

# CSE331 – Computer Organization

Lecture 5: Multiply, Shift and Divide

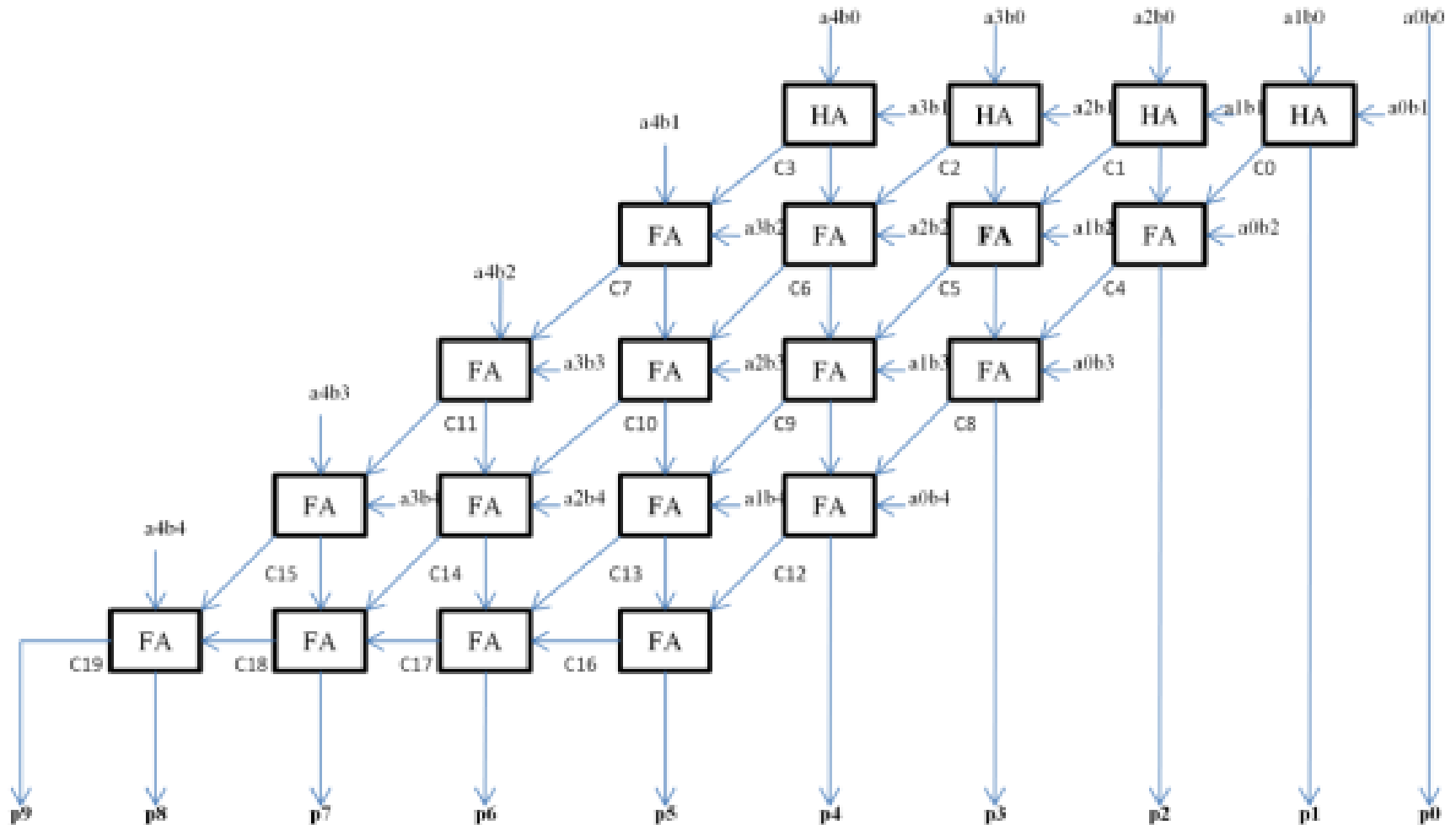
# MULTIPLY (unsigned)

- ▶ Paper and pencil example (unsigned):

$$\begin{array}{rcl} \text{Multiplicand} & \longrightarrow & 1000 = 8 \\ \text{Multiplier} & \longrightarrow & \begin{array}{r} \text{x} \quad 1001 = 9 \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 01001000 = 72 \end{array} \end{array}$$

