CHAPTER 8 O.S. Support

Operating System

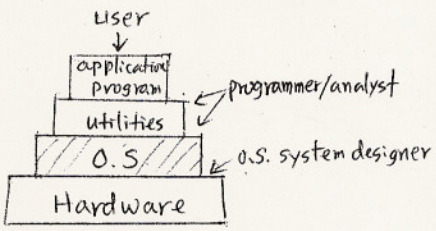
- systems programs
- manages computer system resources (process, memory, device, file)

Function

{ convenience to the users
{ efficient resource management

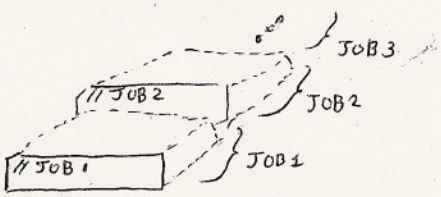
O.S. services

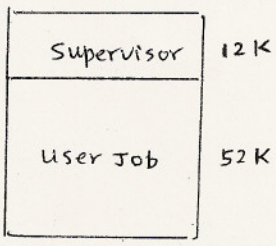
- o program creation – editor, debugger
- o program execution
- o access to I/O devices (Ex. int 21h)
- o access to files
- o system access
- o error detection and response
- o accounting



Types of O.S.

(1) Batch – IBM S/360





cpu utilization is low – cpu wait for I/O completion

(2) Multiprogramming – 1970's

- multiple jobs (compute-bound, I/O bound)
  - . job scheduling (process scheduling)
  - . memory management

(3) Time-sharing

Table 8.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50M	100M	80M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

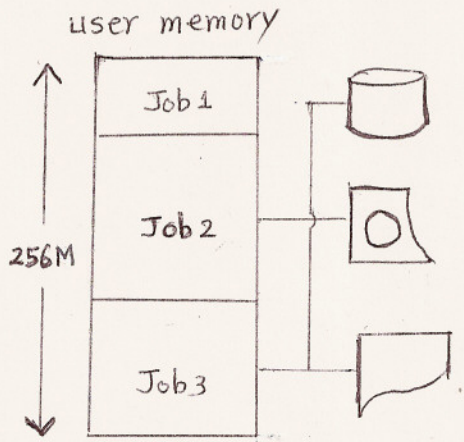


Table 8.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	22%	43%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput rate	6 jobs/h	12 jobs/h
Mean response time	18 min	10 min

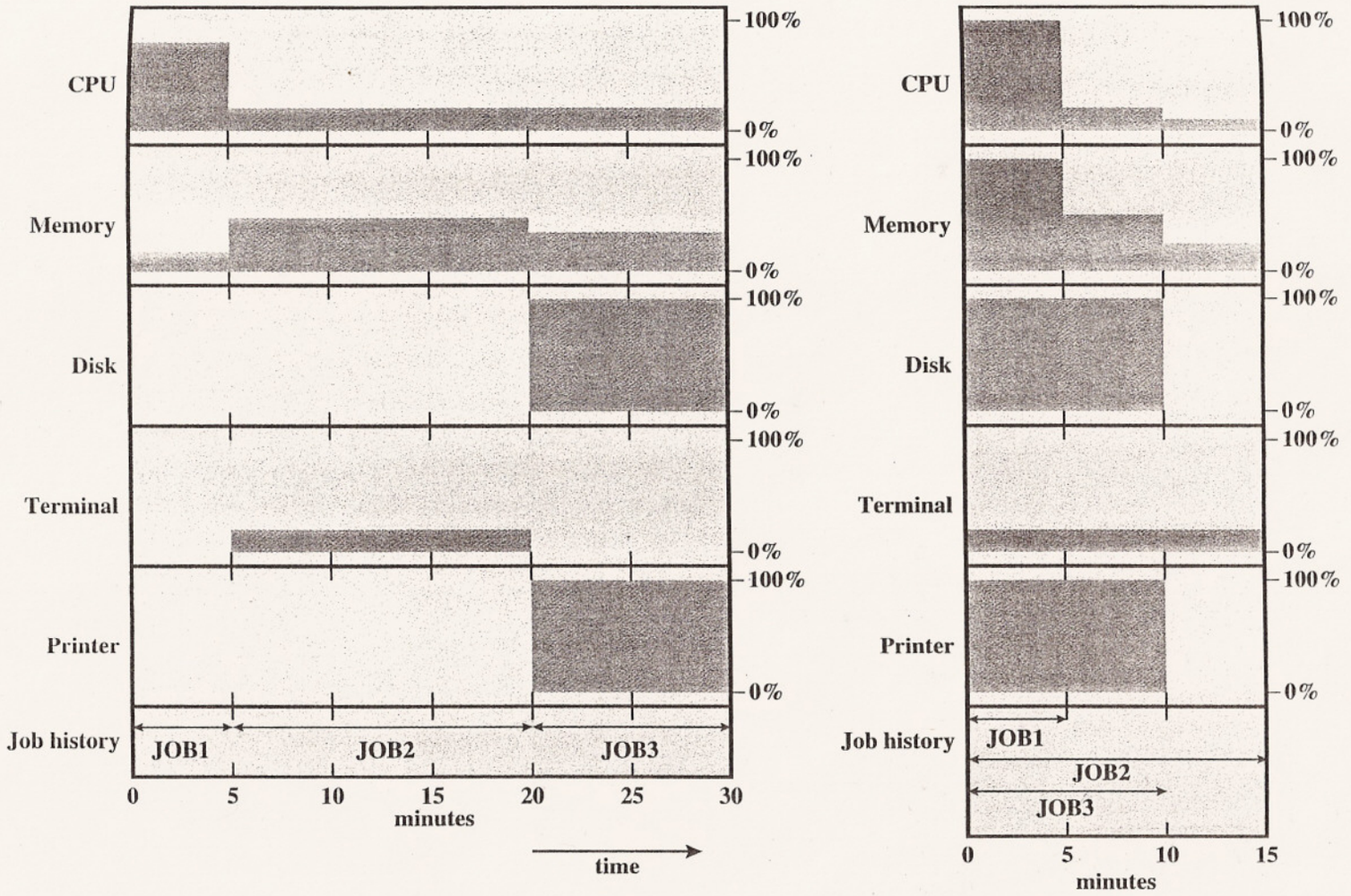


Figure 8.6 Utilization Histograms

8.2 Process Scheduling

Note. Process is subject to schedule.
- compile, link, run
- interrupt is not a process

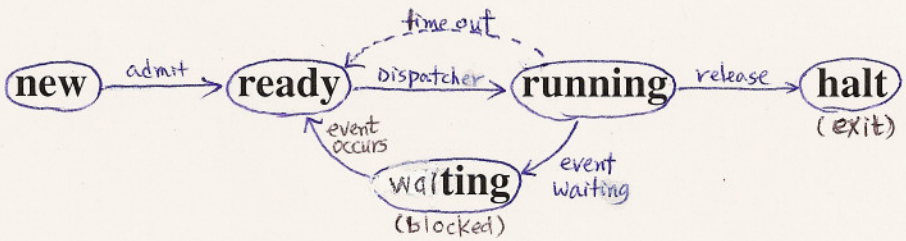
Long-term Scheduling (Job Scheduling)

- degree of multiprogramming
- job queue (disk)
- infrequent decision

Short-term Scheduling (Process Scheduling)

- dispatcher
- frequent, fine-grained decision

Process State Diagram



Process Control Block

Process id
State
Priority
Program counter
Memory pointers
Context data
I/O staus
Accounting
:

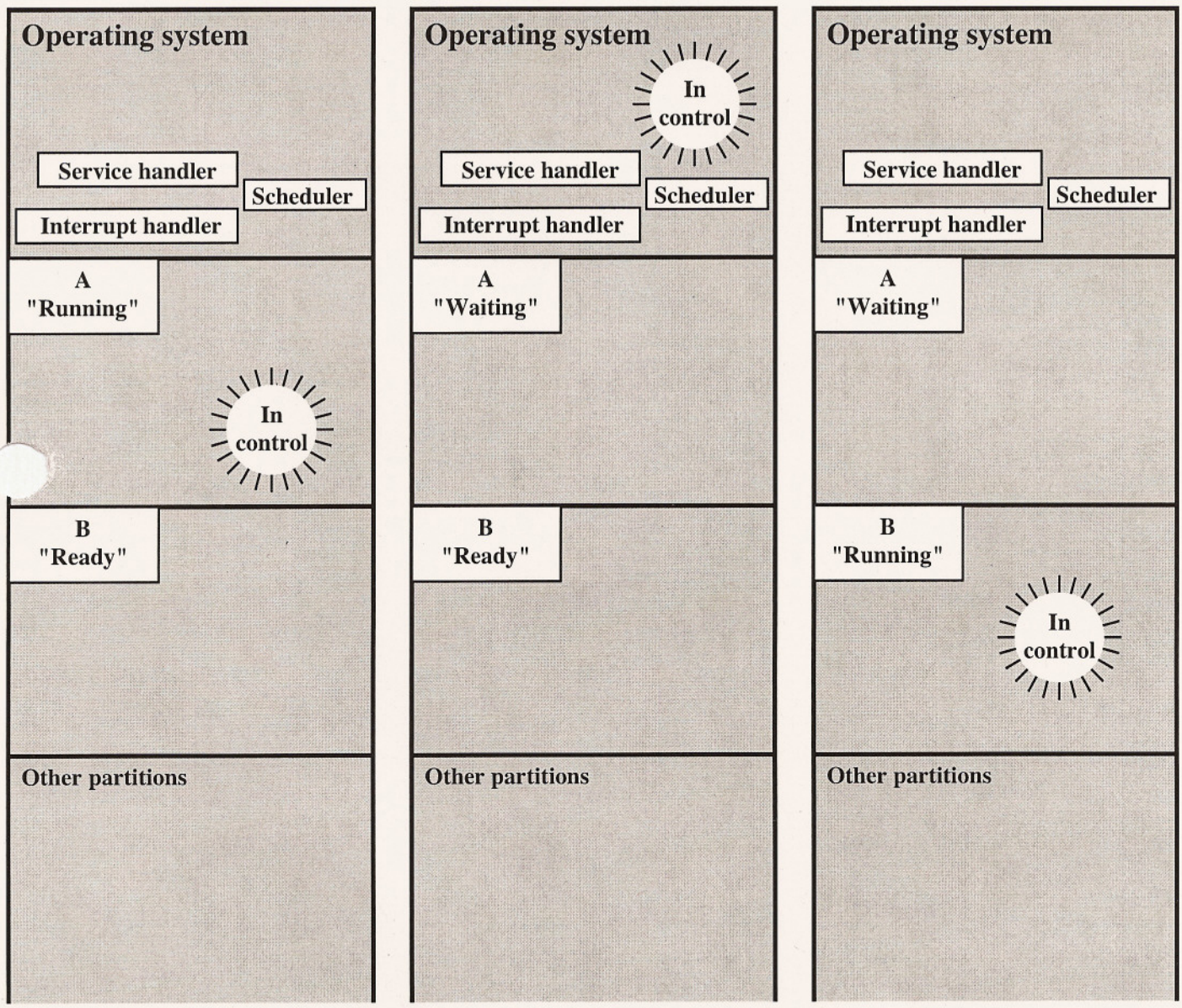


Figure 8.9 Scheduling Example

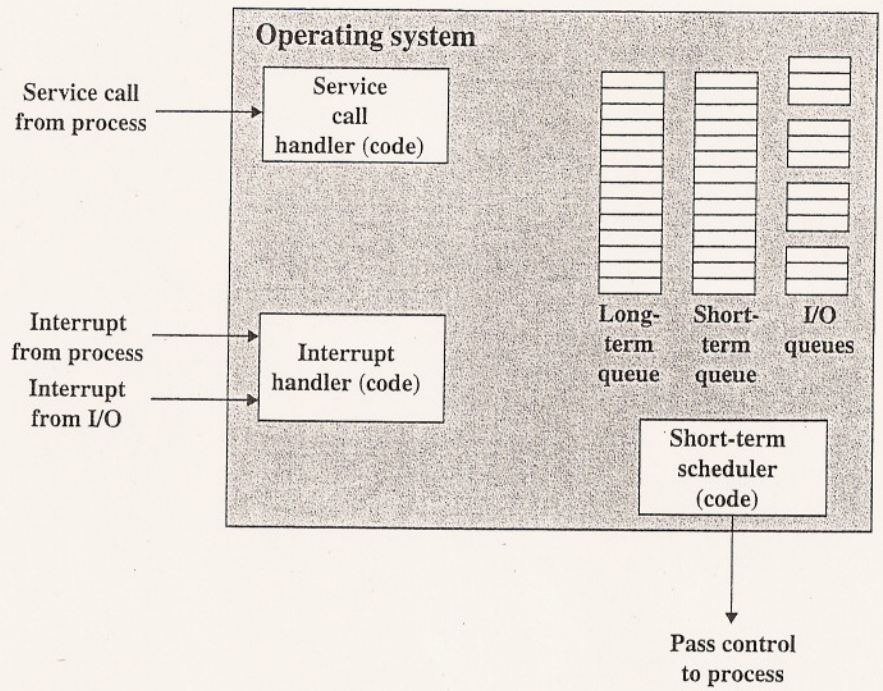


Figure 8.10 Key Elements of an Operating System for Multiprogramming

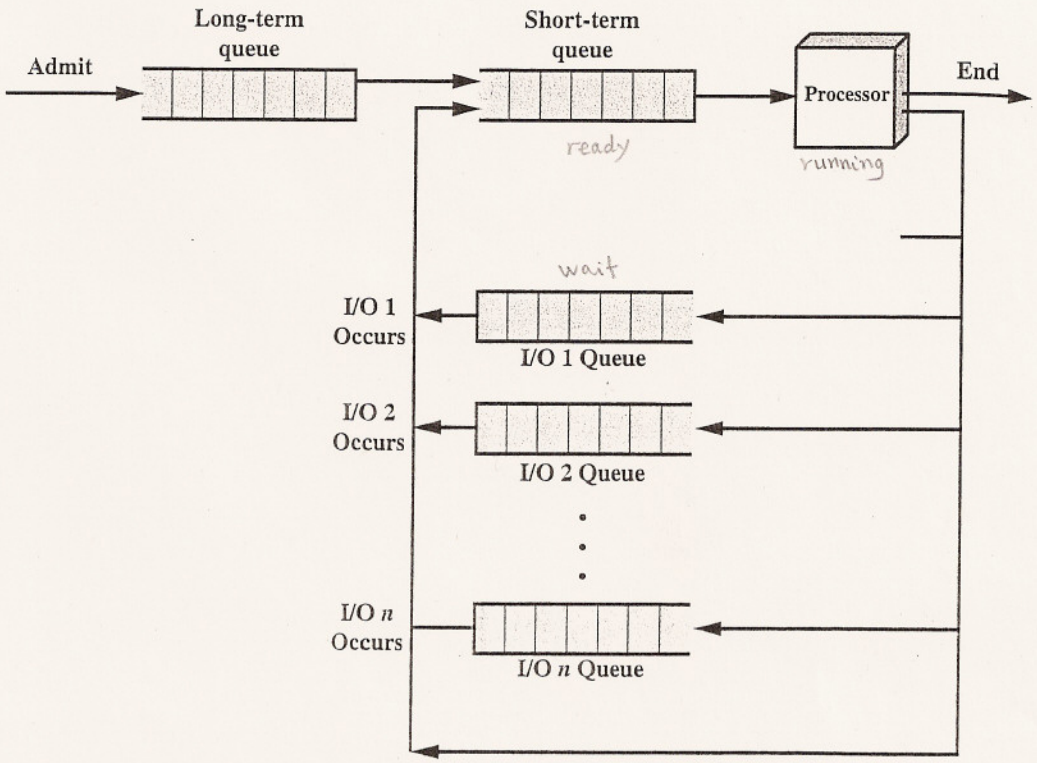


Figure 8.11 Queuing Diagram Representation of Processor Scheduling

Scheduling Algorithm

Performance criteria

1. CPU utilization
   40-90%
2. Throughput - short job preferred
3. Turnaround time
4. Waiting time
5. Response time – interactive case

Scheduling algorithm

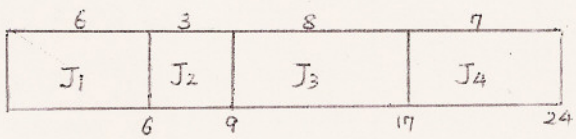
1. FCFS
2. Shortest-job-first
3. Priority
4. Round-robin : time quantum

Example:

Job	Burst time	Priority
1	6	10
2	3	20
3	8	30
4	7	50

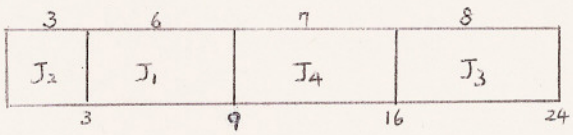
Turnaround time

FCFS



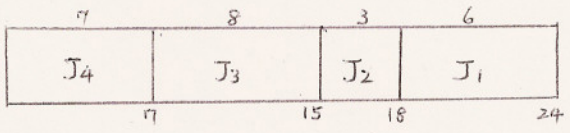
$\frac{6+9+17+24}{4} = 14$

SJF



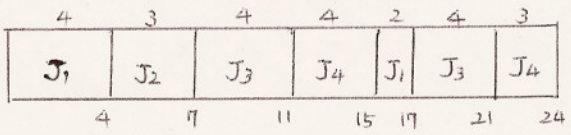
$\frac{3+9+16+24}{4} = 13$

Priority



$\frac{7+15+18+24}{4} = 16$

RR(4)



$\frac{17+7+21+24}{4} = 17\frac{1}{4}$

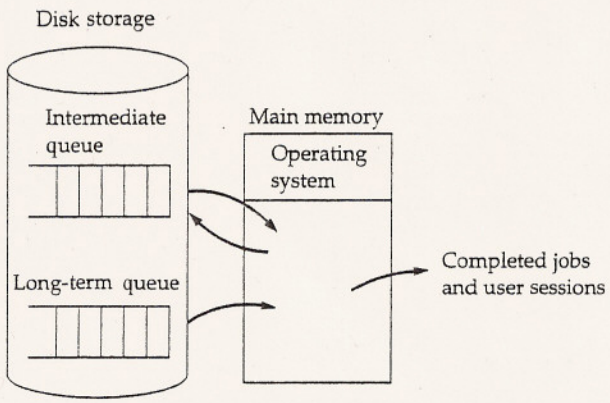
Memory Management

. cpu speed >> I/O speed → cpu idle

Methods to increase performance

(1) Swapping

If all processes are in wait state, then O.S. brings another process from the intermediate queue, or long-term queue
Kicked-out process is placed in intermediate queue

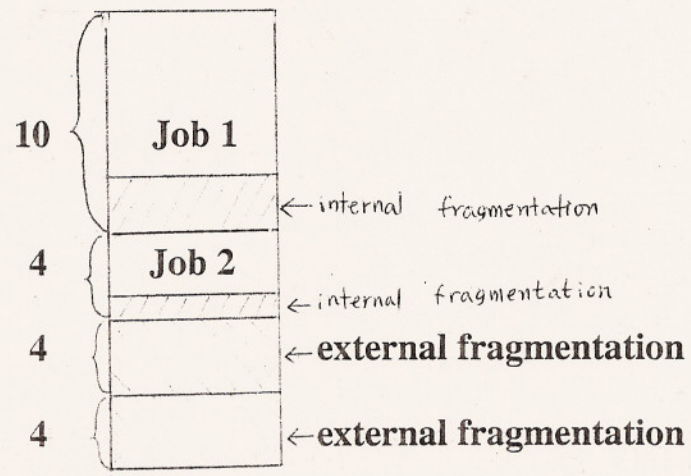


(2) Virtual Memory

Partitioning : fixed-size vs. variable-size

Fixed-Size Partitioning : equal-size vs. unequal-size

Ex. Job 1 – 7M
Job 2 – 3M
Job 3 – 6M
Job 4 – 6M



Variable-size Partitioning (Dynamic partitioning)

