Computer Architecture and Logic Design

Cache Memory Mapping and Processor Architecture

Engr. Rimsha

Learning Objectives Last lectures

- Differentiate the term Computer Architecture and Computer Organisation
- Understand two important concepts of Abstraction and Hierarchy
- ISA vs. Microarchitecture
  - ISA :Agreed upon interface between software and hardware
  - Microarchitecture : Specific implementation of an ISA
- A computer can perform four basic functions:
  - Data processing
  - Data storage
  - Data movement
  - Control

Motivation of Study of Computer Architecture

- Hardware vs. Software Knowledge requirements?
  - Can develop better software if you understand the underlying hardware
  - Can design better hardware if you understand what software it will execute
  - Can design a better computing system if you understand both
- Why study Computer Architecture?
  - Enable Better Systems, Enable new applications, Enable better solutions to problems

Learning Objectives

- What are three ways of main memory mapping on cache memory?
- How is Direct Memory Mapped on Cache Memory?

Reference: William Stallings Book Chapter 4

Cache and Main Memory

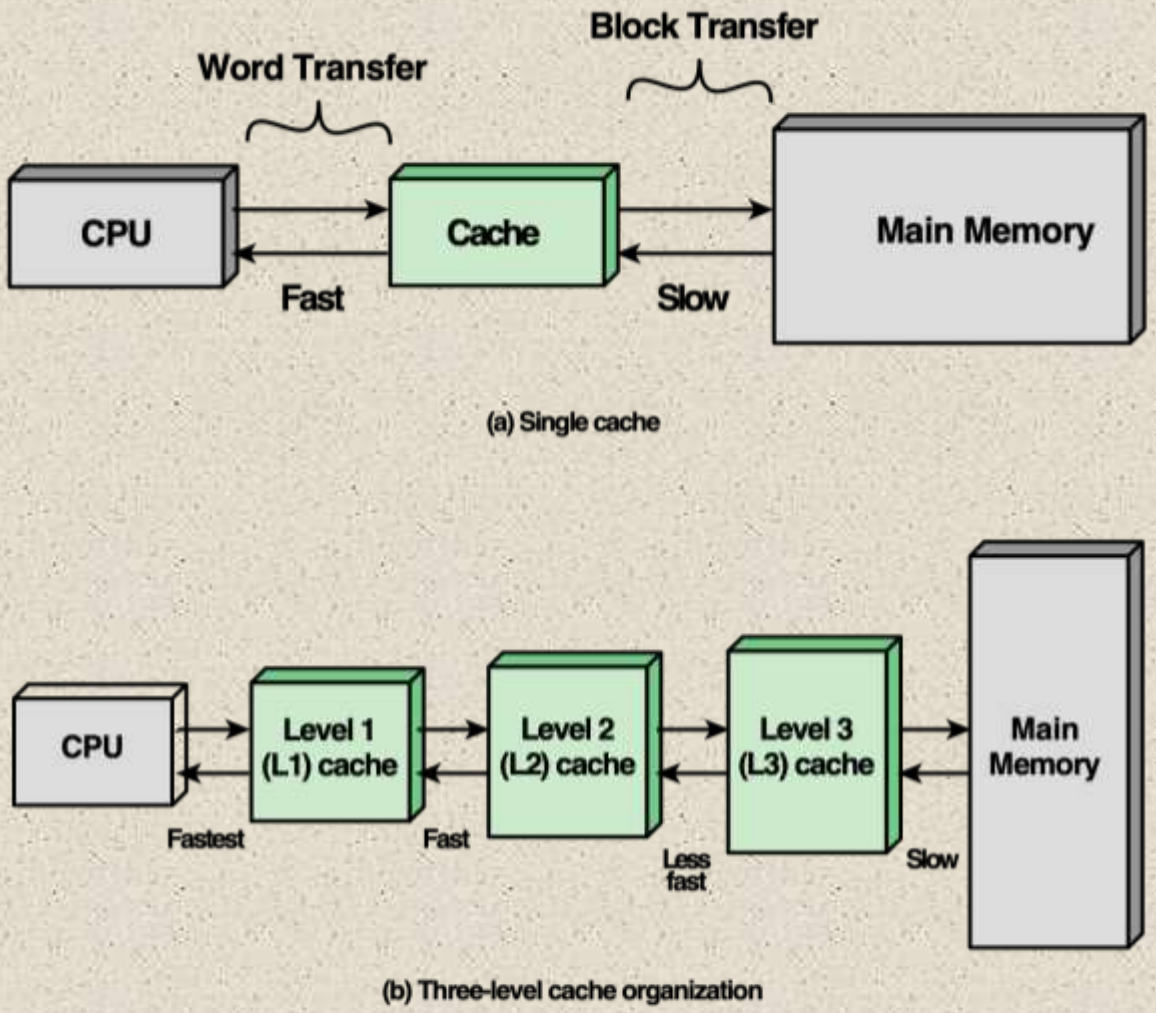
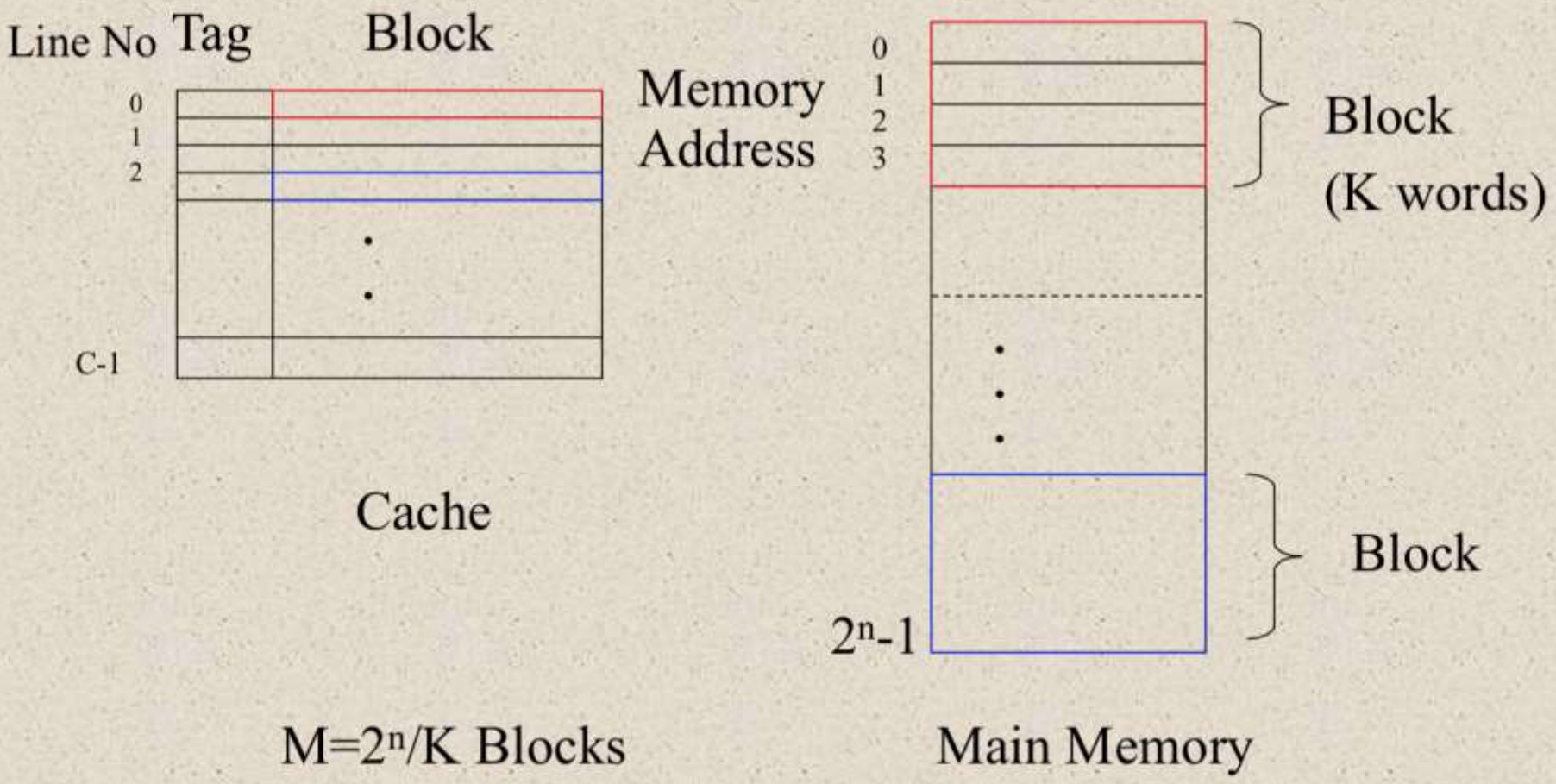


Figure 4.3 Cache and Main Memory

Cache/Main Memory Structure



- Mapping Function
  - Direct Mapping
  - Associative Mapping
  - Set Associative Mapping

Mapping Function

- Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines
- Three techniques can be used:

Direct

- The simplest technique
- Maps each block of main memory into only one possible cache line

Associative

- Permits each main memory block to be loaded into any line of the cache
- The cache control logic interprets a memory address simply as a Tag and a Word field
- To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's Tag for a match

Set Associative

- A compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages

Cache and Memory Sizes Calculations

- Main Memory Size = 16 MB
- Cache Size = 64 KB
- Block Size = 4 Bytes
- How many lines for Cache memory?
  - 64 KB /4 B=16 K lines
  - How many address lines for accessing cache memory lines?
    - 2^{14} =16 K, address lines for Cache is 14
- How many Blocks in main memory?
  - 16 MB/4 B =4 Mega blocks
- How many address lines for accessing Main memory?
  - 2^{24} =16 M, address lines for main memory is 24

Cache and Memory Sizes Calculations

- Main Memory Size = 32B
- Cache Size = 16B
- Block Size = 4 Bytes
- How many lines for Cache memory?
  - 16 B /4 B=4 lines
  - How many address lines for accessing cache memory lines?
    - 2^2 =4, address lines for Cache is 2
- How many Blocks in main memory?
  - 32 B/4 B =8 blocks
- How many address lines for accessing Main memory?
  - 2^5 =32 , address lines for main memory is 5

- Main Memory Size = 32B
- Cache Size = 16B
- Block Size = 4 Bytes

Mapping Example

Main Memory Size	32 B					Address Distribution			Block No	Memory Address in Binary	Memory Address in Hex	Memory Address in Decimal	Memory Word
Cache Size	16B												
Block Size	4B	Line Number	Tag	Line/Block	Possible Blocks	Tag	Line	Word	Tag+Line				
		0			B0, B4	0	00	00	000	00000	0	0	
		1			B1, B5	0	00	01	000	00001	1	1	
		2			B2, B6	0	00	10	000	00010	2	2	
		3			B3, B7	0	00	11	000	00011	3	3	
						0	01	00	001	00100	4	4	
						0	01	01	001	00101	5	5	
						0	01	10	001	00110	6	6	
						0	01	11	001	00111	7	7	
						0	10	00	010	01000	8	8	
						0	10	01	010	01001	9	9	
						0	10	10	010	01010	A	10	
						0	10	11	010	01011	B	11	
						0	11	00	011	01100	C	12	
						0	11	01	011	01101	D	13	
						0	11	10	011	01110	E	14	
						0	11	11	011	01111	F	15	
						1	00	00	100	10000	10	16	
						1	00	01	100	10001	11	17	
						1	00	10	100	10010	12	18	
						1	00	11	100	10011	13	19	
						1	01	00	101	10100	14	20	
						1	01	01	101	10101	15	21	
						1	01	10	101	10110	16	22	
						1	01	11	101	10111	17	23	
						1	10	00	110	11000	18	24	
						1	10	01	110	11001	19	25	
						1	10	10	110	11010	1A	26	
						1	10	11	110	11011	1B	27	
						1	11	00	111	11100	1C	28	
						1	11	01	111	11101	1D	29	
						1	11	10	111	11110	1E	30	
						1	11	11	111	11111	1F	31	

