PIPELINING AND VECTOR PROCESSING

- Parallel Processing
- Pipelining
- Arithmetic Pipeline
- Instruction Pipeline
- RISC Pipeline
- Vector Processing
- Array Processors

PARALLEL PROCESSING

Execution of *Concurrent Events* in the computing process to achieve faster *Computational Speed*

Levels of Parallel Processing

- Job or Program level
- Task or Procedure level
- Inter-Instruction level
- Intra-Instruction level

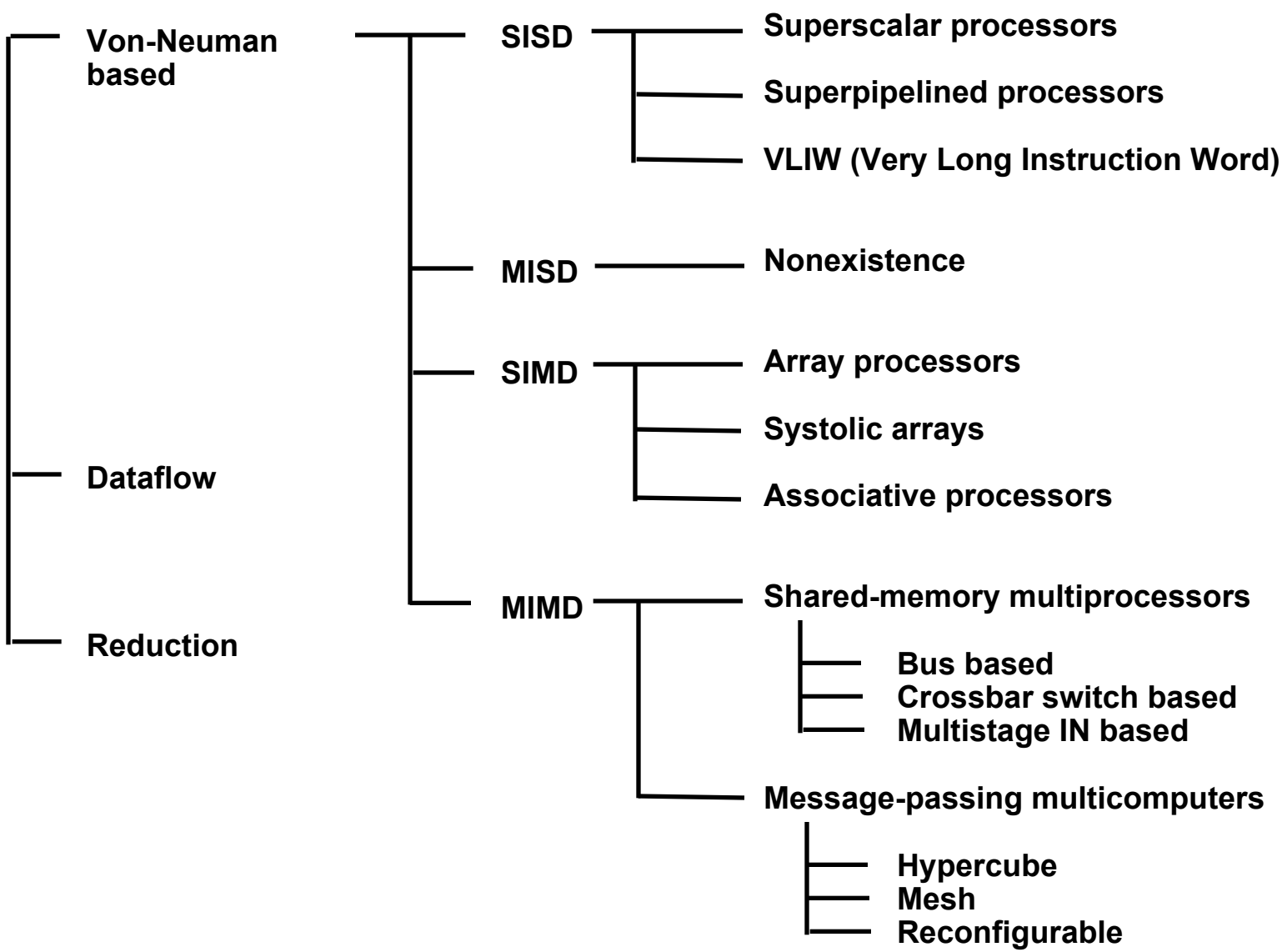
PARALLEL COMPUTERS

Architectural Classification

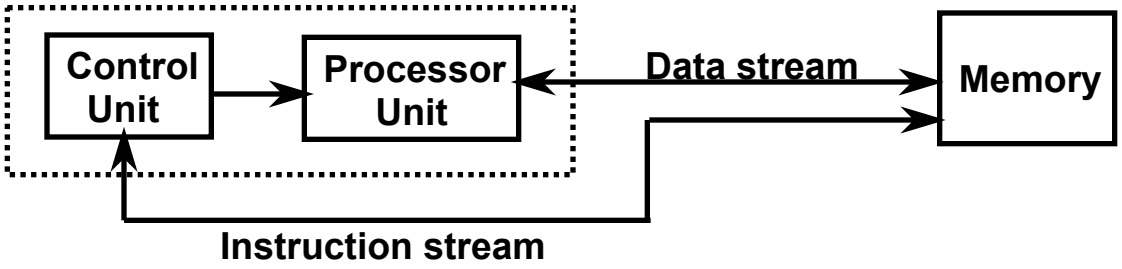
– Flynn's classification
  » Based on the multiplicity of *Instruction Streams* and *Dat*
  » Instruction Stream
    • Sequence of Instructions read from memory
  » Data Stream
    • Operations performed on the data in the processor

		Number of *Data Streams*	
		Single	Multiple
Number of *Instruction Streams*	Single	SISD	SIMD
	Multiple	MISD	MIMD

COMPUTER ARCHITECTURES FOR PARALLEL PROCESSING



SISD COMPUTER SYSTEMS



Characteristics

- Standard von Neumann machine
- Instructions and data are stored in memory
- One operation at a time

Limitations

Von Neumann bottleneck

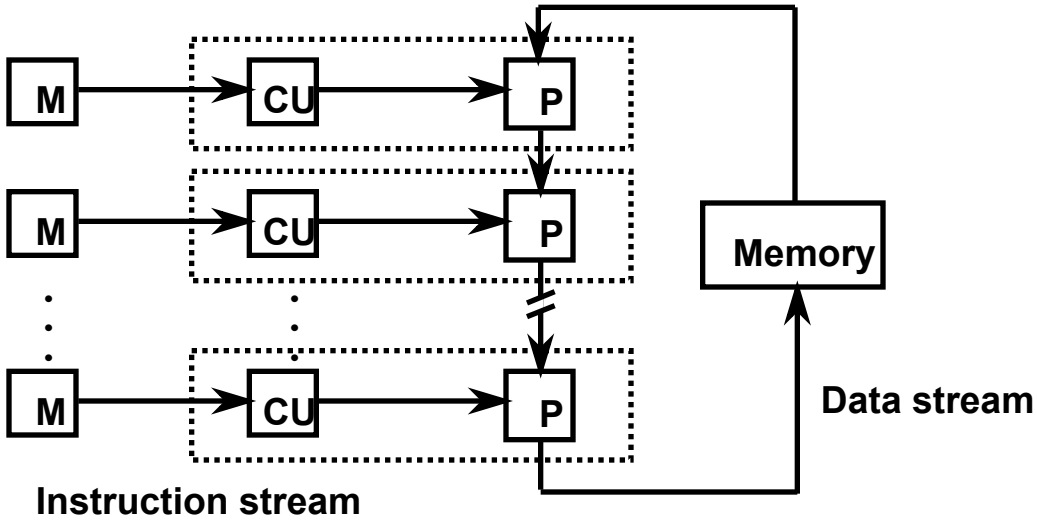
Maximum speed of the system is limited by the *Memory Bandwidth* (bits/sec or bytes/sec)

- Limitation on *Memory Bandwidth*
- Memory is shared by CPU and I/O

SISD PERFORMANCE IMPROVEMENTS

- Multiprogramming
- Spooling
- Multifunction processor
- Pipelining
- Exploiting instruction-level parallelism
  - Superscalar
  - Superpipelining
  - VLIW (Very Long Instruction Word)

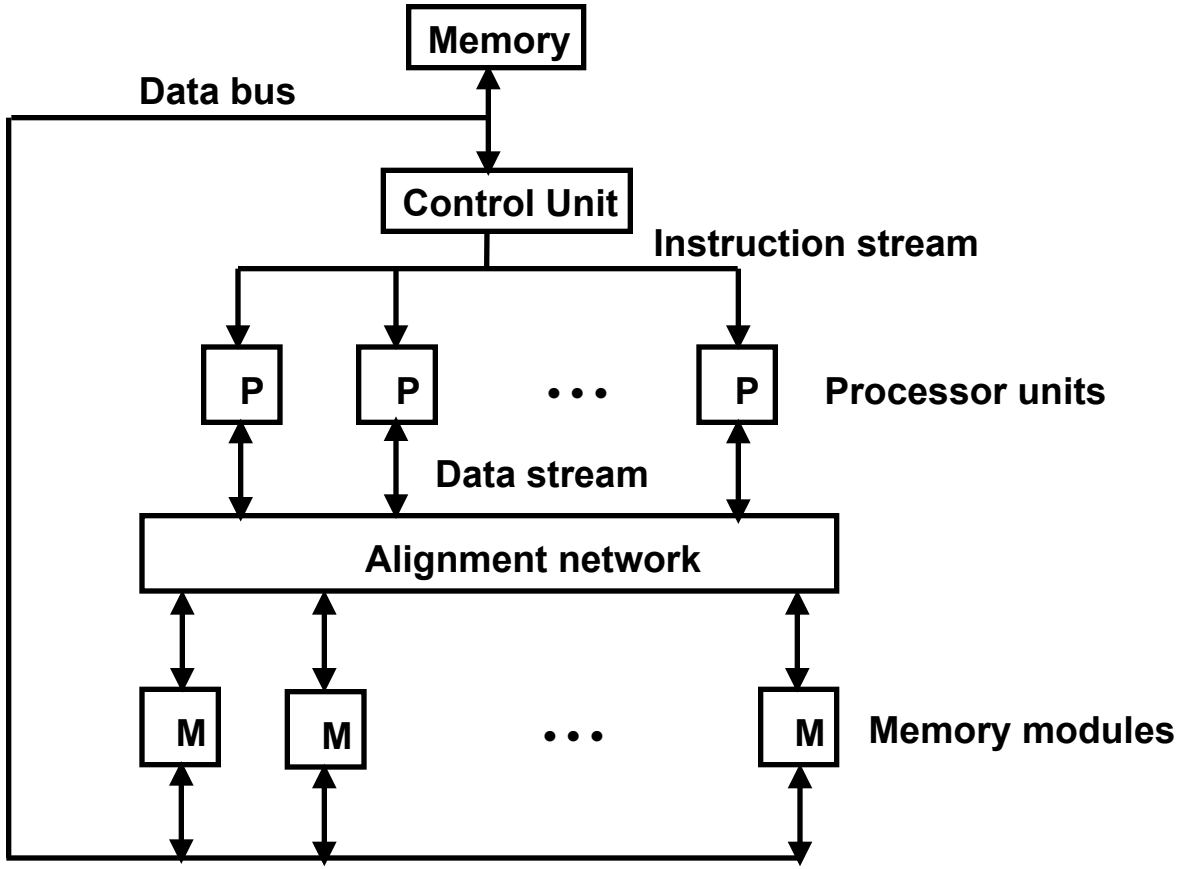
MISD COMPUTER SYSTEMS



Characteristics

- There is no computer at present that can be classified as MISD

SIMD COMPUTER SYSTEMS



Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time

TYPES OF SIMD COMPUTERS

Array Processors

- The control unit broadcasts instructions to all Processor Units, and all active Processor Units execute the same instructions
- ILLIAC IV, GF-11, Connection Machine, DAP, MPP

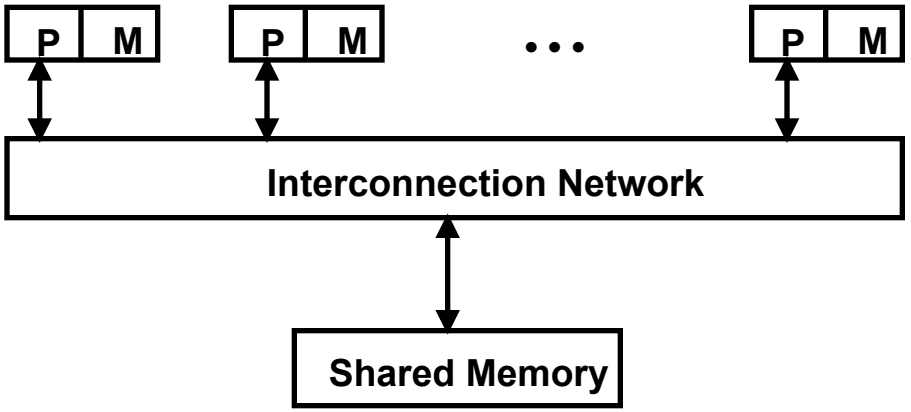
Systolic Arrays

- Regular arrangement of a large number of very simple processors constructed on VLSI circuits
- CMU Warp, Purdue CHiP

Associative Processors

- Content addressing
- Data transformation operations over many sets of arguments with a single instruction
- STARAN, PEPE

MIMD COMPUTER SYSTEMS



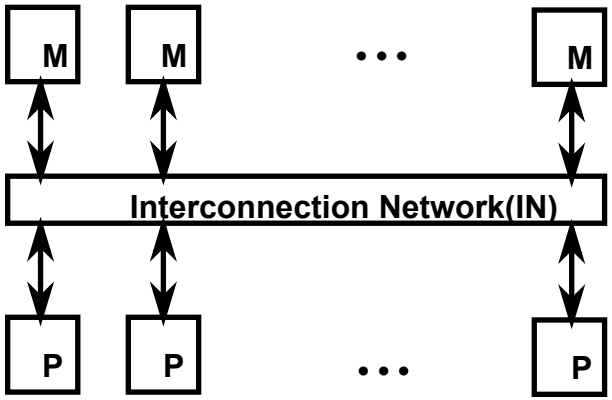
Characteristics

- Multiple processing units
- Execution of multiple instructions on multiple data

Types of MIMD computer systems

- Shared memory multiprocessors
- Message-passing multicomputers

SHARED MEMORY MULTIPROCESSORS



Buses,
Multistage IN,
Crossbar Switch

Characteristics

All processors have equally direct access to one large memory address space

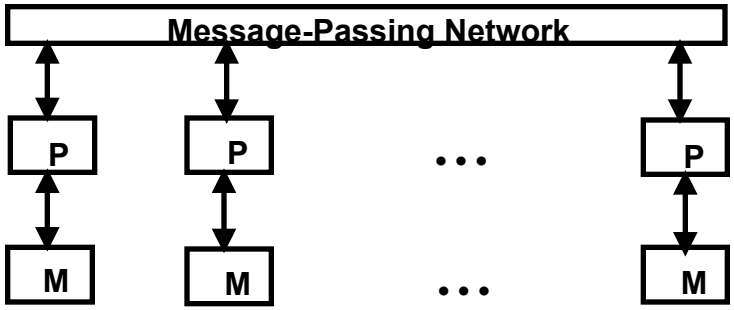
Example systems -

Bus and cache-based systems - Sequent Balance, Encore Multimax
Multistage IN-based systems - Ultracomputer, Butterfly, RP3, HEP
Crossbar switch-based systems- C.mmp, Alliant FX/8

Limitations

Memory access latency, Hot spot problem

MESSAGE-PASSING MULTICOMPUTER



Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

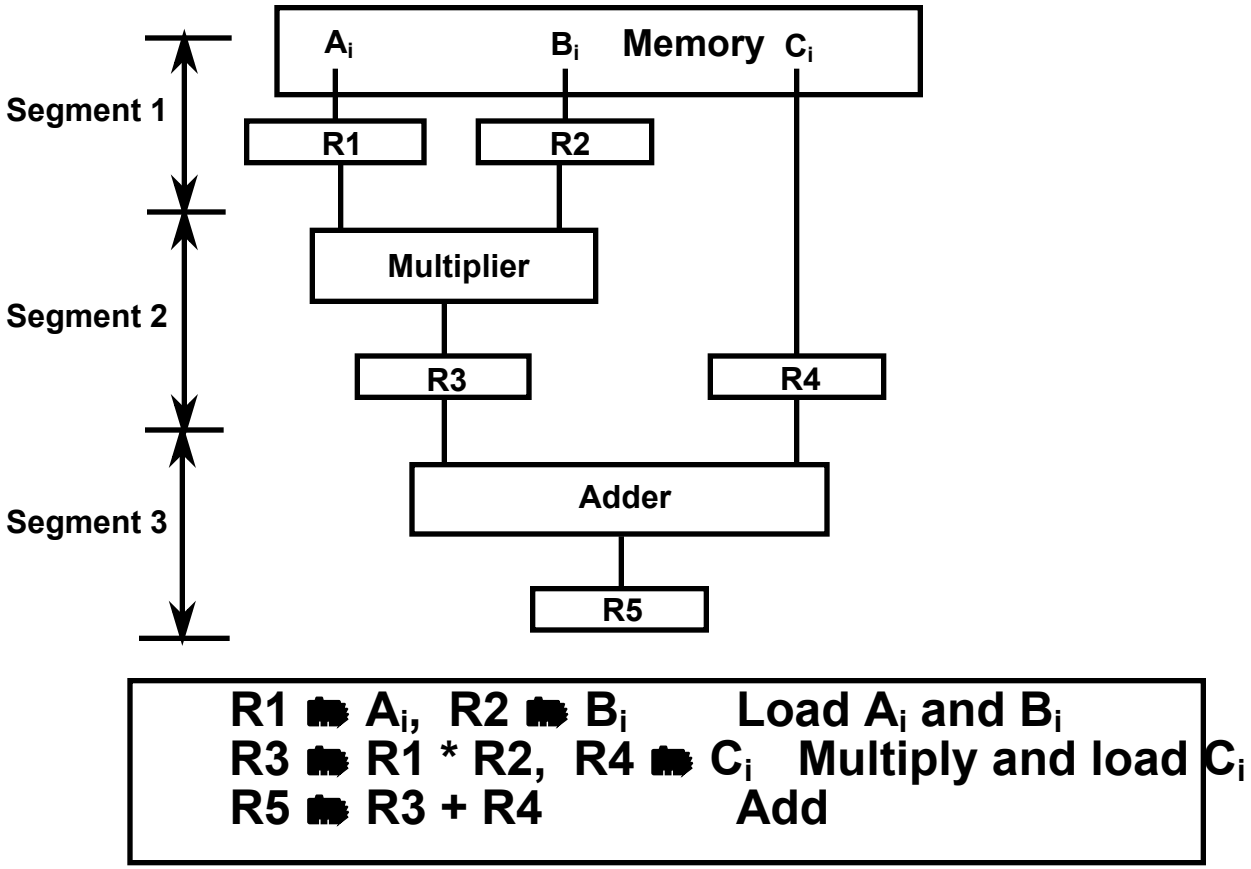
Limitations

- Communication overhead - Hard to programming

PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7



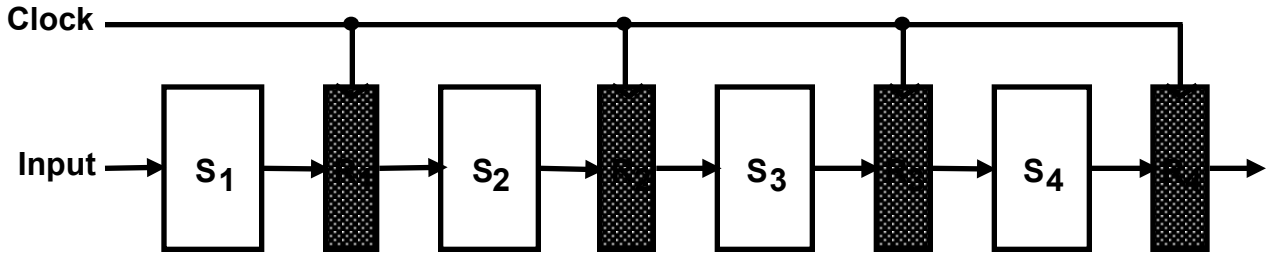
R1 ← A_i, R2 ← B_i Load A_i and B_i
R3 ← R1 * R2, R4 ← C_i Multiply and load C_i
R5 ← R3 + R4 Add

