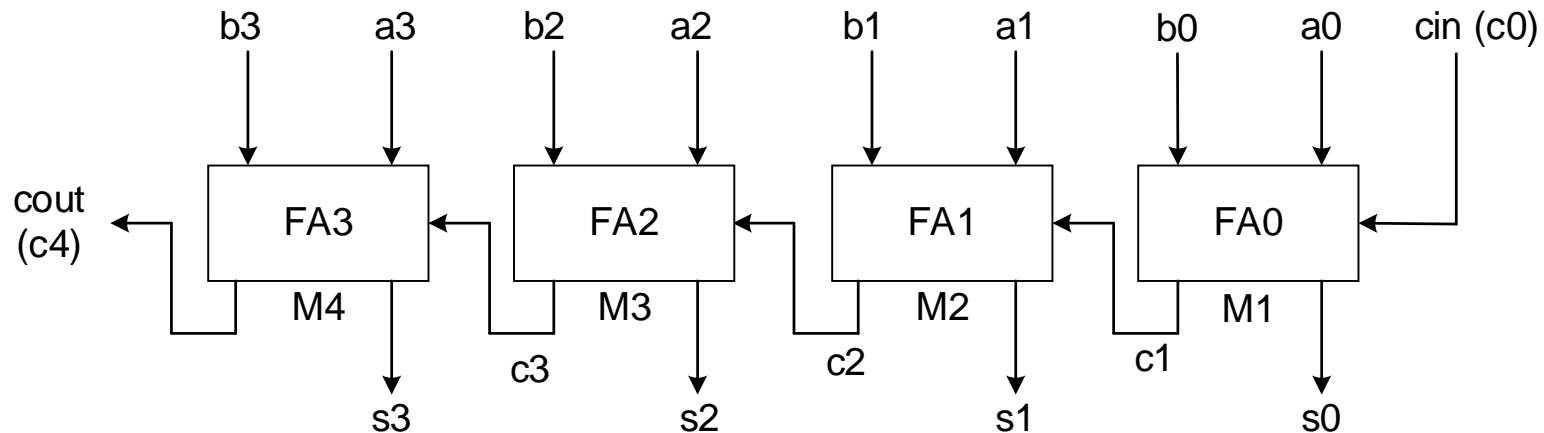


Lecture 7 – Adders and Multipliers

Ripple Carry Adder



A	B	cin	cout	
0	0	0	0	Kill
0	0	1	0	
0	1	0	0	Propagate
1	0	0	0	
0	1	1	1	
1	0	1	1	
1	1	0	1	Generate
1	1	1	1	

- **Key observations**, the value of the carry into any stage of a multi-cell adder depends only on
 - The data bit of previous stage
 - The carry into the 1st stage
 - When both inputs 0, no carry
 - When one is 0, the other is 1, *propagate* carry input
 - When both are 1, then *generate* a carry

Carry-lookahead adder

- Generate
$$G_i = a_i * b_i$$
- Propagate
$$P_i = a_i \text{ xor } b_i; \text{ or } P_i = a_i + b_i$$
- P_i and G_i are mutually exclusive
 - Because P_i is asserted when $(a_i, b_i) = \{(1, 0), (0, 1)\}$;
 G_i is asserted when $(a_i, b_i) = (1, 1)$
- $$S_i = a_i \text{ xor } b_i \text{ xor } c_i = P_i \text{ xor } c_i$$
- $$c_{(i+1)} = (a_i + b_i) * c_i + a * b = (a_i \text{ xor } b_i) * c_i + a * b = P_i * c_i + G_i$$
- Write carry out as function of preceding G , P , and c_{out}
$$c_1 = G_0 + P_0 * c_0$$
$$c_2 = G_1 + P_1 * c_1$$
$$c_3 = G_2 + P_2 * c_2$$
$$c_4 = G_3 + P_3 * c_3$$

