

CSE261: Computer Architecture

5. Arithmetic for Computers (1)

Seongil Wi

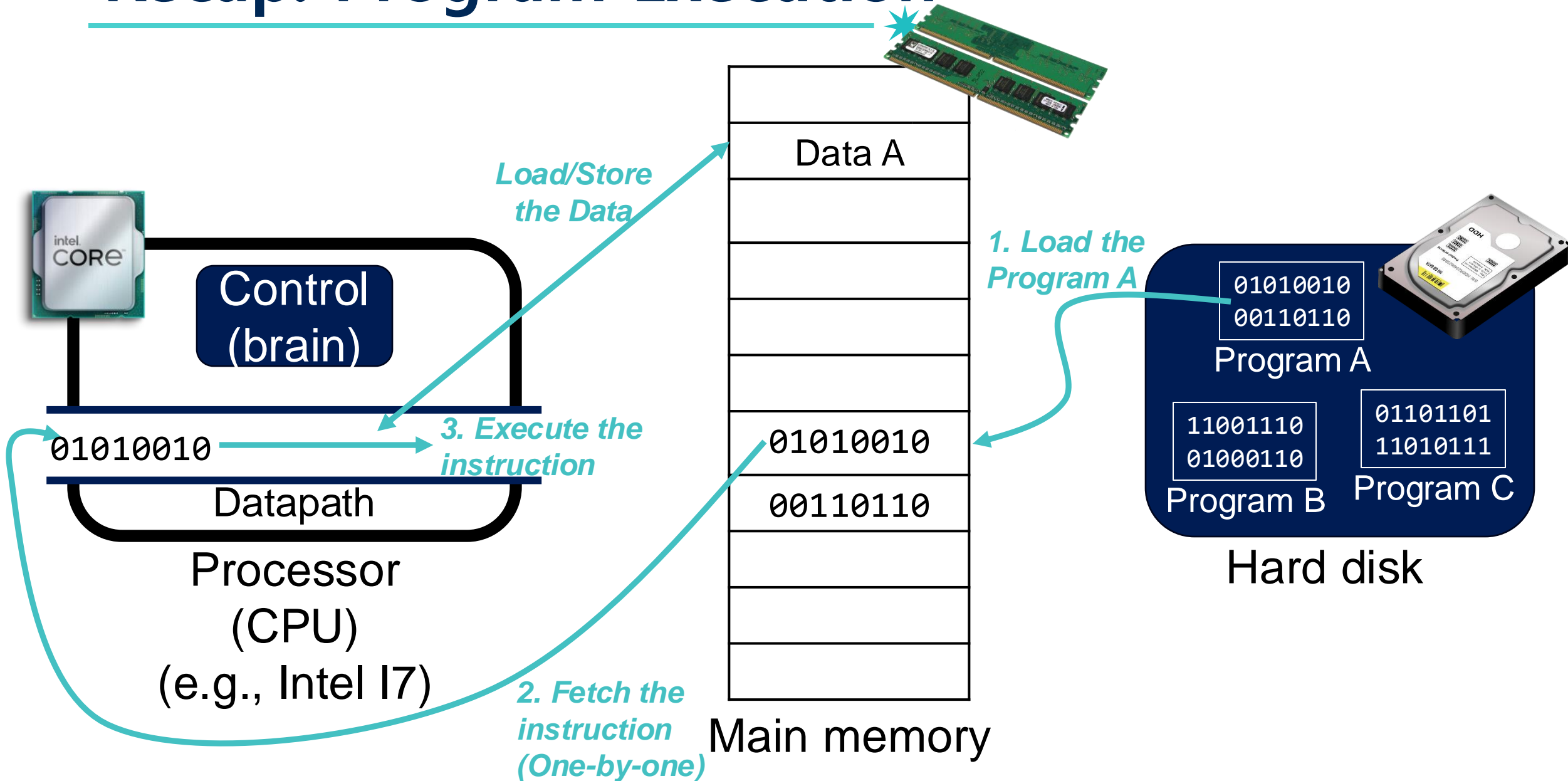
Notification: HW1

2

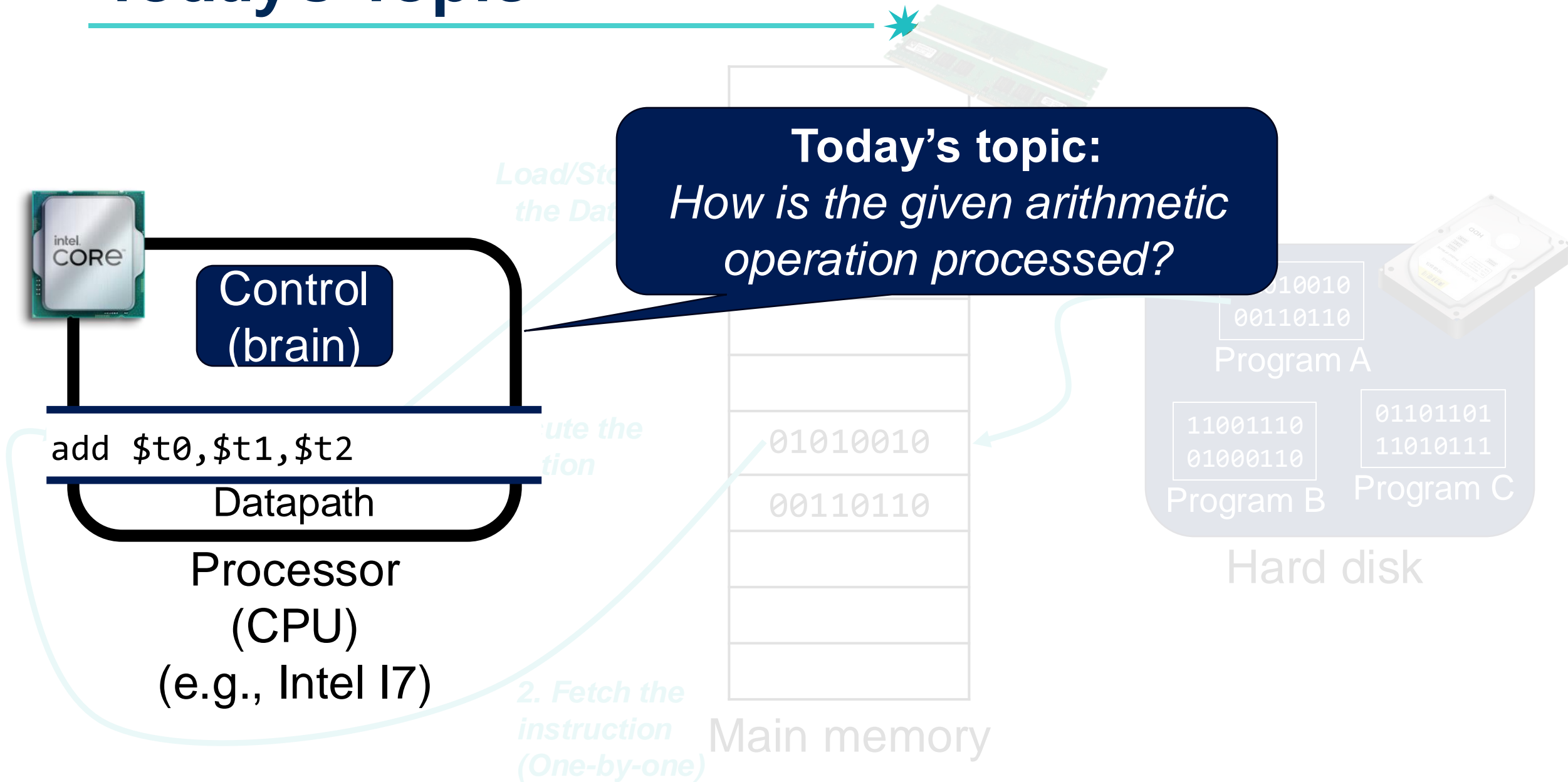


- Due date: **10/10, 11:59PM**

Recap: Program Execution



Today's Topic



Integer Addition & Subtraction

Integer Addition



Example: 7+6

+	...	0	0	0	1	1	1
	...	0	0	0	1	1	0
							1

Integer Addition

7

Carry

Example: 7+6

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

Integer Addition

Example: 7+6

Carry

8

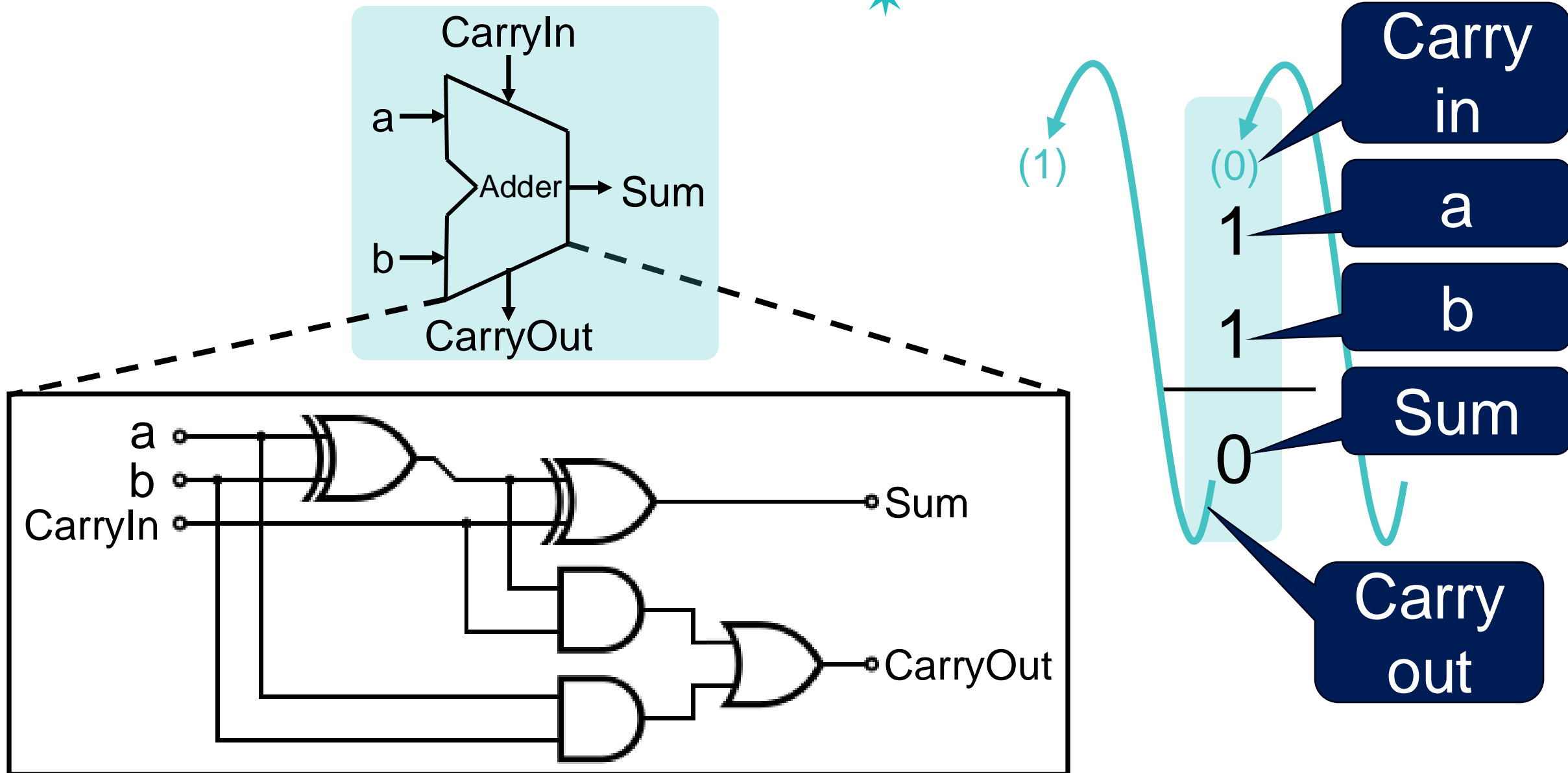
Carry in

+	...	0	0	0	(1) 1	(0) 1	1
	...	0	0	0	1	1	0

0 1

Carry out

Hardware: A 1-bit Adder (a.k.a. Full Adder)



Background: The Basics of Logic Gates¹⁾

10

AND



Input		Output
a	b	
0	0	0
0	1	0
1	0	0
1	1	1

OR



Input		Output
a	b	
0	0	0
0	1	1
1	0	1
1	1	1

XOR

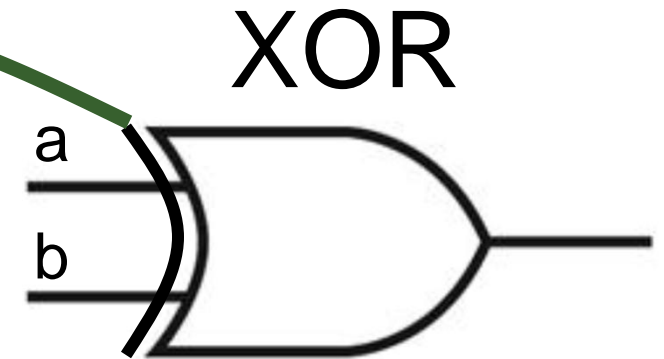
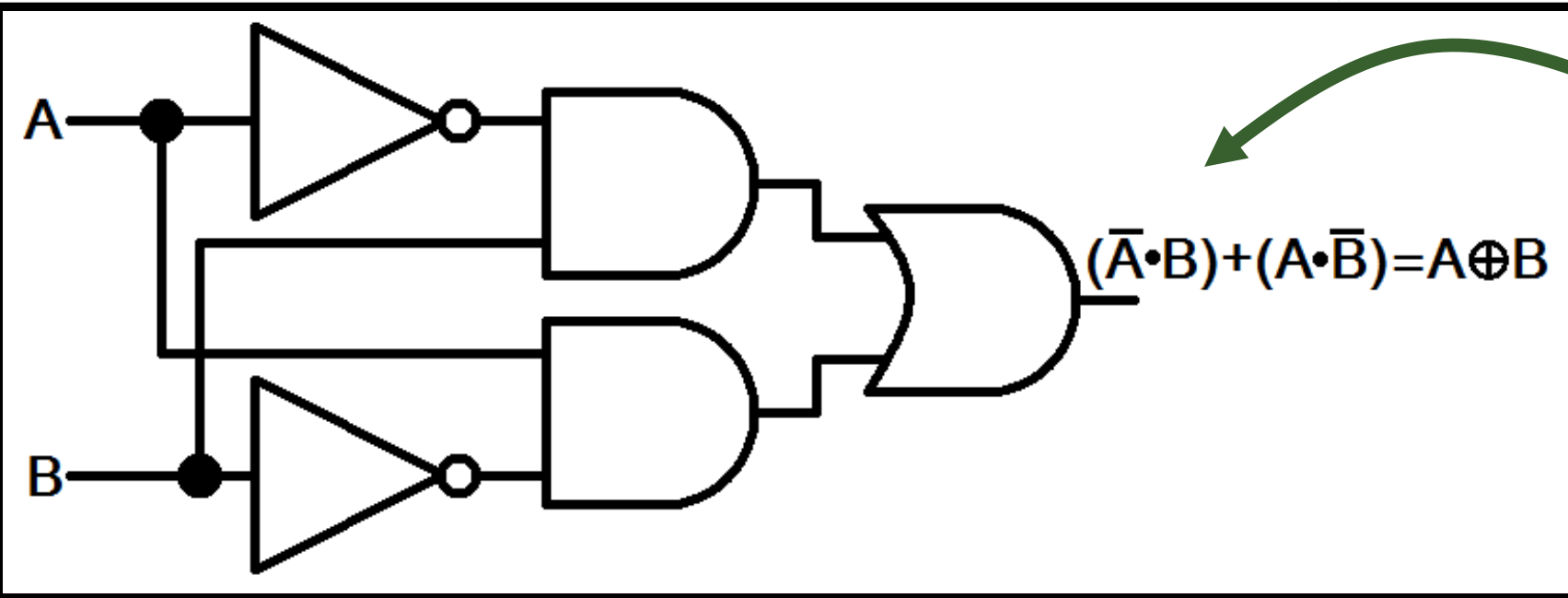


Input		Output
a	b	
0	0	0
0	1	1
1	0	1
1	1	0

¹⁾ A device that implements basic logic functions

Background: The Basics of Logic Gates¹⁾

11



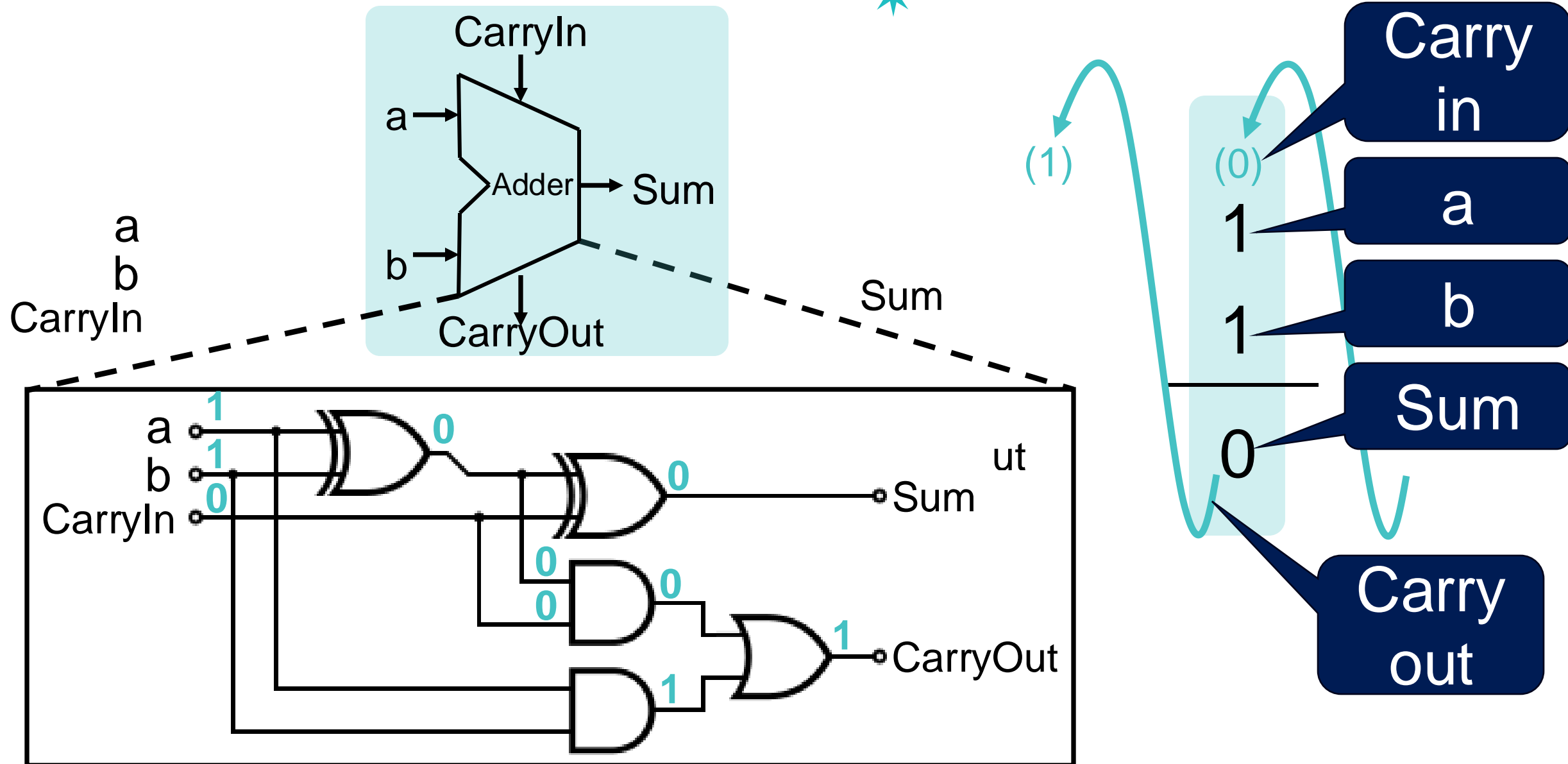
0	0	0
0	1	0
1	0	0
1	1	1

0	0	0
0	1	1
1	0	1
1	1	1

Input		Output
a	b	
0	0	0
0	1	1
1	0	1
1	1	0

¹⁾ A device that implements basic logic functions

Hardware: A 1-bit Adder (a.k.a. Full Adder)

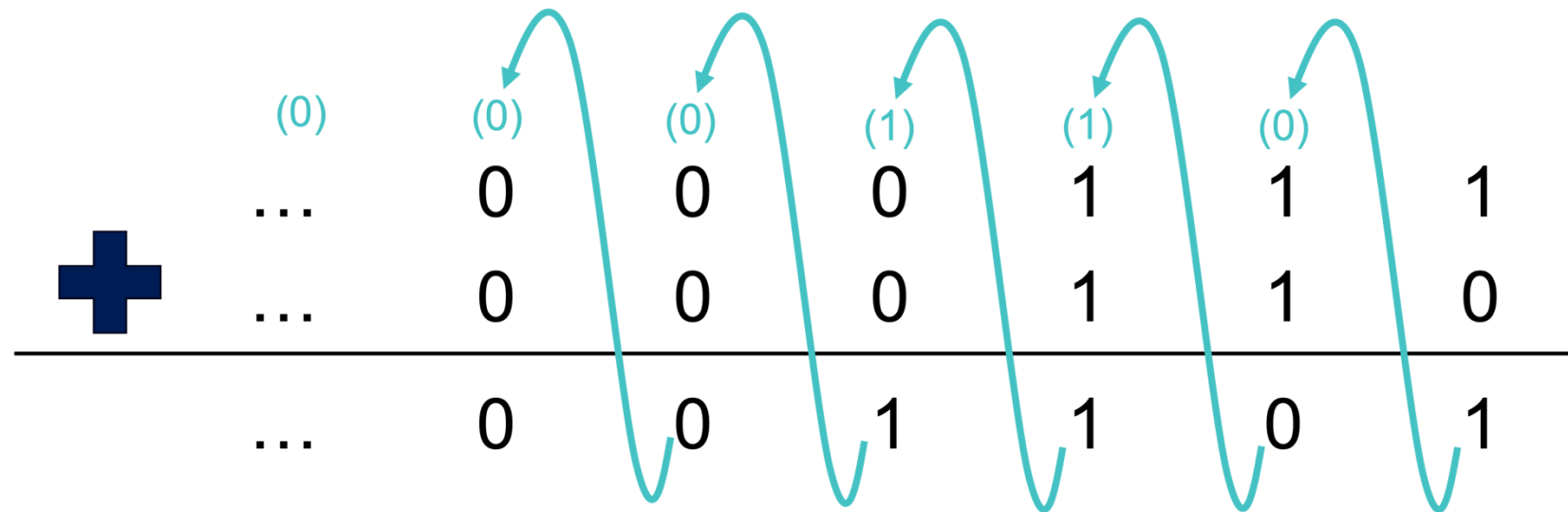
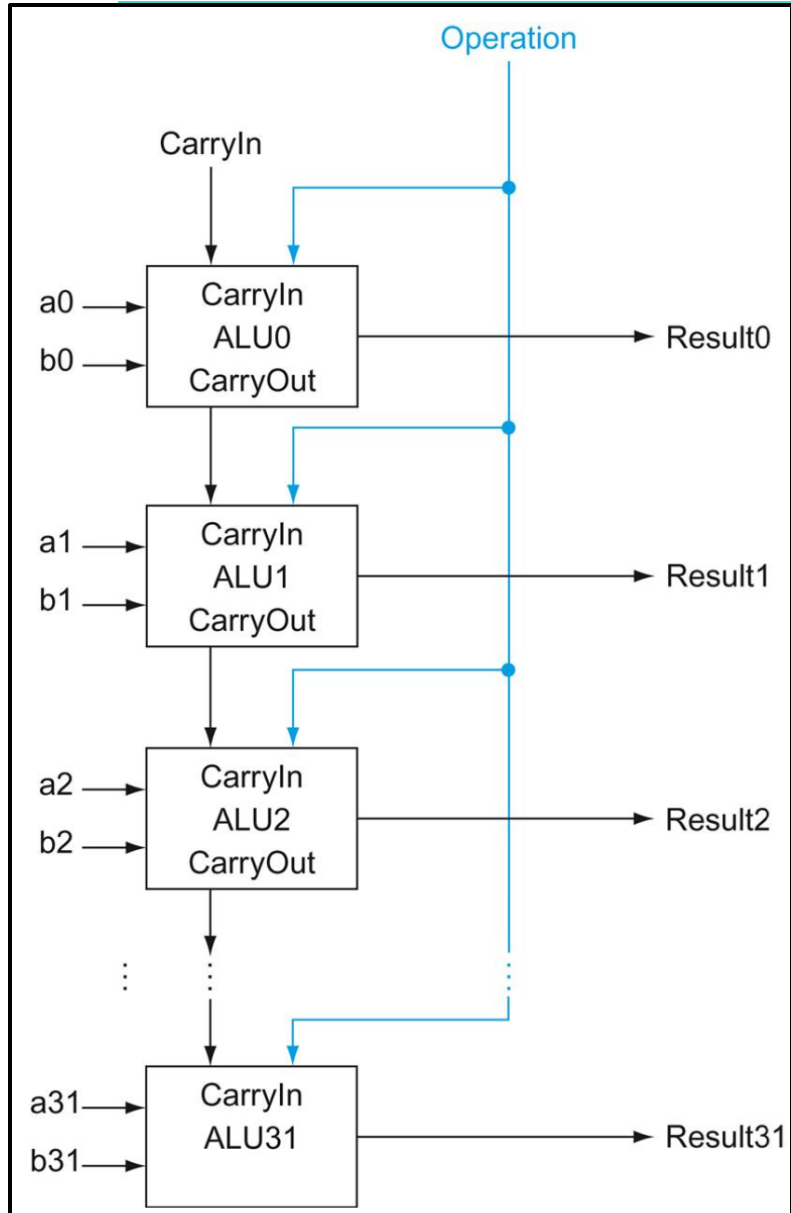


WIP: Truth Table for a Full Adder



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{two}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{two}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{two}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{two}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{two}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{two}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{two}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{two}}$

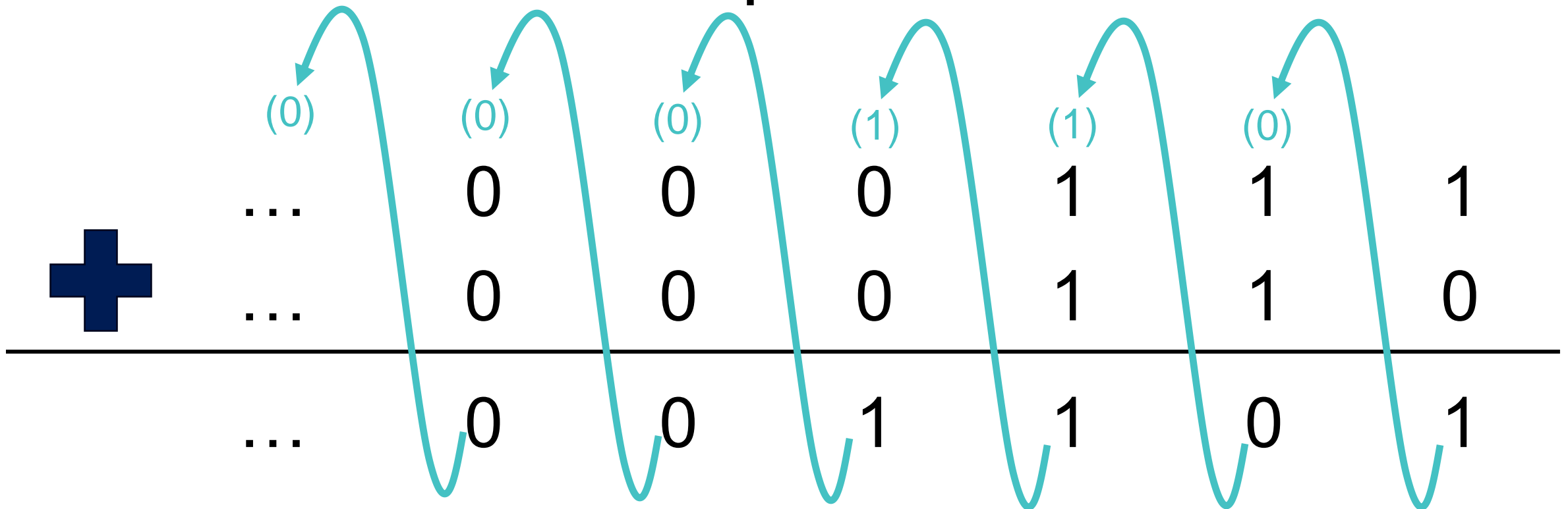
Hardware: N-bit Ripple-Carry Adder



Integer Addition



Example: 7+6



Integer Addition: Dealing with Overflow

16

- Overflow if result out of range
 - Adding **positive number** with **negative number**,
no overflow
 - Adding two **positive number** operands
 - Expected result: positive number
 - Overflow if result sign (MSB) is 1
 - Adding two **negative number** operands
 - Expected result: negative number
 - Overflow if result sign (MSB) is 0

Integer Addition: Dealing with Overflow

17

Basic idea: If ...

