



# **COMPUTER ORGANIZATION AND DESIGN**

---

**Mahesh Awati**

Department of Electronics and Communication Engg.

# COMPUTER ORGANIZATION AND DESIGN

---

## Arithmetic for Computers

### 3.2

### Addition and Subtraction

**Mahesh Awati**

Department of Electronics and Communication Engineering

# Computer Organization and Design

## Range of Signed & Unsigned Numbers

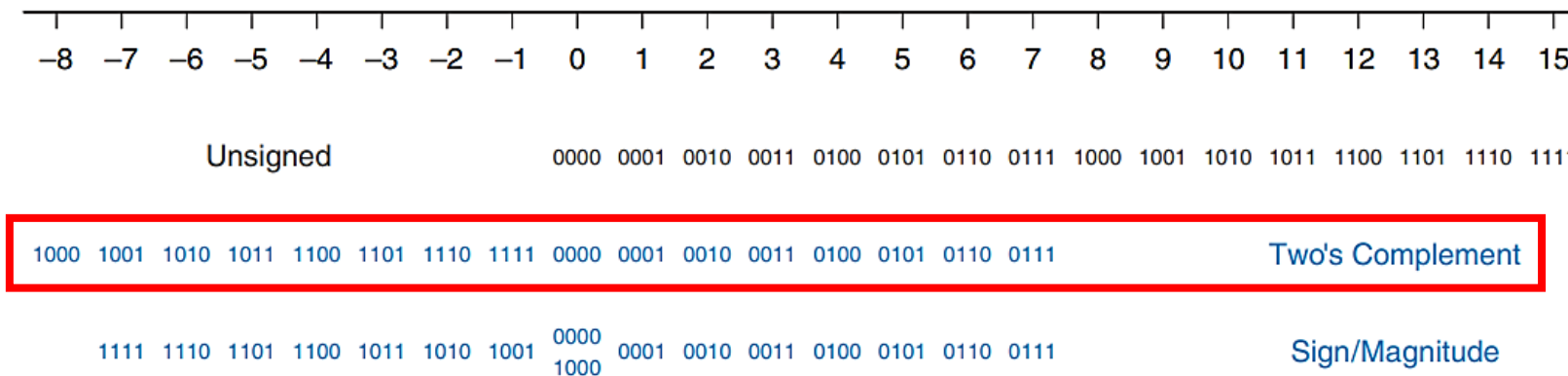
**Table 1.3** Range of  $N$ -bit numbers

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

The negative numbers are stored in two's complement form

The range for a 32 bit signed number is

**-2147483648 to 2147483647**



**Figure 1.11** Number line and 4-bit binary encodings

# Computer Organization and Design

## Range of Signed & Unsigned Numbers

The negative numbers are stored in two's complement form

The range for a 32 bit signed number is

**-2147483648 to 2147483647**

Decimal Value (max: 9223372036854775807)	Binary Value
<input type="text" value="2147483647"/> <input type="checkbox"/> Padding <input type="button" value="Convert"/>	<input type="text" value="01111111111111111111111111111111"/>
swap conversion: <a href="#">Binary To Decimal Converter</a>	

Decimal Value (max: 9223372036854775807)	Hexadecimal Value
<input type="text" value="2147483647"/> <input type="button" value="Convert"/>	<input type="text" value="7FFFFFFF"/>
swap conversion: <a href="#">Hex to Decimal</a>	

Decimal Value (max: 9223372036854775807)	Binary Value
<input type="text" value="-2147483648"/> <input type="checkbox"/> Padding <input type="button" value="Convert"/>	<input type="text" value="10000000000000000000000000000000"/>
swap conversion: <a href="#">Binary To Decimal Converter</a>	

Decimal Value (max: 9223372036854775807)	Hexadecimal Value
<input type="text" value="-2147483648"/> <input type="button" value="Convert"/>	<input type="text" value="80000000"/>
swap conversion: <a href="#">Hex to Decimal</a>	

### Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

Compute (a)  $-2_{10} + 1_{10}$  and (b)  $-7_{10} + 7_{10}$  using two's complement numbers.

**Solution:** (a)  $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$ . (b)  $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$ . The fifth bit is discarded, leaving the correct 4-bit result  $0000_2$ .

### Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

Compute (a)  $5_{10} - 3_{10}$  and (b)  $3_{10} - 5_{10}$  using 4-bit two's complement numbers.

**Solution:** (a)  $3_{10} = 0011_2$ . Take its two's complement to obtain  $-3_{10} = 1101_2$ . Now add  $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$ . Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of  $5_{10}$  to obtain  $-5_{10} = 1011$ . Now add  $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$ .

---

### **Example 1.14** ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

Compute  $4_{10} + 5_{10}$  using 4-bit two's complement numbers. Does the result overflow?

**Solution:**  $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$ . The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result  $01001_2 = 9_{10}$  would have been correct.

---

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	$\geq 0$	$\geq 0$	$< 0$
$A + B$	$< 0$	$< 0$	$\geq 0$
$A - B$	$\geq 0$	$< 0$	$< 0$
$A - B$	$< 0$	$\geq 0$	$\geq 0$

**FIGURE 3.2** Overflow conditions for addition and subtraction.

### Observations:

- ✓ Overflow occurs when adding two positive numbers and the sum is negative, or vice versa.
- ✓ **Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result.**
- ✓ **No overflow can occur when adding positive and negative operands.**

### What about overflow with unsigned integers?

- ✓ Unsigned integers are commonly used for memory addresses where overflows are ignored.
- ✓ Fortunately, the compiler can easily check for unsigned overflow using a branch instruction.
- ✓ **Addition has overflowed if the sum is less than either of the addends, whereas subtraction has overflowed if the difference is greater than the minuend.**

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas **RISC-V only has integer arithmetic operations on full words.**

**What RISC-V instructions should be generated for byte and half-word arithmetic operations?**



Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas **RISC-V only has integer arithmetic operations on full words.**

**What RISC-V instructions should be generated for byte and half-word arithmetic operations?**

1. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`; then store using `sb`, `sh`.

### Saturation Arithmetic:

One feature not generally found in general-purpose microprocessors is saturating operations.

**Saturation means that when a calculation overflows, the result is set to the largest positive number or the most negative number,** rather than a modulo calculation as in two's complement arithmetic.

# COMPUTER ORGANIZATION AND DESIGN

---

## Arithmetic for Computers

### 3.3

### **Multiplication**

**Mahesh Awati**

Department of Electronics and Communication Engineering

### Multiplier : For n=m=4

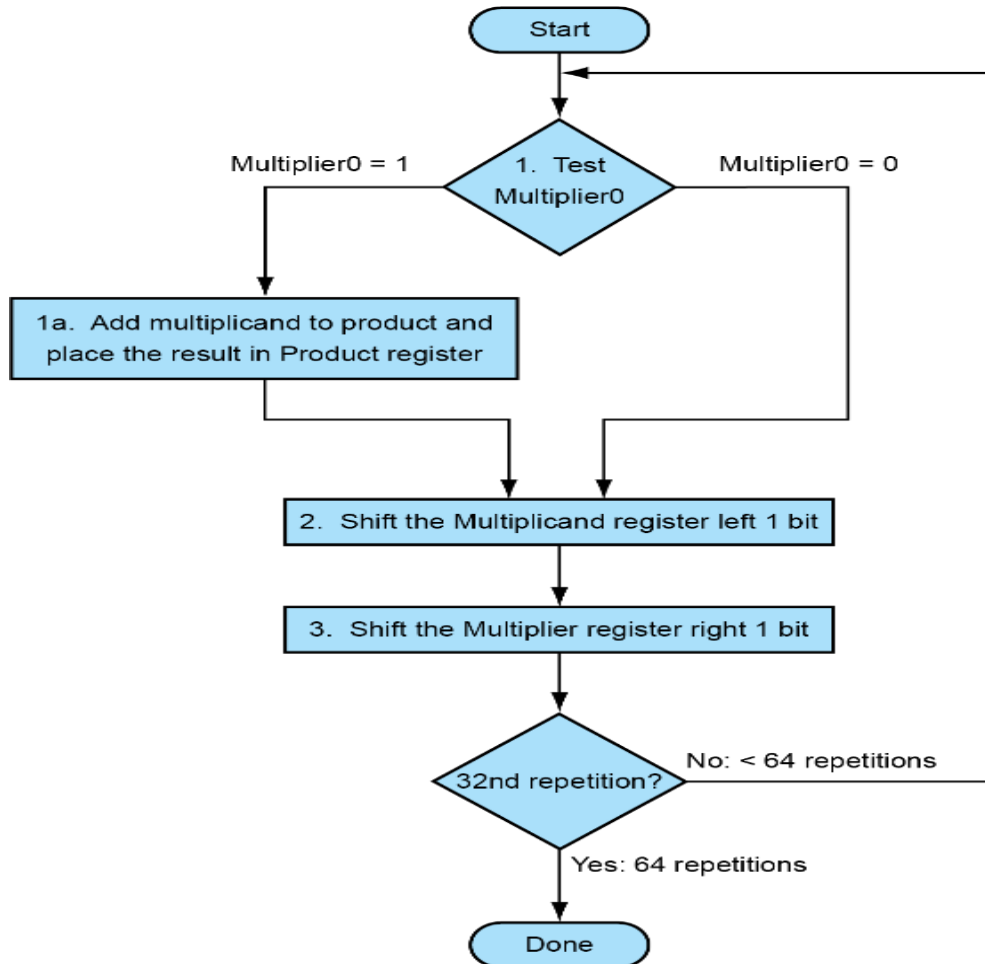
			X	m3 q3	m2 q2	m1 q1	m0 q0	- Multiplicand - Multiplier
j=0				m3q0	m2q0	m1q0	m0q0	Partial Product
j=1		+	m3q1	m2q1	m1q1	m0q1		
j=2		m3q2	m2q2	m1q2	m0q2			
j=3	m3q3	m2q3	m1q3	m0q3				
P7	P6	P5	P4	P3	P2	P1	P0	Product

1 1 0 1	(13) Multiplicand M
x 1 0 1 1	(11) Multiplier Q
1 1 0 1	
0 0 0 0	
1 1 0 1	
1 0 0 0 1 1 1 1	(143) Product P

## MULTIPLICATION : Sequential Circuit Multiplier

Iteration	Multiplier	Steps	Multiplicand	Product
0	001 <b>1</b>	-	0000 0010	0000 0000
1		a) Test bit =1; Product= Product + Multiplicand		0000 0010
		b) Shift left Multiplicand	0000 0100	
	000 <b>1</b>	c) Shift Right Multiplier		
2		a) Test bit =1; Product= Product + Multiplicand	0000 0100	0000 0110
		b) Shift left Multiplicand	0000 1000	
	000 <b>0</b>	c) Shift Right Multiplier		
3		a) Test bit =0; Product= Product ( No operation)	0000 0000	0000 0110
		b) Shift left Multiplicand	0001 0000	
	000 <b>0</b>	c) Shift Right Multiplier		
4		a) Test bit =0; Product= Product ( No operation)	0001 0000	0000 0110
		b) Shift left Multiplicand	0010 0000	
	000 0	c) Shift Right Multiplier		

### First version of the multiplication hardware



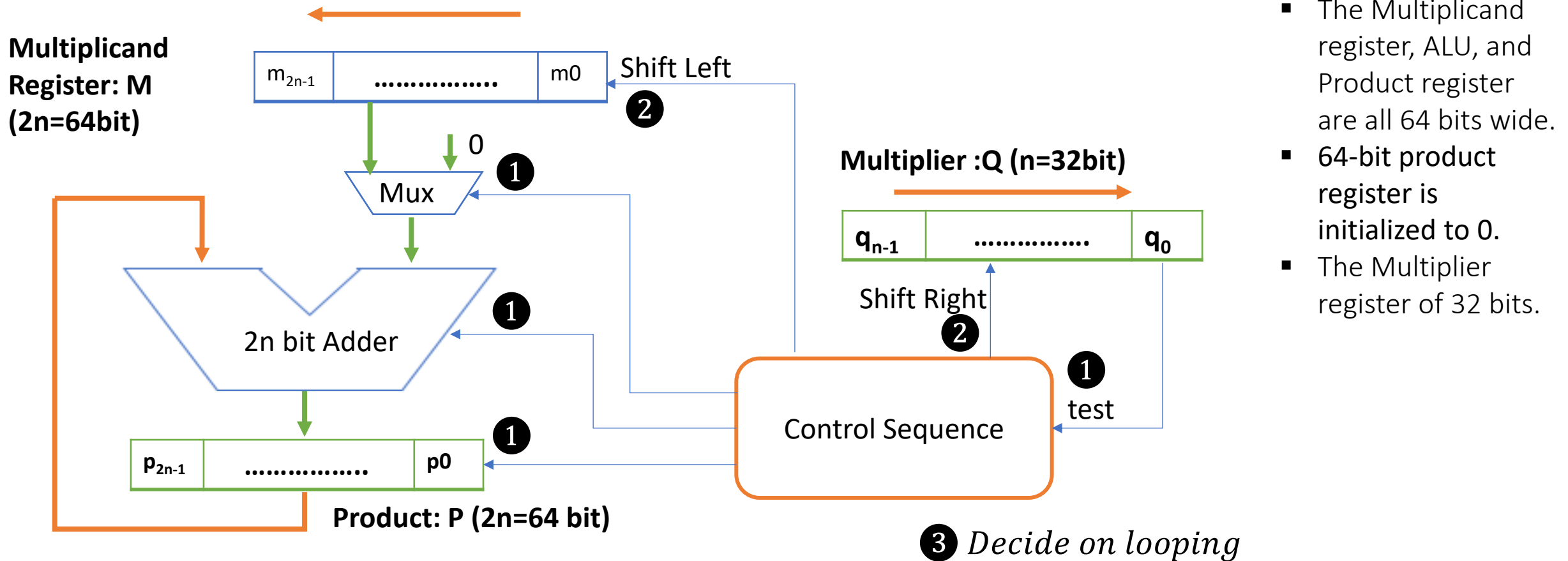
- The Multiplicand register, ALU, and Product register are all 64 bits wide.
- 64-bit product register is initialized to 0.
- The Multiplier register of 32 bits.

These three steps are repeated 32 times to obtain the product. If each step took one clock cycle, this algorithm would require almost 200 clock cycles to multiply two 32-bit numbers.

