Computer Architecture and Organization

INSTRUCTOR: YAN-TSUNG PENG

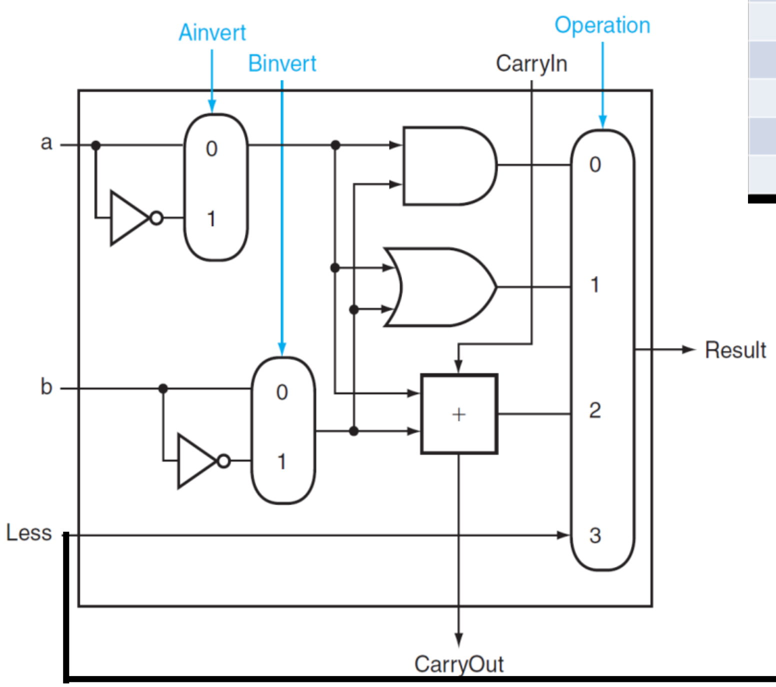
DEPT. OF COMPUTER SCIENCE, NCCU

Arithmetic for Computers: Basics

NATIONAL CHENGCHI UNIVERSITY

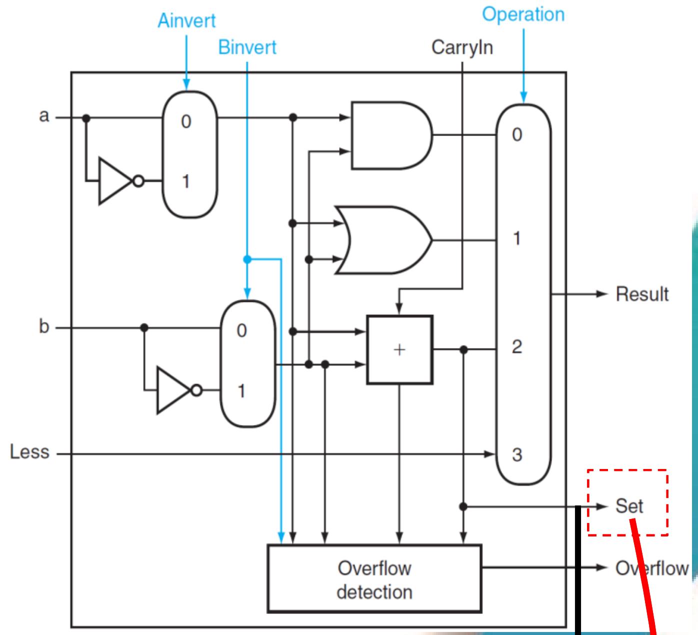
ALU (32-bit)

bit 0



ALU control lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

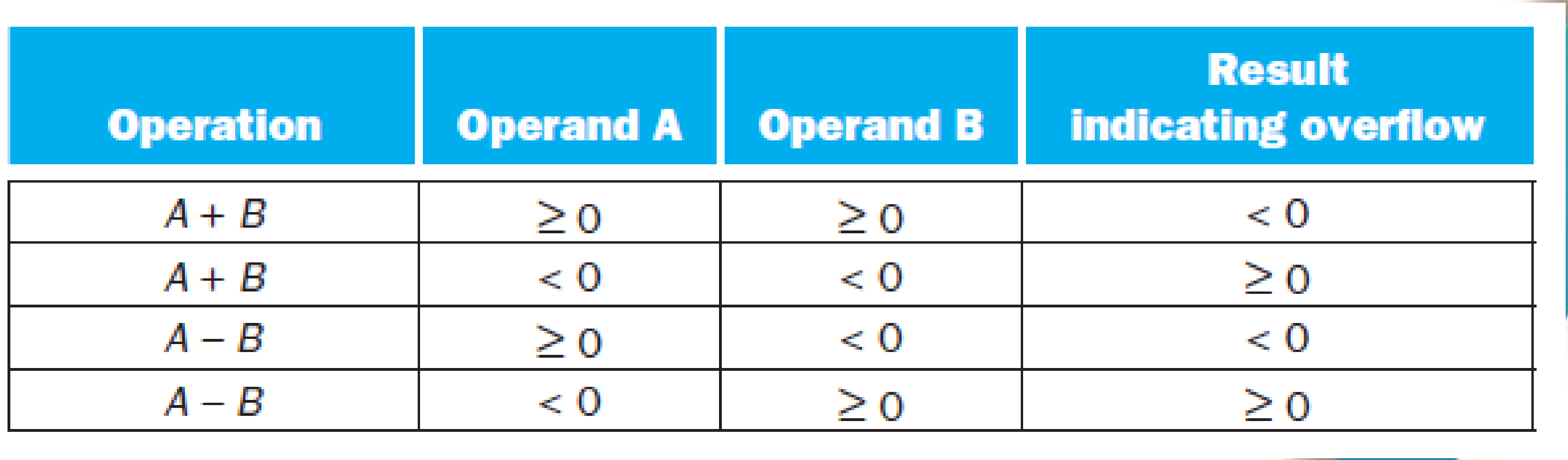
bit 31



$$\text{Less} = \begin{cases} 0 \text{ for bits } 1\text{–}31 \\ \textbf{Set} \text{ for bit } 0 \end{cases}$$

Arithmetic Overflow

- The condition occurs when a calculation of arithmetic operation(s) results in a result causing a given register to wrongly represents it



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

Overflow

- Some languages (e.g., C) ignore overflow
  - MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - MIPS `add`, `addi`, `sub` instructions
  - When an overflow occurs, invoke exception handler
    1. Store PC in EPC(Exception Program Counter)
    2. Jump to the predefined handler address
    3. `mfc0` (move from coprocessor reg) : jump back to where the overflow occurs

Coprocessor0 for Exception

```
# coprocessor0 registers

#  Name      Register Description     (*) simulated by MARS

# (*)BadVAddr  $8 offending memory reference
#     Count       $9 current timer ;incremented every 10ms
#     Compare   $11 interrupt when Count = Compare
# (*)Status       $12 controls which interrupts are enabled
# (*)Cause        $13 exception type, and pending interrupts
# (*)EPC           $14 PC where exception/interrupt occured
```

REF: http://msdl.cs.mcgill.ca/people/hv/teaching/ComputerArchitecture/lectures/exceptions_example.asm.txt

Example: Exception Handler

```
mfc0  $a0, $14      # coprocessor0 EPC register:
                    # address of instruction that caused exception
jal   print_hex

mfc0  $a0, $12      # coprocessor0 Status register
jal   print_hex

mfc0  $a0, $13      # coprocessor0 Cause register
jal   print_hex
```

Arithmetic Overflow

- Overflow occurs when
  - Sum of two positive numbers is negative
  - Sum of two negative numbers is positive
  - Overflow indicator correlates Cin with Cout (MSB)
- Overflow condition: $\mathrm{Cin} \neq \mathrm{Cout}$ for MSB
  - Ex: 4-bit addition $(-2^3 \sim 2^3 - 1)$

Cin

$$\begin{array}{rll} & 0\,1\,0\,1 & \rightarrow 5 \\ + & 0\,1\,1\,0 & \rightarrow 6 \\ \hline & 0\,1\,0\,1\,1 & \end{array}$$

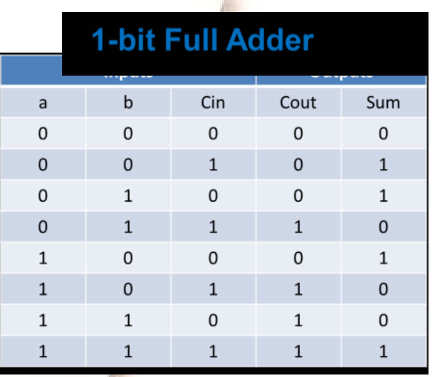
Cout

Cin

$$\begin{array}{rll} & 1\,0\,1\,1 & \rightarrow -5 \\ + & 1\,0\,1\,0 & \rightarrow -6 \\ \hline & 1\,0\,1\,0\,1 & \end{array}$$

Cout

Why not checking MSB and Cout? Think about -5+6



1-bit Full Adder

a	b	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Overflow

Overflow = XOR(Cin, Cout)

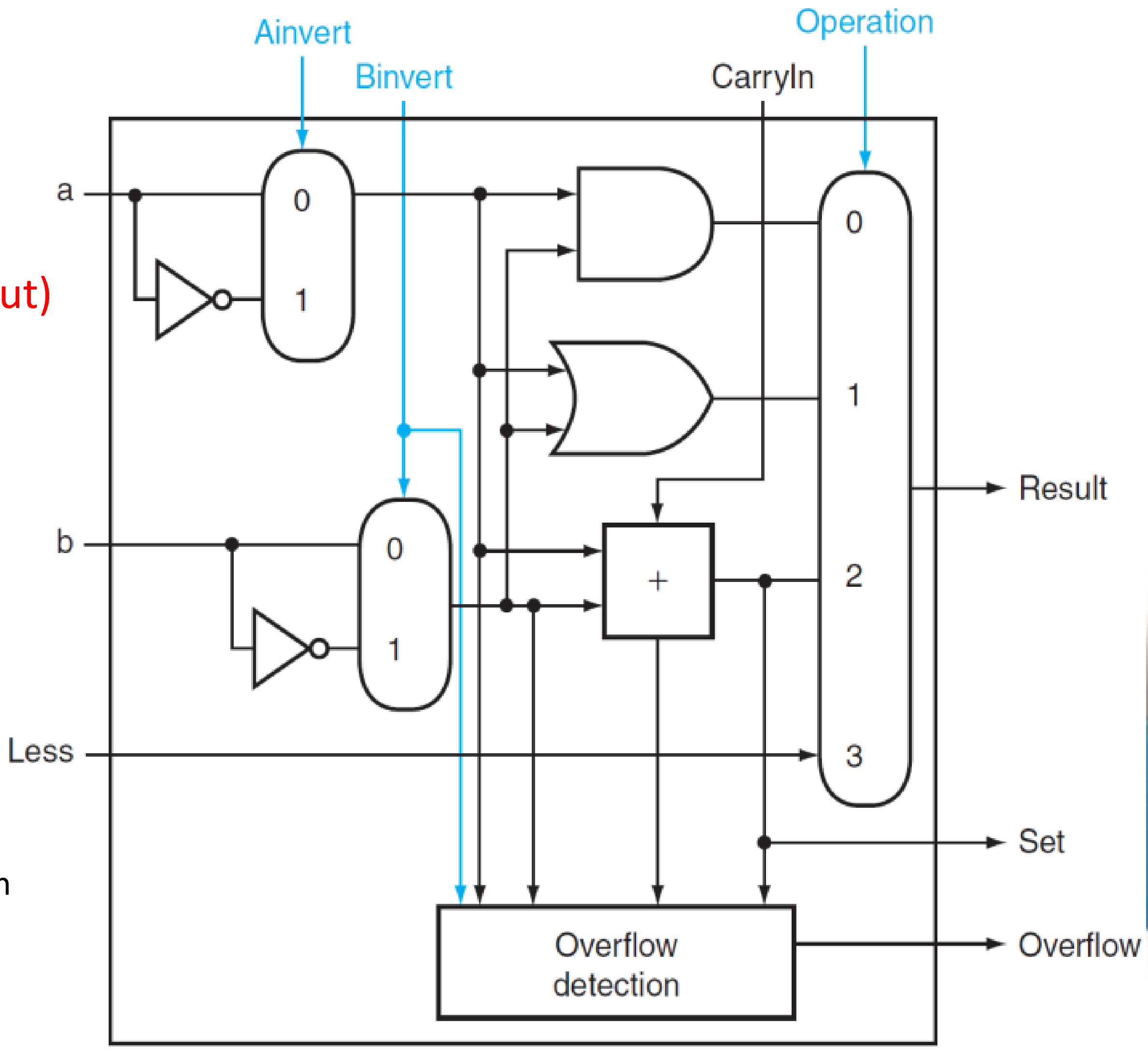
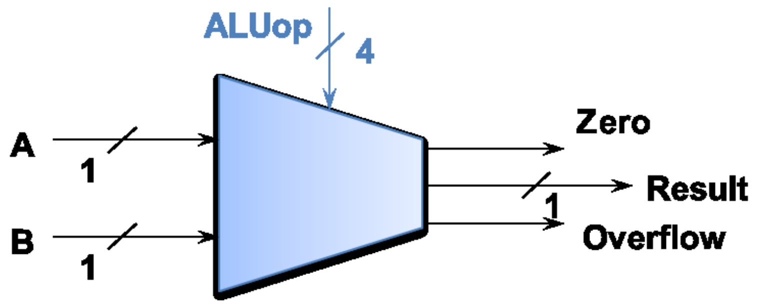
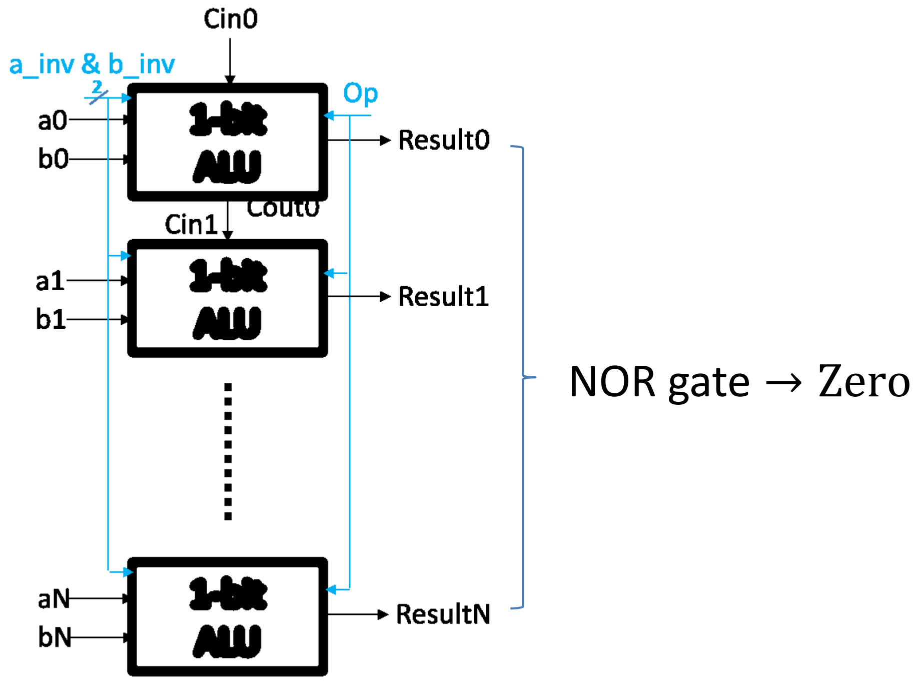


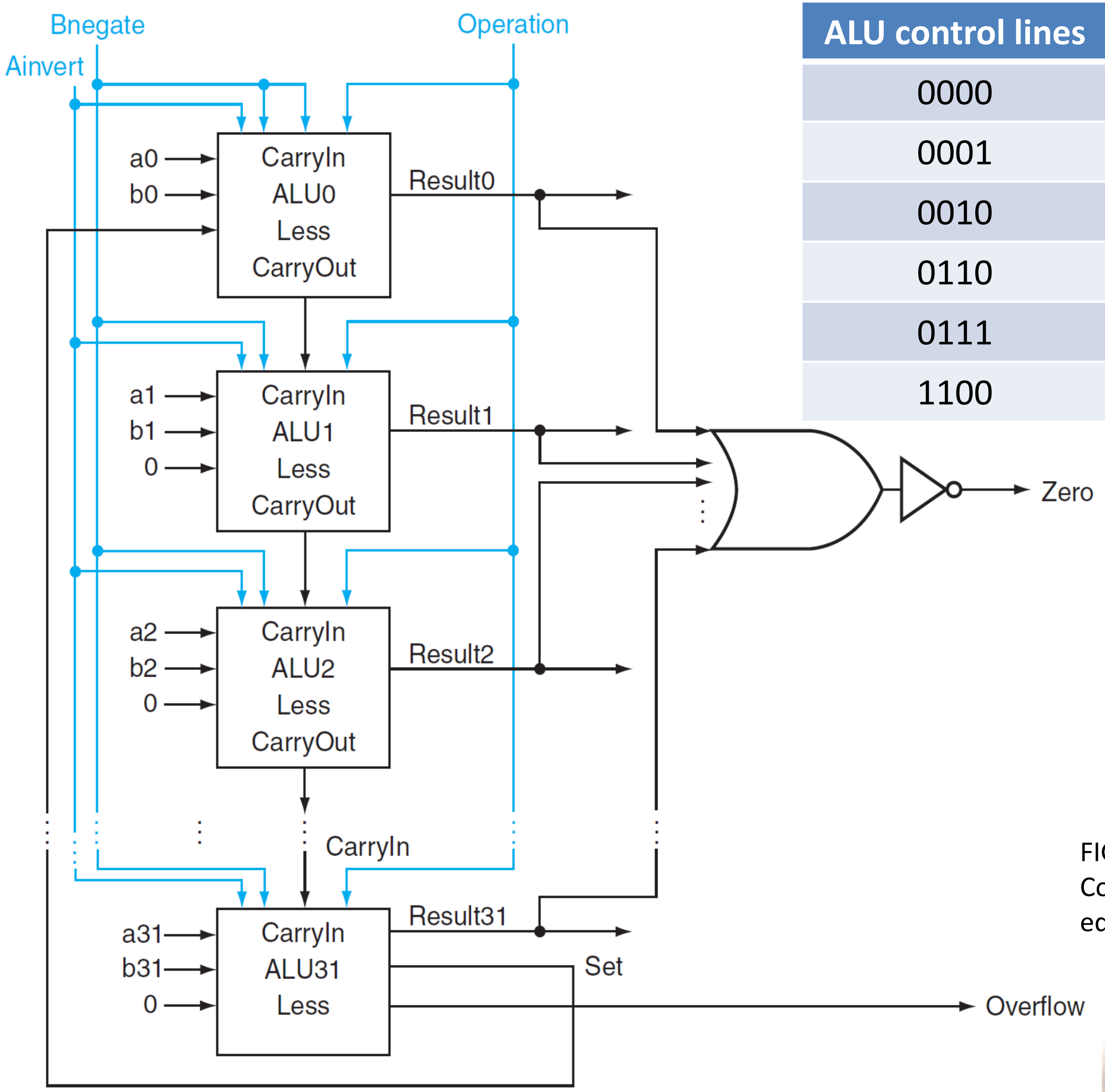
FIGURE B.5.10 from Computer Org. and Design, 5th edition

Equality Test

- Support for branch equal operations
- Zero $= (\text{Result0} + \text{Result1} + \cdots + \text{Result31})'$



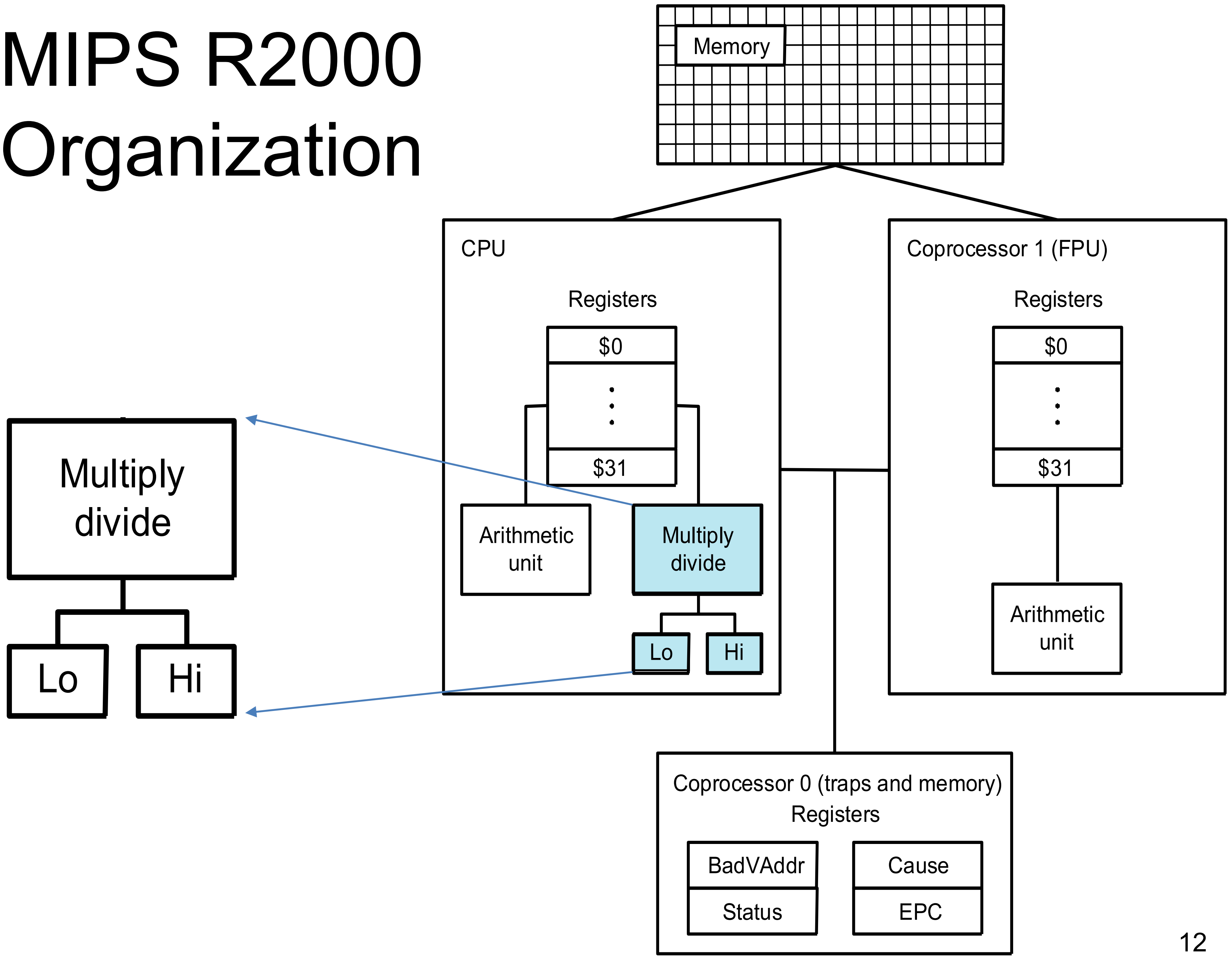




ALU control lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

FIGURE B.5.12 from Computer Org. and Design, 5th edition

MIPS R2000 Organization



Multiplication in MIPS

- Previously, we use sll (**shift left logical**) or srl (**shift right logical**) for multiplication
  - ex: sll $t0, $s0, 2   # $t0 = $s0 * 2^2
- Multiplication without overflow: mul $t1, $t2, $t3
  - Set HI to high-order 32 bits and LO and $t1 to low-order 32 bits of the product of $t2 and $t3

Two 32-bit registers for product

    HI: most-significant 32 bits
    LO: least-significant 32-bits

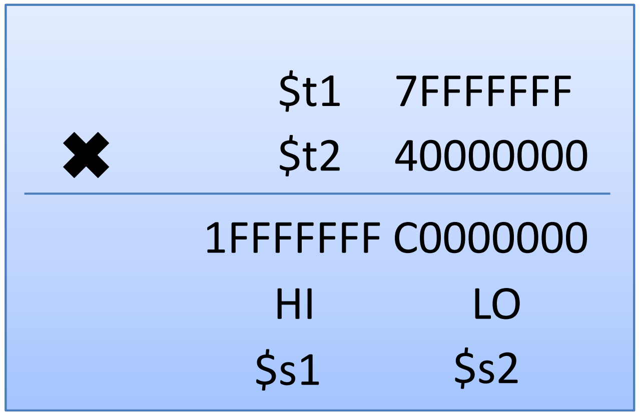
Instructions

    mult rs, rt / multu rs, rt
        64-bit product in HI/LO
    mfhi rd / mflo rd
        Move from HI/LO to rd
    mul rd, rs, rt (when the product only takes 32 bits)
        Least-significant 32 bits of product –> rd

Multiplication in MIPS

- `mult $t1, $t2 # perform $t1×$t2`
- It's a 32-bit value multiplied by another 32-bit value. The product needs to take 64 bits.
- 3-step process

1. mult $t1, $t2
2. mfhi $s1
3. mflo $s2



Unsigned Multiplication

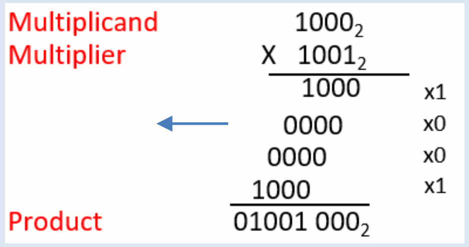
- Example

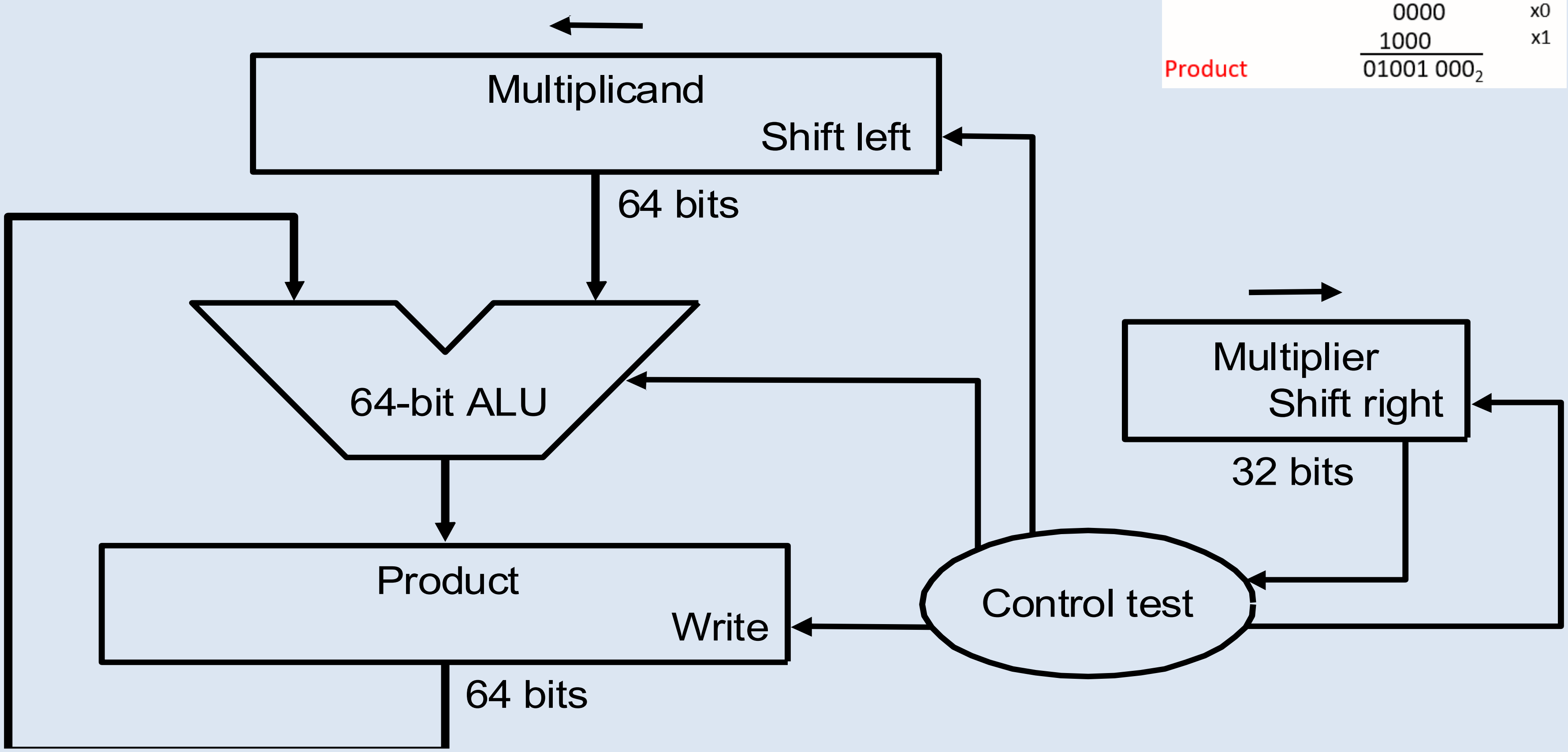
$$\begin{array}{lrl}
\text{Multiplicand} & 1000_2 & \\
\text{Multiplier} & \times \quad 1001_2 & \\
\cline{2-2}
 & 1000 & \text{x1} \\
 & 0000 & \text{x0} \\
 & 0000 & \text{x0} \\
 & 1000 & \text{x1} \\
\cline{2-2}
\text{Product} & 01001\;000_2 &
\end{array}$$

