CO101
Principle of Computer Organization

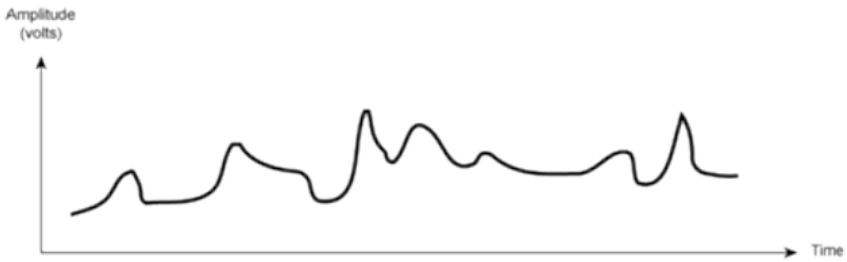
Lecture 02: Arithmetic

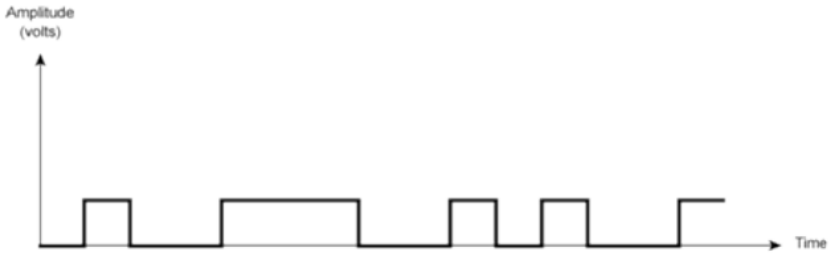
Liang Yanyan

澳門科技大學
Macau of University of Science and Technology

Analog vs. Digital

- Analog Signal
  - Vary in a smooth way over time.
  - Analog data are continuous valued.
    - Example: audio, video
- Digital Signal
  - Maintains a constant level then changes to another constant level (generally operate in one of the two states).
  - Digital data are discrete value.
    - Example: computer data.





Number Systems

- An ordered set of symbols, called digits, with relations defined for addition, subtraction, multiplication, and division.
- Radix or base of the number system is the total number of the number system is the total number of digits allowed in the number system.
- Commonly used numeral systems.

System Name	Decimal	Binary	Octal	Hexadecimal
Radix	10	2	8	16
First seventeen positive integers	0	0	0	0
	1	1	1	1
	2	10	2	2
	3	11	3	3
	4	100	4	4
	5	101	5	5
	6	110	6	6
	7	111	7	7
	8	1000	10	8
	9	1001	11	9
	10	1010	12	A
	11	1011	13	B
	12	1100	14	C
	13	1101	15	D
	14	1110	16	E
	15	1111	17	F
	16	10000	20	10

Conversion from Decimal Integer

- Step 1: Divide the decimal number by the radix (number base).
- Step 2: Save the remainder (first remainder is the least significant digit).

- Repeat step 1 and step 2 until the quotient is zero.
- Result is in reverse order of remainders.

- Exercises:
  - Convert 15_{10} to binary value.
  - Convert 10_{10} to octal value.

Machine number representations

- Machine number representations
  - Fixed point representation
  - Floating point representation
- Fixed point representation
  - Numbers can be represented in any base.
    - 189_{10}, $6EF_{16}$
  - Numbers in computer are represented in binary.
    - A sequence of 0 and 1, each digit is called a bit.
    - 0000 -> 0001 -> 0010 -> 0011 -> 0100 -> 0101 -> …
    - In decimal from 0 to 2^N-1 for N bits.
  - A byte → 8 bits
  - A word → 16 bits, 32 bits? (machine dependent)

Sign and magnitude

- Unsigned number
  - $a_{n-1}a_{n-2}\ldots a_0 = a_{n-1}x2^{n-1} + a_{n-2}x2^{n-2}+\ldots+ a_0x2^0$
  - $101 = 1x2^2 + 0x2^1 + 1x2^0 = 5$
  - For a N-bit number, the range it can represent is $[0, 2^N-1]$
- Sign and magnitude
  - Use the MSB to represent positive or negative number, e.g. 1 → negative, 0 → positive
  - 101 = -1 (since $0x2^1 + 1x2^0 = 1$, MSB = 1, so it is a negative number)
  - For a N bit number, the range it can represent is $[-(2^{N-1}-1), +(2^{N-1}-1)]$

Problems of sign and magnitude

- There are two zeros
  - 0000 = +0
  - 1000 = -0
  - Waste resource
- Computer needs an extra step to check the sign before performing computations.
  - E.g. (1001 + 0010) we cannot add them directly.

Two's complement

- The sign is still indicated by MSB.
  - 1 → negative , 0 → positive.
- Negative number is obtained by two steps.
  - inverse all bits of the positive number
  - add 1

For example, to represent -5 using a 4-bit representation

We know	:	5 = 0101
After inverse all bits:		1010
Plus 1	:	+0001
After plus 1	:	1011 → answer!

Two's complement

- Given a number in two's complement format, what is the actual value?

  Example 1: given 1100, this is a negative number.

We have	:	1100
After inverse all bits:		0011
Plus 1	:	+0001
We have	:	0100 → 4

  As a result, 1100 is -4, since the MSB is 1.

  Example 2: given 0110, since this is a positive number, we just calculate the value as unsigned format.

We have	:	0110 → 6

Two's complement

- The range for an N-bit representation
  - $[-2^{N-1}, +2^{N-1}-1]$
  - E.g. for N=8, the range is [-128, +127]
- Remember to extend the sign when we extend a number. Consider the case that extends a 4-bit number to 8 bits (e.g. a computer uses 4 bits, another one uses 8 bits).
  - 0011 → 0000 0011
  - 1011 → 1111 1011 (the negative sign is extended)
  - If the sign is not extended, 1011 → 0000 1011. After extension, a negative number becomes positive.

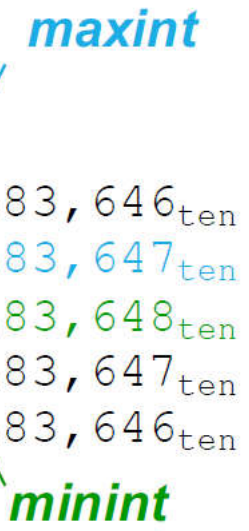
32-bit Binary Representations

- 32-bit signed numbers (2's complement):

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2,147,483,646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2,147,483,648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2,147,483,646_{ten}$
...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$



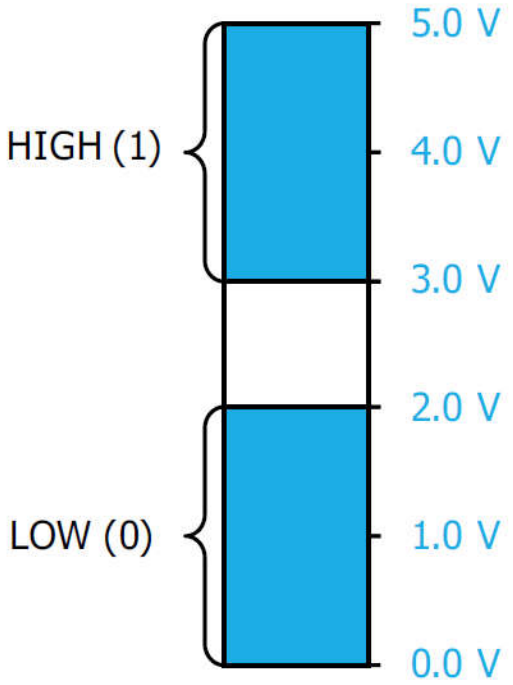
Addition and subtraction

- Addition of X and Y
  - X + Y
- Subtraction
  - X – Y = X + (-Y) = X + (inverse of Y) + 1
- Overflow means a number is out of the representation range.
  - E.g. unable to represent 1023 using 4 bits
- What situations may cause overflow?
  - Addition of two positive numbers, or two negative numbers.
    - 0100 + 0100 or 1000 + 1000
  - Subtraction of a positive and a negative number.
    - 0111 – 1000
- What situations don't cause overflow?
  - Addition of a positive and a negative number
    - 0100 + 1111
  - Subtraction of two negative numbers, or two positive numbers
    - 1000 – 1111 or 0100 - 0111

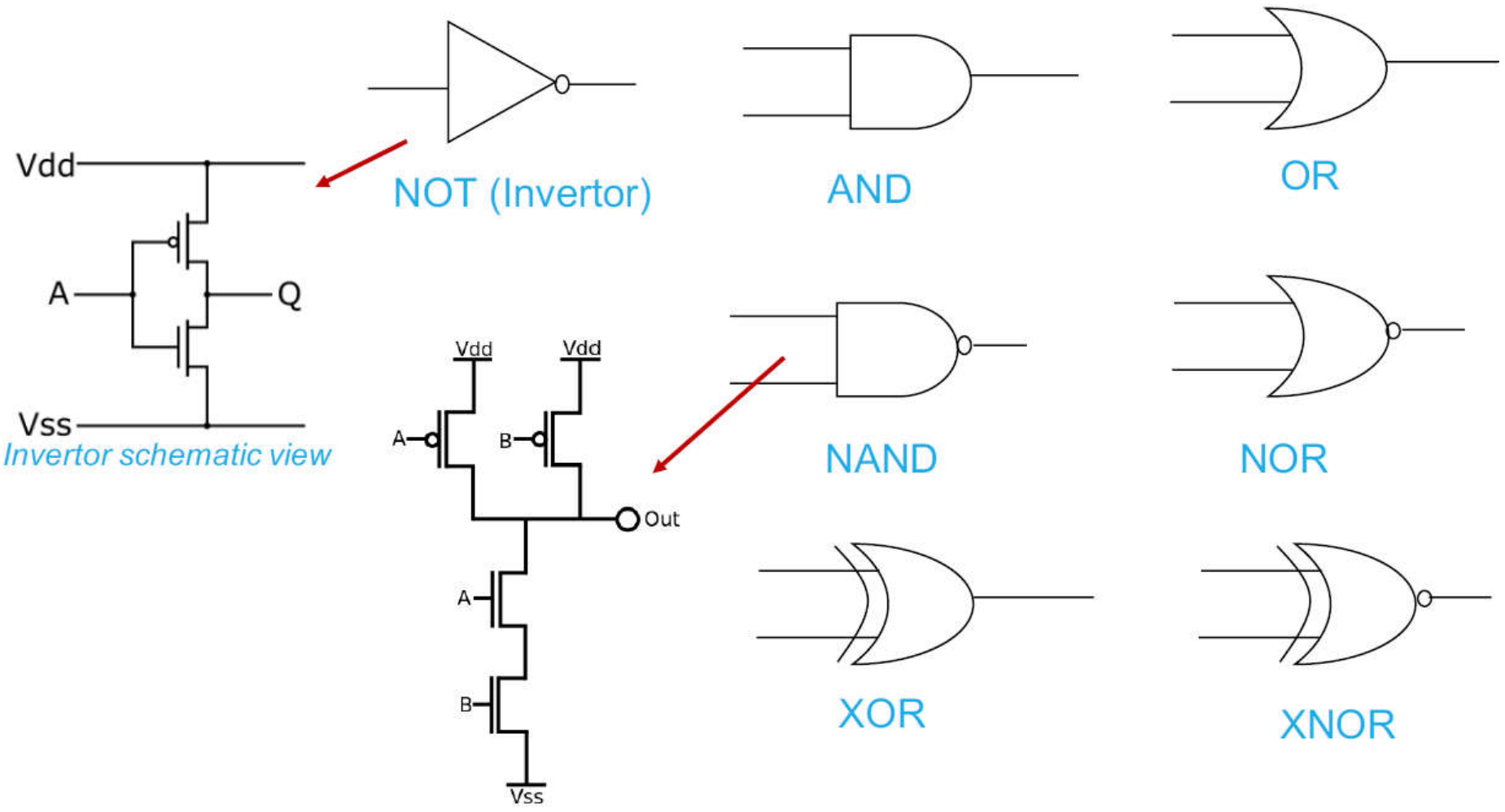
Digital Signal Representation

- Active HIGH
  - High voltage means On
- Active LOW
  - Low voltage means OFF

Logic 0	Logic 1
False	True
Off	On
LOW	HIGH
No	Yes
Open switch	Closed switch



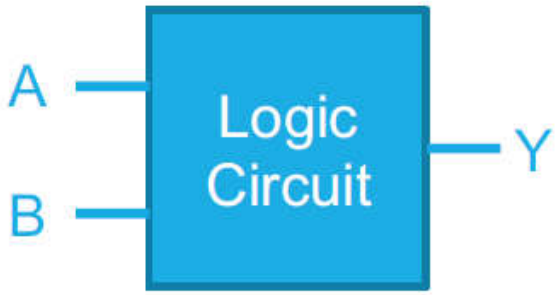
Logic Gates


NOT (Invertor)
AND
OR
Vdd
A
Q
Vss
Invertor schematic view
Vdd
Vdd
A
B
Out
A
B
Vss
NAND
NOR
XOR
XNOR

Truth Table

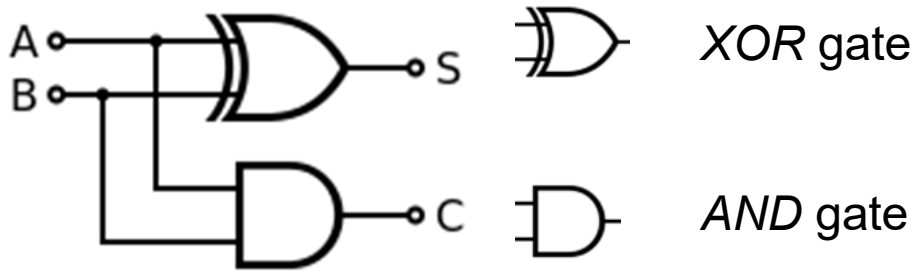
- A means for describing how a logic circuit's output depends on the logic levels present at the circuit's inputs.

- The number of input combinations will equal 2^N for an N-input truth table.



Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

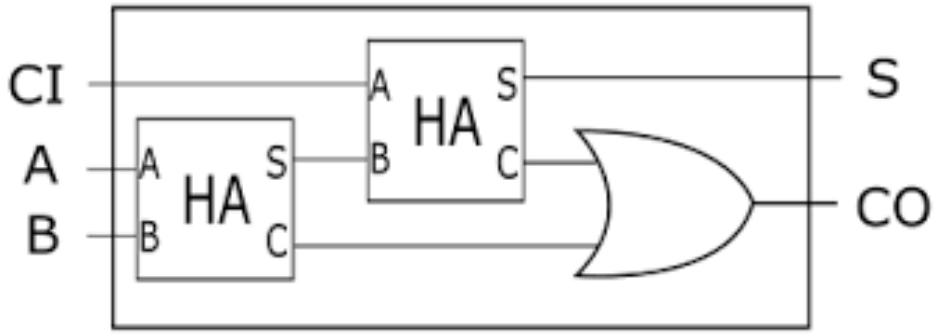
Building a 1-bit Binary Half Adder



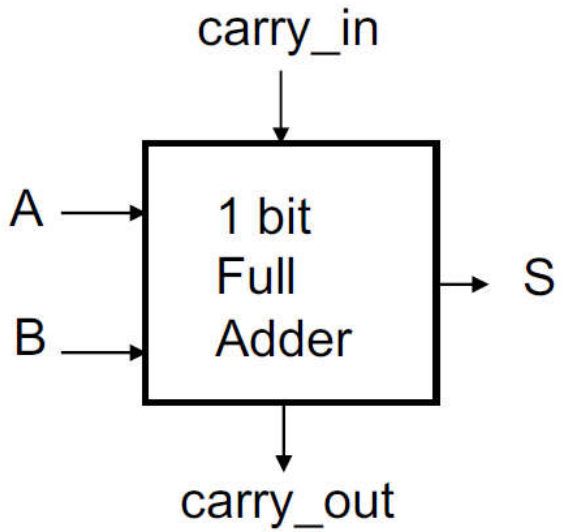
Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

- A: input
- B: input
- S: output
- C: output

- Using two half adders to build a full adder.



Building a 1-bit Binary Full Adder



A	B	carry_in	carry_out	S
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

S = A xor B xor carry_in

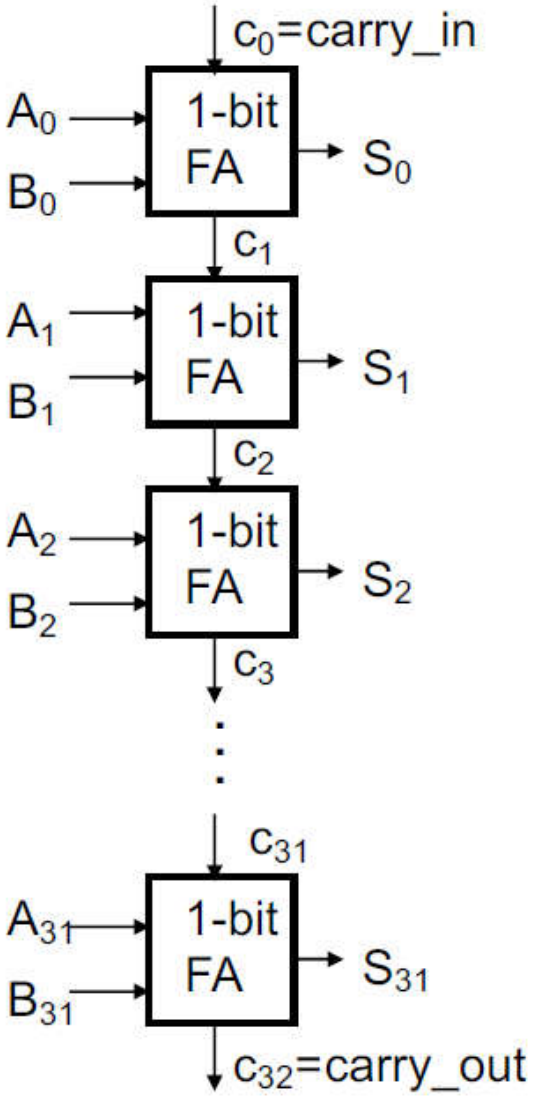
carry_out = A&B | A&carry_in | B&carry_in

(majority function)

How can we use it to build a 32-bit adder?

How can we modify it easily to build an adder/subtractor?

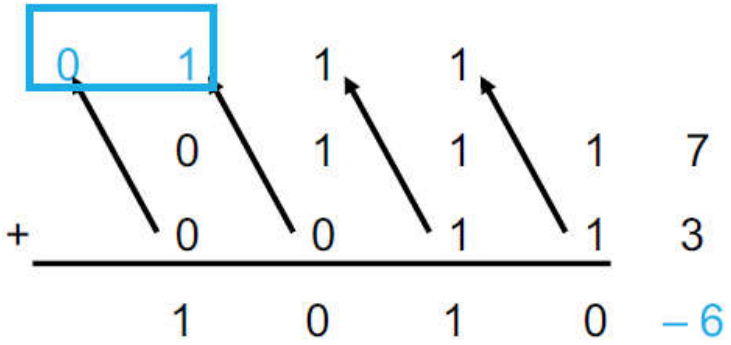
Building 32-bit Adder

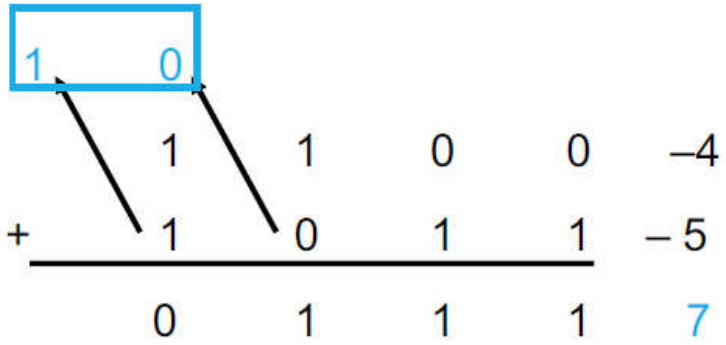


- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect …
- Ripple Carry Adder (RCA)
  - Advantage: simple logic, so small (low cost).
  - Disadvantage: slow and lots of glitching (so lots of energy consumption).

Overflow Detection

- Can detect overflow by:
  - Carry into MSB xor Carry out of MSB.



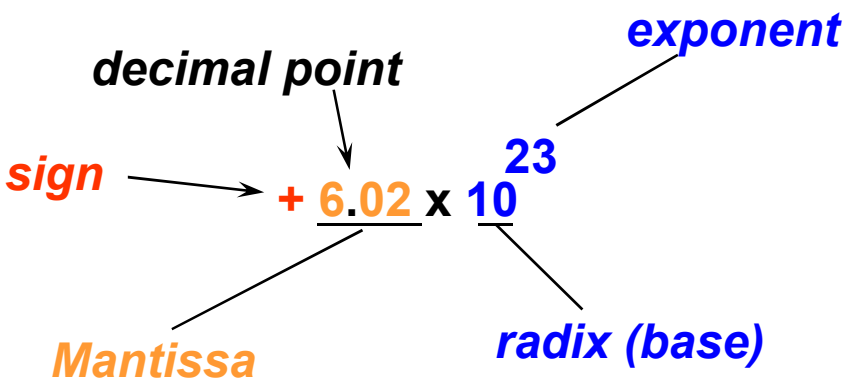


- xor operation:
  - 0 xor 0 = 0;
  - 1 xor 1 = 0;
  - 0 xor 1 = 1;
  - 1 xor 0 = 1;

Floating Point Representation

- Fixed point representation of fractional number (decimal).
  - $a_{n-1}a_{n-2}\ldots a_0 \cdot b_1b_2\ldots b_m = a_{n-1}x2^{n-1} + a_{n-2}x2^{n-2} + \ldots + a_0x2^0 \cdot b_1x2^{-1} + b_2x2^{-2} + \ldots + b_mx2^{-m}$
  - $11.101 = 1x2^1 + 1x2^0 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} = 3.625$
- We need a way to represent
  - Very small numbers, e.g., .000000001. To represent 2^{-98} using fixed point, we need 99 bits.
  - Very large numbers, e.g., $3.15576 * 10^{99}$. To represent 2^{99}-1 using fixed point, we need 99 bits.

Scientific Notation

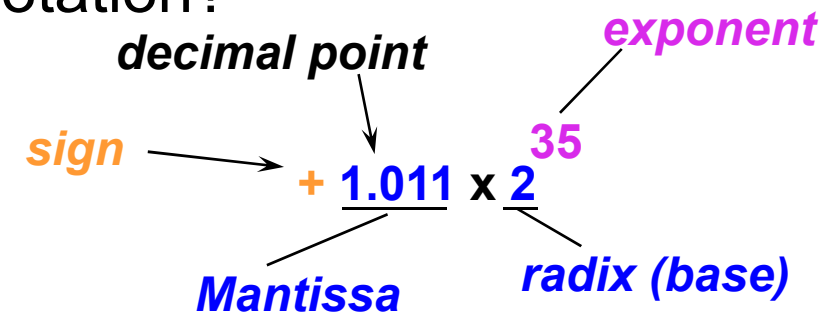


Floating Point Representation

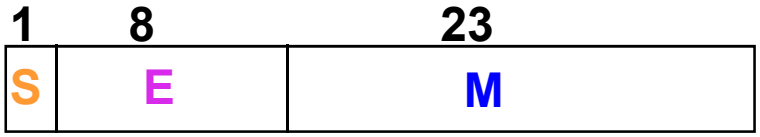
- In mathematics, we have standard (Scientific Notation) for representing fractional numbers.
- In computer systems, we also need a standard to represent fractional numbers → IEEE754 floating point format.
  - The most widely used standard for floating point computation.
- Advantages of the standard (IEEE754):
  - Simplify exchange of data includes fractional number.
  - Simplify floating point arithmetic algorithms.
  - Increase accuracy of the numbers that can be stored.

Floating Point Representation

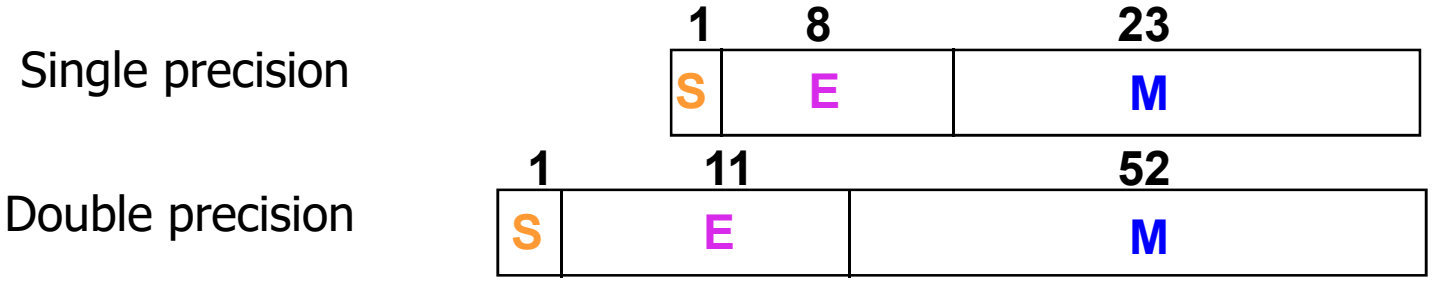
- How do computer systems store a fractional number which is in scientific notation?



- Can you figure out what the number is if I just tell you the mantissa, exponent, and sign?
- In computer systems, fractional numbers are often stored using 32-bit space and divide these 32 bits into 3 parts:
  - S: store the sign.
  - E: store the exponent.
  - M: store the mantissa.
  - No need to store the base.