Reducing the complexity

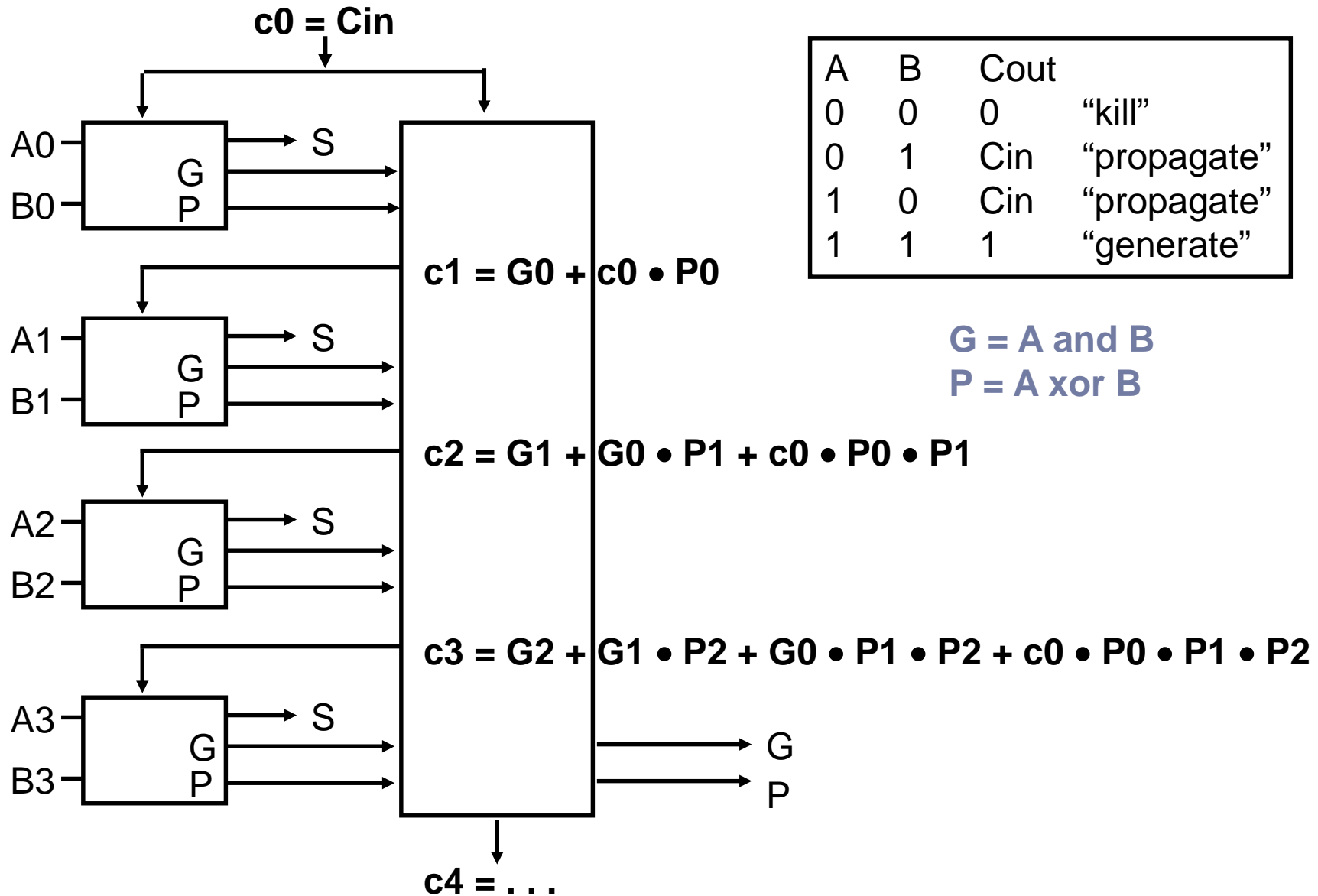
- $c_1 = G_0 + (P_0 * c_0)$
- $c_2 = G_1 + (P_1 * [G_0 + P_0 * c_0])$
 $= G_1 + (P_1 * G_0) + (P_1 * P_0 * c_0)$
- $c_3 = G_2 + (P_2 * G_1) + (P_2 * P_1 * G_0) +$
 $(P_2 * P_1 * P_0 * c_0)$

That is c_i can only use c_0 and $P_{(i-1)} \dots P_0$ and $G_{(i-1)}$
 $\dots G_0$

Increase speed at what cost ?

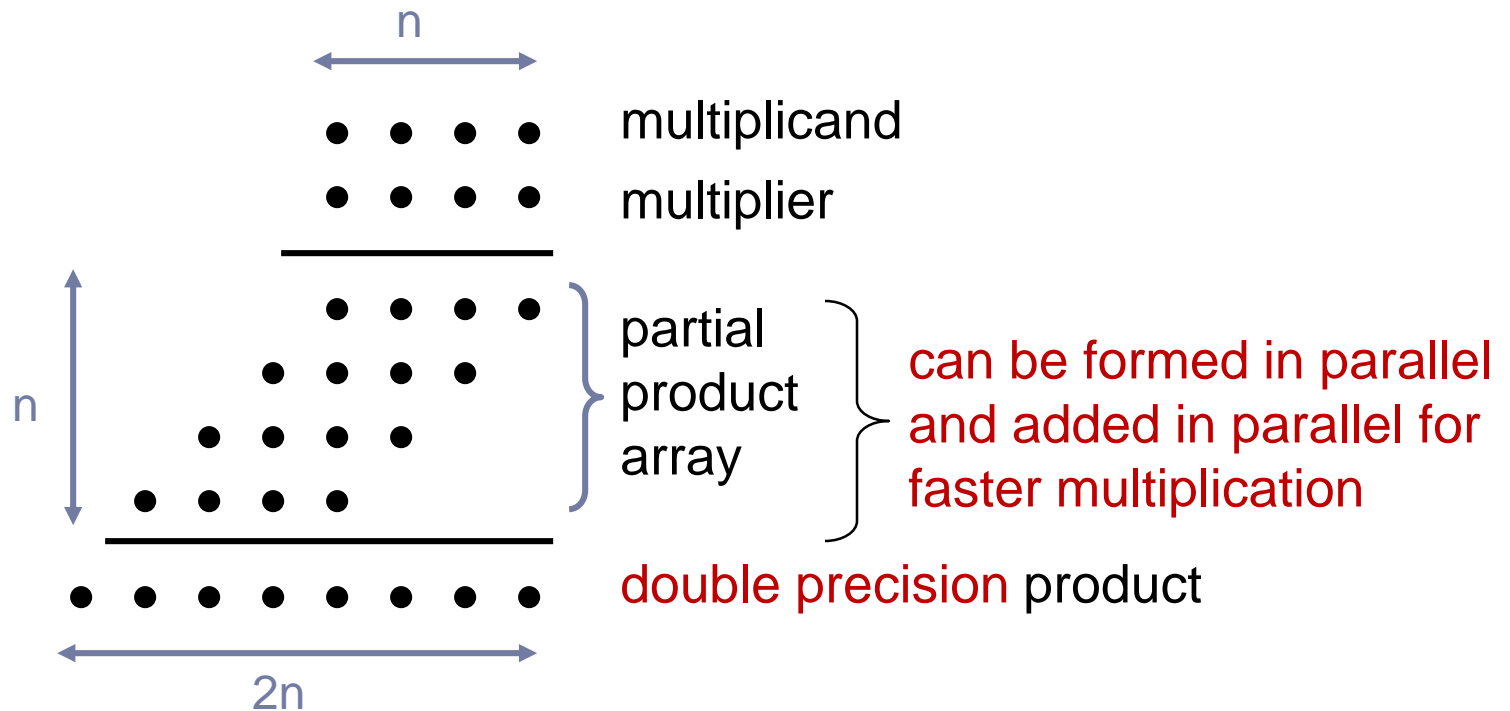
Can you illustrate how to build a 32-bit adder with carry look ahead?

