Module: 3	**ARITHMETIC ALGORITHMS** - Algorithms for multiplication and division (restoring method) of binary numbers — Array multiplier —Booth's multiplication algorithm **Pipelining –** Basic Principles, classification of pipeline processors. instruction and arithmetic pipelines (Design examples not required), hazard detection and resolution.

MULTIPLICATION OF UNSIGNED NUMBERS

Product of 2 n bit numbers is atmost 2n bit number. Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1 the partial product is the multiplicand. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.

Multiplication of two integer numbers 13 and 11 is,

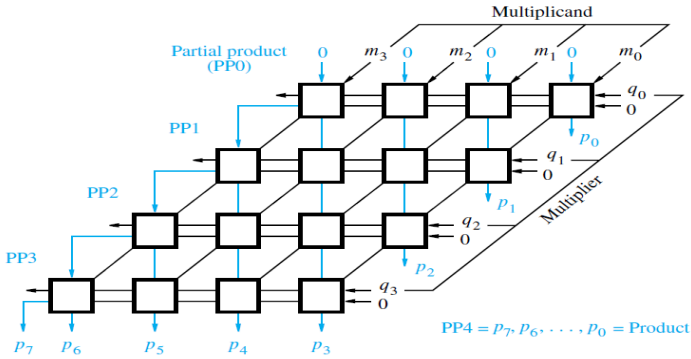
```
          1 1 0 1      (13) Multiplicand M
        × 1 0 1 1      (11) Multiplier Q
        ---------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
  ---------------------
  1 0 0 0 1 1 1 1      (143) Product P
```

(a) Manual multiplication algorithm

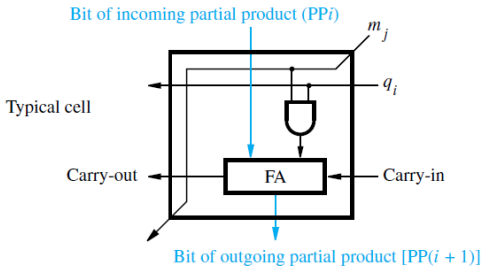
Array Multiplier

Binary multiplication can be implemented in a combinational two-dimensional logic array called **array multiplier.**

- The main component in each in each cell is a full adder, FA.
- The AND gate in each cell determines whether a multiplicand bit m_j, is added to the incoming partial product bit based on the value of the multiplier bit, q_i.
- Each row i, where 0<= i <=3, adds the multiplicand (appropriately shifted) to the incoming parcel product, PP_i, to generate the outgoing partial product, PP(i+1), if q_i.=1.
- If q_i.=0, PP_i is passed vertically downward unchanged. PP0 is all 0's and PP4 is the desired product. The multiplication is shifted left one position per row by the diagonal signal path.



(a)Array multiplication of positive binary operands



(b) Multiplier cell

Disadvantages:

(1) An n bit by n bit array multiplier requires n^2 AND gates and n(n-2) full adders and n half adders.(Half aders are used if there are 2 inputs and full adder used if there are 3 inputs).

(2) The longest part of input to output through n adders in top row, n -1 adders in the bottom row and n-3 adders in middle row. The longest in a circuit is called **critical path.**

Sequential Circuit Multiplier

Multiplication is performed as a series of (n) conditional addition and shift operation such that if the given bit of the multiplier is 0 then only a shift operation is performed, while if the given bit of the multiplier is 1 then addition of the partial products and a shift operation are performed.

The combinational array multiplier uses a large number of logic gates for multiplying numbers. Multiplication of two n-bit numbers can also be performed in a sequential circuit that uses a single n bit adder.

The block diagram in Figure shows the **hardware arrangement for sequential multiplication**. This circuit performs multiplication by using single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders in Figure. Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PPi while multiplier bit qi generates the signal Add/Noadd. This signal causes the multiplexer MUX to select 0 when qi = 0, or to select the multiplicand M when qi = 1, to be added to PPi to generate PP(i + 1). The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP0, of n 0s in register A. The carryout from the adder is stored in flipflop C, shown at the left end of the register C.

Algorithm:

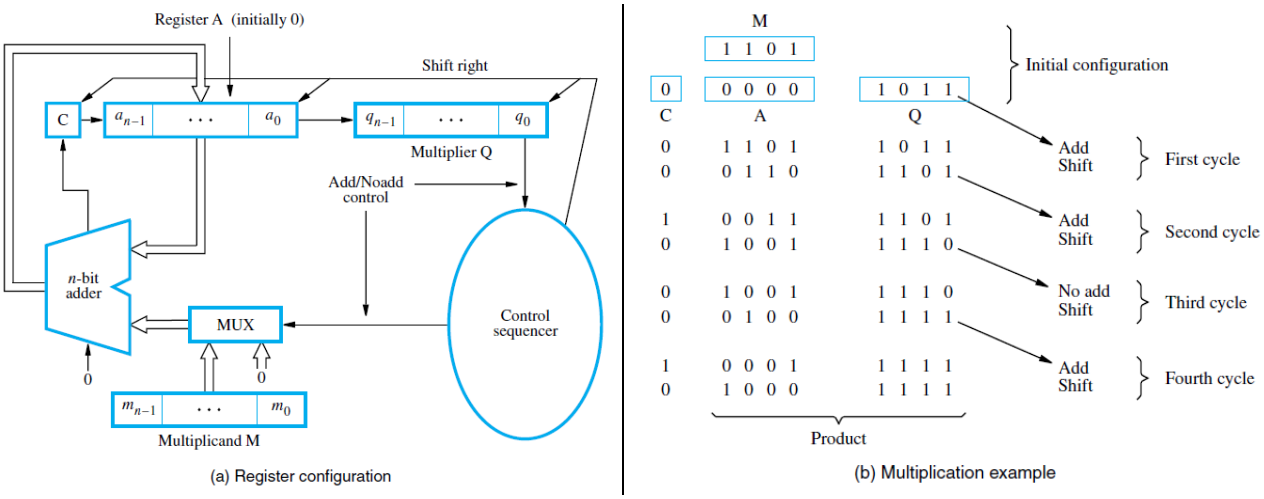
(1) The multiplier and multiplicand are loaded into two registers Q and M. Third register A and C are cleared to 0.