OPERATIONS IN EACH PIPELINE STAGE

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram

Segment	1	2	3	4	5	6	7	8	9
1	T1	T2	T3	T4	T5	T6			
2		T1	T2	T3	T4	T5	T6		
3			T1	T2	T3	T4	T5	T6	
4				T1	T2	T3	T4	T5	T6

Clock cycles

Ti = Task

PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycles

t_1: Time required to complete the n tasks

$$t_1 = n * t_n$$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation, Time of the Largest segment)

t_k: Time required to complete the n tasks

$$t_k = kt_p + (n - 1)t_p = (k + n - 1) * t_p$$

Speedup

S_k: Speedup

$$S_k = n*t_n / (k + n - 1)*t_p$$

$$\lim_{n \to \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k * t_p)$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline (k =4)
- subopertion in each stage; t_p = 20nS
- 100 tasks to be executed (n=100)
- 1 task in non-pipelined system; 20*4 = 80nS (tn or k*tp)

Pipelined System

$(k + n - 1)*t_p = (4 + 99) * 20 = 2060nS$

Non-Pipelined System

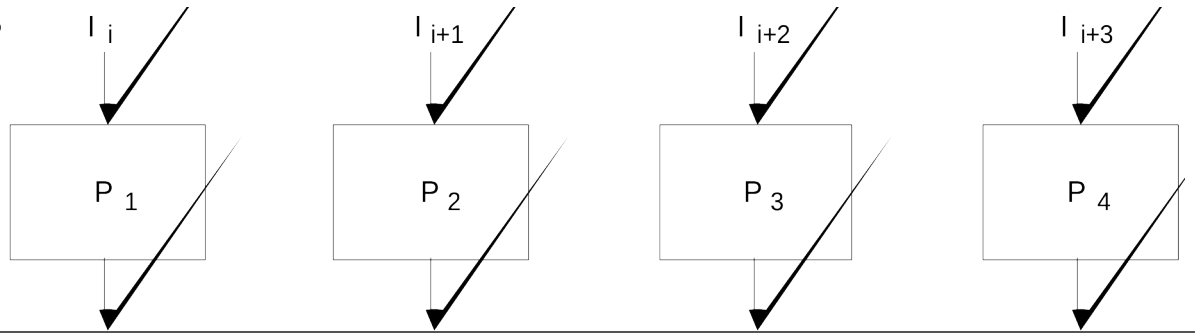
$n* tn = n*k*t_p = 100 * 80 = 8000nS$

Speedup

$S_k = 8000 / 2060 = 3.88$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Multiple Functional Units



ARITHMETIC PIPELINE

Floating-point adder

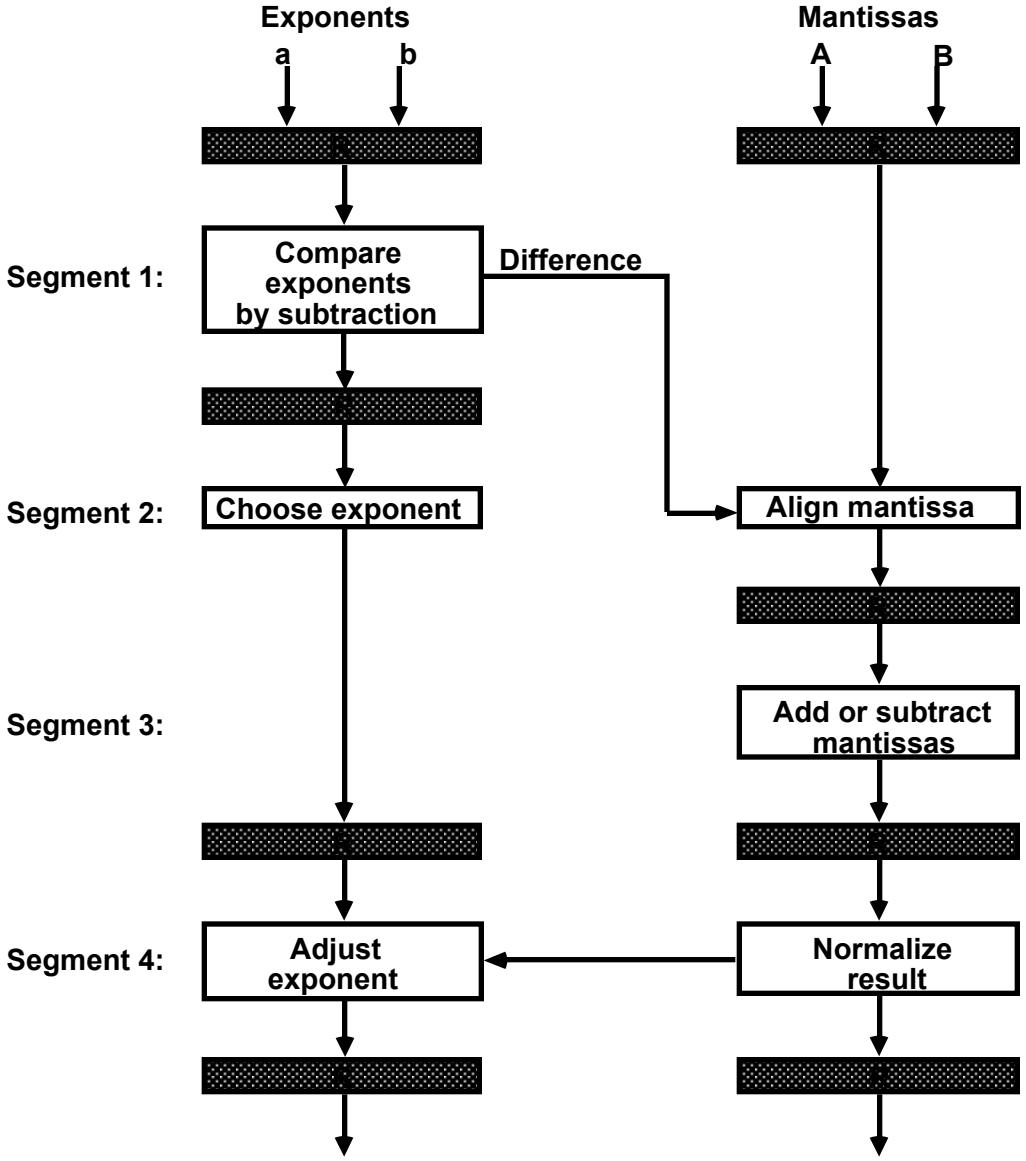
$X = A \times 2^a$
$Y = B \times 2^b$

[1] Compare the exponents
[2] Align the mantissa
[3] Add/sub the mantissa
[4] Normalize the result

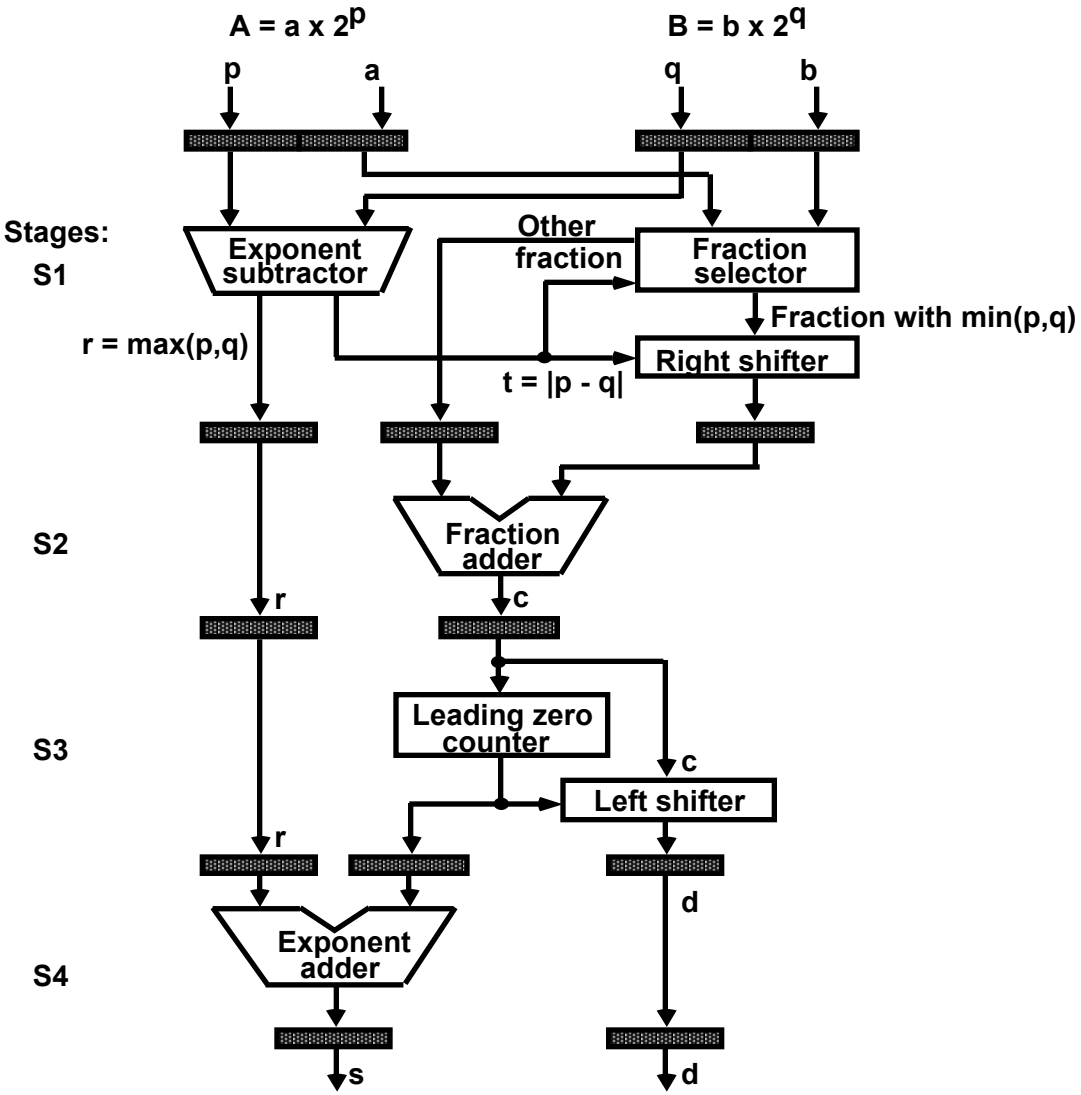
$X = 0.9504 \times 10^3$
$Y = 0.8200 \times 10^2$

Compare the exponent 3-2 = 1
Align the mantissa
$X = 0.9504 \times 10^3$
$Y = 0.0820 \times 10^3$

Add the mantissa
$Z = X + Y = 0.9504 + 0.0820 = 1.0324$
Normalize the result
$Z = 0.10324 \times 10^4$



4-STAGE FLOATING POINT ADDER



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

[1] Fetch an instruction from memory
[2] Decode the instruction
[3] Calculate the effective address of the operand
[4] Fetch the operands from memory
[5] Execute the operation
[6] Store the result in the proper place

* Some instructions skip some phases
* Effective address calculation can be done in the part of the decoding phase
* Storage of the operation result into a register is done automatically in the execution phase

INSTRUCTION CYCLE

==> 4-Stage Pipeline

[1] FI: Fetch an instruction from memory

[2] DA: Decode the instruction and calculate the effective address of the operand

[3] FO: Fetch the operand

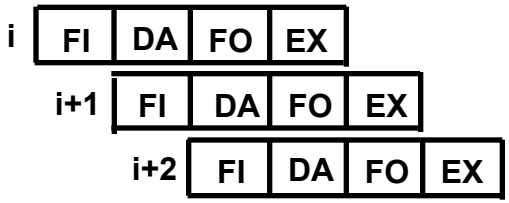
[4] EX: Execute the operation

INSTRUCTION PIPELINE

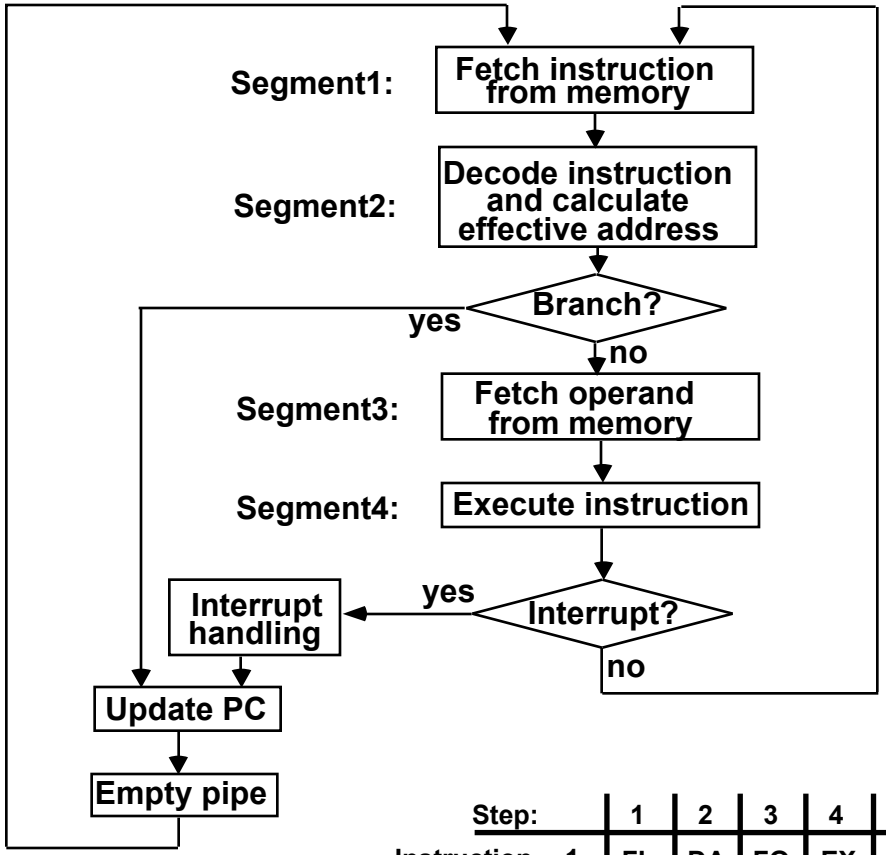
Execution of Three Instructions in a 4-Stage Pipeline

Conventional

Pipelined



INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

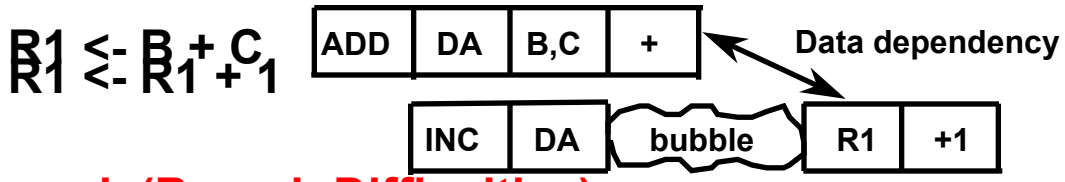
MAJOR HAZARDS IN PIPELINED EXECUTION

Structural hazards(Resource Conflicts)

Hardware Resources required by the instructions in simultaneous overlapped execution cannot be met. It is caused by access memory (or resource) by two or more segment at the same time.

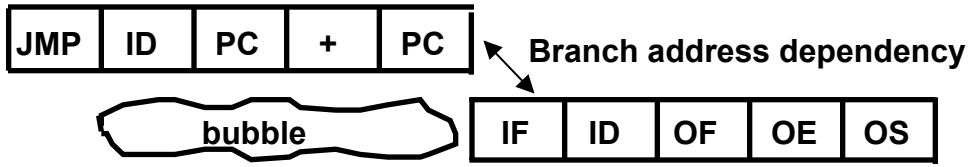
Data hazards (Data Dependency Conflicts)

An instruction scheduled to be executed in the pipeline requires the result of a previous instruction, which is not yet available. It arise when an instruction depends on the result of the previous instruction.



Control hazards(Branch Difficulties)

Branches and other instructions that change the PC make the fetch of the next instruction to be delayed.



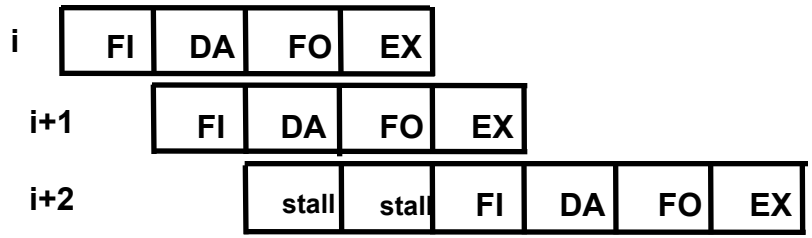
Hazards in pipelines may make it necessary to ***stall*** the pipeline ⇨ Pipeline Interlock: Detect Hazards Stall until it is cleared

STRUCTURAL HAZARDS

Structural hazards(Resource Conflicts)

Occur when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example: With one memory-port, a data and an instruction fetch cannot be initiated in the same clock



The Pipeline is stalled for a structural hazard
<- Two Loads with one port memory
-> Two-port memory will serve without stall

Most of these conflicts can be resolved by using separate instruction and data memories.

DATA HAZARDS

Data Hazards occurs when the execution of an instruction depends on the results of a previous instruction

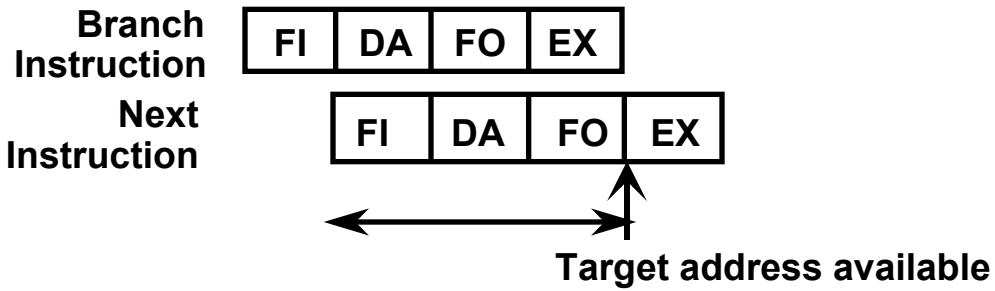
```
ADD R1, R2,
R3
SUB R4, R1,
R5
```

Data hazard can be deal with either hardware techniques or software technique.

Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.

CONTROL HAZARDS

Branch Instructions - Branch target address is not known until the branch instruction is completed



- Stall -> waste of cycle times

Dealing with Control Hazards

* **Prefetch Target Instruction**
* **Branch Target Buffer**
* **Loop Buffer**
* **Branch Prediction**
* **Delayed Branch**

CONTROL HAZARDS

Prefetch Target Instruction

- Fetch instructions in both streams, branch not taken and branch taken
- Both are saved until branch is executed. Then, select the right instruction stream and discard the wrong stream

Branch Target Buffer(BTB; Associative Memory)

- Entry: Address of previously executed branches; Target instruction and the next few instructions
- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

CONTROL HAZARDS

Loop Buffer(High Speed Register file)

- Storage of entire loop that allows to execute a loop without accessing memory

Branch Prediction

- Guessing the branch condition, and fetch an instruction stream based on the guess. Correct guess eliminates the branch penalty

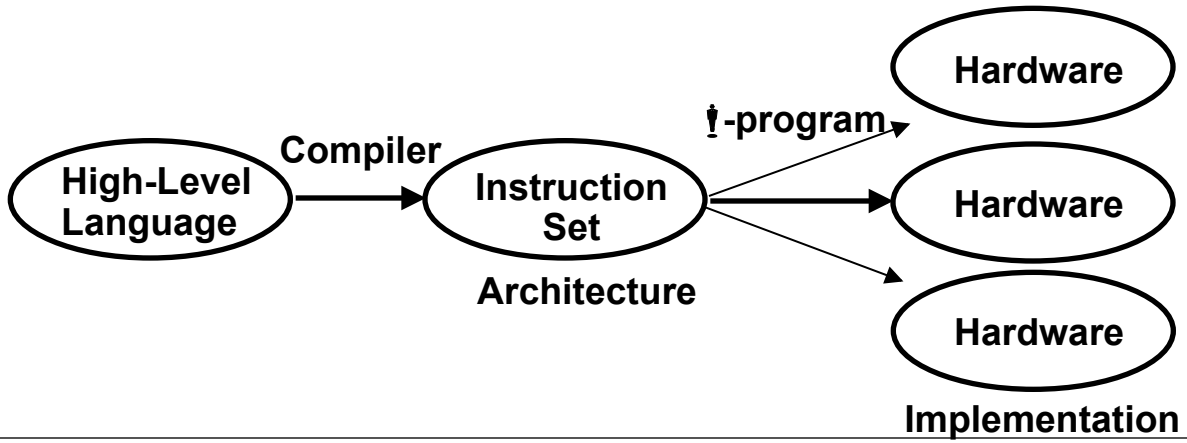
Delayed Branch

- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction

RISC/CISC: Historical Background

IBM System/360, 1964

- **The real beginning of modern computer architecture**
- **Distinction between *Architecture* and *Implementation***
- **Architecture: The abstract structure of a computer**
- **Continuing growth in semiconductor memory and micropr**
  - **A much richer and complicated instruction sets**
  - **CISC(Complex Instruction Set Computer)**



COMPLEX INSTRUCTION SET COMPUTER (CISC)

- These computers with many instructio
- One goal for CISC machines was to hav

VARIABLE LENGTH INSTRUCTIONS

- The large number of instructions
- The large number of instructions

VARIABLE LENGTH INSTRUCTIONS

- In order to manage this large numb
  - More frequently used instructions
  - Less frequently used ones were a

VARIABLE LENGTH INSTRUCTIONS

- Also, multiple operand instructions could specify
  - For example,
    - » Operand 1 could be a directly addressed regi
    - » Operand 2 could be an indirectly addressed m
    - » Operand 3 (the destination) could be an indir
- All of this led to the need to have different length i

VARIABLE LENGTH INSTRUCTIONS

- For example, an instruction that only specifies register
  - One byte to specify the instruction and addressing m
  - One byte to specify the source and destination regis
- An instruction that specifies memory addresses for op
  - One byte to specify the instruction and addressing m
  - Two bytes to specify each memory address
    - Maybe more if there's a large amount of memory.