Direct Mapping Cache Organization

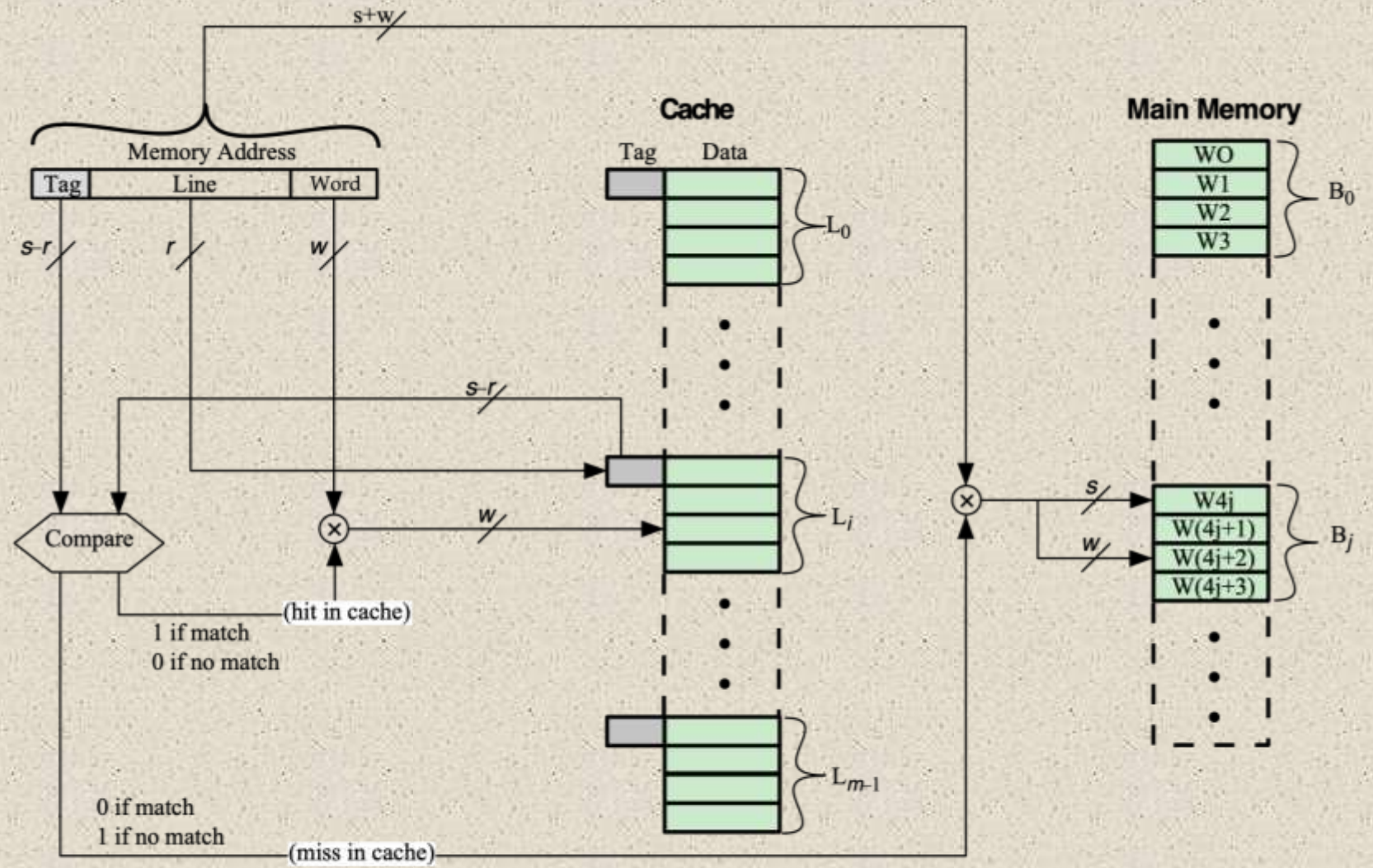


Figure 4.9 Direct-Mapping Cache Organization

Direct Mapping Example

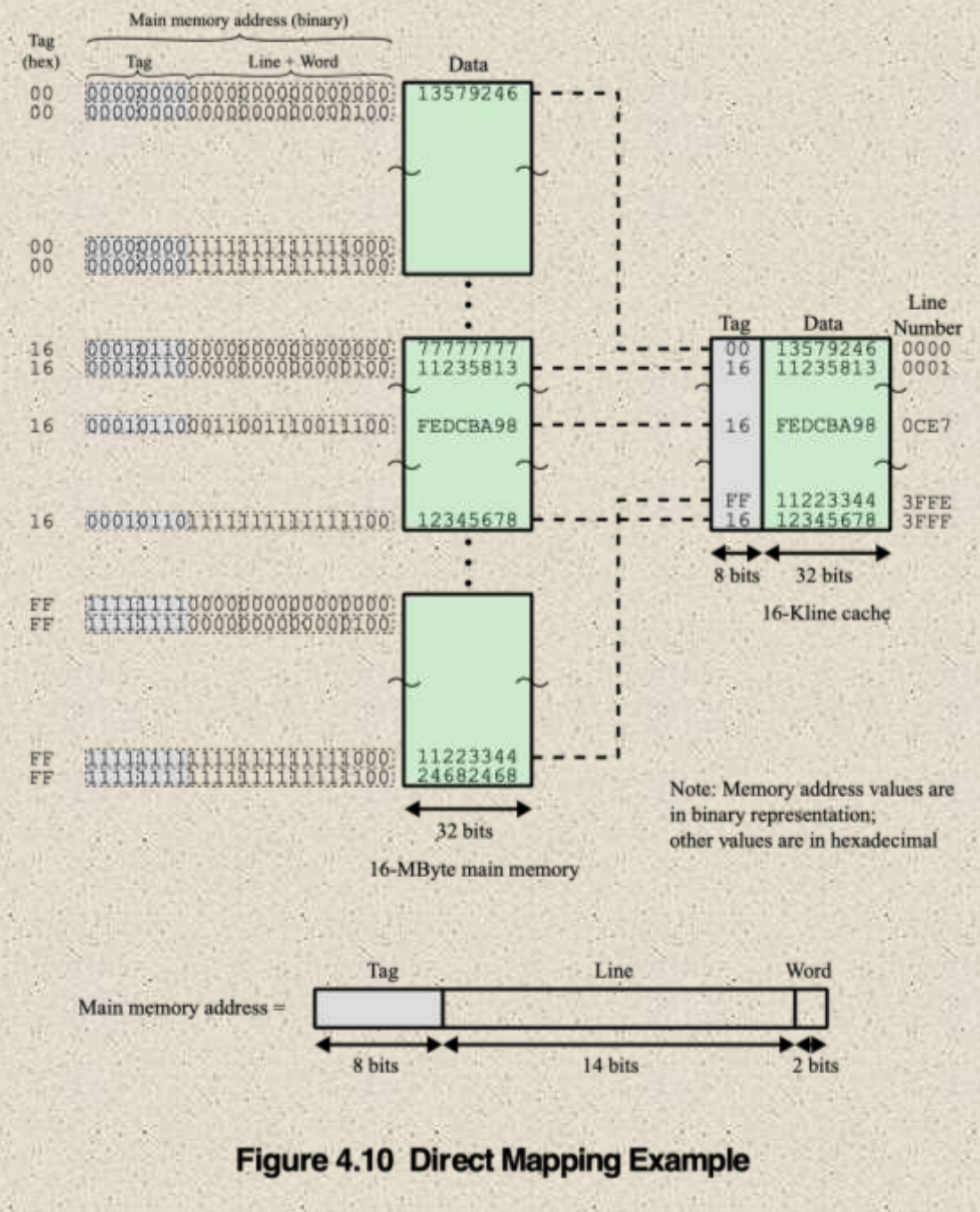


Figure 4.10 Direct Mapping Example

Direct Mapping Summary

$$i = j \text{ modulo } m$$

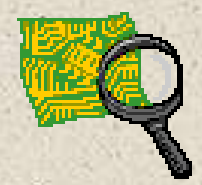
where

i = cache line number

j = main memory block number

m = number of lines in the cache

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w} = 2^{s}$
- Number of lines in cache = m = 2^{r}
- Size of tag = (s – r) bits



Direct Mapping

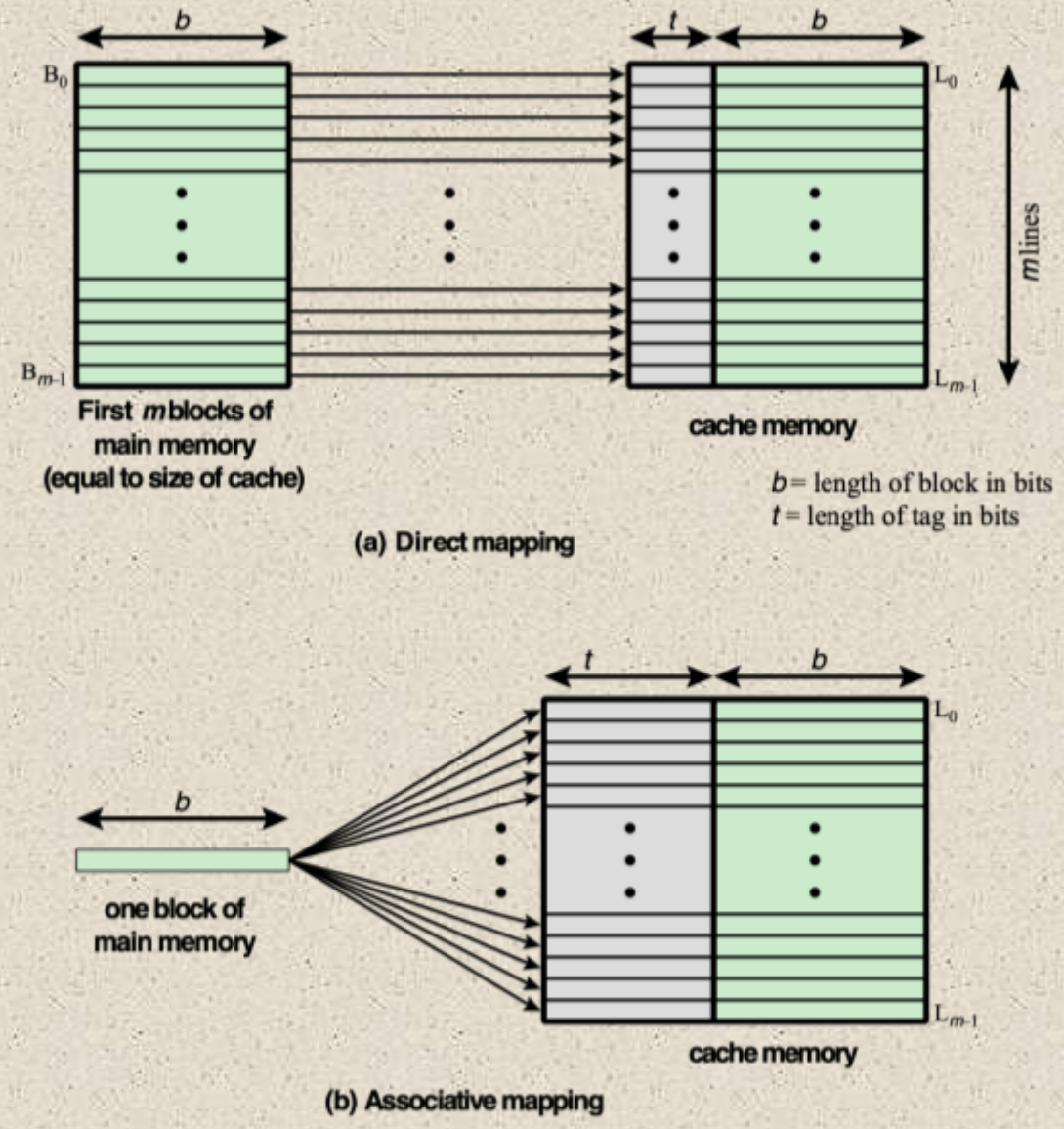
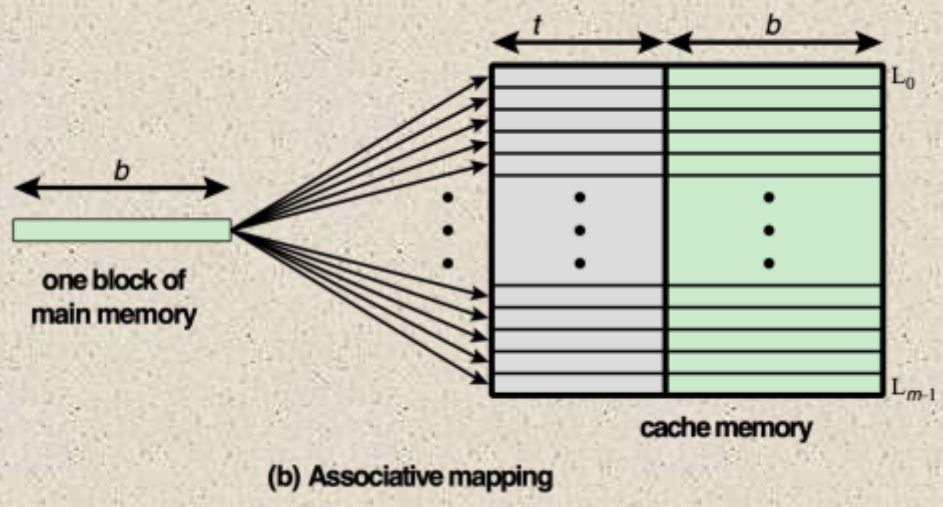


Figure 4.8 Mapping From Main Memory to Cache: Direct and Associative

Associative Mapping



(b) Associative mapping

Fully Associative Cache Organization

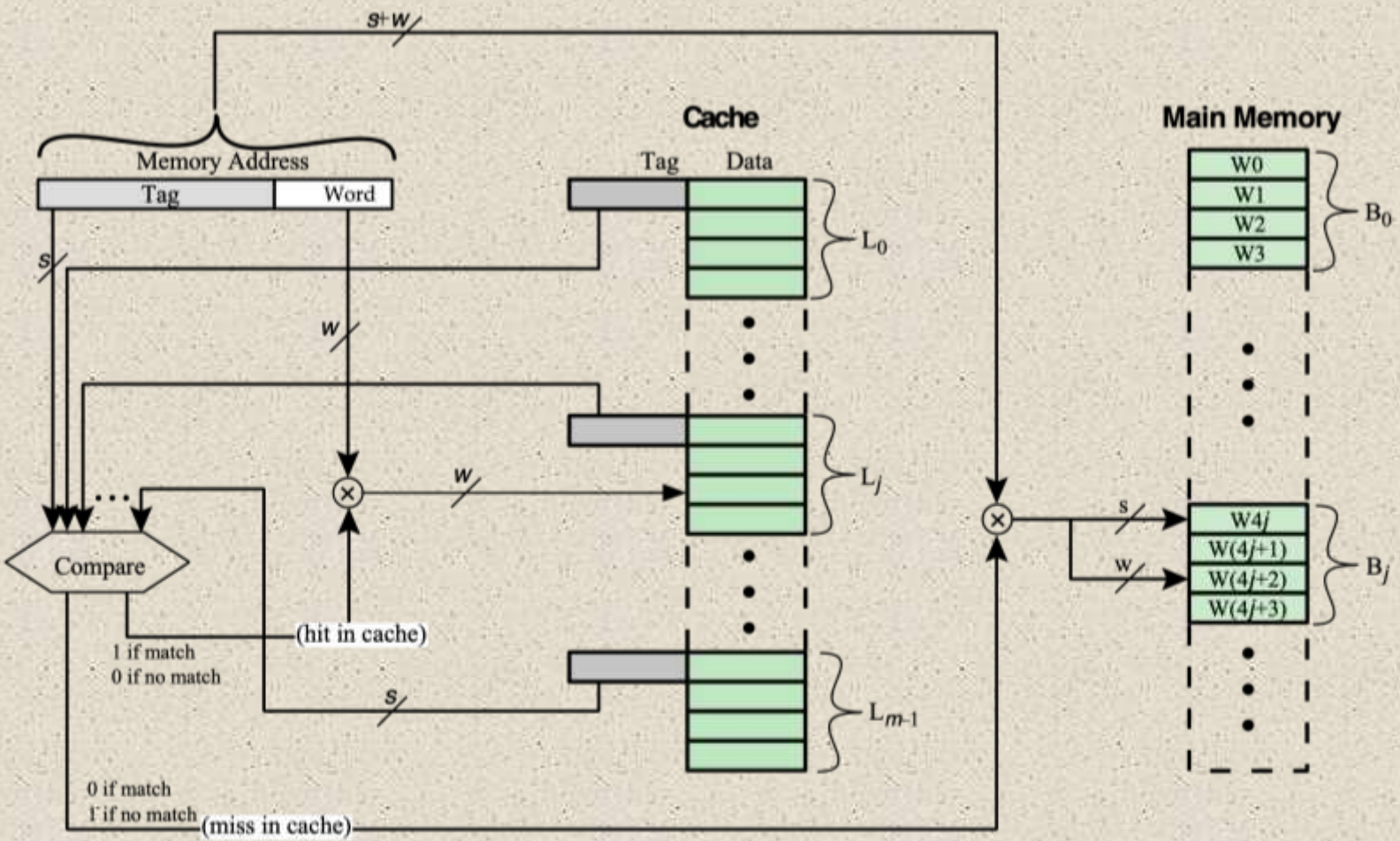


Figure 4.11 Fully Associative Cache Organization

Associative Mapping Example

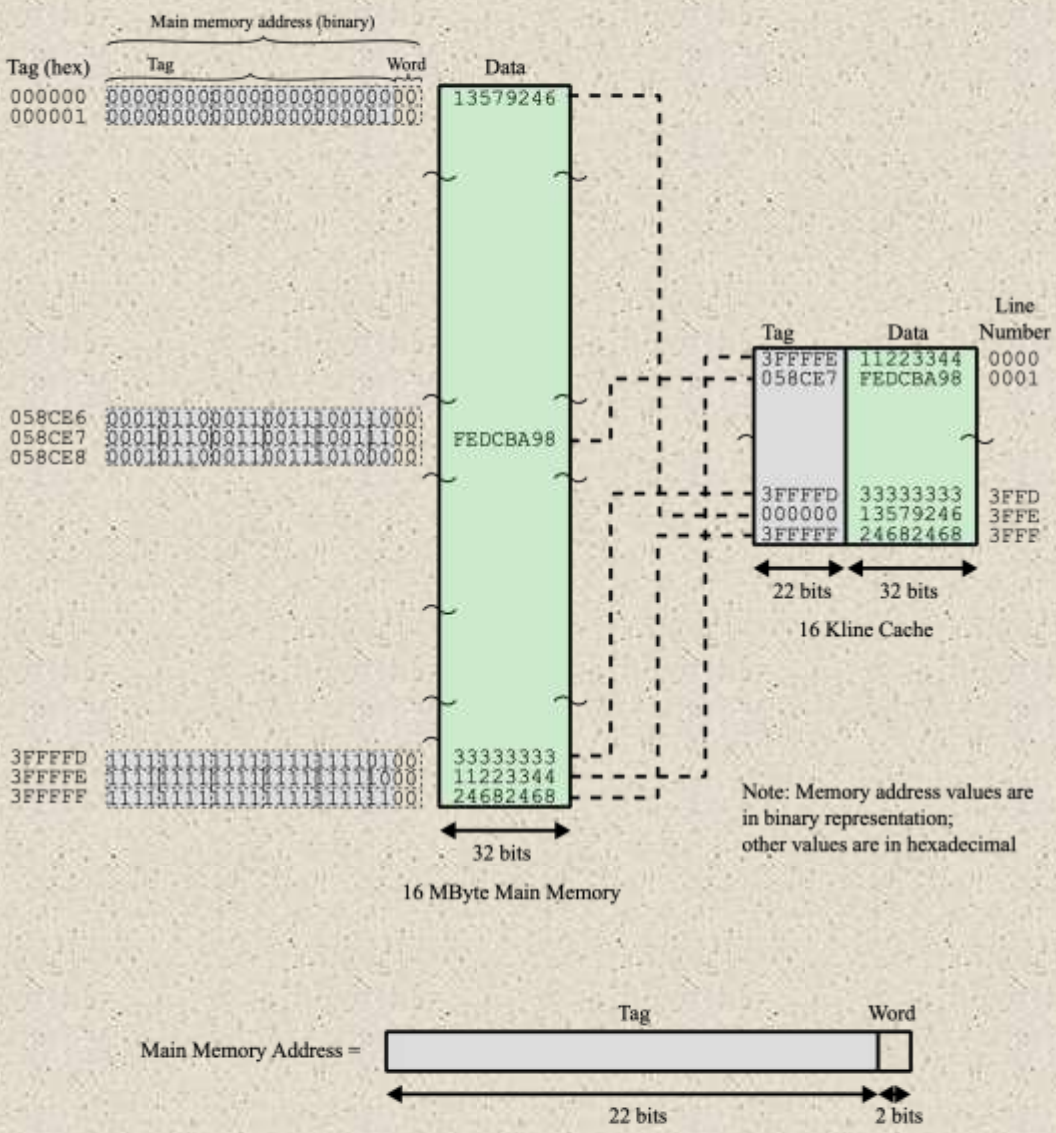
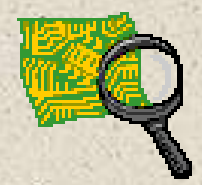


