



COMP 303

Computer Architecture

Lecture 6



MULTIPLY (unsigned)

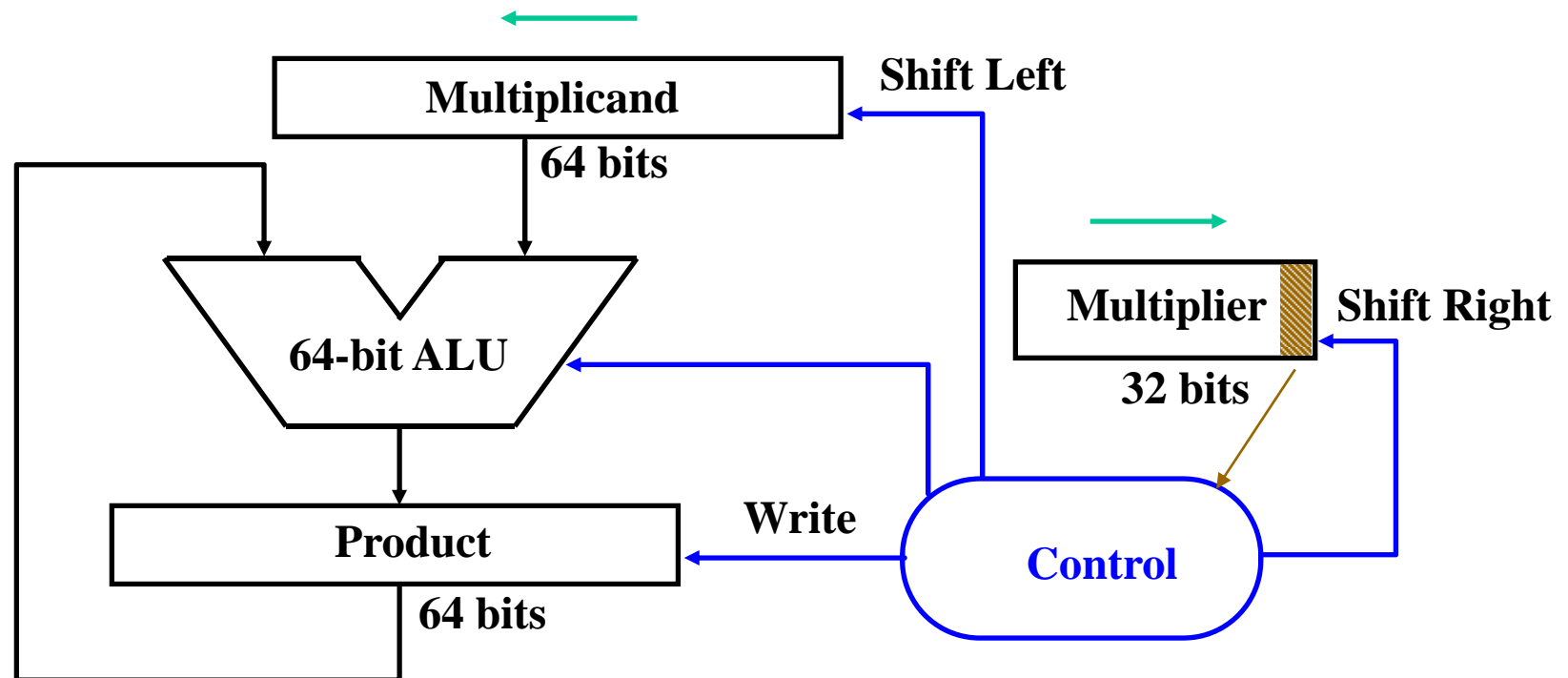
- Paper and pencil example (unsigned):

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

- 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**

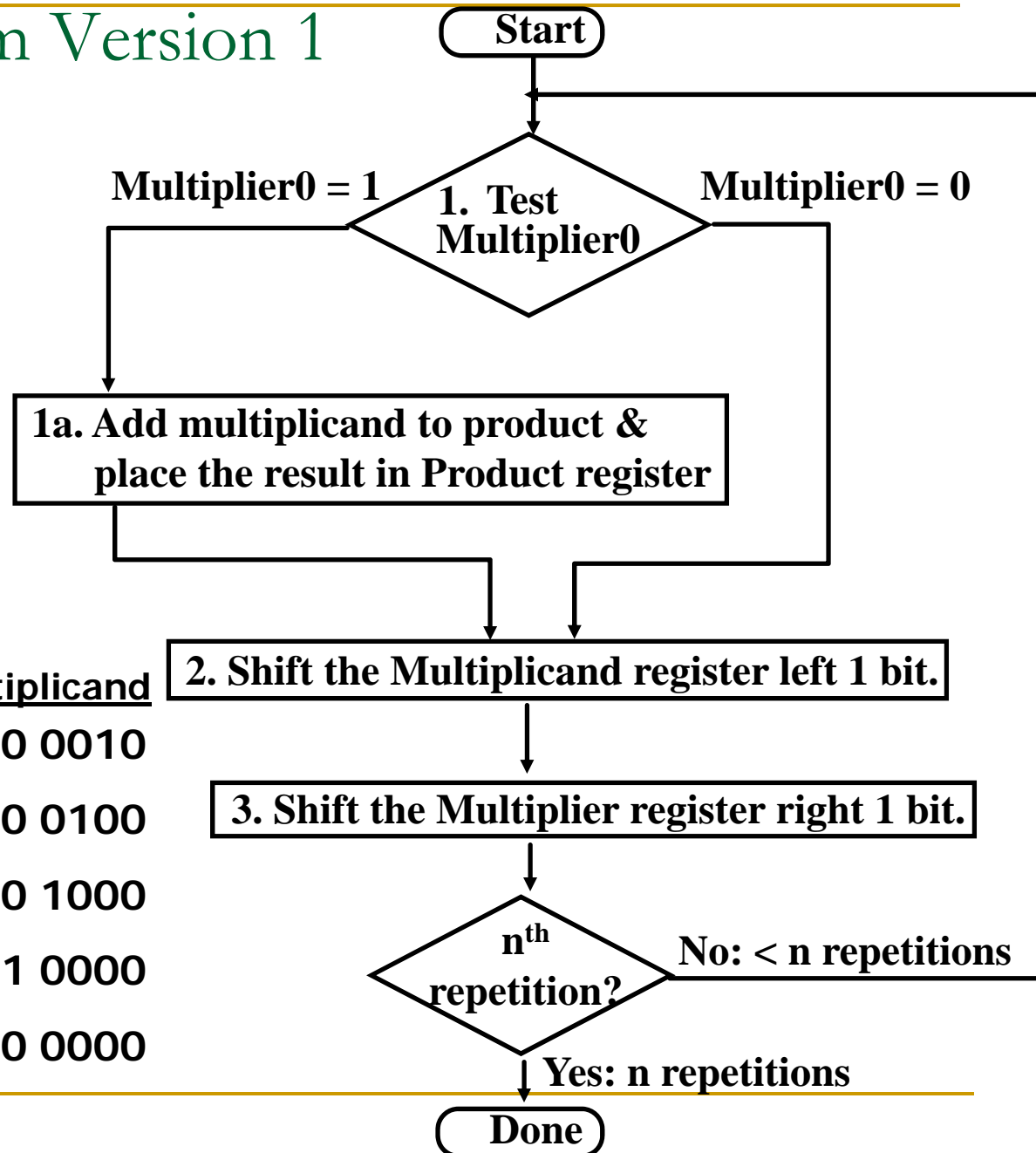
Unsigned shift-add multiplier (version 1)