Positive Number  Positive Number  Negative Number

Or...

Negative Number  Negative Number  Positive Number

Overflow is occurred!

Integer Subtraction



- Add negation of second operand
(Do not have specified hardware for subtraction)
- Example: $7 - 6 = 7 + (-6)$

+7:	0000	0000	...	0000	0111
-6:	1111	1111	...	1111	1010
<hr/>					
+1:	0000	0000	...	0000	0001

Integer Subtraction: Dealing with Overflow

19

- Overflow if result out of range
 - Subtracting two positive numbers or two negative numbers, no overflow
 - Subtracting positive number from negative number operand
 - Expected result: negative number
 - Overflow if result sign (MSB) is 0
 - Subtracting negative number from positive number operand
 - Expected result: positive number
 - Overflow if result sign (MSB) is 1

Integer Subtraction: Dealing with Overflow

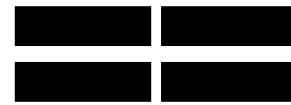
20

Basic idea: If ...

Positive
Number



Negative
Number



Negative
Number

Or...

Negative
Number



Positive
Number



Positive
Number

Overflow is occurred!

Dealing with Overflow

- add, addi, sub cause exceptions on overflow
- addu, addiu, subu do *not* cause exceptions on overflow

The ISA must provide
a way to ignore overflow

Dealing with Overflow

- add, addi, sub cause exceptions on overflow

C ignores overflow

The MIPS C compilers will always generate the unsigned version of the arithmetic instructions

- addu, addiu, subu do *not* cause exceptions on overflow

The ISA must provide
a way to ignore overflow

Dealing with Overflow



- add, addi, sub cause exceptions on overflow

Ada and Fortran require raising an exception

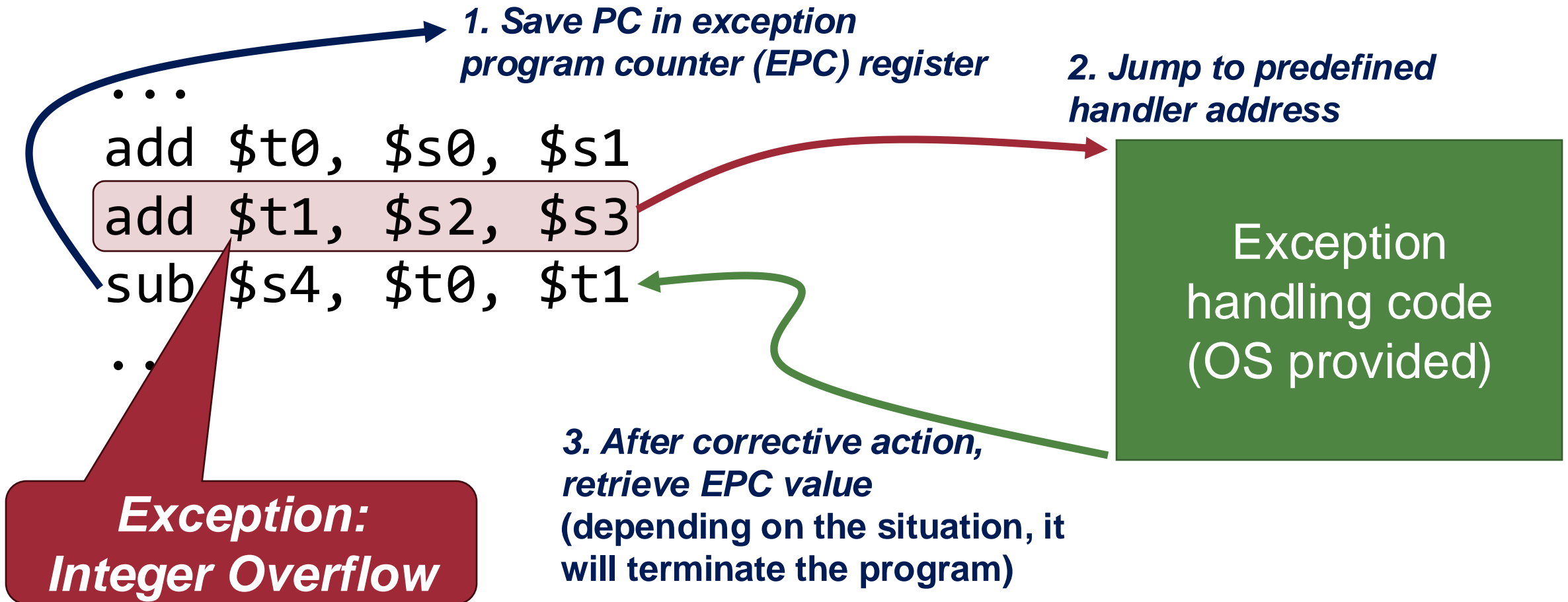
The MIPS Fortran compilers
will pick the appropriate instruction

- addu, addiu, subu do *not* cause exceptions on overflow

The ISA must provide
a way to ignore overflow

FYI: Exception

- add, addi, sub cause exceptions on overflow



Integer Multiplication

How to Multiply Two Integers?



- MIPS multiplies two integers without dedicated multiplying unit

With *add* and *shift* operation!

Multiplication



- Let's think about the multiplication in grade school level

$$\begin{array}{r} 312 \\ \times 203 \\ \hline 936 \\ 000 \\ 624 \\ \hline 63336 \end{array}$$

Multiplicand

Multiplier

Product

Multiplication via Shifting and Addition

28

$$\begin{array}{r} 312 \\ \times 203 \\ \hline 936 \end{array}$$

$$\begin{array}{r} \hline 936 \end{array}$$

Multiplication via Shifting and Addition

29

$$\begin{array}{r} 3120 \\ \times 20 \\ \hline 936 \\ 000 \\ \hline 0936 \end{array}$$



Multiplicand: shift left



Multiplier: shift right

Multiplication via Shifting and Addition

30

$$\begin{array}{r} 312\cancel{00} \\ \times \quad \quad 2 \\ \hline 624 \\ \hline 63336 \end{array}$$



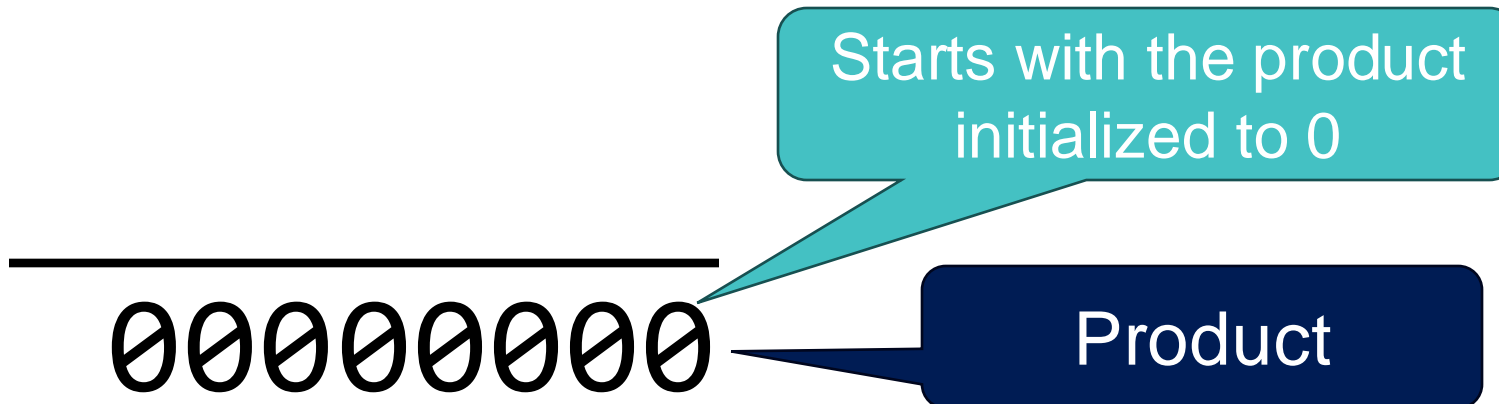
Multiplicand: shift left



Multiplier: shift right

Multiplying Two Binary Numbers

$$\begin{array}{r} 1000 \\ X \quad 1001 \\ \hline \end{array}$$



Multiplying Two Binary Numbers

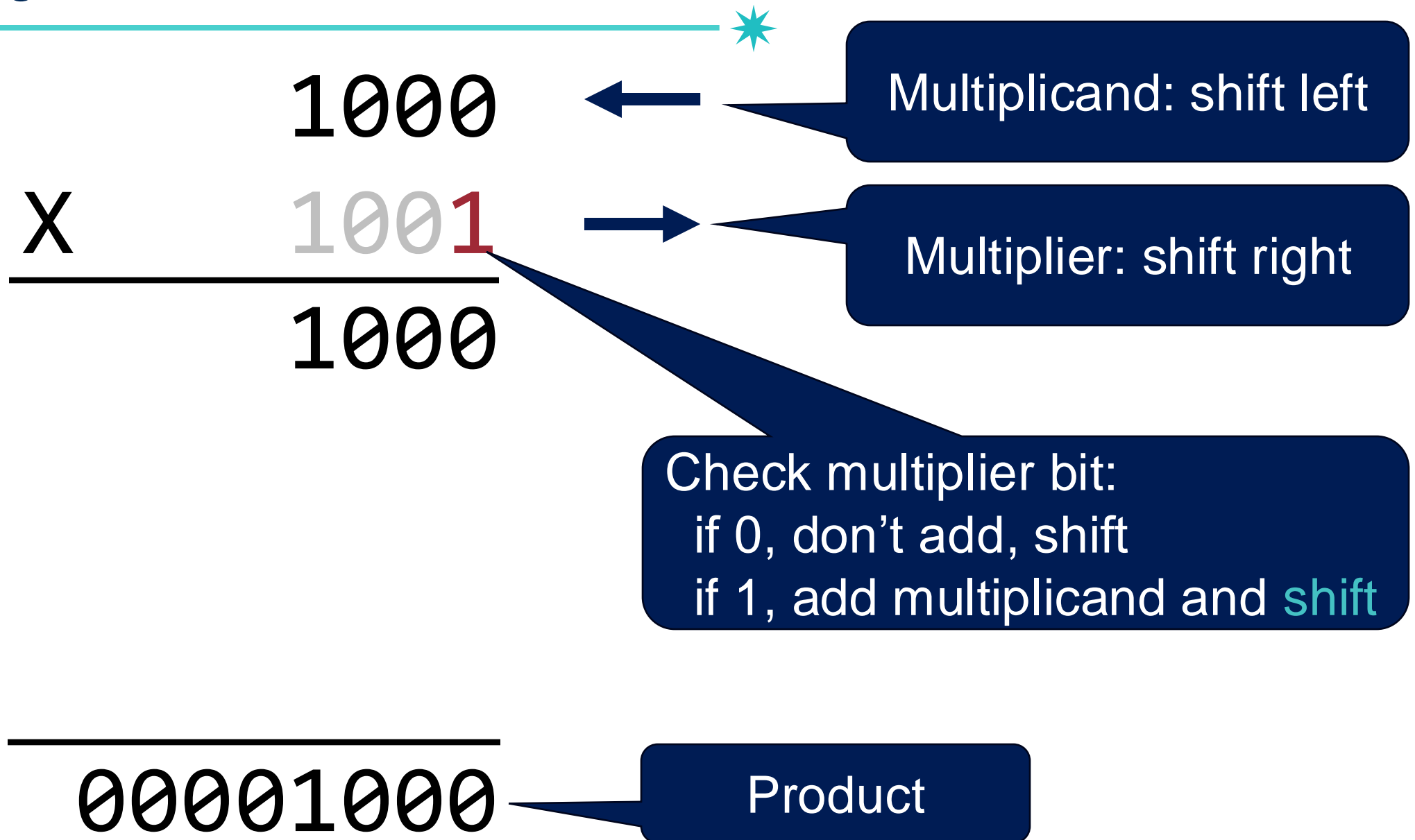
$$\begin{array}{r} 1000 \\ X \quad 1001 \\ \hline \end{array}$$

Check multiplier bit:
if 0, don't add, shift
if 1, add multiplicand and shift

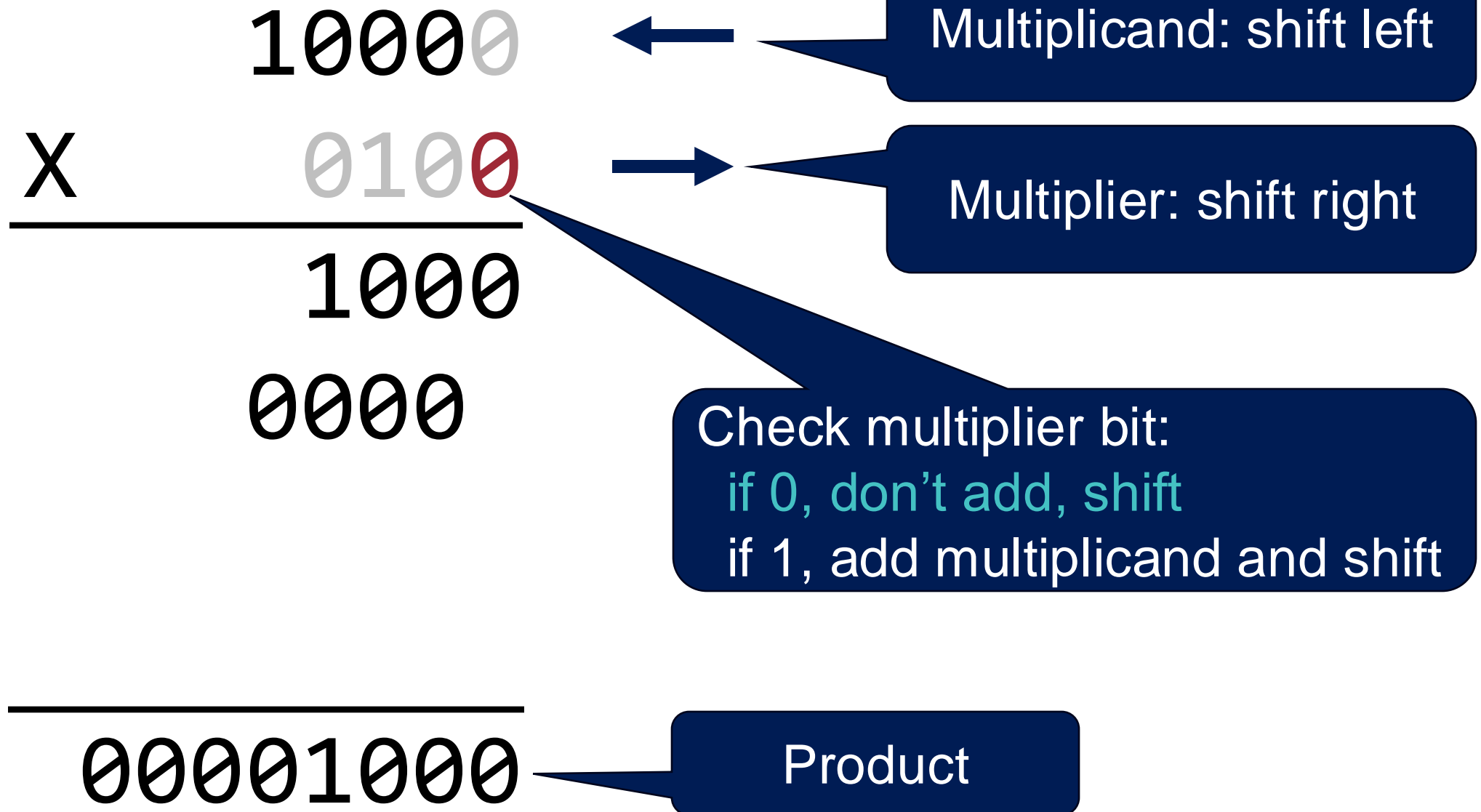
00000000

Product

Shift



Multiplying Two Binary Numbers



Multiplying Two Binary Numbers

10000

X 0010

1000

0000

0000

00001000

← Multiplicand: shift left

→ Multiplier: shift right

Check multiplier bit:
if 0, don't add, shift
if 1, add multiplicand and shift

Product

Multiplying Two Binary Numbers

1000000

X 0001

1000

0000

0000

1000

01001000

← Multiplicand: shift left

→ Multiplier: shift right

Check multiplier bit:
if 0, don't add, shift
if 1, add multiplicand and shift

Product

Final Result

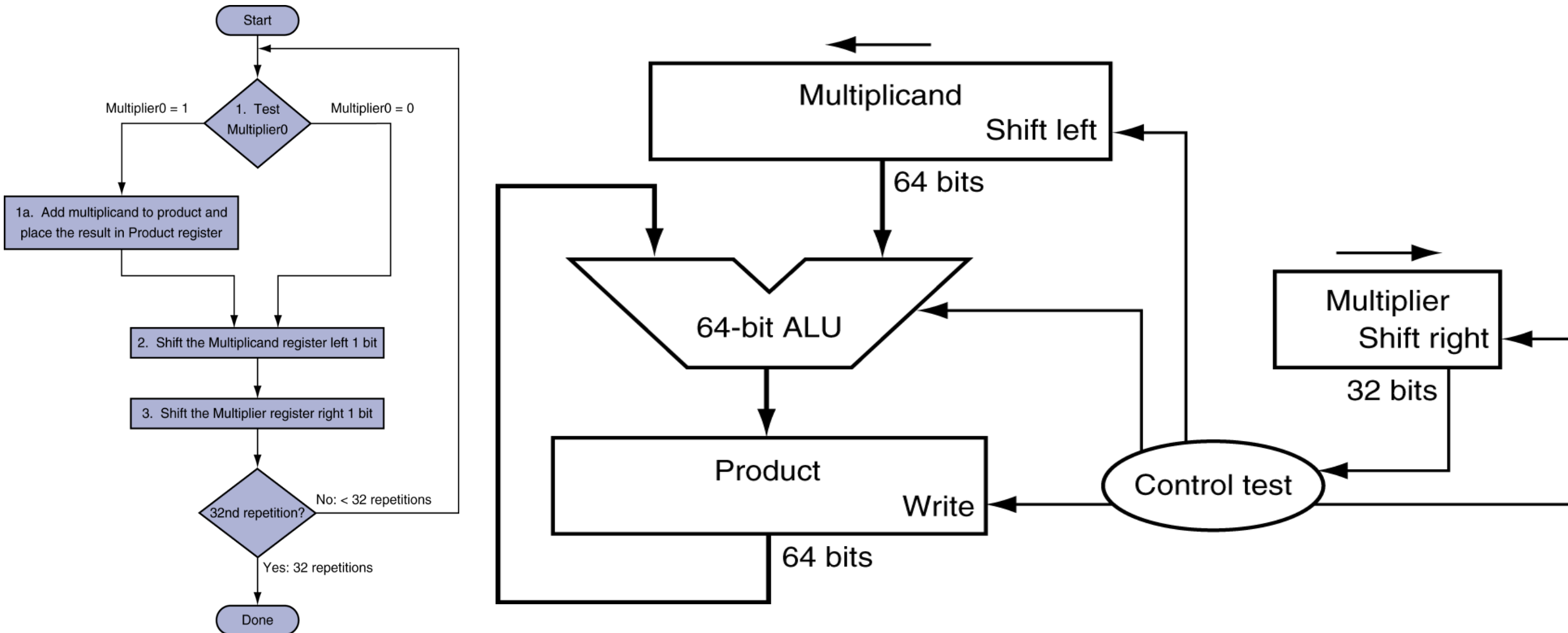


$$\begin{array}{r} 10000000 \\ \times \quad 0000 \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 01001000 \end{array}$$

