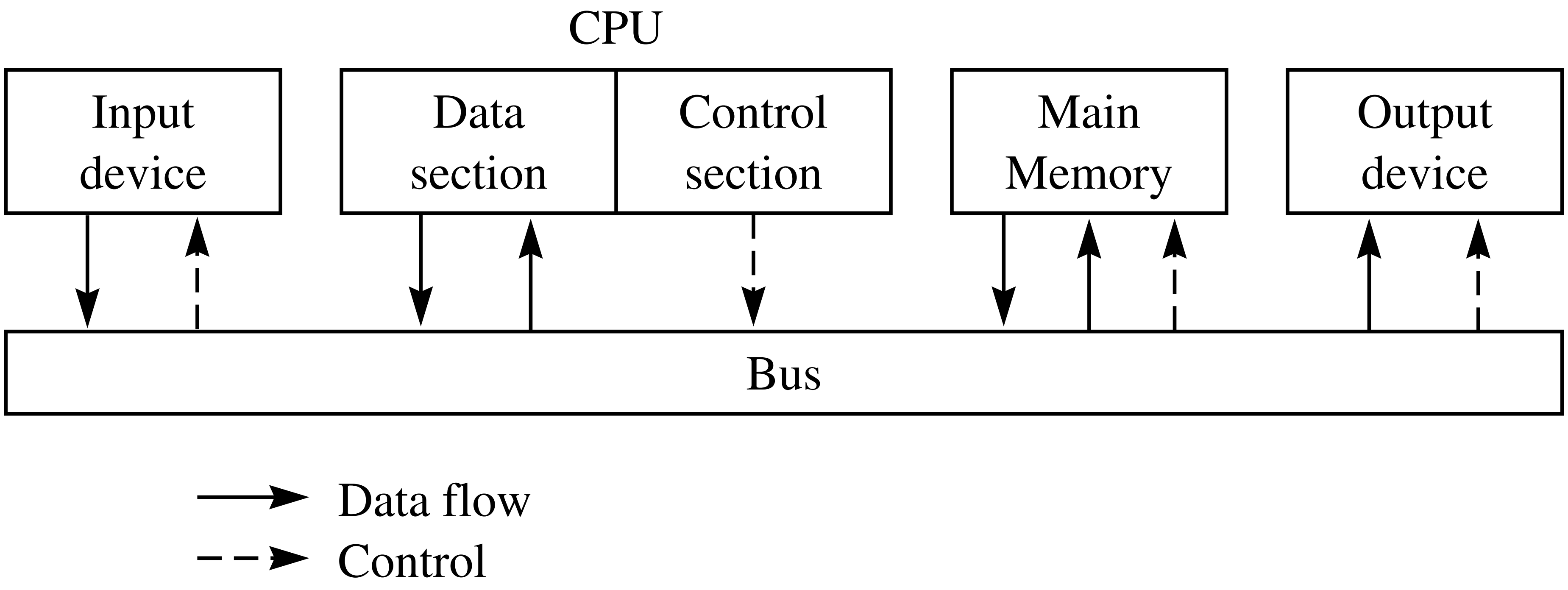
Computer Organization

Central Processing Unit (CPU)

- Data section
  - Receives data from and sends data to the main memory subsystem and I/O devices
- Control section
  - Issues the control signals to the data section and the other components of the computer system

Figure 12.1


CPU
Input device
Data section
Control section
Main Memory
Output device
Bus
Data flow
Control

CPU components

- 16-bit memory address register (MAR)
  - 8-bit MARA and 8-bit MARB
- 8-bit memory data register (MDR)
- 8-bit multiplexers
  - AMux, CMux, MDRMux
  - 0 on control line routes left input
  - 1 on control line routes right input

Control signals

- Originate from the control section on the right (not shown in Figure 12.2)
- Two kinds of control signals
  - Clock signals end in “Ck” to load data into registers with a clock pulse
  - Signals that do not end in “Ck” to set up the data flow before each clock pulse arrives

Figure 12.2

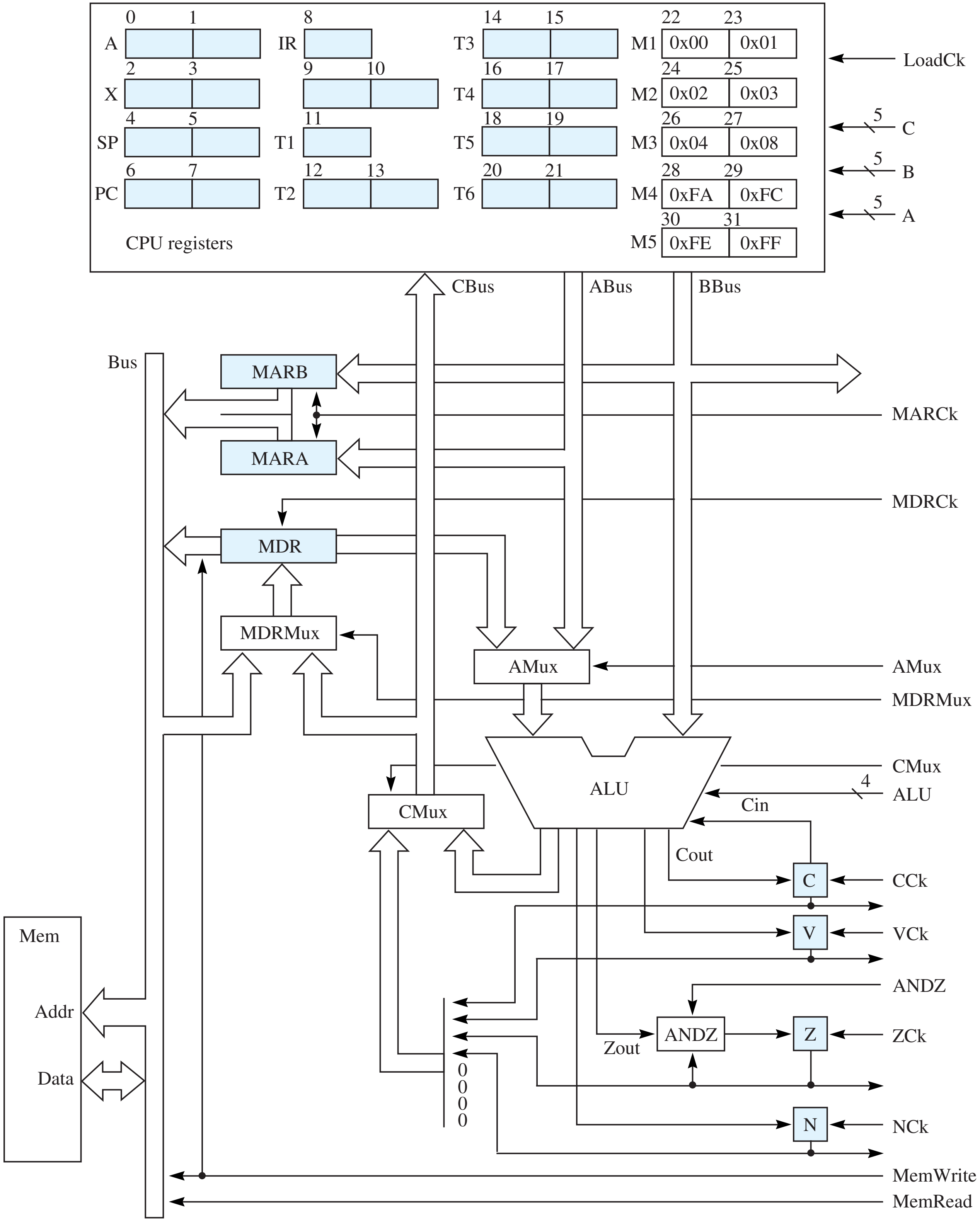

0
1
A
8
IR
14
15
T3
22
23
M1
0x00
0x01
2
3
X
9
10
16
17
T4
24
25
M2
0x02
0x03
4
5
SP
11
T1
18
19
T5
26
27
M3
0x04
0x08
6
7
PC
12
13
T2
20
21
T6
28
29
M4
0xFA
0xFC
30
31
M5
0xFE
0xFF
CPU registers
LoadCk
5
C
5
B
5
A
CBus
ABus
BBus
Bus
MARB
MARA
MARCk
MDRCk
MDR
MDRMux
AMux
AMux
MDRMux
ALU
CMux
4
ALU
Cin
CMux
Cout
C
CCk
V
VCk
ANDZ
ANDZ
Zout
Z
ZCk
N
NCk
0
0
0
0
Mem
Addr
Data
MemWrite
MemRead

Figure 12.2 (Expanded)

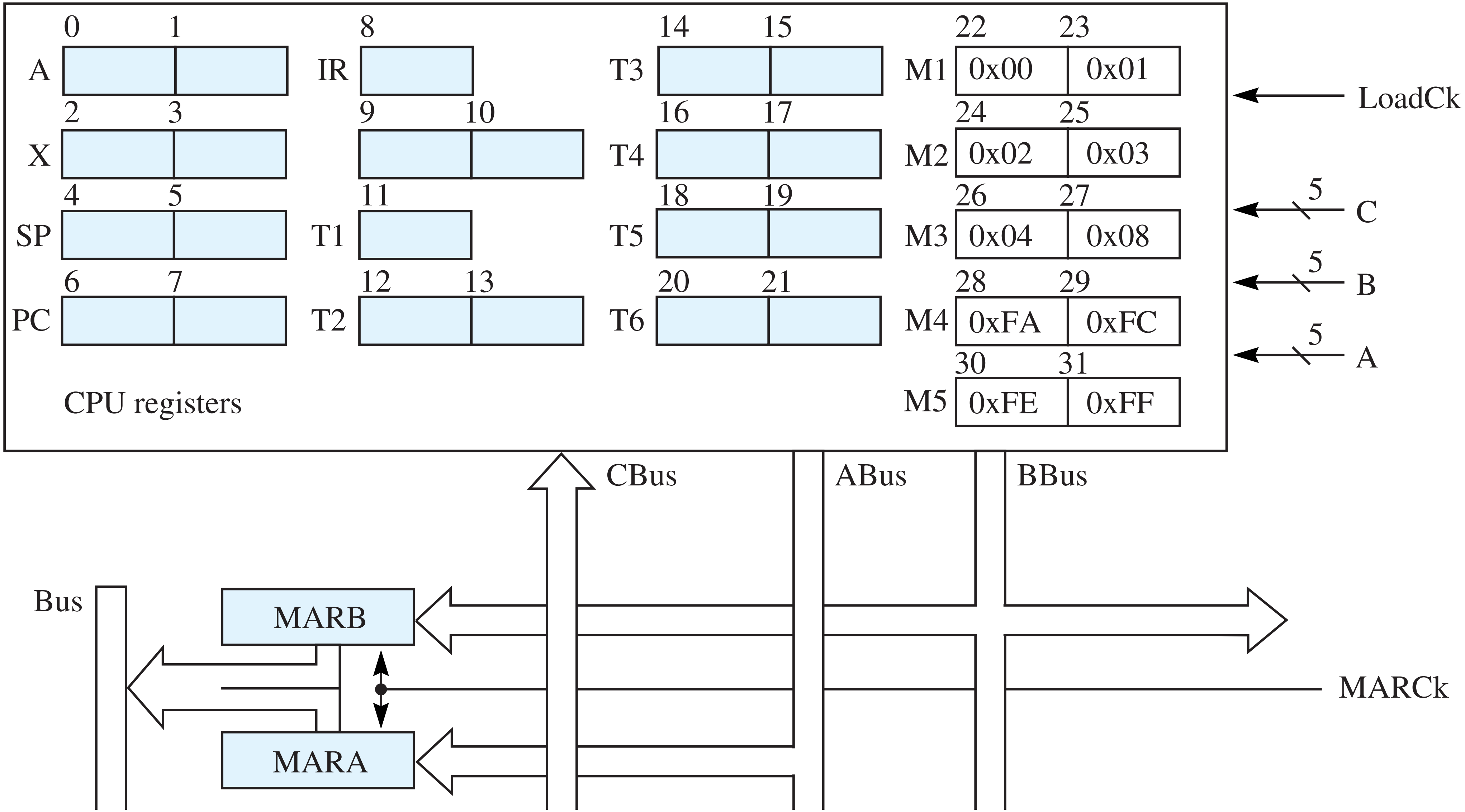
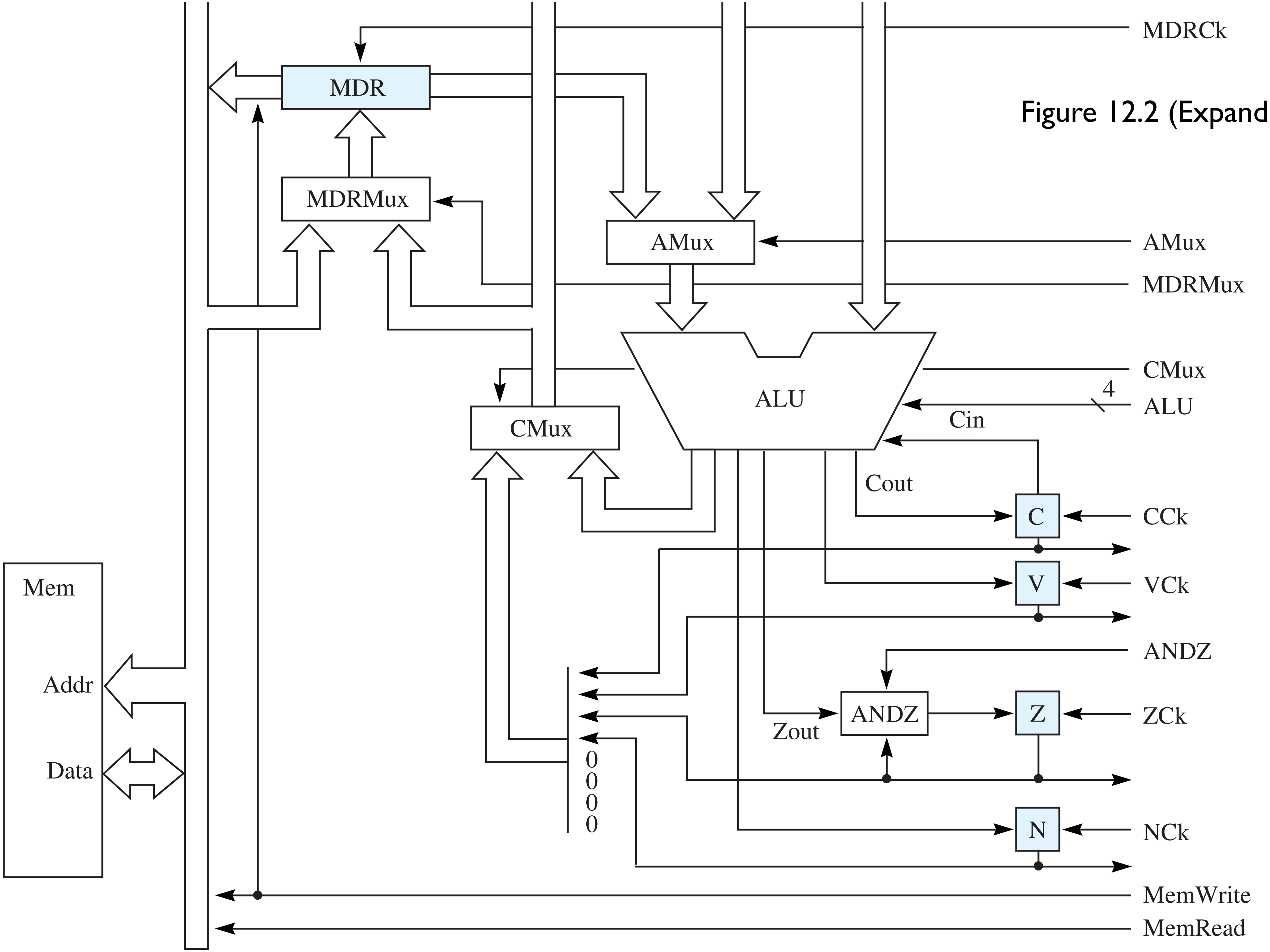


Figure 12.2 (Expanded)



The status bits NZVC

- Each status bit is a one-bit D flip-flop
- Each status bit is available to the control section
- The status bits can be sent as the low-order nybble to the left input of CMux and from there to the register bank

Setting the status bits

- C can be set directly from ALU Cout, and is the Cin input to the ALU
- V and N can be set directly from ALU
- Z can be set in one of two ways
  - If ANDZ control signal is 0, Z is set directly from ALU Zout
  - If ANDZ control signal is 1, Z is set as the AND of ALU Zout and Z

Figure 12.3

ANDZ	Z	Zout	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

von Neumann cycle

- The cycle at level ASMB5
    - Fetch
    - Decode
    - Increment
    - Execute
    - Repeat

Figure 4.30

```
Load the machine language program
Initialize PC and SP
do {
    Fetch the next instruction
    Decode the instruction specifier
    Increment PC
    Execute the instruction fetched
}
while (the stop instruction does not execute)
```

von Neumann cycle

- The cycle at level LG1
  - The instruction could be unary or nonunary
  - If nonunary, the instruction specifier must be fetched one byte at a time because the Pep/8 data bus is eight bits wide

Figure 12.4

```
do {
    Fetch the instruction specifier at address in PC
    PC ← PC + 1
    Decode the instruction specifier
    if (the instruction is not unary) {
        Fetch the high-order byte of the operand specifier at address in PC
        PC ← PC + 1
        Fetch the low-order byte of the operand specifier at address in PC
        PC ← PC + 1
    }
    Execute the instruction fetched
}
while ((the stop instruction does not execute) &&
       (the instruction is legal))
```

Control sequences

- Each line is a clock cycle
- Comma is the parallel separator
- Semicolon is the sequential separator
- Control signals before the semicolon are combinational signals set for the duration of the cycle
- Control signals after the the semicolon are clock pulses at the end of the cycle

Control signals for the von Neumann cycle

- To fetch from memory
  - Put the address in the MAR (MARA and MARB)
  - Assert MemRead for two consecutive cycles
  - At the end of the second cycle, clock the data from the bus into the MDR

Figure 12.2

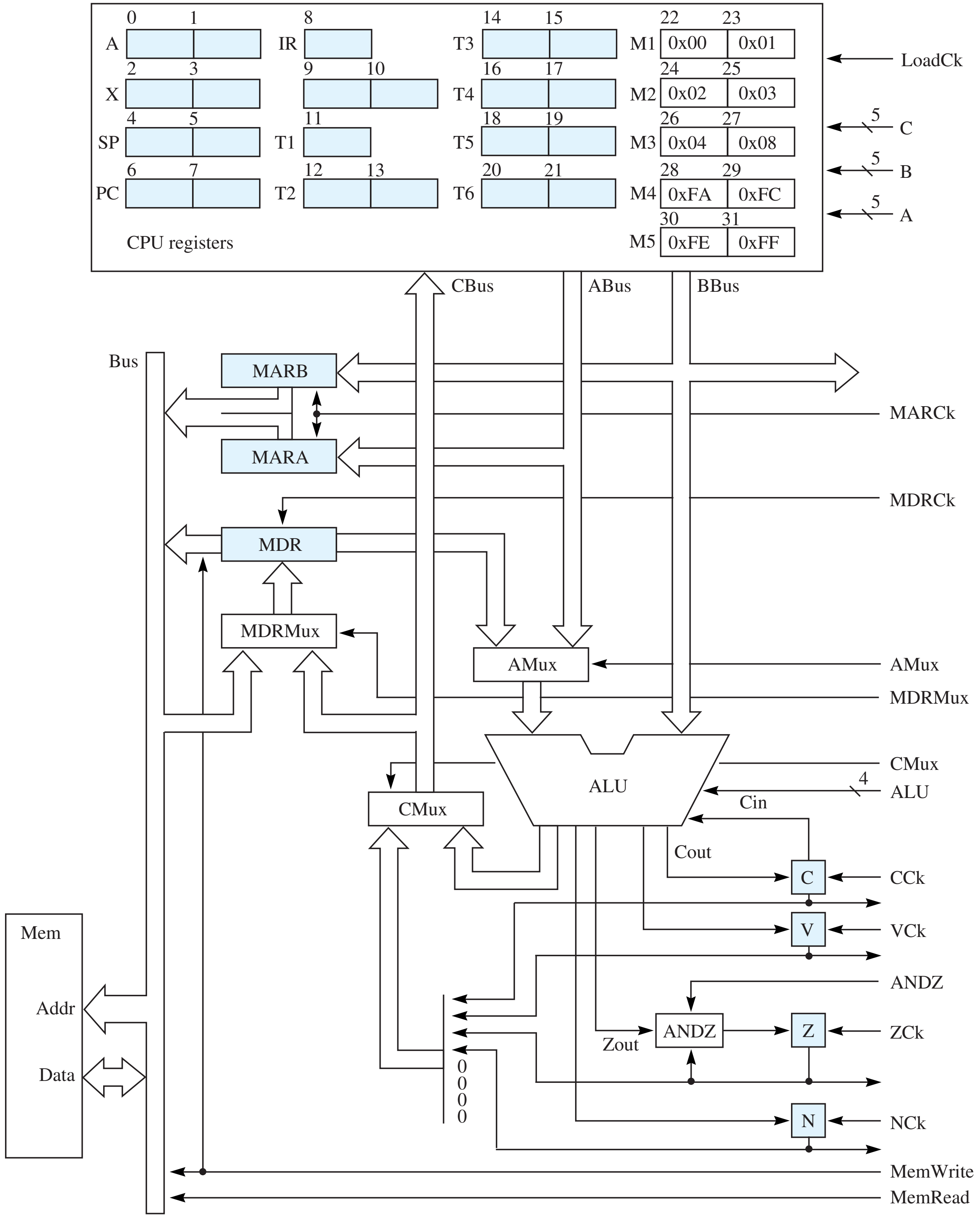
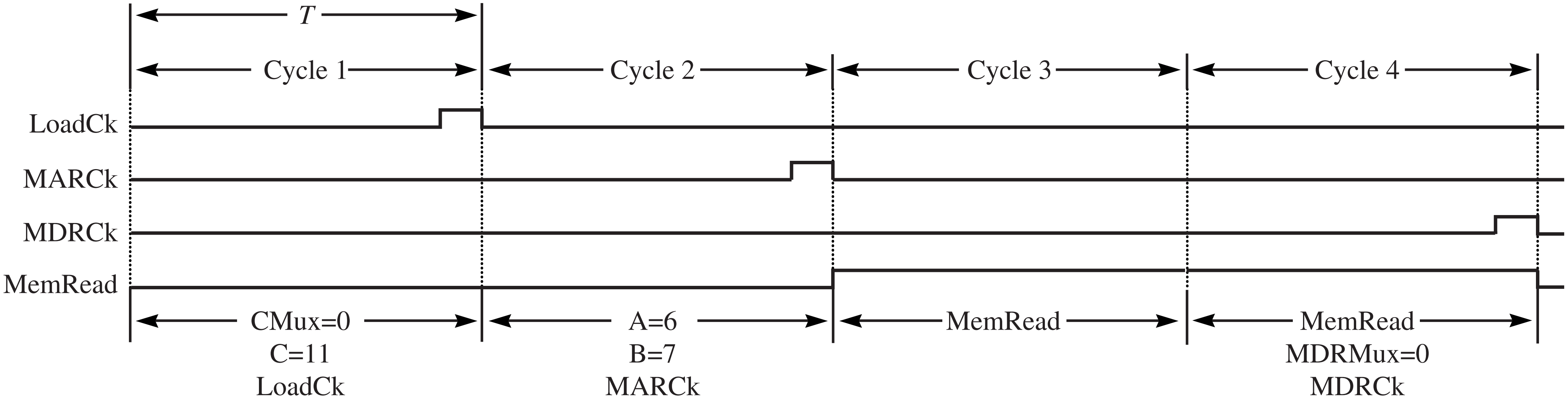