# Computer Organization and Design

## MULTIPLICATION : Sequential Circuit Multiplier

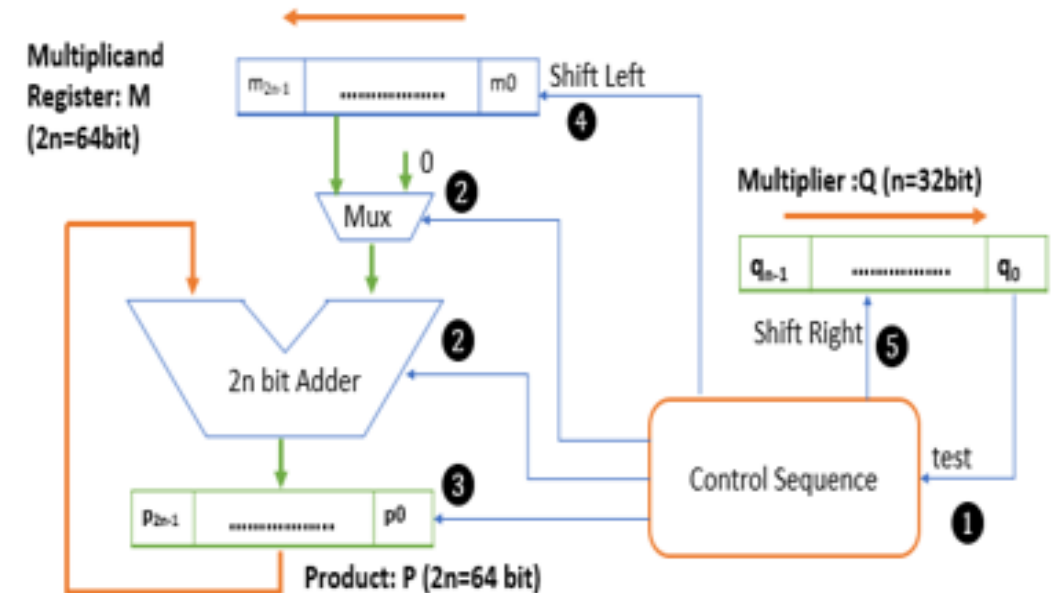
First version of the multiplication hardware



- The Multiplicand register, ALU, and Product register are all 64 bits wide.
- 64-bit product register is initialized to 0.
- The Multiplier register of 32 bits.

### First version of the multiplication hardware

- The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits.
- The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step.
- The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0.
- Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

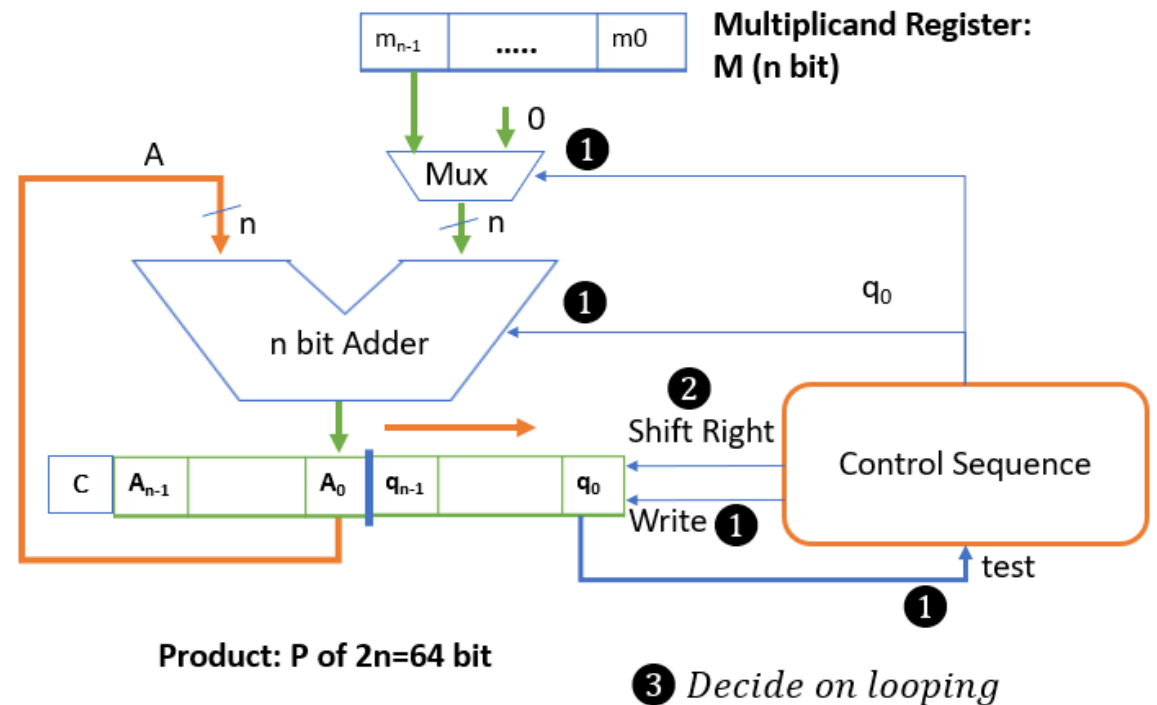




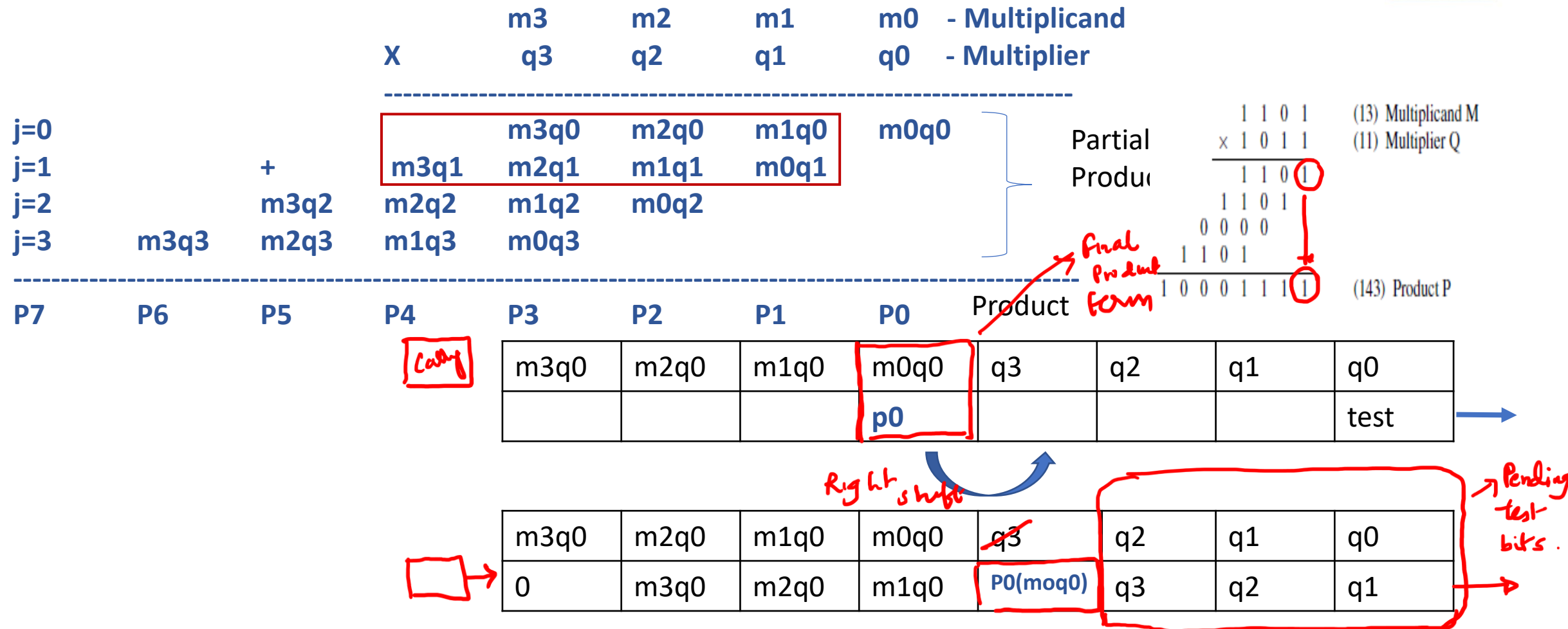
### Refined Version

- This circuit performs multiplication by using
  - ✓ **Single  $n$ -bit adder** instead  $2n$ -bit adders
  - ✓ **Registers P and Q are shift registers**, concatenated as shown, they hold partial product  $PP_i$  while multiplier bit  $q_0$  generates the signal Add/No add.
- ✓ This signal causes the multiplexer MUX to select 0 when  $q_0 = 0$ , or to select the multiplicand  $M$  when  $q_0 = 1$ , to be added to  $PP_i$  to generate  $PP(i + 1)$ . The product is computed in  $n$  cycles.
- ✓ Initially Register A is loaded with  $PP_i$  as 0s ( $n$  bit 0s)