Carry Look Ahead Adder



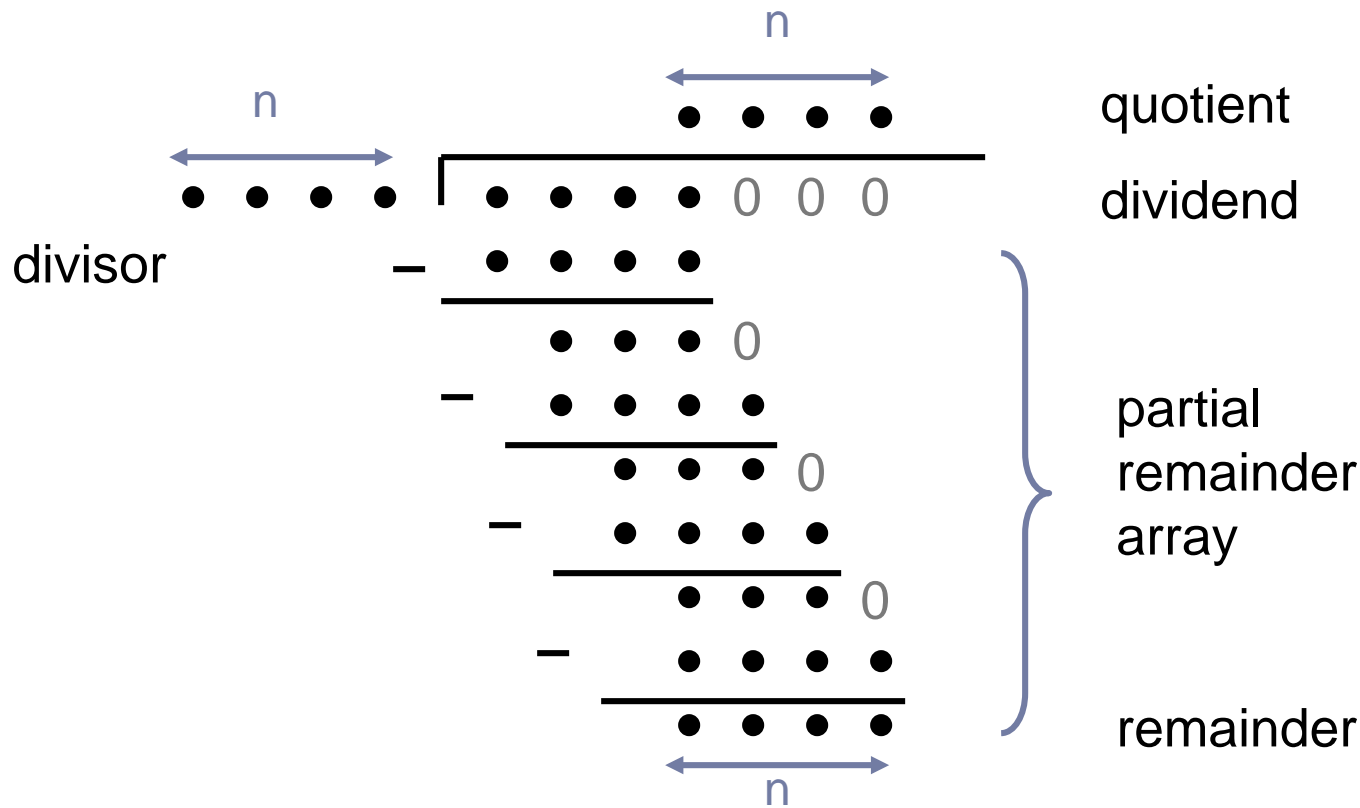
Multiply Overview

- Binary multiplication is just a *bunch* of left shifts and adds



Division Overview

- Division is just a *bunch* of quotient digit guesses and right shifts and subtracts