More details



- Scientific notation in binary: $+1.011 \times 2^{35}$
- S is the sign bit, 0 means +ve, 1 means –ve
- E is the biased exponent, assume E is N bits.
  - E = exponent + bias, e.g. E = 35 + 127
  - bias = $2^{N-1} – 1$, bias is 127 for single precision and 1023 for double precision.
  - exponent = E – bias
- M is the normalized binary number with “hidden” leading ‘1’, called significand.
  - M = Mantissa – 1, e.g. M = 1.011 – 1 = .011
  - Mantissa = 1 + M

$$\text{actual value} = (-1)^S * (1+M) \times 2^{E-\text{bias}}$$

Conversion

- Convert -11.5_{10} into Single Precision FP in Hex
- Convert 0.75_{10} into Single Precision FP in Hex

$-11.5_{10} = -1011.1_2 = -1.0111_2 \times 2^3$ (normalized)

S = 1 (negative number)

E = exponent + 127 = 3+127 = 130 = 10000010

M = Mantissa – 1 = 1.0111 – 1 = .01110000000000000000000

32 bits = 11000001011100000000000000000000

= 0xC1380000

$0.75_{10} = 0.11_2 = 1.1_2 \times 2^{-1}$ (normalized)

S = 0 (positive number)

E = exponent + 127 = -1+127 = 126 =01111110

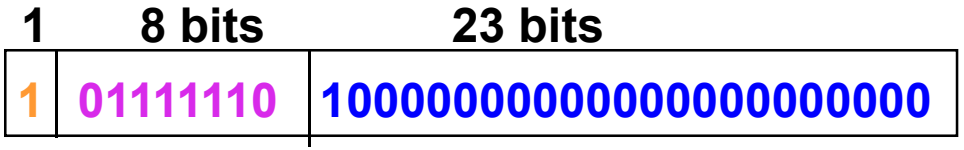
M = Mantissa – 1 = 1.1 – 1 = .10000000000000000000000

32 bits = 00111111010000000000000000000000

= 0x3F400000

Conversion

- What is the decimal value of the following IEEE Single precision number?



sign bit S = 1 = negative

E = 01111110 = 126

exponent = E – 127 = 126-127 = -1

significand M = $0.1000\ldots_2$

Mantissa = 1 + M = $1+0.1000\ldots_2 = 1.1_2$

Value = $- 1.1_2 \times 2^{-1} = -0.11_2 = -(0.5+0.25) = -0.75$

Conversion (Exercises)

- Convert single precision FP 0xC0A00000 to decimal
- Convert single precision FP 0x41380000 to decimal

0xC0A0000 = 11000000101000000000000000000000

sign = 1 = negative

exponent = E – 127 = 129-127 = 2

Mantissa = 1 + M = 1.01_2

value = $-1.01_2 \times 2^2 = -101_2 = -5$

0x41380000 = 01000001001110000000000000000000

sign = 0 = positive

exponent = E – 127 = 130-127 = 3

Mantissa = 1 + M = 1.0111_2

value = $1.0111_2 \times 2^3 = 1011.1_2 = 11.5$

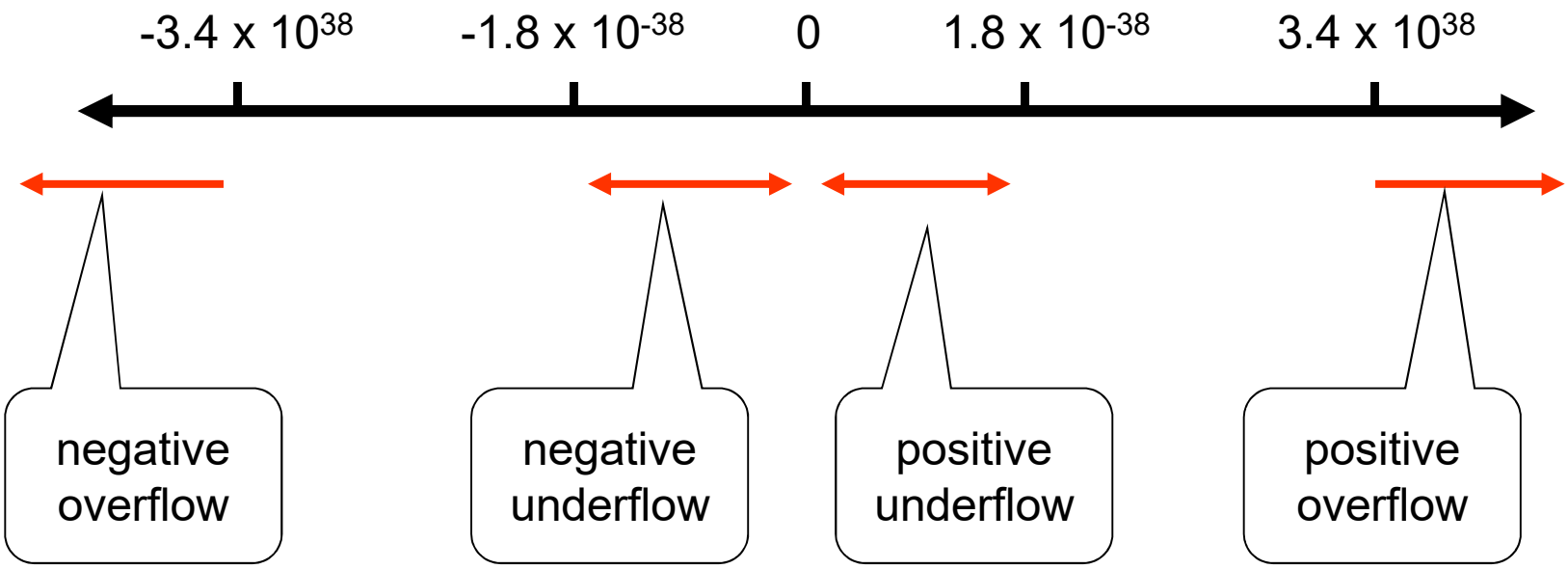
Advantages

- For ease of sorting
  - 1st bit determine whether the number is positive or negative.
  - Sorting is then based on exponent, a larger exponent implies the number is bigger.
- For more accuracy
  - Observe that after normalization, no leading ‘0’ and must start with ‘1’, hidden this leading ‘1’ makes the significand actually 24 bits long (53 bits long for double precision) → can use more bits to store the mantissa.

Range

- Notice that the IEEE 754 standard reserves the smallest E (all 0s) and largest E (all 1s) for special purpose. Will address this issue later.
- Smallest magnitude can be represented
  - Smallest E = 00000001 = 1
  - Smallest M = 00000000000000000000000
  - Smallest magnitude = $1.0_2 \times 2^{(1-127)} \approx 1.8 \times 10^{-38}$
- Largest magnitude can be represented
  - Largest E = 11111110 = 254
  - Largest M = 11111111111111111111
  - Largest magnitude = $1.111_2... \times 2^{(254-127)} \approx 3.4 \times 10^{38}$
- Range the IEEE754 can represent
  - $(+1.8 \times 10^{-38}, +3.4 \times 10^{38})$ and $(-3.4 \times 10^{38}, -1.8 \times 10^{-38})$
- How about zero and numbers outside these range?

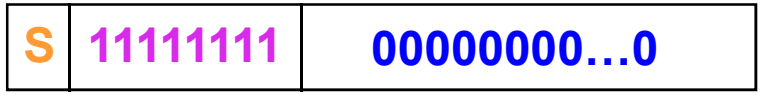
Underflow and overflow



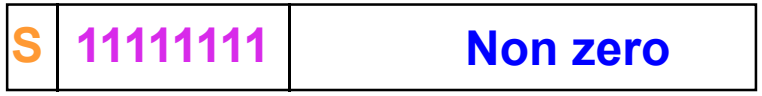
Special number

- Representation of Zero
  - All 0s in E and M
  - +0 and -0 indicated by S (0 mean +)

- Representation of +/- infinity
  - E: all 1s, M: all 0s
  - +/- indicated by S (0 means +)

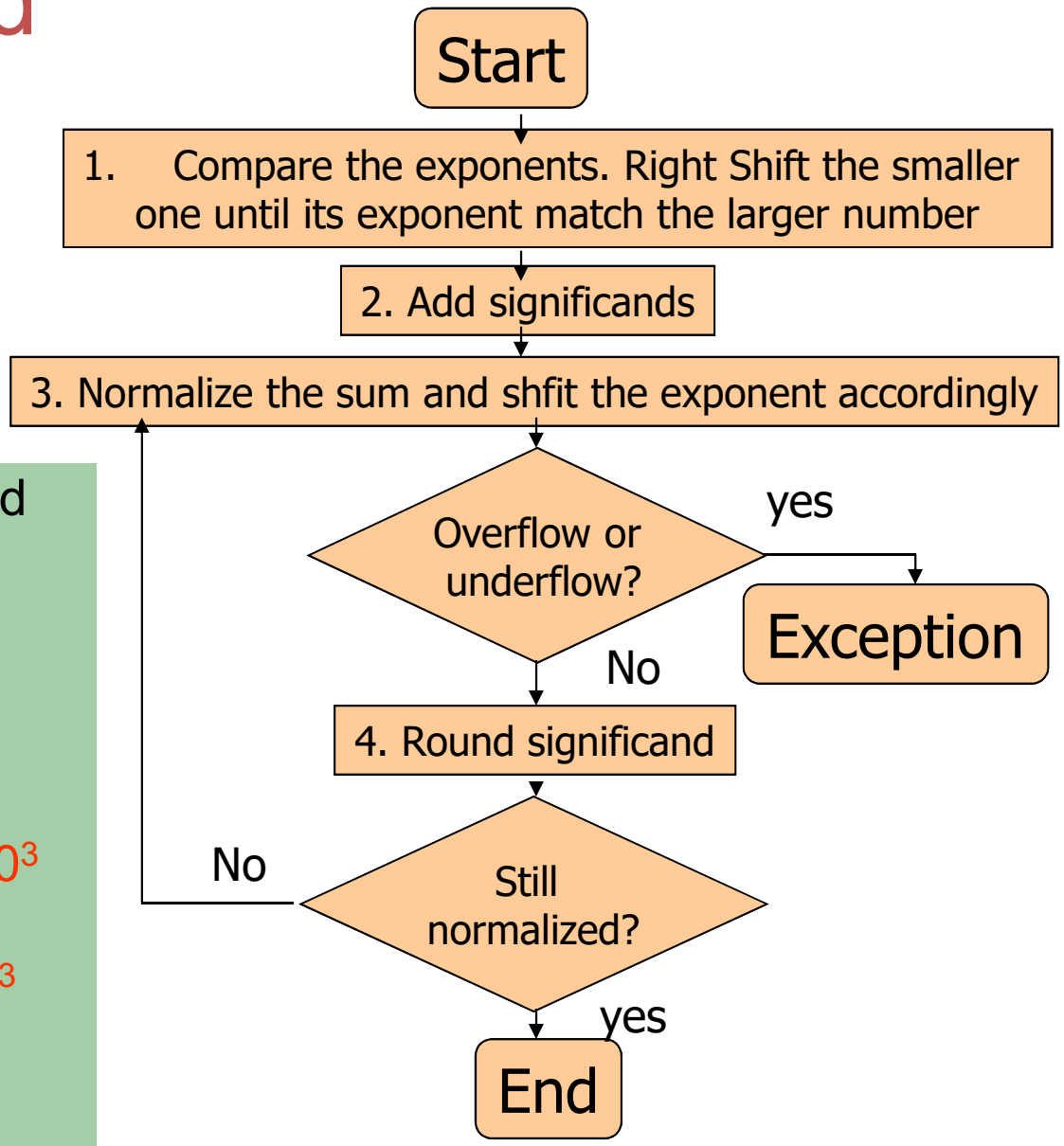


- Representation of Not a Number (NaN)
  - E: all 1s, M: non zero number
  - e.g. sqrt of –ve number
  - HW decides what M is



- Denormalized number
  - E: all 0s, M: non zero number
  - Computers cause exception

Arithmetic Add



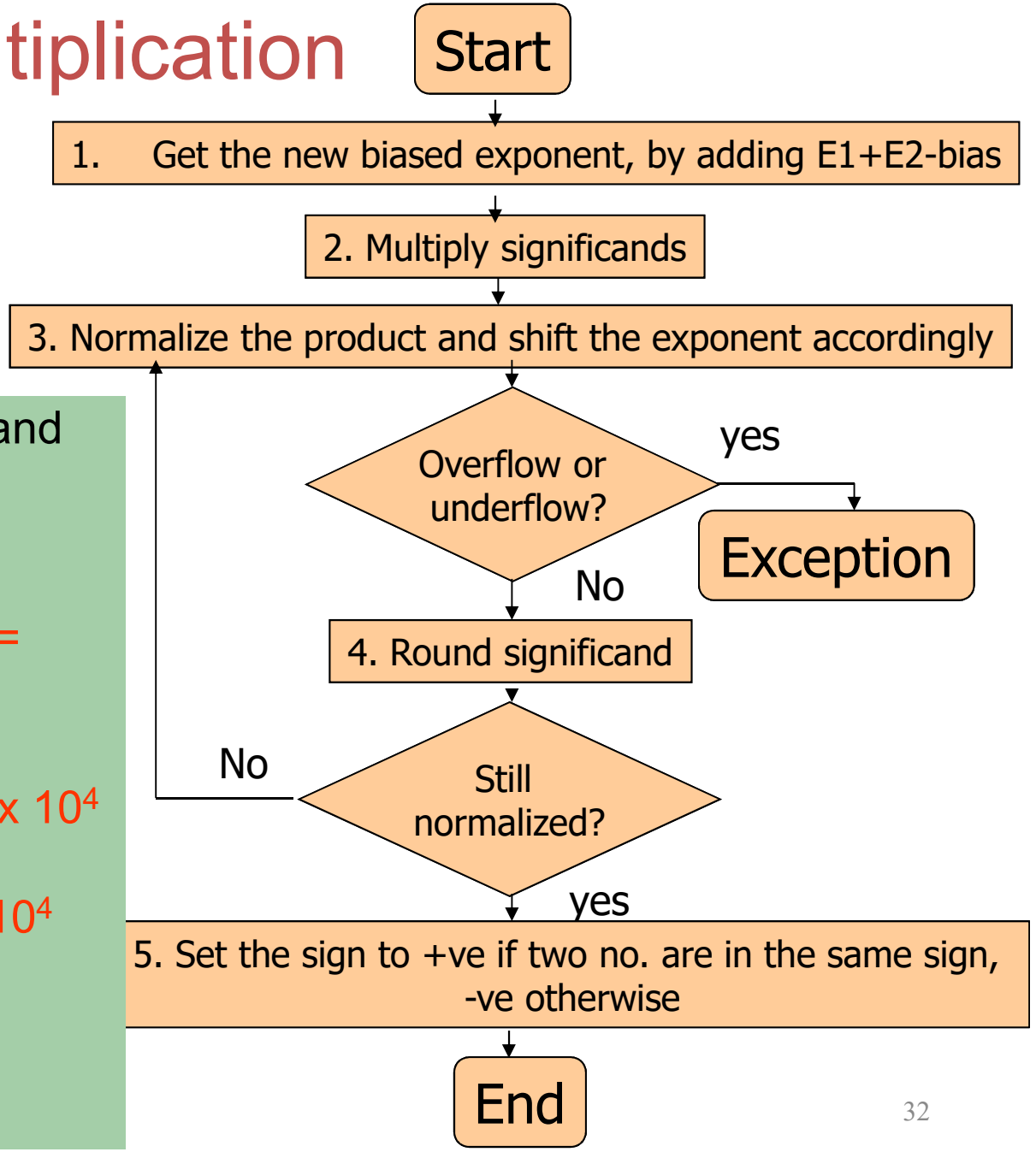
- Assume 3 digits significand and 2 digits exponent
- $6.42 \times 10^1 + 9.51 \times 10^2$

1. $6.42 \times 10^1 = 0.642 \times 10^2$
2. $0.642+9.51 = 10.152$
3. $10.152 \times 10^2 = 1.0152 \times 10^3$ (no over/under flow)
4. After rounding, 1.015×10^3 (normalized)
5. Ans : 1.015×10^3

Arithmetic Multiplication

- Assume 3 digits significand and 2 digits exponent
- $8.76 \times 10^1 * 1.47 \times 10^2$

1. $(1+127)+(2+127) - 127 = (3+127)$
2. $8.76 \times 1.47 = 12.8772$
3. $12.8772 \times 10^3 = 1.28772 \times 10^4$ (No over/under flow)
4. After rounding, 1.288×10^4 (normalized)
5. Set to +ve
6. Ans : 1.288×10^4



Common Pitfall for FP

- Floating point addition is NOT associative.
- Is (a+b)+c equivalent to a+(b+c)?
- No in some circumstances, but why?
  - FP numbers have limited precision, only approximate result can be stored.
  - a = -1.5×10^{38}, b = 1.5×10^{38}, c = 1.0
  - Everyone knows (a+b)+c = a+(b+c) = 1.0 in mathematics.
  - How about do this in C program using *float* type?

Common Pitfall for FP

$a = -1.5 \times 10^{38}$, $b = 1.5 \times 10^{38}$, $c = 1.0$

- printf (“%f”, (a+b)+c); Result = 1.0
- printf (“%f”, a+(b+c)); Result = 0

- It is nearly no effect for adding a very small number to a very big number.
  - The small number becomes zero when normalizes the exponents of two numbers (step 1 of addition algorithm).

- Don’t assume associate rule holds for Floating Point Number! Always avoid adding a very small number to a very big number first.

Common Pitfall for FP

- Case 1
  - $a = -2^{38}$, $b = 2^{38}$, $c = 1.0$
  - Is ((a+b)+c) equal to (a+(b+c))?
- Case 2
  - $a = -2^{28}$, $b = 2^{28}$, $c = 1.0$
  - Is ((a+b)+c) equal to (a+(b+c))?
- Case 3
  - $a = -2^{2}$, $b = 2^{2}$, $c = 1.0$
  - Is (a+b)+c) equal to (a+(b+c))?

NOT EQUAL

Decreasing the exponent.

EQUAL

- Q1: What is the largest exponent that these two equations are still equal?
- Q2: What is the smallest exponent that these two equations are not equal?

Common Pitfall for FP

- Equality test may NOT hold for FP.
- Is 10/3*3 equal to 10?
- No, but why?
  - Round off errors easily occur during each intermediate step!
  - Not wise to test ‘absolute’ equality of two FP numbers.

Common Pitfall for FP

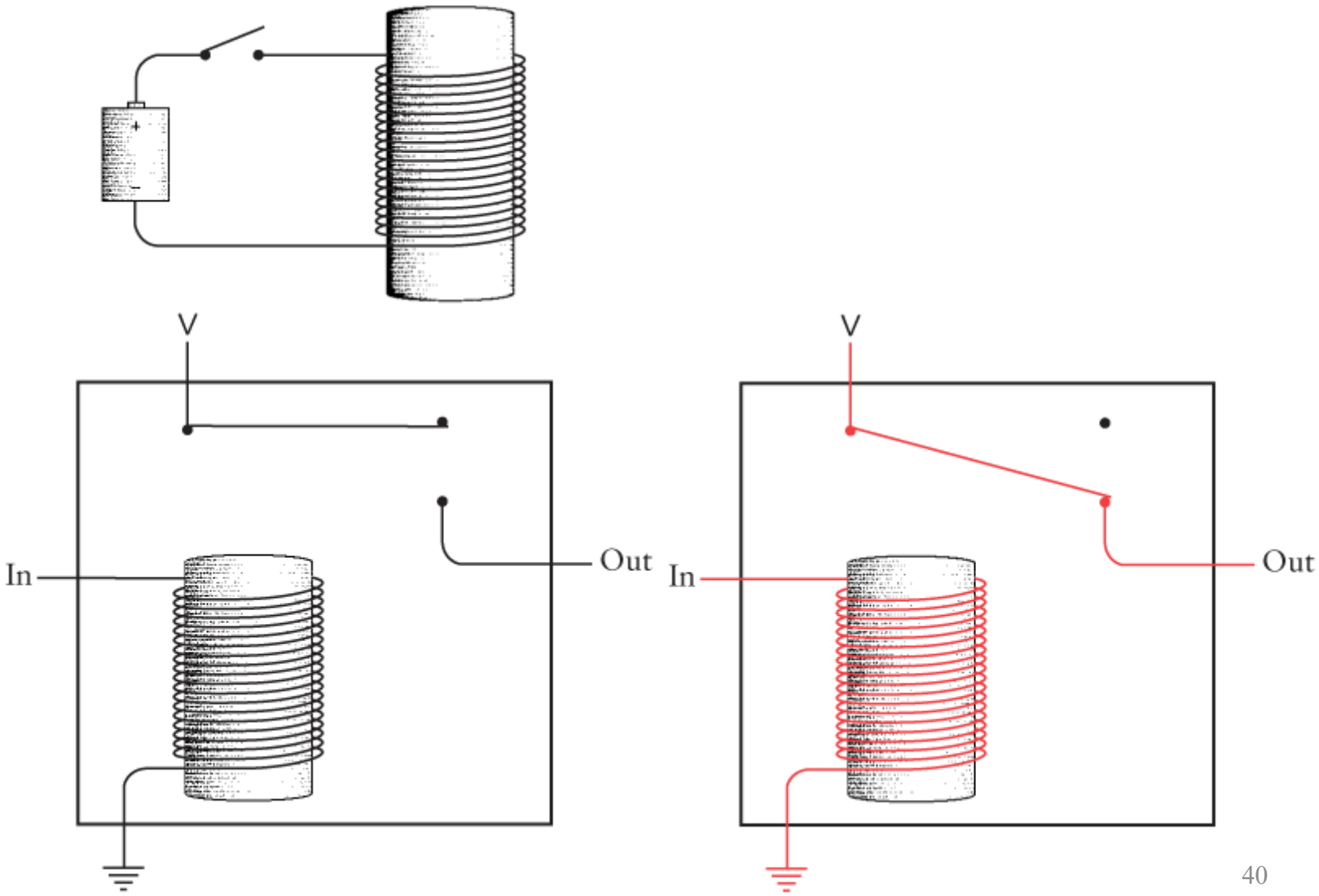
```
float d;
int e = 10;

d = (10/3)*3;

if (d==e)
     printf("1. equality work!\n");
else
     printf("2. equality not work!\n");

if (d > e)
     printf("3. compare magnitude work!\n");
```

Summary

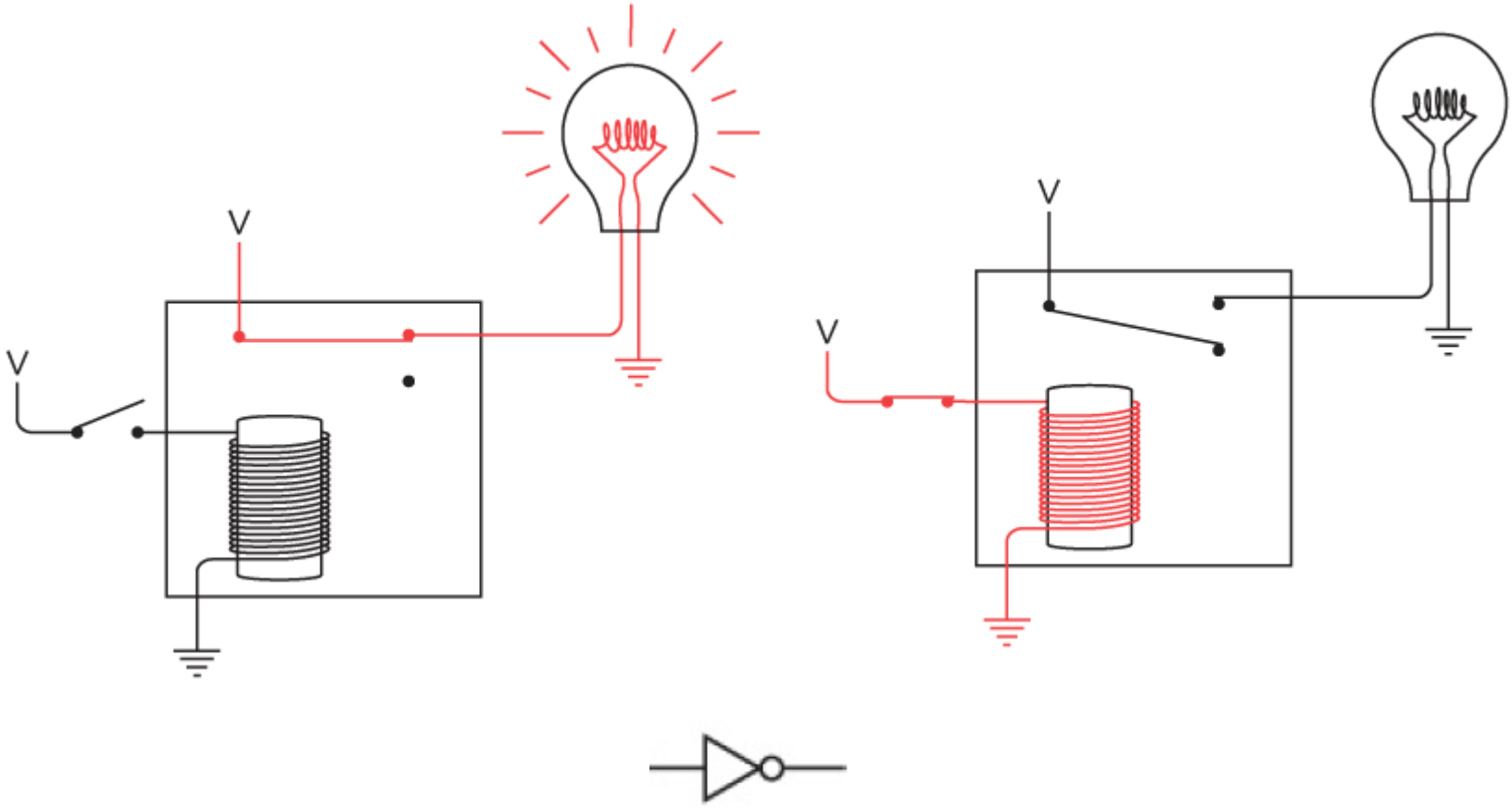
- Fixed point representation
  - Two's complement
- Floating point representation
  - IEEE754
  - Conversion between single precision floating point format and decimal format.
  - Addition and multiplication
  - Pitfall

How does a computer work?