- Buddy system
- Boundary-tag method

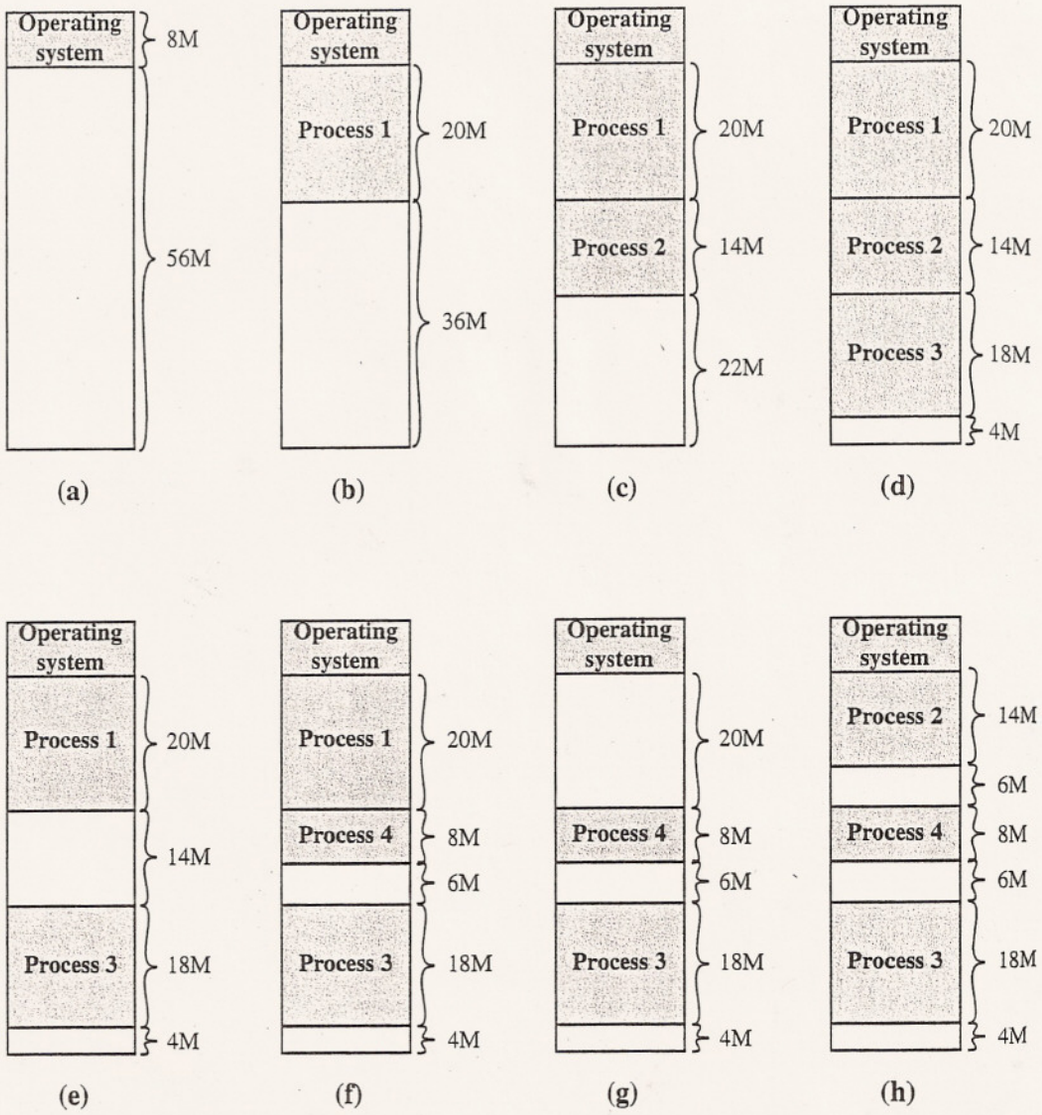


Figure 8.14 The Effect of Dynamic Partitioning

fragments

compaction – time-consuming, addressing problem

Variable-Partition – Fragmentation

Allocation Strategies

(1) Best-fit algorithm

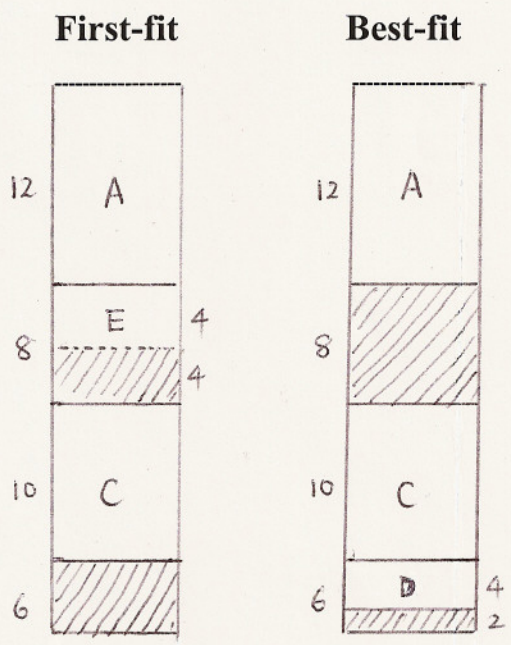
- **use the smallest hole possible to leave the largest holes available for future scheduling**

(2) First-fit algorithm

- **allocate memory from the first hole that can satisfy the request**

Example. Memory size = 36

Job	Size	Action
A	12	start
B	8	start
C	10	start
B	8	stop
D	4	start

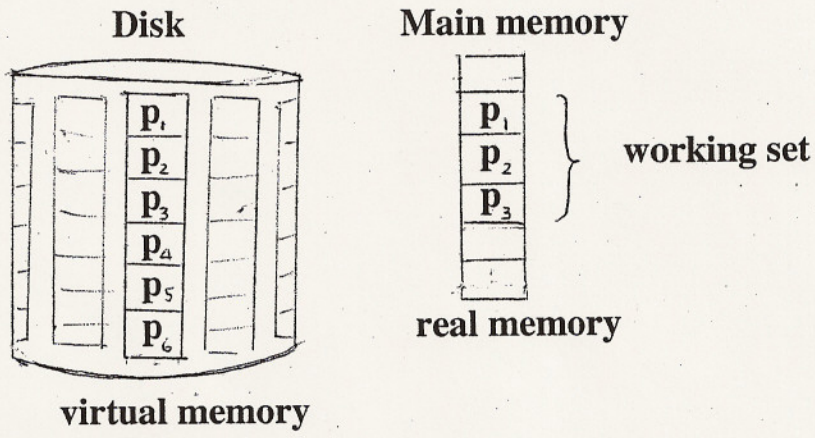


Note. which method performs better?

Virtual Memory

- paging (segmentation) 1962 Atlas

Demand Paging



- page fault
- thrashing

Page table maps virtual (logical) address to real (physical) address.

Ex

Page Table

Page	Frame #	Valid	Dirty
0	2	1	0
1	-	0	-
2	5	1	1
⋮	⋮	⋮	⋮

→ should be rewritten to disk when replaced.

(1) Direct Mapping

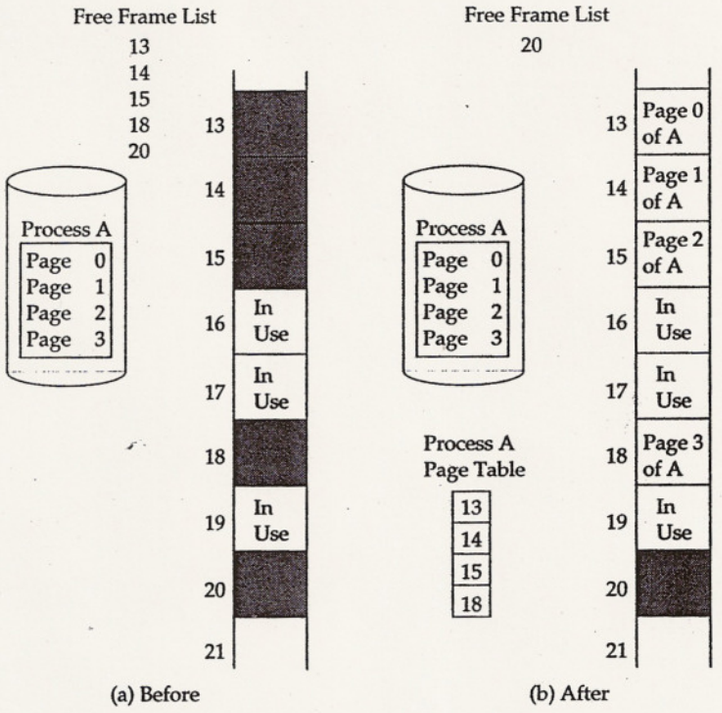


FIGURE 8.15. Allocation of free frames

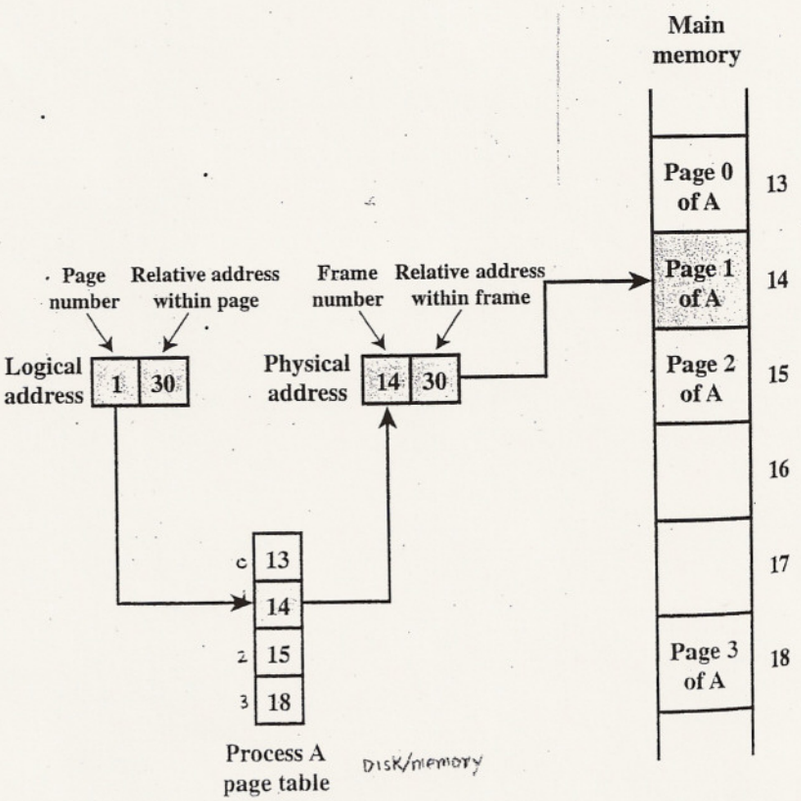


Figure 8.16 Logical and Physical Addresses

2 memory access

(2) Associative mapping
-TLB

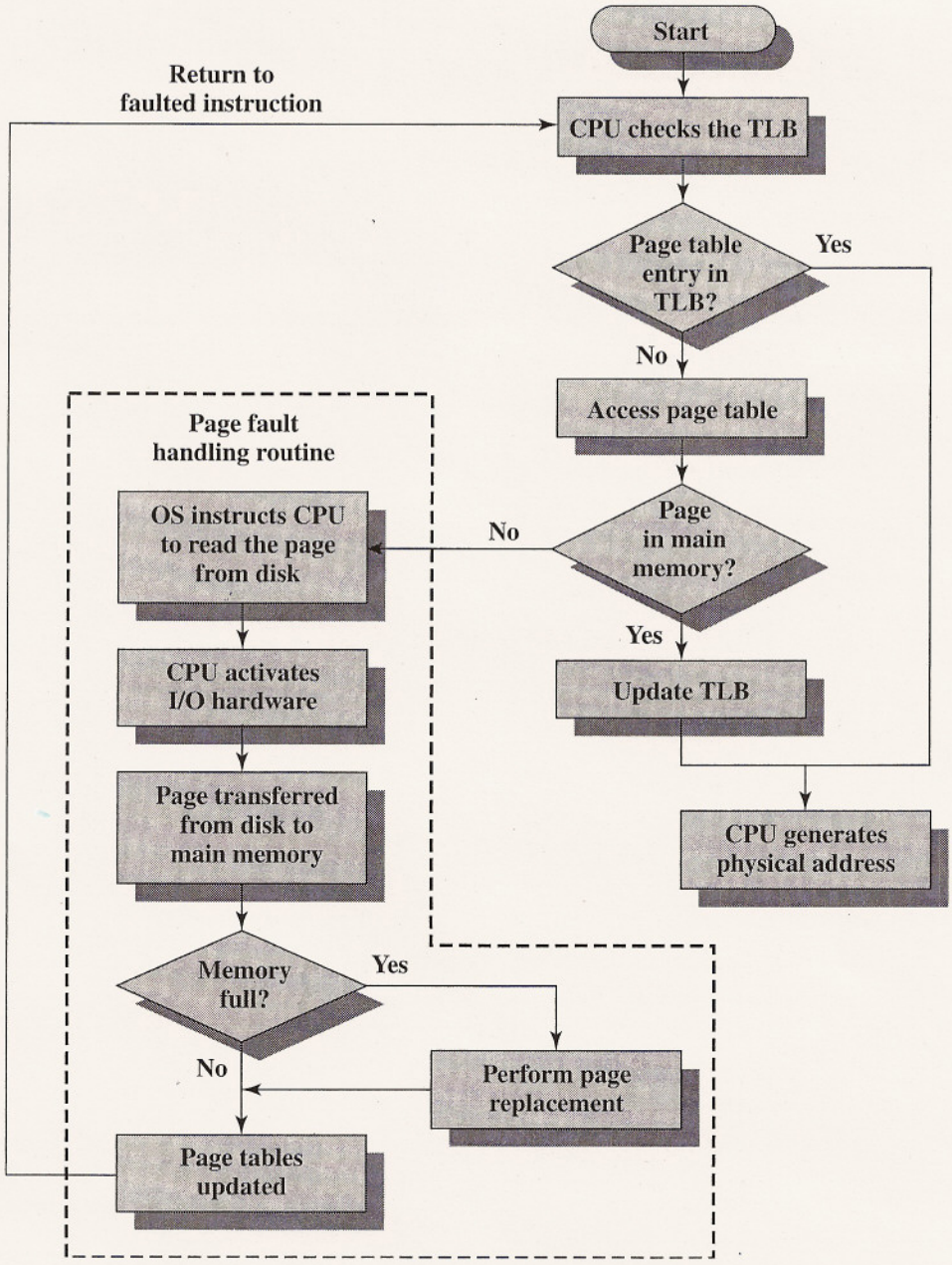


Figure 8.18 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

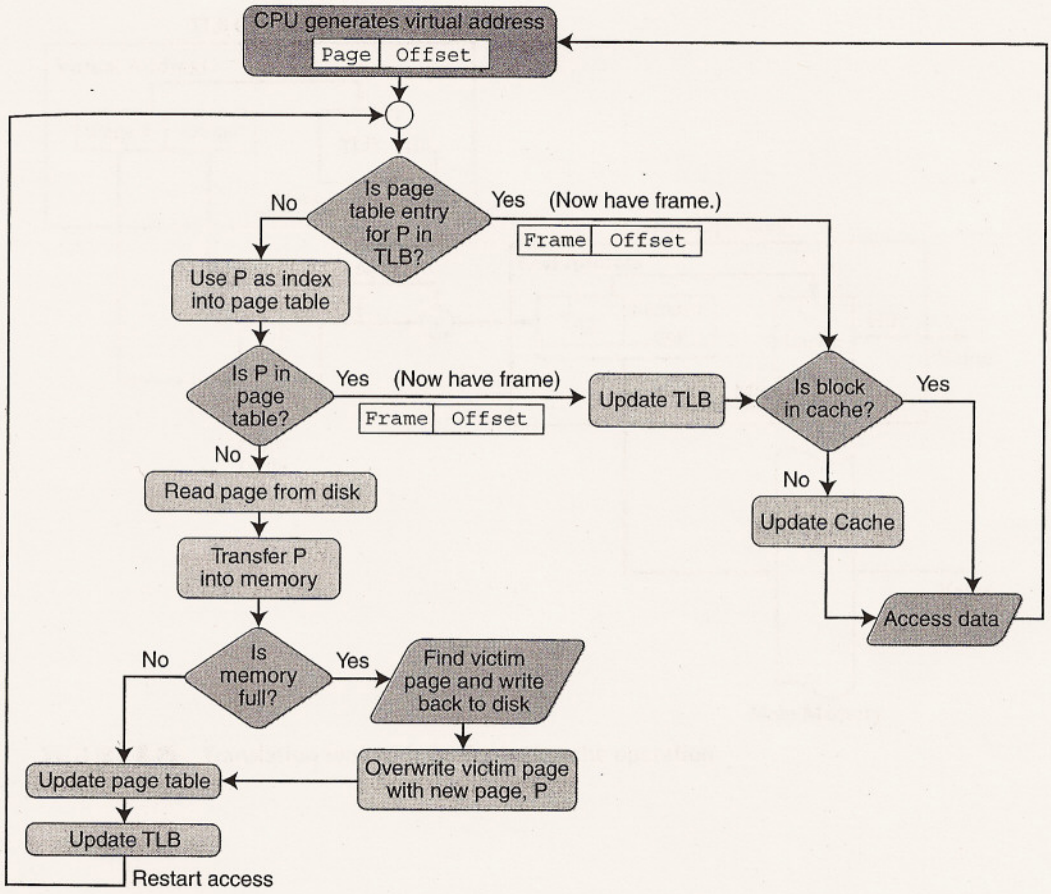


FIGURE 6.20 Putting It All Together: The TLB, Page Table, Cache, and Main Memory

Segmentation

visible
logical

Segmented Page

Page Replacement Policy

- **FIFO**
- **LRU** – age register
- **OPT** – not practical

Example 5 pages, 3 frames

page address stream	2	3	2	1	5	2	4	5	3	2	5	2
FIFO	2*	2*	2*	2*	5	5	5*	5*	3	3	3	3*
		3	3	3	3*	2	2	2	2*	2*	5	5
				1	1	1*	4	4	4	4	4*	2
			h					h		h		
LRU	2*	2*	2	2	2*	2	2	2*	3	3	3*	3*
		3	3*	3*	5	5	5*	5	5	5*	5	5
				1	1	1*	4	4	4*	2	2	2
			h			h		h			h	h

Chapter 9 Computer Arithmetic

Integer Representation

(1) Sign-Magnitude Method

+18 = 0 0 0 1 0 0 1 0
-18 = 1 0 0 1 0 0 1 0

Drawbacks –

(2) Two's Complement Method

Ex. (-3) in 8-bit number system

+3	0 0 0 0 0 0 1 1
complement	1 1 1 1 1 1 0 0
add 1	1 1 1 1 1 1 0 1 (= -3)

Ex. 5 – 3 = 5 + (-3)

5	0 0 0 0 0 1 0 1	
-3	1 1 1 1 1 1 0 1	+

1	0 0 0 0 0 0 1 0	(= 2)
ignore		

Note. No subtractor is necessary.