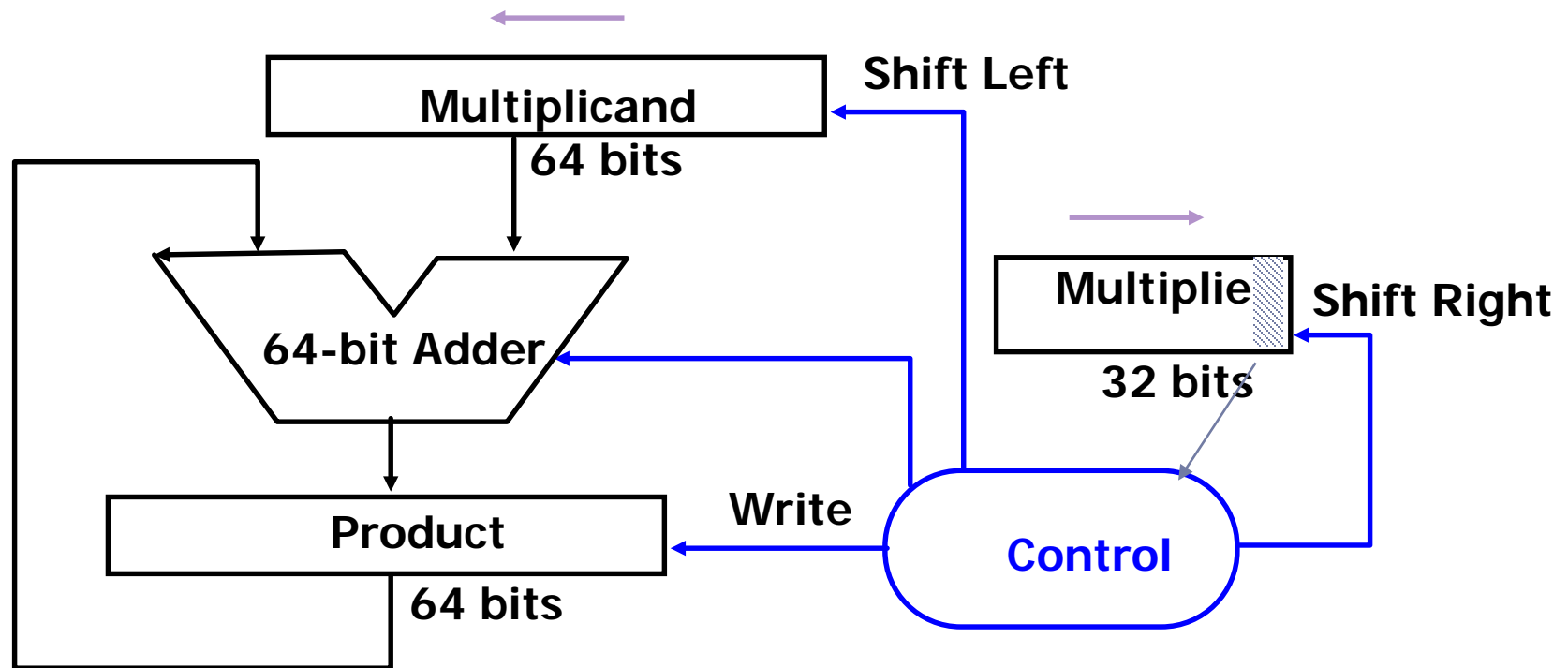


Multiplication: design and implementation

- More complicated than addition
 - accomplished via shifting and addition
- More time and more area
 - m bits \times n bits = $m+n$ bit product
- Let's look at 3 (unsigned) versions of multiplication designs in the next few slides