CPU registers
A
0
1
X
2
3
SP
4
5
PC
6
7
IR
8
9
10
T1
11
T2
12
13
T3
14
15
T4
16
17
T5
18
19
T6
20
21
M1
22
23
0x00
0x01
M2
24
25
0x02
0x03
M3
26
27
0x04
0x08
M4
28
29
0xFA
0xFC
M5
30
31
0xFE
0xFF
LoadCk
5
C
5
B
5
A
CBus
ABus
BBus
Bus
MARB
MARA
MARCk
MDRCk
MDR
MDRMux
AMux
AMux
MDRMux
ALU
CMux
CMux
4
ALU
Cin
Cout
C
CCk
V
VCk
ANDZ
ANDZ
Zout
Z
ZCk
N
NCk
0
0
0
0
Mem
Addr
Data
MemWrite
MemRead

Figure 12.5

```
// Save the status bits in T1
1. CMux=0, C=11; LoadCk

// MAR <- PC, fetch instruction specifier.
2. A=6, B=7; MARCk
3. MemRead
4. MemRead, MDRMux=0; MDRCk
5. AMux=0, ALU=0, CMux=1, C=8; LoadCk

// PC <- PC + 1, low-order byte first.
6. A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; CCk, LoadCk
7. A=6, B=22, AMux=1, ALU=2, CMux=1, C=6; LoadCk

// If the instruction is not unary, fetch operand specifier
// and increment PC
...
// Restore the carry bit from T1
n. A=11, AMux=1, ALU=15; CCk

// Execute the instruction fetched
```

Figure 12.6


T
Cycle 1
Cycle 2
Cycle 3
Cycle 4
LoadCk
MARCk
MDRCk
MemRead
CMux=0
C=11
LoadCk
A=6
B=7
MARCk
MemRead
MemRead
MDRMux=0
MDRCk

Combining cycles

- Can combine cycles 1 and 3
- Eliminate cycle 1
- Keep cycle 2 unchanged
- Cycle 3 (renumbered cycle 2) is

```
MemRead, CMux=0, C=11; LoadCk
```

Combining cycles

- Exercise for the student: Combine cycles 1 through 7 in Figure 12.5 to produce an implementation of just 4 cycles.

Figure 12.7

Addressing mode	Operand
Immediate	OprndSpec
Direct	Mem [OprndSpec]
Indirect	Mem [Mem [OprndSpec]]
Stack-relative	Mem [SP + OprndSpec]
Stack-relative deferred	Mem [Mem [SP + OprndSpec]]
Indexed	Mem [OprndSpec + X]
Stack-indexed	Mem [SP + OprndSpec + X]
Stack-indexed deferred	Mem [Mem [SP + OprndSpec] + X]

Store byte

- Assume operand specifier has been fetched and contains the address of the operand
- To write to memory
  - Put the address in the MAR and the data to to be written in the MDR
  - Assert MemWrite for two consecutive cycles

$$\text{byte Oprnd} \leftarrow r\langle 8..15\rangle$$

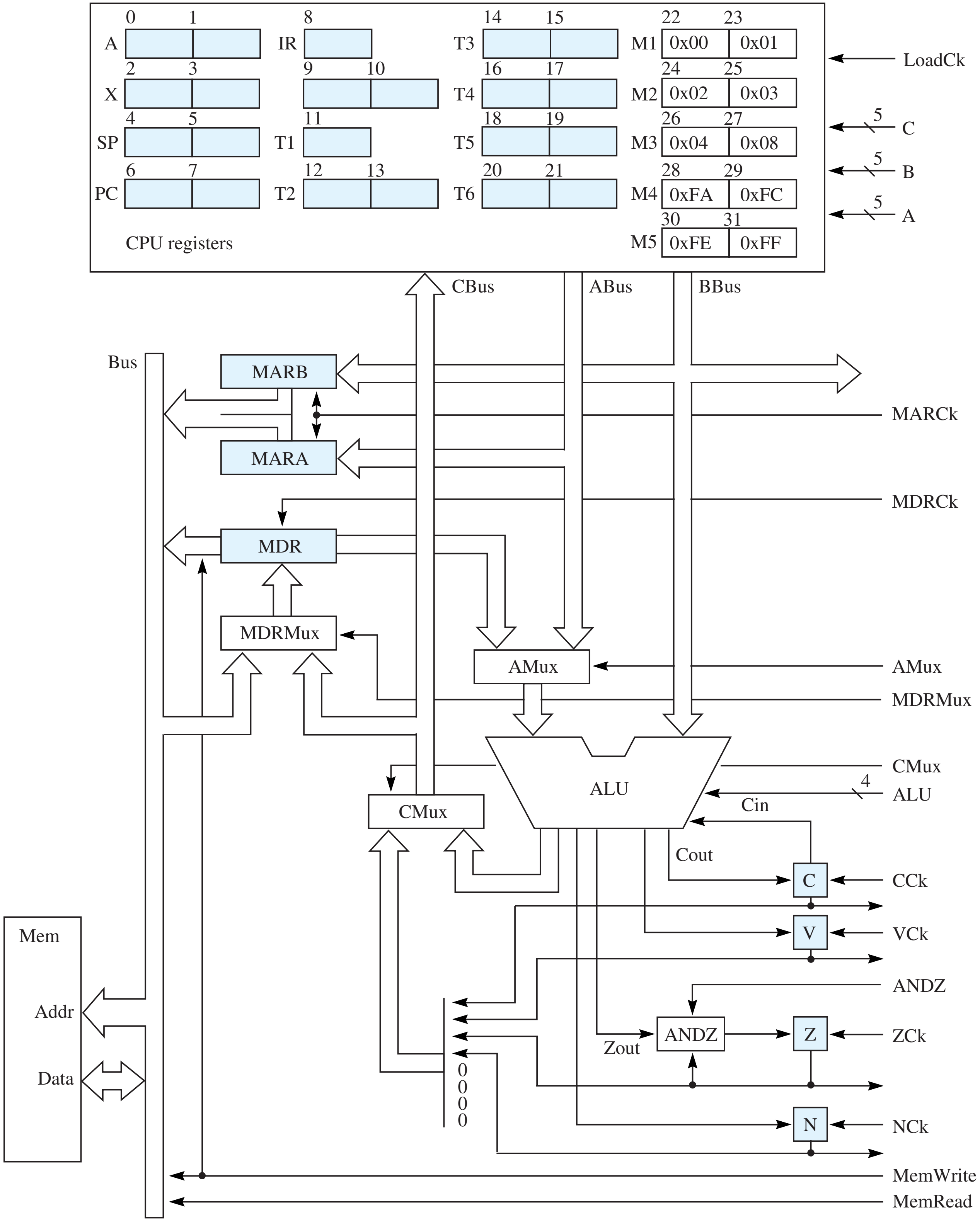

0
1
A
8
IR
14
15
T3
22
23
M1
0x00
0x01
2
3
X
9
10
16
17
T4
24
25
M2
0x02
0x03
4
5
SP
11
T1
18
19
T5
26
27
M3
0x04
0x08
6
7
PC
12
13
T2
20
21
T6
28
29
M4
0xFA
0xFC
30
31
M5
0xFE
0xFF
CPU registers
LoadCk
5
C
5
B
5
A
CBus
ABus
BBus
Bus
MARB
MARA
MARCk
MDRCk
MDR
MDRMux
AMux
AMux
MDRMux
ALU
CMux
CMux
4
ALU
Cin
Cout
C
CCk
V
VCk
ANDZ
ANDZ
Zout
Z
ZCk
N
NCk
0
0
0
0
Mem
Addr
Data
MemWrite
MemRead

Figure 12.2

Figure 12.8

```
// STBYTEA there,d
// RTL: byteOprnd <- A<8..15>
// Direct addressing: Oprnd = Mem[OprndSpec]
// Shortest known implementation: 4 cycles

// MAR <- OprndSpec.
1. A=9, B=10; MARCk

// MBR <- A<low>.
2. A=1, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk

// Initiate memory write.
3. MemWrite

// Complete memory write.
4. MemWrite
```

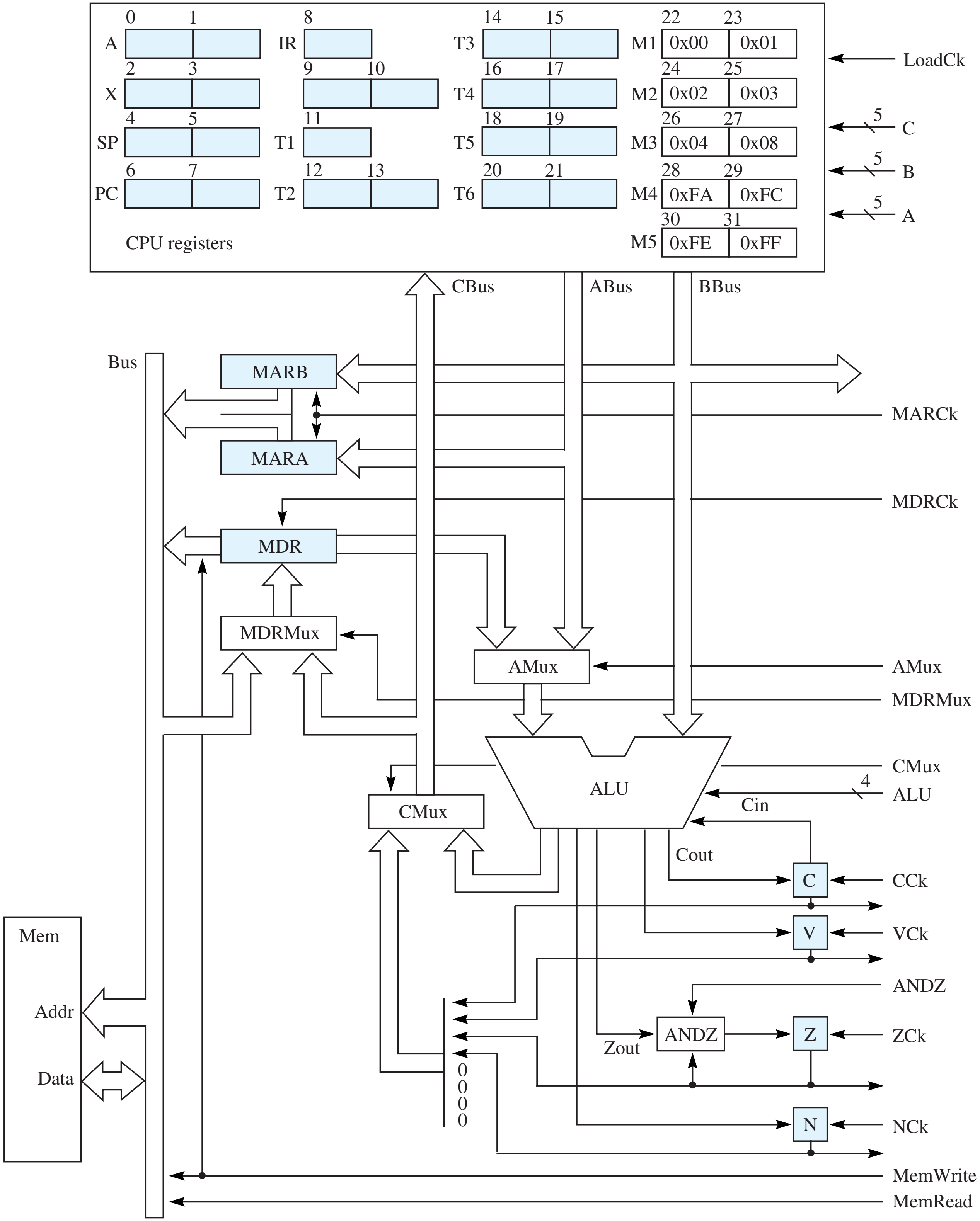
Add accumulator

- Assume immediate addressing
- The number to be added is in the operand specifier

$$\mathrm{r} \leftarrow \mathrm{r} + \mathrm{Oprnd}\ ;\ \mathrm{N} \leftarrow \mathrm{r} < 0\ ,\ \mathrm{Z} \leftarrow \mathrm{r} = 0\ ,$$

$$\mathrm{V} \leftarrow \{\mathit{overflow}\}\ ,\ \mathrm{C} \leftarrow \{\mathit{carry}\}$$

Figure 12.2



0
1
A
8
IR
14
15
T3
22
23
M1
0x00
0x01
2
3
X
9
10
16
17
T4
24
25
M2
0x02
0x03
4
5
SP
11
T1
18
19
T5
26
27
M3
0x04
0x08
6
7
PC
12
13
T2
20
21
T6
28
29
M4
0xFA
0xFC
30
31
M5
0xFE
0xFF
CPU registers
LoadCk
5
C
5
B
5
A
CBus
ABus
BBus
Bus
MARB
MARA
MARCk
MDRCk
MDR
MDRMux
AMux
AMux
MDRMux
ALU
CMux
CMux
4
ALU
Cin
Cout
C
CCk
V
VCk
ANDZ
ANDZ
Z
ZCk
Zout
N
NCk
0
0
0
0
Mem
Addr
Data
MemWrite
MemRead

Figure 12.9

```
// ADDA this,i
// RTL: A <- A + Oprnd; N <- A<0, Z <- A=0, V <- {overflow}, C <- {carry}
// Immediate addressing: Oprnd = OprndSpec
// Shortest known implementation: 2 cycles

// A<low> <- A<low> + Oprnd<low>. Save carry.
1. A=1, B=10, AMux=1, ALU=1, ANDZ=0, CMux=1, C=1; ZCk, CCk, LoadCk

// A<high> <- A<high> plus Oprnd<high> plus saved carry.
2. A=0, B=9, AMux=1, ALU=2, ANDZ=1, CMux=1, C=0; NCk, ZCk, VCk, CCk, LoadCk
```

Load index register

- Assume indirect addressing
- The operand specifier contains the address of the address of the operand

$$r \leftarrow \mathrm{Oprnd} \; ; \; N \leftarrow r < 0 \; , \; Z \leftarrow r = 0$$

Figure 12.11

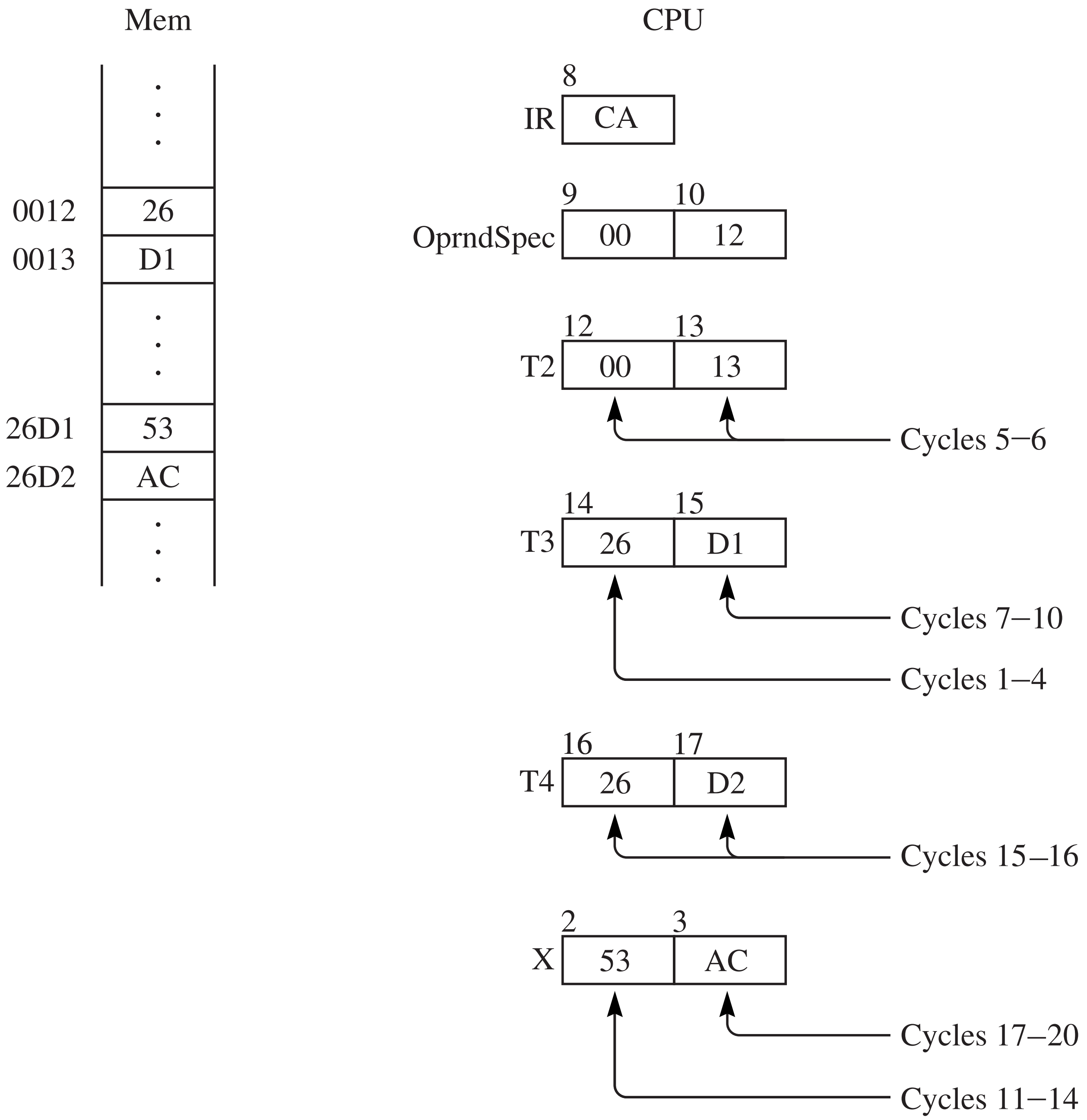

Mem
CPU
0012
26
0013
D1
26D1
53
26D2
AC
8
IR
CA
9
10
OprndSpec
00
12
12
13
T2
00
13
Cycles 5–6
14
15
T3
26
D1
Cycles 7–10
Cycles 1–4
16
17
T4
26
D2
Cycles 15–16
2
3
X
53
AC
Cycles 17–20
Cycles 11–14

Figure 12.10

```
// LDX this,n
// RTL: X <- Oprnd; N <- X<0, Z <- X=0
// Indirect addressing: Oprnd = Mem[Mem[OprndSpec]]
// Shortest known implementation: 17 cycles

// T3<high> <- Mem[OprndSpec].
1. A=9, B=10; MARCk
2. MemRead
3. MemRead, MDRMux=0; MDRCk
4. AMux=0, ALU=0, CMux=1, C=14; LoadCk

// T2 <- OprndSpec + 1.
5. A=10, B=23, AMux=1, ALU=1, CMux=1, C=13; CCk, LoadCk
6. A=9, B=22, AMux=1, ALU=2, CMux=1, C=12; LoadCk

// T3<low> <- Mem[T2].
7. A=12, B=13; MARCk
8. MemRead
9. MemRead, MDRMux=0; MDRCk
10. AMux=0, ALU=0, CMux=1, C=15; LoadCk
```

Figure 12.10 (Continued)

```
// Assert: T3 contains the address of the operand.
// X<high> <- Mem[T3].
11. A=14, B=15; MARCk
12. MemRead
13. MemRead, MDRMux=0; MDRCk
14. AMux=0, ALU=0, ANDZ=0, CMux=1, C=2; NCk, ZCk, LoadCk

// T4 <- T3 + 1.
15. A=15, B=23, AMux=1, ALU=1, CMux=1, C=17; CCk, LoadCk
16. A=14, B=22, AMux=1, ALU=2, CMux=1, C=16; LoadCk

// X<low> <- Mem[T4].
17. A=16, B=17; MARCk
18. MemRead
19. MemRead, MDRMux=0; MDRCk
20. AMux=0, ALU=0, ANDZ=1, CMux=1, C=3; ZCk, LoadCk

// Restore C, assumed in T1 from Fetch.
21. A=11, AMux=1, ALU=15; CCk
```

Combining cycles

- Exercise for the student: Combine cycles 1 through 21 in Figure 12.10 to produce an implementation of just 17 cycles.

