3-4 Floating-Point Representation

mantissa
exponent

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	*Exponent*
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$.

Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

fraction

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	*Exponent*
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

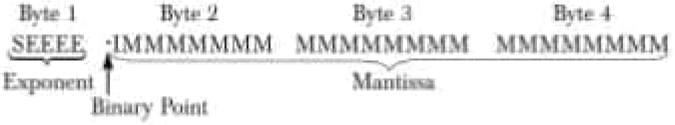
normalization

A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's. The number can be normalized by shifting it

three positions to the left and discarding the leading 0's to obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0's in the mantissa and exponent.

Two main standard forms of floating-point numbers are from the following organizations that decide standards: ANSI (American National Standards Institute) and IEEE (Institute of Electrical and Electronic Engineers). The ANSI 32-bit floating-point numbers in byte format with examples are given below:

Byte Format:



S – Sign of Mantissa, E – Exponent Bits in 2's complement, M – Mantissa Bits

Examples:

$$\begin{aligned} 13 = 1101 &= 0.1101 \times 2^4 \\ &= 00000100\ 11010000\ 00000000\ 00000000 \\ -17 = -10001 &= -0.10001 \times 2^5 \\ &= 10000101\ 10001000\ 00000000\ 00000000 \\ -0.125 = -0.001 &= -.1 \times 2^{-2} \\ &= 11111110\ 10000000\ 00000000\ 00000000 \end{aligned}$$

Q4. Write short notes on floating-point representation of decimal number.

Answer :

As mentioned earlier, the floating-point number is basically divided into two segments, with one segment representing the mantissa and other representing the exponent. Hence it takes the form as,

$$M \times K^e$$

Where 'M' corresponds to mantissa, 'e' corresponds to exponent and 'K' corresponds to radix.

For example, consider two floating-point numbers i.e., + 122.28 and + 5089.34

The floating-point number, i.e., 122.28 can be written as $+0.12228 \times 10^{+3}$ $(M \times K^e)$. Where + 0.12228 forms the fractional part or mantissa, the power '3' is exponent and '10' corresponds to radix. Similarly +5089.34 can be written as $+0.508934 \times 10^{+4}$ $(M \times K^e)$, where 0.5089 corresponds to fractional or mantissa part, + 4 corresponds to the exponent and '10' is the radix. While dealing with floating-point representation of decimal numbers one has to remember that the computer accommodates only the mantissa and exponents part including their signs in register while radix is just assumed.

Q5 What is the

Q37. Explain Booth's algorithm with its theoretical basis.

OR

What is a Booth's algorithm for 2's complement multiplications? Explain with an example.

Answer :

Model Paper-III, Q7

Booth's algorithms is the most strong and effective algorithm for solving signed 2's complement multiplication. It produces 2 n-bit product and gives equal priority to both positive and negative numbers. The algorithm adopts a techniques where in, it represents the multiplies as a difference between numbers. Thereby reducing the number of operations required for multiplication. It takes into consideration the fact that 0's in the multiplier require just shifting, and 1's in the multiplier from bit position n to bit position m can be treated as $2^{n+1} - 2^{m}$. For example, multiplier 00011110 (+30) can be represented as $2^5 - 2^1$ $32 - 2 = 30$. Therefore, the multiplication of multiplicand M and the multiplier 30 (i.e. $M \times 30$) is done as $M \times 2^5 - M \times 2^1$.

In Booth's algorithm, series of events can be described with the help of a flowchart as shown in figure (1).

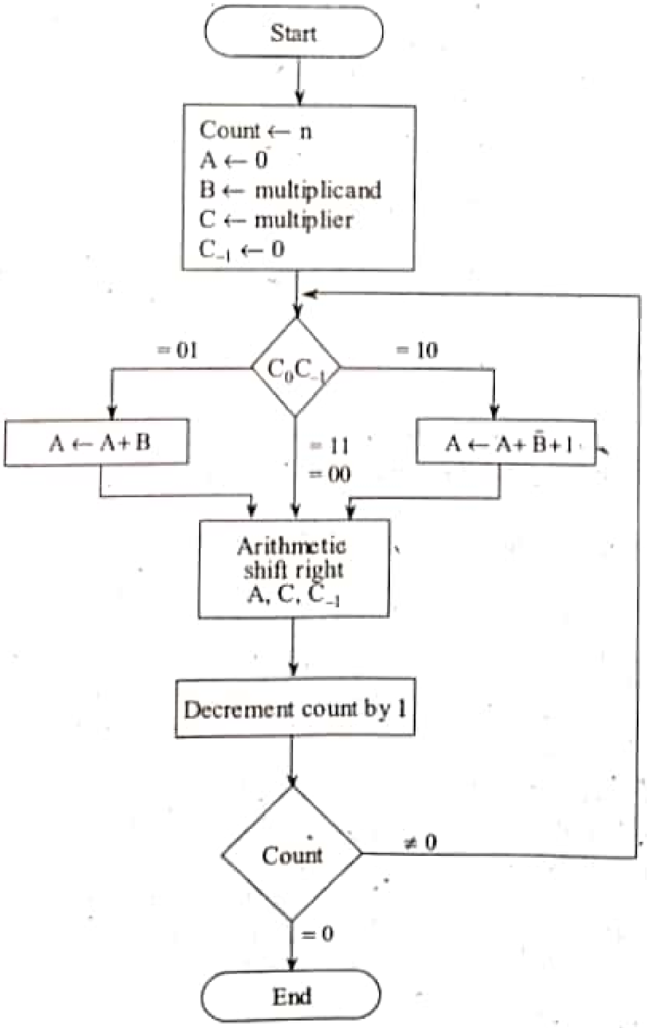


Figure (1): Booth's Algorithm for Signed Multiplication

Q35. Draw a flowchart to explain how addition and subtraction of two fixed point numbers can be done. Also, draw a circuit using full adders for the same.

Answer : **Model Paper-I, Q7(b)**

Flowchart representing the hardware implementation of adder/subtracter circuit is as shown in following figure,

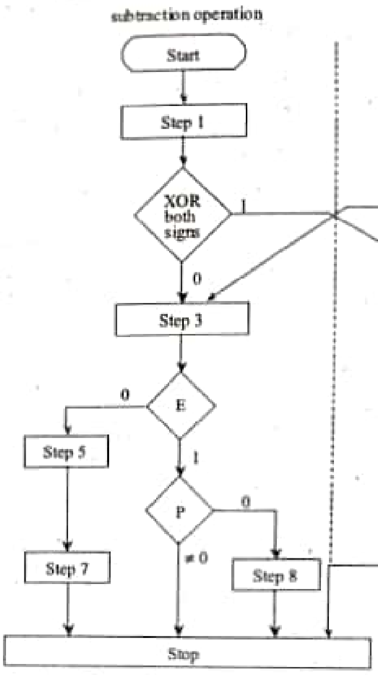


Figure: Flowchart for Signed-magnitude Addition/Subtraction

Following are the sequence of steps important to understand the above flowchart.

Addition Operation

Step 1

Let the two numbers be represented as P and Q.

Step 2

Here, the signs of these two numbers are compared by an XOR logic circuit. If the result is zero it is understood that the sign are same, else they are different. If two signs are same, the two numbers are added i.e. step 3 of addition operation, else they are subtracted i.e. step 3 of subtraction operation.

Step 3

Here, the values of P and Q are summed and the result is stored in RP register.

Step 4

The overflow bit which is stored in R is transferred to add overflow flip-flop E.

In short, the mentioned operations can be rewritten as,

Step 1

Define P, Q

Step 2

$P_{sign} \oplus Q_{sign}$

Step 3

$RP \leftarrow A + B$

Step 4

$E \leftarrow R;$

Subtraction Operation

Step 1

Let the two numbers be represented as P and Q.