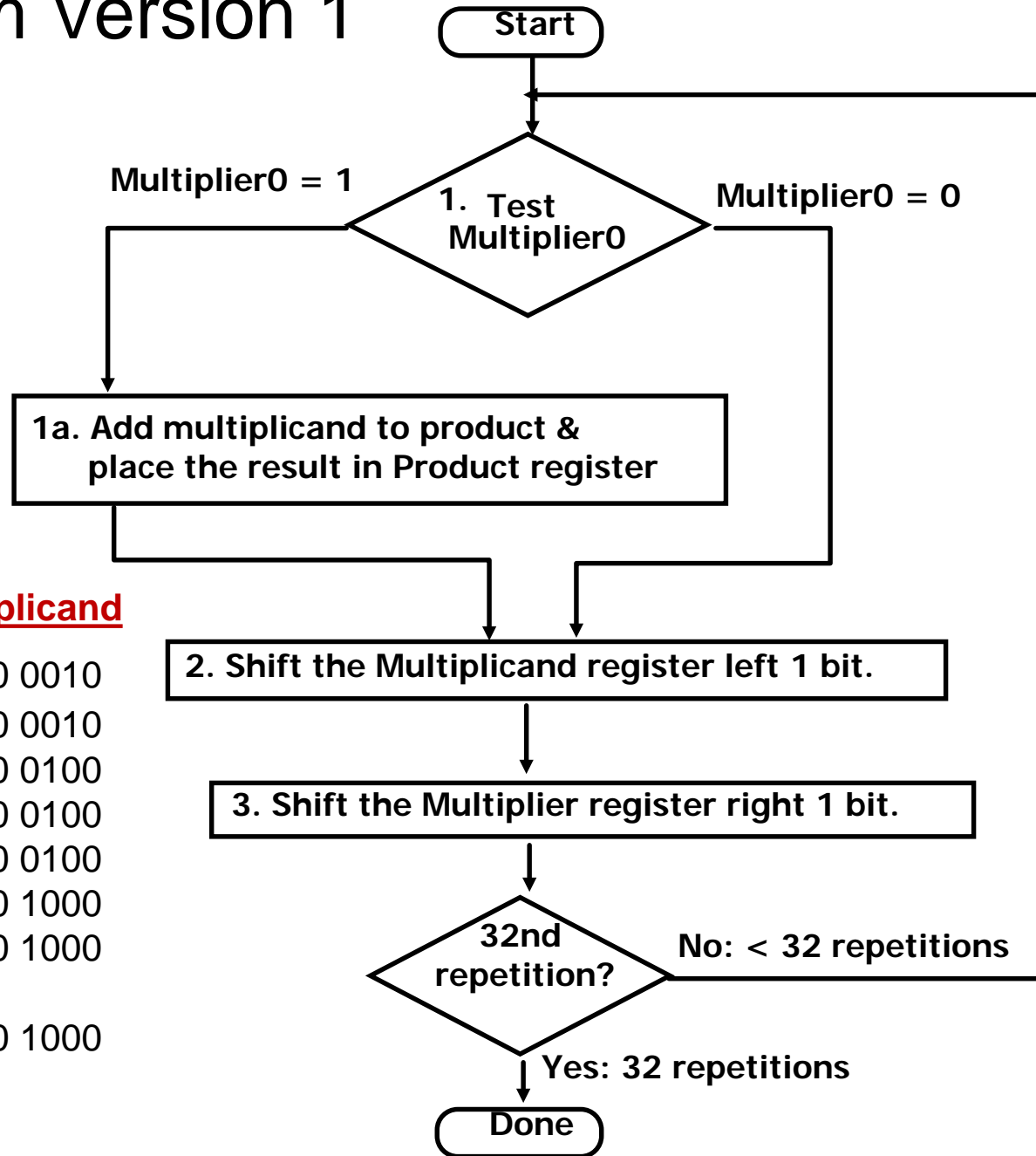
Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit Adder, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

Multiply Algorithm Version 1



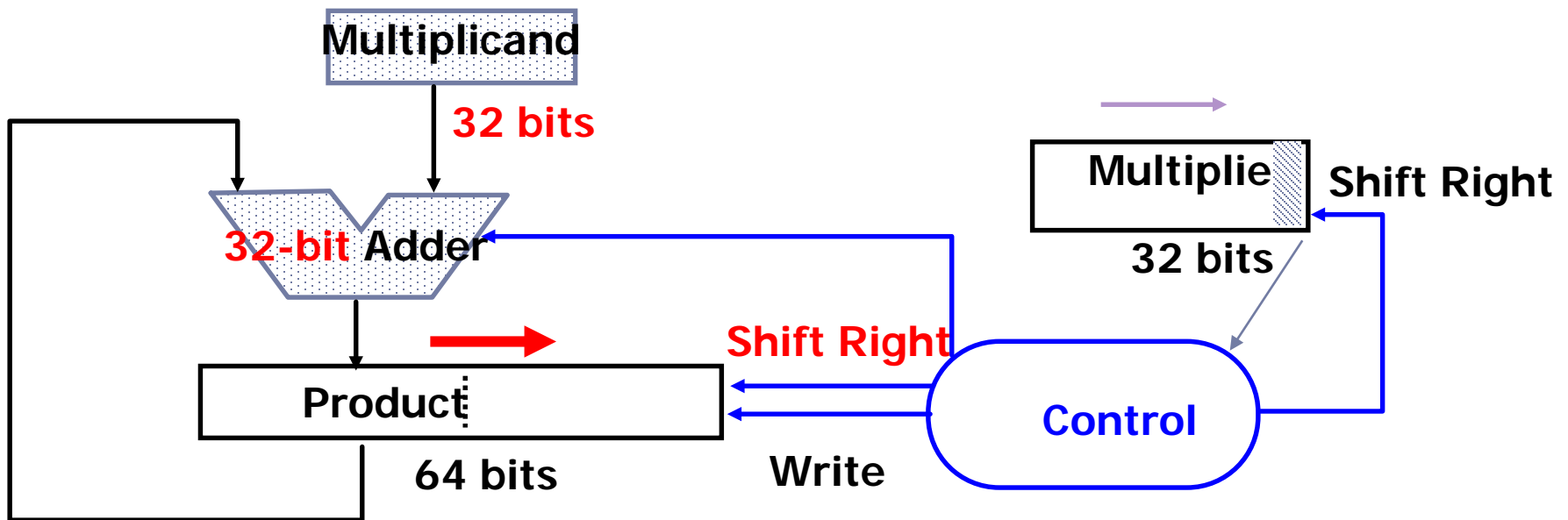
	<u>Product</u>	<u>Multiplier</u>	<u>Multiplicand</u>
	0000 0000	0011	0000 0010
1:	0000 0010	0011	0000 0010
2:	0000 0010	0011	0000 0100
3:	0000 0010	0001	0000 0100
1:	0000 0110	0001	0000 0100
2:	0000 0110	0001	0000 1000
3:	0000 0110	0000	0000 1000
	0000 0110	0000	0000 1000