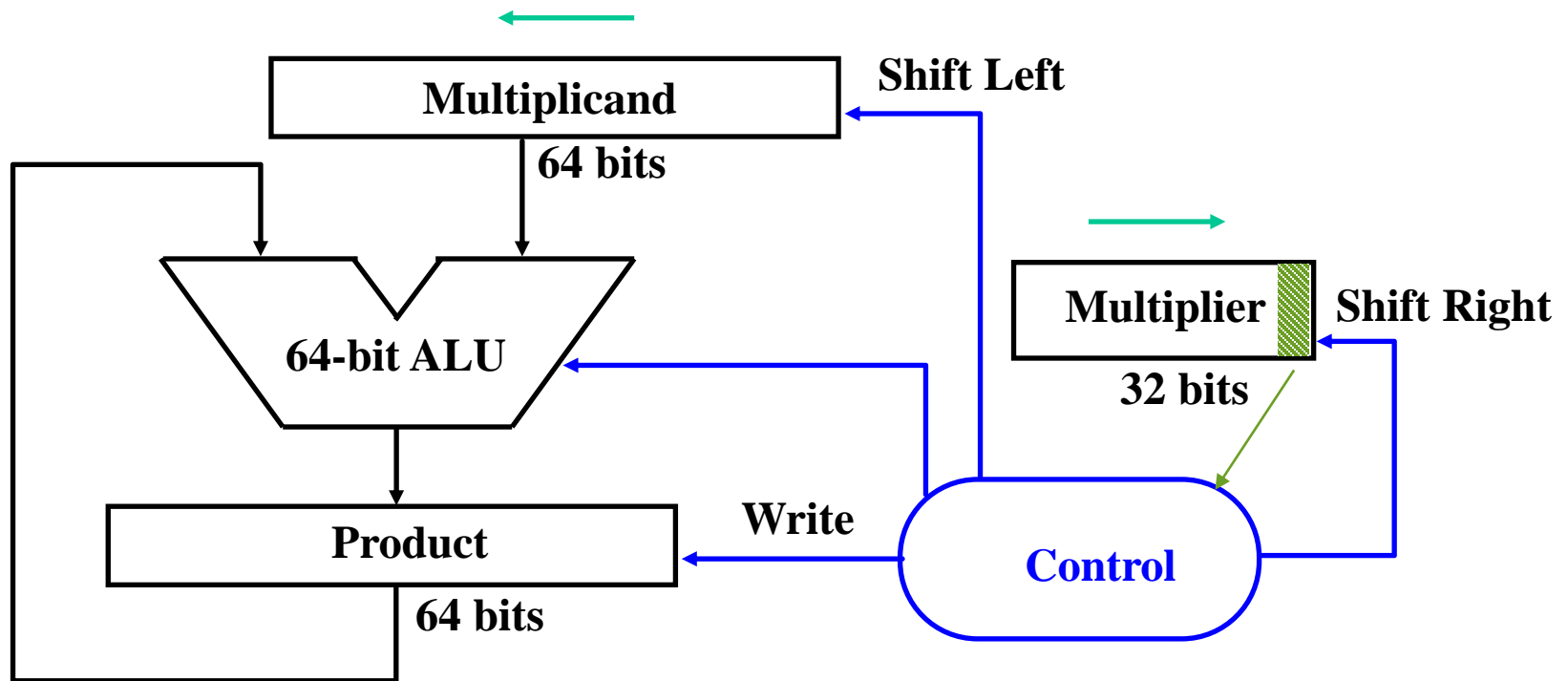
- ▶  $n$  bits  $\times$   $n$  bits =  $2n$  bit product
- ▶ Binary makes it easy:
  - 0  $\Rightarrow$  place 0 (0  $\times$  multiplicand)
  - 1  $\Rightarrow$  place a copy (1  $\times$  multiplicand)
- ▶ 4 versions of multiply hardware & algorithm:
  - **successive refinement**