**Now, let's look at the
multiplication hardware!**

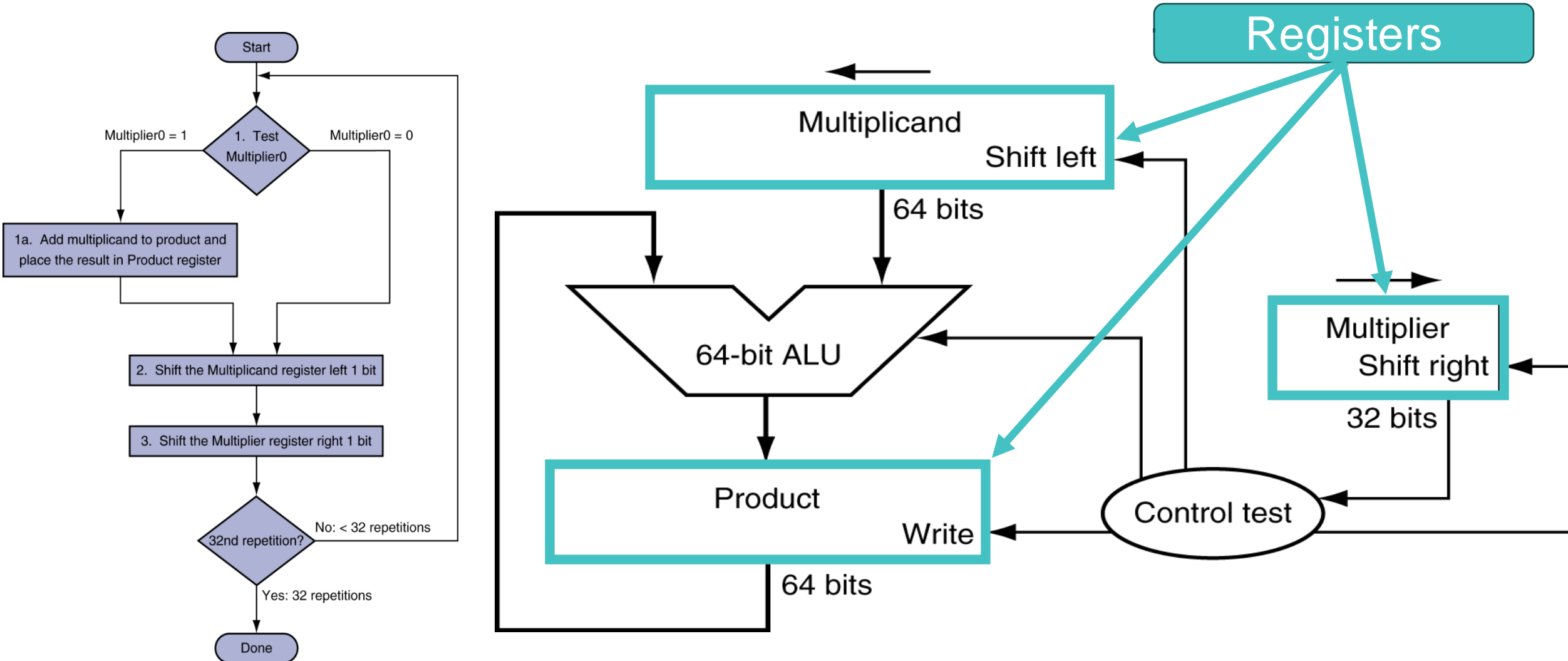
Multiplication Hardware

40



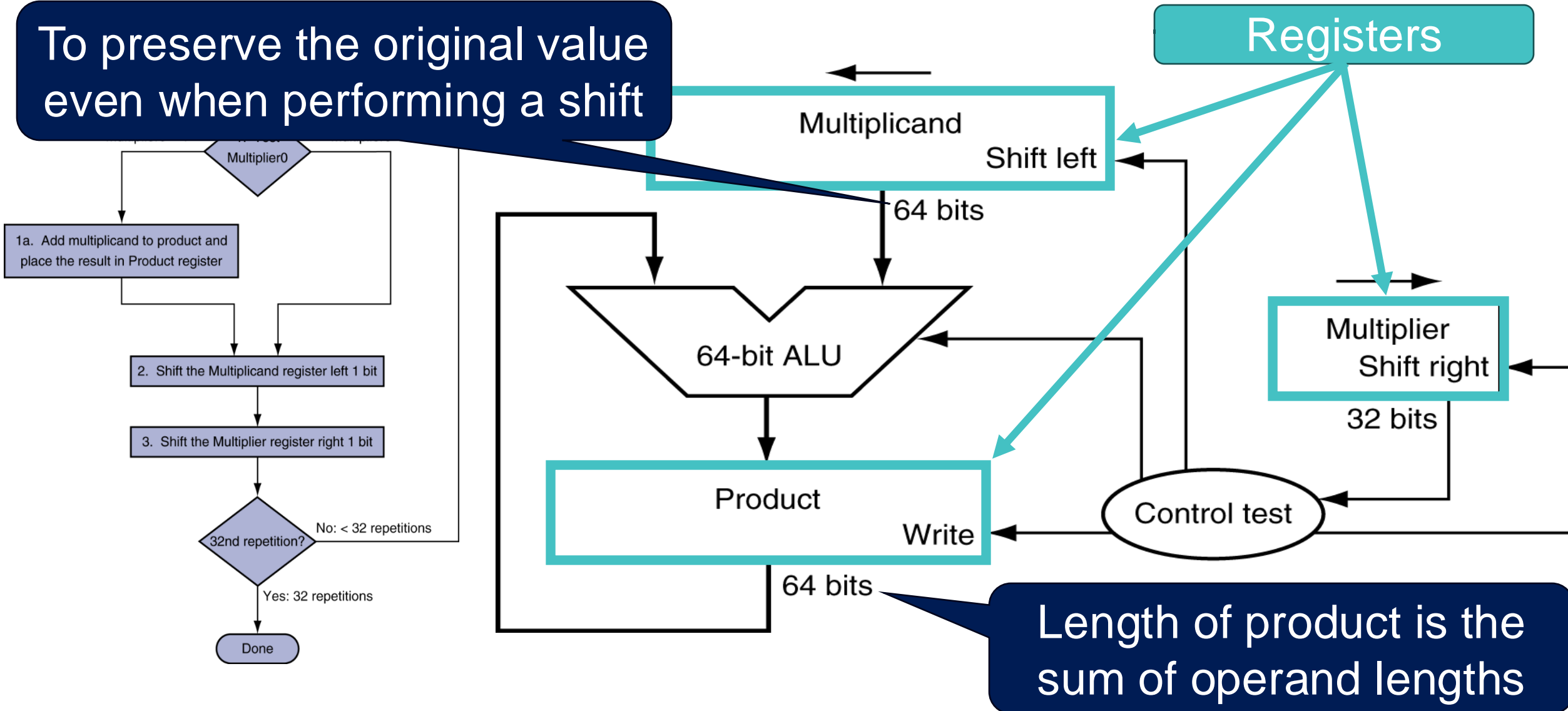
Multiplication Hardware

41



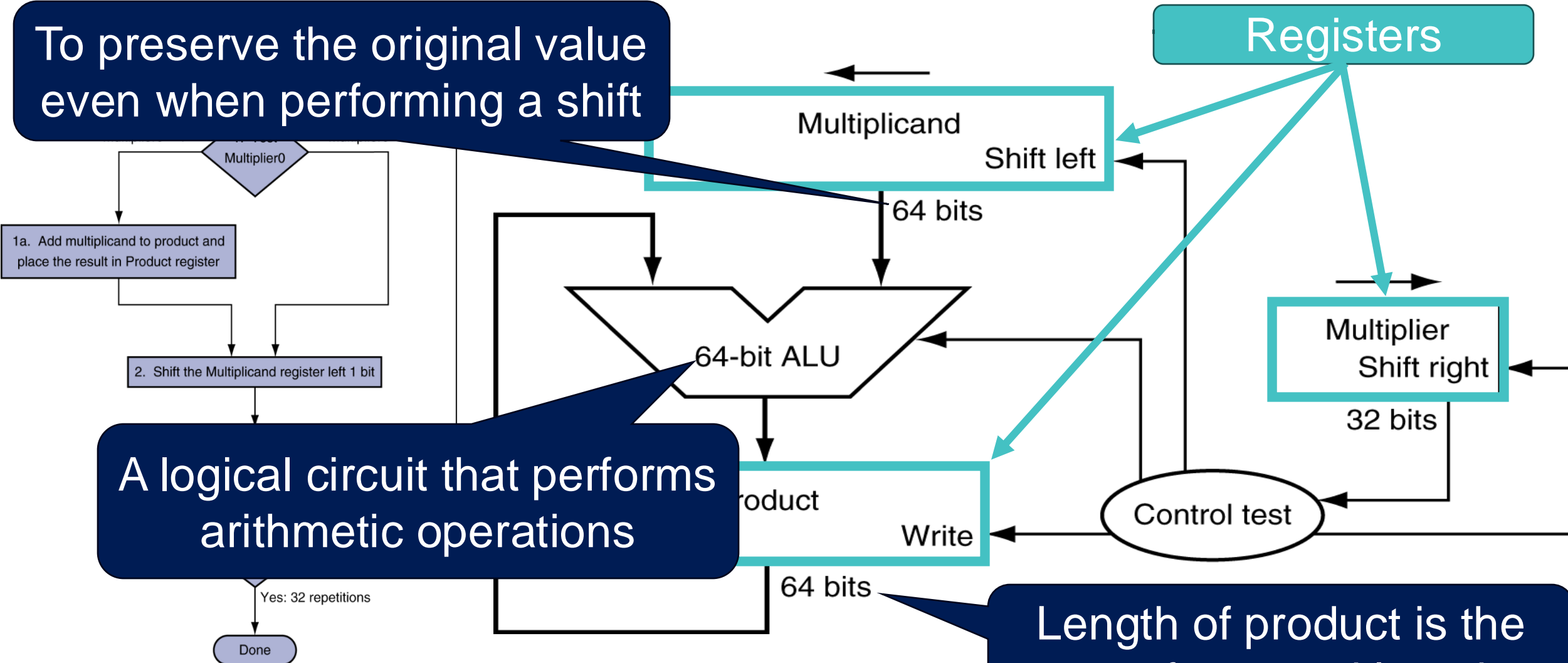
Multiplication Hardware

To preserve the original value even when performing a shift



Multiplication Hardware

To preserve the original value even when performing a shift





Example: 1st Iteration – Test Multiplier Bit

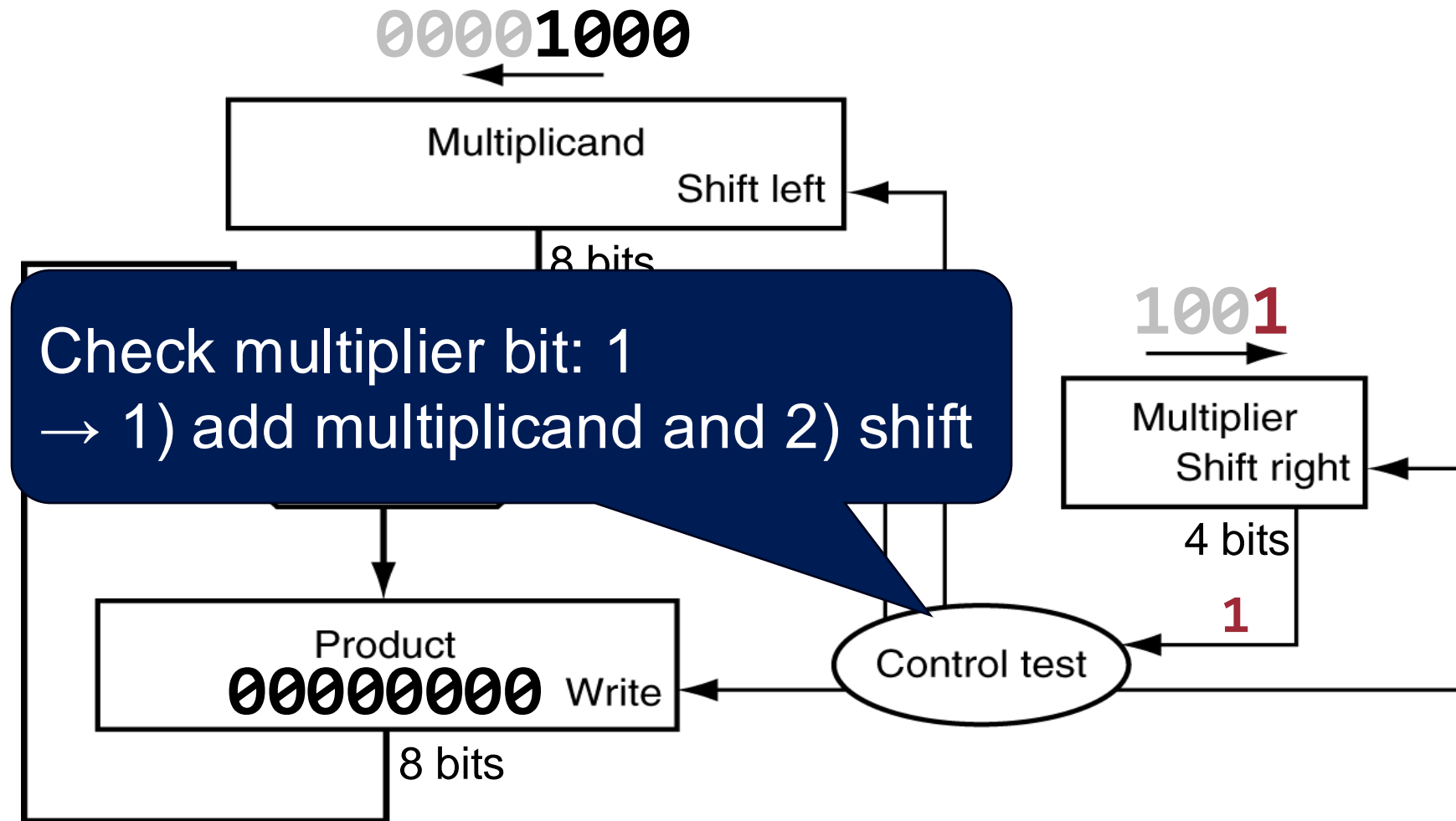
15

Multiplicand 1000

X Multiplier 1001

00000000

Product



1st iteration

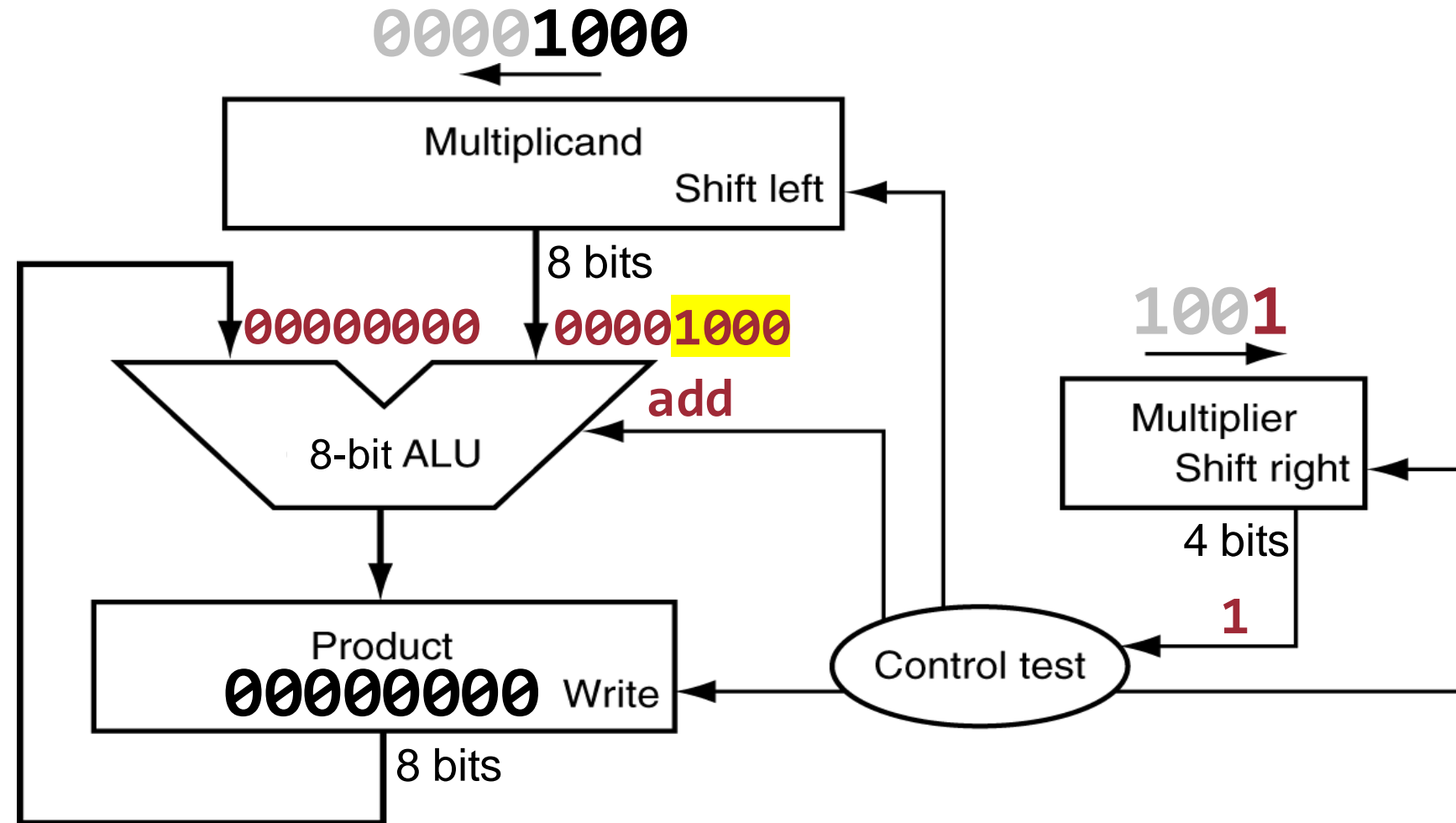
Example: 1st Iteration – Before Addition

46

Multiplicand 1000
X Multiplier 1001

1000

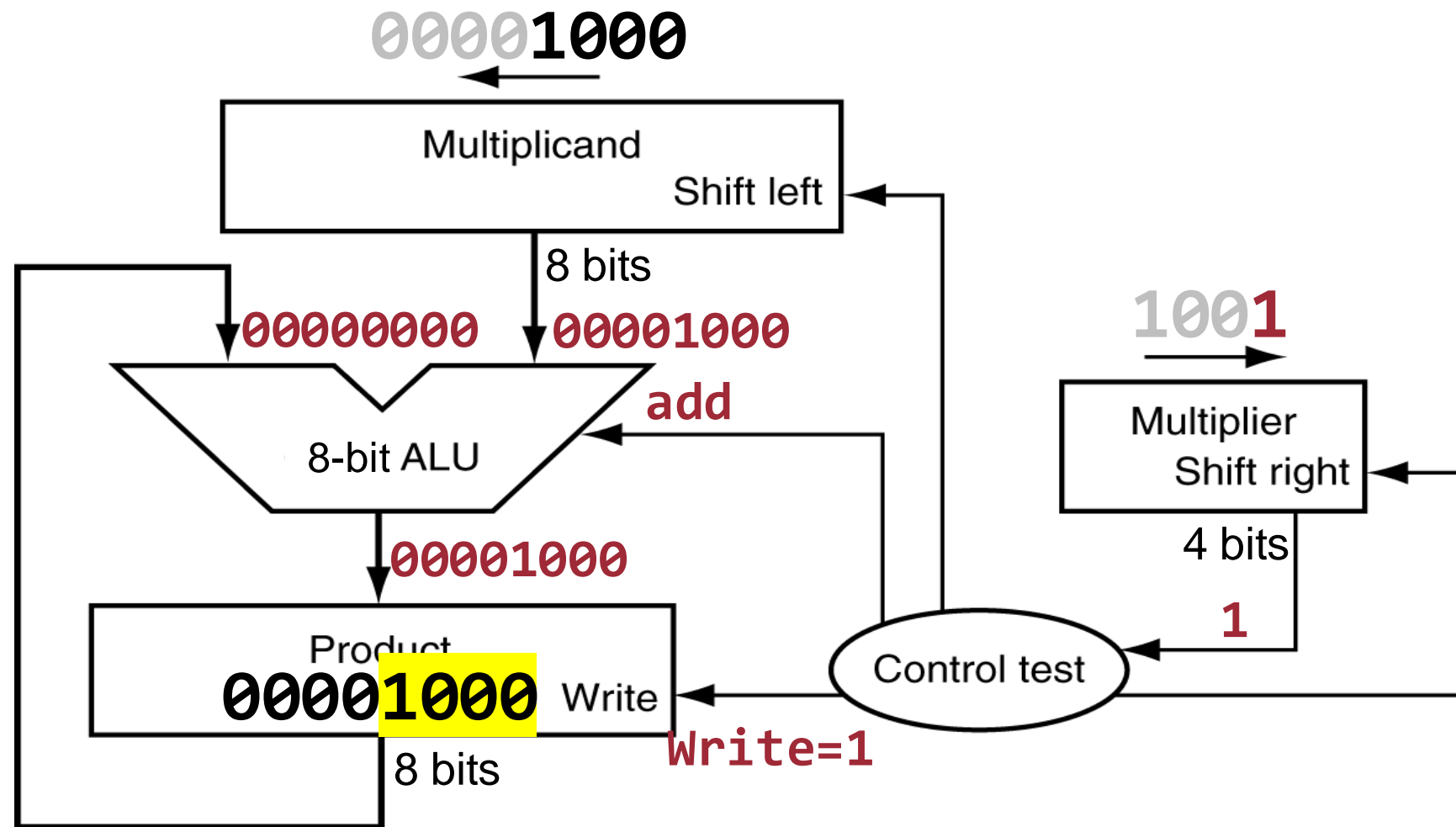
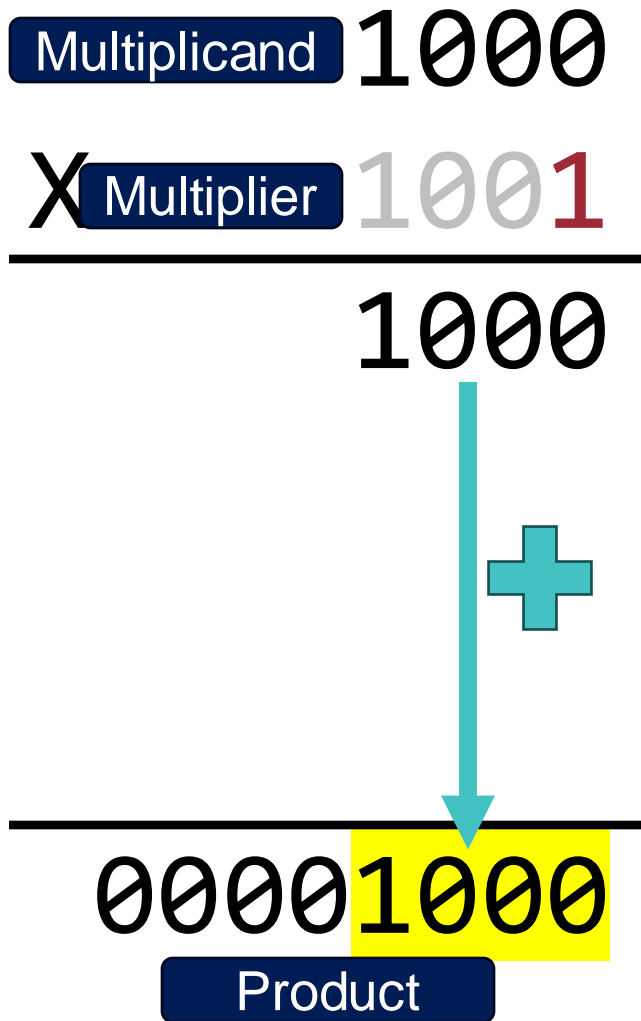
00000000
Product



1st iteration

Example: 1st Iteration – After Addition

47



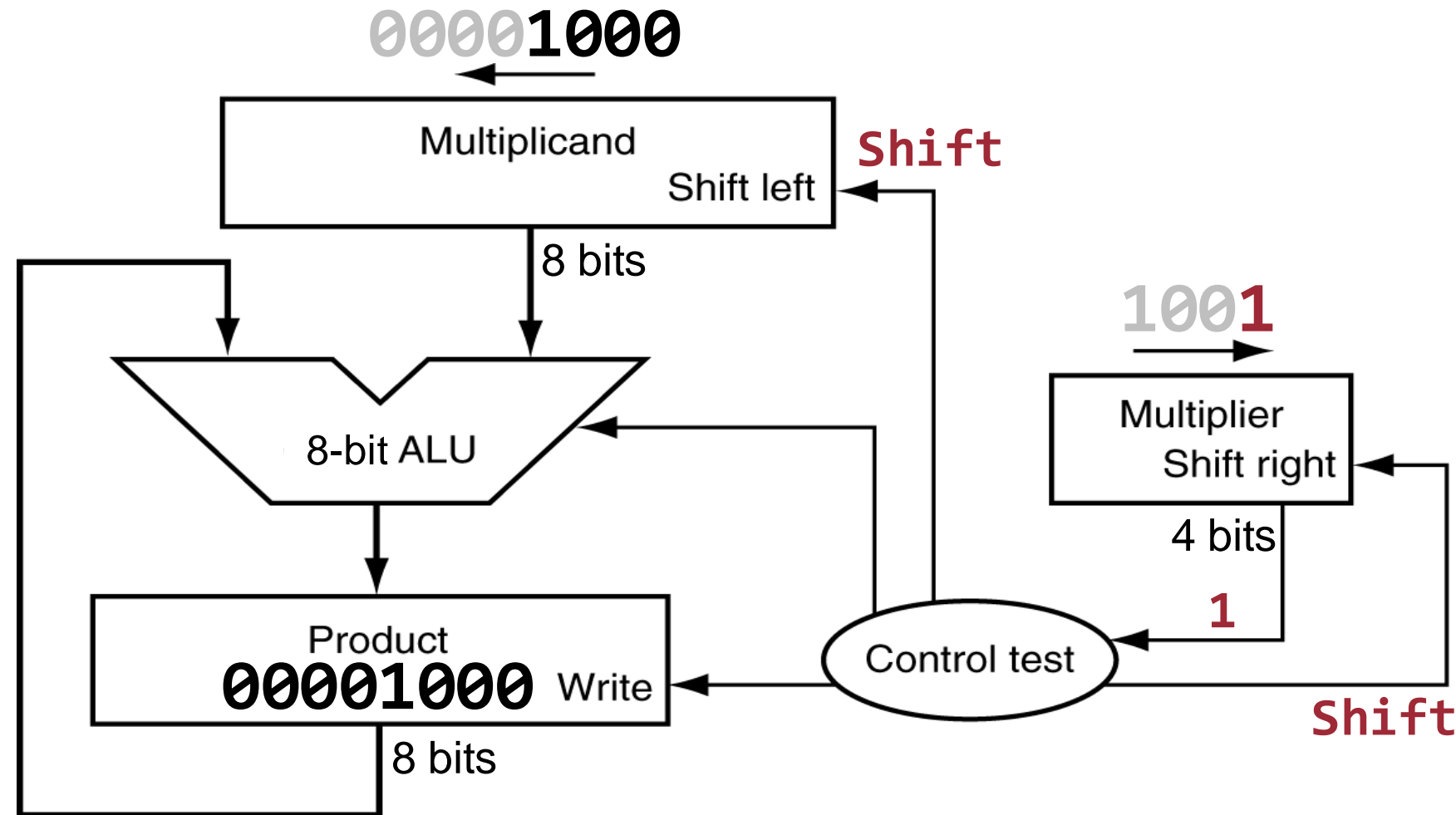
1st iteration

Example: 1st Iteration – Before Shift

Multiplicand **1000**
 X Multiplier **1001**

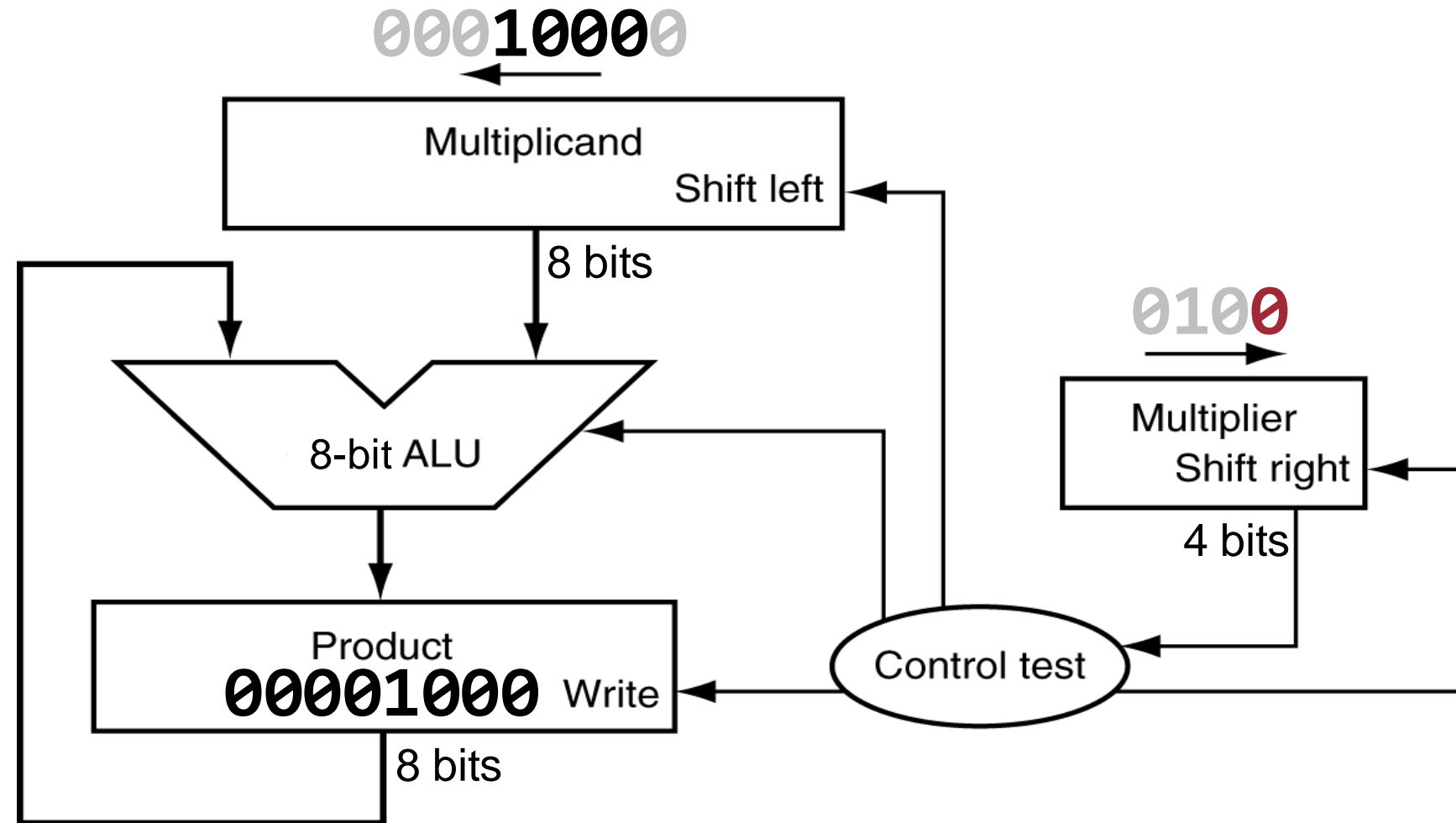
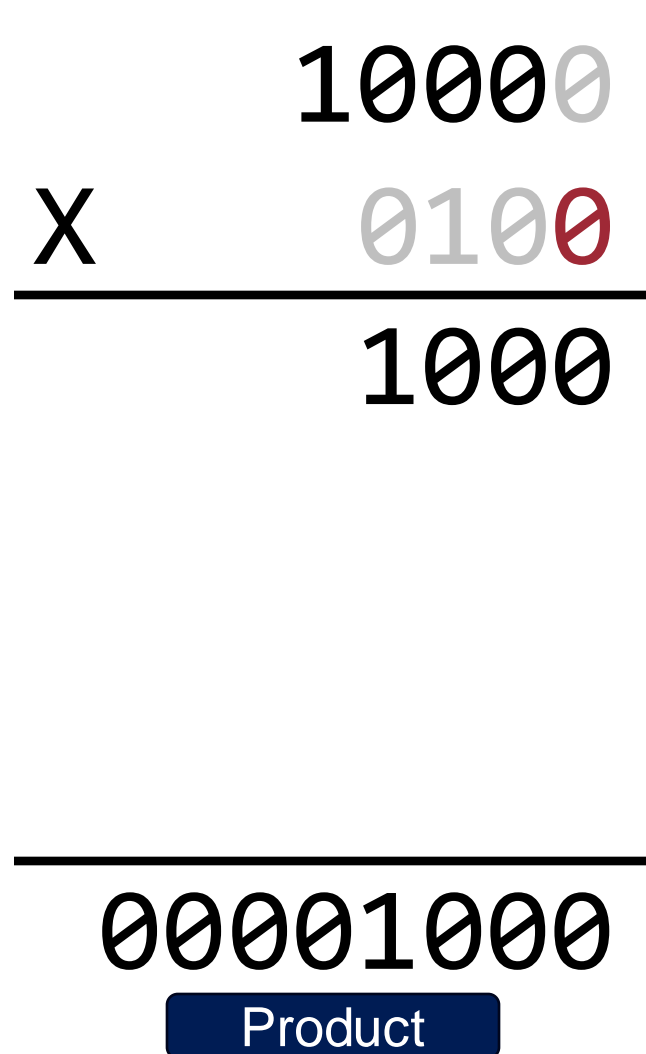
 1000

