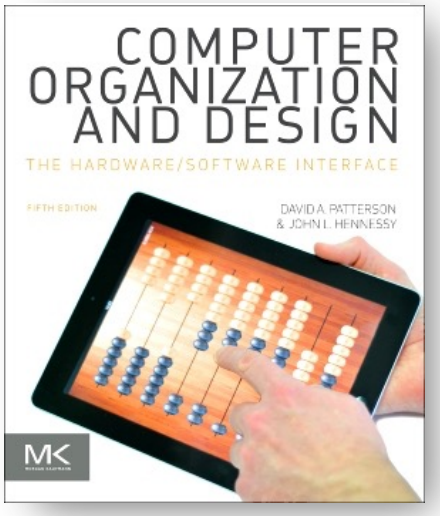


ESE 345 Computer Architecture

Multiply, Divide

MIPS Arithmetic Instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3,	Lo = quotient, Hi = remainder
			Hi = $2 mod $3
divide unsigned	divu $2,$3	Lo = $2 ÷ $3,	Unsigned quotient & remainder
			Hi = $2 mod $3
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

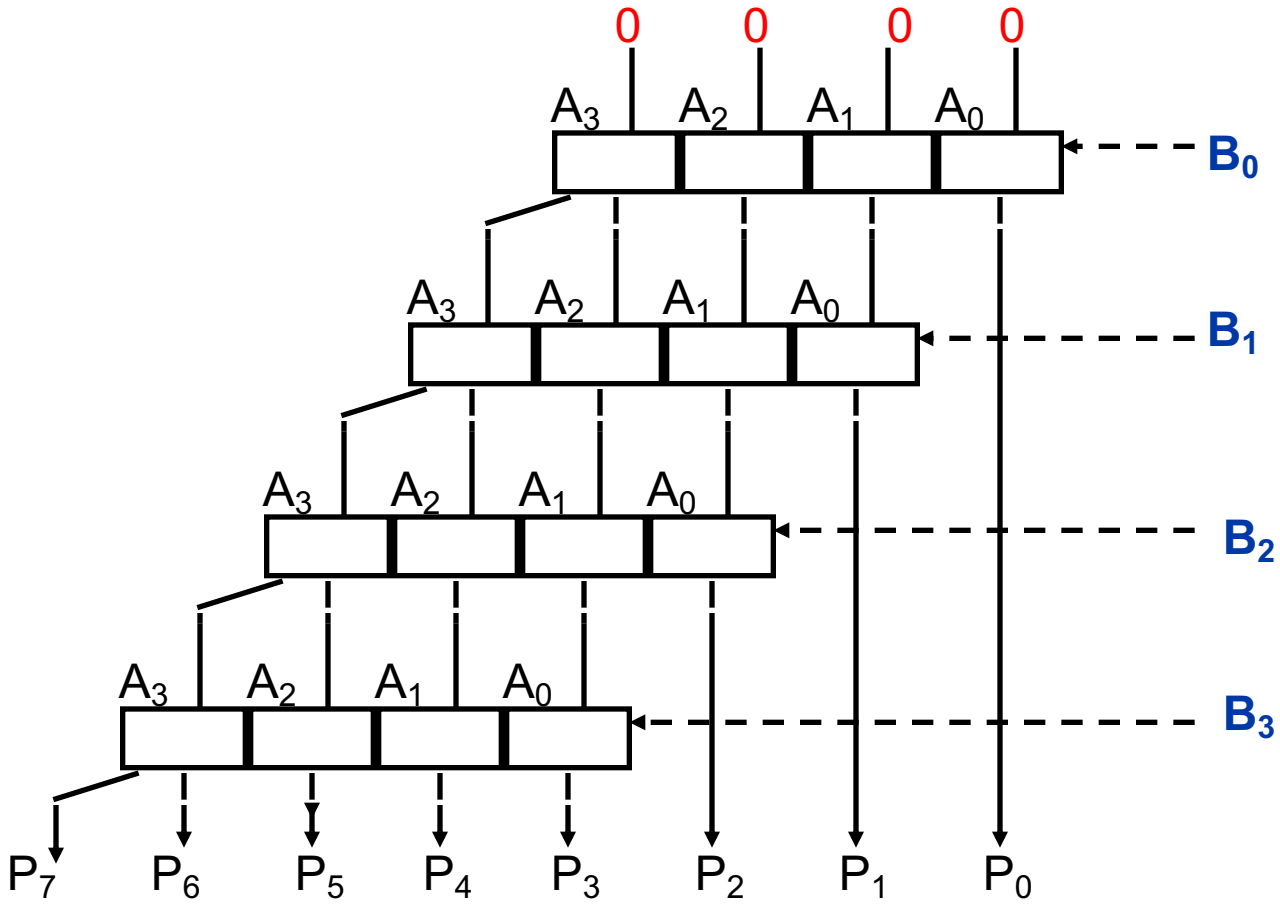
MULTIPLY (Unsigned)

- Paper and pencil example (unsigned):

```
Multiplicand      1000
  Multiplier    x 1001
                -------
                  1000
                 0000
                0000
               1000
Product        --------
               01001000
```

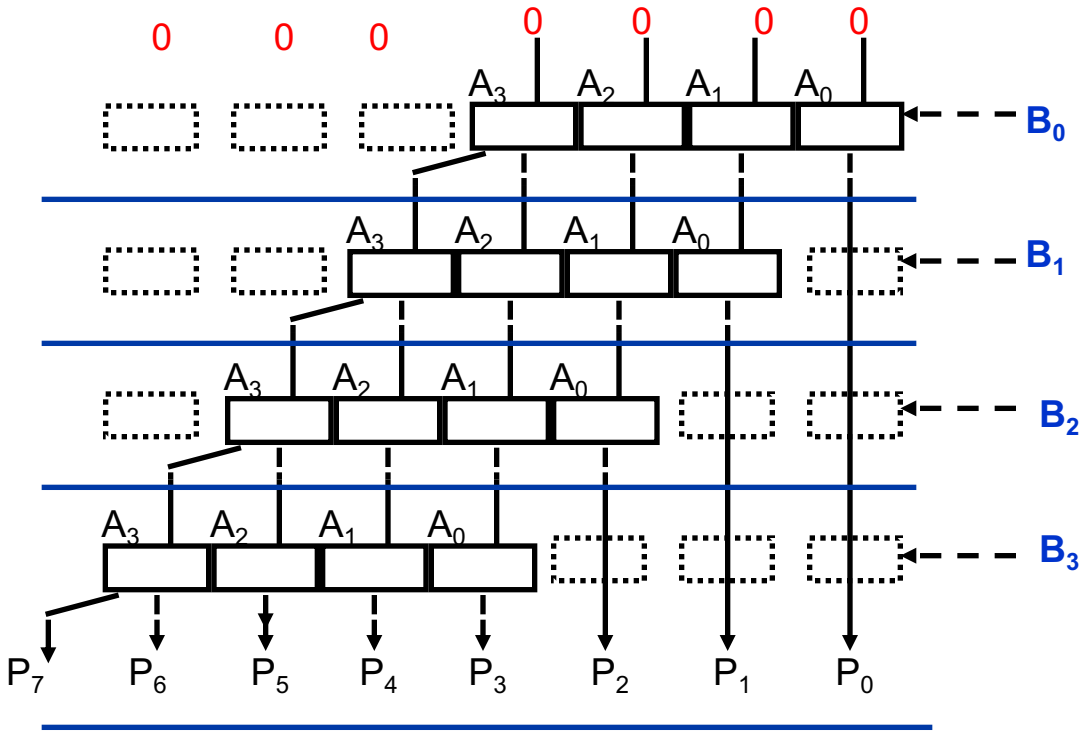
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - •0 => place 0 (0 x multiplicand)
  - •1 => place a copy (1 x multiplicand)
- 4 versions of multiply hardware & algorithm:
  - •successive refinement

Unsigned Combinational Multiplier



- Stage i accumulates $A * 2^i$ if $B_i == 1$
- Q: How much hardware for 32 bit multiplier? Critical path?