- 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

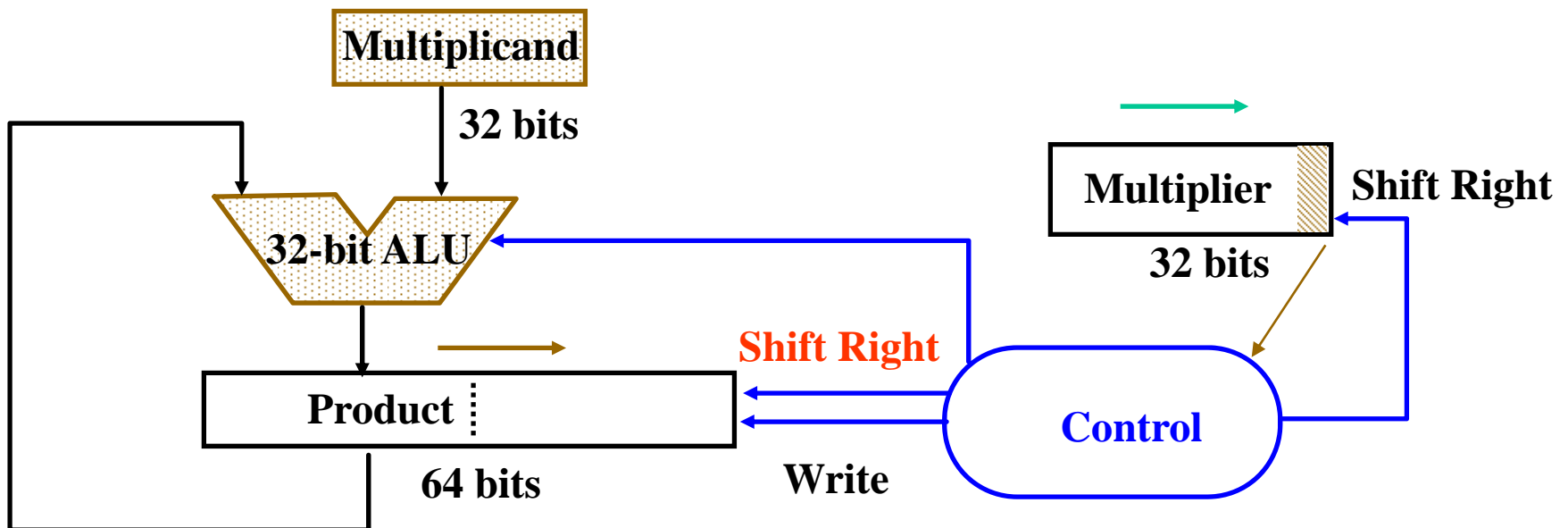
0000 0110

Observations on Multiply Version 1

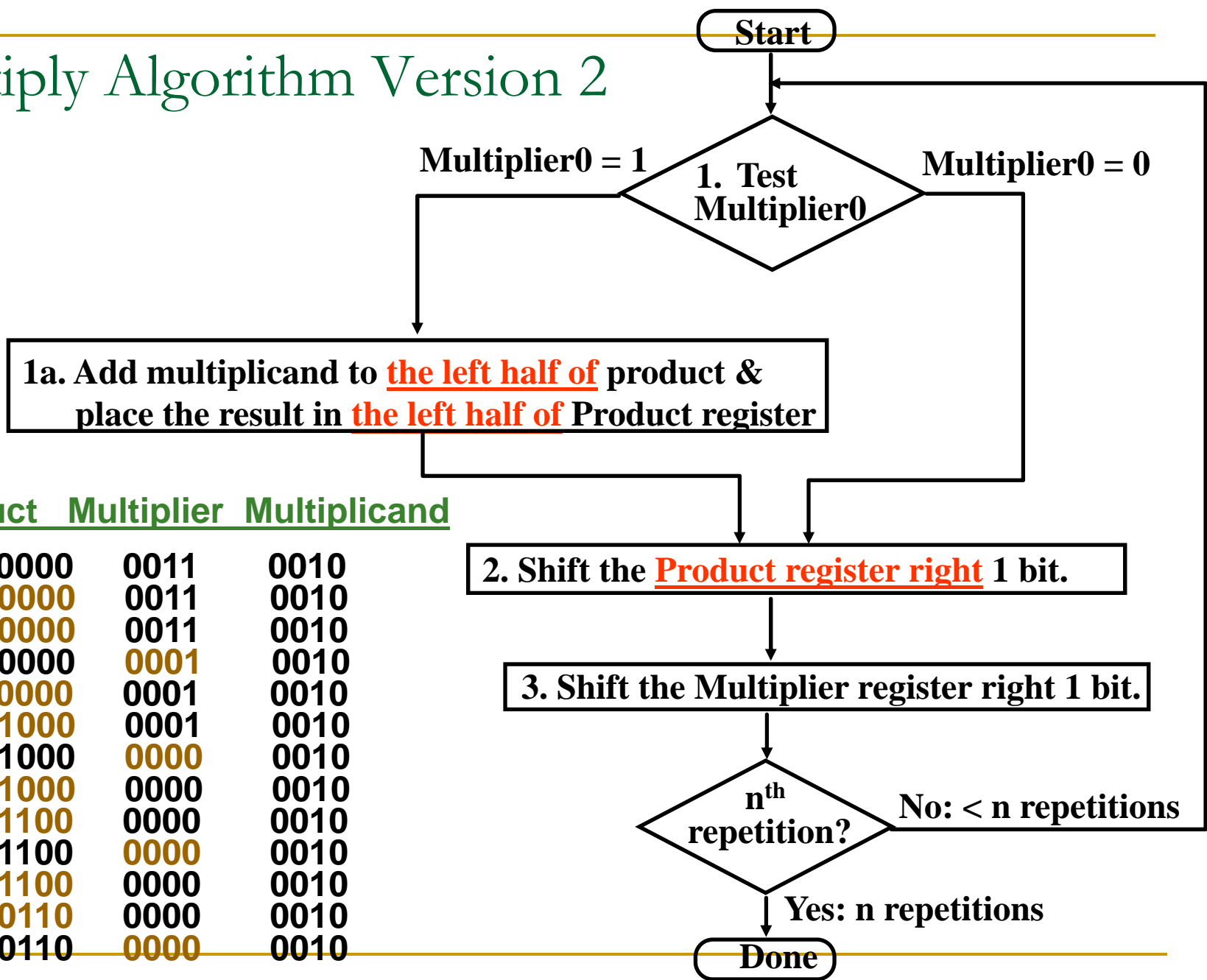
- 1/2 bits in multiplicand always 0
=> 64-bit adder is wasted
- 0's inserted into the least significant bit of multiplicand as shifted => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right.