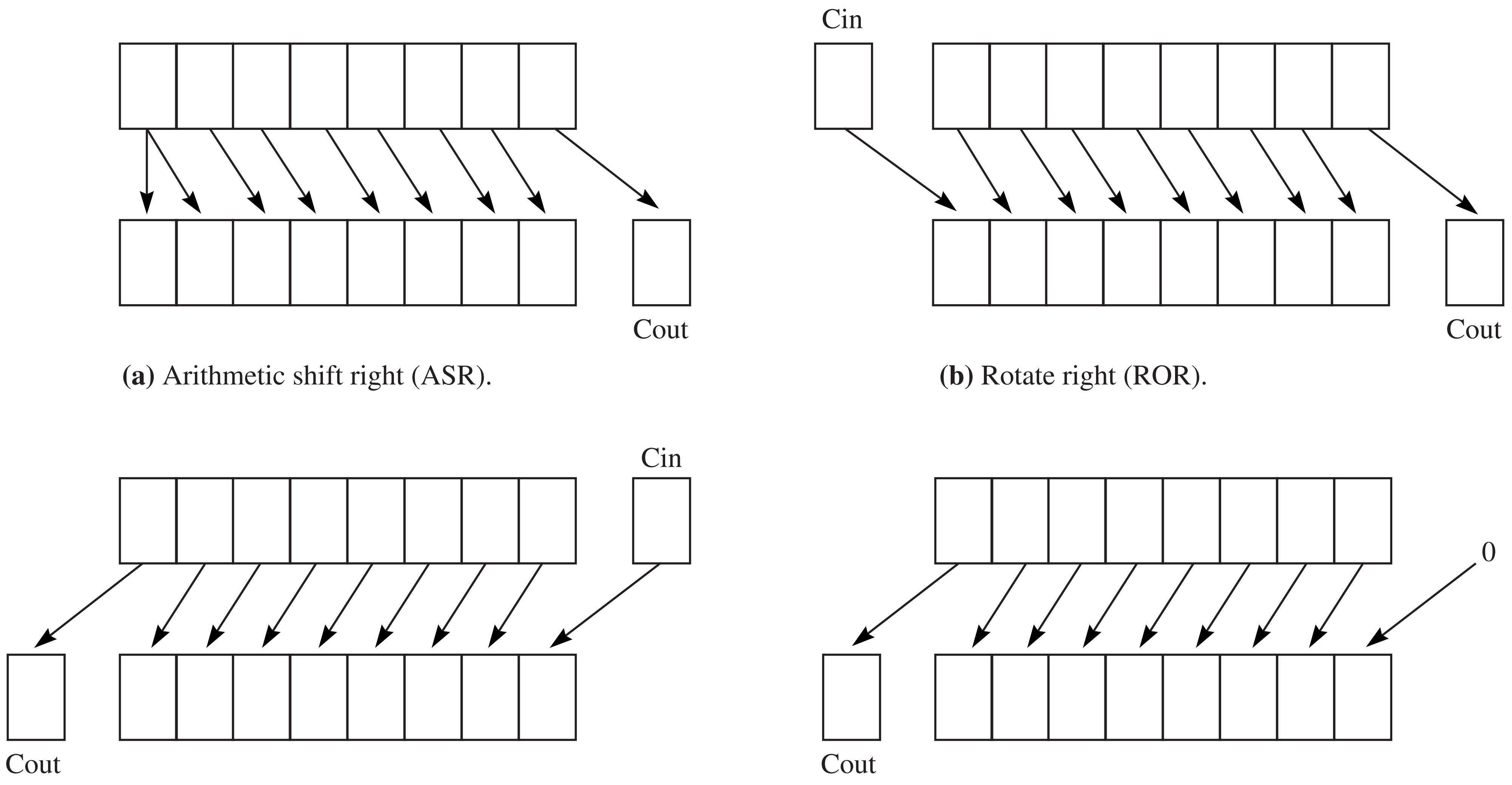
Arithmetic shift right accumulator

- Unary instruction
- No memory read or write

$$C \leftarrow r\langle 15\rangle\ ,\ r\langle 1..15\rangle \leftarrow r\langle 0..14\rangle;$$

$$N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$$

Figure 12.13



(a) Arithmetic shift right (ASR).

(b) Rotate right (ROR).

(c) Rotate left (ROL).

(d) Arithmetic shift left (ASL).

Figure 12.12

```
// ASRA
// RTL: C <- A<15>, A<1..15> <- A<0..14>; N <- A<0, Z <- A=0
// Shortest known implementation: 2 cycles

// Arithmetic shift right of high-order byte
1. A=0, AMux=1, ALU=13, ANDZ=0, CMux=1, C=0; NCk, ZCk, CCk, LoadCk

// Rotate right of low-order byte
2. A=1, AMux=1, ALU=14, ANDZ=1, CMux=1, C=1; ZCk, CCk, LoadCk
```

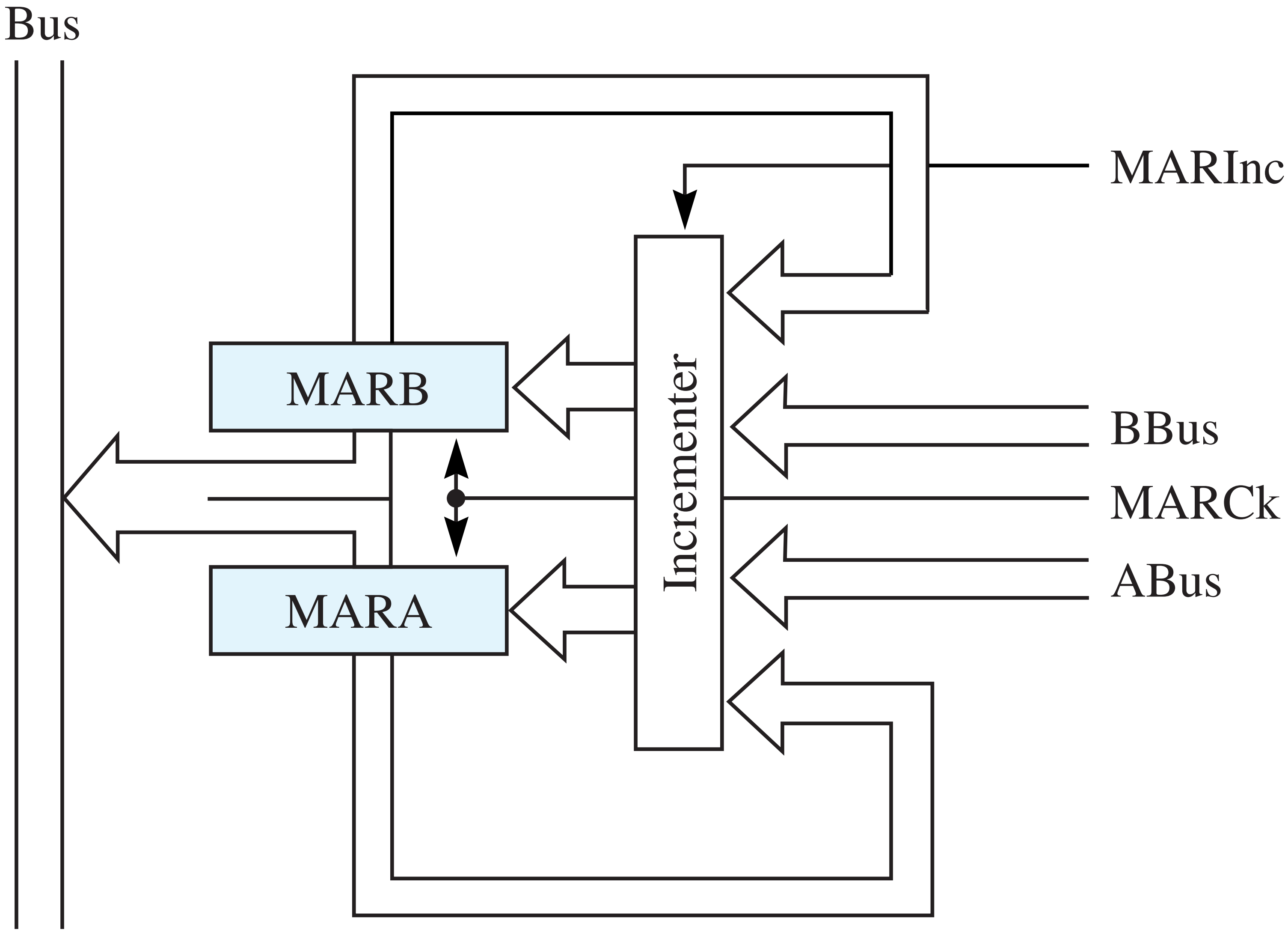
Performance issues

- Fundamental sources of increased performance
  - The space/time tradeoff
  - Parallelism

Specialized hardware units

- ALU is a general-purpose device for 8-bit operations
- To increment a 16-bit address requires three cycles, e.g. Figure 12.10, cycles 5, 6, 7
- Design special-purpose address incrementer
    - MARInc = 0, routes ABus and BBus to MAR
    - MARInc = 1, routes 16-bit MAR + 1 to MAR

Figure 12.17



```
// T3<high> <- Mem[OprndSpec].
// MAR <- OprndSpec + 1.
1. A=9, B=10, MARInc=0; MARCk
2. MemRead
3. MemRead, MDRMux=0, MARInc=1; MDRCk, MARCk
4. MemRead, AMux=0, ALU=0, CMux=1, C=14; LoadCk

// T3<low> <- Mem[OprndSpec + 1].
5. MemRead, MDRMux=0; MDRCk
6. AMux=0, ALU=0, CMux=1, C=15; LoadCk

// Assert: T3 contains the address of the operand.
// X<high> <- Mem[T3].
// MAR <- T3 + 1.
7. A=14, B=15, MARInc=0; MARCk
8. MemRead
9. MemRead, MDRMux=0, MARInc=1; MDRCk, MARCk
10. MemRead, AMux=0, ALU=0, ANDZ=0, CMux=1, C=2; NCk, ZCk, LoadCk

// X<low> <- Mem[T3 + 1].
11. MemRead, MDRMux=0; MDRCk
12. AMux=0, ALU=0, ANDZ=1, CMux=1, C=3; ZCk, LoadCk
```

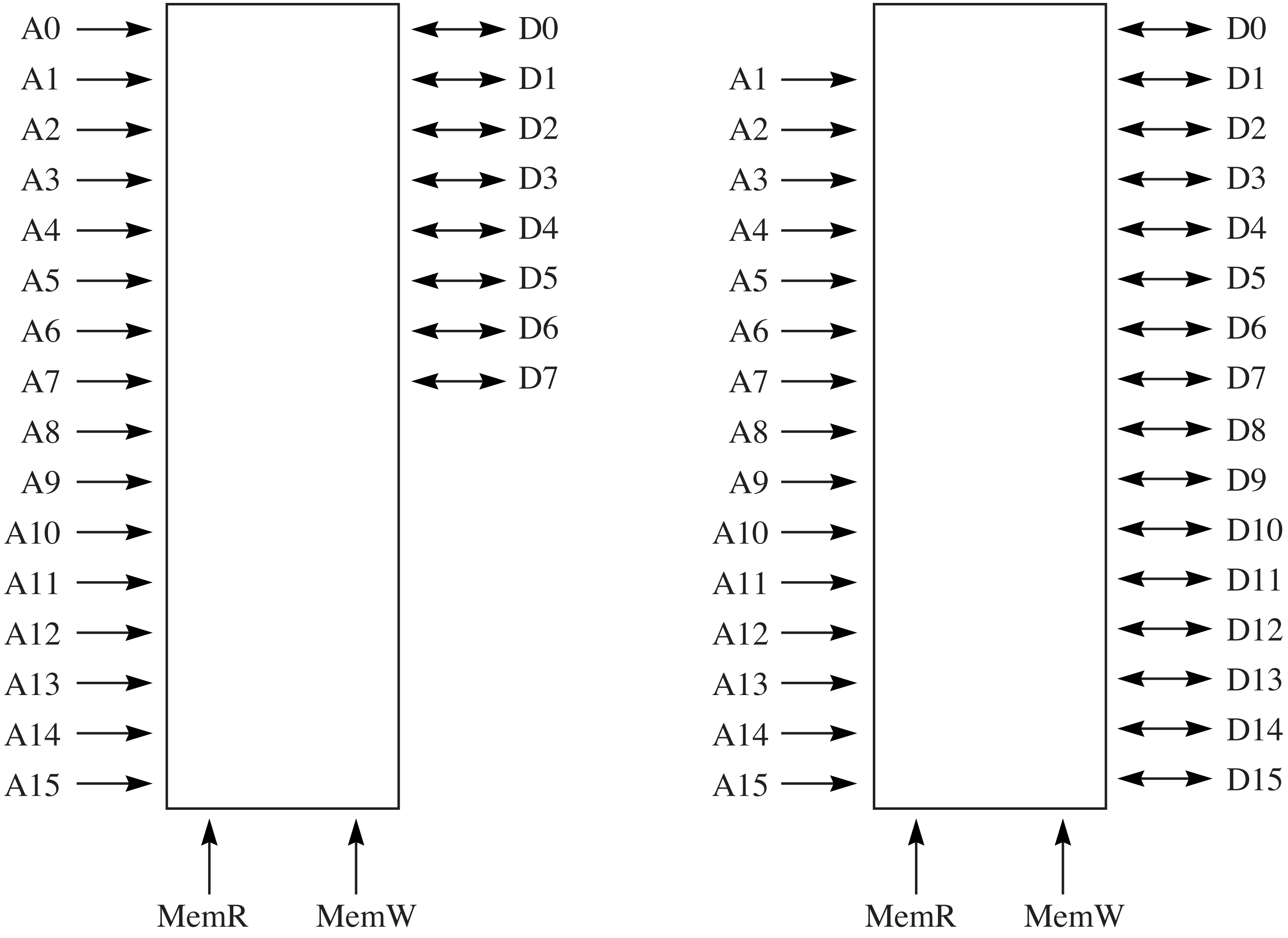
Result

- After combining cycles, original takes 17 cycles
- With incrementer, it takes 12 cycles
- 17 – 12 = 5 cycles saved
- Time savings is 5 / 17 = 29%

Increase data bus width

- Application of the space/time tradeoff
- Could increase data bus from 8 bits to 16 bits
- MDR becomes 16 bits wide
- Problem: Memory is still byte-addressable
- Memory alignment issues

Figure 12.14



(a) The chip of Figure 12.2.

(b) A chip with a 16-bit data bus.

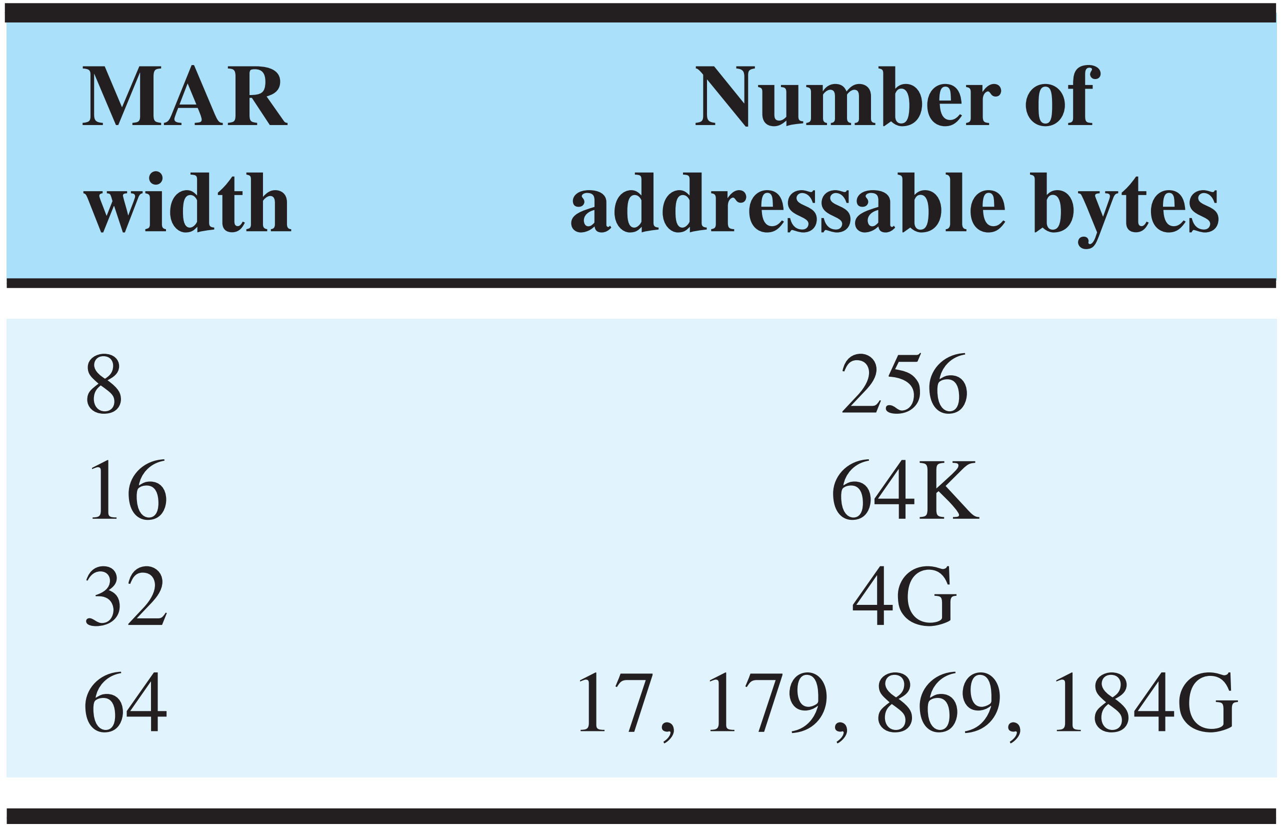
Figure 12.15

Chip	Date	Register width
4004	1971	4-bit
8008	1972	8-bit
8086	1978	16-bit
80386	1985	32-bit

n-bit computers

- An *n*-bit computer has *n* bits in the MAR and in the CPU registers that are visible at level ISA3 (usually equal)
- The registers in the register bank at level LG1 could have width less than *n*
- The data bus could have width greater than *n*
- The address bus could have width less than *n*

Figure 12.16



MAR width	Number of addressable bytes
8	256
16	64K
32	4G
64	17, 179, 869, 184G

Increase bus width

- Split MDR into MDR0 and MDR1 with 8 bits each
- Put a 16-bit bus between MDR and MAR
- Replace incrementer with MARMux
  - MARMux = 0, route MDR to output
  - MARMux = 1, route ABus and BBus to output

