Lecture 5 Multiplication and Division

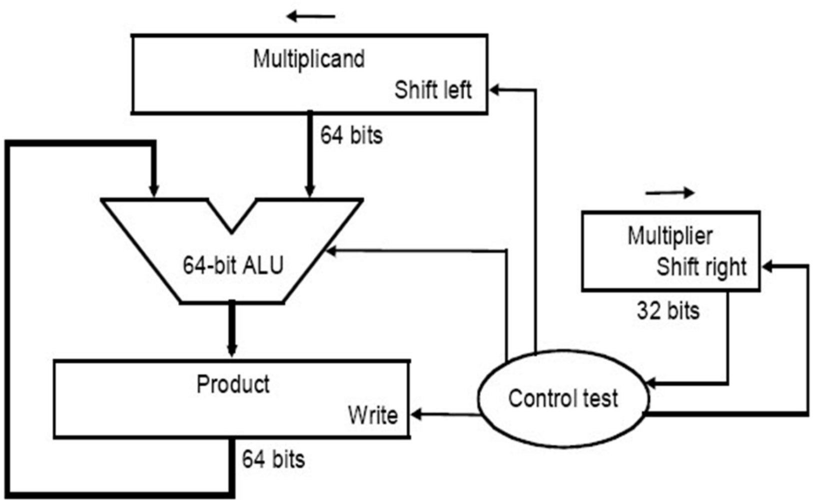
Multiplication

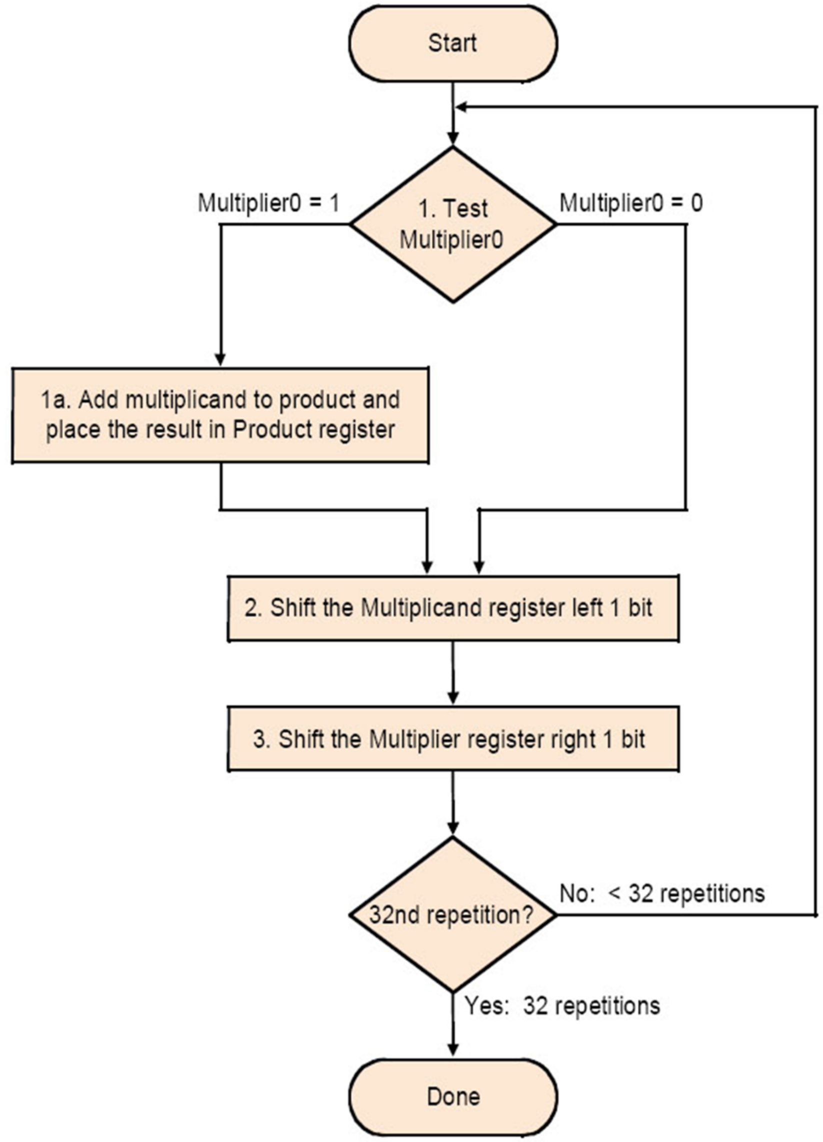
- More complicated than addition
  - A straightforward implementation will involve shifts and adds
- More complex operation can lead to
  - More area (on silicon) and/or
  - More time (multiple cycles or longer clock cycle time)
- Let′s begin from a simple, straightforward method