 00001000
 Product



1st iteration

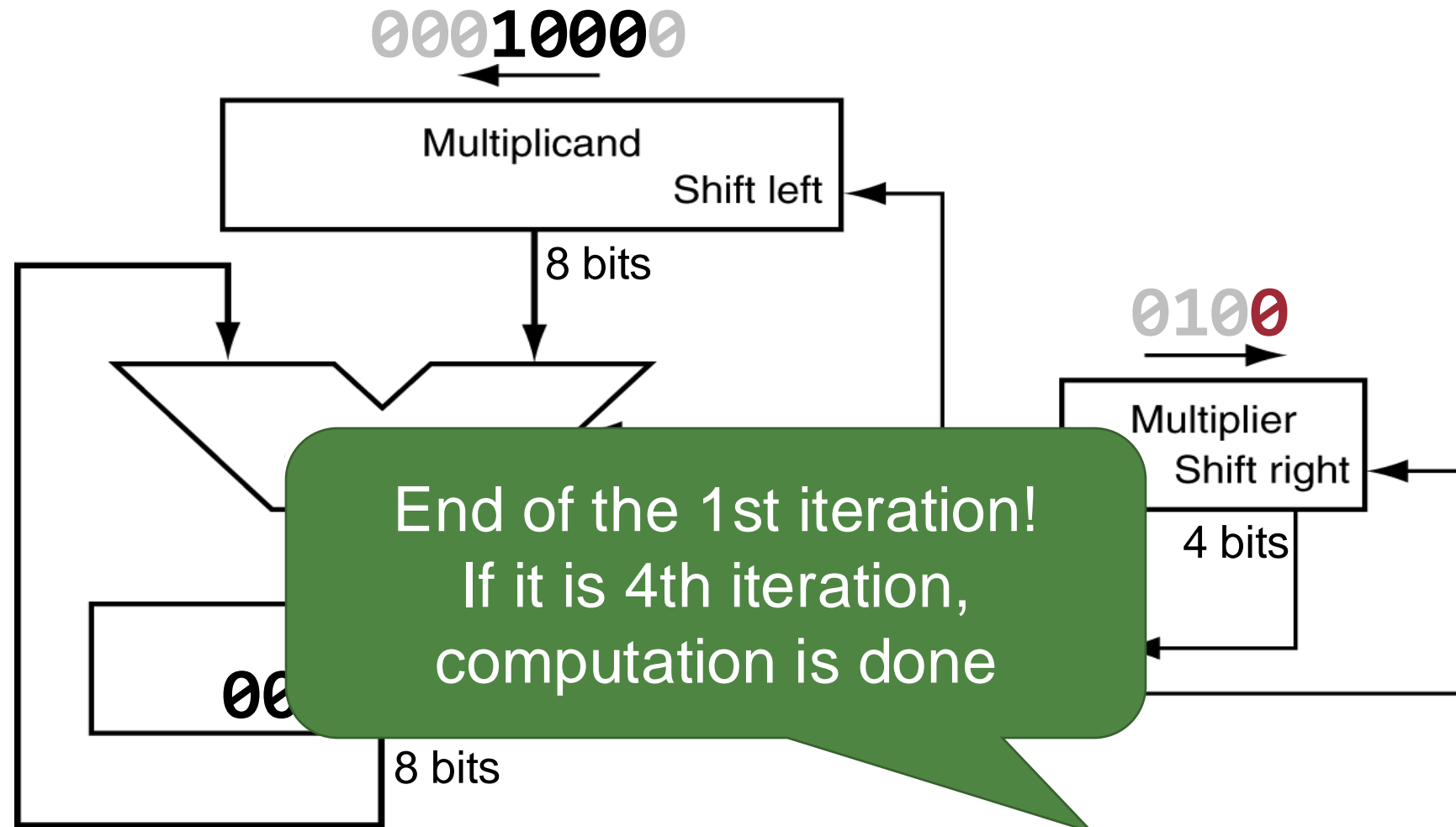
Example: 1st Iteration – After Shift



1st iteration

Example: 1st Iteration – Check Iteration #50

$$\begin{array}{r} X \quad 10000 \\ \underline{0100} \\ 1000 \\ \hline 00001000 \\ \text{Product} \end{array}$$



1st iteration

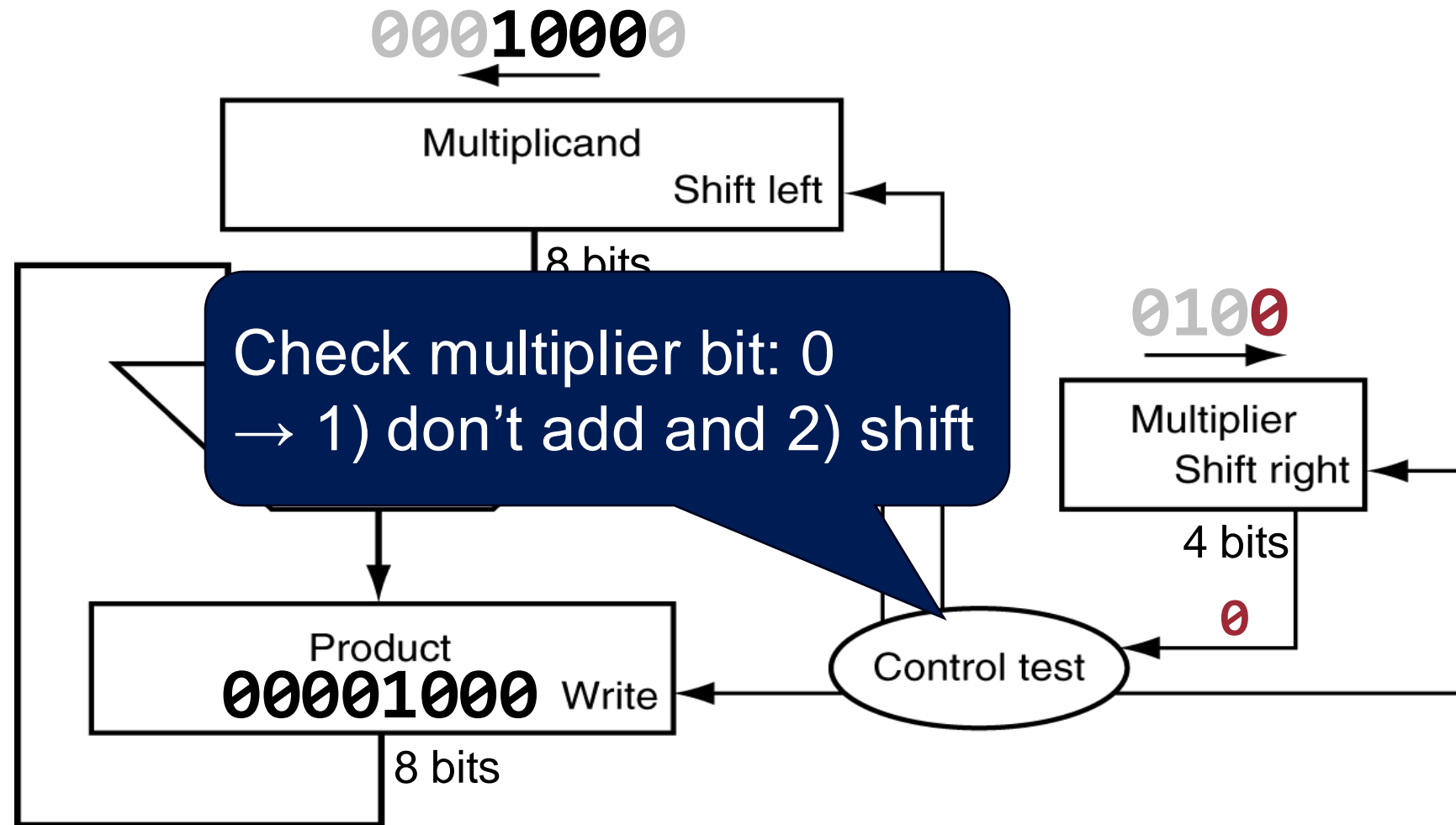
Example: 2nd Iteration – Test Multiplier Bit

X 10000
 0100

 1000

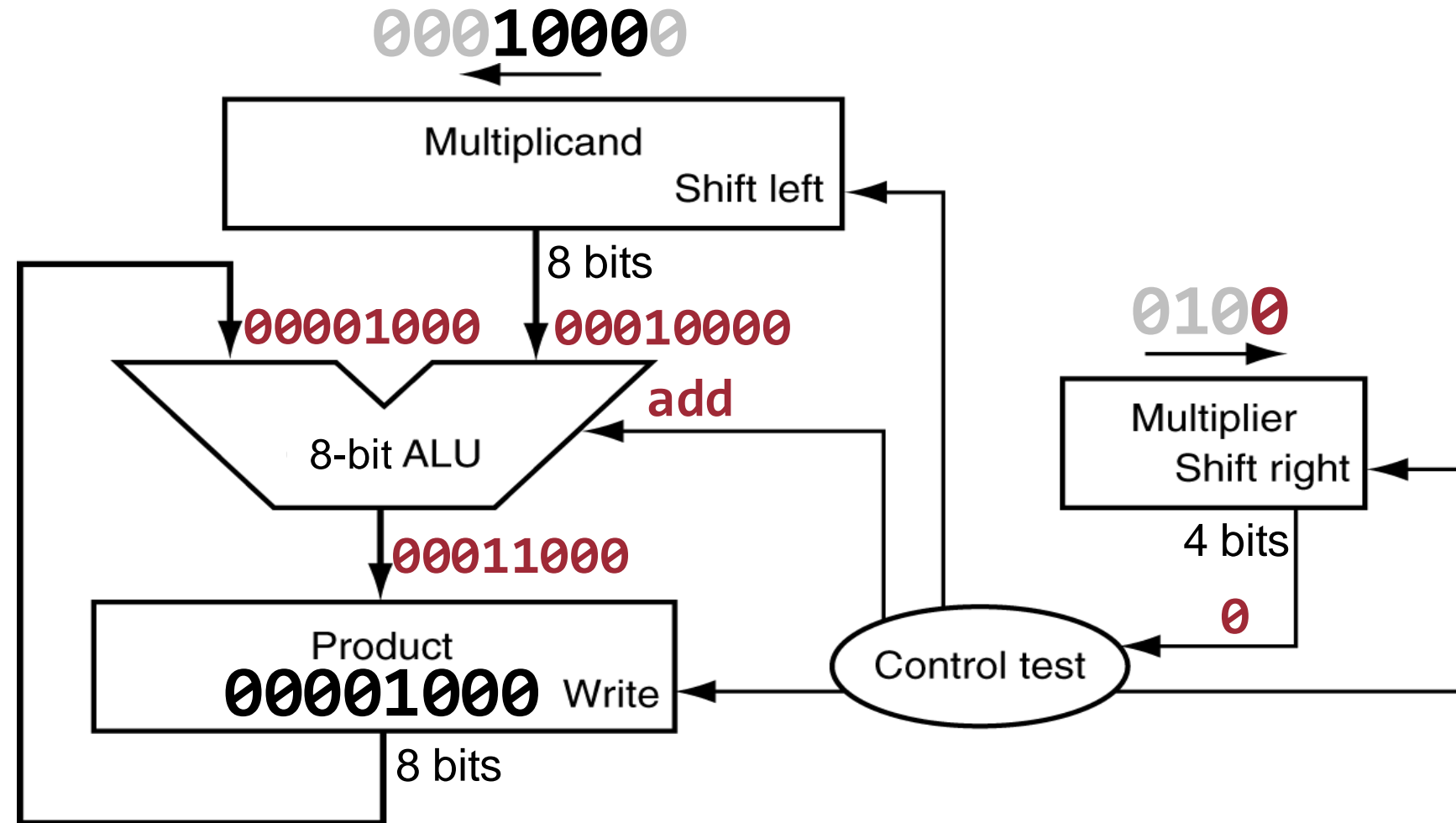
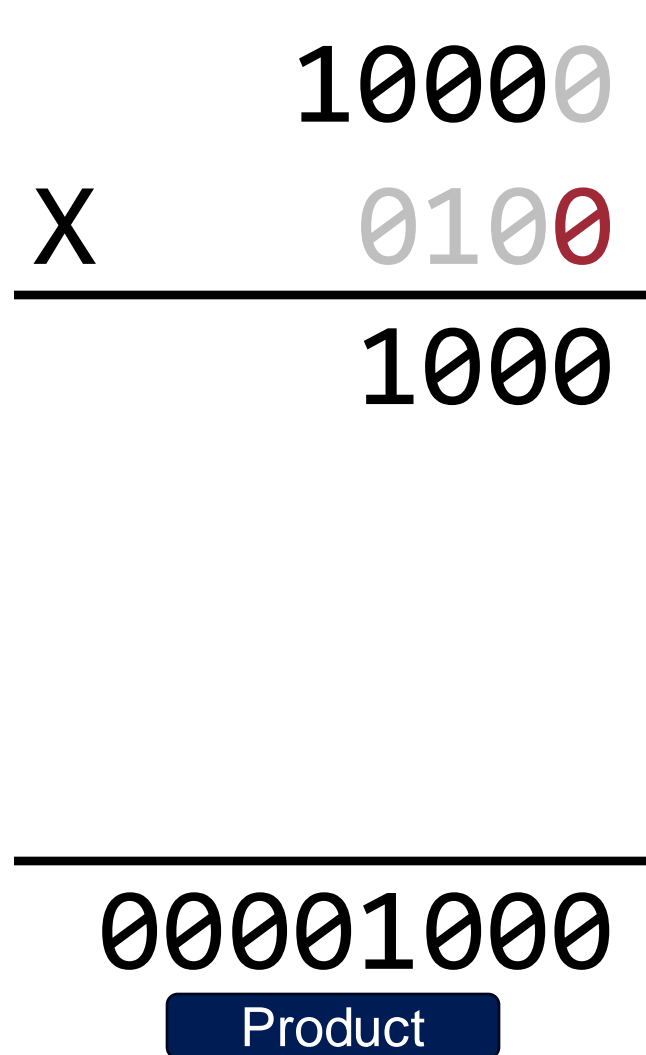
00001000

Product



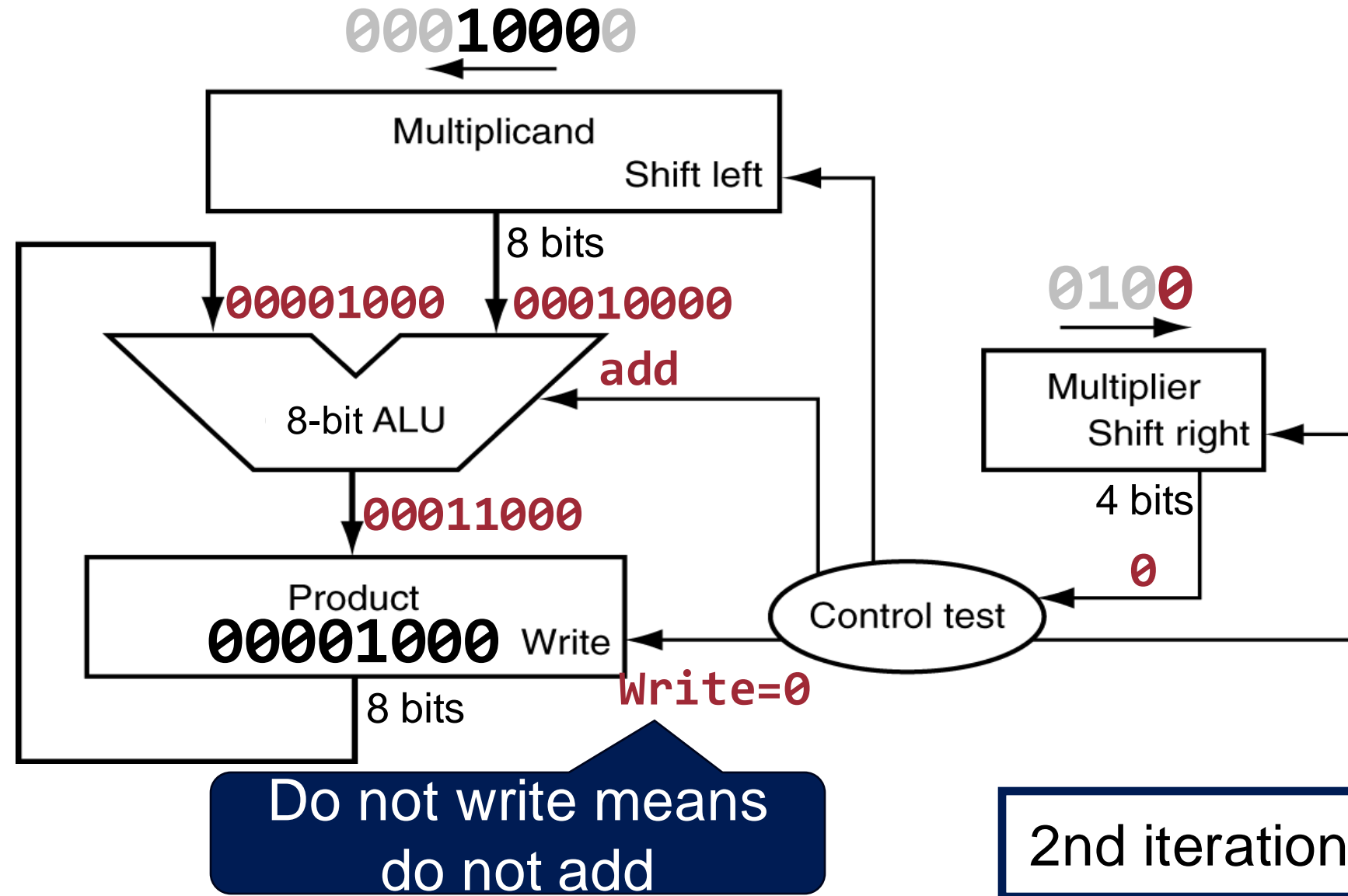
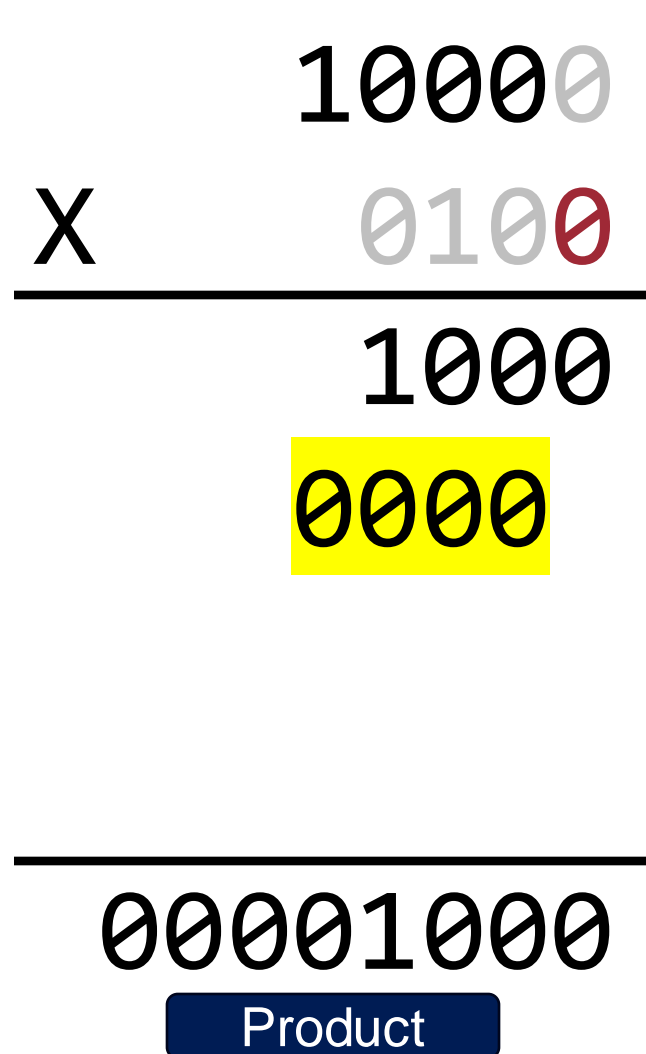
2nd iteration

Example: 2nd Iteration – Don't Add

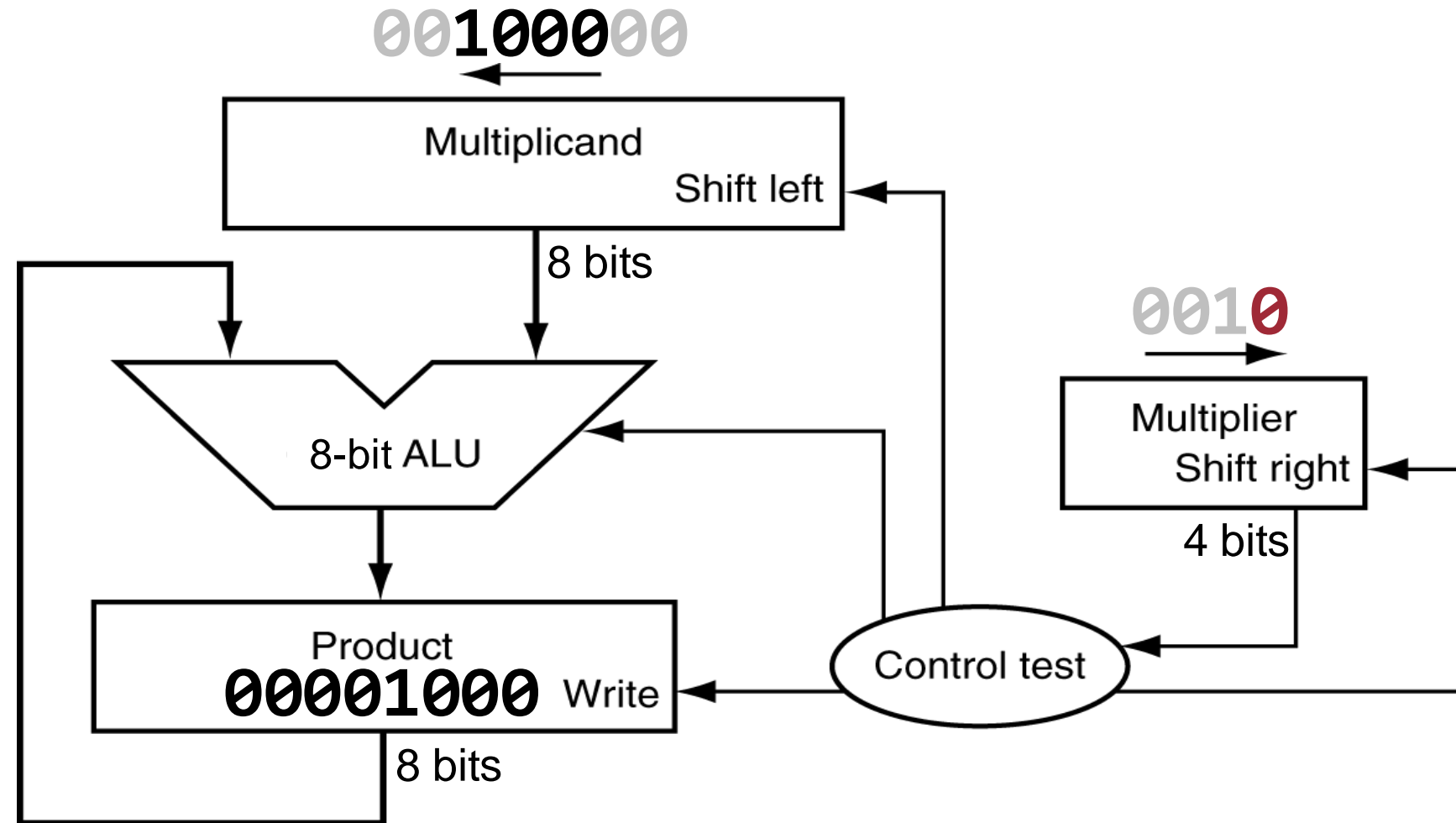
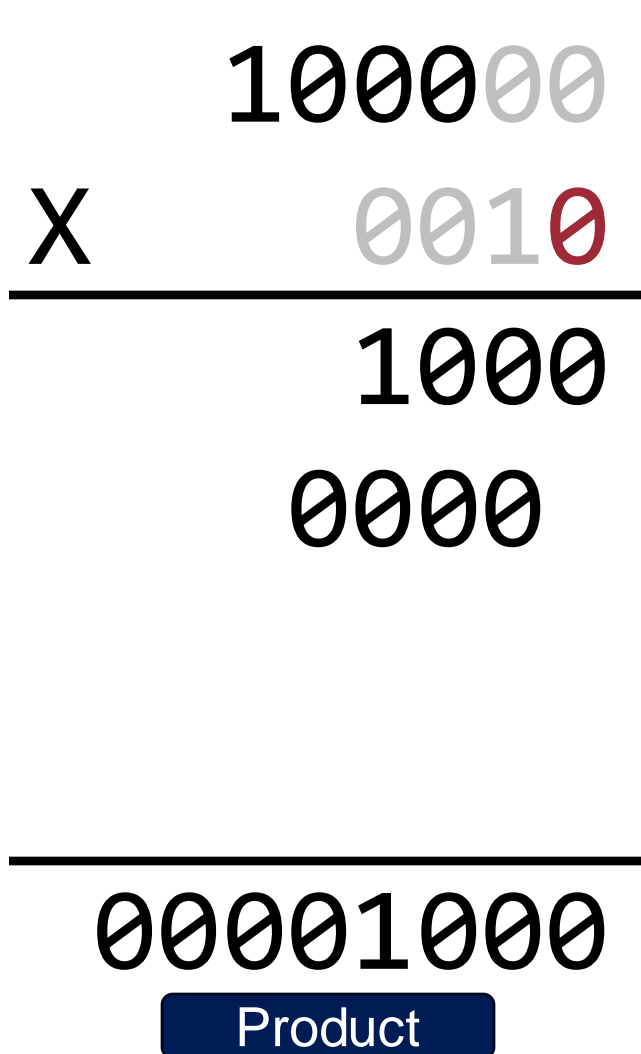