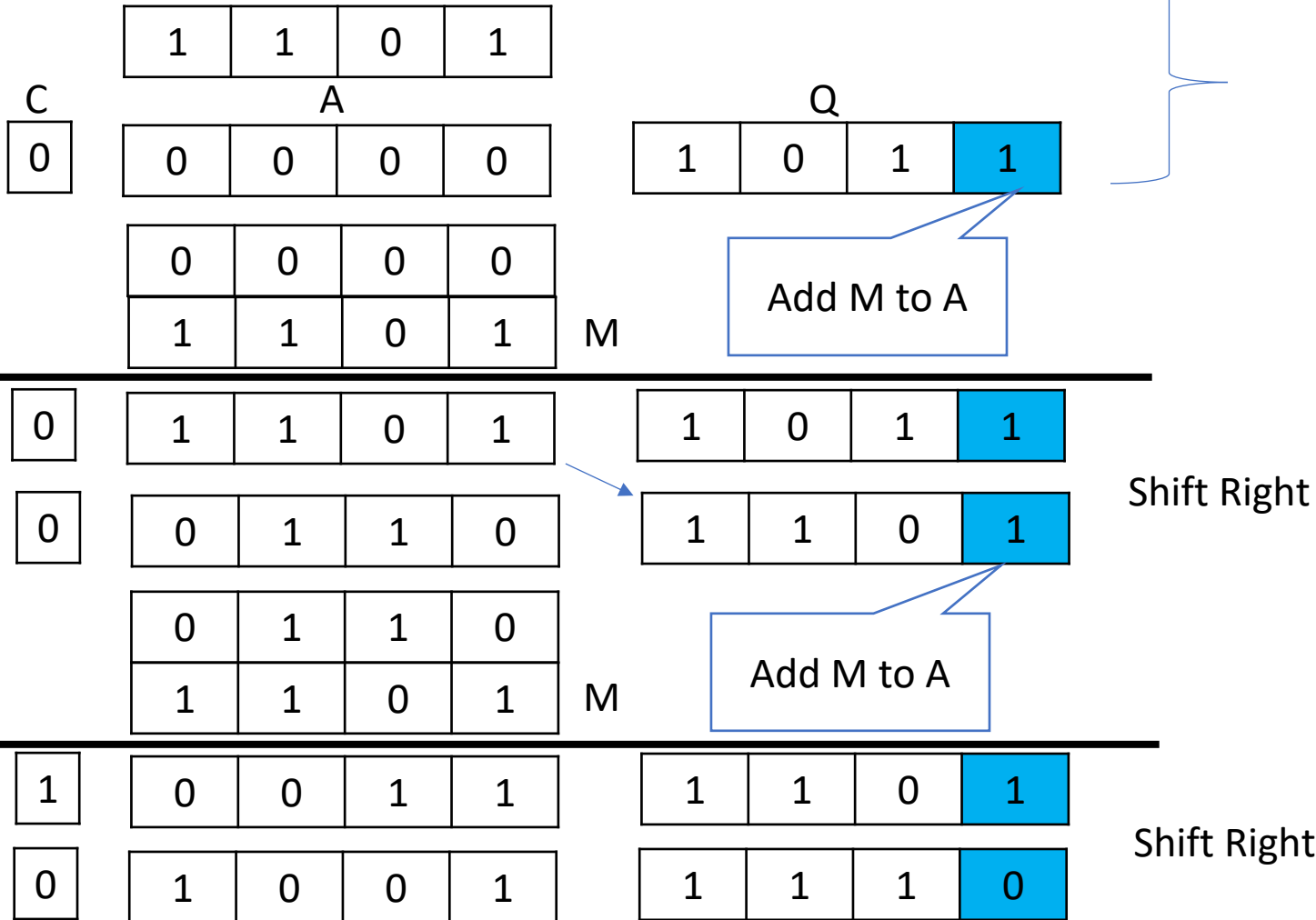
### Refined Version



Multiplier : For  $n=m=4$



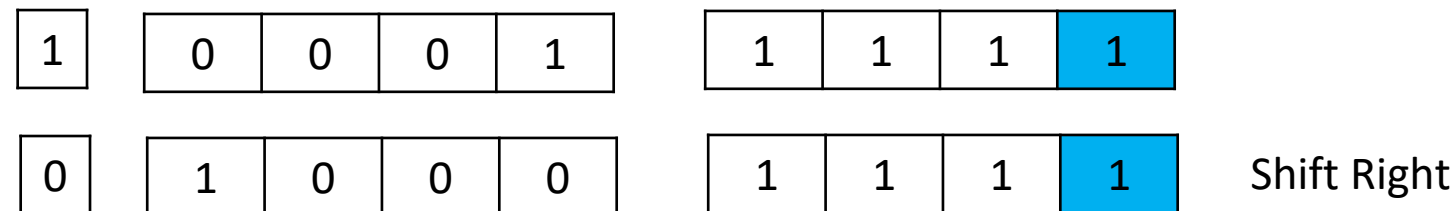
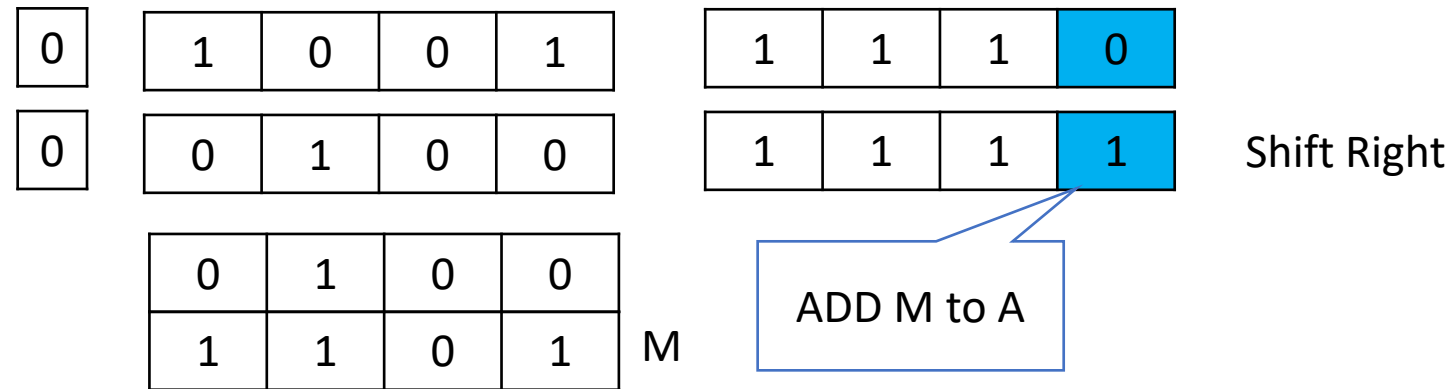
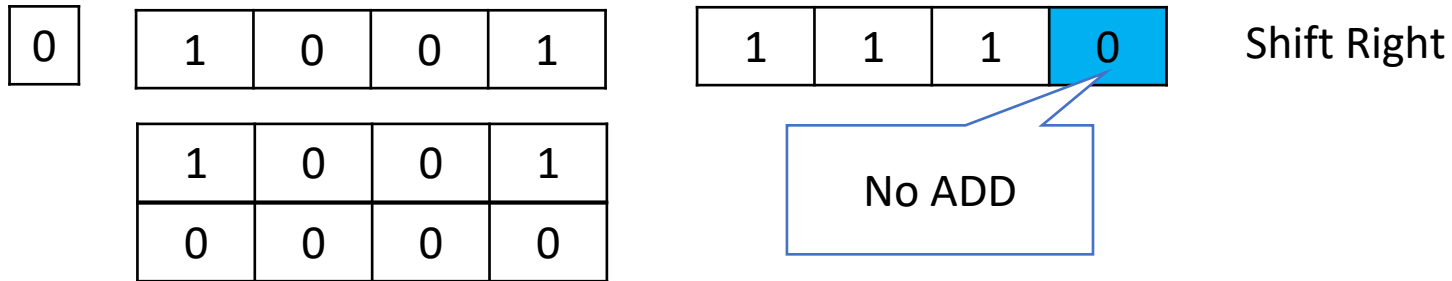
Refined Version M



$$\begin{array}{r}
 1101 \\
 \times 1011 \\
 \hline
 1101 \\
 1101 \\
 0000 \\
 1101 \\
 \hline
 10001111
 \end{array}$$

(13) Multiplicand M  
(11) Multiplier Q  
(143) Product P

### Refined Version



$$\begin{array}{r}
 1101 \\
 \times 1011 \\
 \hline
 1101 \\
 0000 \\
 1101 \\
 1000 \\
 \hline
 10001111
 \end{array}$$

(13) Multipl  
(11) Multipl  
(143) Product P

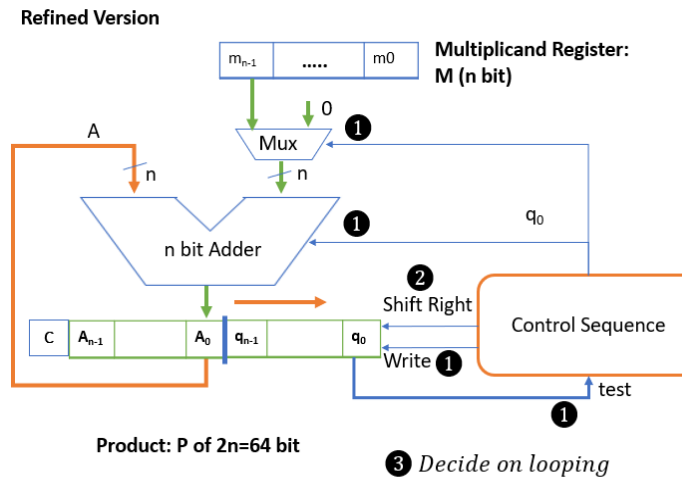
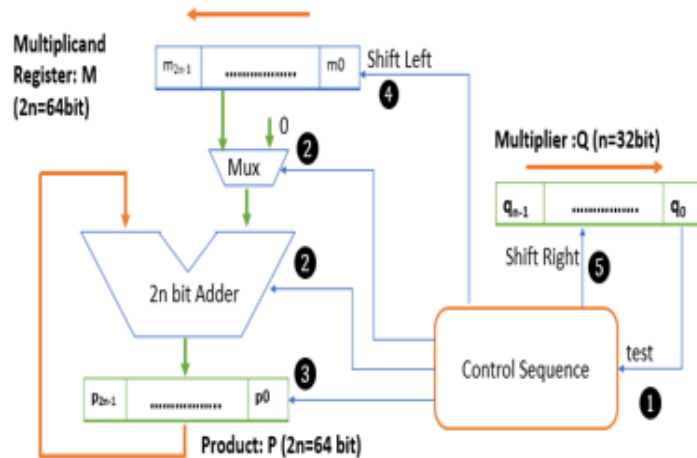
What are the changes in terms of hardware requirements?

Adder :  $n$  bit instead of  $2n$

Register :  
a) One  $(2n+1)$  right shift regis  
b)  $1, n$  bit register for M.



### Comparison of Refined Version Vs First Version



### Compare with the first version

- The **Multiplicand register and ALU** have been reduced to 32 bits.
- Now the **product is shifted right**.
- The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of
- the Product register, which has grown by **one bit to 33 bits to hold the carry-out** of the adder. (Note: It depends on size of data)

No of steps / iteration  
① test and add  
② shift right  
③ Decision.

**3.12** [20] <§3.3> Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.

Step	Action	Multiplier	Multiplicand	Product
0	Initial Vals	001 010	000 000 110 010	000 000 000 000
1	lsb=0, no op	001 010	000 000 110 010	000 000 000 000
	Lshift Mcand	001 010	000 001 100 100	000 000 000 000
	Rshift Mplier	000 101	000 001 100 100	000 000 000 000
2	Prod=Prod+Mcand	000 101	000 001 100 100	000 001 100 100
	Lshift Mcand	000 101	000 011 001 000	000 001 100 100
	Rshift Mplier	000 010	000 011 001 000	000 001 100 100
3	lsb=0, no op	000 010	000 011 001 000	000 001 100 100
	Lshift Mcand	000 010	000 110 010 000	000 001 100 100
	Rshift Mplier	000 001	000 110 010 000	000 001 100 100
4	Prod=Prod+Mcand	000 001	000 110 010 000	000 111 110 100
	Lshift Mcand	000 001	001 100 100 000	000 111 110 100
	Rshift Mplier	000 000	001 100 100 000	000 111 110 100
5	lsb=0, no op	000 000	001 100 100 000	000 111 110 100
	Lshift Mcand	000 000	011 001 000 000	000 111 110 100
	Rshift Mplier	000 000	011 001 000 000	000 111 110 100
6	lsb=0, no op	000 000	110 010 000 000	000 111 110 100
	Lshift Mcand	000 000	110 010 000 000	000 111 110 100
	Rshift Mplier	000 000	110 010 000 000	000 111 110 100

### First Version of Sequential Circuit Multiplier

Octal – 3 bit representation  
of numbers.

## MULTIPLICATION : Sequential Circuit Multiplier

**3.13** [20] <§3.3> Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 8-bit integers 62 and 12 using the hardware described in Figure 3.5. You should show the contents of each register on each step.

**Refined Version of  
Sequential Circuit Multiplier**

Step	Action	Multiplicand	Product/Multiplier
0	Initial Vals	110 010	000 000 001 010
1	lsb=0, no op	110 010	000 000 001 010
	Rshift Product	110 010	000 000 000 101
2	Prod=Prod+Mcand	110 010	110 010 000 101
	Rshift Mplier	110 010	011 001 000 010
3	lsb=0, no op	110 010	011 001 000 010
	Rshift Mplier	110 010	001 100 100 001
4	Prod=Prod+Mcand	110 010	111 110 100 001
	Rshift Mplier	110 010	011 111 010 000
5	lsb=0, no op	110 010	011 111 010 000
	Rshift Mplier	110 010	001 111 101 000
6	lsb=0, no op	110 010	001 111 101 000
	Rshift Mplier	110 010	000 111 110 100



## MULTIPLICATION : Sequential Circuit Multiplier

**3.14** [10] <§3.3> Calculate the time necessary to perform a multiply using the approach given in Figures 3.3 and 3.4 if an integer is 8 bits wide and each step of the operation takes four time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

### Hardware

1. Add- 1 clock
2. Shift – 1 clock
3. Decide looping – 1 clock

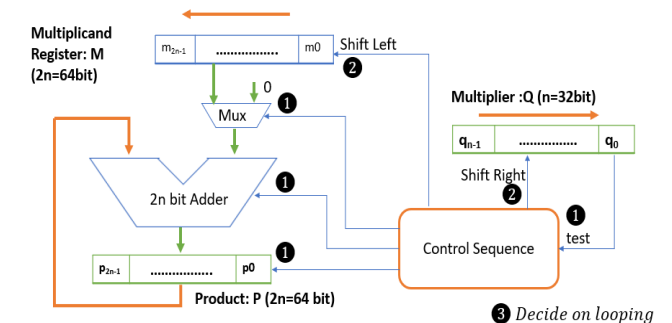
$$(3 \times 8) \times 4tu = 96 \text{ time units for hardware}$$

### Software

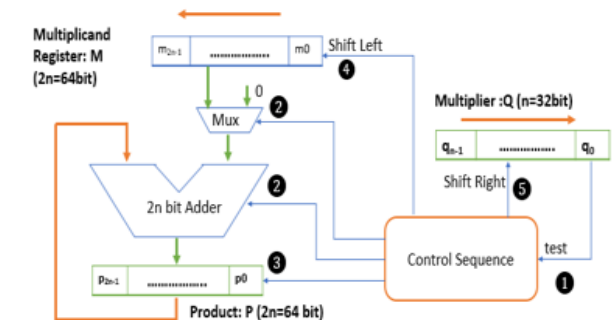
1. Decide what to add( $M/0$ ) -1 clk
2. Add- 1 clk
3. SR- 1 clk
4. SL- 1 clk
5. Decide on looping – 1 clk

$$(5 \times 8) \times 4tu = 160 \text{ time units for software}$$

### Sequential a) Hardware



### Sequential a) software






## FASTER MULTIPLICATION

Case 1: Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: **one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.**

**Multiplier : For  $n=m=4$  ; Number of levels  $=(n-1)$**



$$\begin{array}{r}
 1101 \quad (13) \text{ Multiplicand M} \\
 \times 1011 \quad (11) \text{ Multiplier Q} \\
 \hline
 1101 \\
 0000 \\
 1101 \\
 \hline
 10001111 \quad (143) \text{ Product P}
 \end{array}$$

				m3 q3	m2 q2	m1 q1	m0 q0	- Multiplicand - Multiplier
j=0				m3q0	m2q0	m1q0	m0q0	M.q0
j=1	+	m3q1	m2q1	m1q1	m0q1			M.q1
j=2		m3q2	m2q2	m1q2	m0q2			M.q2
j=3	m3q3	m2q3	m1q3	m0q3				M.q3
P7	P6	P5	P4	P3	P2	P1	P0	

Case 1: Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: **one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.**

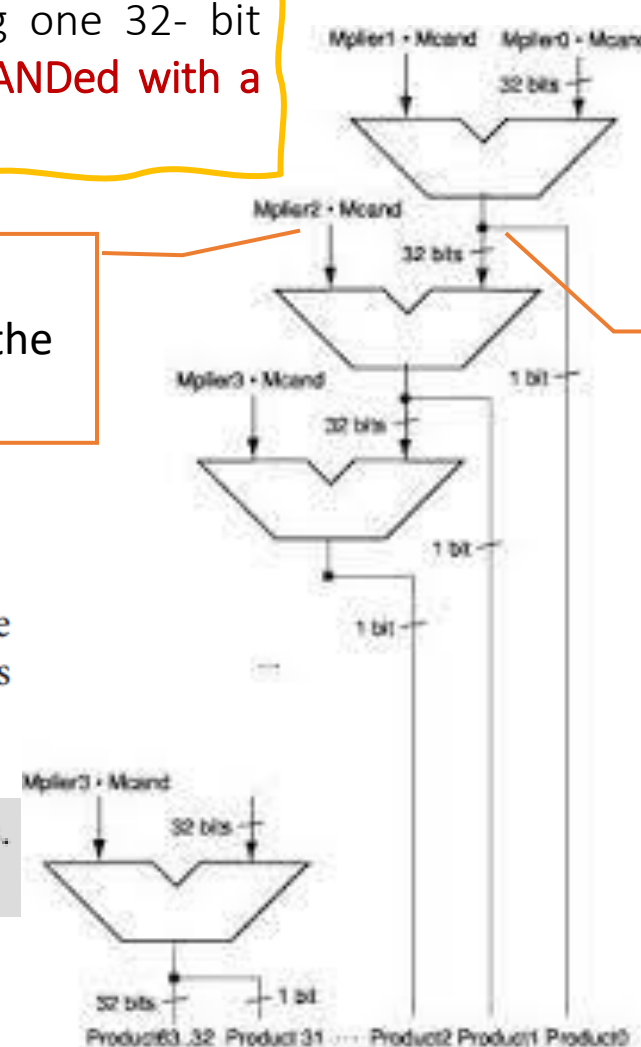
- Use 31,32-bit adders to compute the partial products
- One input is the multiplicand ANDed with the multiplier and the other is the partial product from the previous stage.

multiplicand  
ANDed with the  
multiplier

Partial Product  
from the  
previous stage

**3.15** [10] <§3.3> Calculate the time necessary to perform a multiply using the approach described in the text (31 adders stacked vertically) if an integer is 8 bits wide and an adder takes four time units.