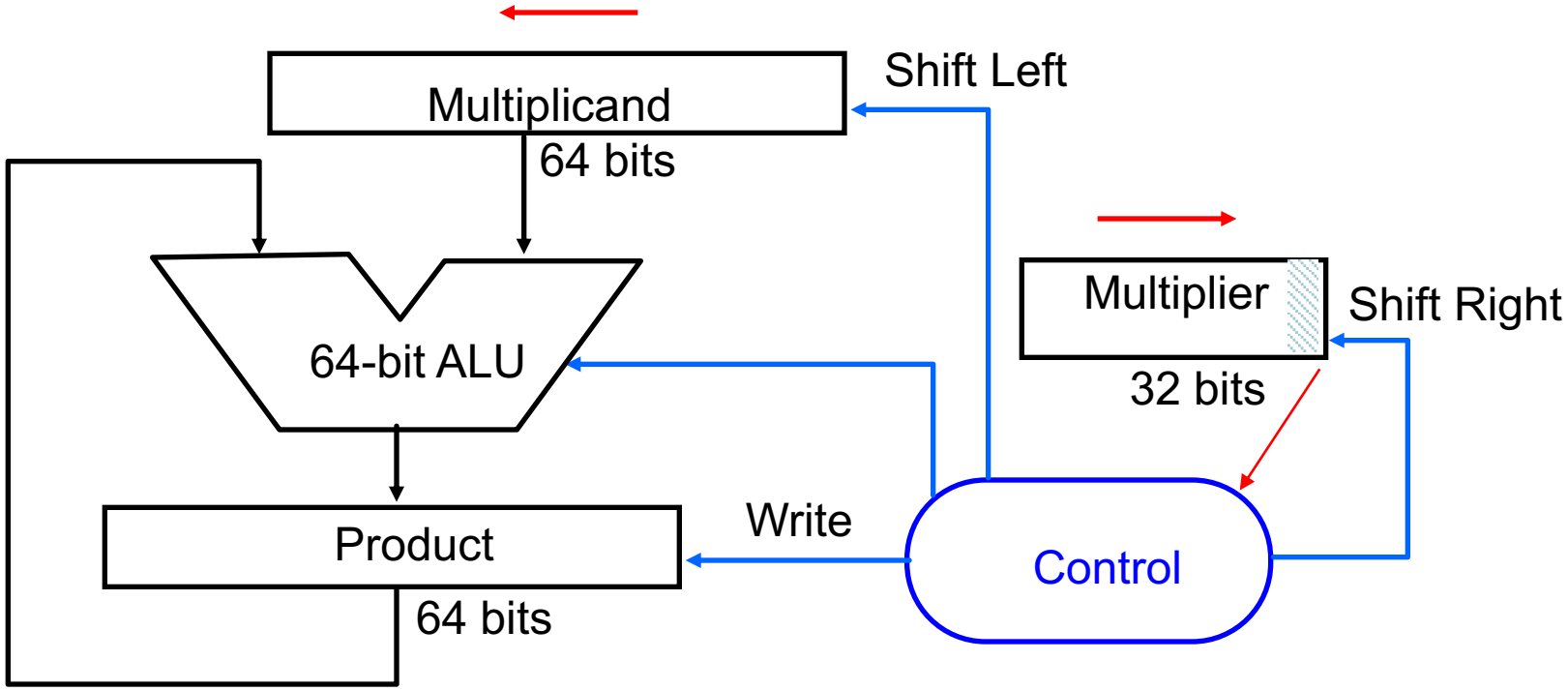
How Does It Work?



- at each stage shift A left (x 2)
- use next bit of B to determine whether to add in shifted multiplicand
- accumulate 2n bit partial product at each stage

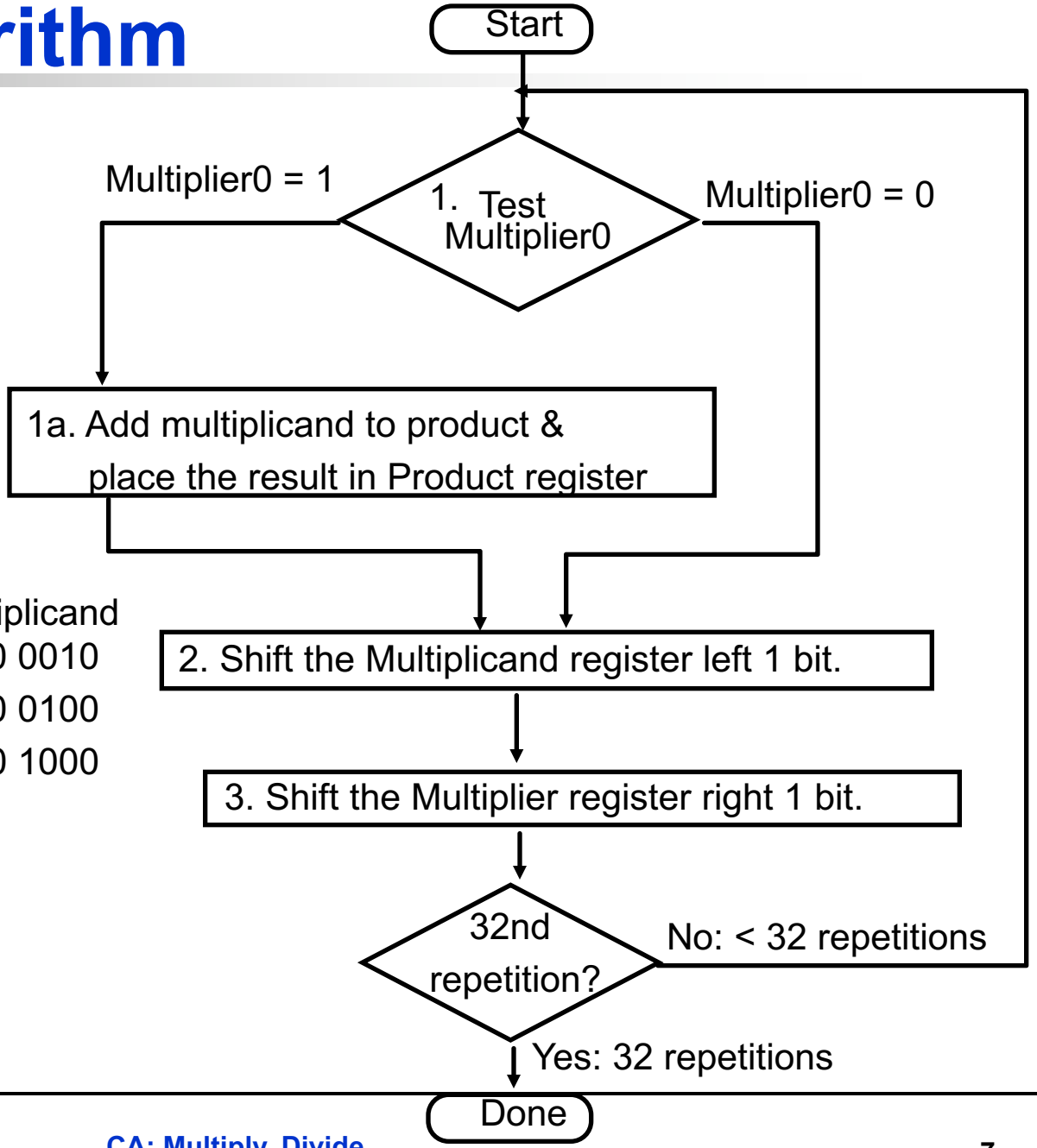
Unisigned Shift-Add Multiplier (Version 1)