2nd iteration

Example: 2nd Iteration – Don't Add



Example: 2nd Iteration – After Shift



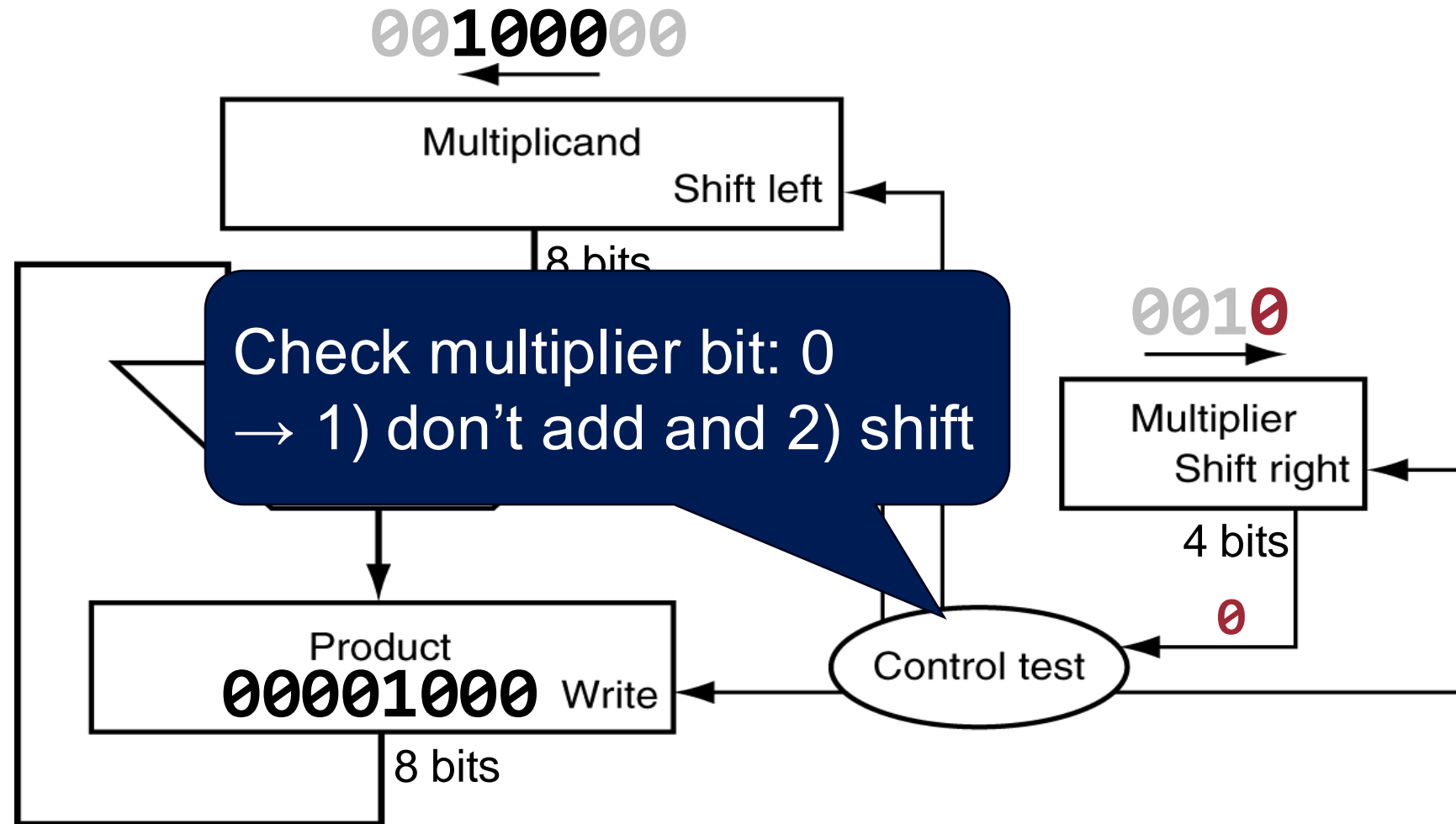
2nd iteration

Example: 3rd Iteration – Test Multiplier Bit

X 100000
 0010

 1000
 0000

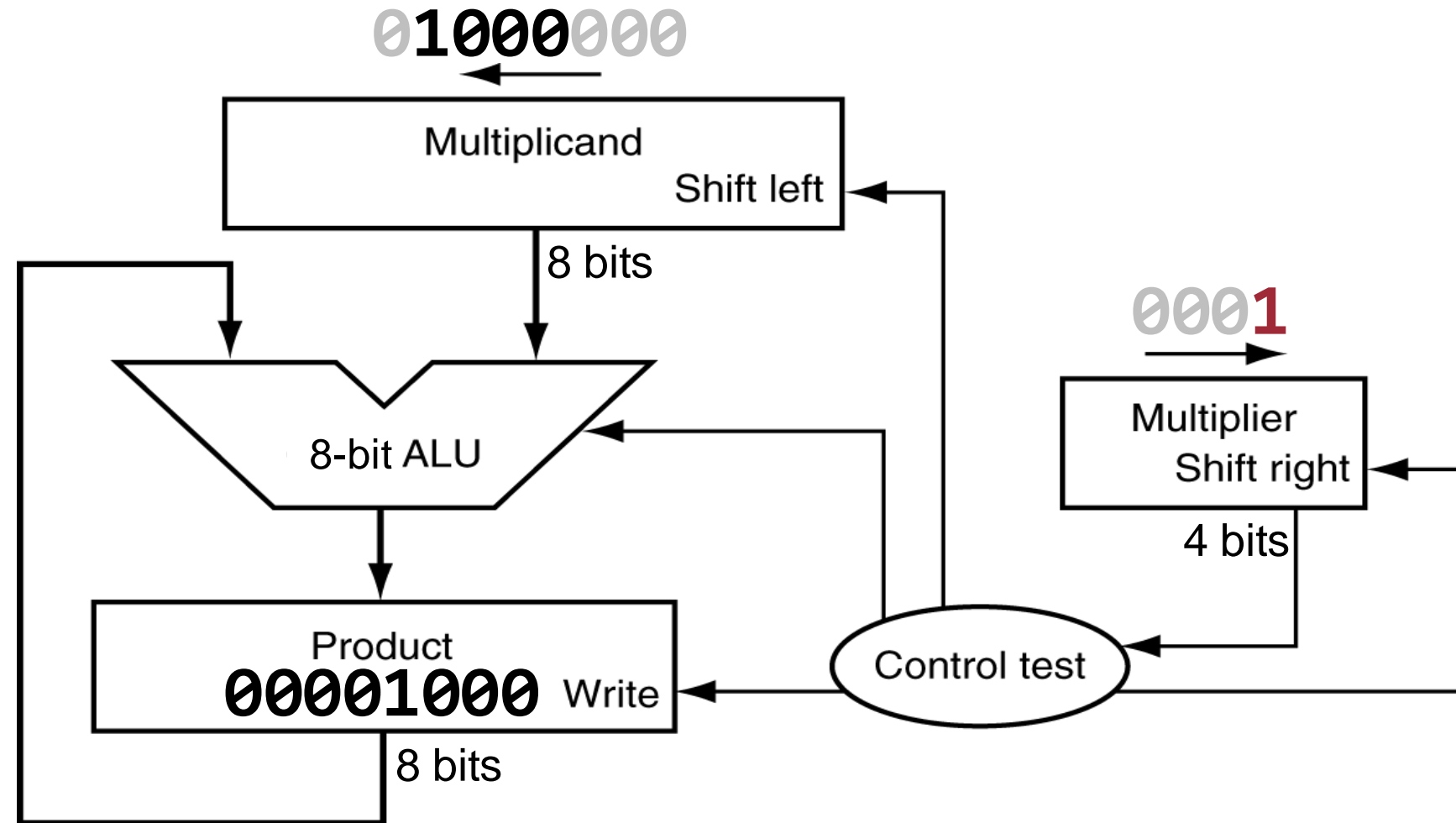
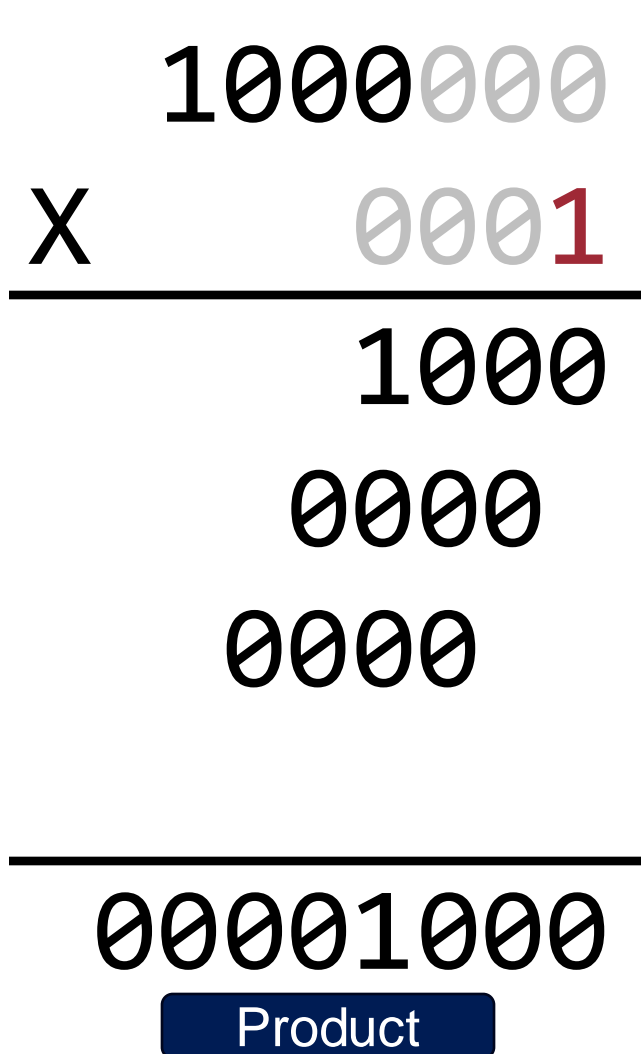
00001000
Product



3rd iteration

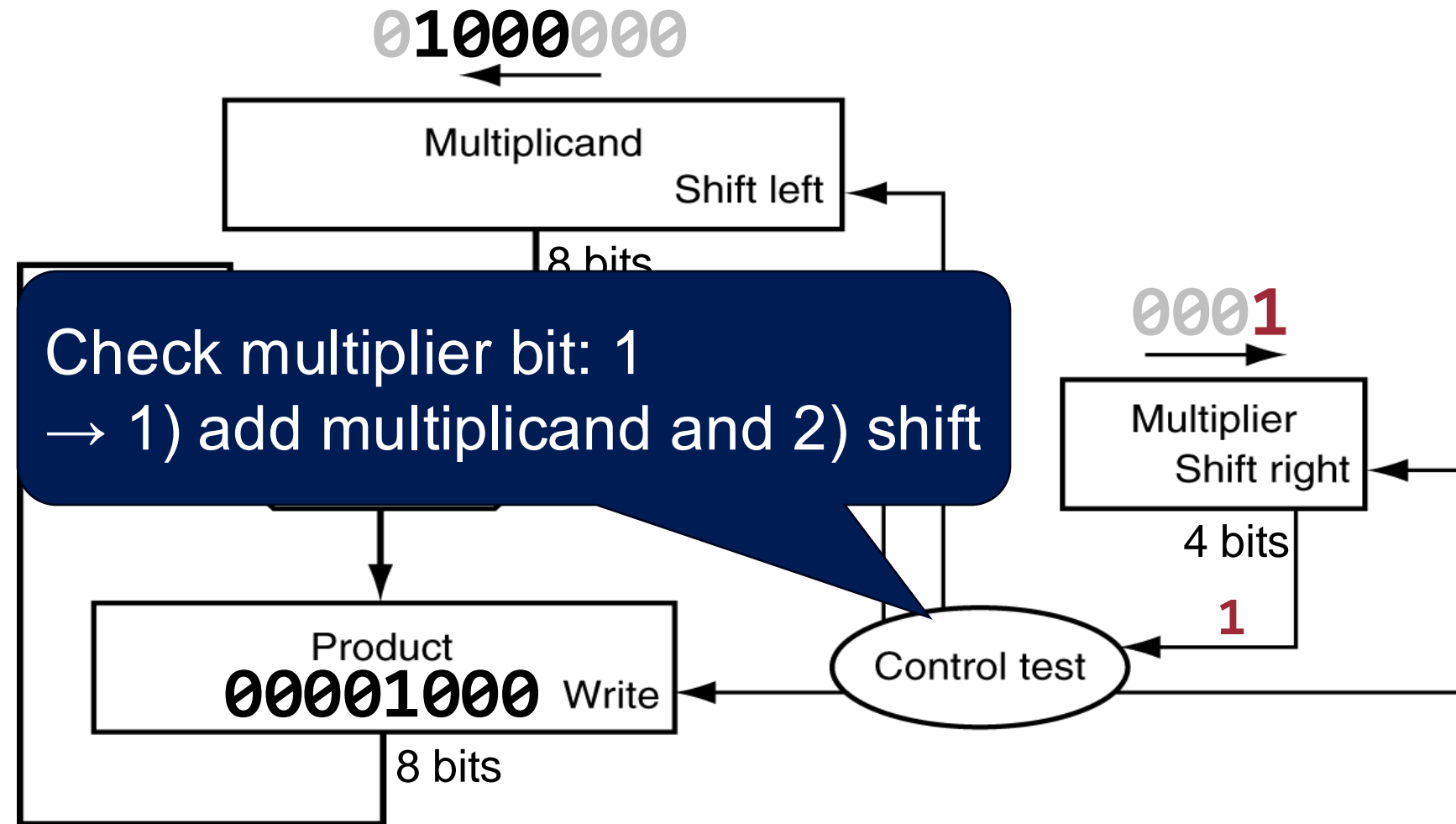
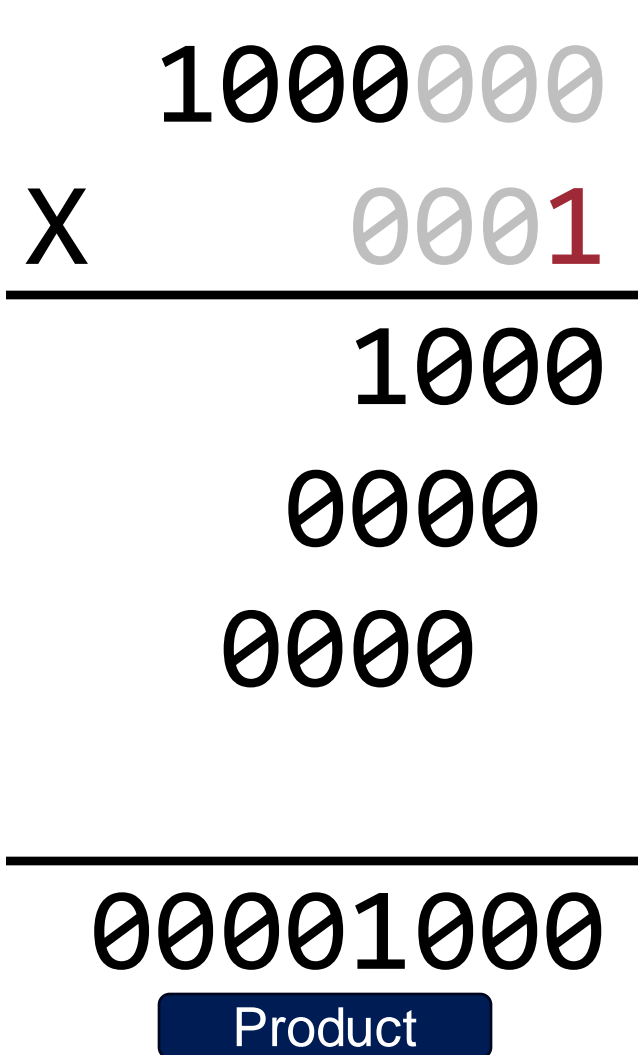


Example: 3rd Iteration – After Shift



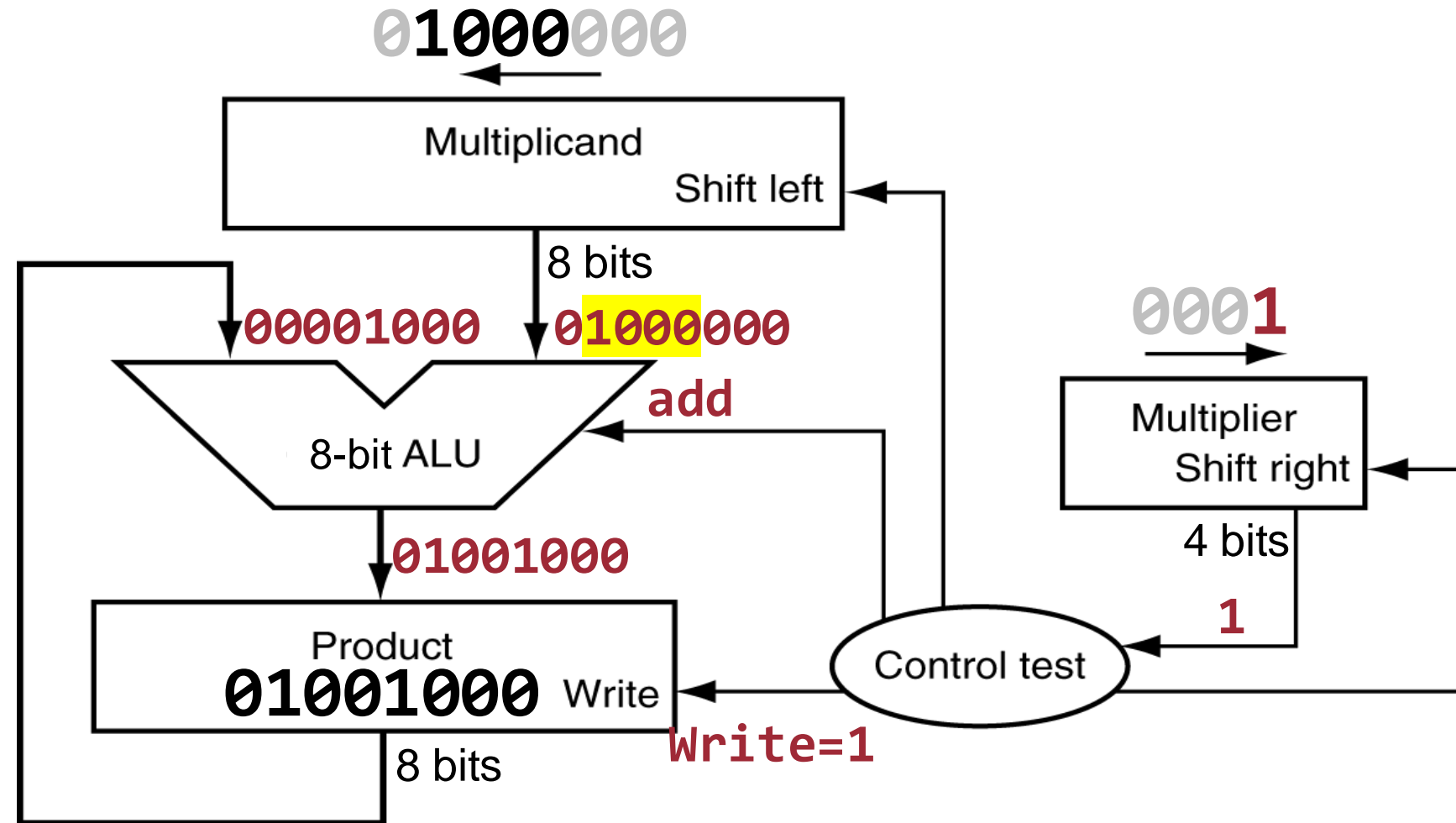
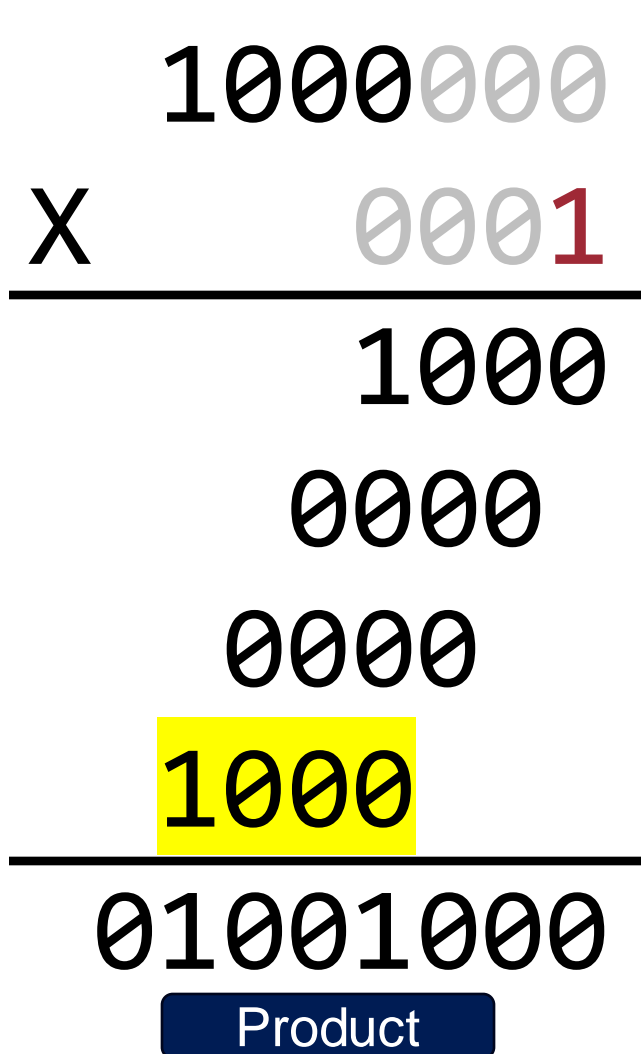
3rd iteration