(2) In each cycle it performs 2 steps:

(a) If LSB of the multiplier q_i =1, control sequencer generates **Add** signal which adds the multiplicand M with the register A and the result is stored in A.

(b) If q_i =0, it generates **Noadd** signal to restore the previous value in register A.

(3) Right shift the registers C, A and Q by 1 bit



MULTIPLICATION OF SIGNED NUMBERS

We now discuss multiplication of 2's-complement operands, generating a **double-length** product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.

First, consider the case of a **positive multiplier and a negative multiplicand**. When we add a negative multiplicand to a partial product, we must **extend the sign-bit value** of the multiplicand to the **left** as far as the product will extend. Figure shows an example in which a 5-bit signed operand, −13, is the multiplicand. It is multiplied by +11 to get the 10-bit product, −143. The sign extension of the multiplicand is shown in blue. The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products.

```
 13   –> 1101
+13  -> 01101       [for +ve number, add 0 to MSB]
-13  -> 10010 +     [for –ve number, find 2's complement]
             1
          ----------
          10011   -> -13
```

[**To extend the sign bit** - since its 5 bit signed operand, 10 bit product should be generated. So, if the partial product's MSB is 1, add 1 for sign extension (to left),
if the partial product's MSB is 0, add 0 for sign extension (to left)]

Example: Sign extension of negative multiplicand

					1	0	0	1	1	(–13)
Sign extension is shown in blue				×	0	1	0	1	1	(+11)
1	1	1	1	1	1	0	0	1	1	
1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	0	1	1	1	0	0	0	1	(–143)

[product is10 bits ->(2n)]

For a **negative multiplier**, a straightforward solution is to form the **2's-complement of both the multiplier and the multiplicand** and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product.

[If the sign bit is 0 then the number is positive, If the sign bit is 1, then the number is negative]

The Booth Algorithm

Algorithm & Flowchart for Booth Multiplication

1. Multiplicand is placed in BR and Multiplier in QR
2. Accumulator register AC, Q_{n+1} are initialized to 0
3. Sequence counter SC is initialized to n (number of bits).
4. Compare Q_n and Q_{n+1} and perform the following

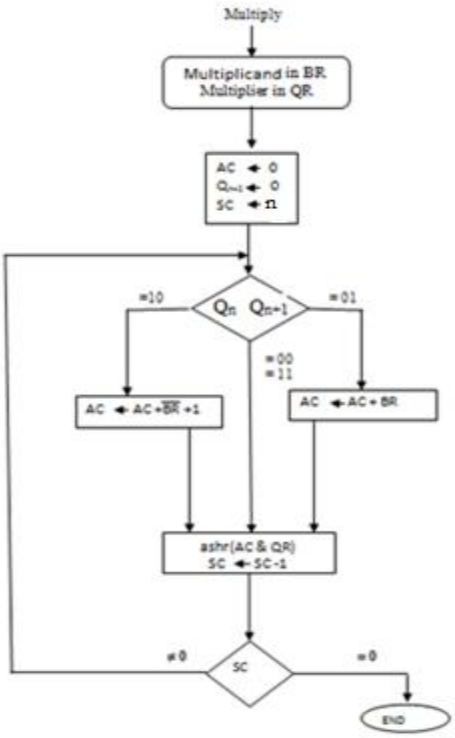
   01 –> AC=AC+BR

   10 –> AC=AC+BR'+1

   00 –> No arithmetic operation

   11-> No arithmetic operation
5. ASHR- Arithmetic Shift right AC,QR
6. Decrement SC by 1

The final product will be store in AC, QR



Multiply -9 x -13 using Booth Algorithm

9 = 1001
+9 = 01001
-9 = 10110+
1

10111 (BR)

13 = 1101
+13 = 01101
-13 = 100010+
1

10011 (Q)

BR= 10111
BR'+1= 01000+
1

01001 (BR'+1)

Qn	Q_{n+1}	BR=10111 BR'+1=01001	AC	Q	Q_{n+1}	SC
		Initial	00000	10011	0	101
1	0	SUB	00000+			
			01001			
			01001	10011	0	101
		ASHR	00100	11001	1	100
1	1	ASHR	00010	01100	1	011
0	1	ADD	00010+			
			10111			
			11001	01100	1	011
		ASHR	11100	10110	0	010
0	0	ASHR	11110	01011	0	001
1	0	SUB	11110			
			01001			
			00111	01011	0	001
		ASHR	00011	10101	1	000

Resultant Product in A and Q = 00011 10101
$= 2^6+2^5+2^4+2^2+2^0$
= 117
==========

Multiply 13 x -6 using Booth Algorithm

13 = 1101
+13 = **01101 (BR)**

6 = 0110
+6 = 00110
-6 = 11001+
1

11010 (Q)

BR= 01101
BR'+1= 10010+
1

10011 (BR'+1)

Qn	Q_{n+1}	BR=01101 BR'+1=10011	AC	Q	Q_{n+1}	SC
		Initial	00000	11010	0	101
0	0	ASHR	00000	01101	0	100
1	0	SUB	00000+			
			10011			
			10011	01101	0	100
		ASHR	11001	10110	1	011
0	1	ADD	11001+			
			01101			
			00110	10110	1	011
		ASHR	00011	01011	0	010
1	0	SUB	00011			
			10011			
			10110	01011	0	010
		ASHR	11011	00101	1	001
1	1	ASHR	11101	10010	1	000

[13x-6 will give a –ve product. so the resultant product's 2's compliment should be determined]

Resultant Product in A and Q = 11101 10010

2's complement = 00010 01101+
1

0001001110
$=2^6+2^3+2^2+2^1$
. **= -78**
==========

Multiply -11 x 8 using Booth Algorithm

11 = 1011
+11 = 01011
-11 = 10100+
1

10101 (BR)

8 = 1000
+8 = **01000 (Q)**
-

BR= 10101
BR'+1= 01010+
1

01011 (BR'+1)