Note: If two numbers are added and they are both positive or both negative, then overflow occurs iff the result has the opposite sign.

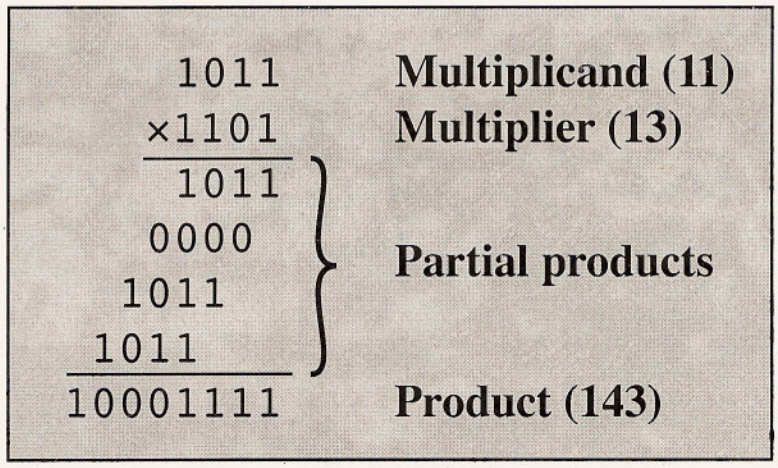
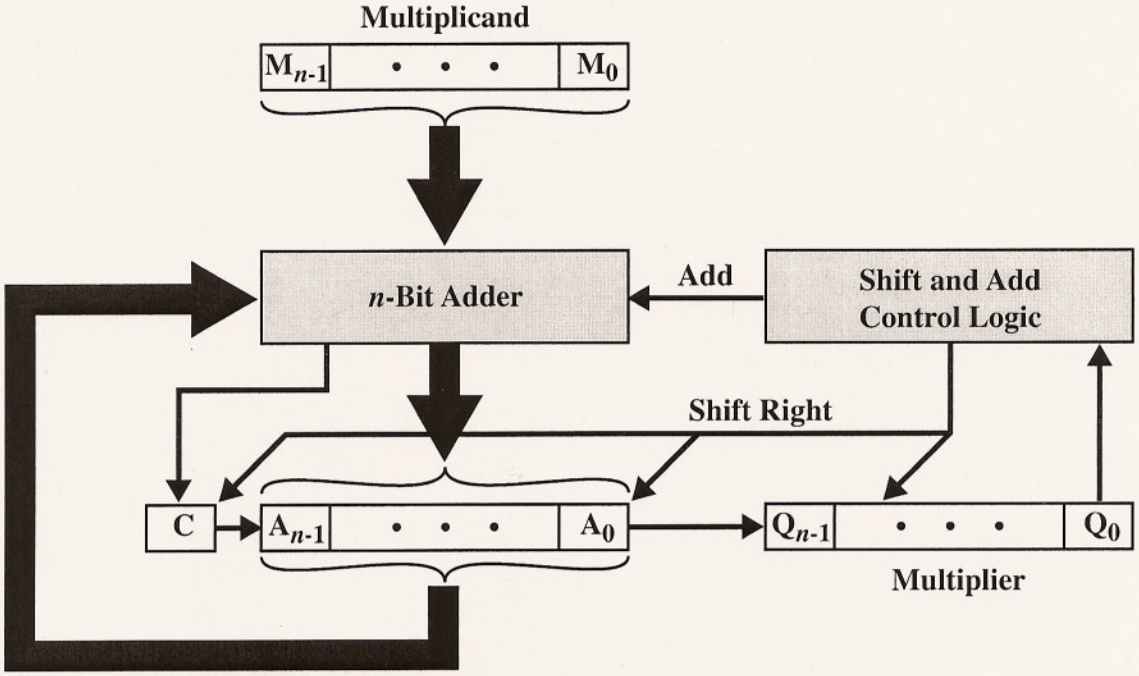


Figure 9.7 Multiplication of Unsigned Binary Integers



(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 9.8 Hardware Implementation of Unsigned Binary Multiplication

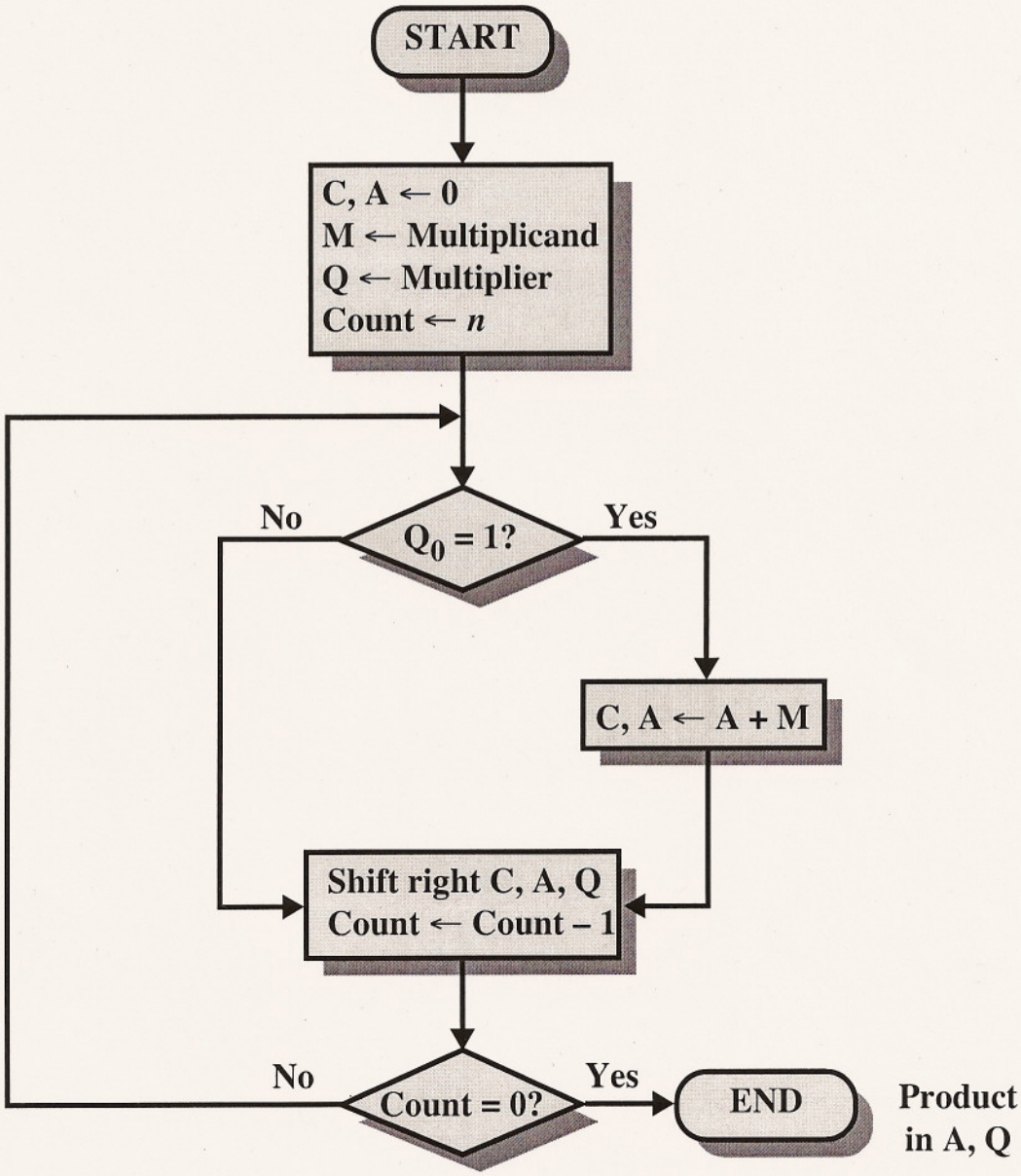


Figure 9.9 Flowchart for Unsigned Binary Multiplication

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A ← A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A ← A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Figure 9.13 Example of Booth's Algorithm (7 × 3)

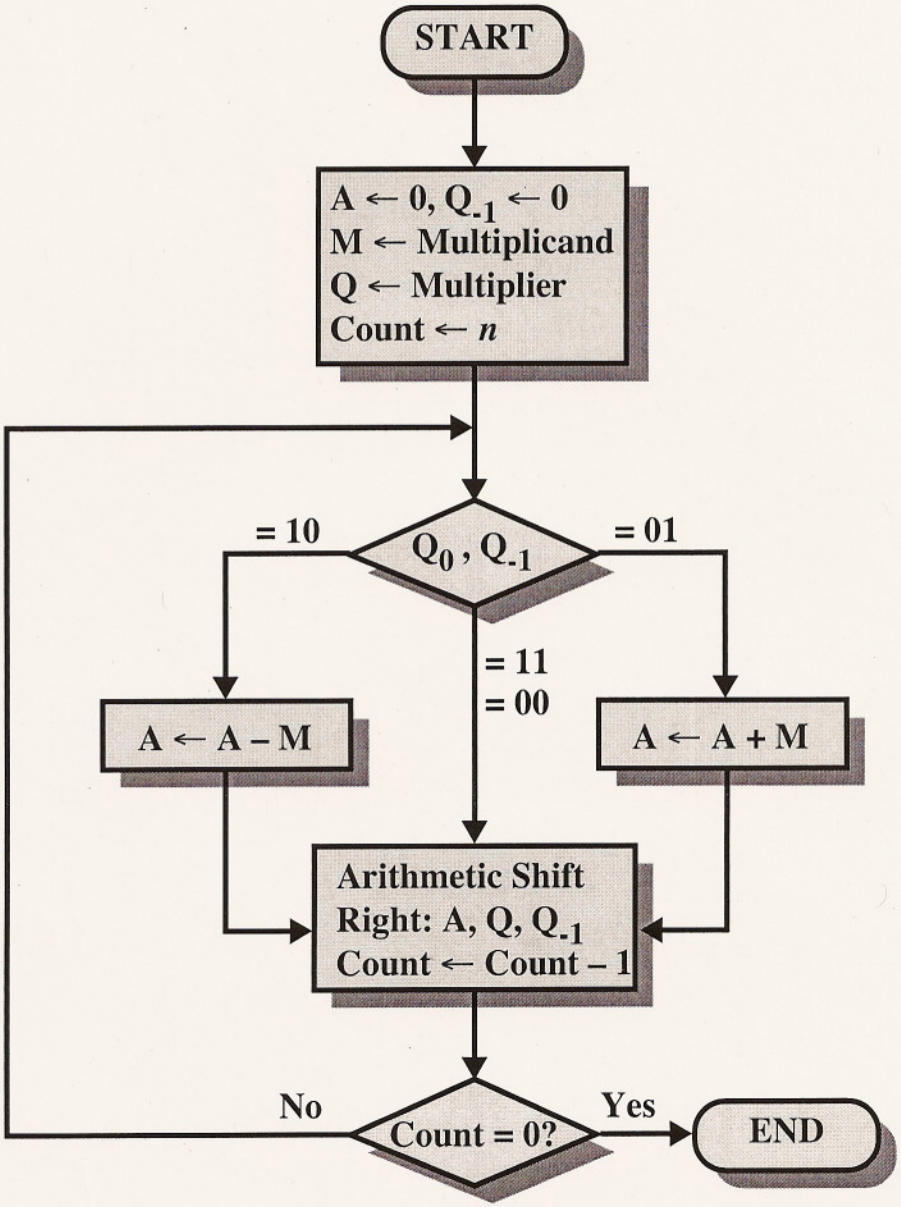


Figure 9.12 Booth's Algorithm for Twos Complement Multiplication

Booth's algorithm - continued

Ex. $0\ 0\ \boxed{0\ (1}\ 1\ 1\ \boxed{1)\ 0}$

(end of run; beginning of run)

Note. $2^4 + 2^3 + 2^2 + 2^1 \equiv 2^5 - 2^1$

Ex. (7) x (-3)

A	Q	Q_{-1}		
0 0 0 0	1 1 0 1	0	Initial	
1 0 0 1	1 1 0 1	0	A ← A – M	First cycle
1 1 0 0	1 1 1 0	1	shift	
0 0 1 1	1 1 1 0	1	A ← A + M	Second cycle
0 0 0 1	1 1 1 1	0	shift	
1 0 1 0	1 1 1 1	0	A ← A – M	Third cycle
1 1 0 1	0 1 1 1	1	shift	
1 1 1 0	1 0 1 1	1	shift	Fourth cycle

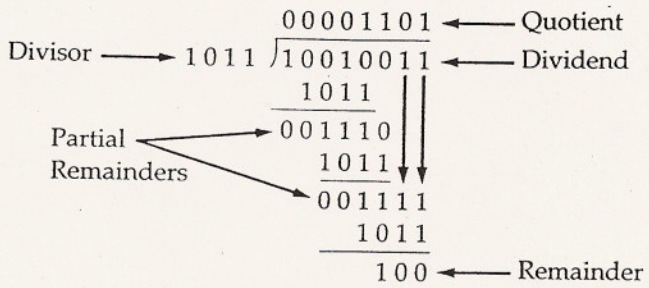


FIGURE 9.15. Division of unsigned binary integers

A	Q	M = 0011
0000	0111	Initial Value
0000	1110	Shift
1101		Subtract
0000	1110	Restore
0001	1100	Shift
1110		Subtract
0001	1100	Restore
0011	1000	Shift
0000		Subtract
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Subtract
0001	0010	Restore

(a) (7) ÷ (3)

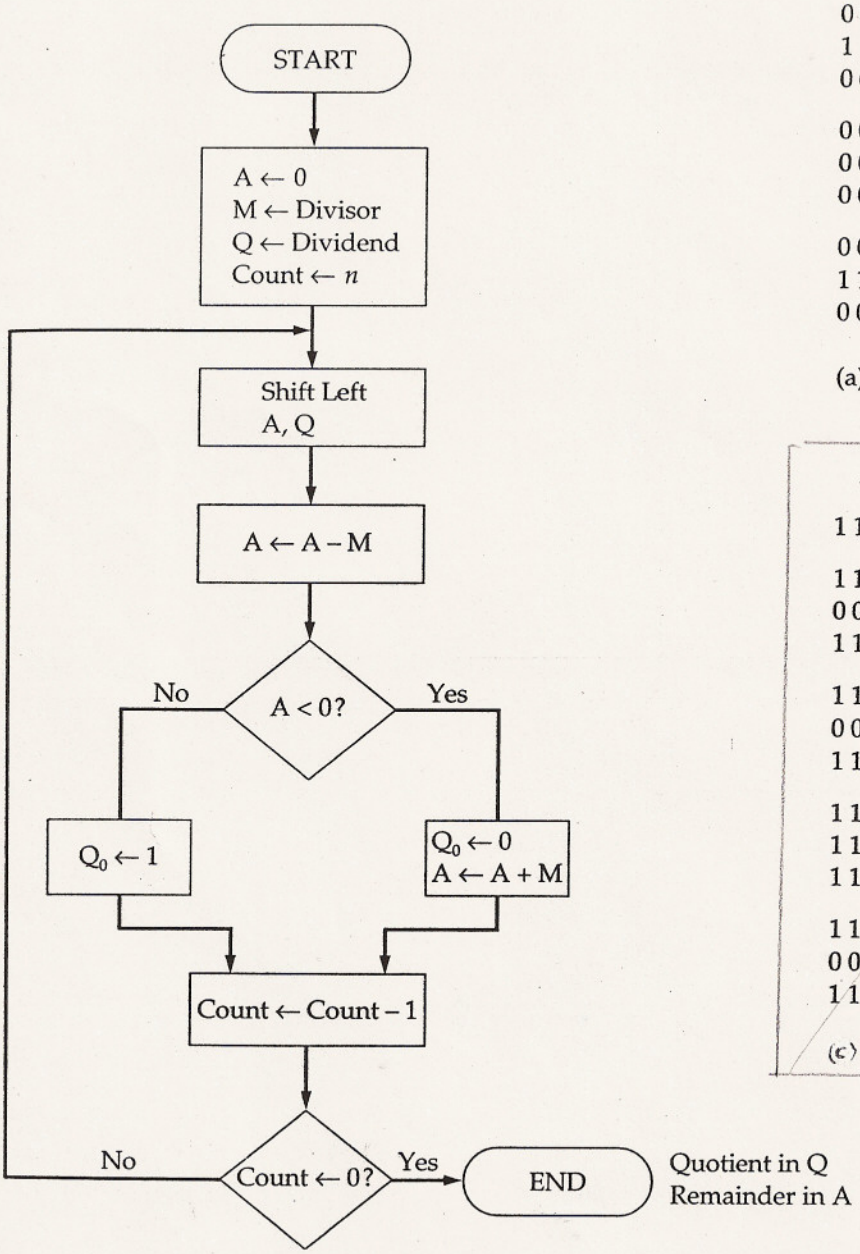


FIGURE 9.16. Flowchart for unsigned binary division

A	Q	M = 0011
1111	1001	Initial Value
1111	0010	Shift
0010		Add
1111	0010	Restore
1110	0100	Shift
0001		Add
1110	0100	Restore
1100	1000	Shift
1111		Add
1111	1001	Set $Q_0 = 1$
1111	0010	Shift
0010		Add
1111	0010	Restore

(c) (-7) ÷ (3)

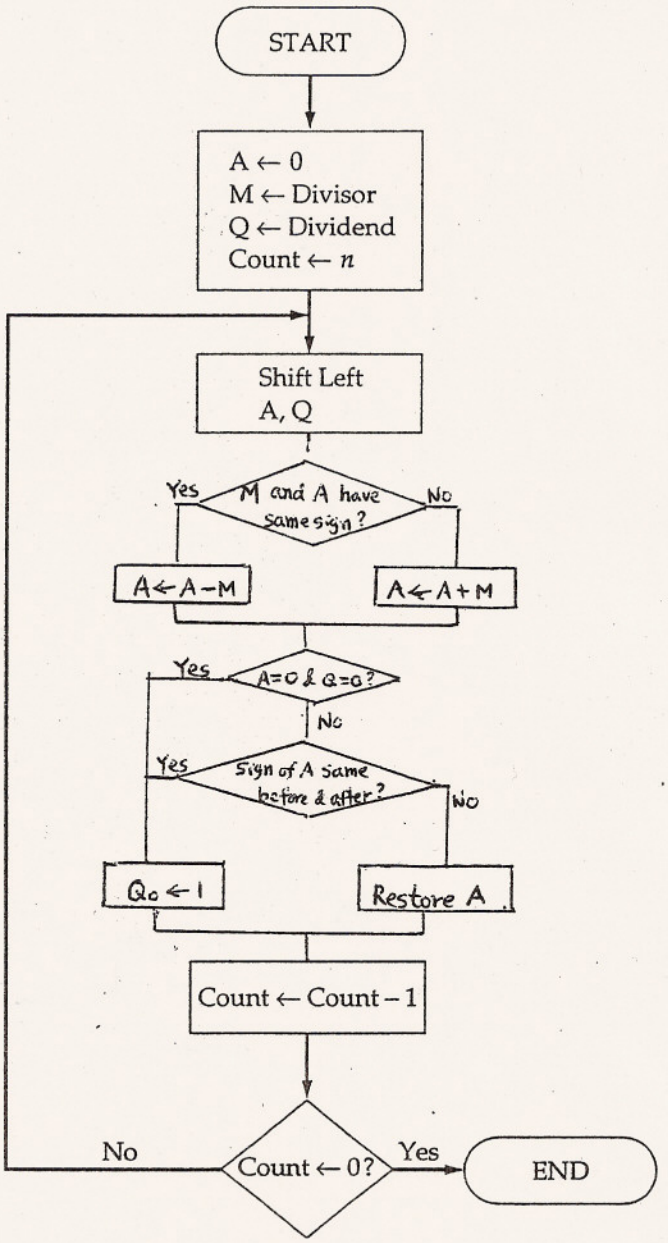


FIGURE **Flowchart for signed binary division**

A	Q	M = 0011
0000	0111	Initial Value
0000	1110	Shift
1101		Subtract
0000	1110	Restore
0001	1100	Shift
1110		Subtract
0001	1100	Restore
0011	1000	Shift
0000		Subtract
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Subtract
0001	0010	Restore

(a) (7) ÷ (3)

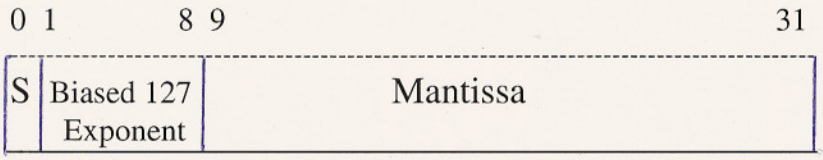
A	Q	M = 0011
1111	1001	Initial Value
1111	0010	Shift
0010		Add
1111	0010	Restore
1110	0100	Shift
0001		Add
1110	0100	Restore
1100	1000	Shift
1111		Add
1111	1001	Set $Q_0 = 1$
1111	0010	Shift
0010		Add
1111	0010	Restore

(c) (−7) ÷ (3)

(*) Dividend와 Divisor의 signal 다르면 2's complement of Q is the correct one.