Observations on Multiply Version 1

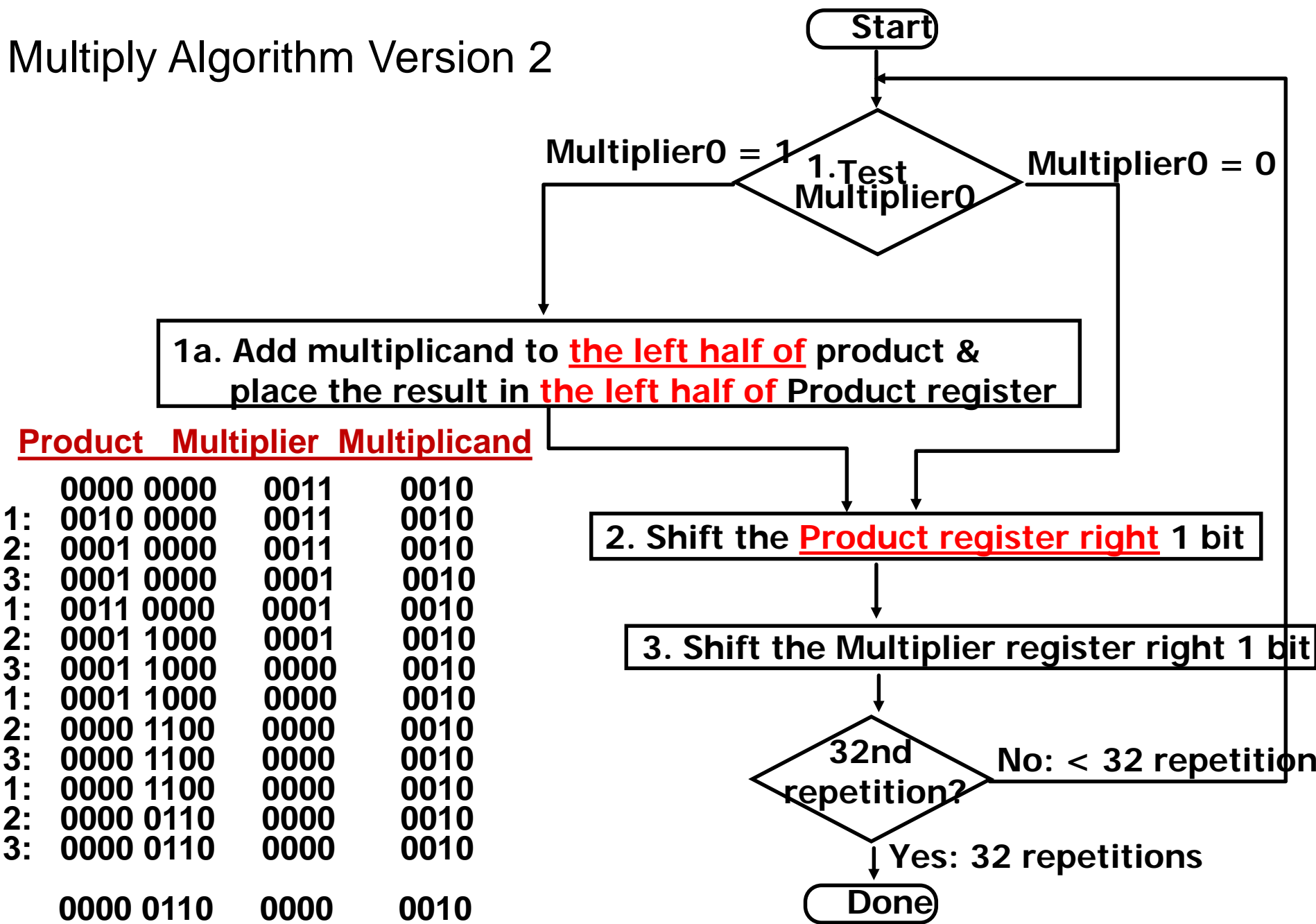
- 1 clock per cycle => ≈ 100 clocks per multiply because of 32 repetitions, 3 steps in one repetition
 - Ratio of add/sub to multiply is from 5:1 to 100:1
 - Slow
- 0's inserted in the rightmost bit of multiplicand as shifting left
 - => least significant bits of product never changed once formed
- 1/2 bits in multiplicand always 0
 - MSB are 0s at the beginning
 - 0 is inserted in LSB as multiplicand shifting left
 - => 64-bit multiplicand register is wasted
 - => 64-bit adder is wasted
- **Instead of shifting multiplicand to left, let's shift product to right**

