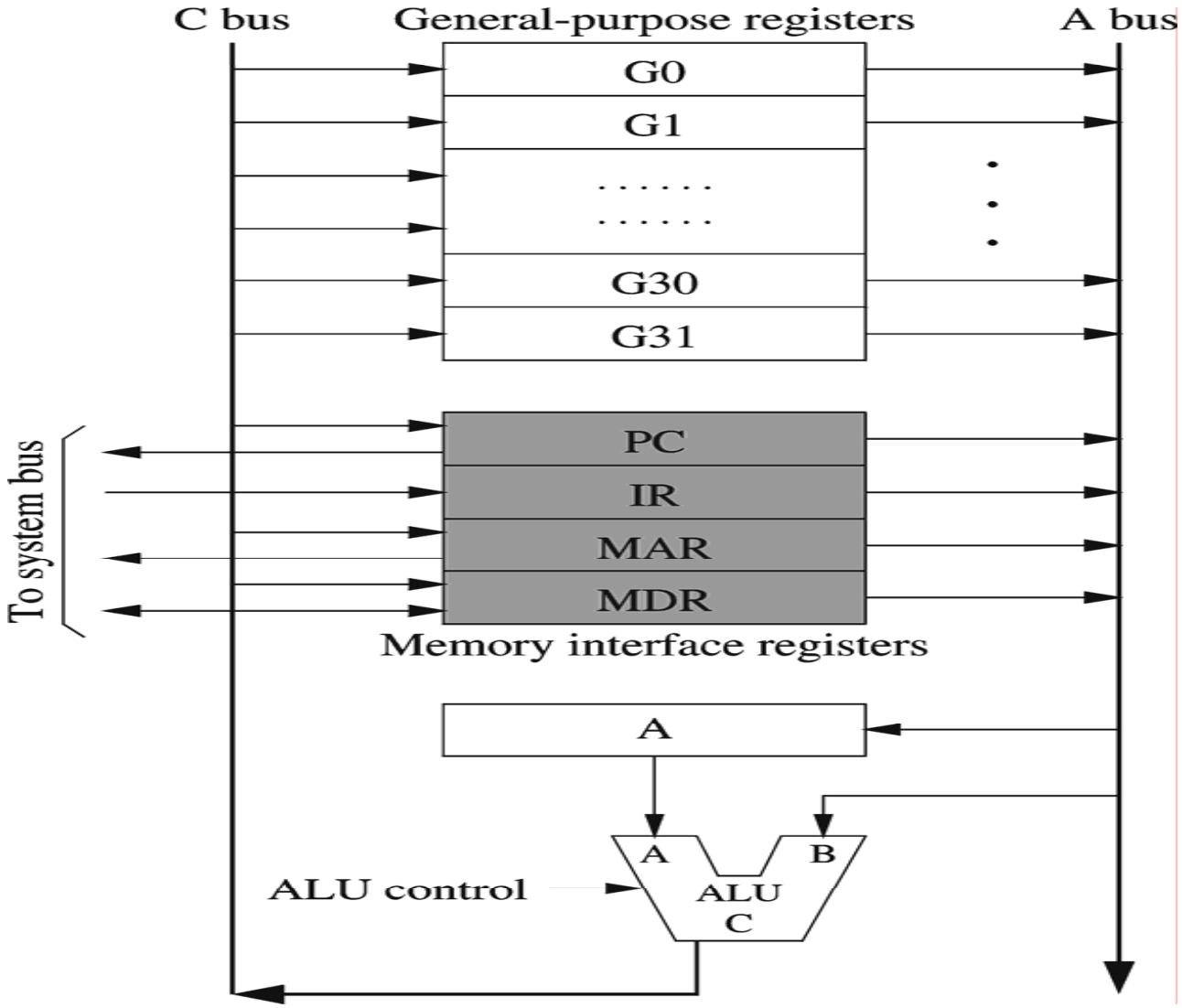
Horizontal microinstruction format

Microprogrammed Control (cont.)

- Vertical organization
  - Encodes to reduce microinstruction length
    - Reduced flexibility
  - Example:
    - Horizontal organization
      - 64 control signals for the 32 general purpose registers
    - Vertical organization
      - 5 bits to identify the register and 1 for in/out



2-bus Datapath



Microprogrammed Control (cont.)

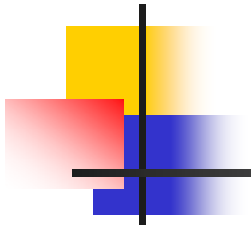
- Adding more buses reduces time needed to execute instructions
  - No need to multiplex the bus
- Example

```
add     %G9,%G5,%G7
```

  - Needed three steps in 1-bus datapath
  - Need only two steps with a 2-bus datapath

```
S1      G5out: Ain;
S2      G7out: ALU=add: G9in;
```

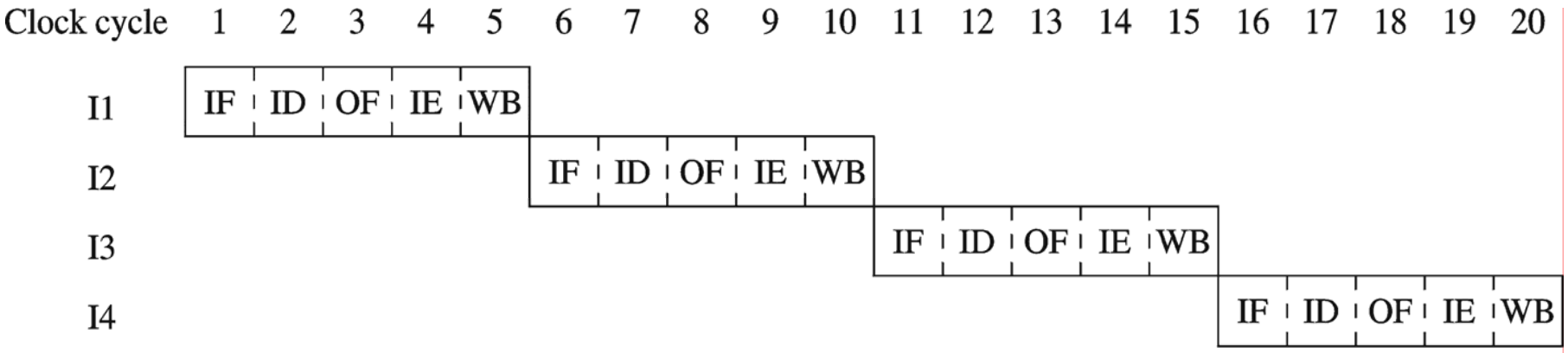
Pipelining



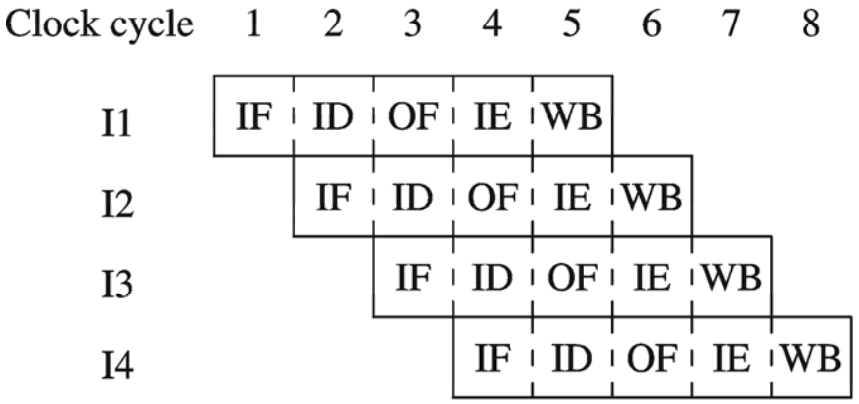
Pipelining

- Introduction
- 3 Hazards
  - Resource, Data and Control Hazards
- 3 Technologies for Performance Improvement
  - Superscalar, Superpipelined, and Very Long Instruction Word

Serial and Pipelining



(a) Serial execution



(b) Pipelined execution

Serial execution: 20 cycles

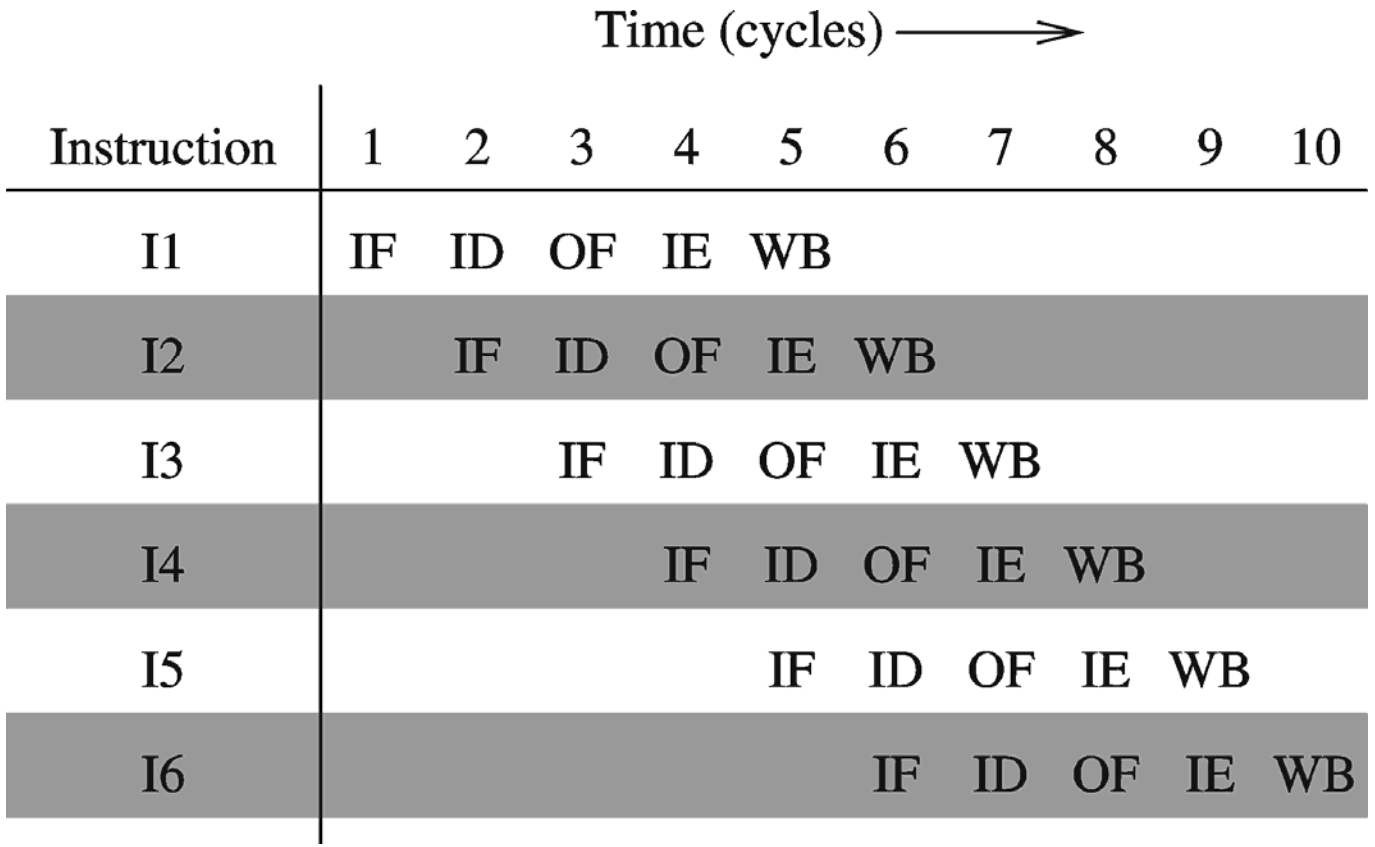
Pipelined execution: 8 cycles

For *k* states and *n* instructions, the number of required cycles is:

$k + (n - 1)$

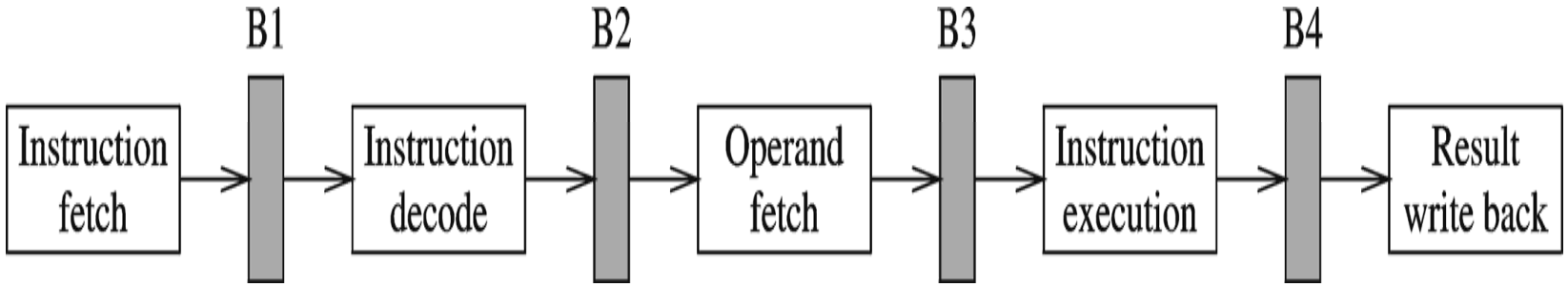
Pipelining

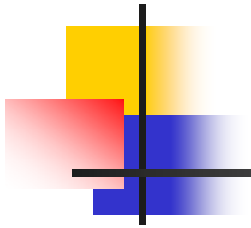
- Pipelining
  - Overlapped execution
  - Increases throughput



Pipelining (cont.)

- Pipelining requires buffers
  - Each buffer holds a single value
  - Uses just-in-time principle
    - Any delay in one stage affects the entire pipeline flow
  - Ideal scenario: equal work for each stage
    - Sometimes it is not possible
    - Slowest stage determines the flow rate in the entire pipeline



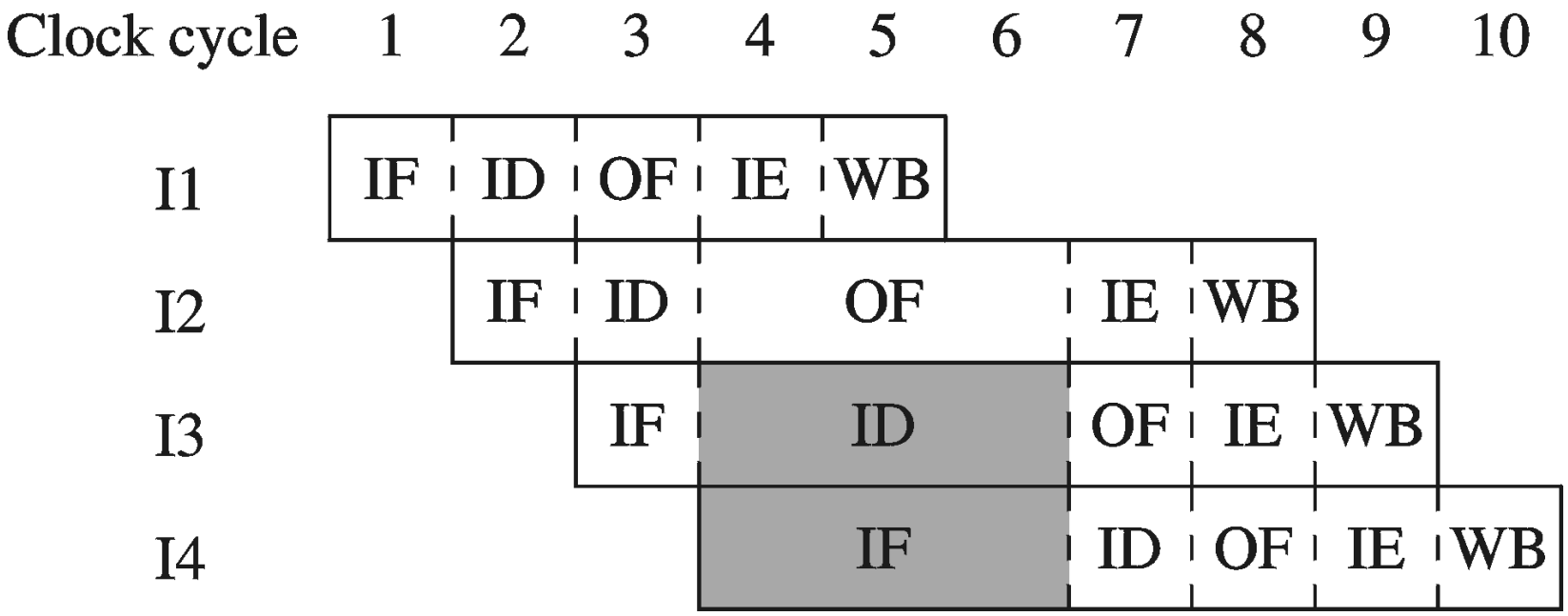


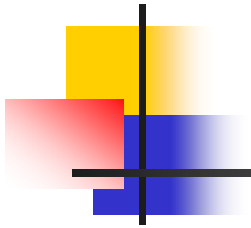
Pipelining (cont.)

- Some reasons for unequal work stages
  - A complex step cannot be subdivided conveniently
  - An operation takes variable amount of time to execute
    - EX: Operand fetch time depends on where the operands are located
      - Registers
      - Cache
      - Memory
  - Complexity of operation depends on the type of operation
    - Add: may take one cycle
    - Multiply: may take several cycles

Pipeline Stall

- Operand fetch of I2 takes three cycles
  - Pipeline ***stalls*** for two cycles
    - Caused by hazards
  - Pipeline stalls reduce overall throughput



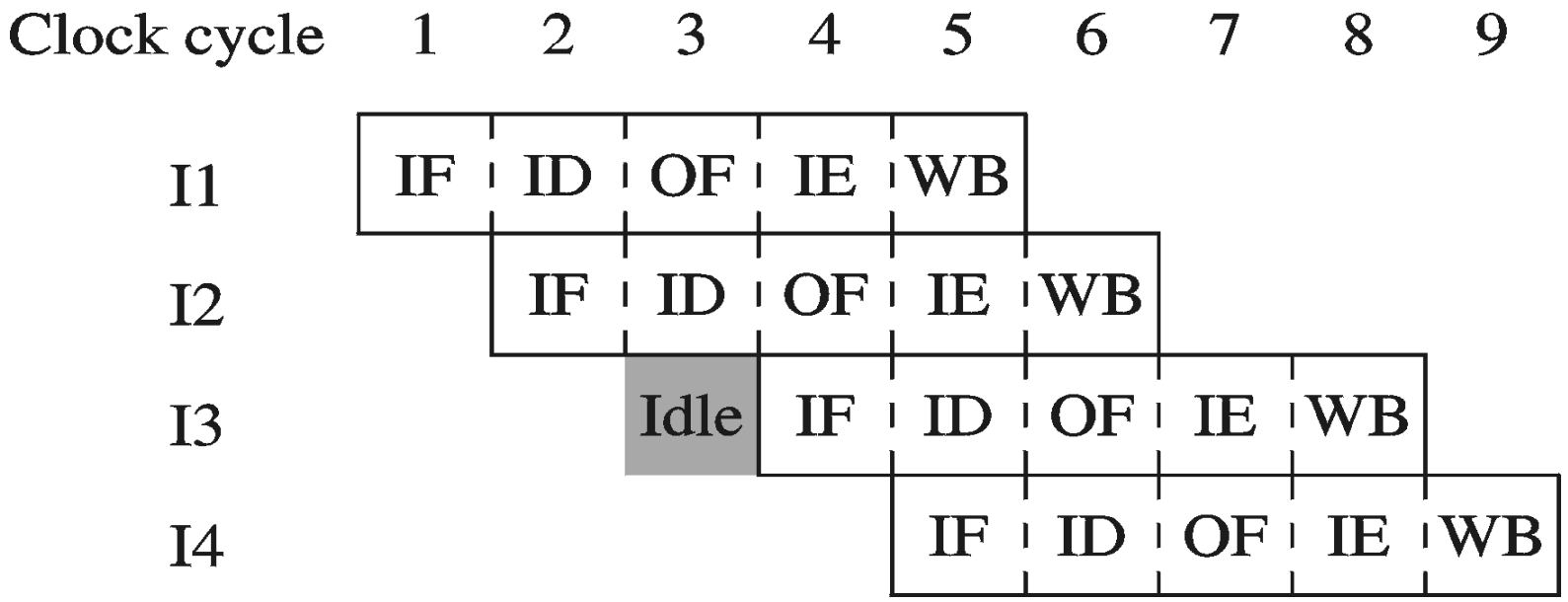


Hazards

- Three types of hazards
  - Resource hazards
    - Occurs when two or more instructions use the same resource
    - Also called ***structural hazards***
  - Data hazards
    - Caused by data dependencies between instructions
      - Example: Result produced by I1 is read by I2
  - Control hazards
    - Default: sequential execution suits pipelining
    - Altering control flow (e.g., branching) causes problems
      - Introduce control dependencies