Qn	Q_{n+1}	**BR=10101** **BR'+1=01011**	AC	Q	Q_{n+1}	SC
		Initial	00000	0100**0**	**0**	101
0	0	ASHR	**0**0000	0010**0**	**0**	100
0	0	ASHR	**0**0000	0001**0**	**0**	011
0	0	ASHR	**0**0000	0000**1**	**0**	010
1	0	SUB	00000+			
			01011			
			01011	00001	0	010
		ASHR	**0**0101	1000**0**	**1**	001
0	1	ADD	00101			
			10101			
			11010	10000	1	010
		ASHR	**1**1101	01000	0	000

[-11x8 will give a –ve product. so the resultant product's 2's compliment should be determined]

Resultant Product in A and Q = 11101 01000

2's complement = 00010 10111+
1

0001011000
$=2^6+2^4+2^3$
= -88
=========

Multiply each of the following pairs of signed 2's complement number using Booth's algorithm. In each of the cases assume A is the multiplicand and B is the multiplier.
A=010111 B=110110

Answer:

A=**0**10111
[sign bit is 0, therefore +ve number]

A=23 [10111]

Therefore, A=+23 [010111]

B=**1**10110
[sign bit is 1, therefore -ve number]
Find 2's complement.
2's complement of 10110 is 01001+ 1
= 01010 => 10

Therefore, B= -10 [110110]

Multiply +23 x -10

+23 = **010111 (BR)**
-10 = **110110 (Q)**

BR= 010111
BR'+1= 101000+
1

101001 (BR'+1)

Qn	Qn+1	BR=010111 BR'+1=101001	AC	Q	Qn+1	SC
		Initial	000000	110110	0	0110
0	0	ASHR	000000	011011	0	0101
1	0	SUB	000000+			
			101001			
			101001	011011	0	0101
		ASHR	110100	101101	1	0100
1	1	ASHR	111010	010110	1	0011
0	1	ADD	111010+			
			010111			
			010001	010110	1	0011
		ASHR	001000	101011	0	0010
1	0	SUB	001000+			
			101001			
			110001	101011	0	0010
		ASHR	111000	110101	1	0001
1	1	ASHR	111100	011010	1	0000

[+23x-10 will give a –ve product. so the resultant product's 2's compliment should be determined]

Resultant Product in A and Q = 111100 011010

2's complement = 000011 100101+

1

000011100110

$=2^7 +2^6+2^5+2^2 + 2^1$

= -230

==========

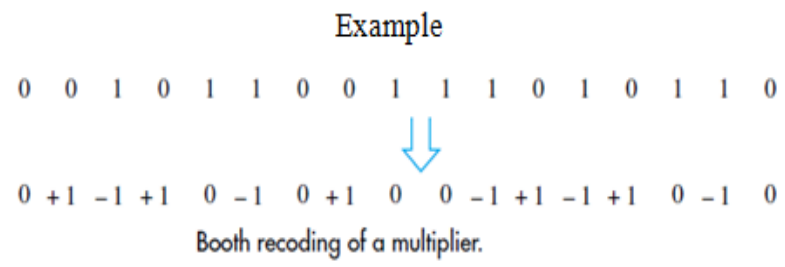
Features of Booth Algorithm:

- Booth algorithm works equally well for both negative and positive multipliers.
- Booth algorithm deals with signed multiplication of given number.
- Speed up the multiplication process.

Booth Recording of a Multiplier:

In general, in the Booth algorithm, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from1 to 0, as the multiplier is scanned from right to left. The case when the LSB of the multiplier is 1, it is handled by assuming that an implied 0 lies to its right.

Multiplier		Version of multiplicand selected by bit *i*
Bit *i*	Bit *i* – 1	
0	0	0 × M
0	1	+1 × M
1	0	–1 × M
1	1	0 × M



Booth recoding of a multiplier.

- In worst case multiplier, numbers of addition and subtraction operations are large.
- In ordinary multiplier, 0 indicates no operation, but still there are addition and subtraction operations to be performed.
- In good multiplier, booth algorithm works well because majority are 0s .
- A good multiplier consists of block/sequence of 1s.

Booth recoded multipliers.

Booth algorithm achieves efficiency in the number of additions required when the multiplier had a **few large blocks of 1s.** The speed gained by skipping over 1s depends on the data. On average, the speed of doing multiplication with the booth algorithm is the same as with the normal multiplication

- Best case – a long string of 1's (skipping over 1s)
- Worst case – 0's and 1's are alternating
- The transformation 011….110 to 100….0-10 is called **skipping over 1s**.

INTEGER DIVISION

Figure shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13into 27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that 27 − 26 = 1 is less than 13, we enter 2 as the quotient and perform the required subtraction.

```
      21
13 ) 274
     26
     14
     13
      1
```

```
        10101
1101 ) 100010010
       1101
       10000
        1101
        1110
        1101
           1
```

Dividend = 274

Divisor = 13

Quotient=21

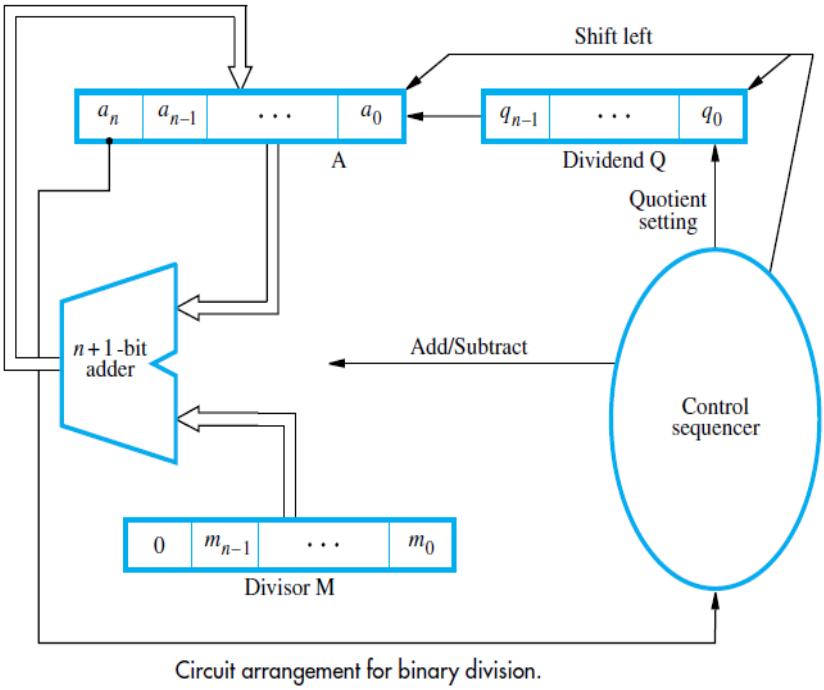
Remainder =1

The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once and the remainder is 1. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 0 and 1.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.