VARIABLE LENGTH INSTRUCTIONS

- Variable length instructions g
- The circuitry to recognize the various i

COMPLEX INSTRUCTION SET COMPUTER

- Another property of CISC computers is that the
  - For example,

    ADD L1, L2, L3

    that takes the contents of *M*[*L1*] adds it to th
- An instruction like this takes three memory acc
- That makes for a potentially very long instructi

COMPLEX INSTRUCTION SET COMPUTER

- The problems
  - The complexity of the design may slow down the processor,
  - –The complexity of the design may res
  - –Many of the instructions and address

SUMMARY: CRITICISMS ON CISC

High Performance General Purpose Instructions

- Complex Instruction
→ Format, Length, Addressing Modes
→ Complicated instruction cycle control due to the complex decoding HW and decoding process

- Multiple memory cycle instructions
→ Operations on memory data
→ Multiple memory accesses/instruction

SUMMARY: CRITICISMS ON CISC

- Microprogrammed control is necessity
  → Microprogram control storage takes substantial portion of CPU chip area
  →Semantic Gap is large between machine instruction and microinstruction

- General purpose instruction set includes all the features required by individually different applications
  → When any one application is running, all the features required by the other applications are extra burden to the application

REDUCED INSTRUCTION SET COMPUTERS

- **In the late ‘70s and early ‘80s there was**
- **Reduced Instruction Set Computers (R**
- **The idea behind RISC processors is to**

REDUCED INSTRUCTION SET COMPUTERS

- RISC processors often feature:
  - Few instructions
  - Few addressing modes
  - Only load and store instructions access mem
  - All other operations are done using on-proce
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogr

REDUCED INSTRUCTION SET COMPUTERS

- Since all but the load and store instructions
- By having all instructions the same length,
- The fetch and decode stages are simple.
- The instruction and address formats are de

REDUCED INSTRUCTION SET COMPUTERS

- **Unlike the variable length CISC instructions,**
- **The control logic of a RISC processor is des**
- **The control logic is simple because of the sr**
- **The control logic is hardwired, rather than m**

REGISTERS in RISC

- By simplifying the instructions and ado
- This extra capacity is used to
  - Pipeline instruction execution to spe
  - Add a large number of registers to th

PIPELINING in RISC

- A very important feature of many RISC proce
- This may seem nonsensical, since it takes at
- It is however possible, because of a techniqu
- Pipelining is the use of the processor to wor

PIPELINING

- For instance, at one time, a pipelined processor m
  - Executing instruction i_t
  - Decoding instruction i_{t+1}
  - Fetching instruction i_{t+2} from memory

- So, if we're running three instructions at once, and
- Pipelined execution is an integral part of all moder

REGISTERS

- By having a large number of general purpose r
- This results in a significant speed up, since me
- Register accesses are fast, since they just use
- To go off-processor to memory requires using

REGISTERS

- It may take many clock cycles to read or write to
  - The memory bus hardware is usually slower th
  - There may even be competition for access to th
- So, for this reason alone, a RISC processor may h
  - for its instructions, and
  - occasionally to load or store a memory value

Properties OF RISC

- **RISC Properties**
  - Relatively few instructions
  - Relatively few addressing modes
  - Memory access limited to load and store instructions
  - All operations done within the registers of the CPU
  - Fixed-length, easily decoded instruction format
  - Single-cycle instruction format
  - Hardwired rather than microprogrammed control

Advantages OF RISC

- **Advantages of RISC**
  - VLSI Realization
  - Computing Speed
  - Design Costs and Reliability
    - High Level Language Support

VECTOR PROCESSING

Vector Processing Applications

- Problems that can be efficiently formulated in terms of vectors
  - Long-range weather forecasting
  - Petroleum explorations
  - Seismic data analysis
  - Medical diagnosis
  - Aerodynamics and space flight simulations
  - Artificial intelligence and expert systems
  - Mapping the human genome
  - Image processing

Vector Processor (computer)

Ability to process vectors, and related data structures such as matrices and multi-dimensional arrays, much faster than conventional computers.

Vector Processors may also be pipelined

VECTOR PROGRAMMING

```
      DO  20  I = 1, 100
20    C(I) = B(I) + A(I)
```

Conventional computer

```
      Initialize I = 0
20    Read A(I)
      Read B(I)
      Store C(I) = A(I) + B(I)
      Increment I = i + 1
      If I ☒ 100 goto 20
```

Vector computer

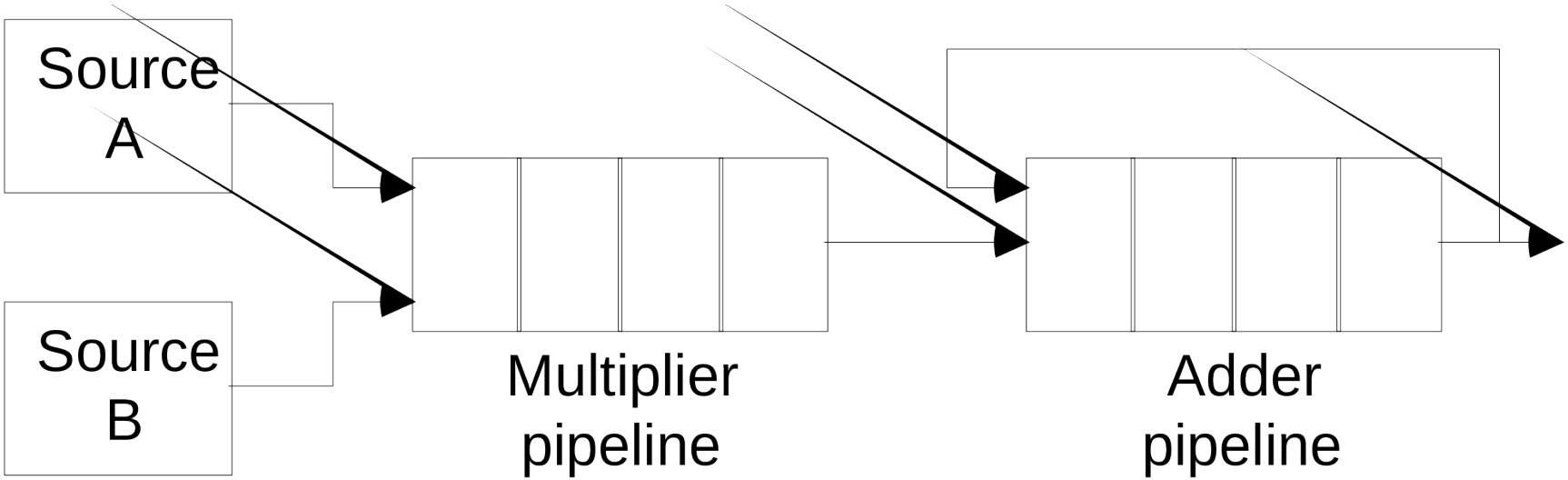
```
C(1:100) = A(1:100) + B(1:100)
```

VECTOR INSTRUCTION FORMAT

Vector Instruction Format

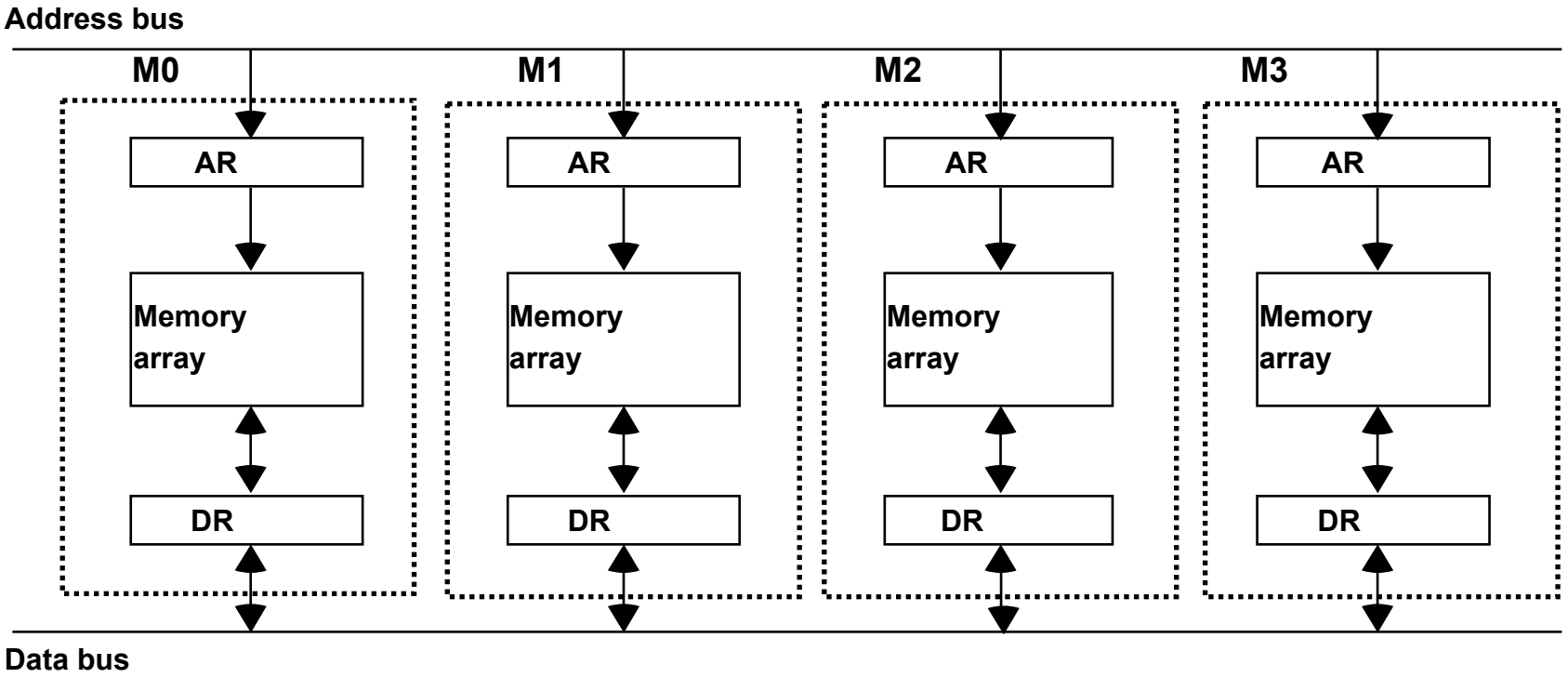
Operation code	Base address source 1	Base address source 2	Base address destination	Vector length

Pipeline for Inner Product



MULTIPLE MEMORY MODULE AND INTERLEAVING

Multiple Module Memory



Address Interleaving

Different sets of addresses are assigned to different memory modules

Supercomputers

- A commercial computer with vector
- Supercomputers are very powerful.

Supercomputers

- To speed up the operations, the cor
- Supercomputers also use special te

Supercomputers

- The instruction set of supercomputer co
- A supercomputer is a computer system

Supercomputers

- The measure used to evaluate comp
- The term *megaflops* is used to denc

Supercomputers

- Typical supercomputer has a basic
- If the processor can calculate a floa

Supercomputers

- The first supercomputer developed in 1
  - It uses vector processing with 12 distinct
  - Each functional unit is segmented to proc
  - A floating-point operation can be performe
  - This gives a rate of 80 megaflops.

Supercomputers

- It has a memory capacity of 4 millions 64-bit wor
- The memory is divided into 16 banks, with each
- This means that when all 16 banks are accessed
- Later version are Cray X-MP, Cray Y-MP, Cray-2

Another supercomputers are Fujitsu VP-200

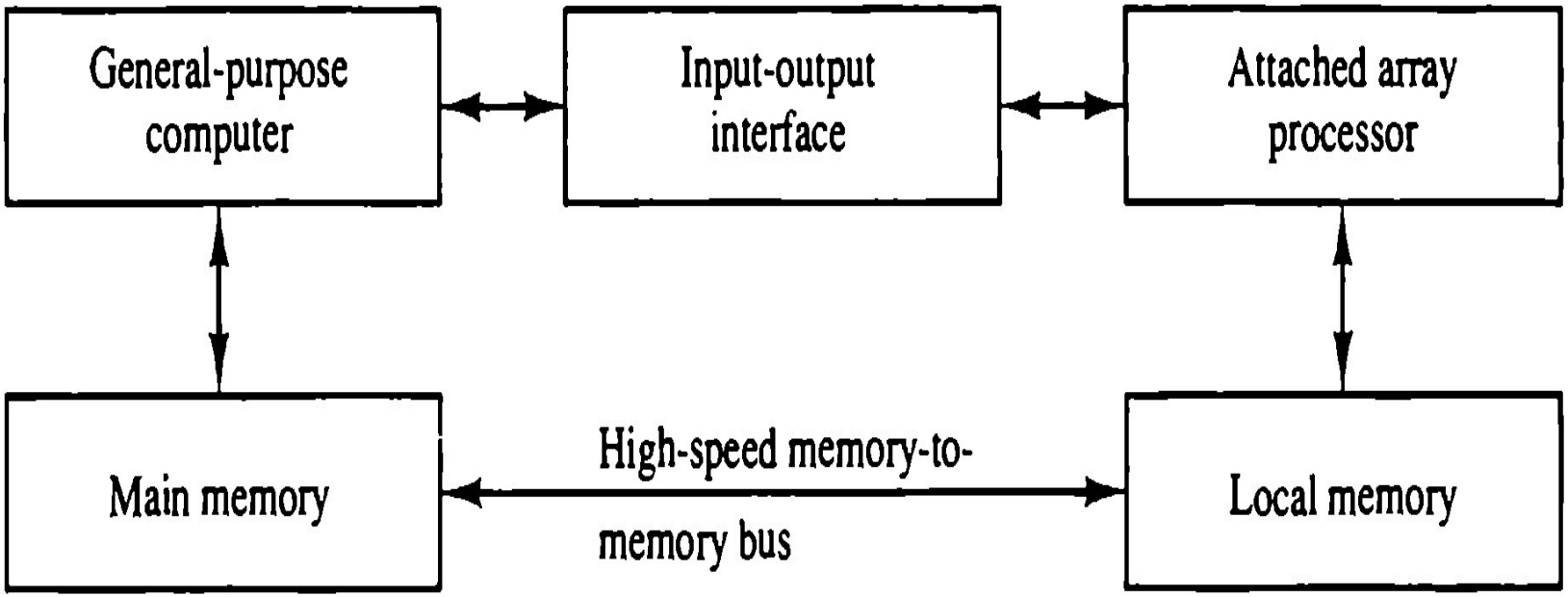
Array Processors

- An array processor is a processor
- The term is used to refer to two dif
  - An attached array processor
  - An SIMD array processor

Array Processors

- **An attached array processor :**

Figure 9-14 Attached array processor with host computer.



Array Processors

- **SIMD array processor :** PEs (processing elements)

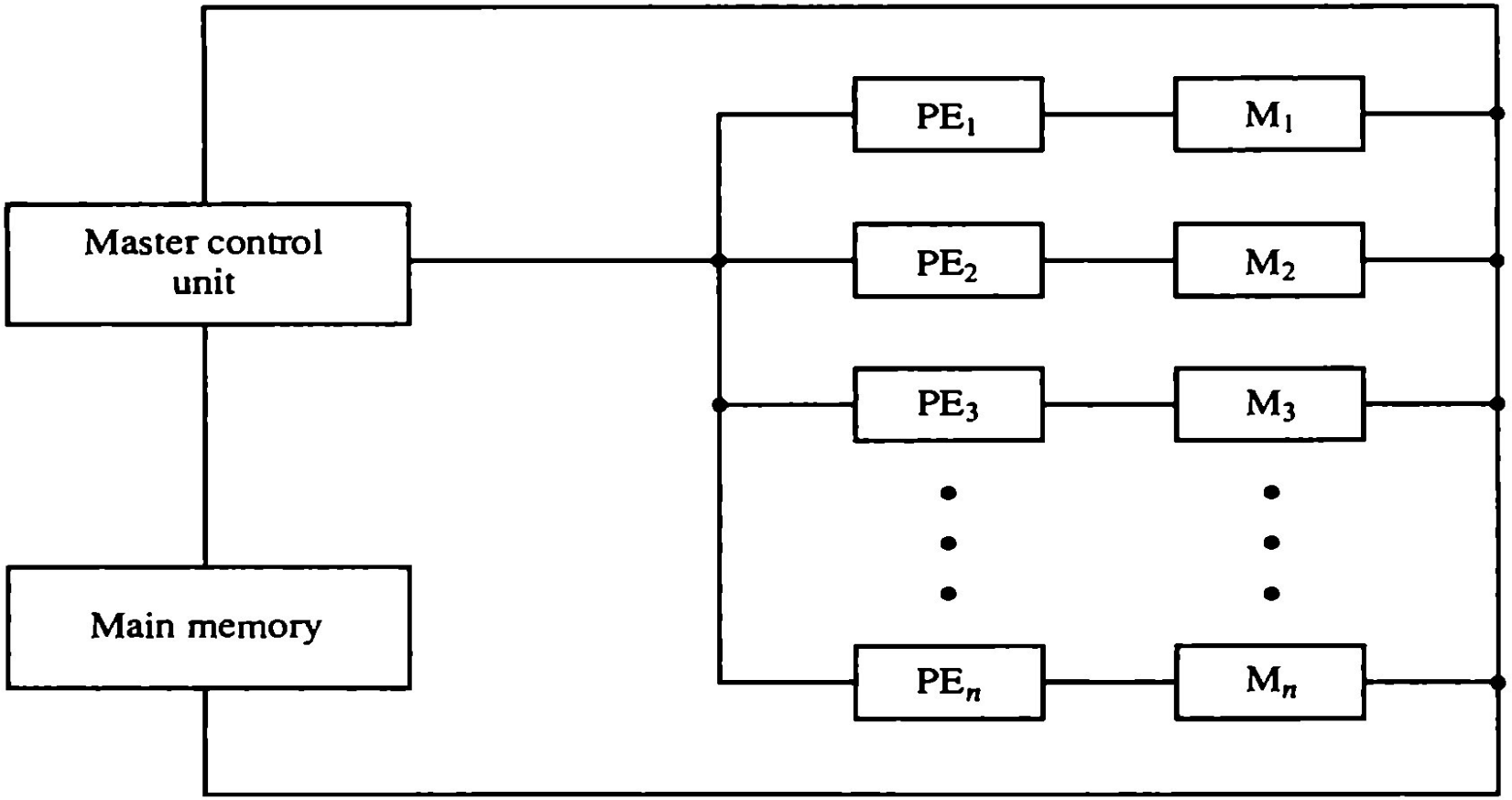


Figure 9-15 SIMD array processor organization.