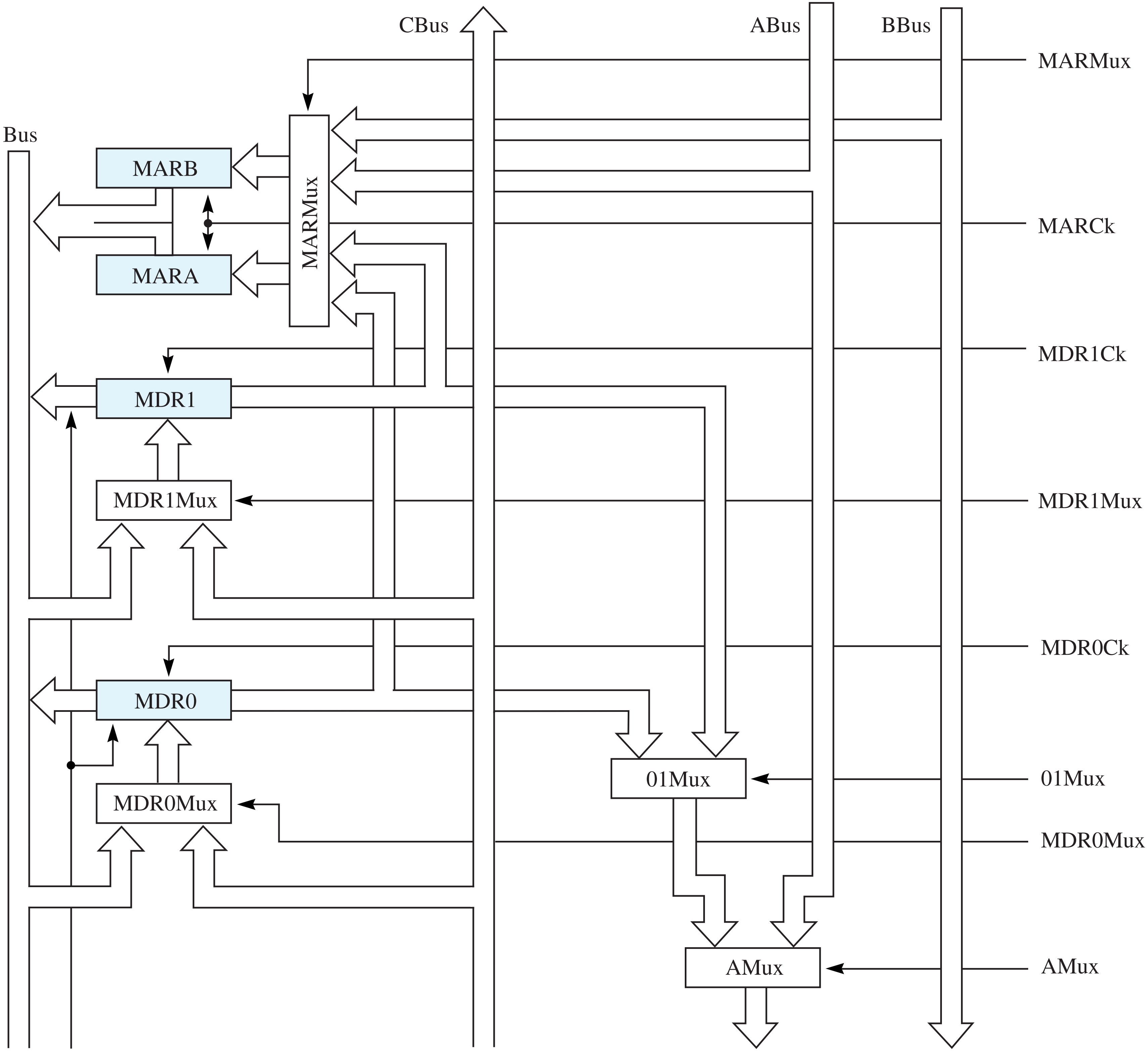

CBus
ABus
BBus
Bus
MARMux
MARB
MARA
MARMux
MARCk
MDR1Ck
MDR1
MDR1Mux
MDR1Mux
MDR0Ck
MDR0
01Mux
01Mux
MDR0Mux
MDR0Mux
AMux
AMux

Figure 12.19

Figure 12.20

```
// MDR <- Mem[OprndSpec].
1. A=9, B=10, MARMux=1; MARCk
2. MemRead
3. MemRead, MDR0Mux=0, MDR1Mux=0; MDR0Ck, MDR1Ck

// MAR <- MDR.
4. MARMux=0; MARCk

// MDR <- two-byte operand.
5. MemRead
6. MemRead, MDR0Mux=0, MDR1Mux=0; MDR0Ck, MDR1Ck

// X <- MDR, high-order first.
7. 01Mux=0, AMux=0, ALU=0, ANDZ=0, CMux=1, C=2; NCk, ZCk, LoadCk
8. 01Mux=1, AMux=0, ALU=0, ANDZ=1, CMux=1, C=3; ZCk, LoadCk
```

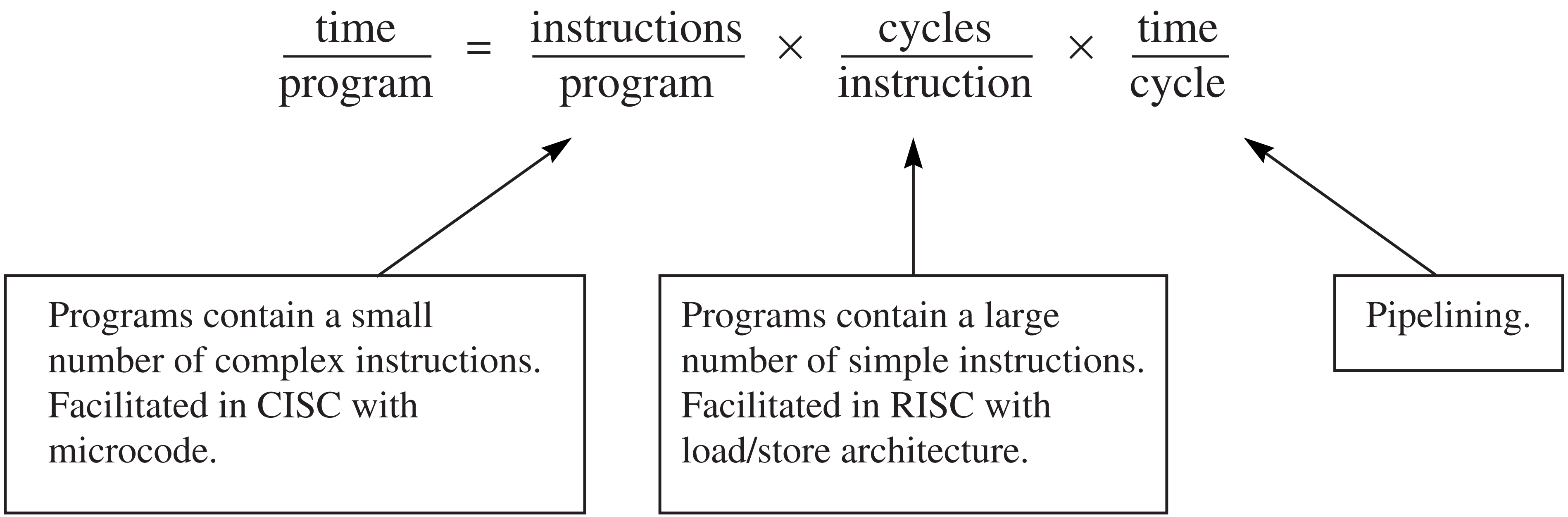
Result

- After combining cycles, original takes 17 cycles
- With 16-bit MDR, it takes 8 cycles
- 17 – 8 = 9 cycles saved
- Time savings is 9 / 17 = 53%

Three components of execution time

- instructions / program
- cycles / instruction
- time / cycle

Figure 12.21



$$\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$

Programs contain a small number of complex instructions. Facilitated in CISC with microcode.

Programs contain a large number of simple instructions. Facilitated in RISC with load/store architecture.

Pipelining.

RISC vs. CISC

- RISC: Reduced Instruction Set Computer
  - Sacrifices instructions / program to decrease cycles / instruction
- CISC: Complex Instruction Set Computer
  - Sacrifices cycles / instruction to decrease instructions / program

Microcode

- Level Mc2
- Used in CISC machines
- Absent in RISC machines

Level Mc2 for Pep/8

- uMem: Microcode ROM memory
- uPC: Microcode program counter
- uIR: Microcode instruction register
- Micro-von Neumann cycle
  - No increment part of the cycle
  - Each micro-instruction contains the address of the next instruction

Figure 12.22

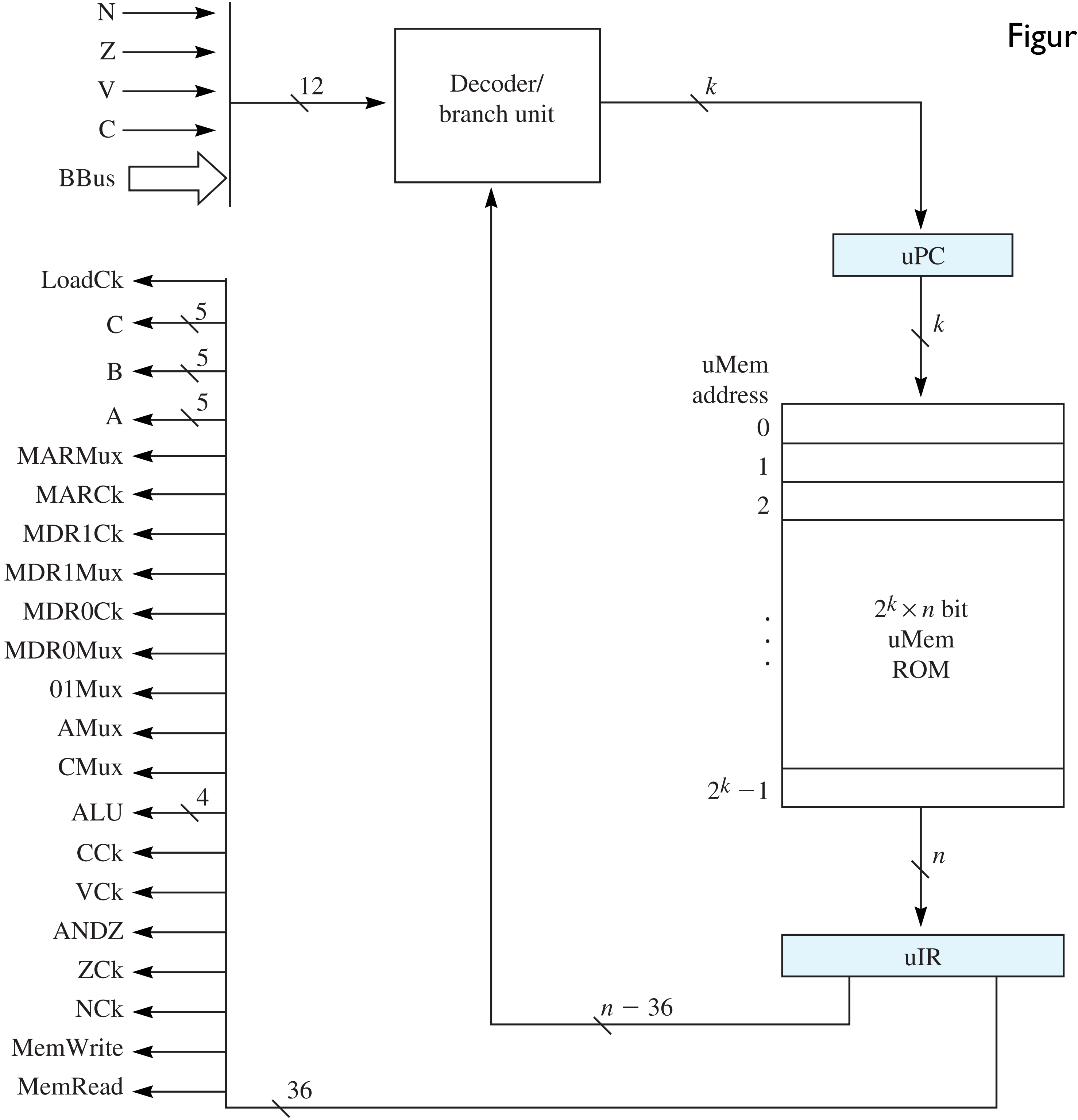
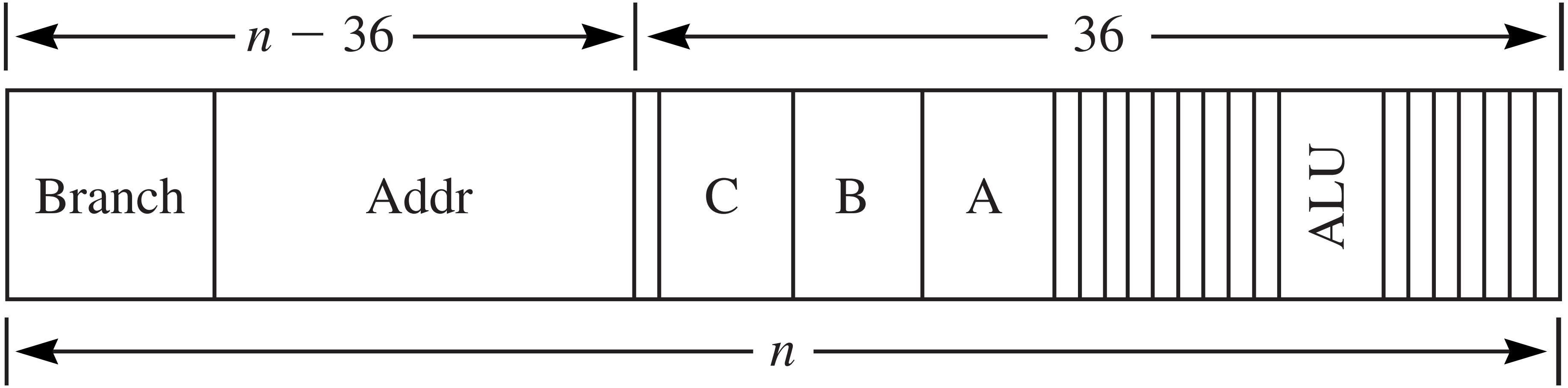
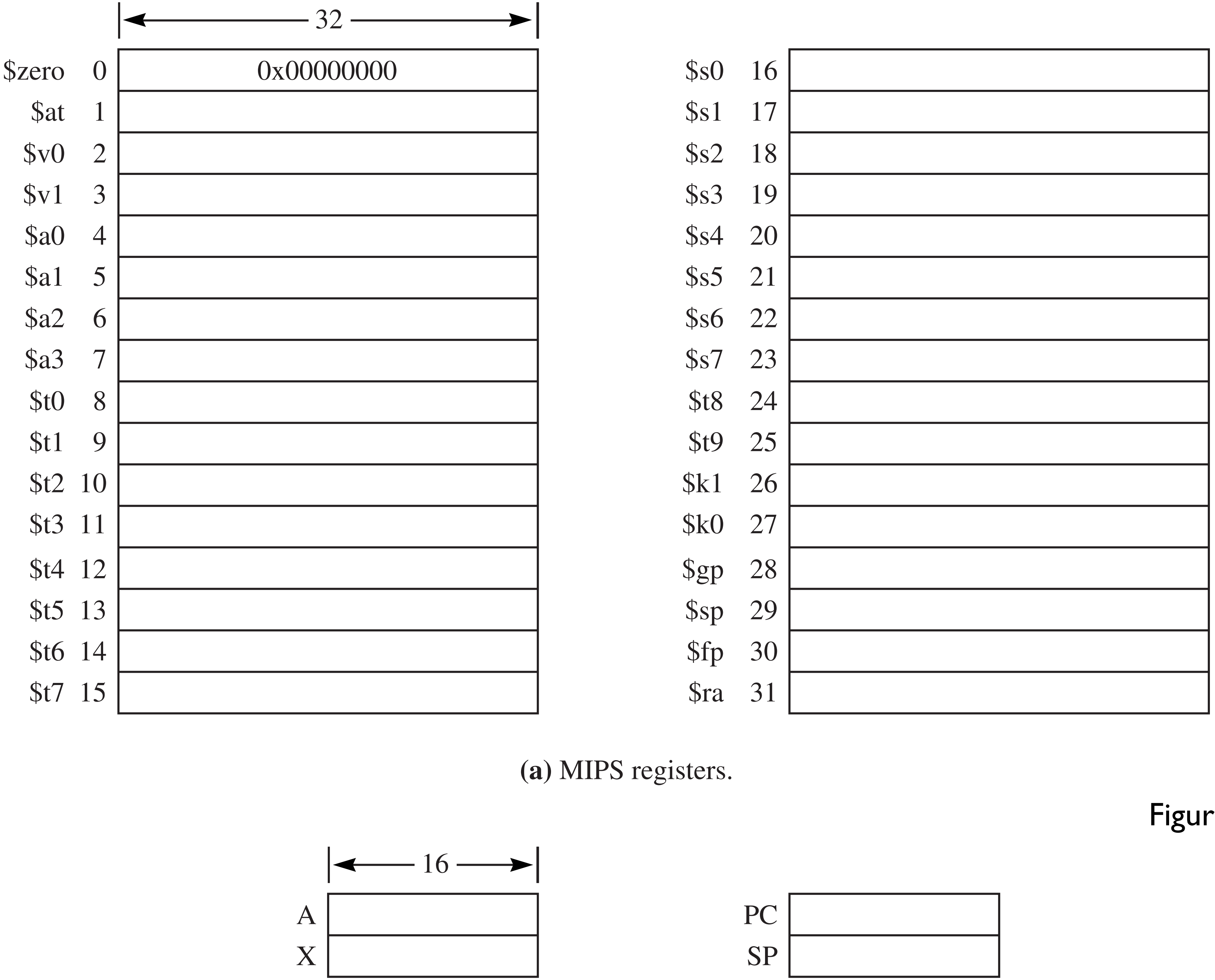

N
Z
V
C
BBus
12
Decoder/
branch unit
k
uPC
k
uMem
address
0
1
2
.
.
.
$2^k \times n$ bit
uMem
ROM
$2^k - 1$
n
uIR
n − 36
36
LoadCk
C
5
B
5
A
5
MARMux
MARCk
MDR1Ck
MDR1Mux
MDR0Ck
MDR0Mux
01Mux
AMux
CMux
ALU
4
CCk
VCk
ANDZ
ZCk
NCk
MemWrite
MemRead

Figure 12.23



Load/store architecture

- Load/store machines are RISC
- Example: 32-bit MIPS machine
- Register bank has 32 32-bit registers



(a) MIPS registers.

Figure 12.24

(b) Pep/8 registers.