If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. This is called the **restoring division algorithm.**

Restoring Division



Circuit arrangement for binary division.

Figure shows a logic circuit arrangement that implements the restoring division algorithm just discussed. An n-bit positive **divisor** is loaded into register **M** and an n-bit positive **dividend** is loaded into register **Q** at the start of the operation. Register **A** is set to **0**. After the division is complete, the n-bit **quotient** is in register **Q** and the **remainder** is in register **A**.

The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following three steps n times:

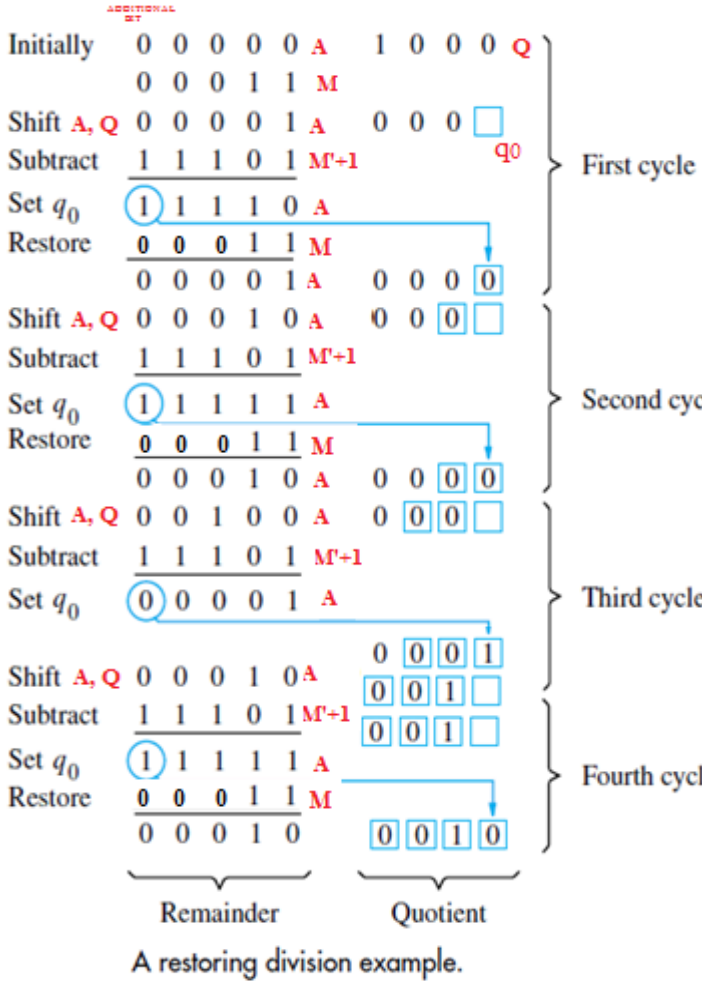
1. Shift A and Q left one bit position.
2. Subtract M from A, ie; (A-M) and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set q_0 to 1.

```
        10
   ┌──────
11 ) 1000   Q
M     11
     ──
      10
```

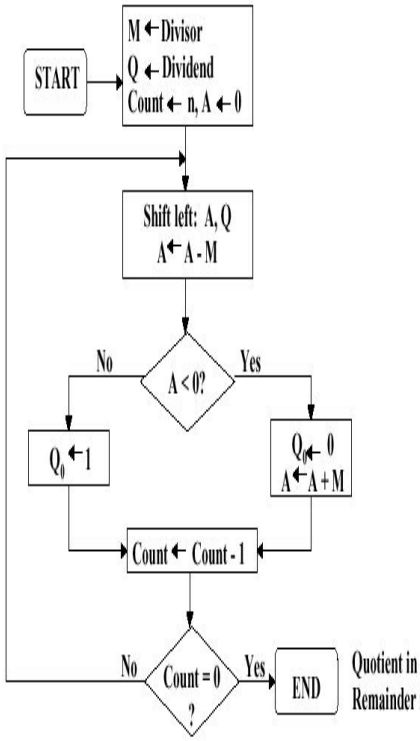
M = 00011

M'+1 = 11100+1

= 11101



A restoring division example.



Dividend=8 [1000], Divisor=3 [11]

Quotient=2 [0010], Remainder=2 [00010]

PIPELINING

Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

A pipeline processor may process each instruction in 4 steps:

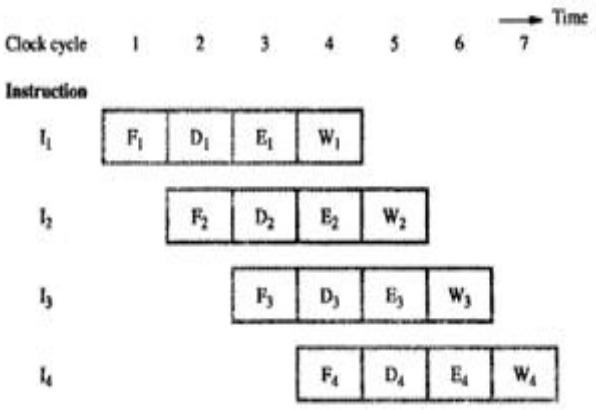
F Fetch: Read the instruction from the memory

D Decode: Decode the instruction and fetch the source operands

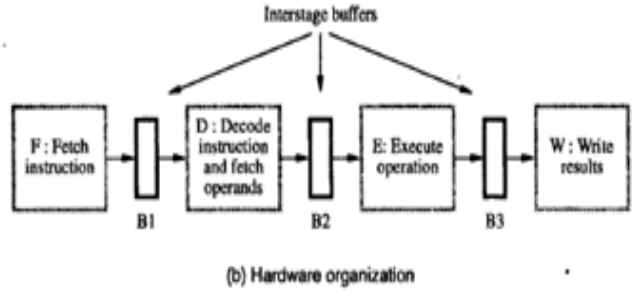
E Execute: Perform the operation specified by the instruction

W Write: Store the result in the destination location.