Floating-Point Numbers

$$\pm S \times B^{\pm E}$$



-127 ~ +128

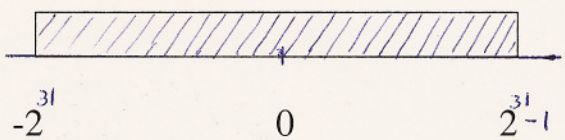
Ex. $1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010\ldots$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010\ldots$

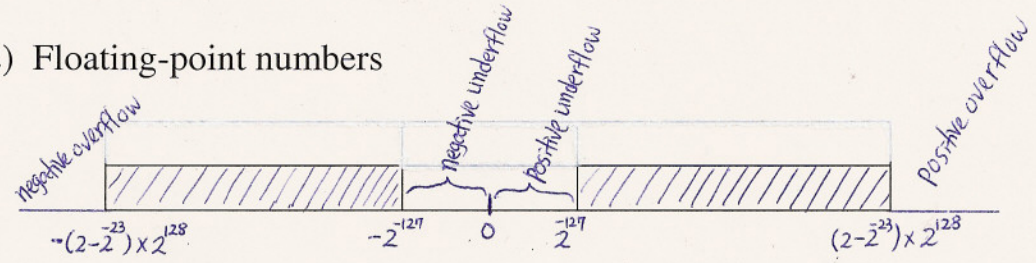
Note. Normalized: $+ 1.bbb..b * 2^{\pm E}$
Base: Either 2 or 16 (IBM S/370)

Expressible numbers in 32-bit word

(1) 2's complement integers

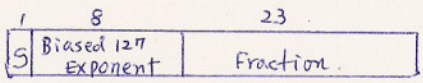


(2) Floating-point numbers



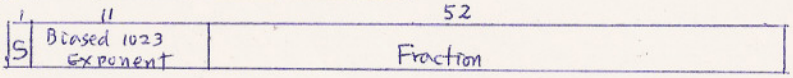
IEEE Standard

Single form



$10^{-38} - 10^{38}$

Double form



$10^{-308} - 10^{308}$

Floating-point Arithmetic

Addition and Subtraction

1. **Check for zero**
2. **Align the significands** // smaller number is shifted right
3. **Add or subtract the significands**
4. **Normalize the result** // shift left until most-significant digit is nonzero.

Example. (decimal, 3 significands)

$123 + 4.56$

Normalized: $1.23 \times 10^2 + 4.56 \times 10^0$

Align

$$\begin{array}{rl} & 1.23 \times 10^2 \\ +) & 0.0456 \times 10^2 \\ \hline & 1.2756 \times 10^2 \end{array}$$

truncate: 1.27×10^2

round: 1.28×10^2

Rounding Policy

- Round to nearest number (default)

Example.

23

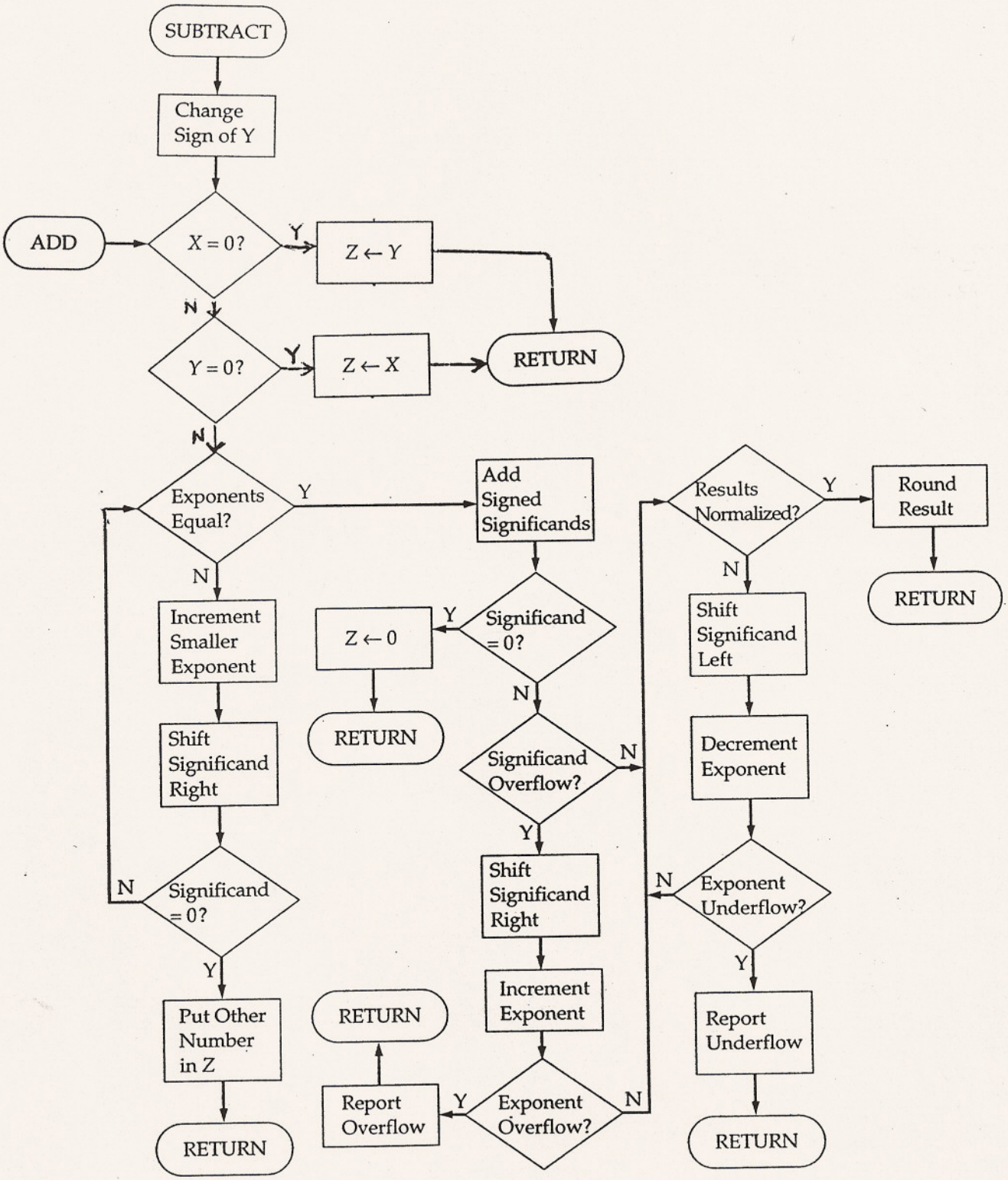
(1) ....xxx | 1 0 0 0 1 0 ... roundup
(2) ....xxx | 0 1 1 0 1 0 ... truncate

Case of tie: .. xxx | 1 0 0 0 0 ...

Solution.

..xx1 | 1 0 0 0 0 ... roundup
..xx0 | 1 0 0 0 0 ... truncate

Floating-point addition and subtraction ($z \leftarrow x \pm y$)



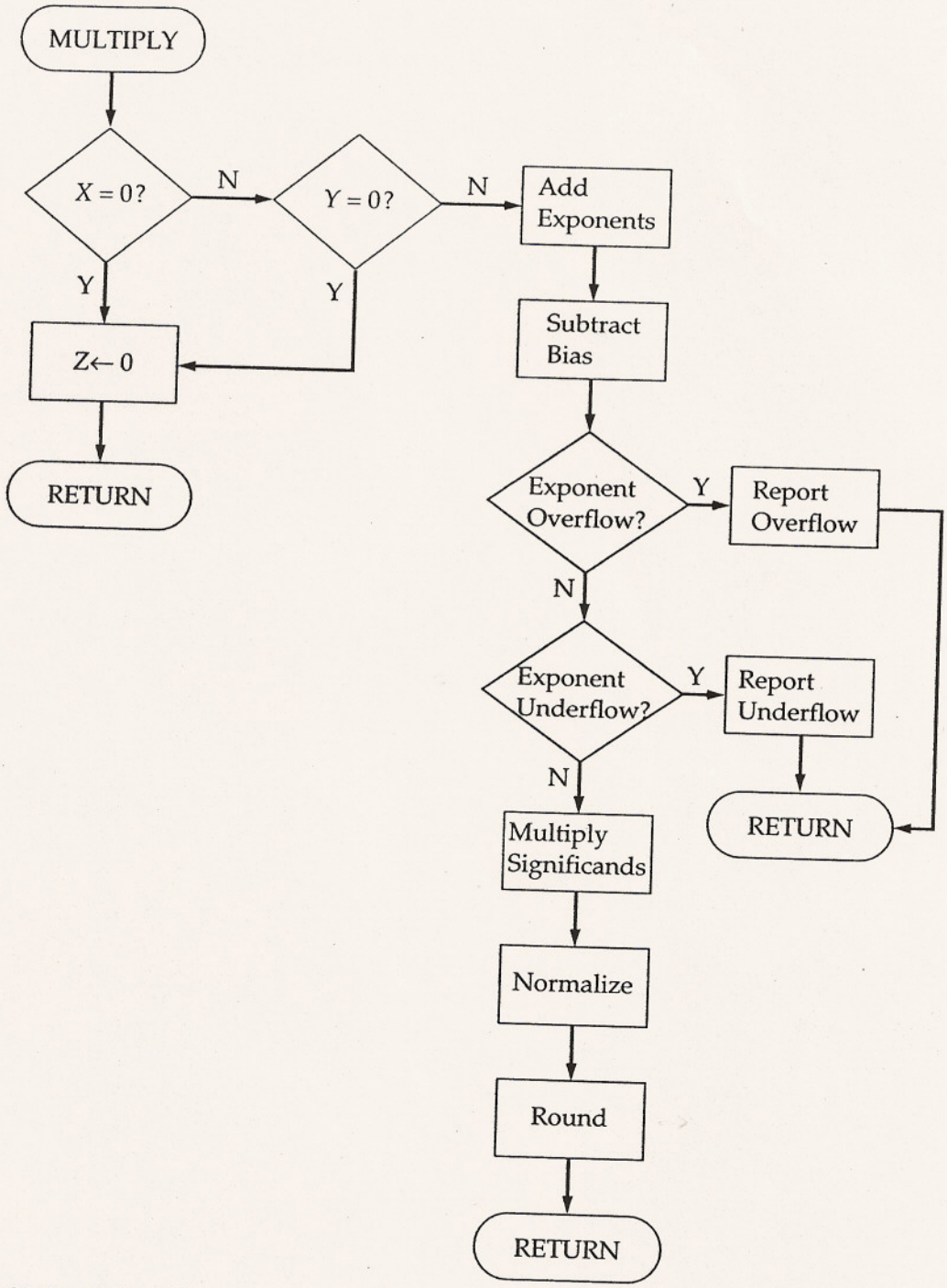
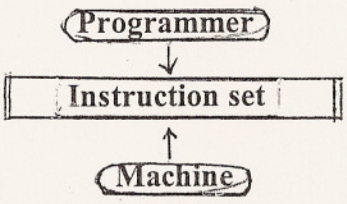


FIGURE 9.23 Floating-point multiplication ($z \leftarrow x \times y$)

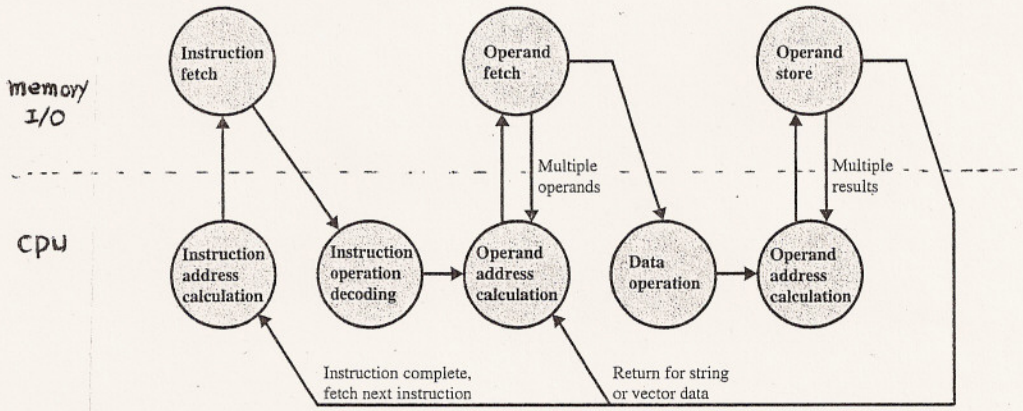
10. Instruction Sets: Characteristics and Functions



register, memory structure

functional requirement for CPU

Instruction cycle



Instruction format

op code	r1	r2
op code	r	m

Op code (symbolic representation: mnemonics)

	Machine 1	Machine 2	Machine 3
Load	LOAD	L	:
Store	STOR	ST	:
Add	ADD	A	:
Subtract	SUB	S	:
Multiply	MUL	M	:
Divide	DIV	D	:

Instruction type

Data processing	-	arithmetic, logical
Data storage	-	load, store
Data movement	-	read, write
Control	-	test, branch

Number of addresses

3-address instruction	op A,B,C	A ← B op C
2-address instruction	op A,B	A ← A op B
1-address instruction	op A	AC ← AC op A
0-address instruction	op	T ← (T-1) op T

Note. Stack computer: Burroughs B5000
Most computer systems use 2 or 3 address instructions.

Instruction Set Design

Issues:

- Operation repertoire
- Data type - integer, float, decimal, character, logical, ...
- Instruction format: length, #addresses, field size, ...
- Registers – no. of registers, kind (general, floating-point, ...)
- Addressing mode – direct / indirect

Y = (A – B) / (C + D * E) → Y = AB-CDE*+/ (postfix)

(a) 3-addr

```
SUB  Y,A,B
MPY  T,D,E
ADD  T,T,C
DIV  Y,Y,T
```

(b) 2-addr

```
MOV  Y,A
SUB  Y,B
MOV  T,D
MPY  T,E
ADD  T,C
DIV  Y,T
```

(c) 1-addr

```
LOAD  D
MPY   E
ADD   C
STOR  Y
LOAD  A
SUB   B
DIV   Y
STOR  Y
```

(d) 0-addr

```
PUSH  A
PUSH  B
SUB
PUSH  C
PUSH  D
PUSH  E
MPY
ADD
DIV
POP   Y
```

Types of Operands

- **Address** — unsigned integer
- **Numbers**
  - . **integer (fixed-point)**
  - . **real (floating-point)**
  - . **decimal**

0	0000
1	0001
:	:
9	1001

Example.

(1) +12345 (IBM S/370)

unpacked decimal:

F1	F2	F3	F4	C5

packed decimal:

12	34	5C

(2) -314

unpacked decimal:

F3	F1	D4

packed decimal:

31	4D

- **characters**
  - . ASCII -- 7 bits
  - . EBCDIC – 8 bits
- **logical data**
  - . bit-level operations
  - . Assembly language (some high-level languages)

Types of operations

TABLE 10.3 Common Instruction Set Operations

Type	*Operation Name*	*Description*
Data Transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Computer sum of two operands (binary)
	Subtract	Compute difference of two operands (binary)
	Multiply	Compute product of two operands (binary)
	Divide	Compute quotient of two operands (binary)
	Absolute	Replace operand by its absolute value (unary)
	Negate	Change sign of operand (unary)
	Increment	Add 1 to operand (unary)
	Decrement	Subtract 1 from operand (unary)
Logical	AND	Perform the specified logical operation bitwise
	OR	
	NOT (Complement)	
	Exclusive-OR	
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class or instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued
Input/Output	Input (read)	Transfer data from specified I/O port or device to destination, e.g., main memory or processor register
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

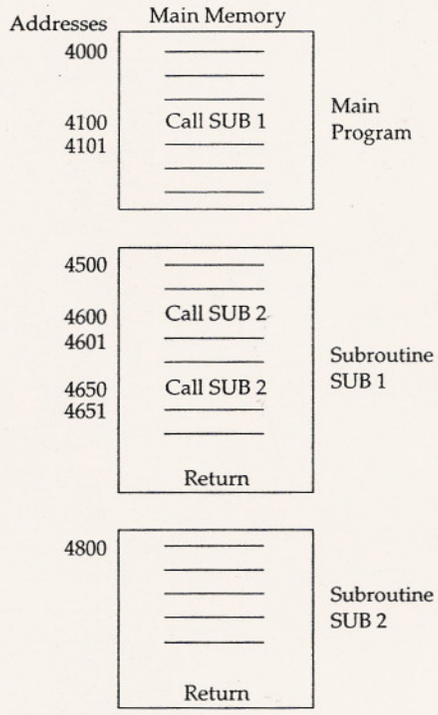
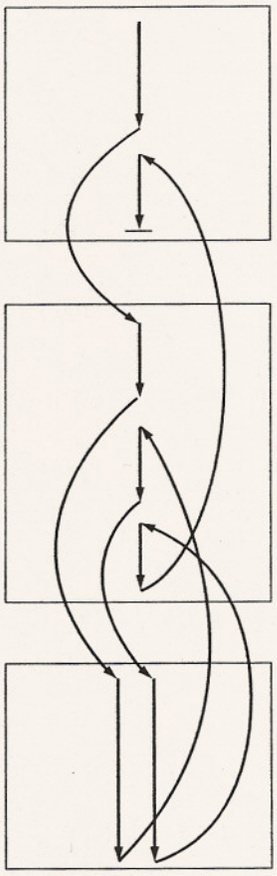


FIGURE 10.7 Nested subroutines



Execution sequence

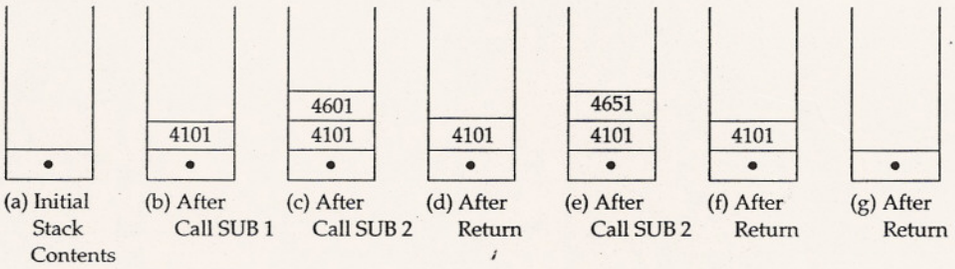


FIGURE 10.8 Use of stack to implement nested subroutines of Figure 10.7

Address	Contents
101	0010 0010 0000 0001
102	0001 0010 0000 0010
103	0001 0010 0000 0011
104	0011 0010 0000 0100
201	0000 0000 0000 0010
202	0000 0000 0000 0011
203	0000 0000 0000 0100
204	0000 0000 0000 0000

(a) Binary Program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal Program

101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic Program

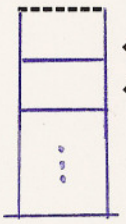
Label	Operation	Opened
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly Program

FIGURE 10.11. Computation of the formula N = I + J + K

10A STACK

Stack (LIFO list) implementation



Usage

(1) sub-procedure call/return
(2) expression evaluation

<stmt> → <assgn stmt> | <cond stmt> | <input stmt> | <while stmt> | ..
<assgn stmt> → <var> := <expr>
<expr> → <expr> + <expr> | <expr> * <expr> | <var>
<var> → a | b | c

Ex. a + b * c (infix)

. precedence: * > +, a + b * c → a + (b * c)

. associativity: left- or right-
a + b + c → (a + b) + c (left-associative)
a ** b ** c → a ** (b ** c) (right-associative)

postfix (reverse Polish):

a b c * +

Advantage:

Expression in postfix can be easily evaluated with a stack.
See Fig 10.16

Note. Many compilers convert infix to postfix, then generate machine code. See Fig 10.17

	Stack	General Registers	Single Register
	Push a Push b Subtract Push c Push d Push e Multiply Add Divide Pop f	Load G[1], a Subtract G[1], b Load G[2], d Multiply G[2], e Add G[2], c Divide G[1], G[2] Store G[1], f	Load d Multiply e Add c Store f Load a Subtract b Divide f Store f
Number of Instructions	10	7	8
Memory Access	10 op + 6 d	7 op + 6 d	8 op + 8 d

Figure 10.15 Comparison of Three Programs to Calculate f = (a − b)/(c + d × e).