MIPS instruction set

- All instructions are exactly 32 bits long
- All binary operations are between two source registers, rs and rt, with the result placed in a destination register, rd
- The *only* instructions that access main memory are the load and store instructions
- *All* instructions except load and store take *one* cycle

Figure 12.25

```
do {
    Fetch the instruction specifier at address in PC
    $PC \leftarrow PC + 4$
    Decode the instruction specifier
    Execute the instruction fetched
}
while (true)
```

Figure 12.26

Addressing mode	Size of OprndSpec	Operand
Immediate	16 bits	OprndSpec
Register	5 bits	Reg [OprndSpec]
Base	5 bits and 16 bits	Mem [Reg[OprndSpec1] + OprndSpec2]
PC-relative	16 bits	Mem [(PC + 4) + OprndSpec * 4]
Pseudodirect	26 bits	Mem [(PC + 4) <0..3> : OprndSpec * 4]

Figure 12.27

Mnemonic	Meaning	Binary instruction encoding
`add`	Add	`0000 00ss ssst tttt dddd d000 0010 0000`
`addi`	Add immediate	`0010 00ss sssd dddd iiii iiii iiii iiii`
`sub`	Subtract	`0000 00ss ssst tttt dddd d000 0010 0010`
`and`	Bitwise AND	`0000 00ss ssst tttt dddd d000 0010 0100`
`andi`	Bitwise AND immediate	`0011 00ss sssd dddd iiii iiii iiii iiii`
`or`	Bitwise OR	`0000 00ss ssst tttt dddd d000 0010 0101`
`ori`	Bitwise OR immediate	`0011 01ss sssd dddd iiii iiii iiii iiii`
`sll`	Shift left logical	`0000 0000 000t tttt dddd dhhh hh00 0000`
`sra`	Shift right arithmetic	`0000 0000 000t tttt dddd dhhh hh00 0011`
`srl`	Shift right logical	`0000 0000 000t tttt dddd dhhh hh00 0010`
`lb`	Load byte	`1000 00bb bbbd dddd aaaa aaaa aaaa aaaa`
`lw`	Load word	`1000 11bb bbbd dddd aaaa aaaa aaaa aaaa`
`lui`	Load upper immediate	`0011 1100 000d dddd iiii iiii iiii iiii`
`sb`	Store byte	`1010 00bb bbbt tttt aaaa aaaa aaaa aaaa`
`sw`	Store word	`1010 11bb bbbt tttt aaaa aaaa aaaa aaaa`

Figure 12.27 (Continued)

beq	Branch if equal to	0001 00ss ssst tttt aaaa aaaa aaaa aaaa
bgez	Branch if greater than or equal to zero	0000 01ss sss0 0001 aaaa aaaa aaaa aaaa
bgtz	Branch if greater than zero	0001 11ss sss0 0000 aaaa aaaa aaaa aaaa
blez	Branch if less than or equal to zero	0001 10ss sss0 0000 aaaa aaaa aaaa aaaa
bltz	Branch if less than zero	0000 01ss sss0 0000 aaaa aaaa aaaa aaaa
bne	Branch if not equal to	0001 01ss ssst tttt aaaa aaaa aaaa aaaa
j	Jump (unconditional branch)	0000 10aa aaaa aaaa aaaa aaaa aaaa aaaa

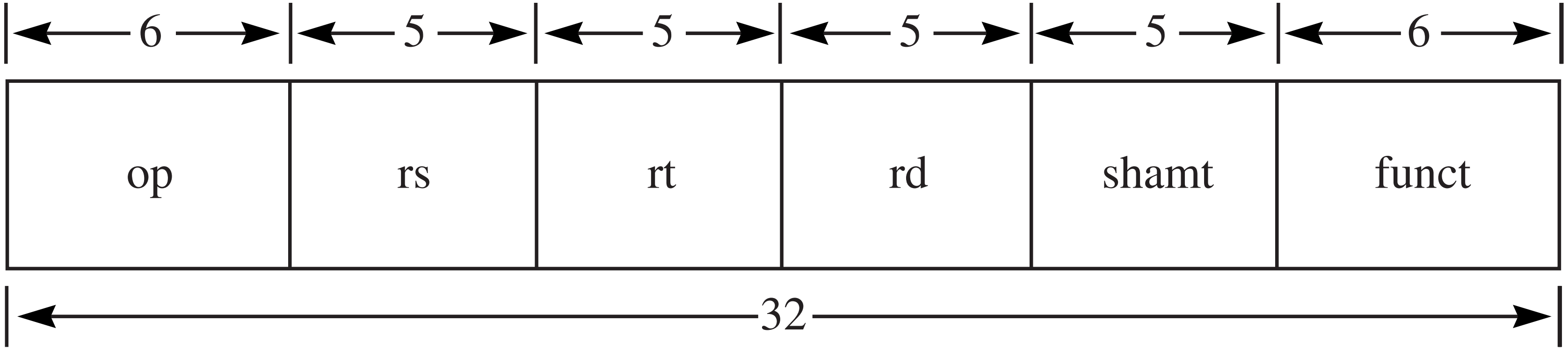
MIPS add instruction

- Uses register addressing
- 32 registers and $2^5 = 32$ implies 5 bits to specify one register
- Assembly language

```
add rd,rs,rt
```

$$\mathrm{rd} \leftarrow \mathrm{rs} + \mathrm{rt}$$

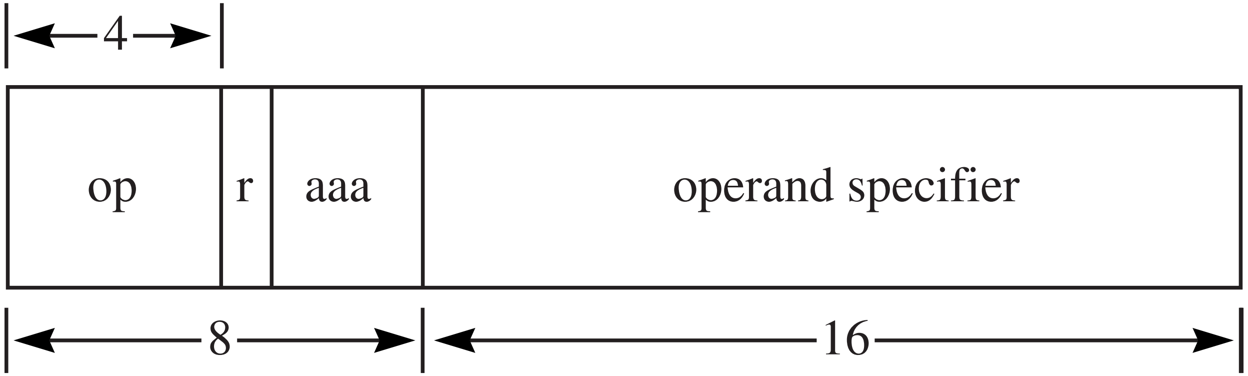
Figure 12.28



6	5	5	5	5	6
op	rs	rt	rd	shamt	funct

32

(a) MIPS register addressing.



4			
op	r	aaa	operand specifier
8			16

(b) Pep/8 general addressing.

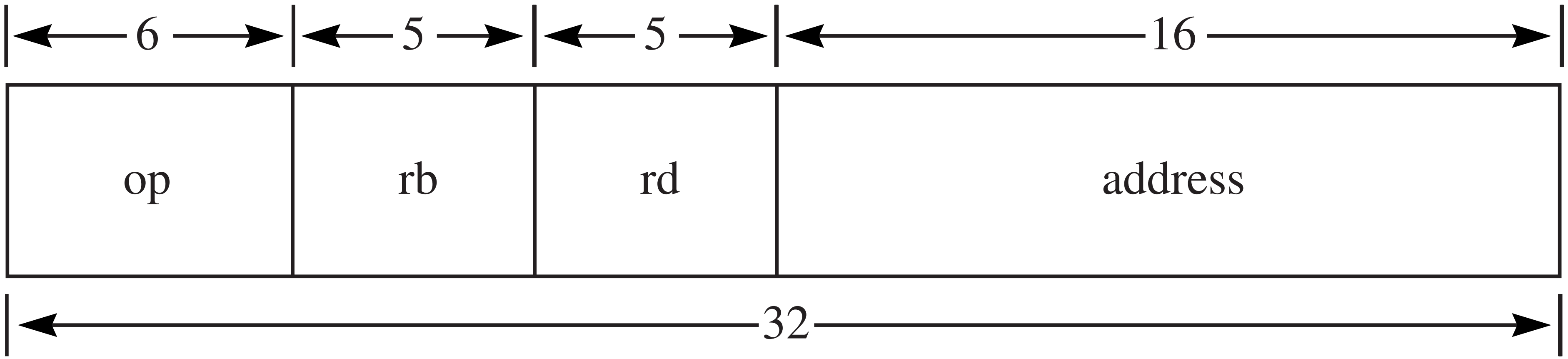
MIPS load instruction

- Uses base addressing
- `address` is a 16-bit signed offset
- Assembly language

```
lw rd,address(rb)
```

$$\mathrm{rd} \leftarrow \mathrm{Mem}[\mathrm{rb} + \mathrm{address}]$$

Figure 12.29



MIPS store instruction

- Uses base addressing
- `address` is a 16-bit signed offset
- Assembly language

```
sw rt,address(rb)
```

$$\mathrm{Mem}[\mathrm{rb} + \mathrm{address}] \leftarrow \mathrm{rt}$$

MIPS arrays

- Use base addressing
- Compiler associates a $s register with an array variable
- It contains the address of the first element of the array

Example 12.1

Compiler associations

```
$s1 → array a
$s2 → variable g
$s3 → array b
```

C++ source code

```
a[2] = g + b[3];
```

MIPS assembly language

Text

```
lw $t0,12($s3)    # Register $t0 gets b[3]
add $t0,$s2,$t0   # Register $t0 gets g + b[3]
sw $t0,8($s1)     # a[2] gets g + b[3]
```

MIPS machine language

```
100011 10011 01000 0000000000001100
000000 10010 01000 01000 00000 100000
101011 10001 01000 0000000000001000
```

Example 12.2

Instruction

Shift left logical 7 bits in register `$s0` and put the results in `$t2`

MIPS assembly language

```
sll $t2,$s0,7
```

MIPS machine language

```
000000 00000 10000 01010 00111 000000
```

Example 12.3

Compiler associations

`$s0` → variable `i`
`$s1` → array `a`
`$s2` → variable `g`

C++ source code

```
g = a[i];
```

MIPS assembly language

```
sll $t0,$s0,2     # $t0 gets $s0 times 4
add $t0,$s1,$t0   # $t0 gets the address of a[i]
lw $s2,0($t0)     # $s2 gets a[i]
```

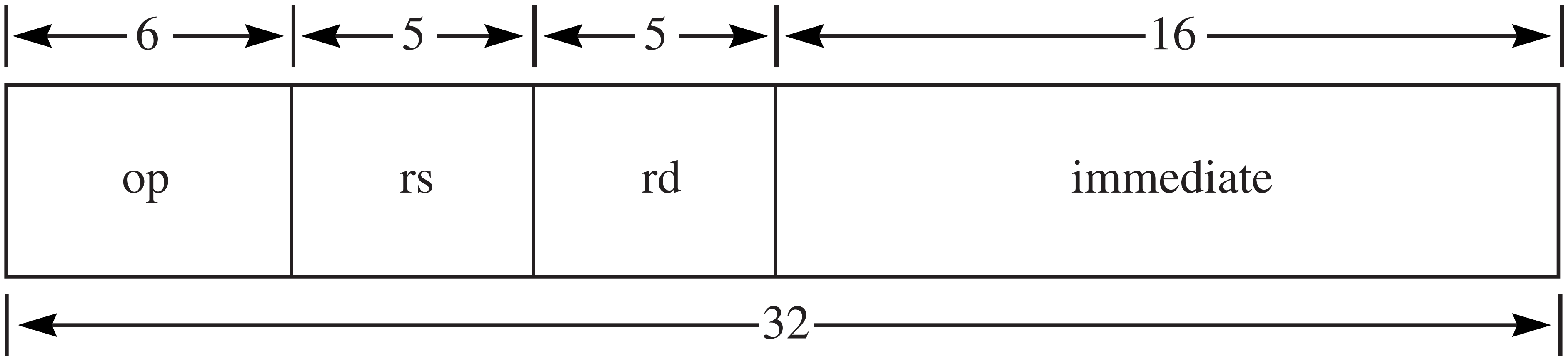
MIPS add immediate

- Uses 16-bit immediate field
- `immediate` is a 16-bit signed field
- Assembly language

```
addi rd,rs,immediate
```

$$\mathrm{rd} \leftarrow \mathrm{rs} + \mathrm{immediate}$$

Figure 12.30



6	5	5	16
op	rs	rd	immediate

32

Example 12.4

Instruction

Allocate four bytes of storage on the run-time stack

MIPS assembly language

```
addi $sp,$sp,-4  # $sp gets $sp - 4
```

MIPS machine language

```
001000 11101 11101 1111111111111100
```

16-bit immediate

- The immediate field is 16 bits
- Registers are 32 bits
- How do you add a 32-bit constant?
  - `lui` Load upper immediate
  - `ori` OR immediate

Example 12.5

Compiler associations

`$s2` → variable `g`

C++ source code

```
g = 491521;
```

MIPS assembly language

```
lui $s2,0x0007
ori $s2,$s2,0x8001
```

Cache memory

- In practice, it takes more than two cycles to fetch data from main memory to the CPU
- Problem: CPU would spend many cycles waiting for memory reads
- Solution: Construct a high-speed memory near the CPU and pre-fetch the data
- The cache has a copy of the data in memory
- Requires that you predict the future

Predicting the future

- Spacial locality — An address close to the previously requested address is likely to be requested in the near future
- Temporal locality — The previously requested address *itself* is likely to be requested in the near future

Cache operation

- When the CPU requests a load from memory it first checks the cache
- A *cache hit* occurs if the requested data is in the cache
  - Can fetch from cache immediately
- A *cache miss* occurs if the requested data is not in the cache
  - Must fetch from memory

Cache levels

- Three levels of cache are common
  - Split L1 instruction and data cache on the CPU chip
  - Unified L2 cache in the CPU package
  - Unified L3 cache on the processor board

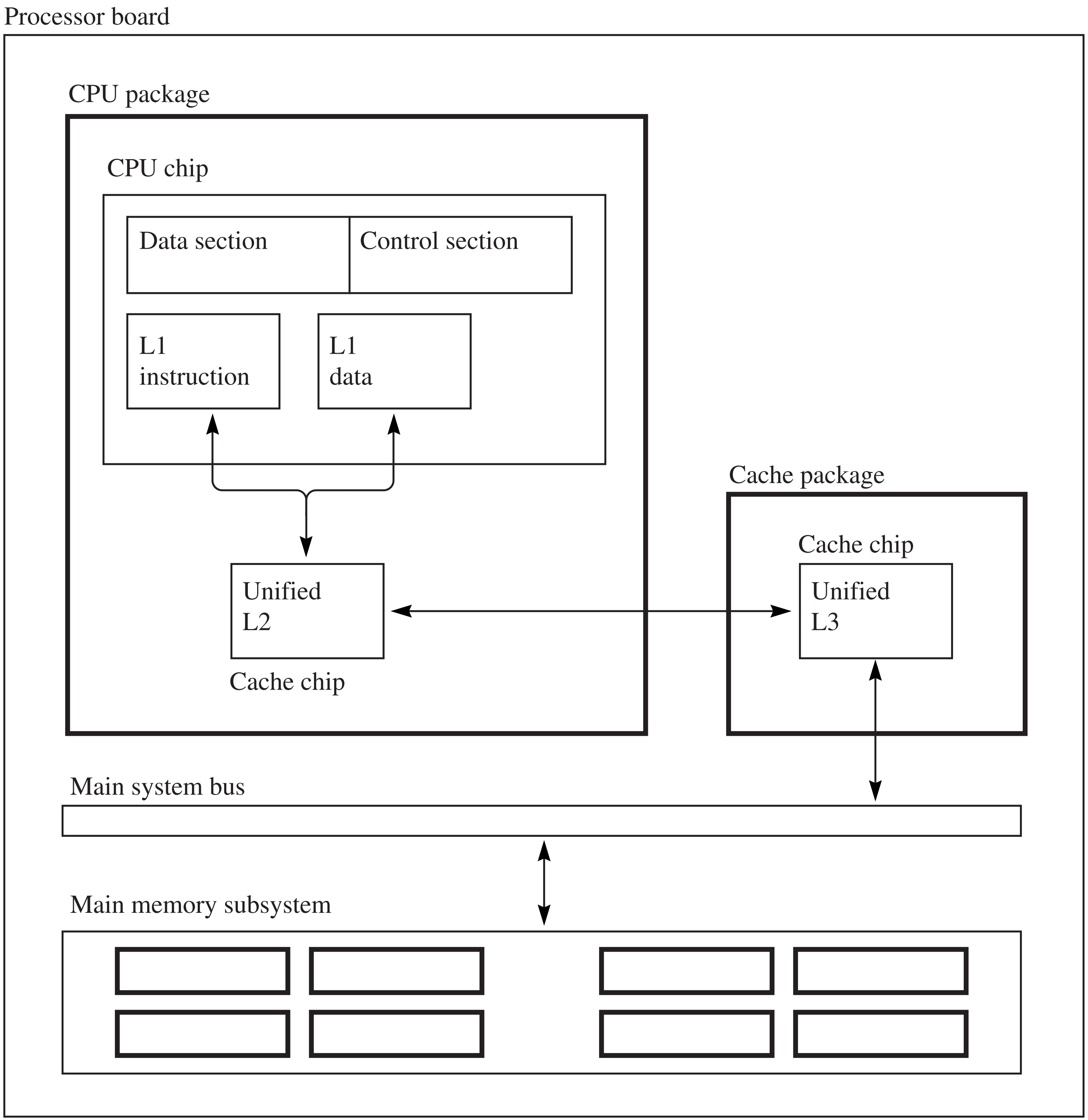

Processor board
CPU package
CPU chip
Data section
Control section
L1
instruction
L1
data
Unified
L2
Cache chip
Cache package
Cache chip
Unified
L3
Main system bus
Main memory subsystem

Figure 12.31

Cache designs

- There are two types of cache designs
  - Direct-mapped cache
  - Set-associative cache

Direct-mapped cache

- Physical address is divided into three fields
  - A high-order tag field
  - A line field
  - A low-order byte field
- Memory is divided into lines or blocks
- When one byte is requested on a cache miss, the entire line containing the byte is loaded into the cache

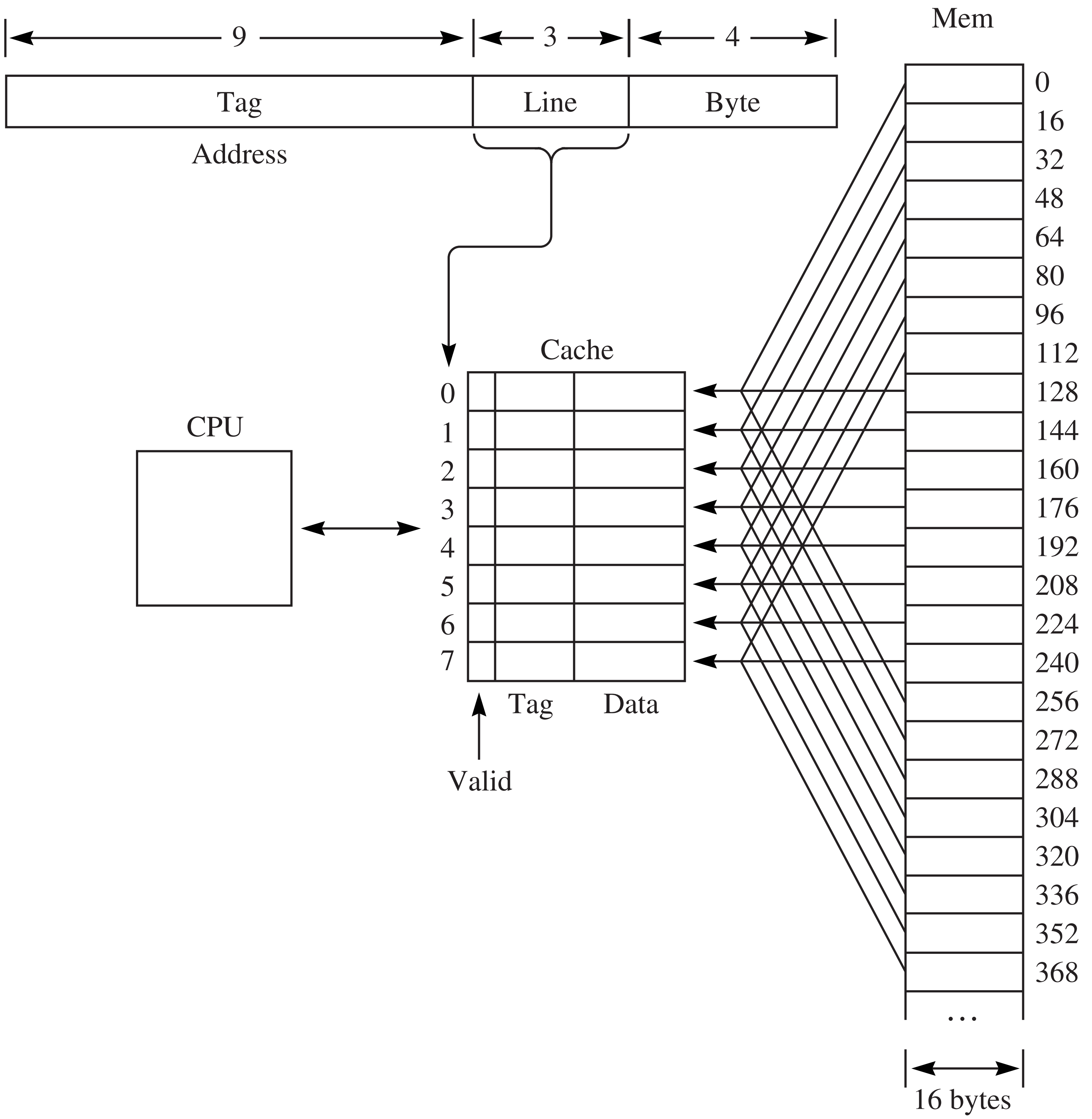

9
3
4
Tag
Line
Byte
Address
Mem
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304
320
336
352
368
...
16 bytes
Cache
0
1
2
3
4
5
6
7
Tag
Data
Valid
CPU

Figure 12.32

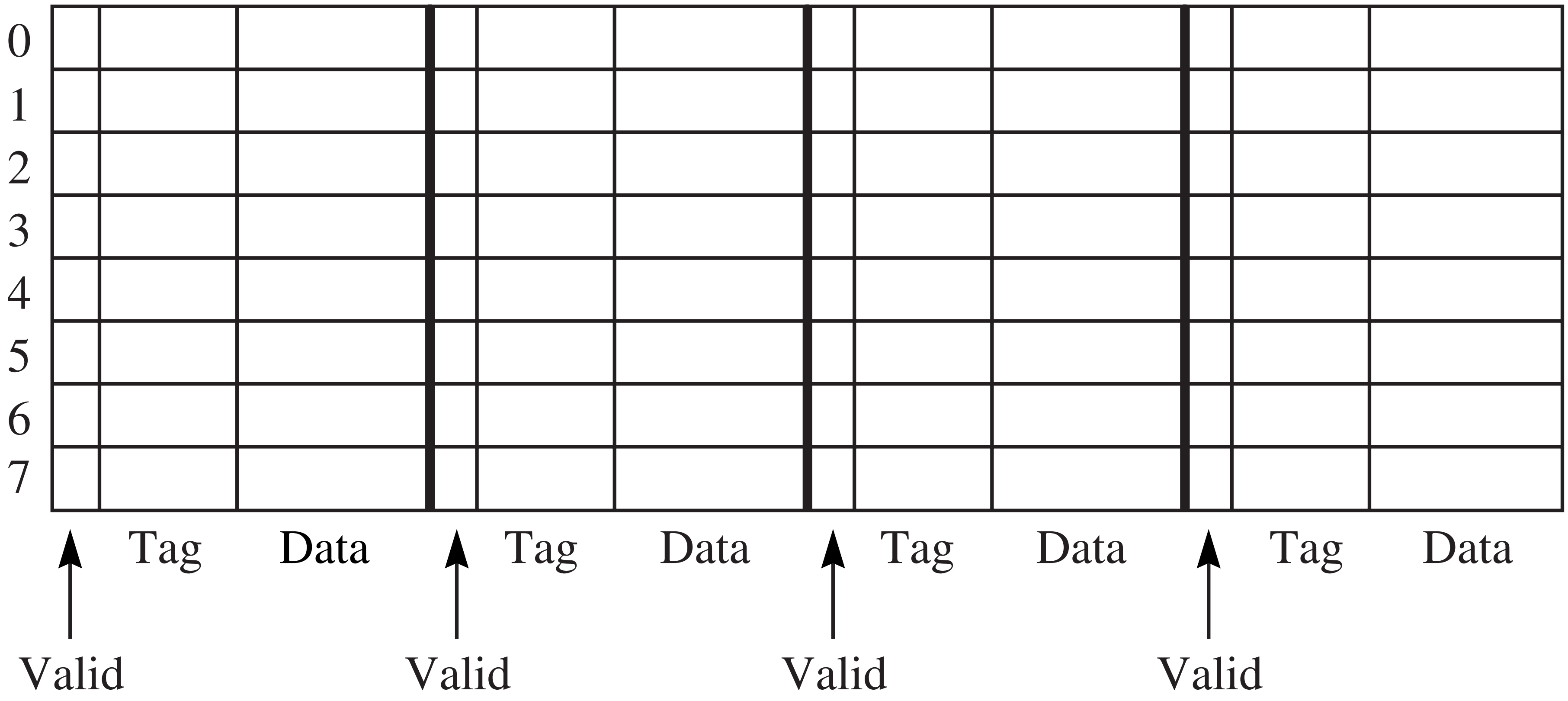
Direct-mapped cache

- It is possible to get a pattern of requests that result in a high cache miss rate
- The program switches back and forth between a low region of memory and a high region that map to the same cache entry
- Example: Pep/8 heap at low region of memory and local pointers on the run-time stack at high region of memory

Set-associative cache

- Increases the cache hit rate
- Each cache entry can hold several lines of data from memory
- A cache that can store up to four lines of data in each cache entry is a *four-way set-associative* cache

Figure 12.33(a)
Block diagram of cache storage.