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



Multiplier = datapath + control

Multiply Algorithm Version 1



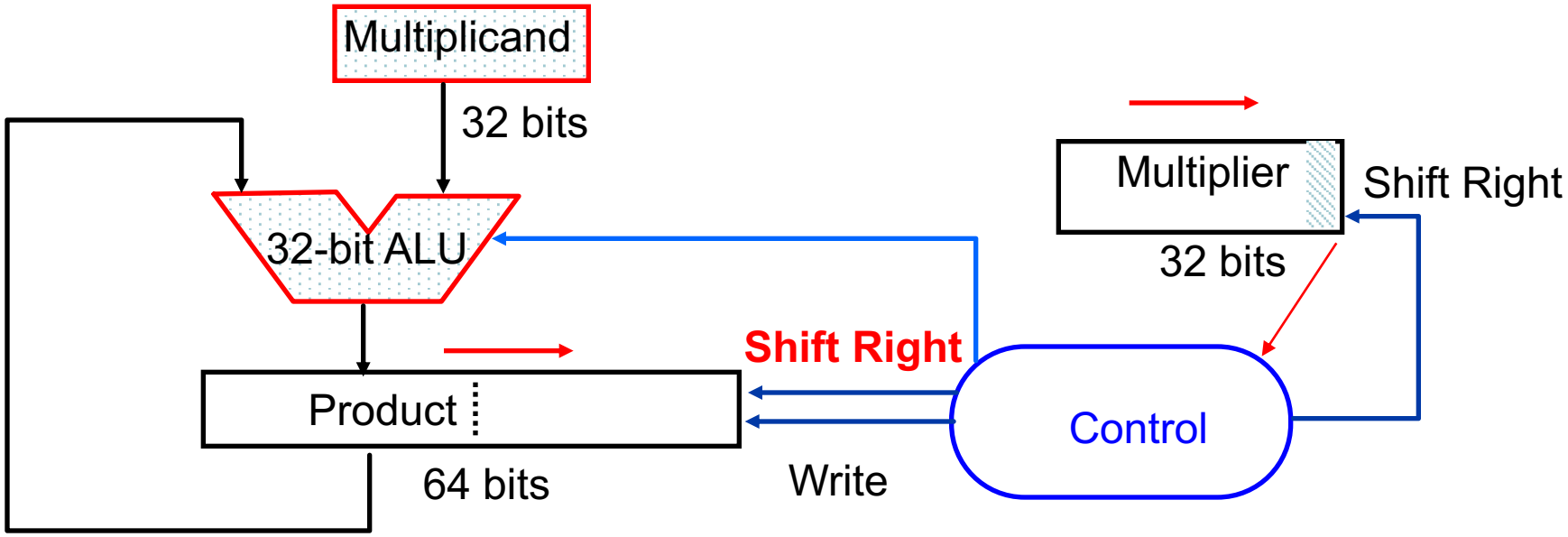
Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

Observations on Multiply Version 1

- 1 clock per cycle => ≈ 100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted, 32-bit “sliding window”
- 0’s inserted in left of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

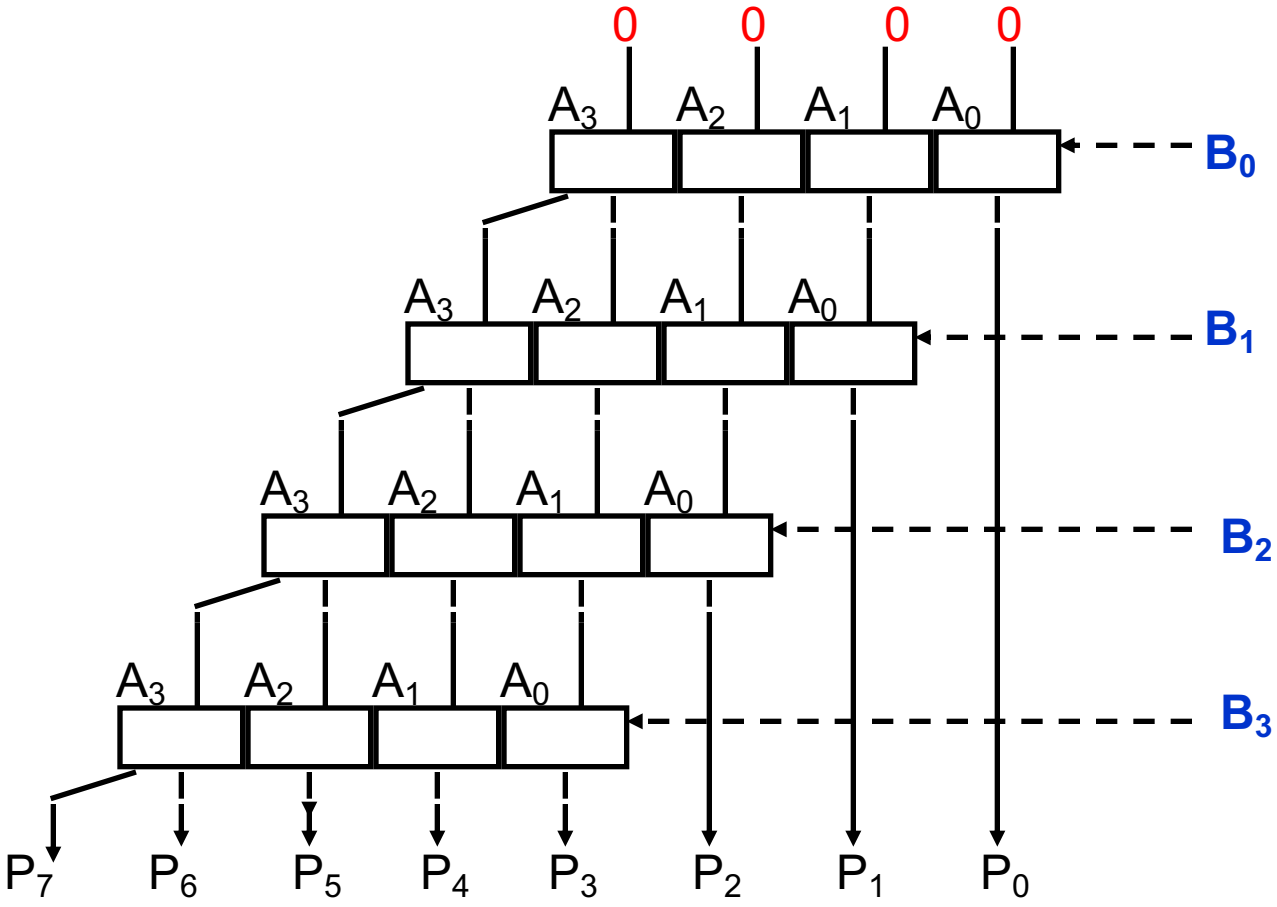
MULTIPLY HARDWARE Version 2

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

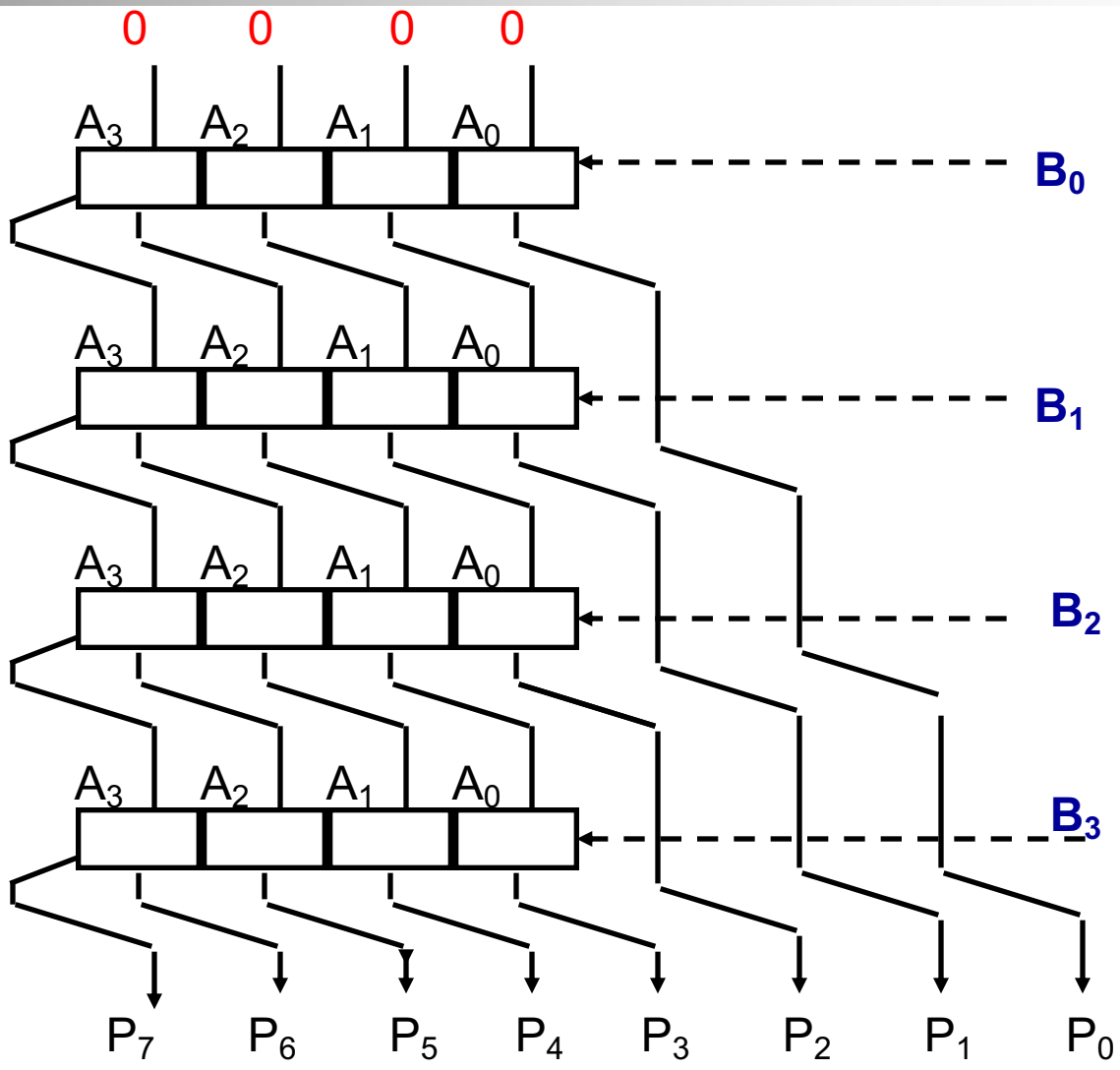


How to Think of This?