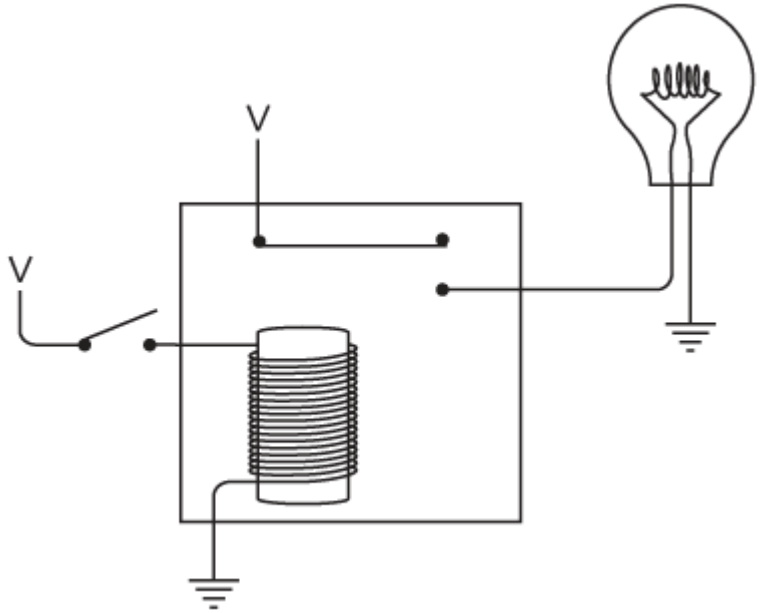
The basic unit: repeater

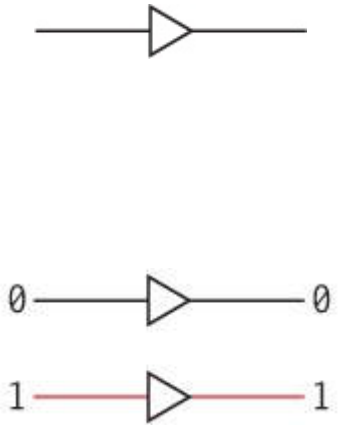


Basic logic unit: NOT

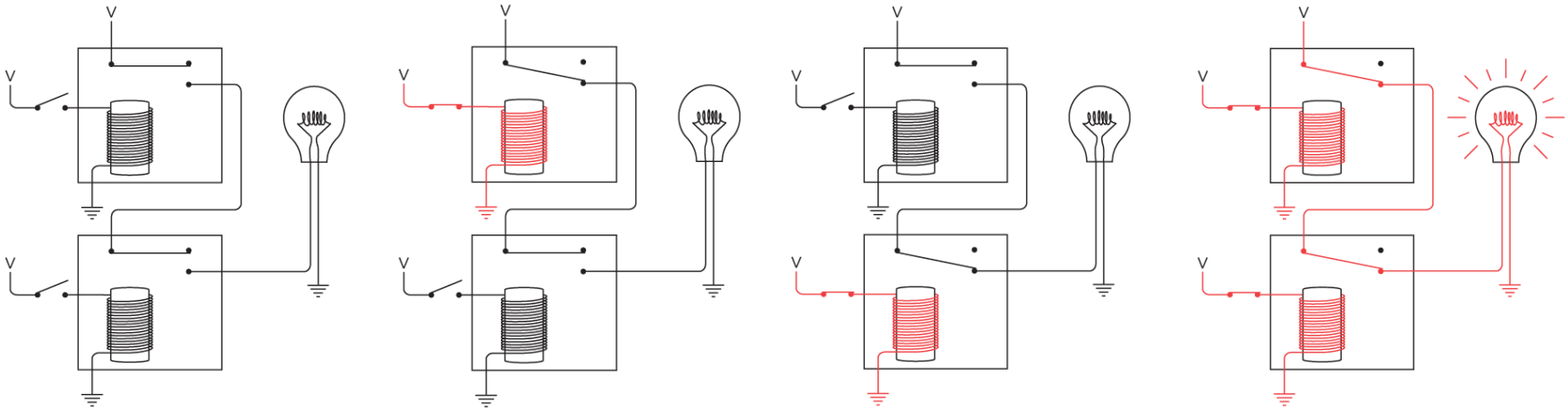


Basic logic unit: buffer

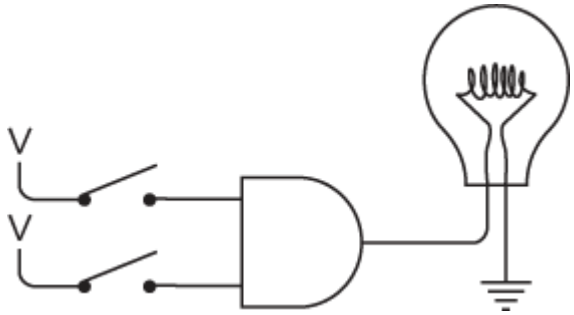


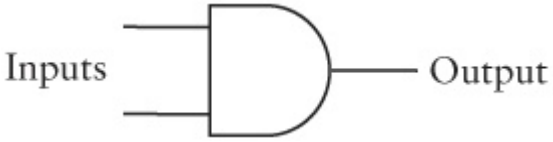


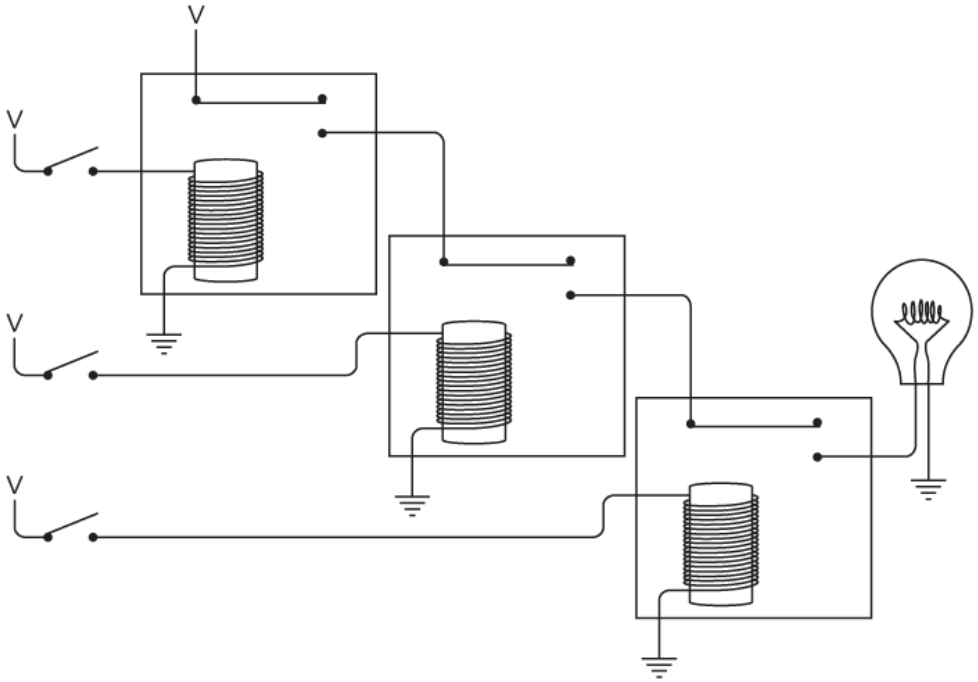
Basic logic unit: AND

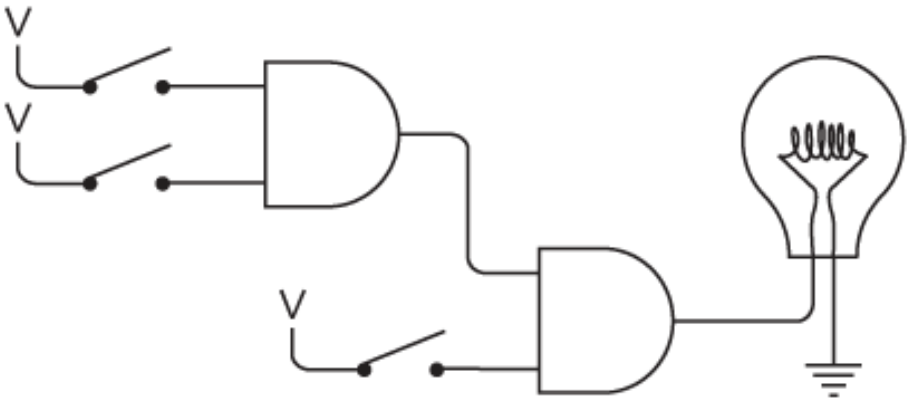


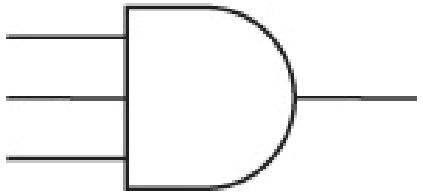
AND	0	1
0	0	0
1	0	1



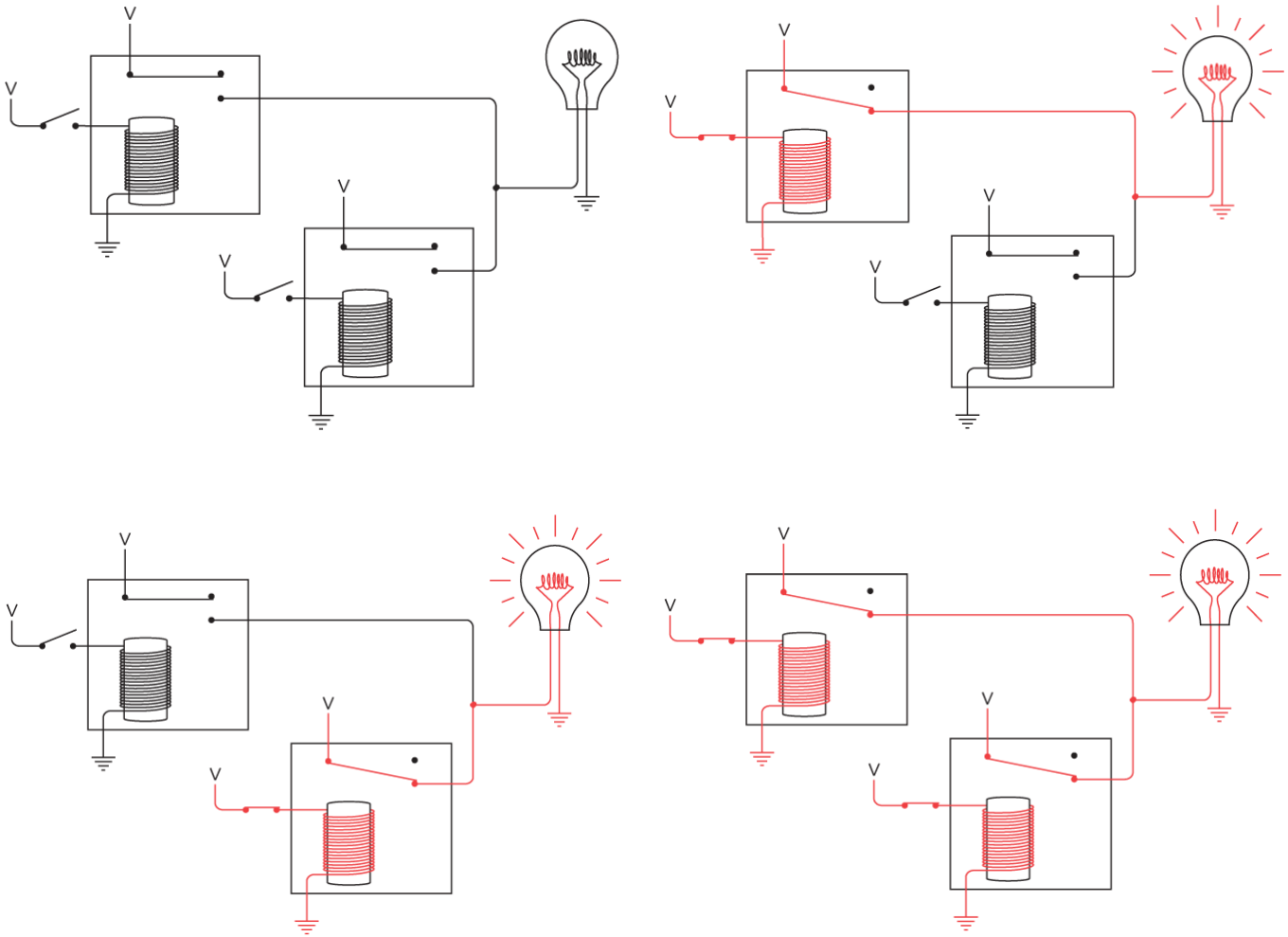



V
V
V
V

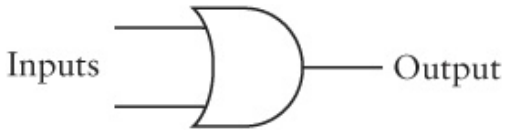

V
V
V



Basic logic unit: OR



OR	0	1
0	0	1
1	1	1



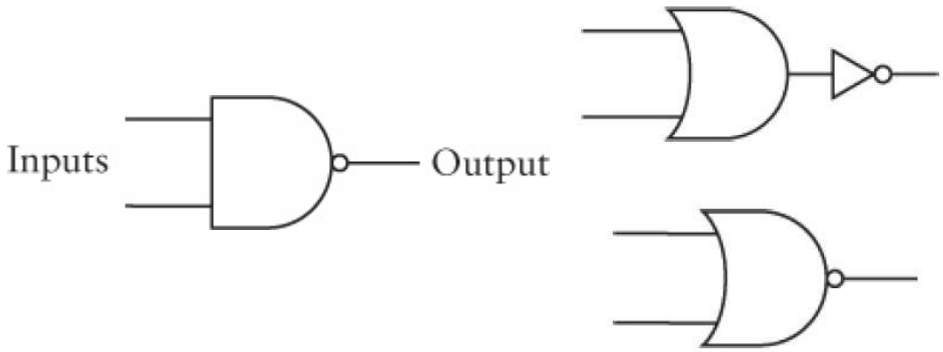
Basic logic unit: NAND and NOR

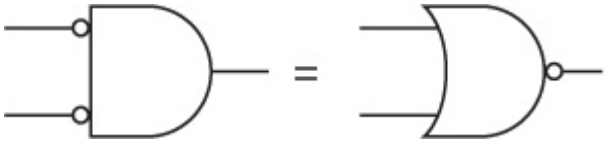
AND	0	1
0	0	0
1	0	1

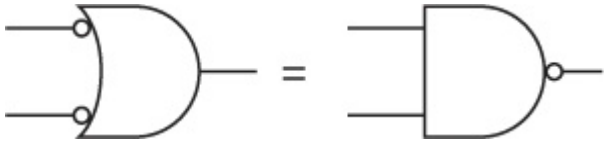
OR	0	1
0	0	1
1	1	1

NAND	0	1
0	1	1
1	1	0

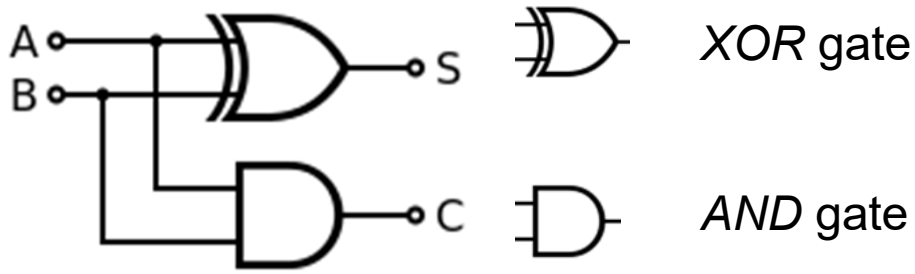
NOR	0	1
0	1	0
1	0	0







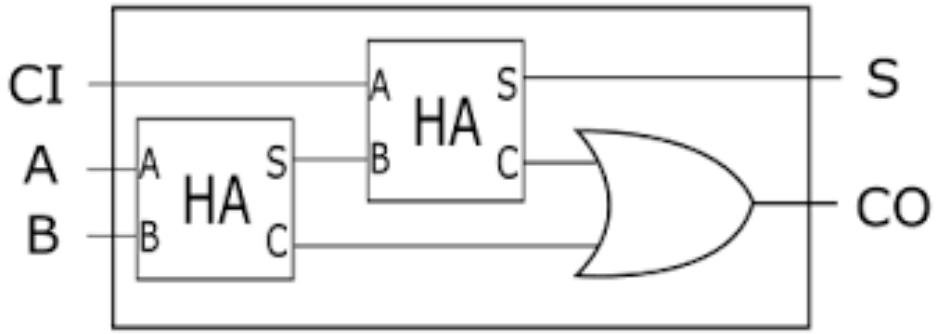
1-bit Half Adder and Full Adder



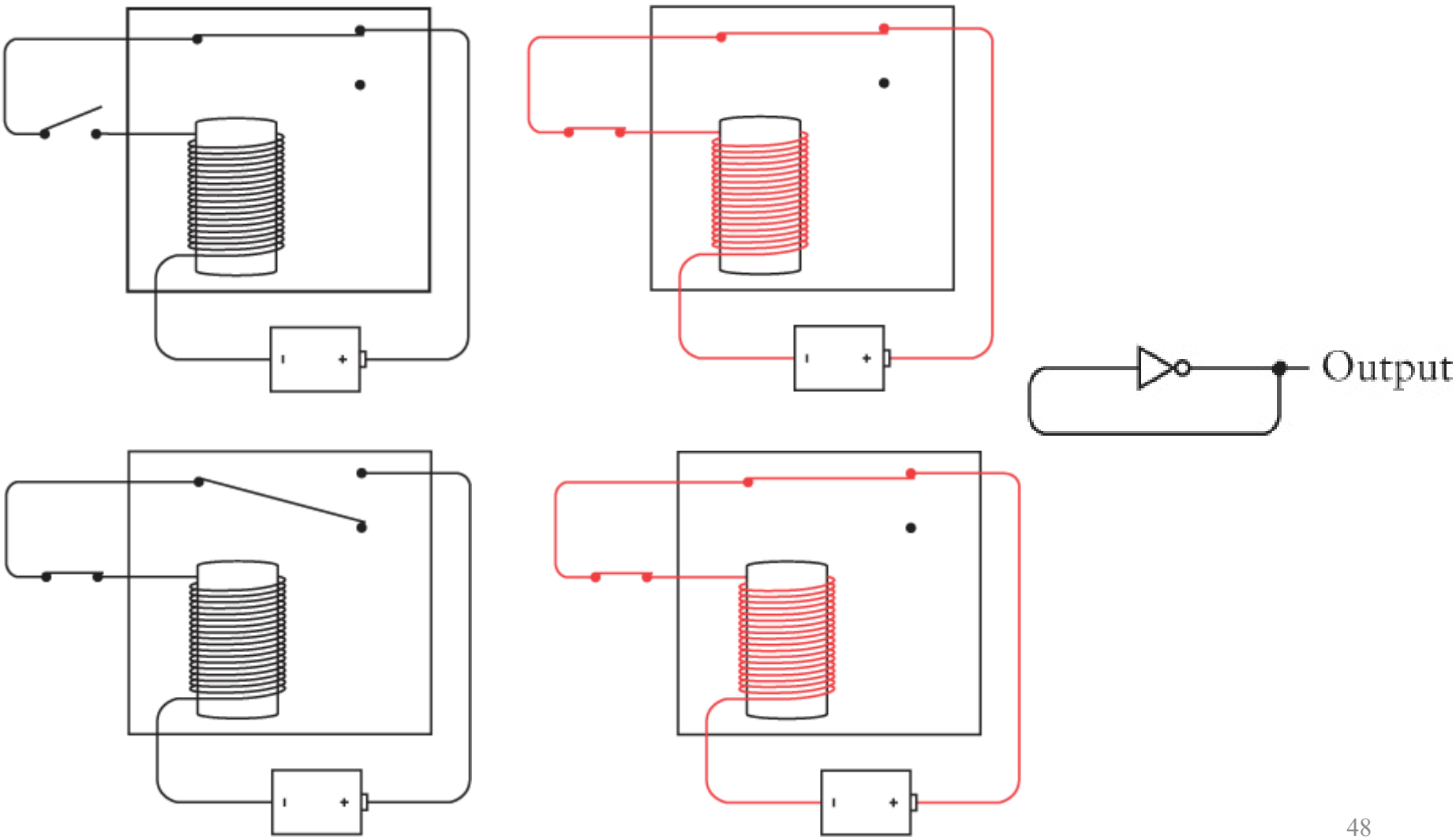
Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

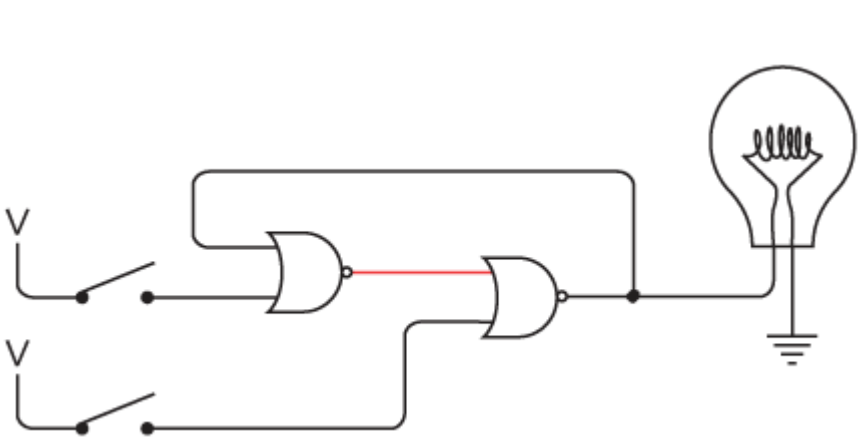
- A: input
- B: input
- S: output
- C: output

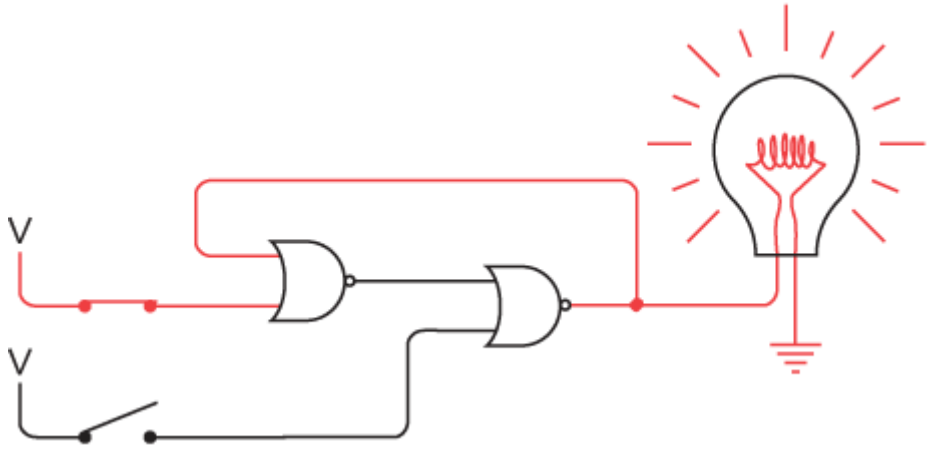
- Using two half adders to build a full adder.

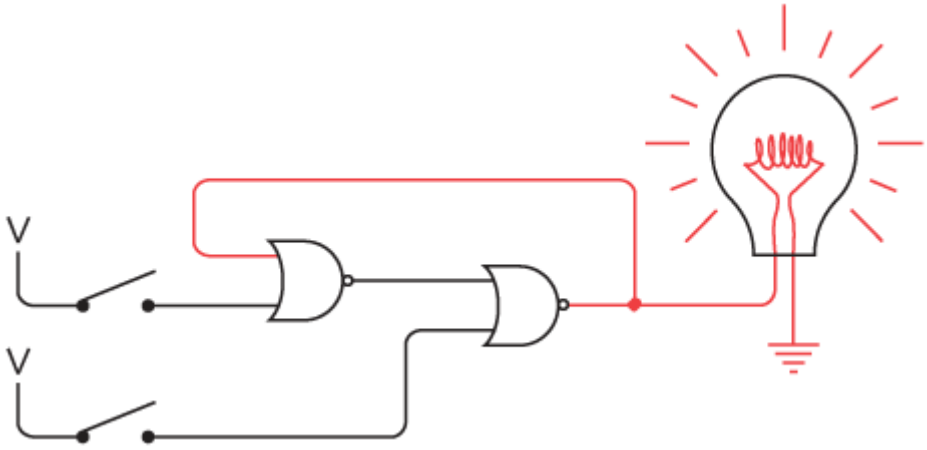


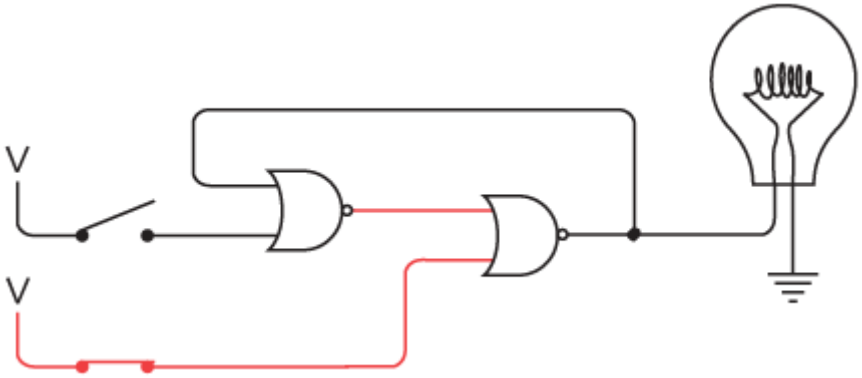
Feedback and Flip-Flops




V
V

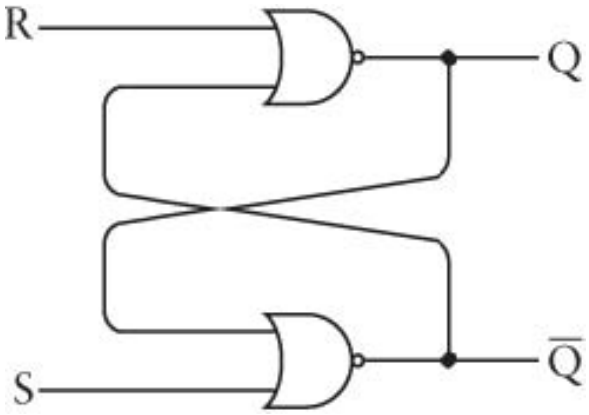

V
V

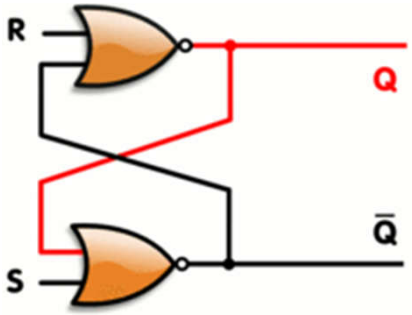

V
V

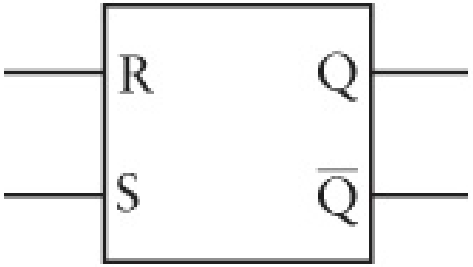

V
V

- Truth Table

Inputs		Outputs	
S	R	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	Q	$\overline{Q}$
1	1	Disallowd	