In figure (a) four instructions progress at any given time. This means that four distinct hardware units are needed as in figure (b). These units must be capable of performing their tasks simultaneously without interfering with one another. Information is passed from one unit to next through a storage buffer. As an instruction progresses through the pipeline. all the information needed by the stages downstream must be passed along.



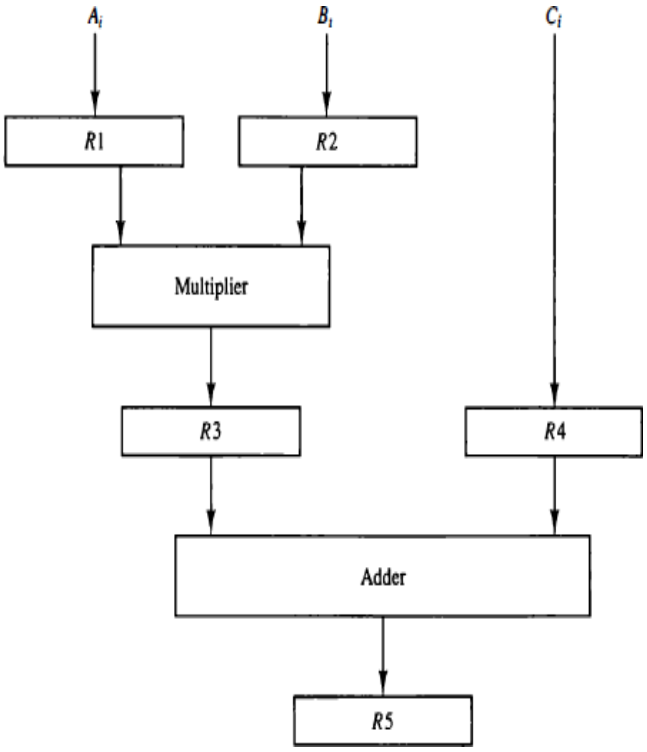
(a) Instruction execution divided into four steps



(b) Hardware organization

Pipeline Organization

The simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit.The register holds the data and the combinational circuit performs the sub operation in the particular segment. The output of the combinational circuit is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In thisway the information flows through the pipeline one step at a time. Example demonstrating the pipeline organization



Suppose we want to perform the combined multiply and add operations with astream of numbers.

Ai*Bi + Ci for i=1, 2, 3 ….7

Each sub operation is to implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in fig.

R1 through r5 are registers that receive new data with every clock pulse.

The multiplier and adder are combinational circuits.

The sub operationsperformed in each segment of the pipeline are as follows:

R1<- Ai	R2<-Bi	Input Ai and Bi
R3<-R1*R2	R4<-Ci	multiply and input Ci
R5<-R3+R4		add Ci to product

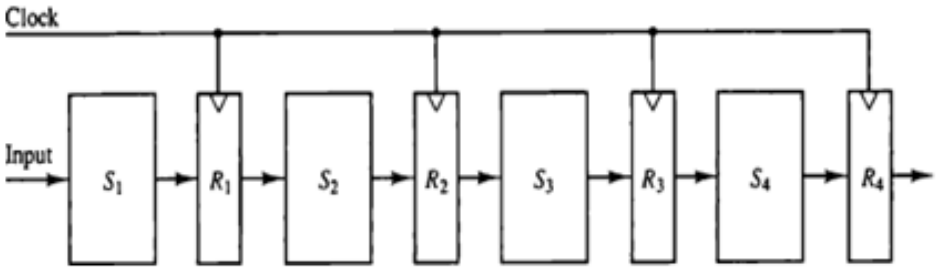
The five registers are loaded with new data every clock pulse.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

The first clock pulse transfers A1 and B1 into R1 and R2. The second clock pulse transfers the product of R1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places A3 and B3 into R1 and R2, transfers the product of R1 and R2 into R3, transfers C2 into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system.

Four Segment Pipeline

The general structure of four segment pipeline is shown in fig. the operands are passed through all four segments in affixed sequence. Each segment consists of a combinational circuit Si that performs a sub operation over the data stream flowing through the pipe. The segments are separated by registers Ri that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registerssimultaneously.

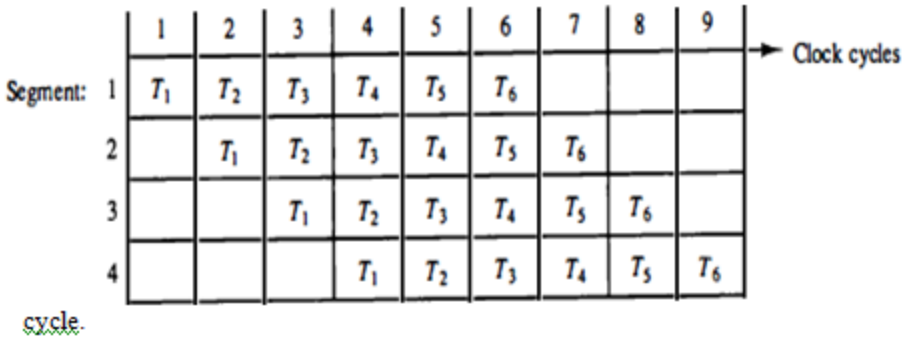


Space Time Diagram

The behavior of a pipeline can be illustrated with a space time diagram. This is a diagram that shows the segment utilization as a function of time.

Fig - The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks T1 through T6 executed in four segments. Initially, task T1 is handled by segment 1. After thefirst clock, segment 2 is busy with T1, while segment 1 is busy with task T2. Continuing in this manner, the first task T1 is completed after fourth clock cycle. From then,the pipe completes a task every clock cycle.

Consider the case where a k-segment pipeline with a clock cycle time tp is used to execute n tasks. The first task T1 requires a time equal to ktp to complete its operation since there are k segments in a pipe. The remaining n-1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n-1) tp. Therefore, to complete n tasksusing a k segment pipeline requires k+ (n-1) clock cycles.