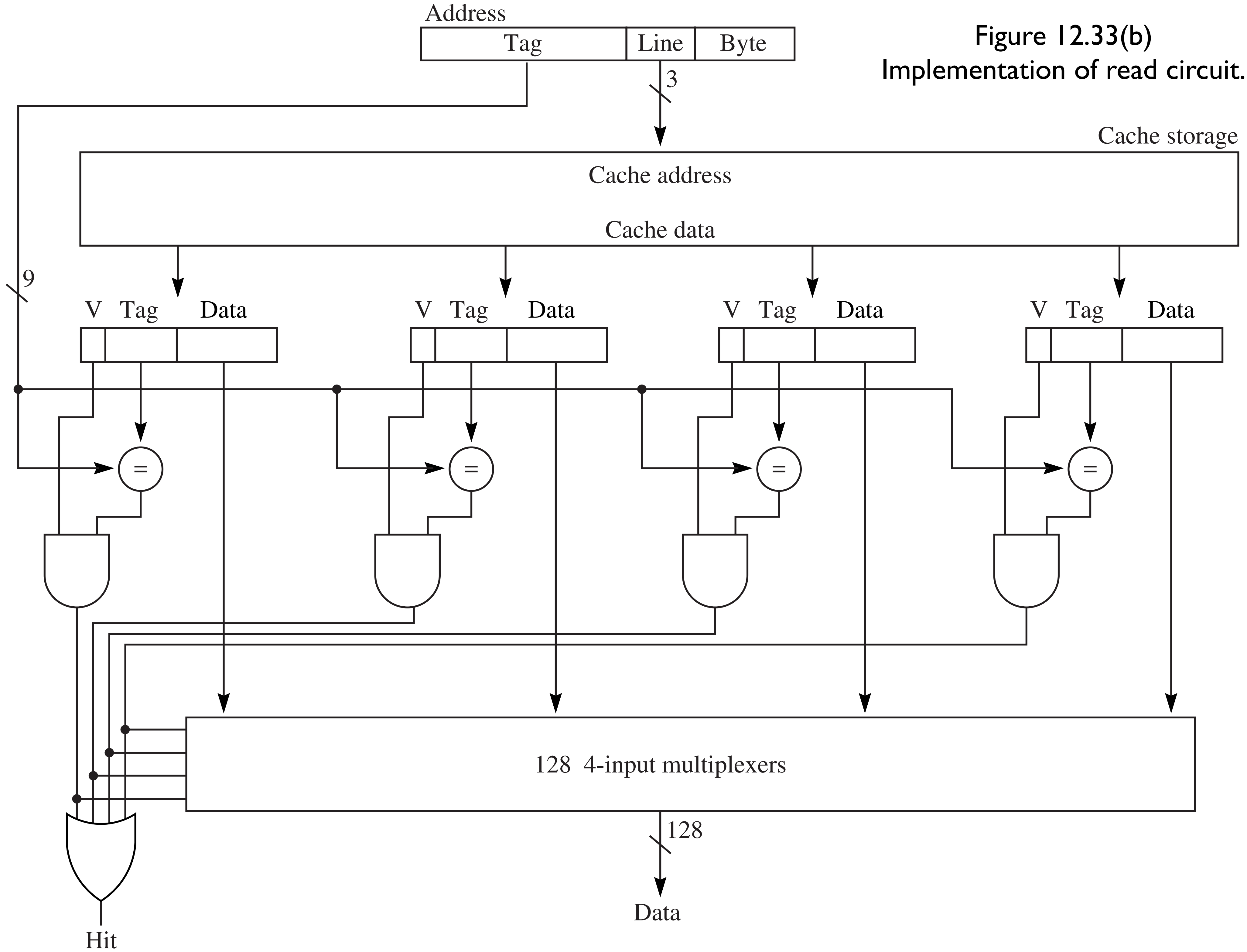


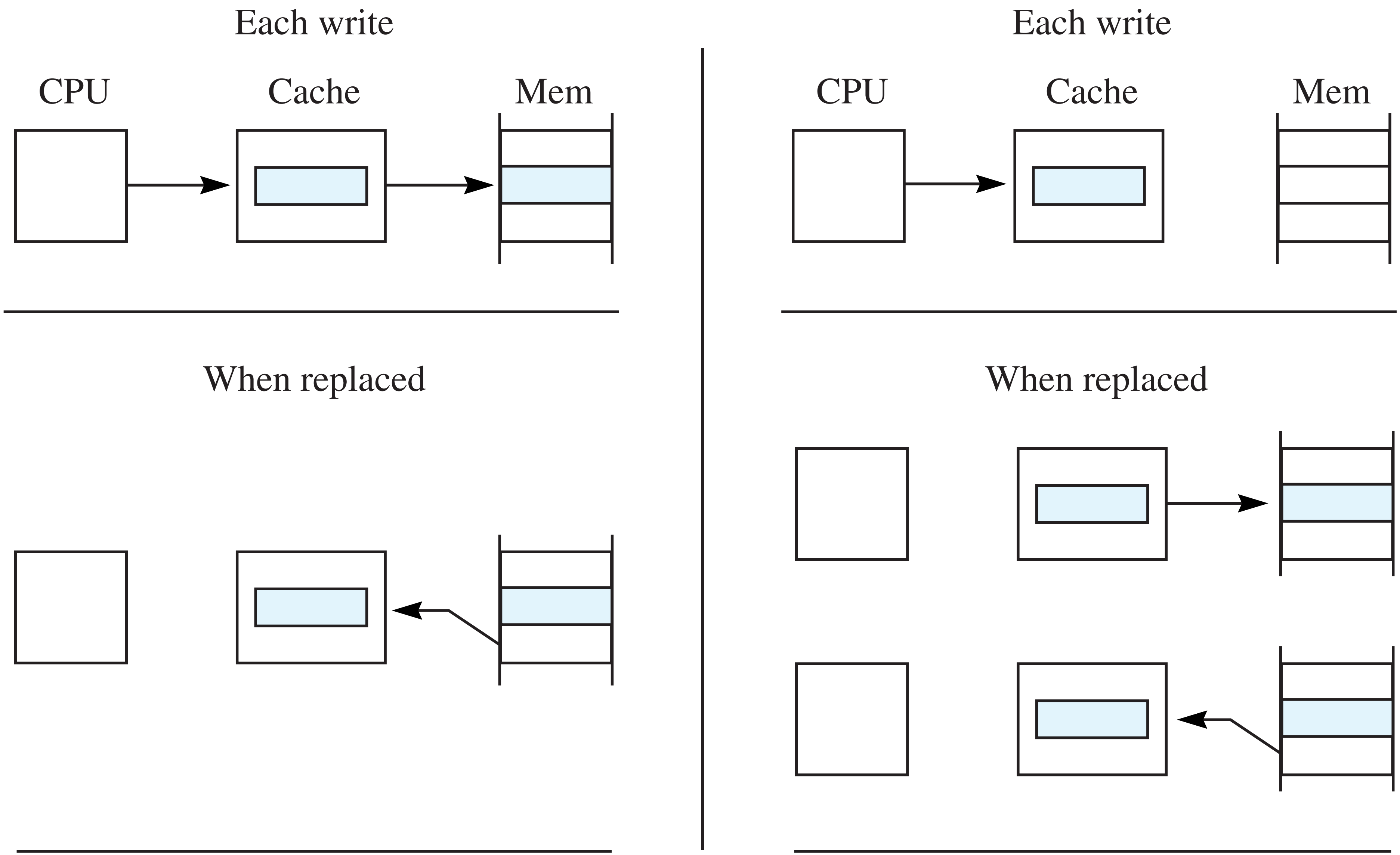
Figure 12.33(b)
Implementation of read circuit.

Replacement policy

- If a cache miss occurs and all parts of the cache are filled, which one is overwritten by the new data?
- Least Recently Used (LRU)
- Requires only one bit per entry with a two-way set associative cache
- Four-way set associative is more difficult, and LRU approximation is common

Cache write policies with cache hits

- Write through
  - Every write request updates the cache and the corresponding block in memory
- Write back
  - A write request only updates the cache copy
  - A write to memory only happens when the cache line is replaced



(a) Write through.

(b) Write back.

Write through

- Simpler design
- When cache line is replaced, memory always has the latest update
- Excessive bus traffic can reduce performance of other devices using the bus

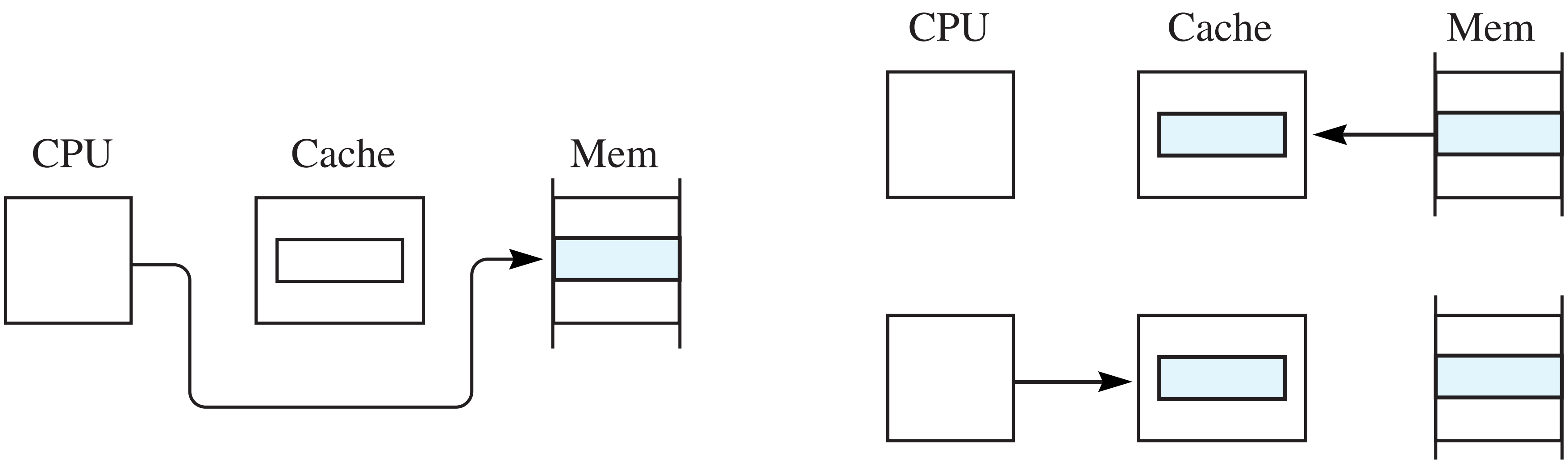
Write back

- Better performance of other devices using the bus, especially when you get a burst of write requests
- There is a delay when the cache line must be replaced, because memory must be updated before the new data can be loaded into the cache

Cache write policies with cache misses

- Without write allocation
  - Bypass the cache altogether
  - Normally used with write through
- With write allocation
  - Load the block from memory into the cache and update the cache line
  - Normally used with write back

Figure 12.35



(a) Without write allocation.

(b) With write allocation.

Memory hierarchies

- Spans two extremes of small high-speed memory and large low-speed memory
- Hardware: *Virtual memory* with small fast main memory and large slow disk
- Hardware: *Cache* with small fast register bank and large slow main memory
- Software: *Hash table* with small fast array and large slow file

MIPS data section

- IF Instruction fetch
- ID Instruction decode / Register file read
- Ex Execute / Address calculation
- Mem Memory access
- WB Write back

Figure 12.36

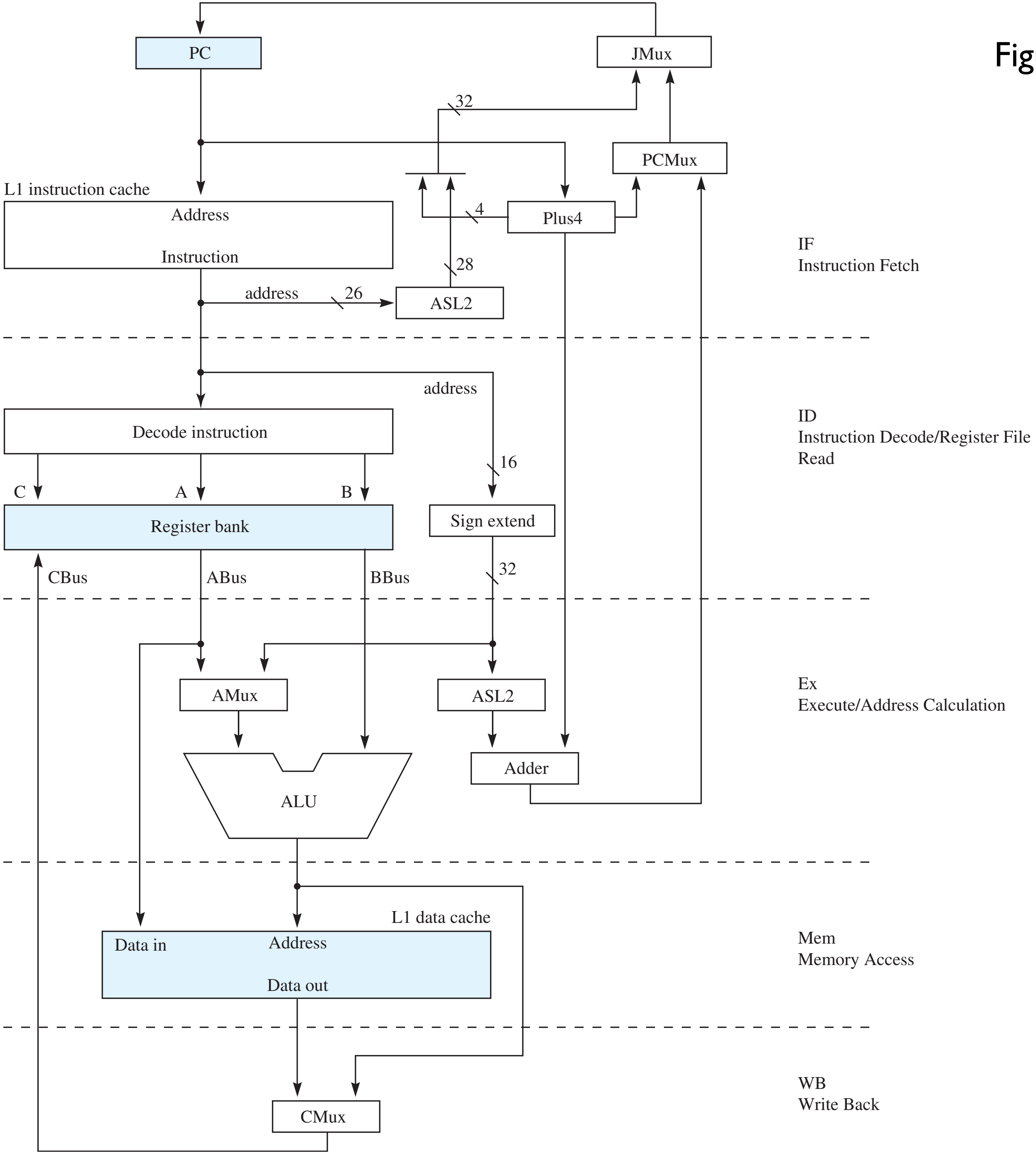
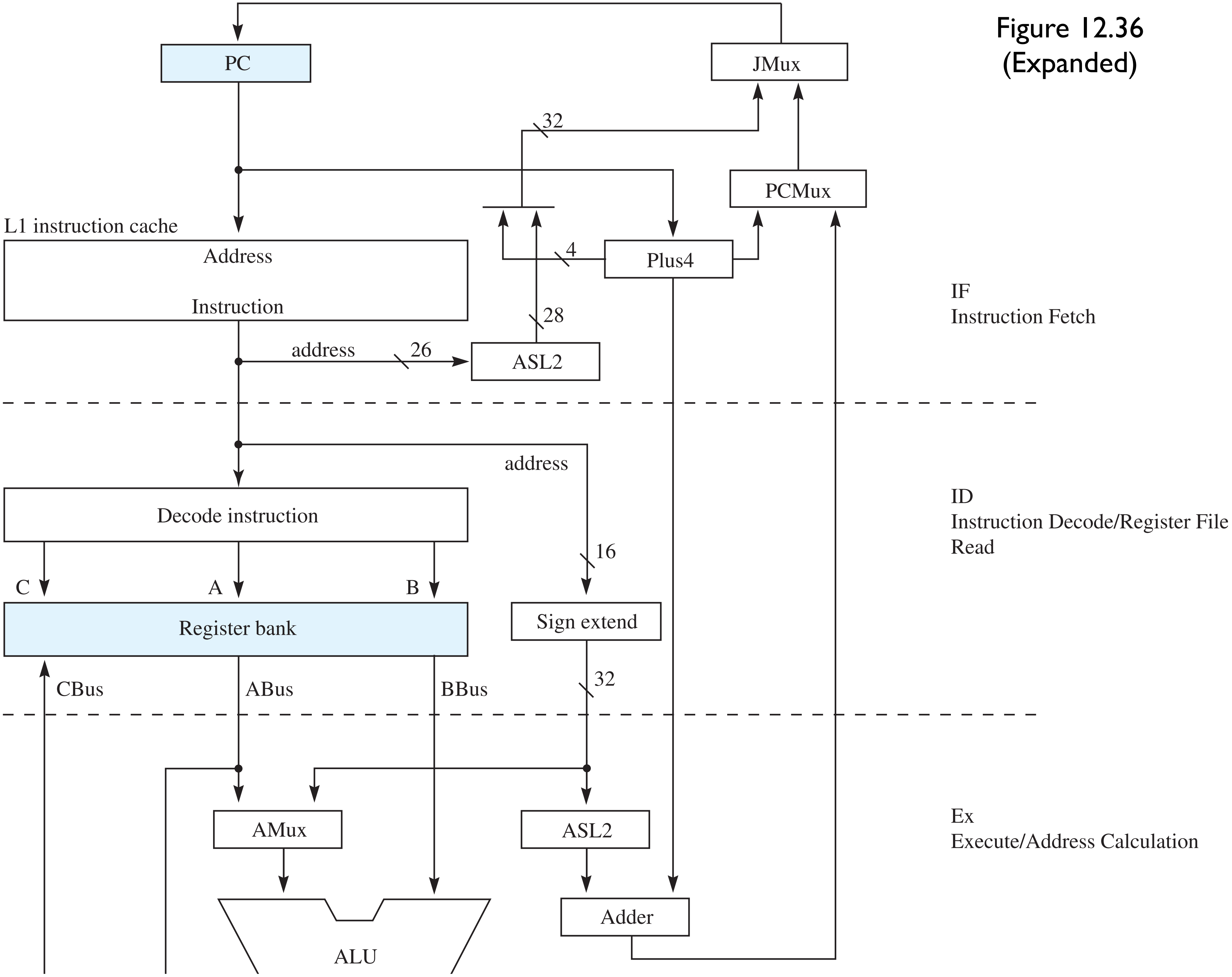