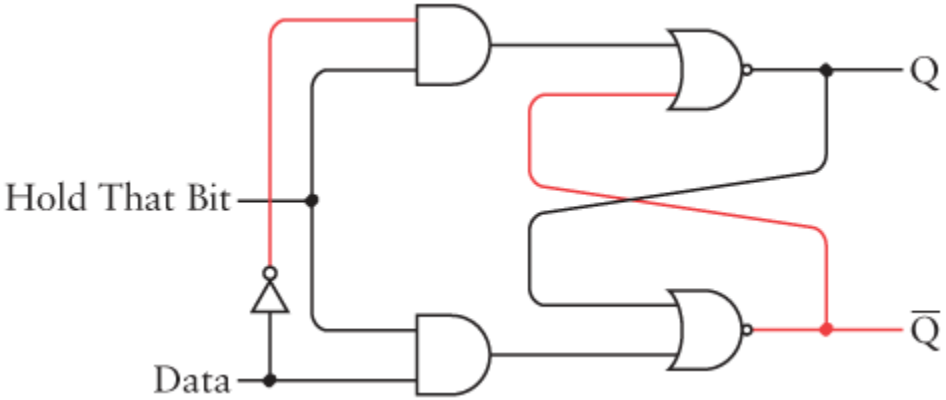


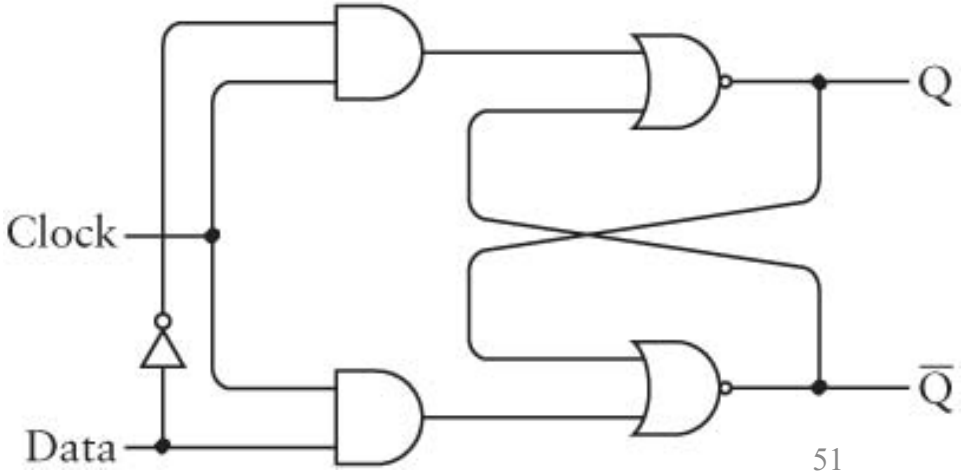




Inputs		Outputs
Data	Hold That Bit	Q
0	1	0
1	1	1
0	0	Q
1	0	Q

Inputs		Outputs	
D	Clk	Q	Q-bar
0	1	0	1
1	1	1	0
X	0	Q	Q-bar

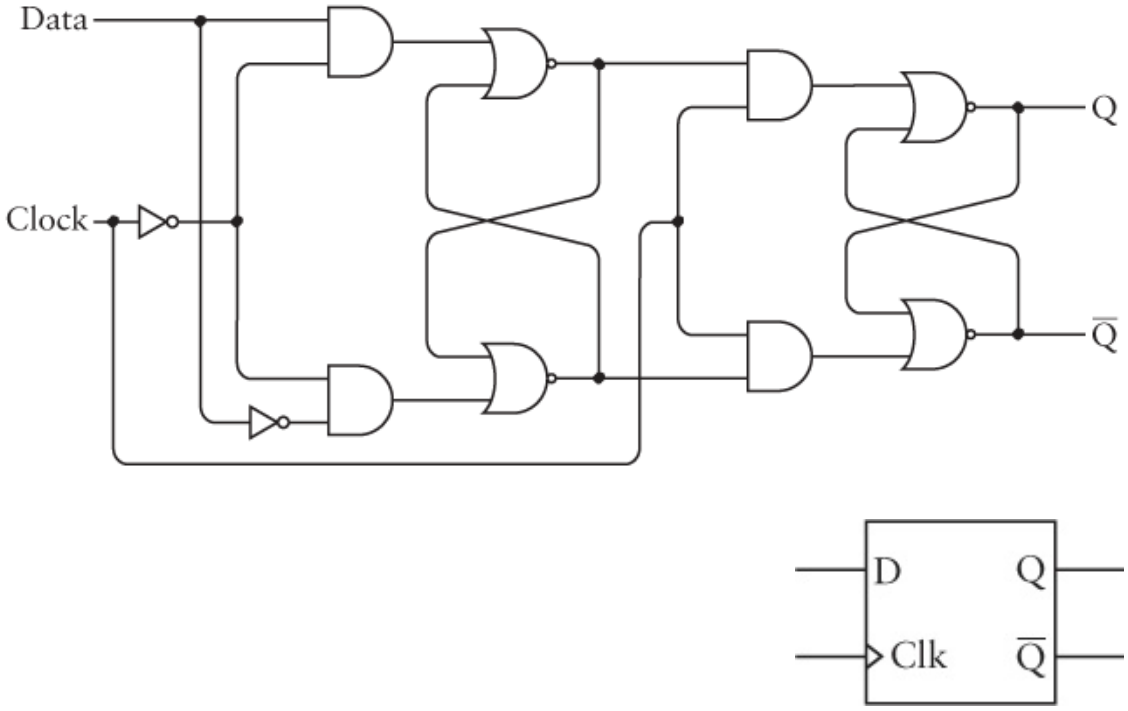




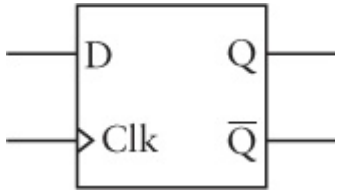
An edge-triggered D-type flip-flop

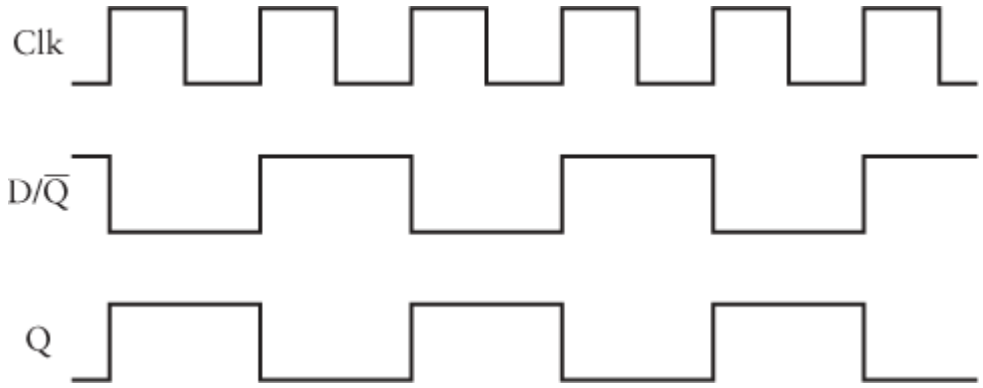
- The idea here is that the Clock input controls both the first stage and the second stage. But notice that the clock is inverted in the first stage. This means that the first stage works exactly like a D-type flip-flop except that the Data input is stored when the Clock is 0. The outputs of the second stage are inputs to the first stage, and these are saved when the Clock is 1. The overall result is that the Data input is saved when the Clock changes from 0 to 1.

Inputs		Outputs	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1
1	1	0	1



An edge-triggered D-type flip-flop



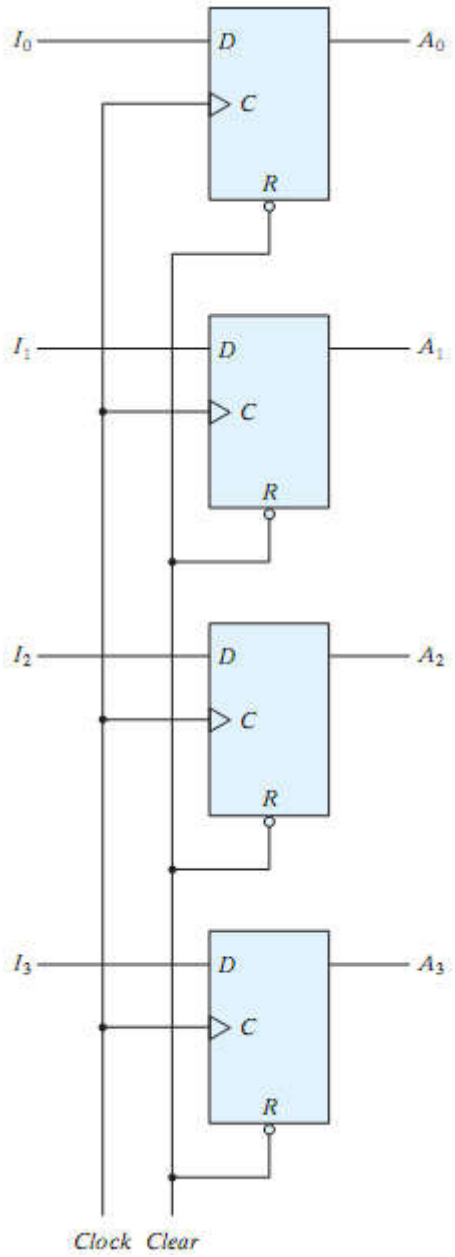


Registers

- A register is a group of flip-flops.
- An n-bit register is made of n flip-flops and can store n bits.
- A register may have additional combinational gates to perform certain operations.

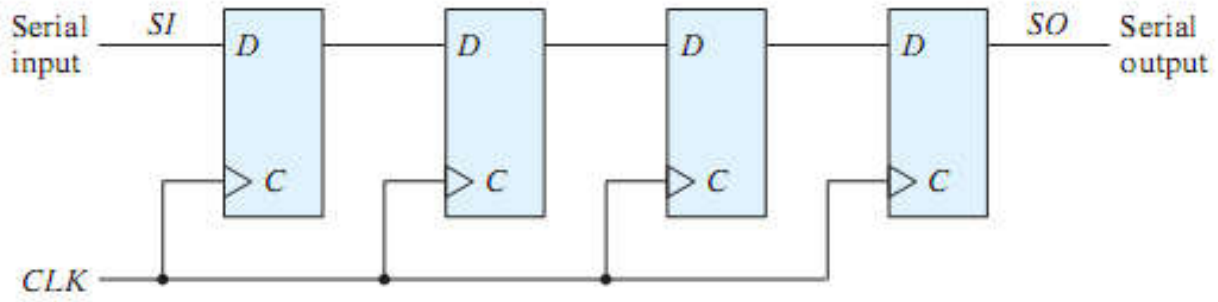
4-Bit Register

- A simple 4-bit register can be made with 4 D-type flip-flops.
- They have a Common Clock.
  - At each positive-edge, 4 bits are loaded in parallel.
  - Previous data is overwritten.
- Common Clear
  - Asynchronous clear
  - When clear = 0, all flip-flops are cleared. i.e. 0 is stored.



4-bit Shift Register

- Serial-in and Serial-out (SISO)



- A simple 4-bit shift register can be made with 4 D-type flip-flops.
- They have a Common Clock.
  - At each positive-edge, 1 bit is shifted in
  - Rightmost bit is discarded
- Which direction is this register shifting?

Multiply (unsigned)

- Paper and pencil example (unsigned):

- Multiplicand 1000
  Multiplier 1001

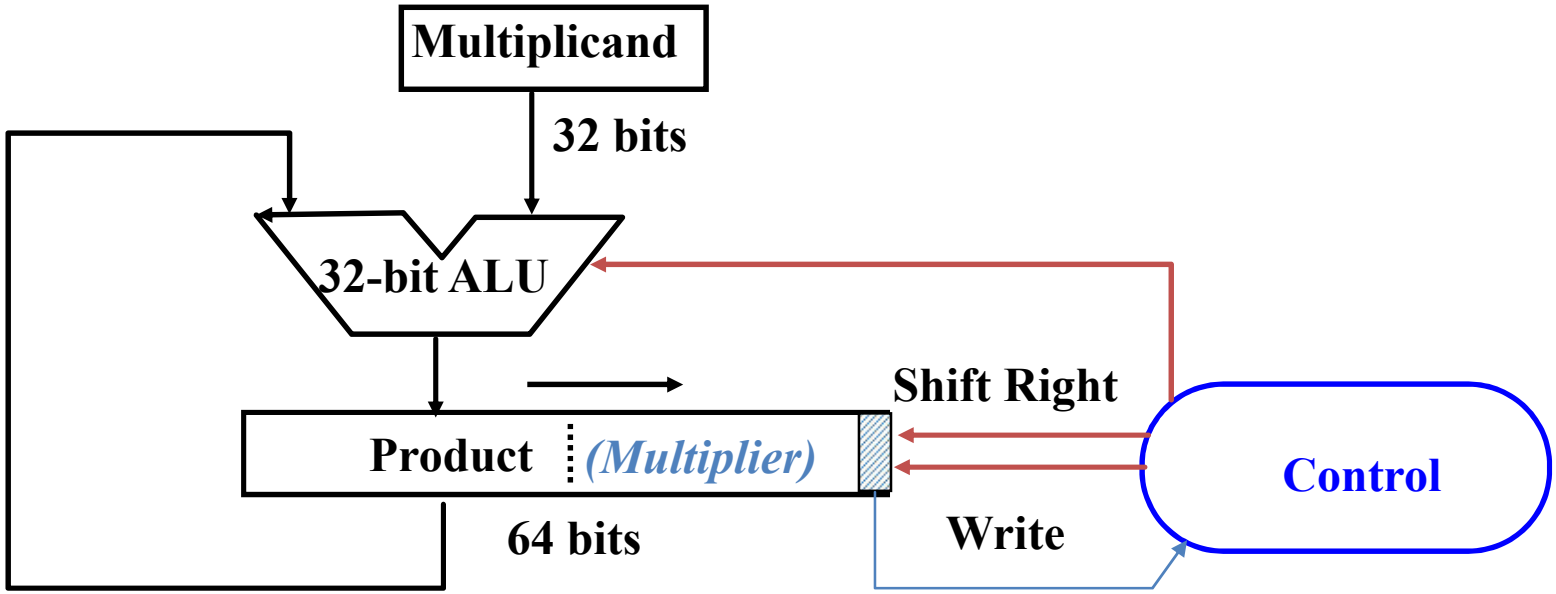
```
      1000
      0000
     0000
    1000
   01001000
```

  Product

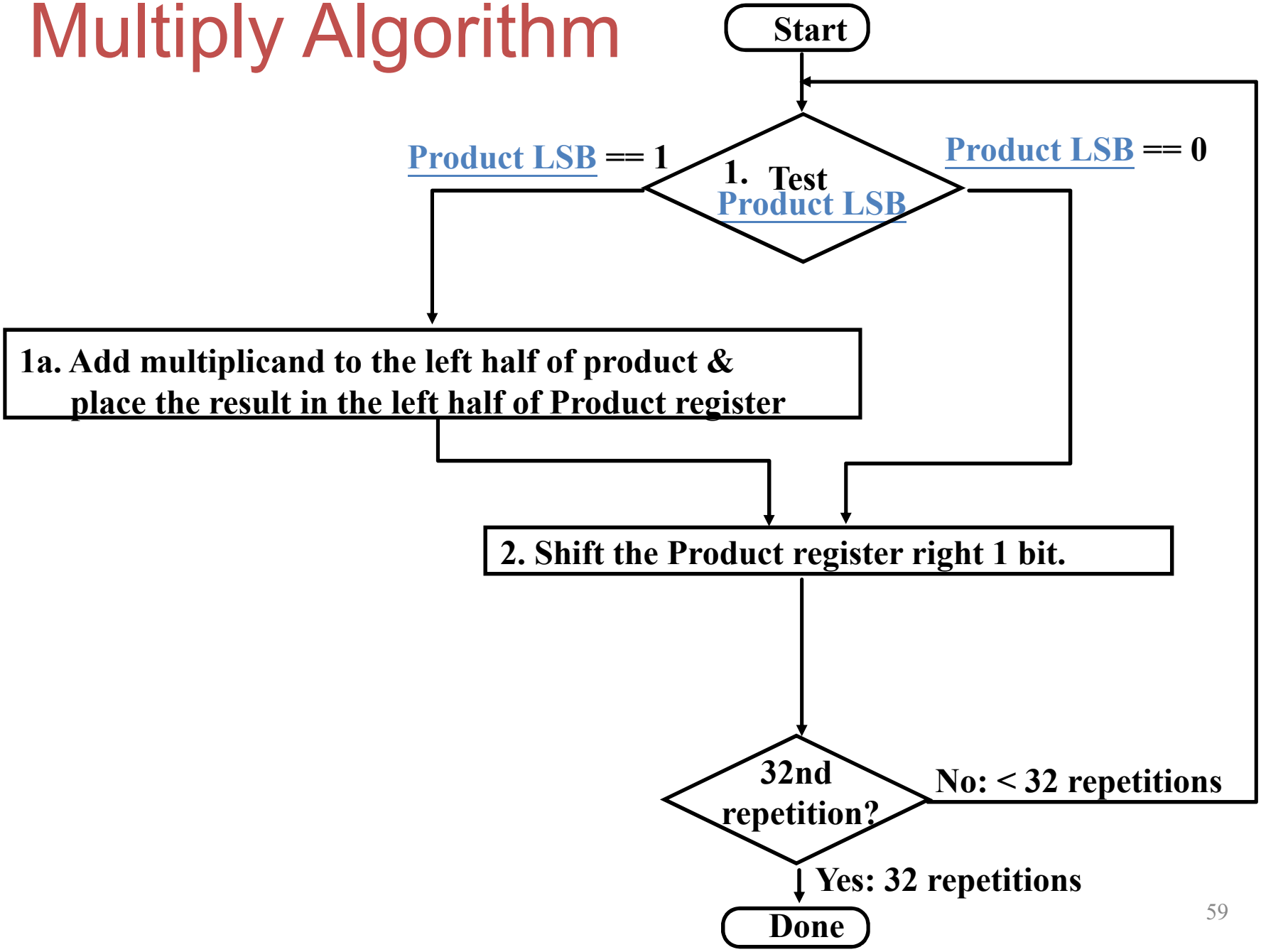
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)

Multiply Hardware

- 32-bit Multiplicand register, 32-bit ALU, 64-bit Product register, (Multiplier register is encapsulated in product register).



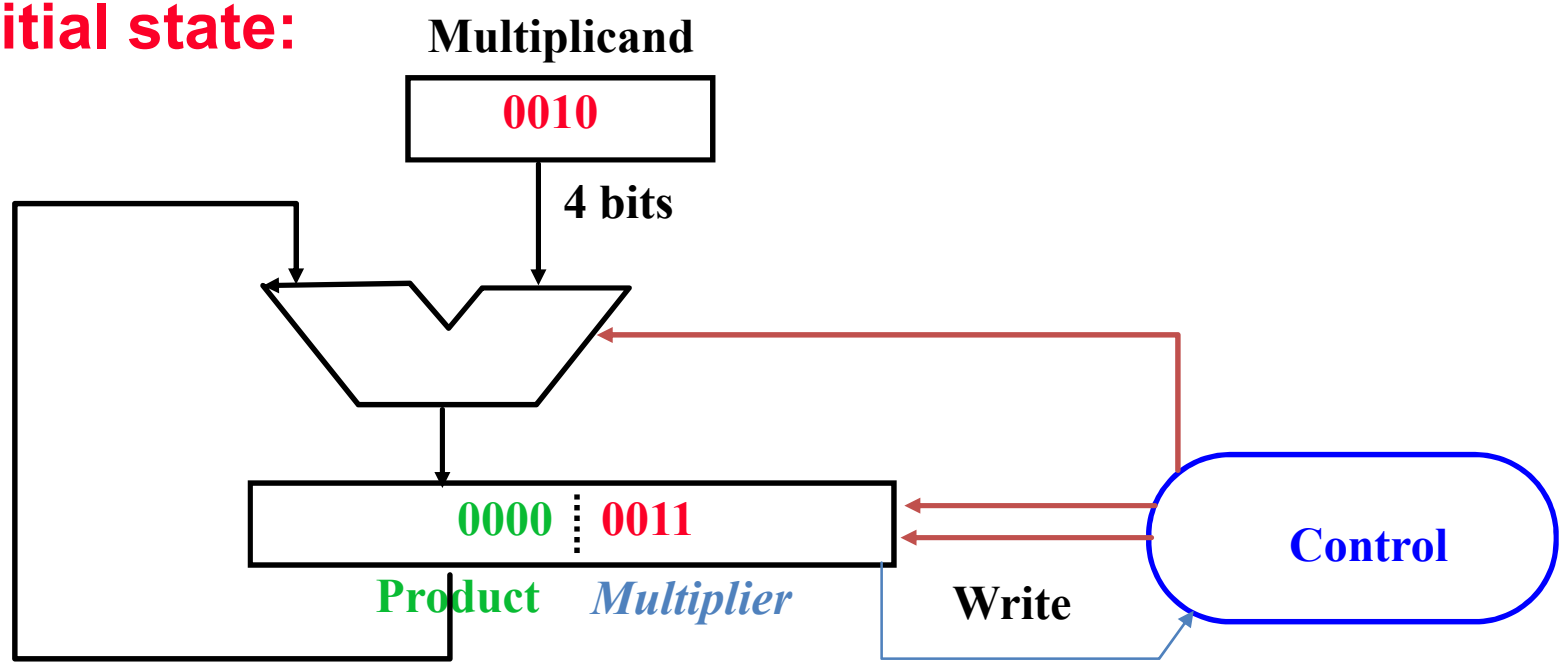
Multiply Algorithm



Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

Initial state:



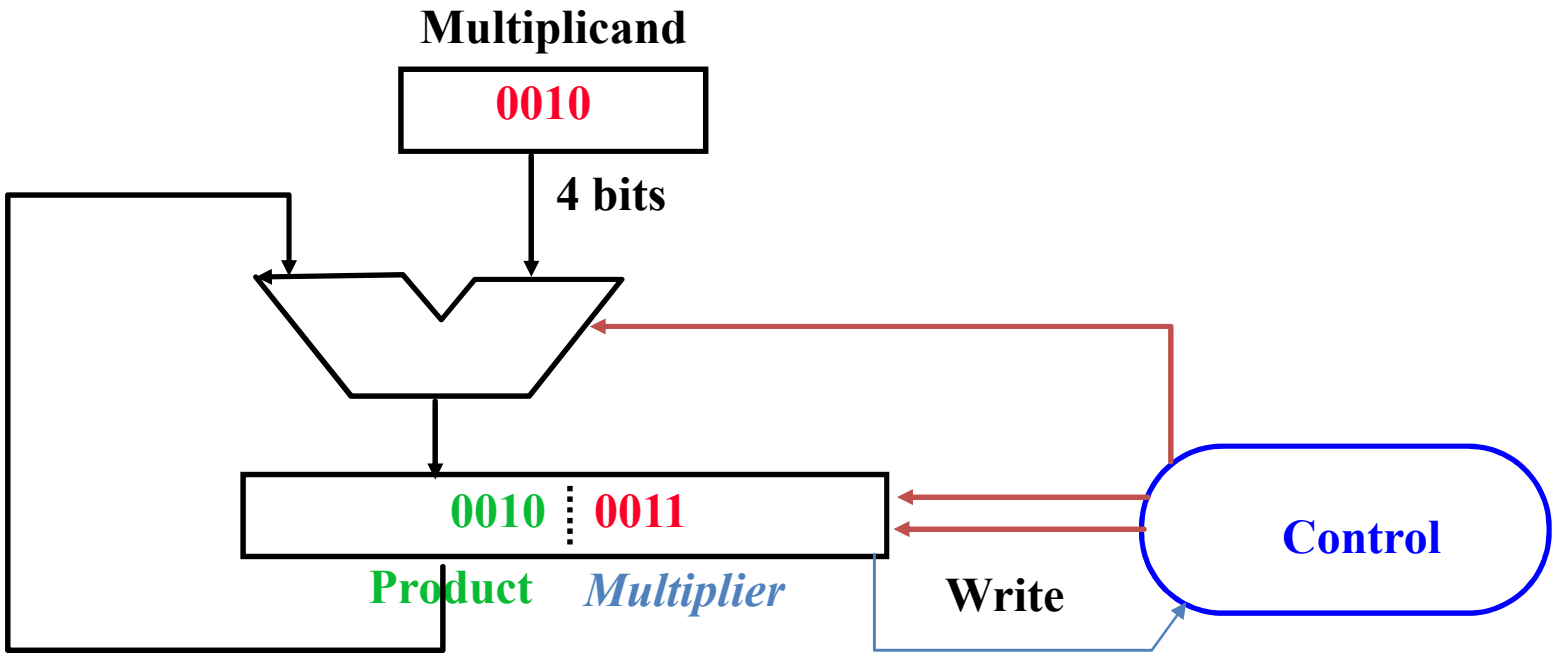
Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

1st repetition:

Step 1: Product LSB == 1

Step 1a: Add multiplicand to the left half of product

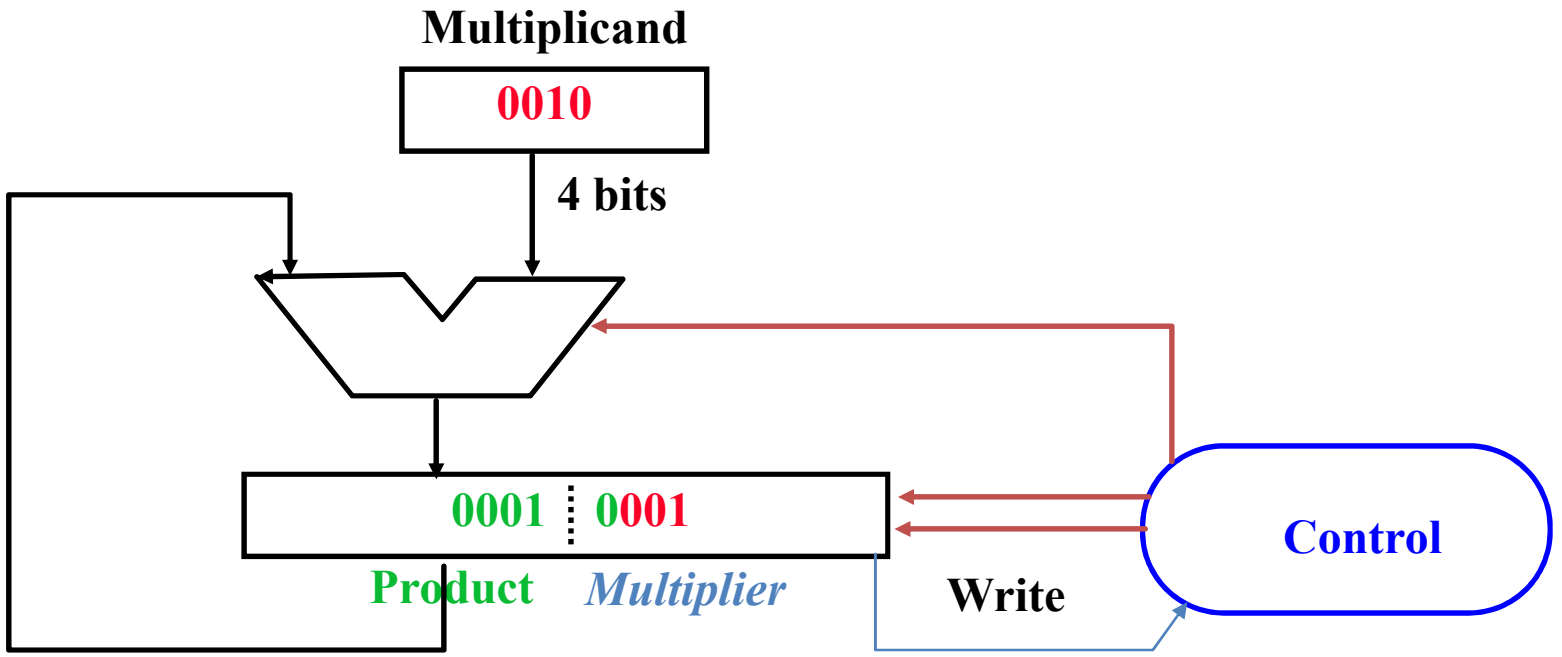


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

1st repetition:

Step 2: Shift product register right 1 bit



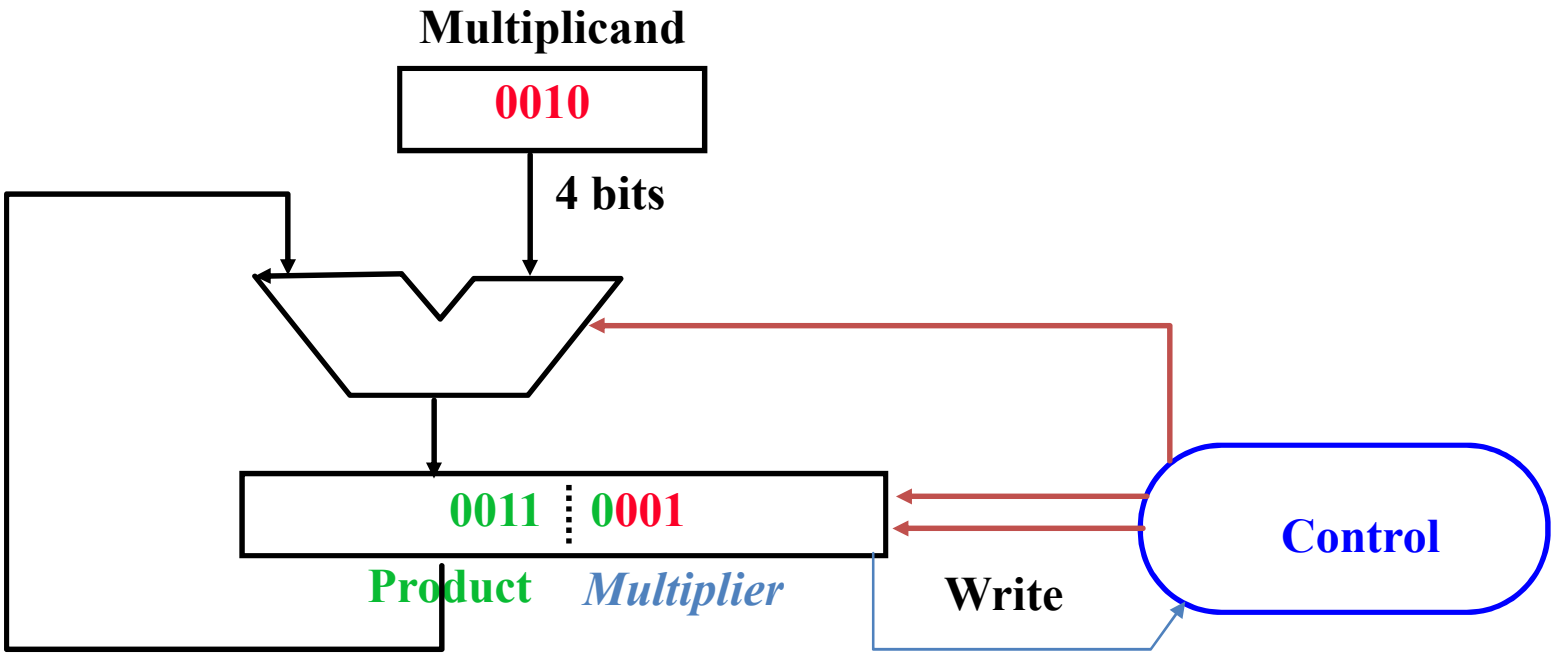
Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

2nd repetition:

Step 1: Product LSB == 1

Step 1a: Add multiplicand to the left half of product

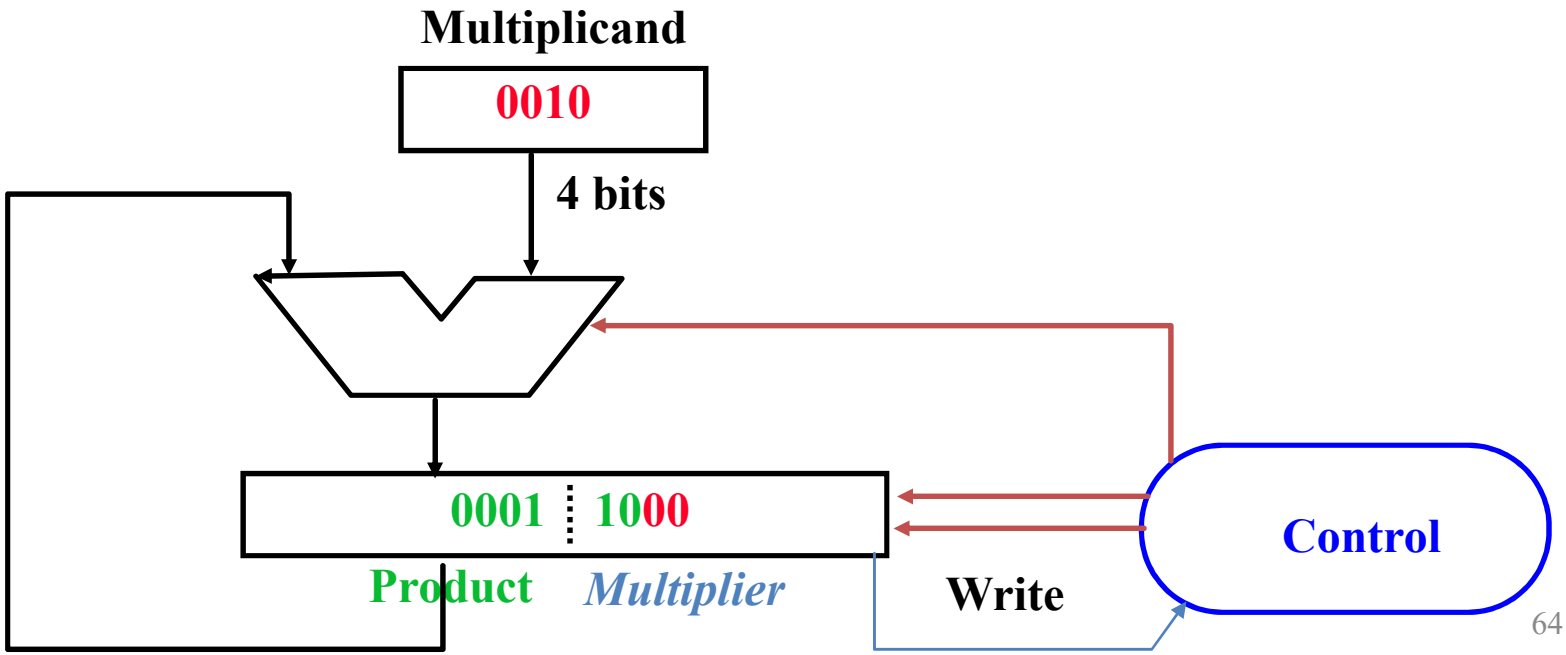


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

2nd repetition:

Step 2: Shift product register right 1 bit

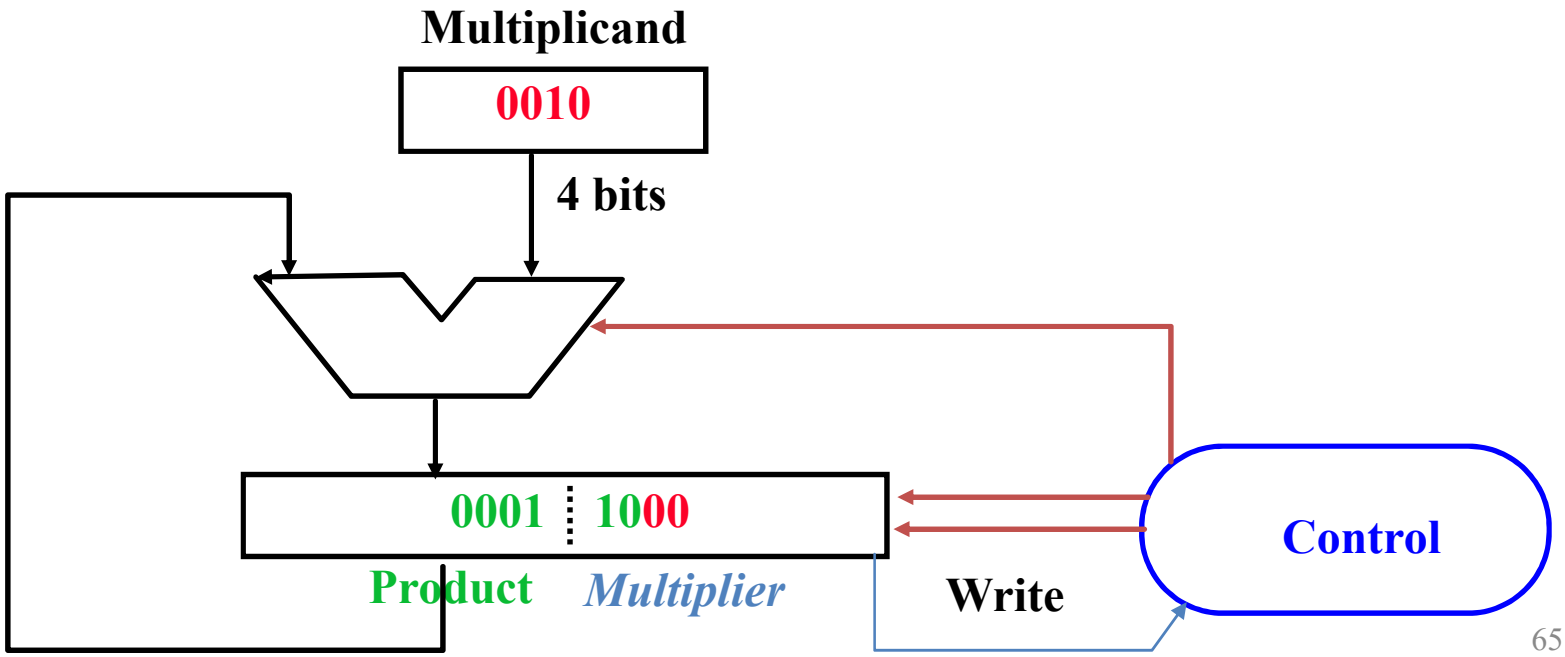


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

3rd repetition:

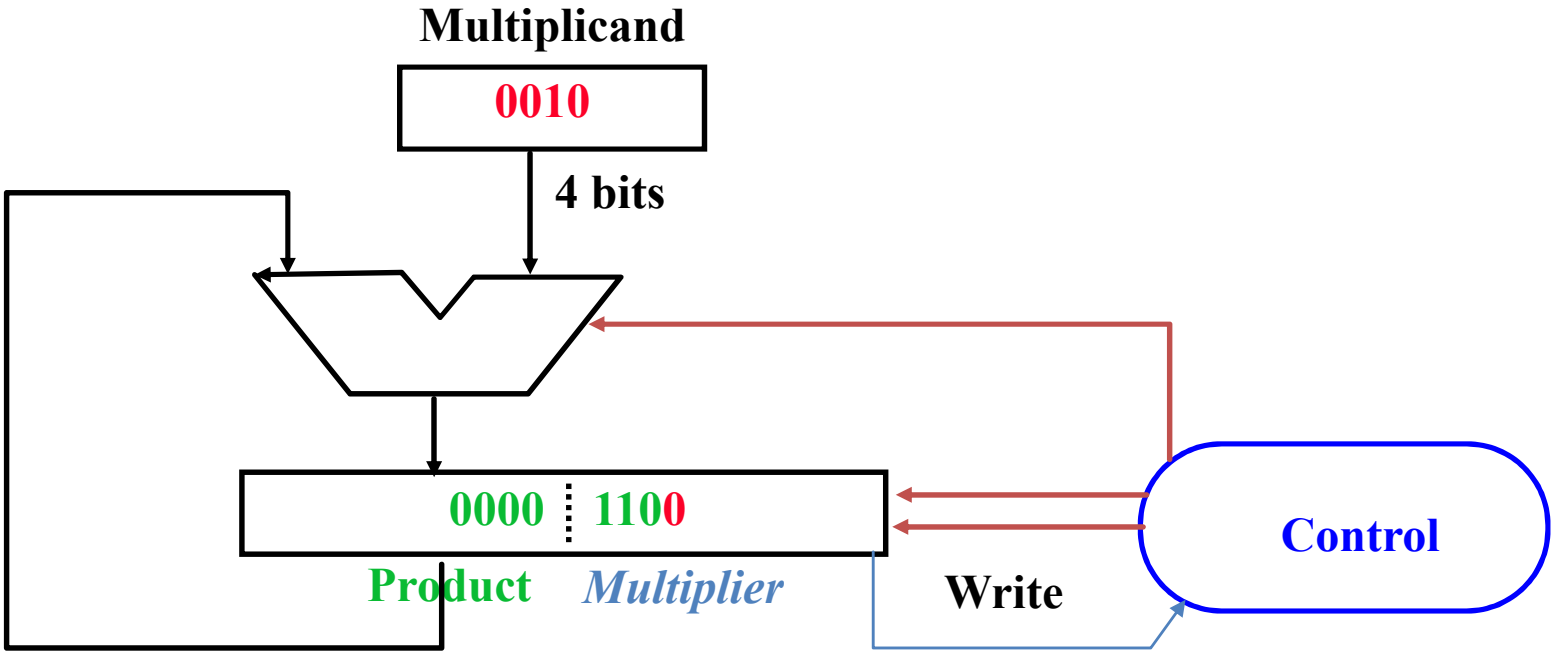
Step 1: Product LSB == 0
No addition is required.



Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

3rd repetition: **Step 2: Shift product register right 1 bit**

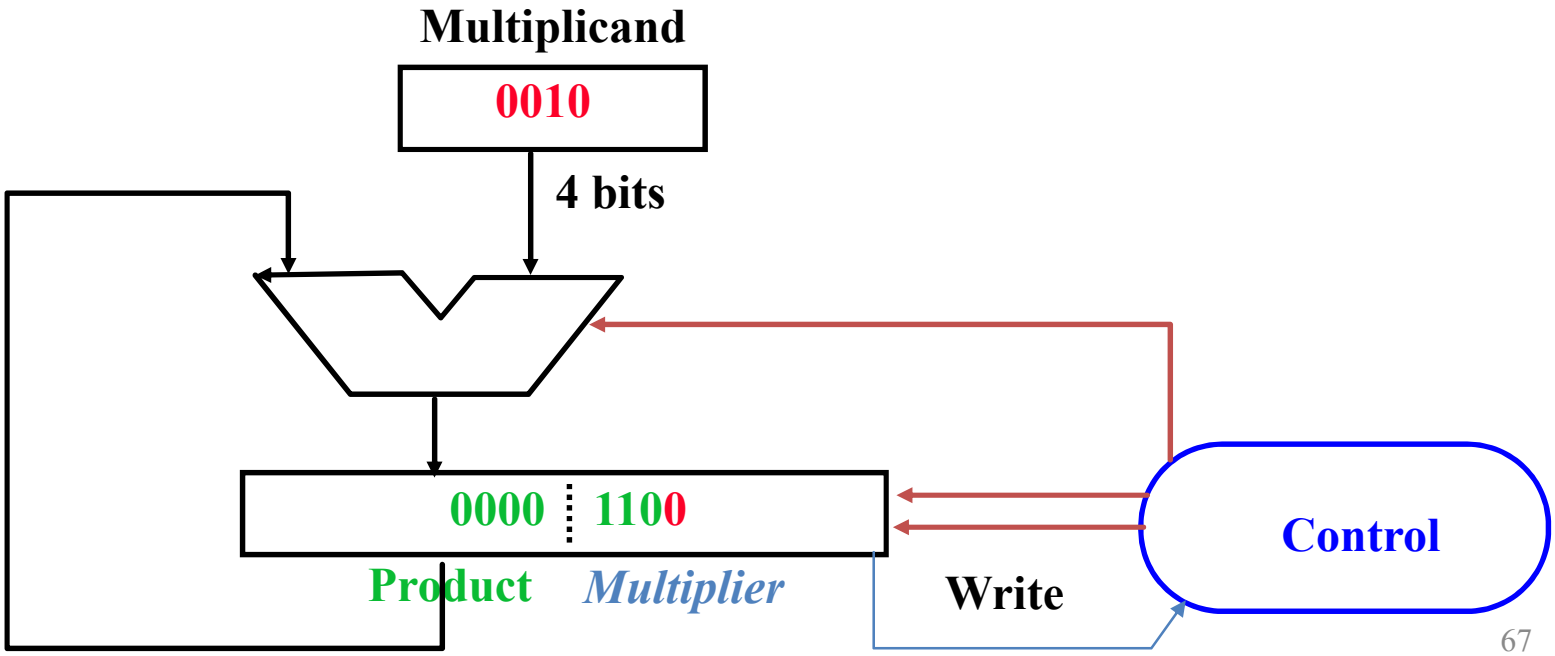


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

4th repetition:

Step 1: Product LSB == 0
No addition is required.



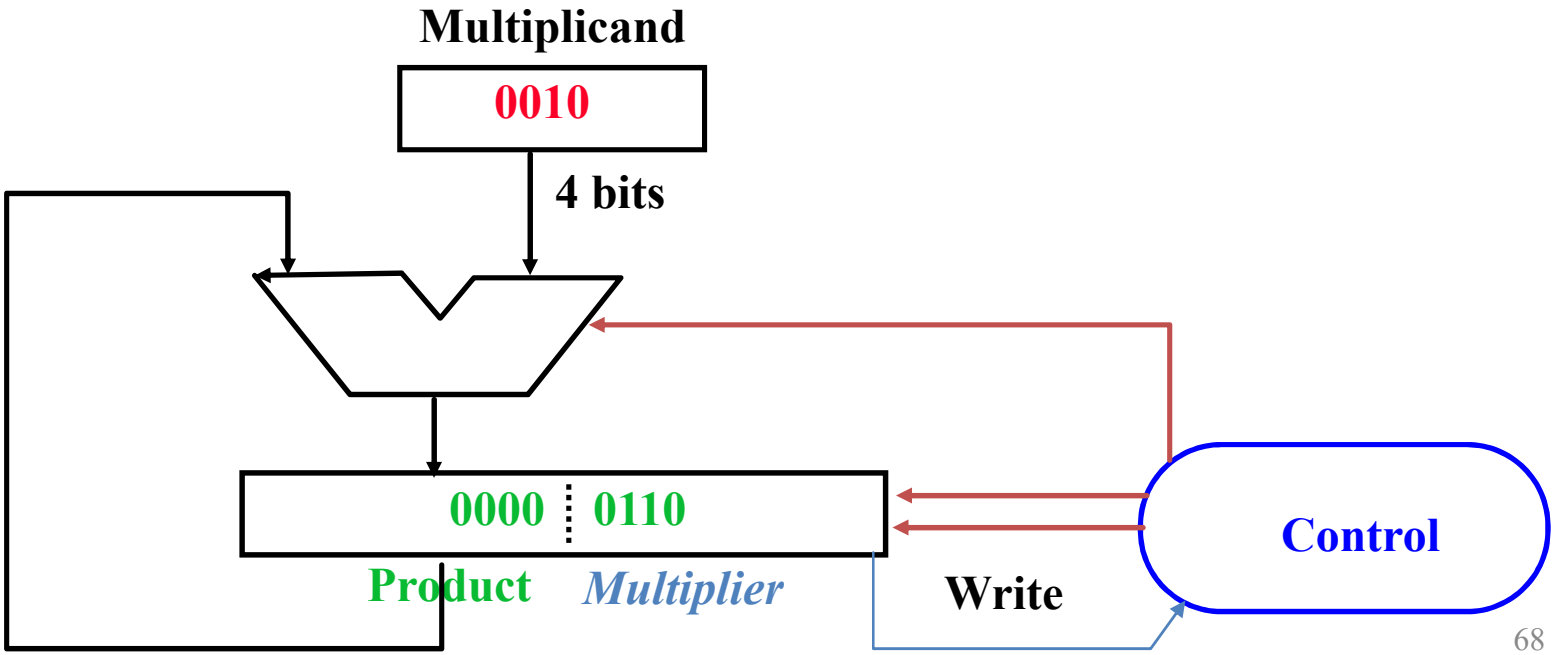
Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = 0011, multiplicand = 0010.

4th repetition:

Step 2: Shift product register right 1 bit

4th repetition, STOP

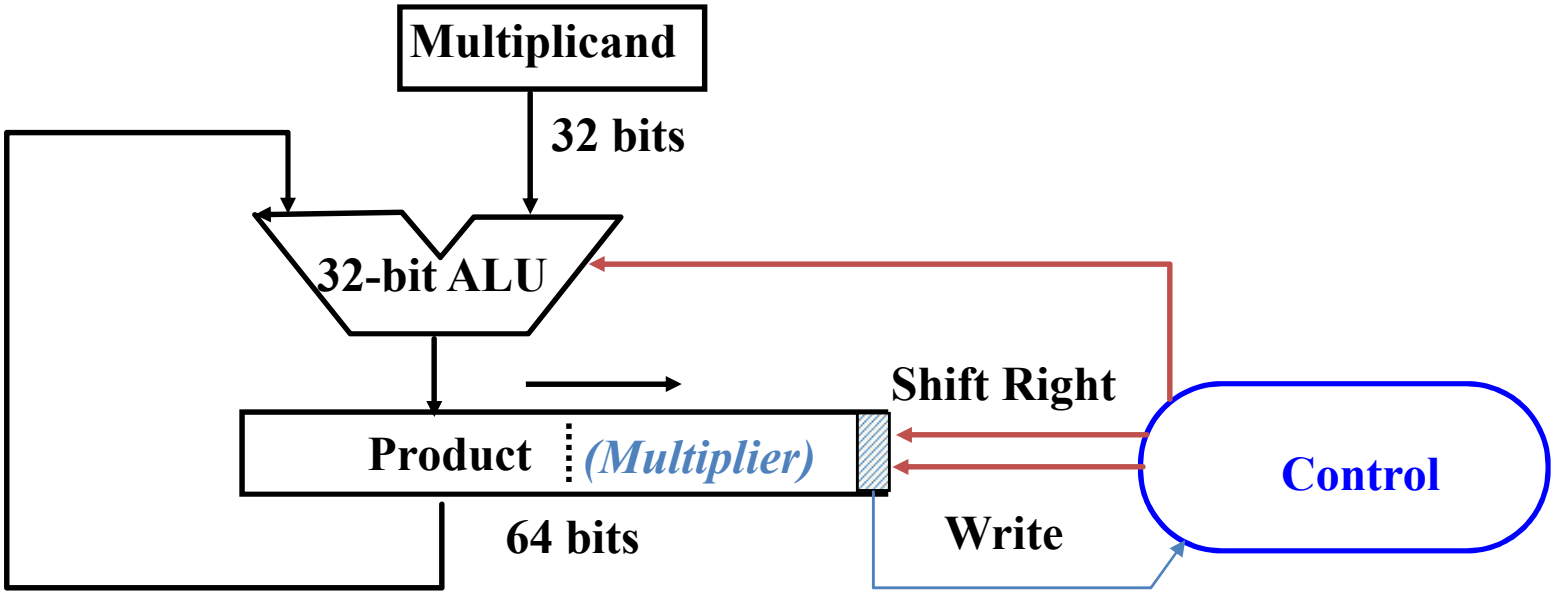


Observations on Multiply Version 3

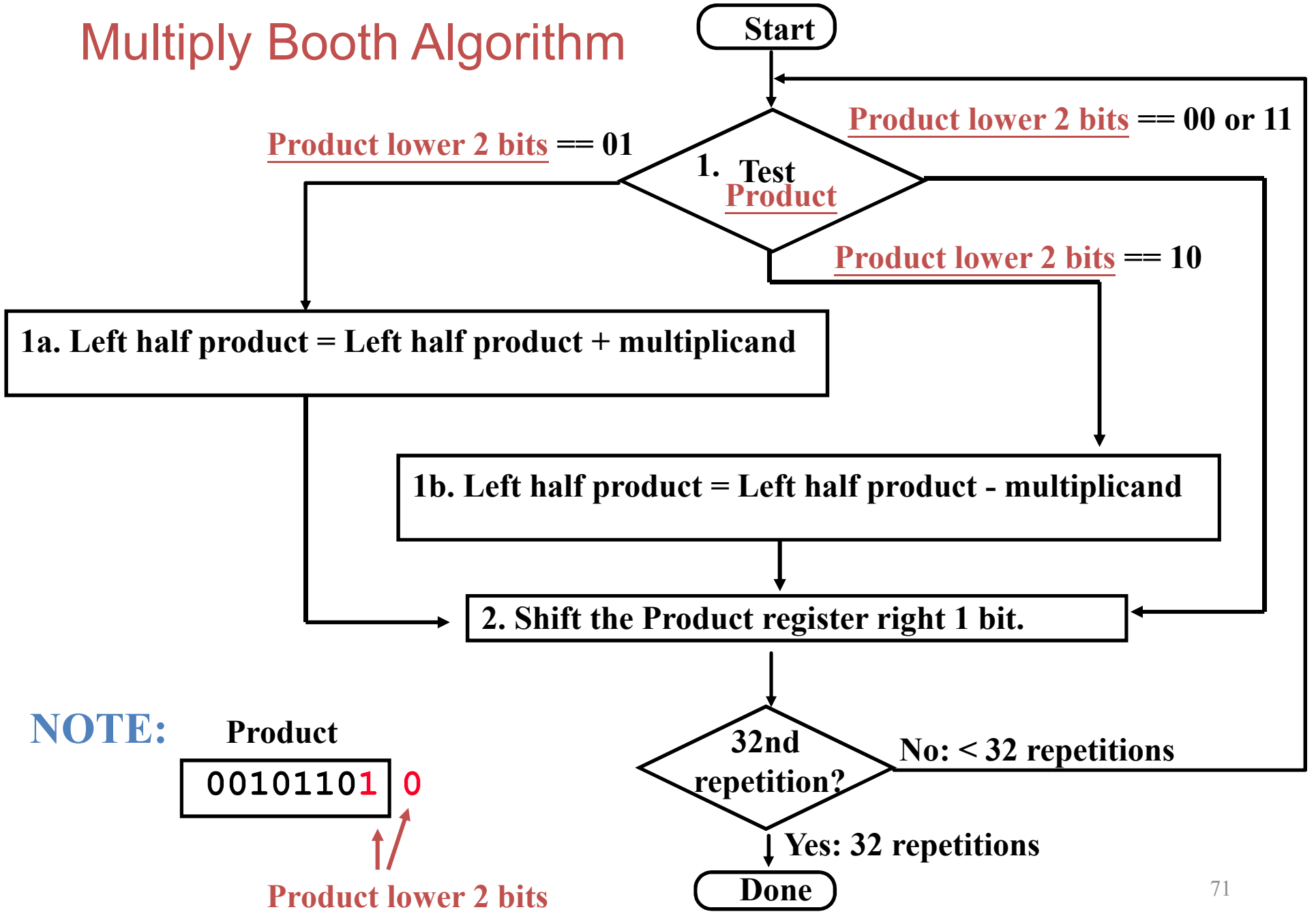
- Both Multiplier & Product are shifted at the same time
  - As they are combined in the same register
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to inverse product when done (leave out the sign bit, run for 31 steps)
  - apply definition of 2's complement
    - need to sign-extend partial products and subtract at the end
  - Booth's Algorithm is elegant way to multiply signed numbers using same hardware as before

Multiply Hardware for Booth Algorithm

- Same as version 3 before
- 32-bit Multiplicand register, 32-bit ALU, 64-bit Product register, (Multiplier register is encapsulated in product register)



Multiply Booth Algorithm



NOTE:

Product

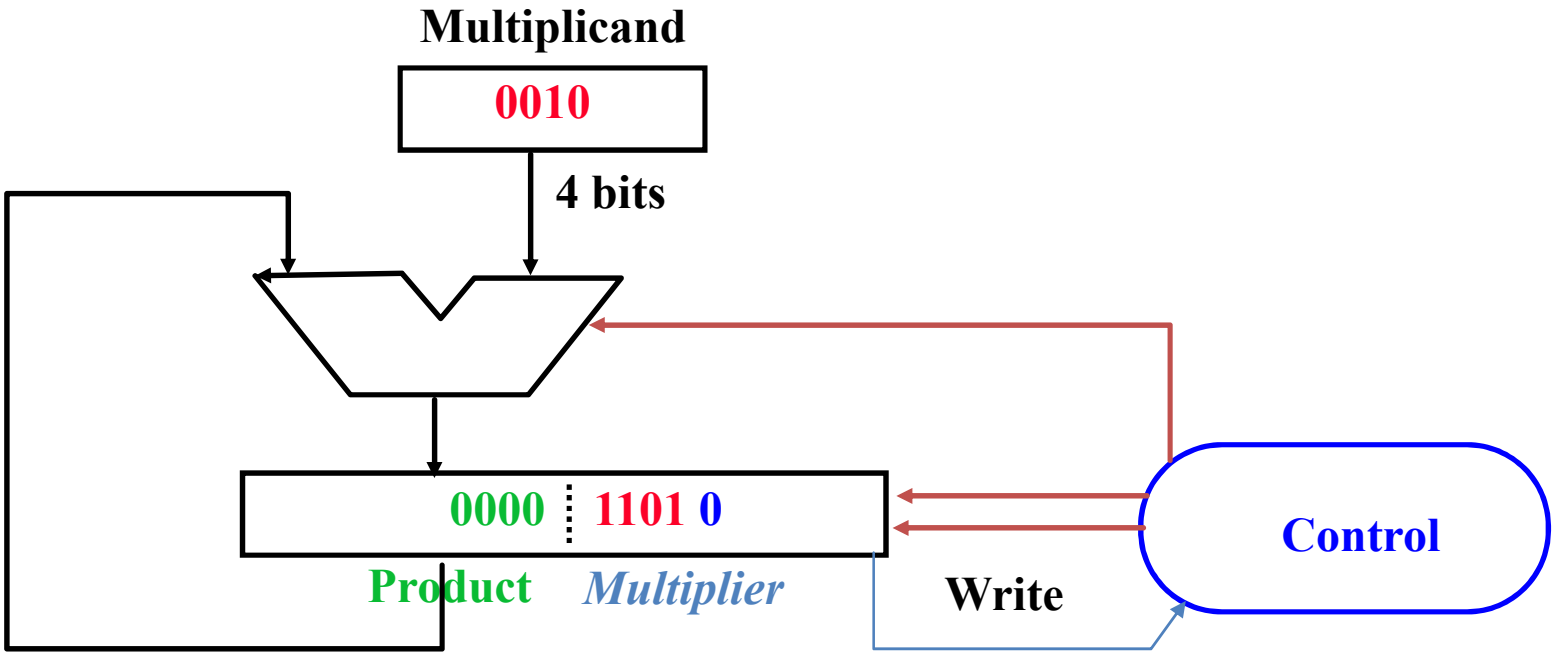
00101101 0

Product lower 2 bits

Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = –3 = 1101, multiplicand = 0010.