Step 2

Here, the signs of these two numbers are compared by an XOR logic circuit. If the result is zero, it is understood that the signs are same, else they are different. If two signs are same, the two numbers are subtracted i.e. step 3 of subtraction operation else they are added i.e. step 3 of the addition operation.

Step 3

Here, the value of Q is subtracted with the value P (i.e. $P+\overline{Q}+1$) and the result is stored in a separate RP register which combines the values of R and P. Here the input is derived from two sources i.e. from step 2 of subtraction operation and step 2 of addition operation later the additional overflow flip flop is assigned a value zero.

Step 4

In this step the value of R is checked, this is to ascertain whether R is empty or contains the overflow value. If R is 0, it indicates that R is empty or if R is 1, it indicates that R contains an overflow value. Here if R is empty, step 5 is performed else step 6 is performed.

Step 5

When R is 0 it indicates that $P < Q$. Therefore, the value of P is complemented and is stored again in P.

Step 6 or C6

Here, the value in P is checked, if it is equal to zero, then step 8 is performed or if P does not contain a value zero then the process is directly terminated.

Step 7

In this step the value of P is added to excess 1 and the result is stored in P. Also the value of P_{sign} flip-flop is complemented and result is stored in the same flip-flop.

Step 8

A value zero is stored in the P_{sign} flip-flop and the process is halted.

In short, the above mentioned steps can be rewritten as,

Step 1

Define P, Q

Step 2

$P_{sign} \oplus Q_{sign}$

Step 3

$ER \leftarrow P + \overline{Q} + 1$

$E \leftarrow 0$

Step 4

Compare R

Step 5

$P \leftarrow \overline{P}$

Step 6

Compare P

Step 7

$P \leftarrow P+1$

$\text{sign}(P) \leftarrow \text{sign}(\overline{P})$

Step 8

$\text{sign}(P) \leftarrow 0$

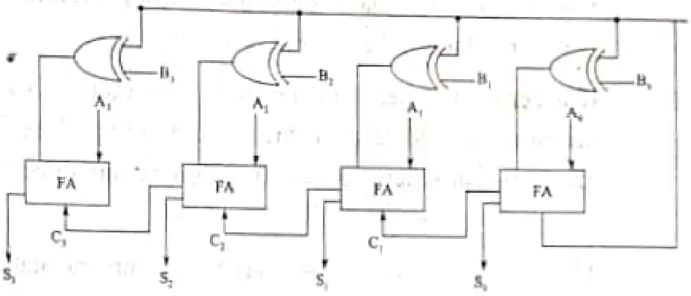


Figure: Hardware for Integer 4-bit Addition/Subtraction

3.2.2 Multiplication Algorithms

Q36. Draw a flowchart which explains multiplication of two signed magnitude fixed point numbers.

Answer :

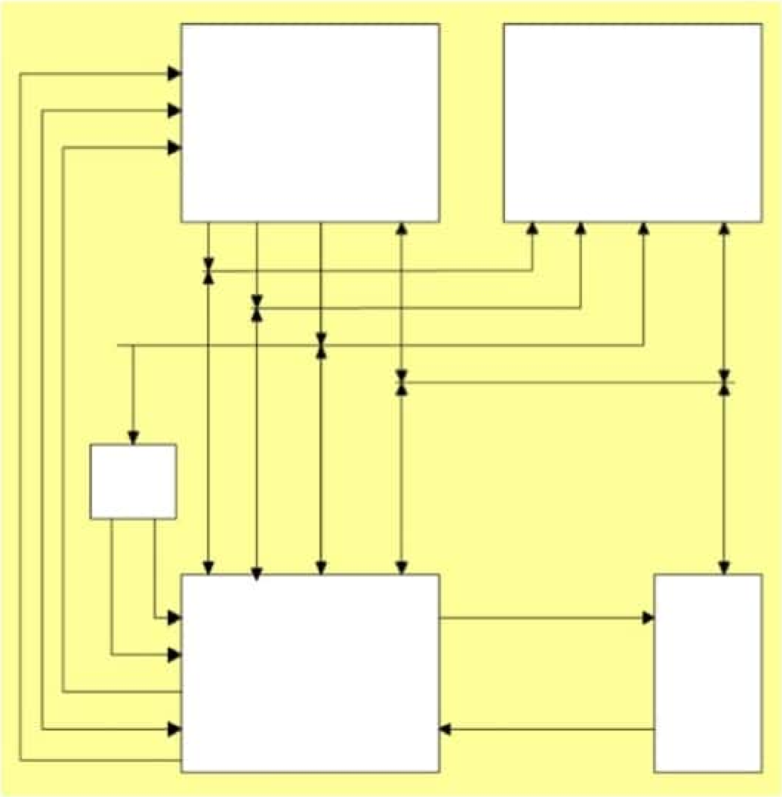
The process of multiplying two signed-magnitude fixed-point numbers can be illustrated with an example which is given below,

```
Multiplicand 17 :         1 0 0 0 1
 Multiplier 11  :         0 1 0 1 1
                          ---------
                          1 0 0 0 1
                        1 0 0 0 1
                      0 0 0 0 0
                    1 0 0 0 1
                  0 0 0 0 0
Final product 187 : 0 1 0 1 1 1 0 1 1
```

The multiplication process begins by taking the least significant bit of the multiplier. If this bit is 0, then zeros are written at the corresponding places in the product and if it is 1, then all the bits in the multiplicand are written without any change. In successive lines, the numbers written are moved towards left by one position from the previous number. The process of multiplication continues till the most significant bit is encountered. Finally, all the numbers are added to generate the product of two binary numbers. By considering the signs of both multiplier and multiplicand the product's sign can be determined i.e., if both the signs are similar then the product's sign is positive, else it is negative.

DMA Transfer (I/O to Memory)

- 1) I/O Device sends a DMA request
- 2) DMAC activates the **BR** line
- 3) CPU responds with **BG** line
- 4) DMAC sends a DMA acknowledge to the I/O device
- 5) I/O device puts a word in the data bus (*for memory write*)
- 6) DMAC write a data to the address specified by **Address register**
- 7) Decrement **Word count register**
- 8) **Word count register** = 0 이면 **EOT** interrupt 발생하여 CPU에 알림
- 9) **Word count register** ≠ 0 이면 DMAC checks the DMA request from I/O device