Postfix : a b − c d e * + /

Algorithm

Scan postfix expression from left to right.

1. If the element is a variable or constant, push it onto the stack.
2. If the element is an operator, pop the top two items of the stack, perform the operation, and push the result.

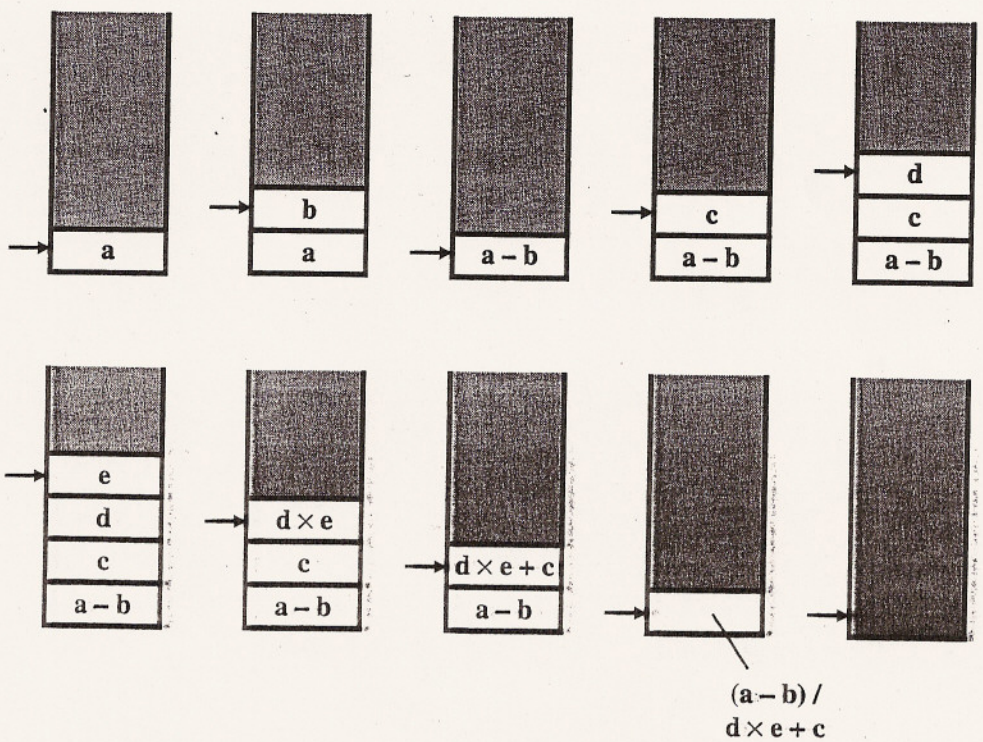


Figure 10.16 Use of Stack to Compute f = (a − b)/(d × e + c).

Input	Output	Stack (top on right)
A + B × C + (D + E) × F	empty	empty
+ B × C + (D + E) × F	A	empty
B × C + (D + E) × F	A	+
× C + (D + E) × F	A B	+
C + (D + E) × F	A B	+ ×
+ (D + E) × F	A B C	+ ×
(D + E) × F	A B C × +	+
D + E) × F	A B C × +	+ (
+ E) × F	A B C × + D	+ (
E) × F	A B C × + D	+ (+
) × F	A B C × + D E	+ (+
× F	A B C × + D E +	+
F	A B C × + D E +	+ ×
empty	A B C × + D E + F	+ ×
empty	A B C × + D E + F × +	empty

Figure 4.17 Conversion of an Expression from Infix to Postfix Notation.

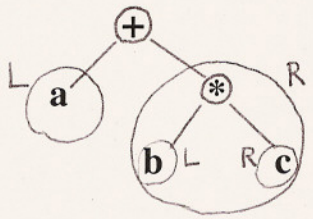
+ (D + E) × F	A B C ×	+
+ (D + E) × F	A B C × +	empty

[DIJK63]. The infix expression is scanned from left to right, and the postfix expression is developed and output during the scan. The steps are as follows:

1. Examine the next element in the input.
2. If it is an operand, output it.
3. If it is an opening parenthesis, push it onto the stack.
4. If it is an operator, then
   - If the top of the stack is an opening parenthesis, then push the operator.
   - If it has higher priority than the top of the stack (multiply and divide have higher priority than add and subtract), then push the operator.
   - Else, pop operation from stack to output, and repeat step 4.
5. If it is a closing parenthesis, pop operators to the output until an opening parenthesis is encountered. Pop and discard the opening parenthesis.
6. If there is more input, go to step 1.
7. If there is no more input, unstack the remaining operands.

Figure 4.17 illustrates the use of this algorithm. This example should give the reader some feel for the power of stack-based algorithms.

Binary Tree Traversal



(rt) L R → 3! different ordering

(rt)	L	R	≡	preorder	(+ a * b c)
L	(rt)	R	≡	inorder	(a + b * c)
L	R	(rt)	≡	postorder	(a b c * +)

Note. Postorder

- Reverse Polish notation
- Jan Łukasiewicz
- Parenthesis free

Note.

- inorder + preorder → unique binary tree
- inorder + postorder → unique binary tree

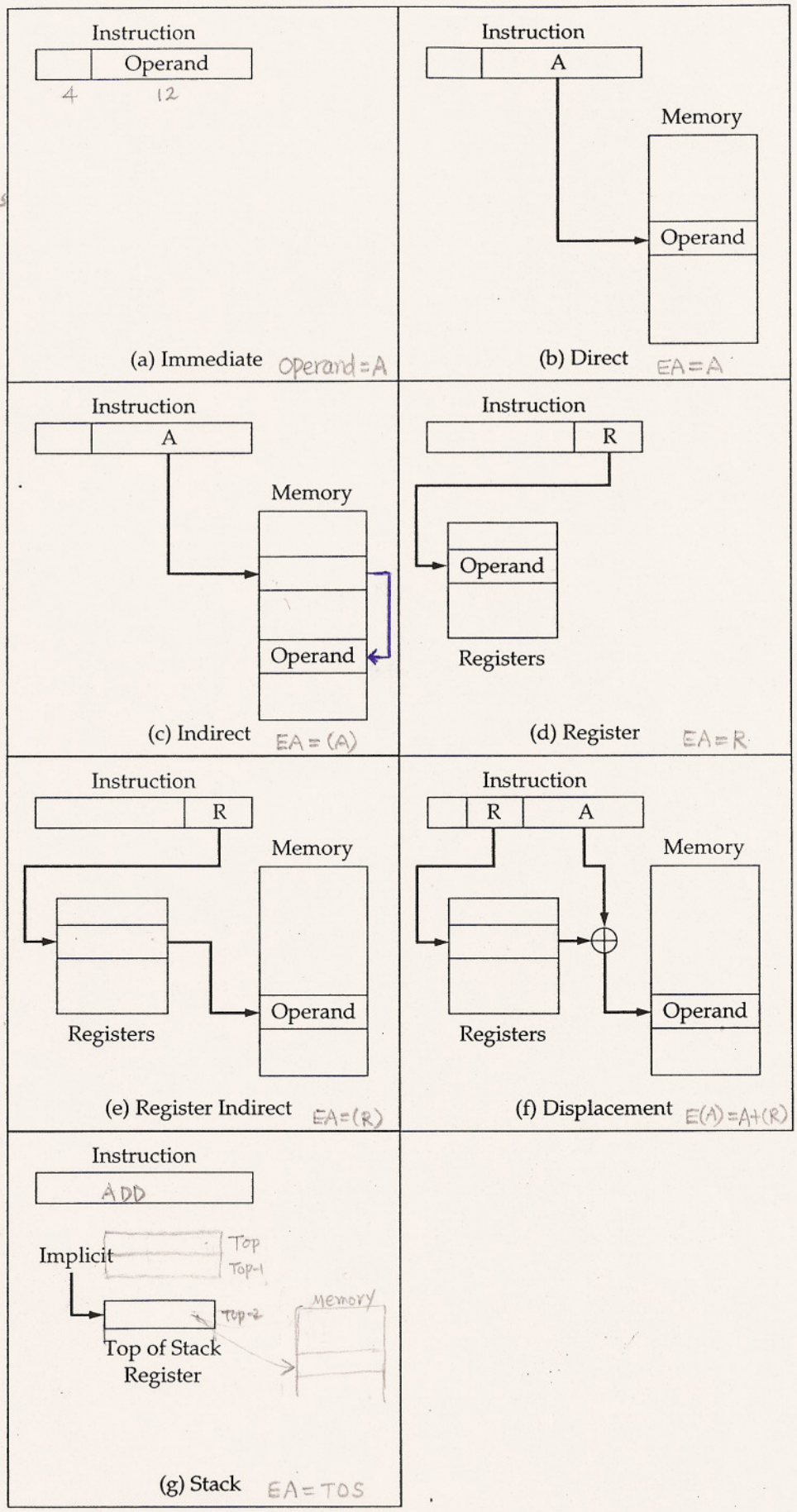
Infix → Postfix

- using stack (Fig 10.17)
- using parenthesis

Little-Endian, Big-Endian

Ex. 12 34 56 78h

184	12	184	78
185	34	185	56
186	56	186	34
187	78	187	12
	big-endian		little-endian
	IBM S/370		Intel
	Motorola		Vax
	Sun SPARC		Alpha



no memory access

Addressible memory space가 작다. (2^{12})

one memory access

Addressible memory space가 크다. (2^{16})

Two memory accesses

Small address field

no memory access

one memory access

0-operand instruction

FIGURE 11.1. Addressing modes

11.3 Instruction Formats

Issues:

(1) Instruction Length

More opcodes
More operands → shorter program — Trade-off
More addressing modes → flexibility
Larger memory address range

Word size
- one character = 8 bits
- multiple of 8

Example. IBM S/360

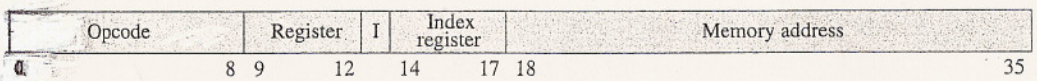
1 word = 32 bits (4 bytes)
1 half word = 2 bytes
1 double word = 8 bytes

Note. IBM 700/7000 series: 36 bits

(2) Allocation of bits

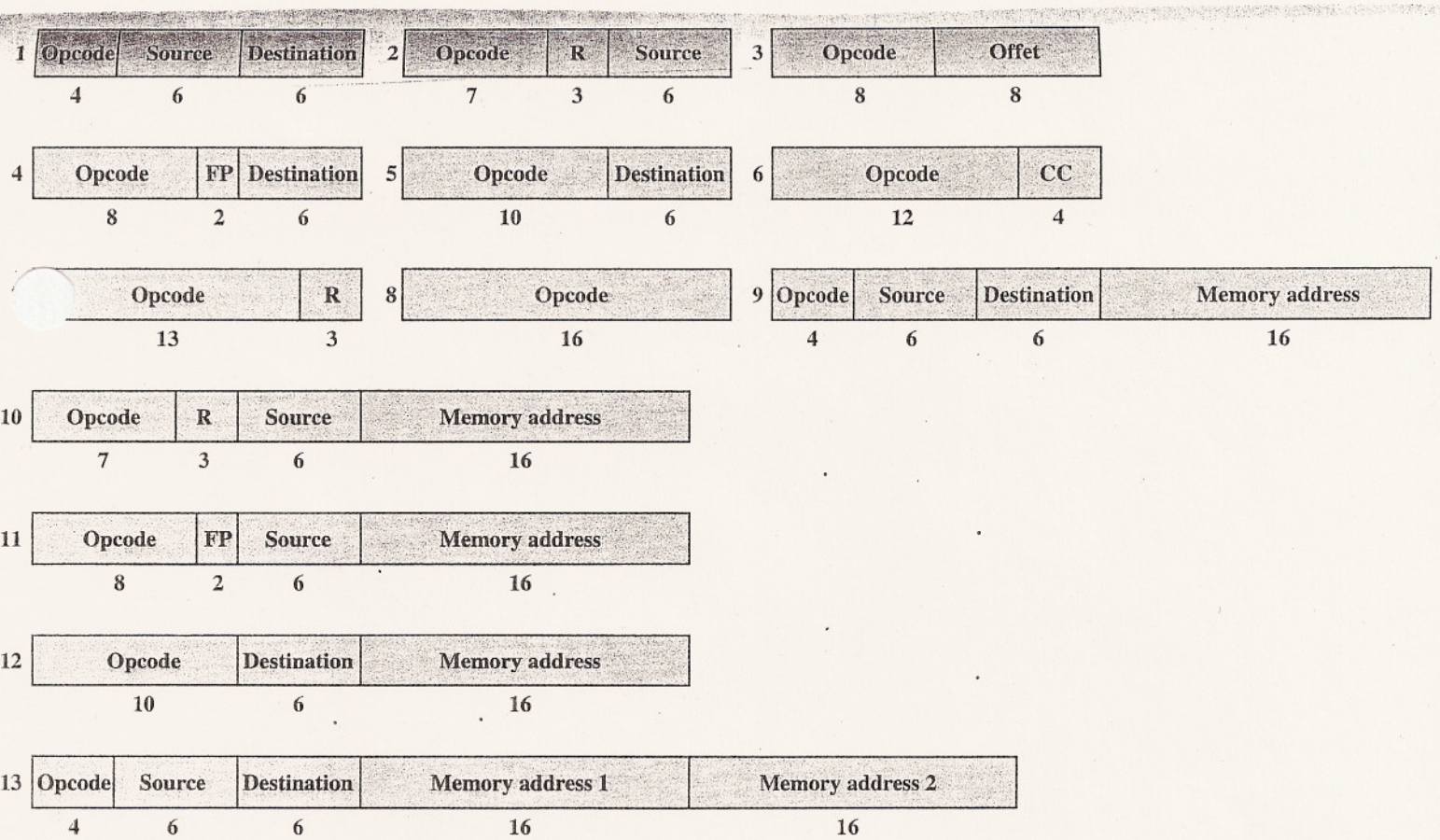
Factors

- number of addressing modes
  implicit or explicit
- number of operands
  majority 2 operands
- register vs. memory
  1. single register (accumulator) – implicit
  2. multiple registers
     4 bits → 16 registers
     5 bits → 32 registers
- number of register sets
  general-purpose
  specialized
  Ex. Index (Si,Di), counter (CX)
- address range
  direct addressing – limited
  displacement addressing – length of the address register
- addressing granularity
  byte or word



I Indirect bit

Figure 11.5 PDP-10 Instruction Format



Numbers below fields indicate bit length
Source and destination each contain a 3-bit addressing mode field and a 3-bit register number
FP indicates one of four floating-point registers
R indicates one of the general-purpose registers
CC is the condition code field

Figure 11.6 Instruction Formats for the PDP-11

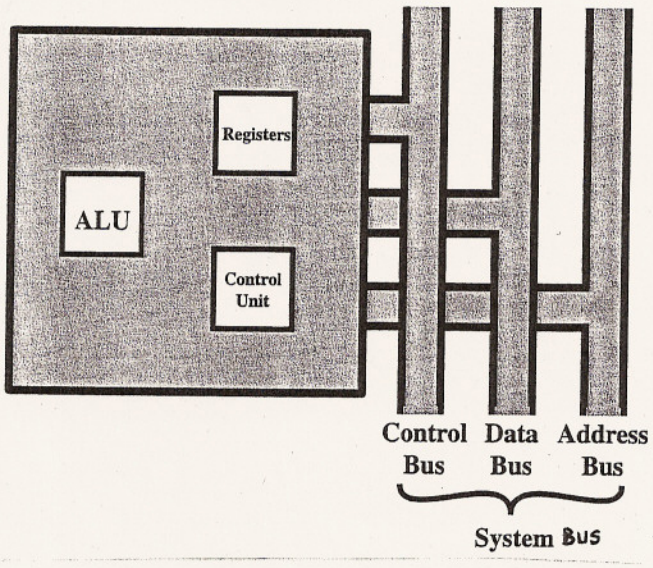
CHAPTER 12. Processor Structure and Function

12.1 PROCESSOR ORGANIZATION

function

- **Fetch instruction:**
- **Interpret instruction:**
- **Fetch data:**
- **Process data:**
- **Write data:**

Structure



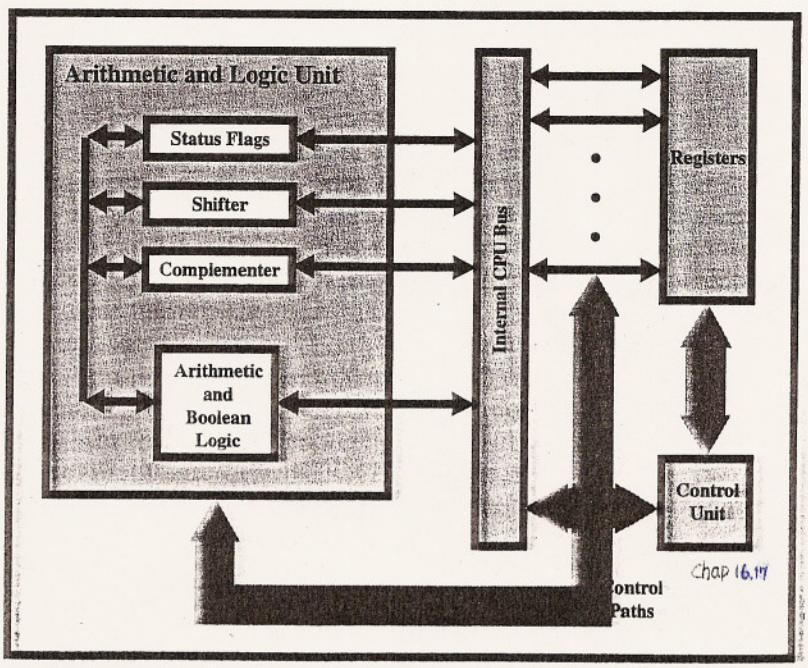


Figure 12.2 Internal Structure of the CPU.