# Conventional Array Multiplier



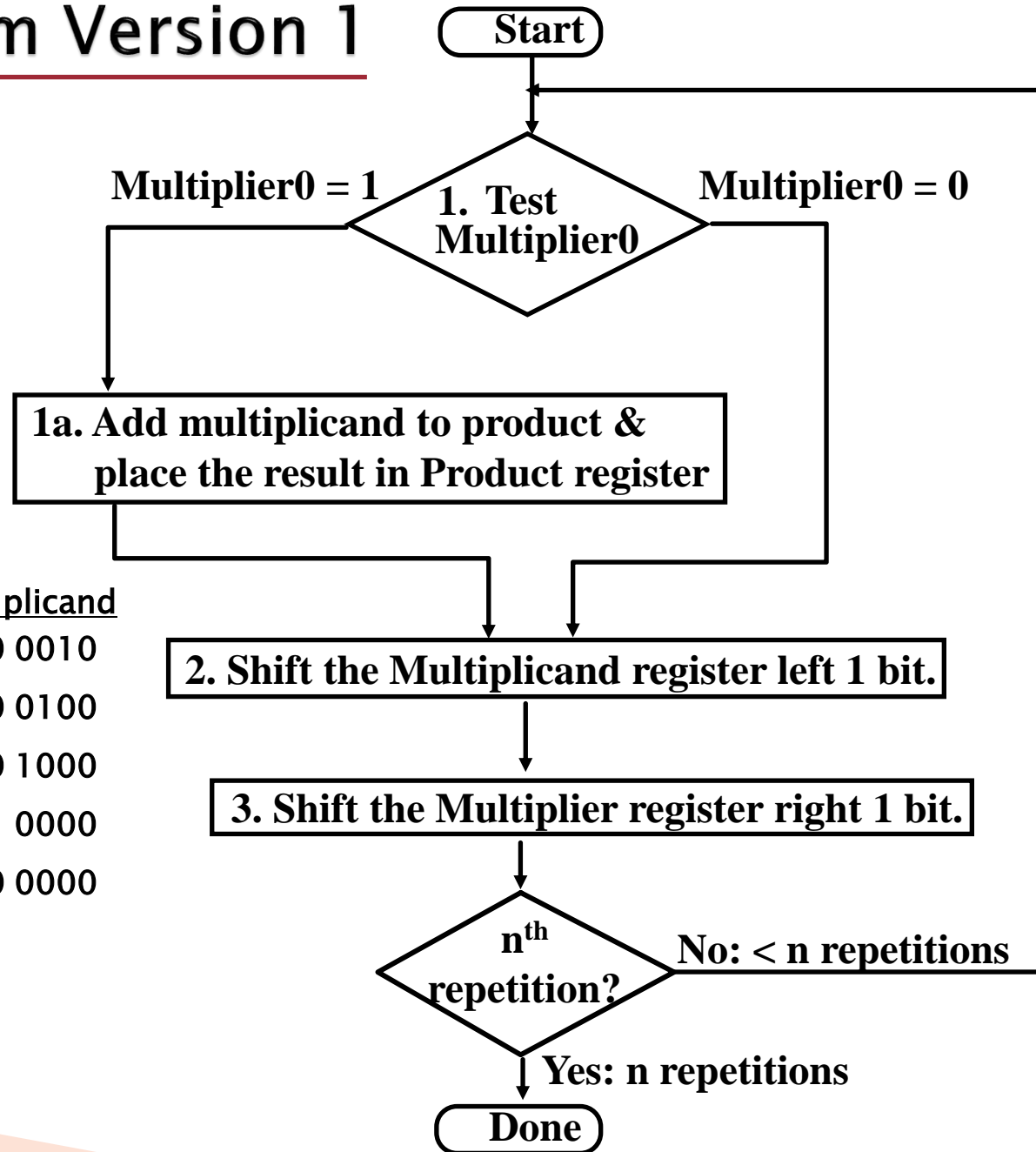
# Unsigned shift-add multiplier (version 1)