Straightforward Algorithm

```
          01010010 (multiplicand)
        x 01101101 (multiplier)
        -----------------------
          01010010
         00000000
        01010010
       01010010
      00000000
     01010010
    01010010
   00000000
------------------------------
   010001011101010
```

Implementation 1





Example (Implementation 1)

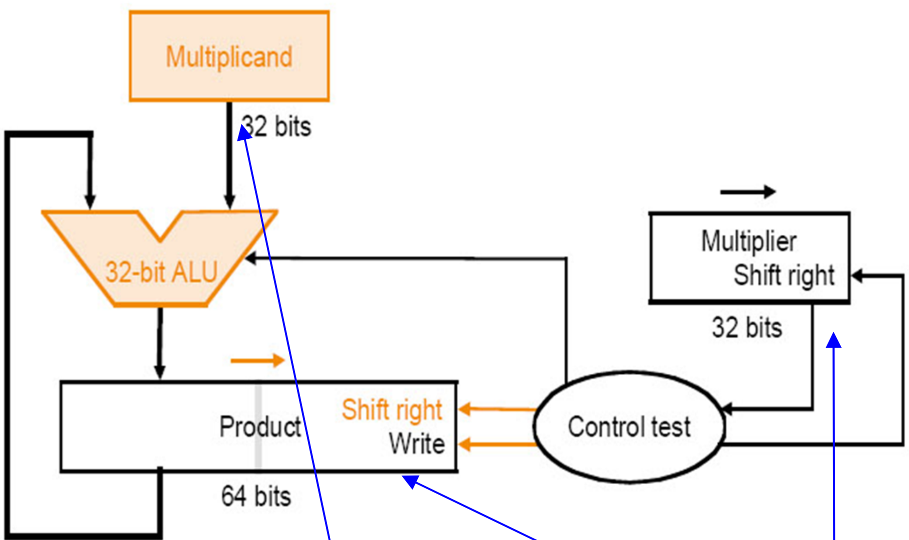
- Let's do 0010 x 0110 (2 x 6), unsigned