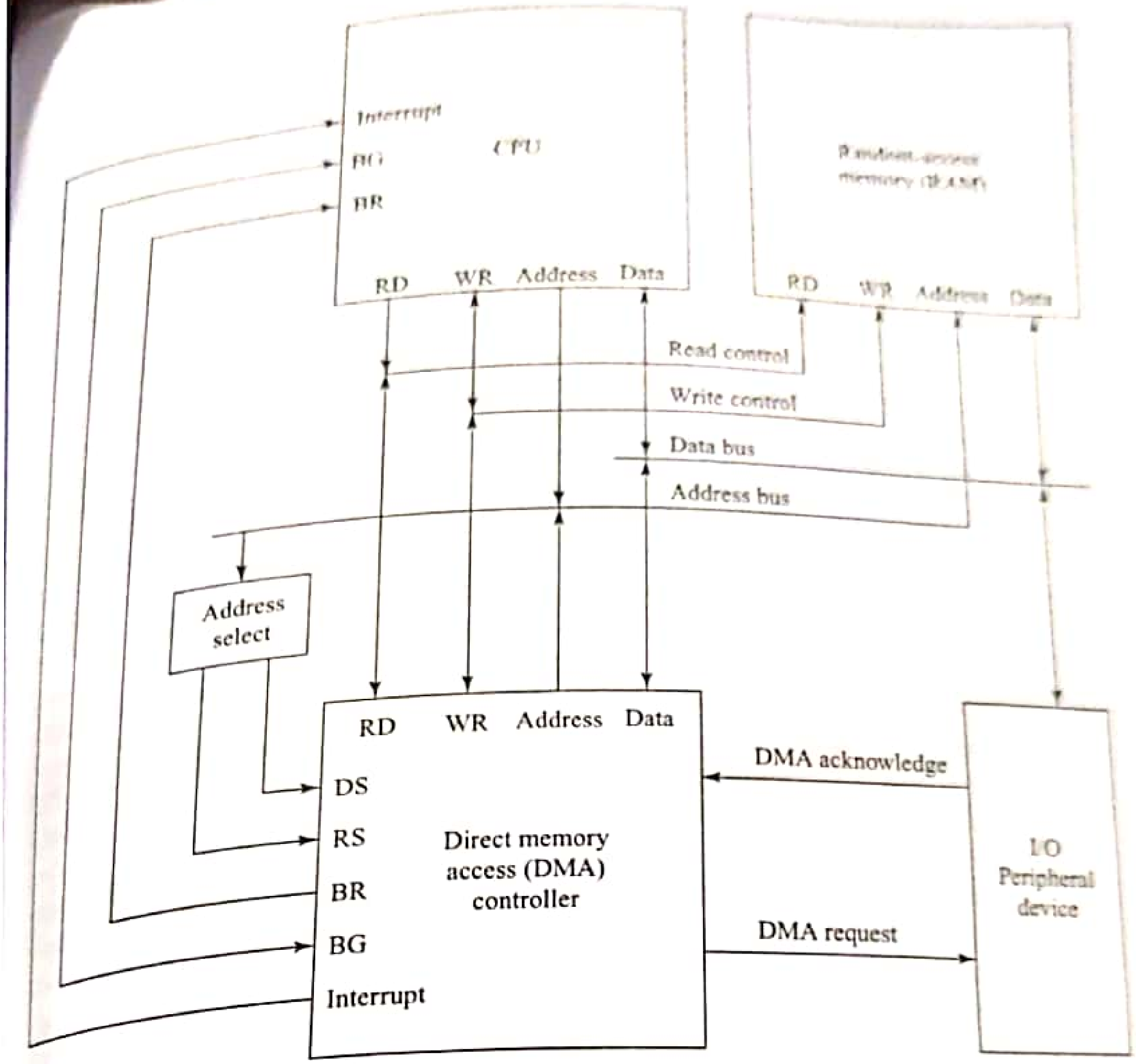


FIGURE 12.18 DMA transfer in a computer system.

DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the *DS* and *RS* lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the *BR* line, informing the CPU to relinquish the buses. The CPU responds with its *BG* line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the *RD* or *WR* signal, and sends a DMA acknowledge to the peripheral device. Note that the *RD* and *WR* lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the *BG* line. When *BG* = 0, the *RD* and *WR* are input lines allowing the CPU to communicate with the internal DMA registers. When *BG* = 1, the *RD* and *WR* are output

Scanned with CamScanner

Subject:Computer Organization and Architecture

MODES OF DATA TRANSFER:

The data that we use is actually binary information that we send or receive between external device and memory storage.

Processor tries to execute any I/O instruction that accepts data for temporarily. Whatever Data transfer between processor and memory unit ultimately.

The data transfer between i/o device and CPU that has various modes:

1)Programmed i/o
2)Interrupt intiated i/o
3)DMA Direct Memory access

Prog. I/o:

1)whatever instruction you have provided for the I/O operations the result of these instructions are written here.

2)Data you transfer from or to(CPU-external device)or(external device to CPU), whatever is sent are monitored by the CPU.

3) This data transfer is done by instruction program

4) once you have done the data transfer, the data is transferred is monitored by the CPU by using interface. Else what next time they could sent data again.

Interrupt i/o initiated input/output:

1)We are using Interrupt facility to avoid time consumption that we face from Programmed I/O method

2) Prog. I/O method keeps processor busy between program loops

3)Interrupt: to stop voluntarily: they ask interface to generate interrupt signal when data is available from the device

4) with the help of interrupt, they able to stop current program and move to another so this way the data is not comes under program loop.

This how processor does not need to work unnecessarily.

DMA:

1)Whatever data we transfer we do with the help of memory bus and data is transfer between CPU and external Device.

2)It transfers data with the help of interface. As soon as it starts to supply. The interface collect the starting address and it also collect the number of words and they jump to another work

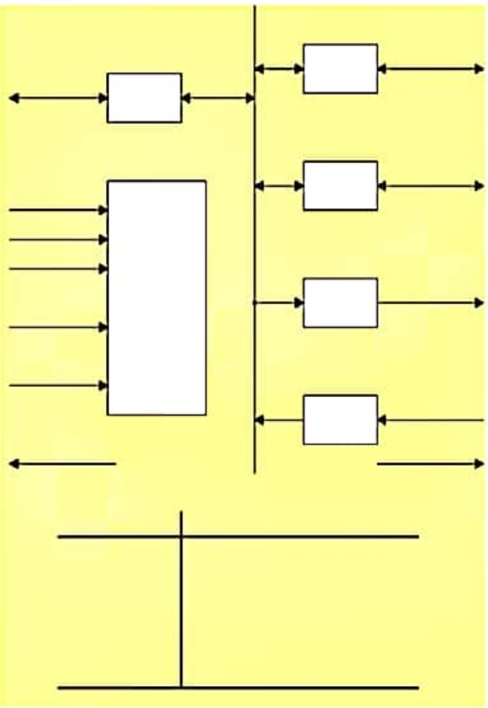
3)DMA request for memory cycles from memory bus. As it granted by the Memory unit It directly transfer data to memory with the help of the bus.

4)**Work faster than other modes.**

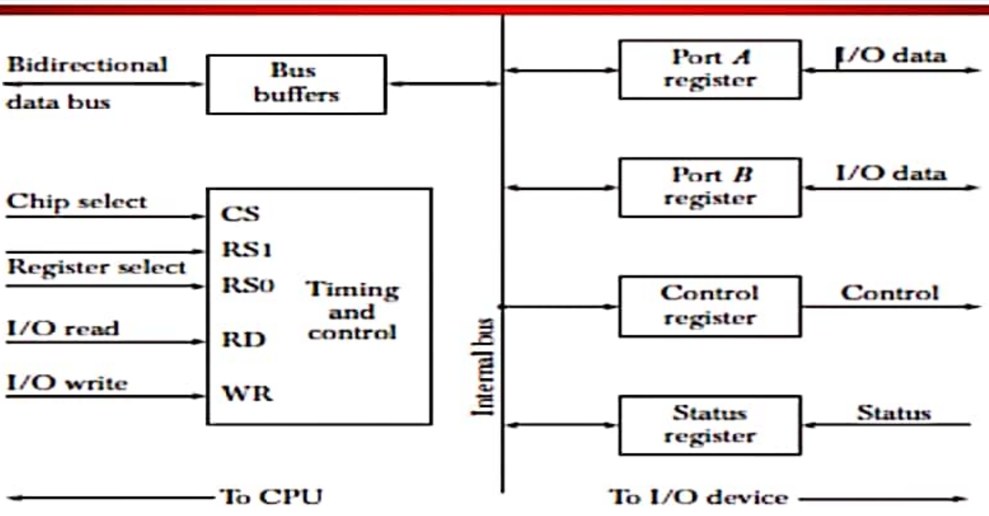
- Example of I/O Interface : *Fig. 11-2*
  - 4 I/O port : Data port A, Data port B, Control, Status
    - » 비교 : 8255 PIO (port A, B, C, Control/Status)
  - Address Decode : CS, RS1, RS0

11-3 Asynchronous Data Transfer

- Synchronous Data Transfer
  - All data transfers occur simultaneously during the occurrence of a clock pulse
  - Registers in the **interface** share a common clock with **CPU** registers
- Asynchronous Data Transfer
  - Internal timing in each unit (*CPU and Interface*) is independent
  - Each unit uses its own private clock for internal registers



따라서 **Asynchronous** 에서는 언제 data transfer 가 발생하는지 알리는 신호가 필요하며 **Strobe** 와 **Handshake** 방식을 사용함



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 11-2 Example of I/O interface unit.