MULTIPLY HARDWARE Version 2

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



Multiply Algorithm Version 2

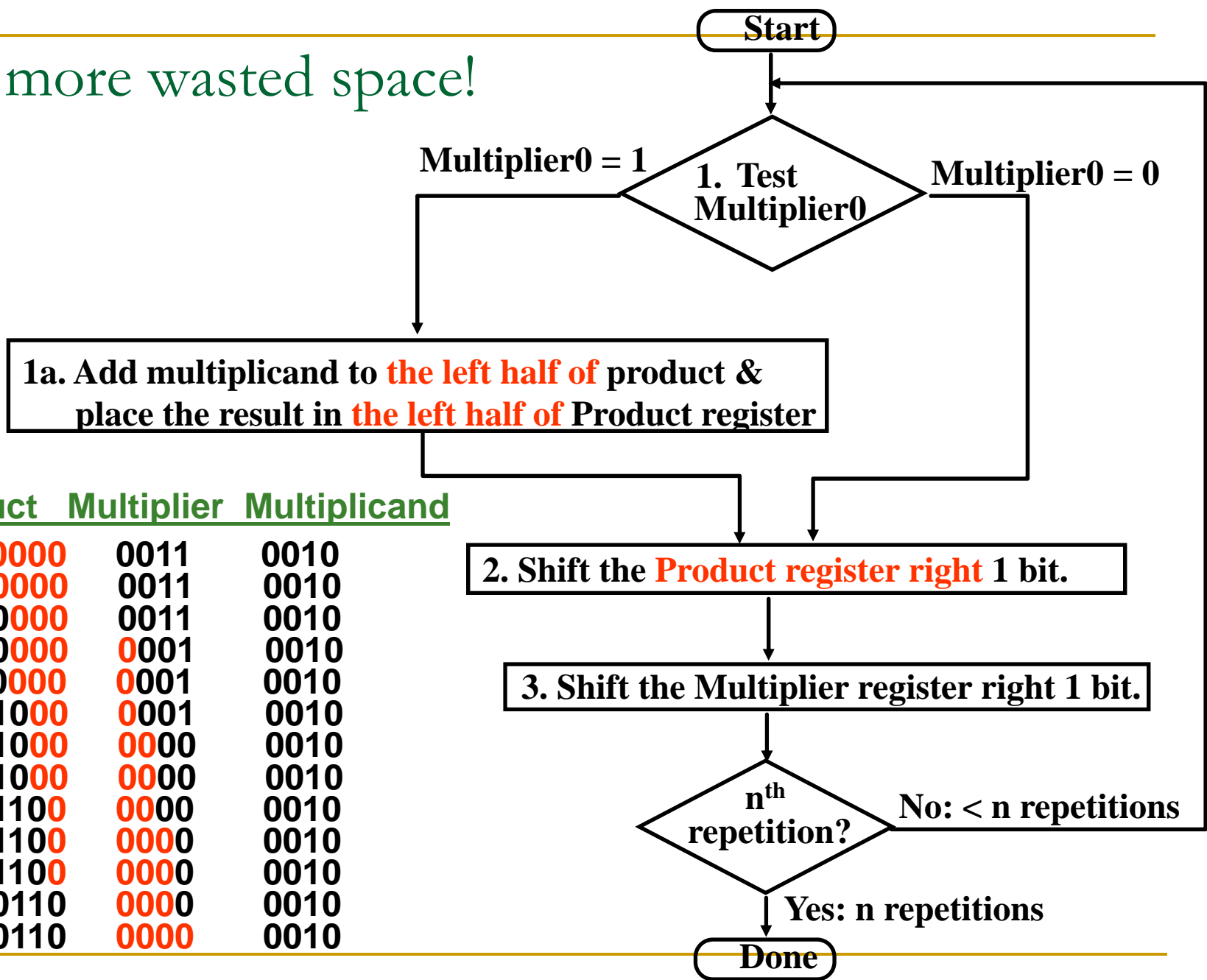


Product Multiplier Multiplicand

	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!



Product Multiplier Multiplicand

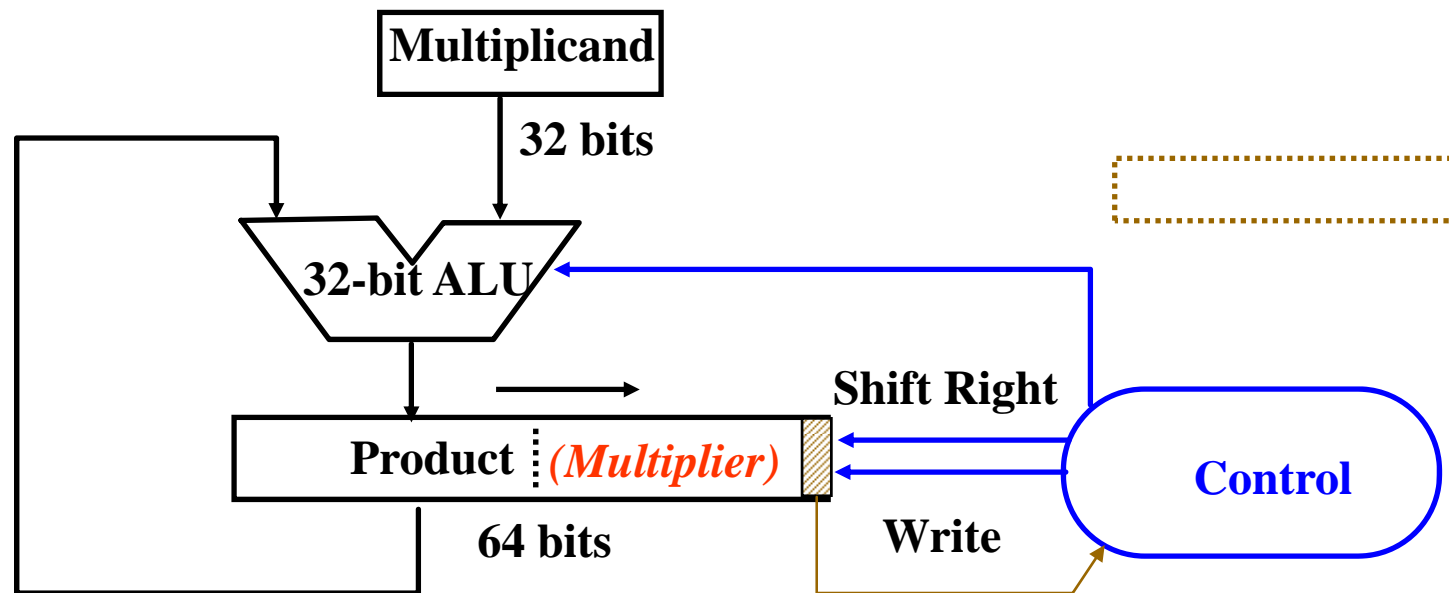
	0000	0000	0011	0010
1:	0010	0000	0011	0010
2:	0001	0000	0011	0010
3:	0001	0000	0001	0010
1:	0011	0000	0001	0010
2:	0001	1000	0001	0010
3:	0001	1000	0000	0010
1:	0001	1000	0000	0010
2:	0000	1100	0000	0010
3:	0000	1100	0000	0010
1:	0000	1100	0000	0010
2:	0000	0110	0000	0010
3:	0000	0110	0000	0010
	0000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