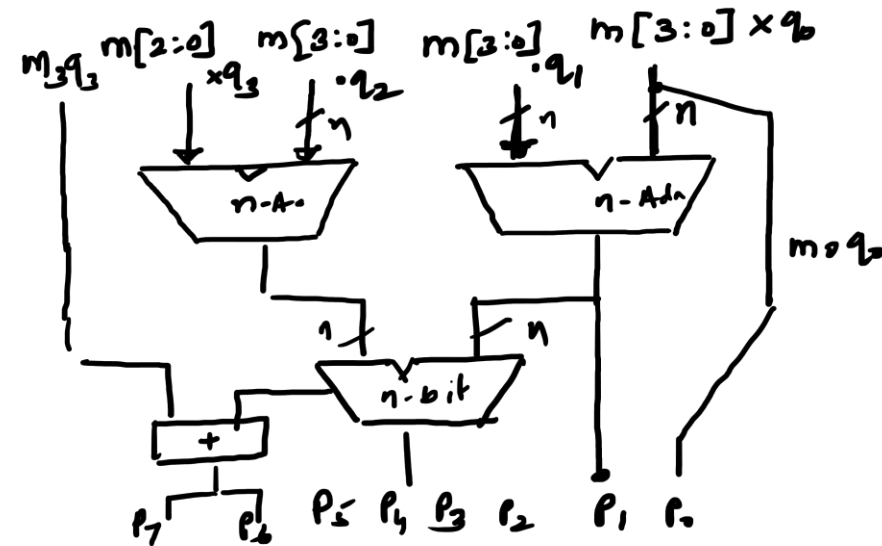
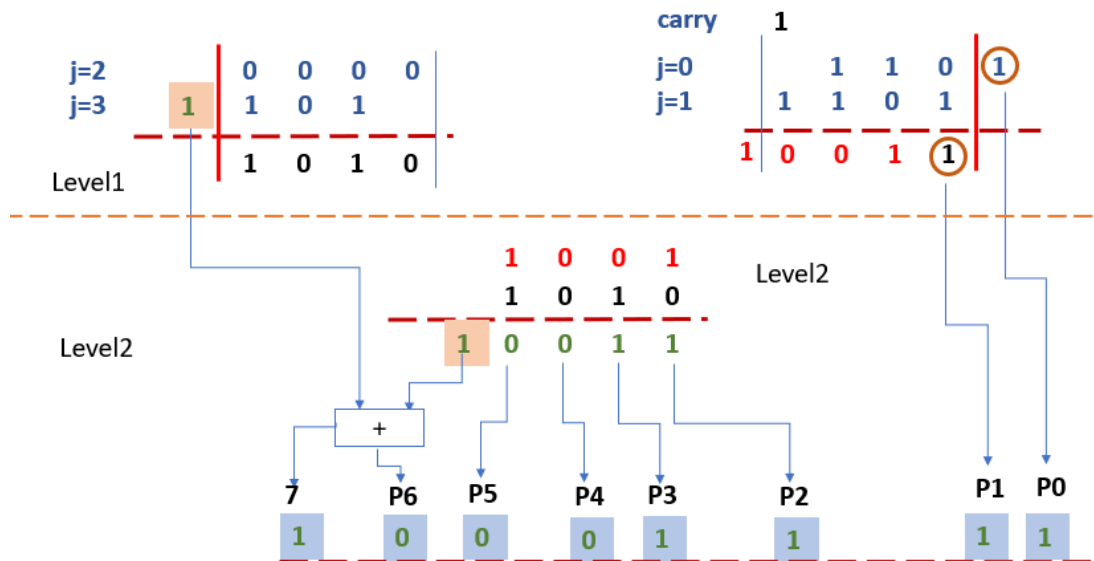
It takes  $B$  time units to get through an adder, and there will be  $A - 1$  adders. Word is 8 bits wide, requiring 7 adders.  $7 \times 4\text{tu} = 28$  time units.



## FASTER MULTIPLICATION

Case 2: An alternative way to organize these 32 additions is in a parallel tree. Instead of waiting for 32 add times, we wait just the  $\log_2(32)$  or five 32-bit add times.

**Multiplier :** For  $n=m=4$  ; Number of levels  $\log_2(4) = 2$



## FASTER MULTIPLICATION

### Case 2:

- An alternative way to organize these 32 additions is in a **parallel tree**. Instead of waiting for 32 add times, **we wait just the  $\log_2(32)$  or five 32-bit add times.**

① No of levels =  $\log_2(32) = 5$

② No of Adder in level 1 =  $\frac{n}{2}$

③ Successive levels, have  $\frac{1}{2}$  of previous levels

16, 32 bit adders

Level 1

8, 32 bit adders

Level 2

4, 32 bit adders

Level 3

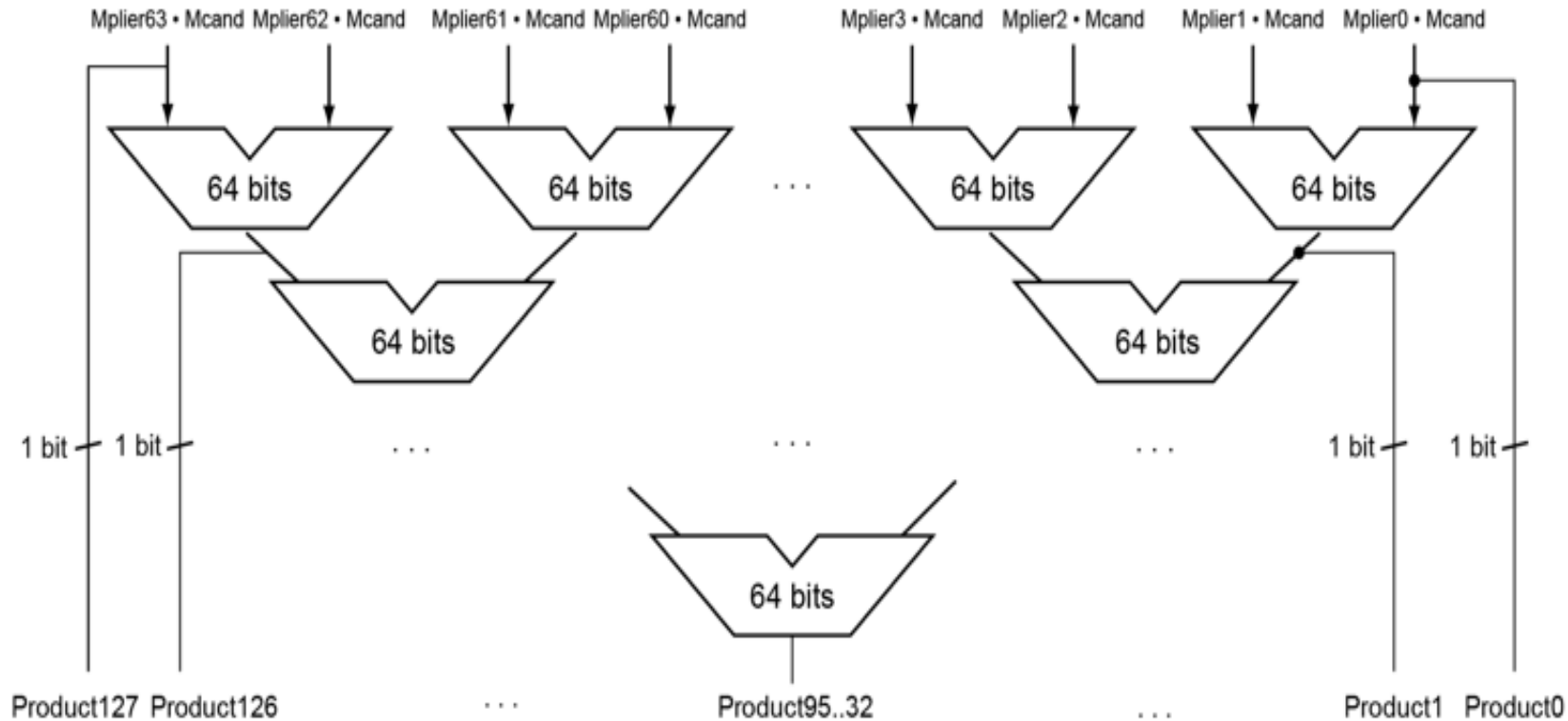
2, 32 bit adders

Level 4

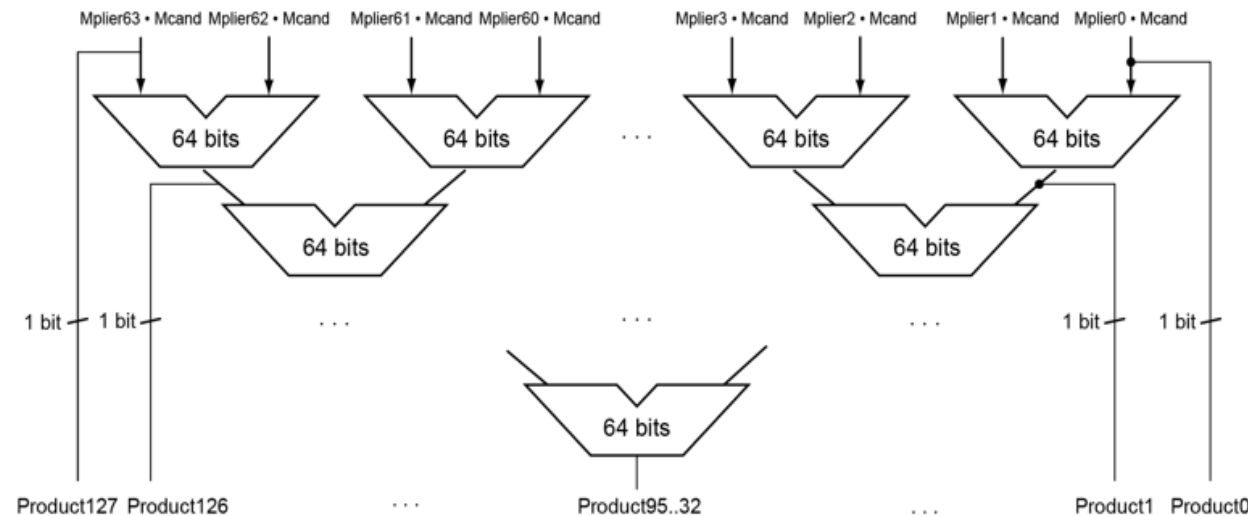
1, 32 bit adders

Level 5

Case 2:  $n=64$  bit



**3.16** [20] <§3.3> Calculate the time necessary to perform a multiply using the approach given in Figure 3.7 if an integer is 8 bits wide and an adder takes four time units.



It takes  $B$  time units to get through an adder, and the adders are arranged in a tree structure. It will require  $\log_2(A)$  levels. An 8 bit wide word requires seven adders in three levels.  $3 \times 4tu = 12$  time units.

### M Extension

Multiply	<code>mul x5, x6, x7</code>	$x5 = x6 \times x7$	Lower 32 bits of 64-bit product
Multiply high	<code>mulh x5, x6, x7</code>	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit signed product
Multiply high, unsigned	<code>mulhu x5, x6, x7</code>	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit unsigned product
Multiply high, signed-unsigned	<code>mulhsu x5, x6, x7</code>	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit signed-unsigned product
Divide	<code>div x5, x6, x7</code>	$x5 = x6 / x7$	Divide signed 32-bit numbers
Divide unsigned	<code>divu x5, x6, x7</code>	$x5 = x6 / x7$	Divide unsigned 32-bit numbers
Remainder	<code>rem x5, x6, x7</code>	$x5 = x6 \% x7$	Remainder of signed 32-bit division
Remainder unsigned	<code>remu x5, x6, x7</code>	$x5 = x6 \% x7$	Remainder of unsigned 32-bit division

# Computer Organization and Design

## RISC V Multiplication and Division Instruction

### M Extension

Programmer

FFFFFFFF × FFFFFFFF =

**FFFF FFFE 0000 0001**

Programmer

-1 × -1 =

**1**

HEX 1

FFFFFFFFFFFFFFFF × FFFFFFFF =

**FFFF FFFF 0000 0001**

HEX FFFF FFFF 0000 0001

DEC -4,29,49,67,295

### Case1: Unsigned Numbers

Source code

```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulhu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xfffffffffe

### Case2: Signed Numbers

Source code

```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulh x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0x00000000

### Case3: Signed Numbers x Unsigned Number

Source code

```
1 li x5,0xffffffff
2 li x6,0xffffffff
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xffffffff



# Computer Organization and Design

## RISC V Multiplication and Division Instruction

### M Extension

Programmer

FFFFFFFF × FFFFFFFF =

**FFFF FFFE 0000 0001**

Programmer

-1 × -1 =

**1**

HEX 1

FFFFFFFFFFFFFFFF × FFFFFFFF =

**FFFF FFFF 0000 0001**

HEX FFFF FFFF 0000 0001

DEC -4,29,49,67,295

### Case1: Unsigned Numbers

Source code

```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulhu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xfffffffffe

### Case2: Signed Numbers

Source code

```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulh x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0x00000000

### Case3: Signed Numbers x Unsigned Number

Source code

```
1 li x5,0xffffffff
2 li x6,0xffffffff
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xffffffff

# Computer Organization and Design

## RISC V Multiplication and Division Instruction

### M Extension

Programmer

FFFFFFFF × FFFFFFFF =

**FFFF FFFE 0000 0001**

Programmer

-1 × -1 =

**1**

HEX 1

FFFFFFFFFFFFFFFF × FFFFFFFF =

**FFFF FFFF 0000 0001**

HEX FFFF FFFF 0000 0001

DEC -4,29,49,67,295

### Case1: Unsigned Numbers

Source code

```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulhu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xfffffffffe

### Case2: Signed Numbers

Source code

```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulh x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0x00000000

### Case3: Signed Numbers x Unsigned Number

Source code

```
1 li x5,0xffffffff
2 li x6,0xffffffff
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xffffffff

# Computer Organization and Design

## RISC V Multiplication and Division Instruction

### M Extension

Programmer

FFFFFFFF × FFFFFFFF =

**FFFF FFFE 0000 0001**

Programmer

-1 × -1 =

**1**

HEX 1

FFFFFFFFFFFFFFFF × FFFFFFFF =

**FFFF FFFF 0000 0001**

HEX FFFF FFFF 0000 0001

DEC -4,29,49,67,295

### Case1: Unsigned Numbers

Source code