Figure 4.12 Associative Mapping Example

Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w} = 2^{s}$
- Number of lines in cache = undetermined
- Size of tag = s bits



Set Associative Mapping

- Compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages
- Cache consists of a number of sets
- Each set contains a number of lines
- A given block maps to any line in a given set
- e.g. 2 lines per set
  - 2 way associative mapping
  - A given block can be in one of 2 lines in only one set

Set Associative Mapping

In this case, the cache consists of a number sets, each of which consists of a number of lines. The relationships are

$$m = v \times k$$
$$i = j \text{ modulo } v$$

where

i = cache set number
j = main memory block number
m = number of lines in the cache
v = number of sets
k = number of lines in each set

Mapping From Main Memory to Cache:

k-Way Set Associative

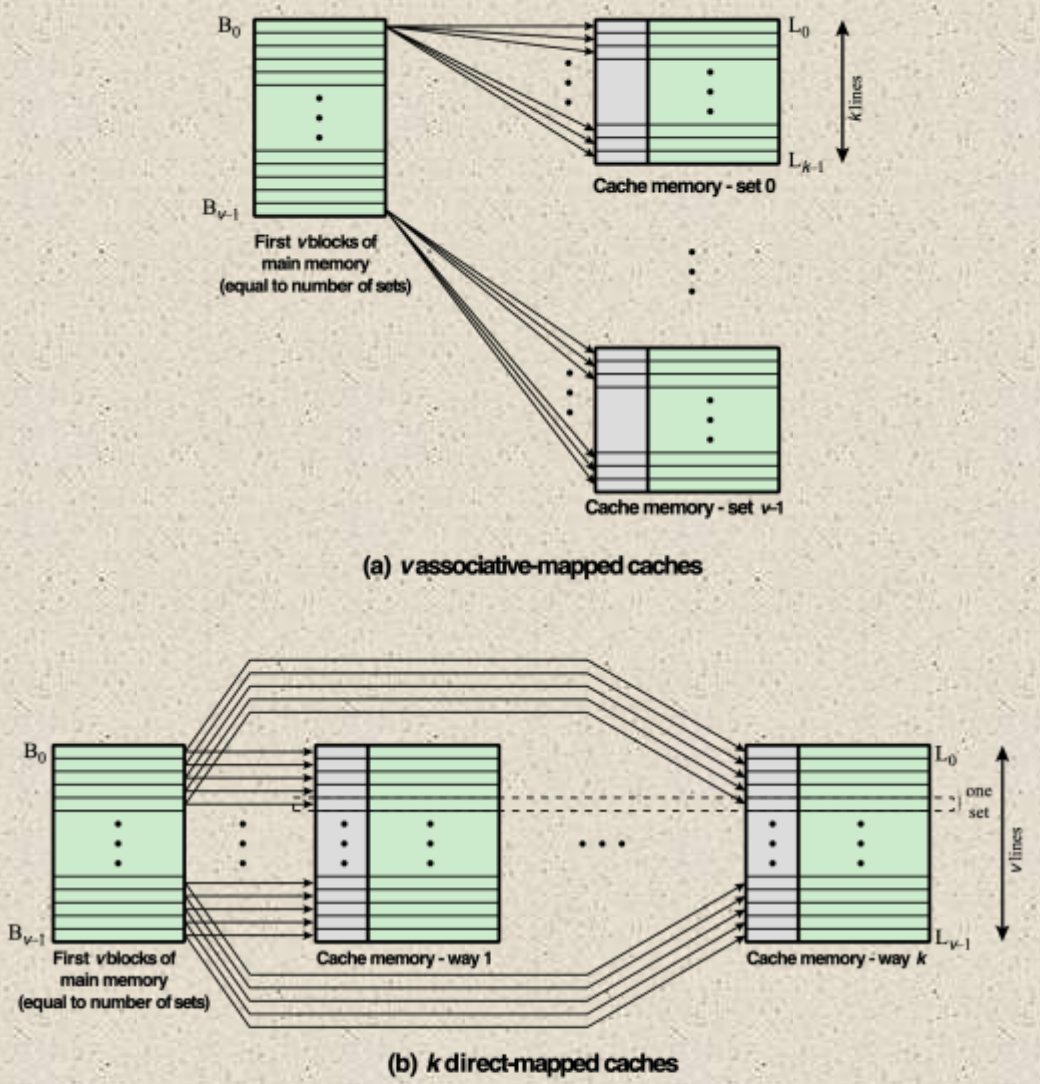


Figure 4.13 Mapping From Main Memory to Cache: *k*-way Set Associative

k-Way Set Associative Cache Organization

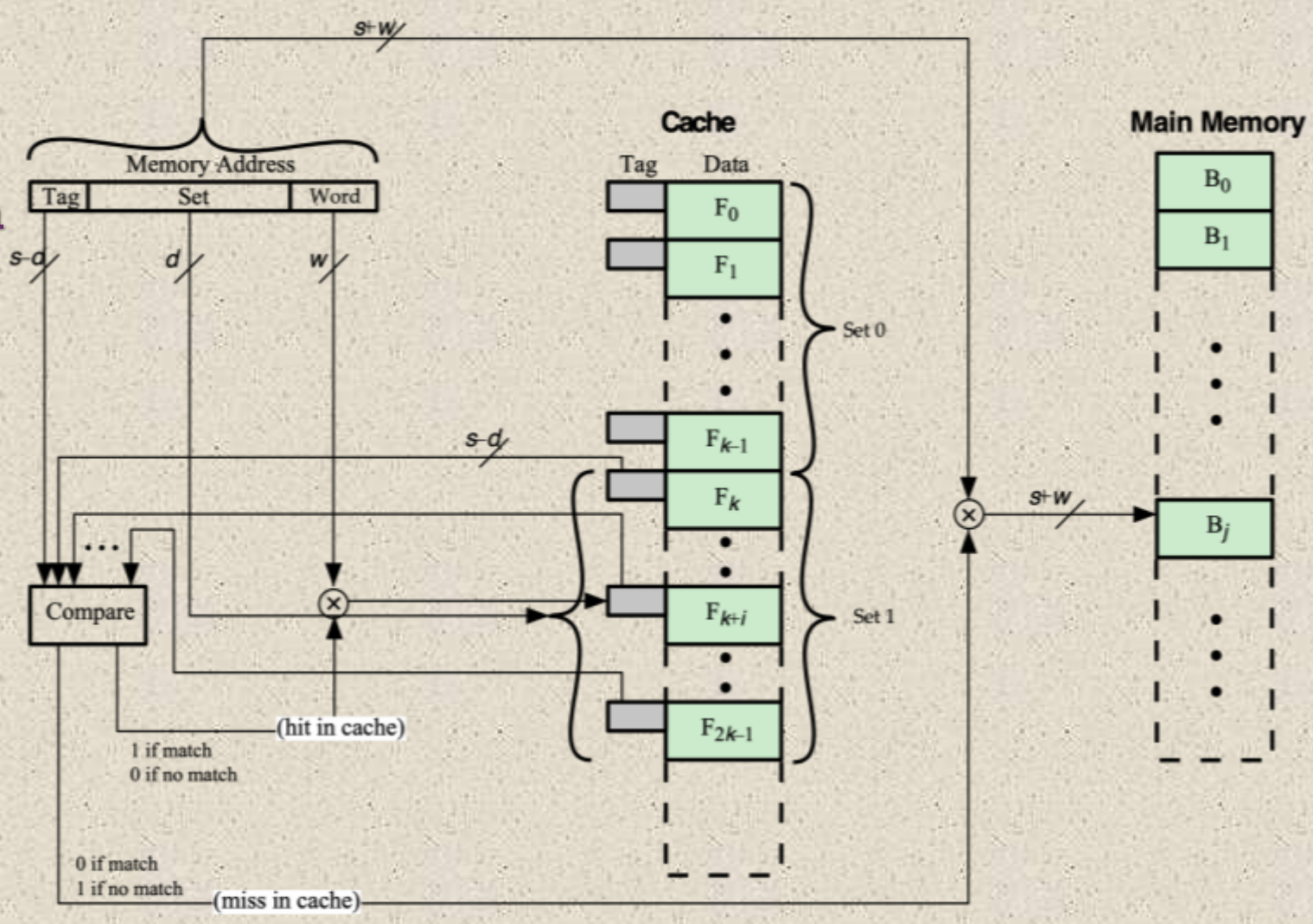
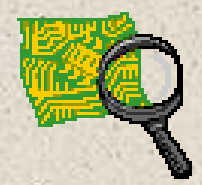


Figure 4.14 *k*-Way Set Associative Cache Organization

Set Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w}=2^{s}$
- Number of lines in set = k
- Number of sets = v = 2^{d}
- Number of lines in cache = m=kv = k * 2^{d}
- Size of cache = $k * 2^{d+w}$ words or bytes
- Size of tag = (s – d) bits