Observations on Multiply Version 2

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

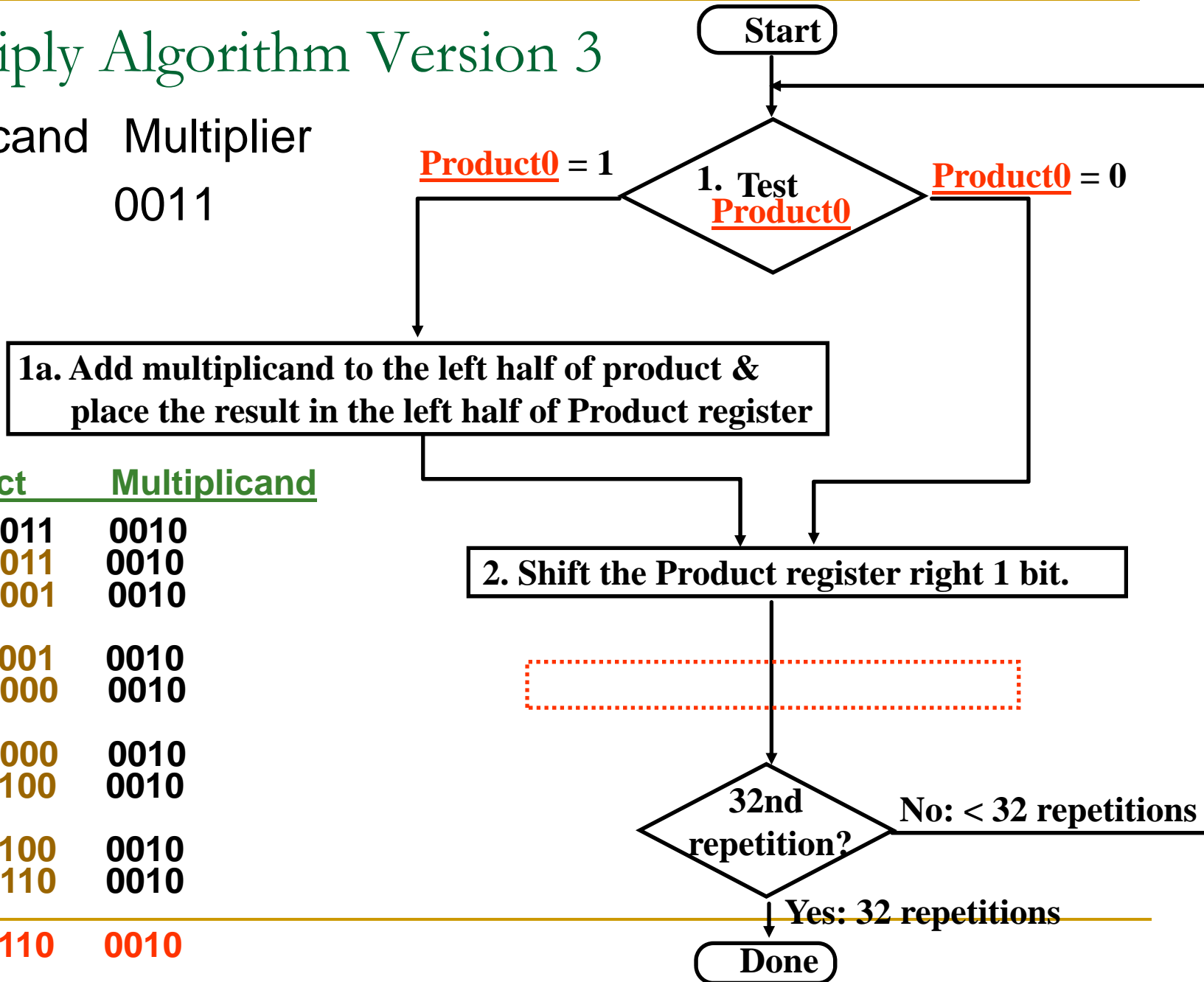
MULTIPLY HARDWARE Version 3

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



Multiply Algorithm Version 3

Multiplicand Multiplier
0010 0011



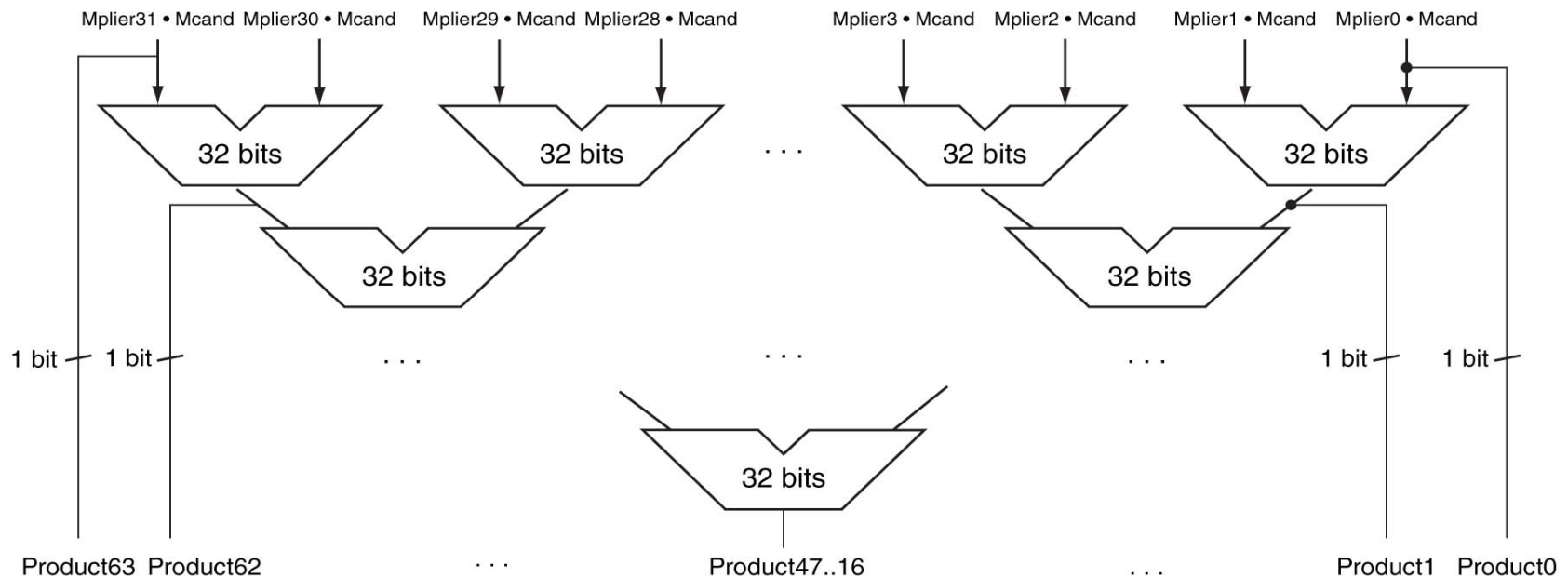
	<u>Product</u>	<u>Multiplicand</u>
	0000 0011	0010
1:	0010 0011	0010
2:	0001 0001	0010
1:	0011 0001	0010
2:	0001 1000	0010
1:	0001 1000	0010
2:	0000 1100	0010
1:	0000 1100	0010
2:	0000 0110	0010
	0000 0110	0010