Remember original combinational multiplier:

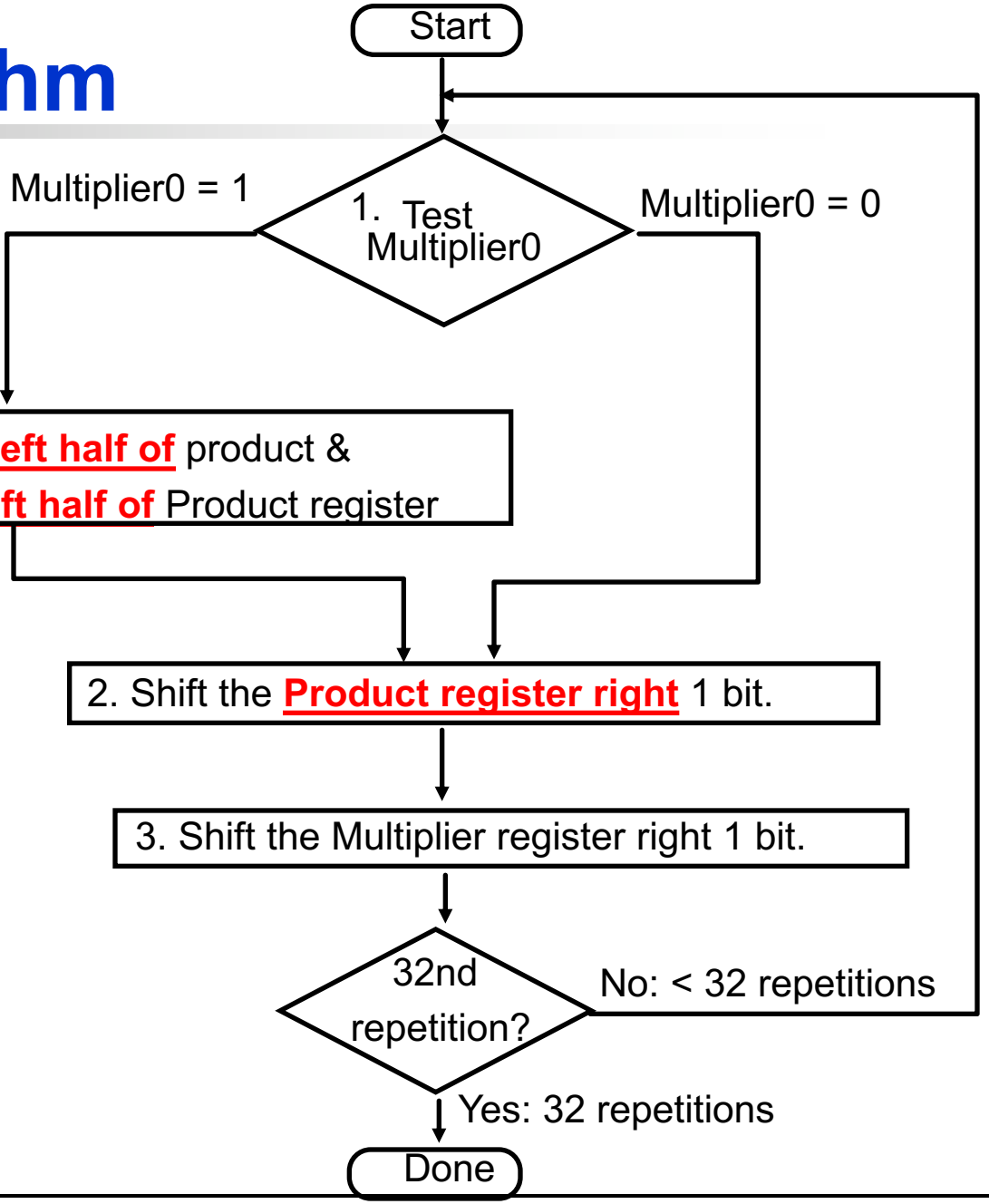


Simply warp to let product move right...



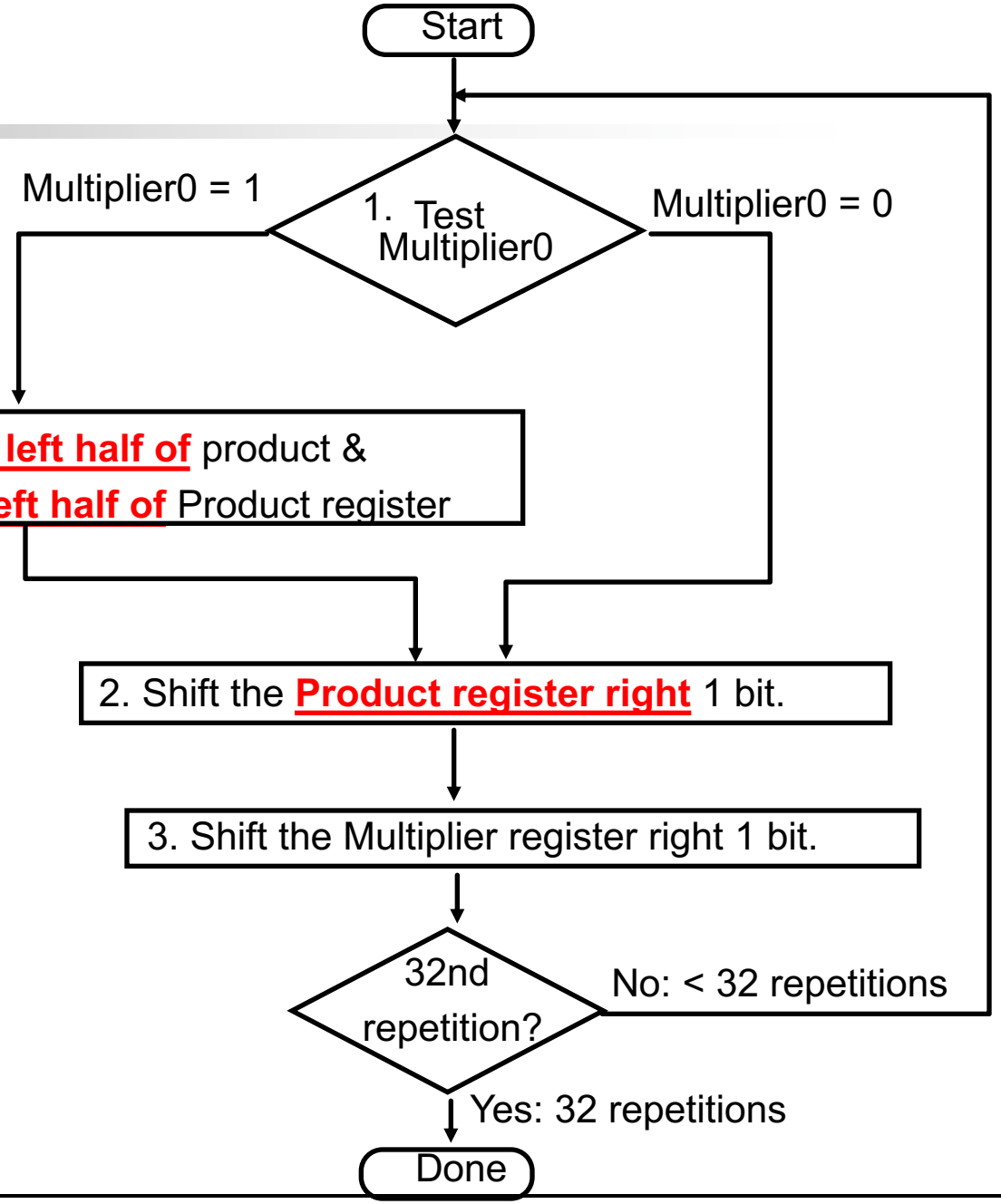
- Multiplicand stay's still and product moves right

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!