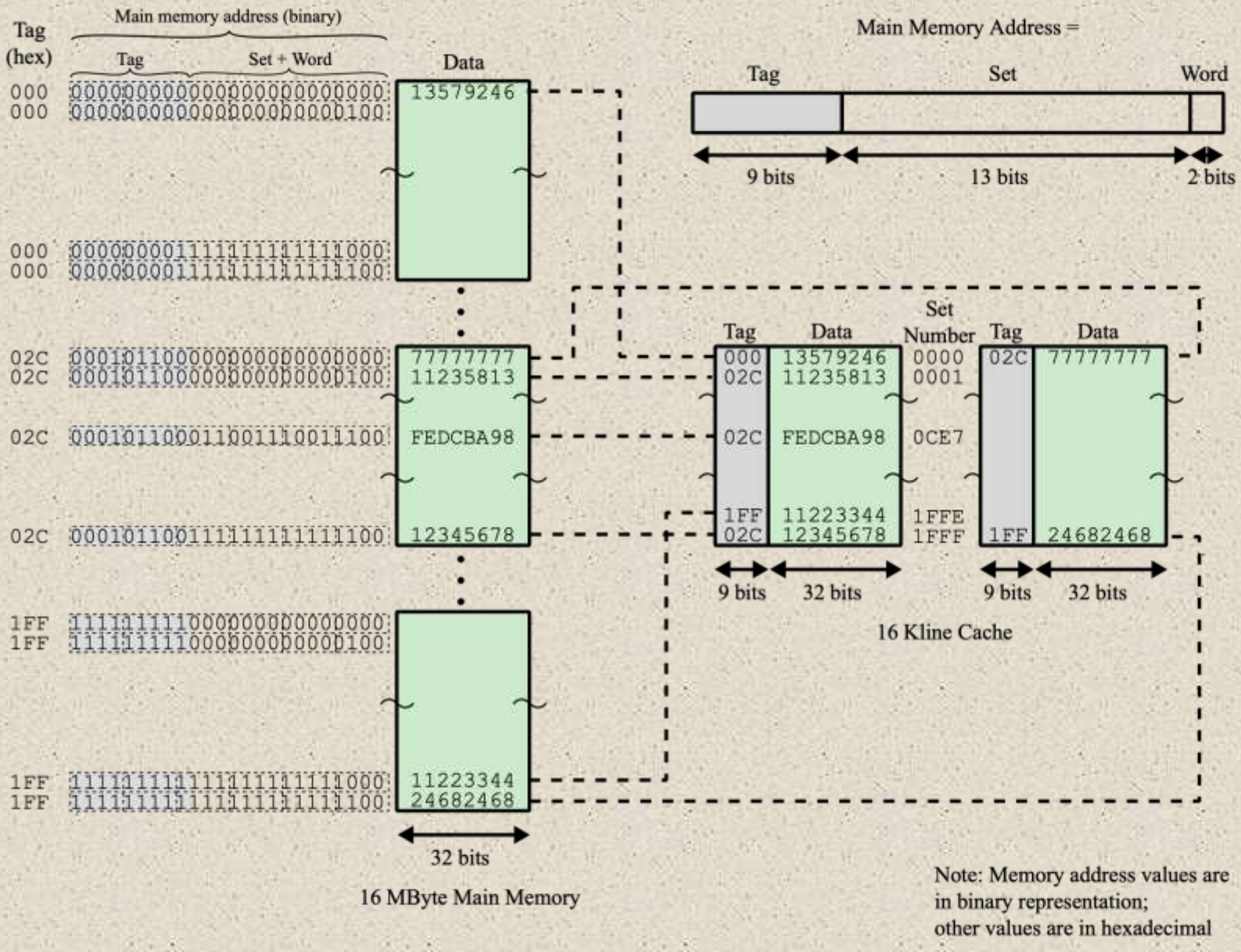


Figure 4.15 Two-Way Set Associative Mapping Example

Varying Associativity Over Cache Size

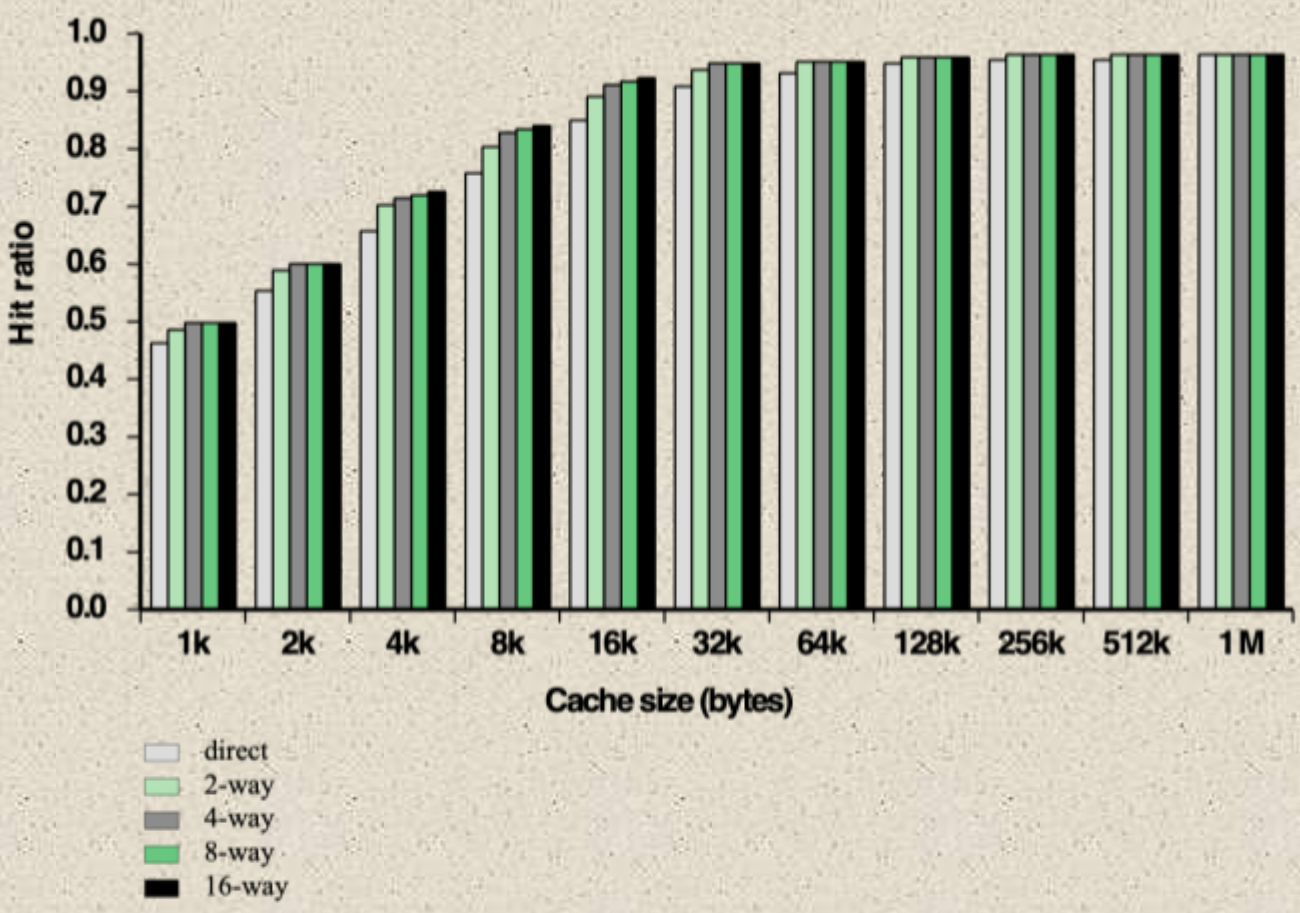
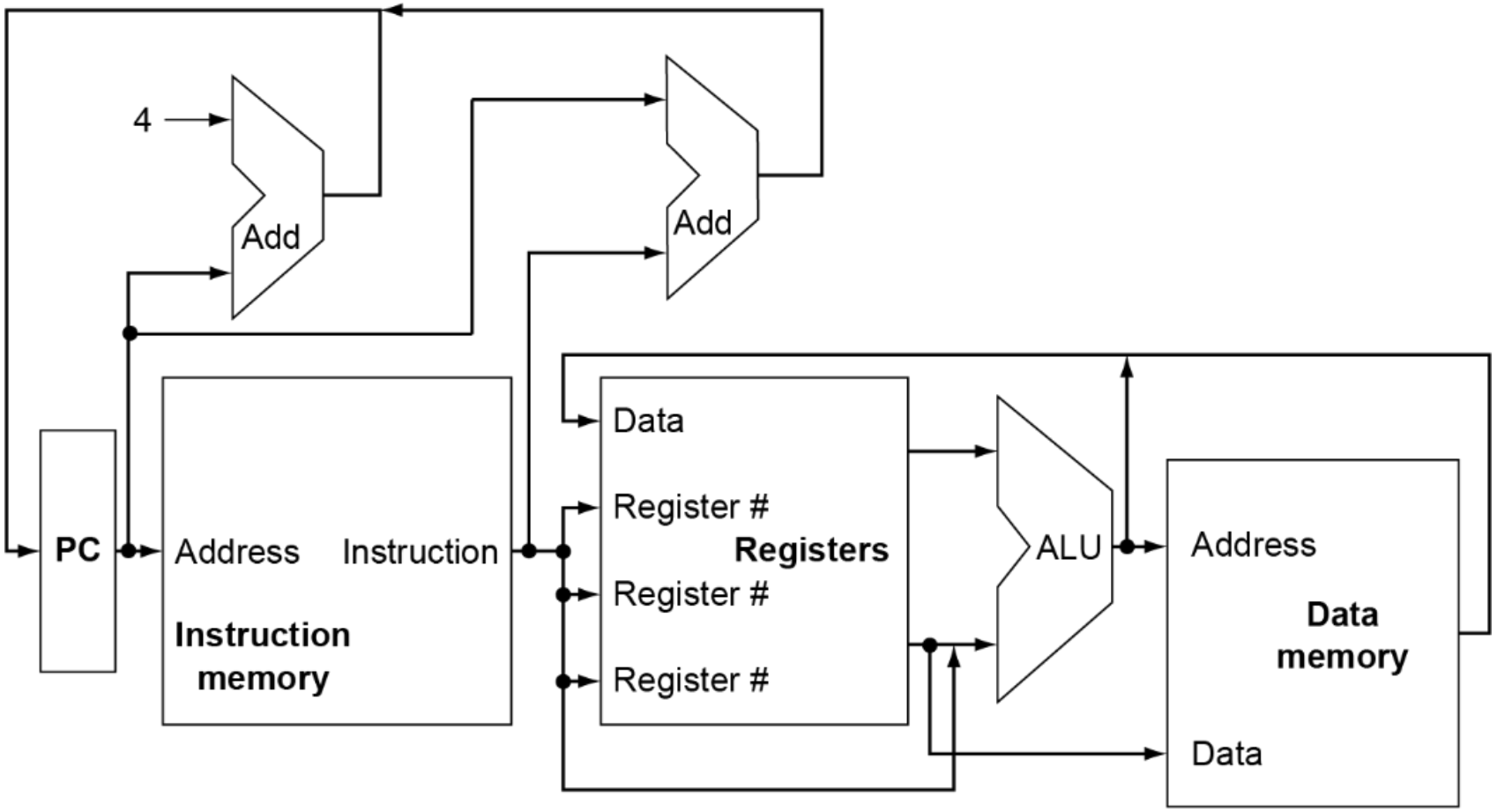
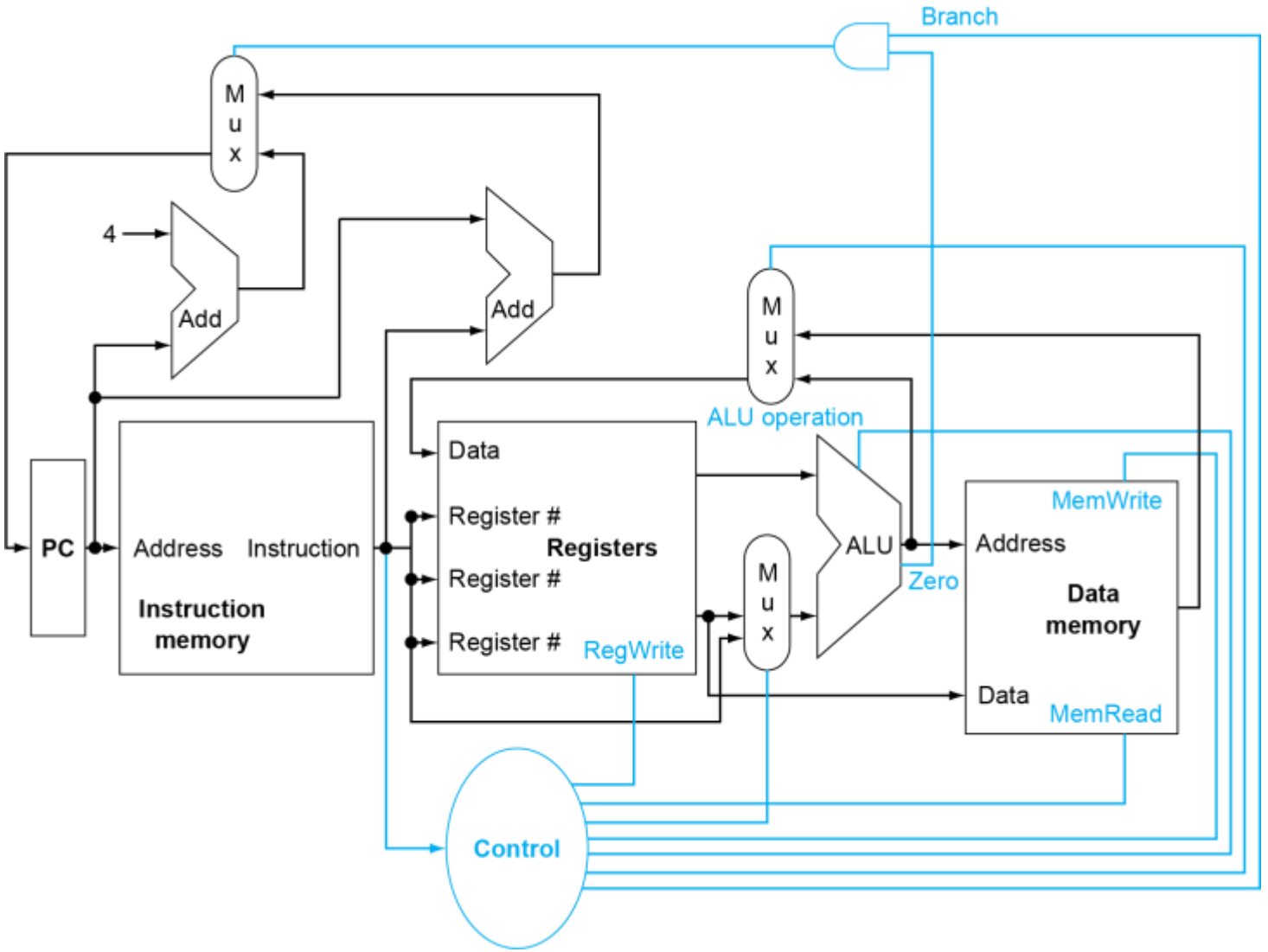


Figure 4.16 Varying Associativity over Cache Size

CPU Overview



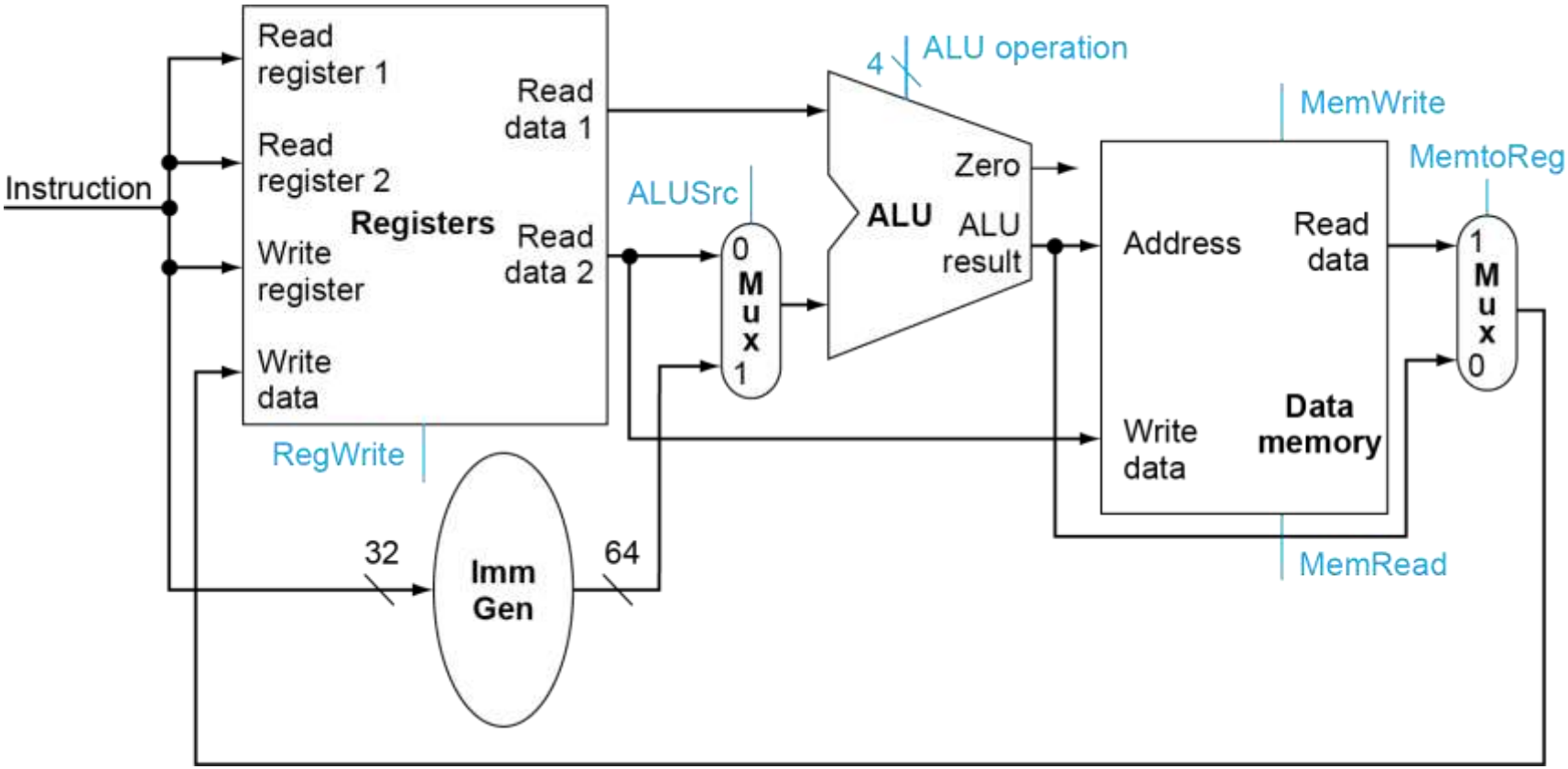
Control


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

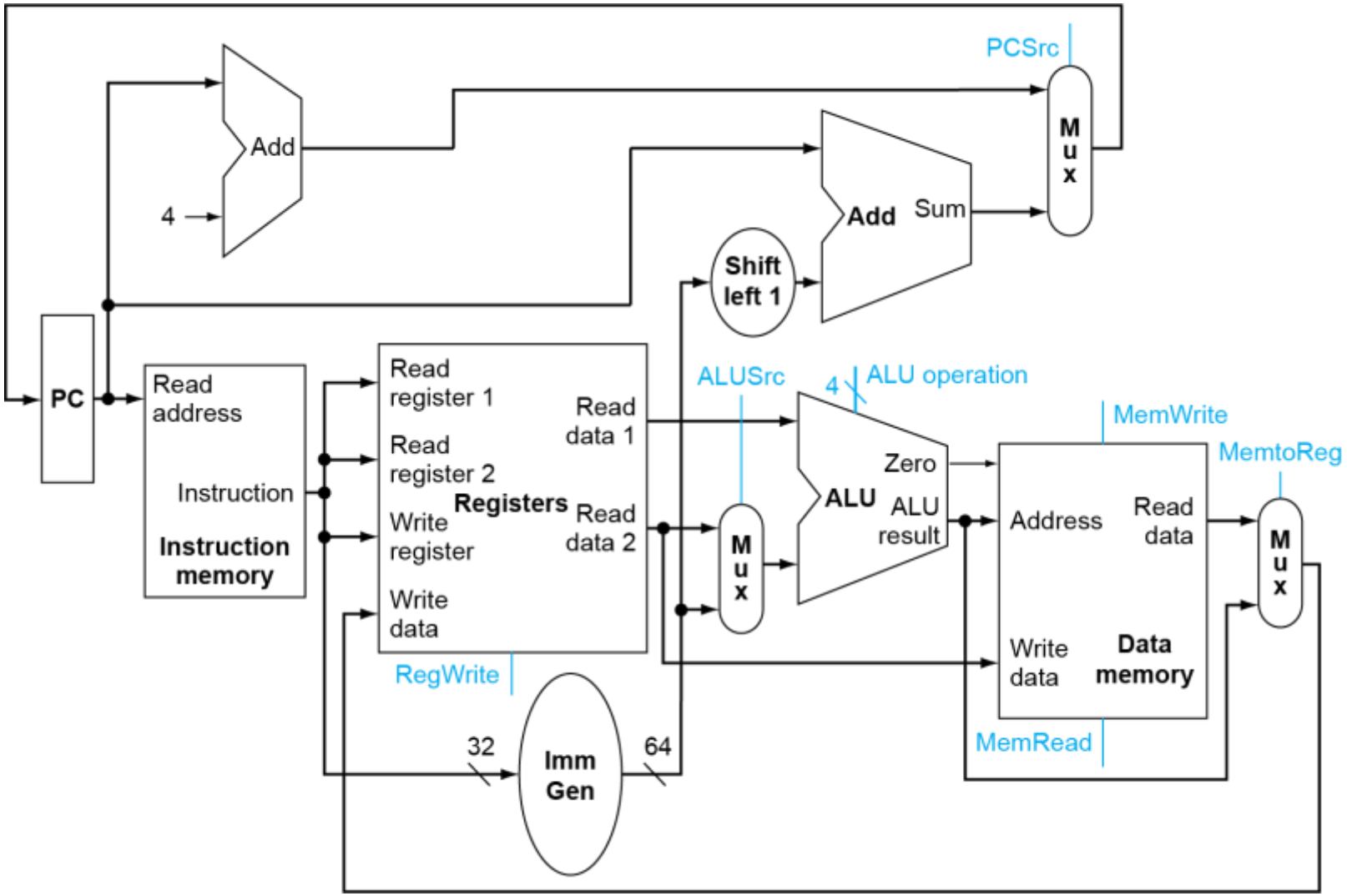
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