- Total bits: m bits x n bits = m+n bit product
  - 0 => place 0 (0 x multiplicand)
  - 1 => place multiplicand (1 x multiplicand)

Unsigned Multiplier

- 32-bit multiplication hardware





4-bit Multiplication Hardware

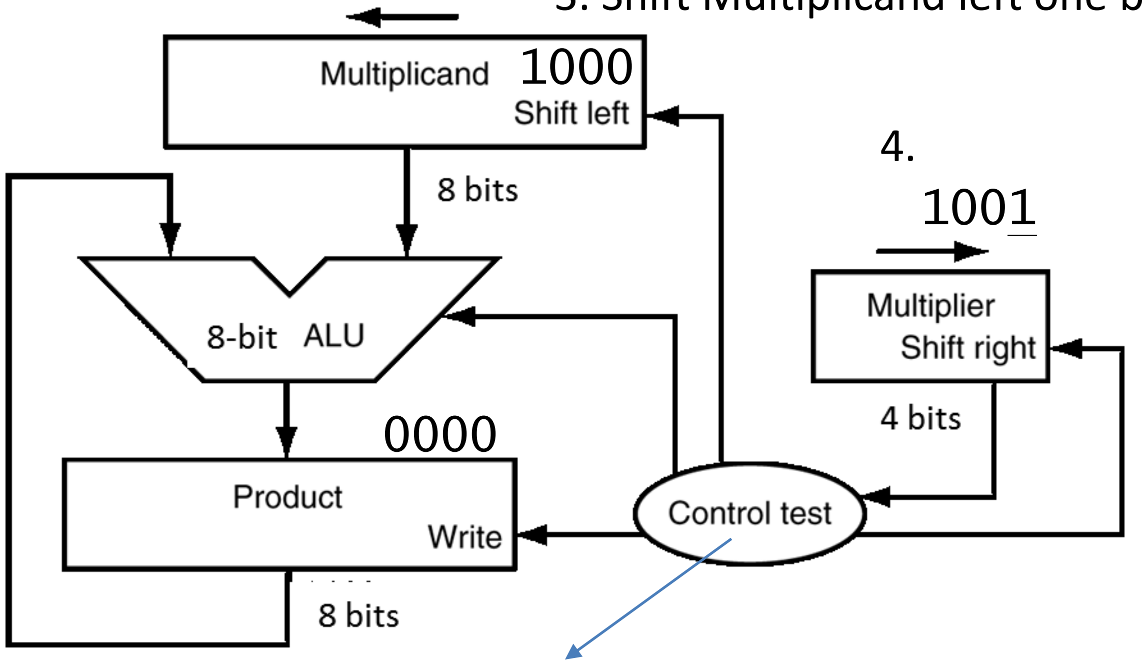
multiplicand

multiplier

product

```
   1000
×  1001
   1000
  0000
 0000
1000
1001000
```

3. Shift Multiplicand left one bit

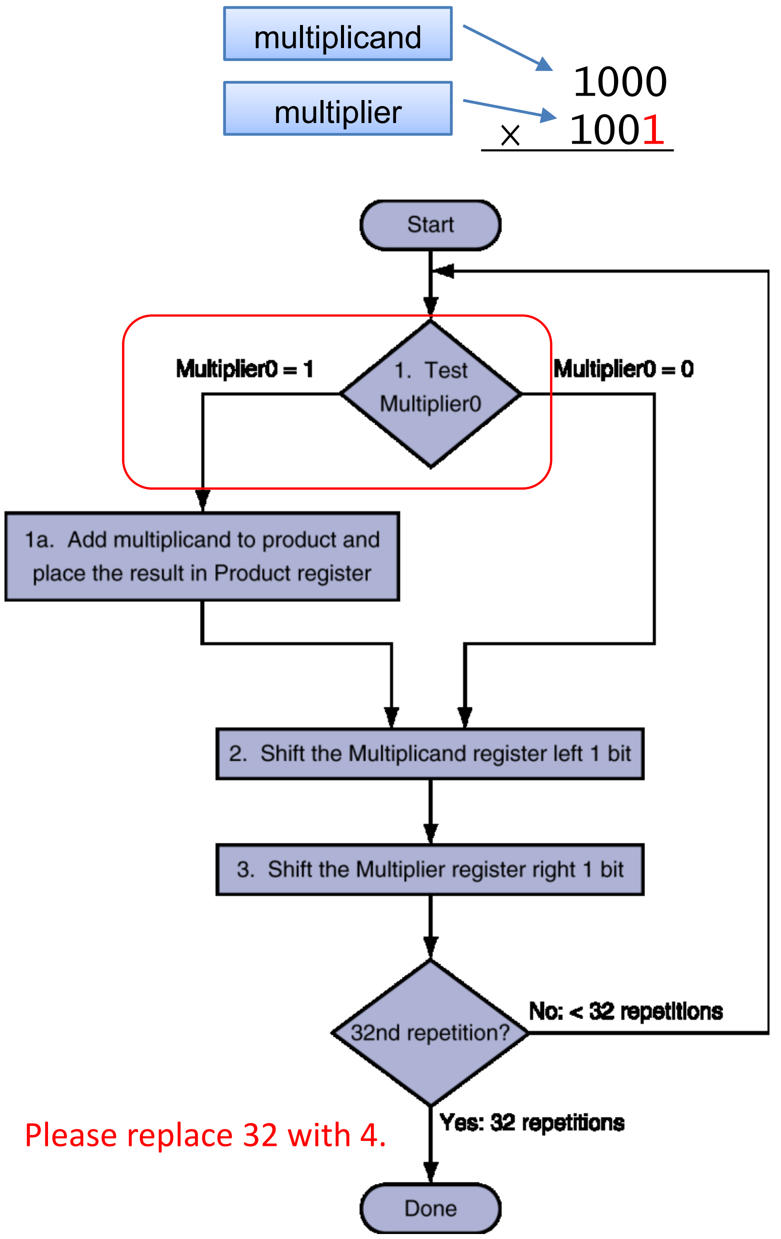


1. If lsb of multiplier = 0 → do nothing
   else (lsb of multiplier = 1)→ add multiplicand to product

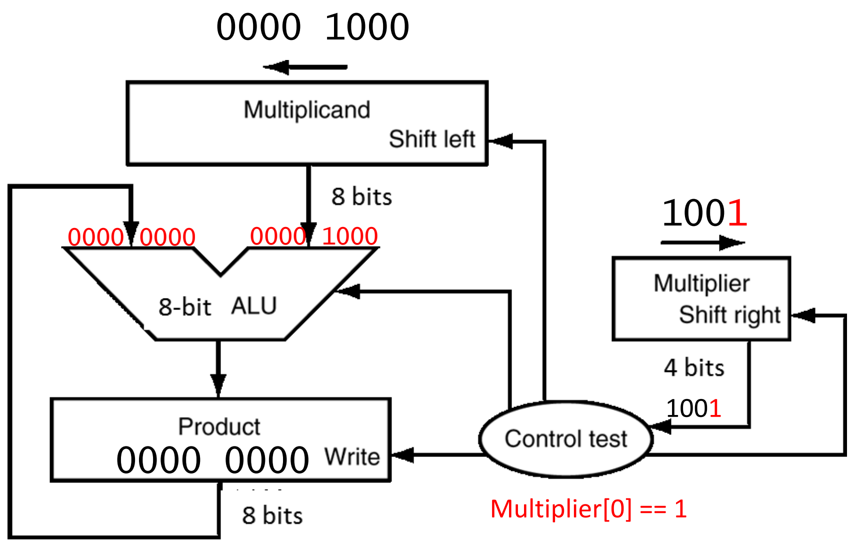
2. Sum multiplicand and product

4-bit Multiplication Breakdown

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 $\Longrightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 $\Longrightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 $\Longrightarrow$ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 $\Longrightarrow$ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

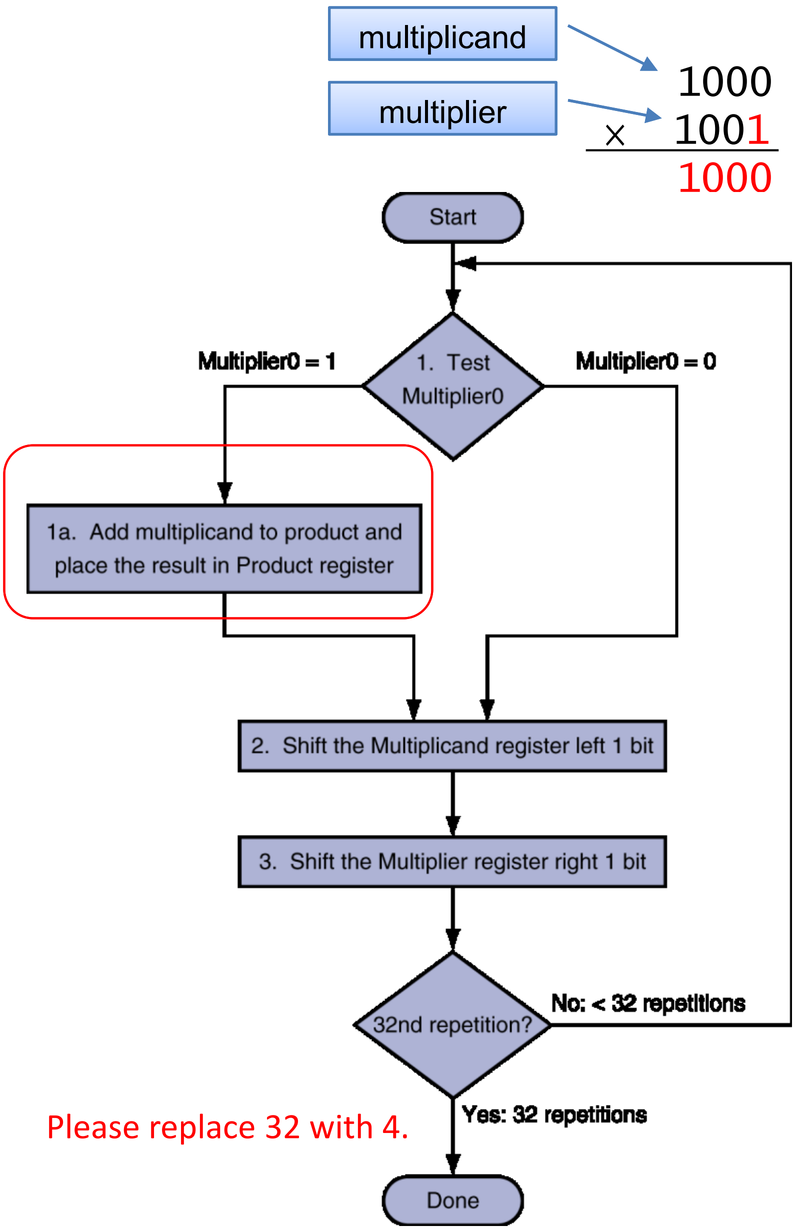


multiplicand
multiplier
1000
× 1001
Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



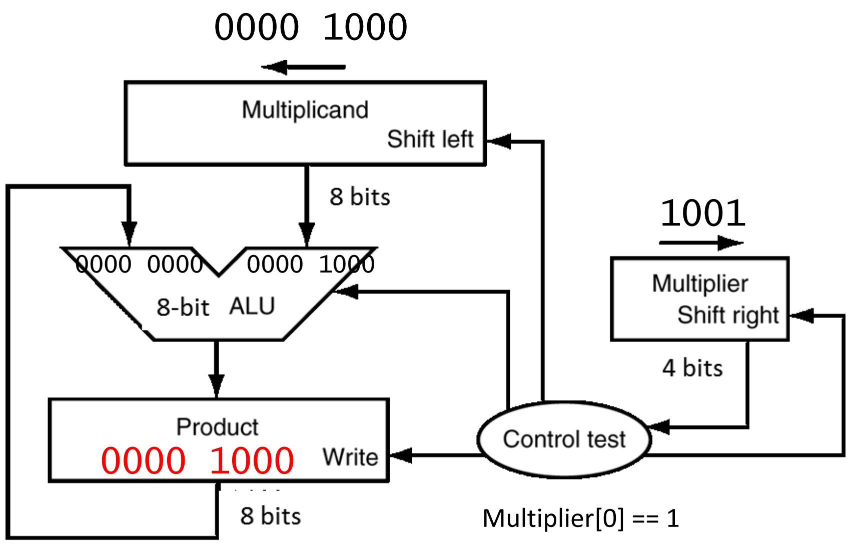
Repetition=1

Please replace 32 with 4.

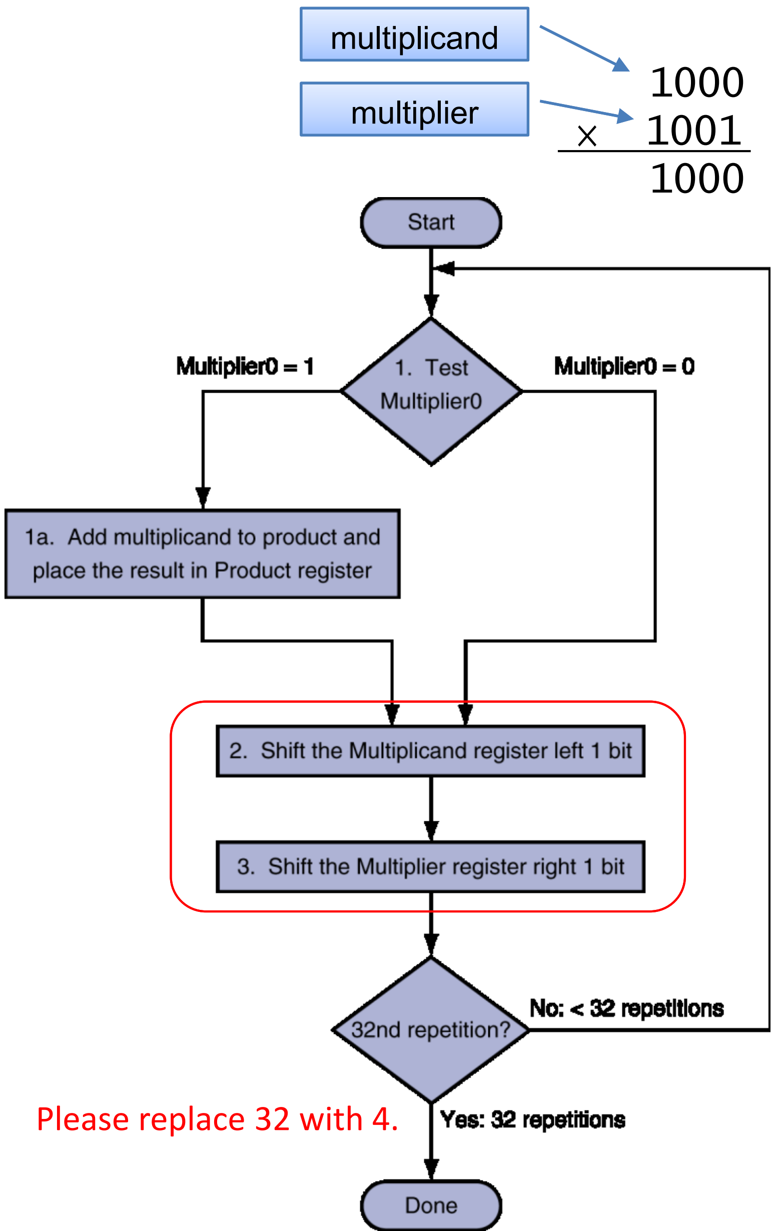


multiplicand
multiplier
1000
× 1001
1000
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Please replace 32 with 4.

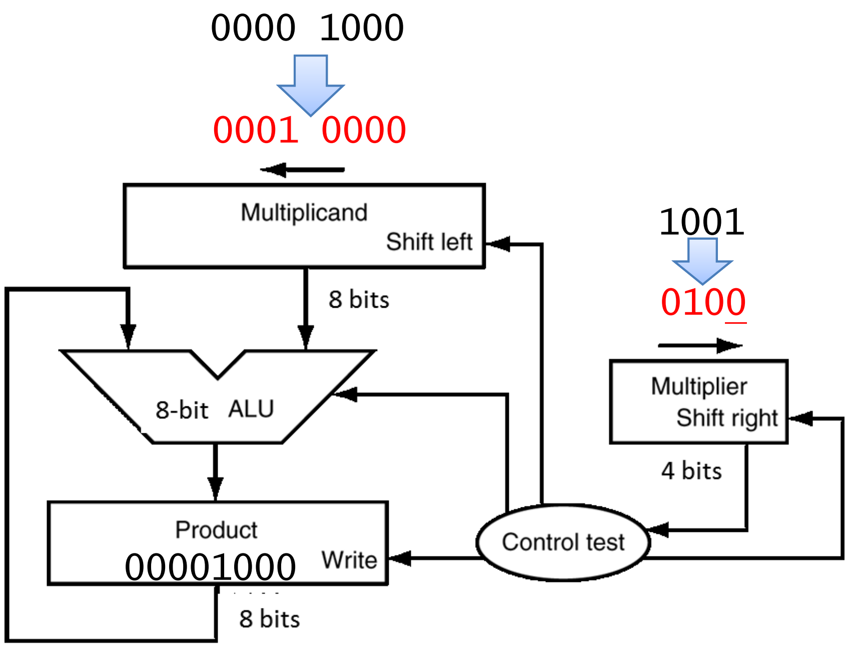


Repetition=1

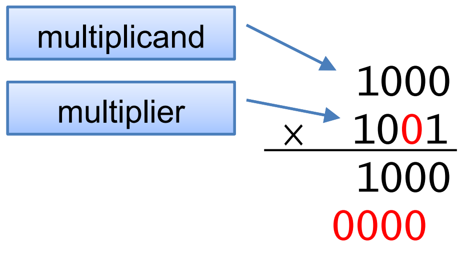


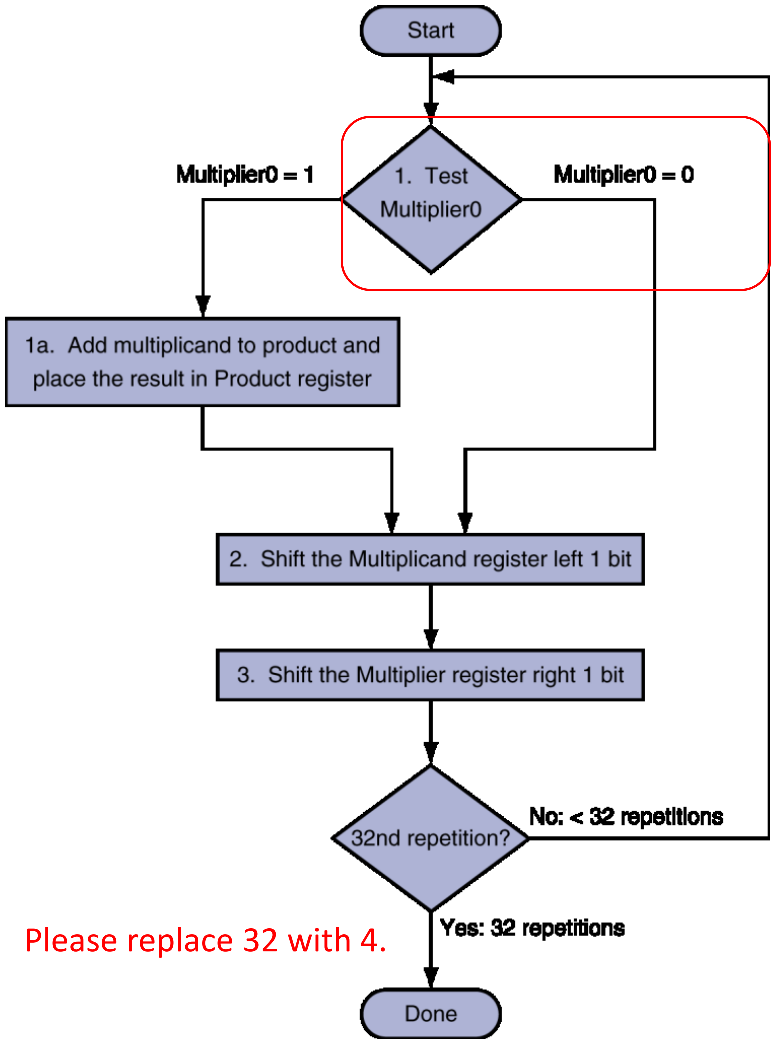
multiplicand
multiplier
1000
× 1001
1000
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

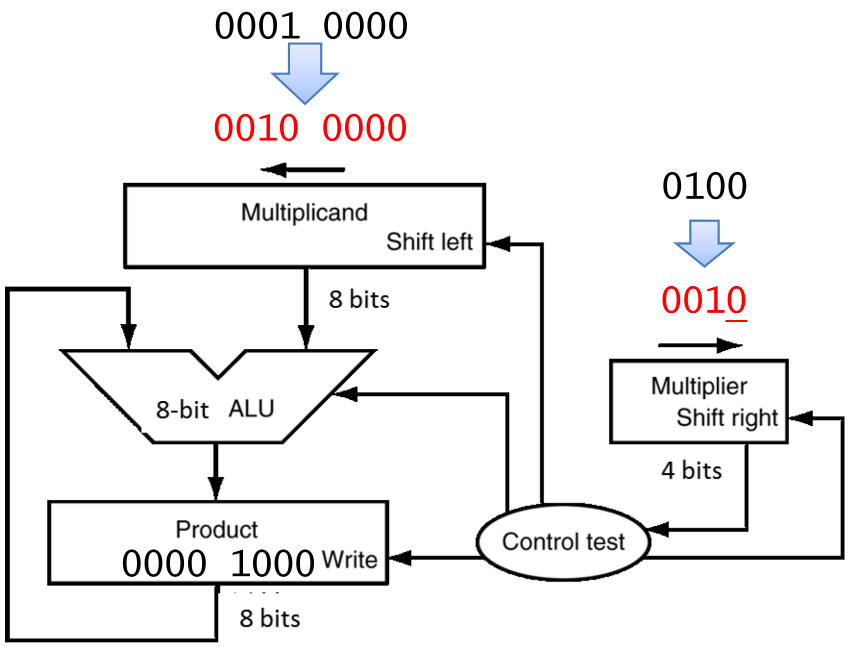
Please replace 32 with 4.



Repetition=1




Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



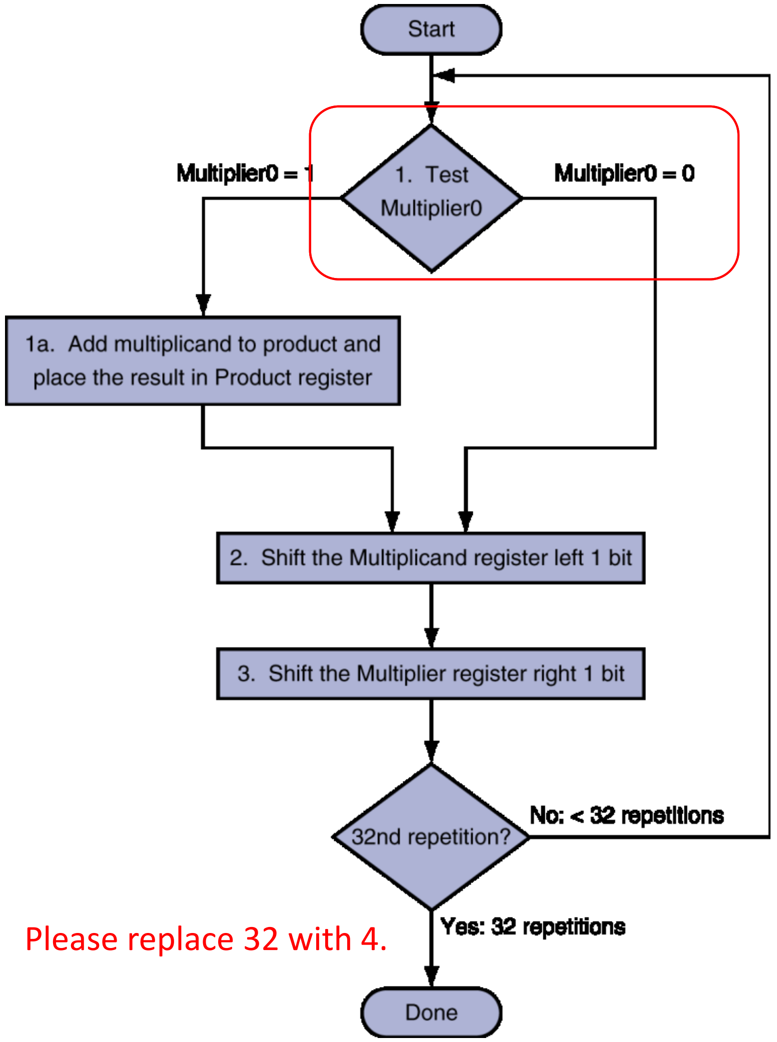
Repetition=2

Please replace 32 with 4.

multiplicand

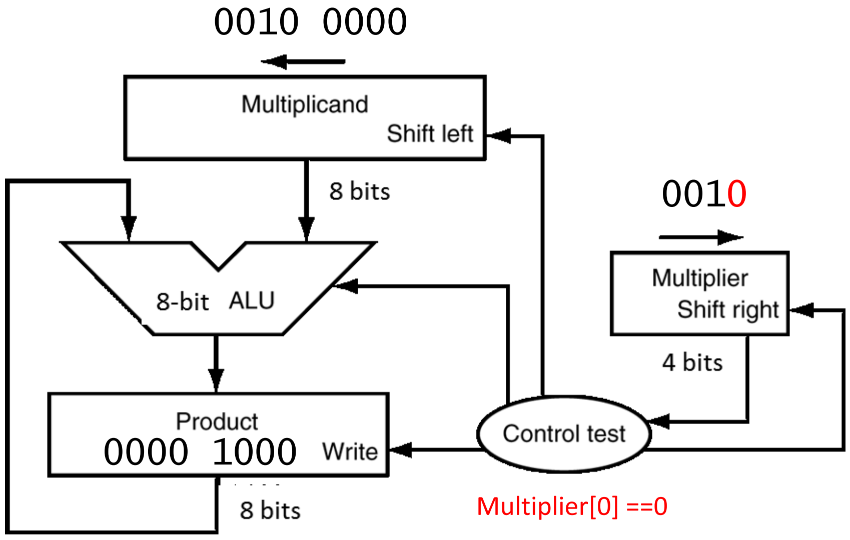
multiplier

```
      1000
×     1001
      1000
     0000
    0000
```



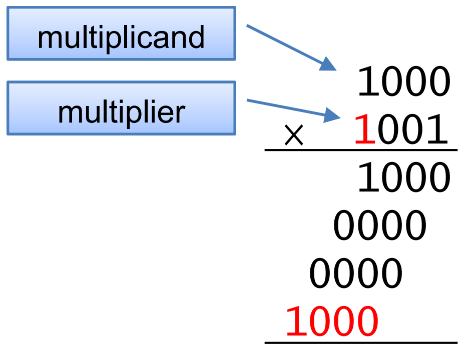
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

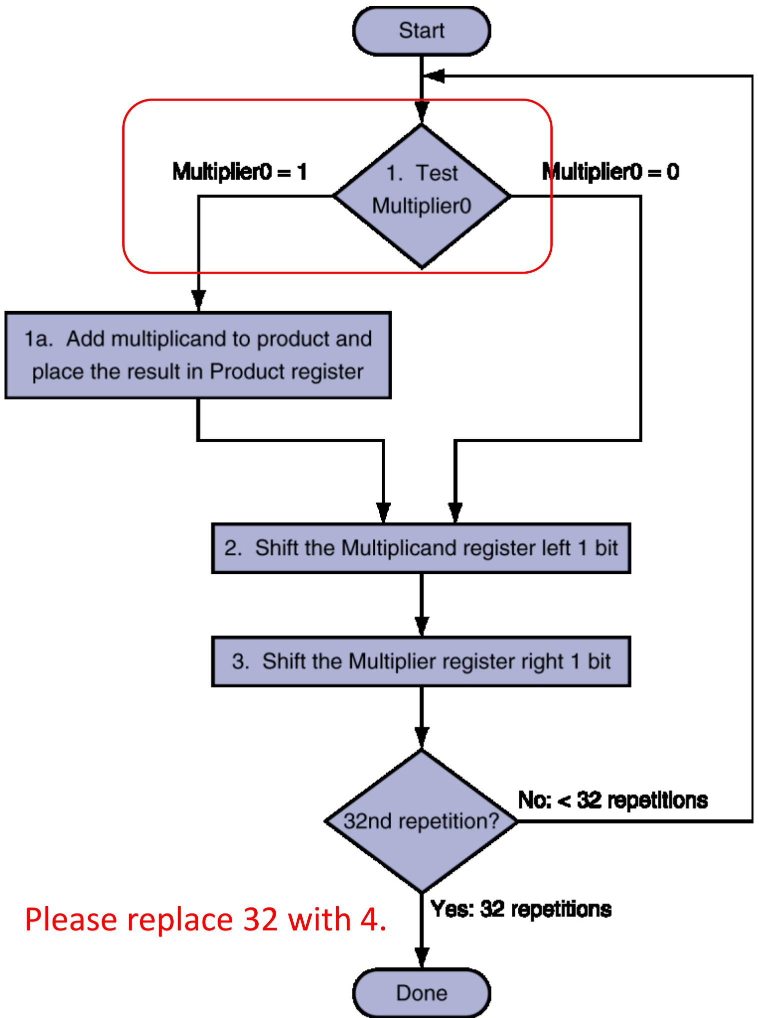
Please replace 32 with 4.