- ▶ 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



**Multiplier = datapath + control**

# Multiply Algorithm Version 1



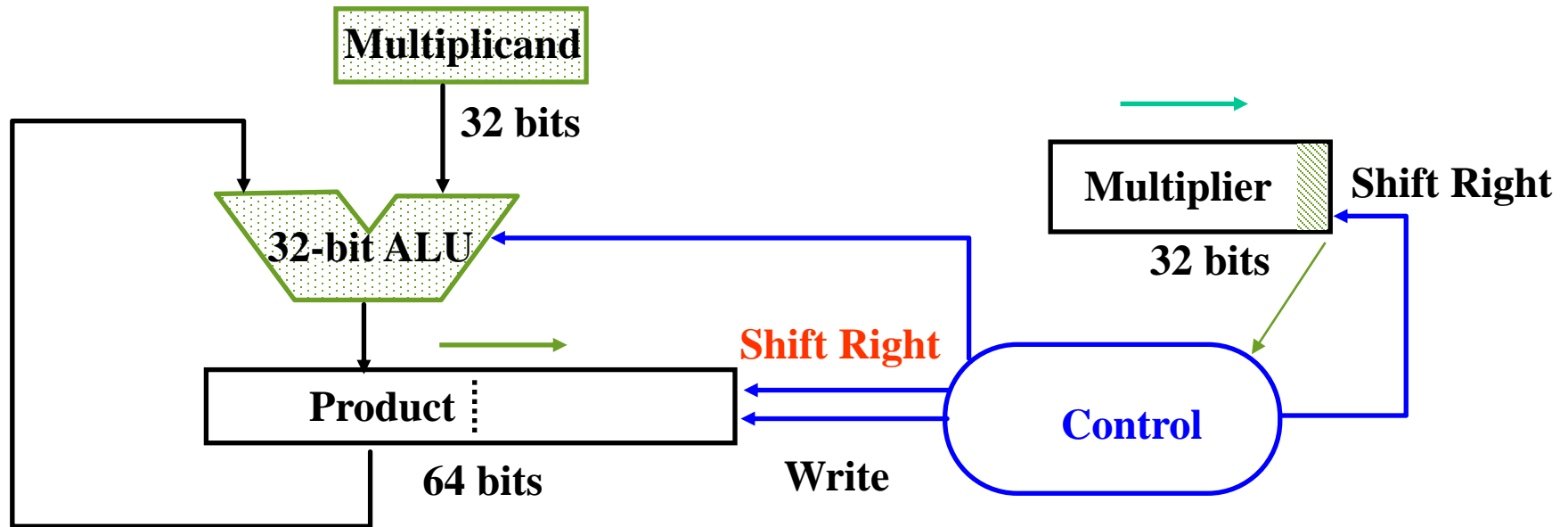
<u>Product</u>	<u>Multiplier</u>	<u>Multiplicand</u>
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110	0000	0001 0000
0000 0110	0000	0010 0000
0000 0110		

# Observations on Multiply Version 1

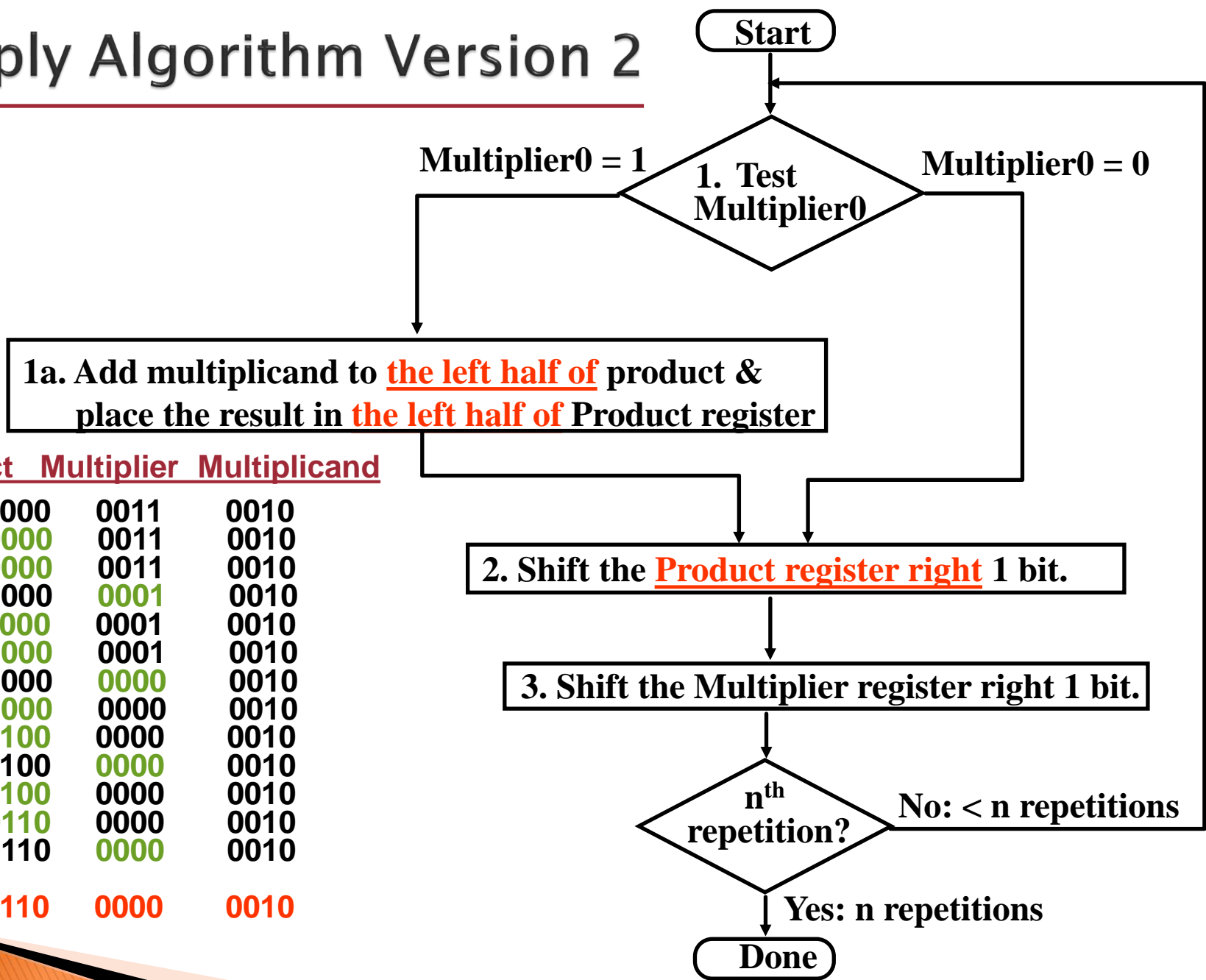
- ▶ 1 / 2 bits in multiplicand always 0  
=> 64-bit adder is wasted
- ▶ 0's inserted into the least significant bit of multiplicand as shifted => least significant bits of product never changed once formed
- ▶ Instead of shifting multiplicand to left, shift product to right.