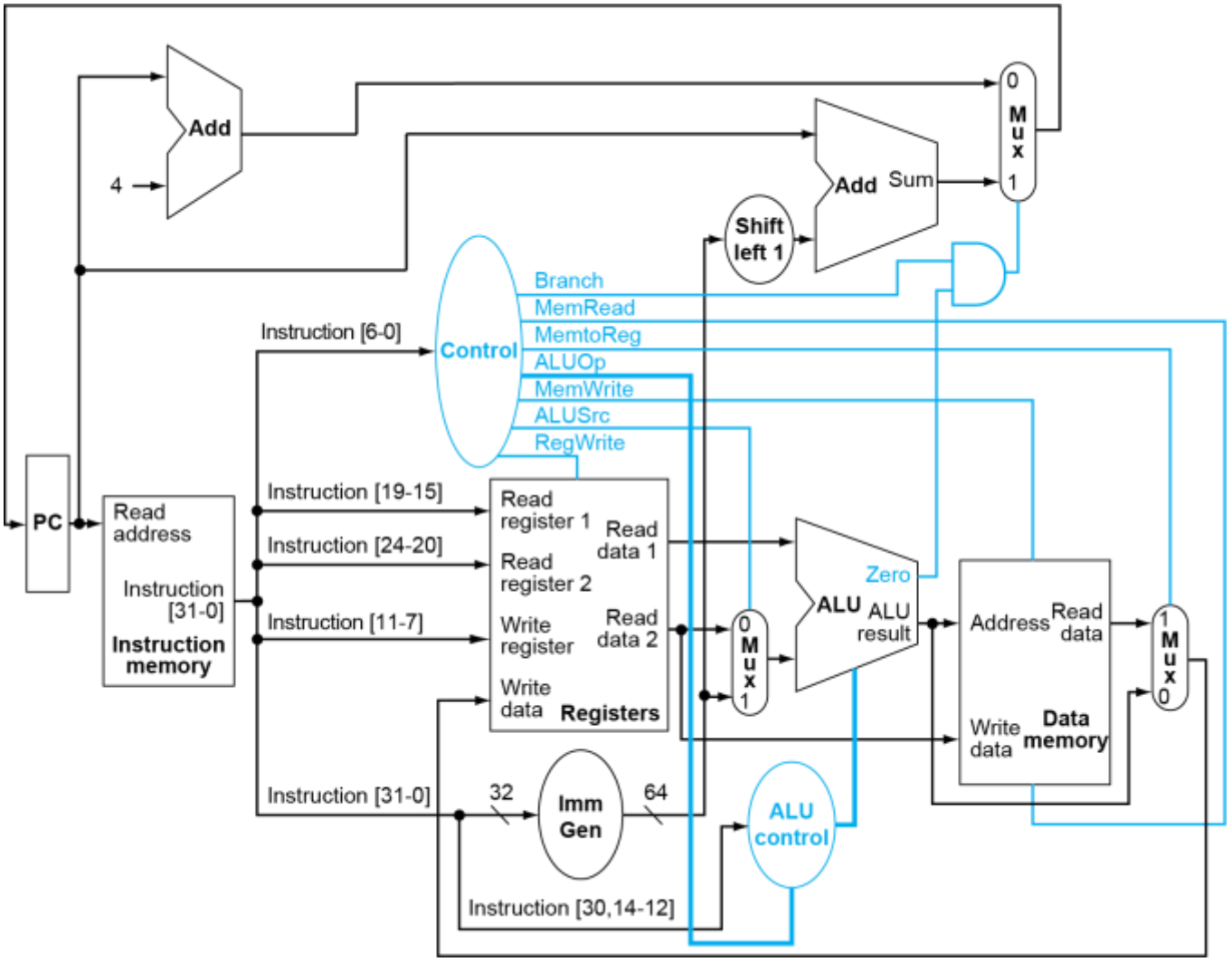
The Main Control Unit

- Control signals derived from instruction

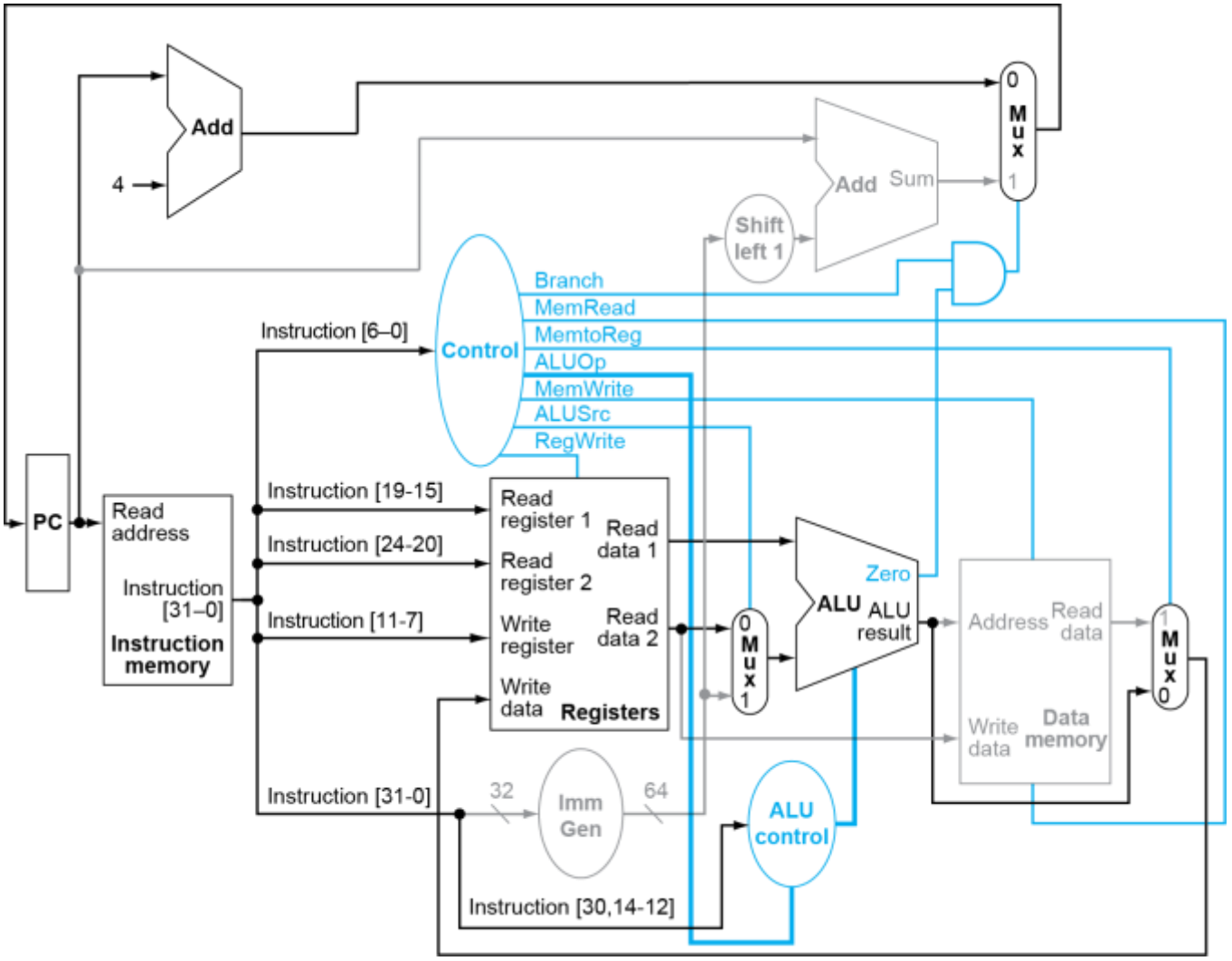
Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