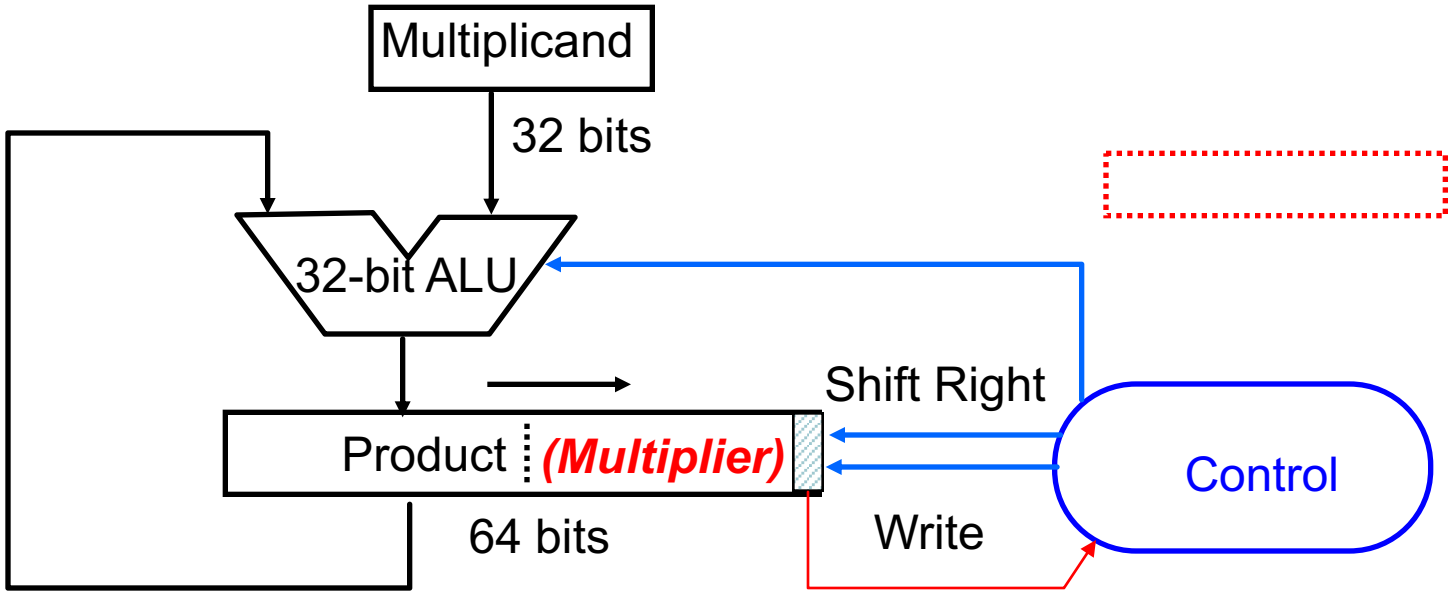
	Product	Multiplier	Multiplicand
	0000 **0000**	0011	0010
1:	0010 **0000**	0011	0010
2:	0001 0**000**	0011	0010
3:	0001 0**000**	**0**001	0010
1:	0011 0**000**	**0**001	0010
2:	0001 10**00**	**0**001	0010
3:	0001 10**00**	**00**00	0010
1:	0001 10**00**	**00**00	0010
2:	0000 110**0**	**00**00	0010
3:	0000 110**0**	**000**0	0010
1:	0000 110**0**	**000**0	0010
2:	0000 0110	**0000**	0010
3:	0000 0110	**0000**	0010
	0000 0110	**0000**	0010

Observations on Multiply Version 2

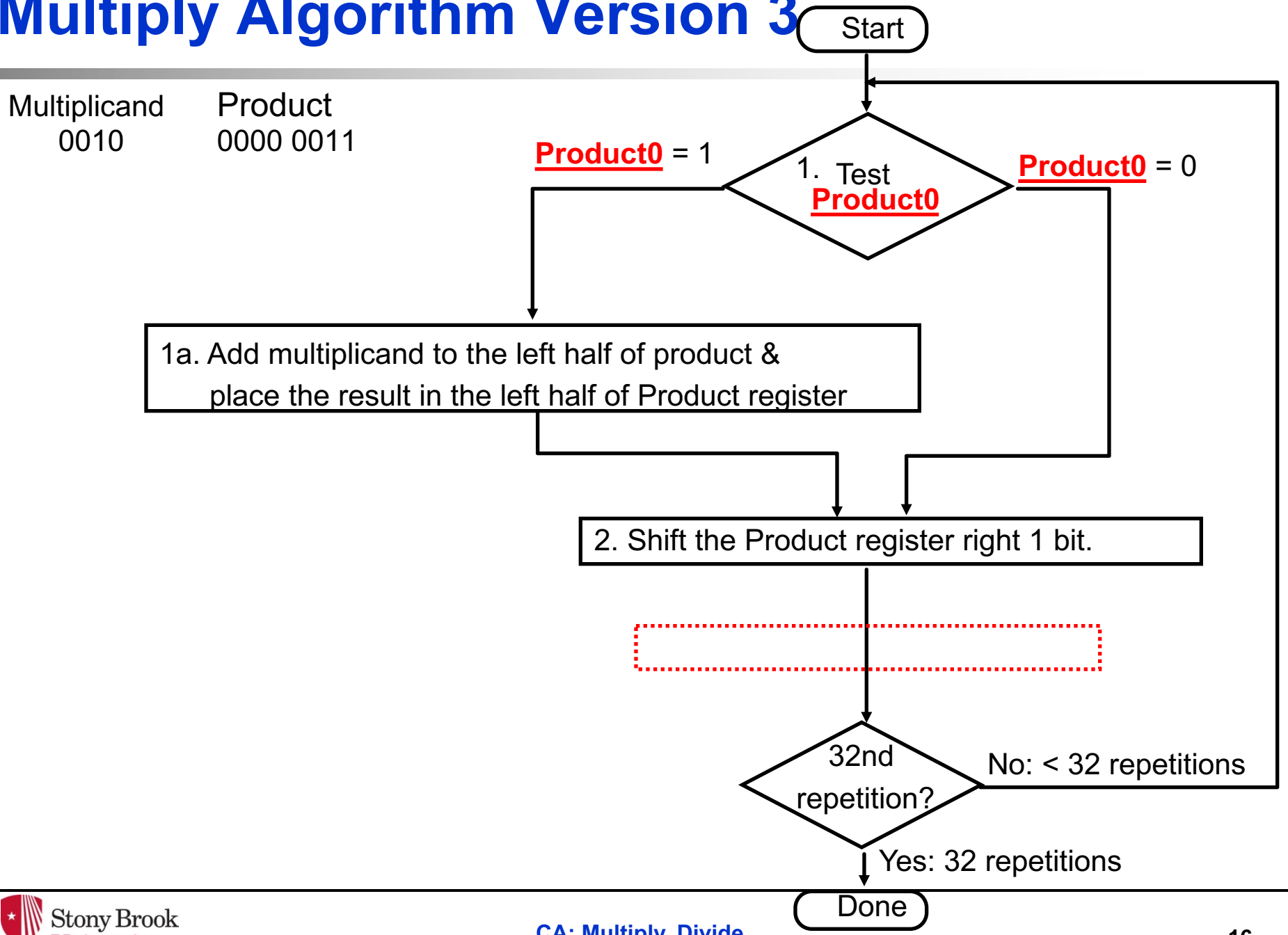
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

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



Multiply Algorithm Version 3



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
- How can you make it faster?
- 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)
  - 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 and save cycles
    - can handle multiple bits at a time