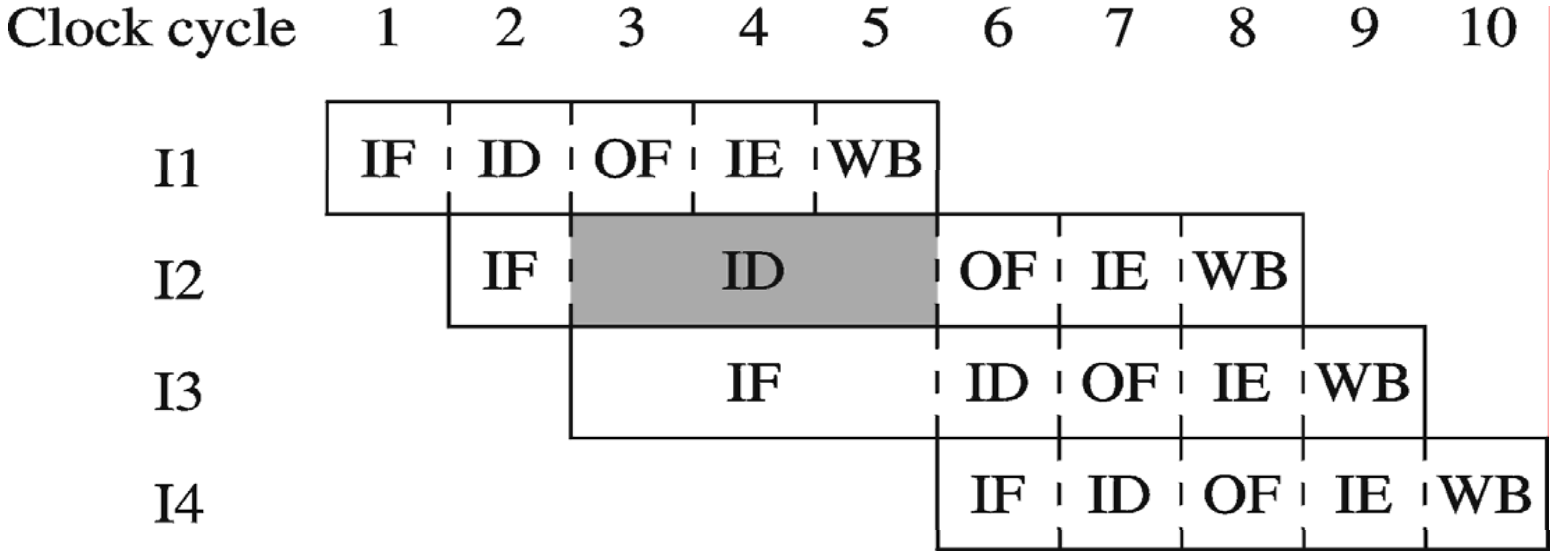
Resource Hazards

- Example
  - Conflict for memory in clock cycle 3
    - I1 fetches operand
    - I3 delays its instruction fetch from the same memory

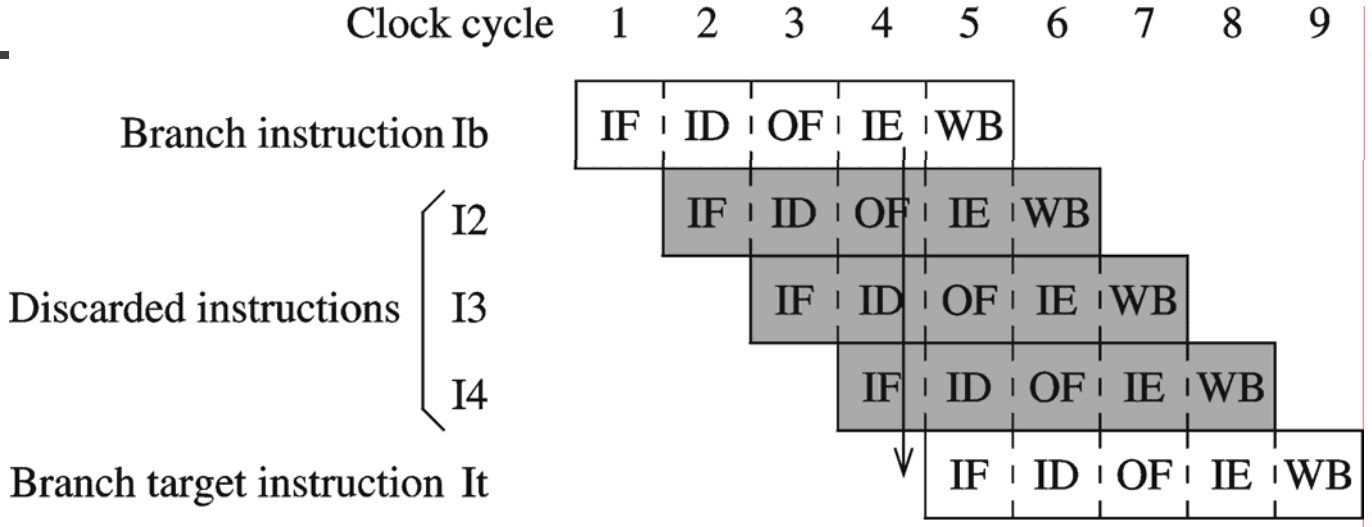


Data Hazards

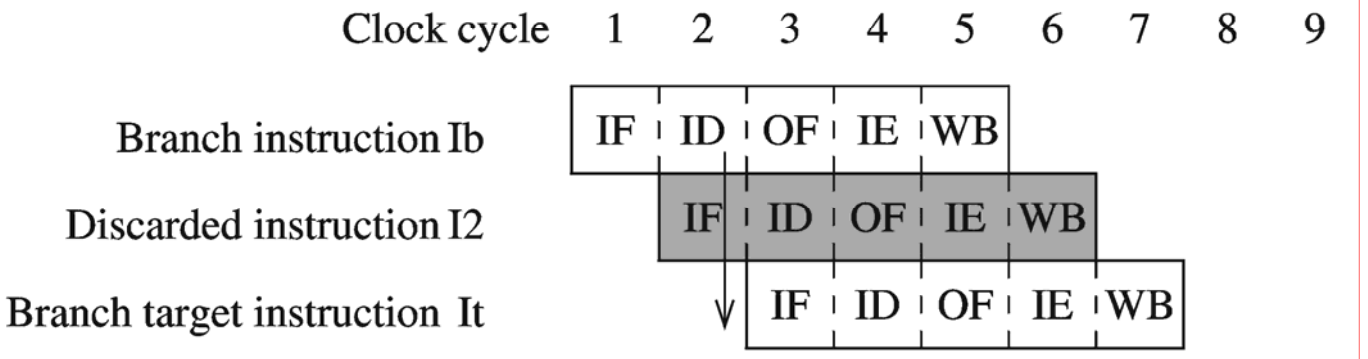
- Example
  - `I1: add  R2,R3,R4  /* R2 = R3 + R4 */`
  - `I2: sub  R5,R6,R2  /* R5 = R6 - R2 */`
- Introduces data dependency between I1 and I2



Control Hazards



(a) Branch decision is known during the IE stage



(b) Branch decision is known during the ID stage

»Determine branch decision early

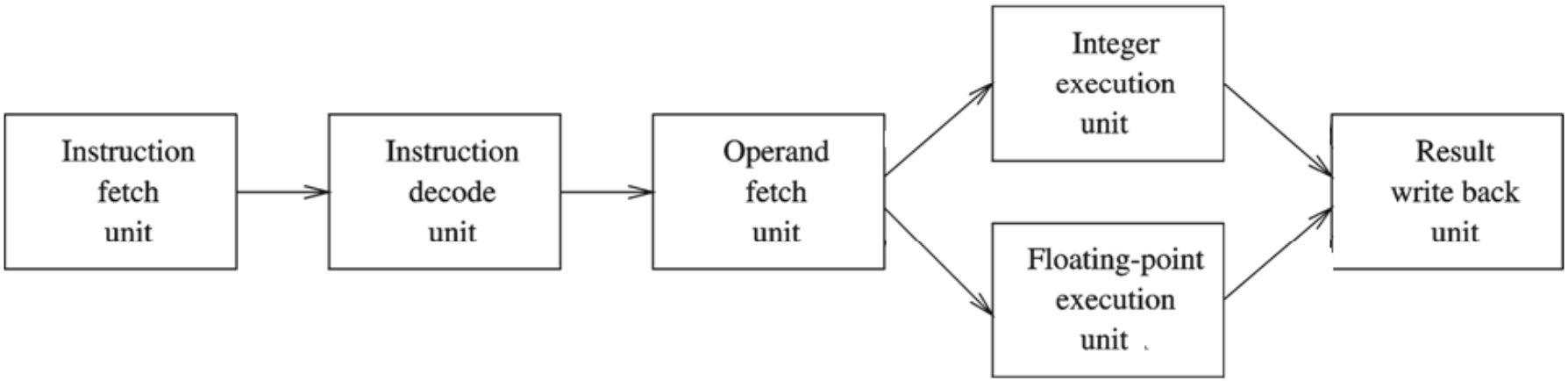
Performance Improvement

- Several techniques to improve performance of a pipelined system
  - Superscalar
    - Replicates the pipeline hardware
  - Superpipelined
    - Increases the pipeline depth
  - Very long instruction word (VLIW)
    - Encodes multiple operations into a long instruction word
    - Hardware schedules these instructions on multiple functional units (No run-time analysis)
    -
```
add R1, R2, R3     ; R1 = R2 + R3
sub R5, R6, R7     ; R5 = R6 – R7
and R4, R1, R5     ; R4 = R1 AND R5
xor  R9, R9, R9    ; R9 = R9 XOR R9
```

cycle 1: add, sub, xor

cycle 2: and

Superscalar Processor



Ex: Pentium

Wasted Cycles (pipelined)

- When one of the stages requires two or more clock cycles, clock cycles are again wasted.

Stages

exe

Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3	I-3	I-2	I-1			
4		I-3	I-2	I-1		
5			I-3	I-1		
6				I-2	I-1	
7				I-2		I-1
8				I-3	I-2	
9				I-3		I-2
10					I-3	
11						I-3

For k states and n instructions, the number of required cycles is:

$$k + (2n - 1)$$

Superscalar

A superscalar processor has multiple execution pipelines. In the following, note that Stage S4 has left and right pipelines (u and v).

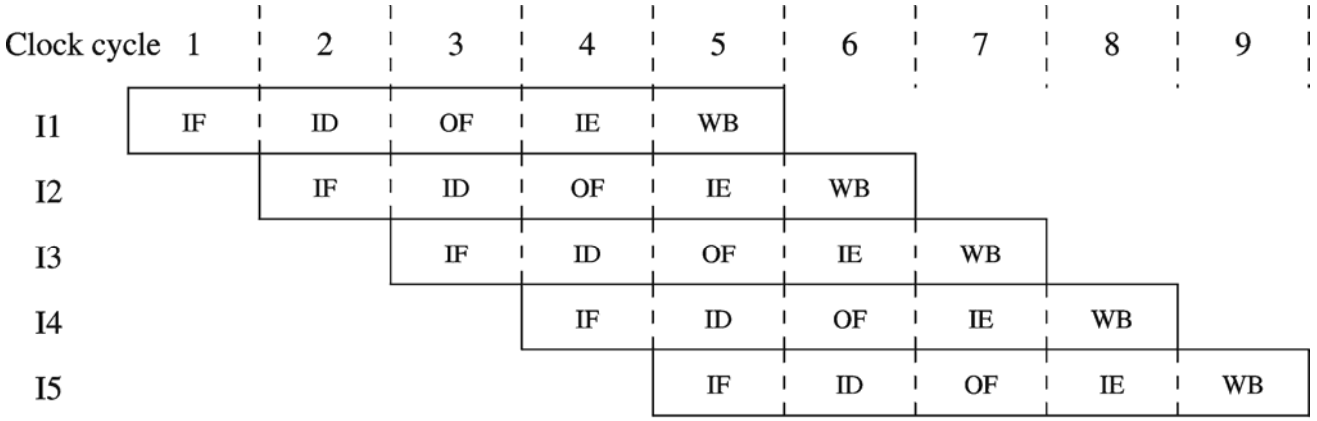
Stages

Cycles	S1	S2	S3	S4 (u)	S4 (v)	S5	S6
1	I-1						
2	I-2	I-1					
3	I-3	I-2	I-1				
4	I-4	I-3	I-2	I-1			
5		I-4	I-3	I-1	I-2		
6			I-4	I-3	I-2	I-1	
7				I-3	I-4	I-2	I-1
8					I-4	I-3	I-2
9						I-4	I-3
10							I-4