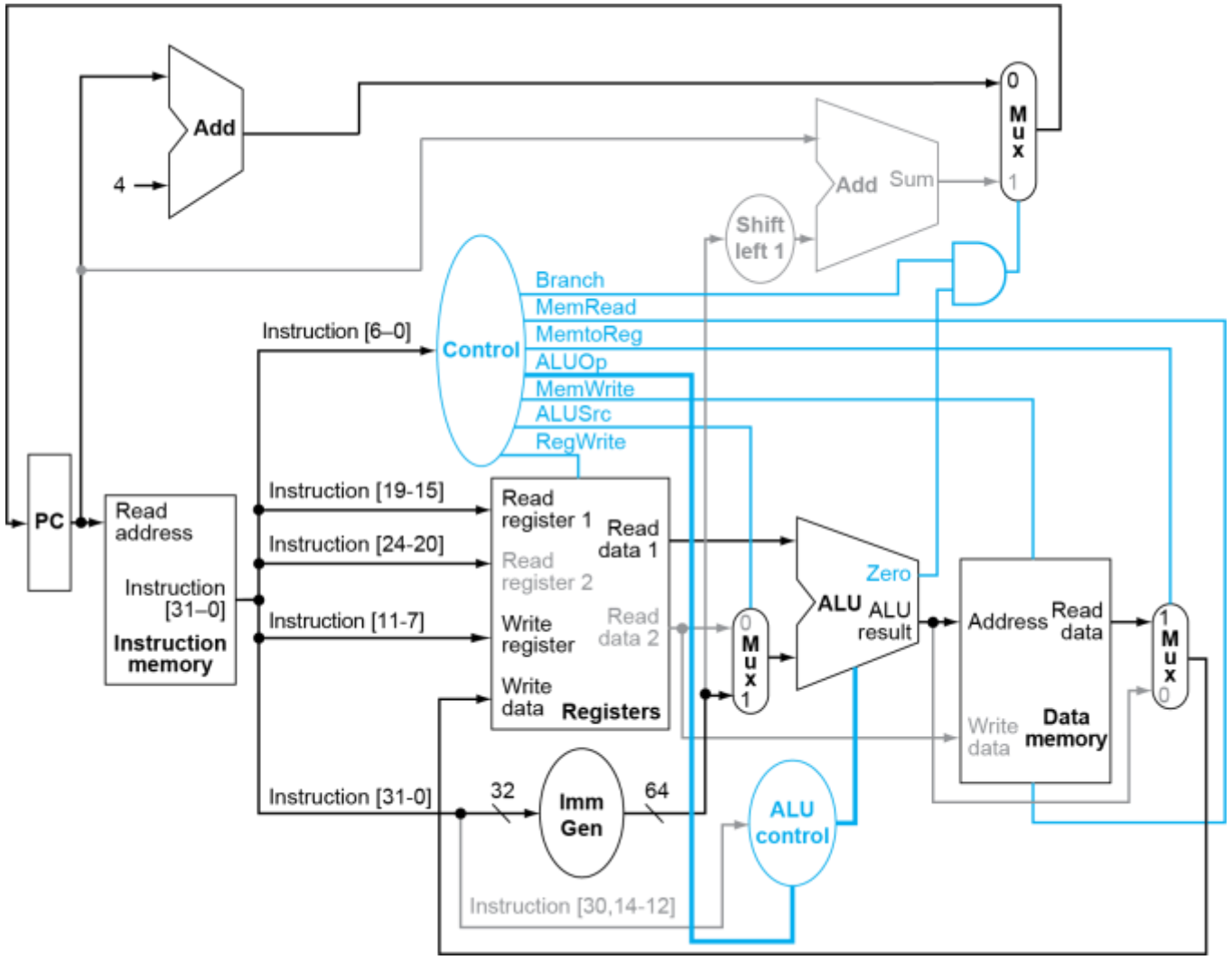
Datapath With Control



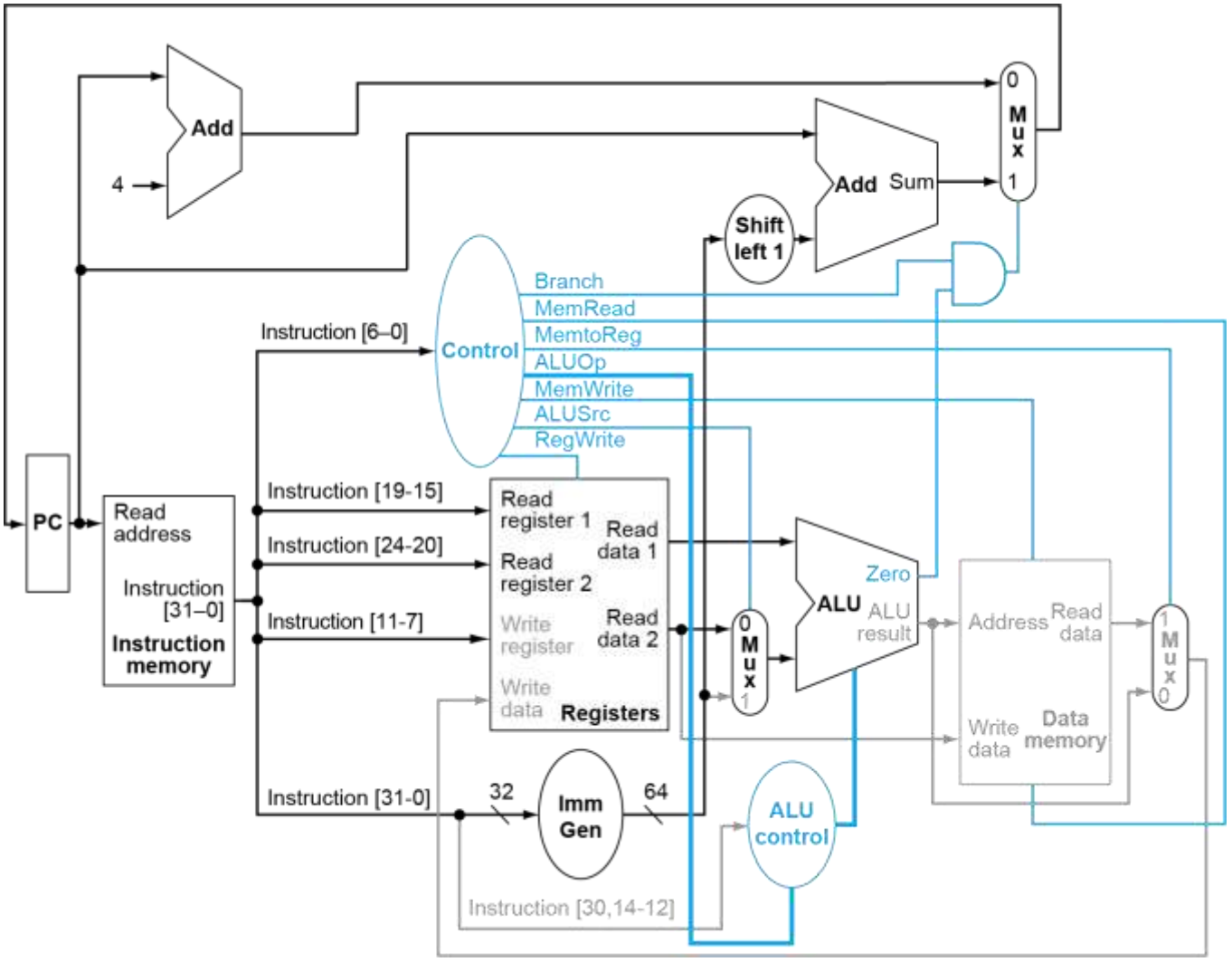
R-Type Instruction



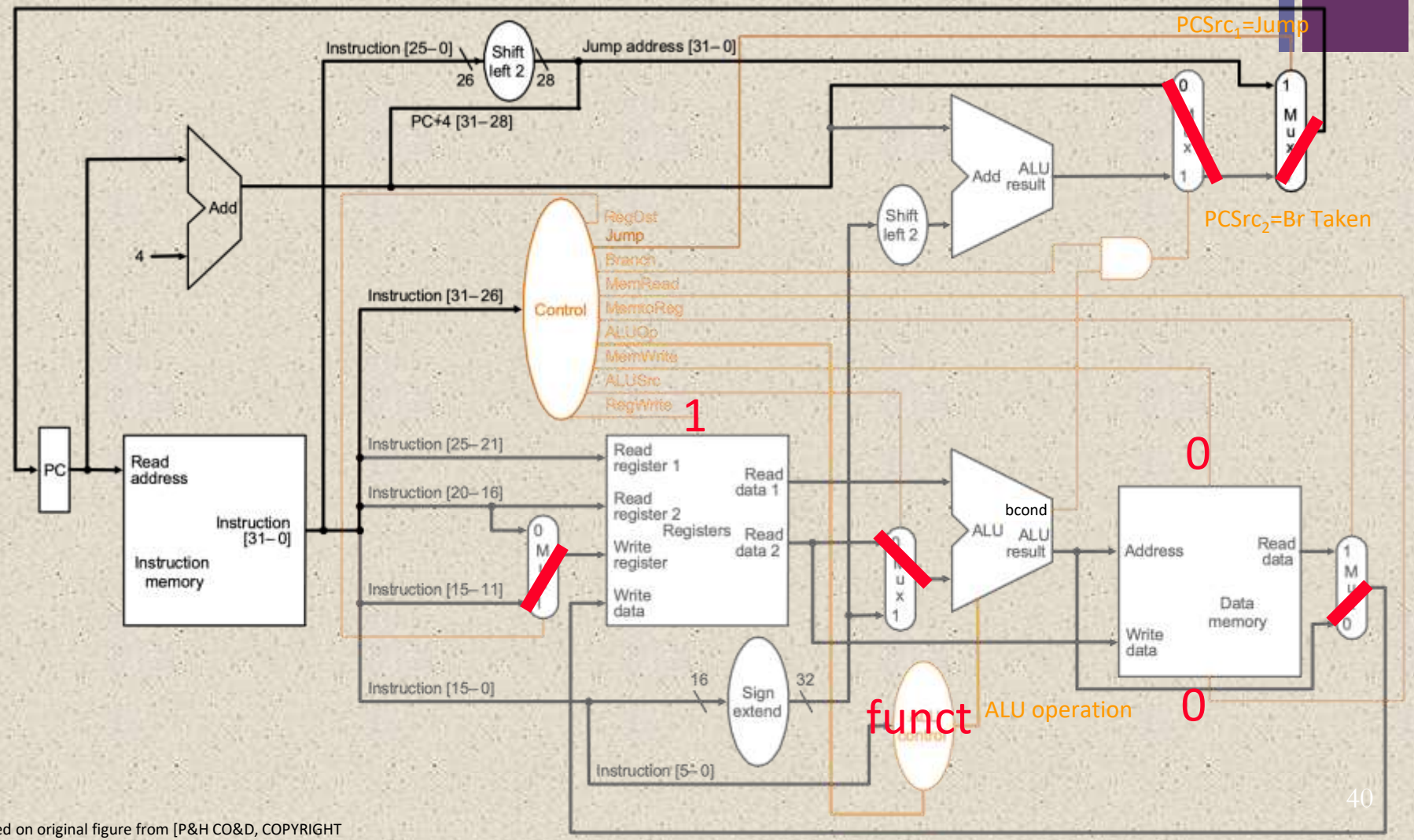
Load Instruction



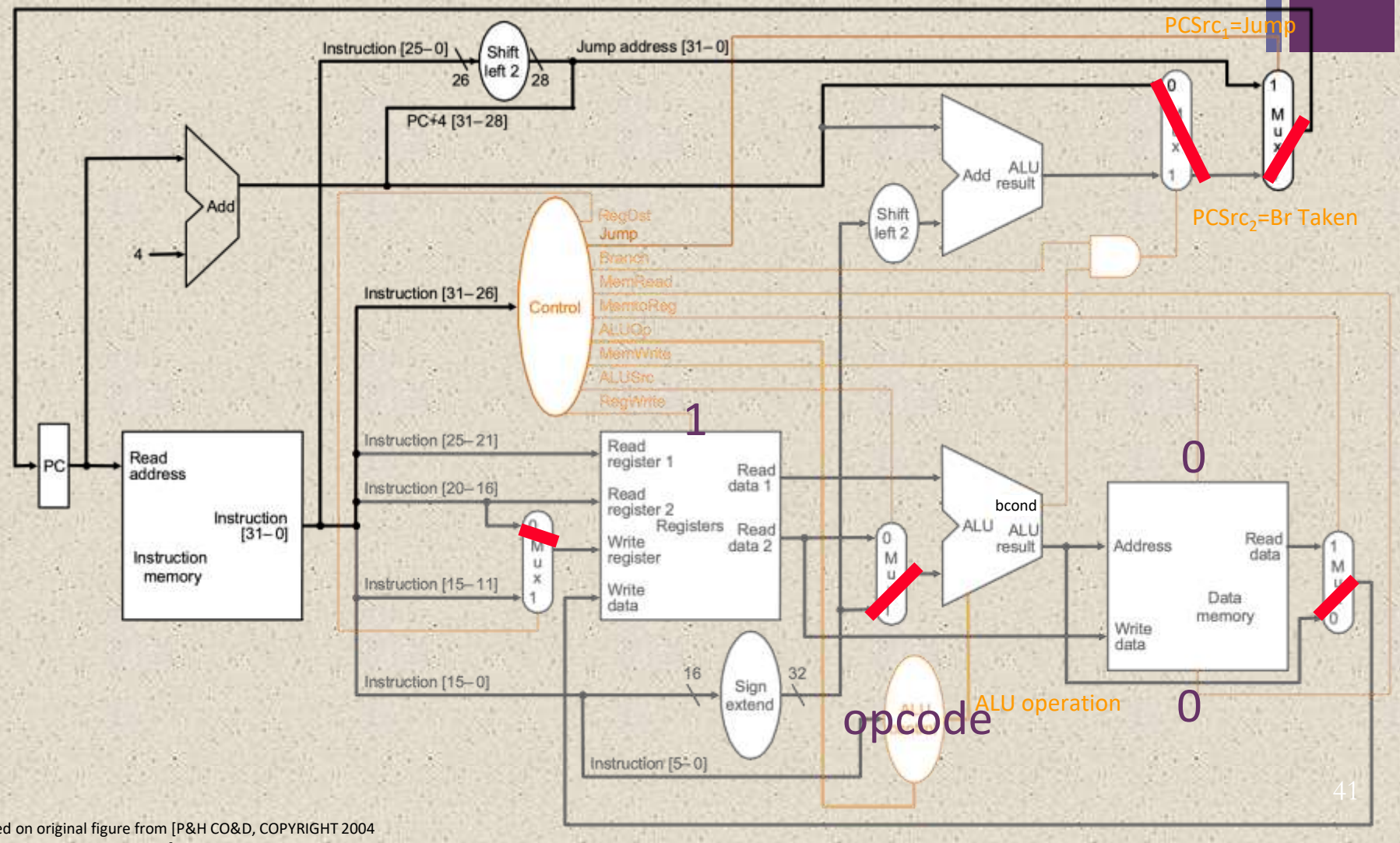
BEQ Instruction



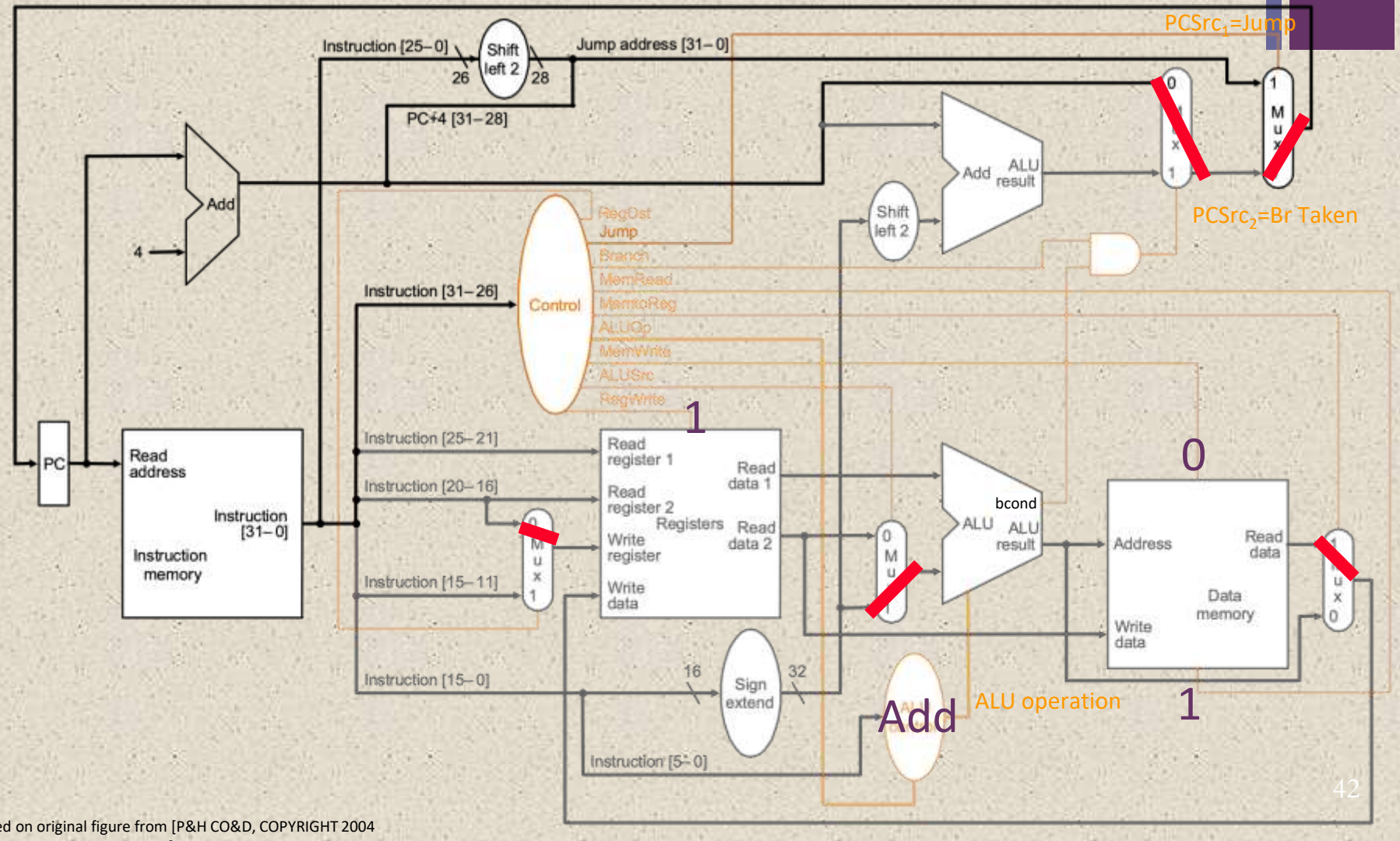
R-Type ALU



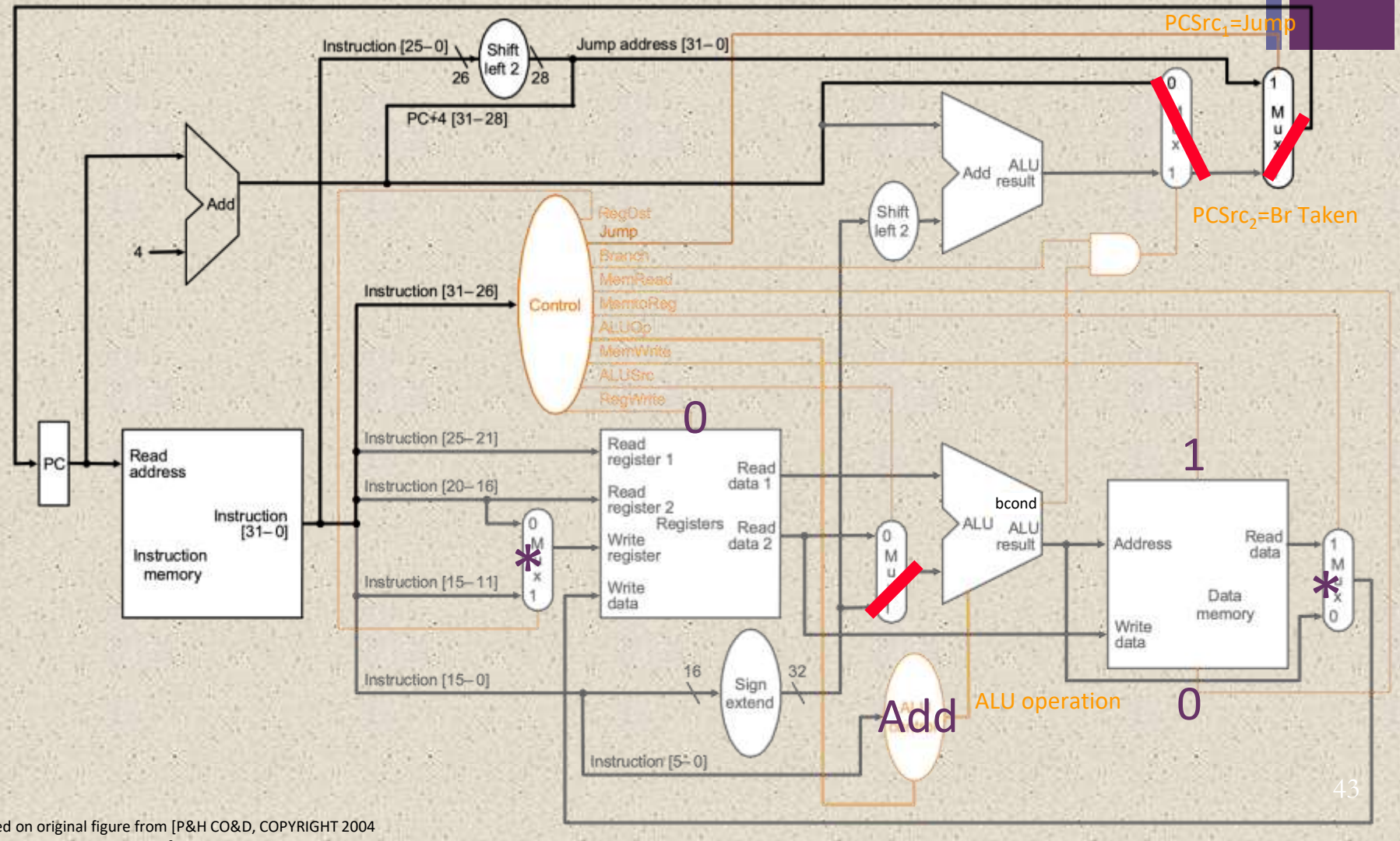
I-Type ALU



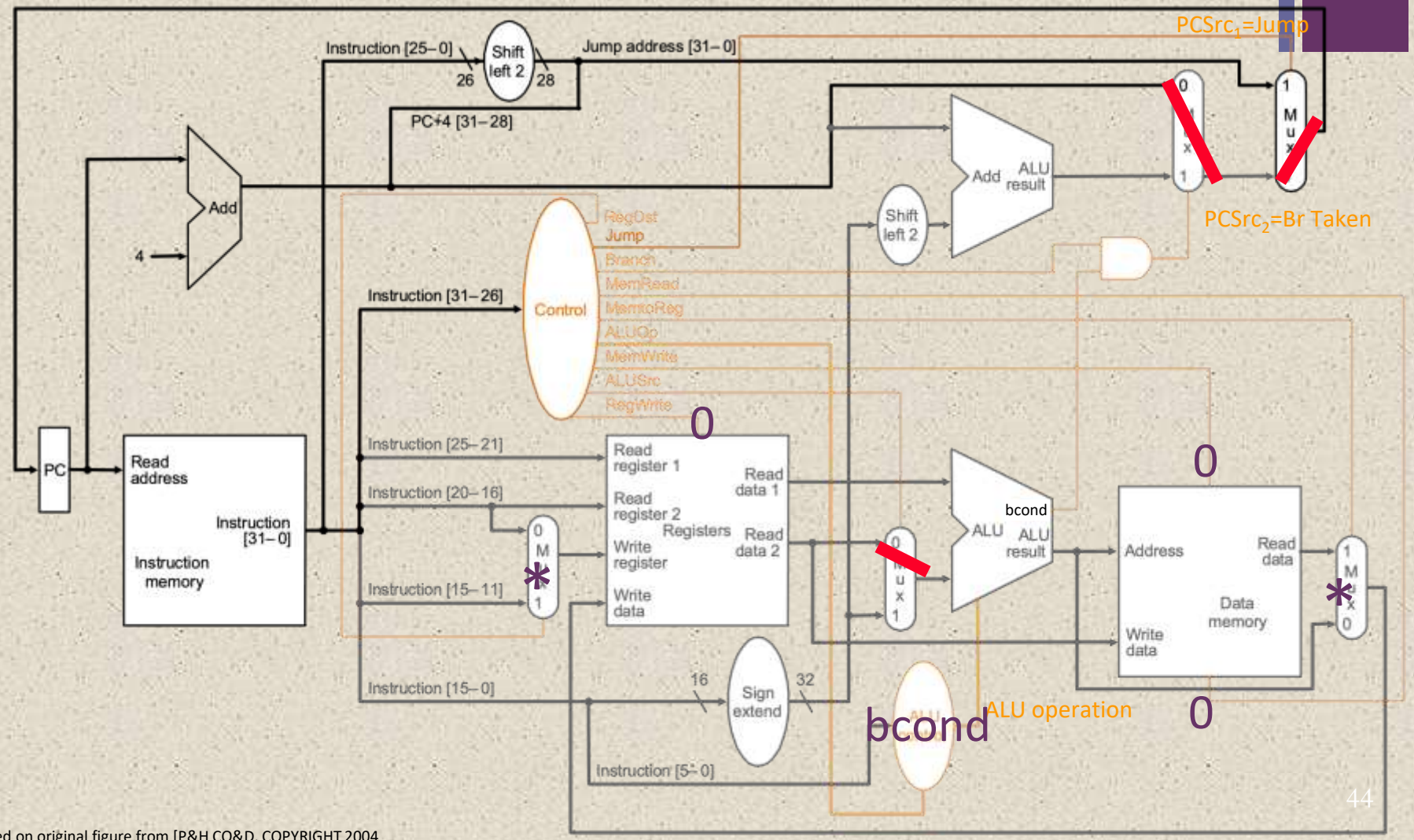
LW



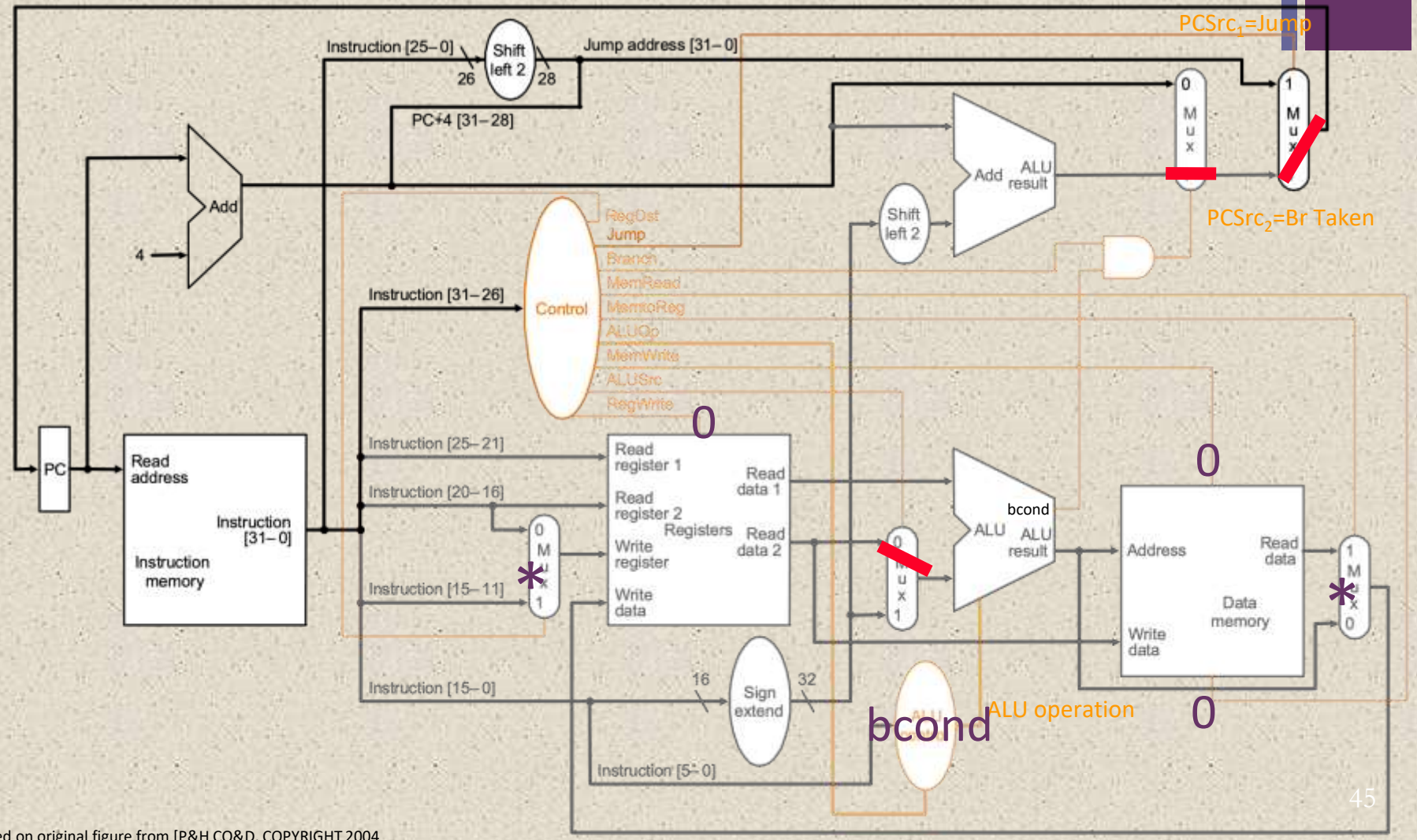
SW