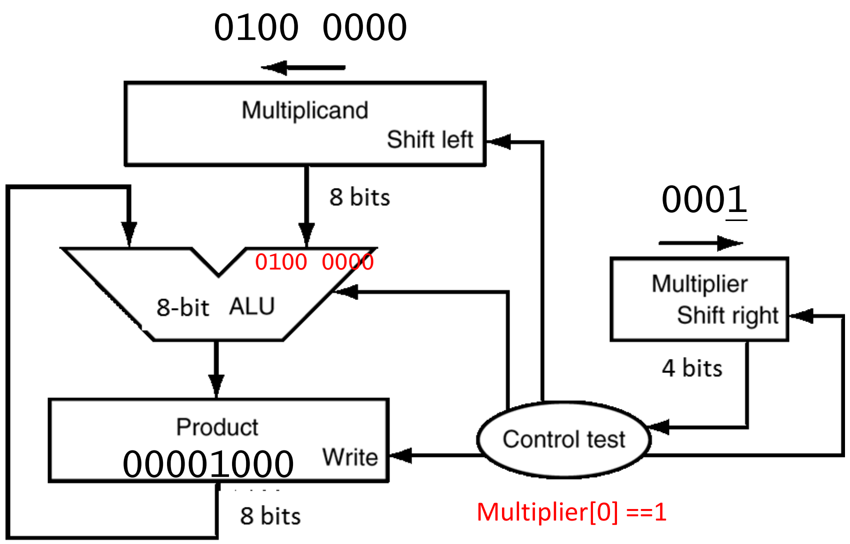


Multiplier[0] ==0

Repetition=3



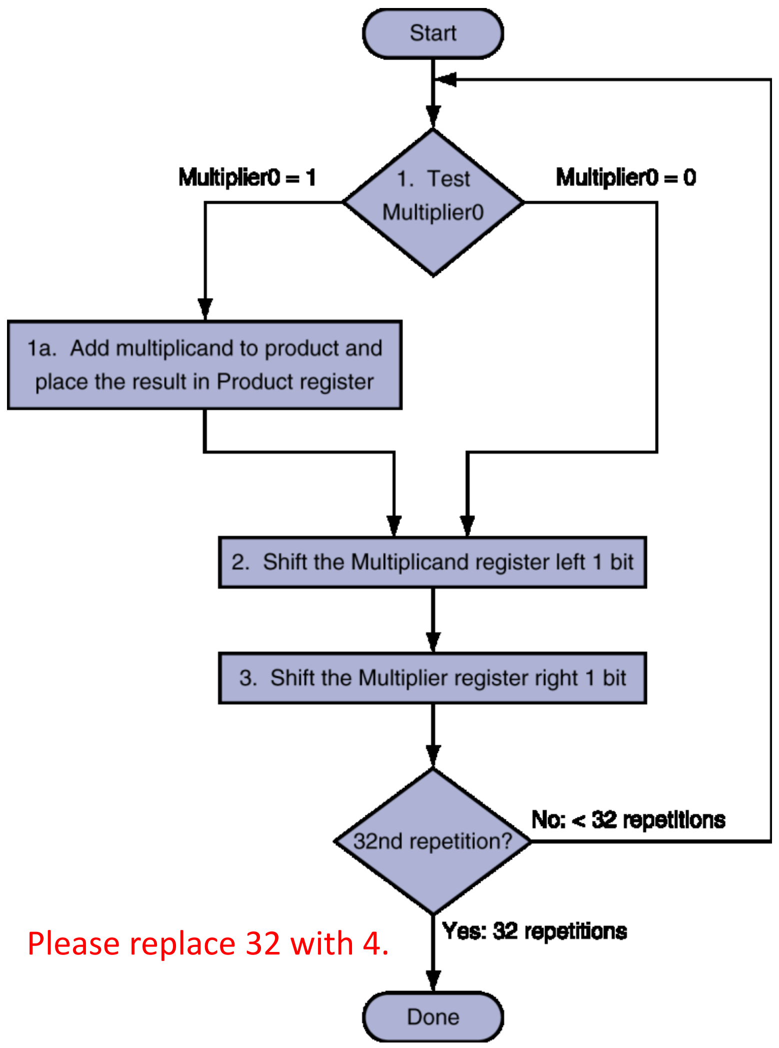

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



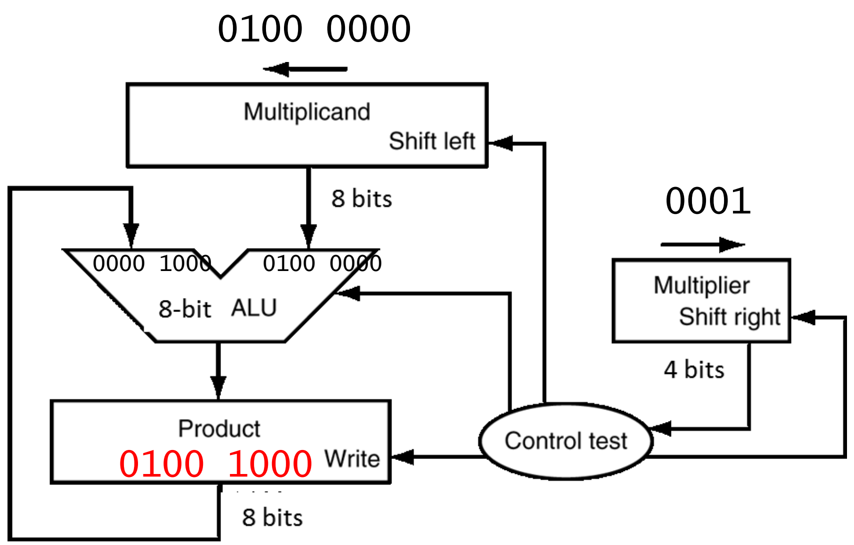
Repetition=4

Please replace 32 with 4.

```
      1000
  ×   1001
  --------
      1000
     0000
    0000
   1000
  --------
   1001000
```



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

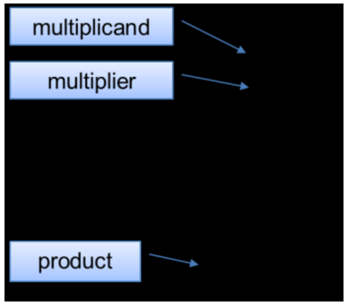


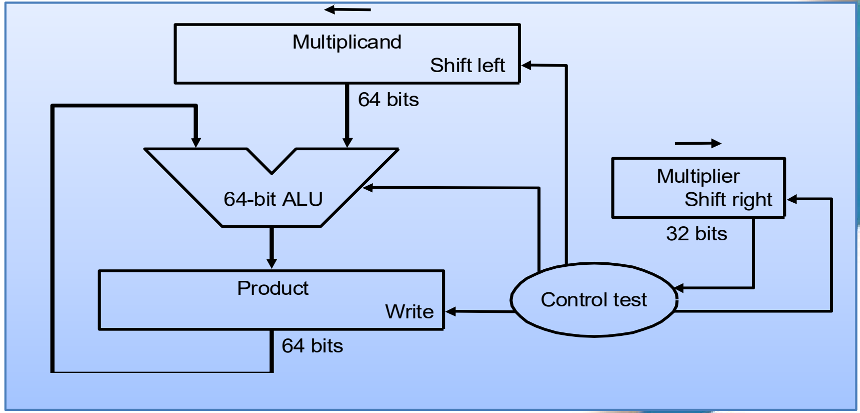
Repetition=4

Please replace 32 with 4.

Some Observations

- Multiplication needs more clock cycles than Addition
- That Multiplicand has only 32 bits but takes 64 bits is a waste
- Once computed, LSB of product would not change
- LSB of Multiplier can be dropped once checked (computed)
- Shifting Product to right instead of shifting Multiplicand to left





Improved Design

4-bit multiplier

Lower cost with less hardware used

Multiplier

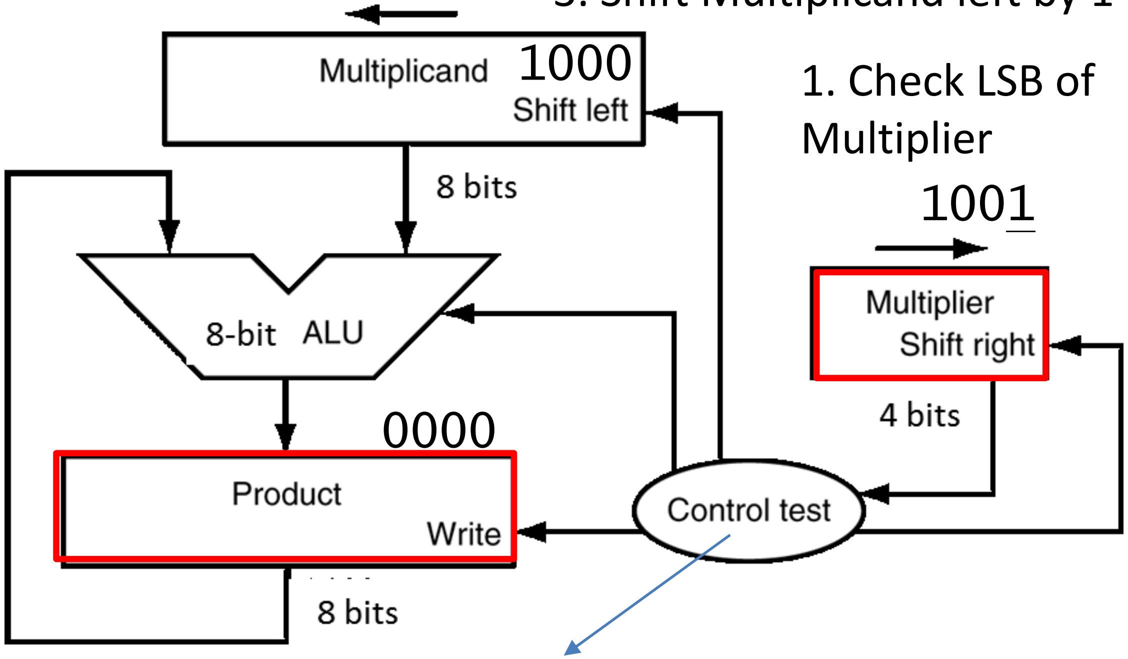
Step 1:	100$\underline{1}$
Step 2:	10$\underline{0}$
Step 3:	1$\underline{0}$
Step 4:	$\underline{1}$

Valid digits for product

$$
\begin{array}{lr}
 & 1000 \\
\times & 1001 \\
\hline
4 \rightarrow & 1000 \\
5 \rightarrow & 0000 \\
6 \rightarrow & 0000 \\
7 \rightarrow & 1000 \\
\hline
 & 1001000
\end{array}
$$

3. Shift Multiplicand left by 1

1. Check LSB of Multiplier



2. LSB=0→Do nothing
LSB=1→Add Multiplicand to Product

Improved Design 4-bit multiplier

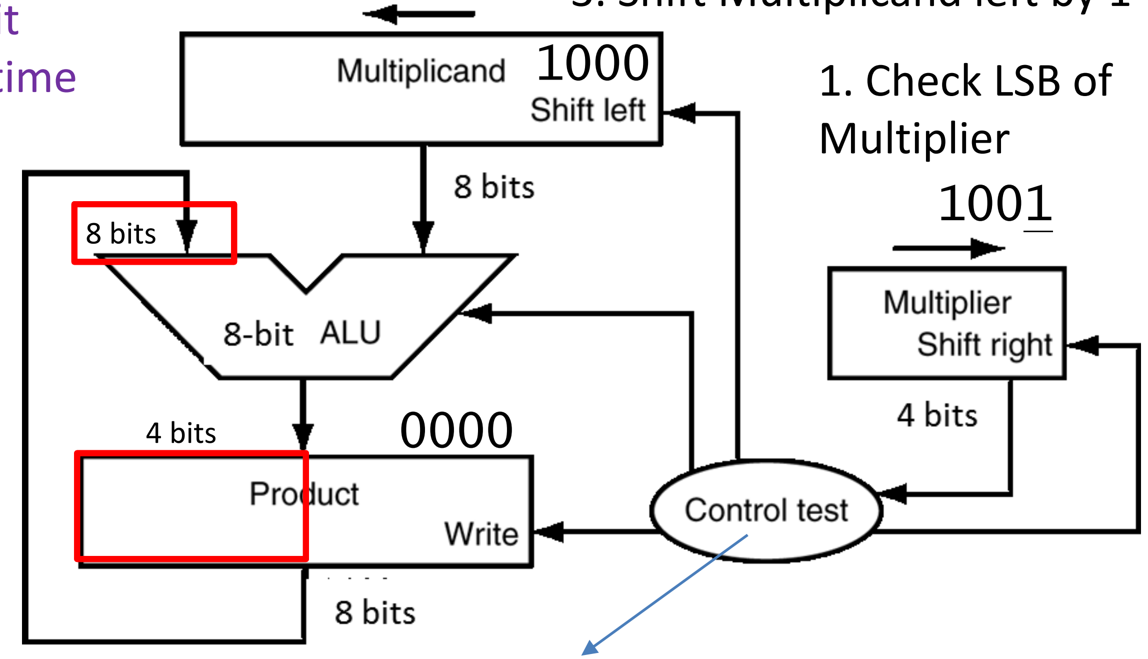
Lower cost with less hardware used

Observation: only 4-bit addition needed at a time

```
      1000
×     1001
----------
      1000
     0000
    0000
   1000
----------
   1001000
```

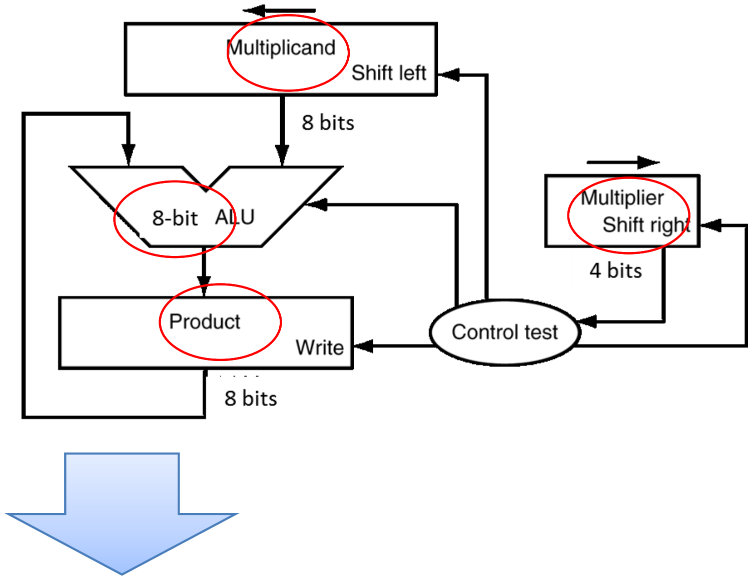
3. Shift Multiplicand left by 1

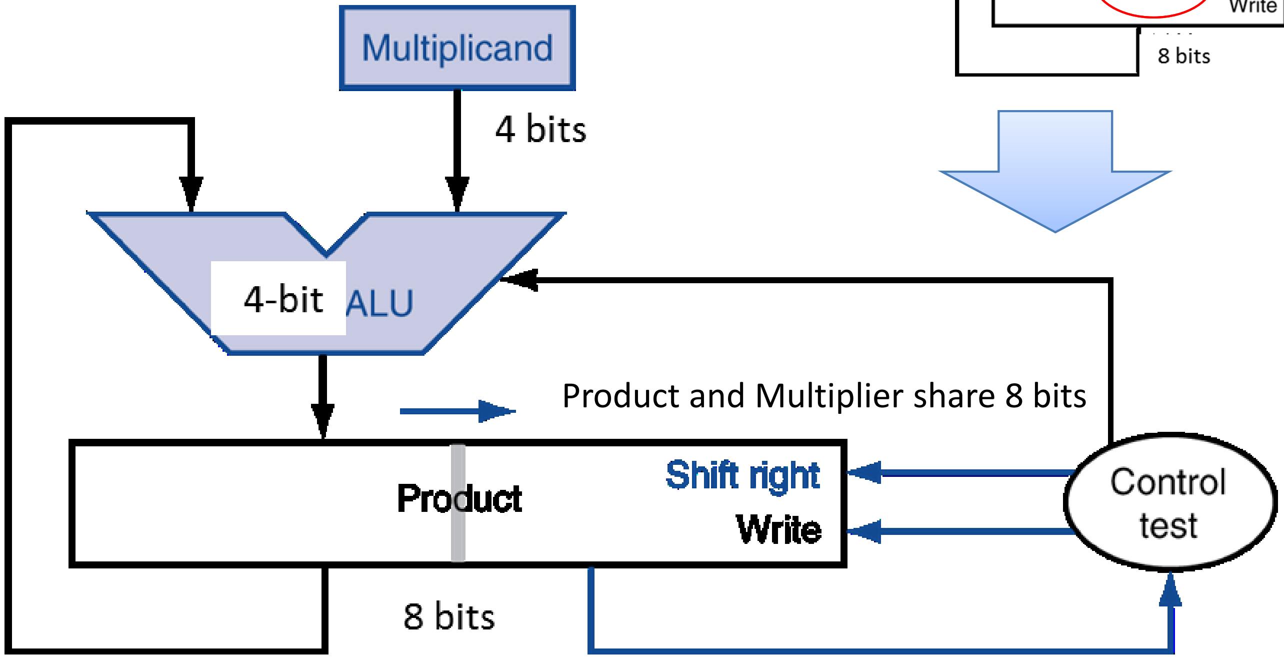
1. Check LSB of Multiplier



2. LSB=0→Do nothing
   LSB=1→Add Multiplicand to Product

Improved Design – 4-bit Multiplier

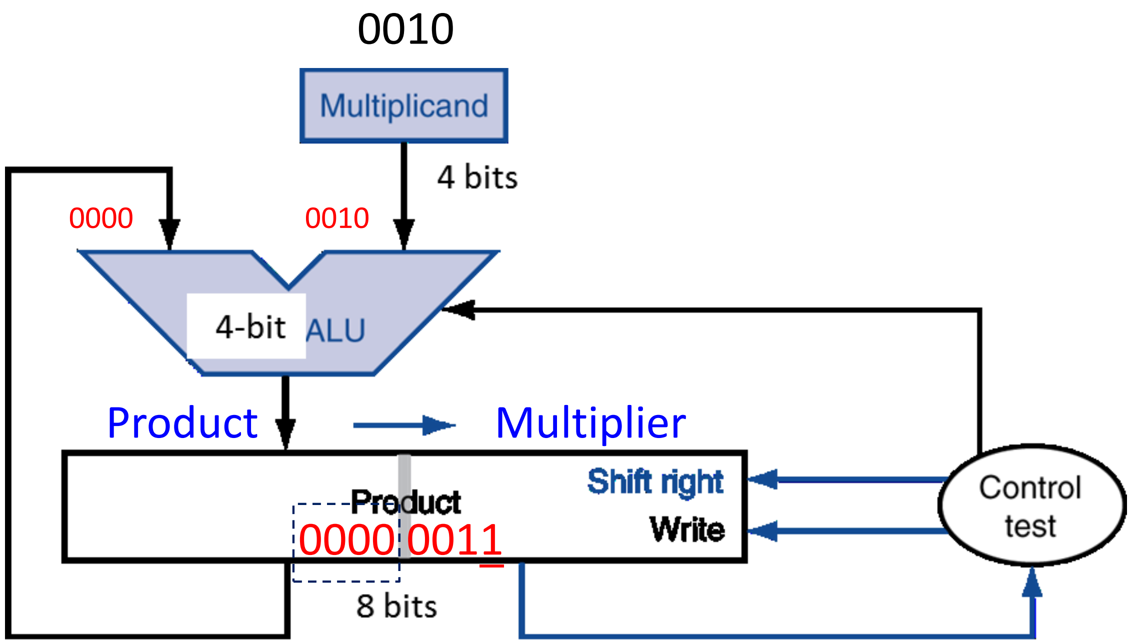




Only 4 MSBs need to be added

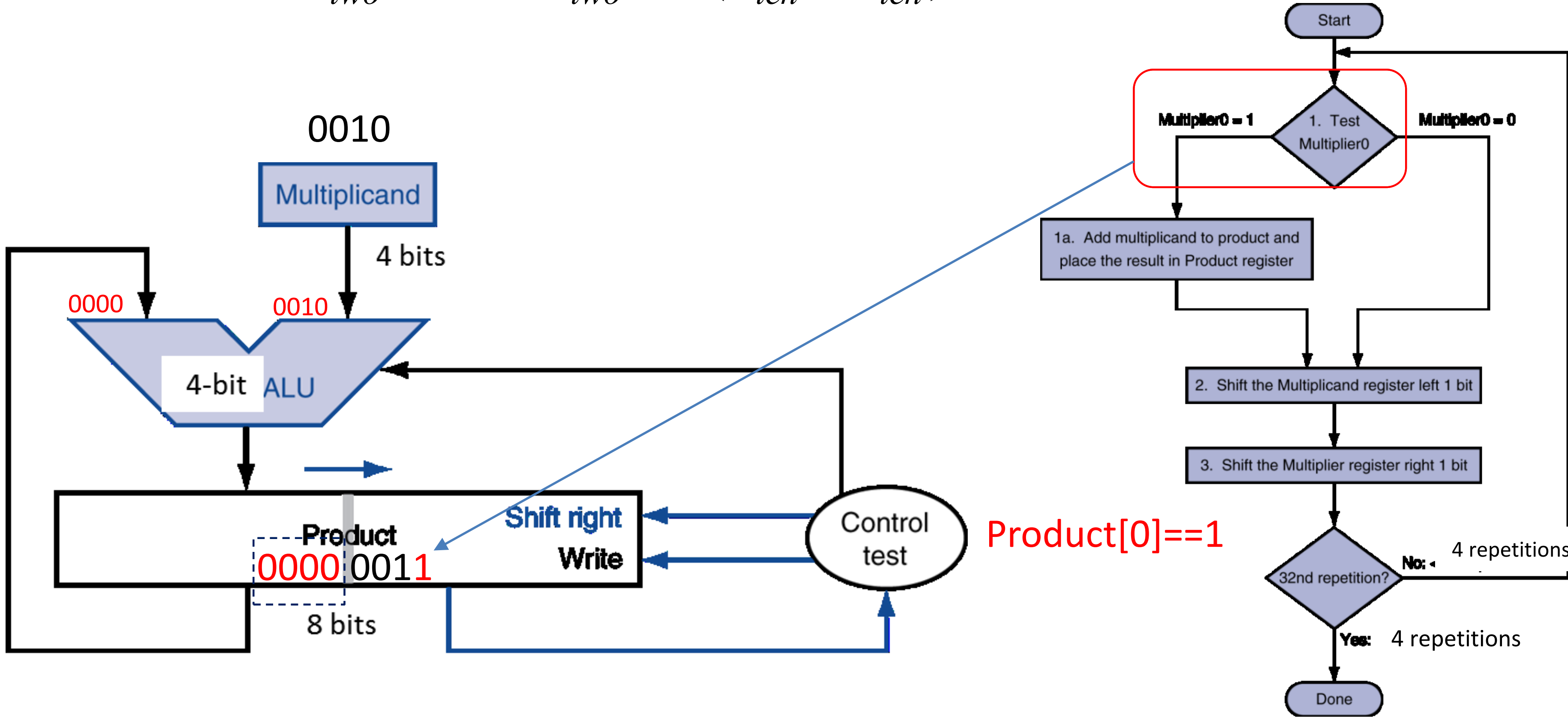
Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$



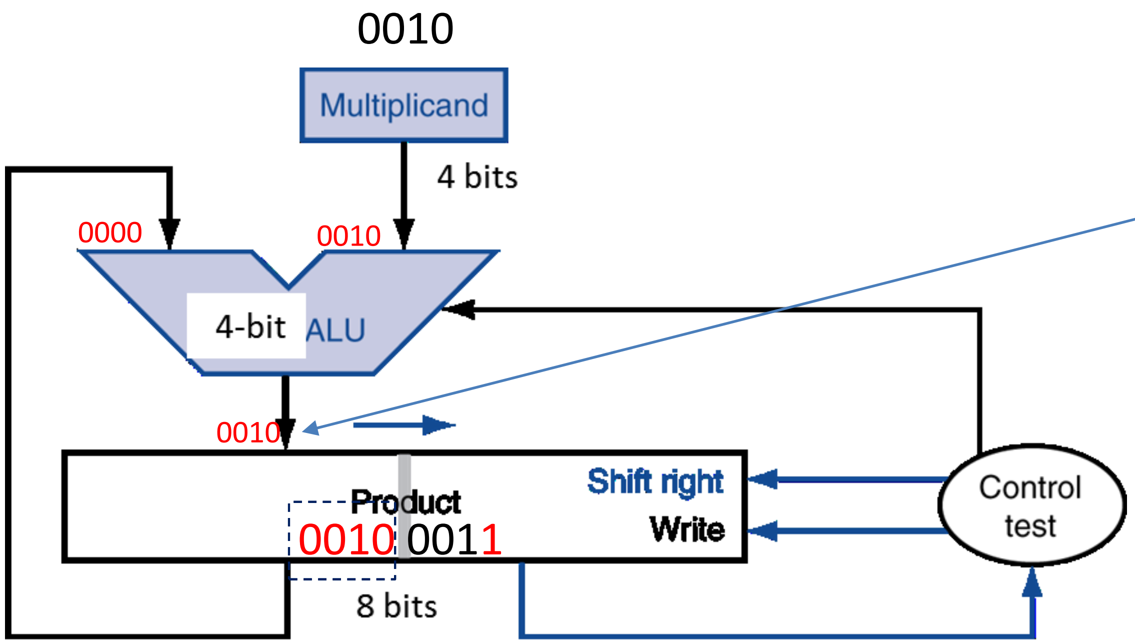
Optimized Multiplier

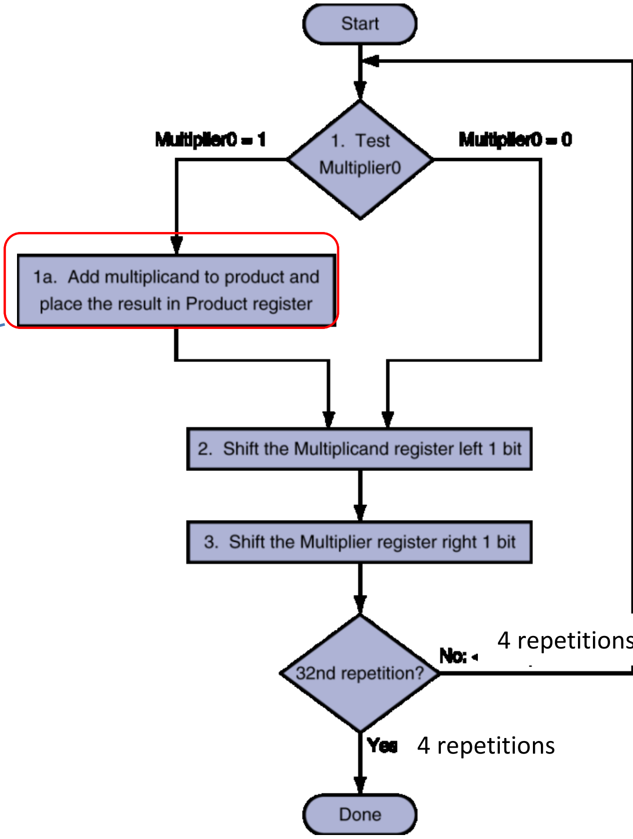
Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$



Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

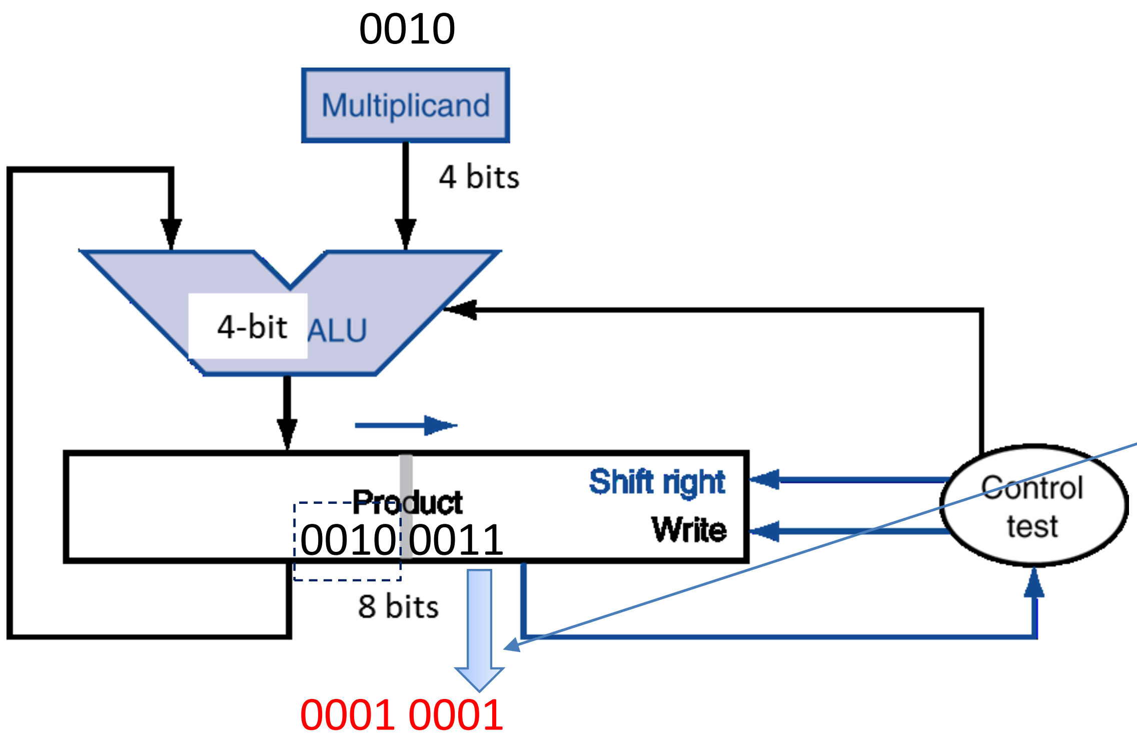


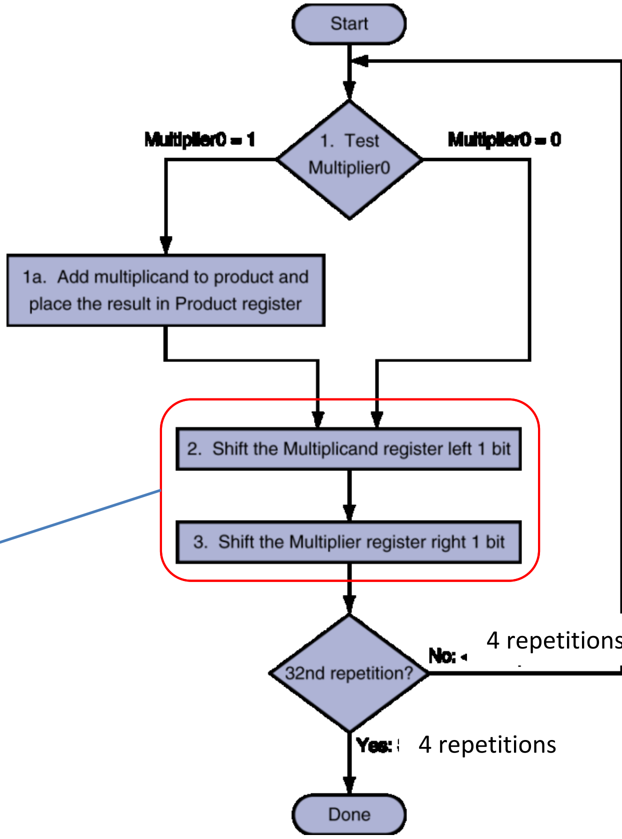


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No:
4 repetitions
Yes
4 repetitions
Done

Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

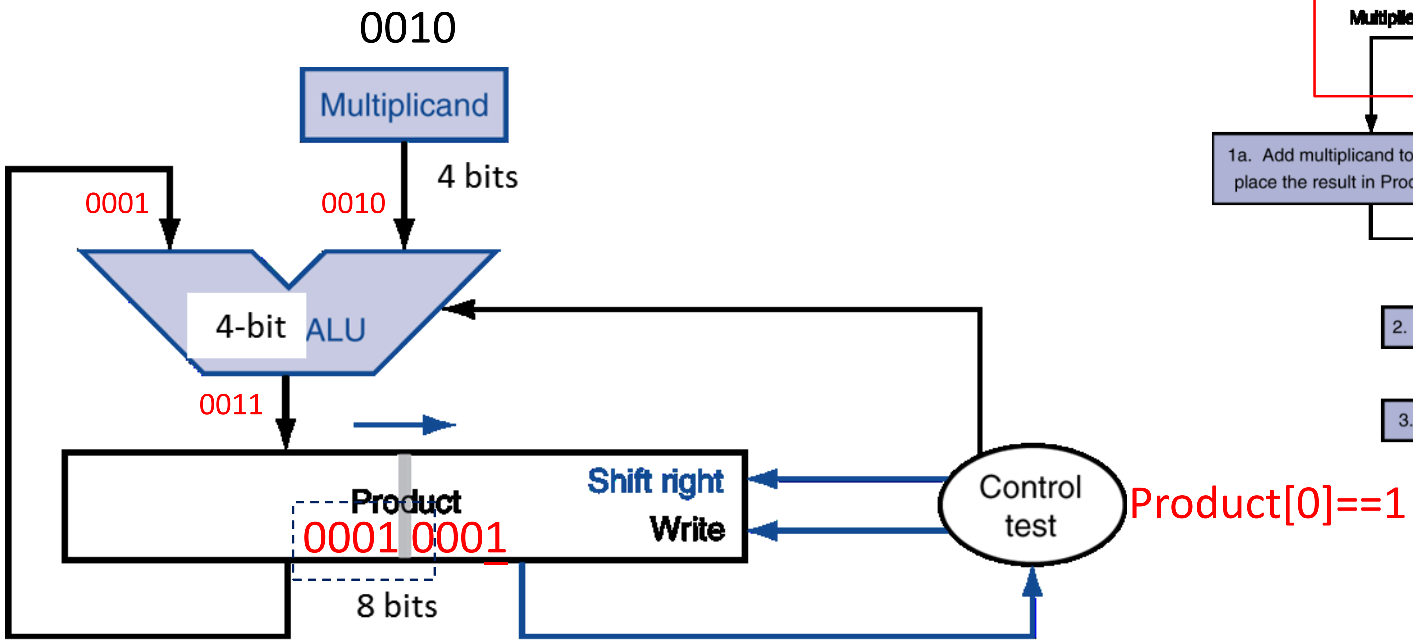


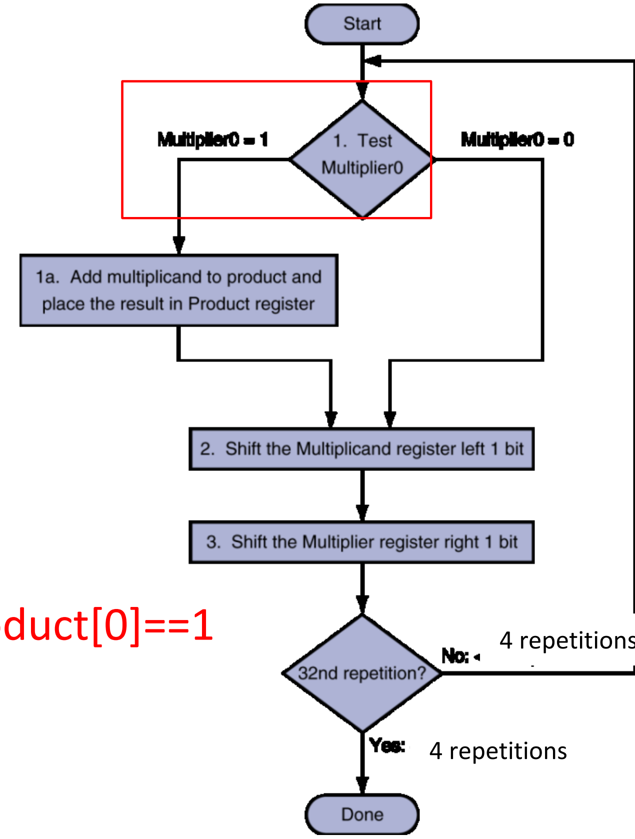


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No:
4 repetitions
Yes:
4 repetitions
Done

Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

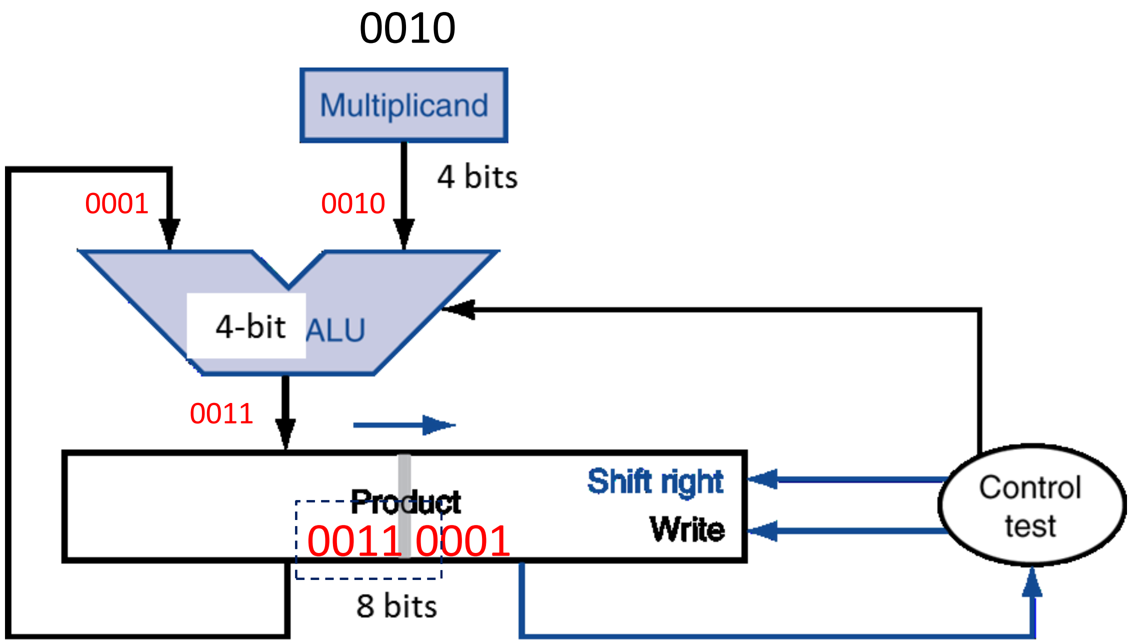


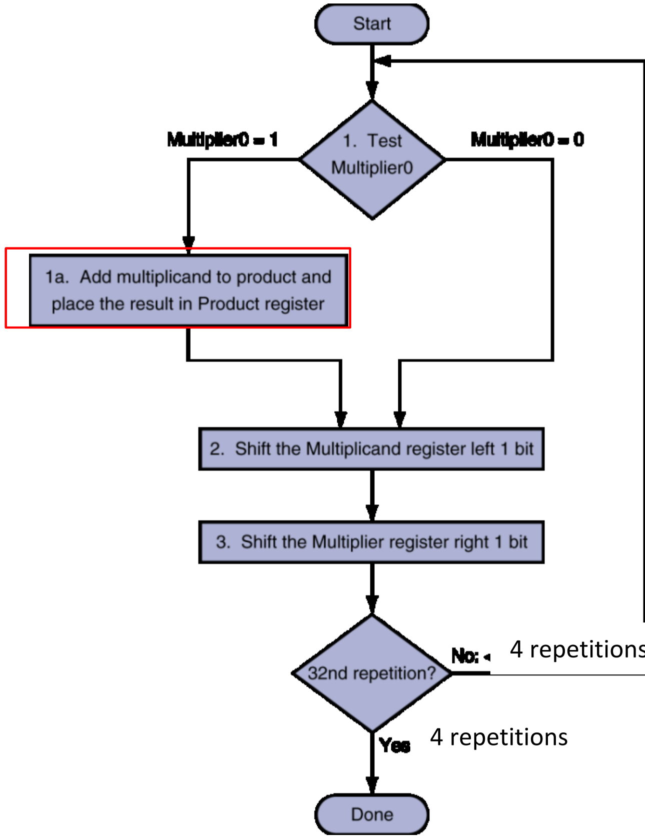


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: 4 repetitions
Yes: 4 repetitions
Done

Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

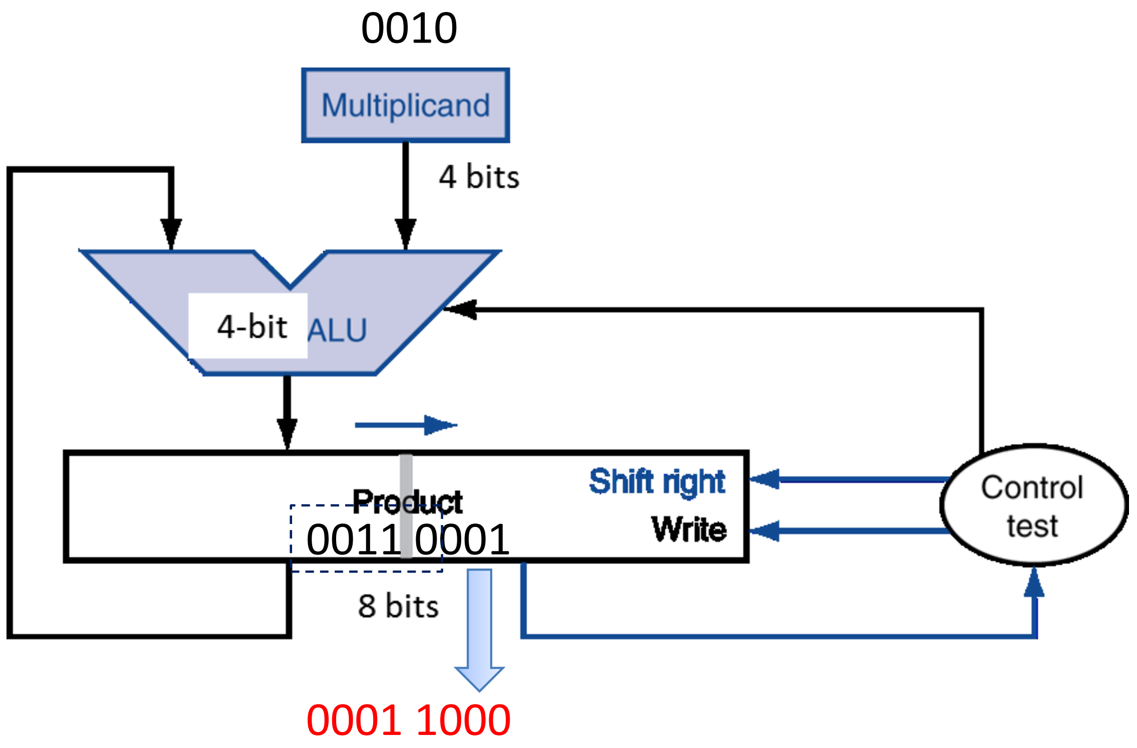


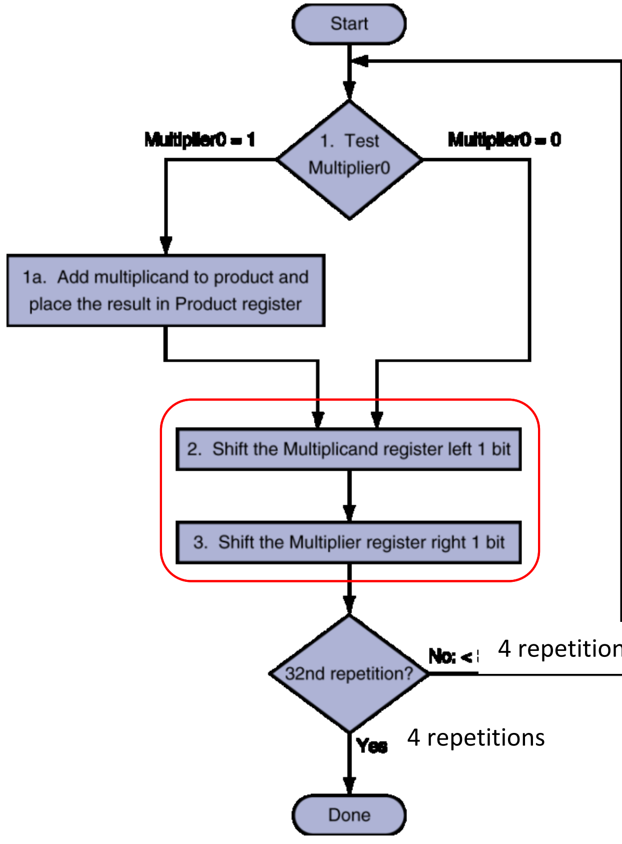


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 4 repetitions
Yes
4 repetitions
Done

Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

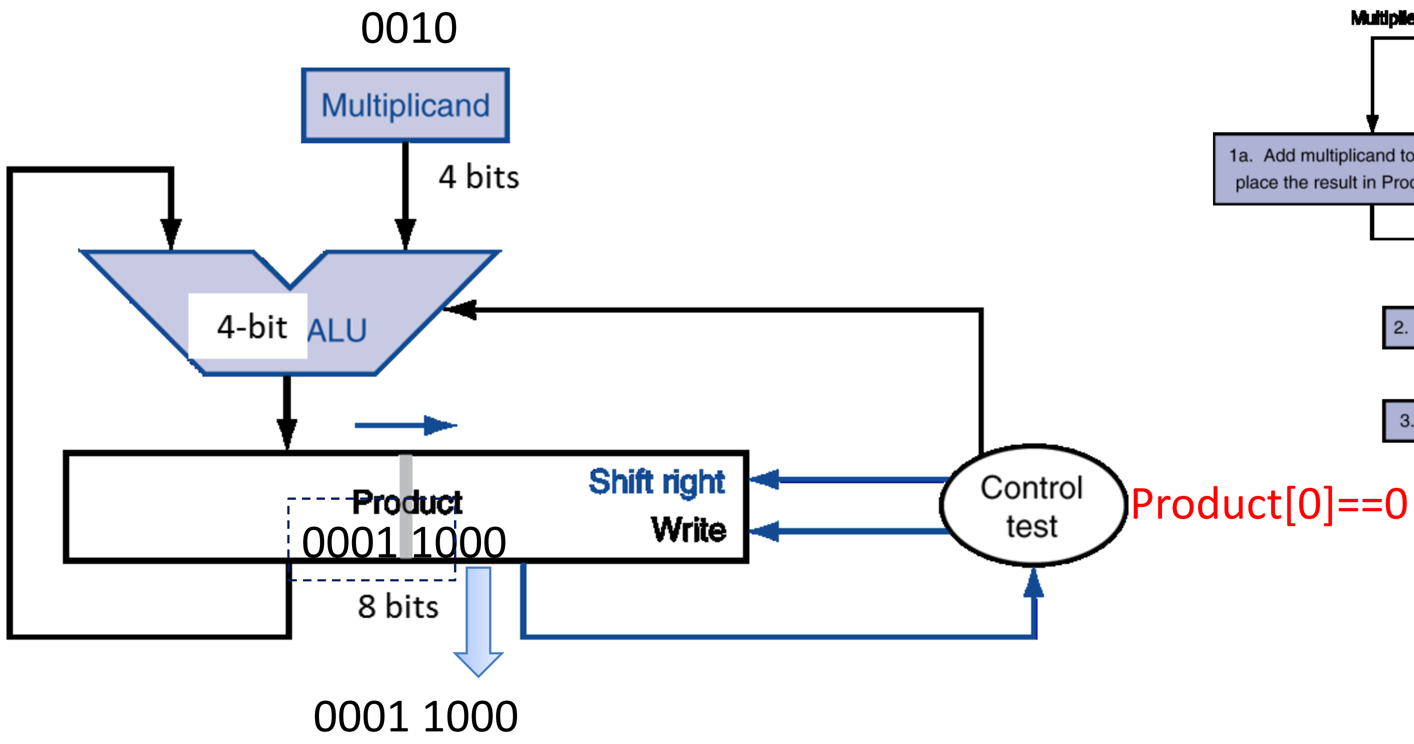


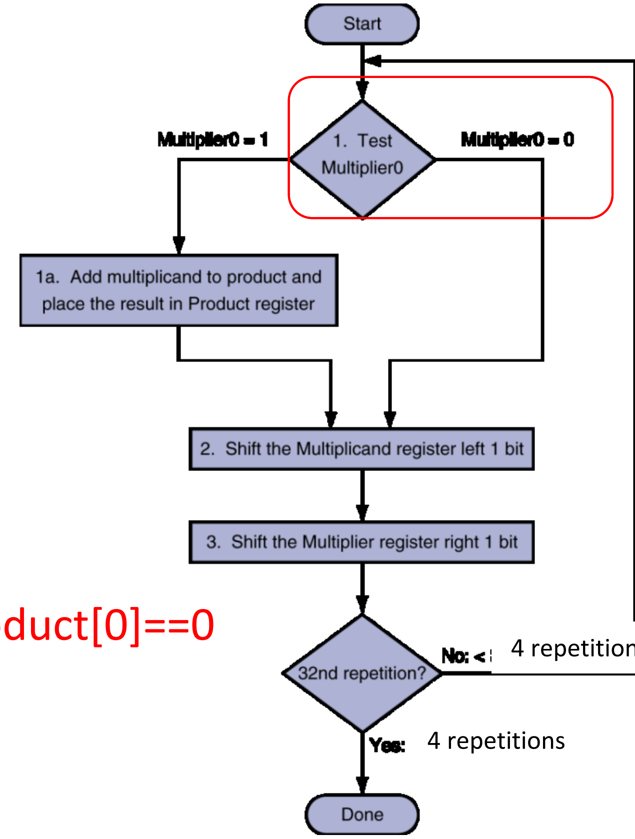


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <
4 repetition
Yes
4 repetitions
Done

Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

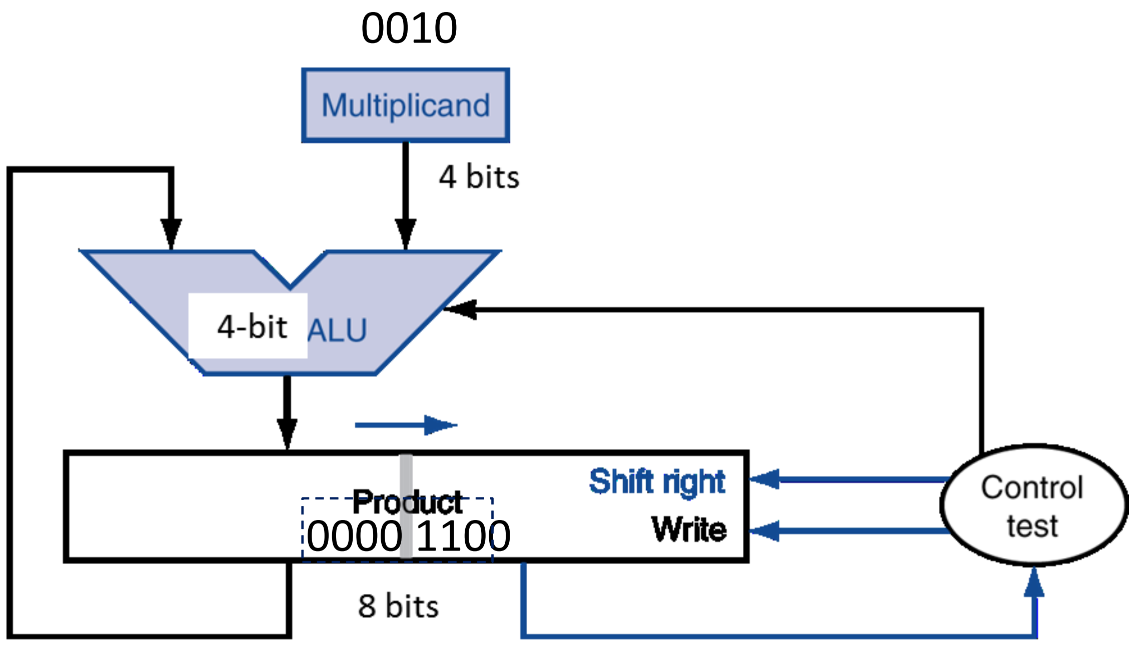


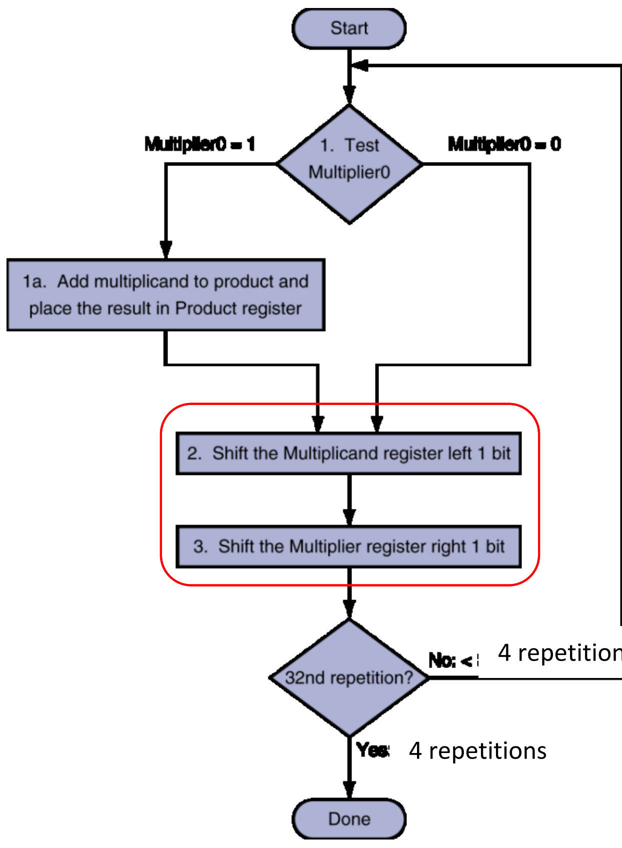


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <
4 repetition
Yes:
4 repetitions
Done

Optimized Multiplier

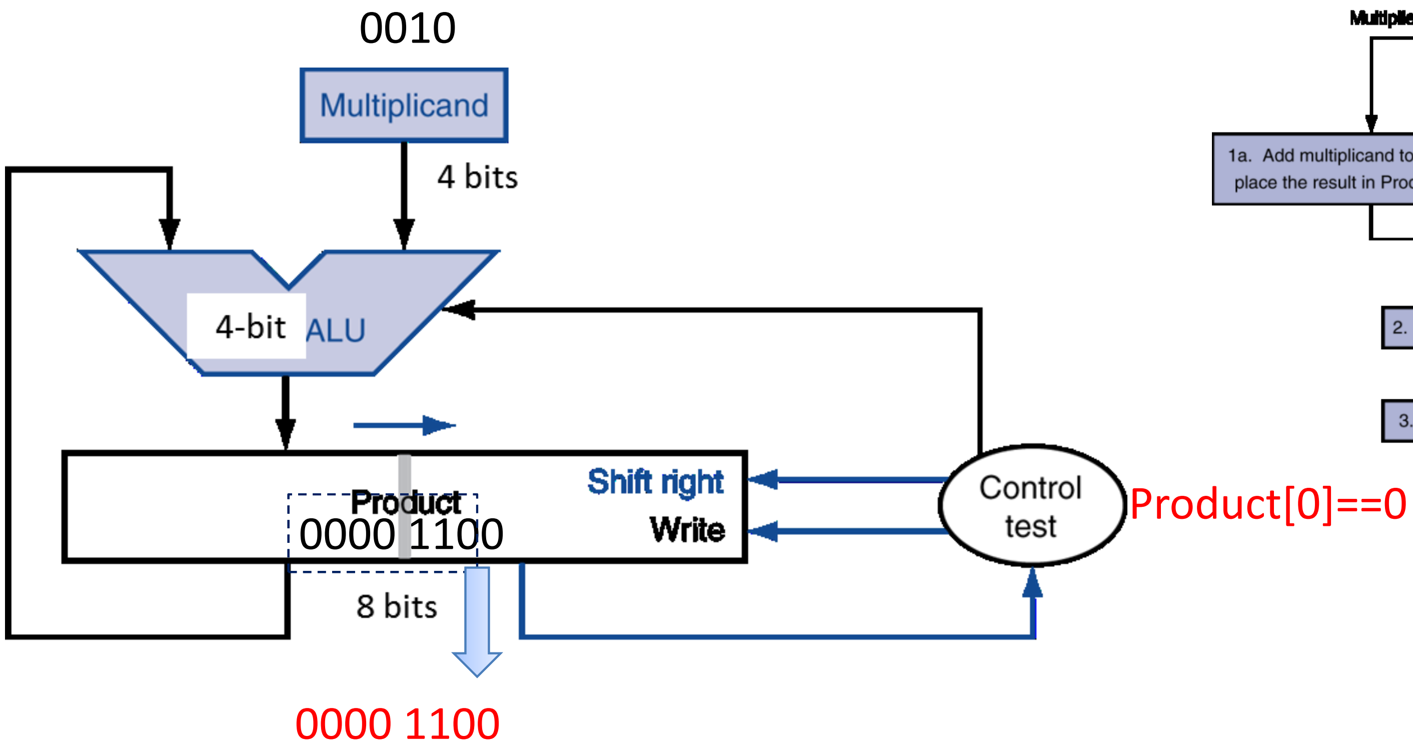
Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