Initial state:



Example

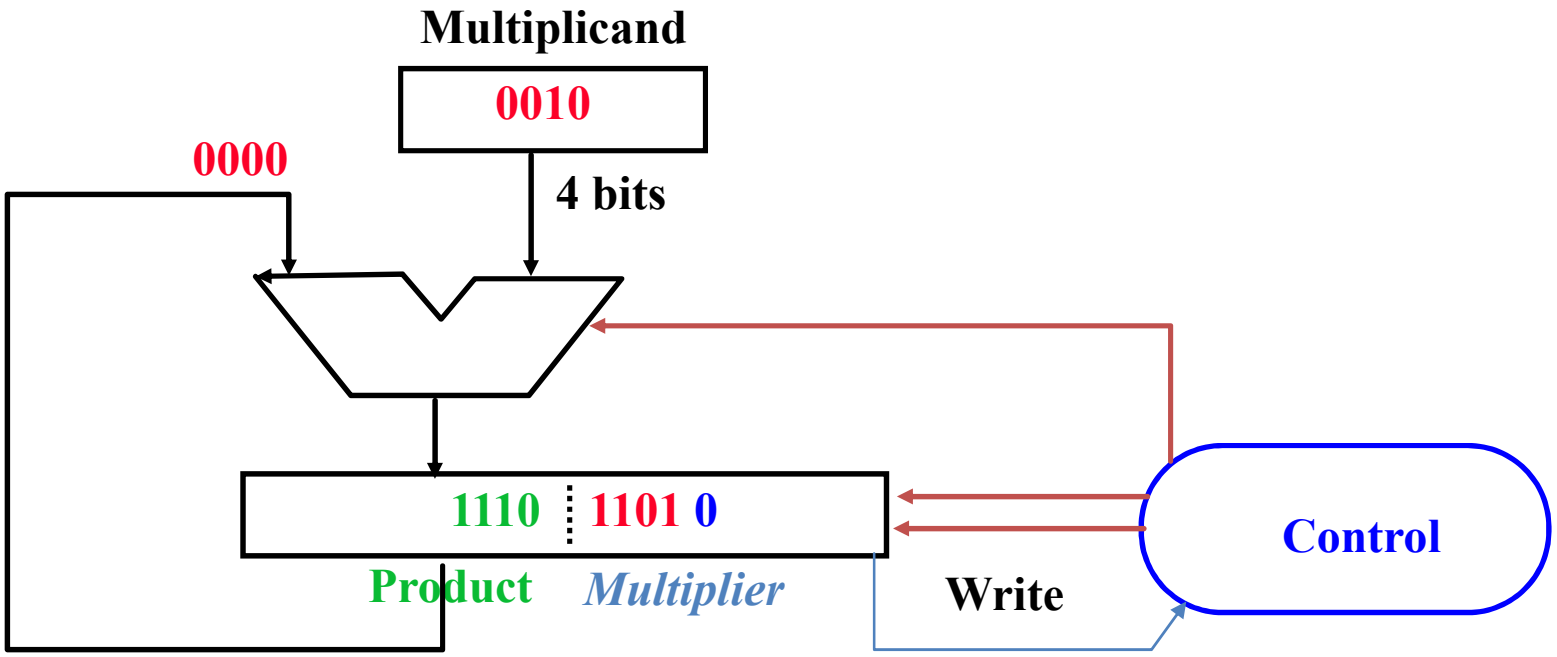
- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = −3 = 1101, multiplicand = 0010.

1st repetition:

Step 1: Product lower 2 bits == 10

Step 1b: Product left half = Product left half – multiplicand

0000 – 0010 = 0000 + 1110 = 1110

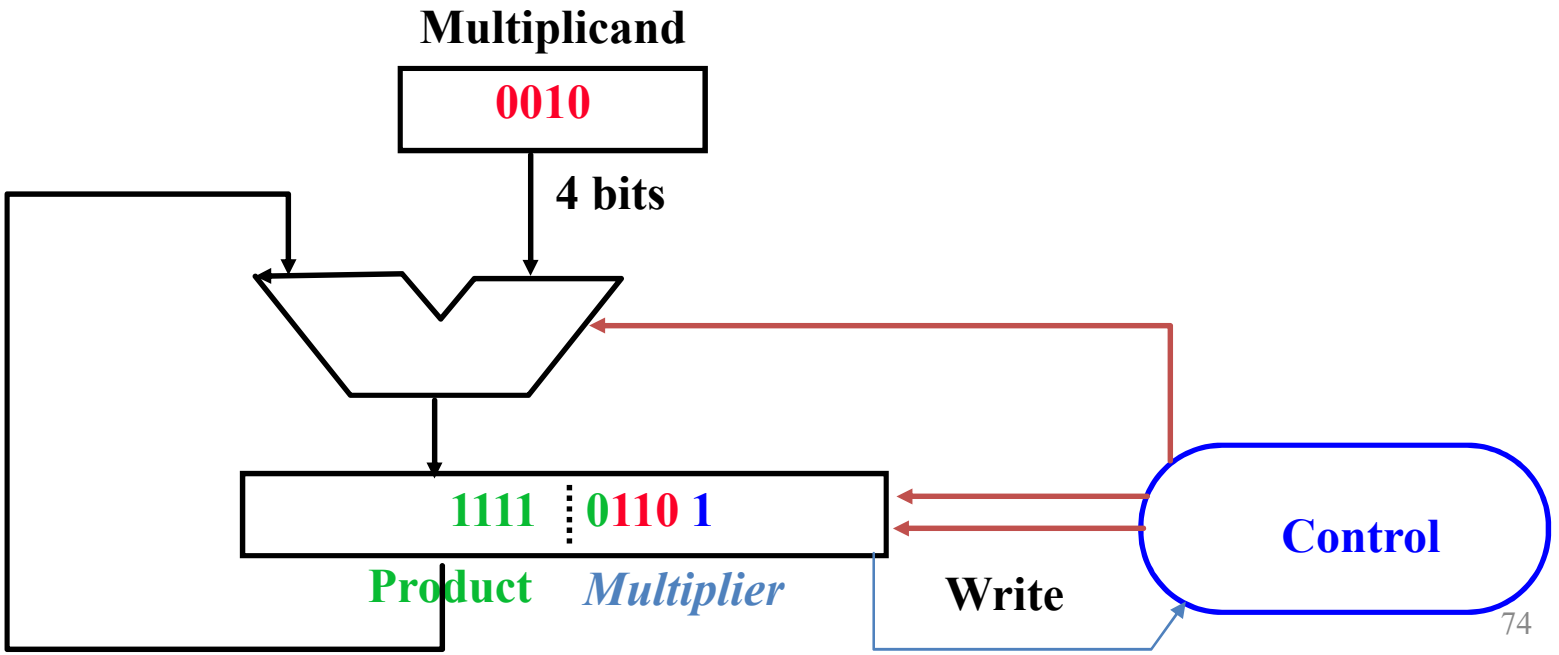


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = −3 = 1101, multiplicand = 0010.

1st repetition:

Step 2: Shift product register right 1 bit (Remember sign extend)



Example

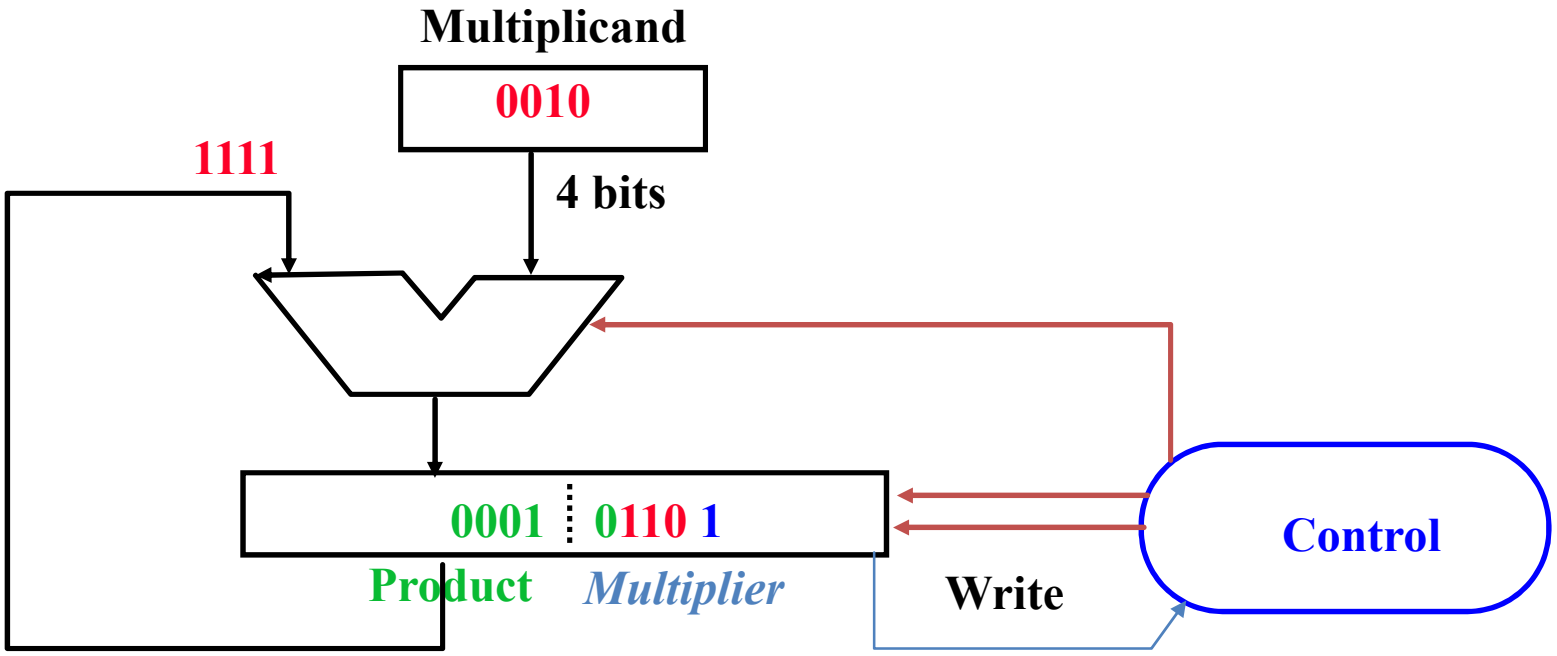
- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = −3 = 1101, multiplicand = 0010.

2nd repetition:

Step 1: Product lower 2 bits == 01

Step 1a: Product left half = Product left half + multiplicand

0010 + 1111 = 0001

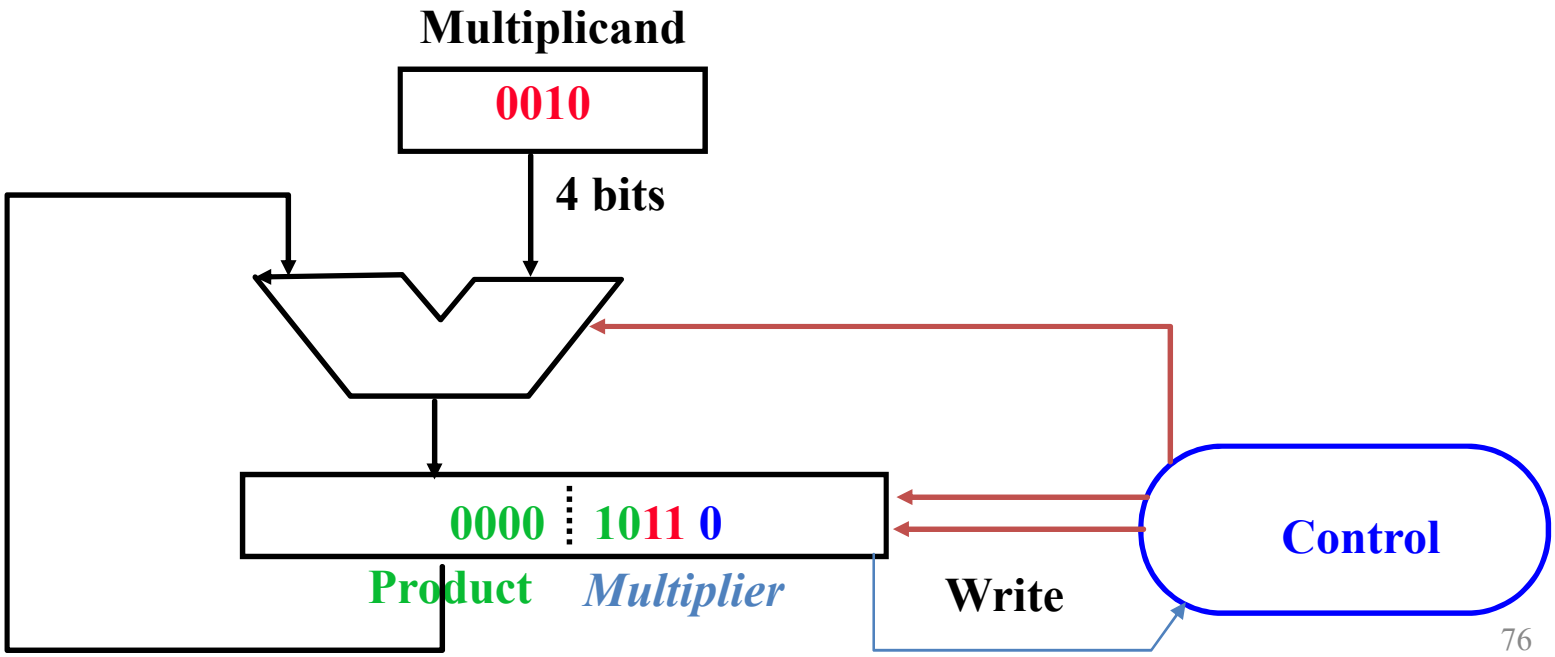


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = −3 = 1101, multiplicand = 0010.

2nd repetition:

Step 2: Shift product register right 1 bit



Example

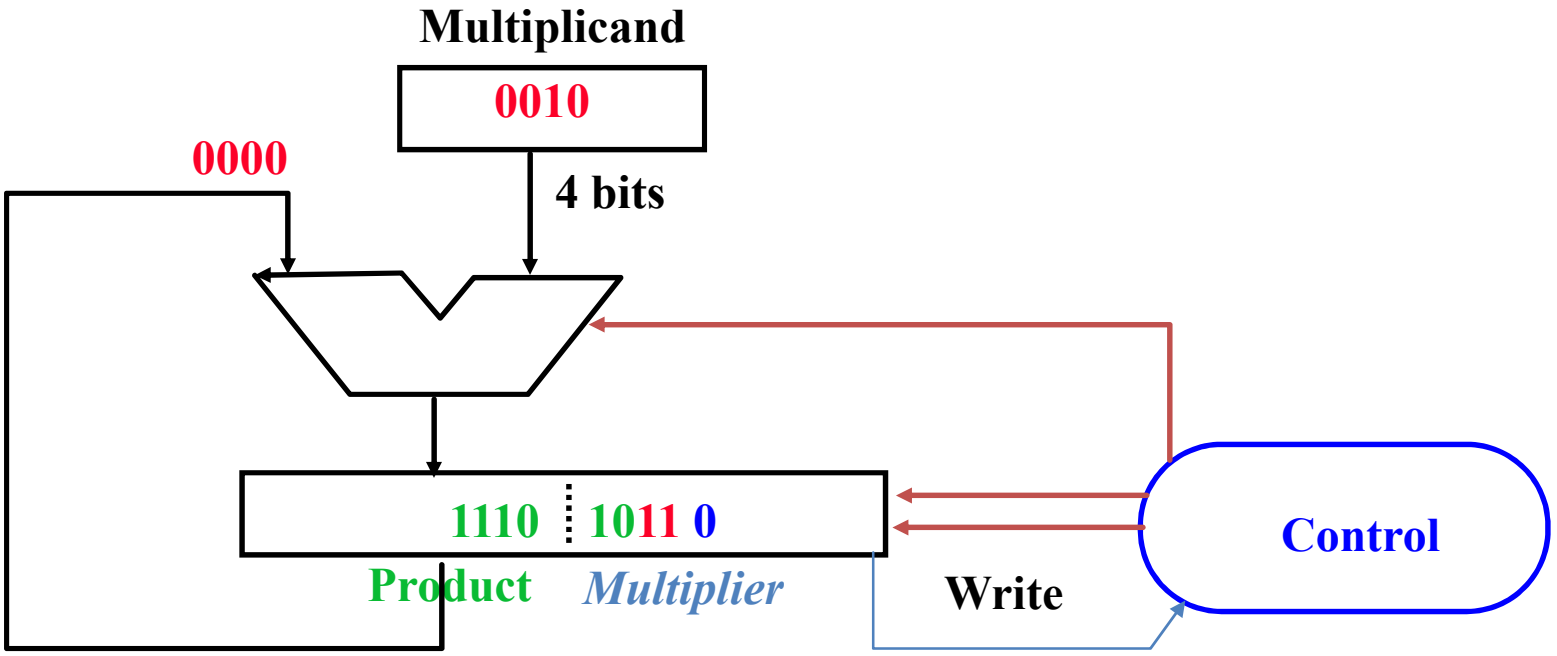
- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = −3 = 1101, multiplicand = 0010.

3rd repetition:

Step 1: Product lower 2 bits == 10

Step 1b: Product left half = Product left half - multiplicand

0000 - 0010 = 0000 + 1110 = 1110

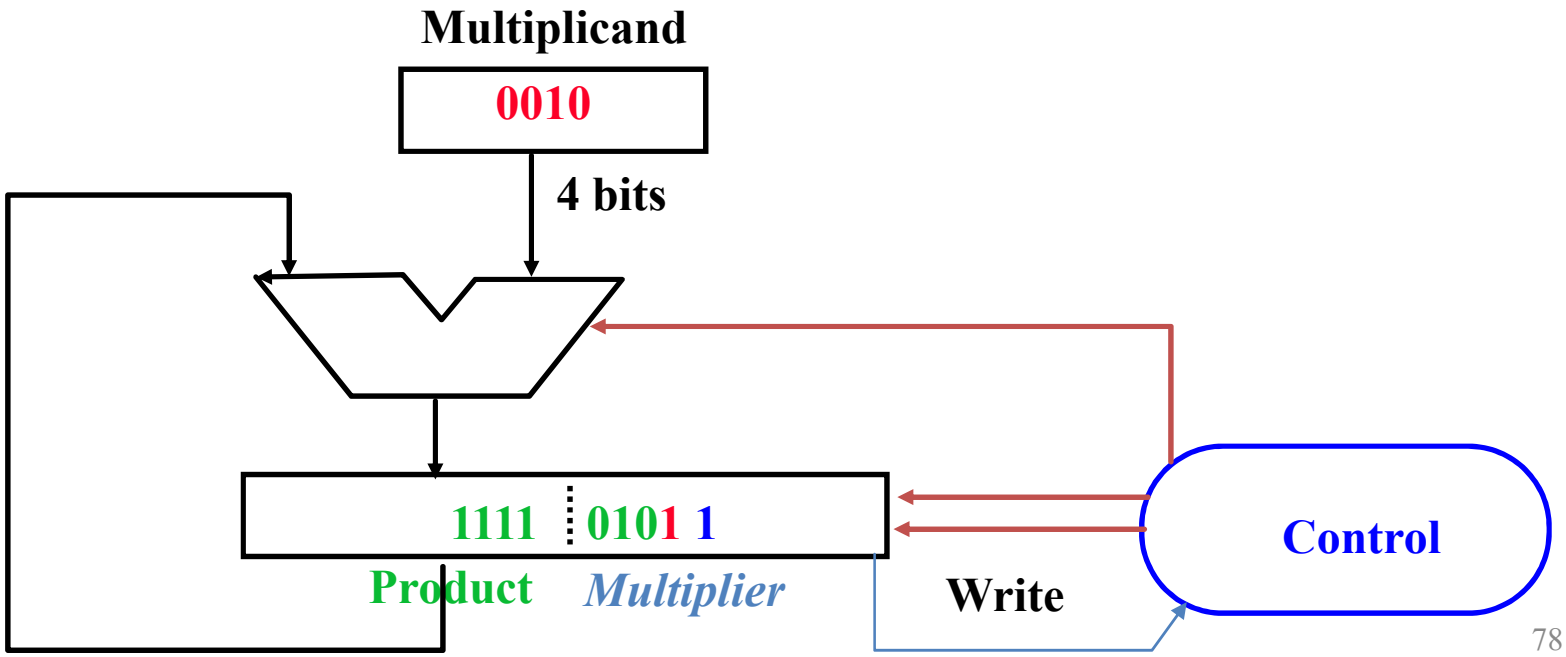


Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = −3 = 1101, multiplicand = 0010.

3rd repetition:

Step 2: Shift product register right 1 bit



Example

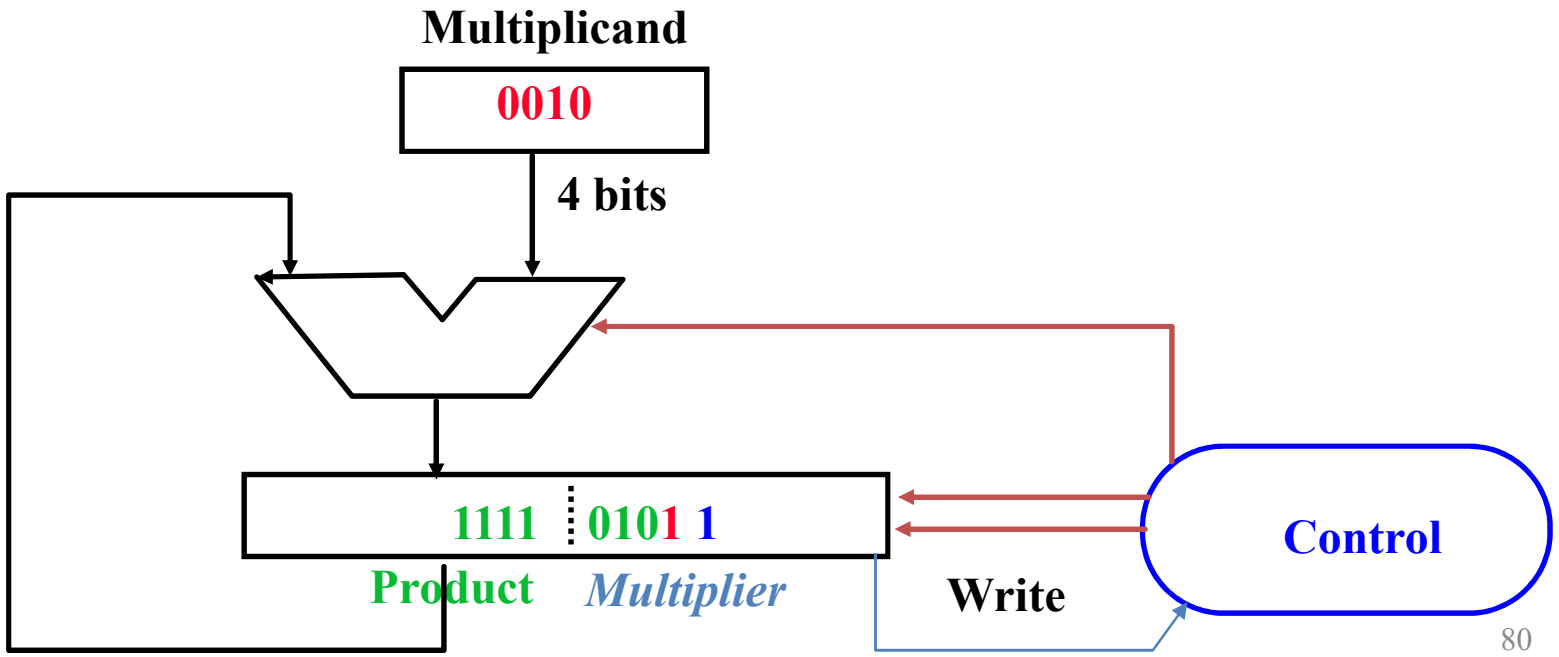
- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = -3 = 1101, multiplicand = 0010.

Example

- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = –3 = 1101, multiplicand = 0010.

4th repetition:

Step 1: Product lower 2 bits == 11
No need to add or sub.