MULTIPLY HARDWARE Version 2

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



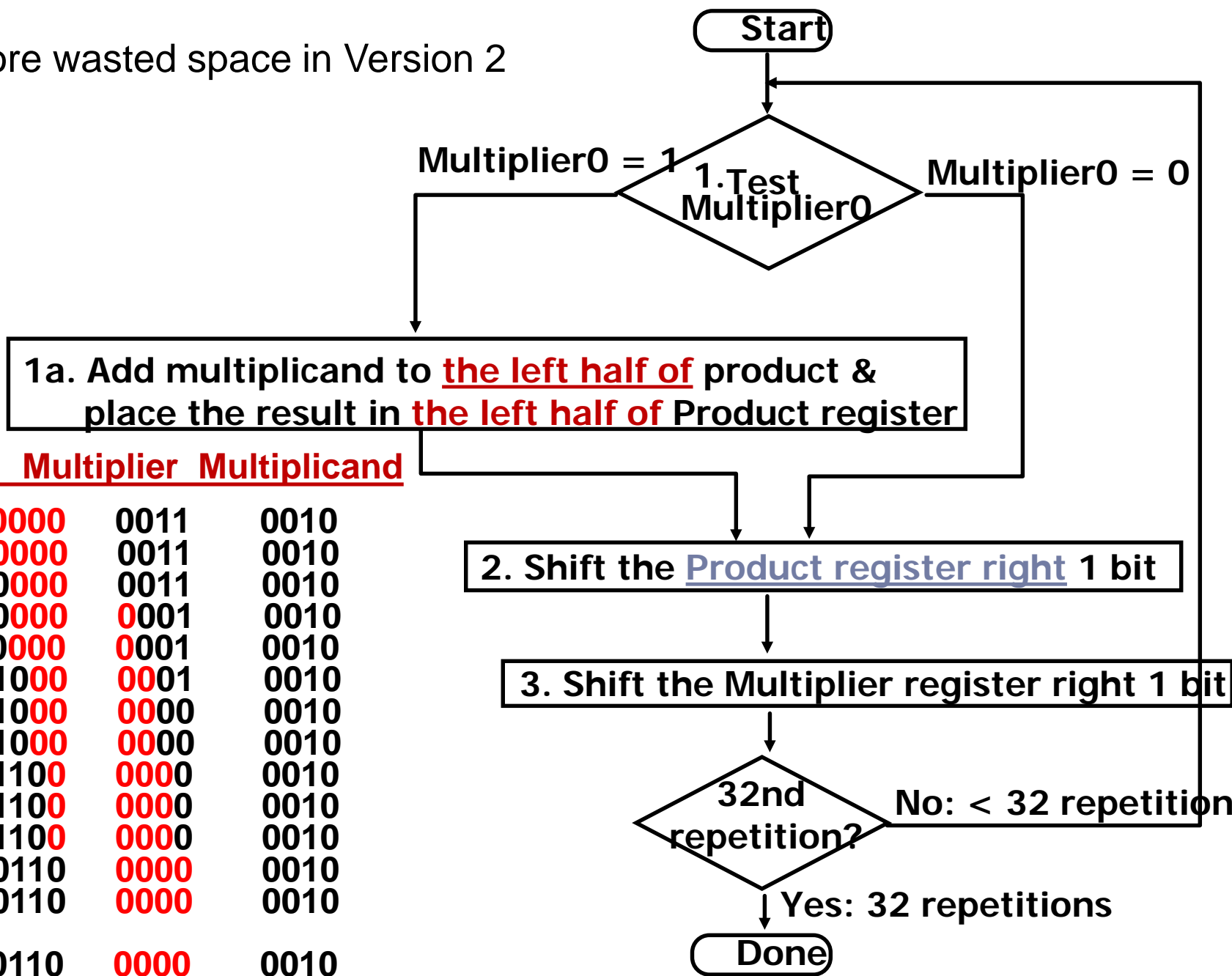
Multiply Algorithm Version 2



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 in Version 2



Product Multiplier Multiplicand

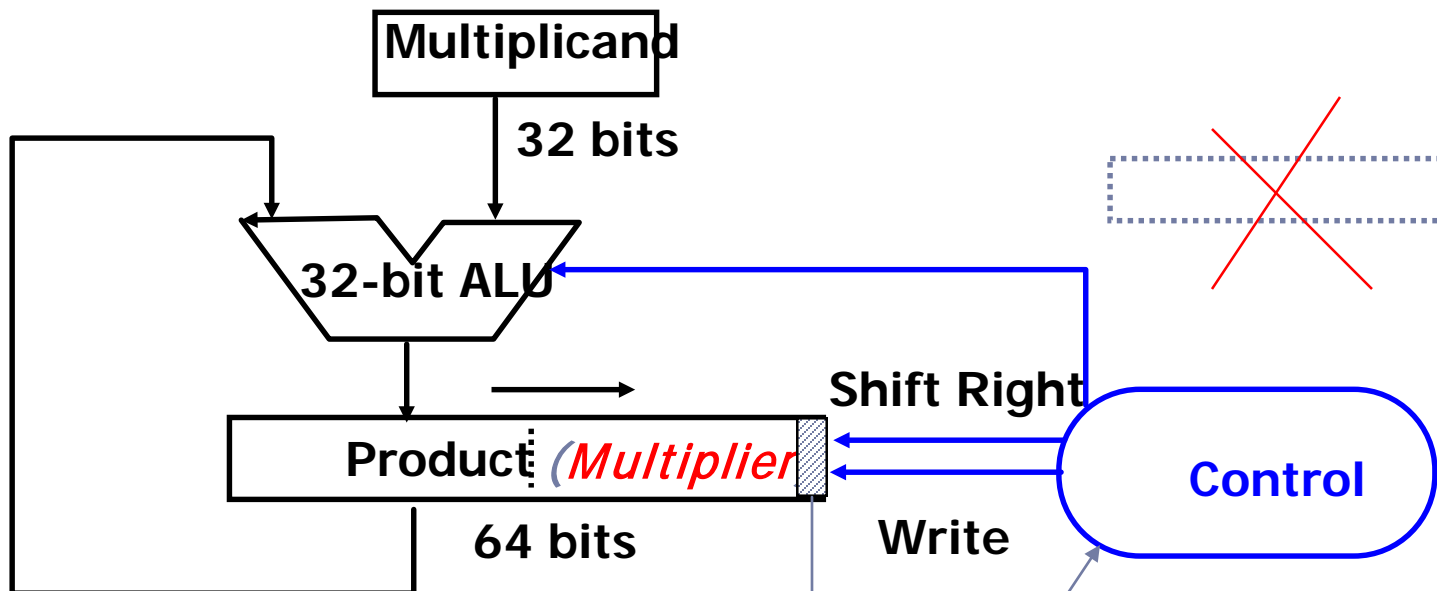
	0000	0000	0011	0010
1:	0010	0000	0011	0010
2:	0001	0000	0011	0010
3:	0001	0000	0001	0010
1:	0011	0000	0001	0010
2:	0001	1000	0001	0010
3:	0001	1000	0000	0010
1:	0001	1000	0000	0010
2:	0000	1100	0000	0010
3:	0000	1100	0000	0010
1:	0000	1100	0000	0010
2:	0000	0110	0000	0010
3:	0000	0110	0000	0010
	0000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