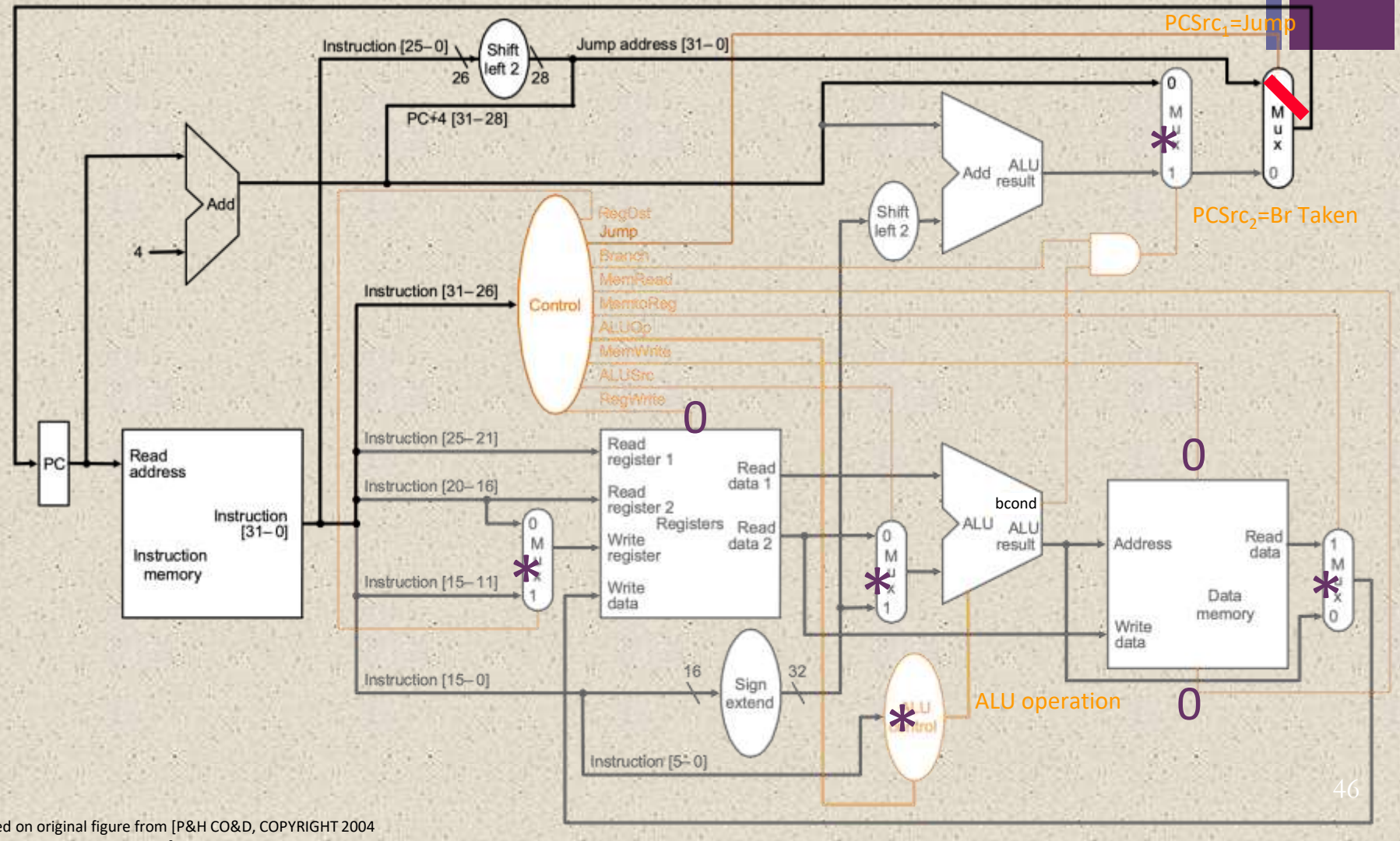
Branch Not Taken



Branch Taken



Jump



Can We Use the Idle Hardware to Improve Concurrency?

- Goal: Concurrency → throughput (more "work" completed in one cycle)

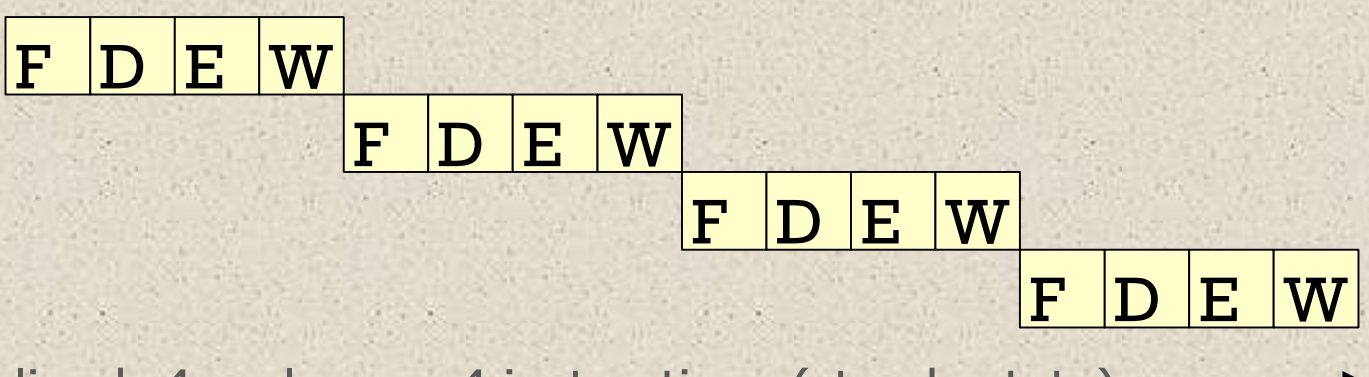
- Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Pipelining: Basic Idea

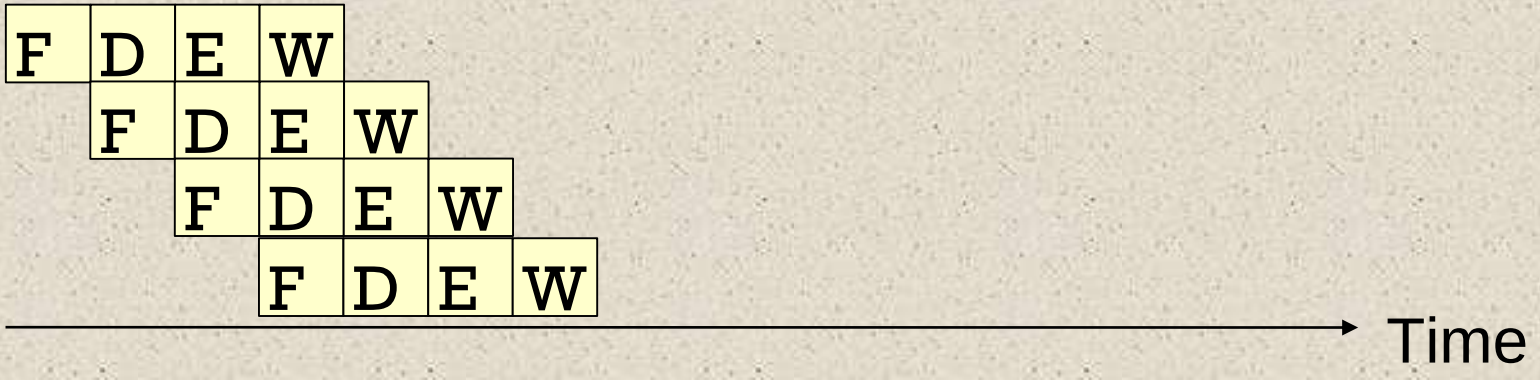
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…

Example: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)