Consider a non pipeline unit that performs the same operation and takes atime equal to tn to complete each task. The total time required for n tasks is n tn. The speedup of a pipeline processing over an equivalent non pipeline processing is defined by the ratio

S=ntn / (k+n-1)tp

As the number of tasks increases, n becomes much larger than k-1, and k+n-1 approaches the value of n. under this condition the speed up ratio becomes

S=tn/tp

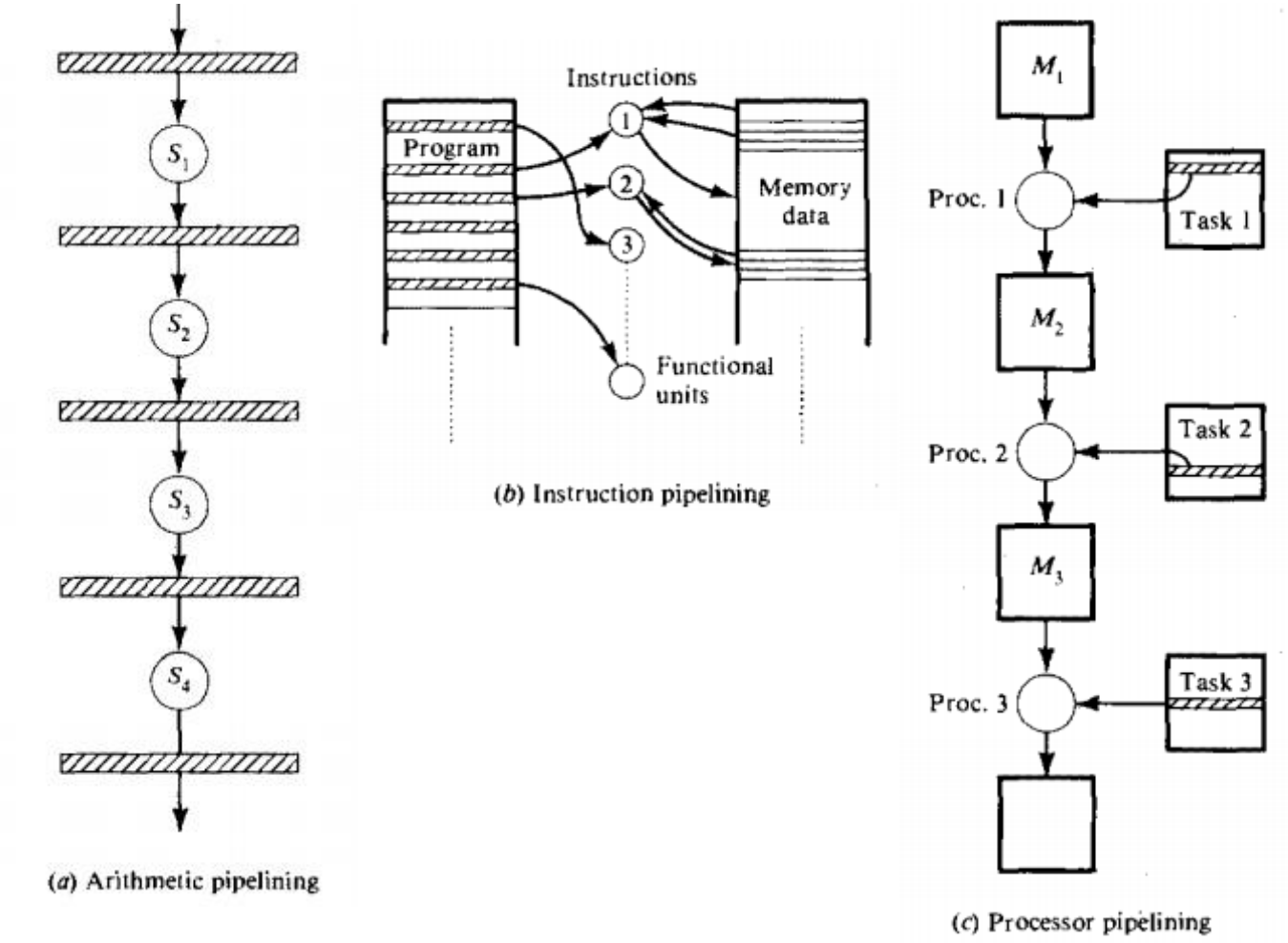
If we assume that the time it takes to process a task is the same in the pipeline and non pipeline circuits, we will have tn=ktp. Including this assumption speedup ratio reduces to

S=ktp/tp=k

CLASSIFICATION OF PIPELINE PROCESSORS

1. **Arithmetic Pipelining:** The arithmetic logic units of a computer can be segmented for pipeline operations in various data formats.
2. **Instruction Pipelining:** The execution of stream of instructions can be pipelined by overlapping the execution of current instruction with the fetch, decode and execution of subsequent instructions. This technique is known as **instruction lookahead.**

3. **Processor Pipelining**: Pipeline processing of the same data stream by a cascade of processors, each of which processes a specific task. The data stream passes the first processor with the results stored in memory block which is also accessible by the second processor. The second processor then passes the refined results to the third and so on.