Motivation for Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
                0010
 x              0110
 +              0000      shift (0 in multiplier)
 +             0010       add (1 in multiplier)
 +            0010        add (1 in multiplier)
 +           0000         shift (0 in multiplier)
            00001100
```

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

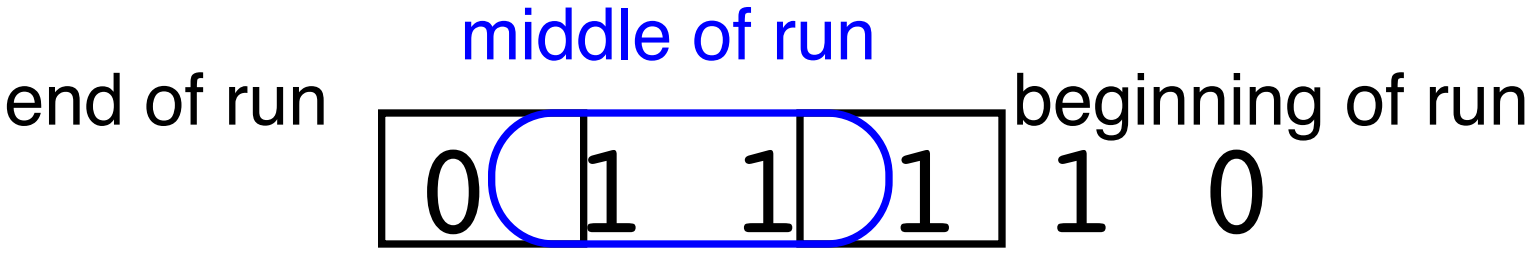
```
6       = – 2 + 8
0110    = – 00010 + 01000 = 11110 + 01000
```

- For example

-

```
                  0010
 x                0110
                  0000   shift (0 in multiplier)
 –                0010   sub (first 1 in multpl.)
                 0000    shift (mid string of 1s)
 +              0010      add (prior step had last
                00001100
```

Booth's Algorithm



Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	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**

```
  -1
+ 10000
-------
  01111
```

Booths Example (2 x 7)

Operation	Multiplicand	Product	next?
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)

Operation	Multiplicand	Product	next?
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

Radix-4 Modified Booth's ⇒ Multiple representations

Once admit new symbols (i.e. $\overline{1}$), can have multiple representations of a number:

Current Bits	Bit to the Right	Explanation	Example	Recode
0 0	0	Middle of zeros	00 00 00 **00** 00	00 (0)
0 1	0	Single one	00 00 00 **01** 00	01 (1)
1 0	0	Begins run of 1s	00 01 11 **10** 00	$\overline{1}$0 (-2)
1 1	0	Begins run of 1s	00 01 11 **11** 00	0$\overline{1}$ (-1)