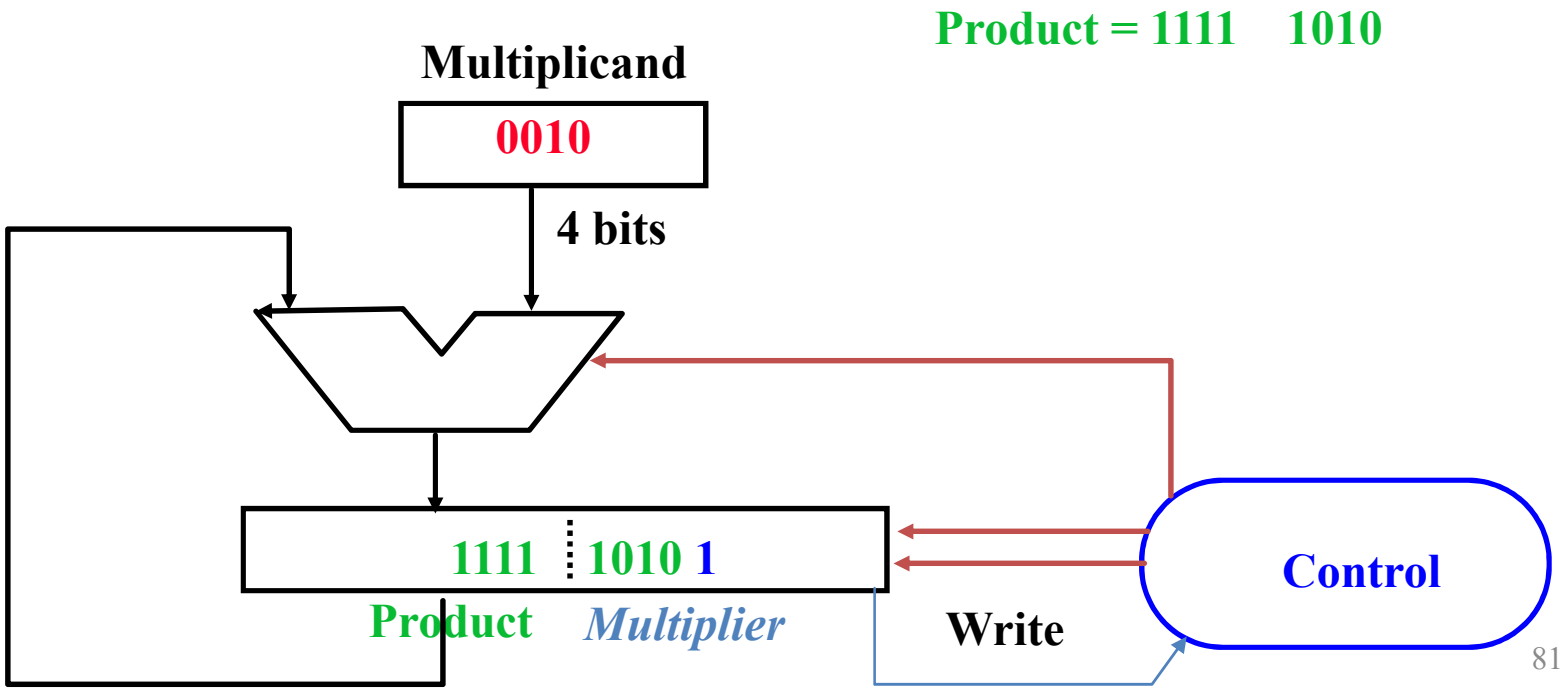


Example

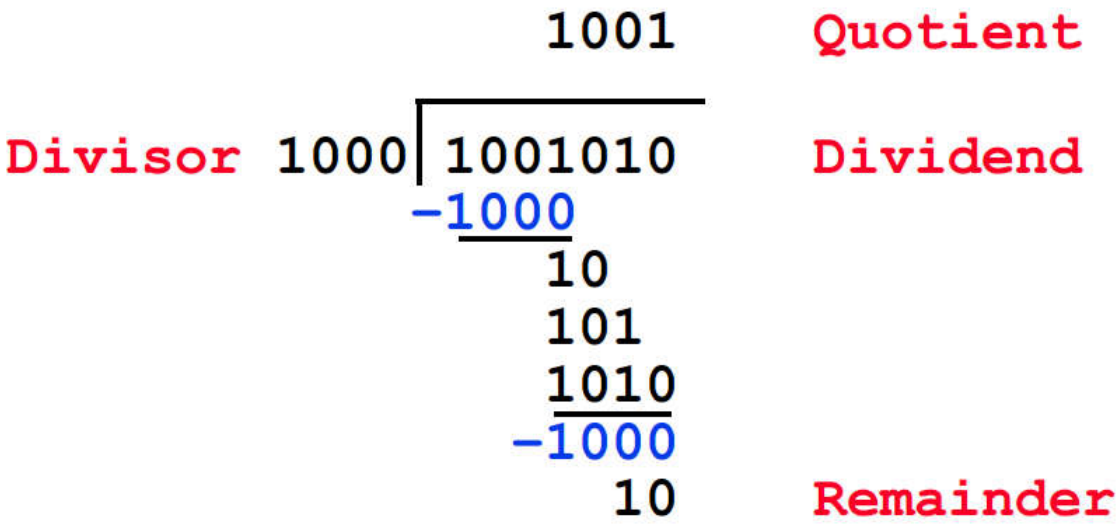
- 4-bit Multiplicand register, 4-bit ALU, 8-bit Product register, (Multiplier register is encapsulated in product register).
- Multiplier = –3 = 1101, multiplicand = 0010.

4th repetition: **Step 2: Shift product register right 1 bit**

4th repetition, STOP



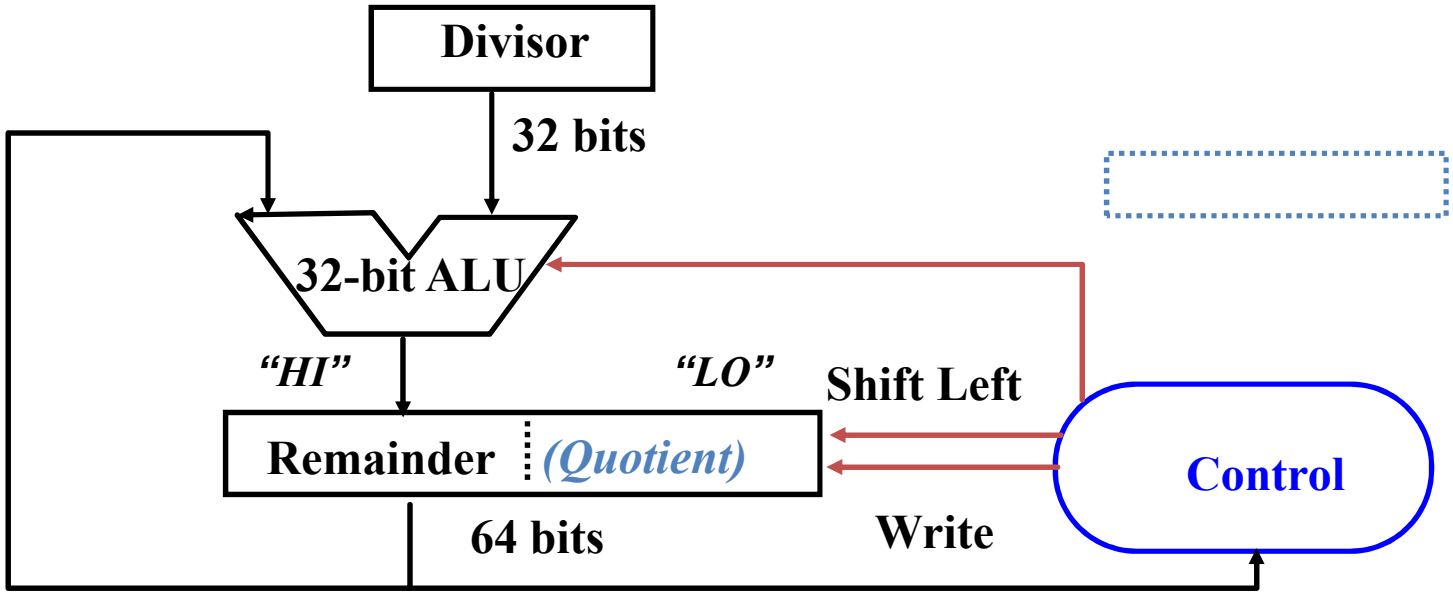
Divide: Paper & Pencil



- See how big a number can be subtracted, creating quotient bit on each step
- Dividend = Quotient x Divisor + Remainder
- 3 versions of divide, successive refinement

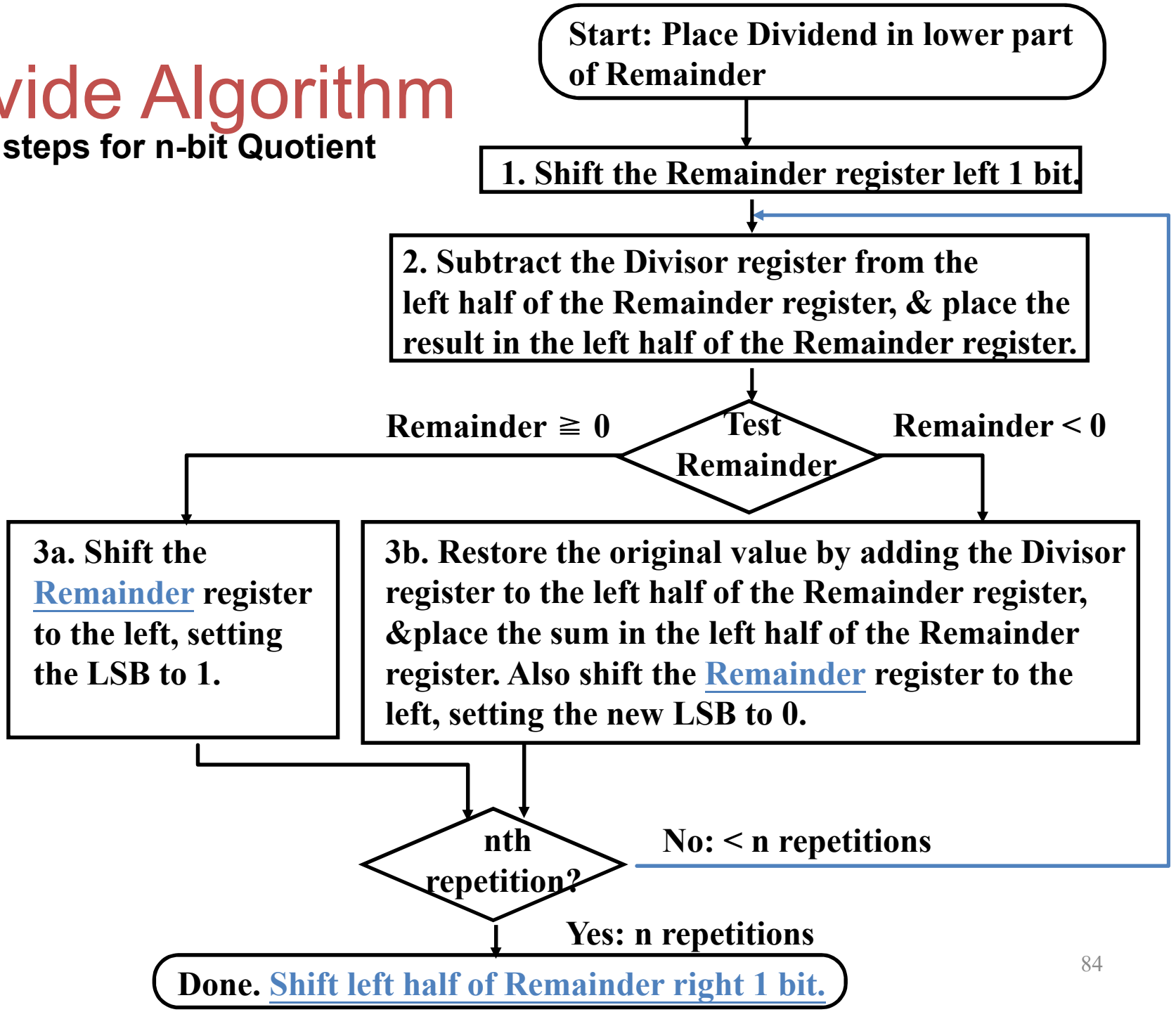
Divide Hardware

- 32-bit Divisor register, 32-bit ALU, 64-bit Remainder register, (0-bit Quotient register, combined in remainder register)



Divide Algorithm

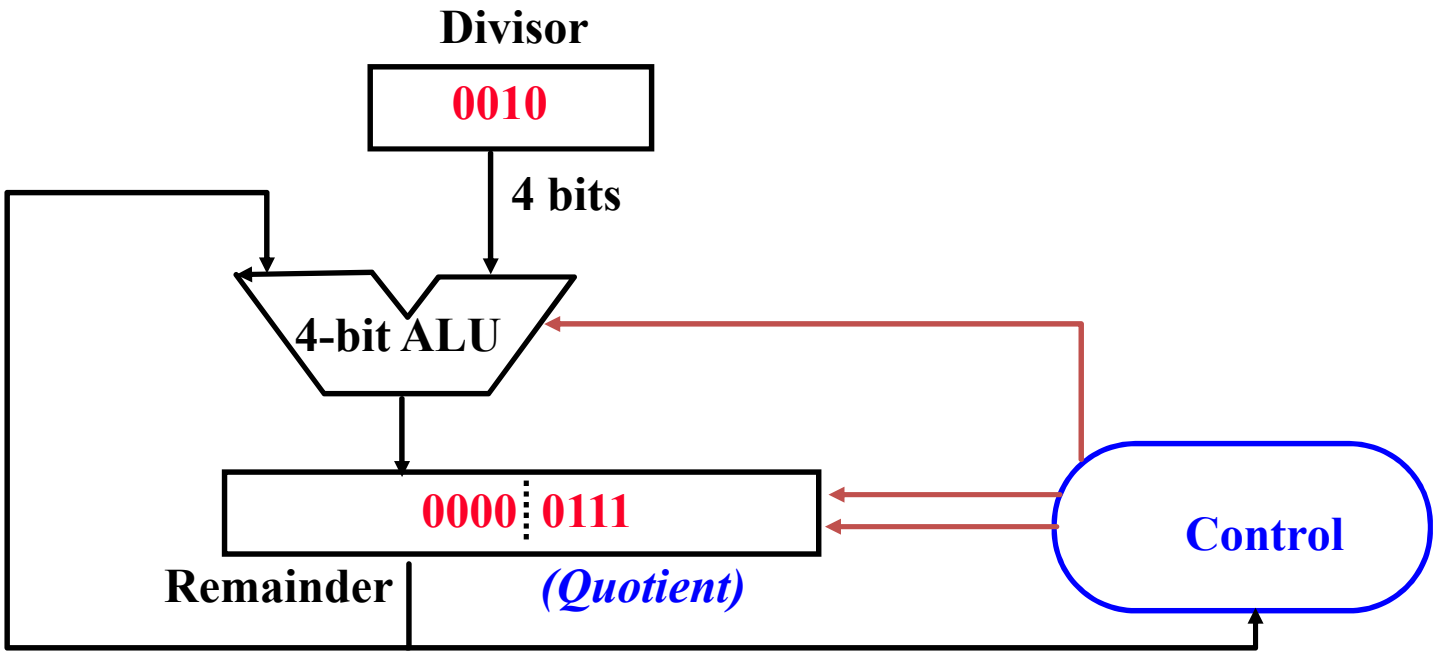
Takes n steps for n-bit Quotient



Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

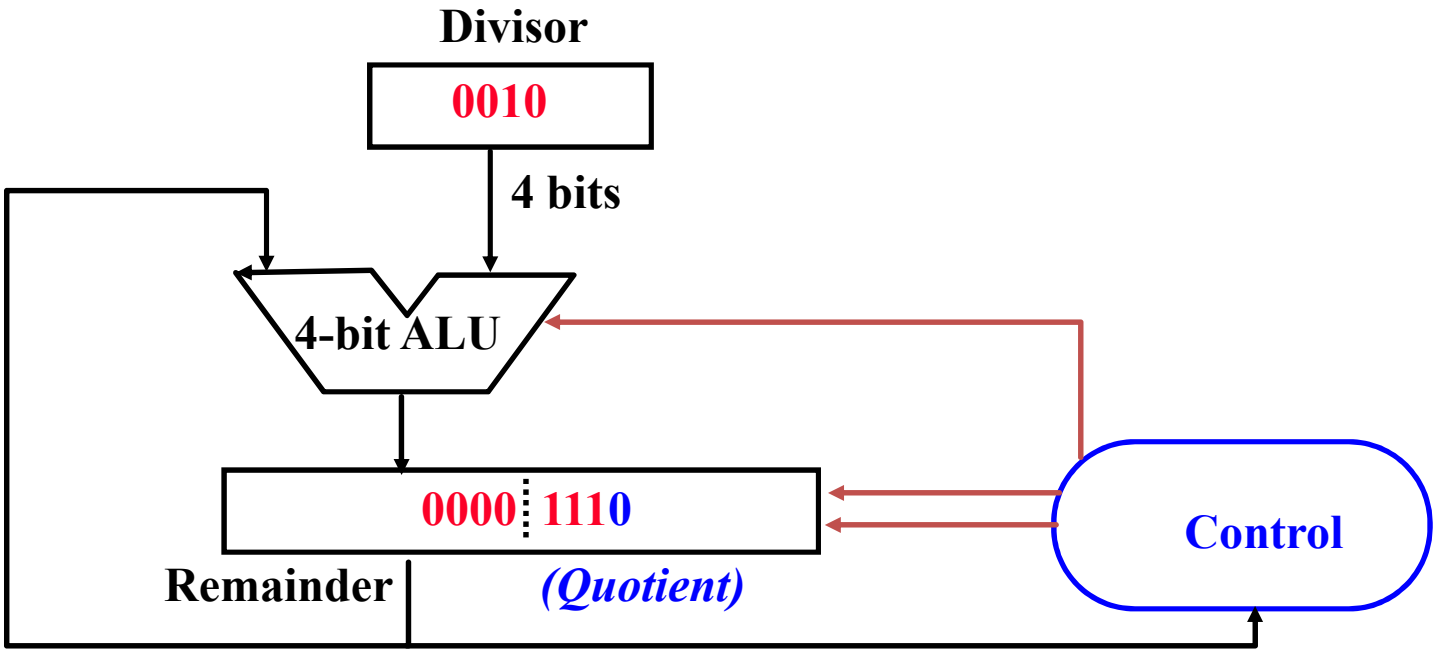
Initial state:



Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

Initial state: **Step 1: shift remainder left 1 bit**



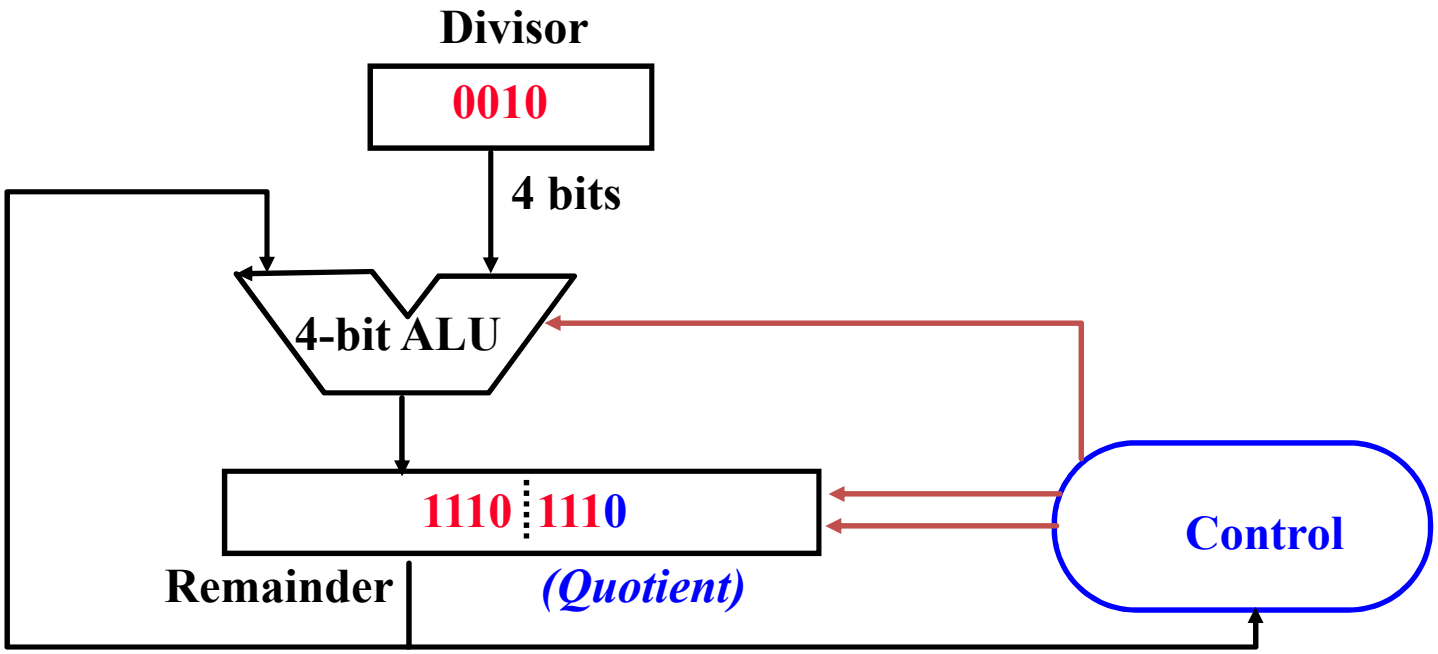
Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

1st repetition: **Step 2: remainder upper half = remainder upper half – divisor**

= 0000 – 0010

= 1110

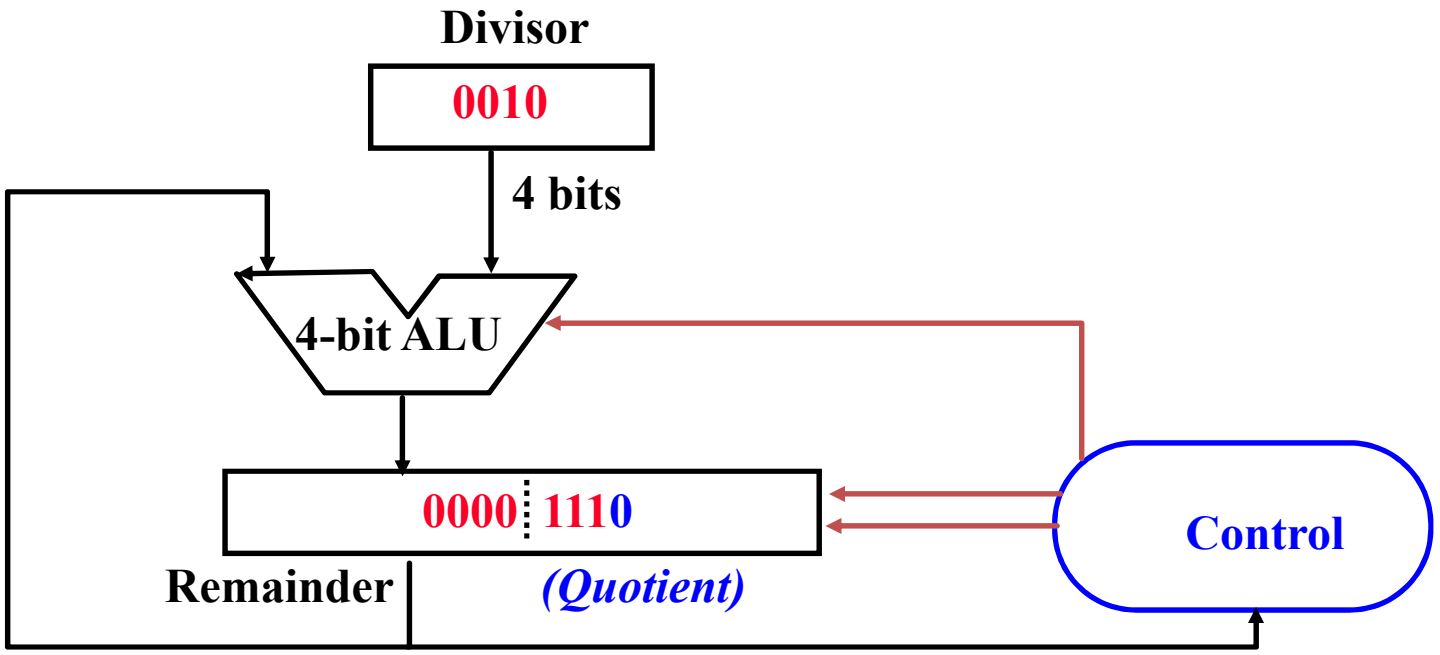


Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

1st repetition: **As remainder < 0**

Step 3b: restore remainder upper half

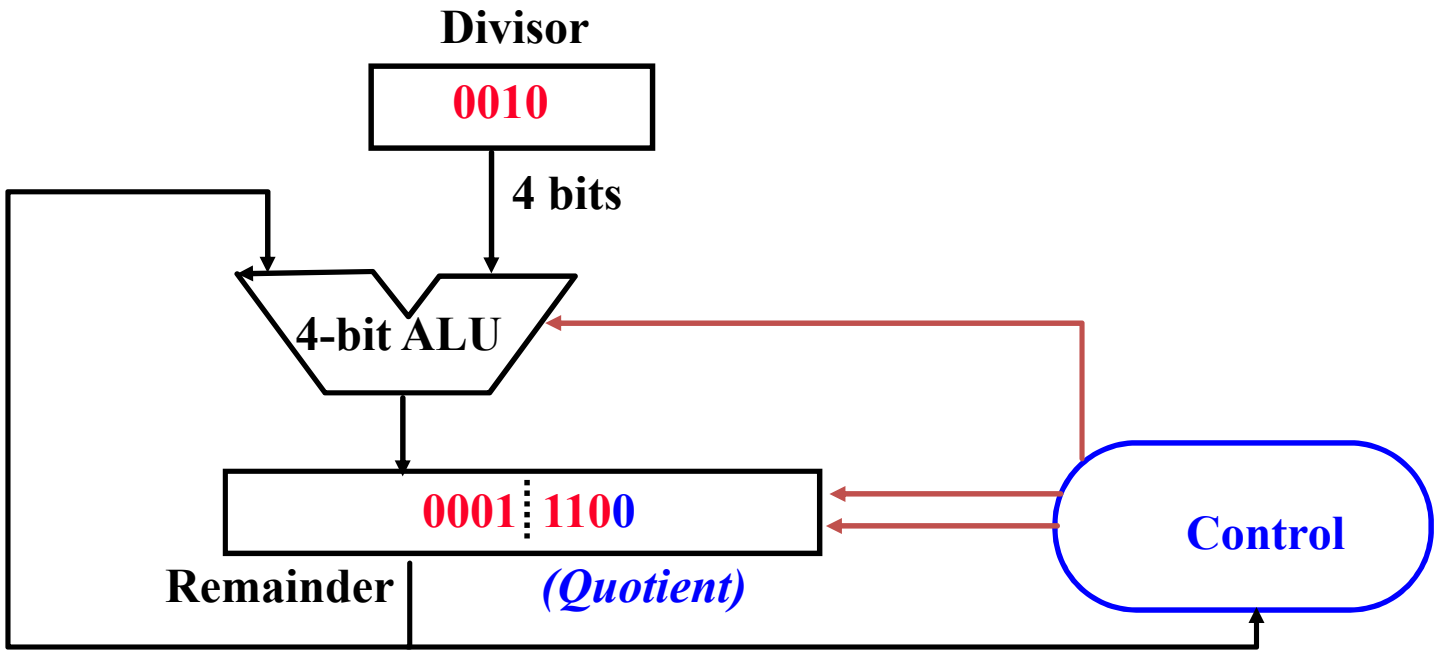


Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

1st repetition: **Step 3b: shift remainder left 1 bit**

set remainder LSB = 0



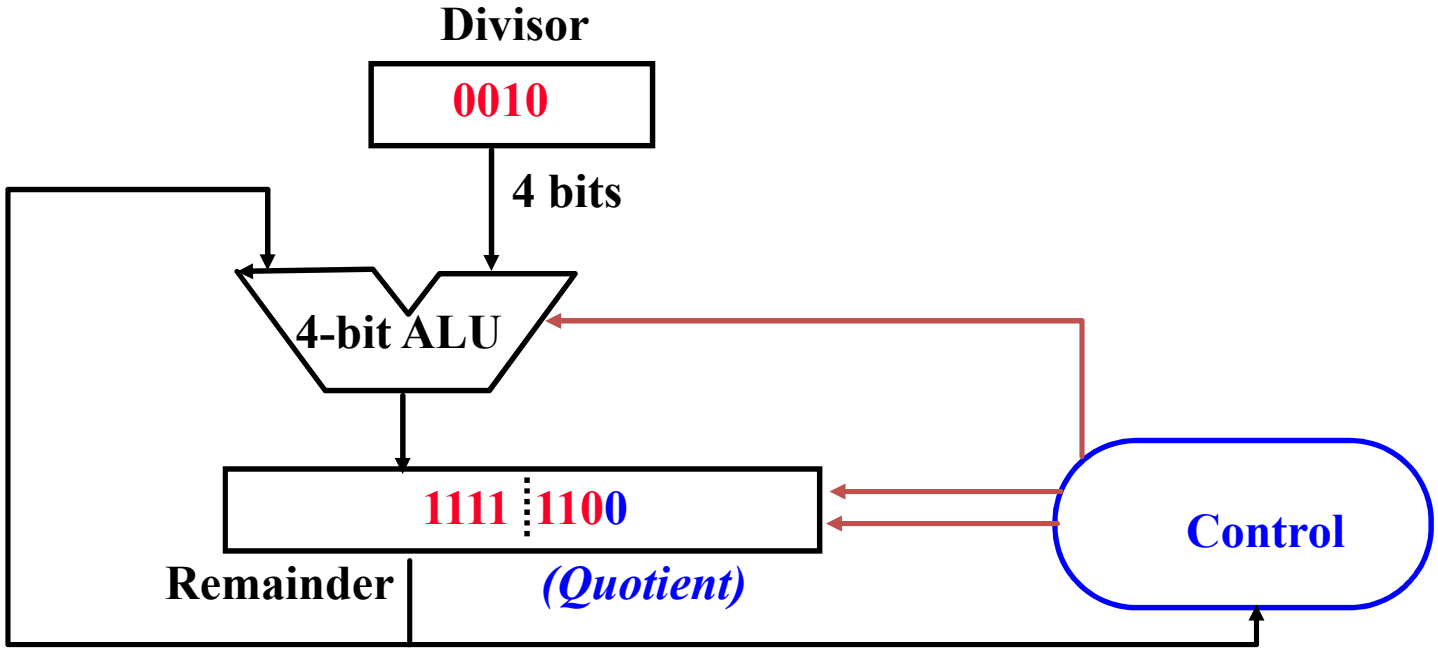
Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