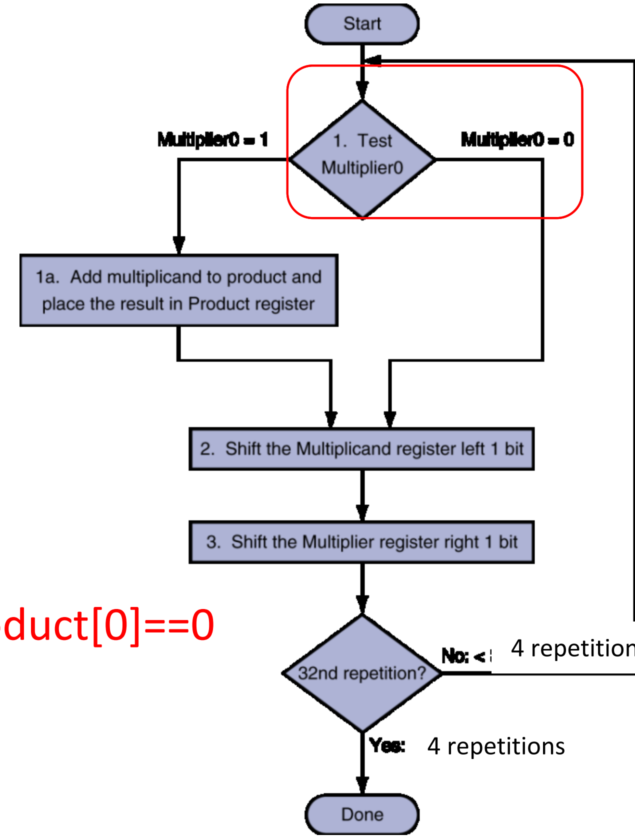



Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <
4 repetition
Yes:
4 repetitions
Done

Optimized Multiplier

Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$

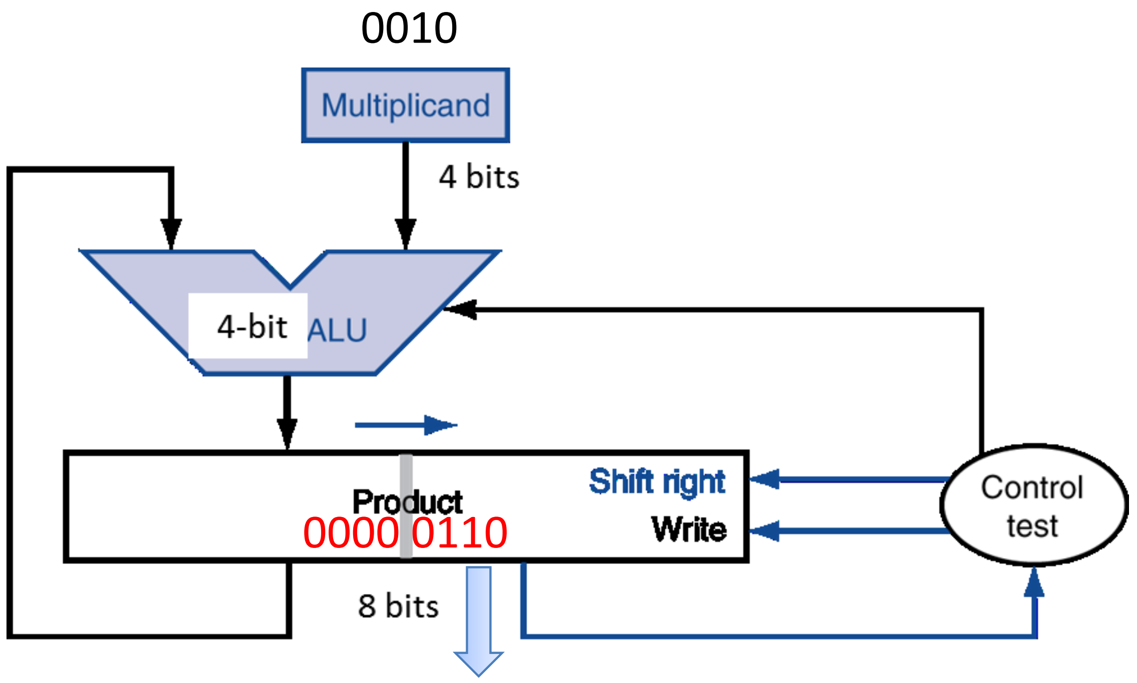


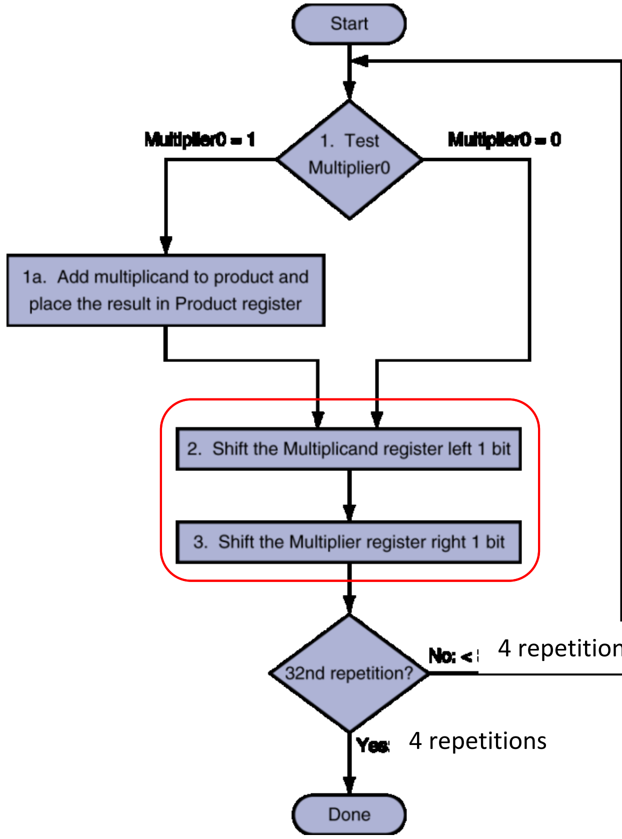


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <
4 repetition
Yes:
4 repetitions
Done

Optimized Multiplier

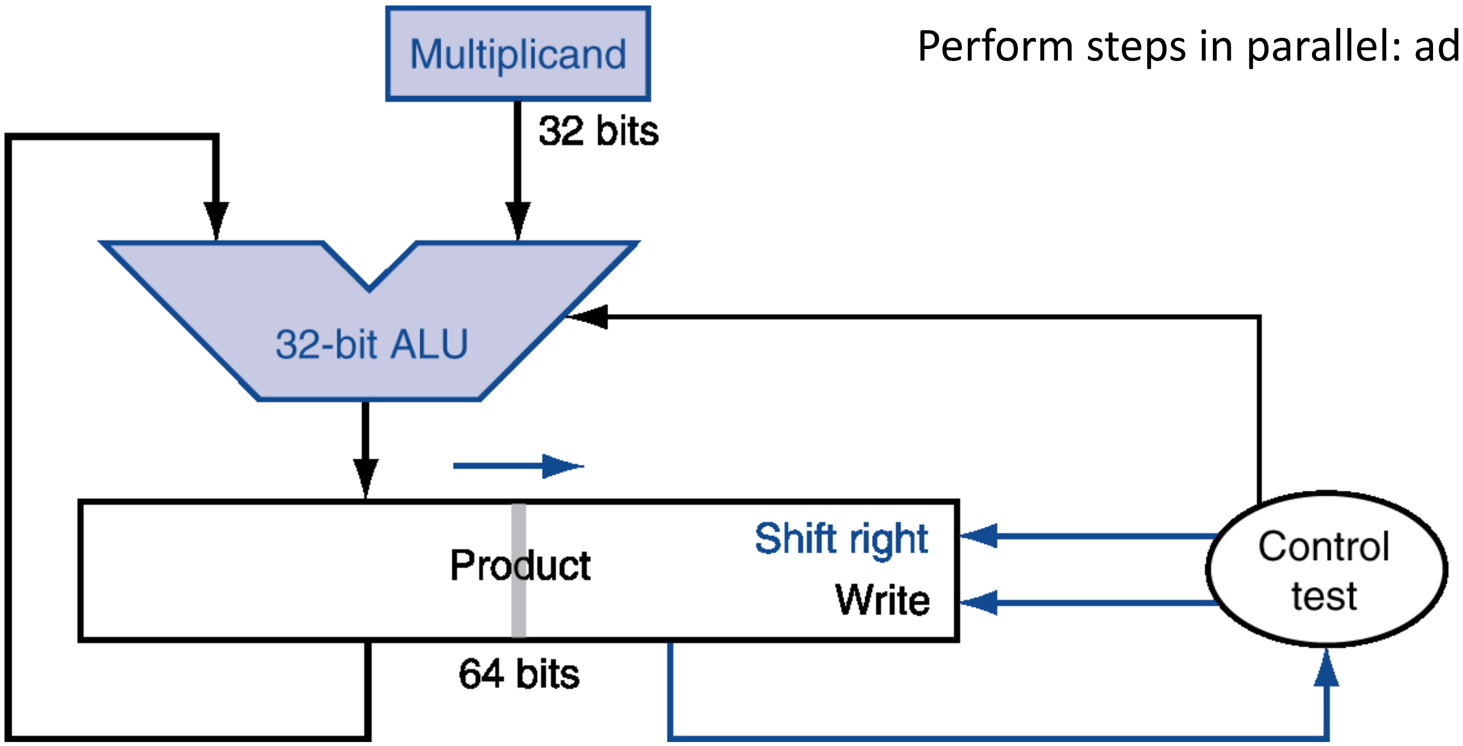
Ex: $0010_{two} \times 0011_{two} \equiv (2_{ten} \times 3_{ten})$





Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <
4 repetition
Yes:
4 repetitions
Done

Optimized 64-bit Multiplier



Perform steps in parallel: add/shift

One cycle per partial-product addition (Slow)

Signed Multiplication

- What about signed multiplication?
  - Make both positive
  - Leave out sign bit, run for 31 steps
  - Complement product afterwards when needed

Signed Multiplication

- How to do signed multiplication?
  - Make both multiplier and multiplicand positive and complement the produce when done if needed
  - Multiply 2’s complement numbers directly
    - sign-extend partial products
    - (if multiplier is negative) subtract at the end

Signed Multiplication

n-bit multiplier

Step 1: Sign-extend Multiplicand to n bits
Step 2: Proceed with multiplication process until sign bit
Step 3: Check sign bit of Multiplier
0 => 0 x multiplicand
1 => -1 x multiplicand

n-bit Value (2's complement)

$$s\ v_{n-2}\ v_{n-3} \ldots v_0 = -s \cdot 2^{n-1} + \sum_{i=0}^{n-2} v_i \cdot 2^i$$

$$\begin{aligned} -7 &= 1001 \\ &= -1 \times 2^3 + 1 \times 2^0 \end{aligned}$$

$$\begin{array}{r} 1101 \\ \times \quad 0010 \\ \hline 00000000 \\ 1111101 \\ \vdots \\ \hline 11111010 \end{array}
\qquad
\begin{array}{rl} & 0010 \\ \times & 1101 \\ \hline + & 00000010 \\ + & 0000000 \\ + & 000010 \\ - & 00010 \xrightarrow{\quad} 11110 \\ \hline & 11111010 \end{array}$$

Signed Multiplication – Booth's Algorithm

- Intuition
  - $3 \times 99 = 3 \times (100 - 1)$
  - Assuming Multiplicand M is multiplied by Multiplier Q "00111110"

$$
\begin{array}{r}
1000 \\
-\ 0001 \\
\hline
0111
\end{array}
$$

$$M \times 00111110 = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1)$$

$$\Downarrow$$

$$M \times 010000(-1)0 = M \times (2^6 - 2^1) = M \times 2^6 - M \times 2^1$$

We need
- positive M
- negative M
- Multiplier Q transformation

$00111110 \text{->} 010000(-1)0$

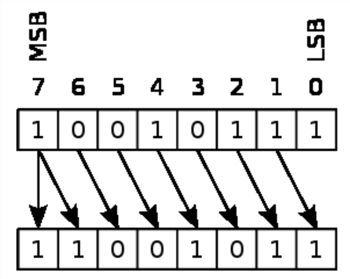
Ref: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

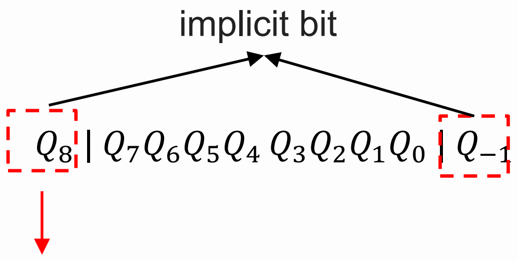
Booth’s Algorithm

Multiplicand M, Multiplier Q, Product P

Q_i	Q_{i-1}	Action
0	0	ARS
1	1	ARS
0	1	$P \leftarrow P + M$, then ARS
1	0	$P \leftarrow P - M$, then ARS

Arithmetic right shift (ARS)





Need it when dealing with the exception:

The multiplicand is the most negative number

https://en.wikipedia.org/wiki/Arithmetic_shift
Ref: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

Booth’s Algorithm

- Example : $3 \times (-4) \quad \rightarrow \quad 0011 \times 1100$

A 4-bit number is multiplied by a 4-bit number →

- m = 0011, -m = 1101, q=1100

All of the numbers must have a length equal to 4+4+1=9

- Initialize M = 0011 0000 0, -M = 1101 0000 0
  The product P = 0000 1100 0 (with q in the last 4 bits except for the implicit bit)

- Then do the following actions 4 times (4-bit multiplication)

P = 0000 $\underbrace{1100}_{q}$ 0

1. P = 0000 1100 0. The last two bits are 00.
   P = 0000 0110 0. Arithmetic right shift.
2. P = 0000 0110 0. The last two bits are 00.
   P = 0000 0011 0. Arithmetic right shift.
3. P = 0000 0011 0. The last two bits are 10.
   P = 1101 0011 0. P ← P – M.
   P = 1110 1001 1. Arithmetic right shift.
4. P = 1110 1001 1. The last two bits are 11.
   P = 1111 0100 1. Arithmetic right shift.

The product is 1111 0100

Ref: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

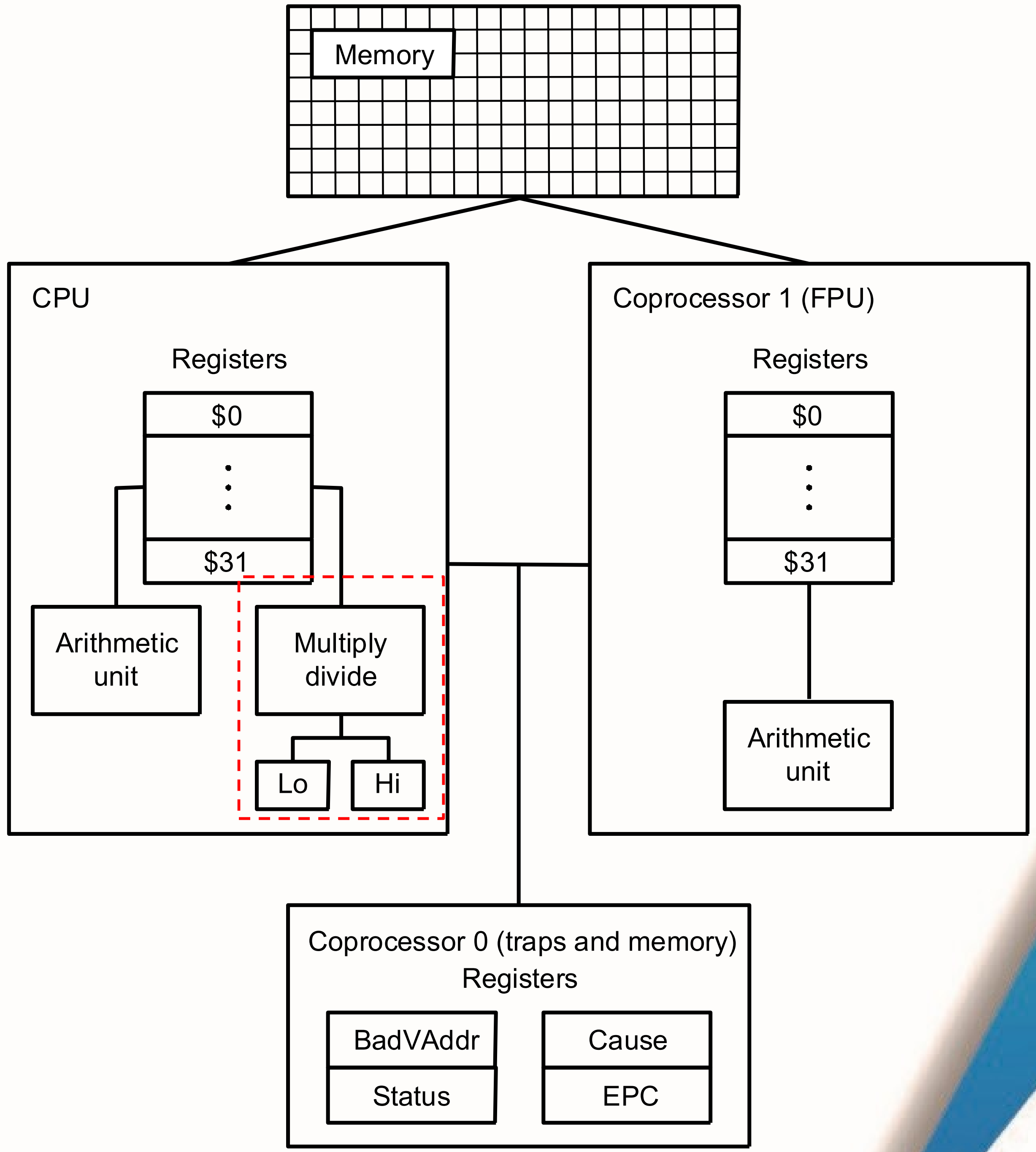
Booth’s Algorithm - Exception

- When the multiplicand is the most negative number (e.g. 4-bit => -8)
- Add one more implicit bit to the MSB for – M (signed extension)
- Example - $-8 \times 2 \quad \rightarrow \quad 1000 \times 0010$
- Initialize M = 1 1000 0000 0, -M = 0 1000 0000 0
  The product P = 0 0000 0010 0

1. P = 0 0000 0010 0. The last two bits are 00.
   P = 0 0000 0001 0. Arithmetic right shift.
2. P = 0 0000 0001 0. The last two bits are 10.
   P = 0 1000 0001 0. P ← P − M.
   P = 0 0100 0000 1. Arithmetic right shift.
3. P = 0 0100 0000 1. The last two bits are 01.
   P = 1 1100 0000 1. P ← P + M.
   P = 1 1110 0000 0. Arithmetic right shift.
4. P = 1 1110 0000 0. The last two bits are 00.
   P = 1 1111 0000 0. Arithmetic right shift.

The product is 1111 0000

Ref: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm


Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Cause
Status
EPC

MIPS Division

- Instructions
  - `div rs, rt  /  divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must check overflow if needed
  - Use `mflo`, `mfhi` to access quotient and remainder

```
mflo $t1    #copy quotient to t1
mfhi $t2    #copy remainder to t2
```

- HI/LO registers
  - HI: 32-bit remainder
  - LO: 32-bit quotient

Division

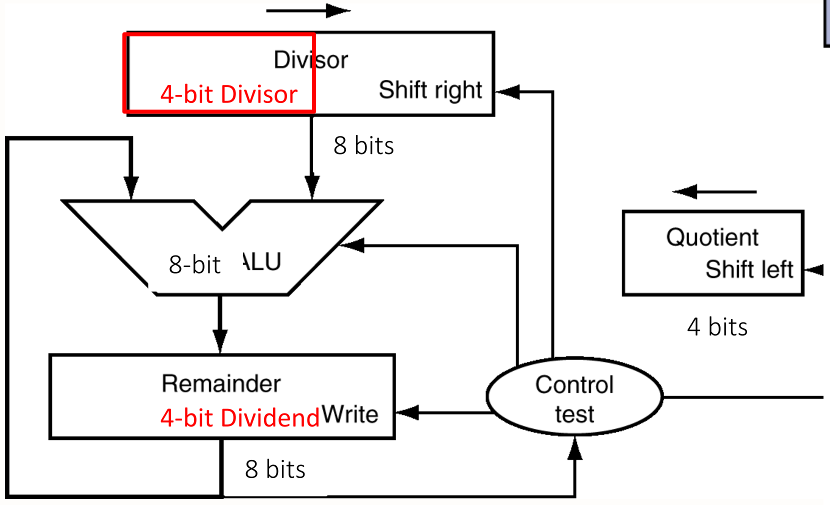
```
                                1001ten      Quotient
Divisor  1000ten             1001010ten      Dividend
                            -1000
                              0010
                               0101
                                1010
                               -1000
                                  10ten      Remainder
```

觀察:每次補位後，和除數比較
此時的除數每補1次位就降1個數量級

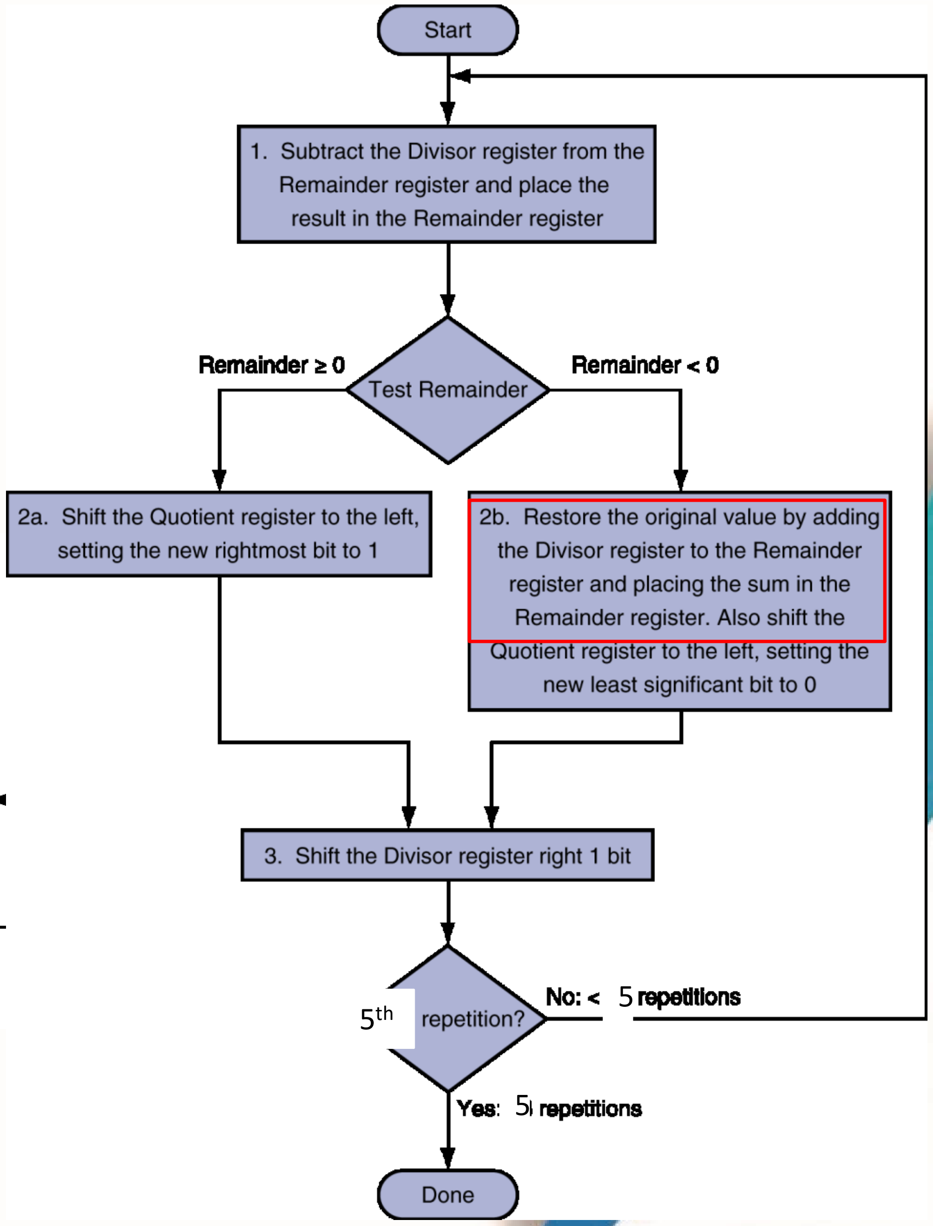
- ♦ Check devisor to see if it is 0 (error)
- ♦ If divisor ≤ dividend
  - Add 1 bit in quotient, subtract divisor from dividend
- ♦ Else
  - Add 0 bit in quotient, bring down the next dividend bit

4-bit Division Hardware

```
       0011(3)
10(2)) 0111(7)
       010
       ----
         11
         10
        ----
          1
```