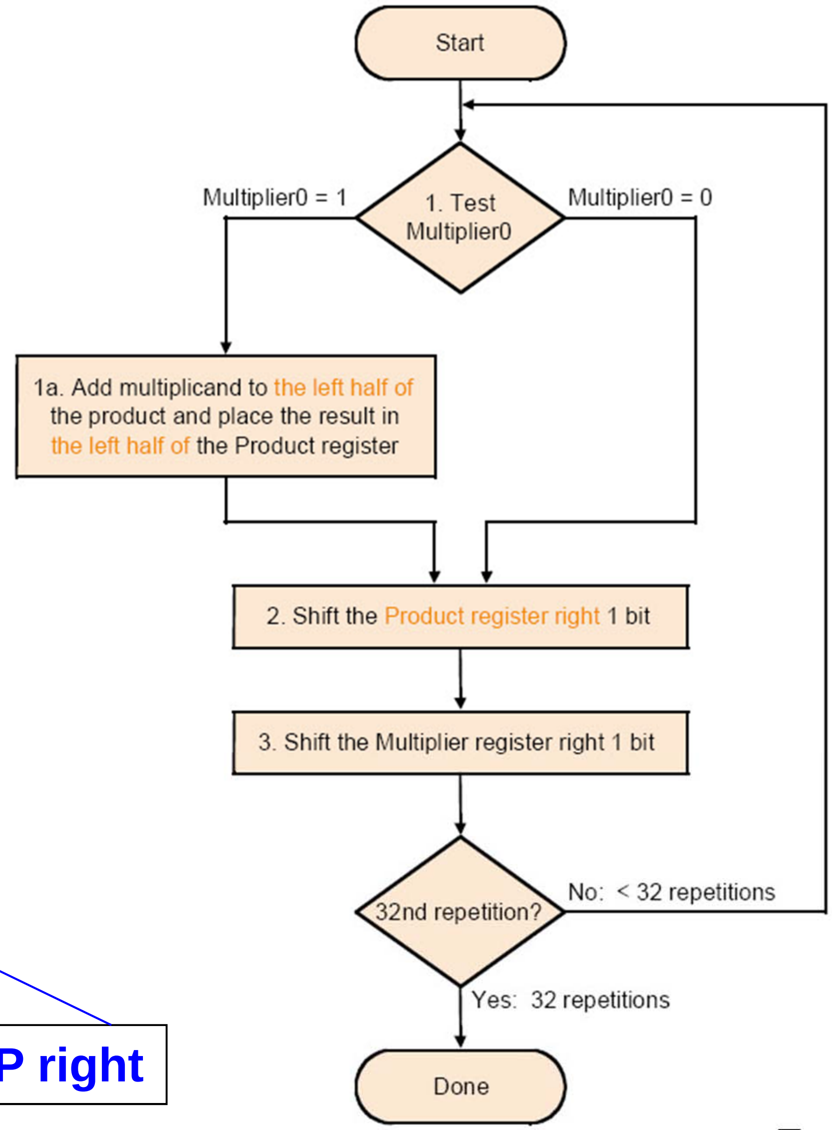
Iteration	Implementation 1			
	Step	**Multiplier**	**Multiplicand**	**Product**
0	initial values	0110	0000 0010	0000 0000
1	1: 0 -> no op	0110	0000 0010	0000 0000
	2: Multiplier shift right/ Multiplicand shift left	→ 011	0000 0100 ←	0000 0000
2	1: 1 -> product = product + multiplicand	011	0000 0100	0000 0100
	2: Multiplier shift right/ Multiplicand shift left	→ 01	0000 1000 ←	0000 0100
3	1: 1 -> product = product + multiplicand	01	0000 1000	0000 1100
	2: Multiplier shift right/ Multiplicand shift left	→ 0	0001 0000 ←	0000 1100
4	1: 0 -> no op	0	0001 0000	0000 1100
	2: Multiplier shift right/ Multiplicand shift left		0010 0000	

Drawbacks

- The ALU is twice as wide as necessary
- The multiplicand register takes twice as many bits as needed
- The product register won't need 2n bits till the last step
  - Being filled
- The multiplier register is being emptied during the process