# MULTIPLY HARDWARE Version 2

- ▶ 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg



# Multiply Algorithm Version 2

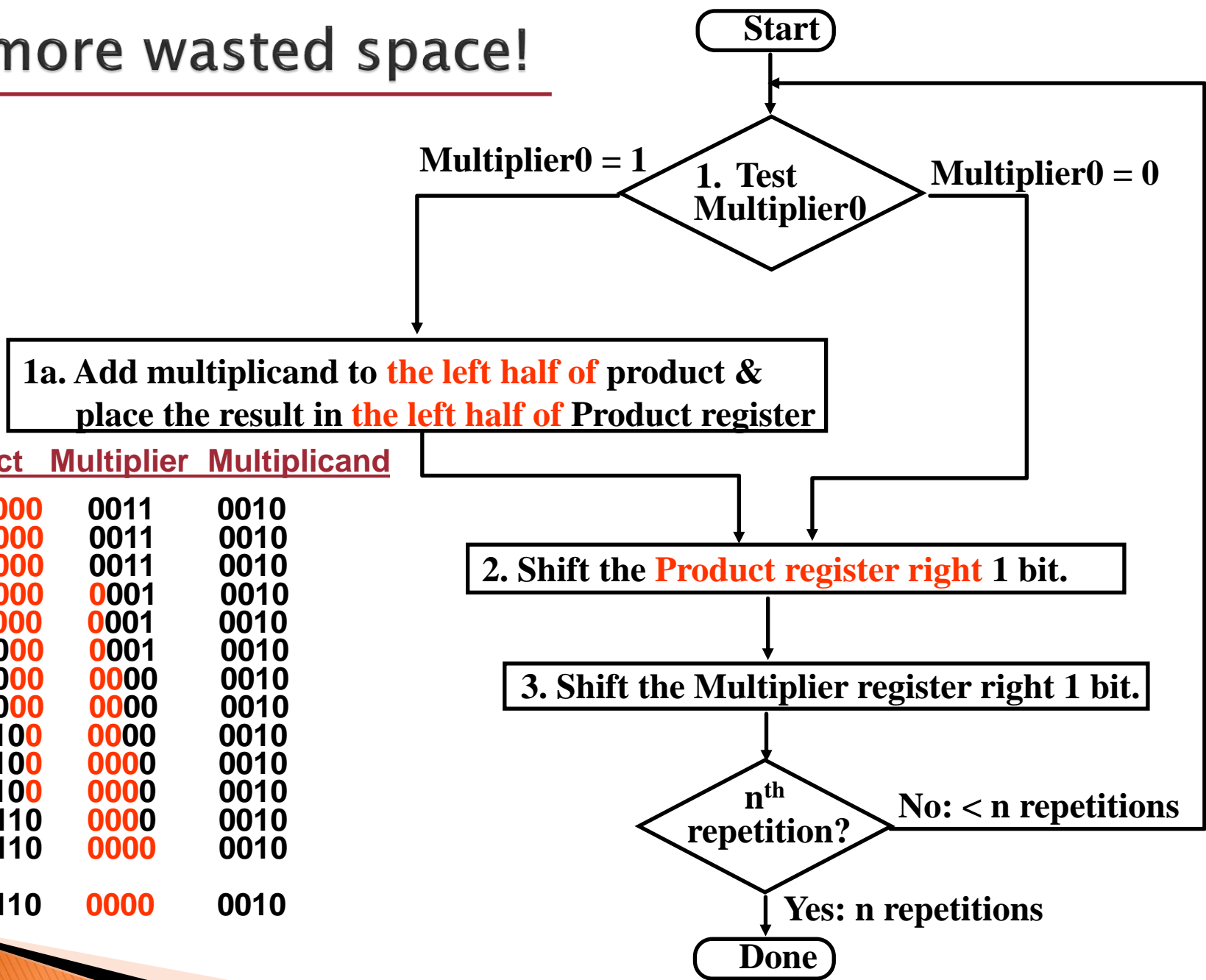


Product Multiplier Multiplicand

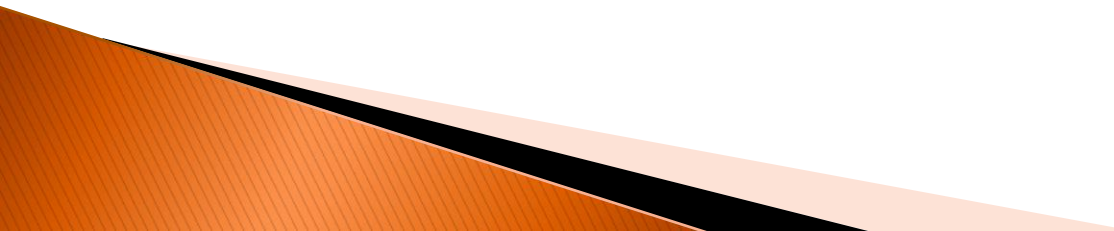
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010