Example: 4th Iteration – Test Multiplier Bit



4th iteration

Example: 4th Iteration – After Addition



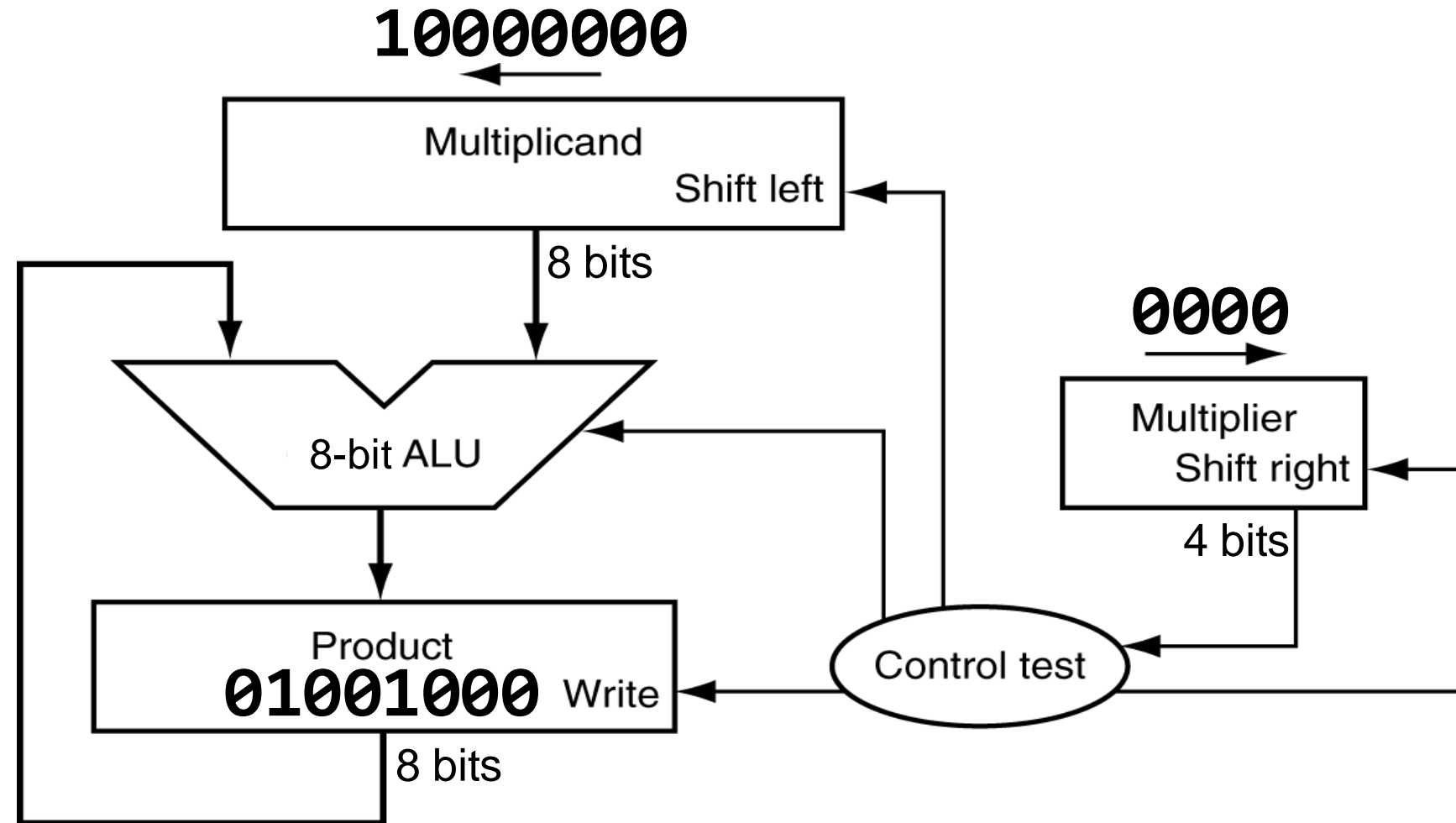
4th iteration

Example: 4th Iteration – After Shift

10000000
 X 0000

 1000
 0000
 0000
 1000

 01001000
 Product

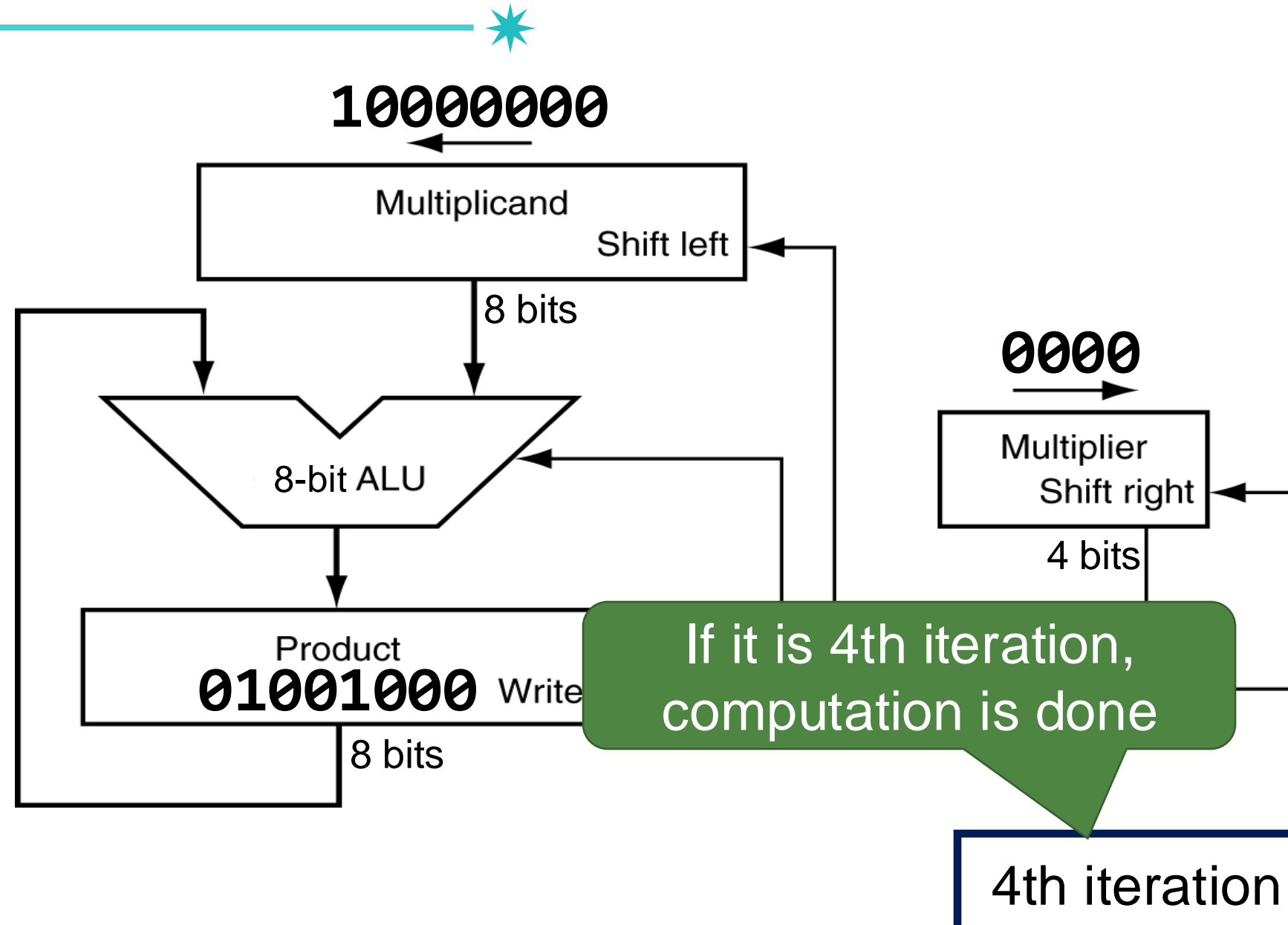


4th iteration

Example: Final Result

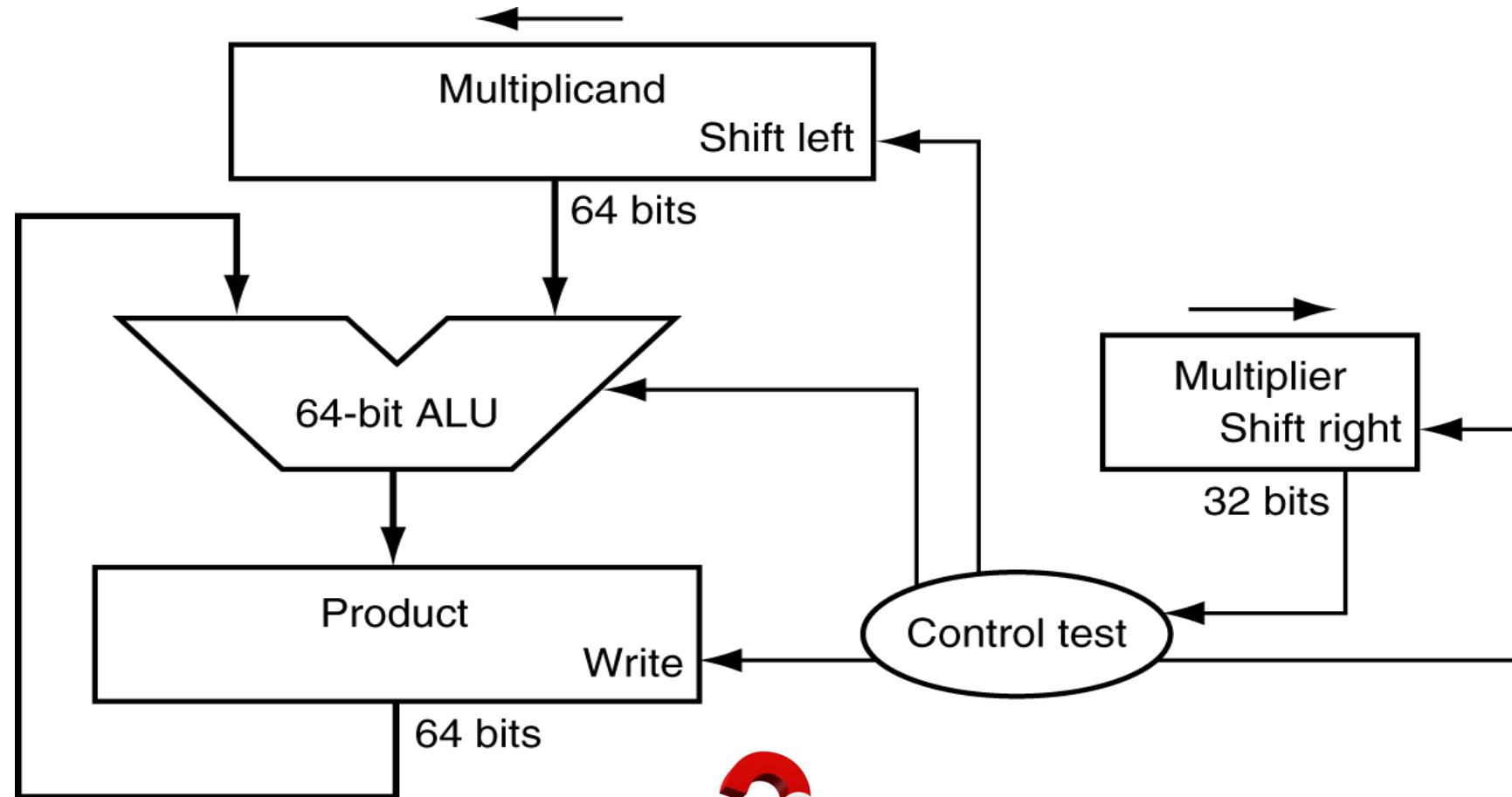
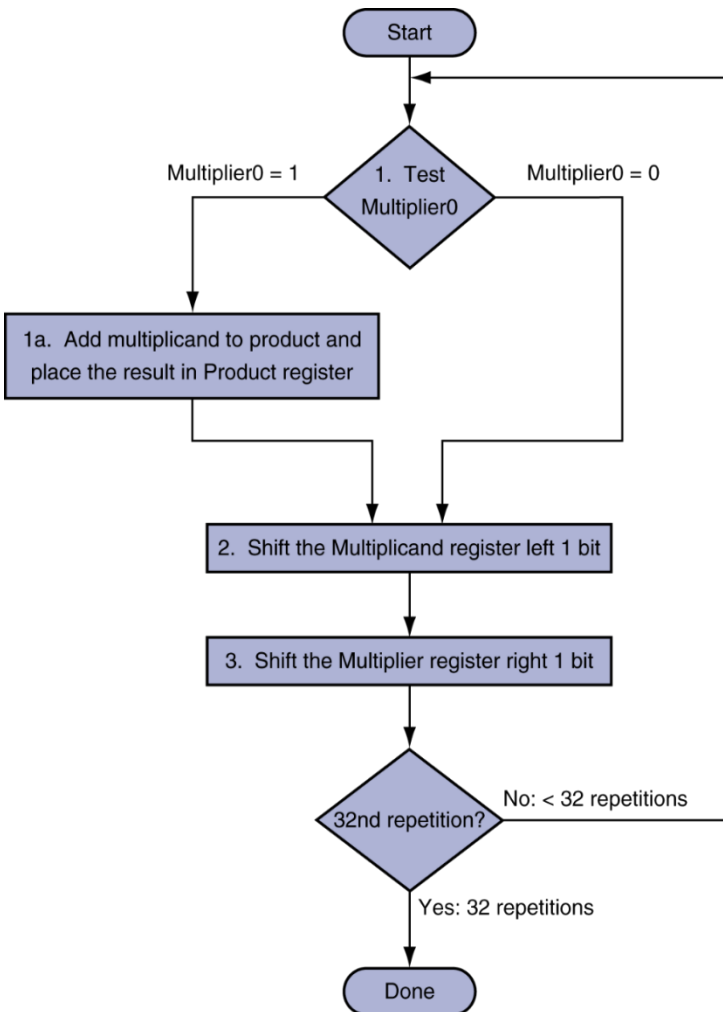
$$\begin{array}{r}
 10000000 \\
 \times \quad 0000 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$

Product



Multiplication Hardware: Algorithm Summary

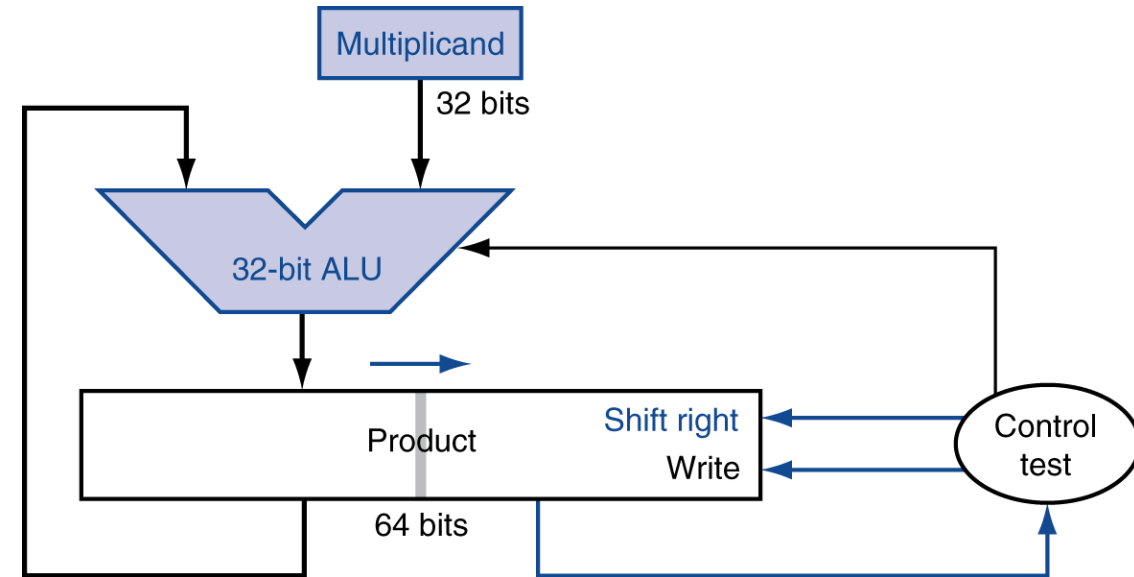
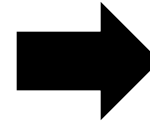
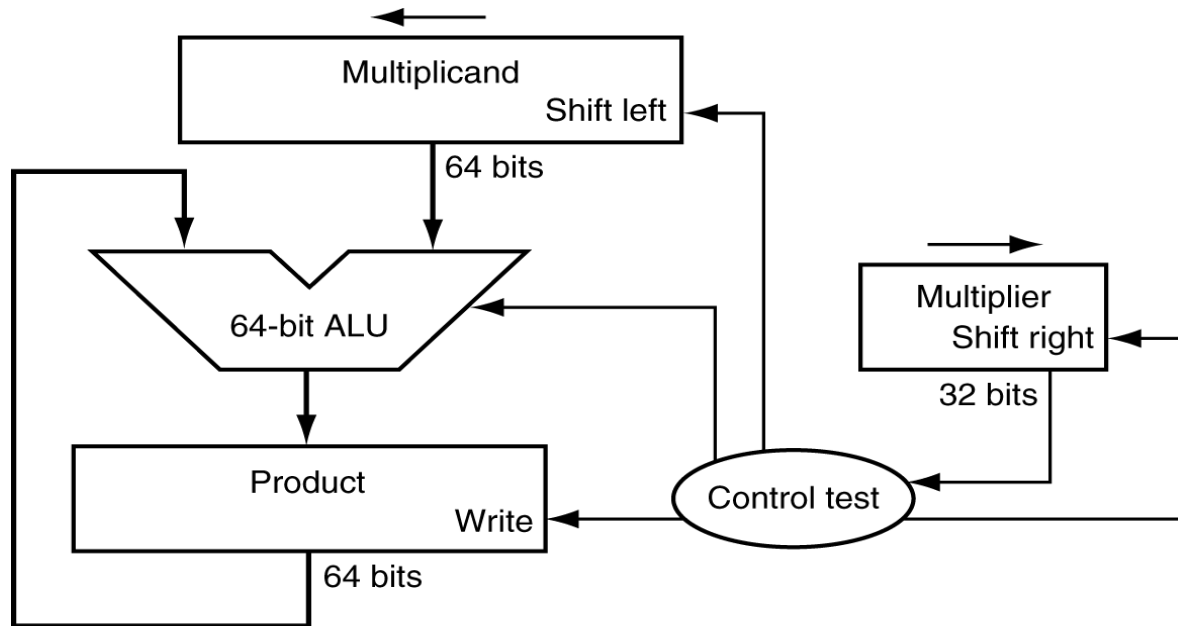
62



*Can we optimize this
Multiplication hardware?*

Optimized Multiplier

63

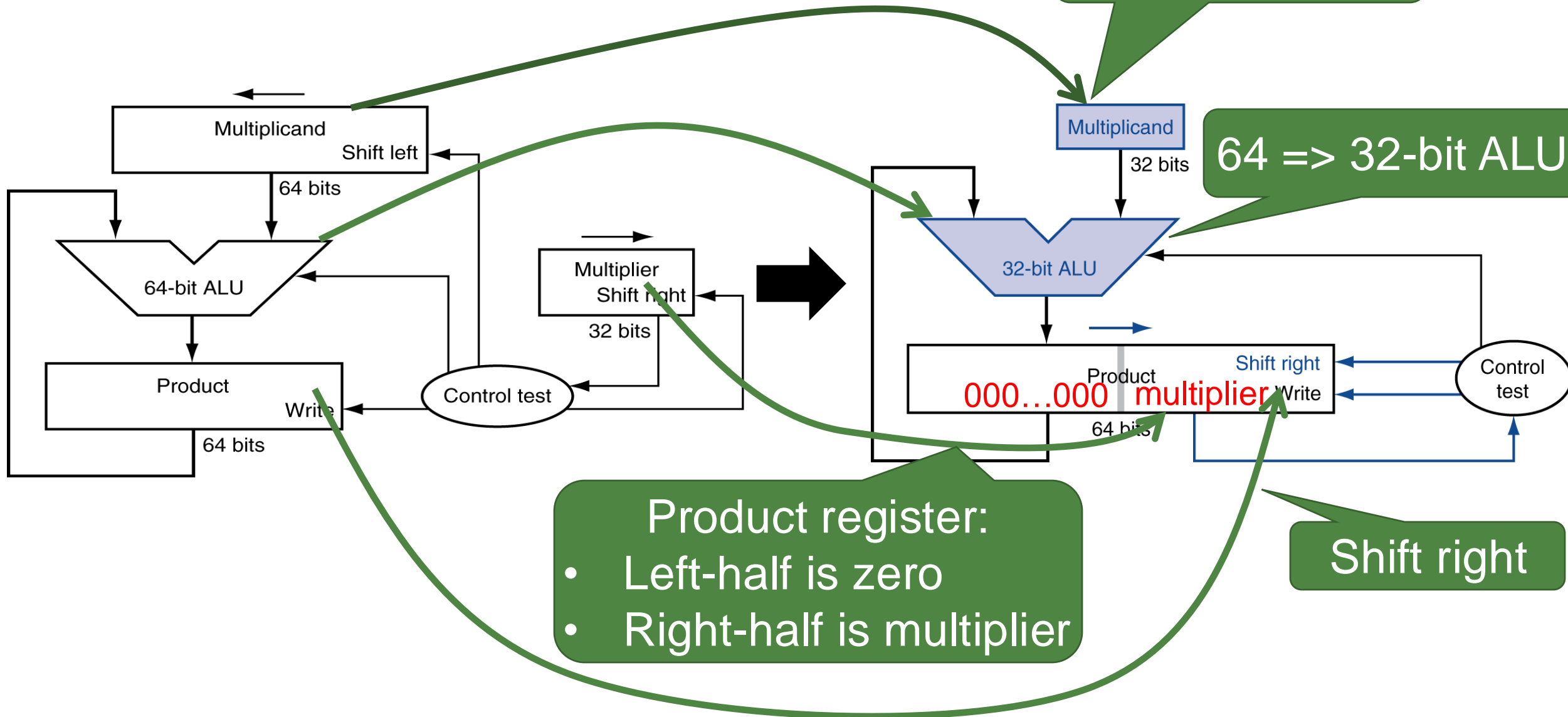


What has Changed?

64

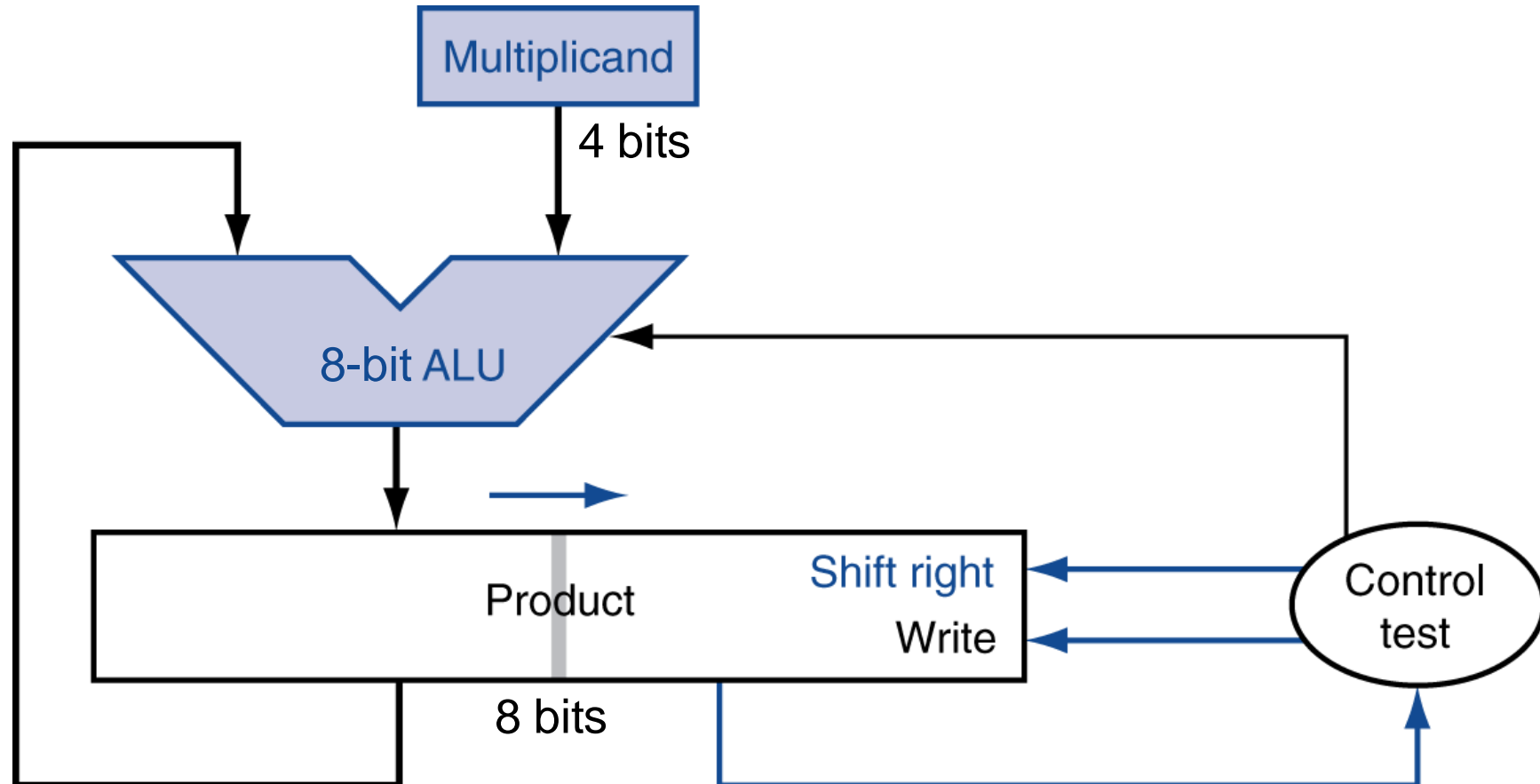
- 64 => 32 bits
- No shift

64 => 32-bit ALU



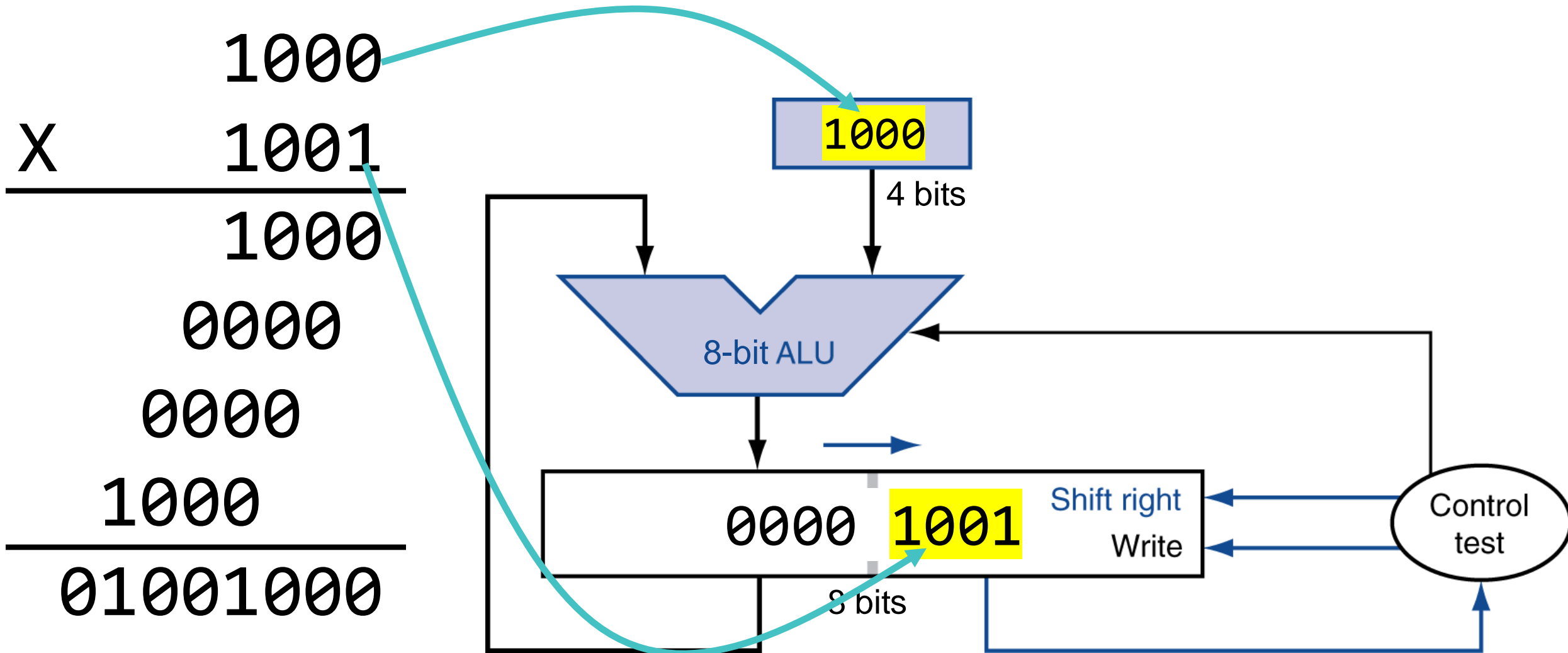
Example: 4-bit Operands

$$\begin{array}{r} 1000 \\ X 1001 \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 01001000 \end{array}$$