Observations on Multiply Version 2

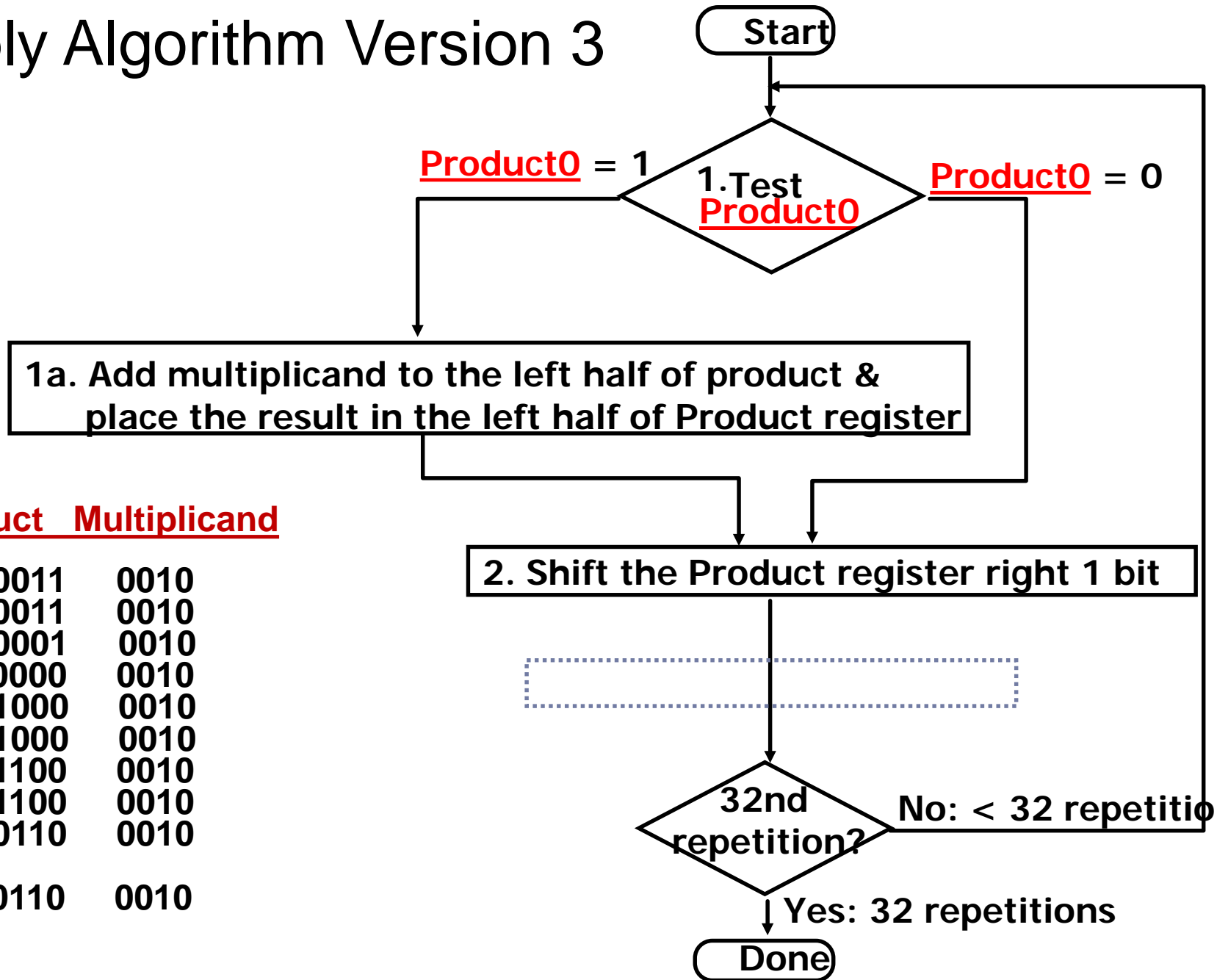
- Product register wastes space that exactly matches size of multiplier
=> combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

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



Multiply Algorithm Version 3



Product Multiplicand

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