# Still more wasted space!

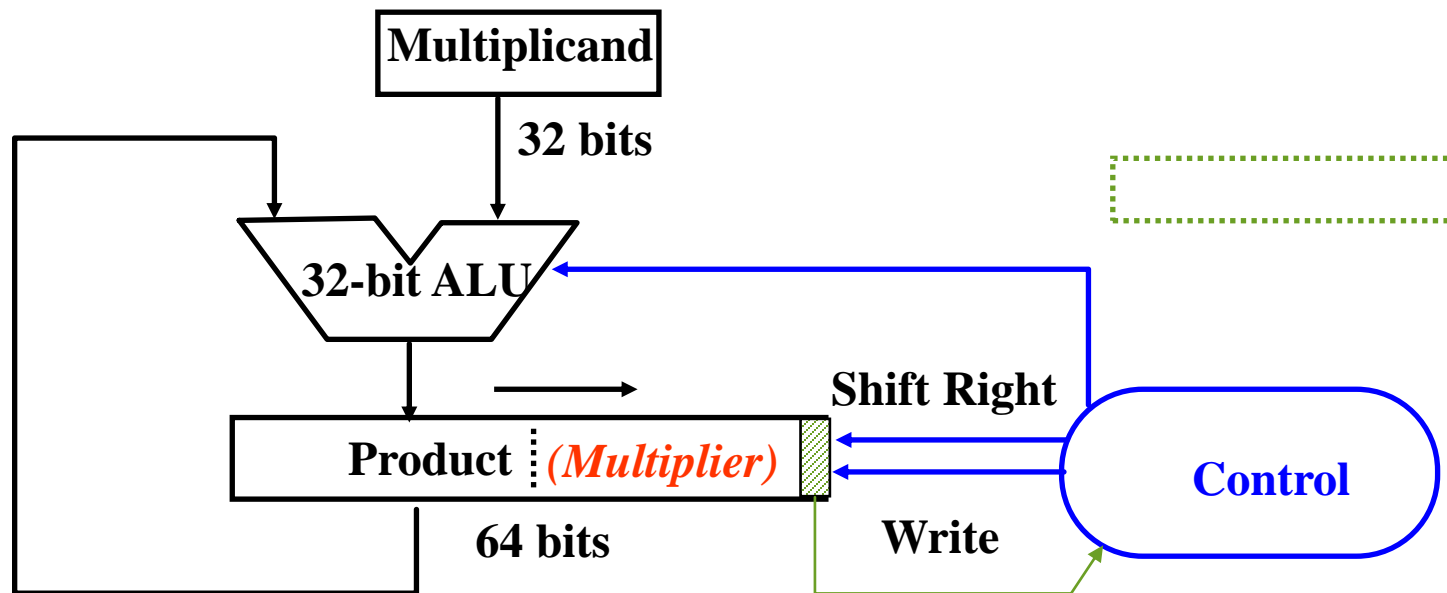


# Observations on Multiply Version 2

- ▶ Product register wastes space that exactly matches size of multiplier
  - ▶ Both Multiplier register and Product register require right shift
  - ▶ Combine Multiplier register and Product register
- 