Computer Arithmetic

IN THIS CHAPTER

Computer Arithmetic

In this chapter we develop the various arithmetic algorithms and show the procedure for implementing them with digital hardware. We consider addition, subtraction, multiplication, and division for the following types of data:

1. Fixed-point binary data in signed-magnitude representation
2. Fixed-point binary data in signed-2's complement representation
3. Floating-point binary data
4. Binary-coded decimal (BCD) data

Addition and Subtraction of Signed-Magnitude Numbers

TABLE 10-1 Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Hardware Implementation

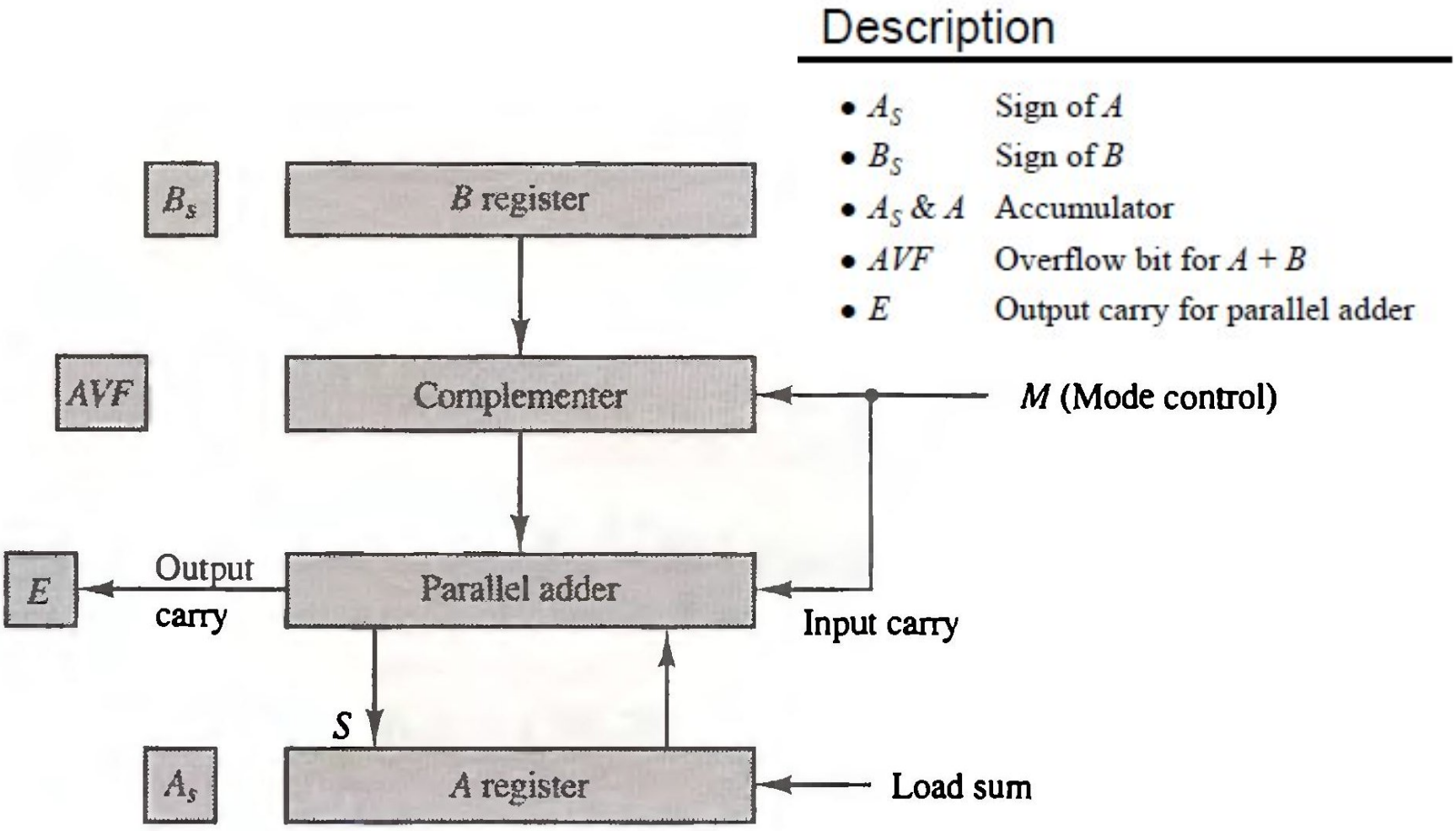


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

Flow Chart for add and subtract Operation

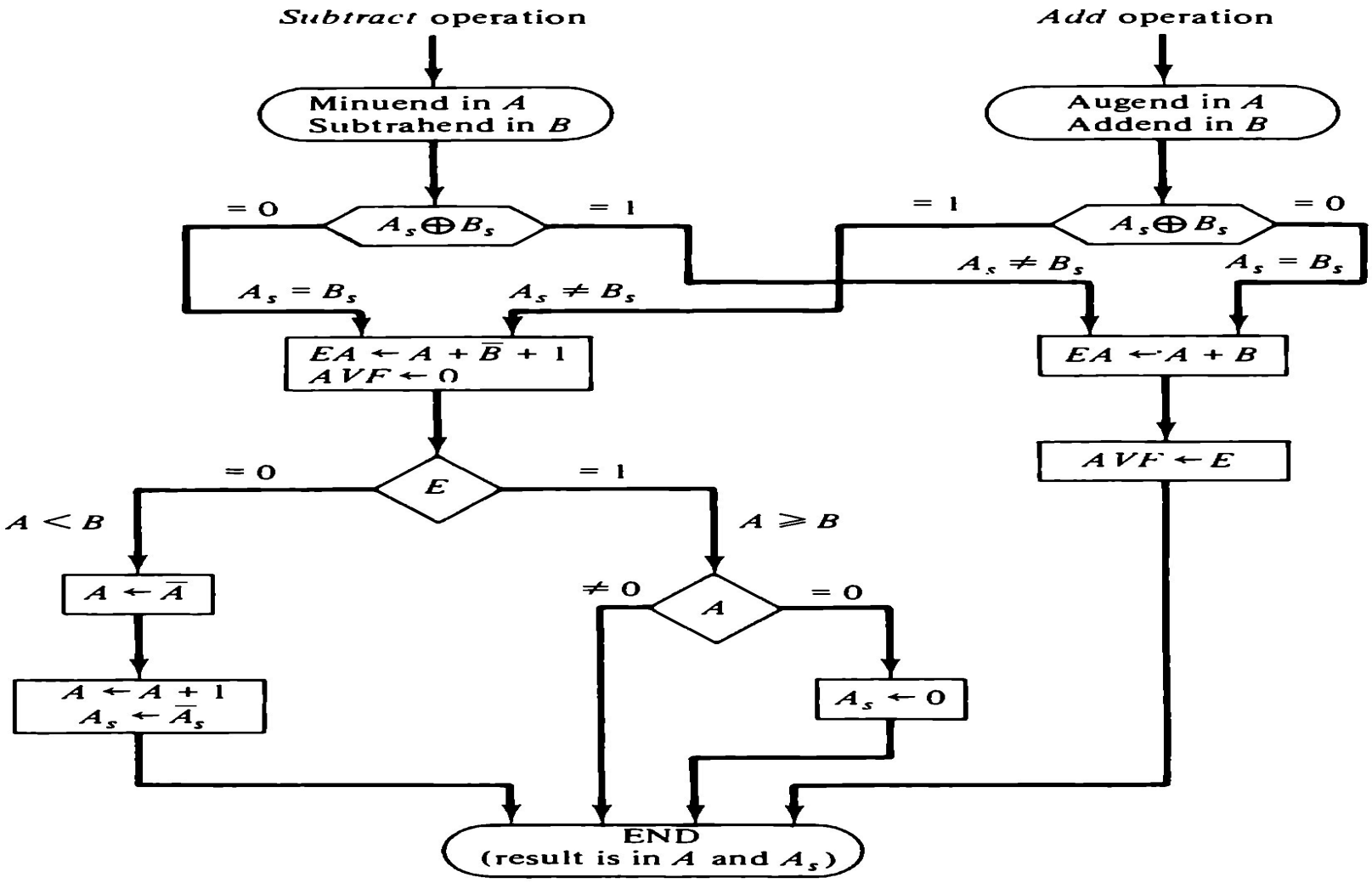
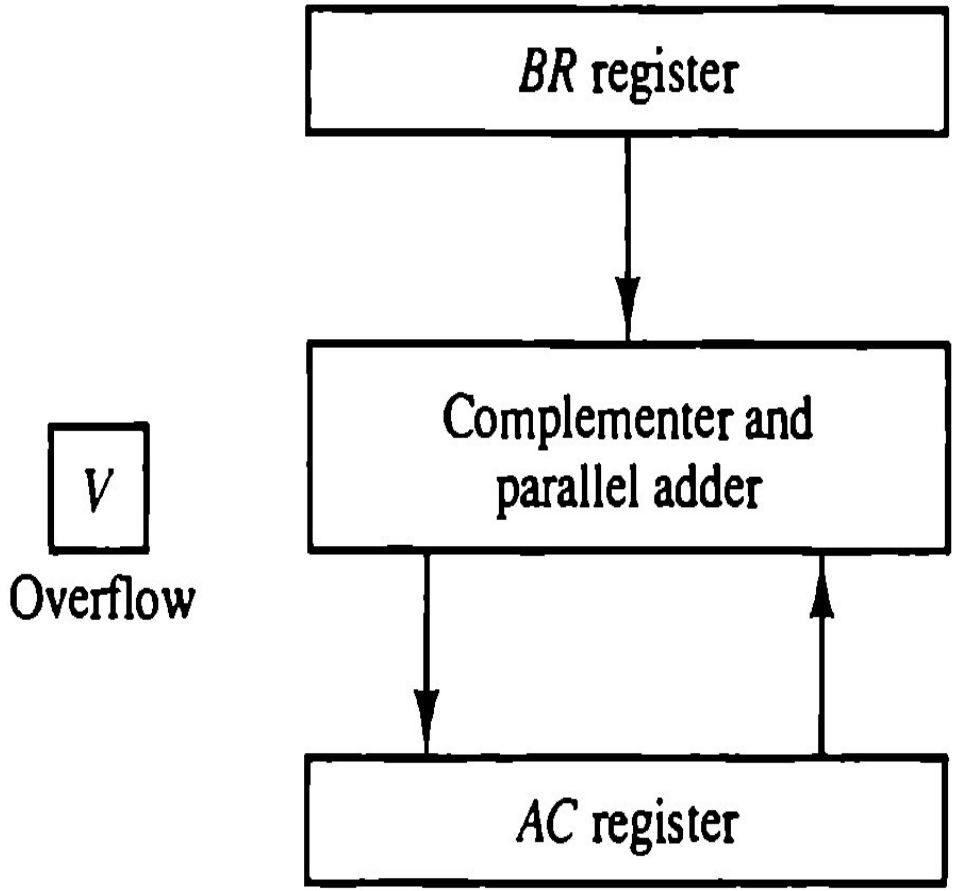


Figure 10-2 Flowchart for add and subtract operations.

Hardware for signed-2's complement addition and subtraction

Figure 10-3 Hardware for signed-2's complement addition and subtraction.



Algorithm for signed-2's complement addition and subtraction

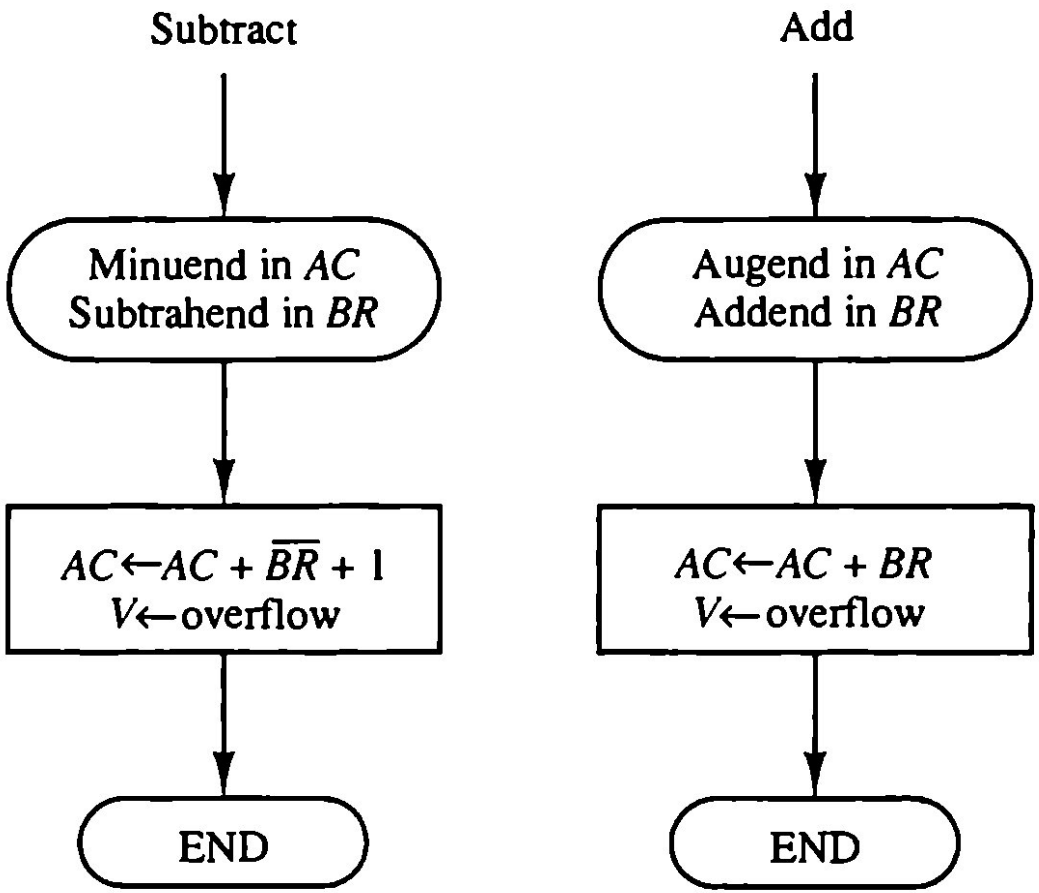


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

Multiplication Algorithm for signed magnitude

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example.

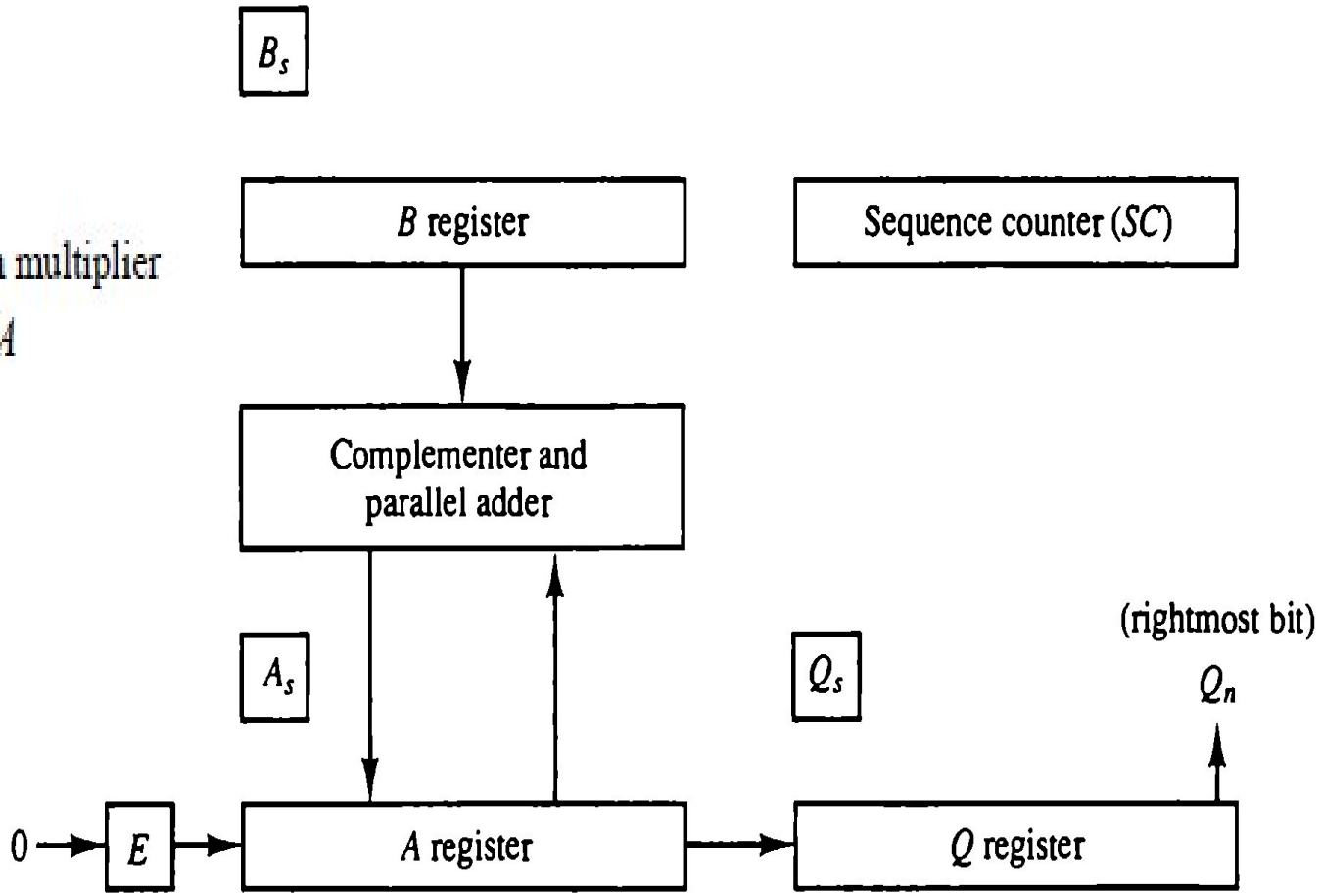
```
 23      10111    Multiplicand
 19    x 10011    Multiplier
       ------
        10111
       10111
      00000   +
     00000
    10111
   ---------
437 110110101    Product
```

Hardware Implementation of Multiplication Algorithm for signed magnitude

Description

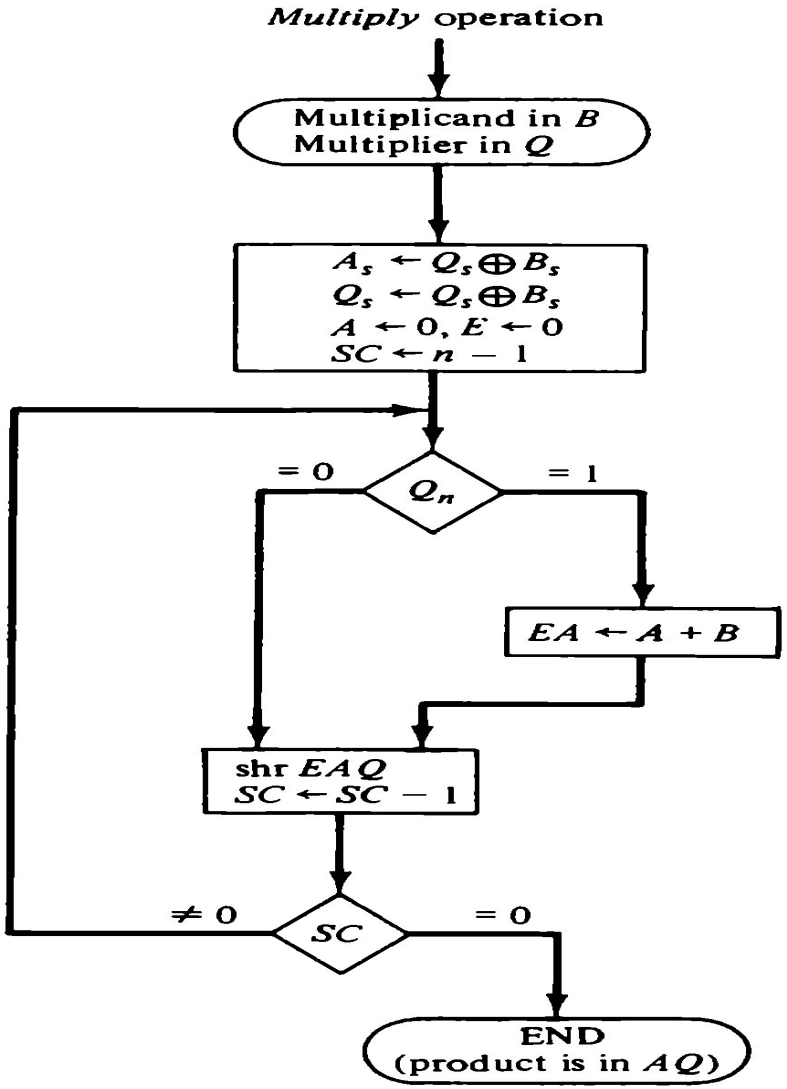
- Q multiplier
- B multiplicand
- A 0
- SC number of bits in multiplier
- E overflow bit for A
- Do SC times
  - If low-order bit of Q is 1
    - $A \leftarrow A + B$
  - Shift right EAQ
- Product is in AQ

Figure 10-5 Hardware for multiply operation.