Implementation 2





Multiplicand stationary - Multiplier right - PP right

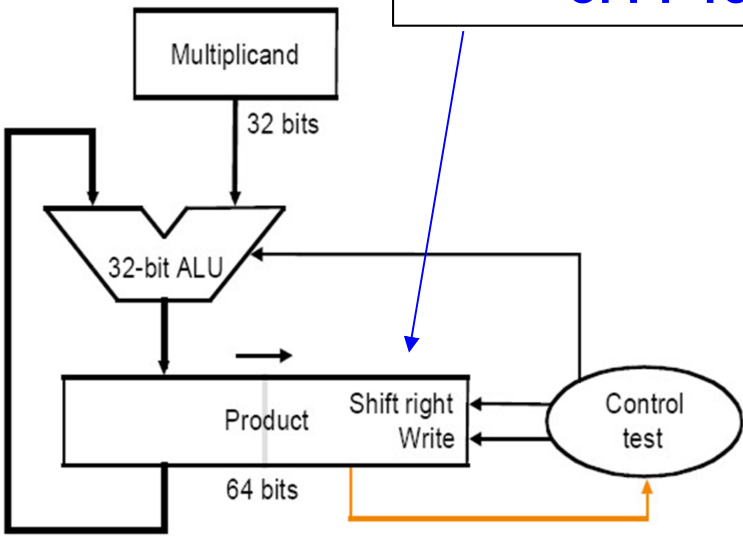
Example (Implementation 2)

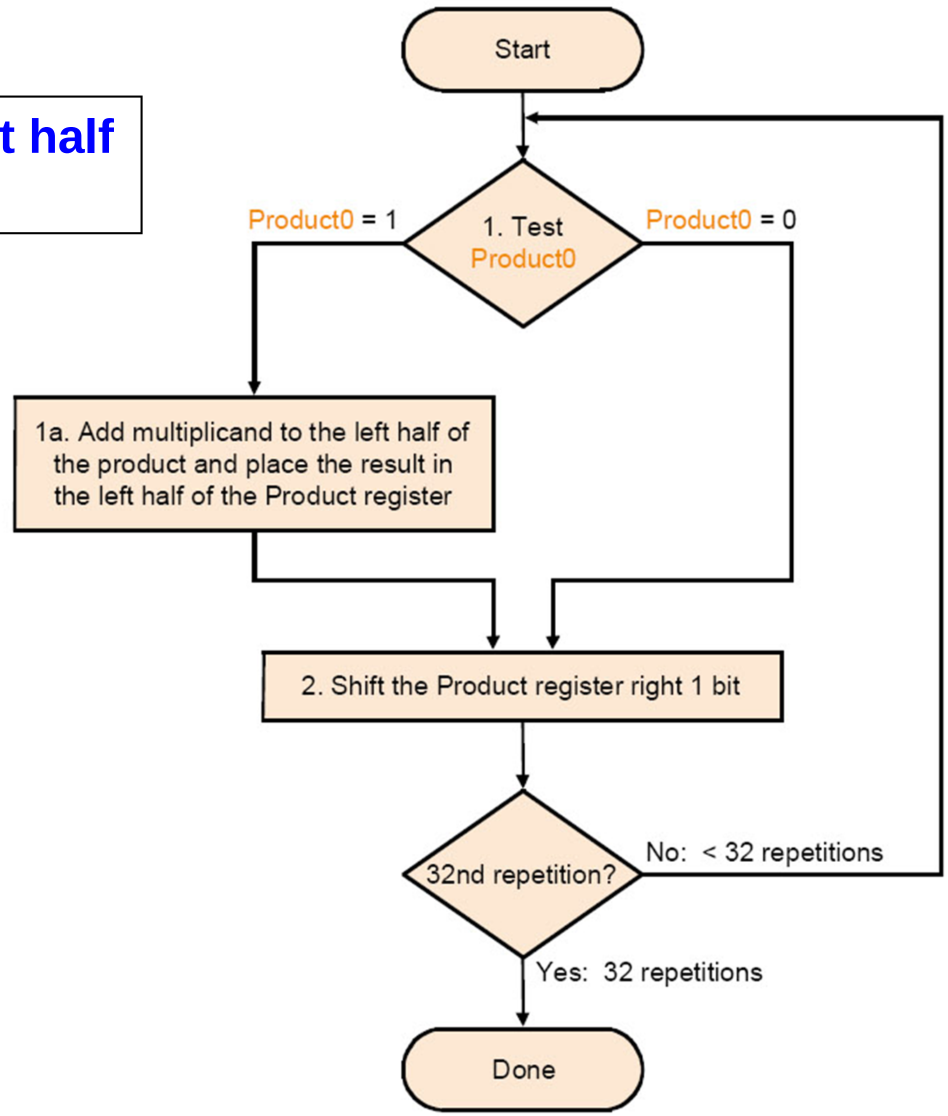
- Let′s do 0010 x 0110 (2 x 6), unsigned