(*a*) Arithmetic pipelining

(*b*) Instruction pipelining

(*c*) Processor pipelining

ARITHMETIC PIPELINES

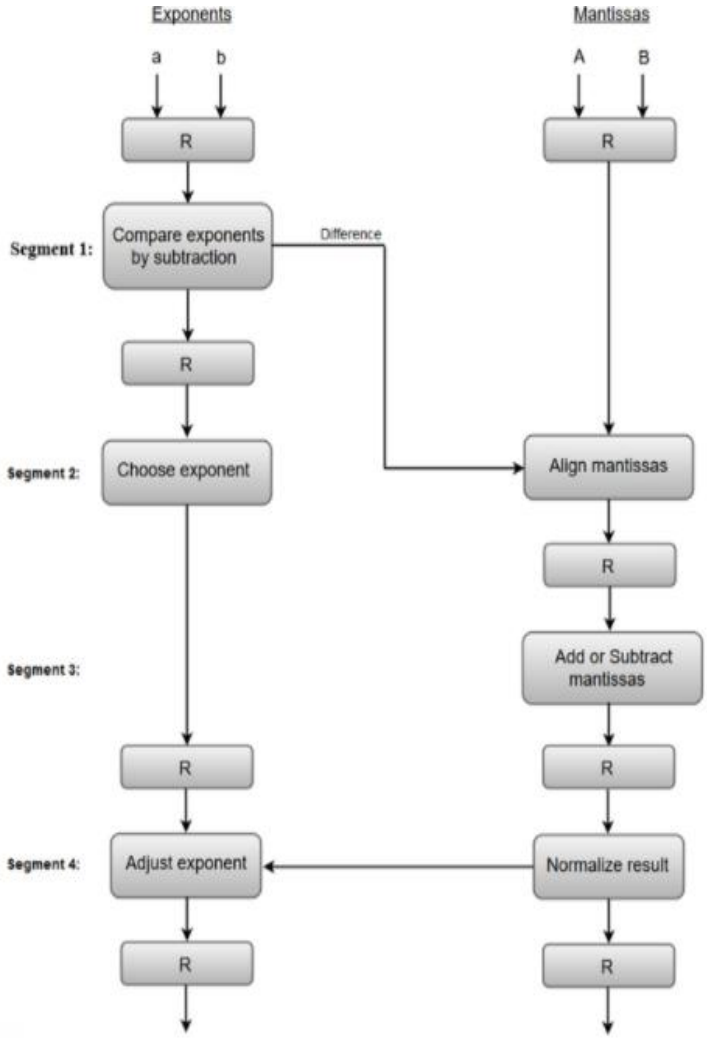
An arithmetic pipeline divides an arithmetic operation into sub operations for execution in the pipeline segments. Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations, multiplication of fixed point numbers, and similar computations encountered in scientific problems.

Pipeline Unit For Floating Point Addition And Subtraction:

The inputs to the floating point adder pipeline are two normalized floating point binary numbers. $X=A*2^a$, $Y=B*2^b$

A and B are two fractions that represent the mantissa and a and bare the exponents. The floating point addition and subtraction can be performed in four segments. The registers labeled are placed between the segments to store intermediate results. The sub operations that are performed in the four segments are:

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



Representation of floating point number: **$X=M * r^e$**

[M-mantissa, r- radix, e- exponent]

Example: $X=0.9143 * 10^3$

$y=0.5429 * 10^2$

Segment 1: Compare the exponents by subtraction

X-> 3, Y-> 2 Difference=3-2 => **1**

Segment 2: Align the Mantissa

Mantissa of smaller exponent is **shifted right** by 1

$(0.5429 * 10^2 * 10)/10 = \mathbf{0.05429 * 10^3}$

Segment 3:Add/Subtract the Mantissas

Assume Add operation

$0.9143+0.05429=\mathbf{0.96859}$

Segment 4: Normalize the result

Here the result is already normalized.

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 ofthe two mantissas.

The two mantissas are added or subtracted in segment3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted to right and the exponent incremented by one. If the underflow occurs, the number of leading zeroes in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

[**Overflow** – When the result of an Arithmetic operation is finite but larger in magnitude than the largest floating point which can be stored by the precision, **Underflow** – When the result of an Arithmetic operation is smaller in magnitude than the smallest floating point which can be stored]

INSTRUCTION PIPELINE

An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of instruction cycle. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and executes phases to overlap and perform simultaneous operations.

Consider a computer with an instruction fetch unit and an instruction execute unit designed to provide a **two segment pipeline**. The instruction fetch segment can be implemented by means of a first in first out (FIFO) buffer. Whenever the execution unit is not using memory, the control increments the program counter and uses it address value to read consecutive instructions frommemory. The instructions are inserted into the FIFO buffer so that they can be executed on a first in first out basis. Thus an instruction stream can be placed inqueue, waiting for decoding and processing by the execution segment.

Four Segment Instruction Pipeline

In general the computer needs to process each instruction with the following sequence of steps. (6 steps in 4 segments)

1. Fetch the instruction. → **FI**
2. Decode the instruction.
3. Calculate the effective address. → **DA**
4. Fetch the operands from memory. → **FO**
5. Execute the instruction.
6. Store the result in the proper place. → **EX**

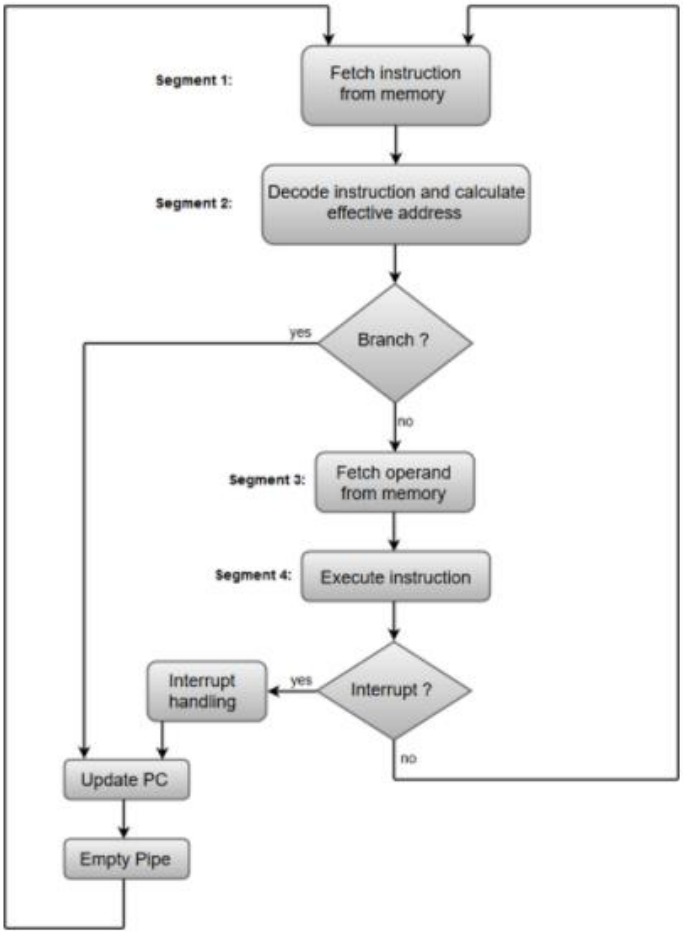


Fig shows the operation of the instruction pipeline. The clock in thehorizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

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

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates theeffective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

Here the instruction is fetched **(FI)** on first clock cycle in segment 1. it is decoded **(DA)** on second clock cycle , the operands are fetched **(FO)** on third clock cycle and finally the instruction is executed **(EX)** in the fourth cycle. Here the fetch and decode phase overlap due to **pipelining**. By the time the first instruction is being decoded, next instruction is fetched by the pipeline.