Flow Chart of Multiplication Algorithm for signed magnitude

Figure 10-6 Flowchart for multiply operation.



Example of Multiplication Algorithm for signed magnitude

TABLE 10-2 Numerical Example for Binary Multiplier

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Booth Multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001110 (+14) has a string of 1's from 2^3 to 2^1 ($k = 3, m = 1$). The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$. Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^4 - M \times 2^1$. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

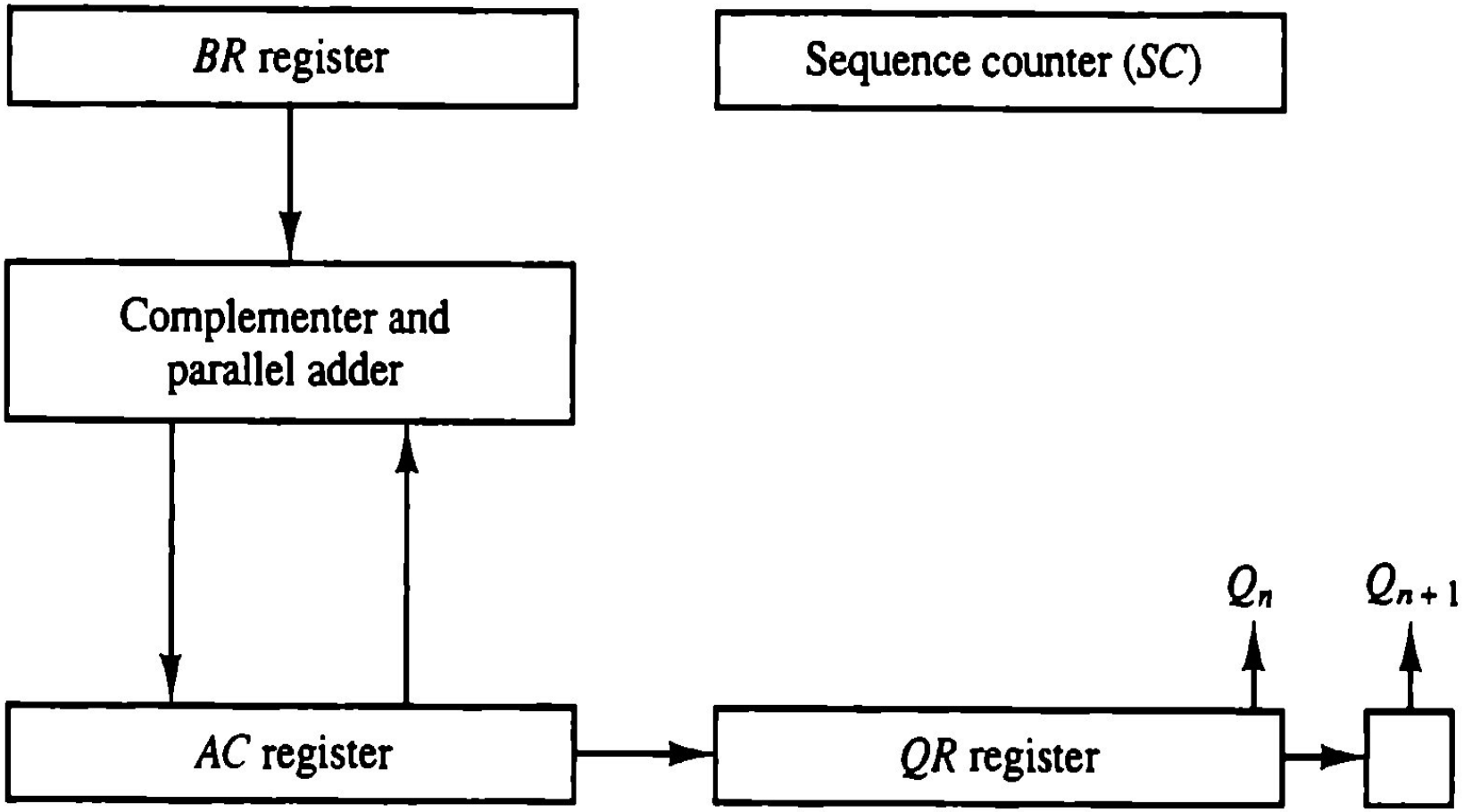
Booth Multiplication Algorithm

As in all multiplication schemes, Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Booth Multiplication Algorithm

Figure 10-7 Hardware for Booth algorithm.



Booth Multiplication Algorithm

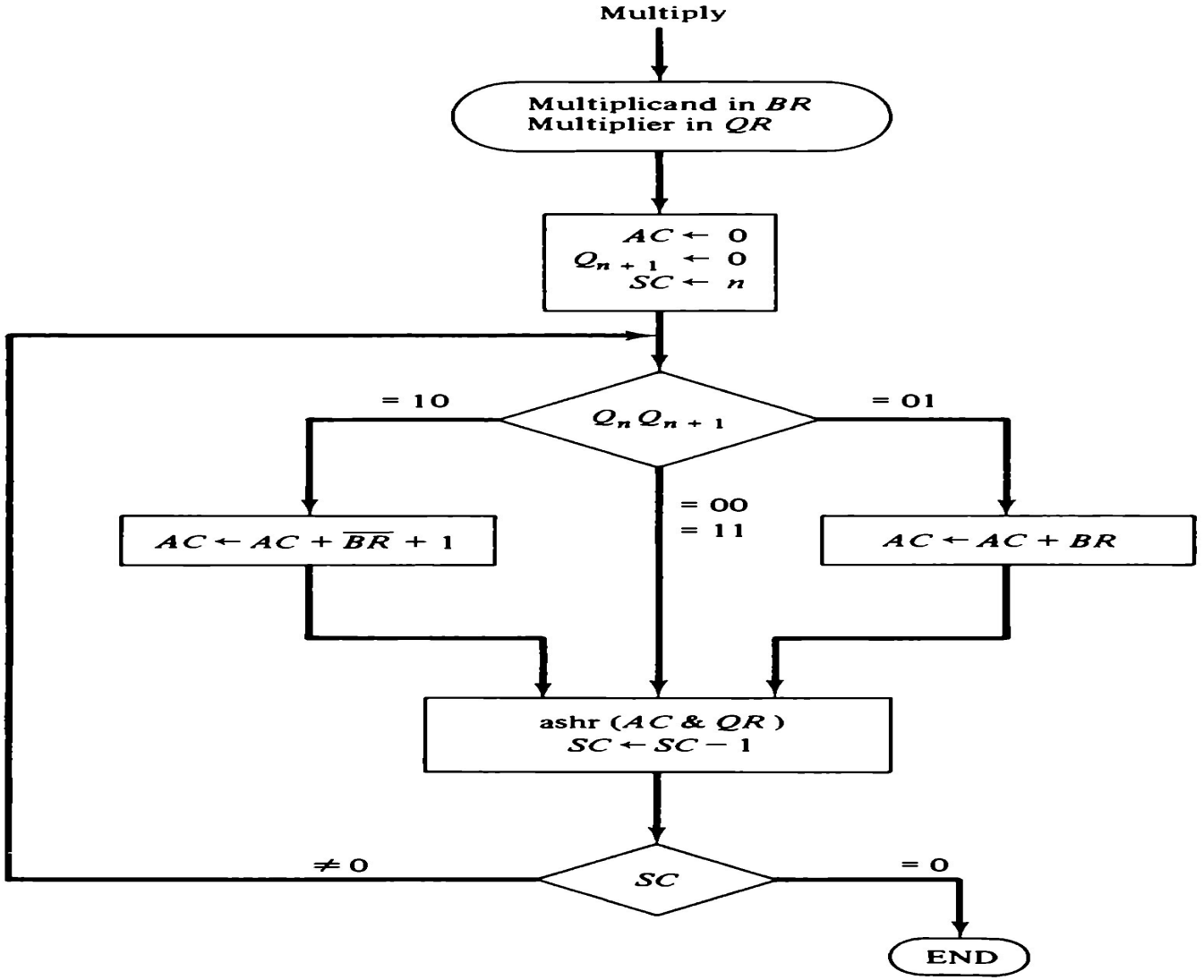


Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

Booth Multiplication Algorithm

TABLE 10-3 Example of Multiplication with Booth Algorithm

$Q_n\ Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

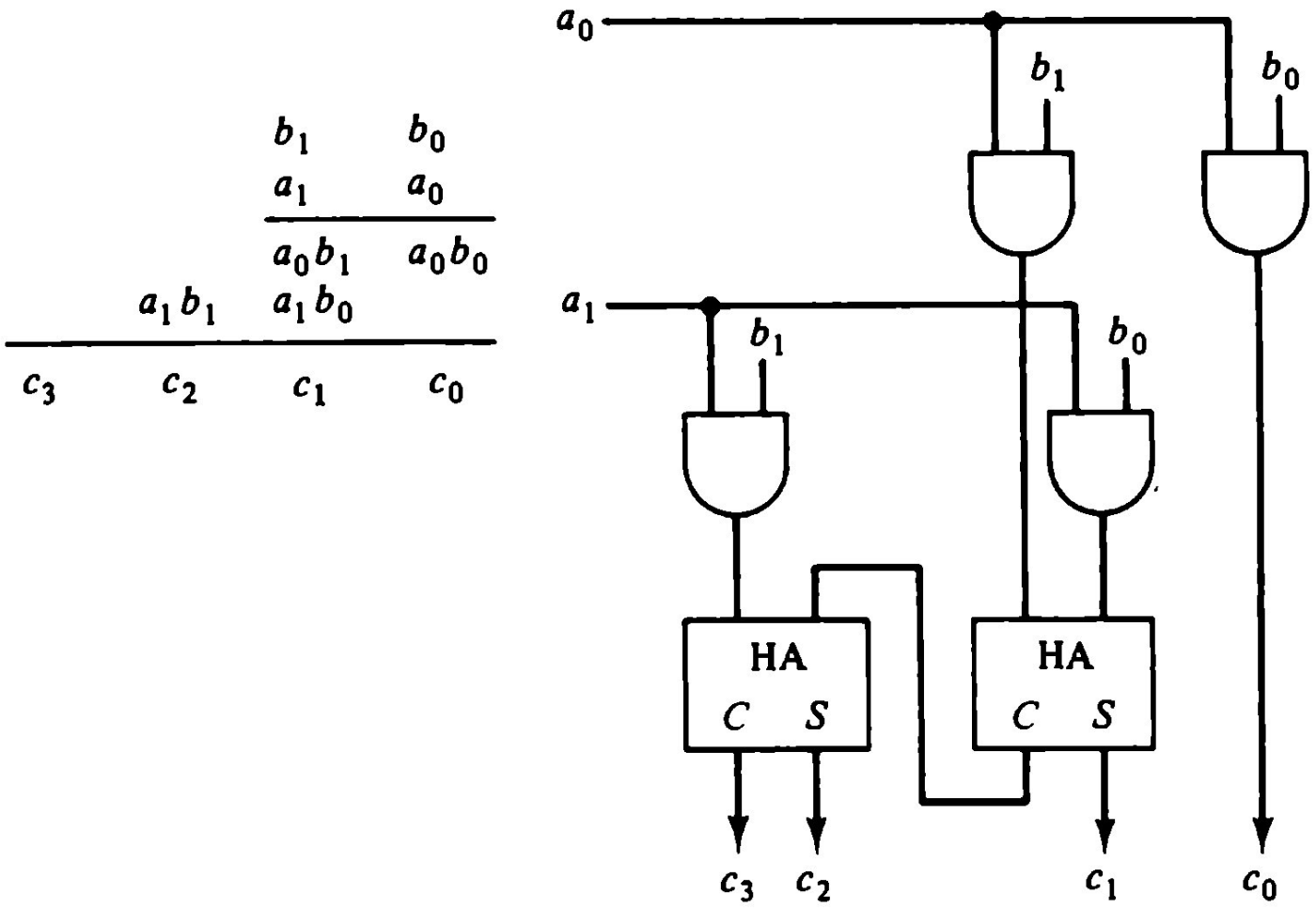
Array Multiplier

Array Multiplier

- Combination circuit
- Product generated in one microoperation
- Requires large number of gates
- Became feasible after integrated circuits developed
- Needed for j multiplier and k multiplicand bits
  - $j \times k$ AND gates
  - $j - 1$ k-bit adders to produce product of $j + k$ bits

Array Multiplier

Figure 10-9 2-bit by 2-bit array multiplier.



4 Bit by 3Bit Array Multiplier

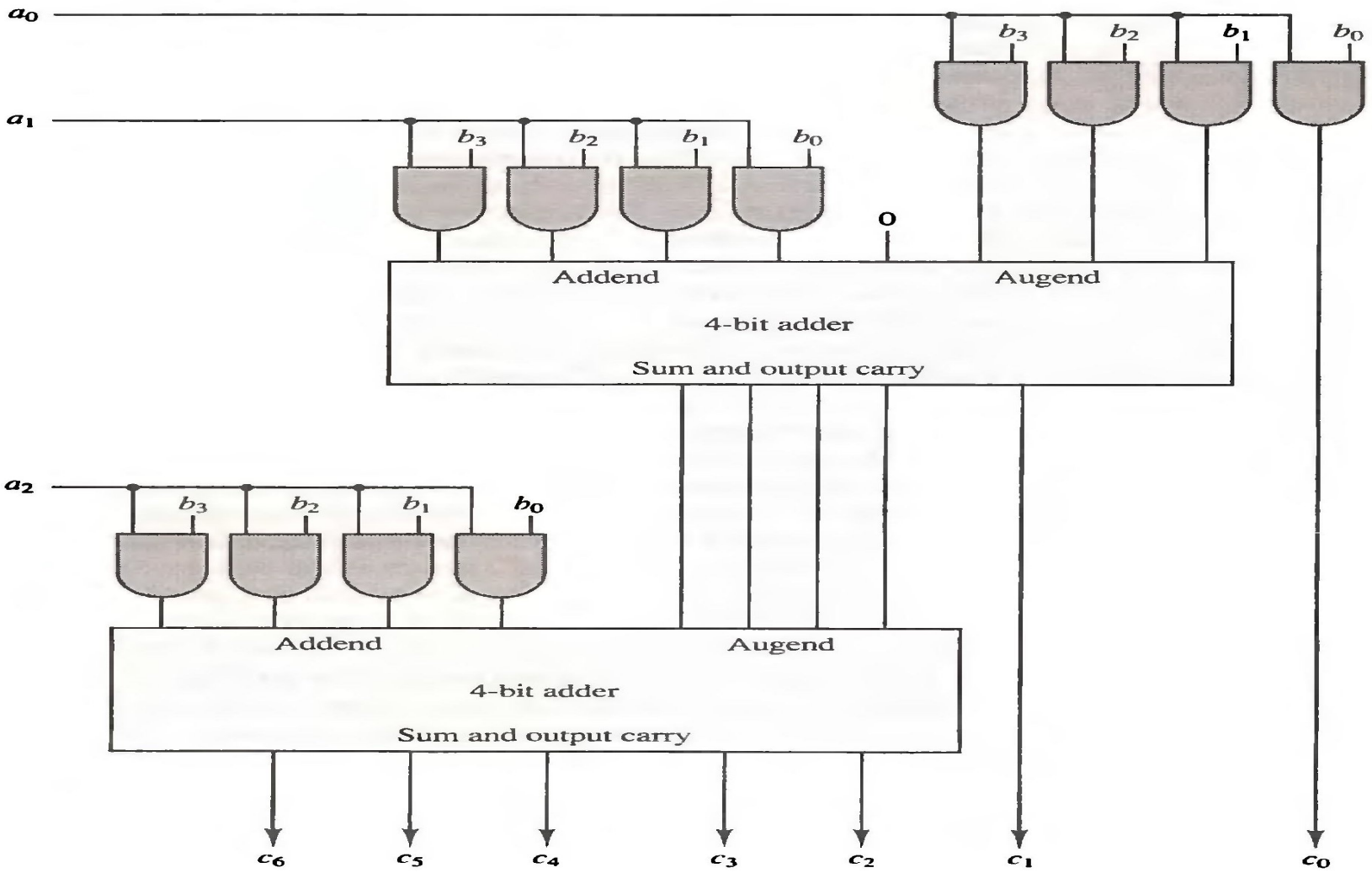


Figure 10-10 4-bit by 3-bit array multiplier.

Division of Fixed-Point Signed magnitude Numbers

Figure 10-11 Example of binary division.

Divisor:	11010	Quotient = Q
B = 10001	0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Division of Fixed-Point Signed magnitude Numbers

Divisor $B = 10001$, $\overline{B} + 1 = 01111$ Dividend in A and Q

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 10-12 Example of binary division with digital hardware.

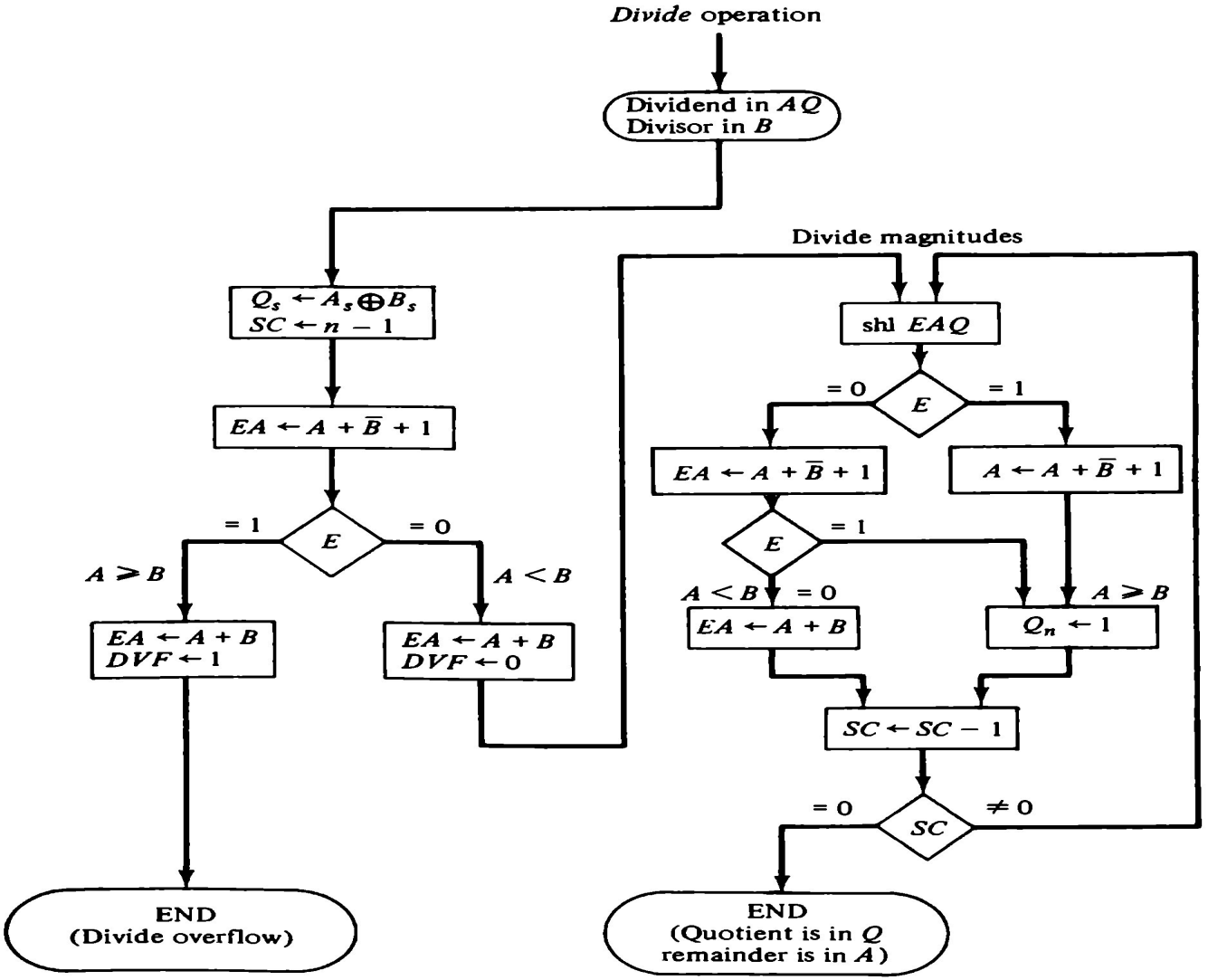
Division of Fixed-Point Signed magnitude Numbers

Divide Overflow

The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Fig. 10-11 we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor.