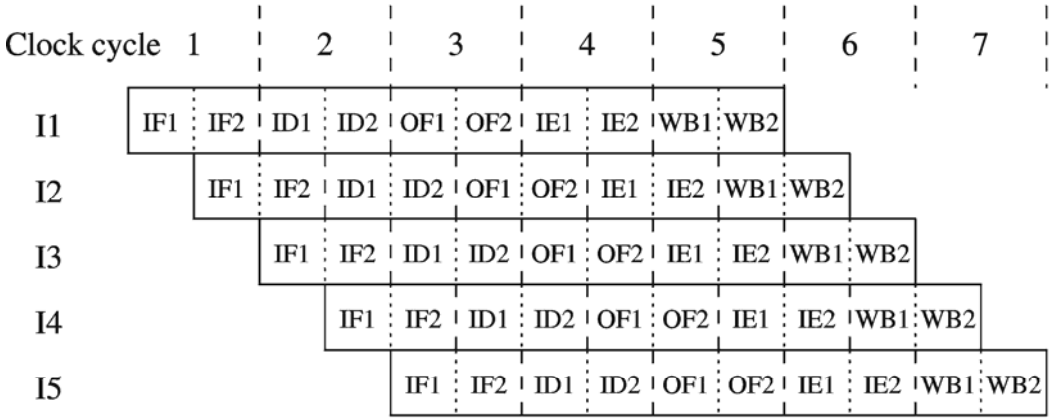
For k states and n instructions, the number of required cycles is:

$$k + n$$

Superpipelined Processor



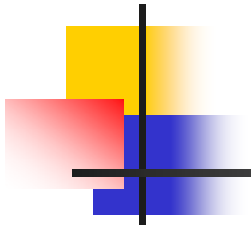
(a) Pipelined execution



(b) Superpipelined execution

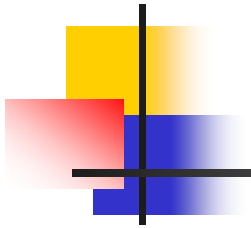
Ex: MIPS R4000

Memory



Memory

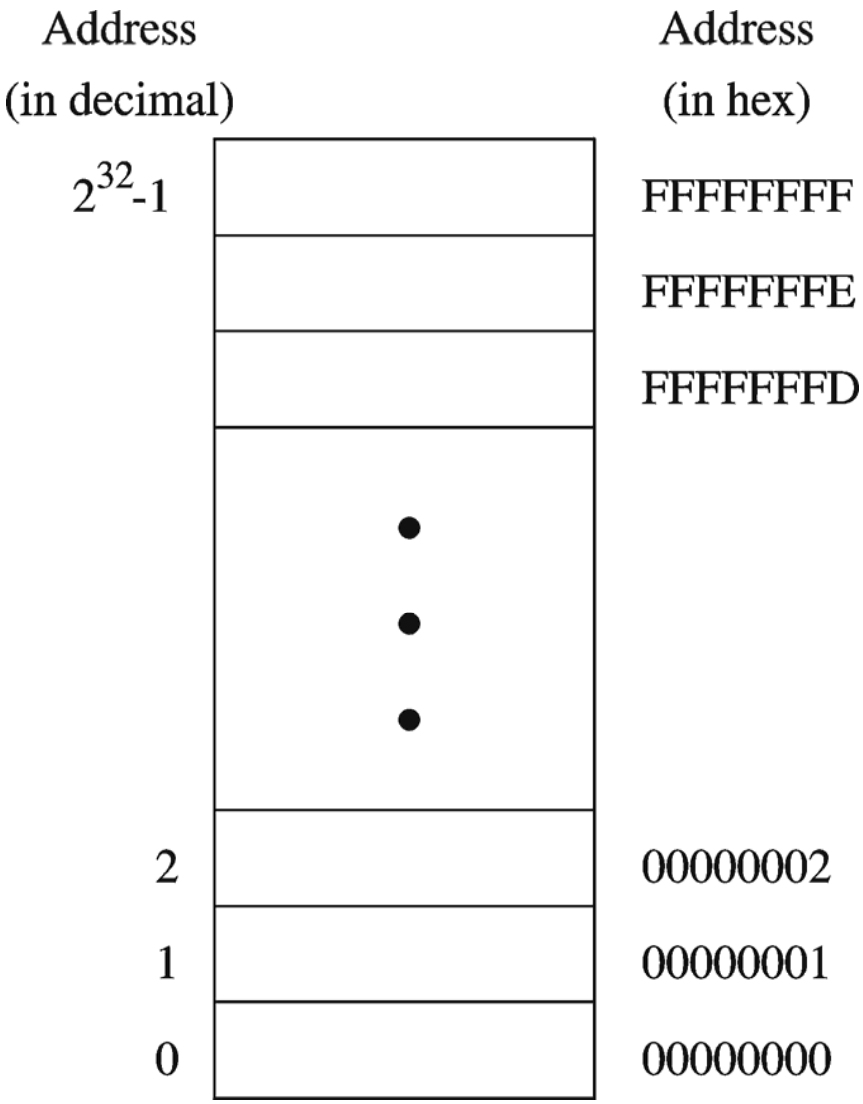
- Introduction
- Building Memory Blocks
- Alignment of Data
- 2 Memory Design Issues
  - Cache
  - Virtual Memory

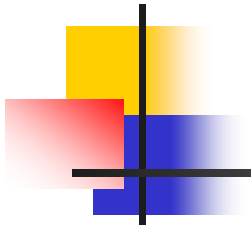


Memory (cont.)

- Ordered sequence of bytes
  - The sequence number is called the ***memory address***
  - Byte addressable memory
    - Each byte has a unique address
    - Almost all processors support this
- Memory address space
  - Determined by the address bus width
  - Pentium has a 32-bit address bus
    - address space = 4GB ($\mathbf{2^{32}}$)
  - Itanium with 64-bit address bus supports
    - $\mathbf{2^{64}}$ bytes of address space

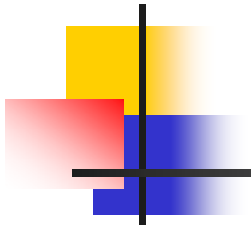
Memory (cont.)


Address
(in decimal)
Address
(in hex)
2^{32}-1
FFFFFFFF
FFFFFFFE
FFFFFFFD
2
00000002
1
00000001
0
00000000



Memory (cont.)

- Read cycle
  1. Place address on the address bus
  2. Assert memory read control signal
  3. Wait for the memory to retrieve the data
     - Introduce ***wait states*** if using a slow memory
  4. Read the data from the data bus
  5. Drop the memory read signal
- In Pentium, a simple read takes three clocks cycles
  - Clock 1: steps 1 and 2
  - Clock 2: step 3
  - Clock 3 : steps 4 and 5



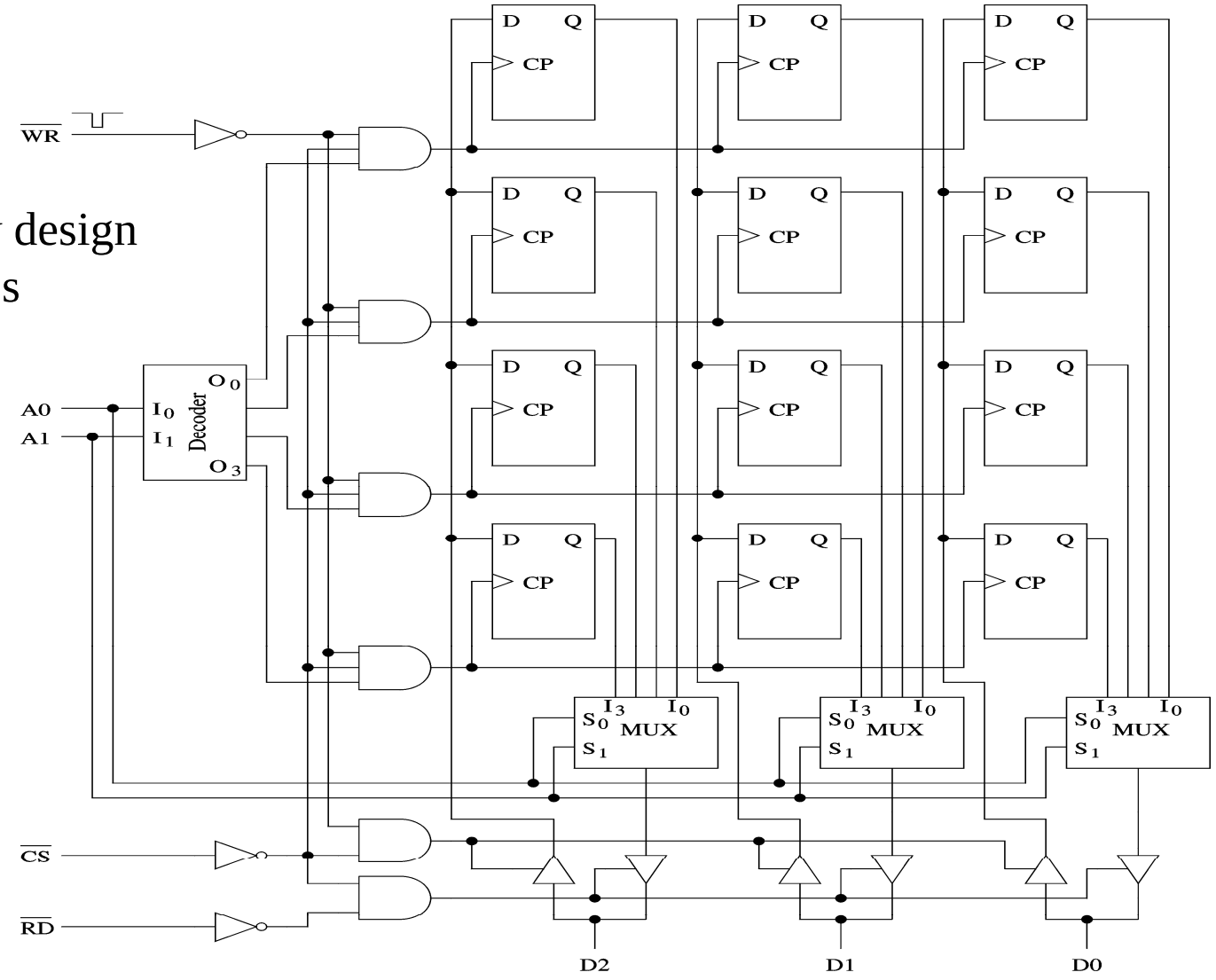
Memory (cont.)