```
1 li x5,0xffffffff
2 li x6,0x0f
3 mul x7,x5,x6
4 mulhu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0x0000000f
x7	t2	0xffffffff1
x8	s0	0x0000000e

### Case2: Signed Numbers

Source code

```
1 li x5,0xffffffff
2 li x6,0x0f
3 mul x7,x5,x6
4 mulh x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0x0000000f
x7	t2	0xffffffff1
x8	s0	0xffffffff

### Case3: Signed Numbers x Unsigned Number

Source code

```
1 li x5,0xffffffff
2 li x6,0x0f
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0x0000000f
x7	t2	0xffffffff1
x8	s0	0xffffffff

# COMPUTER ORGANIZATION AND DESIGN

---

## Arithmetic for Computers

### 3.4

#### Division

**Mahesh Awati**

Department of Electronics and Communication Engineering

←

Quotient 1001<sub>ten</sub>

Divisor 1000<sub>ten</sub> | 1001010<sub>ten</sub> Dividend

→

-1000

10

101

1010

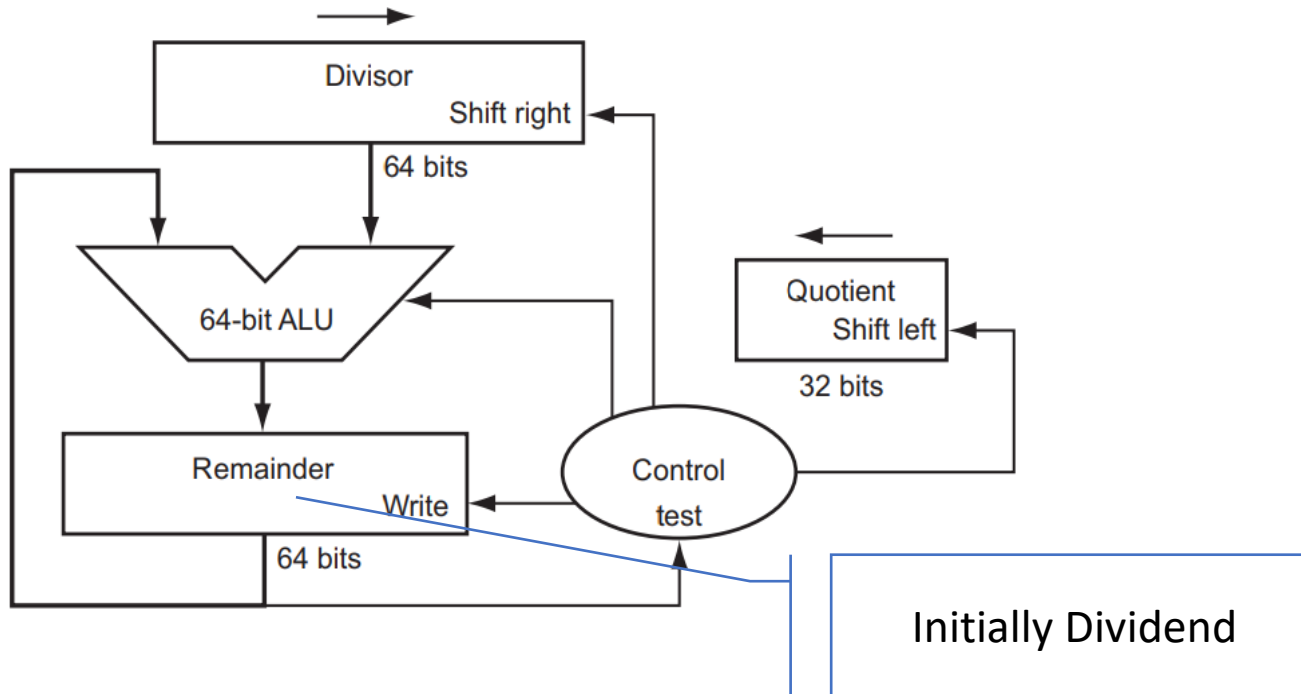
-1000

10<sub>ten</sub> Remainder

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

### First version of the division Algorithm

- We start with the 32-bit Quotient register set to 0.
- **Each iteration of the algorithm needs to move the divisor to the right one digit**, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend.
- The Remainder register is initialized with the dividend

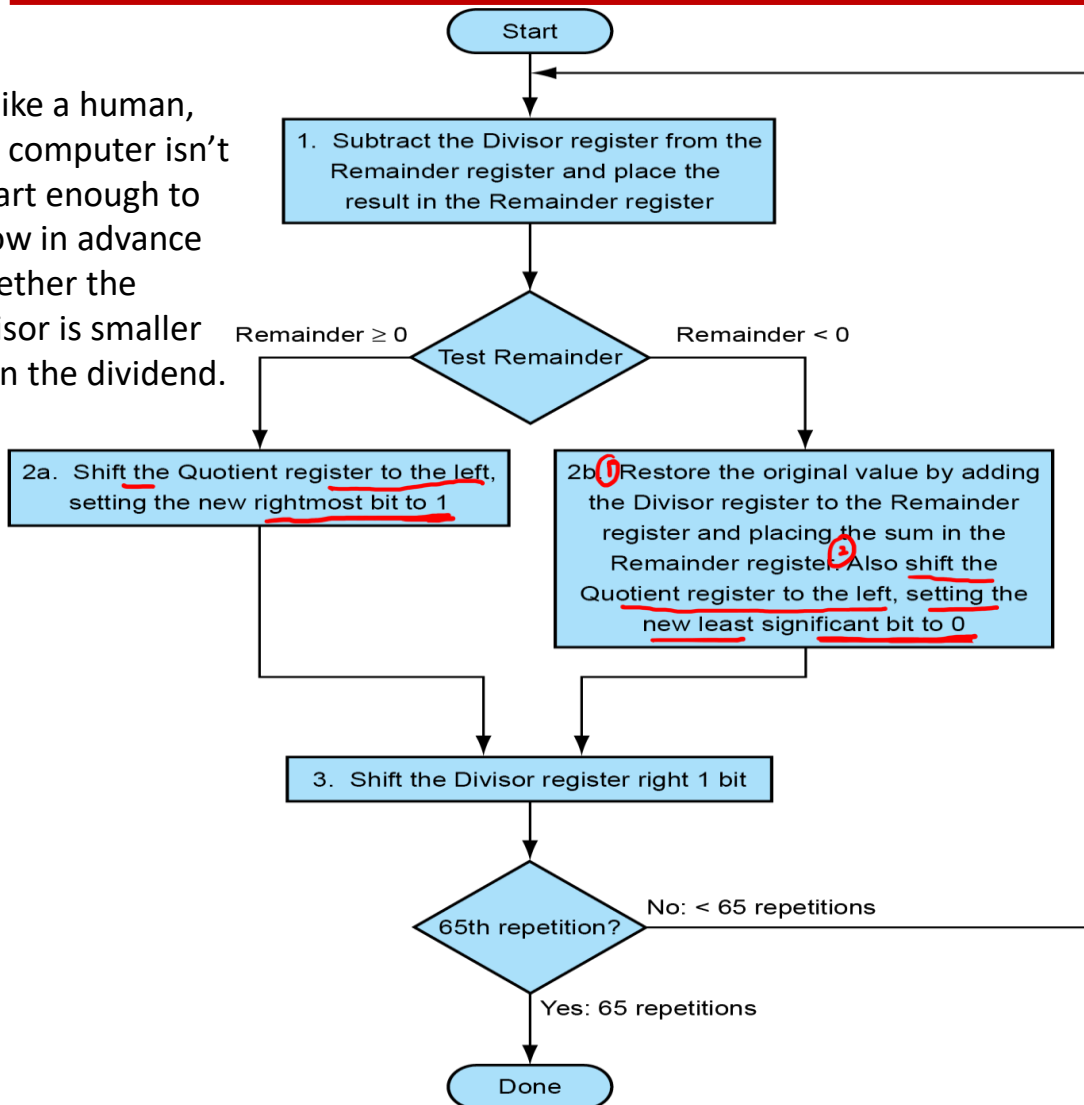


- This means, The Divisor register, ALU, and Remainder register are all 64(2n)bits wide, with only the Quotient register being 32 bits.
- Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

# Computer Organization and Design

## Division

Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend.



$$\begin{array}{r}
 \text{Divisor } 1000_{\text{ten}} \quad \leftarrow \\
 \text{Dividend } 1001010_{\text{ten}} \\
 \hline
 1001_{\text{ten}} \quad \text{Quotient} \\
 -1000 \\
 \hline
 10 \\
 101 \\
 1010 \\
 -1000 \\
 \hline
 10_{\text{ten}} \quad \text{Remainder}
 \end{array}$$

- (n+1) times.
- ①  $\text{Rem} = \text{Rem} - \text{Divisor}$ .
  - ② if  $(\text{Rem} < 0)$ 
    - $\text{Rem} = \text{Rem} + \text{Divisor}$  (Restore)
    - and shift Left Quotient & make  $Q_0 = 0$ .
    - else
    - shift Left Quotient & make  $Q_0 = 1$ .
  - ③ shift Right Divisor 1 time.

# Computer Organization and Design

## Division

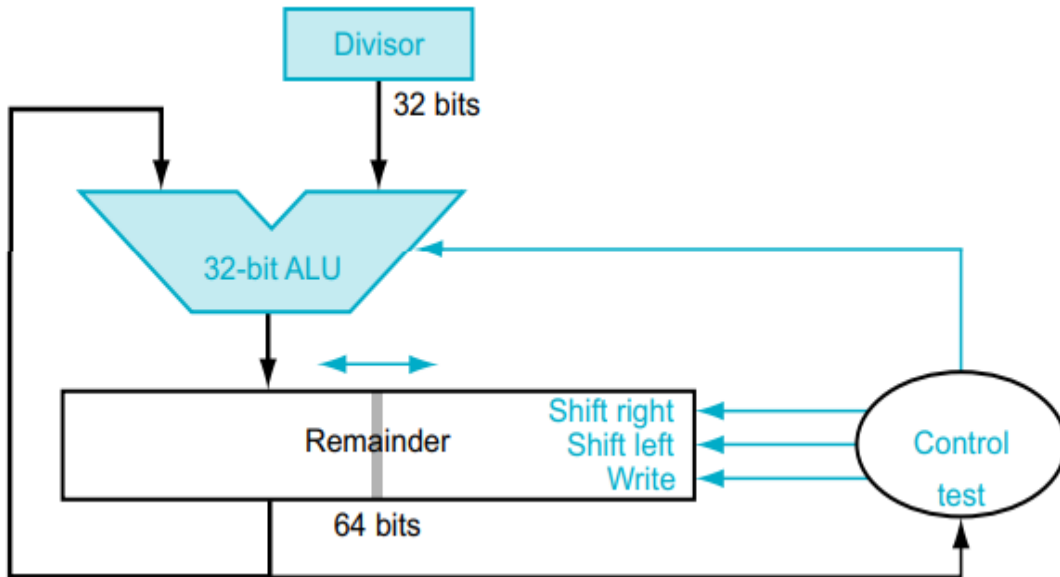
### A Divide Algorithm

Using a 4-bit version of the algorithm to save pages, let's try dividing  $7_{\text{ten}}$  by  $2_{\text{ten}}$ ,  
or  $0000\ 0111_{\text{two}}$  by  $0010_{\text{two}}$ .

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	<sup>2</sup> 0010 0000	0000 <sup>7</sup> 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 $\Rightarrow$ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 $\Rightarrow$ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 $\Rightarrow$ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	①000 0011
	2a: Rem $\geq$ 0 $\Rightarrow$ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	①000 0001
	2a: Rem $\geq$ 0 $\Rightarrow$ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

The surprising requirement of this algorithm is that it takes **(n+1)** steps to get the proper quotient and remainder.





- The Divisor register, ALU, and Quotient register are all 32 bits wide. i.e., the ALU and Divisor registers are halved in comparison with version 1 and the remainder is shifted left.
- This version also combines the Quotient register with the right half of the Remainder register.

### M Extension

Multiply	<code>mul x5, x6, x7</code>	$x5 = x6 \times x7$	Lower 32 bits of 64-bit product
Multiply high	<code>mulh x5, x6, x7</code>	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit signed product
Multiply high, unsigned	<code>mulhu x5, x6, x7</code>	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit unsigned product
Multiply high, signed-unsigned	<code>mulhsu x5, x6, x7</code>	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit signed-unsigned product
Divide	<code>div x5, x6, x7</code>	$x5 = x6 / x7$	Divide signed 32-bit numbers
Divide unsigned	<code>divu x5, x6, x7</code>	$x5 = x6 / x7$	Divide unsigned 32-bit numbers
Remainder	<code>rem x5, x6, x7</code>	$x5 = x6 \% x7$	Remainder of signed 32-bit division
Remainder unsigned	<code>remu x5, x6, x7</code>	$x5 = x6 \% x7$	Remainder of unsigned 32-bit division

## Signed Division

Remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Complication of signed division is that we must also set the sign of the remainder.

The following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

Example:  $\pm 7$ ten by  $\pm 2$ ten:

Case 1:  $+7 / +2$ : Quotient = + 3, Remainder = + 1: check -  $+ 7 \ 3 \times 2 + (+1) = 6 + 1$

Case 2:  $-7 / +2$ : Quotient = - 3, Remainder = -1

Case 3:  $+7 / -2$ : Quotient = - 3, Remainder = + 1

Case 4:  $-7 / -2$ : Quotient = +3, Remainder = -1

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

Dividend	Divisor	Quotient	Remainder
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

# COMPUTER ORGANIZATION AND DESIGN

---

## Arithmetic for Computers

### 3.5

### Floating Point

**Mahesh Awati**

Department of Electronics and Communication Engineering

# Computer Organization and Design

## FLOATING POINT NUMBERS AND OPERATIONS

### 1. FLOATING POINT REPRESENTATION

Why Floating Point Representation needed?