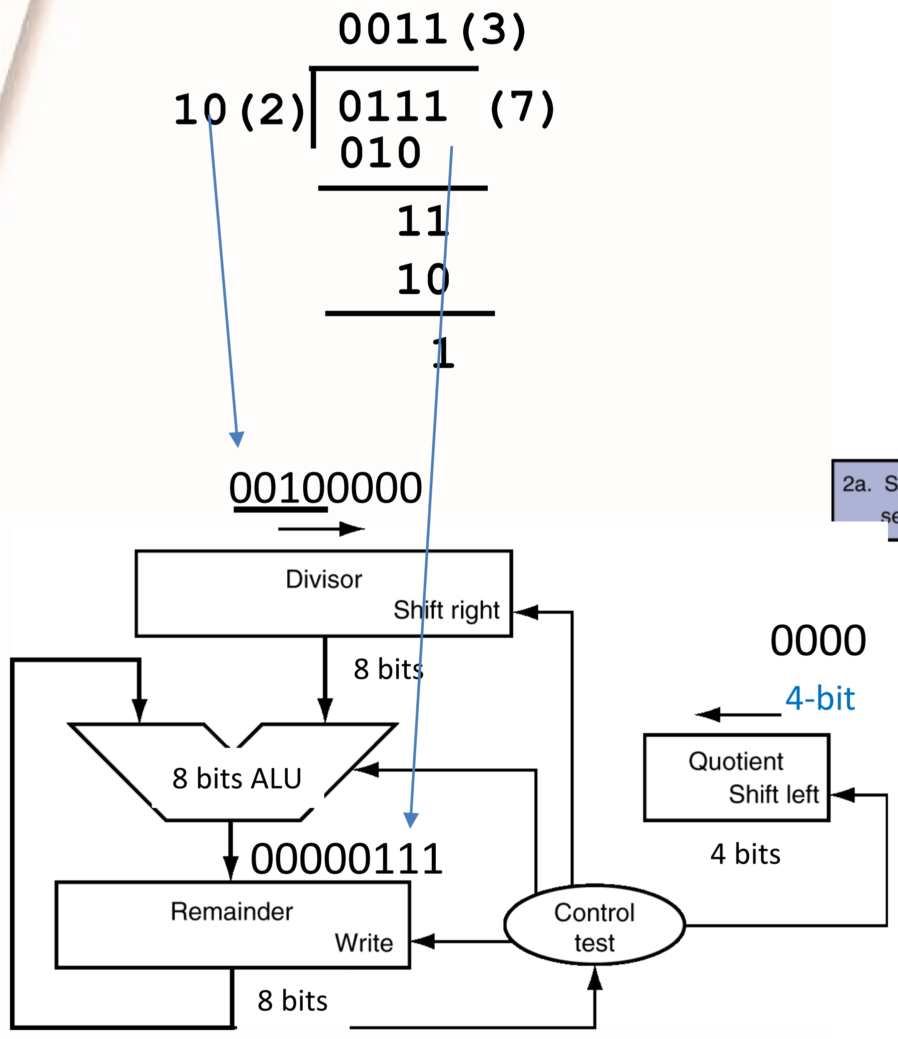


4-bit division flowchart

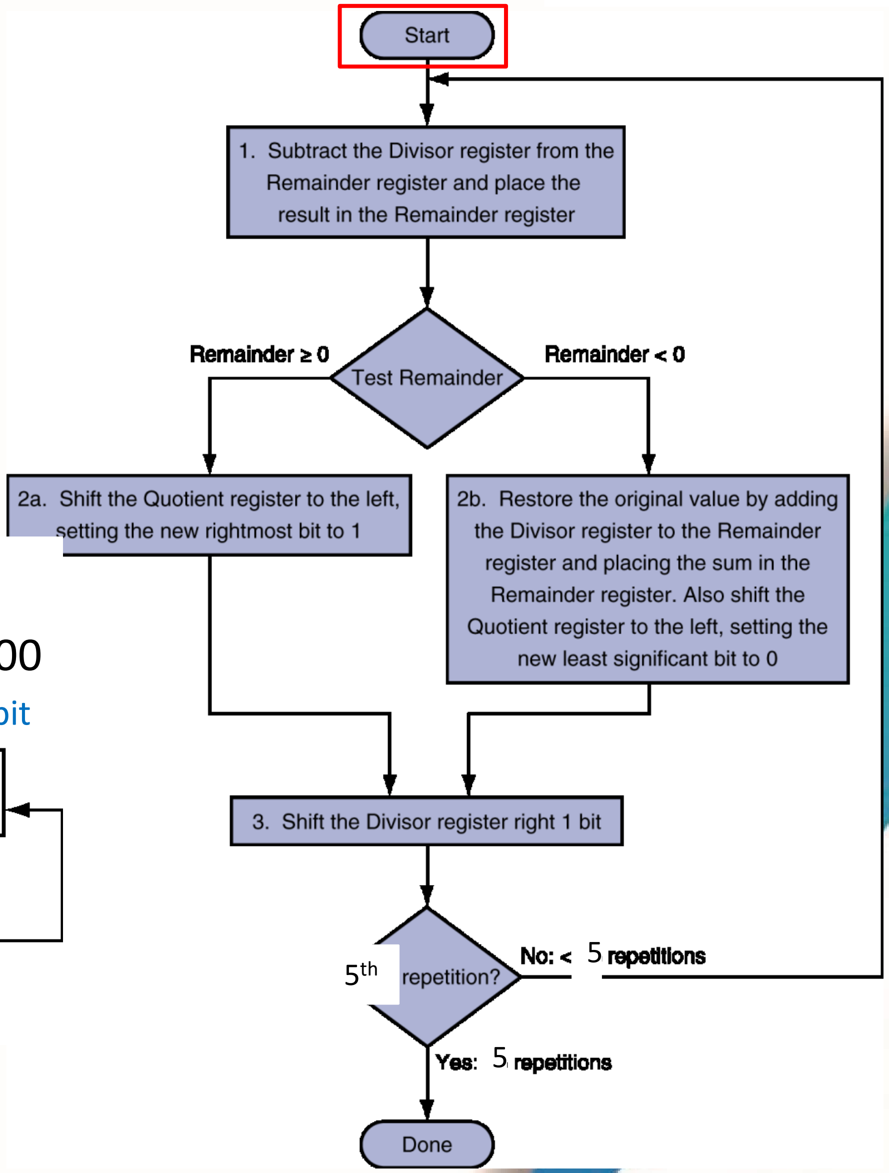


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

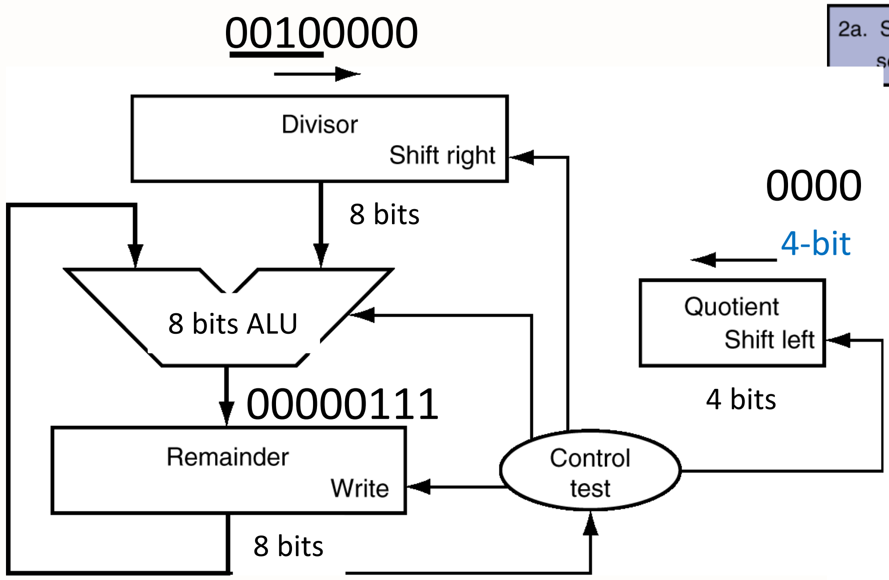
4-bit Division Hardware



4-bit division flowchart

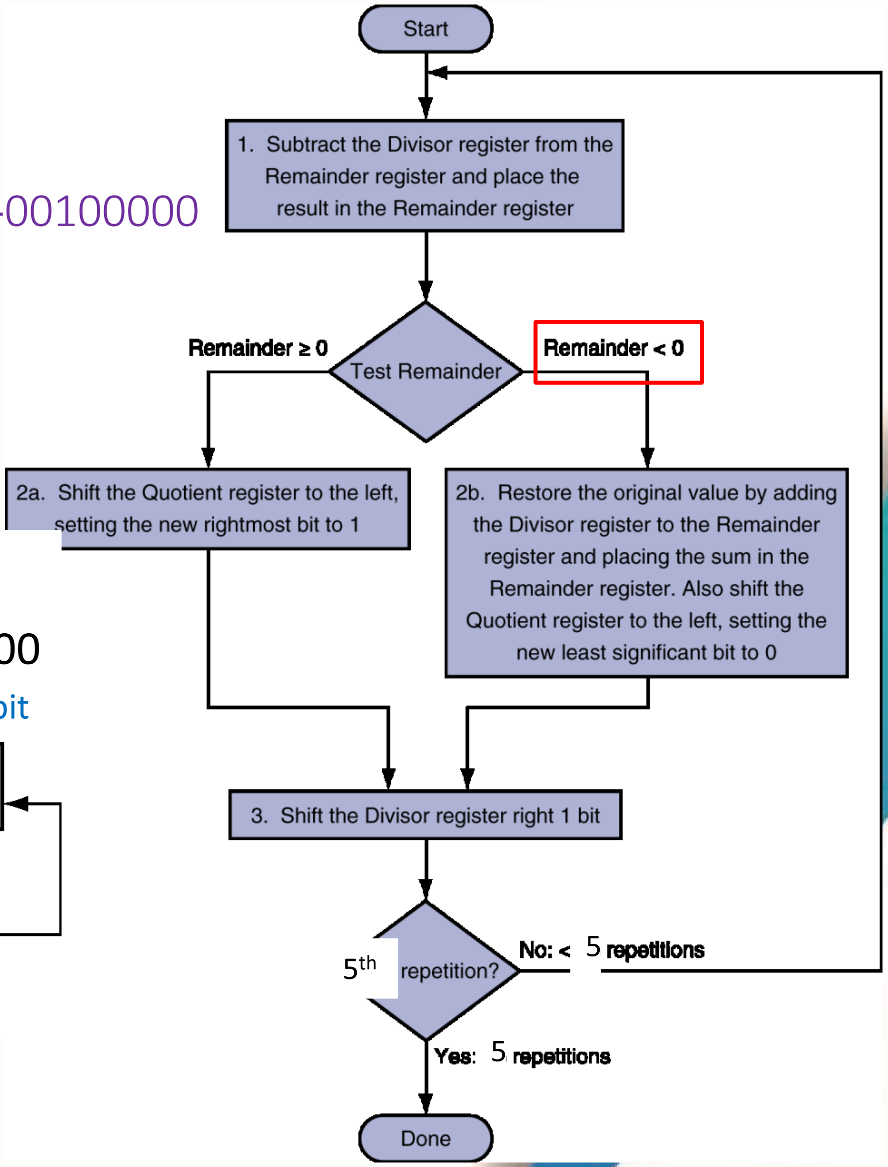

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware



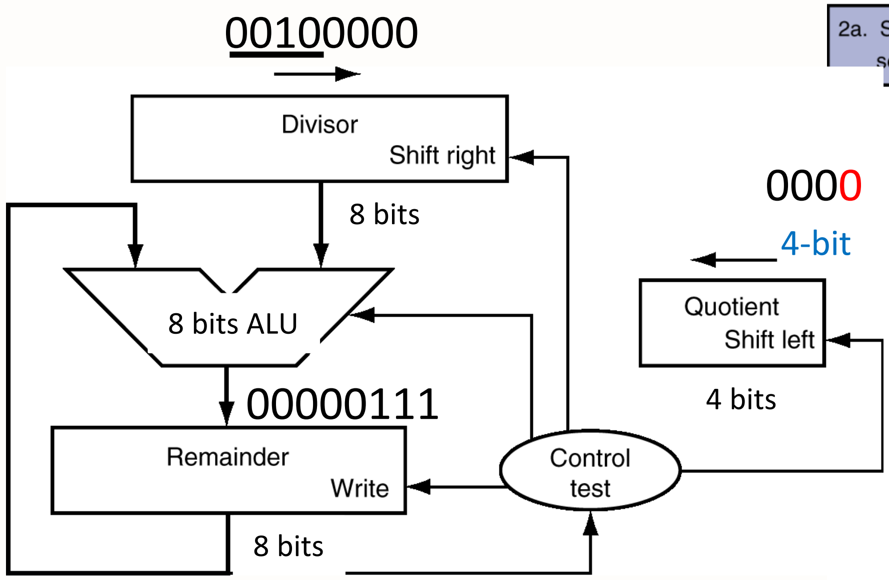
00000111 -00100000

4-bit division flowchart



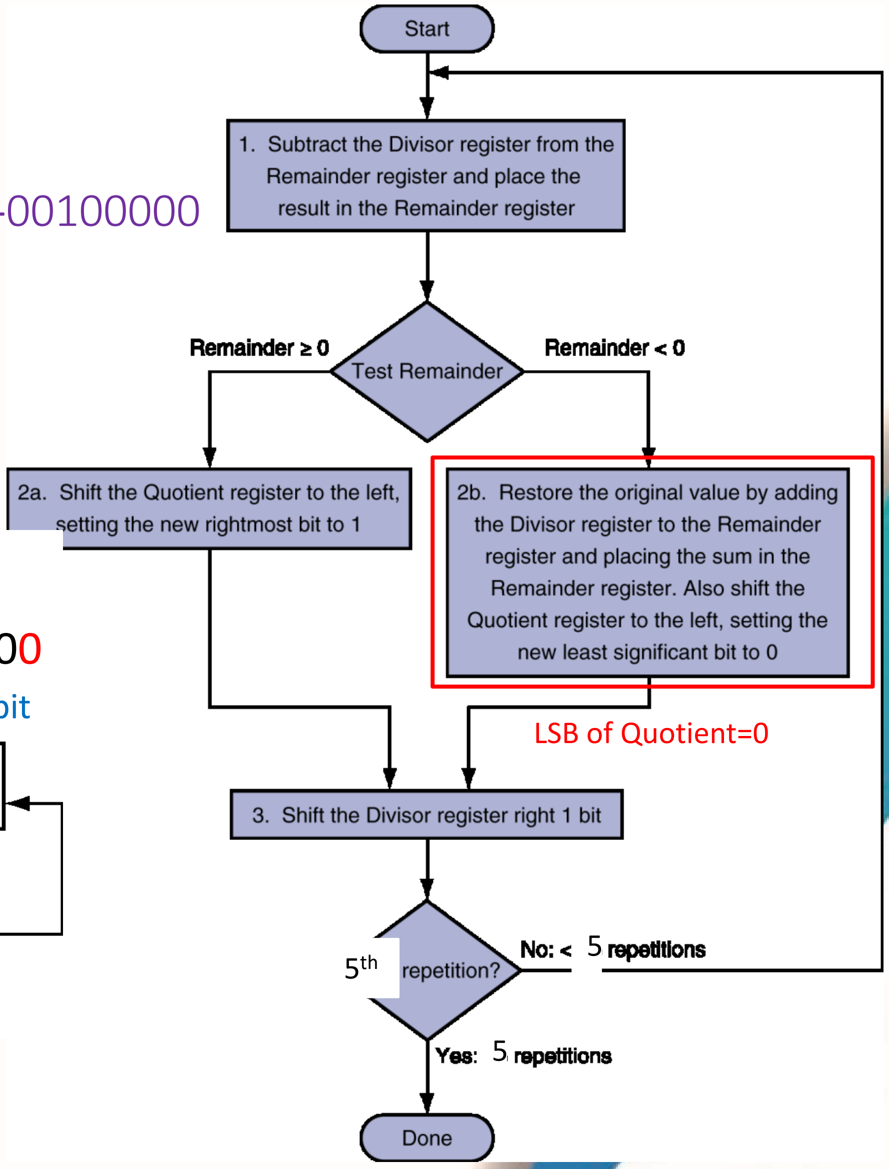
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware



4-bit division flowchart

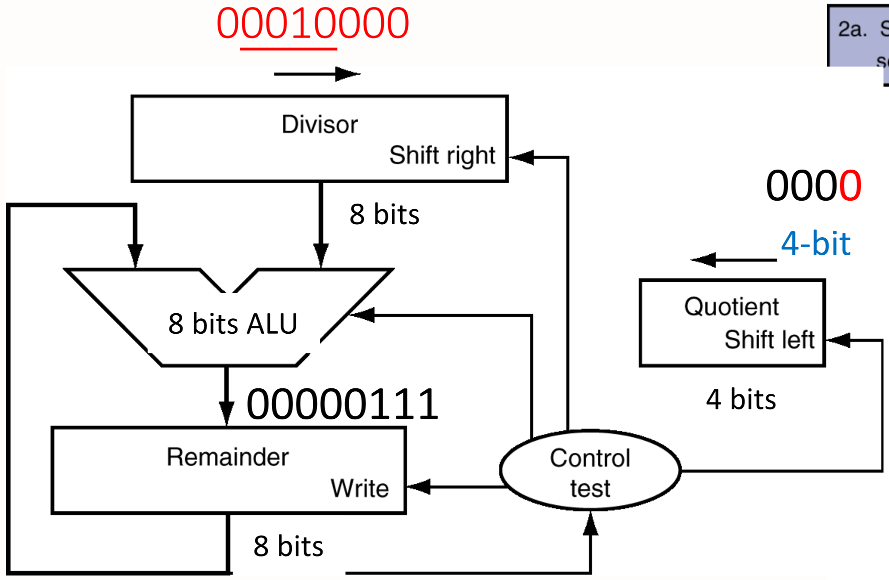
00000111 -00100000



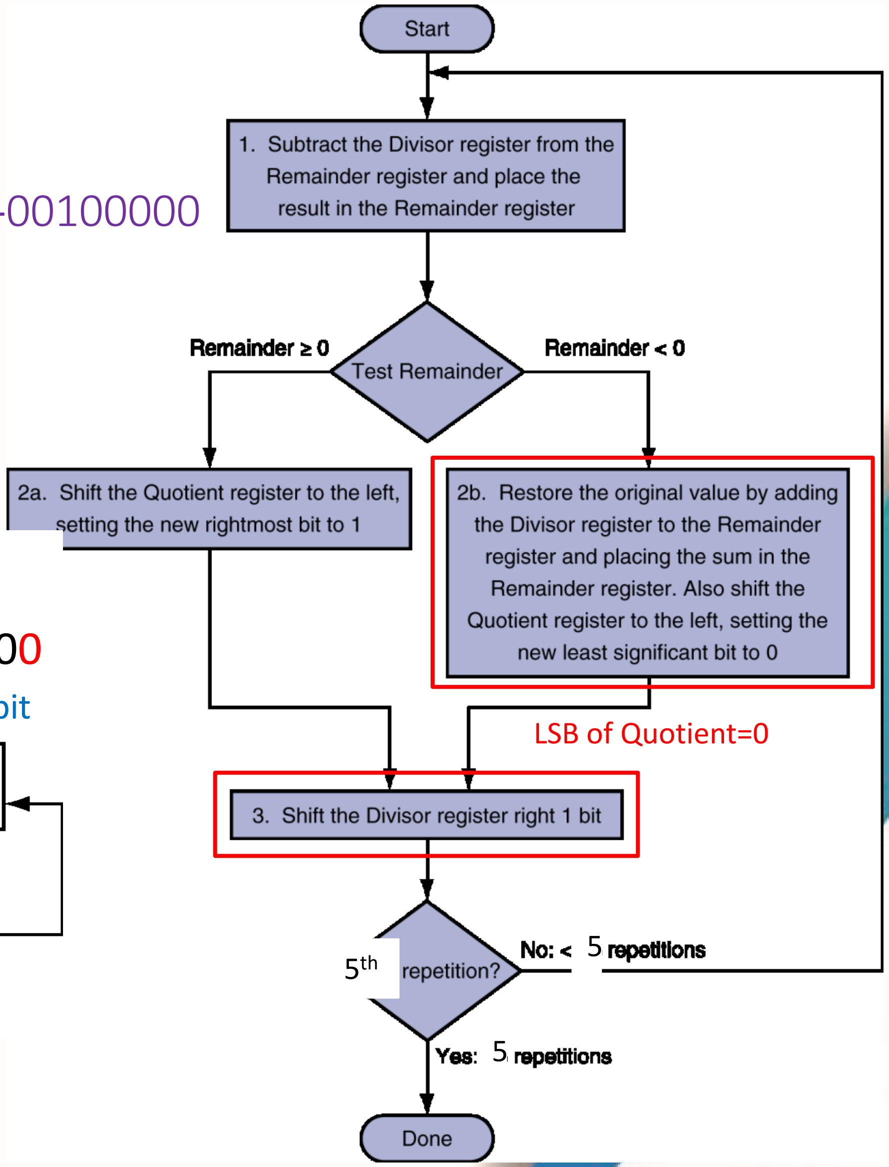
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
LSB of Quotient=0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

4-bit division flowchart

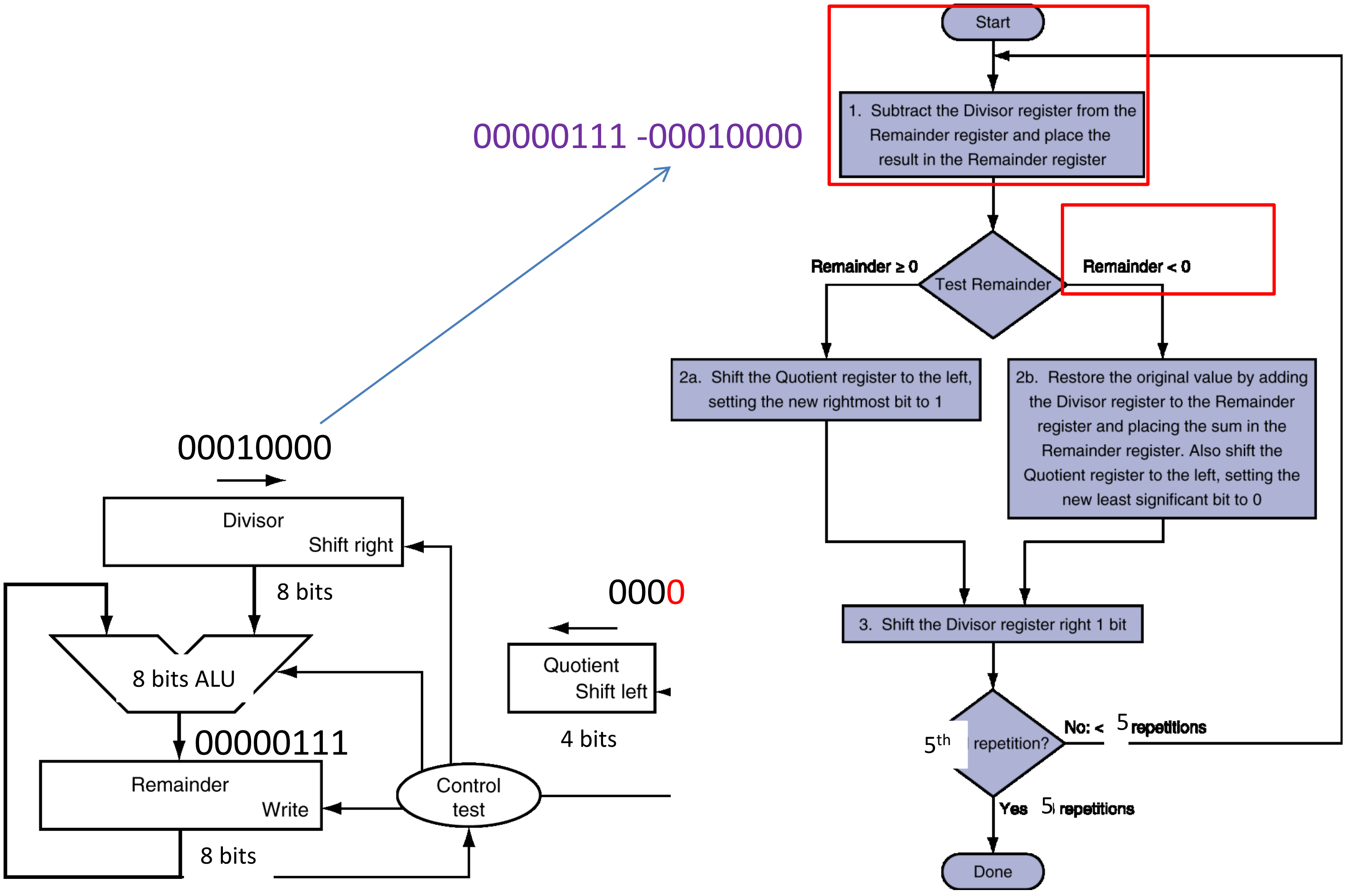


00000111 -00100000


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
LSB of Quotient=0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

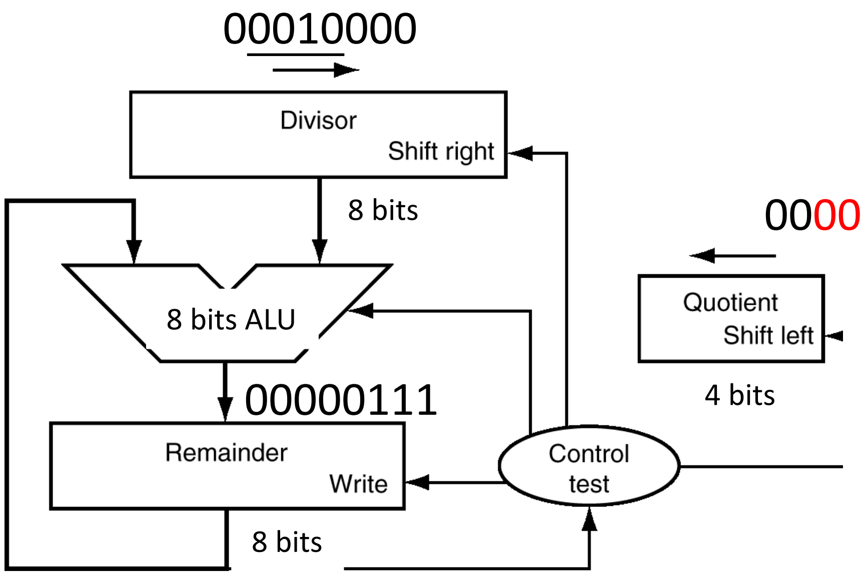
2nd iteration

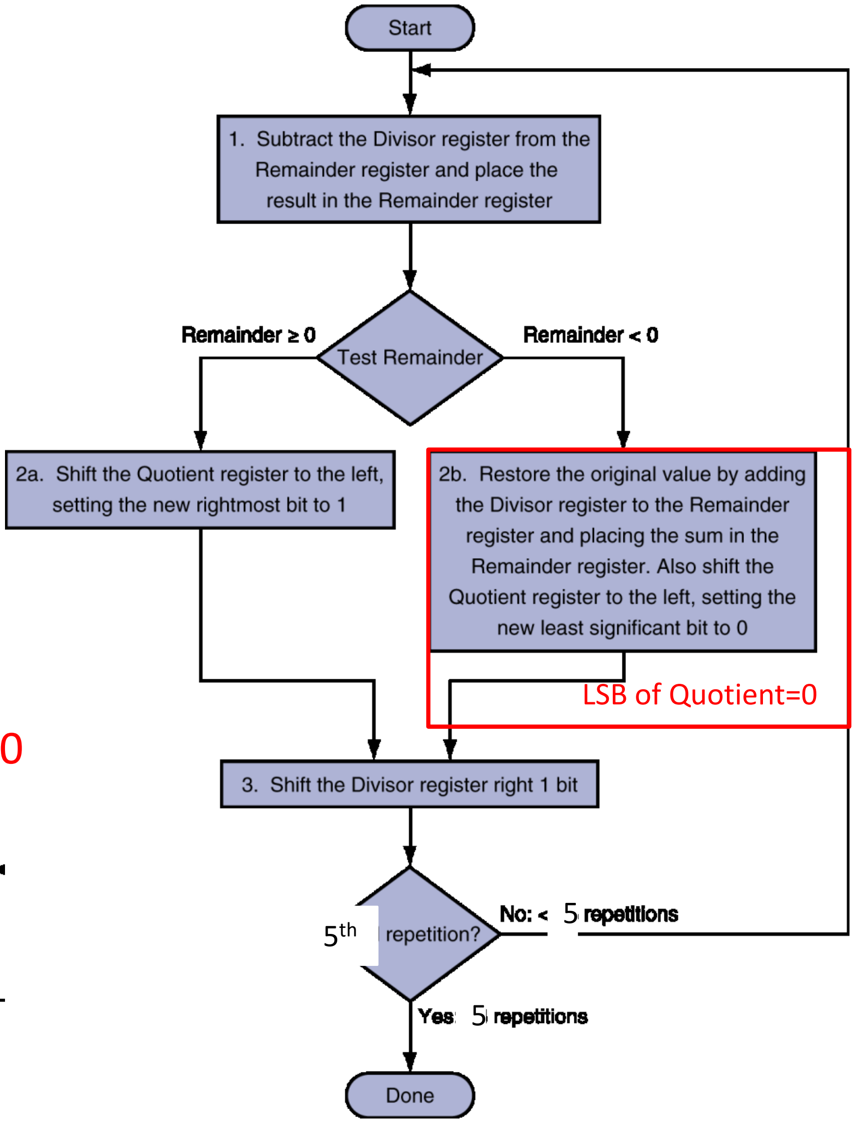


00000111 -00010000
00010000
Divisor
Shift right
8 bits
8 bits ALU
00000111
Remainder
Write
8 bits
Control
test
0000
Quotient
Shift left
4 bits
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes 5 repetitions
Done

4-bit Division Hardware

2nd iteration

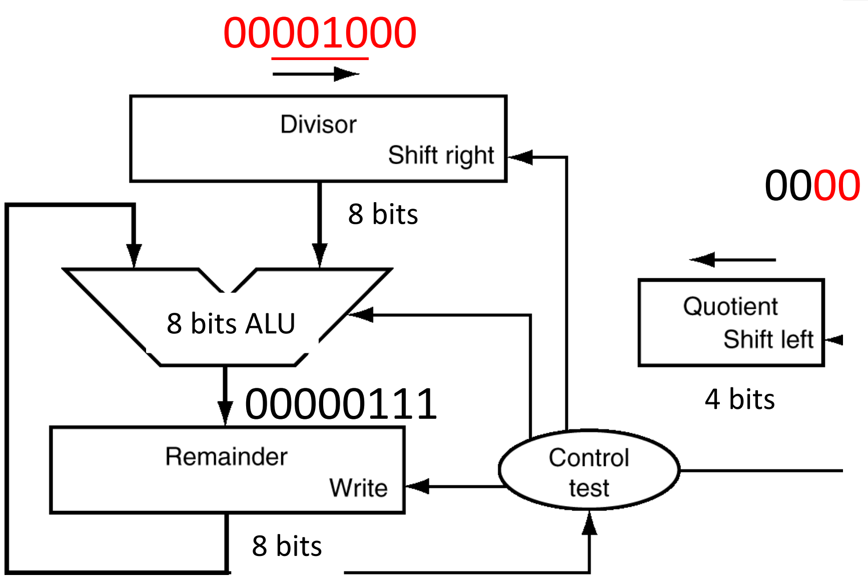


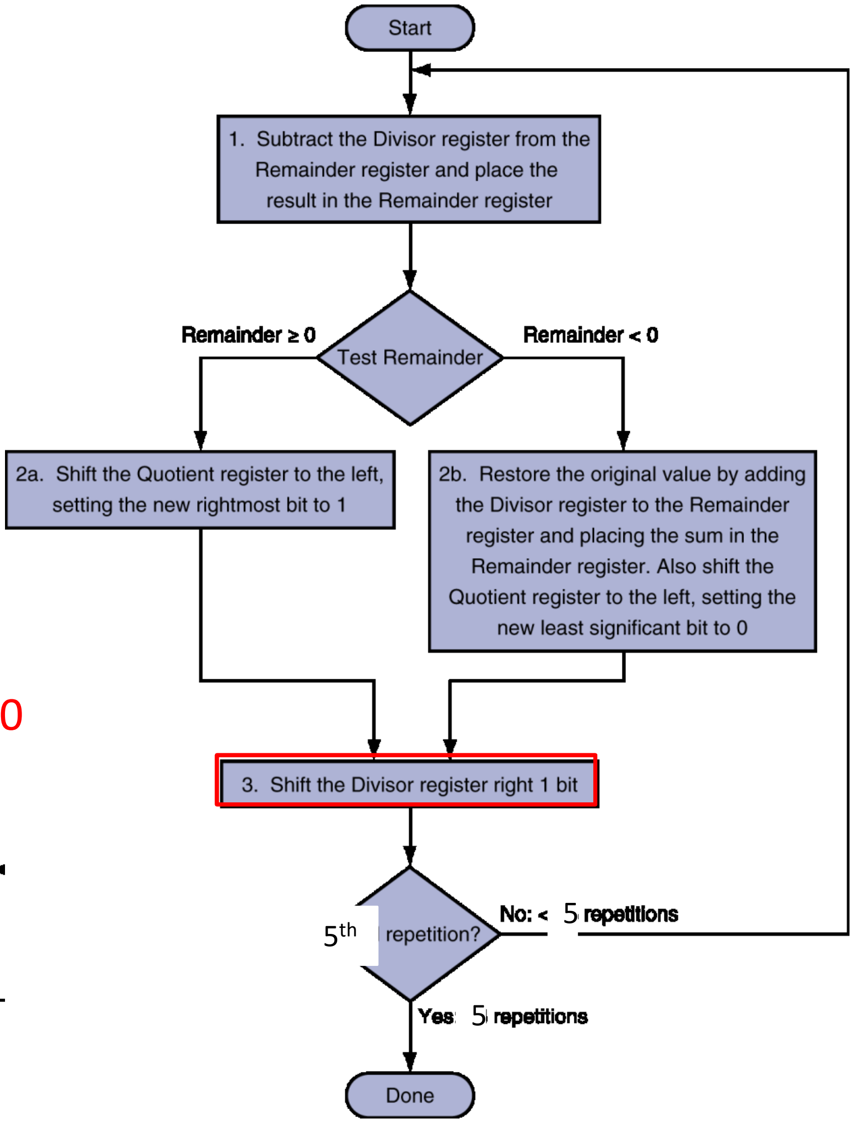


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
LSB of Quotient=0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