- Write cycle
  1. Place address on the address bus
  2. Place data on the data bus
  3. Assert memory write signal
  4. Wait for the memory to retrieve the data
     - Introduce ***wait states*** if necessary
  5. Drop the memory write signal
- In Pentium, a simple write also takes three clocks
  - Clock 1: steps 1 and 3
  - Clock 2: step 2
  - Clock 3 : steps 4 and 5

How Hardware Implements Memory Systems

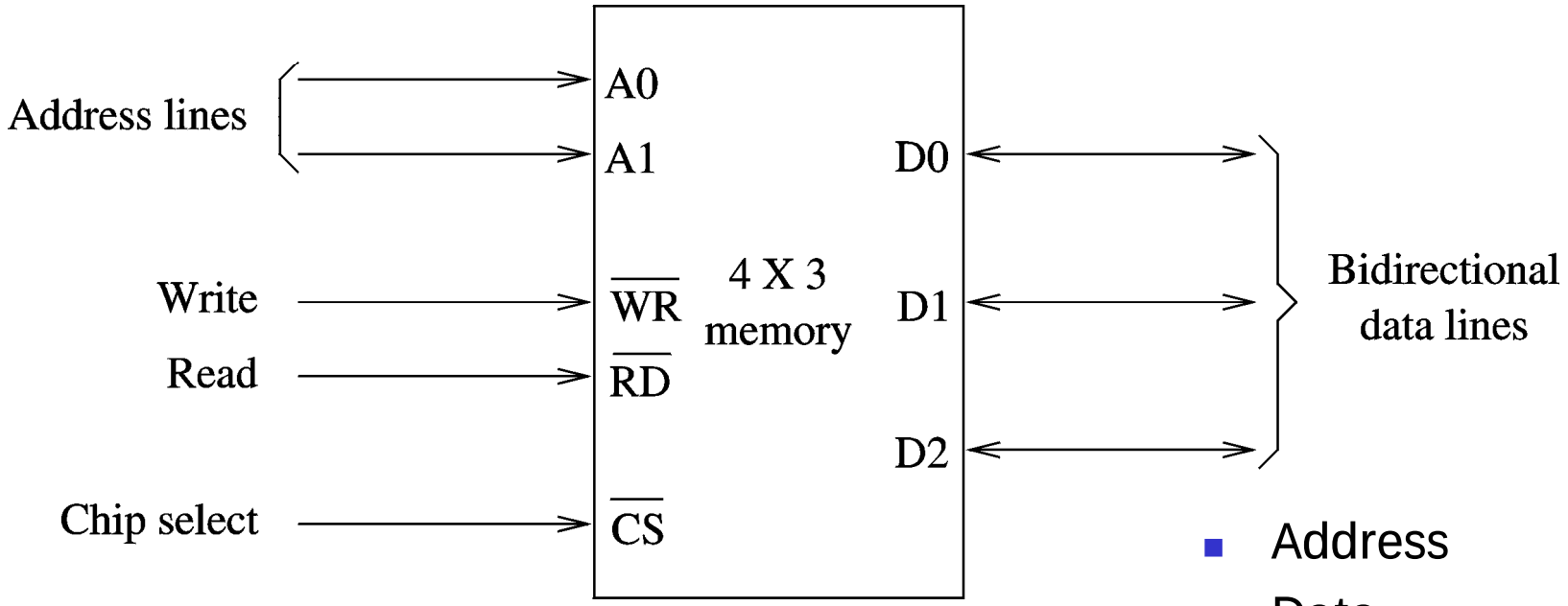
Building a Memory Block

A 4 X 3 memory design using D flip-flops



Building a Memory Block (cont'd)

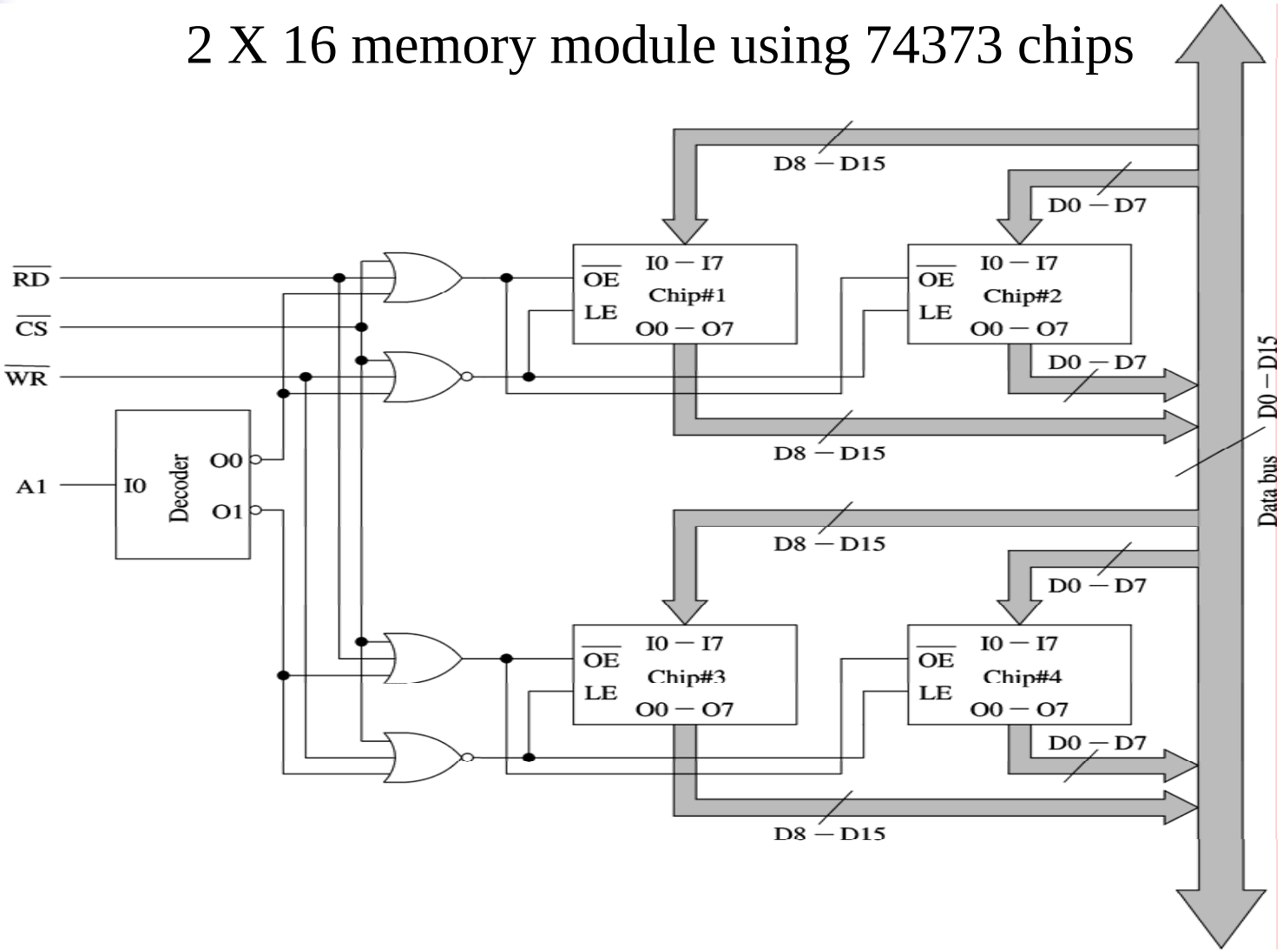
Block diagram representation of a 4x3 memory



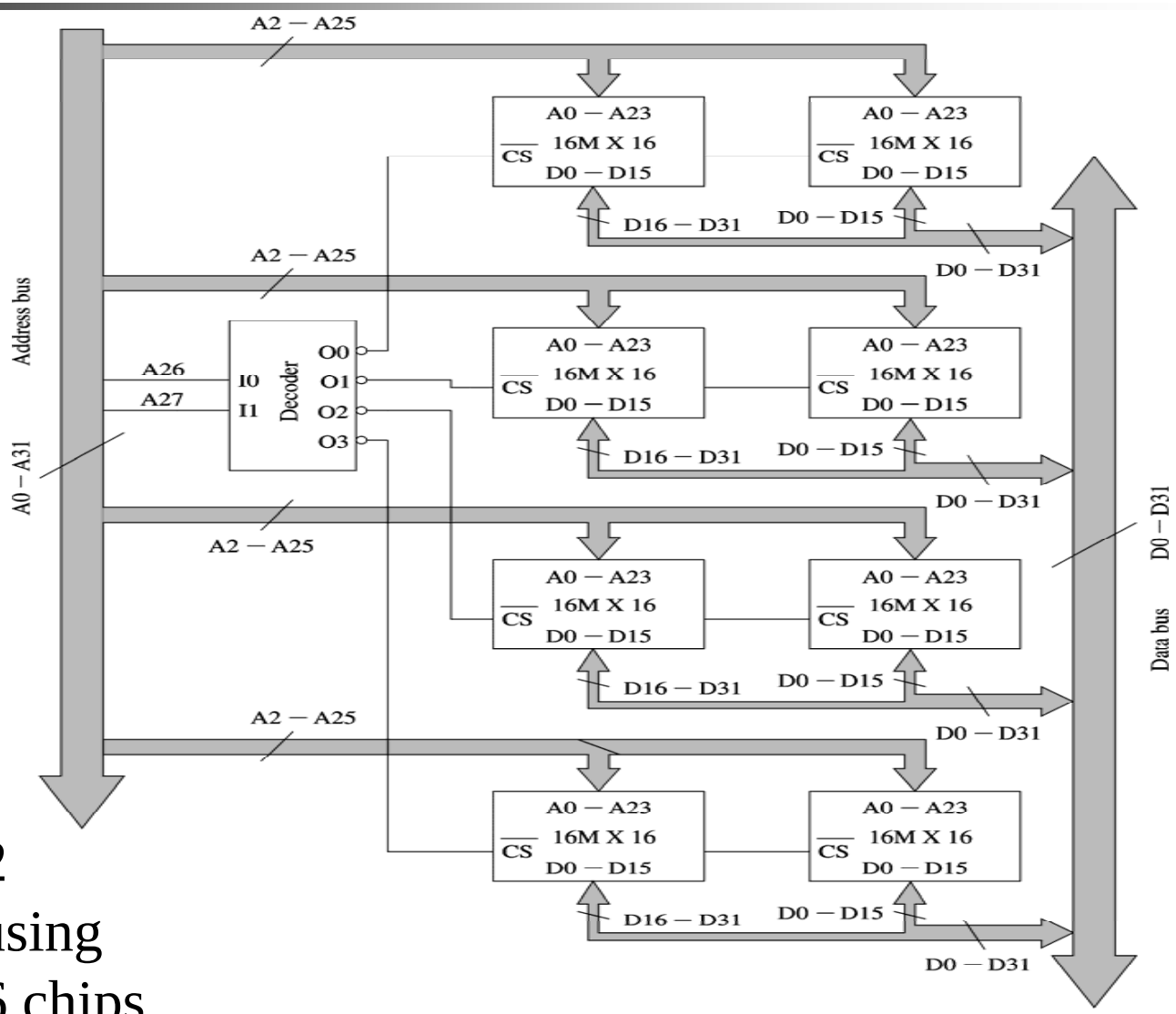
- Address
- Data
- Control signals
  - Read
  - Write