Iteration	Implementation 2			
	Step	Multiplier	Multiplicand	Product
0	initial values	0110	0010	0000 ××××
1	1: 0 -> no op	0110	0010	0000 ××××
	2: Multiplier shift right/ Product shift right	×011	0010	0000 0×××
2	1: 1 -> product = product + multiplicand	×011	0010	0010 0×××
	2: Multiplier shift right/ Product shift right	××01	0010	0001 00××
3	1: 1 -> product = product + multiplicand	××01	0010	0011 00××
	2: Multiplier shift right/ Product shift right	×××0	0010	0001 100×
4	1: 0 -> no op	×××0	0010	0001 100×
	2: Multiplier shift right/ Product shift right	××××	0010	0000 1100

Implementation 3

Multiplier on right half of PP reg





Multiplicand stationary - Multiplier right - PP right

Example (Implementation 3)

- Let′s do 0010 x 0110 (2 x 6), unsigned

Iteration	Implementation 3			
	Step	Multiplier	Multiplicand	Product|Multiplier
0	initial values	0110	0010	0000 0110
1	1: 0 -> no op	0110	0010	0000 0110
	2: Multiplier shift right/ Product shift right	×011	0010	0000 0011
2	1: 1 -> product = product + multiplicand	×011	0010	0010 0011
	2: Multiplier shift right/ Product shift right	××01	0010	0001 0001
3	1: 1 -> product = product + multiplicand	××01	0010	0011 0001
	2: Multiplier shift right/ Product shift right	××00	0010	0001 1000
4	1: 0 -> no op	×××0	0010	0001 1000
	2: Multiplier shift right/ Product shift right	××××	0010	0000 1100

Example (signed)

- **Note:**
  - **Sign extension of partial product**
  - **If most significant bit of multiplier is 1, then *subtract***

00111_2 * 0111_2	11001_2 * 1001_2	00111_2 * 1001_2	11001_2 * 0111_2
00000 0111	00000 1001	00000 1001	00000 0111
+ 00111 0111	+ 11001 1001	+ 00111 1001	11001 0111
→ 00011 1011	→ 11100 1100	→ 00011 1100	11100 1011
+ 01010 1011	→ 11110 0110	→ 00001 1110	10101 1011
→ 00101 0101	→ 11111 0011	→ 00000 1111	11010 1101
+ 01100 0101	subtract	subtract	10011 1101
→ 00110 0010	00110 0011	11001 1111	11001 1110
→ 00011 0001	→ 00011 0001	→ 11100 1111	11100 1111

Booth′s Encoding

- Three symbols to represent numbers: 1, 0, -1
- -1 in 8 bits
  - 11111111 (two′s complement)
  - 0000000-1 (Booth′s encoding)
- 14 in 8 bits
  - 00001110 (two′s complement)
  - 000100-10 (Booth′s encoding)
- Bit transitions show Booth′s encoding
  - 0 to 0: 0
  - 0 to 1: -1
  - 1 to 1: 0
  - 1 to 0: 1
- Partial results are obtained by
  - Adding multiplicand
  - Adding 0
  - Subtracting multiplicand

Booth′s Algorithm Example

- Let′s do 0010 x 1101 (2 x -3)

Iteration	Implementation 3		
	Step	Multiplicand	Product
0	initial values	0010	0000 1101 0
1	10 -> product = product – multiplicand	0010	1110 1101 0
	shift right		1111 0110 1
2	01 -> product = product + multiplicand	0010	0001 0110 1
	shift right		0000 1011 0
3	10 -> product = product – multiplicand	0010	1110 1011 0
	shift right		1111 0101 1
4	11 -> no op	0010	1111 0101 1
	shift right		1111 1010 1

Why it works?