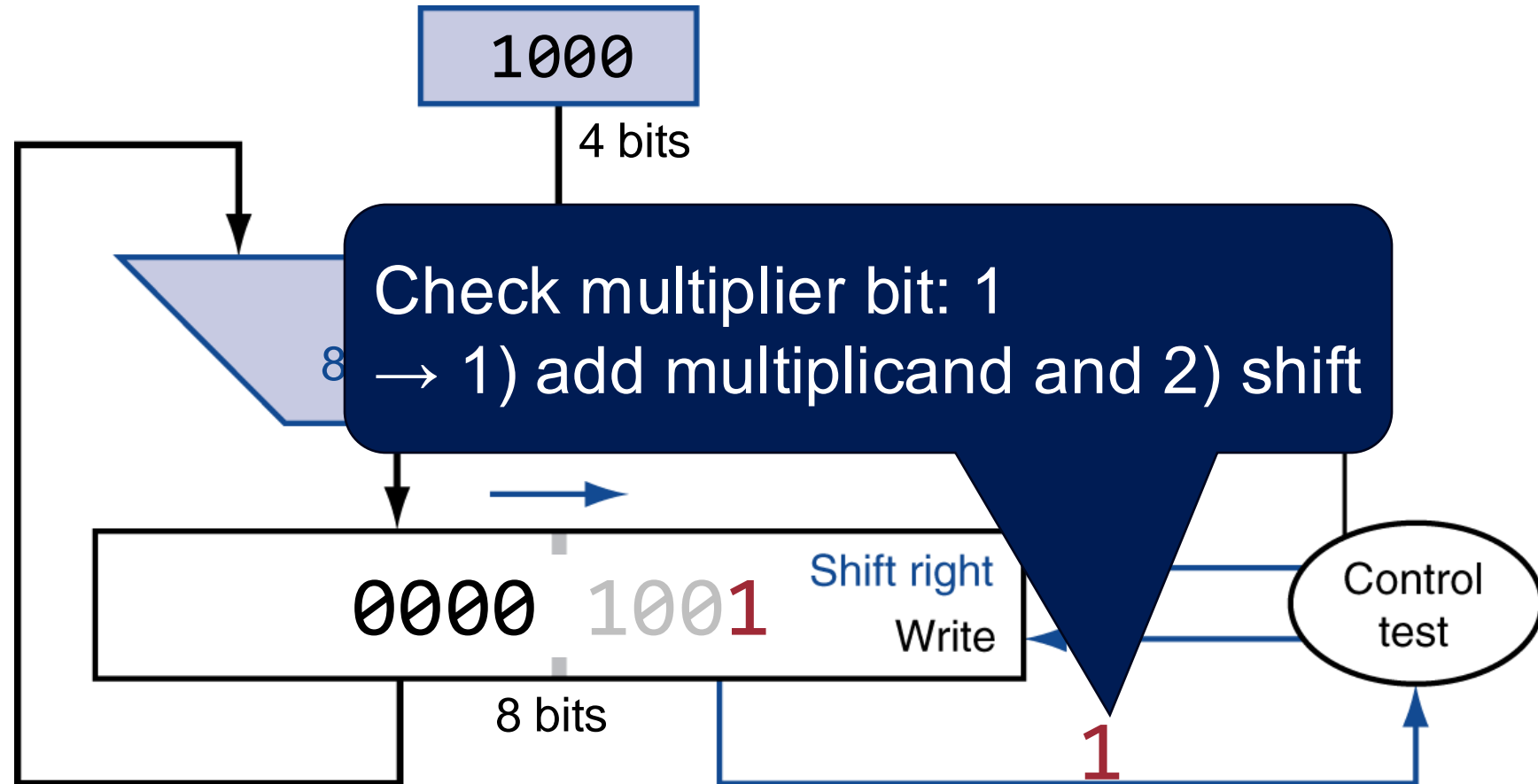
Example: 4-bit Operands – Initial State

66



Example: 1st Iteration – Test Multiplier Bit

$$\begin{array}{r} 1000 \\ X 100\mathbf{1} \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 01001000 \end{array}$$



1st iteration

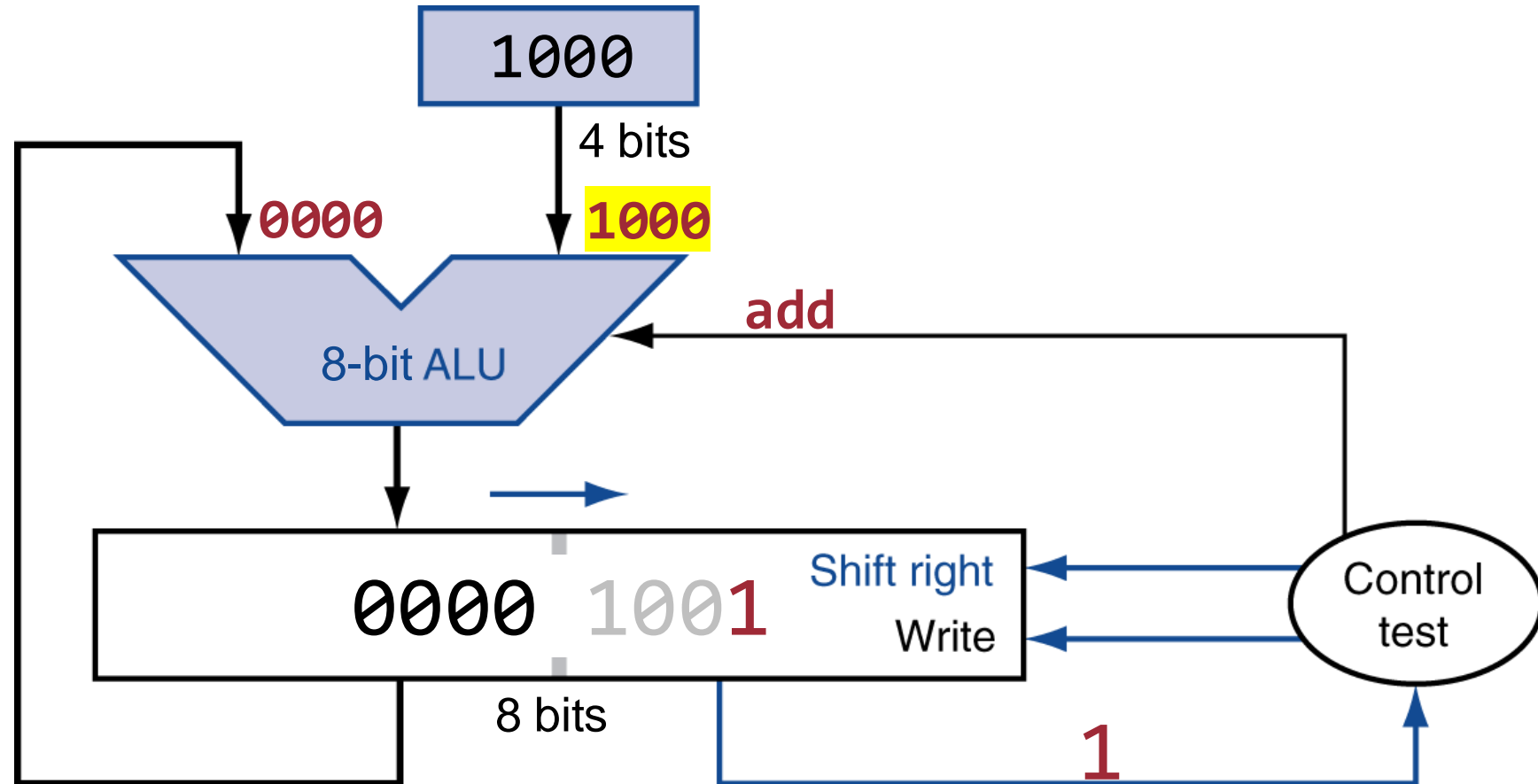
Example: 1st Iteration – Before Addition

68

X 1000
 1001

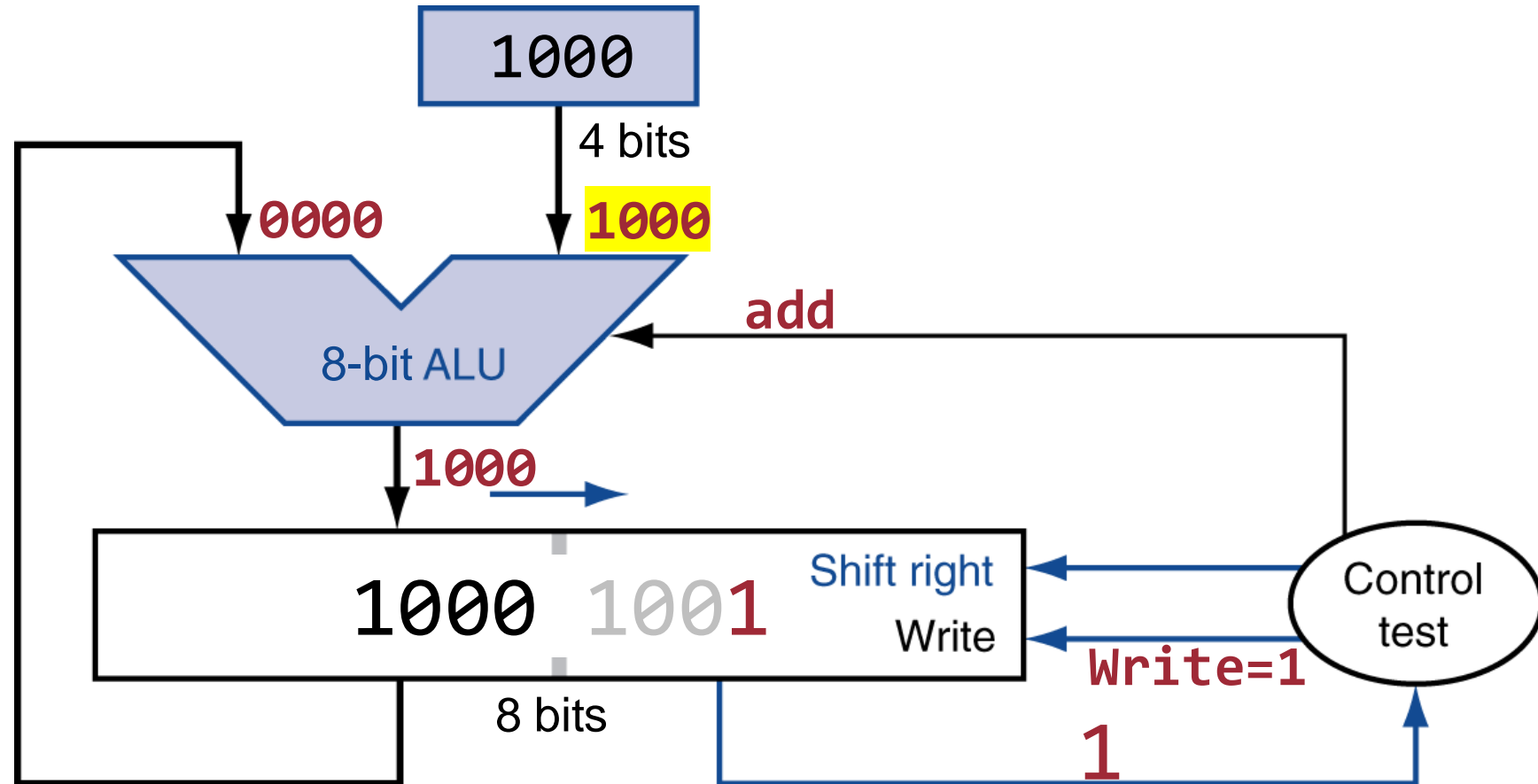
 1000
 0000
 0000
 1000

01001000



Example: 1st Iteration – After Addition

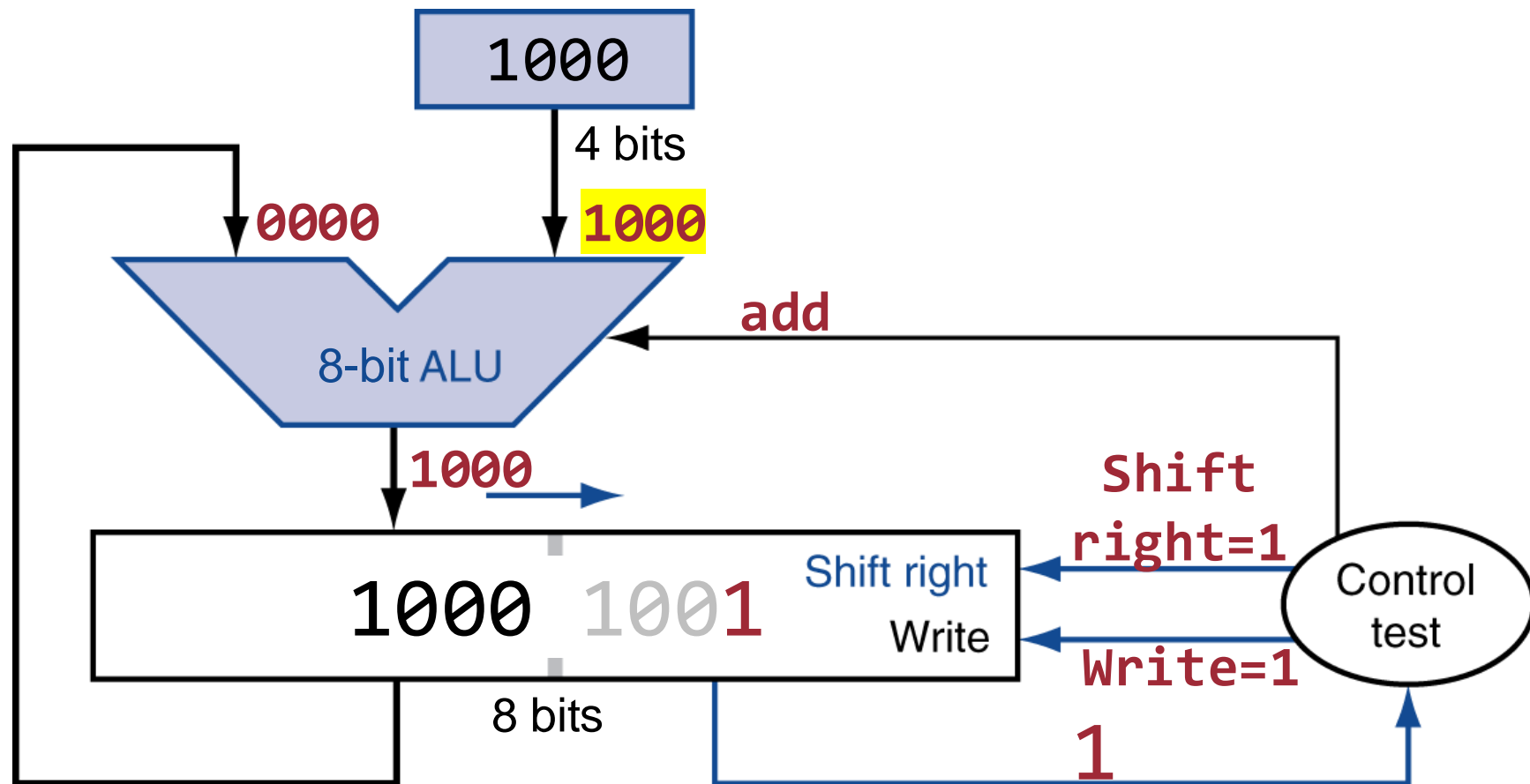
$$\begin{array}{r}
 1000 \\
 X \quad 1001 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$



1st iteration

Example: 1st Iteration – Before Shift

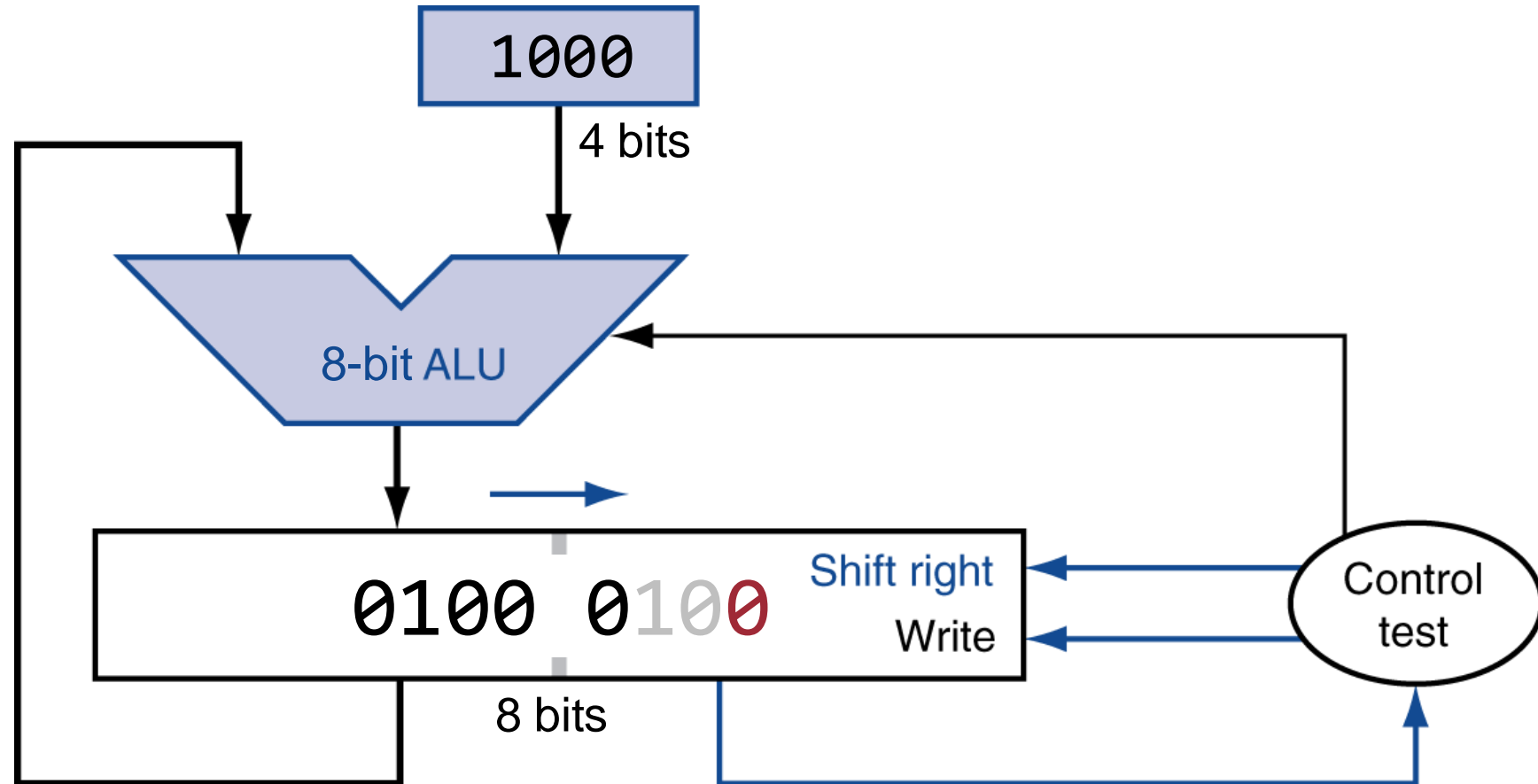
$$\begin{array}{r}
 1000 \\
 X \quad 1001 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$



1st iteration

Example: 1st Iteration – After Shift

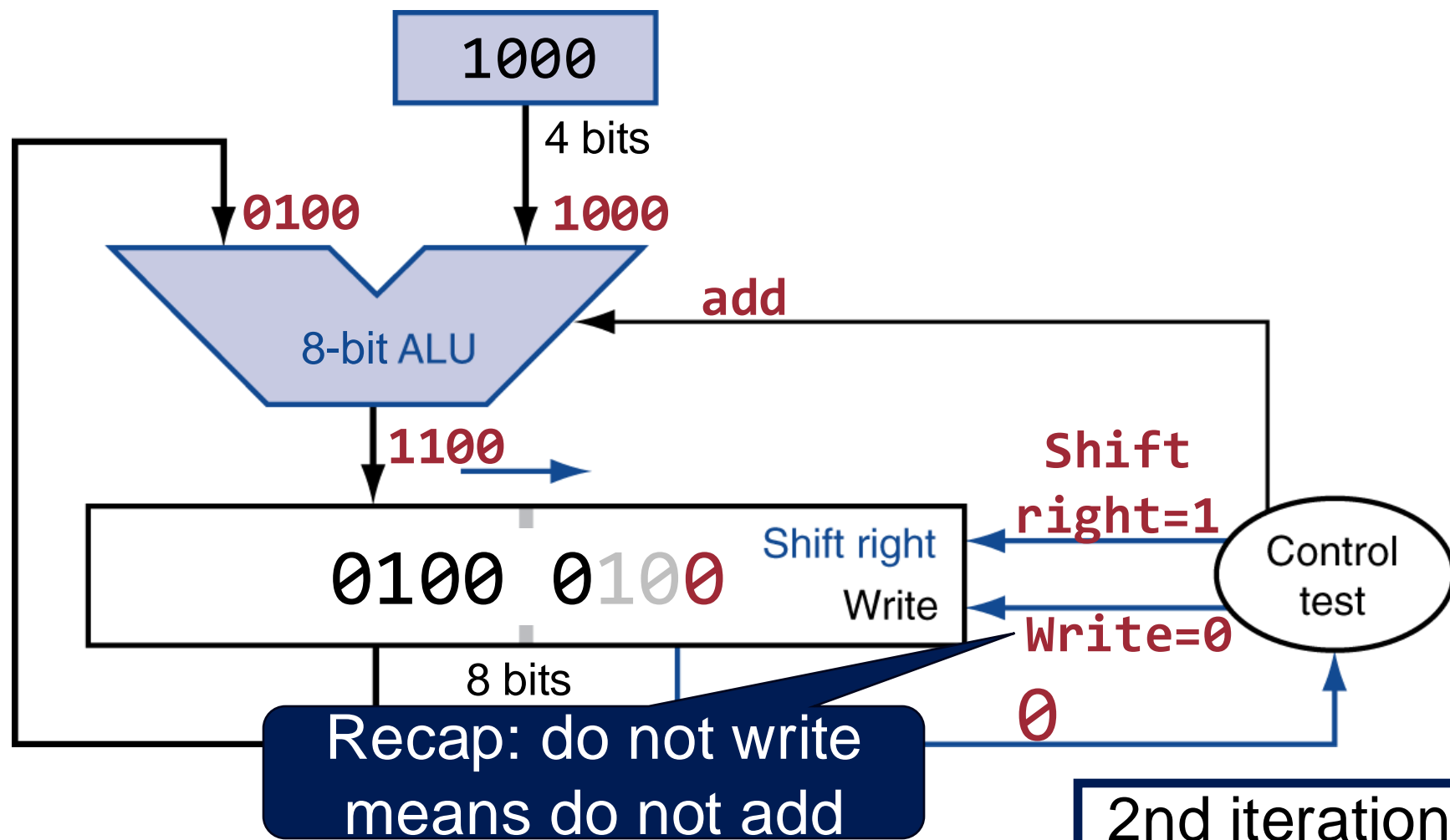
$$\begin{array}{r}
 1000 \\
 X \quad 1001 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$



1st iteration

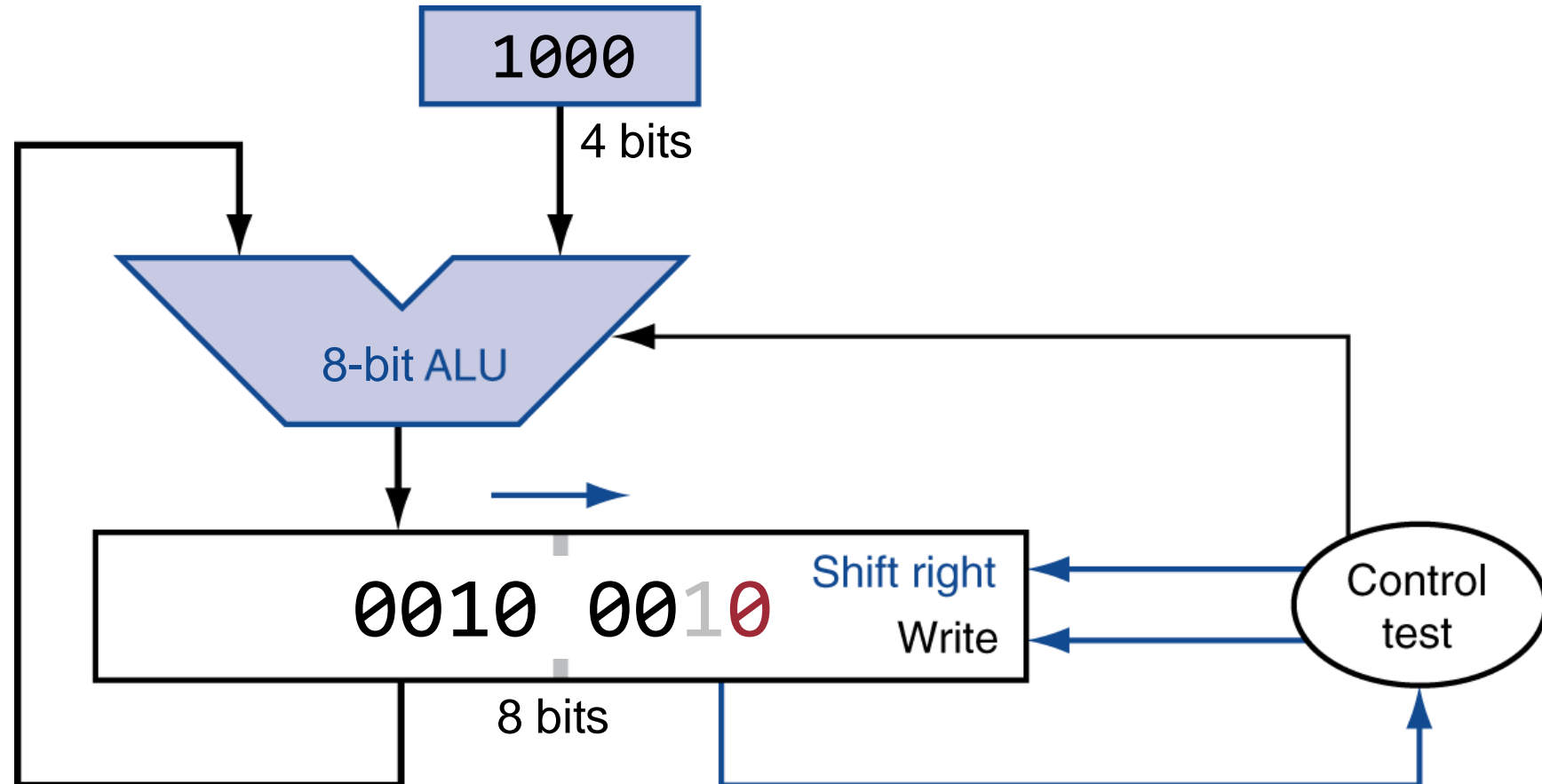
Example: 2nd Iteration – Before Shift

$$\begin{array}{r}
 1000 \\
 X \quad 1001 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$