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)
 - Multiply algorithm 3 will work for signed numbers if partial products are sign-extended as shifted
 - 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

Faster Multiplication

- Whether the multiplicand is to be added or not is known at the beginning of the operation
- Provide a 32-bit adder for each bit of the multiplier
- One input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder.
- Speed: just the overhead of a clock for each bit of the product.
 $\log_2(32)$



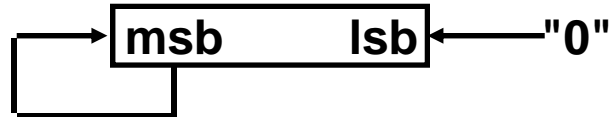
Shifters

Two kinds:

logical-- value shifted in is always "0"

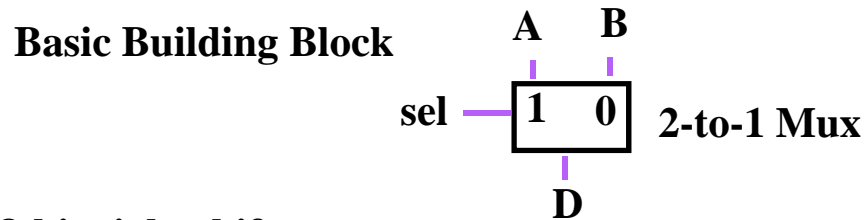


arithmetic-- on right shifts, sign extend

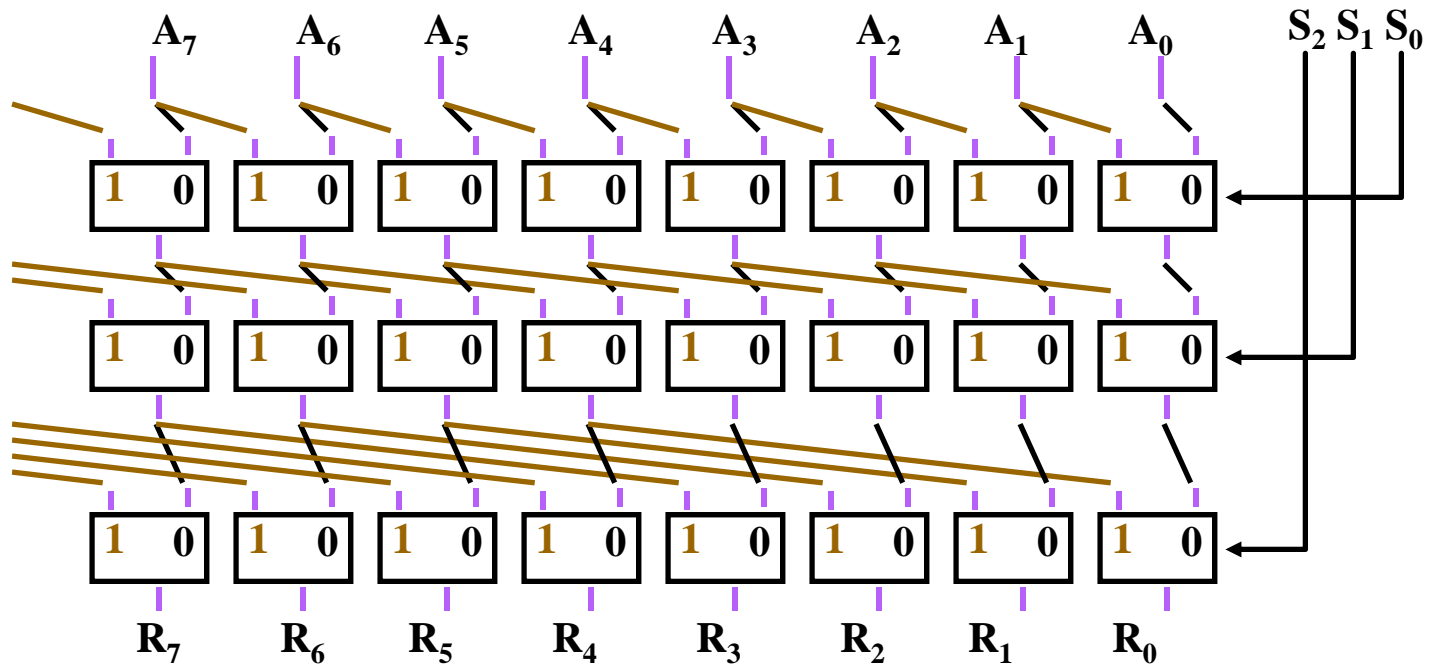


Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Combinational Shifter from MUXes



8-bit right shifter



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

Unsigned Divide: Paper & Pencil

	1001	Quotient
Divisor 1000	$\overline{) 1001010}$	Dividend
	$\underline{-1000}$	
	10	
	101	
	1010	
	$\underline{-1000}$	
	10	Remainder (or Modulo result)