0 0	1	Ends run of 1s	00 **00** 11 11 00	01 (1)
0 1	1	Ends run of 1s	00 **01** 11 11 00	10 (2)
1 0	1	Isolated 0	00 11 **10** 11 00	0$\overline{1}$ (-1)
1 1	1	Middle of run	00 11 **11** 11 00	00 (0)

Divide: Paper & Pencil

```
                    1001        Quotient
Divisor 1000 | 1001010          Dividend
             -1000
              ----
                 10
                101
               1010
              -1000
              -----
                 10             Remainder (or Modulo result)
```

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

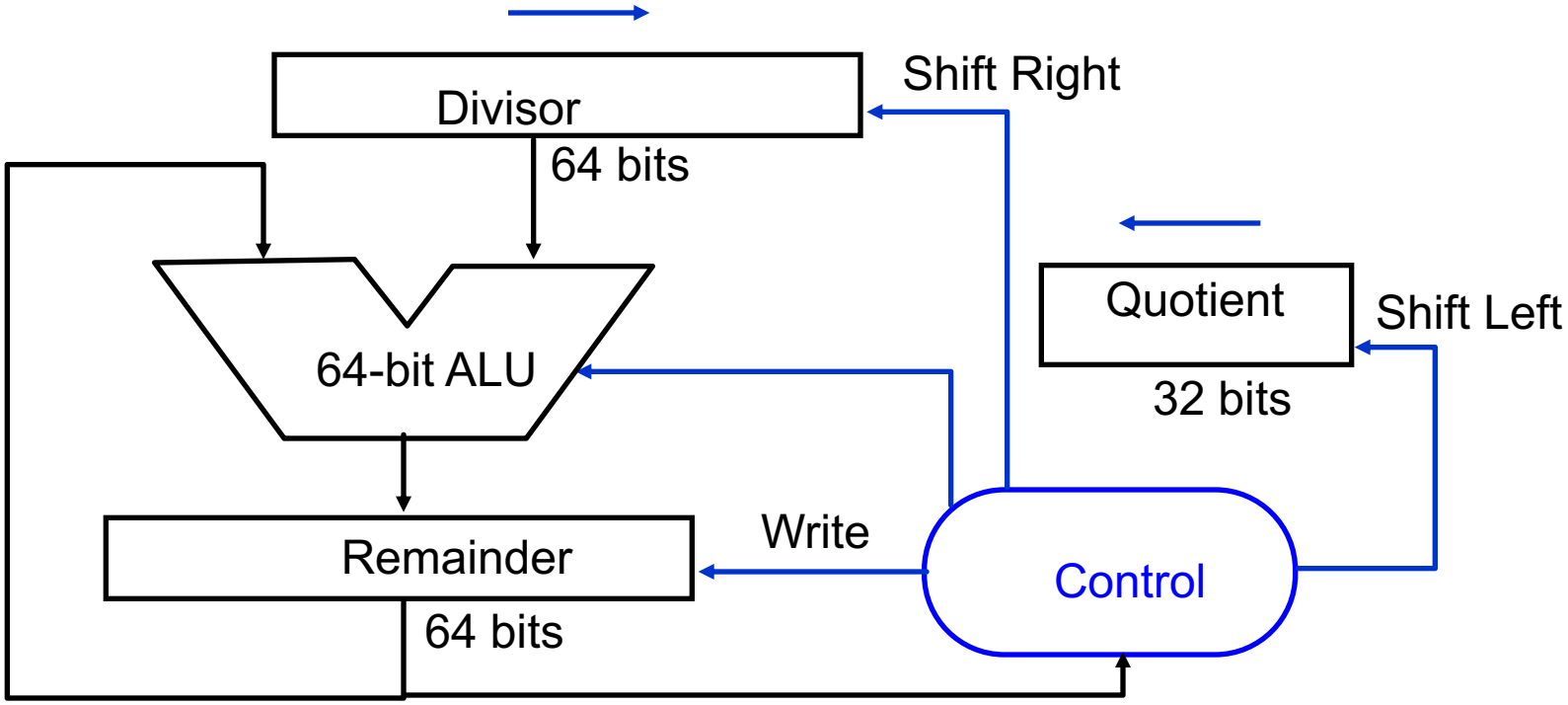
Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder

3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

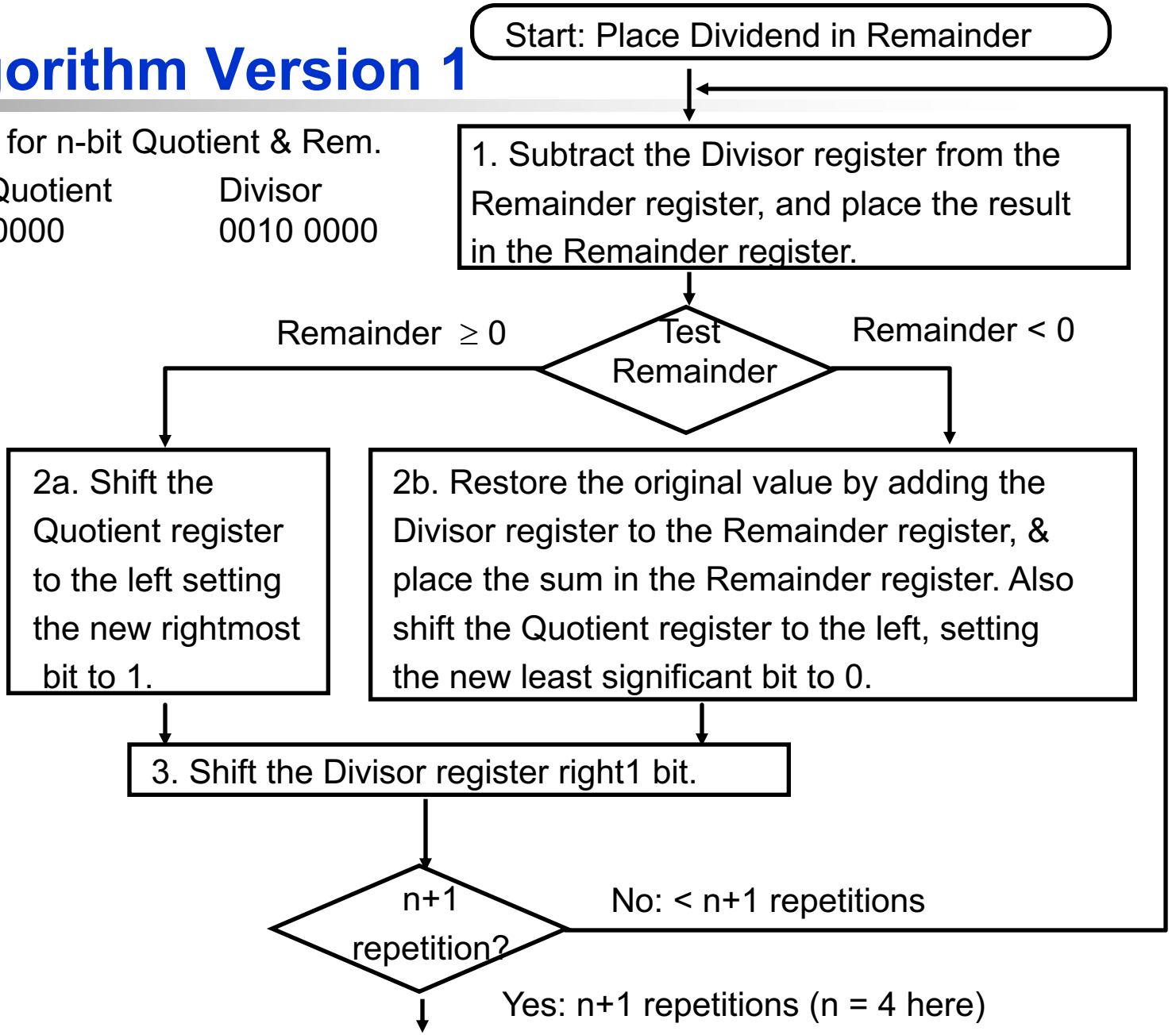
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

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

Remainder	Quotient	Divisor
0000 0111	0000	0010 0000



Divide Algorithm I example (7 / 2)

	Remainder	Quotient	Divisor
	0000 0111	00000	0010 0000
1:	1110 0111	00000	0010 0000
2:	0000 0111	00000	0010 0000
3:	0000 0111	00000	0001 0000
1:	1111 0111	00000	0001 0000
2:	0000 0111	00000	0001 0000
3:	0000 0111	00000	0000 1000
1:	1111 1111	00000	0000 1000
2:	0000 0111	00000	0000 1000
3:	0000 0111	00000	0000 0100
1:	0000 0011	00000	0000 0100
2:	0000 0011	00001	0000 0100
3:	0000 0011	00001	0000 0010
1:	0000 0001	00001	0000 0010
2:	0000 0001	00011	0000 0010
3:	0000 0001	00011	0000 0010

Answer:

Quotient = 3

Remainder = 1

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

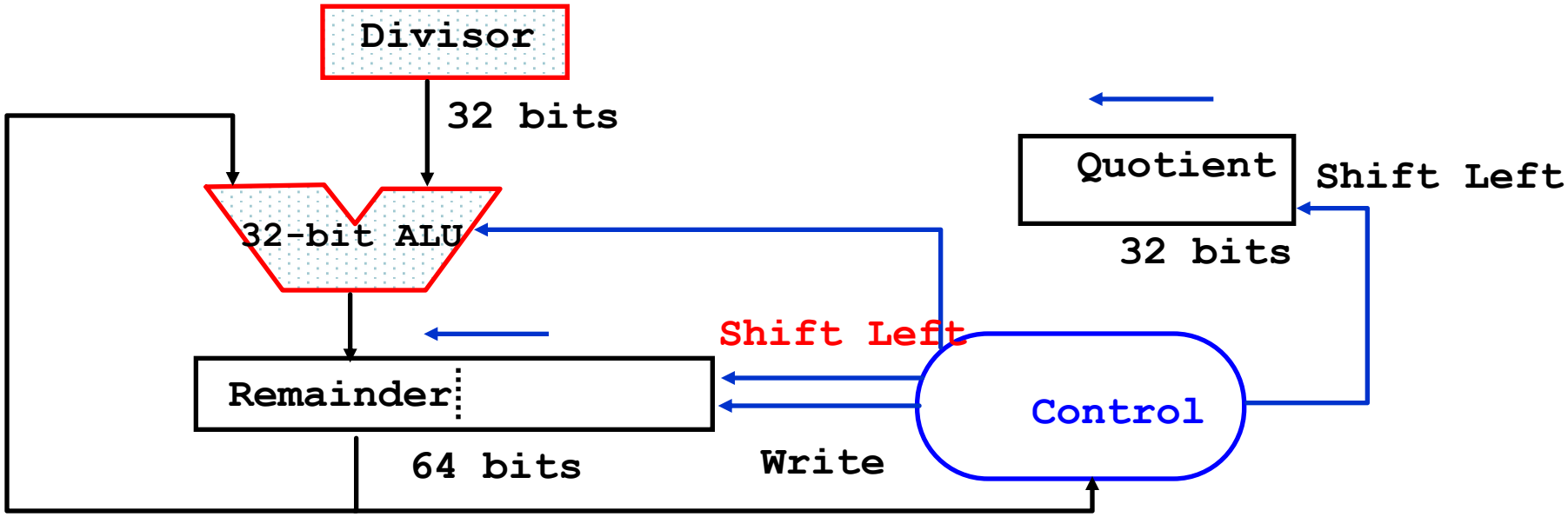
Divide: Paper & Pencil