12.2 REGISTERS

(1) User-visible Registers

Ex. IBM S/370

- o 16 general-purpose registers – addressing, data, indexing
- o 4 floating-point registers

Ex. Intel Microprocessor

- data register: AX, BX, CX, DX
- address register
  - segment register: SS, DS, CS, ES
  - index register: SI, DI
  - stack pointer: BP

Issues

1. general vs. special - trade-off, no clear-cut
2. number of registers
   - one register – accumulator
   - 8 ~ 32 registers (common)
   - hundreds – RISC
3. length – word, double-word

(2) System Registers

- o control (to execute programs)
  - . PC
  - . IR
  - . MAR – address bus
  - . MBR – data bus
- o status (PSW)
  - sign
  - zero
  - carry
  - overflow
  -

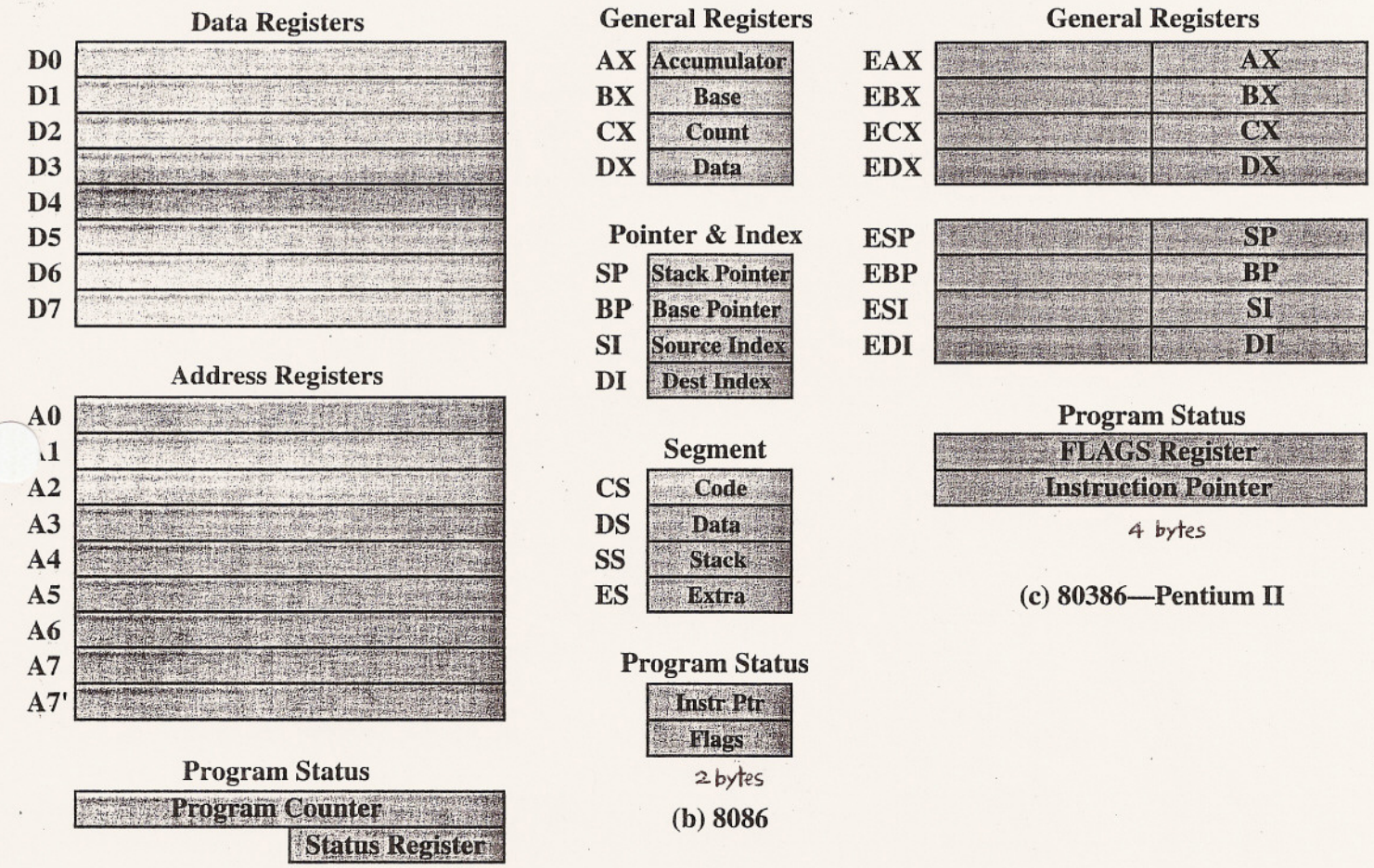


Figure 12.3 Example Microprocessor Register Organizations.

12.3 INSTRUCTION CYCLE

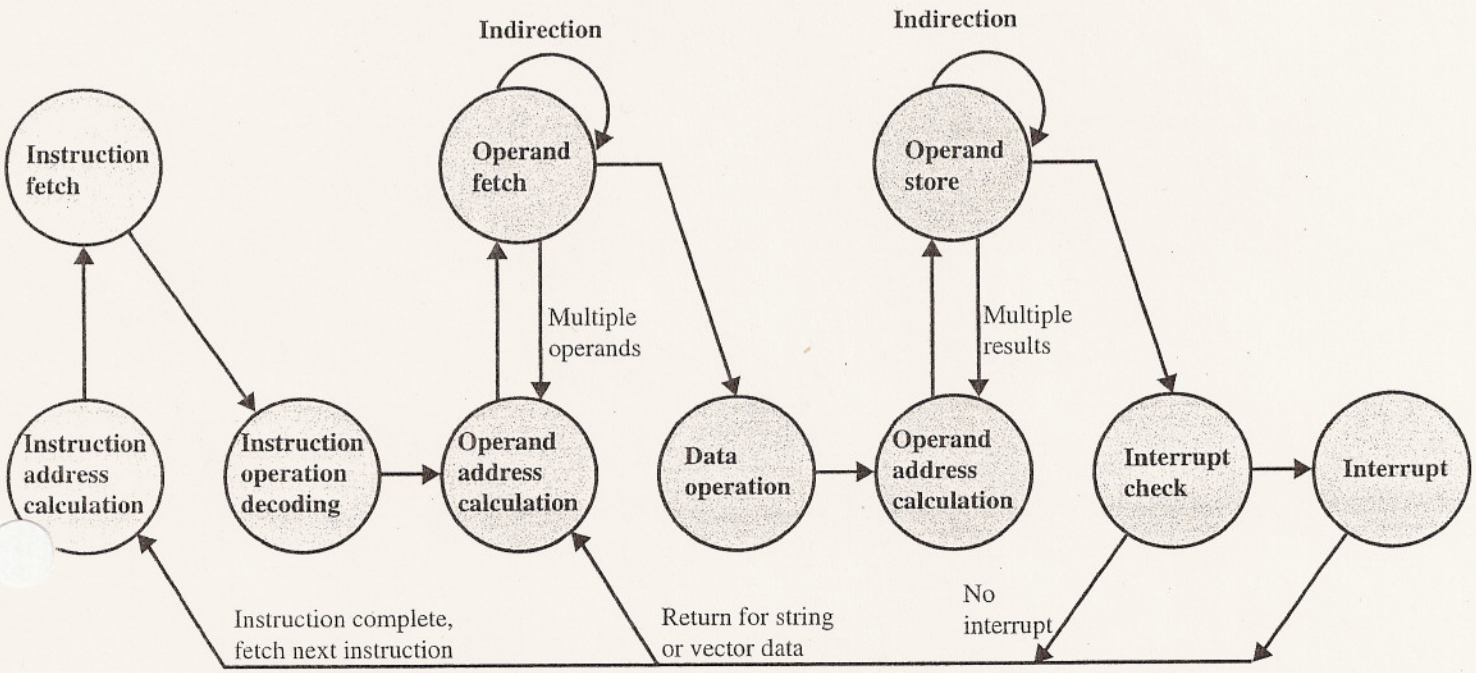


Figure 12.5 Instruction Cycle State Diagram

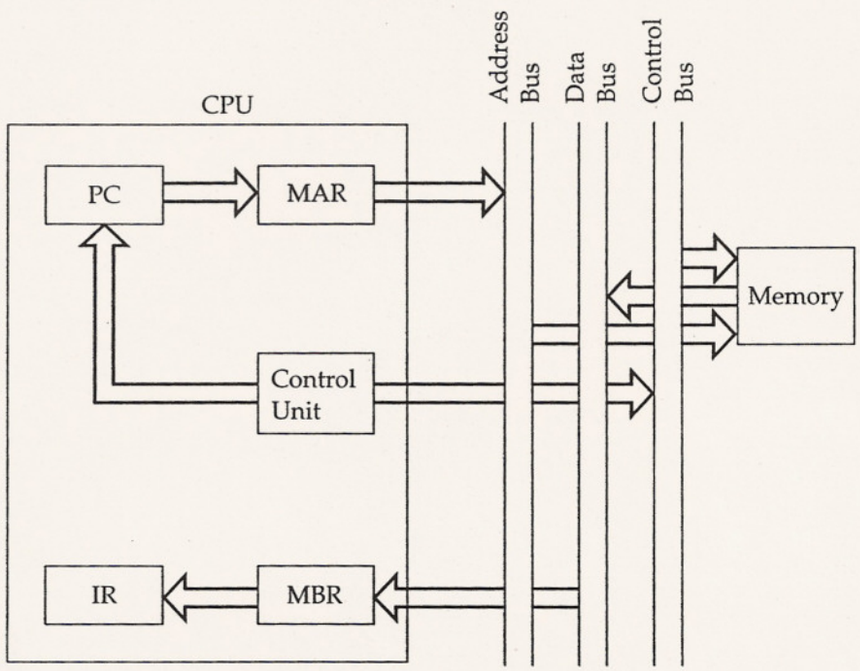


FIGURE 12.6. Data flow, fetch cycle

t_1: MAR ← (PC)
t_2: MBR ← Memory
PC ← (PC)+1
t_3: IR ← (MBR)

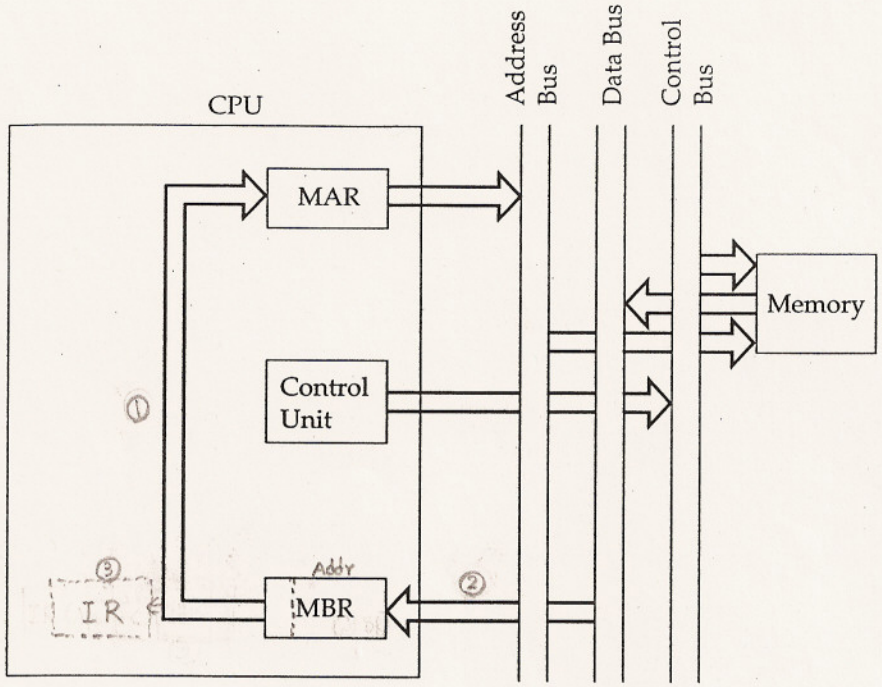


FIGURE 12.7. Data flow, indirect cycle

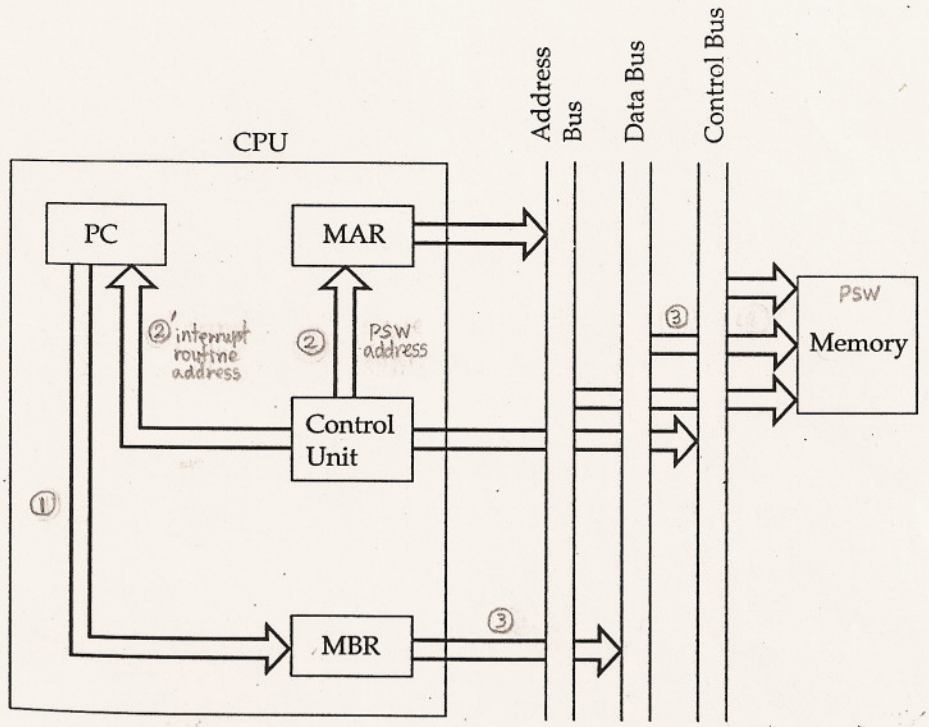


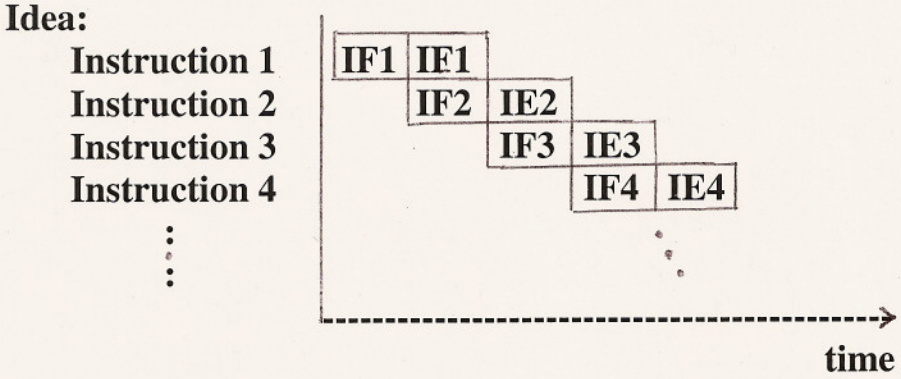
FIGURE 12.8. Data flow, interrupt cycle.

Methods that improves Performance

- bit-parallel arithmetic (carry look-ahead adder)
- separate I/O processor – interrupt-driven (DMA)
- memory interleaving – lower memory contentions
- associative memory – cache
- multiple registers
- instruction pipelining – IBM 7090 (1959)
- multiple functional unit – CDC 6600/7600
- pipelined functional unit
  - STAR 100 – pipelined adder (4 pipes)
  - Vector supercomputers (Cray, Fujitsu, NEC)

Instruction Pipeline

Idea:



Note. Pipeline lengths may not be the same. Usually, |IE| > |IF|
Conditional branch instruction
The more stages, the higher speedup
As the number of pipes increases, overhead increases, too.
(logic controlling the gating between stages)