- **Strobe** : Control signal to indicate the time at which data is being transmitted
  - 1) Source-initiated strobe : *Fig. 11-3*
  - 2) Destination-initiated strobe : *Fig. 11-4*

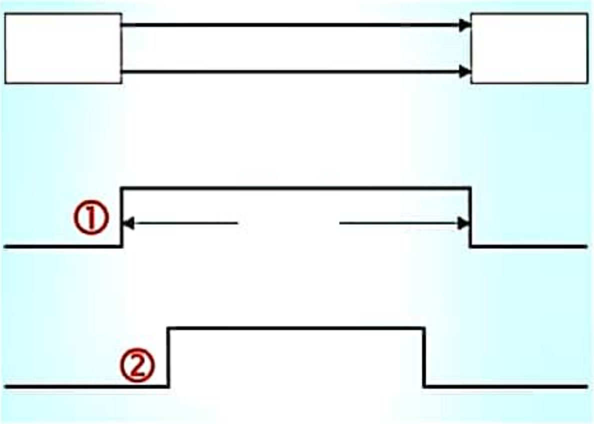


Fig. 11-3 Source-initiated strobe

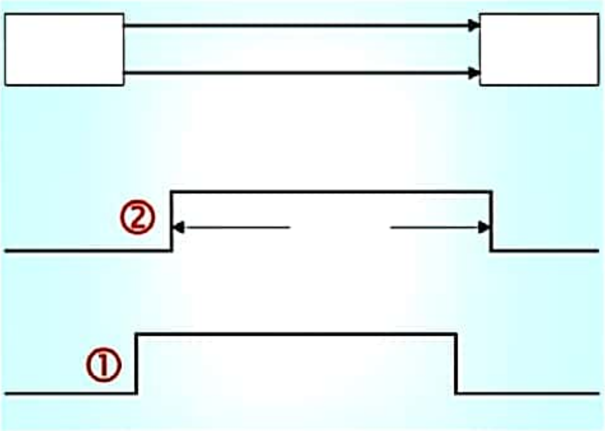
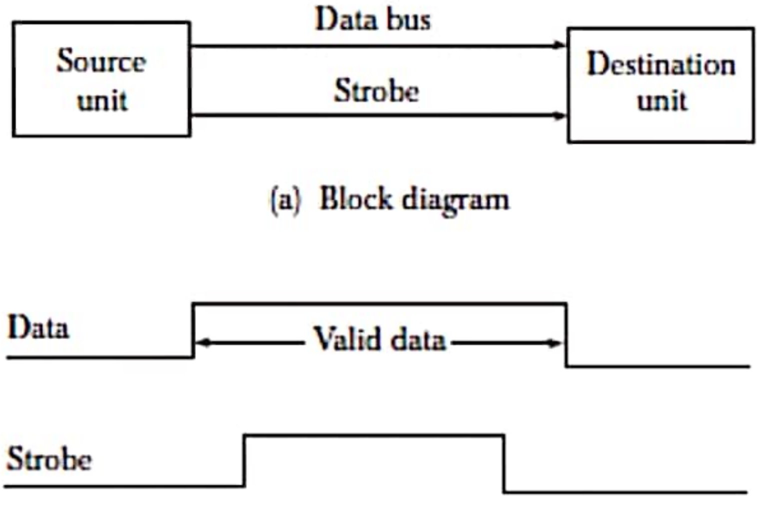


Fig. 11-4 Destination-initiated strobe

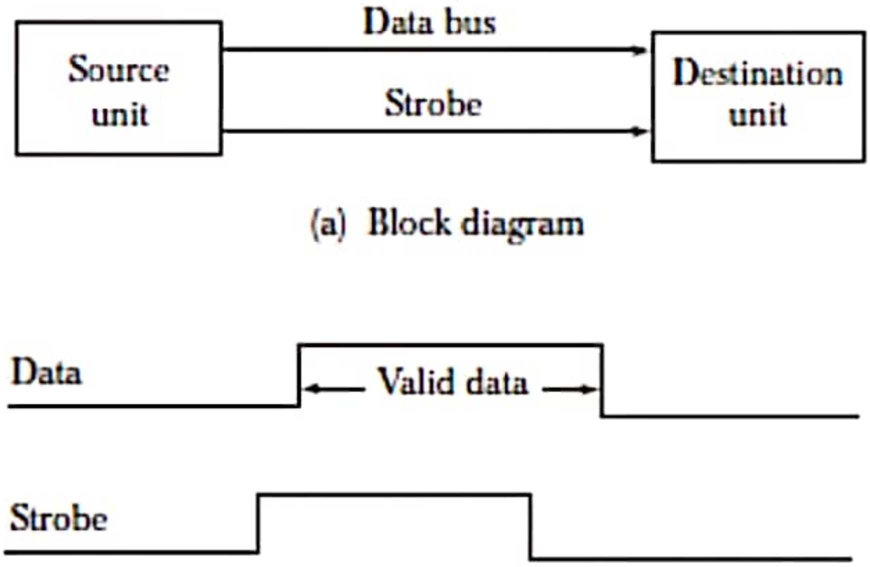
- Disadvantage of strobe method
  - » Destination 이 Data를 아무 이상 없이 잘 가져 갔는지 알 수가 없다.
  - » 따라서 Handshake method를 사용하여 Data 전송을 확인함



(a) Block diagram

(b) Timing diagram

Figure 11-3 Source-initiated strobe for data transfer.



(a) Block diagram

(b) Timing diagram

Figure 11-4 Destination-initiated strobe for data transfer.

- Handshake : Agreement between two independent units
  - 1) Source-initiated handshake : *Fig. 11-5*
  - 2) Destination-initiated handshake : *Fig. 11-6*