Building Larger Memories

2 X 16 memory module using 74373 chips

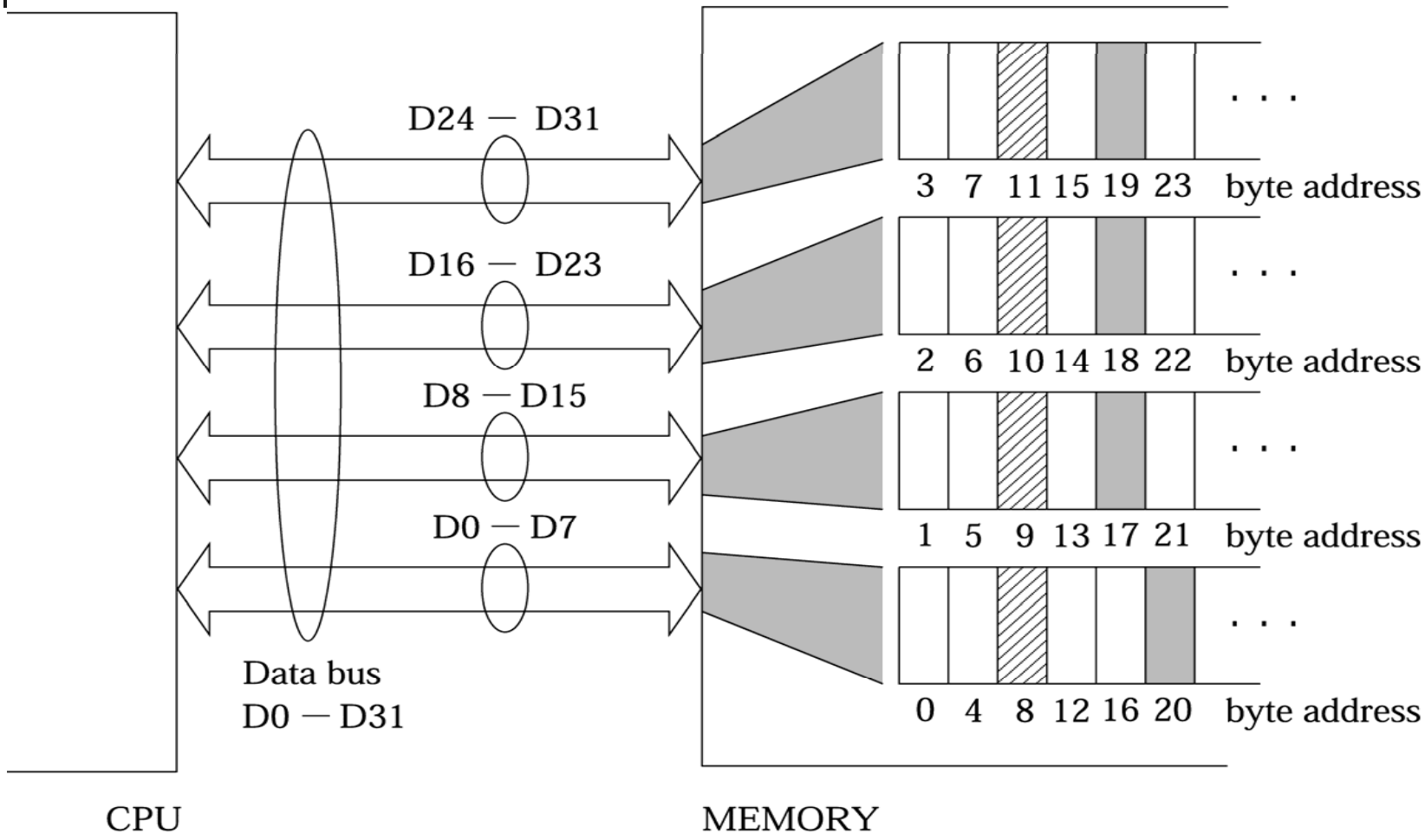


Designing Larger Memories

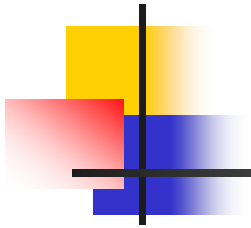


64M X 32
memory using
16M X 16 chips

Alignment of Data

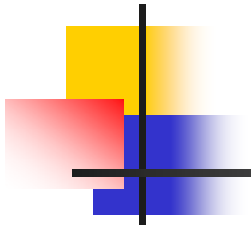


Get 32-bit data in one or more read cycle?



Alignment of Data (cont.)

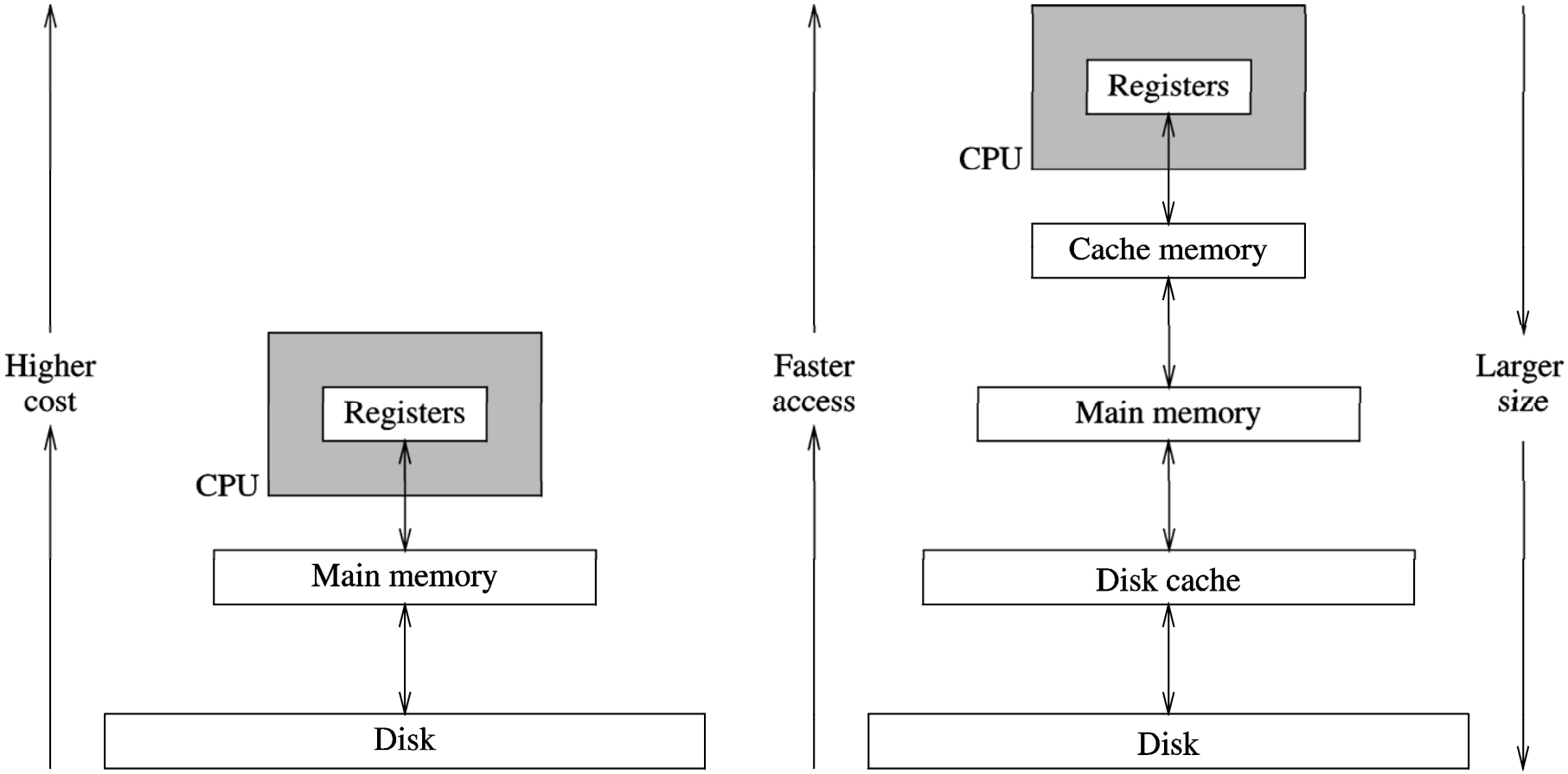
- Alignment
  - 2-byte data: Even address
    - Rightmost address bit should be zero
  - 4-byte data: Address that is multiple of 4
    - Rightmost 2 bits should be zero
  - 8-byte data: Address that is multiple of 8
    - Rightmost 3 bits should be zero
  - Soft alignment
    - Can handle aligned as well as unaligned data
  - Hard alignment
    - Handles only aligned data (enforces alignment)

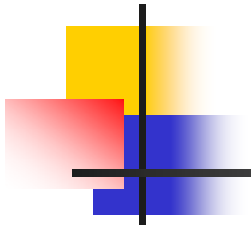


Memory Design Issues

- Slower memories
  - Problem: Speed gap between processor and memory
  - Solution: Cache memory
    - Use small amount of fast memory
    - Make the slow memory appear faster
    - Works due to “reference locality”
- Size limitations
  - Limited amount of physical memory
    - Overlay technique
      - Programmer managed
  - Virtual memory
    - Automates overlay management
    - Some additional benefits

Memory Hierarchy





Cache Memory

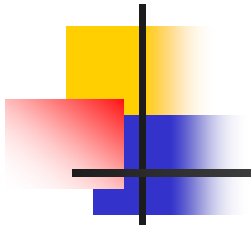
- High-speed expensive static RAM both inside and outside the CPU.
  - Level-1 cache: inside the CPU
  - Level-2 cache: outside the CPU
- Prefetch data into cache before the processor needs it
  - Need to predict processor future access requirements
  - ***Locality of reference***
- Cache hit: when data to be read is already in cache memory
- Cache miss: when data to be read is not in cache memory. When? compulsory, capacity and conflict.
- Cache design: cache size, n-way, block size, replacement policy

Why Cache Memory Works

- Example

```
for (i=0; i<M; i++)
    for(j=0; j<N; j++)
        X[i][j] = X[i][j] + K;
```

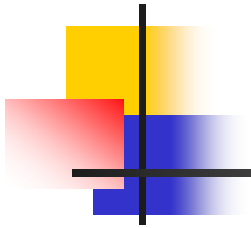
  - Each element of X is **double** (eight bytes)
  - Loop is executed (M*N) times
    - Placing the code in cache avoids access to main memory
      - Repetitive use
      - Temporal locality
    - Prefetching data
      - Spatial locality



Cache Design Basics

- On every read miss
  - A fixed number of bytes are transferred
    - More than what the processor needs
      - Effective due to spatial locality
- Cache is divided into blocks of B bytes
    - b-bits are needed as offset into the block