Ex. Six-stage pipeline

FI → DI → CO → FO → EI → WO

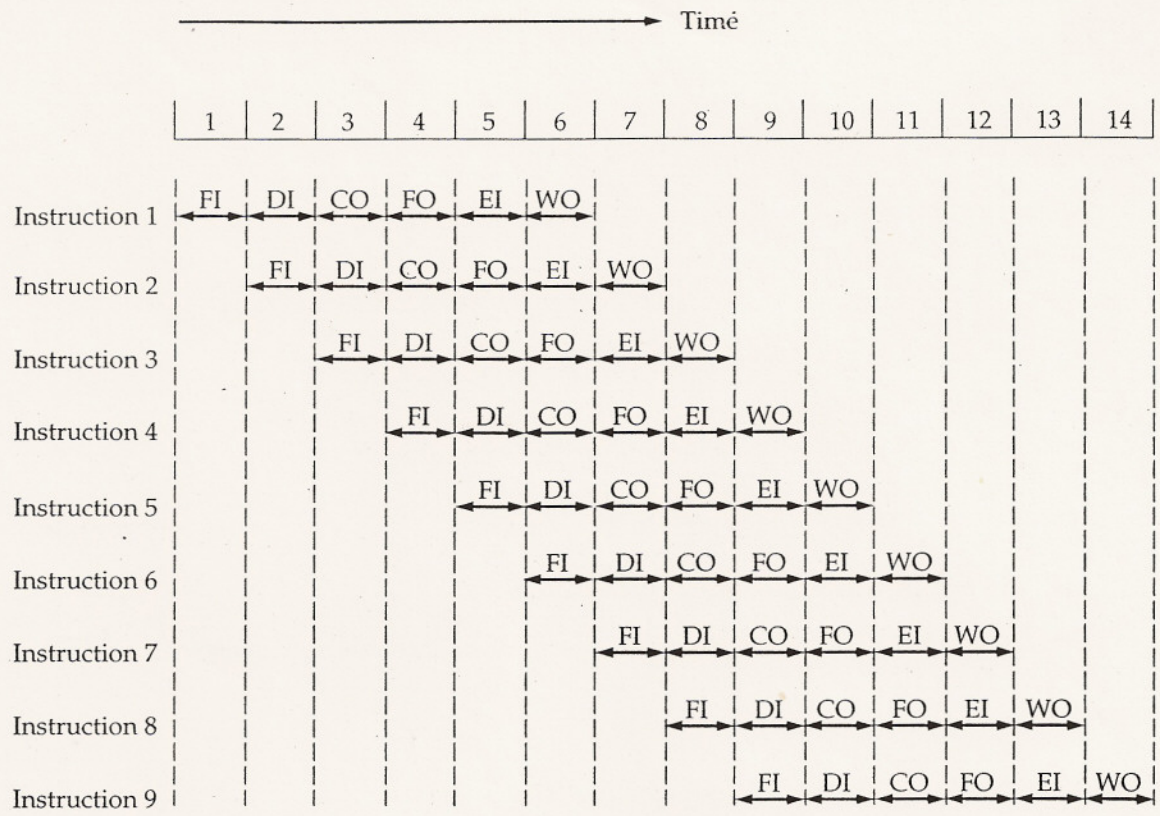


FIGURE 12.10. Timing Diagram for instruction pipeline operation

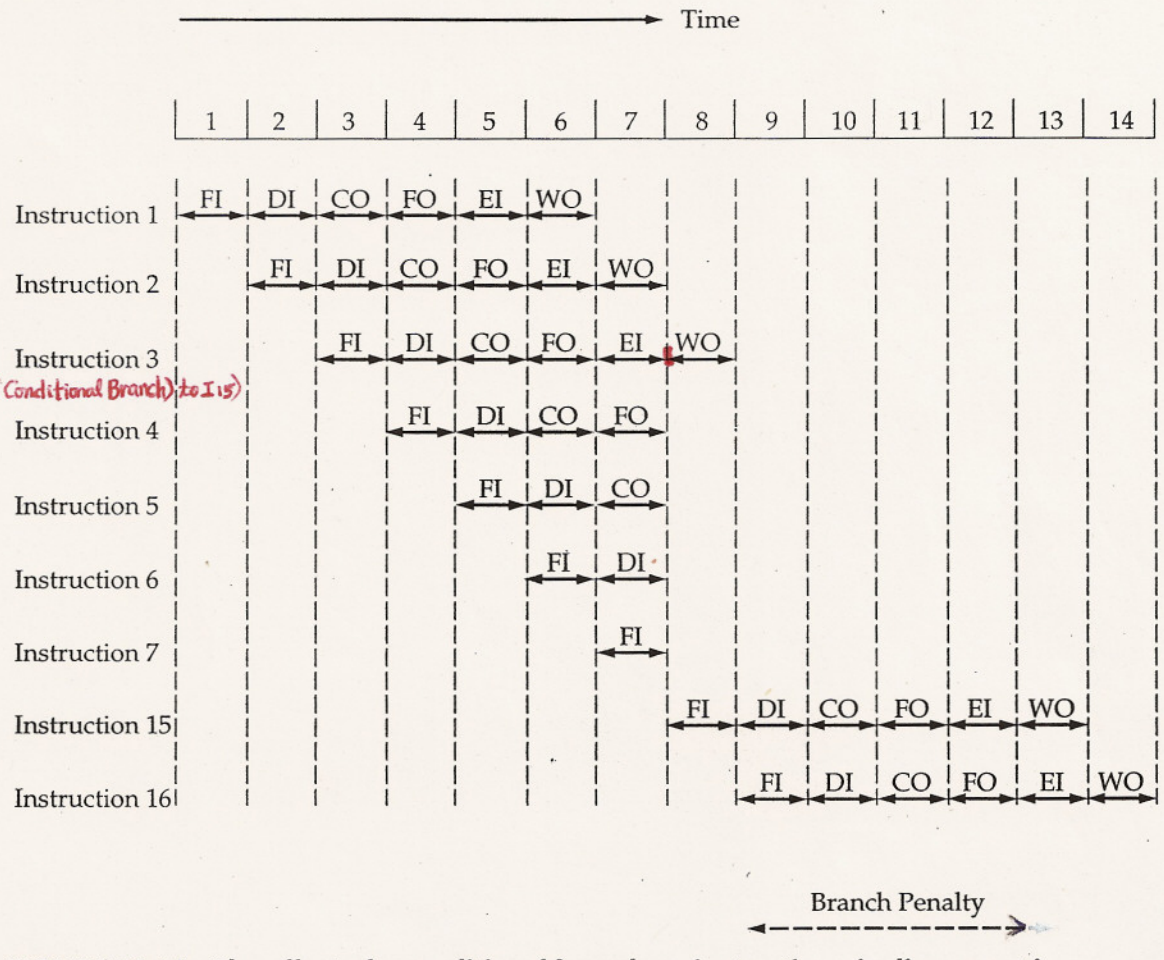


FIGURE 12.11. The effect of a conditional branch on instruction pipeline operation

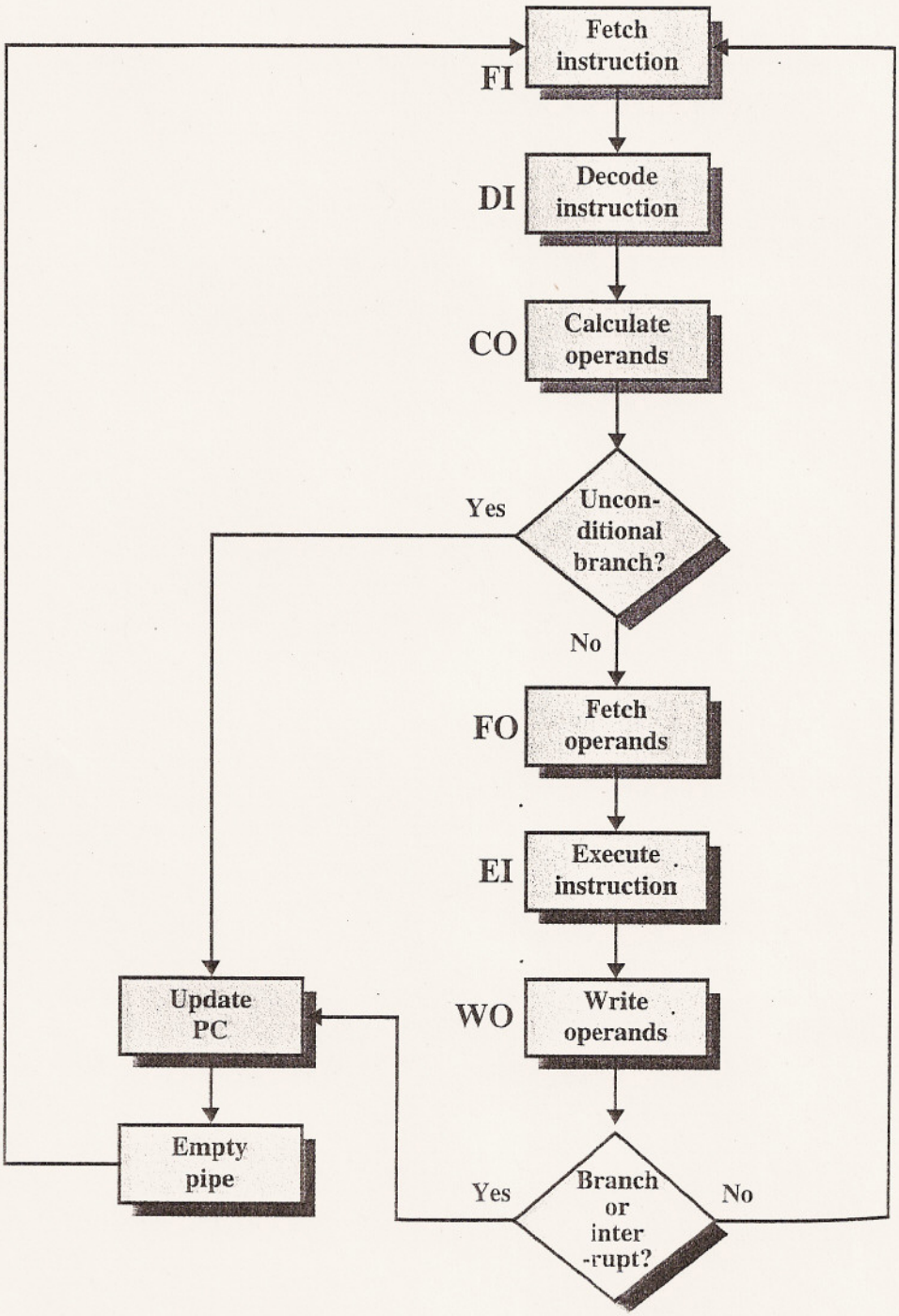


Figure 12.12 Six-Stage CPU Instruction Pipeline

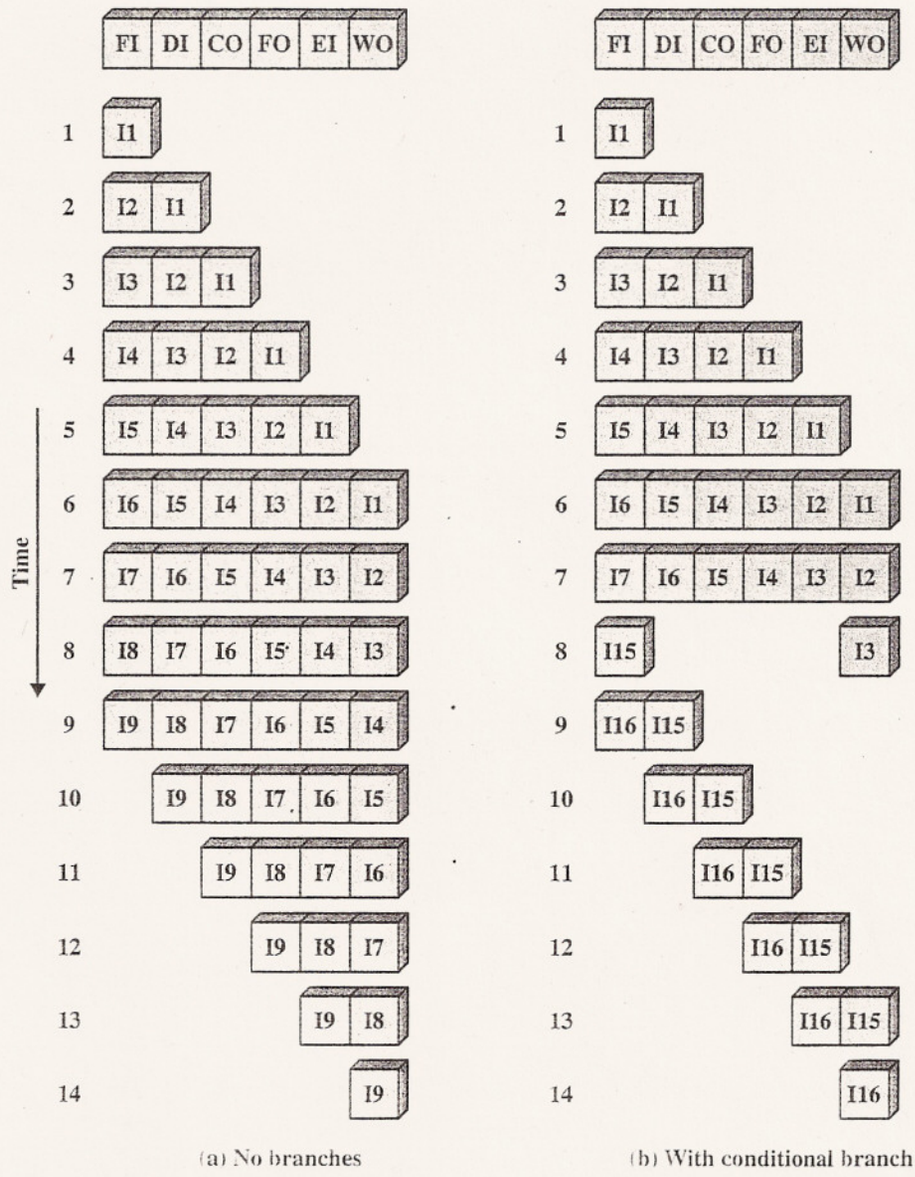
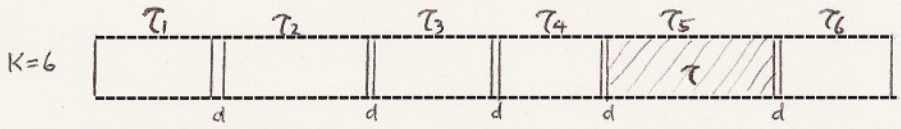


Figure 12.13 An Alternative Pipeline Depiction

Instruction Pipelining Performance



$\tau = \max_i [\tau_i] + d, \quad 1 \le i \le k$

$= \tau_m + d$ // d is time delay between stages – to move signals/data //

In general, $\tau_m >> d \rightarrow \tau \fallingdotseq \tau_m$

Let Tk,n be total time taken for n instructions on k-stage pipeline

$T_{k,n} = [k + (n-1)]\,\tau$ // (n-1) for initial pipe filling / pipe flushing //

Example. 8th instruction completion time (Fig 12.10)

Here, k = 6, n = 8

T6,8 = [6 + 7] = 13

Def. Speedup

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{n\,k\,\tau}{[k + (n-1)]\,\tau} = \frac{nk}{k + (n-1)}$$

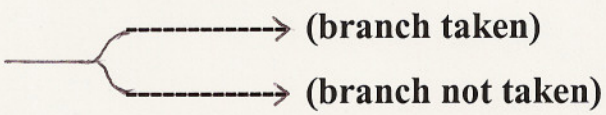
Note. $\lim_{n \to \infty} S_k \fallingdotseq k$

- The larger the number of pipeline stages, the larger the speedup.
- As the number of stages increases, the complexity/cost increases.

Conclusion. Instruction pipelining is a powerful technique for enhancing performance, but requires careful design to achieve optimum results with reasonable complexity.

Conditional Branch

(1) Multiple pipelines

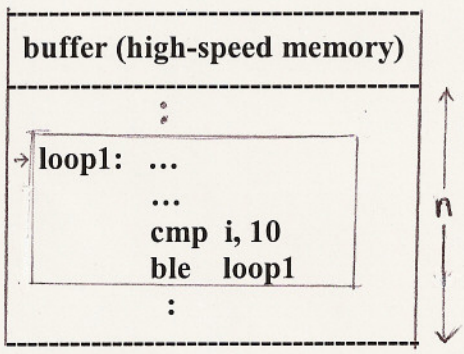


- IBM System 370/168, IBM 3033
- expensive

(2) Pre-fetch branch target

- IBM System 360/91

(3) Loop buffer



- n most recently used instructions – pre-fetched (only once)
- consecutive instructions, such as loop
- similar to cache
- CDC 6600, STAR 100, Cray 1

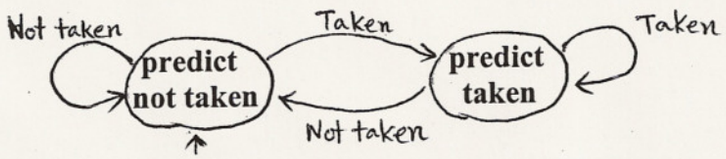
(4) Branch prediction

Static

- branch never taken – Vax 11/780
- branch always taken
  (not applied when page fault expected)
- predict by operation code – 75%

Dymanic

- **one-bit dynamic branch prediction** (switch)



- **Two-bit dynamic branch prediction** (last 2 branch history)

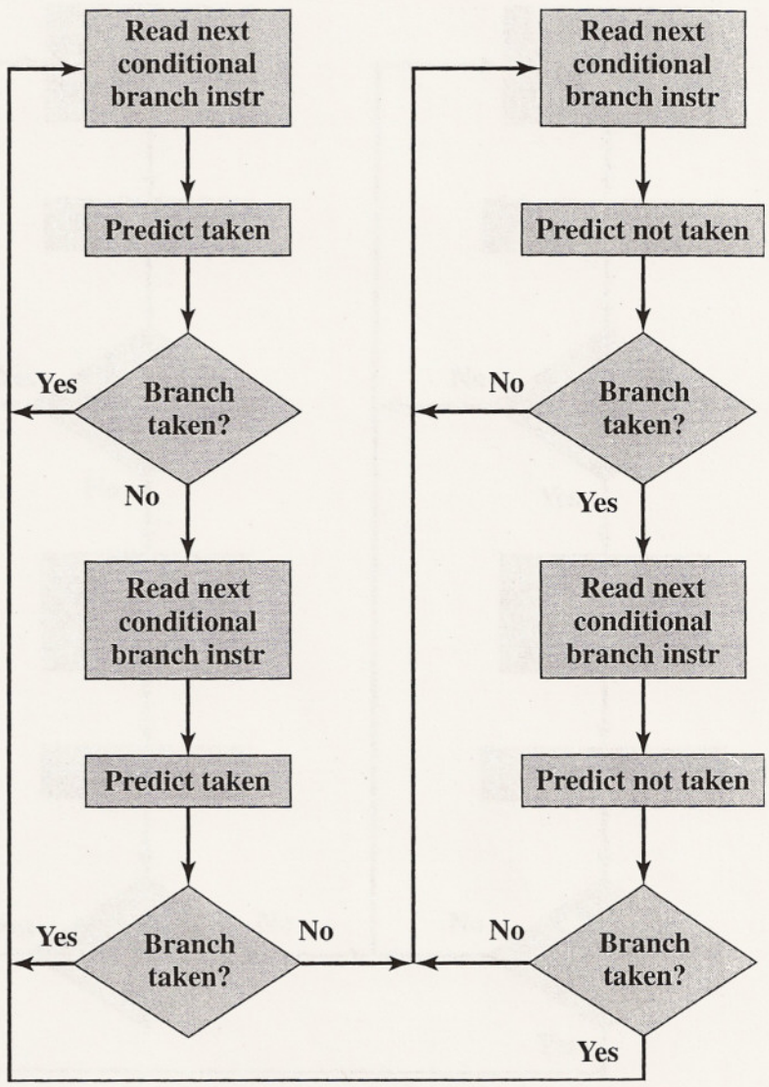


Figure 12.16 Branch Prediction Flowchart

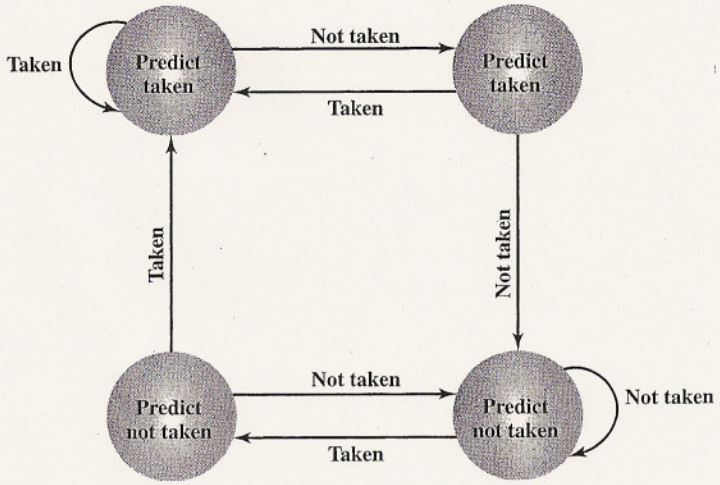


Figure 12.17 Branch Prediction State Diagram

Example - nested loop

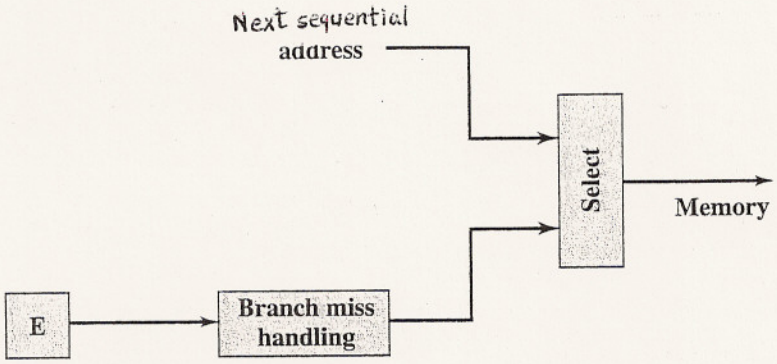
```
for i:= 1 to n do
    for j:= 1 to 4 do
```