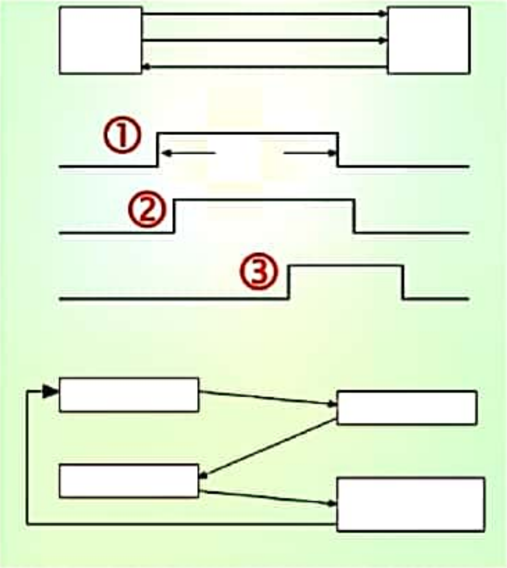


Fig. 11-5 Source-initiated handshake

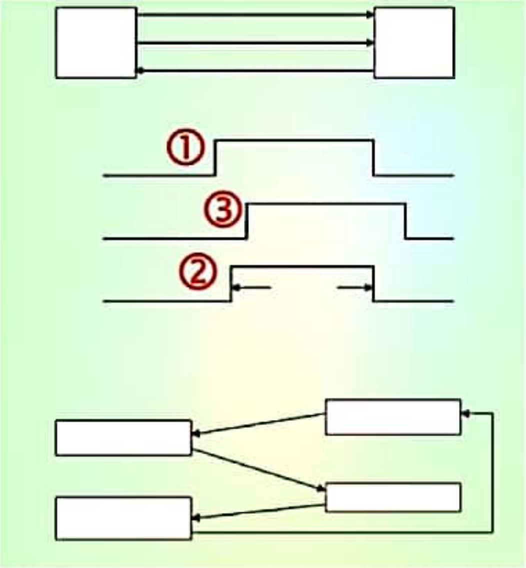
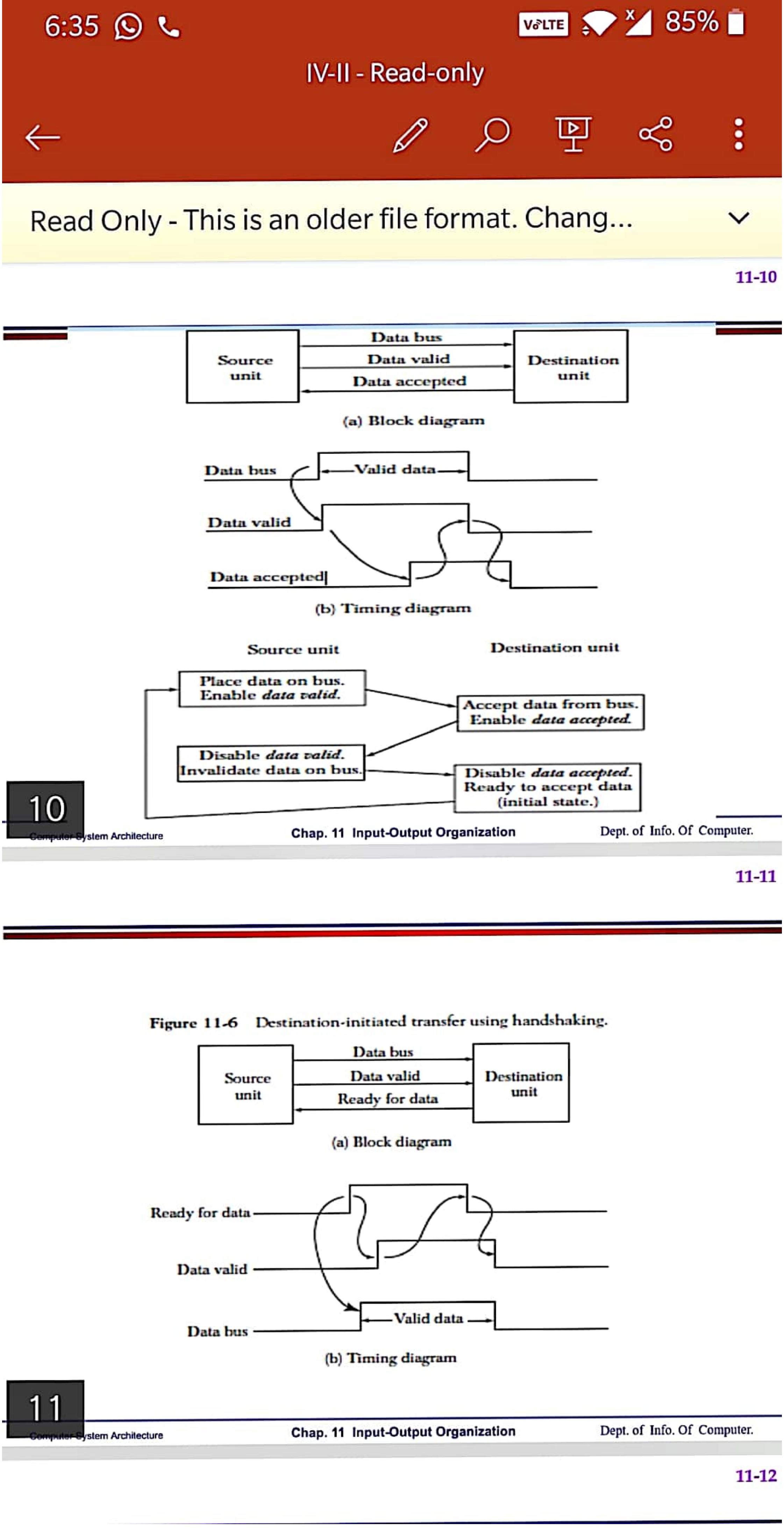


Fig. 11-6 Destination-initiated handshake

- **Timeout** : If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred.



Read Only - This is an older file format. Chang...

(a) Block diagram

(b) Timing diagram

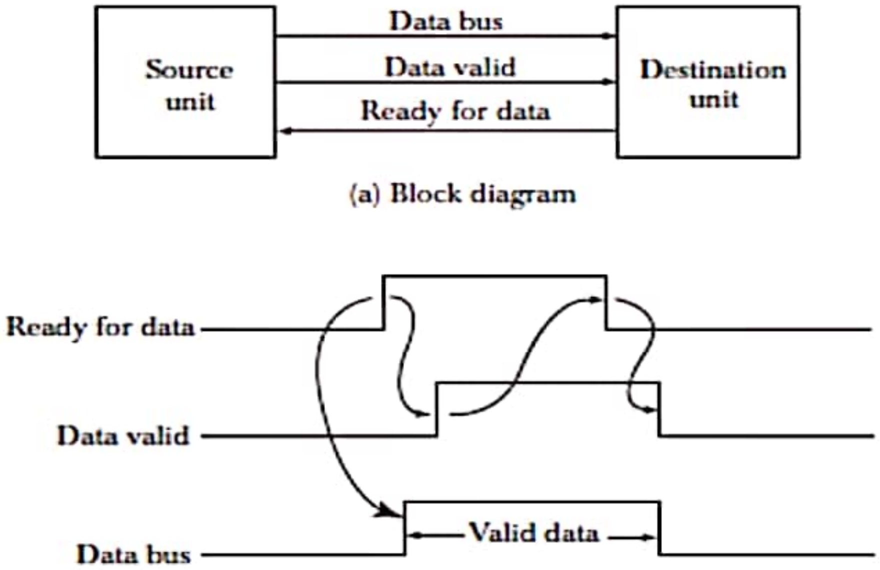
Source unit
Destination unit
Place data on bus. Enable *data valid.*
Accept data from bus. Enable *data accepted.*
Disable *data valid.* Invalidate data on bus.
Disable *data accepted.* Ready to accept data (initial state.)

Figure 11-6 Destination-initiated transfer using handshaking.

(a) Block diagram

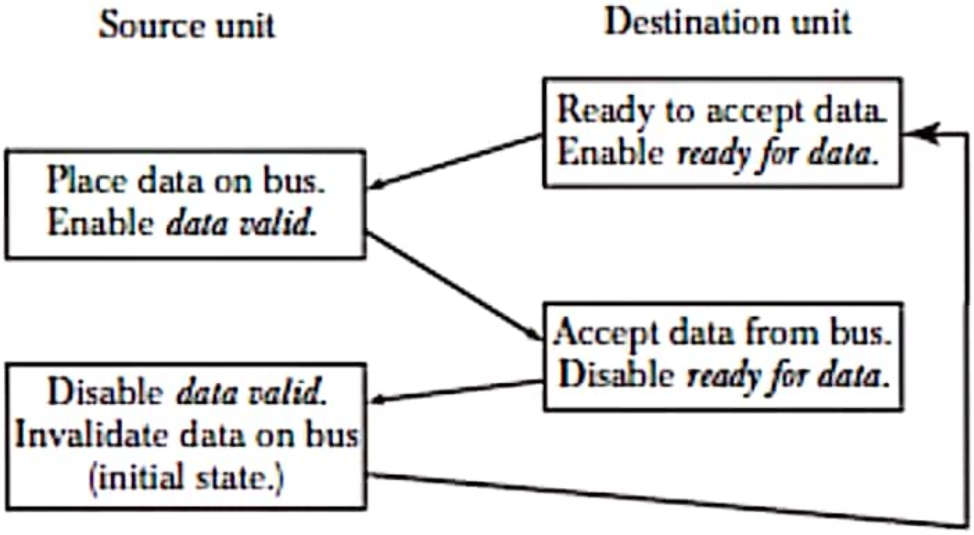
(b) Timing diagram

Figure 11-6 Destination-initiated transfer using handshaking.