$b \times (a_{31}a_{30}....a_0)$

$= a_0 \times b \times 2^0 +$
$a_1 \times b \times 2^1 +$
$\ldots$
$a_{31} \times b \times 2^{31}$

$= (0 - a_0) \times b \times 2^0 +$
$(a_0 - a_1) \times b \times 2^1 +$
$\ldots$
$(a_{30} - a_{31}) \times b \times 2^{31} +$
$a_{31} \times b \times 2^{32}$

?

- **Booth's algorithm performs an addition when it encounters the first digit of a block of ones (0 1) and a subtraction when it encounters the end of the block (1 0). When the ones in a multiplier are grouped into long blocks, Booth's algorithm performs fewer additions and subtractions than the normal multiplication algorithm.**

Why it works?

- **Works for positive multiplier coz** a_{31} **is 0**
- **What happens for negative multipliers?** a_{31} **is 1**
  - **Remember that we are using 2's complement binary**

$$b \times (a_{31}a_{30}\ldots a_0) = \mathbf{b} \times (- a_{31} \times 2^{31} + a_{30} \times 2^{30} + \ldots a_0 \times 2^0)$$

Same derivation applies

Division

Integer Division

- Dividend = <u>Quotient</u> × Divisor + <u>Remainder</u>

  Q ? R ?

- How to do it using paper and pencil?

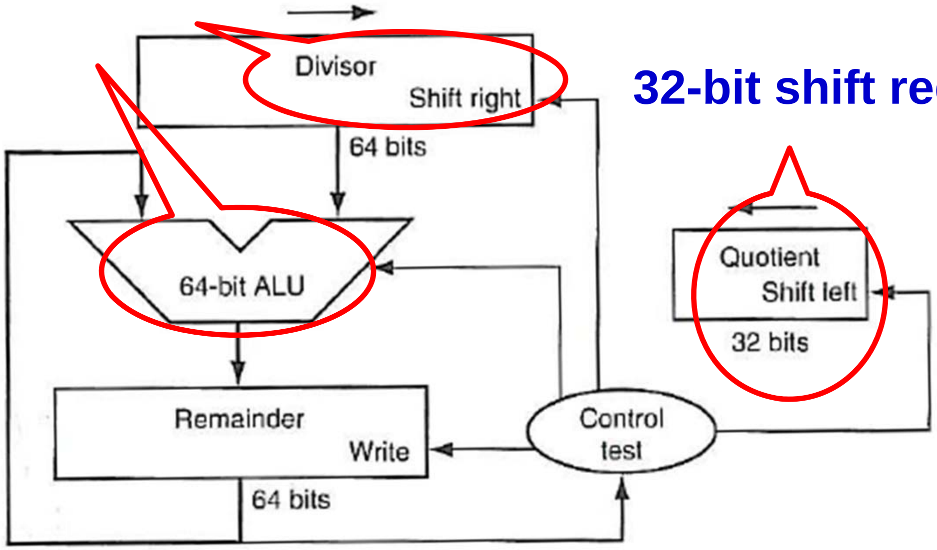
```
          1001
1000  | 1001010
       -1000
          10
          101
          1010
         -1000
            10
```