```
                    01010      Quotient
Divisor 0001 | 00001010        Dividend
               00001
              -0001
                0000
                 0001
                -0001
                    0
                   00          Remainder (or Modulo result)
```

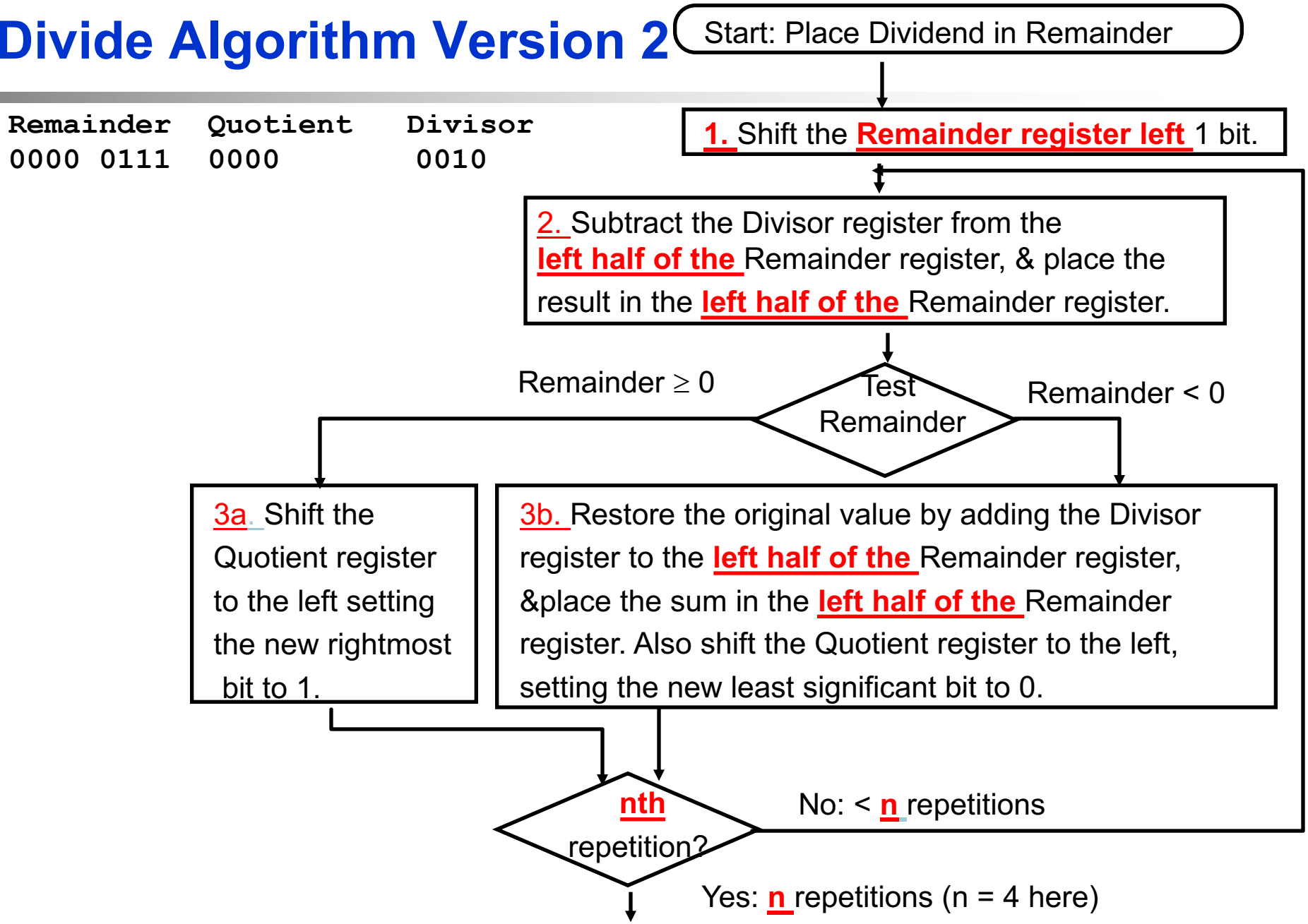
- Notice that there is no way to get a 1 in leading digit! (this would be an overflow, since quotient would have n+1 bits)

DIVIDE HARDWARE Version 2

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



Divide Algorithm Version 2

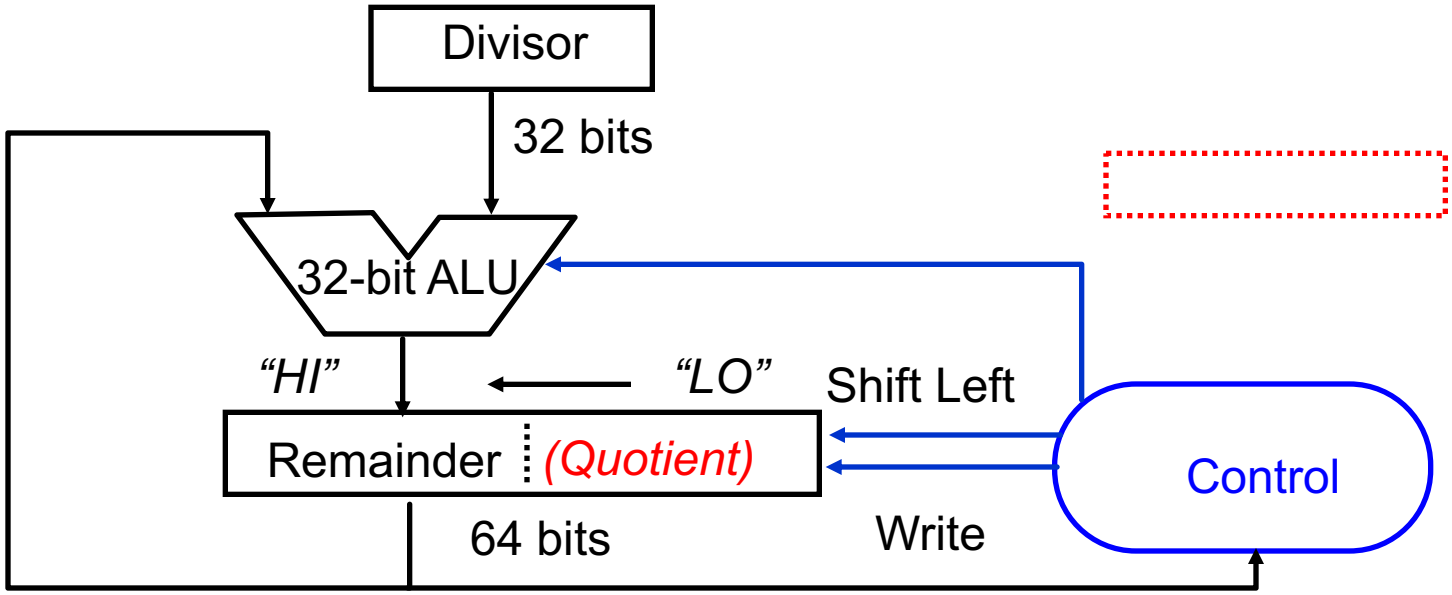


Observations on Divide Version 2

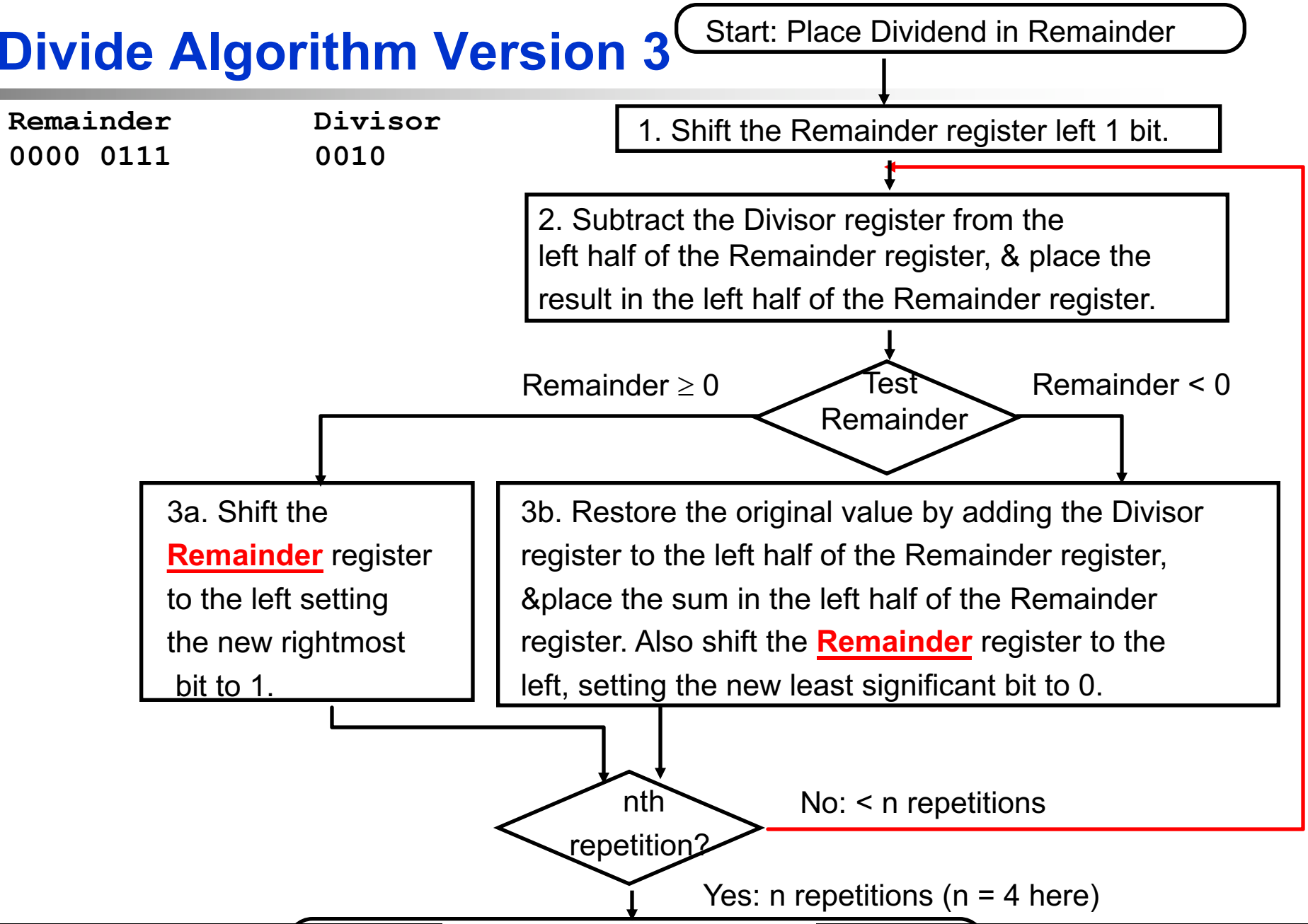
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The 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, (0-bit Quotient reg)



Divide Algorithm Version 3



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 ÷ 2 = –3, remainder = –1
  - What about? –7 ÷ 2 = –4, remainder = +1
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (“called saturation”)

Summary

- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth's algorithm to handle signed multiplies
  - There are algorithms that calculate many bits of multiply per cycle

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)