So far we have seen how a decimal number is represented in binary and how it is classified as signed and unsigned, represented and how they are stored

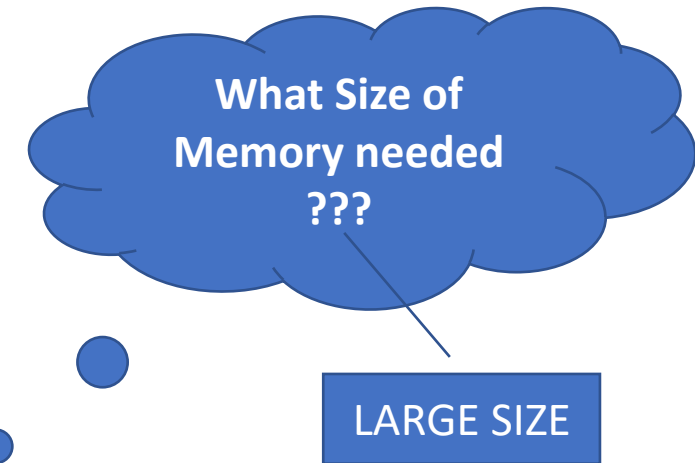
Ex:  $(65535)_{10} = 0xFFFF = 1111\ 1111\ 1111\ 1111$

But, How about storing a small number like  $0.00000000005 = 0.5 \times 10^{-10}$   
Or How about storing a Large number like  $500000000000 = 5 \times 10^{11}$

If Conventional Binary Representation is used –The Number is needed to be Converted into Binary and then need to be stored in memory.

Can we represent these by using Fixed Size Memory?

**Yes, There are standard representation to do and are called Floating Point Representation defined by IEEE commonly used in all processors**



$3.14159265..._{10}$  (pi)

$2.71828..._{10}$  (e)

$0.000000001_{10}$  or  $1.0_{10} \times 10^{-9}$  (seconds in a nanosecond)

$3,155,760,000_{10}$  or  $3.15576_{10} \times 10^9$  (seconds in a typical century)

### 1. FLOATING POINT REPRESENTATION

There are three standards used for floating point number representation defined by IEEE

Precision	Sign	Exponent	Mantissa	Total
single	1	8	23	32
double	1	11	52	64
long double	1	15	64	80

#### a) Single Precision - 32 bit Number representation

It is called a *single-precision* representation **because it occupies a single 32-bit word**. IEEE 32 bit Number Floating Point representation is as shown below

31	30	23	22	0
Sign	Bias Exponent (E')		M- Mantissa fraction (23 bits)	

# Computer Organization and Design

## FLOATING POINT NUMBERS AND OPERATIONS



### a) Single Precision - 32 bit Number representation

where,

- ✓ **Bit 31:** S- Sign bit

If S=1 – Negative Sign and S=0 – Positive Sign

$$\text{Value Represented} = \pm 1.M \times 2^{E'-127}$$

- ✓ **Bit [30:23]: Biased Exponent** -8 bits allows us to represent  $2^8=256$  different integers.

$$(-1)^S \times F \times 2^E$$

We need both positive and negative integers and , we keep two integers for special cases, so we have 254 left to **cover -126 to +127** (Actual sign exponent E excluding special cases).

In this, 127 is exponent bias ,added to the actual exponent ( E ) and stored in exponent field and is called **excess 127** representation (E') which is **an unsigned integer**

$E' = E + 127$  , Therefore, the range of E' for normal values is  $-126+127=1 \leq E' \leq 127+127=254$  & E' including special cases is in the range  $0 \leq E' \leq 255$ .

- ✓ **Bit [22:0]:** 23 bit **Mantissa fraction**

The fractional part of the significant bits called mantissa, **always has a leading 1, with the binary point immediately to its right.** Therefore, the

mantissa

$B = 1.M = 1.b_{-1} b_{-2} b_{-3} \dots b_{-23}$  has the value

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \dots + b_{-23} \times 2^{-23}$$

### a) Single Precision - 32 bit Number representation

#### ✓ Bit [22:0]: 23 bit Mantissa fraction

- To pack even more bits into the number, IEEE 754 makes the **leading 1 bit of normalized binary numbers implicit**.
- The number is actually **24 bits long in single precision** (implied 1 and a 23-bit fraction), and **53 bits long in double precision** (1 + 52).
- To be precise, we use the term **significand** to represent the 24- or 53-bit number that is 1 plus the fraction, and fraction when we mean the 23- or 52-bit number.
- 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \dots + b_{-23} \times 2^{-23}$$

$$\text{Value Represented} = \pm 1.M \times 2^{E-127}$$



# Computer Organization and Design

## FLOATING POINT NUMBERS AND OPERATIONS

### a) Single Precision - 32 bit Number representation

Example:  $+(257.3)_{10}$  convert to IEEE 754 32 Bit Floating Point Representation

#### Step1 : Represent data into Binary Form

$(257)_{10} \rightarrow (100000001)_2$   
 $(0.3)_{10} \rightarrow (0100110011.....)_2$

Process of Moving decimal point to the position just right to the most significant non zero digit (bit in binary)

0 1 0 0 0 0 0 0 0 1 . 0 1 0 0 1 1 0 0 1 1 ....

#### Step2: Normalize the data represented in binary form

0 1 0 0 0 0 0 0 0 1 . 0 1 0 0 1 1 0 0 1 1 ....

#### Step3: Identify the exponent and biased Exponent

0 1 . 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 1 1 ....

$\times 2^8$

Exponent  $E' = E + 127 = 8 + 127 = 135$

31	30	23	22	0														
Sign	Exponent –Excess 127- E'			M- Mantissa fraction (23 bits)														
0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	1	.....

0.3	X2	0.6	0
0.6	X2	1.2	1
0.2	X2	0.4	0
0.4	X2	0.8	0
0.8	X2	1.6	1
0.6	X2	1.2	1

# Computer Organization and Design

## FLOATING POINT NUMBERS AND OPERATIONS



### a) Single Precision - 32 bit Number representation

Example

31	30		23		22		0
0	1	0	0	0	0	0	1 1 . 0 0 0 1 0 0 1..... 0

31	Sign	0	It is a Positive Number
----	------	---	-------------------------

30:23	E'	1 0 0 0 0 0 1 1	Exponent
		$2^7$ $2^1$ $2^0$	$E' = 2^7 + 2^2 + 2^1 = 128 + 2 + 1 = 131$

22:0	M	0 0 0 1 0 0 1 ....	Mantissa = $1/8 + 1/128 = 9/128$
		$2^{-1}$ $2^{-4}$ $2^{-7}$	$= 0.0703125$

Value Represented =  $\pm 1.M \times 2^{E'-127} = +1.0001001..... \times 2^{(131-127)} = 1.0001001 \times 2^4$   
 Sign (1+Mantissa)  $\times 2^{(E'-127)} = + (1+0.0703125) \times 2^{(131-127)} = +(1.0703125) \times 2^4$   
 $= + 17.125$

1. Identify Sign of the Number
2. Find the biased Exponent.
3. Find out actual Exponent.
4. Identify the Fractional Part and Find the decimal Value

**Value Represented =**  
 $\pm 1.M \times 2^{E'-127}$  or  $(-1)^S \times F \times 2^E$

### Examples

Show the IEEE 754 binary representation of the number  $-0.75_{\text{ten}}$  in single and double precision.

### Examples

**3.22** [10] <§3.5> What decimal number does the bit pattern  $0 \times 0C000000$  represent if it is a floating point number? Use the IEEE 754 standard.

### Examples

**3.23** [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

**3.24** [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

# Computer Organization and Design

## FLOATING POINT NUMBERS AND OPERATIONS

### Examples

#### Converting Binary to Decimal Floating Point

#### EXAMPLE

What decimal number does this single precision float represent?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

# Computer Organization and Design

## FLOATING POINT NUMBERS AND OPERATIONS

### a) Single Precision - 32 bit Number representation

#### Converting Binary to Decimal Floating Point

#### EXAMPLE

What decimal number does this single precision float represent?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The sign bit is 1, the exponent field contains 129, and the fraction field contains  $1 \times 2^{-2} = 1/4$ , or 0.25. Using the basic equation,

#### ANSWER

$$\begin{aligned} (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129 - 127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0 \end{aligned}$$

### a) Single Precision - 32 bit Number representation

**Overflow (floating point)** : A situation in which a positive exponent becomes too large to fit in the exponent field.

**Underflow (floating point)**: A situation in which a negative exponent becomes too large to fit in the exponent field.

What should happen on an overflow or underflow to let the user know that a problem occurred?

- Some computers signal these events by raising an exception, sometimes called an interrupt.
- **Exception/Interrupt** - unscheduled event that disrupts program execution; used to detect overflow.
- The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
- **RISC-V computers do not raise an exception** on overflow or underflow; instead, software can read the **floating-point control and status register (fcsr)** to check whether overflow or underflow has occurred.



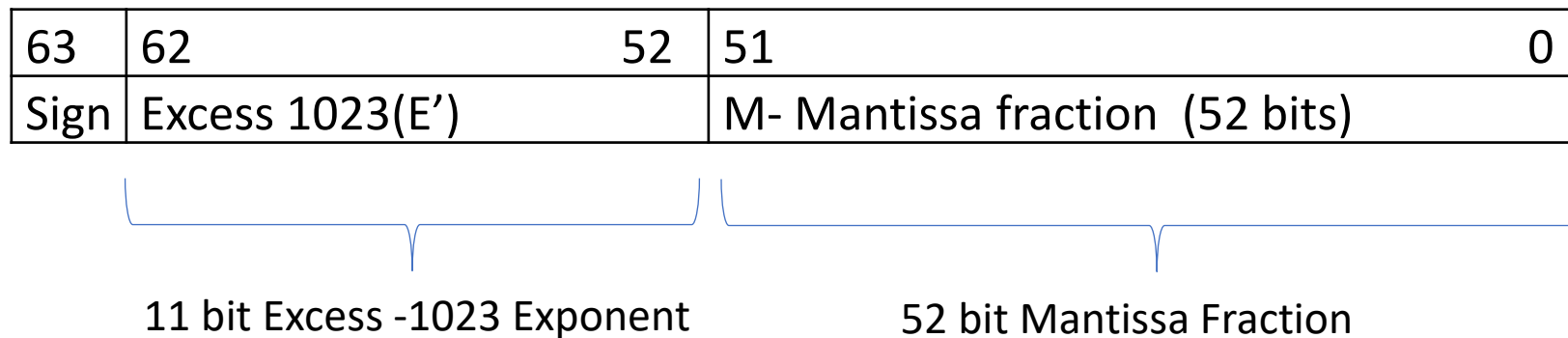
### b) Double Precision FP Representation - 64 bit

This tradeoff is between precision and range:

- Increasing the size of the **fraction enhances the precision of the fraction**,
- While increasing the size of the **exponent increases the range of numbers** that can be represented.

### b) Double Precision FP Representation - 64 bit

To provide **more precision and range** for floating-point numbers, the IEEE standard also specifies a *double-precision* format



The double-precision format has increased exponent and mantissa ranges.

The 11-bit **excess-1023 exponent**  $E$  has the range  $1 \leq E \leq 2046$  for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent  $E$  is in the range  $-1022 \leq E \leq 1023$ , providing scale factors of  $2^{-1022}$  to  $2^{1023}$

The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional.

The standard also specifies certain **optional extended versions** of both of these formats. The extended versions provide **increased precision** and **increased exponent range** for the representation of intermediate values in a sequence of calculations.

The use of extended formats helps to reduce the size of the **accumulated round-off error in a sequence of calculations leading to a desired result.**

### 2. ADDITION AND SUBTRACTION

**Step1 :** Choose the number with the smaller exponent and find difference in exponents.

Difference in the Exponents  $n = |E_A - E_B|$

Shift Mantissa of the Number with the smaller Exponent right by **n steps**.  
In this **n=2**

**Step 2:** Add the Mantissa

**Step 3:** Normalizing

**Step 4:** Rounding

Single precision requires only 23 fraction bits. As the Normalized results has additional bit, it should be rounded of as follows

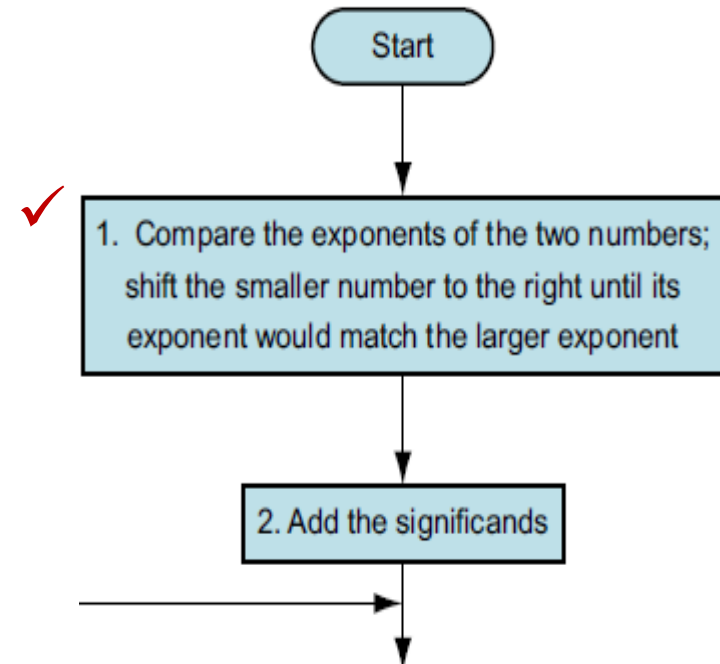
- a) Round bit (R)- The bit after the Normalized bit is Round bit
- b) Sticky Bit (S)-The bits after the round bit are Sticky bits - All sticky bits should be ORed and taken as One bit

G	R	S	
X	0	0	Truncate
X	0	X	Truncate
0	1	0	Tie, Truncate
1	1	0	Tie, +1 to LSB
X	1	1	+1 LSB

### 2. ADDITION AND SUBTRACTION

#### Example

<b>+1.</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>x</b>	<b>2<sup>4</sup></b>	
<b>+1.</b>	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	x	2 <sup>2</sup>



### 2. ADDITION AND SUBTRACTION

**Step1 :** Choose the number with the smaller exponent and find difference in exponents.

+1.	1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	x	2 <sup>4</sup>
+1.	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1	x	2 <sup>2</sup>