7÷2 = 3 … 1

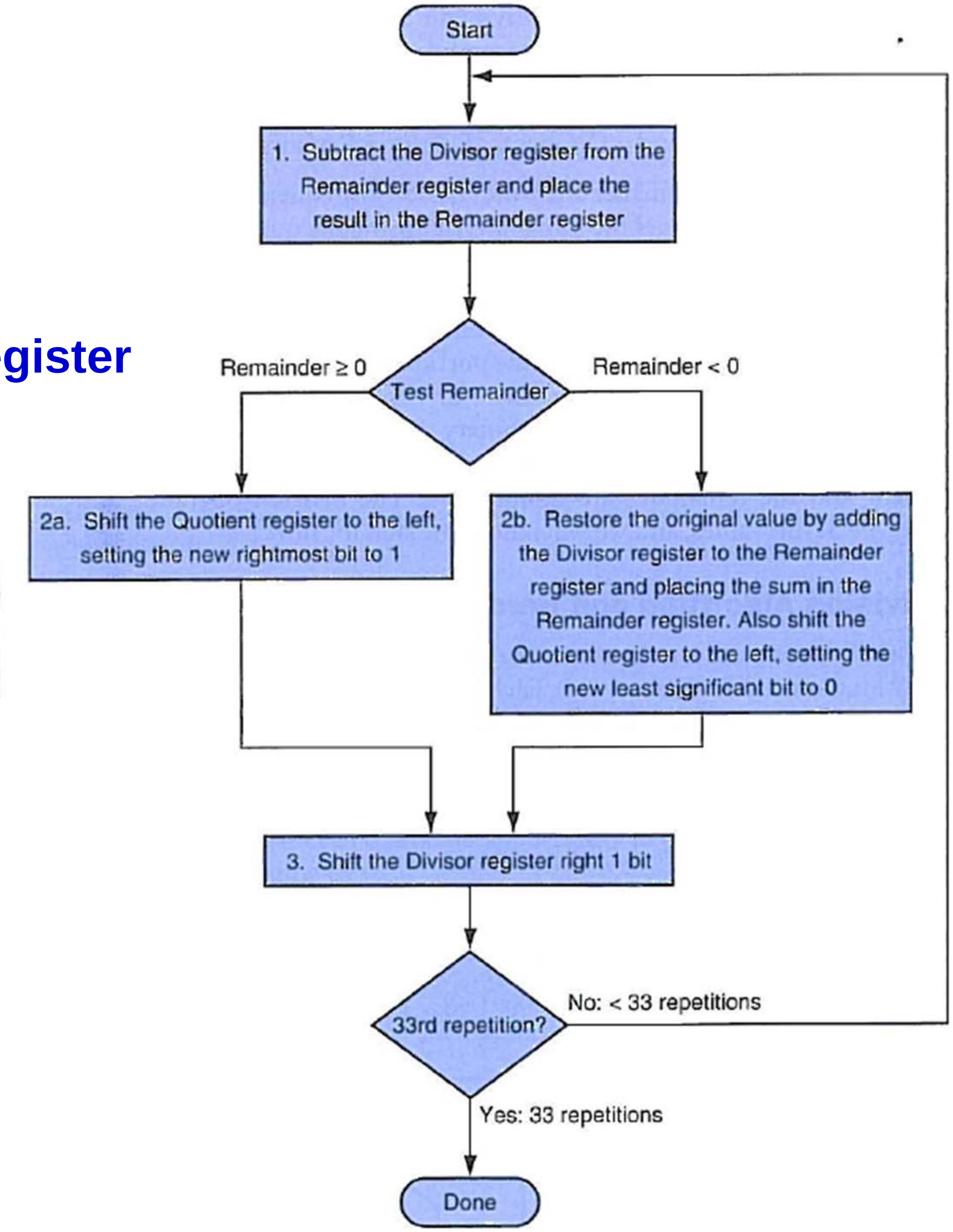
```
             11
0010  | 00000111
           -0010
           00011
            -0010
            0001
```

Implementation 1

64-bit wide



32-bit shift register



Example (7÷2)

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem=Rem-Div	0000	0010 0000	1110 0111
	2b: Rem<0=>+Div, sll Q, Q_0=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=>+Div, sll Q, Q_0=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=>+Div, sll Q, Q_0=0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem=Rem-Div	0000	0000 0100	0000 0011
	2a: Rem≥0=> sll Q, Q_0=1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem=Rem-Div	0001	0000 0010	0000 0001
	2a: Rem≥0=> sll Q, Q_0=1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Implementation 2