In case of third instruction we see that it is a **branched** instruction. Here when it is being decoded, 4th instruction is fetched simultaneously. But as it is a branched instruction it may point to some other instruction when it is decoded. Thus fourth instruction is **kept on hold** until the branched instruction is executed. When it gets executed then the fourth instruction is

copied back and the other phases continue as usual. In the absence of a branch instruction, each segment operates on different instructions.

PIPELINE CONFLICTS:

1. **Resource Conflicts**: They are caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

2. **Data Dependency**: these conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

3. **Branch Difference**: they arise from branch and other instructions that change the value of PC.

PIPELINE HAZARDS DETECTION AND RESOLUTION

Pipeline hazards are caused by resource usage conflicts among various instructions in the pipeline. Such hazards are triggered by inter instruction dependencies when successive instructions overlap their fetch, decode and execution through a pipeline processor, inter instruction dependencies may arise to prevent the sequential data flow in the pipeline.

For example an instruction may depend on the results of a previous instruction. Until the completion of the previous instruction, the present instruction cannot be initiated into the pipeline. In other instances, two stages of a pipeline may need to update the same memory location. Hazards of this sort, if not properly detected and resolved could result in an **inter lock** situation in the pipeline or produce unreliable results by overwriting.

There are **three classes of data dependent hazards**, according to various data update patterns:

1. Write After Read hazards (WAR)
2. Read After Write hazards (RAW)
3. Write After Write hazards (WAW)

Note that **Read After Read** does not pose a problem because nothing is changed.

We use resource object to refer to working registers, memory locations and special flags. The contents of these resource objects are called **data objects**. Each instruction can be considered a mapping from a set of data objects to a set of data objects. The domain **D(I)** of an instruction I is a set of resource objects whose data objects may affect the execution of instruction I. The range of an instruction **R(I)** is the set of resource objects whose data objects may be modified by the execution of instruction I. Obviously, the operands to be used in an instruction execution are retrieved (read) from its domain and the results will be stored (written) in its range.

Consider the execution of two instructions I and J in a program. Instruction J appears after instruction I in the program. There may be none or other instructions between instruction I and J. The latency between the two instructions is a very subtitle matter. Instruction J may enter the execution pipe before or after the completion of the execution of instruction l. The improper timing and the data dependencies may create some other hazardous situations.

1. **RAW** hazard between the two instructions I and J may occur when they attempt to read some data object that has been modified by I.
2. **WAR** hazard may occur when J attempt to modify some data object that is read by I.
3. **WAW** hazard may occur if both I and J attempt to modify the same data object.

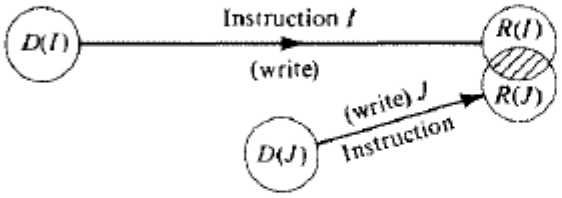
The necessary conditions for these hazards are stated as follows:

$$\begin{aligned} R(I) \cap D(J) &\neq \phi \quad \text{for RAW} \\ R(I) \cap R(J) &\neq \phi \quad \text{for WAW} \\ D(I) \cap R(J) &\neq \phi \quad \text{for WAR} \end{aligned} \tag{3.18}$$

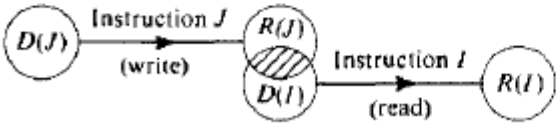
Possible hazards are listed in table. Recognizing the existence of possible hazards, computer designers wish to detect the hazard and then to resolve it efficiently. Hazard detection can be done in the instruction fetch stage of a pipeline processor by comparing the domain and the range of incoming instruction with those of the instructions being processed in the pipe. Should any of the condition in equation 3.18 be detected, a warning signal can be generated to prevent the hazard from taking place. Another approach is to allow the incoming instruction through the pipe and distribute the detection to all the potential pipeline stages.

This distributed approach offers better flexibility at the expense of increased hardware control. Note that the necessary conditions in the equation 3.18 may not be sufficient conditions.

(a) RAW hazard



(b) WAW hazard



(c) WAR hazard

Figure 3.36 Illustration of RAW, WAW, and WAR hazard conditions.

Table 3.3 Possible hazards for various instruction types

	Instruction *I* (first)			
Instruction *J* (second)	Arithmetic and load type	Store type	Branch type	Conditional branch type
Arithmetic and load type	RAW WAW WAR	RAW WAR	WAR	WAR
Store type	RAW WAR	WAW		
Branch type	RAW		WAW	WAW
Conditional branch type	RAW		WAW	WAW

Once the hazard is detected, the system should resolve the interlock situation. Consider the instruction sequence {.. I, I+1,....J, J+1,...} in which a hazard has been detected between the

current instructions J and a previous instruction I. A straightforward approach is to stop the pipe and to suspend the execution of the instructions J, J+ 1, J+2....until instruction I has passed the point of resource conflict. A more sophisticated approach is to suspend only instruction J and continue the flow of instructions is J+1, J+2,... down the pipe. Of course, the potential hazards due to the suspension of J should be continuously checked as instructions J+1, J+2 to move ahead of J. Multi level hazard detection may be encountered, requiring more complex control mechanisms to resolve a stack of hazards

In order to avoid RAW hazards, IBM engineers developed a **short circuiting approach** which gives a copy of the data object to be written directly to the instruction waiting to read the data. This concept was generalized into a technique known as **data forwarding**, which forward multiple copies of the data to as many waiting instructions as may wish to read it. A data forwarding chain can be established in some cases. The internal forwarding and register-tagging techniques are helpful in resolving logic hazards in pipelines.