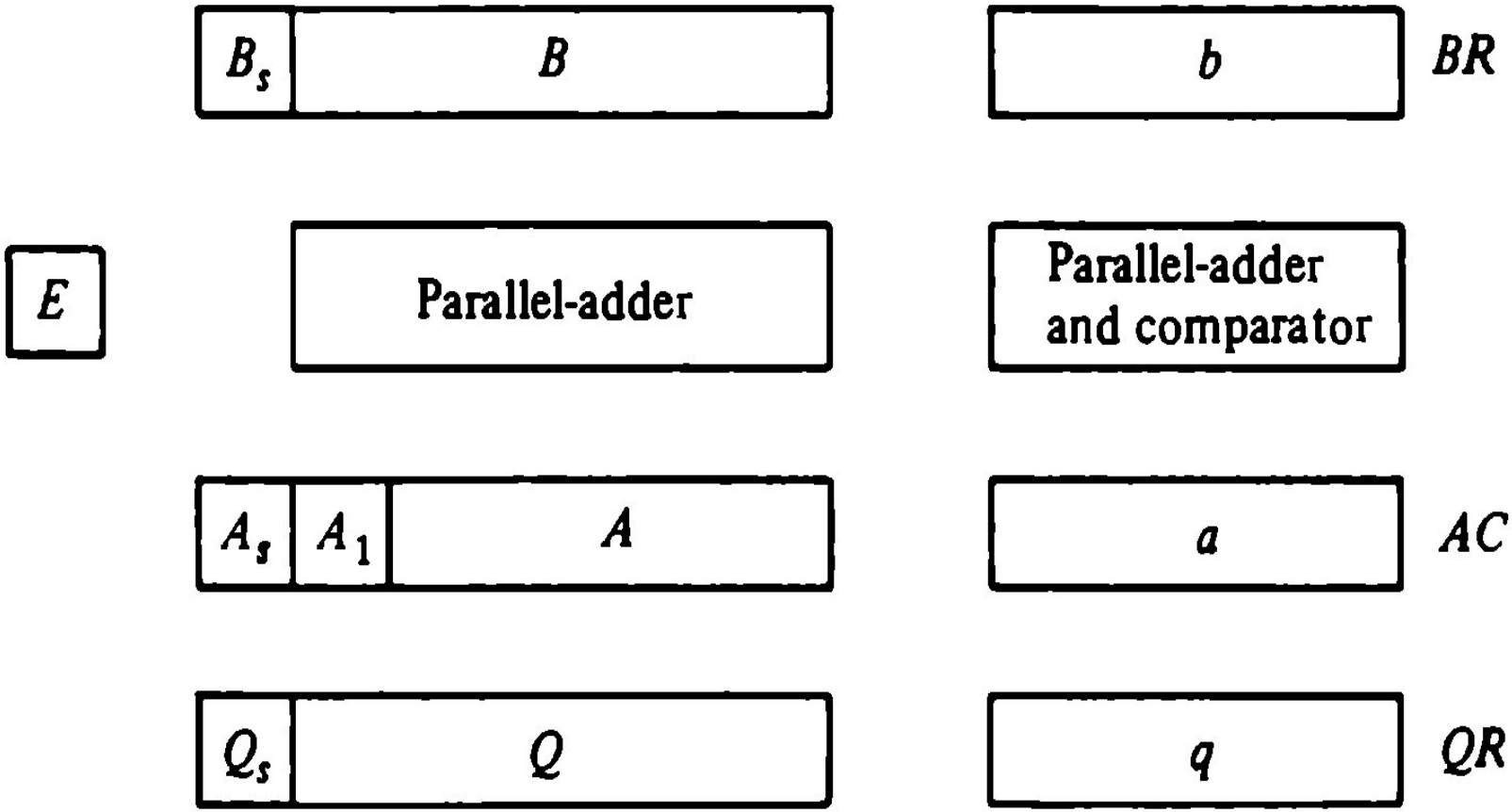
Division of Fixed-Point Signed magnitude Numbers

Figure 10-13 Flowchart for divide operation.



Floating Point Arithmetic Operation

Figure 10-14 Registers for floating-point arithmetic operations.



Floating Point Arithmetic Operation

Addition and Subtraction

During addition or subtraction, the two floating-point operands are in *AC* and *BR*. The sum or difference is formed in the *AC*. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

Floating Point Arithmetic Operation

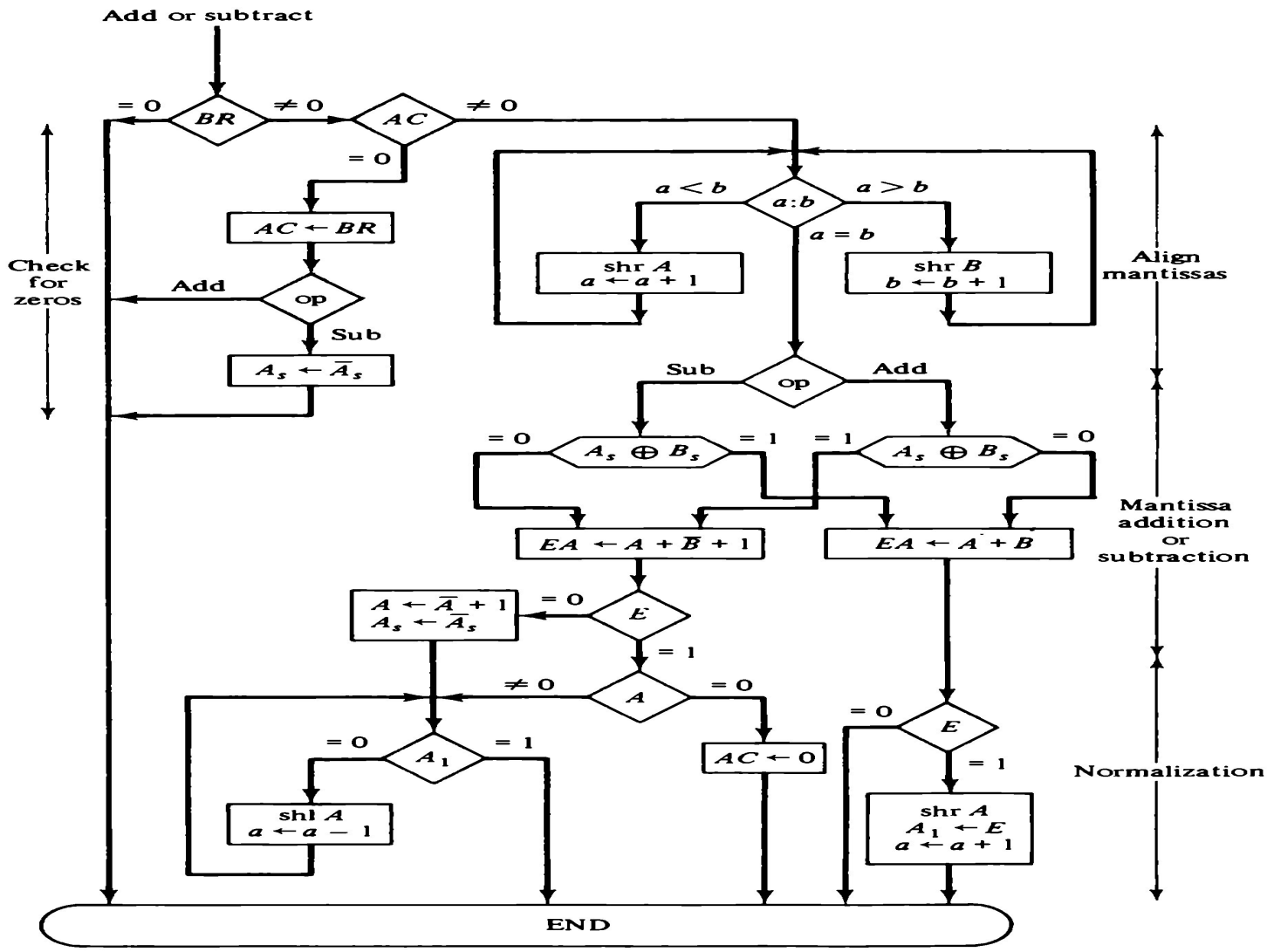


Figure 10-15 Addition and subtraction of floating-point numbers.

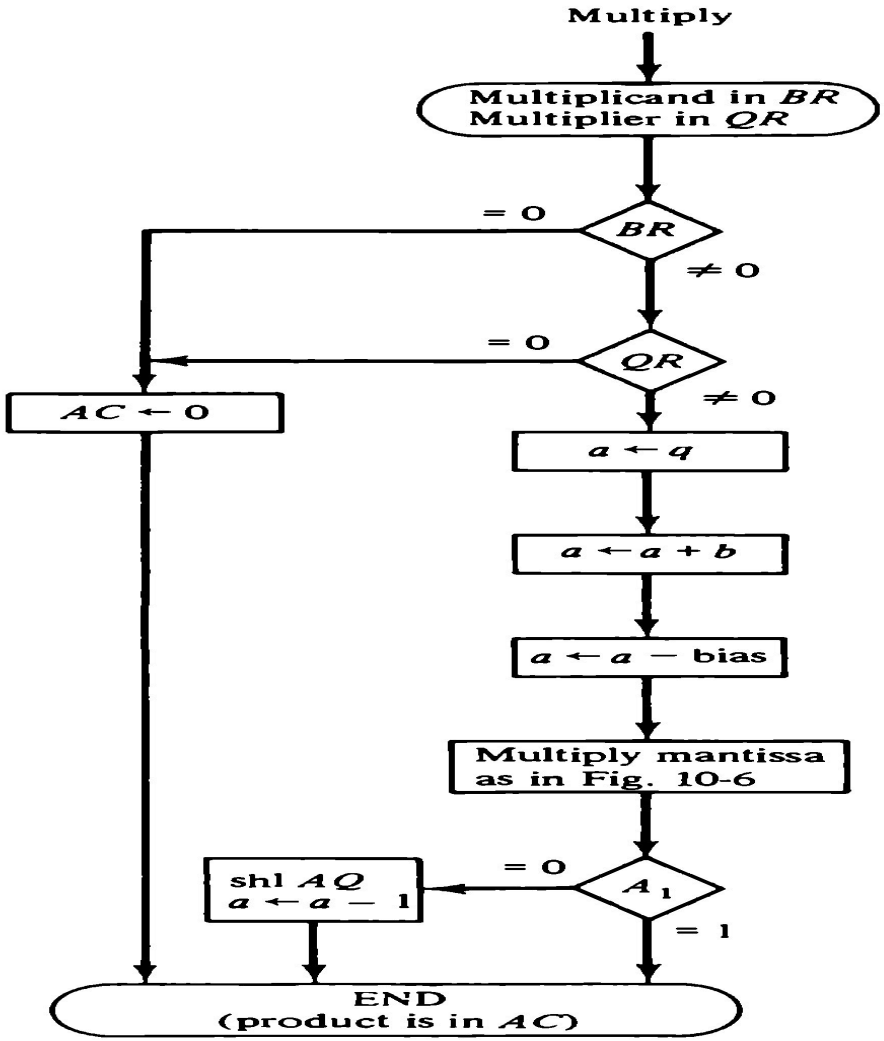
Floating Point Arithmetic Operation

The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.

Floating Point Arithmetic Operation

Figure 10-16 Multiplication of floating-point numbers.



Floating Point Arithmetic Operation

The division algorithm can be subdivided into five parts:

1. Check for zeros.
2. Initialize registers and evaluate the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

Floating Point Arithmetic Operation

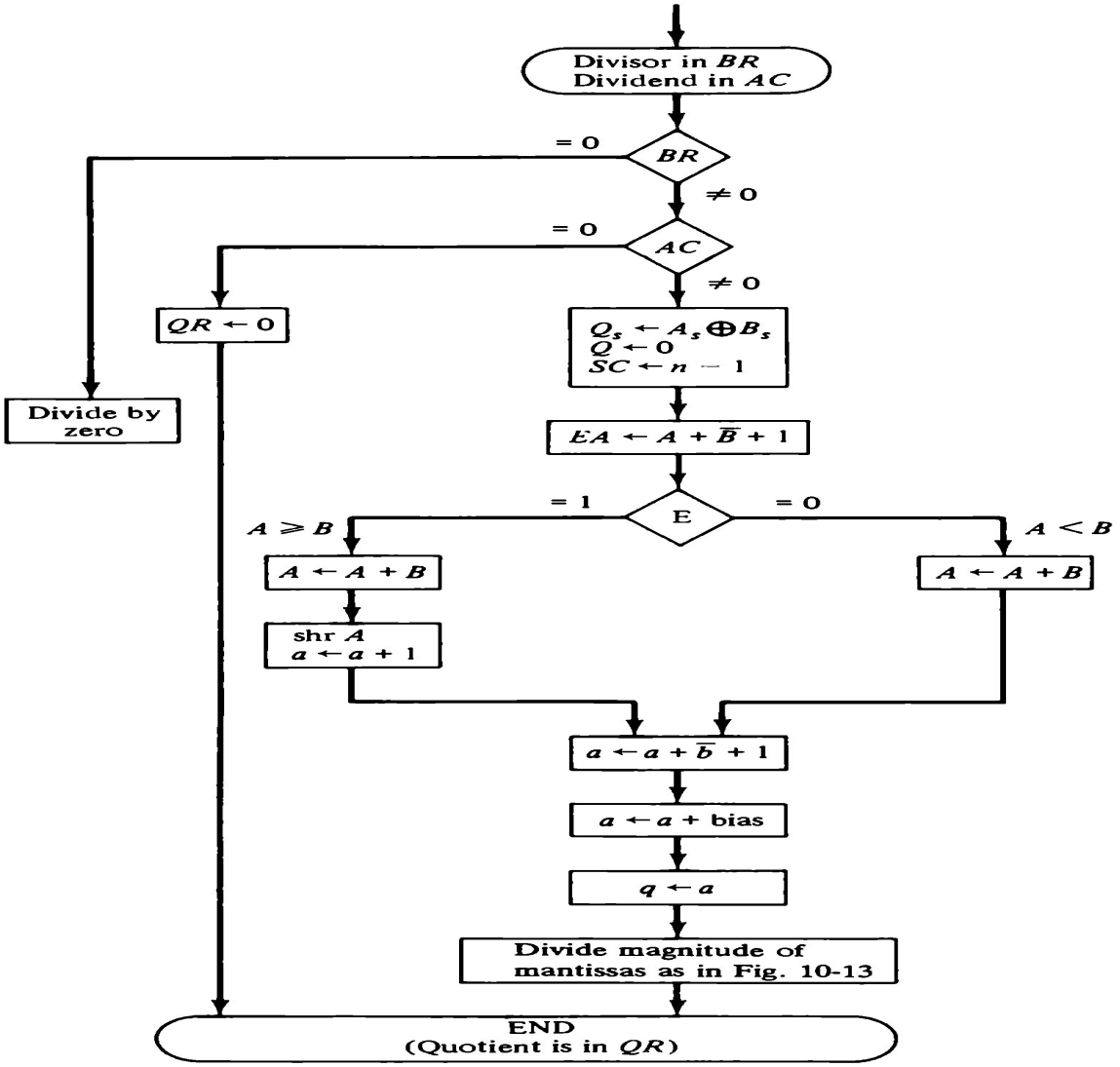


Figure 10-17 Division of floating-point numbers.

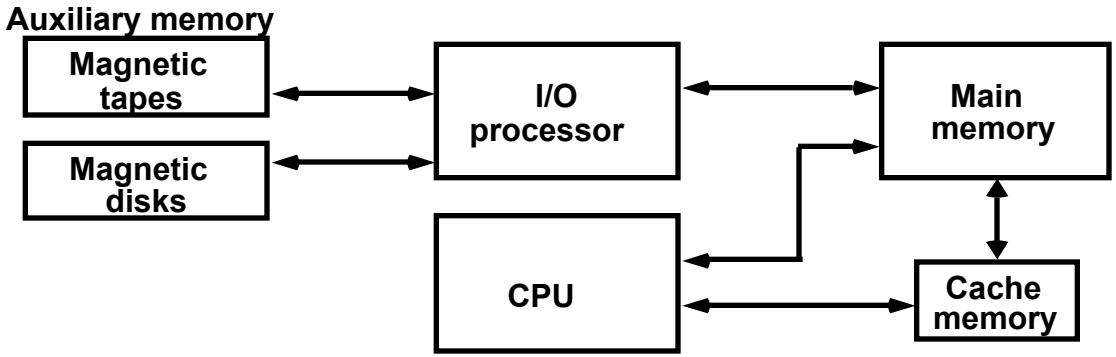
MEMORY ORGANIZATION

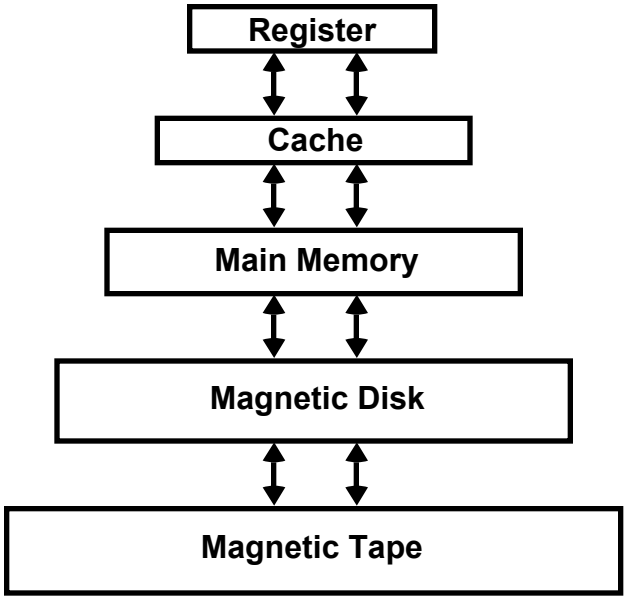
- Memory Hierarchy
- Main Memory
- Auxiliary Memory
- Associative Memory
- Cache Memory
- Virtual Memory
- Memory Management Hardware

calab.kaist.ac.kr/~hyoon/courses/cs311/cs311_2006/Ch11.ppt

MEMORY HIERARCHY

Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system

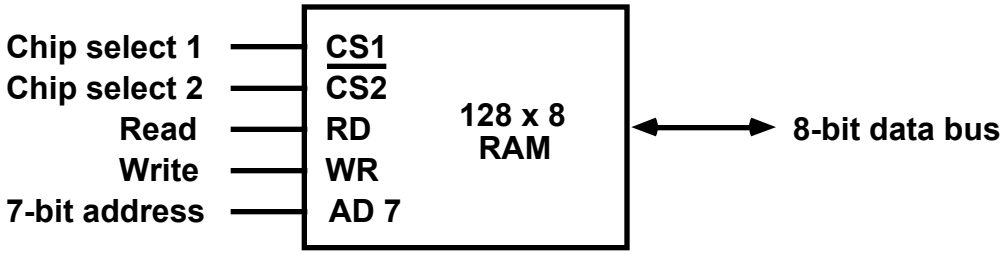




MAIN MEMORY

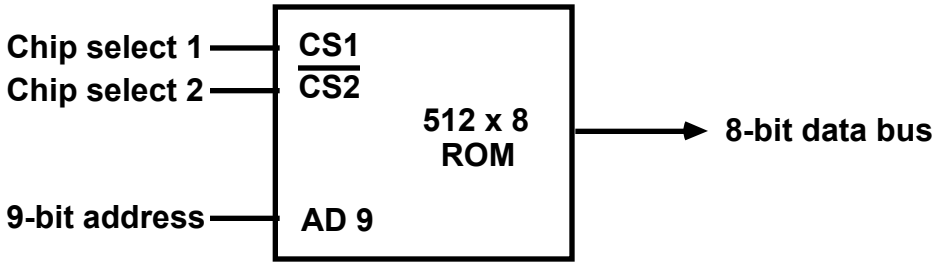
RAM and ROM Chips

Typical RAM chip



CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedence
0	1	x	x	Inhibit	High-impedence
1	0	0	0	Inhibit	High-impedence
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	Inhibit	High-impedence

Typical ROM chip



For the same size of chip, it is possible to have more bits of ROM than a RAM, because the internal binary cells in ROM occupy less space than in RAM.