PC
JMux
32
PCMux
L1 instruction cache
Address
Instruction
4
Plus4
28
address
26
ASL2
IF
Instruction Fetch
address
Decode instruction
16
ID
Instruction Decode/Register File
Read
C
A
B
Register bank
Sign extend
CBus
ABus
BBus
32
AMux
ASL2
Ex
Execute/Address Calculation
ALU
Adder
L1 data cache
Data in
Address
Data out
Mem
Memory Access
WB
Write Back
CMux

Figure 12.36
(Expanded)



Instruction fetch

- The program counter (PC) is not a general register in the register bank
- Plus 4 is a special-purpose adder to increment PC by 4
- The two least-significant bits in an address field are assumed to be 00 and not stored in the address field
- An ASL2 box shifts the address field two bits to the left to compensate

Next instruction

- Non-branch instructions
- Immediate, register, and base addressing
- PCMux and JMux route PC + 4 to PC as the address of the next instruction
- Control signals for PC calculation part of the instruction:

```
1. PCMux=0, JMux=1; PCCk
```

Next instruction

- Conditional branch instructions
- PC-relative addressing
- 16-bit address field sign extended to 32 bits is shifted left two bits and added to PC + 4 as the address of the next instruction
- Control signals for complete instruction:

```
1. PCMux=1, JMux=1; PCCk
```

Next instruction

- Unconditional jump instruction
- Pseudodirect addressing
- 26-bit address field shifted left two bits and concatenated on the right with the four high-order bits of PC + 4 as the address of the next instruction
- Control signals for complete instruction:

```
1. JMux=0; PCCk
```

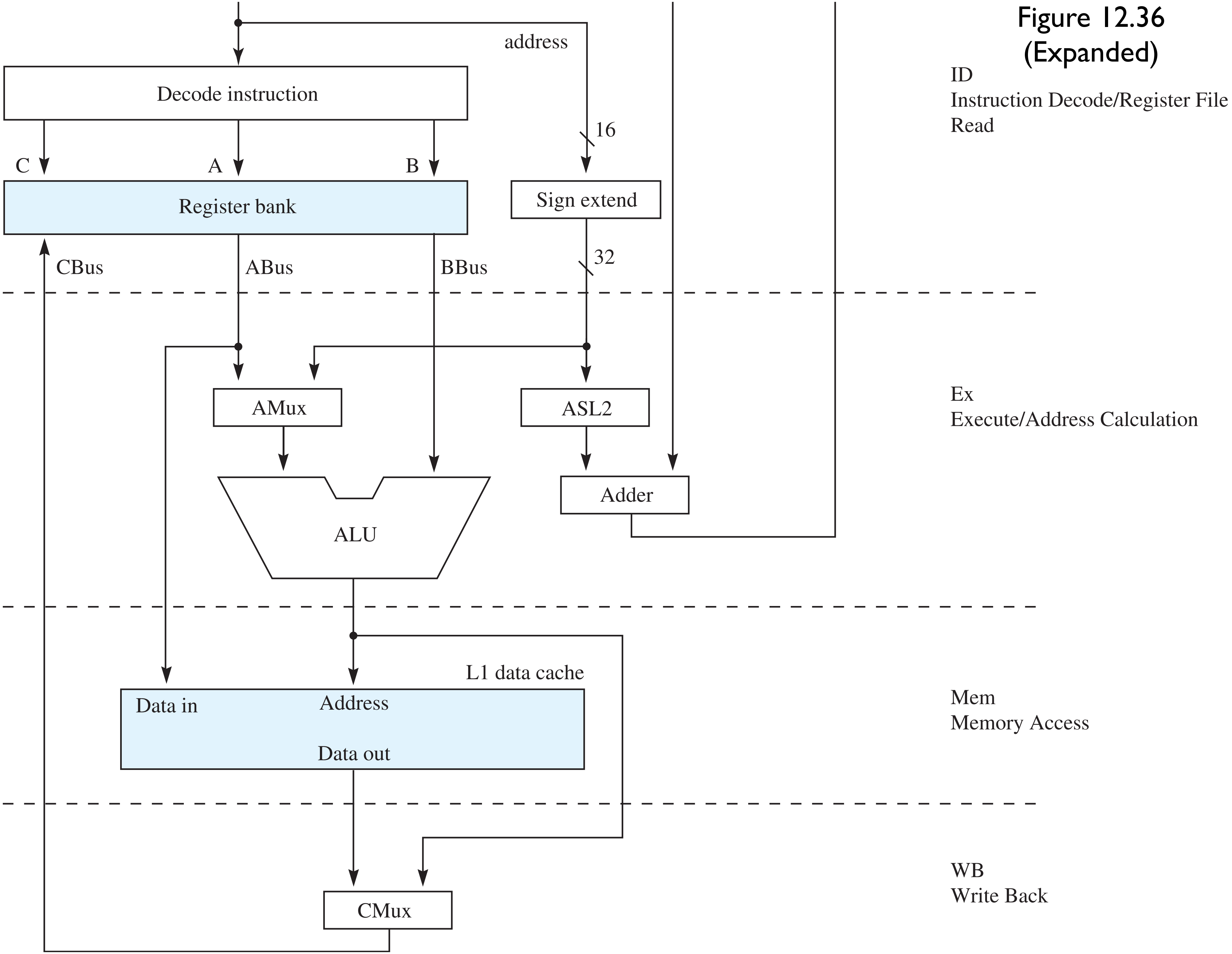


Figure 12.36 (Expanded)

Instruction decode

- The Register bank contains the 32 32-bit MIPS registers
- The Decode instruction box sets the 5-bit A, B, and C register address lines
- Some control signals (not shown) originate from the Decode instruction box

Store word

- Mnemonic: **`sw`**
- RTL: $\mathrm{Mem}[\mathrm{rb} + \mathrm{address}] \leftarrow \mathrm{rt}$
- Control signals:

```
1. PCMux=0, JMux=1, A=rt, AMUX=1, B=rb,
ALU=A plus B; PCCk, DCCk
```

Add

- Mnemonic: **`add`**
- RTL: $\mathrm{rd} \leftarrow \mathrm{rs} + \mathrm{rt}$
- Control signals:

```
1. PCMux=0, JMux=1, A=rs, AMux=0, B=rt,
ALU=A plus B, CMux=1, C=rd; PCCk, LoadCk
```

Signal propagation

- IF: The instruction cache, Plus4, ASL2, PCMux, JMux
- ID: The Decode instruction box, the Register bank, the sign extend box
- Ex: AMux, ASL2, the ALU, the Adder
- Mem: The data cache
- WB: CMux, the address decoder of the Register bank

Figure 12.36

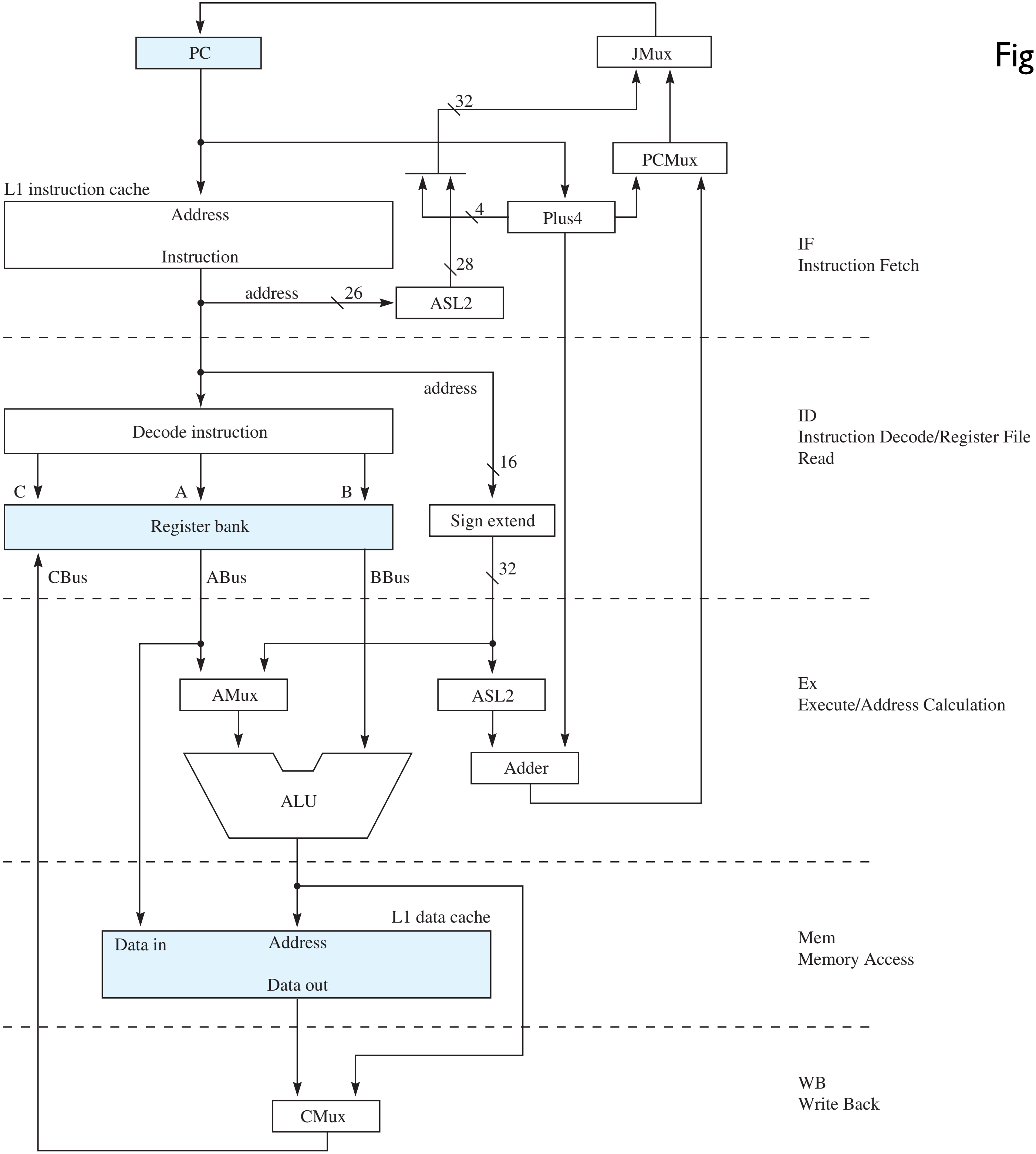
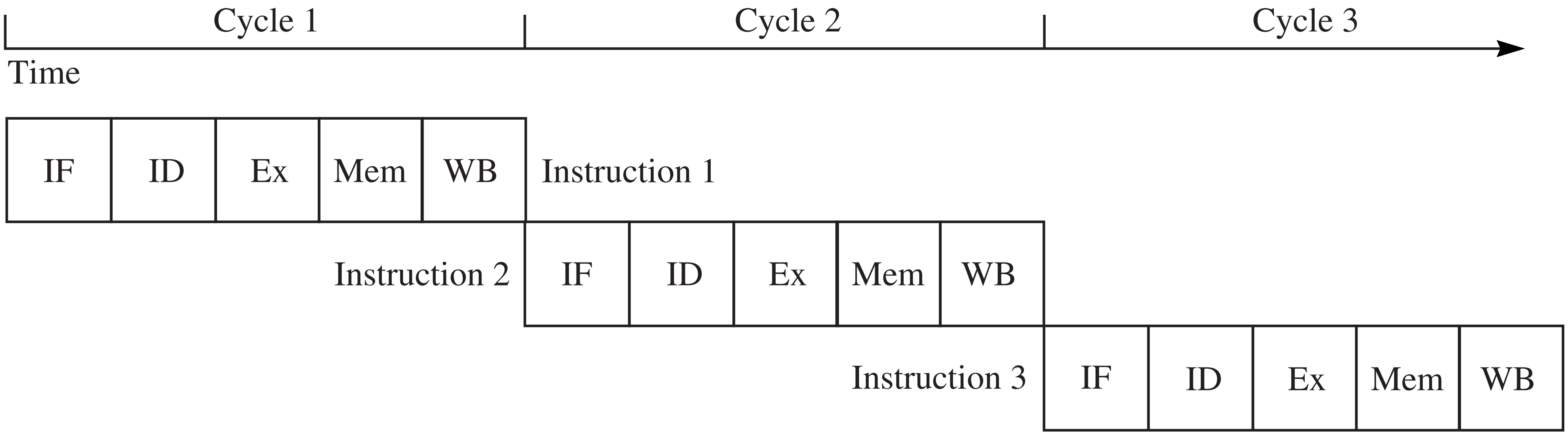

PC
JMux
32
PCMux
L1 instruction cache
Address
Instruction
4
Plus4
28
address
26
ASL2
IF
Instruction Fetch
address
Decode instruction
16
ID
Instruction Decode/Register File
Read
C
A
B
Register bank
Sign extend
CBus
ABus
BBus
32
AMux
ASL2
Ex
Execute/Address Calculation
ALU
Adder
L1 data cache
Data in
Address
Data out
Mem
Memory Access
CMux
WB
Write Back

Figure 12.37



Cycle 1 Cycle 2 Cycle 3

Time

IF ID Ex Mem WB Instruction 1

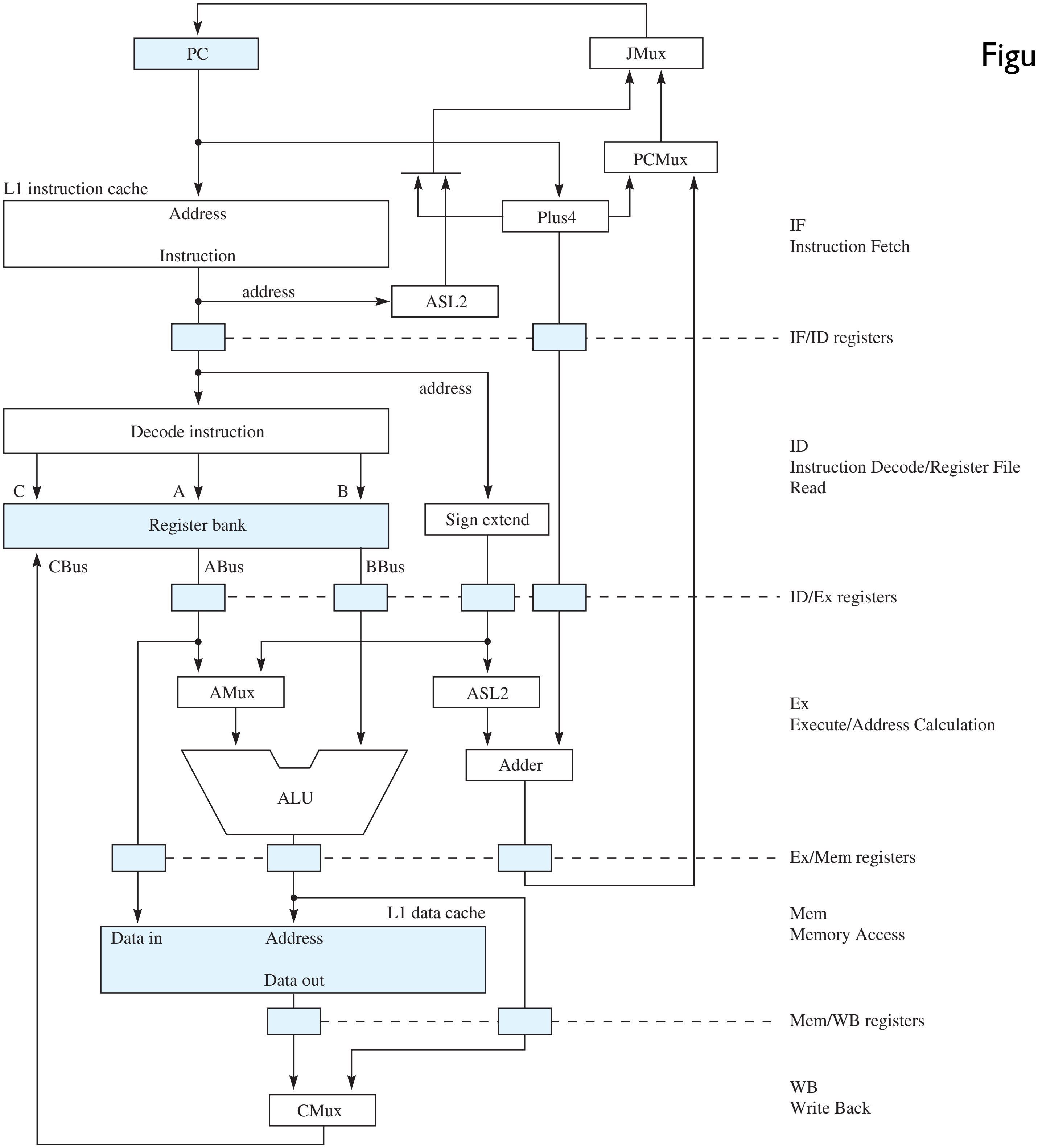
Instruction 2 IF ID Ex Mem WB

Instruction 3 IF ID Ex Mem WB

Pipelining

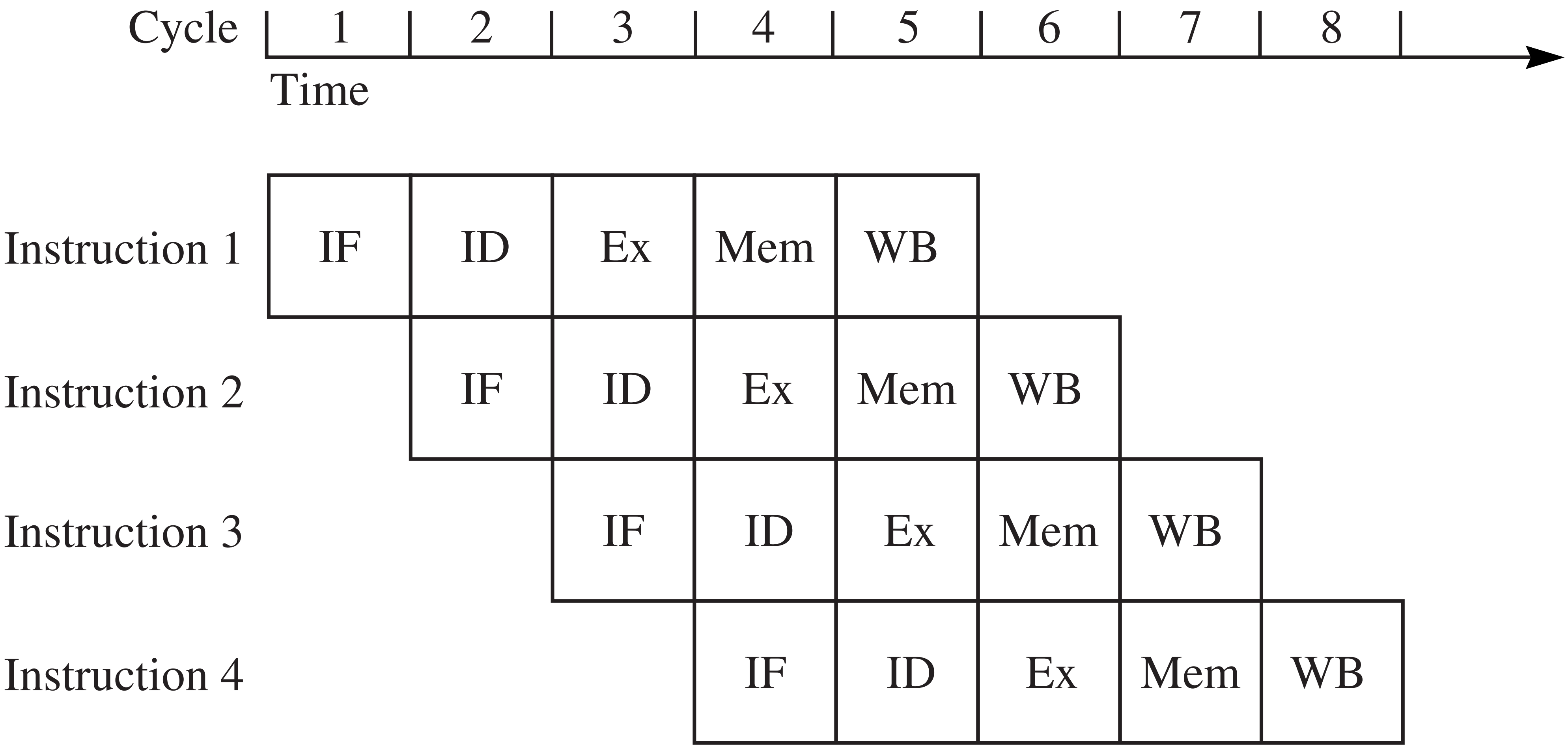
- Assembly line analogy
- Overlap the IF, ID, Ex, Mem, and WB stages
- Allows a decrease in the cycle time (an increase in the MHz rating)
- Provides parallelism in the CPU circuitry
- Requires boundary registers to store the intermediate results between stages