Example: 2nd Iteration – After Shift

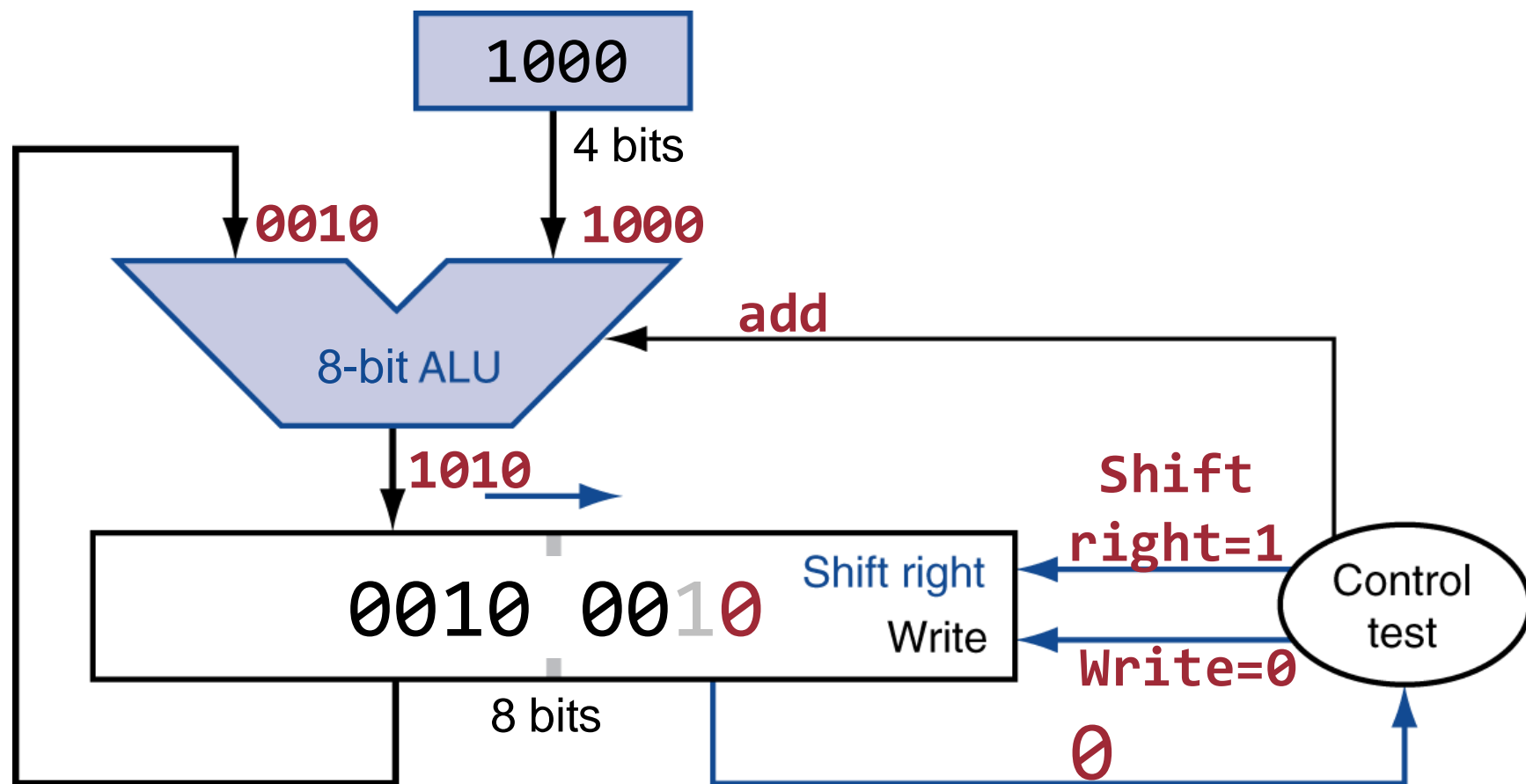
$$\begin{array}{r}
 1000 \\
 X \quad 1\cancel{0}01 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$



2nd iteration

Example: 3rd Iteration – Before Shift

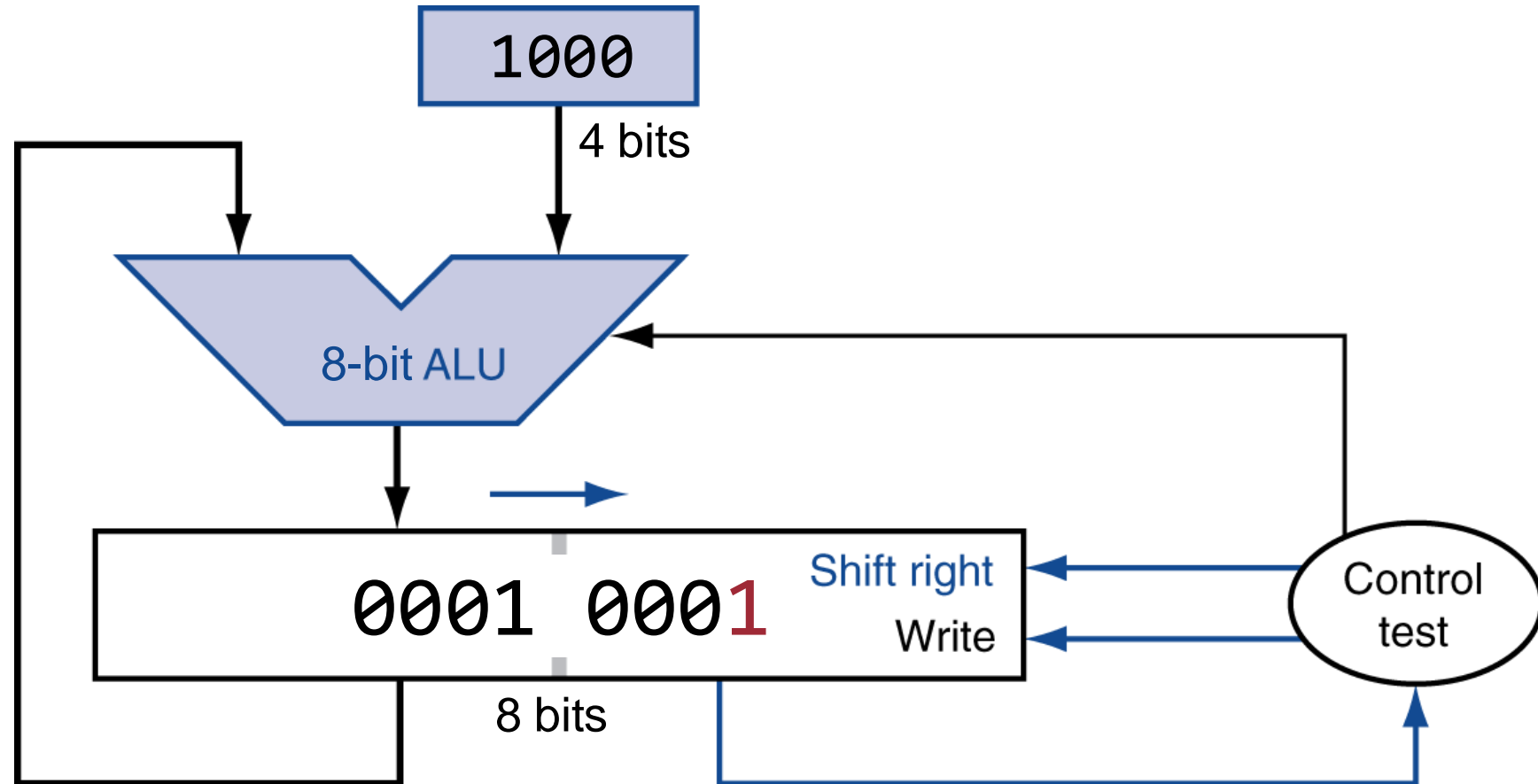
$$\begin{array}{r}
 1000 \\
 X \quad 1\cancel{0}01 \\
 \hline
 1000 \\
 0000 \\
 \text{0000} \\
 1000 \\
 \hline
 01001000
 \end{array}$$



3rd iteration

Example: 3rd Iteration – After Shift

$$\begin{array}{r}
 1000 \\
 X \quad 1001 \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 01001000
 \end{array}$$



3rd iteration

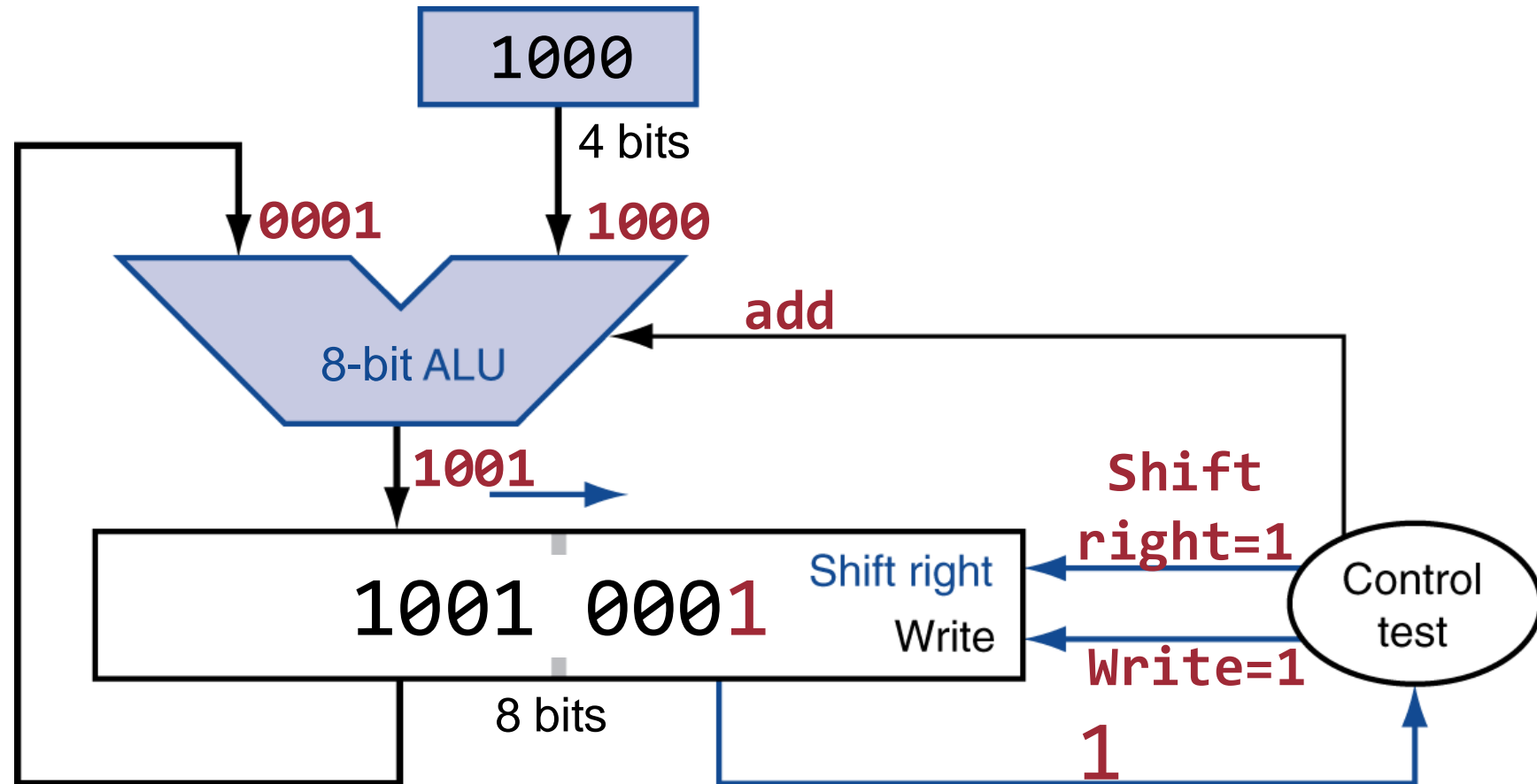
Example: 4th Iteration – After Addition

76

X 1000
 1001

 1000
 0000
 0000
 1000

01001000



4th iteration

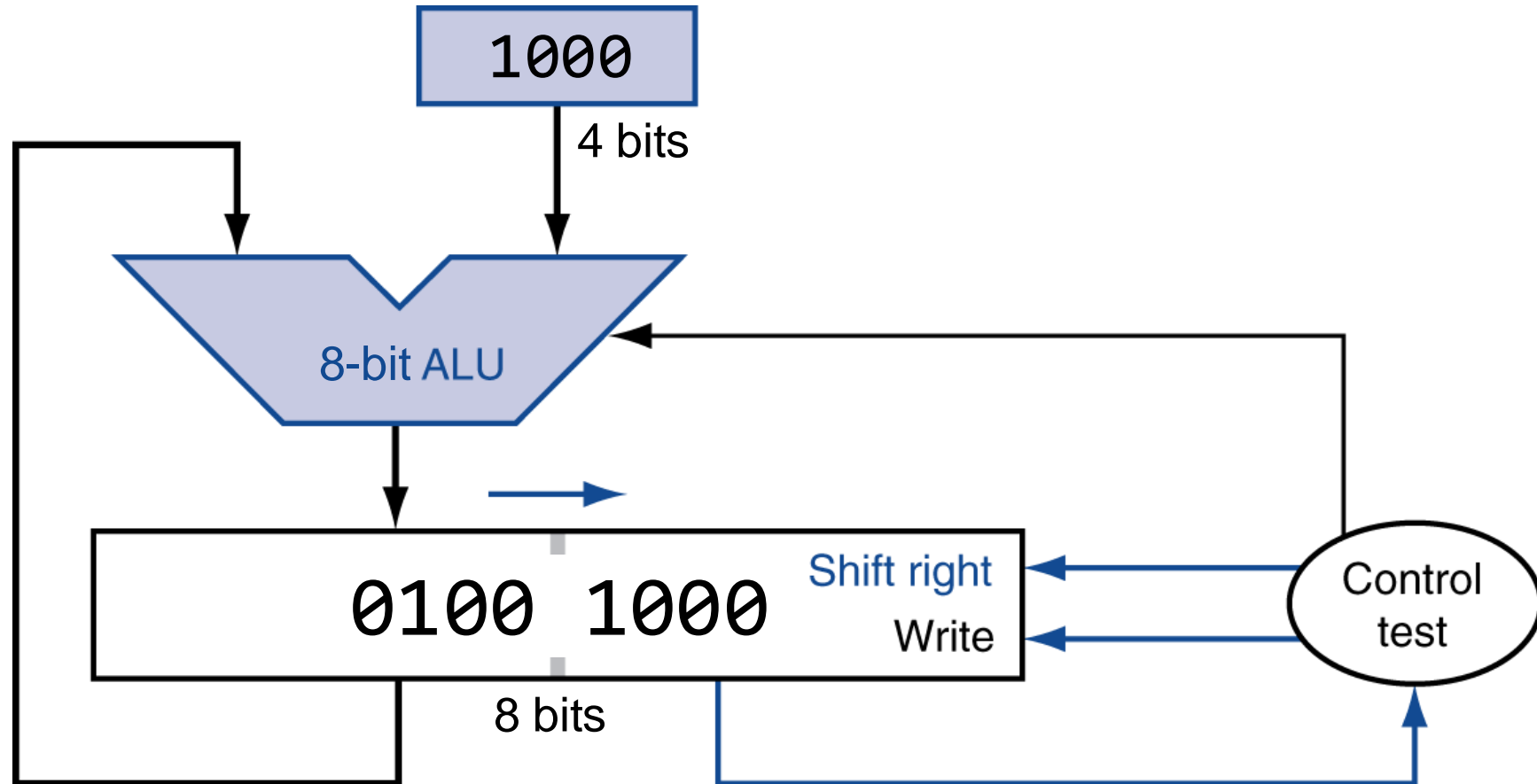
Example: 4th Iteration – After Shift

77

X 1000
 1001

 1000
 0000
 0000
 1000

01001000



4th iteration

Example Summary

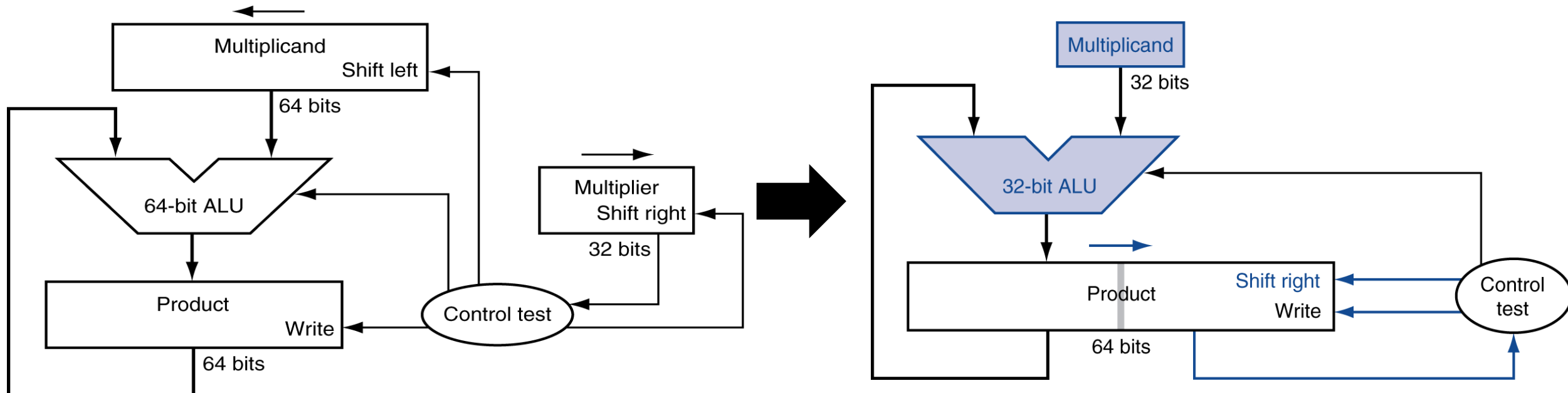


- $1000 \times 1001 = 8 \times 9 = 196$

Iteration–Operation	Product – <i>before</i> operation	Product – <i>after</i> operation
1st – add	0000 100 1	1000 1001
1st – shift	1000 1001	0100 0100
2nd – shift	0100 010 0	0010 0010
3rd – shift	0010 001 0	0001 0001
4th – add	0001 000 1	1001 0001
4th – shift	1001 0001	0100 1000

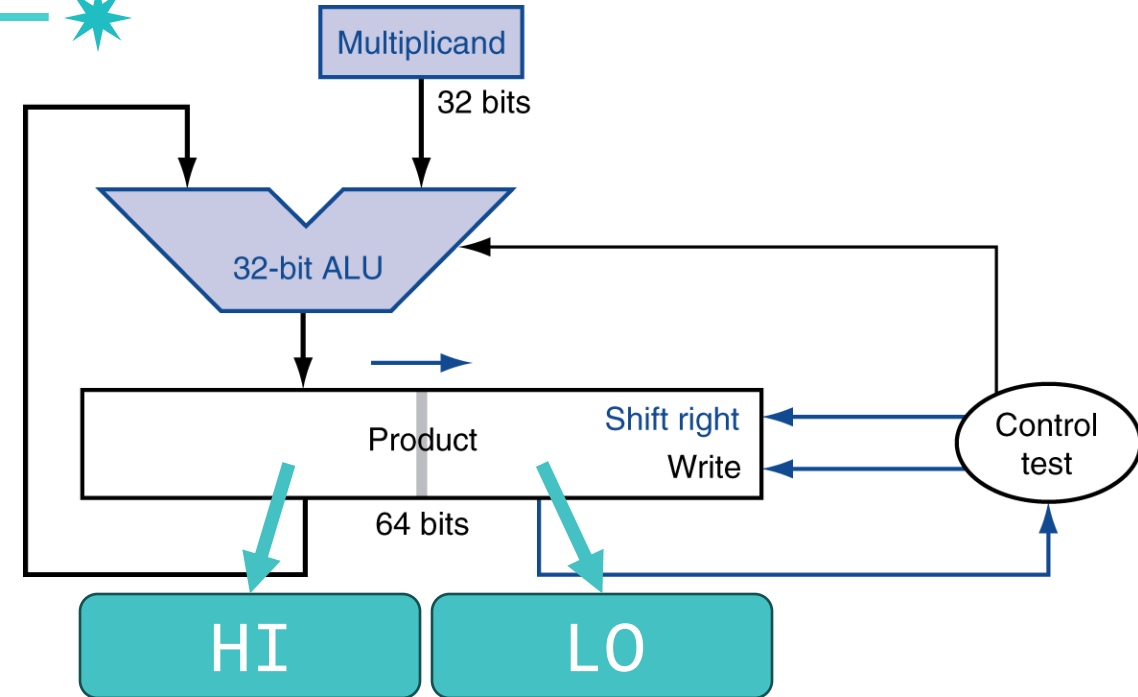
Summary: Optimized Multiplier

- The hardware is optimized to halve the width of the ALU and registers (64 bits \Rightarrow 32 bits, Clock cycle time \downarrow)
- Perform steps in parallel: add/shift (# of clock cycle \downarrow)



MIPS Multiplication Instructions

- 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
 - Can test HI value to see if product overflows 32 bits
 - `mul rd, rs, rt`
 - A pseudoinstruction
 - Least-significant 32 bits of product → rd



85

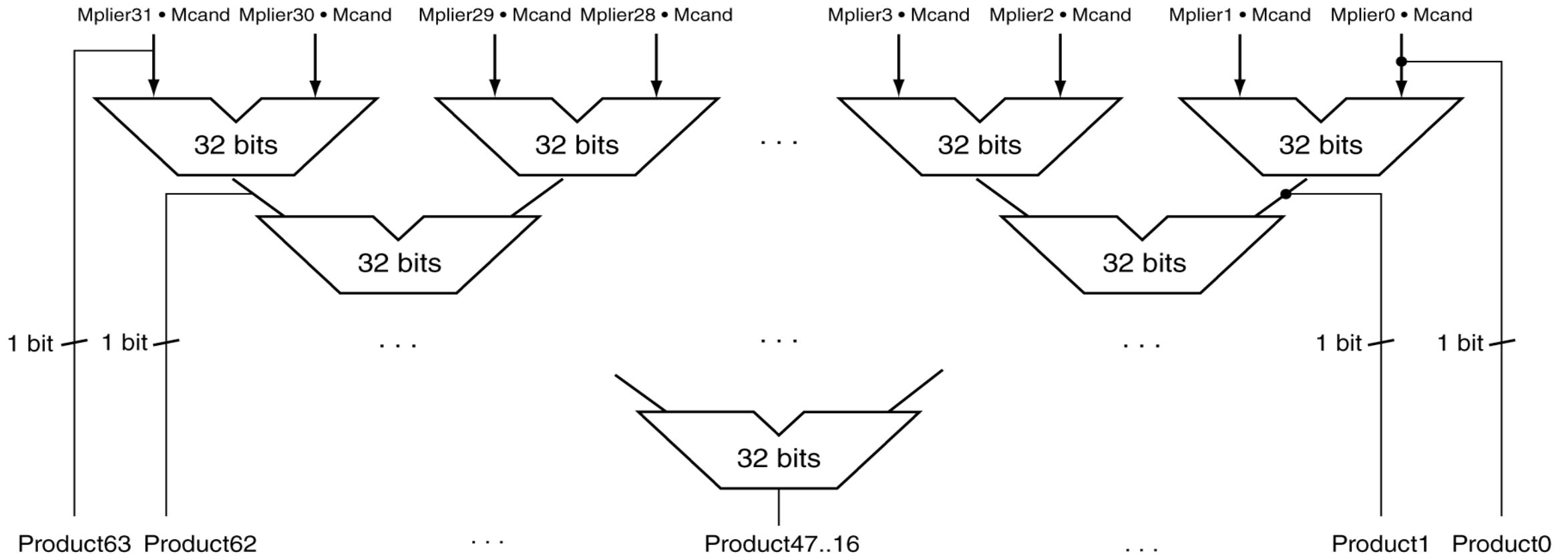
- ```

 M3 M2 M1 M0 (Mcand)
 * m3 m2 m1 m0 (Mplier)

 a3 a2 a1 a0 (Mplier0*Mcand)
+ b3 b2 b1 b0 (Mplier1*Mcand)
+ c3 c2 c1 c0 (Mplier2*Mcand)
+ d3 d2 d1 d0 (Mplier3*Mcand)

p7 p6 p5 p4 p3 p2 p1 p0 (Product)

```



# Multiplication of Negative Numbers



*What about the multiplication of negative number(s)?*

1. Convert the multiplier and multiplicand to positive numbers
2. Perform unsigned multiplication
3. Set the sign of the result based on the signs of the original numbers

**Question?**