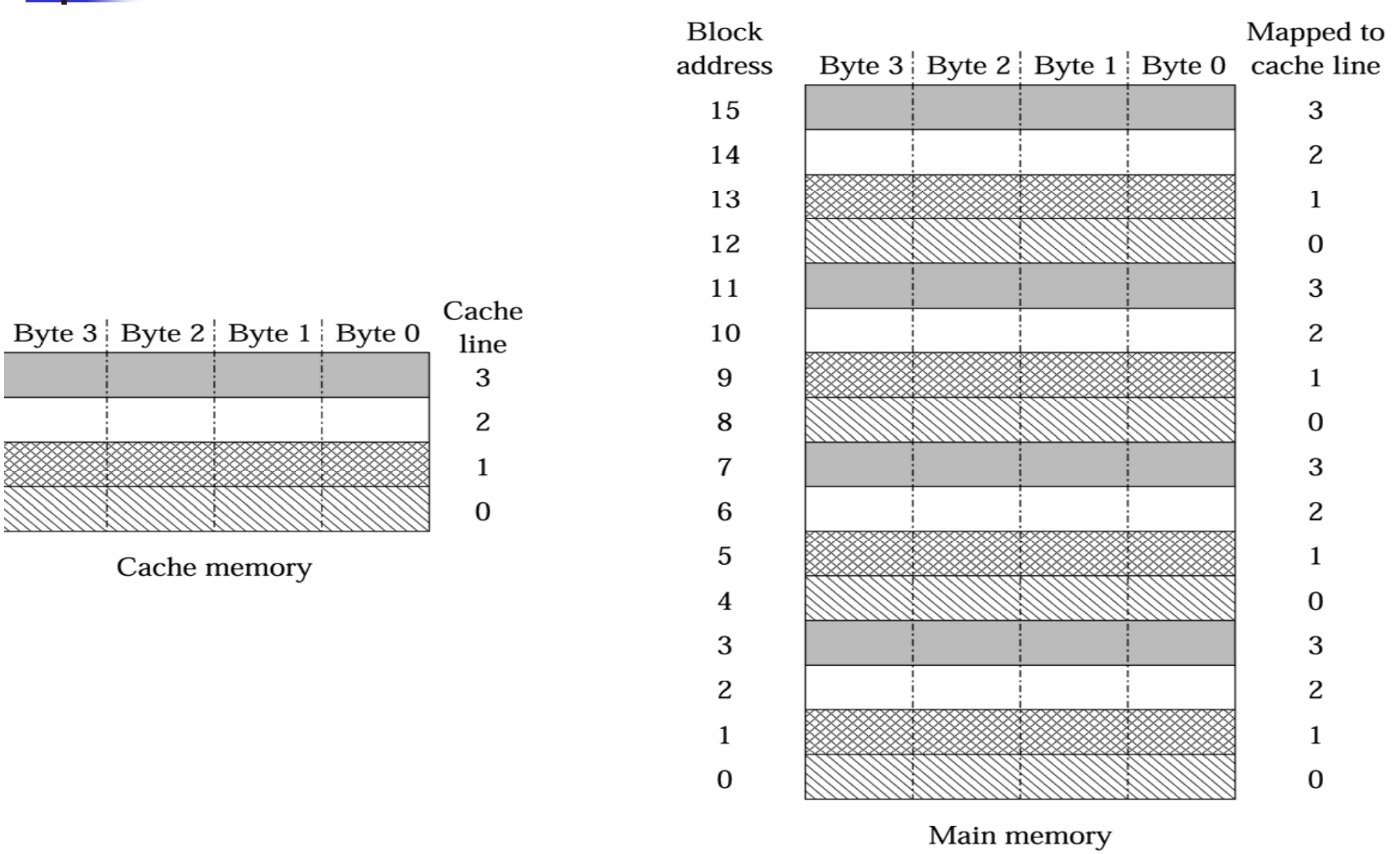
      $b = \log_2 B$
    - Block are called *cache lines*
- Main memory is also divided into blocks of same size



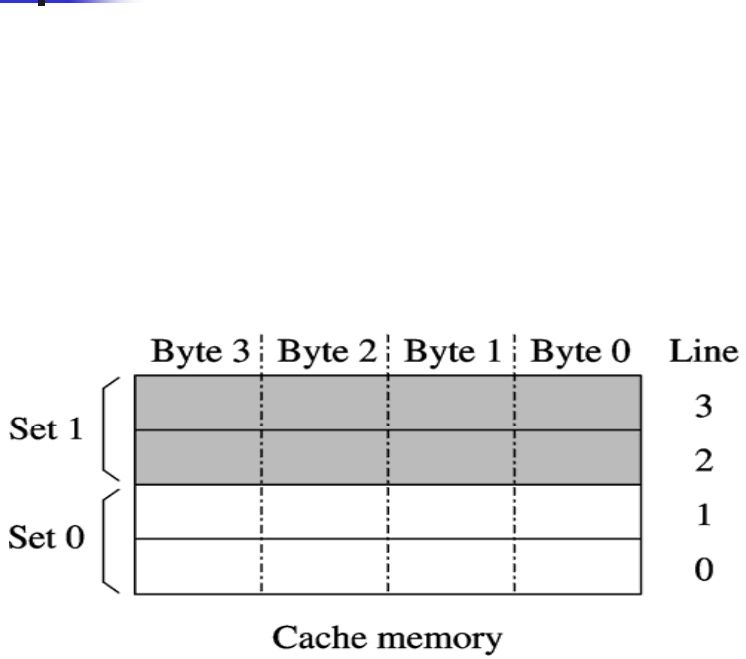
Mapping Function

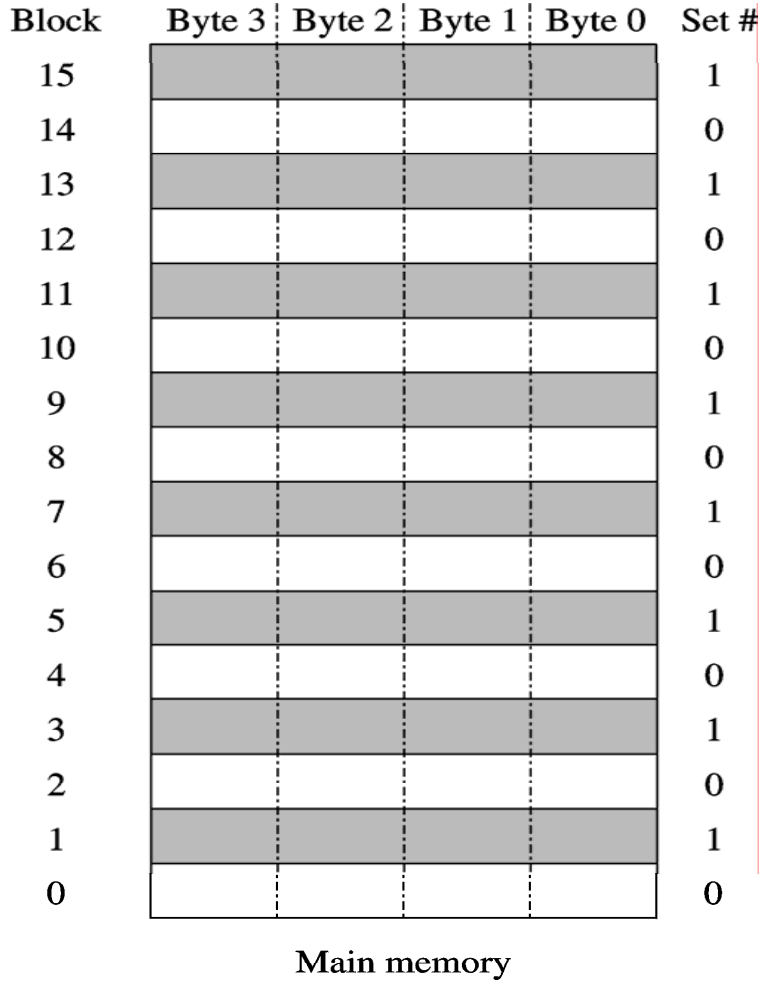
- Determines how memory blocks are mapped to cache lines
- Three types
  - Direct mapping
    - Specifies a single cache line for each memory block
  - Set-associative mapping
    - Specifies a set of cache lines for each memory block
  - Associative mapping
    - No restrictions
      - Any cache line can be used for any memory block