See how big a number can be subtracted, creating quotient bit on each step

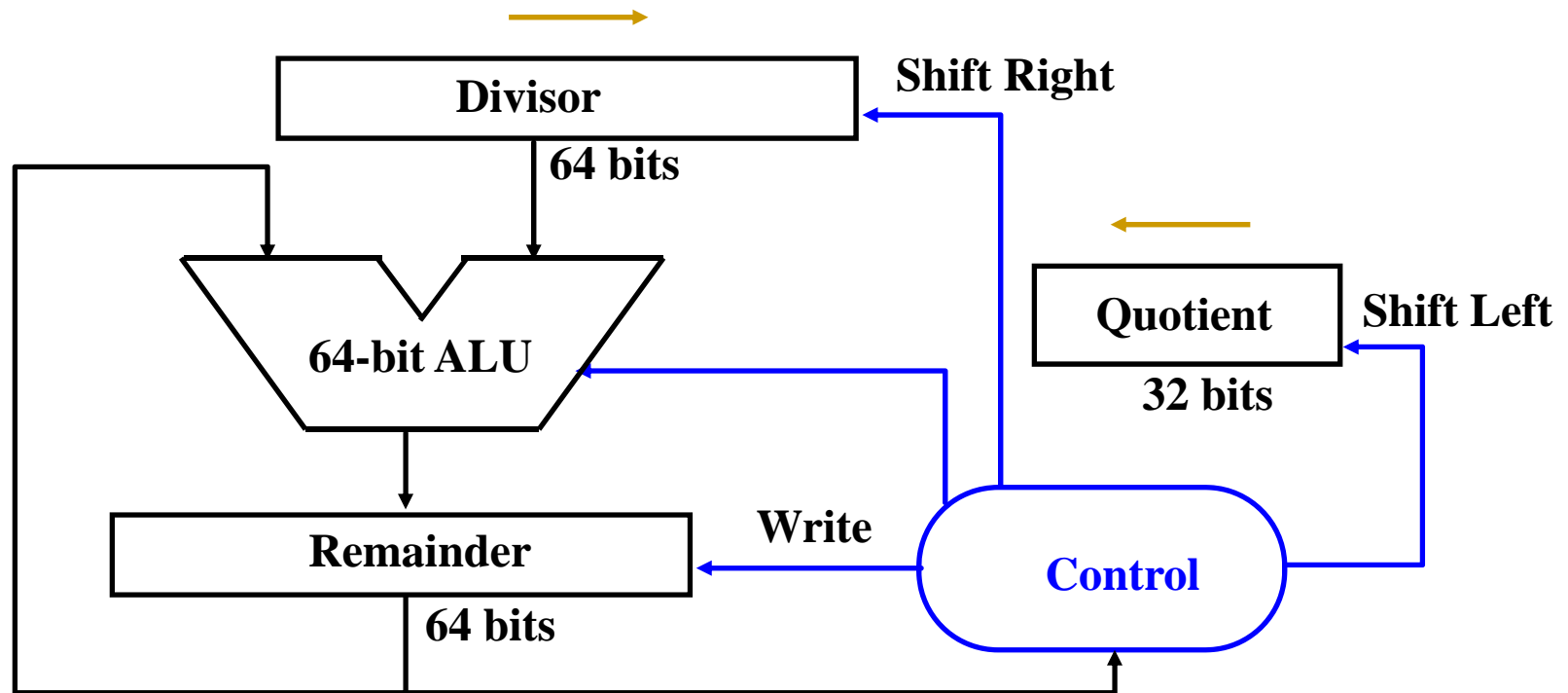
Binary $\Rightarrow 1 * \text{divisor}$ or $0 * \text{divisor}$

Dividend = Quotient x Divisor + Remainder

3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

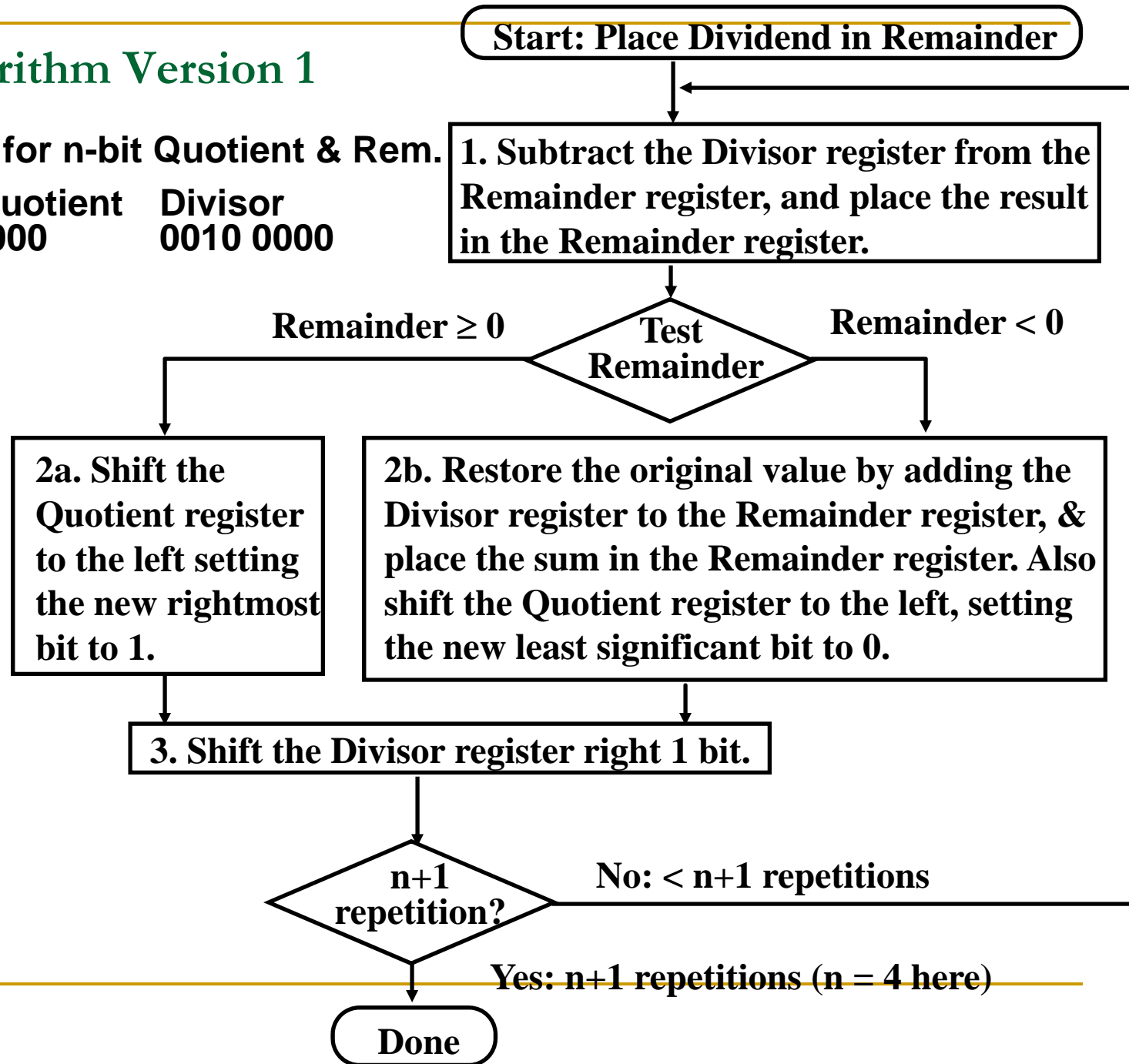
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

■ Takes $n+1$ steps for n -bit Quotient & Rem.