(1) 1-bit dynamic branch prediction

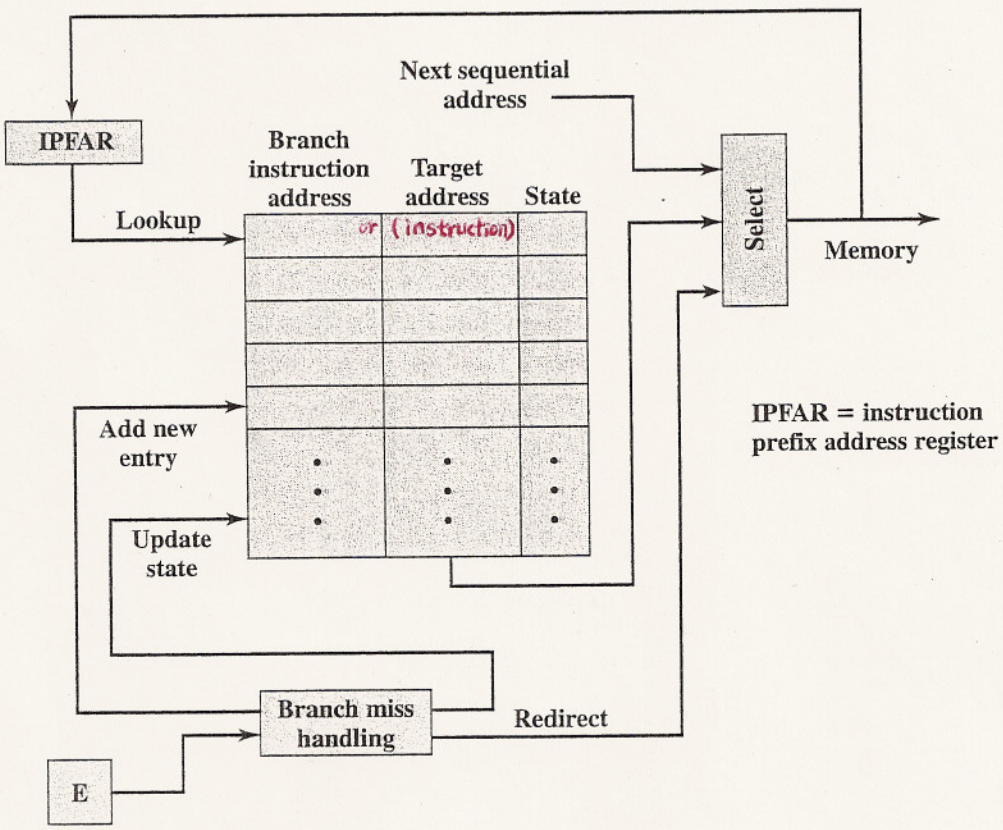
Prediction	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	...
Taken	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	...
incorrect	x				x	x				x	x					

(2) 2-bit dynamic branch prediction

Prediction	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	...
Taken	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	...
incorrect	x	x			x					x						

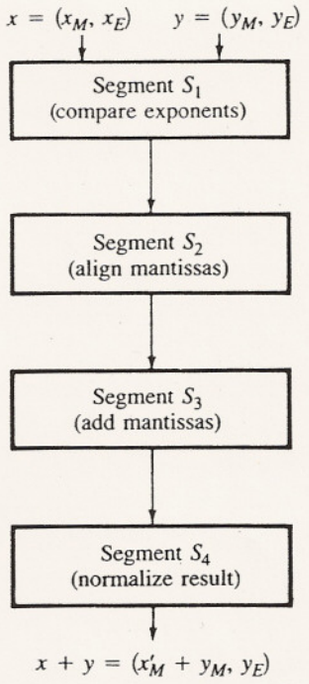


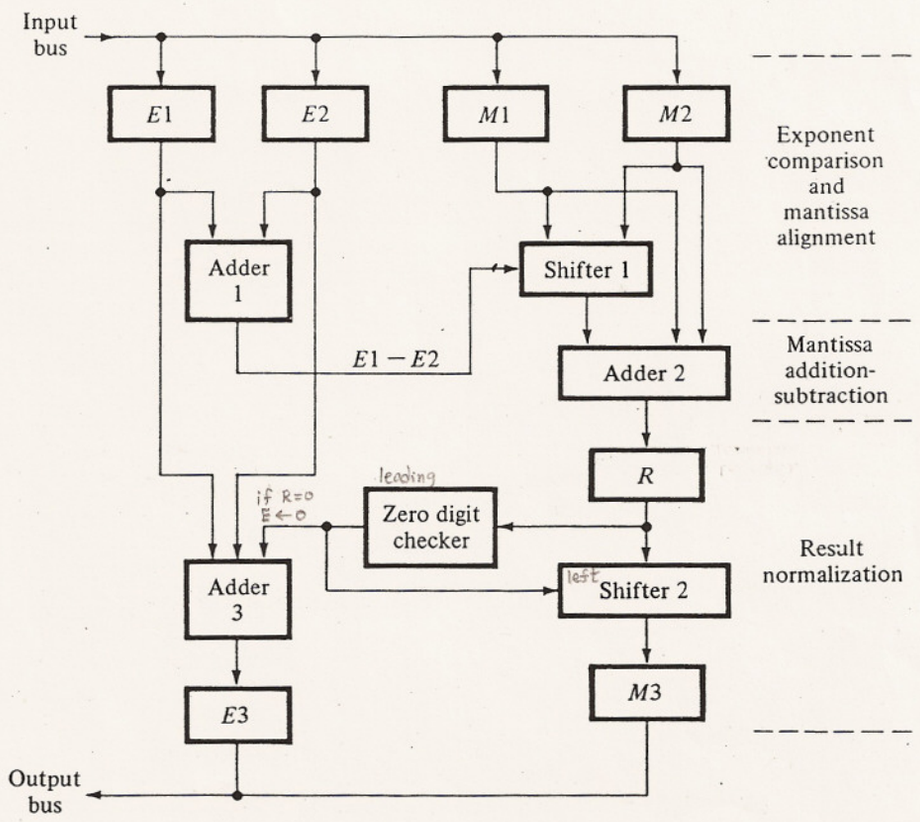
(a) Predict never taken strategy



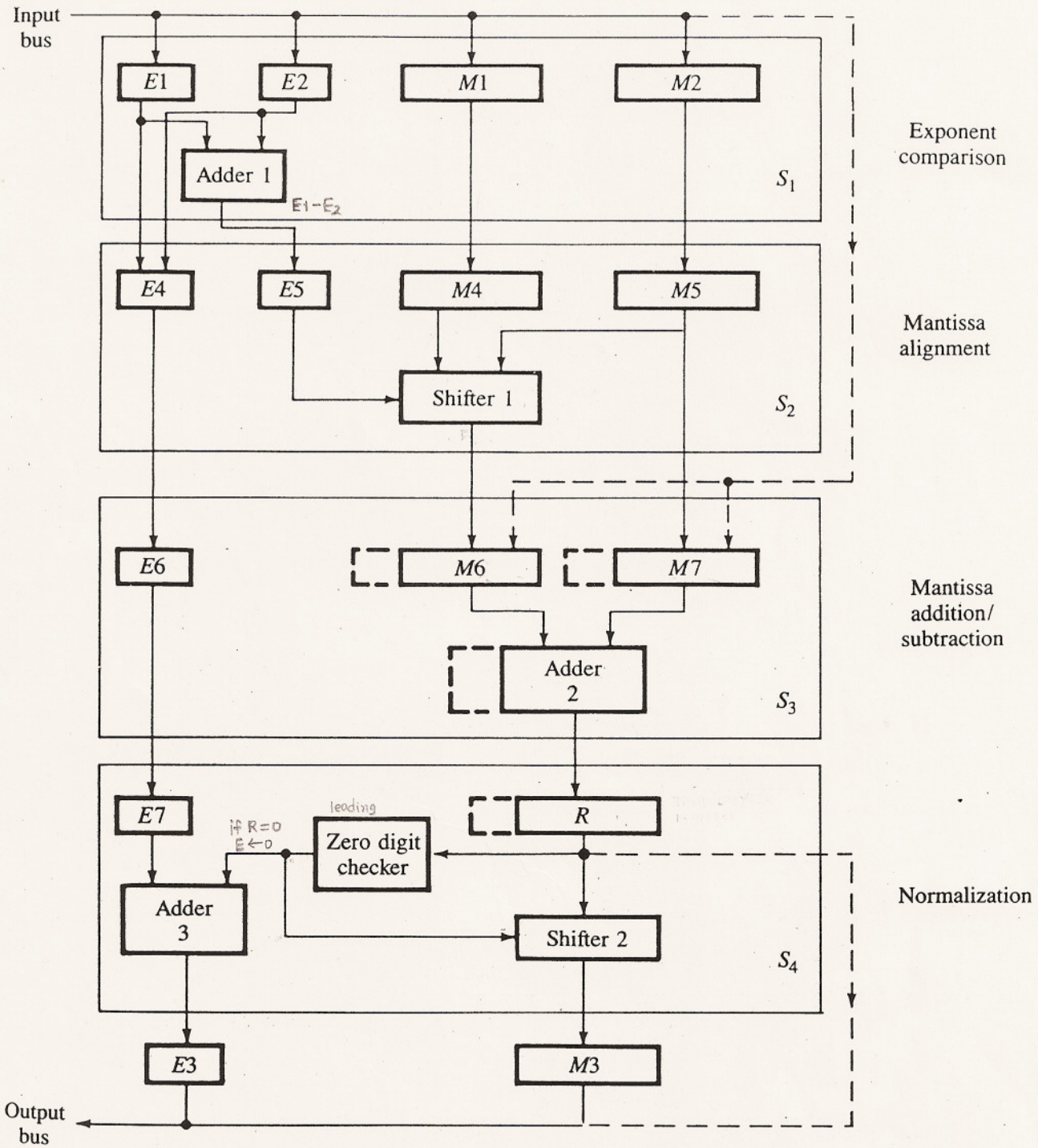
(b) Branch history table strategy

Figure 12.18 Dealing with Branches AMD 29000





Floating-point adder of the IBM System/360 Model 91.



Pipelined version of the floating-point adder

Chapter 13 RISC

Major Advances

- separate I/O processor
- family concept
- microprogrammed control unit
- memory hierarchy – cache, virtual memory (demand page)
- pipelining – instruction pipeline, vector processor (data pipeline)
- parallel processor
- RISC architecture

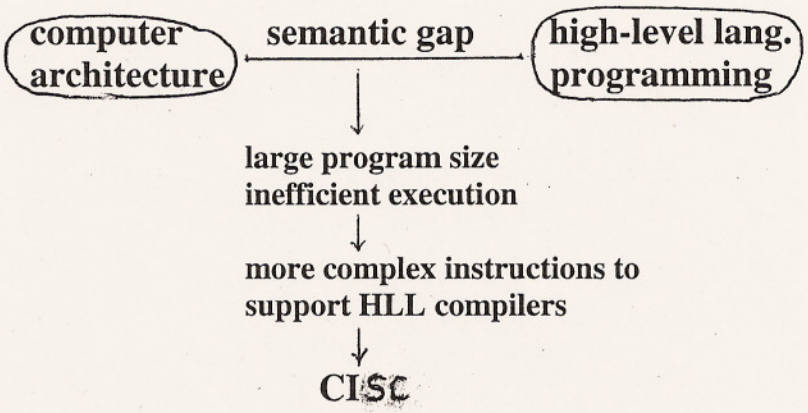
RISC Architecture

- a limited and simple instruction set
- a large number of general-purpose registers – fast operand referencing
- careful design of instruction pipeline – conditional branch and procedure call

Table 13.1 Characteristics of Some CISCs, RISCs, and Superscalar Processors

	Complex Instruction Set (CISC) Computer			Reduced Instruction Set (RISC) Computer		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	Berkeley RISC I
Year developed	1973	1978	1989	1987	1991	1981
Number of instructions	208	303	235	69	94	31
Instruction size (bytes)	2–6	2–57	1–11	4	4	
Addressing modes	4	22	11	1	1	
Number of general-purpose registers	16	16	8	40 - 520	32	138
Control memory size (Kbits)	420	480	246	—	—	
Cache size (KBytes)	64	64	8	32	128	

Instruction Execution Characteristics



RISC ➔ make the architecture that supports the HLL simpler

Background and Motivation

Instruction execution characteristics
- **operations performed**
- **operands used**
- **execution sequencing**

- **Operation**

TABLE 13.2 Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

	Dynamic Occurrence		*Machine-Instruction Weighted*		*Memory-Reference Weighted*	
	Pascal	*C*	*Pascal*	*C*	*Pascal*	*C*
ASSIGN	45	38	13	13	14	15
LOOP	5	3	42	32	33	26
CALL	15	12	31	33	44	45
IF	29	43	11	21	7	13
GOTO	—	3	—	—	—	—
OTHER	6	1	3	1	2	1

dynamic frequency: Assignment 45%
Sequence control 34% } **optimize**
timing: procedure call/return – most time-consuming

→ careful design of instruction pipelining (∵ high propotion of conditional branch and procedure call)

- **Operands**

TABLE 13.3 Dynamic Percentage of Operands

	Pascal	*C*	*Average*
Integer Constant	16	23	20
Scalar Variable	58	53	55
Array/Structure	26	24	25

80% are local variables → localized scalar - subject to optimize
→ large number of registers (to reduce memory access)

o Procedure Call

arguments: 98% fewer than 6 arguments
local variables: 92% fewer than 6 local variables
depth is narrow

→ simple instruction set

Large Number of Registers

- register is the fastest among all (cache, main memory, …)
strategy: most frequently accessed operands should be kept in registers.
1. compiler – smart compiler for optimization (software approach)
2. more registers (hardware approach)

Overlapping Register Windows

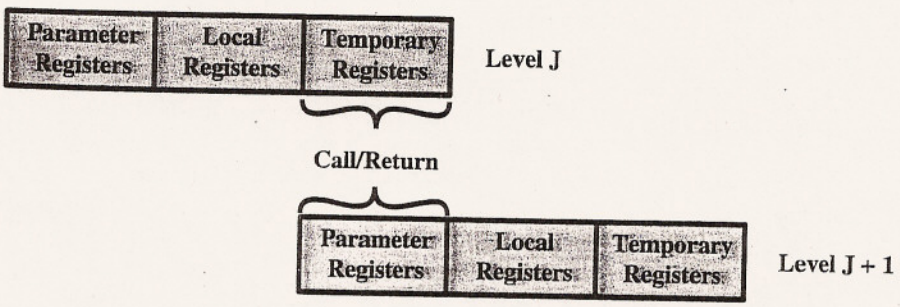
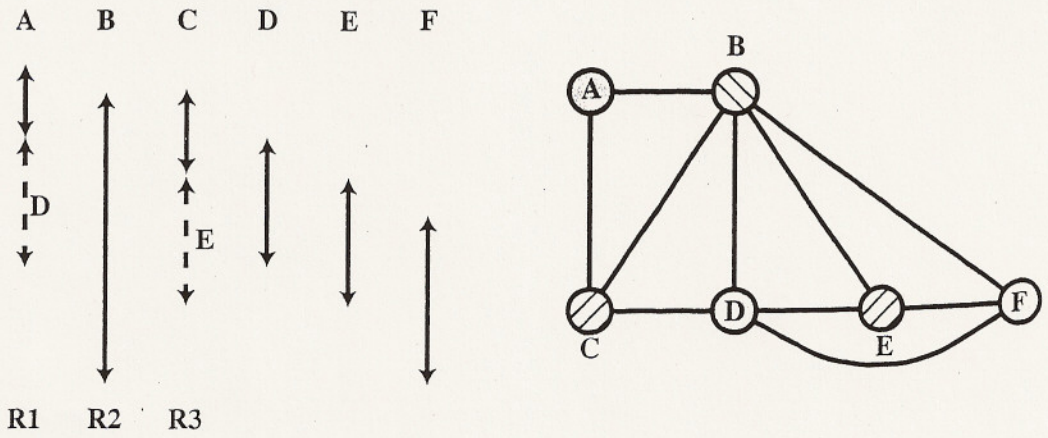


Figure 13.1 Overlapping Register Windows.

13.3 Compiler-Based Register Optimization

symbolic registers
physical registers
mapping (Find the way to use minimum # of registers)



(a) Time Sequence of Active Use of Registers (b) Register Interference Graph

Figure 13.4 Graph Coloring Approach.

Note. 32 ~ 64 registers are enough with register optimization technique

13.4 Reduced Instruction Set Architecture

Why CISC?

(1) Simple compiler – machine instruction ≒ HLL instruction
(2) Smaller program – expect less memory, fewer instruction fetch, less page faults, etc
(3) Faster program

Reality

- CISC program not much shorter

Table 13.6 Code Size Relative to RISC I

	[PATT82a] 11 C Programs	[KATE83] 12 C Programs	[HEAT84] 5 C Programs
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

- Speed
  CICS – memory
  RISC – registers See Fig 13.5

Note: Because of large number of registers in use, RISC instruction needs less number of memory access → fast execution

Characteristics of RISC

- one instruction per cycle – hard-wired machine instruction (instead of microcoding)
- register-to-register operations – simple instruction set and simple control unit – one size
- simple addressing modes – no indirect addressing
- simple instruction formats

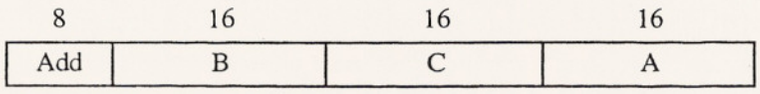
→ Microchip design process becomes easier.

Table 13.7 Design and Layout Effort for Some Microprocessors

CPU	Transistors (thousands)	Design (person-months)	Layout (person-months)
RISC I	44	15	12
RISC II	41	18	12
M68000	68	100	70
Z8000	18	60	70
Intel iAPx-432	110	170	90

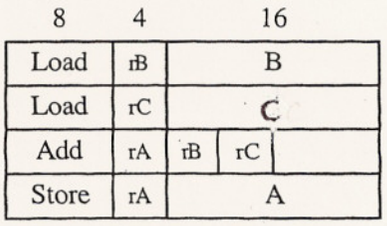
Trend

Hybrid



8	16	16	16
Add	B	C	A

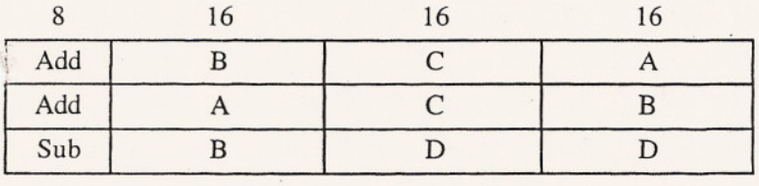
Memory-to-Memory
I = 56, D = 96, M = 152



8	4	16		
Load	rB	B		
Load	rC	C		
Add	rA	rB	rC	
Store	rA	A		

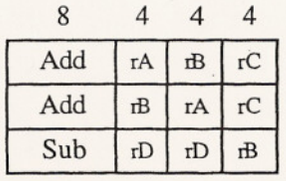
Register-to-Memory
I = 104, D = 96, M = 200

(a) $A \leftarrow B + C$



8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

Memory-to-Memory
I = 168, D = 288, M = 456



8	4	4	4
Add	rA	rB	rC
Add	rB	rA	rC
Sub	rD	rD	rB

Register-to-Memory
I = 60, D = 0, M = 60

(b) $A \leftarrow B + C$; $B \leftarrow A + C$; $D \leftarrow D - B$

I = Size of executed instructions
D = Size of executed data
M = I + D = Total memory traffic

Figure 13.5 Two Comparisons of Register-to-Register and Memory-to-Memory Approaches.

IW = 32 bits

Table 13.7 Characteristics of Some Processors

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

[a]RISC that does not conform to this characteristic.
[b]CISC that does not conform to this characteristic.