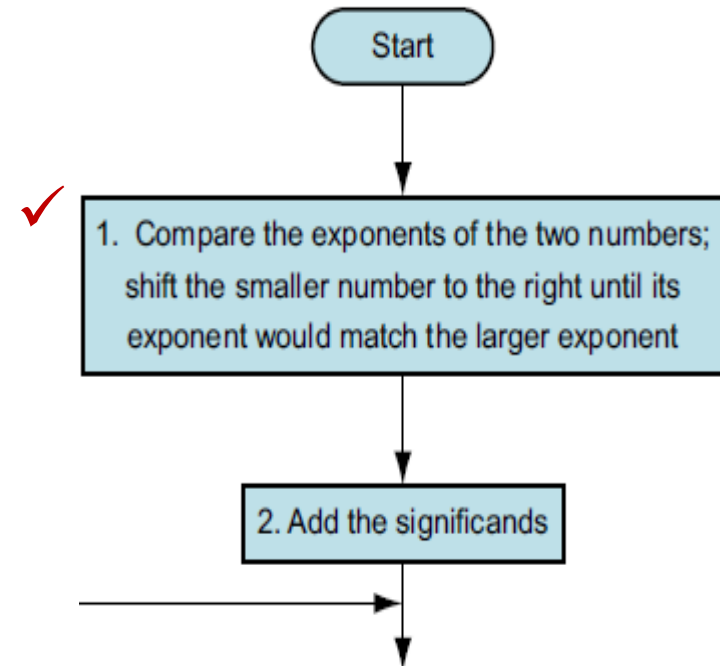
Difference in the Exponents  $n = |E_A - E_B| = |4 - 2| = 2$

Shift Mantissa of the Number with the smaller Exponent right by **n times**  
**i.e, Exponents of number match.** In this **n=2**

0.	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1	x	2 <sup>4</sup>
----	---	---	----------------

**Step 2:** Add the Mantissa

1.	1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	x	2 <sup>4</sup>
0.	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1	x	2 <sup>4</sup>
1 0.	0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 1	x	2 <sup>4</sup>



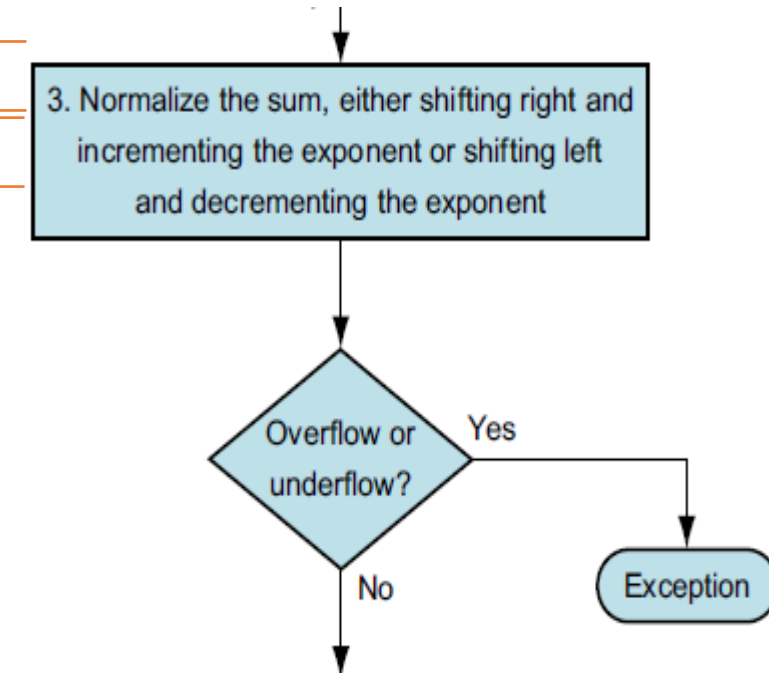
### 2. ADDITION AND SUBTRACTION

#### Step 3: Normalizing

1 0. 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 1      x  $2^4$

1 .0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 1      x  $2^5$

Since  $127 \geq +5 \geq -126$ , there is no overflow or underflow. (The biased exponent would be  $+5 + 127$ , or 132, which is between 1 and 254, the smallest and largest unreserved biased exponents.)



### 2. ADDITION AND SUBTRACTION

**Step4:** Rounding Single precision requires only 23 fraction bits. As the Normalized results has additional bit, it should be rounded of as follows

a) Round bit (R)- The bit after the Normalized bit is Round bit

b) Sticky Bit (S)-The bits after the round bit are Sticky bits - All sticky bits should be ORed and taken as One bit

	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	R	S	S	
	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0							
1.	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	1	x	2 <sup>5</sup>

In the above example R=1 and Oring of Sticky Bits (0|1 =1). Therefore RS=11, Since RS=11 the increment the fraction by 1 to round it off.

1.	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1						x	2 <sup>5</sup>
																												1		
1.	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0						x	2 <sup>5</sup>

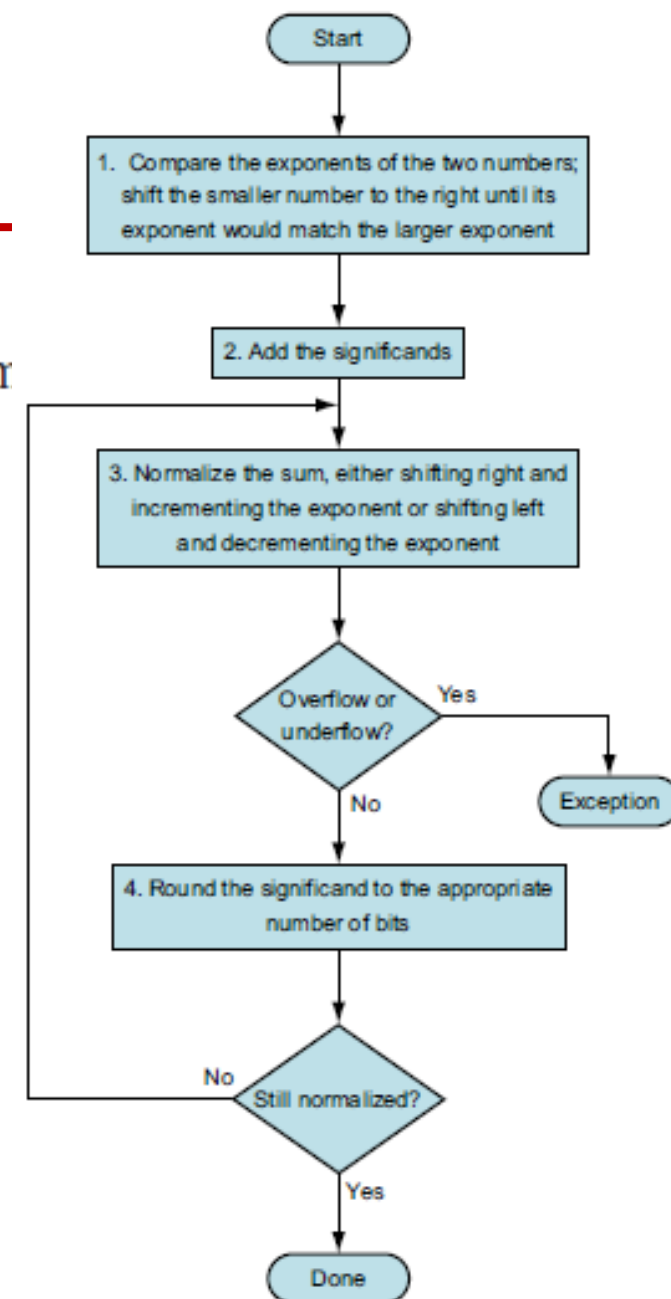
G	R	S	
X	0	0	Truncate
X	0	X	Truncate
0	1	0	Tie, Truncate
1	1	0	Tie, +1 to LSB
X	1	1	+1 LSB

**Step 5:** Repeat Step3 and 4 if the final result is Not in Normalized state

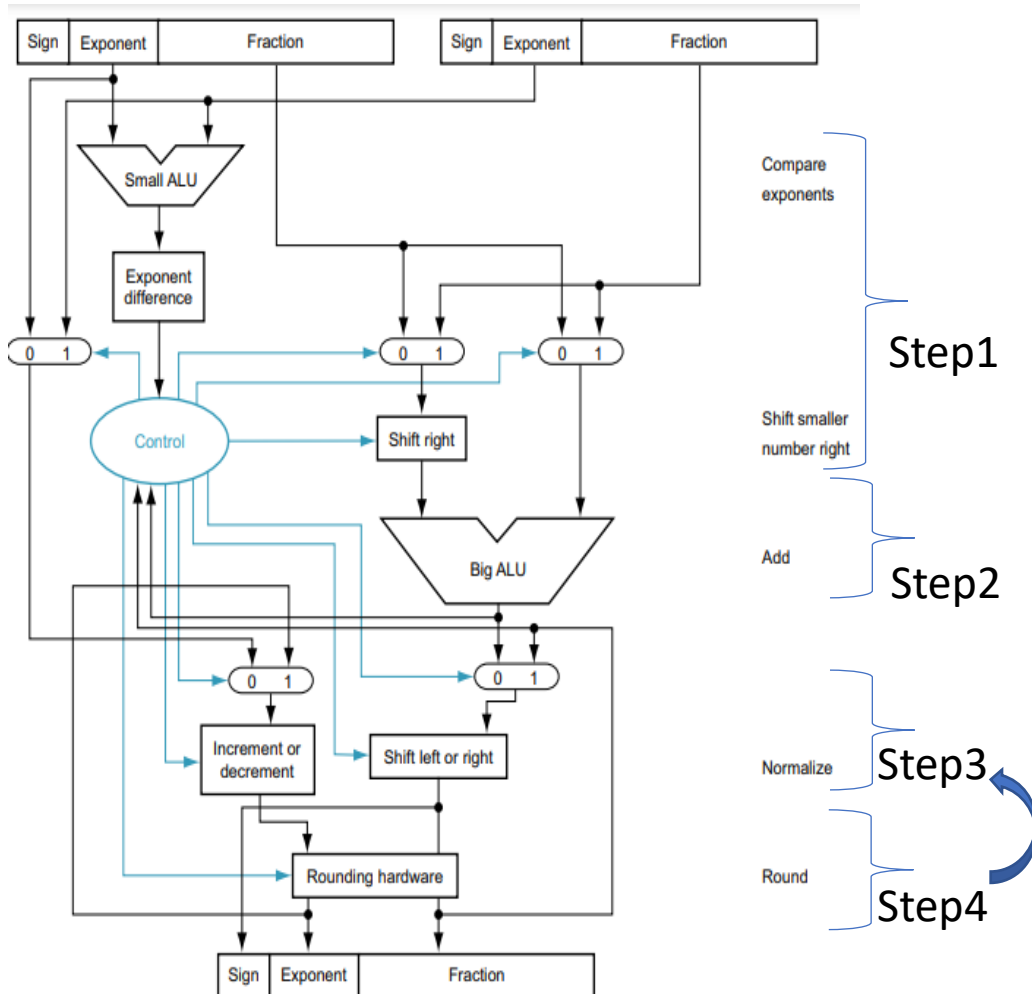


### 2. ADDITION AND SUBTRACTION

Try adding the numbers  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$  in binary using the algorithm



### 2. ADDITION AND SUBTRACTION



Exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much.

This difference controls the three multiplexors; from left to right, they select

1. the **larger exponent**,
2. the **significand of the smaller number**, and
3. the **significand of the larger number**.

The smaller significand is **shifted right**,

then the **significands are added** together using the big ALU

The normalization step then **shifts the sum left or right** and **increments or decrements the exponent**.

Rounding then creates the final result, which may require normalizing again to produce the actual final result

### 2. ADDITION AND SUBTRACTION

Add / Sub	SA	SB	Function	Description
0	0	0	Addition	As signs of A & B are same and Operation is Addition
0	0	1	Subtraction	As sign of A & B are different and Operation required is addition
0	1	0	Subtraction	As sign of A & B are different and Operation required is addition
0	1	1	Addition	Addition As signs of A & B are same and Operation is Addition
1	0	0	Subtraction	As signs of A & B are same and Operation is Subtraction
1	0	1	Addition	As sign of A & B are different and Operation required is Subtraction
1	1	0	Addition	As sign of A & B are different and Operation required is Subtraction
1	1	1	Subtraction	As signs of A & B are same and Operation is Subtraction

$$AS \oplus SA \oplus SB$$

### 3. MULTIPLICATION

Let two numbers be  $M1 \times 2^{E1}$  and  $M2 \times 2^{E2}$

**Step1** : Add the exponents  $E1$  and  $E2$  of two numbers

**Note:** If exponents biased, then Add them and Subtract the bias value

$E_R = E1 + E2$  ; In case  $E1'$  and  $E2'$  are given for Single Precision Numbers, then  $ER' = (E1' + E2') - 127$  . **As bias has got added twice 127 is subtracted once so that added result is biased only once.**

31	30	23		22	0	$E'$
1	0 0 0 0 0 1 1 1	.		1 0 .....	0	7
0	1 1 1 0 0 0 0 0	.		1 0 .....	0	224

$E'R = (E'1 + E'2) - 127 = (7 + 224) - 127 = 104$  ( 0 1 1 0 1 0 0 0 ) - Biased

**Step2** : Multiply the Mantissa  $M1$  and  $M2$  of two numbers and determine the Sign of the result

$M1 = 1.1000000...0$  &  $M2 = 1.100000 .....0$ ;  $M1 \times M2 = 10.0100000 \times 2^{104}$

$$(M1 \times 2^{E1}) * (M2 \times 2^{E2})$$

$$= \pm M1 \times M2. 2^{(E1+E2)}$$

Implicit Part of Mantissa

### 3. MULTIPLICATION

**Step3** : Normalize the resulting value, If Necessary

Result =  $M1 \times M2 \times 2^{104} = 10.010000 \times 2^{104} = 1.0010000 \times 2^{105}$  (Normalized)