(a) Block diagram

(b) Timing diagram



Ready to accept data. Enable *ready for data.*

Place data on bus. Enable *data valid.*

Accept data from bus. Disable *ready for data.*

Disable *data valid.* Invalidate data on bus (initial state.)

(c) Sequence of events

Q9. Write a short note on cache coherence.

Answer : Model Paper-II, Q1(j)

Cache coherence refers to situation wherein multiple cached copies of the shared data are maintained such that all copies have the same value.

It is a known fact that, in order to balance between the speeds of a processor and the main memory, a small memory unit known as *cache* is introduced. Due to this, processor need not search the entire main memory unit for the sake of the required data, rather it refers the cache memory for satisfying the requirement. In a multiprocessor system, apart from the shared memory (the memory shared by all the processors) each processor maintains it's own set of memories referred to as *local memories*. The processor can utilize either a part or the entire local memory as a cache memory.

13-5 Cache Coherence

The operation of cache memory is explained in Sec. 12-6. The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory. In the *write-through* policy, both cache and main memory are updated with every write operation. In the *write-back* policy, only the cache is updated and the location is marked so that it can be copied later into main memory.

In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory. To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical. This requirement imposes a *cache coherence* problem. A memory scheme is *coherent* if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Without a proper solution to the cache coherence problem, caching cannot be used in bus-oriented multiprocessors with two or more processors.

Conditions for Incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data. Read-only data can safely be replicated

without cache coherence enforcement mechanisms. To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12. Sometime during the operation an element X from main memory is loaded into the three processors, P_1, P_2, and P_3. As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory.

If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache. This is shown in Fig. 13-13. A store to X (of the value of 120) into the cache of processor P_1 updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value. In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy. I/O-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.

Solutions to the Cache Coherence Problem

Various schemes have been proposed to solve the cache coherence problem in shared memory multiprocessors. We discuss some of these schemes briefly here. See references 3 and 10 for more detailed discussions.

A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. This method violates the principle of closeness of CPU to cache and increases the average memory access time. In effect, this scheme solves the problem by avoiding it.

For performance considerations it is desirable to attach a private cache to each processor. One scheme that has been used allows only nonshared and read-only data to be stored in caches. Such items are called *cachable*. *Shared writable data are noncachable*. The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches. The noncachable data remain in main memory. This method

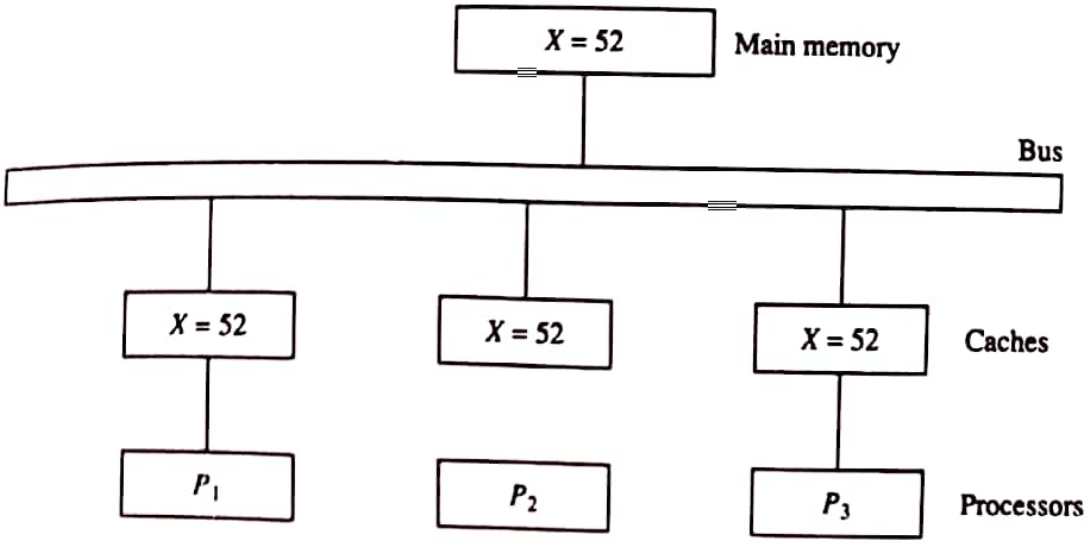
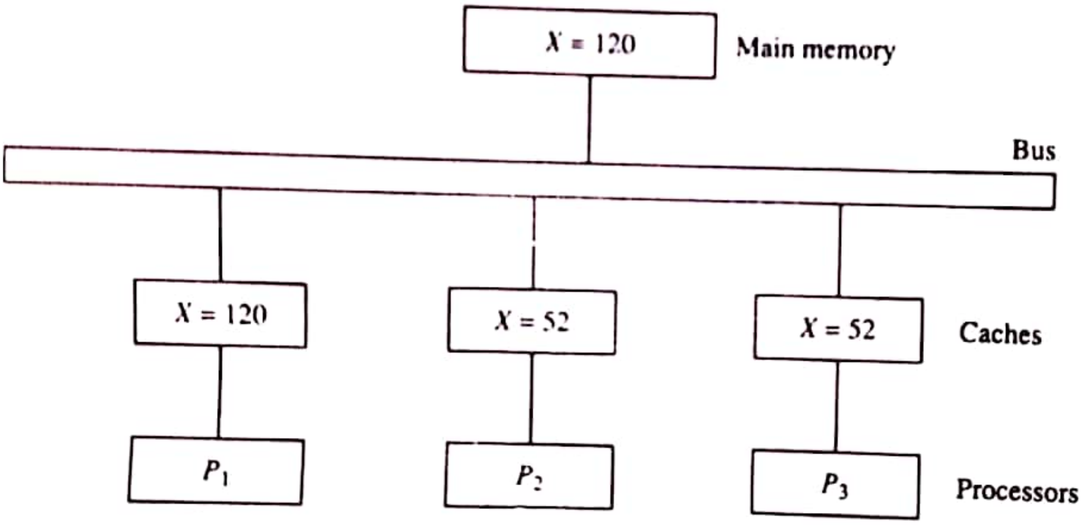
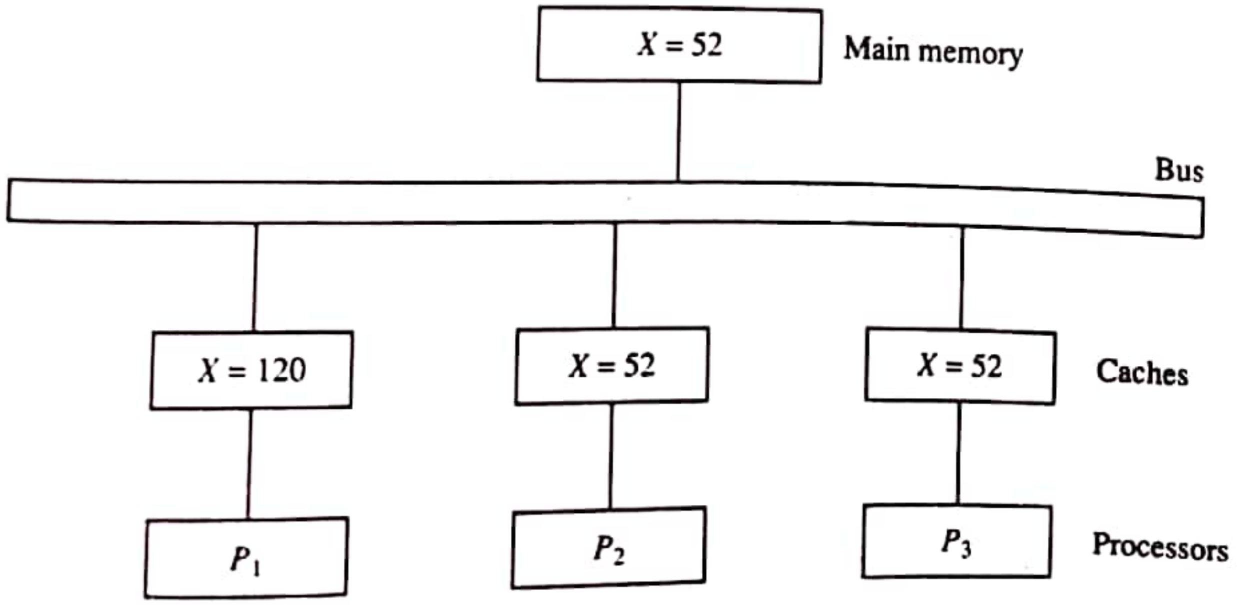


Figure 13-12 Cache configuration after a load on X.