2nd repetition: **Step 2: remainder upper half = remainder upper half – divisor**

= 0001 – 0010

= 1111



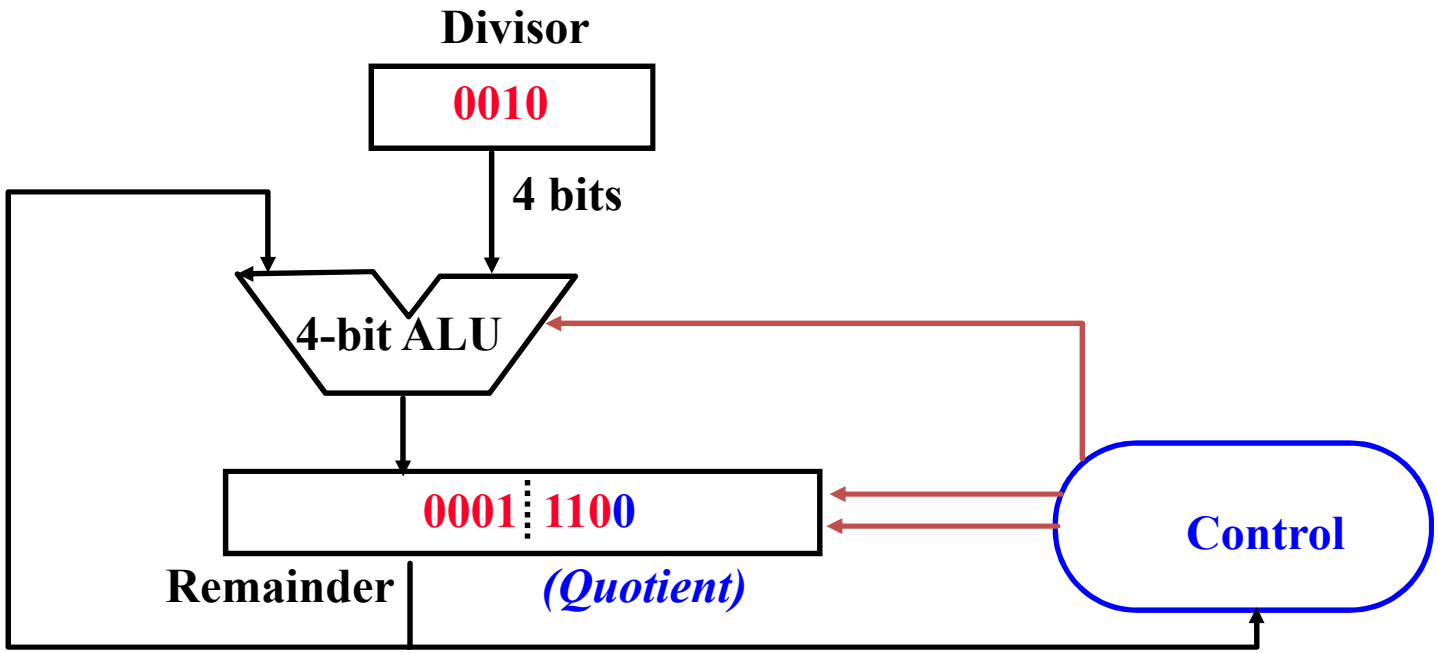
Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

2nd repetition:

Since remainder < 0

Step 3b: restore remainder



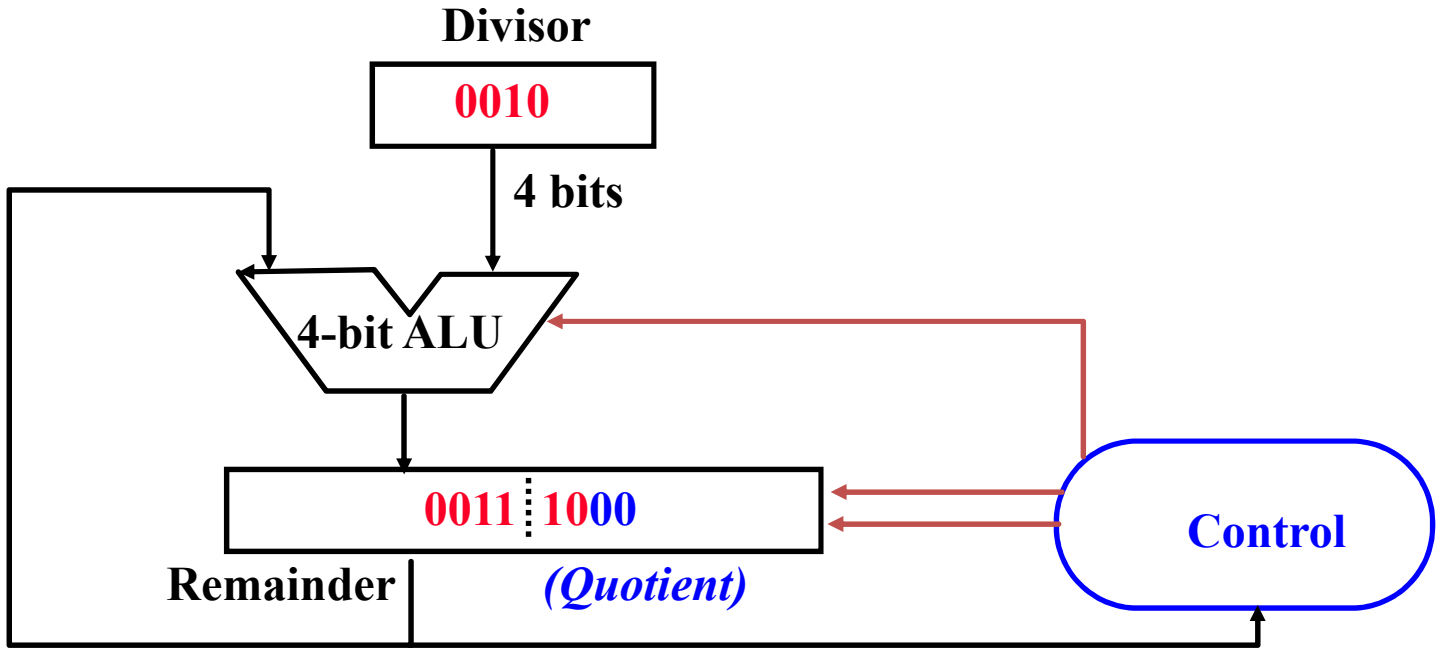
Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

2nd repetition:

Step 3b: shift remainder left 1 bit

set remainder LSB = 0

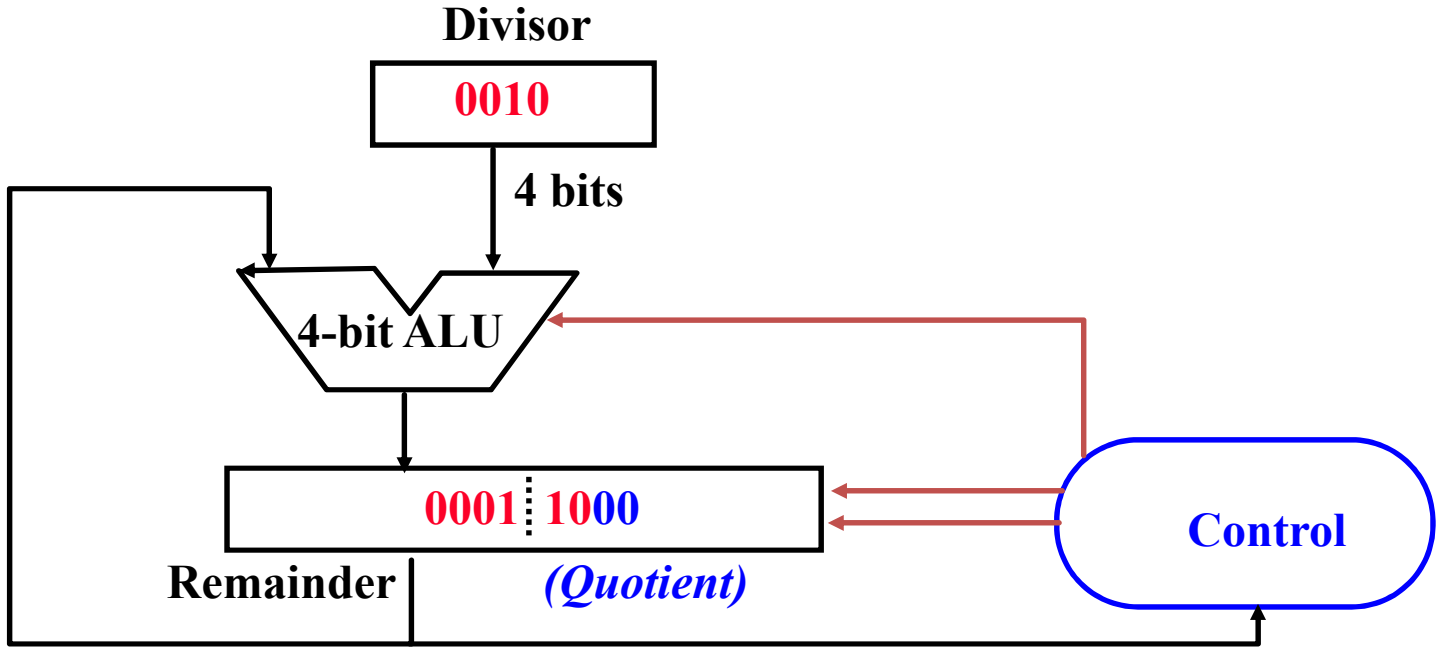


Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

3rd repetition:

Step 2: remainder upper half = remainder upper half – divisor
= 0011 – 0010
= 0001

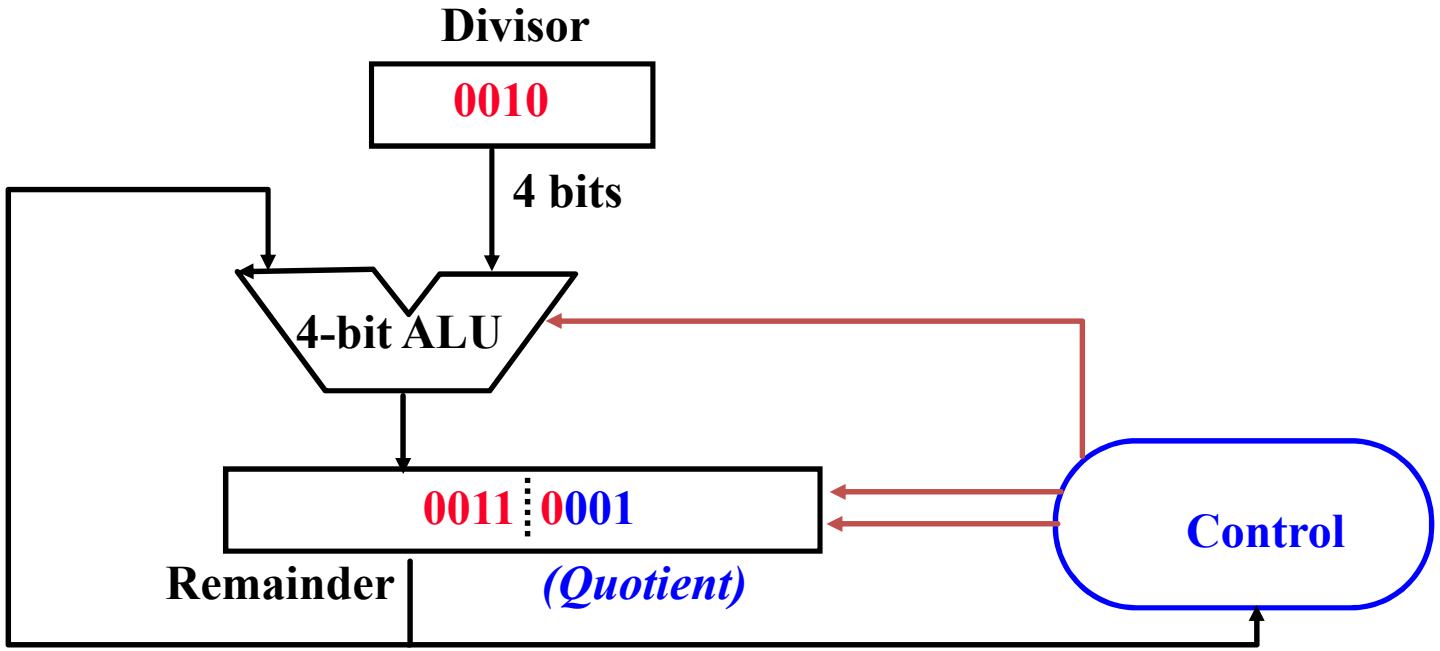


Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

3rd repetition:

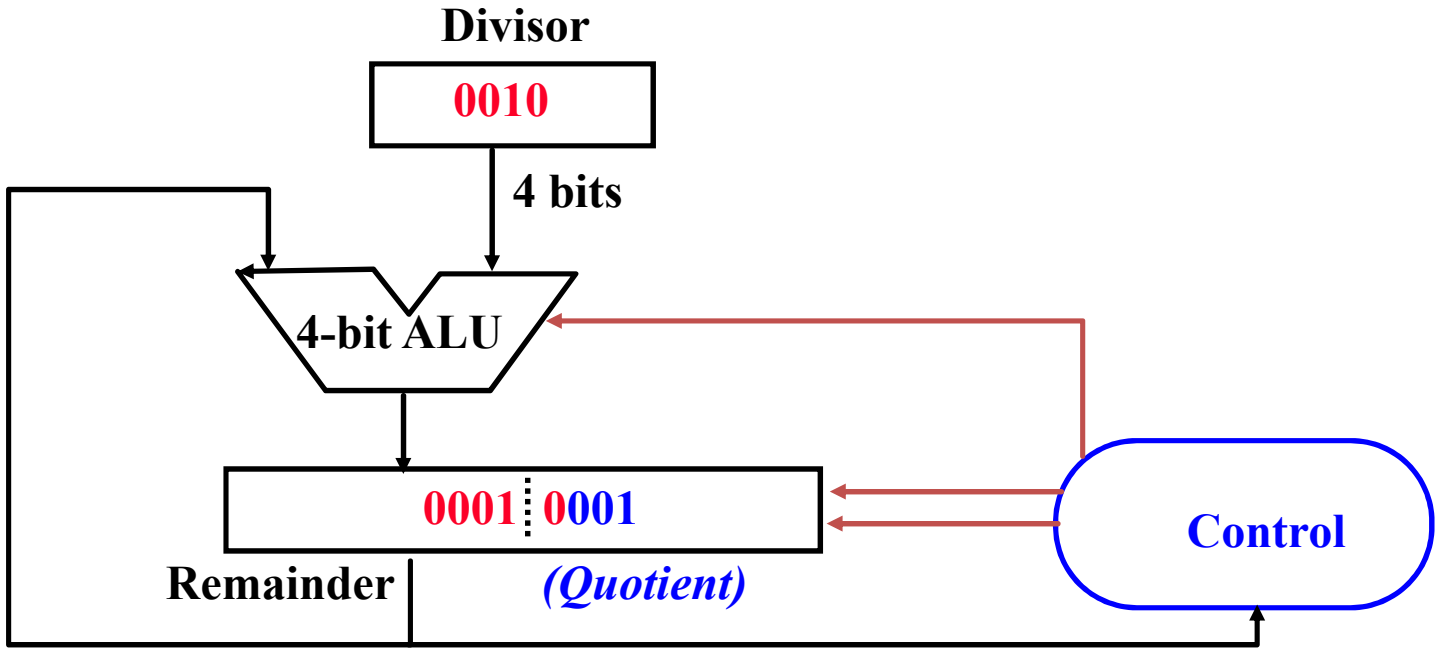
Since remainder >= 0
Step 3a: shift remainder left 1 bit
set remainder LSB = 1



Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

4th repetition: **Step 2: remainder upper half = remainder upper half – divisor**
= 0011 – 0010
= 0001

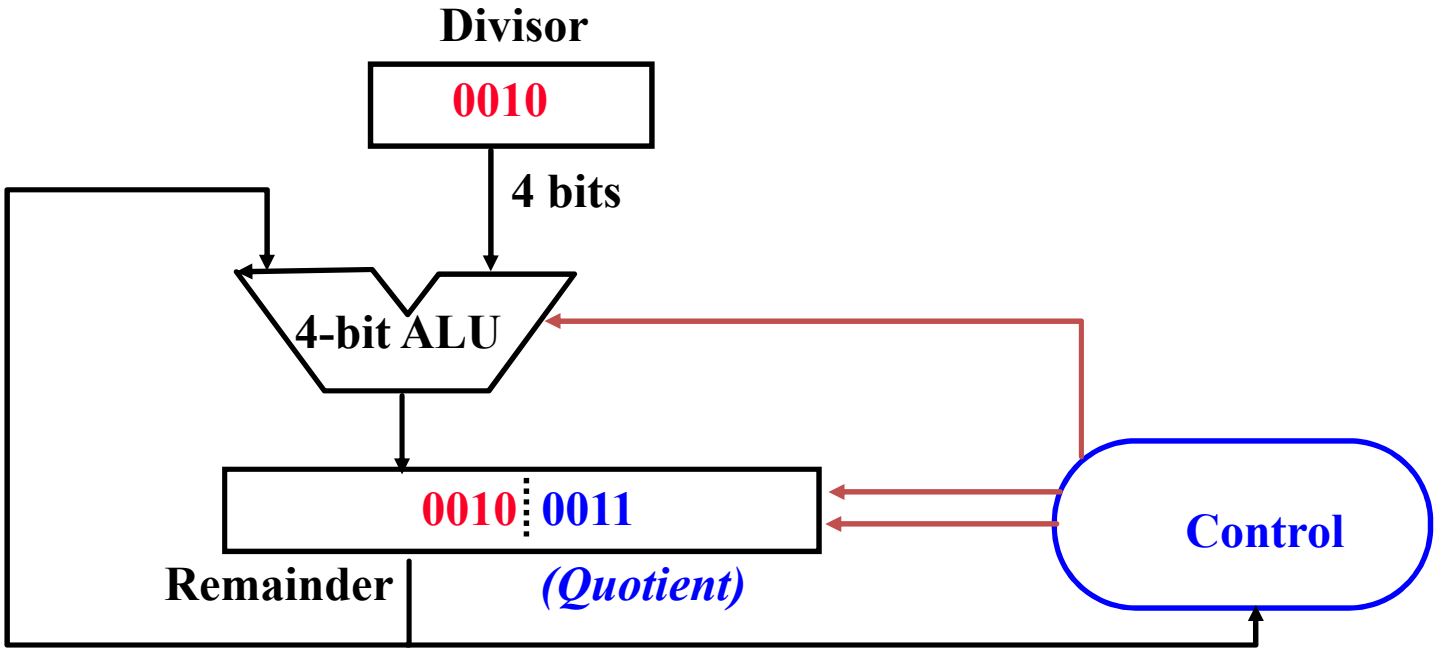


Example

- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

4th repetition:

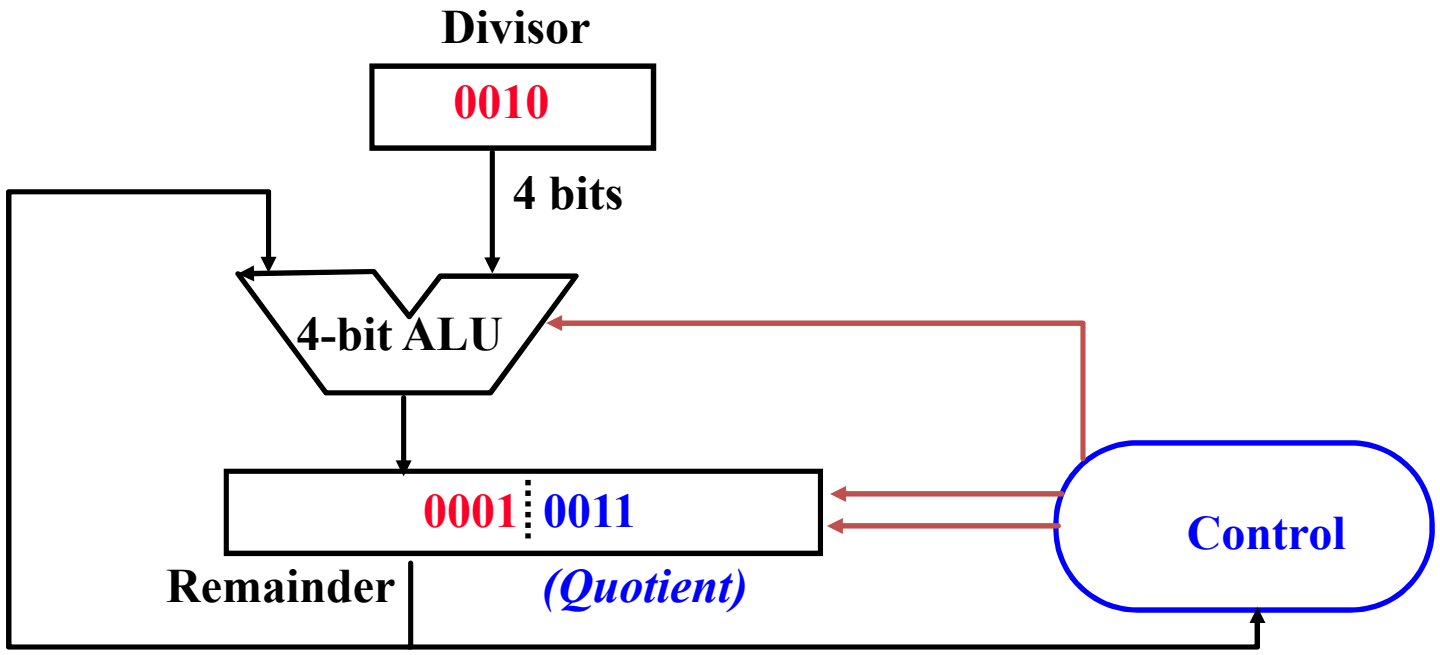
Since remainder >= 0
Step 3a: shift remainder left 1 bit
set remainder LSB = 1



Example

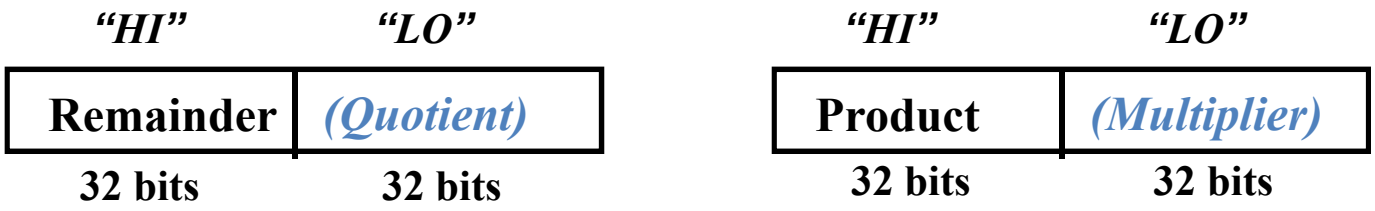
- 4-bit Divisor register, 4-bit ALU, 8-bit Remainder register.
- Dividend = 0111, divisor = 0010.

4th repetition: **4th repetition, quite the loop**
Shift upper half of remainder right 1 bit
STOP



Summary

- Divide and Multiply use the same hardware architecture
  - Just need ALU to add or subtract
- 64-bit register for multiply and divide



- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: the sign of Dividend and Remainder must be the same
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., –7 ÷ 2 = –3, remainder = –1

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

Lab01: Implementation of Multiplier and Divider

- **Purpose**
  - Understand the basic principle of multiplier and divider.
  - Know how to implement a multiplier and a divider.

- **Introduction**
  - In this lab experiment, you are going to implement a 32-bit *multiplier* simulator as well as a 32-bit *divider* simulator using C language. Your simulators will perform multiply and divide operations for 32-bit binary numbers. Some C files implementing each component of the *multiplier* and the *divider* will be provided to you, you are required to modify and fill in the body of the functions in these files.