Remainder	Quotient	Divisor
0000 0111	0000	0010 0000



Observations on Divide Version 1

- 1/2 bits in divisor always 0
 - => 1/2 of 64-bit adder is wasted
 - => 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise too big)
 - => switch order to shift first and then subtract, can save 1 iteration

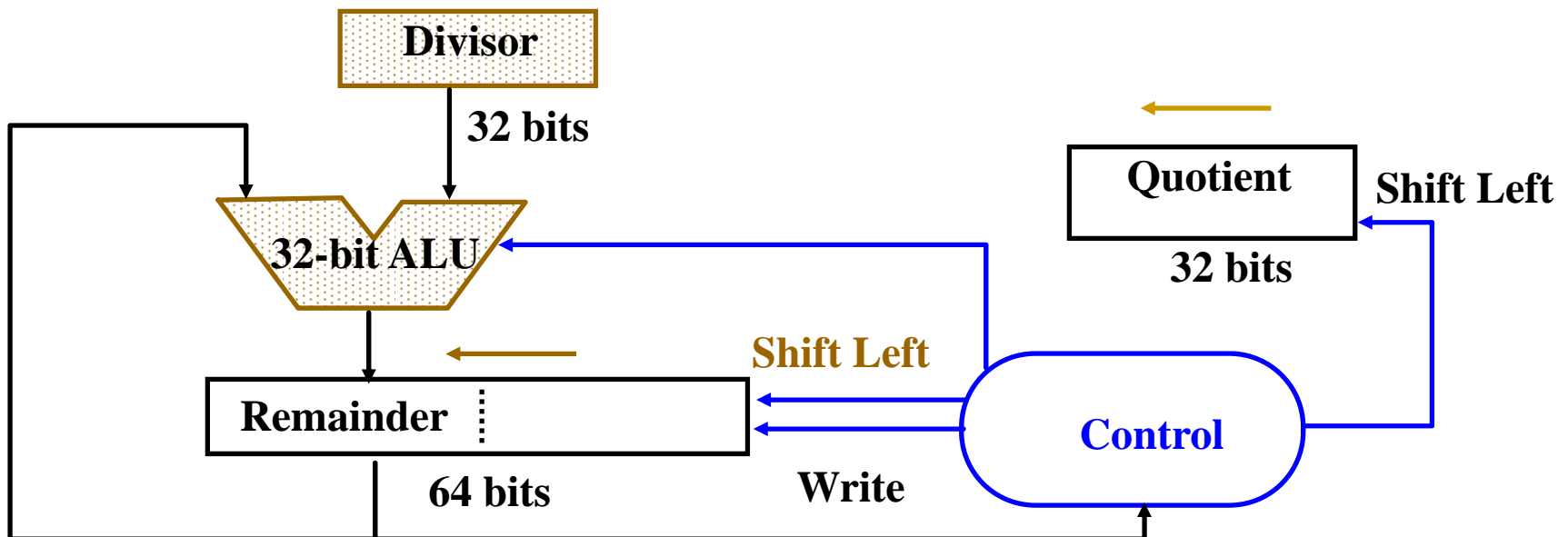
Divide: Paper & Pencil

		01010	Quotient
Divisor	0001	$\overline{)00001010}$	Dividend
		0001	
		-0001	
		0000	
		0001	
		-0001	
		$\underline{0}$	
		00	Remainder (or Modulo result)

- Notice that there is no way to get a 1 in leading digit!
(this would be an overflow, since quotient would have $n+1$ bits)

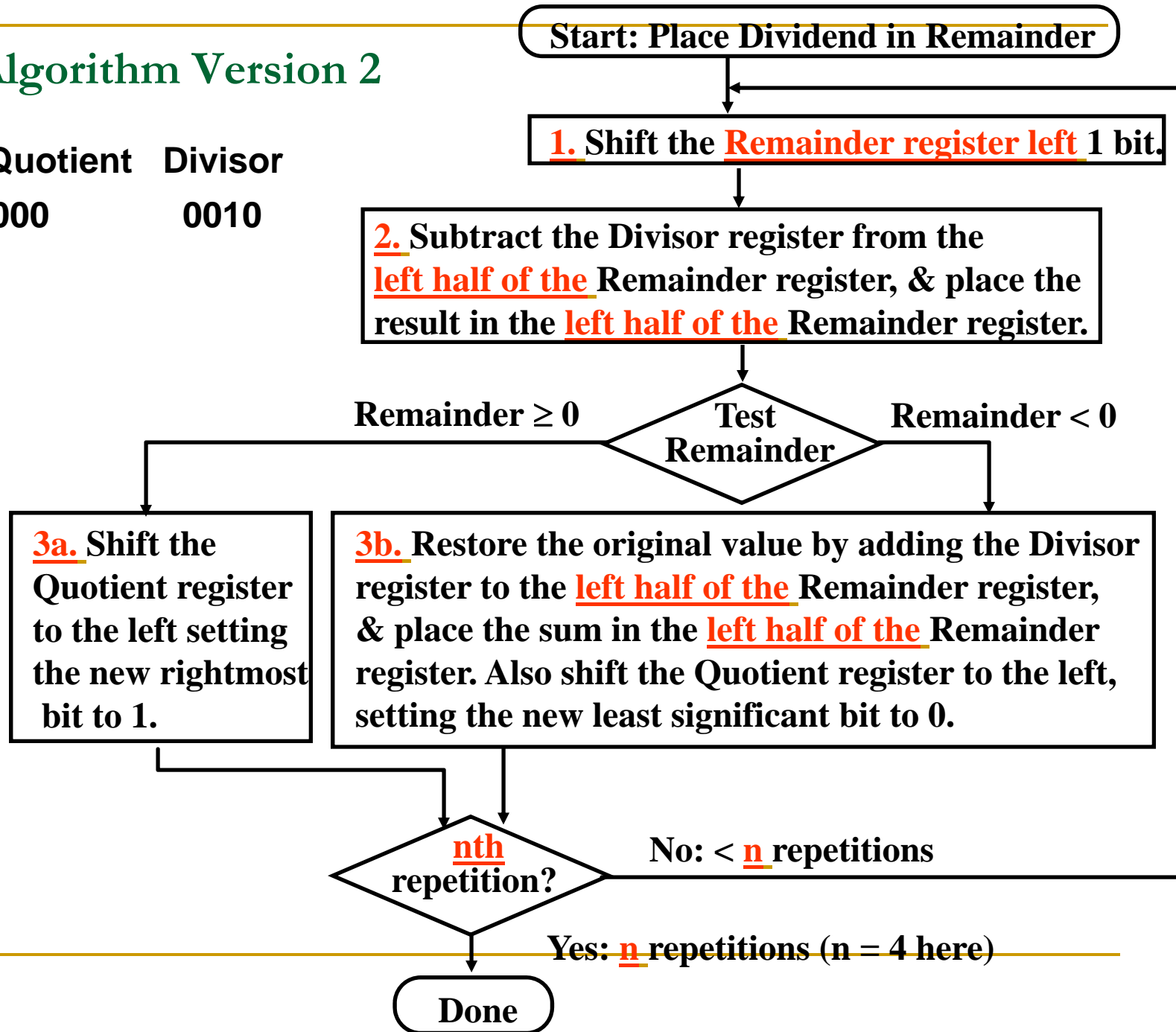
DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 2