MEMORY ADDRESS MAP

Address space assignment to each memory chip

Example: 512 bytes RAM and 512 bytes ROM

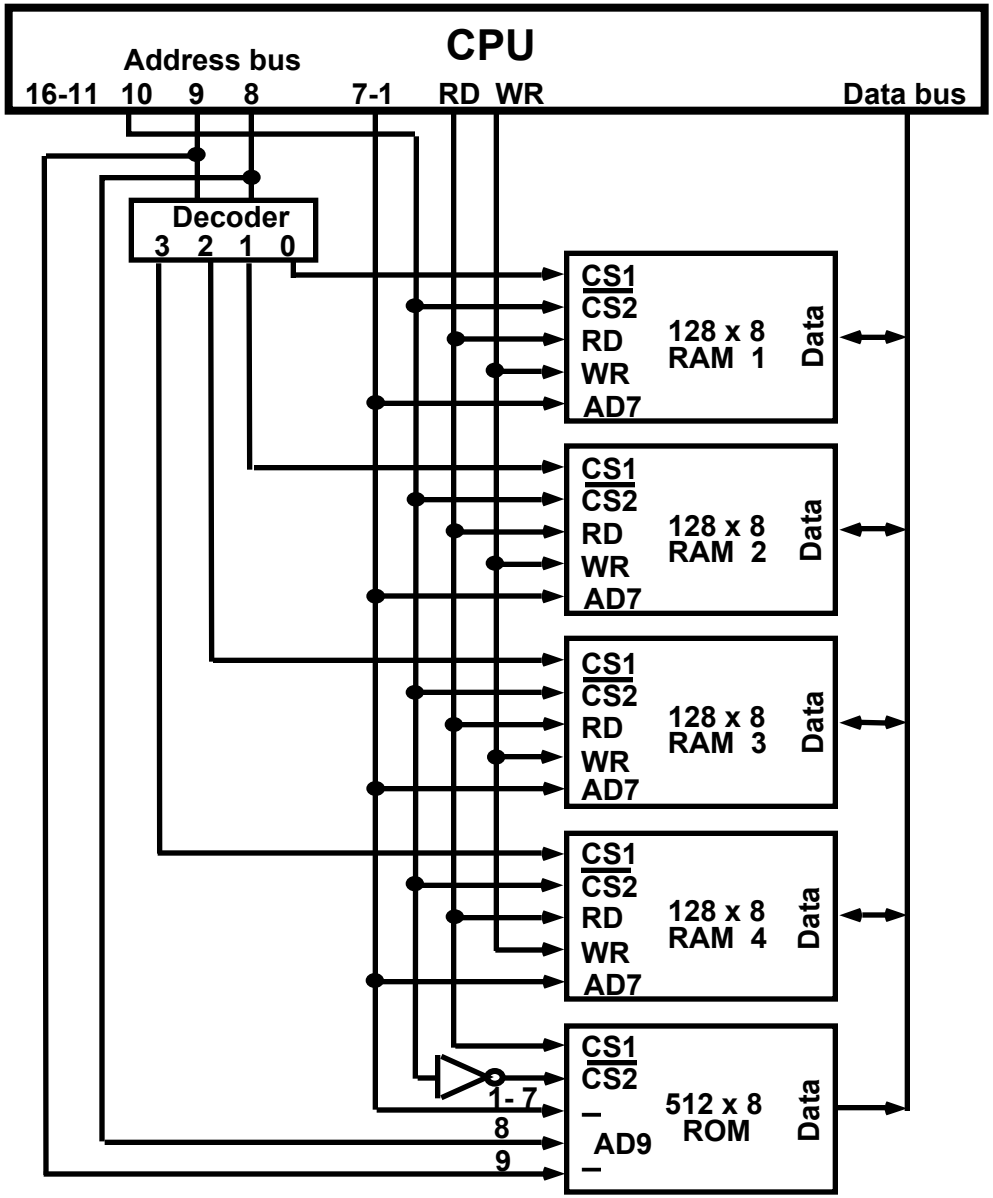
Component	Hexa address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000 - 007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

Memory Connection to CPU

- **RAM and ROM chips are connected to a CPU through the data and address buses**

- **The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs**

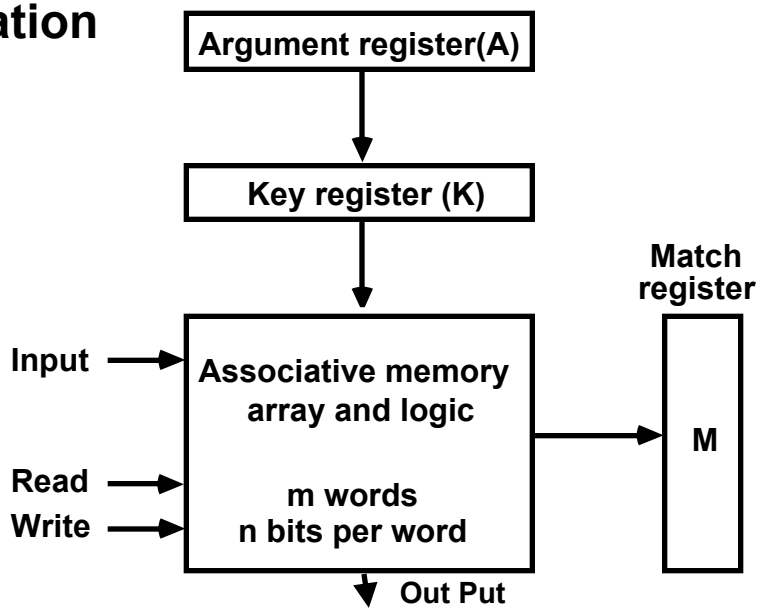
CONNECTION OF MEMORY TO CPU



ASSOCIATIVE MEMORY

- Accessed by the content of the data rather than by an address
- Also called Content Addressable Memory (CAM)

Hardware Organization

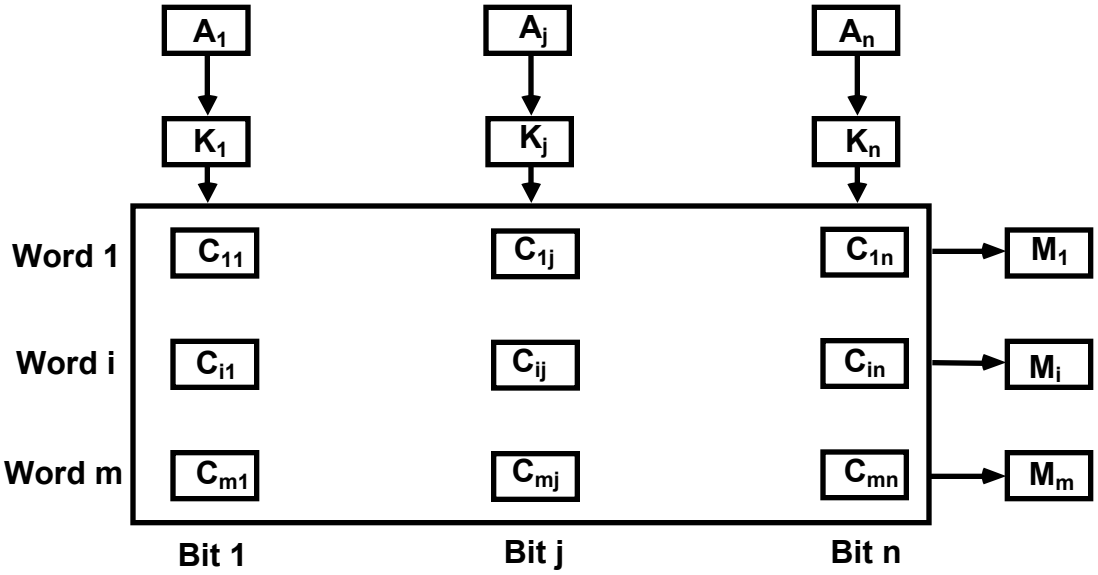


- Compare each word in CAM in parallel with the content of A(Argument Register)
- If CAM Word[i] = A, M(i) = 1
- Read sequentially accessing CAM for CAM Word(i) for M(i) = 1
- K(Key Register) provides a mask for choosing a particular field or key in the argument in A (only those bits in the argument that have 1's in their corresponding position of K are compared)

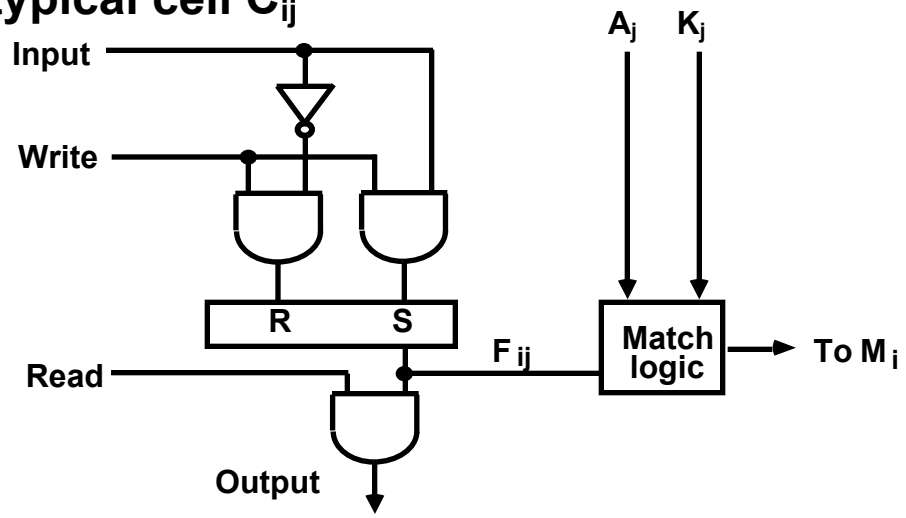
Example

A		101	111100
K		111	000000
Word 1	100	111100	no match
Word 2	101	000001	match

ORGANIZATION OF CAM



Internal organization of a typical cell C_{ij}



Match Logic

- The match logic for each word can be derived from t

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

Where $x_j = 1$ if the pair of the bits in position j are equ

For a word I to be equal to the argument in A we must

$$Mi = x_1 x_2 x_3 \ldots\ldots\ldots x_n$$

Match Logic

Now we include the key bit kj in the comparison logic. The req

$$x_j + K_j' = \begin{cases} 1 & \text{if } K_j = 0 \end{cases}$$

A term $(x_j + K_j')$ will be in 1 state if its pair of bits is not compar

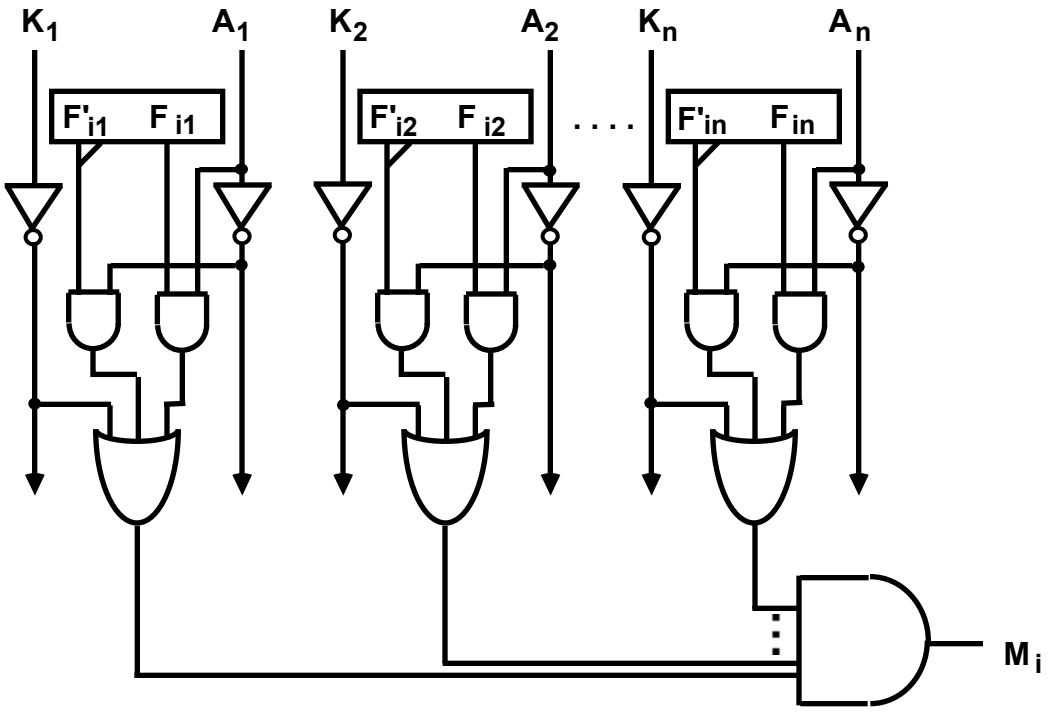
$$M_i = (x1 + K1')(x2 + K2')(x3 + K3').....(xn + Kn')$$

It can be expressed as

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

$\prod$ is a product symbol designing the AND operation of all n terms

MATCH LOGIC



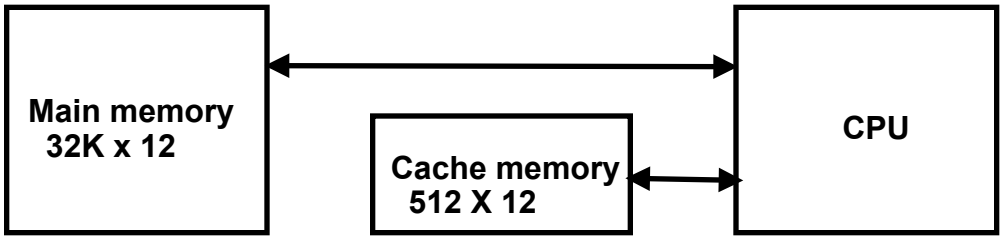
CACHE MEMORY

Locality of Reference

- The references to memory at any given time interval tend to be confined within a localized areas
- This area contains a set of information and the membership changes gradually as time goes by
- *Temporal Locality*
  The information which will be used in near future is likely to be in use already(e.g. Reuse of information in loops)
- *Spatial Locality*
  If a word is accessed, adjacent(near) words are likely accessed soon (e.g. Related data items (arrays) are usually stored together; instructions are executed sequentially)

Cache

- The property of Locality of Reference makes the Cache memory systems work
- Cache is a fast small capacity memory that should hold those information which are most likely to be accessed



PERFORMANCE OF CACHE

Memory Access

All the memory accesses are directed first to Cache
If the word is in Cache; Access cache to provide it to CPU
If the word is not in Cache; Bring a block (or a line) including that word to replace a block now in Cache

- How can we know if the word that is required is there ?
- If a new block is to replace one of the old blocks, which one should we choose ?

Performance of Cache Memory System

Hit Ratio - % of memory accesses satisfied by Cache memory system
Te: Effective memory access time in Cache memory system
Tc: Cache access time
Tm: Main memory access time

$$Te = Tc + (1 - h)\, Tm$$

Example: Tc = 0.4 μs, Tm = 1.2 μs, h = 0.85%
$Te = 0.4 + (1 - 0.85) * 1.2 = 0.58$ μs

MEMORY AND CACHE MAPPING - ASSOCIATIVE MAPPLING -

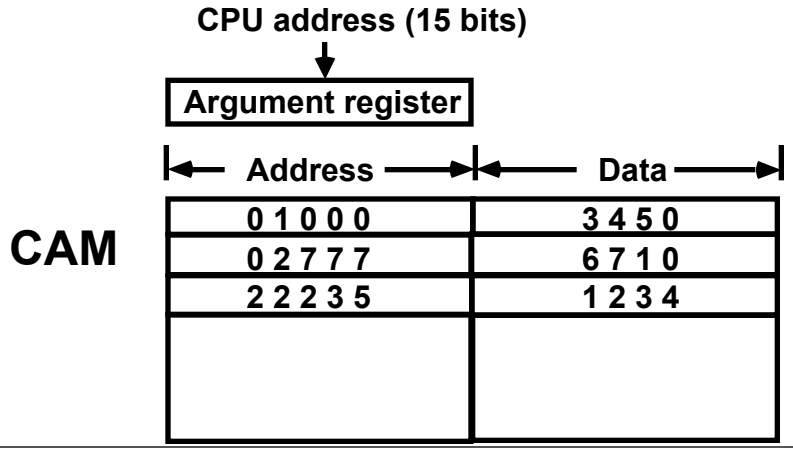
Mapping Function

Three types of mapping procedures are of practical interested when considering the organization of cache memory:

- Associative mapping
- Direct mapping
- Set-associative mapping

Associative Mapping

The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory.

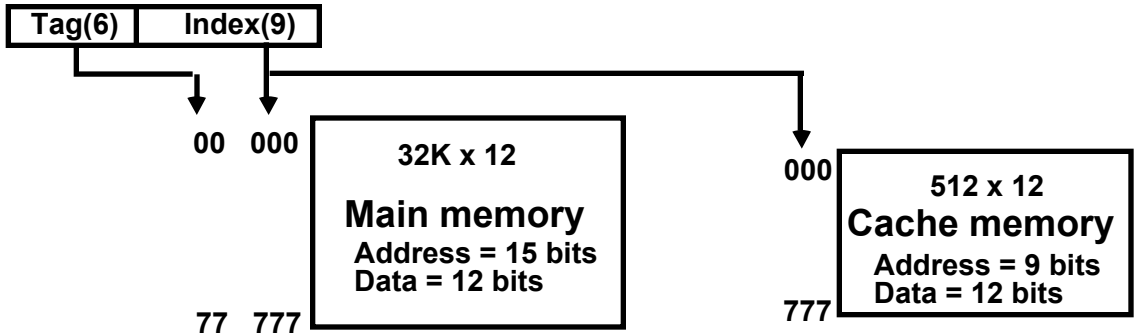


All numbers are in octal

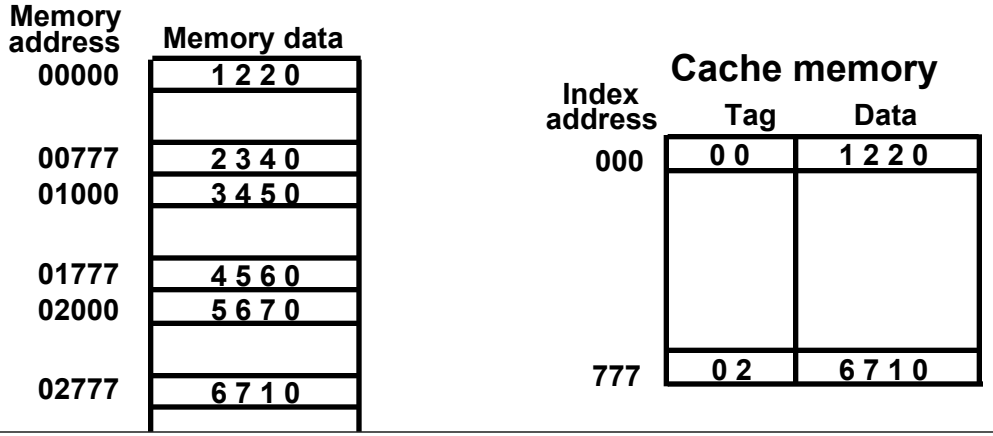
MEMORY AND CACHE MAPPING - DIRECT MAPPING -

- Each memory block has only one location to load in Cache
- Mapping Table is made of RAM instead of CAM
- n-bit memory address consists of 2 parts; k bits of Index field and n-k bits of Tag field
- n-bit addresses are used to access main memory and k-bit Index is used to access the Cache

Addressing Relationships



Direct Mapping Cache Organization



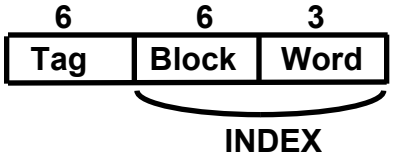
DIRECT MAPPING

Operation

- CPU generates a memory request with (TAG;INDEX)
- Access Cache using INDEX ; (tag; data)
  Compare TAG and tag
- If matches -> Hit
  Provide Cache[INDEX](data) to CPU
- If not match -> Miss
  Take the word from main memory, also bring the word in cache.

Direct Mapping with block size of 8 words

	Index	tag	data
Block 0	000	0 1	3 4 5 0
	007	0 1	6 5 7 8
Block 1	010		
	017		
Block 63	770	0 2	
	777	0 2	6 7 1 0



MEMORY AND CACHE MAPPING - SET ASSOCIATIVE MAPPING -

- **Each word of cache can store two or more words of memory under the same index address.**

<u>Set Associative Mapping Cache with set size of two</u> : Each Tag require 6 bits and each data word has 12 bits so the word length is 2(6+12)=36 bits. [so size will be 512 X 36]. So It can accommodate 1024 words Of Main memory.