Figure 13-13 Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy



(b) With write-back cache policy

restricts the type of data stored in caches and introduces an extra software overhead that may degradate performance.

A scheme that allows writable data to exist in at least one cache is a method that employs a *centralized global table* in its compiler. The status of memory blocks is stored in the central global table. Each block is identified as *read-only* (RO) or *read and write* (RW). All caches can have copies of blocks identified as RO. Only one cache can have a copy of an RW block. Thus if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes. The two methods mentioned previously use software-based procedures that require the ability to tag information in order to disable caching of shared writable data. Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency. In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor the network for possible write operations. Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected. The bus controller that monitors this action is referred to as a *snoopy cache controller*. This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

snoopy cache controller

Various schemes have been proposed to solve the cache coherence problem by means of snoopy cache protocol. The simplest method is to adopt a write-through policy and use the following procedure. All the snoopy controllers watch the bus for memory store operations. When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated. The local snoopy controllers in all other caches check their memory to determine if they have a copy of the word that has been overwritten. If a copy eyists in a remote cache, that location is marked invalid. Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches. If at some future time a processor accesses the invalid item from its cache, the response is equivalent to a cache miss, and the updated item is transferred from main memory. In this way, inconsistent versions are prevented.

Difference –

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and micro programmed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction execute in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word	Instructions are larger than the size of one word

every instruction should posses equal number of

CISC

Complex instruction set computing is another design strategy which consists of full set of computer instructions. In this type of CPU design strategy, a single instruction instructions are responsible for providing the required capabilities efficiently. In this type of CPU design strategy, a single instruction can execute multiple low-level operations. Moreover, the instruction-set is even capable of executing multi-step operations addressing modes within single instruction. The instruction-set consists of 120 to 350 instructions.

Q12. Differentiate between RISC and CISC characteristics.

Model Paper-II

Answer :

	RISC		CISC
1.	In RISC, simple instructions consume only one cycle to execute and an average CPI value is less than 1.5.	1.	In CISC, complex instructions consumes multiple cycles to execute and an average CPI value lies between 2 and 15.
2.	It consists of less number of instructions.	2.	It consists of more number of instructions.
3.	It has fixed instruction format.	3.	It has variable instruction format.
4.	Hardware is responsible for executing instructions.	4.	Microprogram is responsible for executing instructions.
5.	It is highly pipelined.	5.	It is either less pipelined or not pipelined.

ficiency of the

Basically two types of organizations are possible :

(i) Distributed type

(ii) Multiprocessor type.

(i) **Distributed Type**

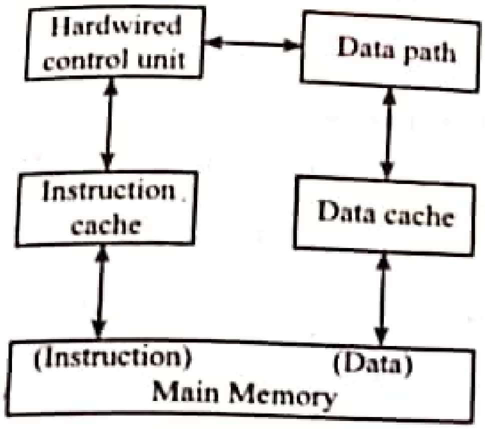
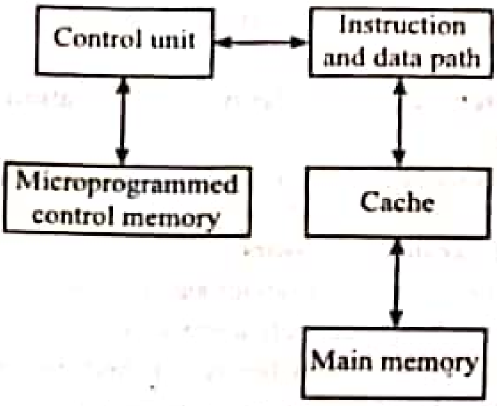
Distributed type or distributed organization c computers (processing their own sets of inst

(ii) **Multiprocessor Type**

Multiprocessor type or multiprocessor org in order to execute a single or multiple in

Consider the following organization wh

Figure: Organiz

6	The complexity exists in complier.	6.	The complexity exists in microprogram.
7	It supports few addressing modes. Usually, instructions have register to register addressing mode.	7.	It supports many addressing modes.
8	It posses multiple register set.	8.	It posses only single register set.
9	Memory is referenced by less number of instructions.	9.	Memory is referenced by more number of instructions.
10.	The RISC architecture is as shown below,  Hardwired control unit Data path Instruction cache Data cache (Instruction) (Data) Main Memory **Figure: RISC Architecture**	10.	The CISC architecture is as shown below,  Control unit Instruction and data path Microprogrammed control memory Cache Main memory **Figure: CISC Architecture**

5.2 PIPELINE AND VECTOR PROCESSING

is granted to it.

Q41. Discuss various dynamic arbitration techniques.

Answer : **Model Paper-II, Q10(b)**

Dynamic Arbitration Techniques

The dynamic priority allocation is a mechanism that has the ability to change the priorities of the processors while the system is in operation. Following are the various dynamic arbitration techniques.

1. **Polling Technique**

   Polling technique, also known as busy/waiting, is one in which the processors repeatedly checks the address generated by the controller to see if the generated address belongs to it or to others. If the address matches, then it activates the busy line to inform other processors that the bus is being used. If there are more pending requests to be processed, the controller generates the next address and the process is repeated. The following figure shows the mechanism of polling technique.

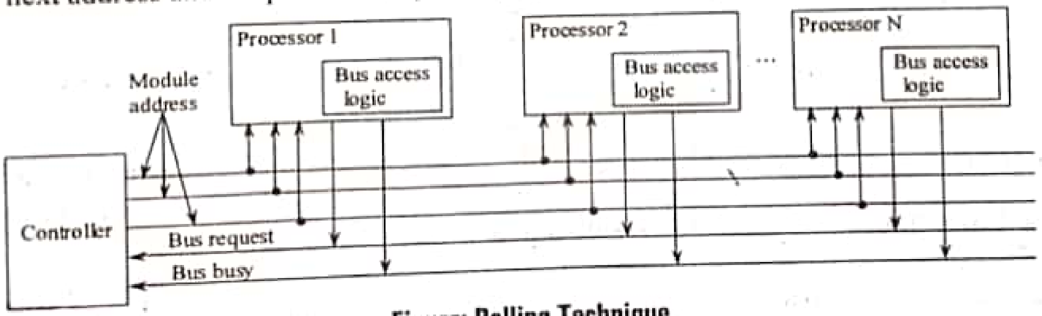


Figure: Polling Technique

2. **Time Slice Technique**

   In this scheme, each processor gains access to the bus for a fixed amount of time in a round-robin fashion. This technique is fairly unbiased, since allocated time for each processor is same, no preference is given to any particular device.