Remainder Quotient Divisor
0000 0111 0000 0010

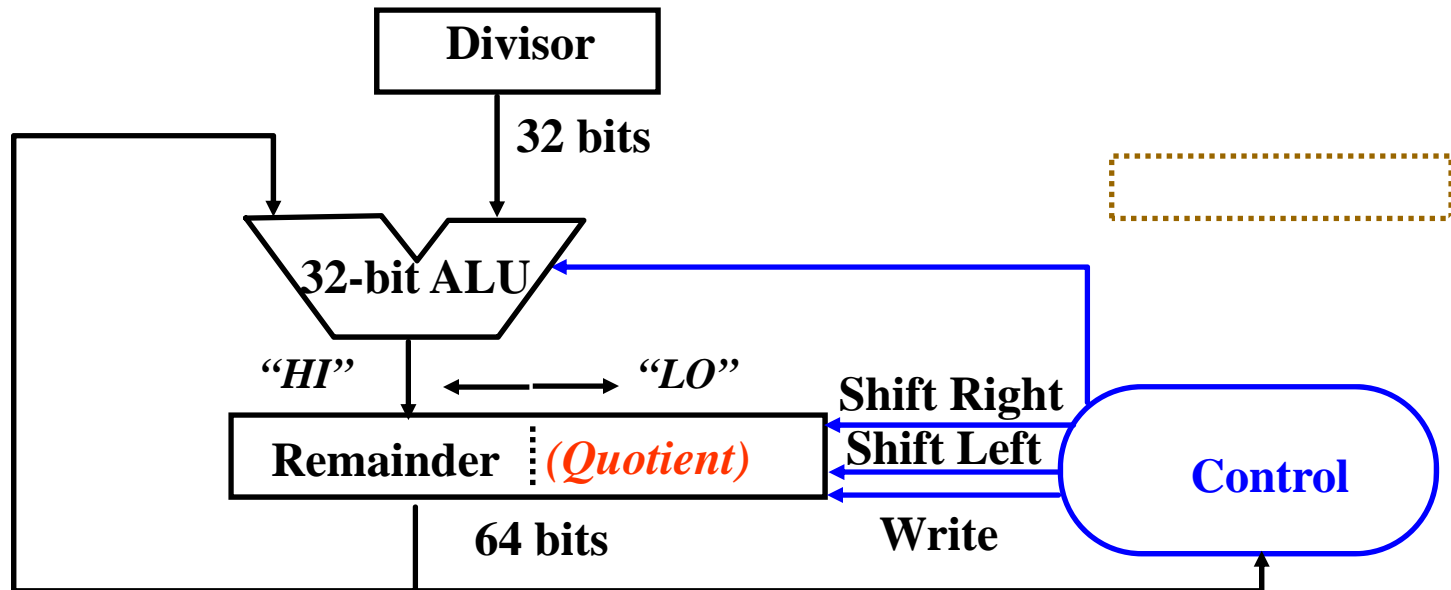


Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
 - Start by shifting the Remainder left as before.
 - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
 - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
 - Thus the final correction step must shift back only the remainder in the left half of the register

DIVIDE HARDWARE Version 3

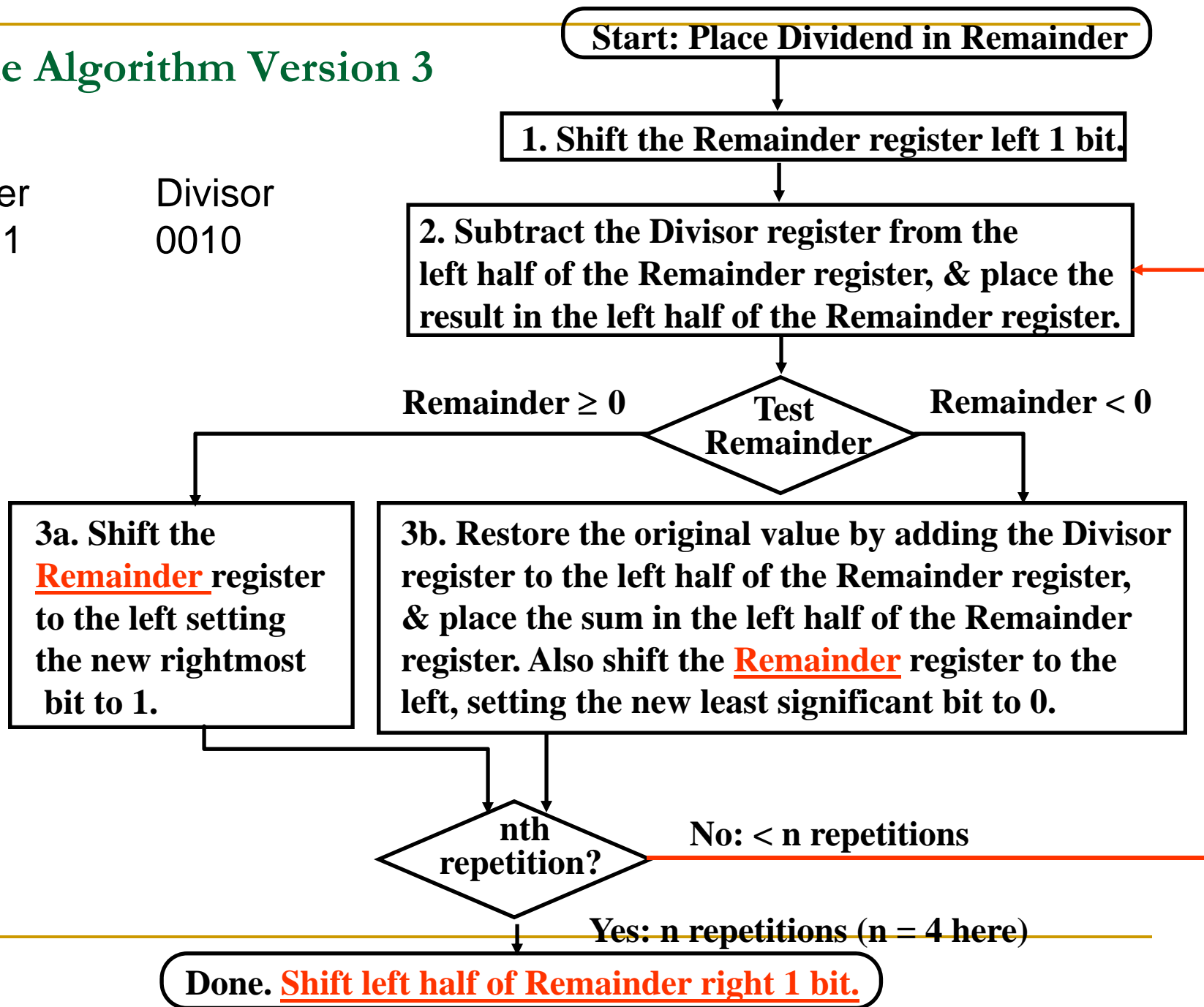
- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (Q-bit Quotient reg)



Divide Algorithm Version 3

Remainder
0000 0111

Divisor
0010



Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
 - Note: Dividend and Remainder must have same sign
 - Note: Quotient negated if Divisor sign & Dividend sign disagree
e.g., $-7 \div 2 = -3$, remainder = -1