# MULTIPLY HARDWARE Version 3

- ▶ 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



# Multiply Algorithm Version 3

MultiplicandMultiplier

0010

0011

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

	<u>Product</u>	<u>Multiplicand</u>
	0000 0011	0010
1:	0010 0011	0010
2:	0001 0001	0010
1:	0011 0001	0010
2:	0001 1000	0010
1:	0001 1000	0010
2:	0000 1100	0010
1:	0000 1100	0010
2:	0000 0110	0010
	0000 0110	0010

Product0 = 1

1. Test  
Product0

Product0 = 0

2. Shift the Product register right 1 bit.

32nd  
repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

# Observations on Multiply Version 3

- ▶ 2 steps per bit because Multiplier & Product combined
- ▶ MIPS registers Hi and Lo are left and right half of Product
- ▶ Gives us MIPS instruction Multu
- ▶ What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - Booth's Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
    - can be modified to handle multiple bits at a time

# Motivation for Booth's Algorithm

- Example  $2 \times 6 = 0010 \times 0110$ :

$$\begin{array}{r}
 \phantom{x} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0010 \\
 \phantom{x} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0110 \\
 \hline
 x \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0000 \\
 + \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0000 \\
 + \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0010 \\
 + \phantom{+} \phantom{+} \phantom{+} 0010 \\
 + \phantom{+} \phantom{+} 0000 \\
 \hline
 00001100
 \end{array}$$

shift (0 in multiplier)  
 add (1 in multiplier)  
 add (1 in multiplier)  
 shift (0 in multiplier)

- ALU with add or subtract gets same result in more than one way:

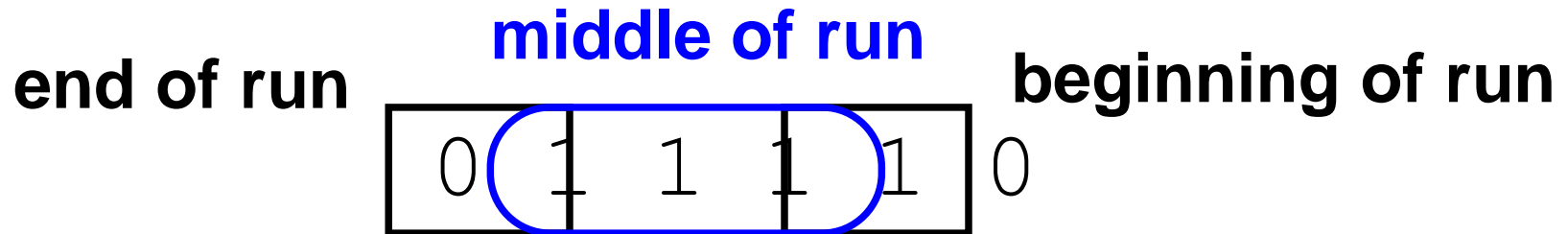
$$\begin{array}{lcl}
 12 & = & -4 + 16 \\
 01100 & = & -00100 + 10000
 \end{array}$$

- For example

$$\begin{array}{r}
 \phantom{x} \phantom{+} \phantom{+} \phantom{+} 0010 \\
 \phantom{x} \phantom{+} \phantom{+} \phantom{+} 0110 \\
 \hline
 x \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0000 \\
 -0010 \\
 \phantom{x} \phantom{+} \phantom{+} 0000 \\
 + 0010 \\
 \hline
 00001100
 \end{array}$$

shift (0 in multiplier)  
 sub (start string of 1's)  
 shift (mid string of 1's)  
 add (end string of 1's)

# Booth's Algorithm



<u>Current Bit</u>	<u>Bit to the Right</u>	<u>Explanation</u>	<u>Example</u>	<u>Op</u>
1	0	Begins run of 1s	000111 <u>1</u> 000	sub
1	1	Middle of run of 1s	00011 <u>11</u> 000	none
0	1	End of run of 1s	00 <u>01</u> 111000	add
0	0	Middle of run of 0s	0 <u>00</u> 1111000	none

Originally for Speed (when shift was faster than add)

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

# Booths Example (2 x 7)

<u>Operation</u>	<u>Multiplicand</u>	<u>Product</u>	<u>next?</u>
0. initial value	0010	0000 0111 0	10 -> sub
1a. $P = P - m$	1110	+ 1110 1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+ 0010 0001 1100 1	shift
4b.	0010	0000 1110 0	done