3. **LRU Technique**

   LRU stands for Least Recently Used. In this scheme, a device which has not used the bus for a longer period of time is given the highest priority. Here, the priorities are assigned dynamically. Each device is given a chance to access the bus due to dynamically varying priorities.

4. **FIFO Technique**

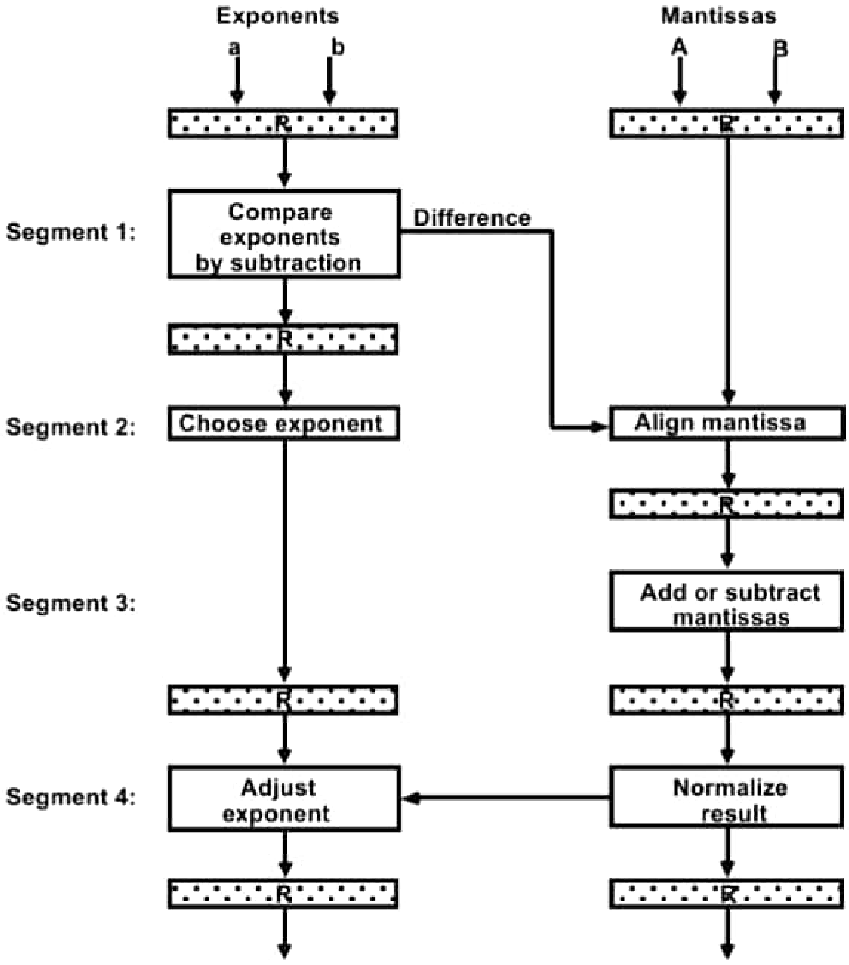
   FIFO stands for First-in-First-out. In this technique, the processor requests are served in the order in which they are received. That is, the first request is served first and the last request is served last. To implement this technique, the bus controller maintains a queue of processors requests.

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



following sections.

9-3 Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

A and B are two fractions that represent the mantissas and a and b the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6. The registers labeled R are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the suboperations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain 3 − 2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

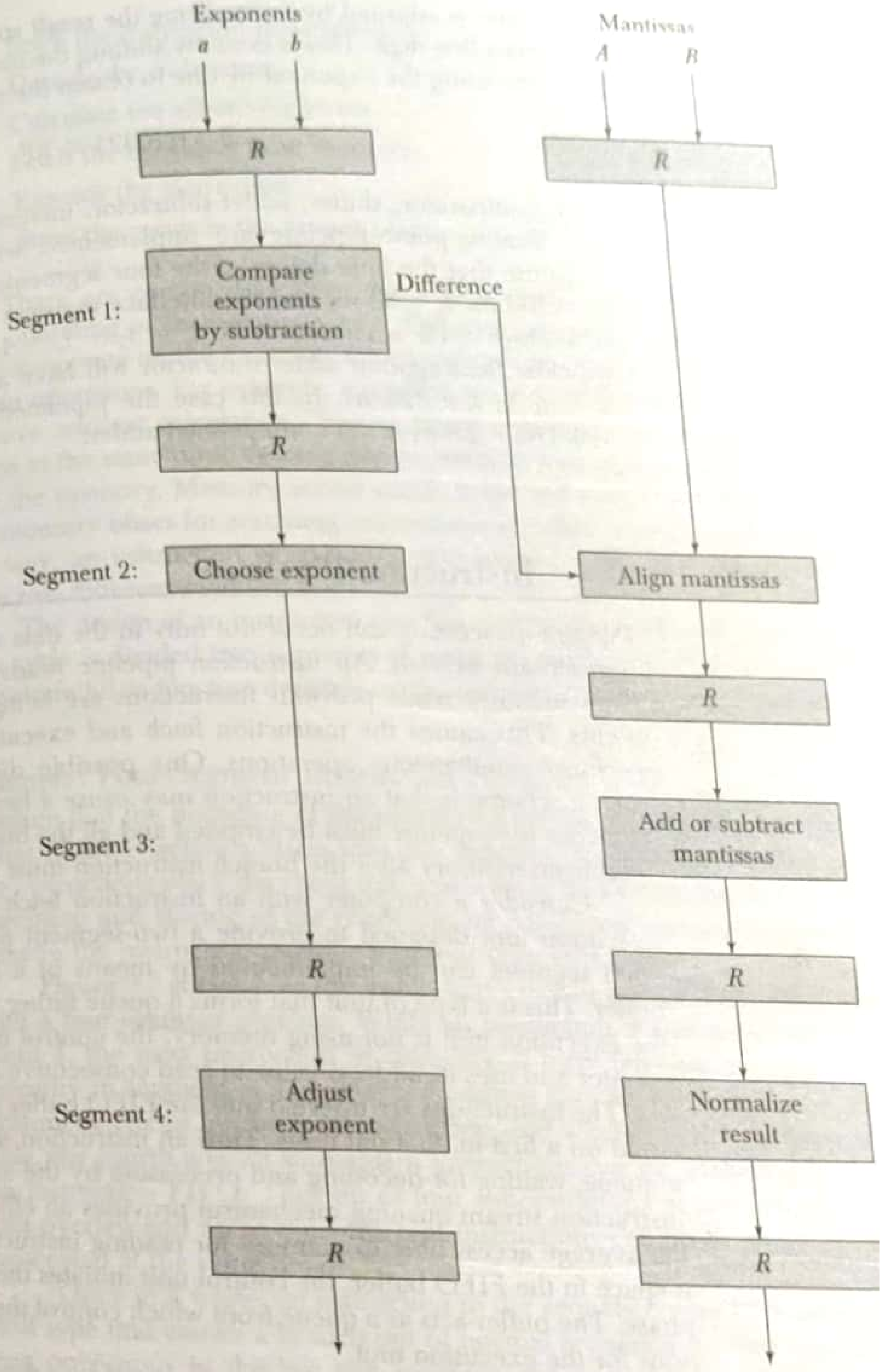


Figure 9-6 Pipeline for floating-point addition and subtraction.

The sum is adjusted by normalizing the result so that it has a fraction
nonzero first digit. This is done by shifting the mantissa once to the ri
incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decremen
the floating-point pipeline are implemented with combinational c
Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 =$
$t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r =$
The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent
pipeline floatingpoint adder-subtractor will have a delay time $t_n = t_1 +$
$t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speed
320/110 = 2.9 over the nonpipelined adder.