2nd iteration

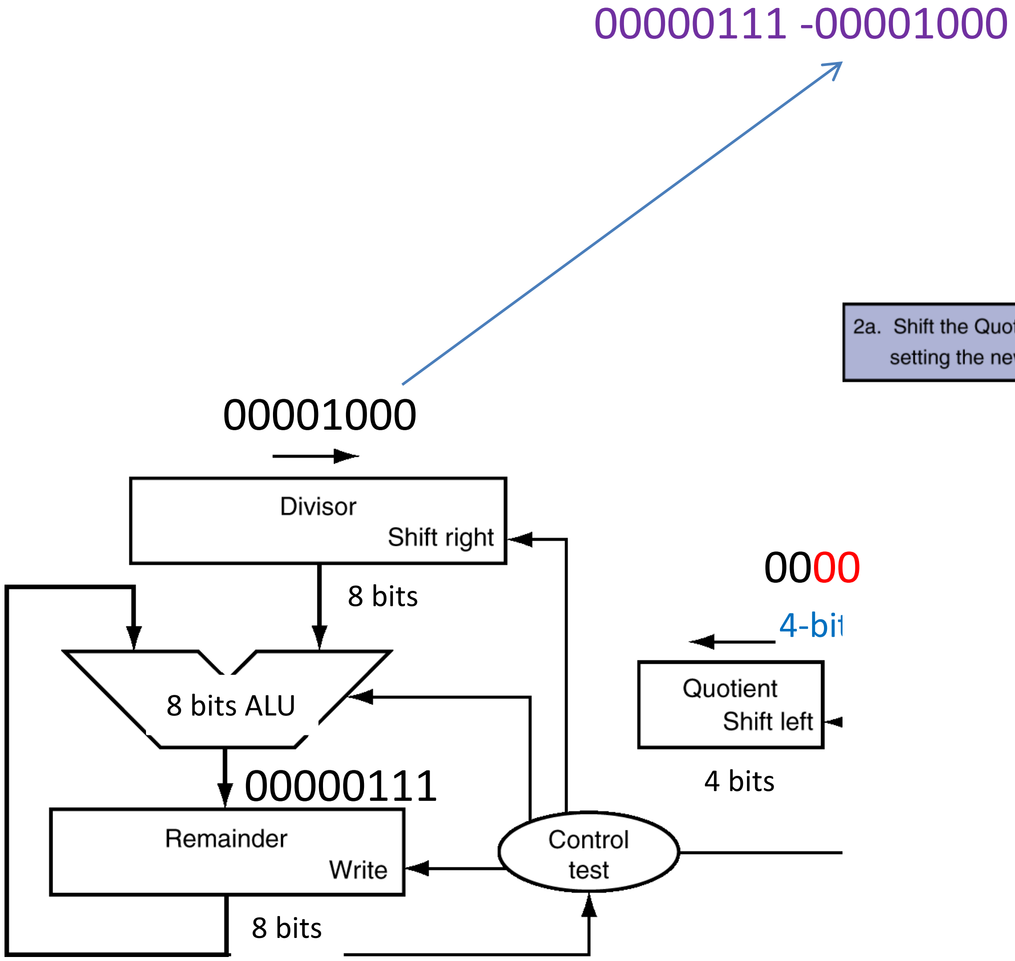


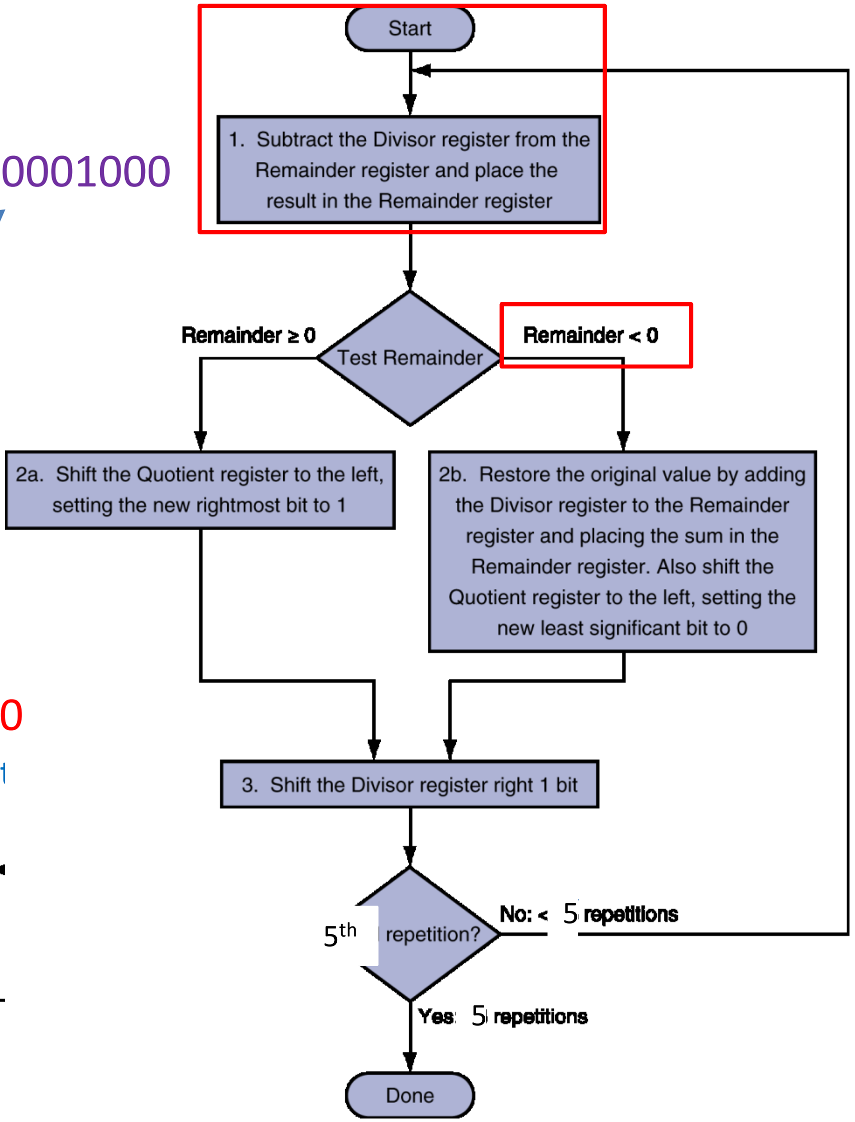


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

3rd iteration

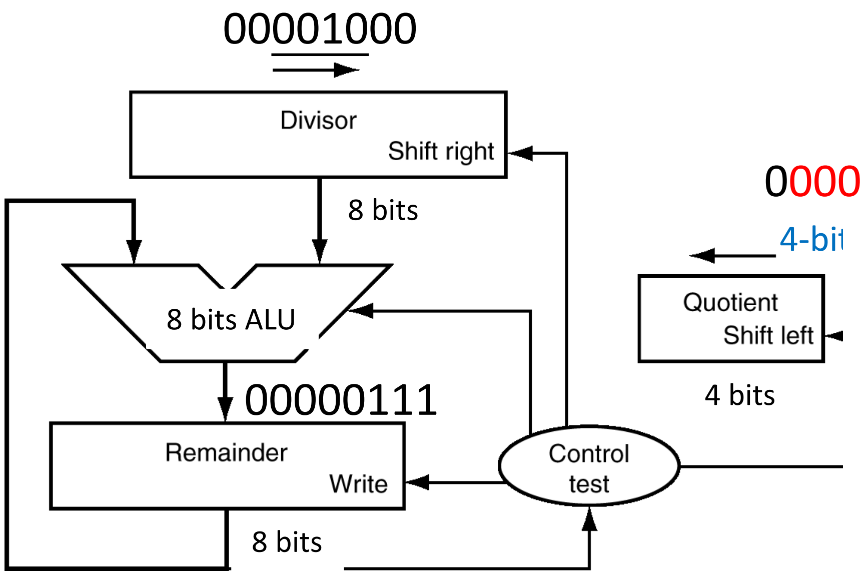


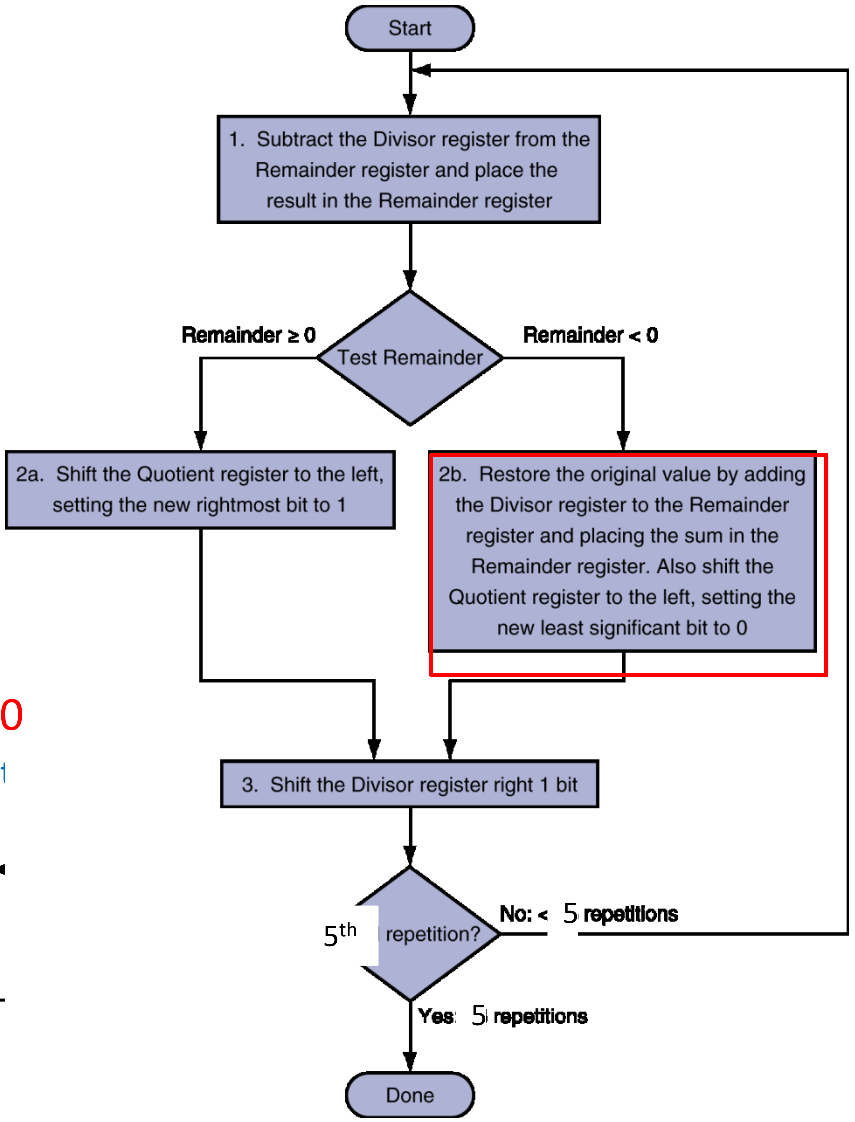


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

3rd iteration

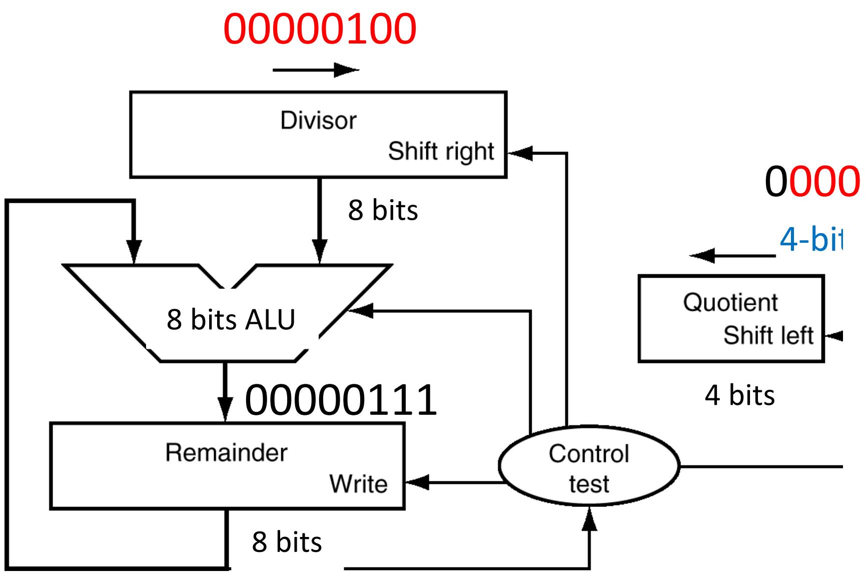


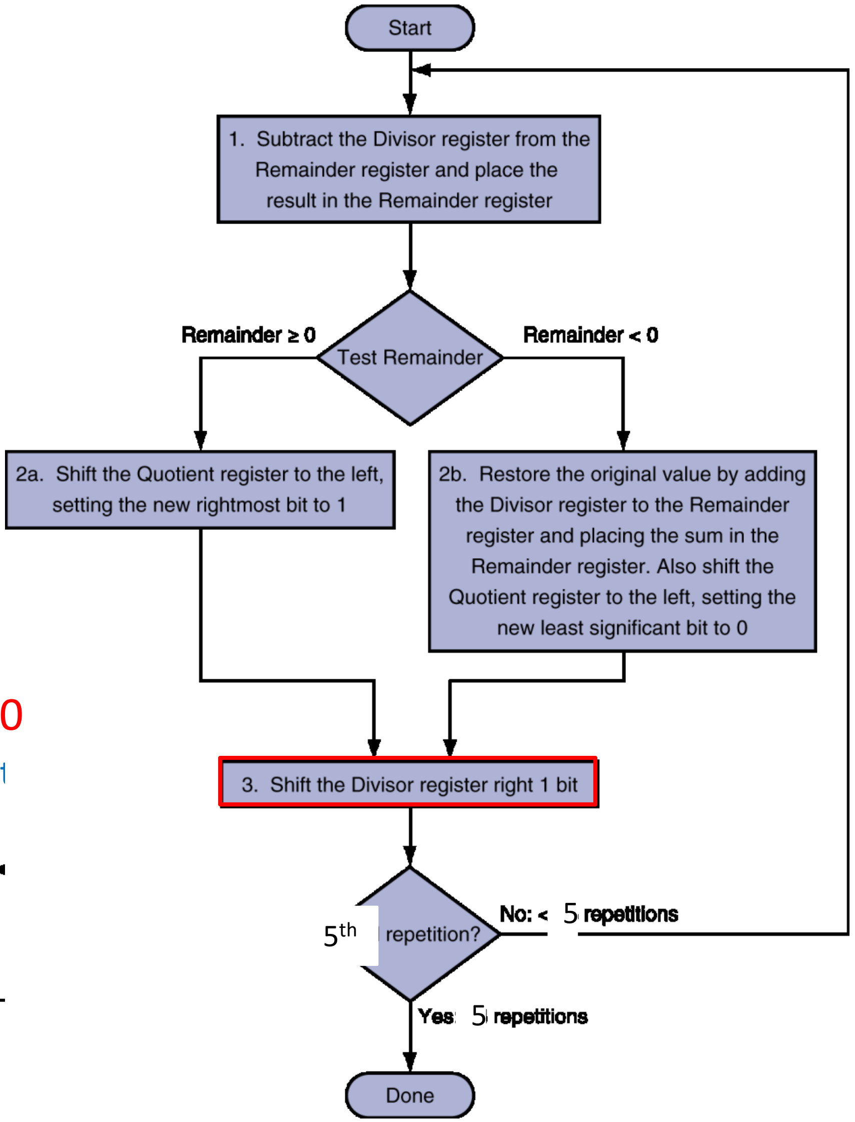


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

3rd iteration

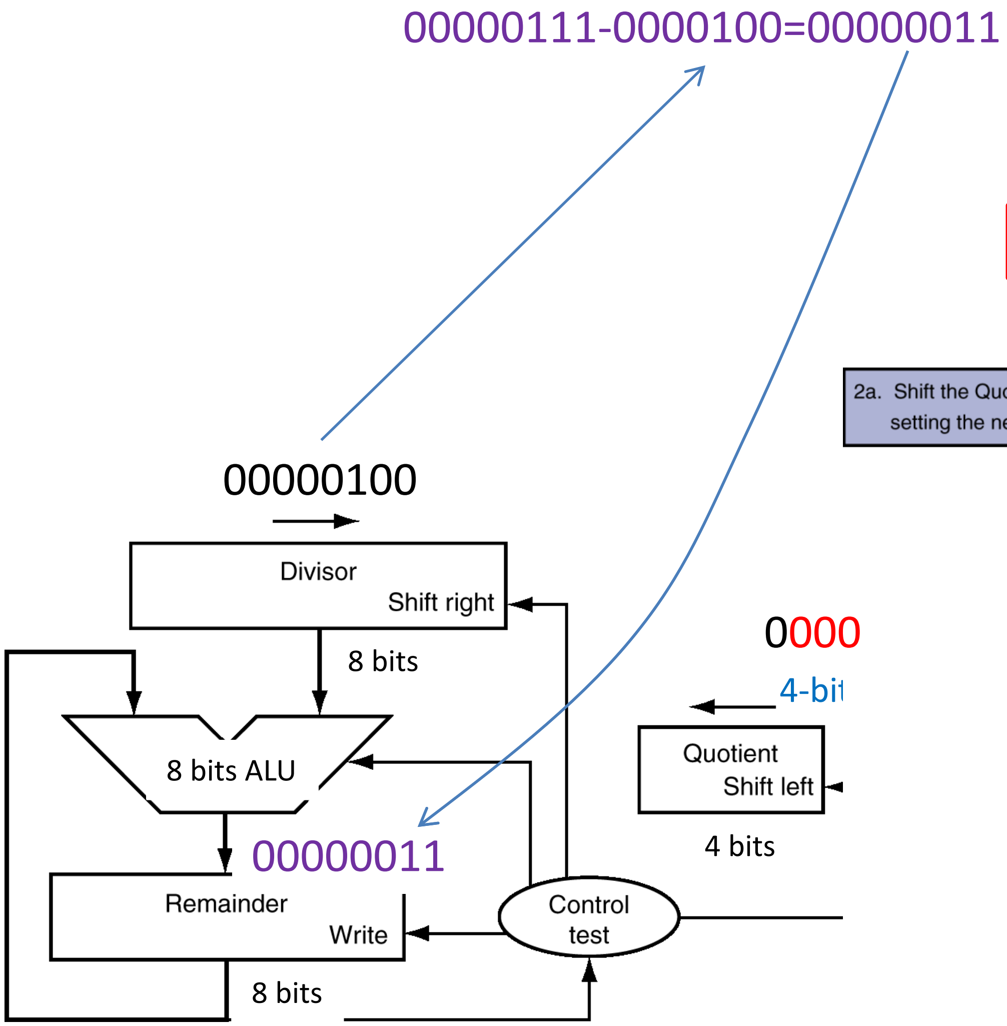


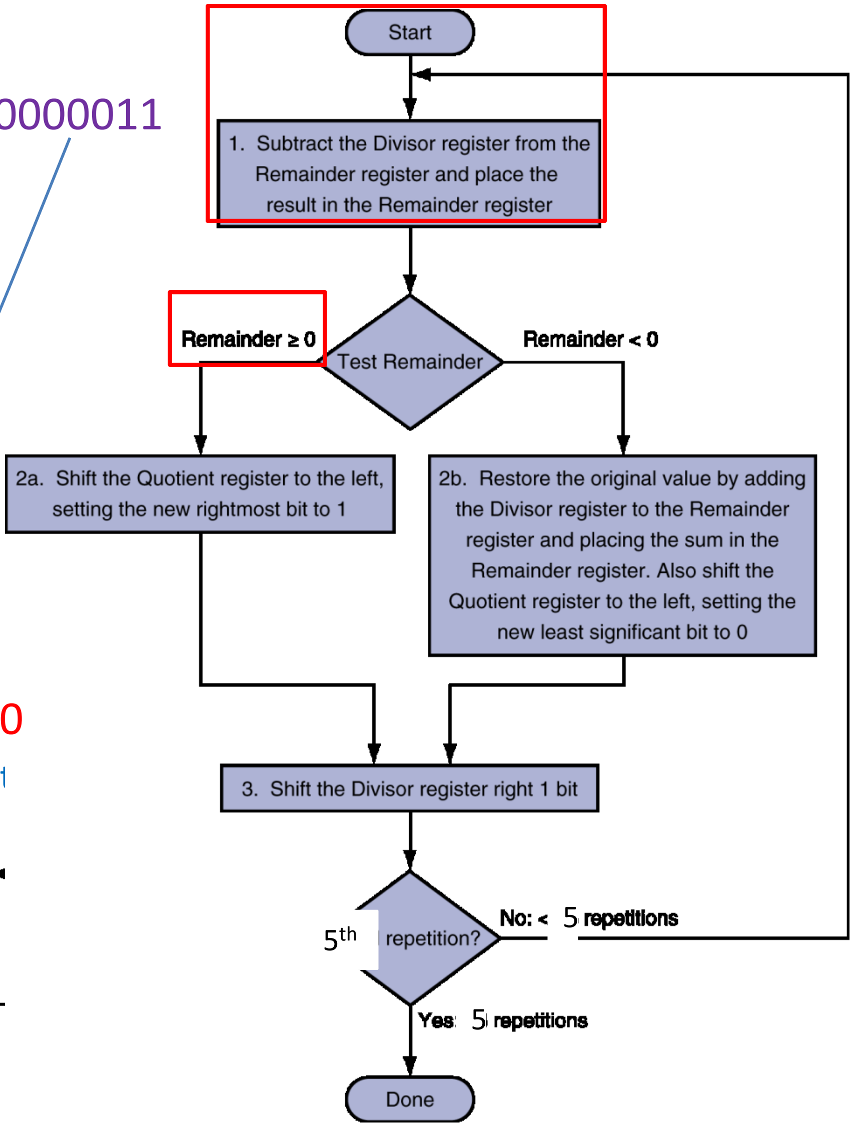


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

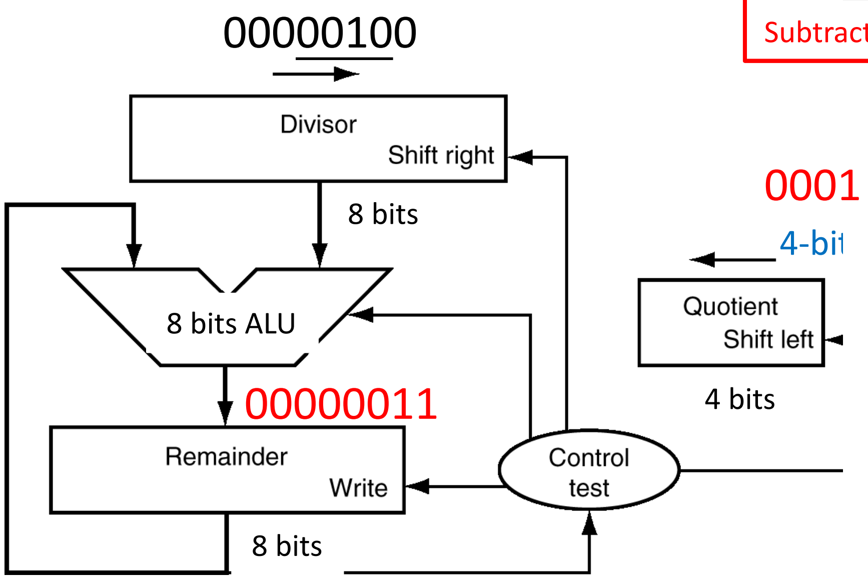
4th iteration

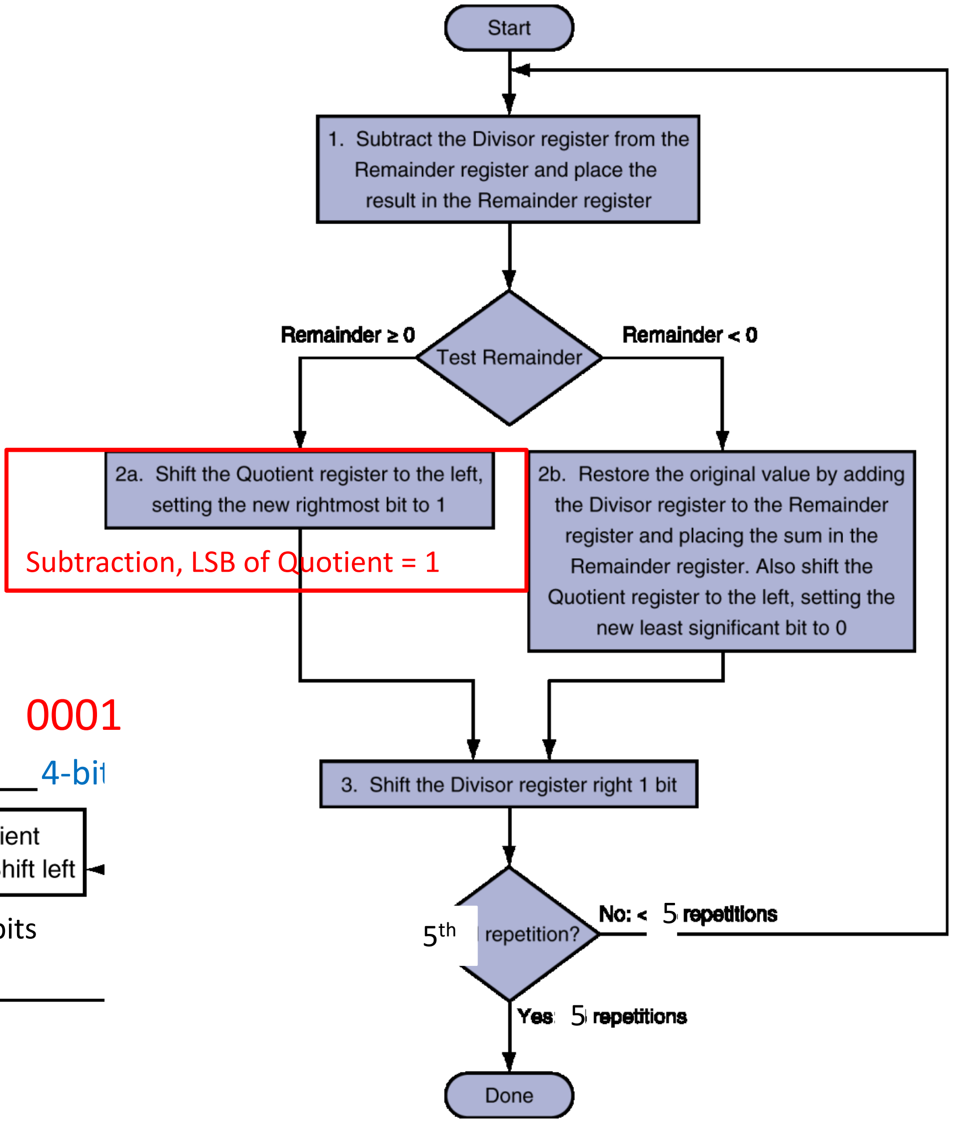



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

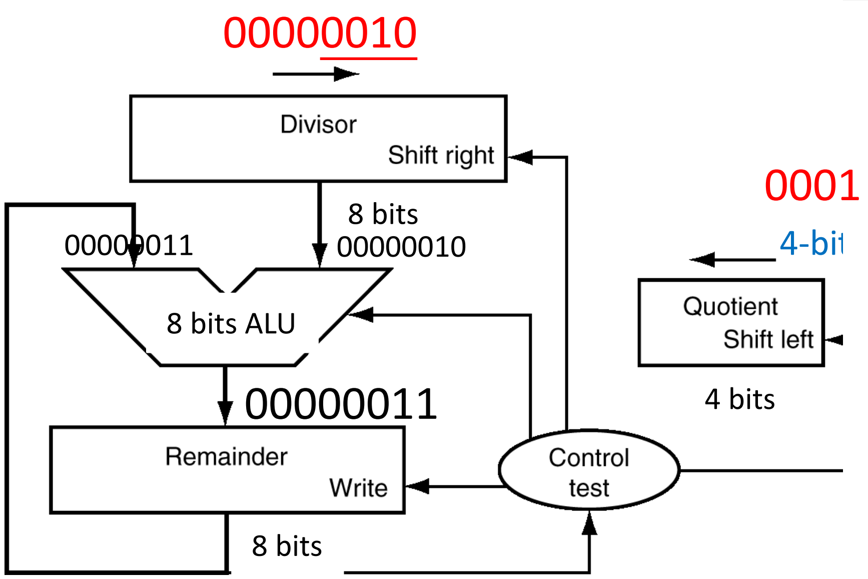
4th iteration

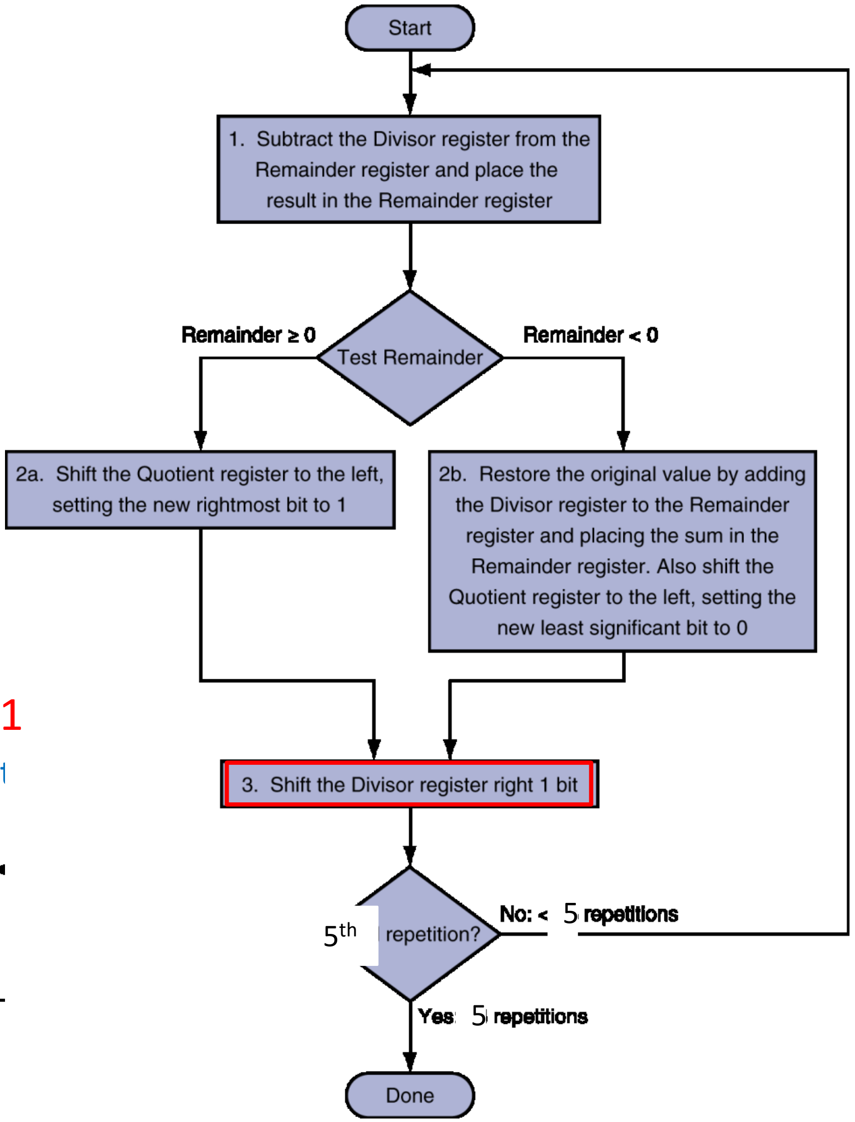



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
Subtraction, LSB of Quotient = 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

4th iteration



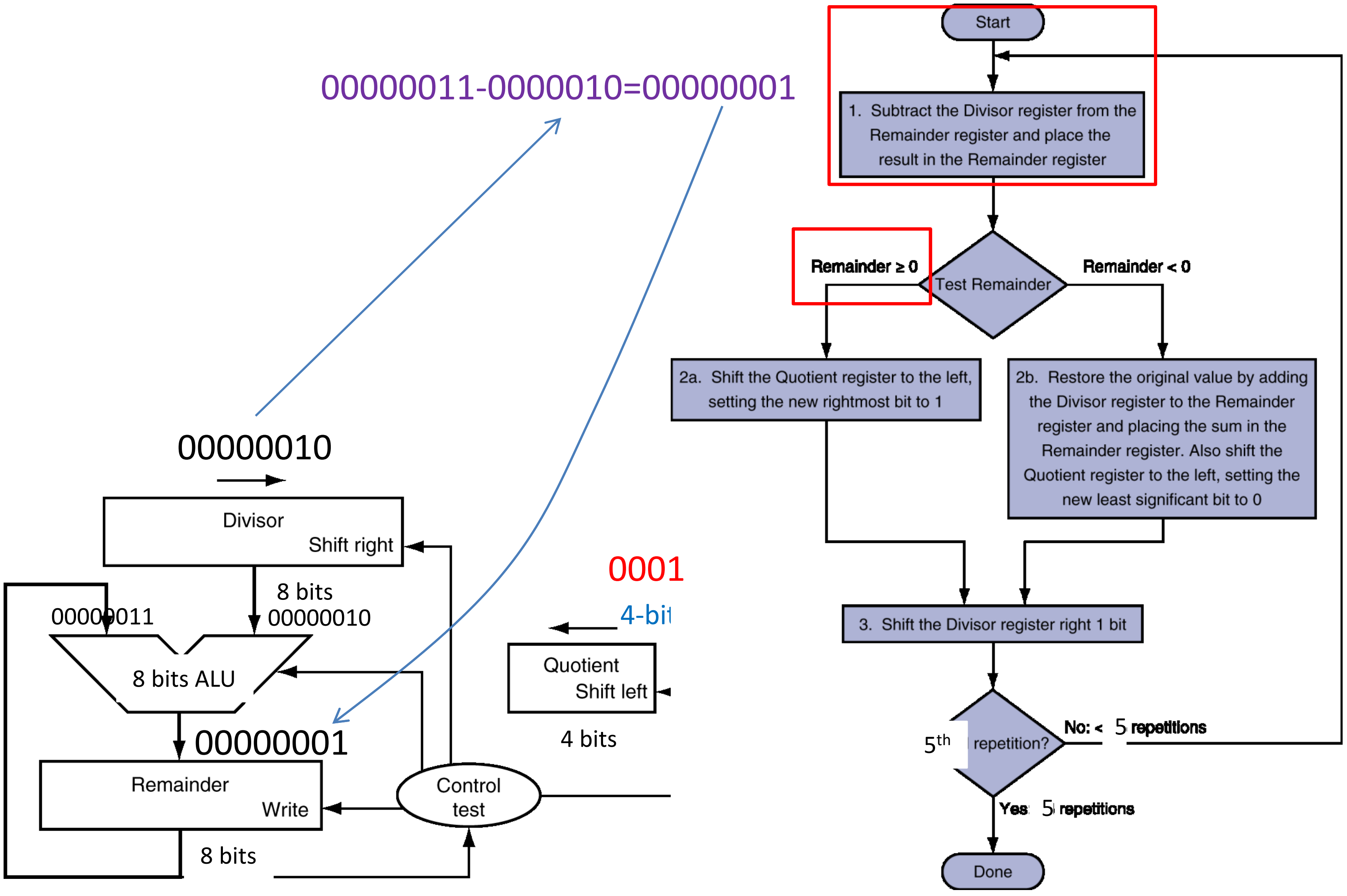


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

5th iteration

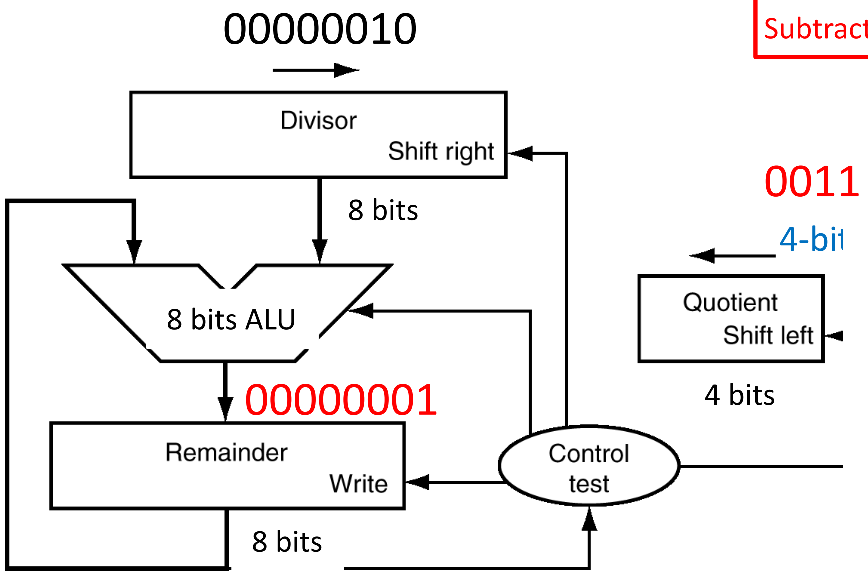
00000011-0000010=00000001

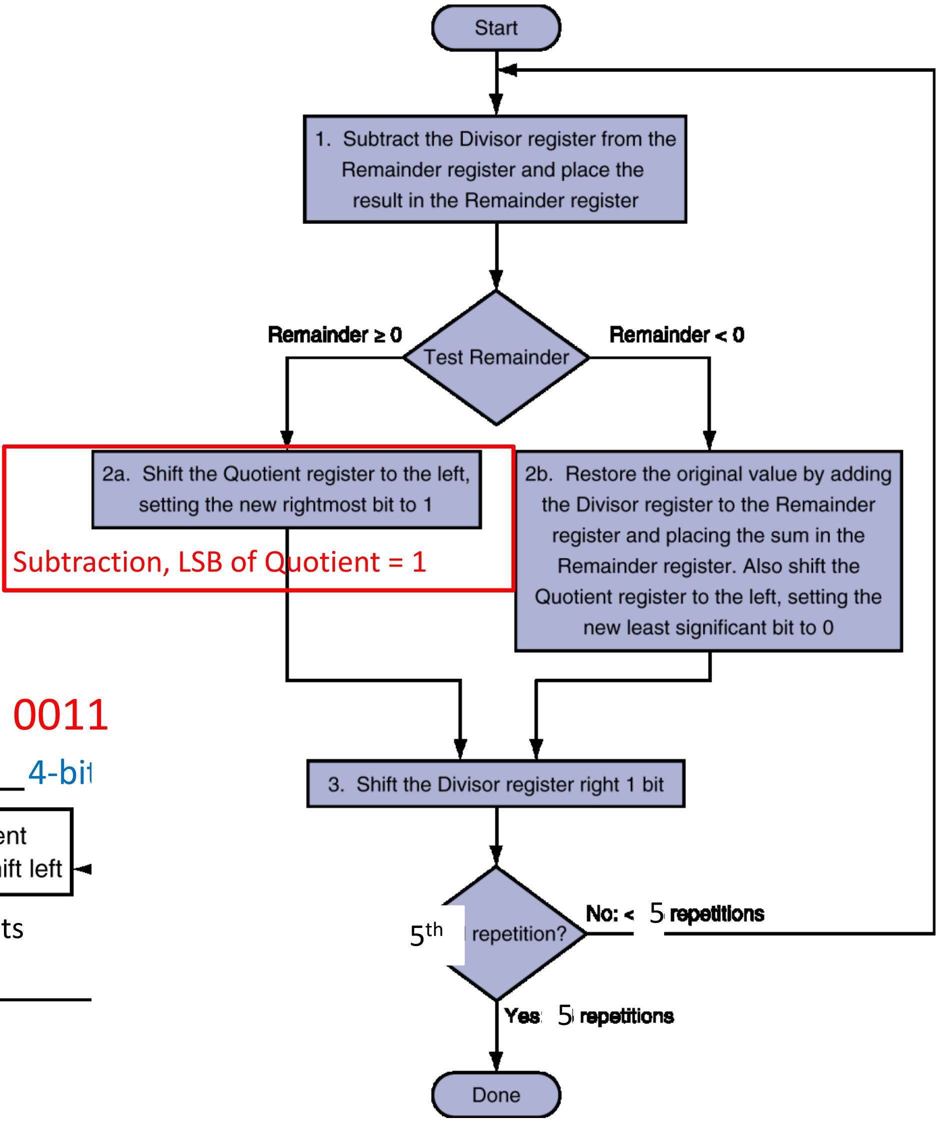


00000010
Divisor
Shift right
8 bits
00000011
00000010
8 bits ALU
00000001
Remainder
Write
8 bits
Control
test
0001
4-bit
Quotient
Shift left
4 bits
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

5th iteration

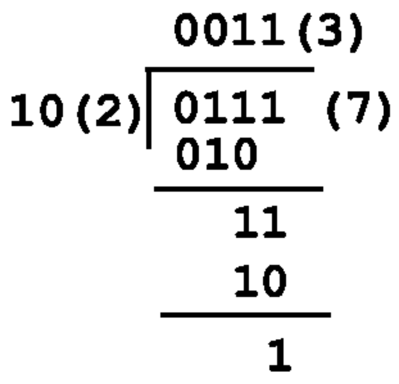


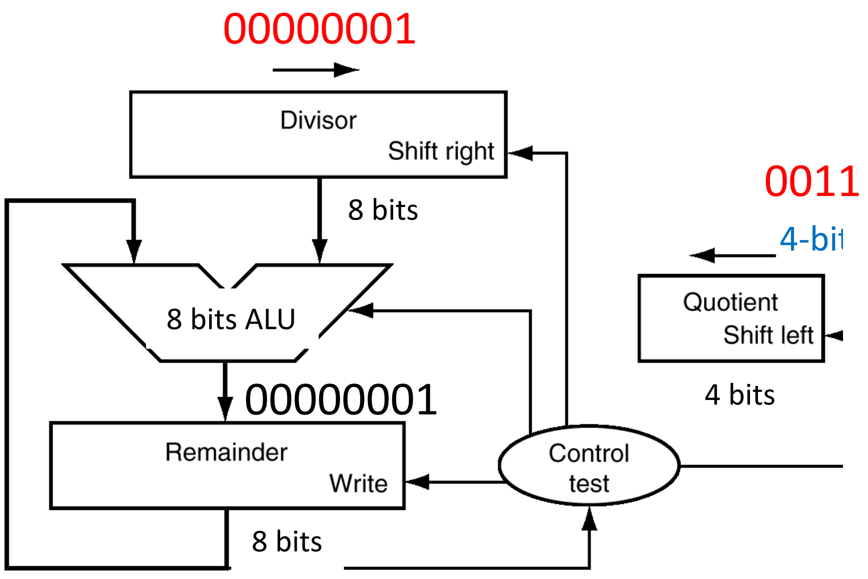


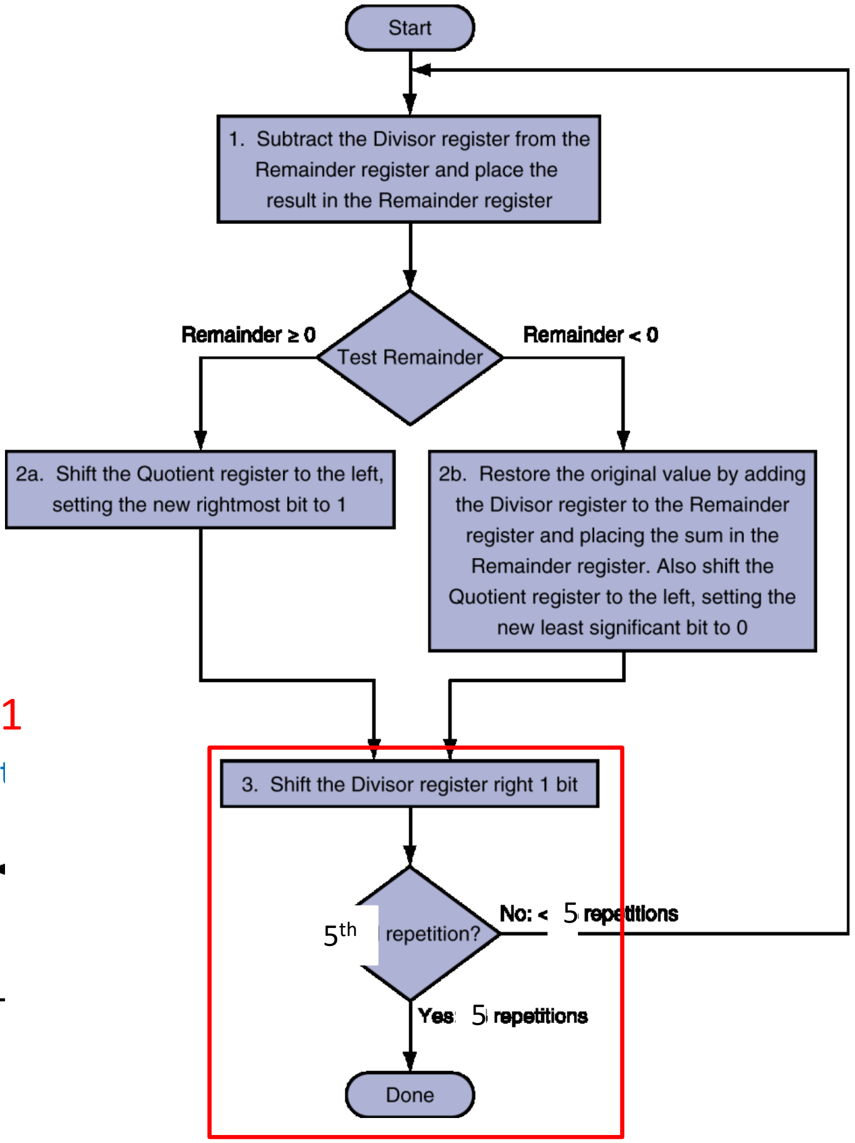
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
Subtraction, LSB of Quotient = 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

4-bit Division Hardware

5th iteration

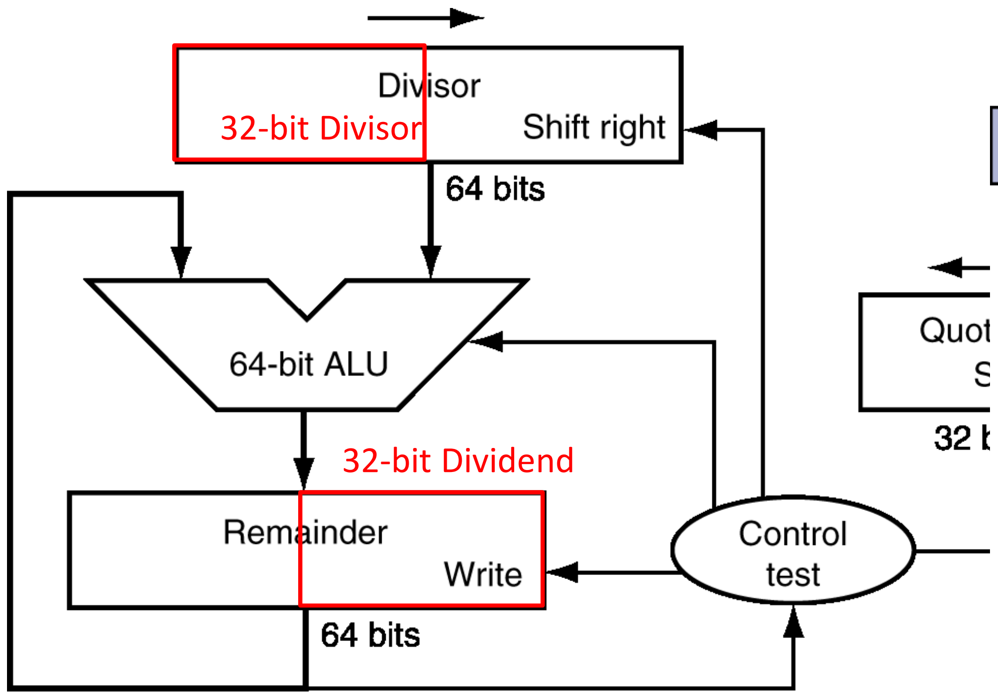


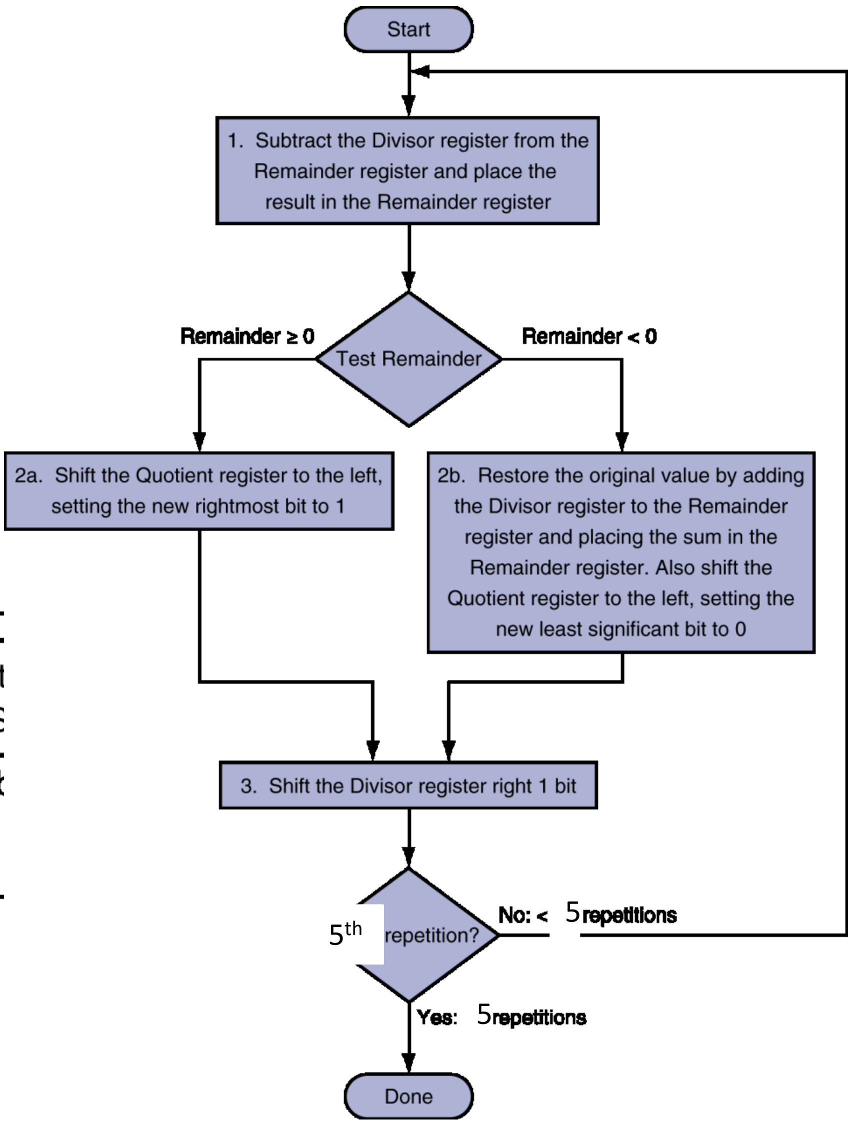




Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

32-bit Division



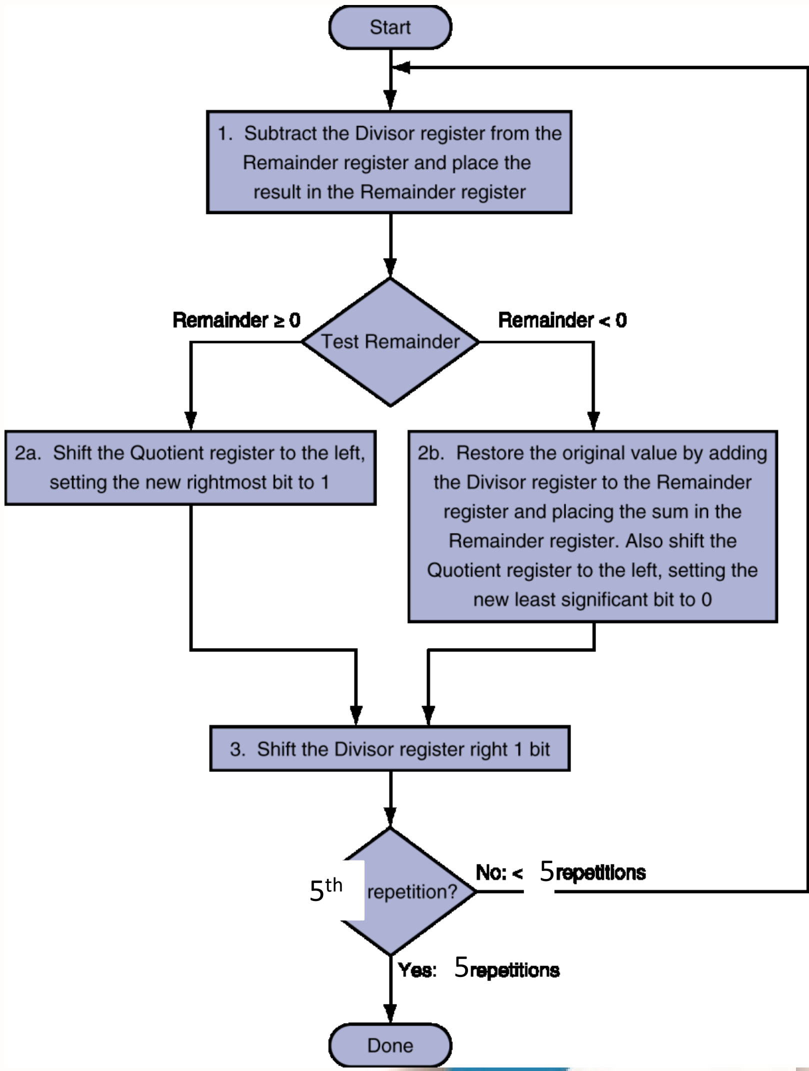

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

0111 / 0010 → 7/2=3…1

-Divisor
11100000

Quot.	Divisor	Remainder	
0000	00100000	00000111	R–D
	Negative	11100111	
		00000111	Restore R
0000	00010000	00000111	lshift Q and set lsb=0 and rshift D
	Negative	11110111	R–D
		00000111	Restore R
0000	00001000	00000111	lshift Q and set lsb=0 and rshift D
	Negative	11111111	R–D
		00000111	Restore R
0000	00000100	00000111	lshift Q and set lsb=0 and rshift D
	Positive	00000011	R–D
0001		00000011	
0001	00000010	00000011	lshift Q and set lsb=1 and rshift D
	Positive	00000001	R–D
0011	5th times	00000001	
0011	00000001	00000001	lshift Q and set lsb=1 and rshift D

Q=0011
R=00000001


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
5th repetition?
No: < 5repetitions
Yes: 5repetitions
Done

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 $\Longrightarrow$ +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 $\Longrightarrow$ +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 $\Longrightarrow$ +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	0000 0011
	2a: Rem ≥ 0 $\Longrightarrow$ 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	0000 0001
	2a: Rem ≥ 0 $\Longrightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Signed Division

- Dividend (Dd) = Quotient (Q) × Divisor (Dv) + Reminder (R)
- The sign of the dividend (Dd) must be the same as that of the reminder (R)
- If the sign of the dividend is not the same as that of the divisor, the quotient (Q) must be negative
- Assuming all are positive, and change the signs in the end

+7 ÷ 2 = 3, remainder = +1

+7 ÷ -2 = -3, remainder = +1

-7 ÷ +2 = -3, remainder = -1

-7 ÷ -2 = +3, remainder = -1

Dd	Q	Dv	R
+	+	+	+
+	−	−	+
−	+	−	−
−	−	+	−