Figure 12.38



PC
JMux
PCMux
Plus4
L1 instruction cache
Address
Instruction
address
ASL2
IF
Instruction Fetch
IF/ID registers
address
Decode instruction
ID
Instruction Decode/Register File
Read
C
A
B
Register bank
Sign extend
CBus
ABus
BBus
ID/Ex registers
AMux
ASL2
Ex
Execute/Address Calculation
ALU
Adder
Ex/Mem registers
L1 data cache
Mem
Memory Access
Data in
Address
Data out
Mem/WB registers
WB
Write Back
CMux

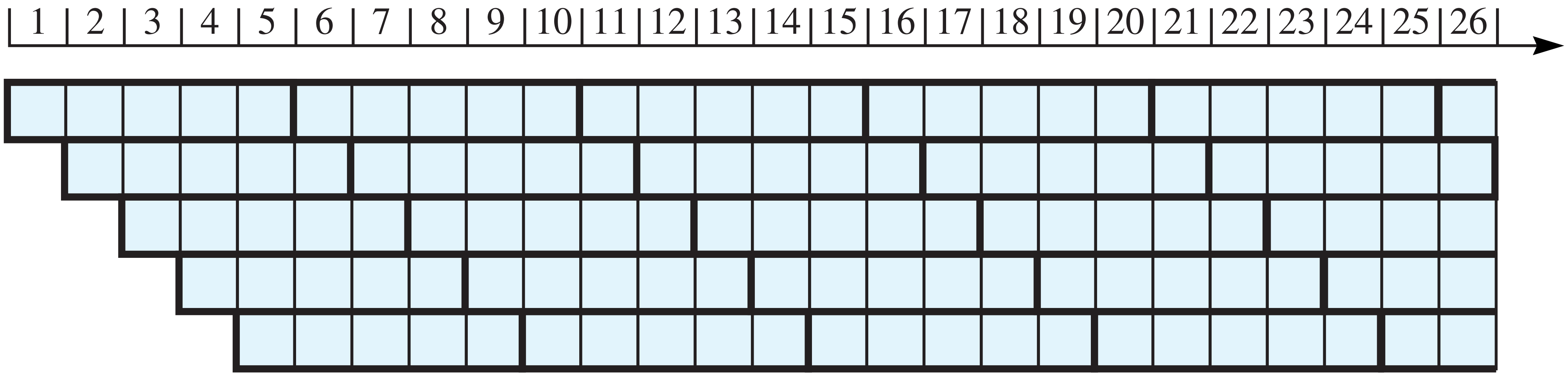
Figure 12.39



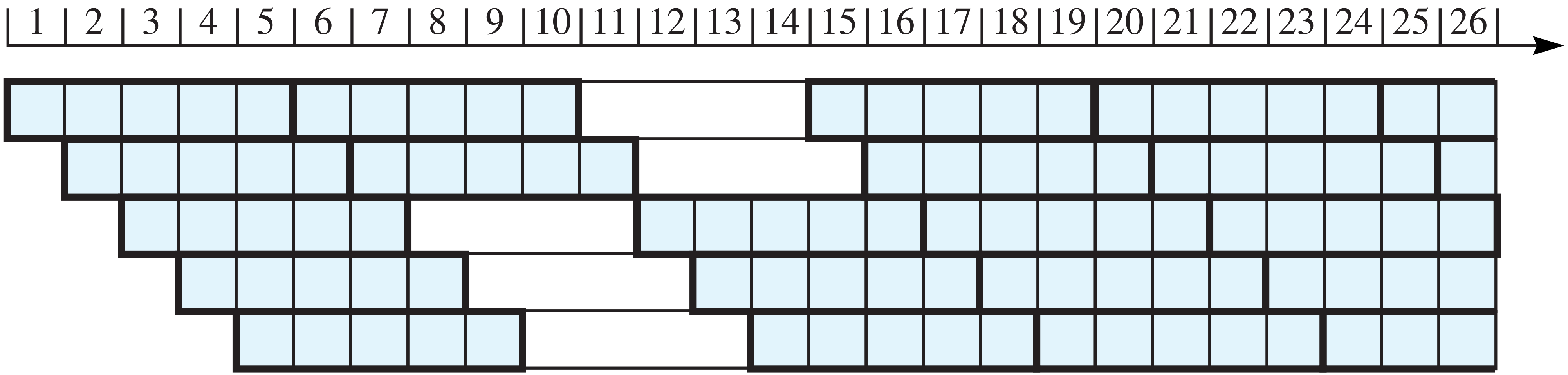
Cycle	1	2	3	4	5	6	7	8
Instruction 1	IF	ID	Ex	Mem	WB			
Instruction 2		IF	ID	Ex	Mem	WB		
Instruction 3			IF	ID	Ex	Mem	WB	
Instruction 4				IF	ID	Ex	Mem	WB

Hazards

- Control hazards from unconditional and conditional branches
- Data hazards from data dependencies between instructions
- A hazard causes the instruction that cannot continue to stall, which creates a *bubble* in the pipeline that must be flushed out before peak performance can be resumed



(a) No hazards.



(b) A branch hazard.

Figure 12.41

Instruction	Frequency
Arithmetic	50%
Load/Store	35%
Branch	15%

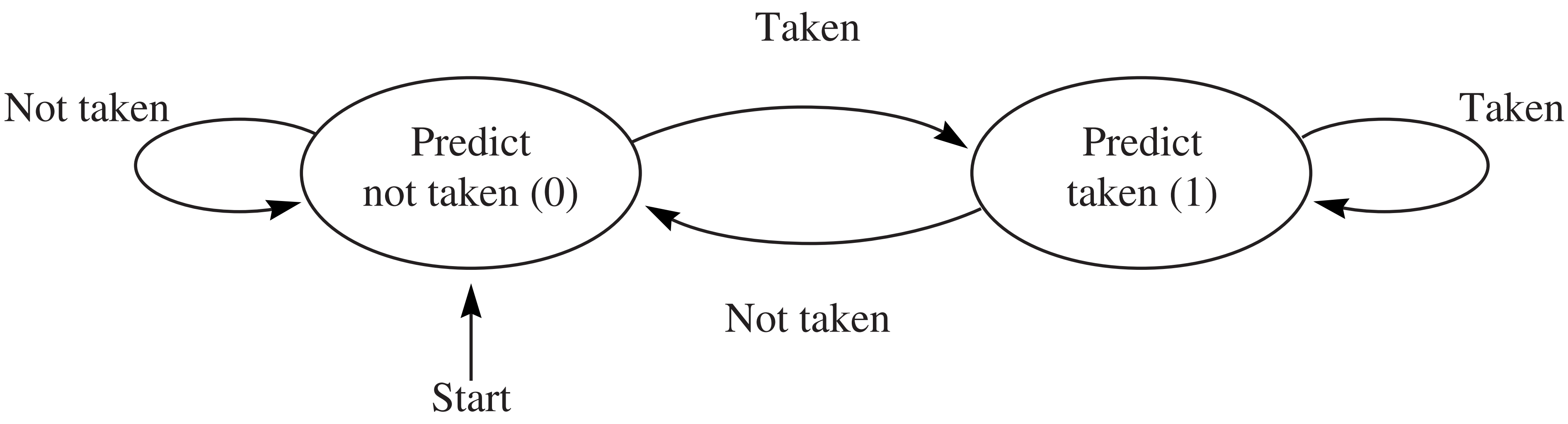
Speedup techniques

- Eliminate the Write back stage of the branch instructions
- Conditional branches
  - Assume branch will not be taken
  - Process the following instructions until you know for sure
  - If you predict right, no wasted cycles

Dynamic branch prediction

- The higher the percentage of the time your prediction is correct, the better the performance
- Use one bit to store whether the branch was taken previously
- Predict that the branch will be taken this time if it was taken the previous time

One-bit dynamic branch prediction



Nested loop pattern

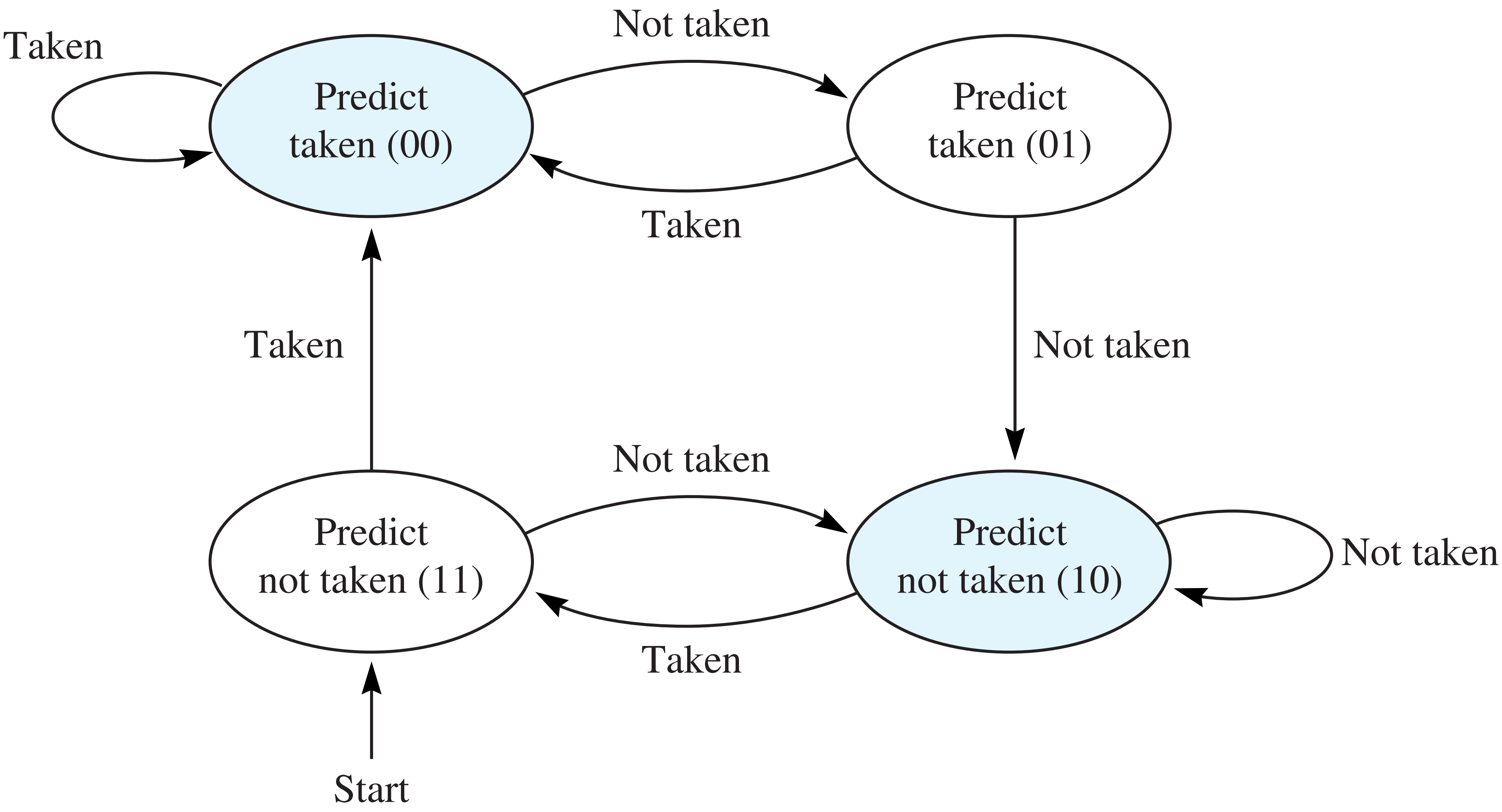
One-bit dynamic branch prediction

Taken:	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N
Prediction:	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y
Incorrect:	x				x	x				x	x				x	x				x

Two-bit dynamic branch prediction

- If you have a run of branches taken and you encounter one branch not taken, do not change your prediction right away
- To change your prediction, you must get two consecutive identical branch types

Two-bit dynamic branch prediction



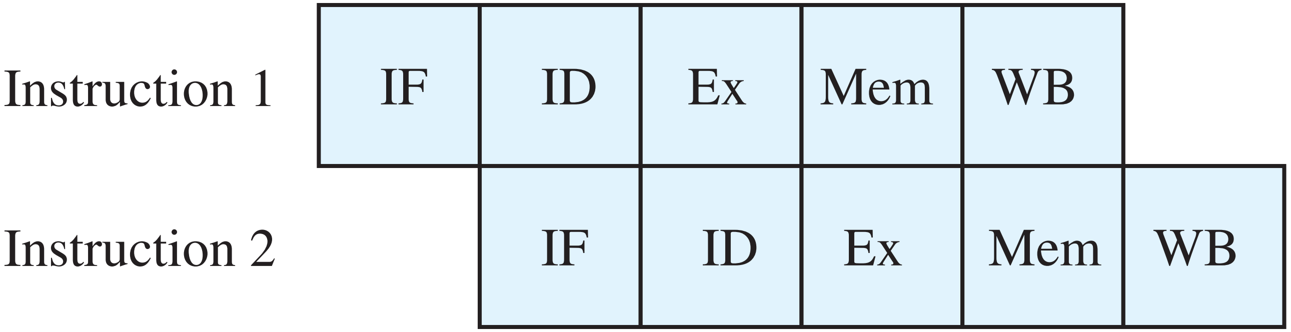
Duplicate pipelines

- One pipeline assuming the branch is taken
- One pipeline assuming the branch is not taken
- When you find out which pipeline is valid discard the other pipeline

Data hazard

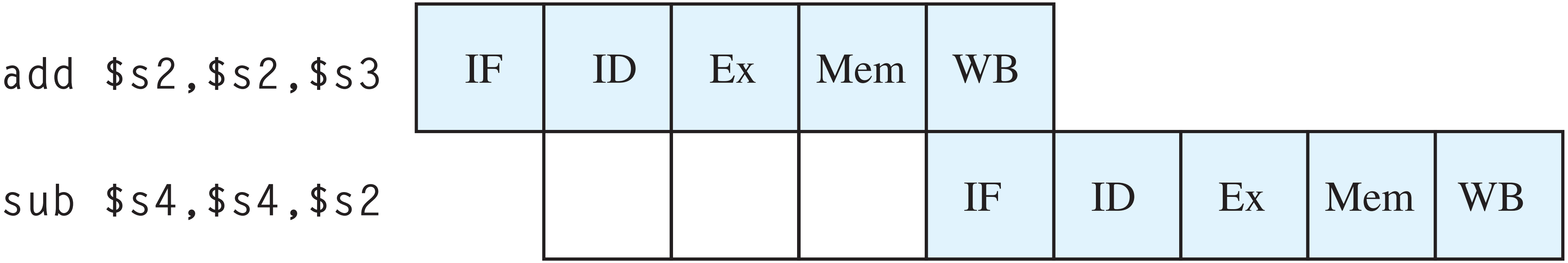
- One instruction needs the result of a previous instruction
- Must stall until it gets the result
- Called *read-after-write* (RAW) hazard

Figure 12.44



Instruction 1	IF	ID	Ex	Mem	WB	
Instruction 2		IF	ID	Ex	Mem	WB

(a) Consecutive instructions without a RAW hazard.



`add $s2,$s2,$s3`	IF	ID	Ex	Mem	WB				
`sub $s4,$s4,$s2`					IF	ID	Ex	Mem	WB

(b) Consecutive instructions with a RAW hazard.

Instruction reordering

- Find some later instructions that can be executed out of order with no change to the results of the program
- Execute them after the conditional branch
- Each instruction that can be executed out of order decreases the bubble penalty by one cycle

Abstraction versus performance

- The compiler must help construct the assembly language program knowing the lower-level details of the pipeline
- Adds complexity to the compiler, but gains performance

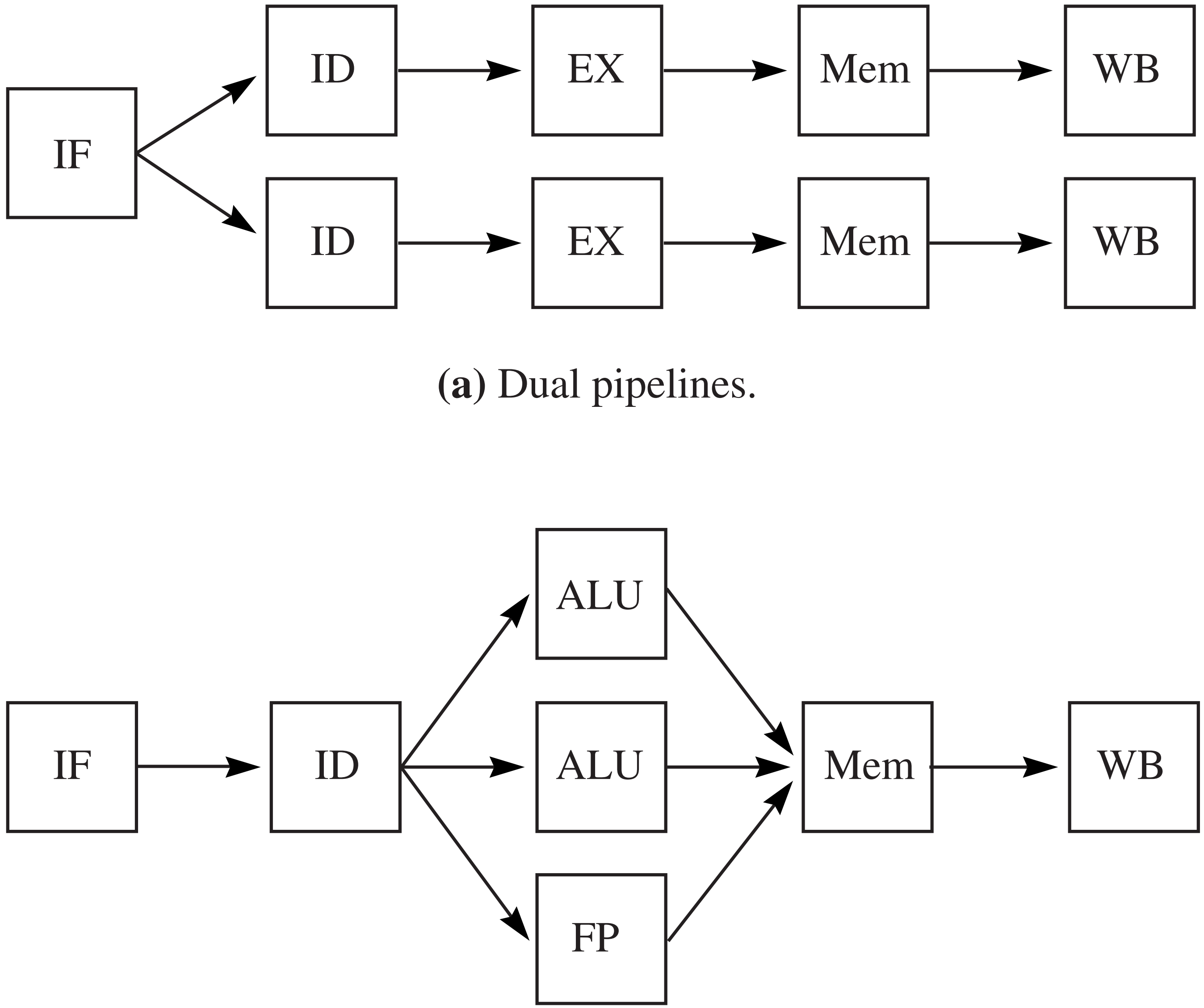
Data forwarding

- For RAW hazard of Figure 12.44(b)
- Construct a data path from Ex/Mem ALU register to a special ID/Ex register
- Reduces the bubble penalty in Figure 12.44(b) from three cycles to one

Superscalar machines

- Based on the fact that two instructions with no data dependencies can execute in parallel
- Two approaches
  - Multiple pipelines
  - Multiple execution units
- Out-of-order execution inherent in both approaches

Figure 12.45



(a) Dual pipelines.

(b) Multiple execution units.

Write-after-read hazard

- Write-after-read hazard (WAR) occurs when one instruction writes to a register after the previous instruction reads it
- Example

```
add $s3,$s3,$s2  # read $s2
sub $s2,$s4,$s5  # write $s2
```

- Only a potential problem with out-of-order execution

The megahertz myth

- Of two different machines with two different megahertz ratings, the one with the higher rating is the one with better performance
- Only true if you compare *identical machines* with two different megahertz ratings
- Cycles per second must be multiplied by work done per cycle
- The myth is valuable for advertising

Simplifications

- Edge-triggered flip-flops are more common than master-slave
- Timing constraints on buses are more severe and bus protocols are more complex
- Input/Output subsystems are more complex

Direct memory access

- A DMA controller, instead of the CPU, controls the bus
- Data can flow between disk and memory in parallel with the CPU performing computation work for applications
- Requires an arbitration protocol when two devices want to use the bus at the same time and a system of interrupt priority levels

Finite state machines

- Finite state machines are the basis of all computing
  - The Turing machine is a finite state machine with an infinite tape
  - FSMs are the basis of lexical analysis
  - A computer is one big FSM with its state stored in the flip-flops of its circuitry

Abstraction

- Simplicity is the key to harnessing complexity
- It is possible to construct a machine as complicated as a computer only because of the simple concepts that govern its behavior at every level of abstraction