# Booths Example (2 x -3)

<u>Operation</u>	<u>Multiplicand</u>	<u>Product</u>	<u>next?</u>
0. initial value	0010	0000 1101 0	10 -> sub
1a. $P = P - m$	1110	+ 1110 1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1 + 0010	01 -> add
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0 + 1110	10 -> sub
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a.		1111 0101 1	shift
4b.	0010	1111 1010 1	done

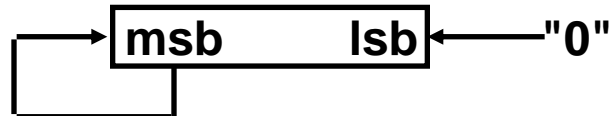
# Shifters

Two kinds:

*logical*-- value shifted in is always "0"



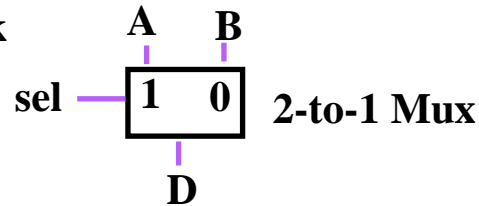
*arithmetic*-- on right shifts, sign extend



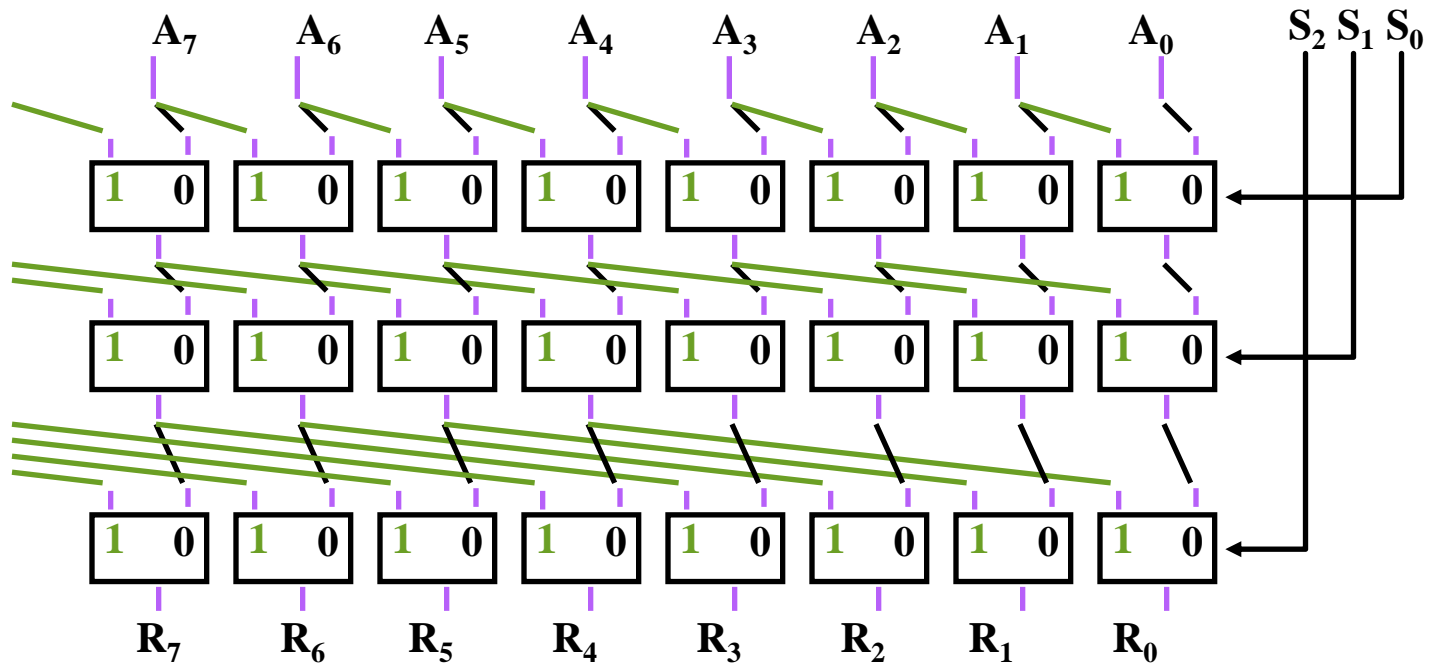
**Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!**

# Combinational Shifter from MUXes

Basic Building Block



8-bit right shifter



- ▶ What comes in the MSBs?
- ▶ How many levels for 32-bit shifter?