Direct Mapping

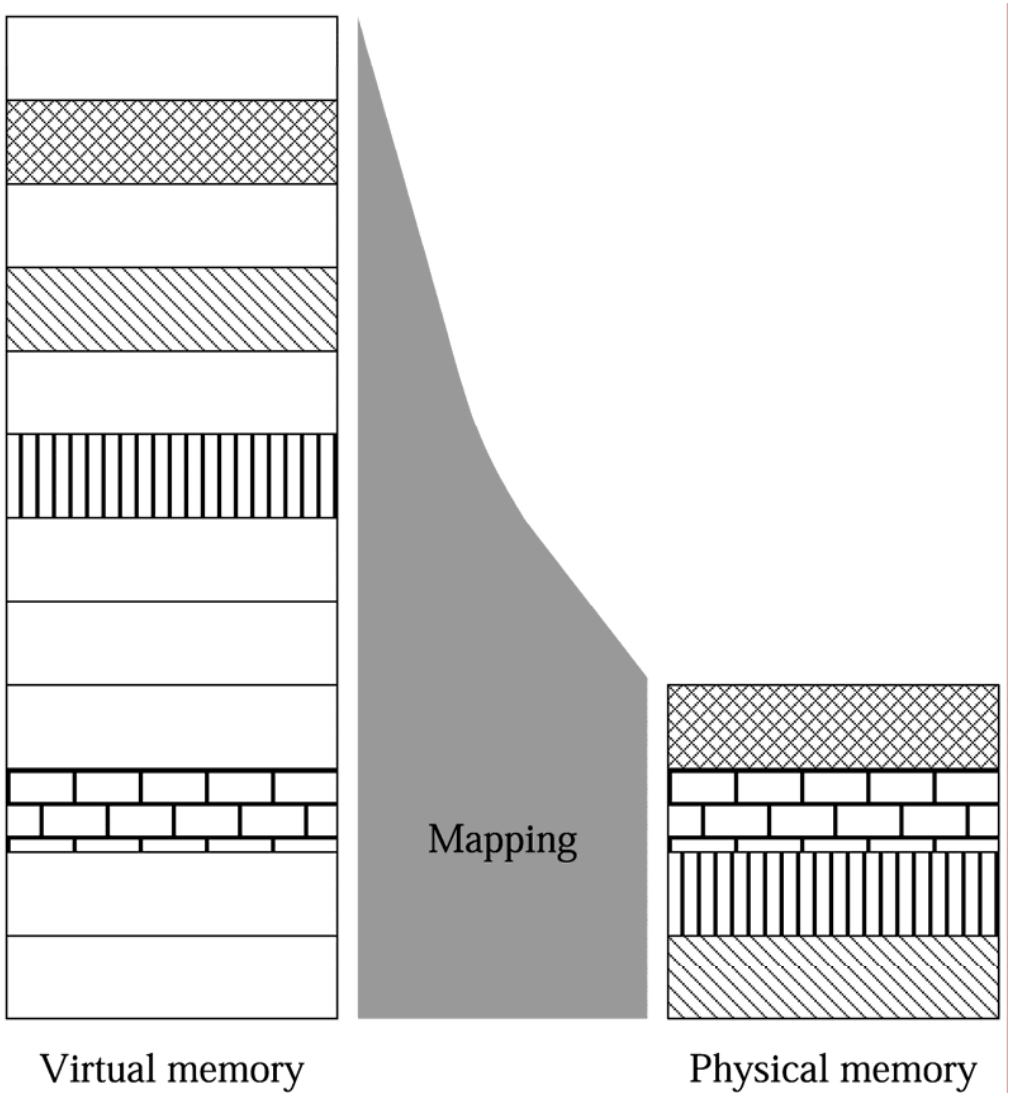


Set-Associate Mapping

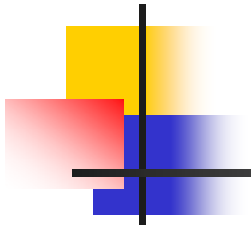




Virtual Memory



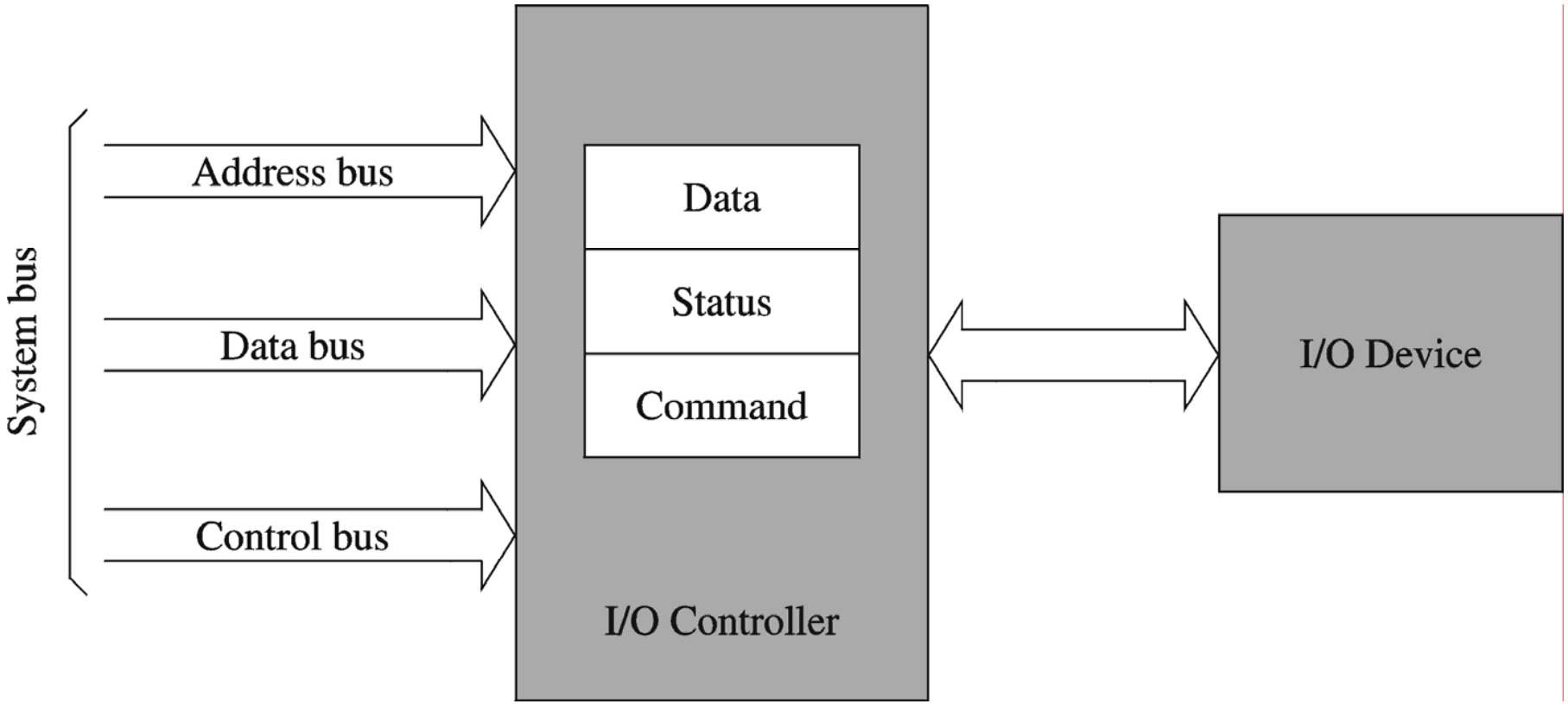
I/O Devices

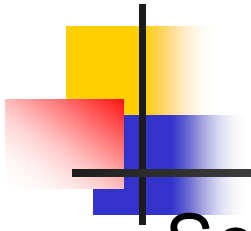


Input/Output

- I/O devices are interfaced via an I/O controller
  - Takes care of low-level operations details
- Several ways of mapping I/O
  - Memory-mapped I/O
    - Reading and writing similar to memory read/write
    - Uses same memory read and write signals
    - Most processors use this I/O mapping
  - Isolated I/O
    - Separate I/O address space
    - Separate I/O read and write signals are needed
    - Pentium supports isolated I/O
      - Also supports memory-mapped I/O

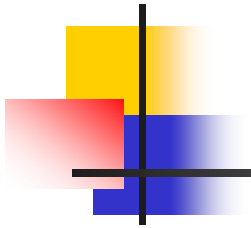
Input/Output (cont.)





Input/Output (cont.)

- Several ways of transferring data
  - Programmed I/O
    - Program uses a busy-wait loop
      - Anticipated transfer
  - Direct memory access (DMA)
    - Special controller (DMA controller) handles data transfers
    - Typically used for bulk data transfer
  - Interrupt-driven I/O
    - Interrupts are used to initiate and/or terminate data transfers
      - Powerful technique
      - Handles unanticipated transfers



Interconnection

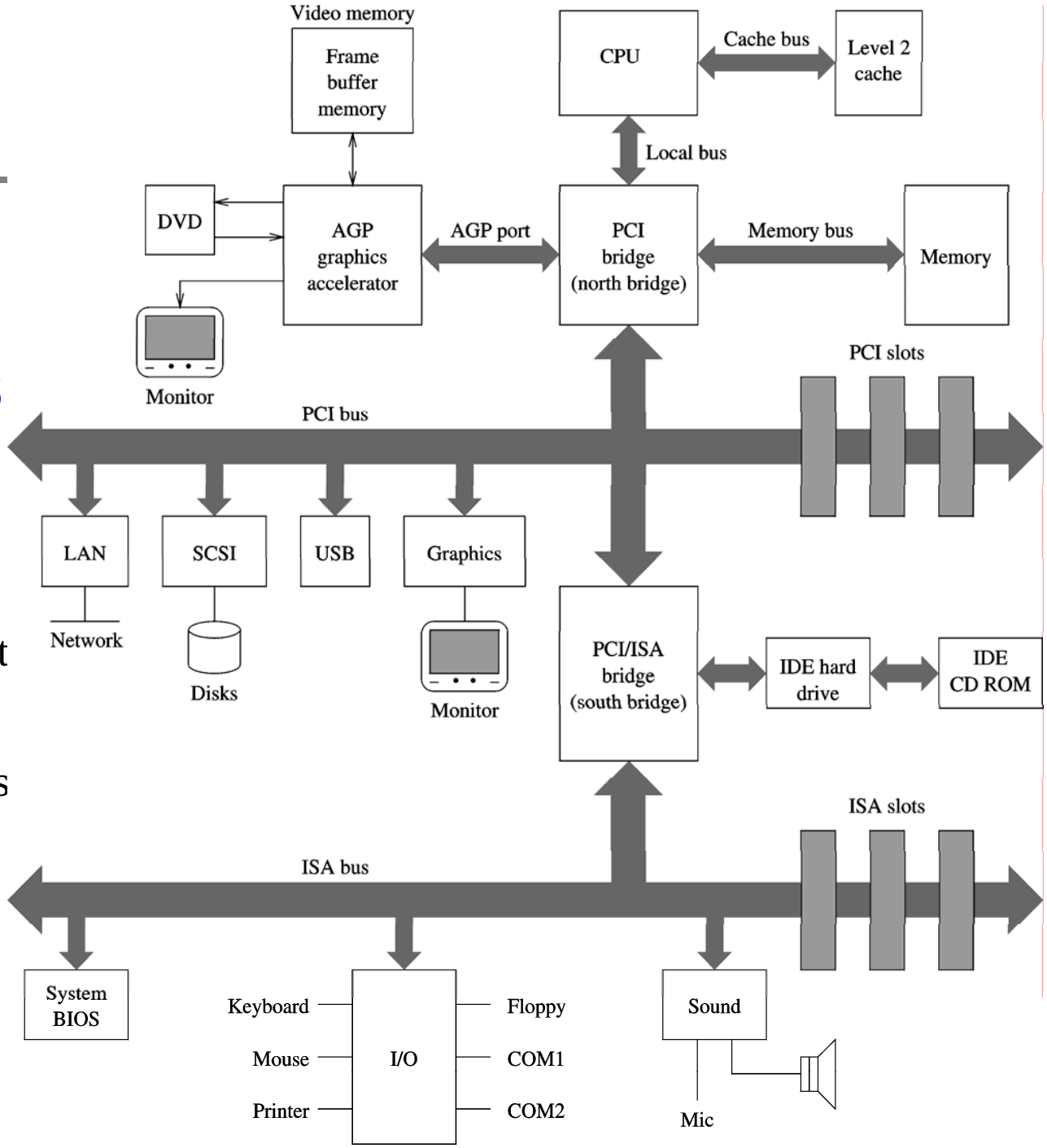
- System components are interconnected by buses
  - Bus: a bunch of parallel wires
- Uses several buses at various levels
  - On-chip buses
    - Buses to interconnect ALU and registers
      - A, B, and C buses in our example
    - Data and address buses to connect on-chip caches
  - Internal buses
    - PCI, AGP, PCMCIA
  - External buses
    - Serial, parallel, USB, IEEE 1394 (FireWire)

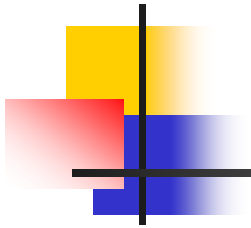
PC System Buses

ISA (Industry Standard Architecture)

PCI (Peripheral Component Interconnect)

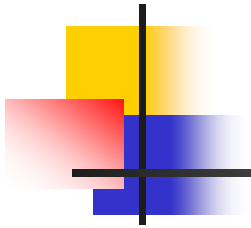
AGP (Accelerated Graphics Port)





Interconnection (cont.)

- Bus is a shared resource
  - Bus transactions
    - Sequence of actions to complete a well-defined activity
    - Involves a master and a slave
      - Memory read, memory write, I/O read, I/O write
  - Bus operations
    - A bus transaction may perform one or more bus operations
      - Pentium burst read
        - Transfers four memory words
        - Bus transaction consists of four memory read operations
  - Bus arbitration



Summary

- General Concepts of Computer Organization
  - Overview of Microcomputer
    - CPU, Memory, I/O
    - Instruction Execution Cycle
  - Central Processing Unit (CPU)
    - CISC vs. RISC
    - 6 Instruction Set Design Issues
  - How Hardwares Execute Processor's Instructions
    - Digital Logic Design (Combinational & Sequential Circuits)
    - Microprogrammed Control
  - Pipelining
    - 3 Hazards
    - 3 technologies for performance improvement
  - Memory
    - Data Alignment
    - 2 Design Issues (Cache, Virtual Memory)
  - I/O Devices