$E'R = (105) = 0110\ 1001$

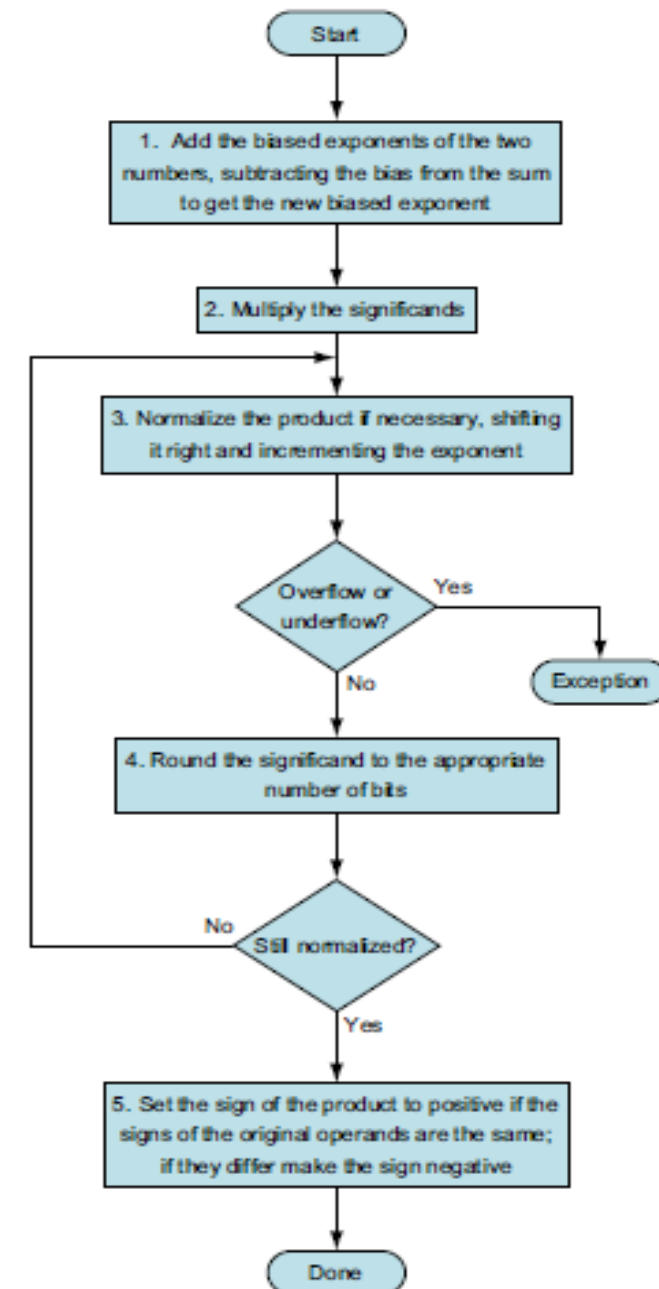
Sign = Negative as One of the Number is Negative

31	30		23		22		0
1	0	1	1	0	1	0	0
				.	0	0	1
					0	0	.....0

### 3. MULTIPLICATION

#### Binary Floating-Point Multiplication

Let's try multiplying the numbers  $0.5_{\text{ten}}$  and  $-0.4375_{\text{ten}}$ ,



### ROUNDING AND TRUNCATION

While doing arithmetic operations on Floating Point numbers, it might have led to increase in the number of mantissa bits beyond the defined size of mantissa. In case of Single Precision floating point representation the size of mantissa of resulted number might have become more than 23 (24 bits including the implied leading 1) bits.

**1. Chopping:** The simplest way is to remove the guard bits and make no changes in the retained bits.

Actual Result	Minimum	Maximum	Truncated	Error Range
$b_{-1} b_{-2} b_{-3} b_{-4} b_{-5} b_{-6}$	$b_{-1} b_{-2} b_{-3} 0 0 0$	$b_{-1} b_{-2} b_{-3} 1 1 1$	$b_{-1} b_{-2} b_{-3}$	0 to 0.000111

The error in chopping ranges **from 0 to almost 1 in the least significant position** of the retained bits.

**Drawback:** The result of chopping is a *biased* approximation because the error range is not symmetrical about 0.

### ROUNDING AND TRUNCATION

#### 2. Von Neumann rounding.

$b_{-1} b_{-2} b_{-3} b_{-4} b_{-5} b_{-6}$	Truncation Action	Error Range
$b_{-1} b_{-2} b_{-3} 0 0 0$	If the bits to be removed are all 0s, they are simply dropped, with no changes to the retained bits.	0
$b_{-1} b_{-2} b_{-3} 0 0 1$ $b_{-1} b_{-2} b_{-3} 0 1 0$ $b_{-1} b_{-2} b_{-3} 0 1 1$ $b_{-1} b_{-2} b_{-3} 1 0 0$ $b_{-1} b_{-2} b_{-3} 1 0 1$ $b_{-1} b_{-2} b_{-3} 1 1 0$ $b_{-1} b_{-2} b_{-3} 1 1 1$	if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1. In our 6-bit to 3-bit truncation example, all 6-bit fractions with $b_{-4}b_{-5}b_{-6}$ not equal to 000 are truncated to $0.b_{-1}b_{-2}1$ .	-1 and +1 in the LSB position of the retained bits.

**Advantage:** the approximation is **unbiased because the error range is symmetrical about 0**. Unbiased approximations are advantageous because **positive errors tend to offset negative errors as the computation proceeds**.



### ROUNDING AND TRUNCATION

#### 3. Rounding

$b_{-1}$	$b_{-2}$	$b_{-3}$	$b_{-4}$	$b_{-5}$	$b_{-6}$	Truncation Action
		G	R	S	S	
$b_{-1}$	$b_{-2}$	$b_{-3}$	0	0	0	Result is exact, No need of rounding
$b_{-1}$	$b_{-2}$	$b_{-3}$	0	X	X	It is rounded to $0.b_{-1}b_{-2}b_{-3}$ by discarding RS
			1	0	0	If $R=1$ and $S=0$ ; It is Tie case-either truncate or increment. Need to break the tie in an unbiased way, one possibility is to choose the retained bits to be the nearest even number. This is now decided by Guard bit G. If $G=0$ - It is truncated to the value $0.b_{-1}b_{-2}0$
$b_{-1}$	$b_{-2}$	0	1	0	0	

G	R	S	
X	0	0	Truncate
X	0	X	Truncate
0	1	0	Tie, Truncate
1	1	0	Tie, +1 to LSB
X	1	1	+1 LSB

### ROUNDING AND TRUNCATION

#### 3. Rounding

$b_{-1}$	$b_{-2}$	1	1	0	0	If $G=1$ - It is rounded to $0.b_{-1}b_{-2}1 + 0.001$ .
$b_{-1}$	$b_{-2}$	$b_{-3}$	1	X	X	R=1 and S=1 ; add 1 to LSB. It is rounded to $0.b_{-1}b_{-2}b_{-3} + 0.001$ Note: All don't care conditions can't be 0
$b_{-1}$	$b_{-2}$	$b_{-3}$	1	X	X	
$b_{-1}$	$b_{-2}$	$b_{-3}$	1	X	X	

This rounding technique is the default mode for truncation specified in the IEEE floating-point standard[1].

- IEEE 754 encoding of floating-point numbers: A separate sign bit determines the Sign.

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	$\pm$ denormalized number
1-254	Anything	1-2046	Anything	$\pm$ floating-point number
255	0	2047	0	$\pm$ infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

- Note: Check slide 17-18*

RISC-V supports the IEEE 754 single-precision and double-precision formats with these instructions:

- Floating-point *addition, single* (`fadd.s`) and *addition, double* (`fadd.d`)
- Floating-point *subtraction, single* (`fsub.s`) and *subtraction, double* (`fsub.d`)
- Floating-point *multiplication, single* (`fmul.s`) and *multiplication, double* (`fmul.d`)
- Floating-point *division, single* (`fdiv.s`) and *division, double* (`fdiv.d`)
- Floating-point square root, single (`fsqrt.s`) and square root, double (`fsqrt.d`)
- Floating-point equals, single (`feq.s`) and equals, double (`feq.d`)
- Floating-point less-than, single (`flt.s`) and less-than, double (`flt.d`)
- Floating-point less-than-or-equals, single (`fle.s`) and less-than-or-equals, double (`fle.d`)

# Computer Organization and Design

## FLOATING POINT Instructions in RISC V

### RISC-V floating-point assembly language

Arithmetic	FP add single	<code>fadd.s f0, f1, f2</code>	<code>f0 = f1 + f2</code>	FP add (single precision)
	FP subtract single	<code>fsub.s f0, f1, f2</code>	<code>f0 = f1 - f2</code>	FP subtract (single precision)
	FP multiply single	<code>fmul.s f0, f1, f2</code>	<code>f0 = f1 * f2</code>	FP multiply (single precision)
	FP divide single	<code>fdiv.s f0, f1, f2</code>	<code>f0 = f1 / f2</code>	FP divide (single precision)
	FP square root single	<code>fsqrt.s f0, f1</code>	<code>f0 = <math>\sqrt{f1}</math></code>	FP square root (single precision)
	FP add double	<code>fadd.d f0, f1, f2</code>	<code>f0 = f1 + f2</code>	FP add (double precision)
	FP subtract double	<code>fsub.d f0, f1, f2</code>	<code>f0 = f1 - f2</code>	FP subtract (double precision)
	FP multiply double	<code>fmul.d f0, f1, f2</code>	<code>f0 = f1 * f2</code>	FP multiply (double precision)
	FP divide double	<code>fdiv.d f0, f1, f2</code>	<code>f0 = f1 / f2</code>	FP divide (double precision)
	FP square root double	<code>fsqrt.d f0, f1</code>	<code>f0 = <math>\sqrt{f1}</math></code>	FP square root (double precision)
Comparison	FP equality single	<code>feq.s x5, f0, f1</code>	<code>x5 = 1 if f0 == f1, else 0</code>	FP comparison (single precision)
	FP less than single	<code>flt.s x5, f0, f1</code>	<code>x5 = 1 if f0 &lt; f1, else 0</code>	FP comparison (single precision)
	FP less than or equals single	<code>fle.s x5, f0, f1</code>	<code>x5 = 1 if f0 &lt;= f1, else 0</code>	FP comparison (single precision)
	FP equality double	<code>feq.d x5, f0, f1</code>	<code>x5 = 1 if f0 == f1, else 0</code>	FP comparison (double precision)
	FP less than double	<code>flt.d x5, f0, f1</code>	<code>x5 = 1 if f0 &lt; f1, else 0</code>	FP comparison (double precision)
	FP less than or equals double	<code>fle.d x5, f0, f1</code>	<code>x5 = 1 if f0 &lt;= f1, else 0</code>	FP comparison (double precision)
Data transfer	FP load word	<code>flw f0, 4(x5)</code>	<code>f0 = Memory[x5 + 4]</code>	Load single-precision from memory
	FP load doubleword	<code>fld f0, 8(x5)</code>	<code>f0 = Memory[x5 + 8]</code>	Load double-precision from memory
	FP store word	<code>fsw f0, 4(x5)</code>	<code>Memory[x5 + 4] = f0</code>	Store single-precision from memory
	FP store doubleword	<code>fsd f0, 8(x5)</code>	<code>Memory[x5 + 8] = f0</code>	Store double-precision from memory



# THANK YOU

---

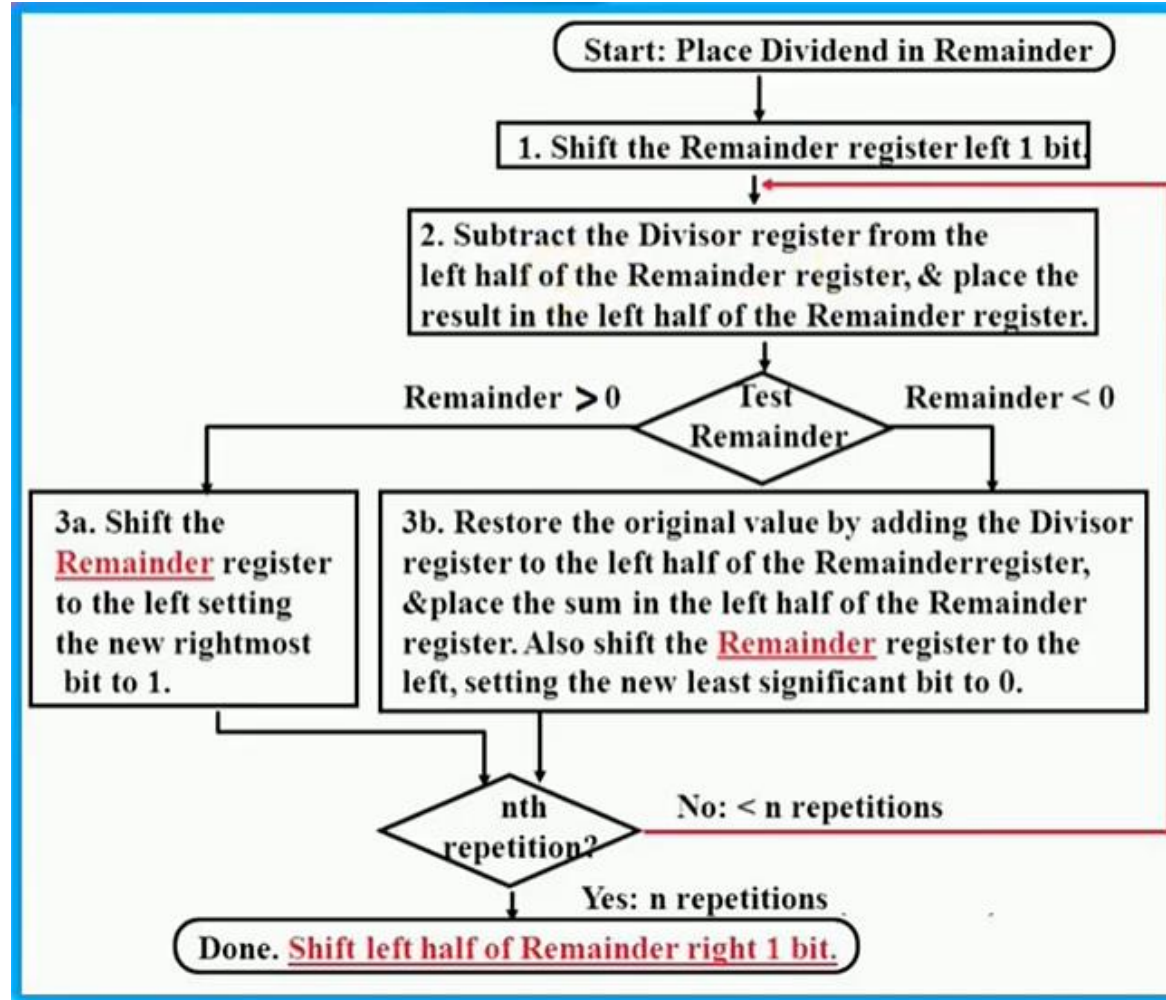
**Mahesh Awati**

Department of Electronics and Communication

**mahesha@pes.edu**

**+91 9741172822**





A improved version,  
1. Divisor Register need to be of 32 bit size only