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

BLOCK REPLACEMENT POLICY

Many different block replacement policies are available:

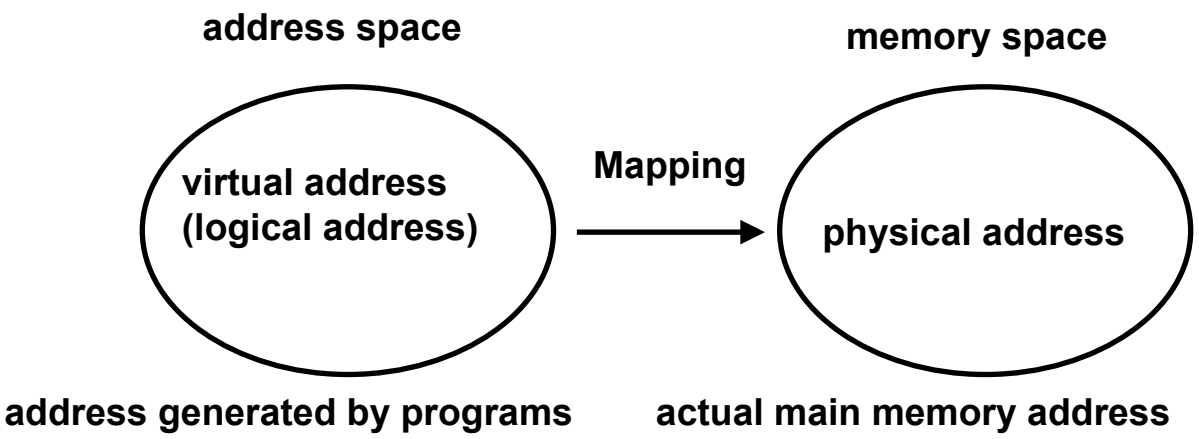
Random replacement

First in First out (FIFO) replacement

Least recently used (LRU) replacement

Virtual Memory

Give the programmer the illusion that the system has a
Address Space (Logical) and Memory Space (Physical
A virtual memory system provides a mechanism for tra

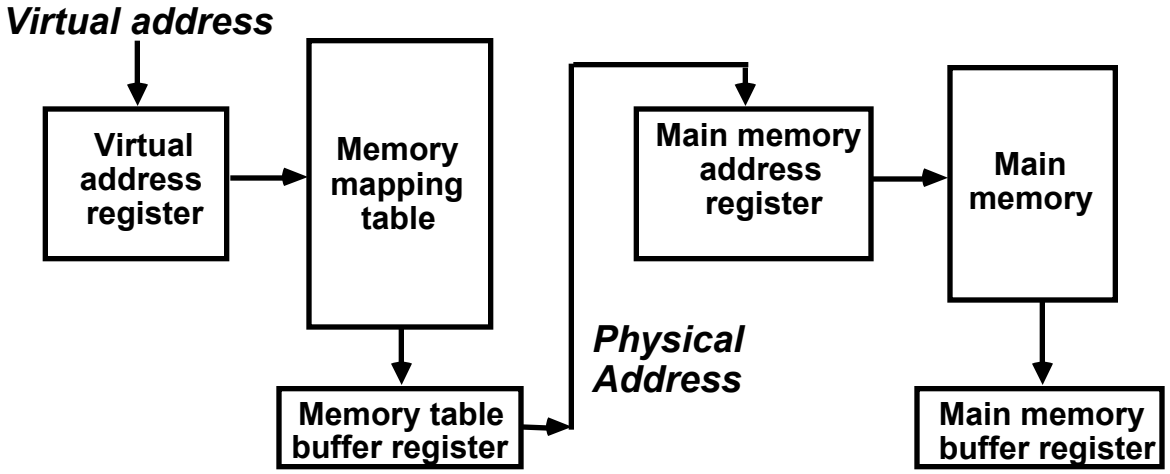


VIRTUAL MEMORY

Address Mapping

Memory *Mapping Table* for
Virtual Address -> Physical Address

A mapping table may be stored in a separate memory or in main memory . Virtual address generated by the program is matched with the mapping table , Then actual physical address is send to the memory address register.



ADDRESS MAPPING

Address Space and Memory Space are each divided into fixed size group of words called *blocks* or *pages*

1K words group

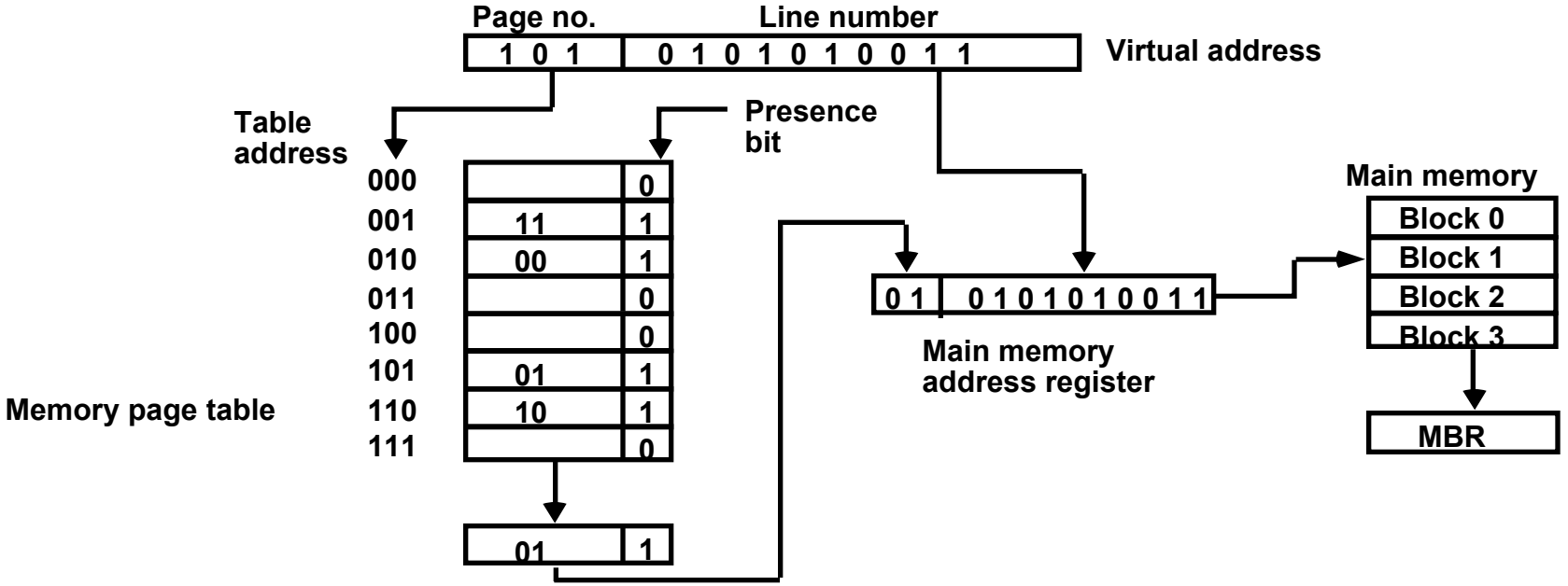
Address space
$N = 8K = 2^{13}$

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7

Memory space
$M = 4K = 2^{12}$

Block 0
Block 1
Block 2
Block 3

Organization of memory Mapping Table in a paged system



ASSOCIATIVE MEMORY PAGE TABLE

Assume that

Number of Blocks in memory = m

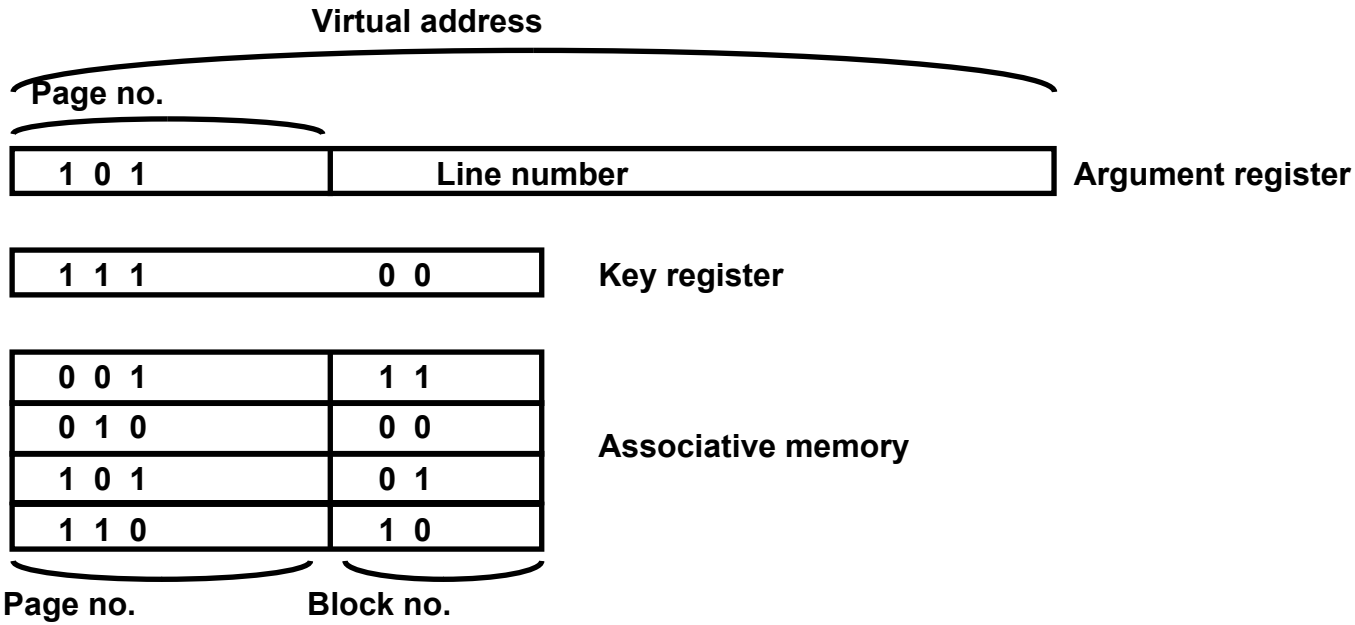
Number of Pages in Virtual Address Space = n

Page Table

- Straight forward design -> n entry table in memory
Inefficient storage space utilization <- n-m entries of the table is empty

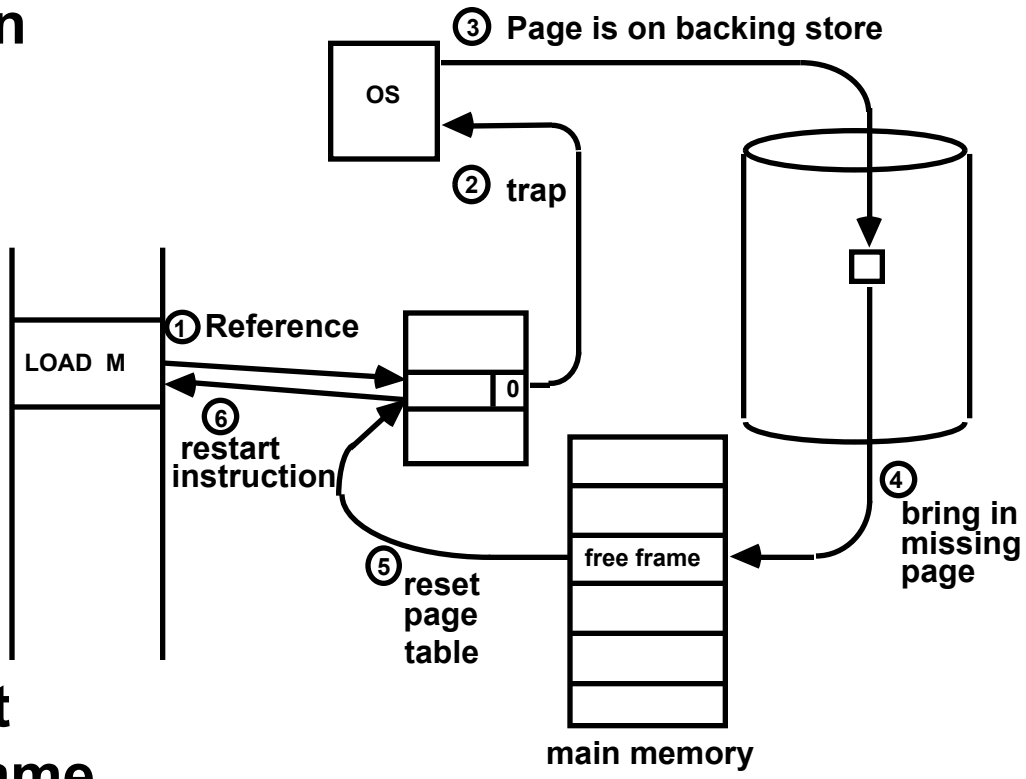
- More efficient method is m-entry Page Table

Page Table made of an Associative Memory m words;
(Page Number:Block Number)



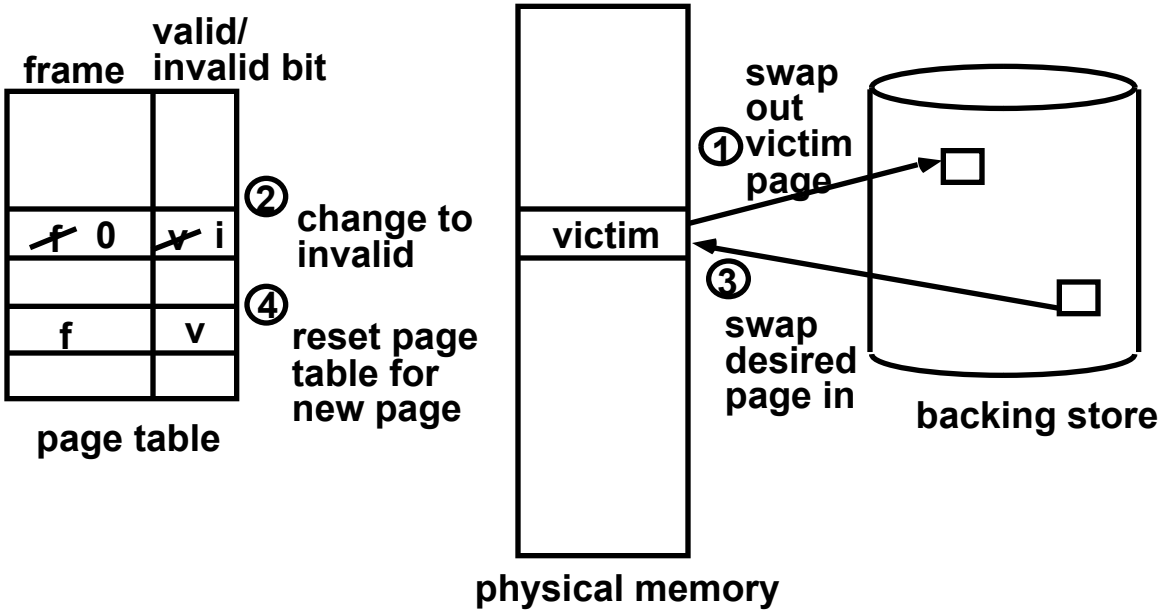
PAGE FAULT

1. CPU reference any page, in memory page table the presence bit is invalid.
2. Trap to the Operating system.
3. OS goes to the backing storage device where page is stored.
4. OS bring the page in main empty memory frame (block) , if any frame is not empty then replace any frame (by using replacement algo.)
5. Reset the page table.
6. CPU restart the execution.



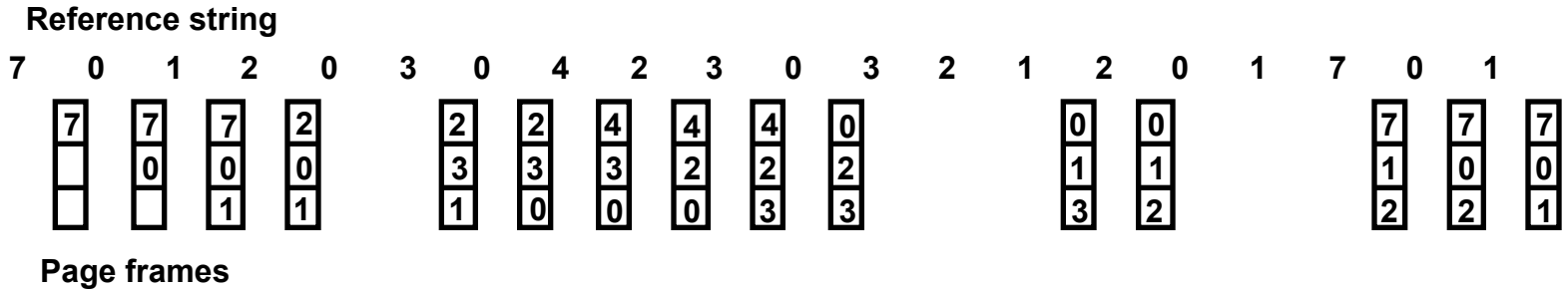
PAGE REPLACEMENT

1. Find the location of the desired page on the backing store
2. Find a free frame
   - If there is a free frame, use it
   - Otherwise, use a page-replacement algorithm to select a *victim* frame
   - Write the victim page to the backing store
3. Read the desired page into the (newly) free frame
4. Restart the user process



PAGE REPLACEMENT ALGORITHMS

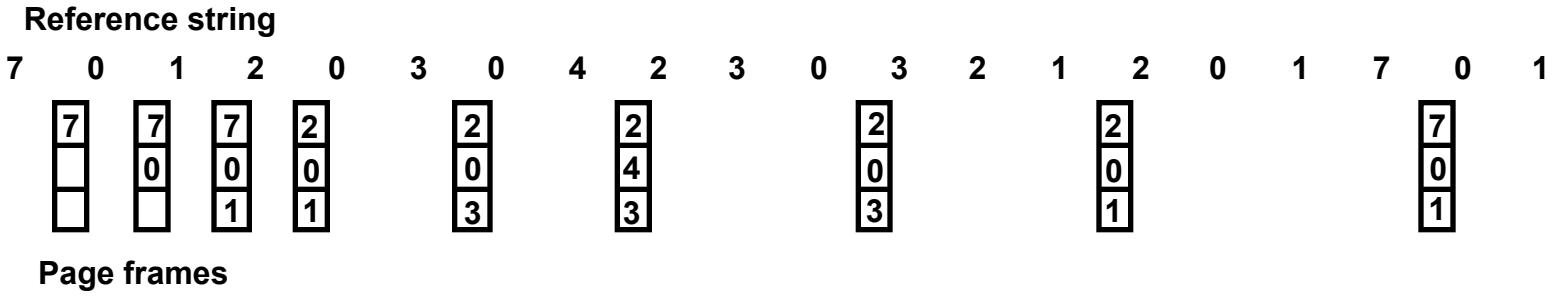
FIFO



FIFO algorithm selects the page that has been in memory the longest time
Using a queue - every time a page is loaded, its
- identification is inserted in the queue
Easy to implement
May result in a frequent page fault

Optimal Replacement (OPT) - Lowest page fault rate of all algorithms

Replace that page which will not be used for the longest period of time

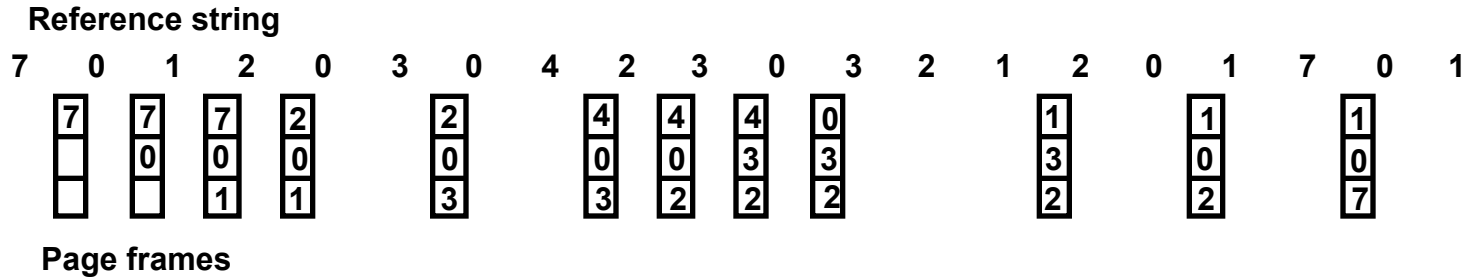


PAGE REPLACEMENT ALGORITHMS

LRU

- OPT is difficult to implement since it requires future knowledge
- LRU uses the recent past as an approximation of near future.

Replace that page which has not been used for the longest period of time



MRU (Most Recently used): **Replace the page which is used most Recently.**