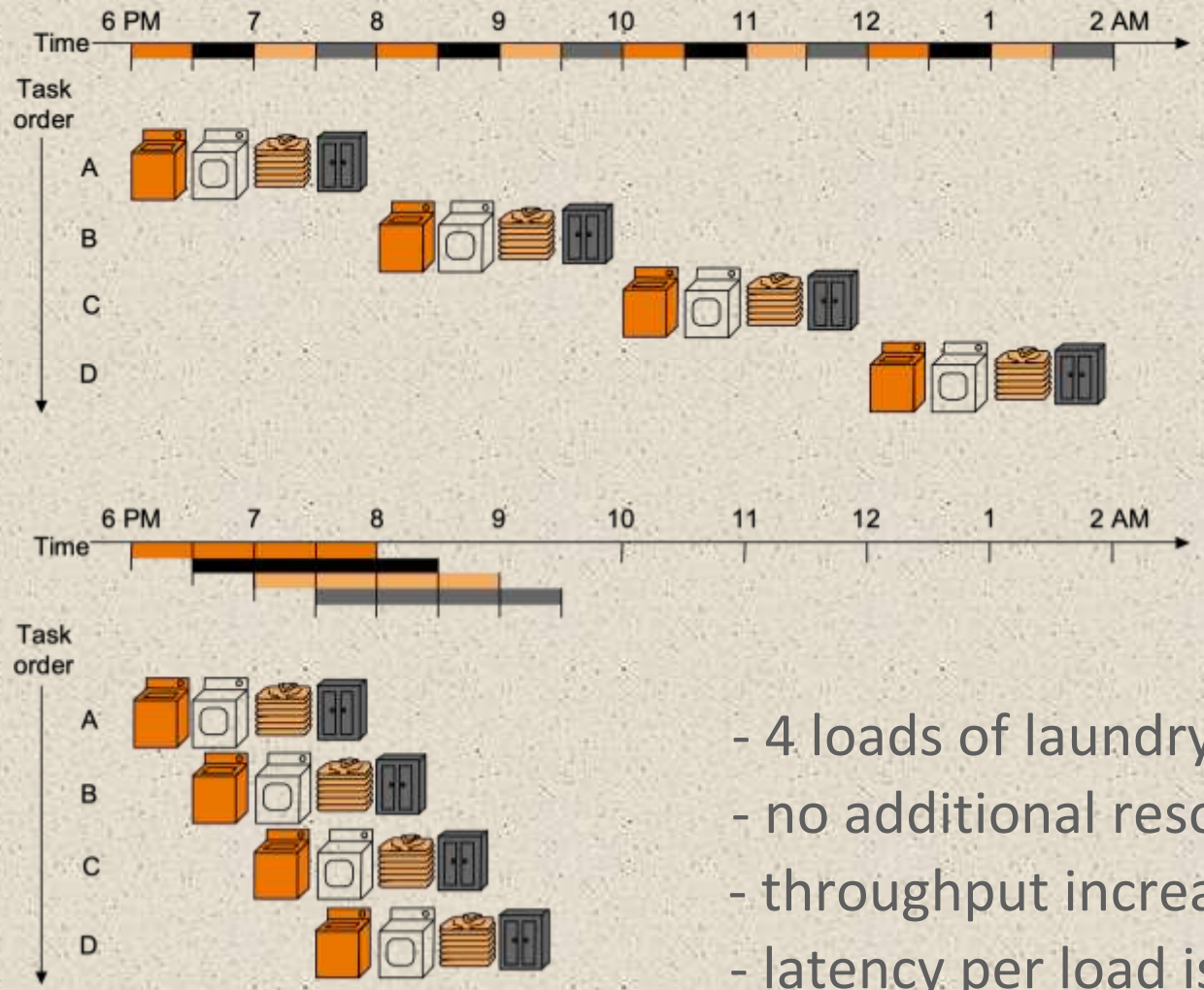


The Laundry Analogy

- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

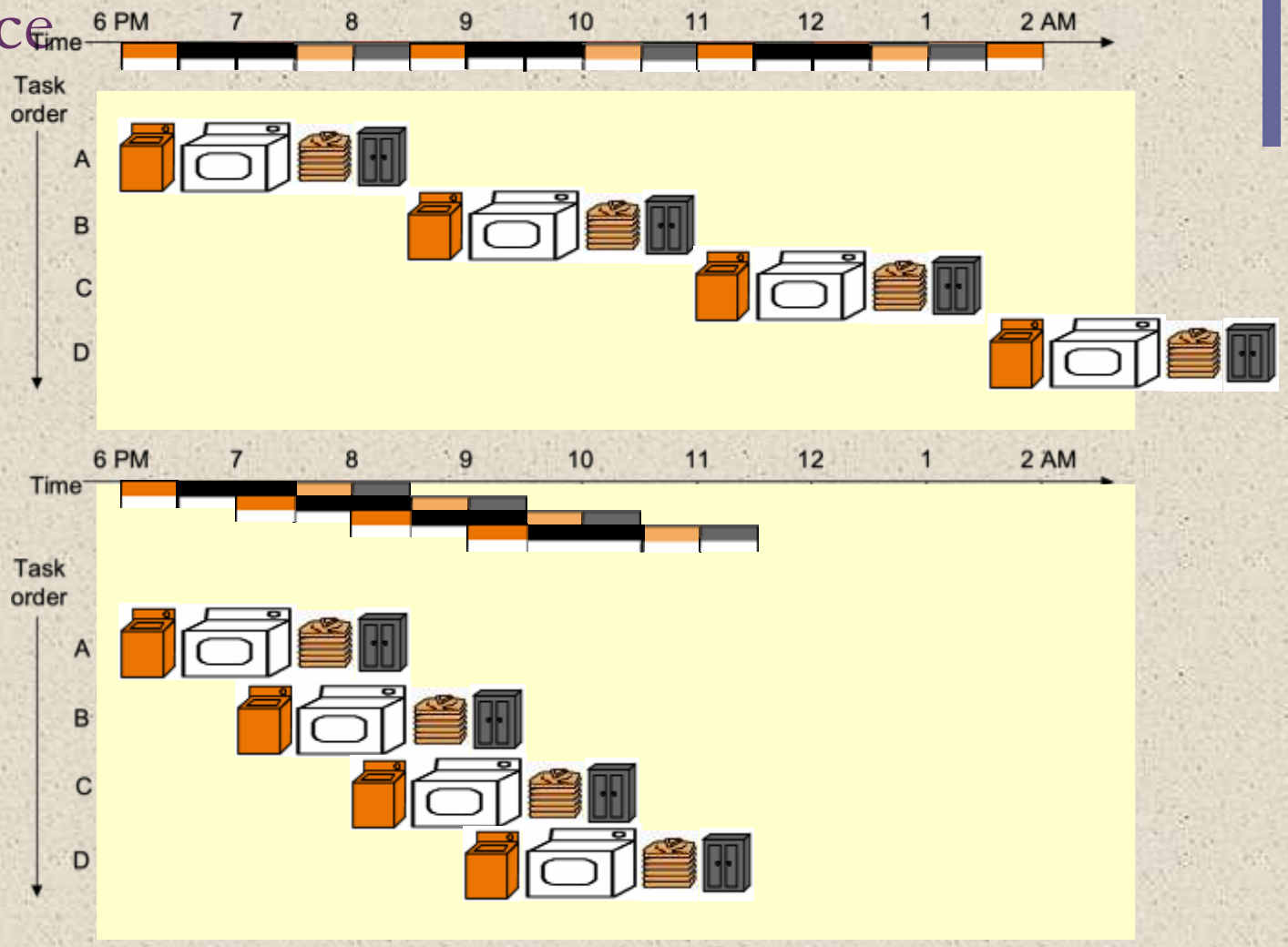
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

Pipelining Multiple Loads of Laundry



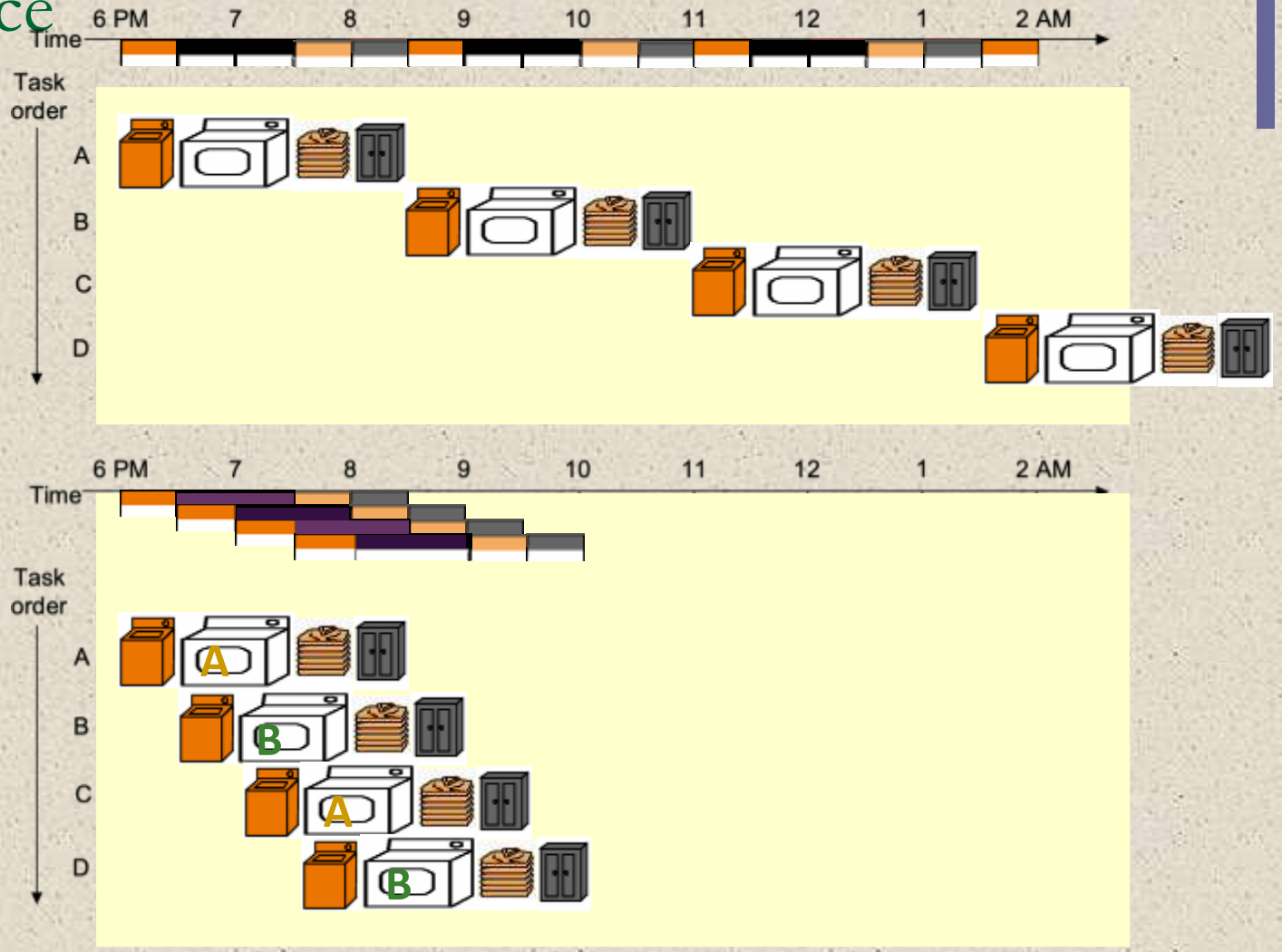
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice

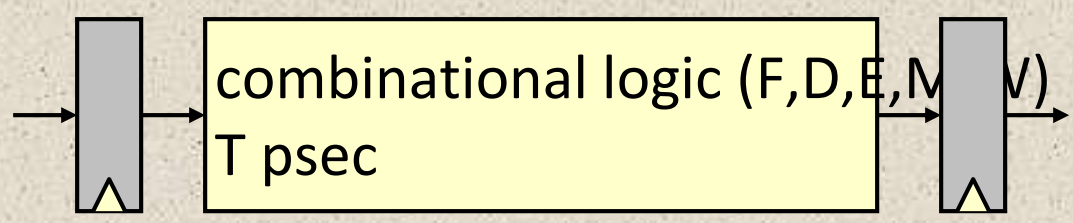


Throughput restored (2 loads per hour) using 2 dryers

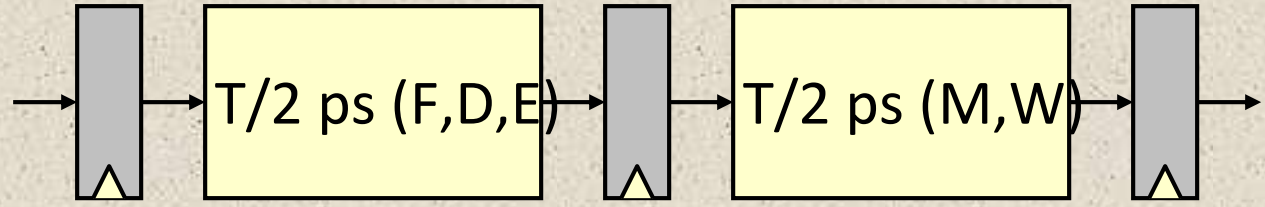
An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing "cycle"?

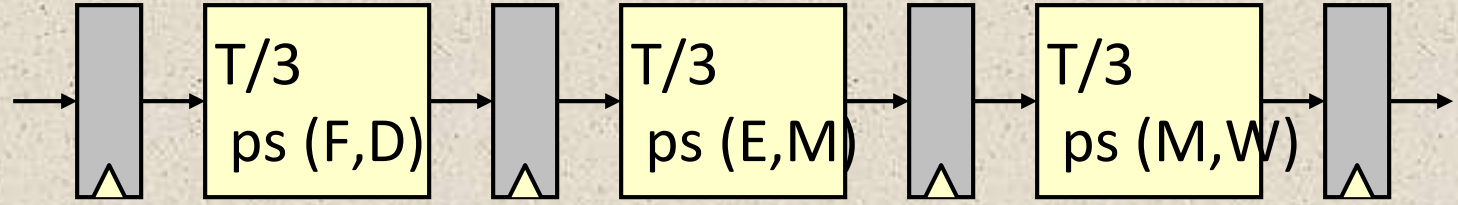
Ideal Pipelining



BW=~(1/T)



BW=~(2/T)

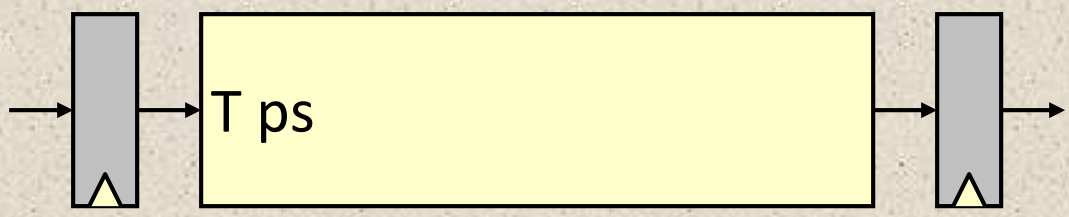


BW=~(3/T)

More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

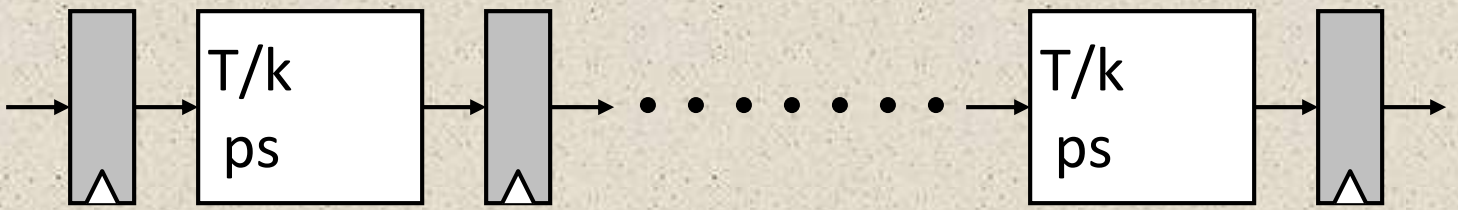
BW = 1/(T+S) where S = latch delay



- k-stage pipelined version

$BW_{k\text{-stage}} = 1 / (T/k + S)$

$BW_{max} = 1 / (1 \text{ gate delay} + S)$



End of Lecture