Computer Organization

Lecture 10 - Multiplication and Division

Reading: 3.3-3.4

Outline - Multiplication and Division

- **Multiplication**
  - **Review: Shift & Add Multiplication**
  - **Review: Booth's Algorithm**
  - **Combinational Multiplication**
  - **MIPS Multiplication Instructions**
- **Division**
- **Summary**

Multiplication

- Basic algorithm analogous to decimal multiplication
  - Break multiplier into digits
  - Multiply one digit at a time;
    shift multiplicand to form partial products
  - Create product as sum of partial products

피승수

승수

```
Multiplicand        0110     (6)
Multiplier        X 0011     (3)
                  ----------------
                    0110          
                   0110           
                  0000            
                 0000             
                 ----------------
Product         00010010    (18)
```

Partial Products

곱의 결과

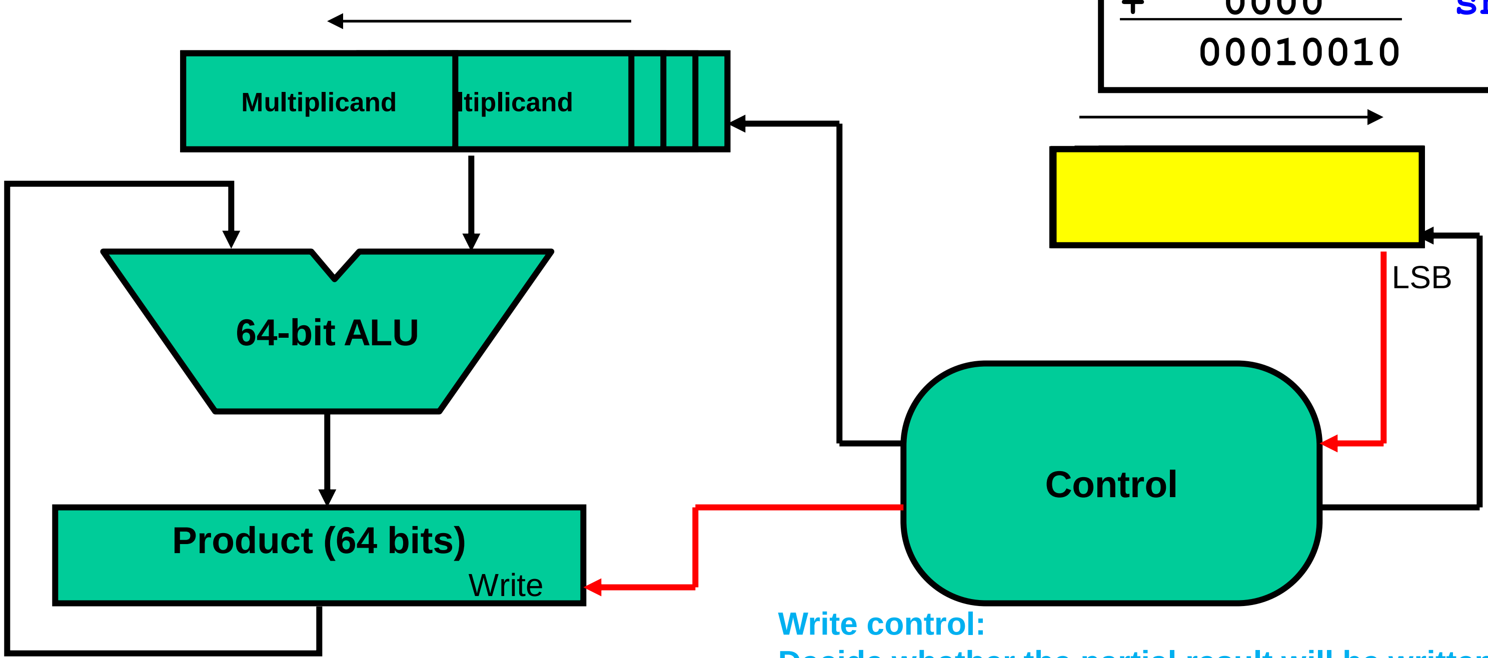
- n bit multiplicand X m bit multiplier = (n+m) bit product

Animation - 1st Cut Multiplier

Multiplicand의 자릿수를 하나씩 올리면서 (shift left) 연산한다.

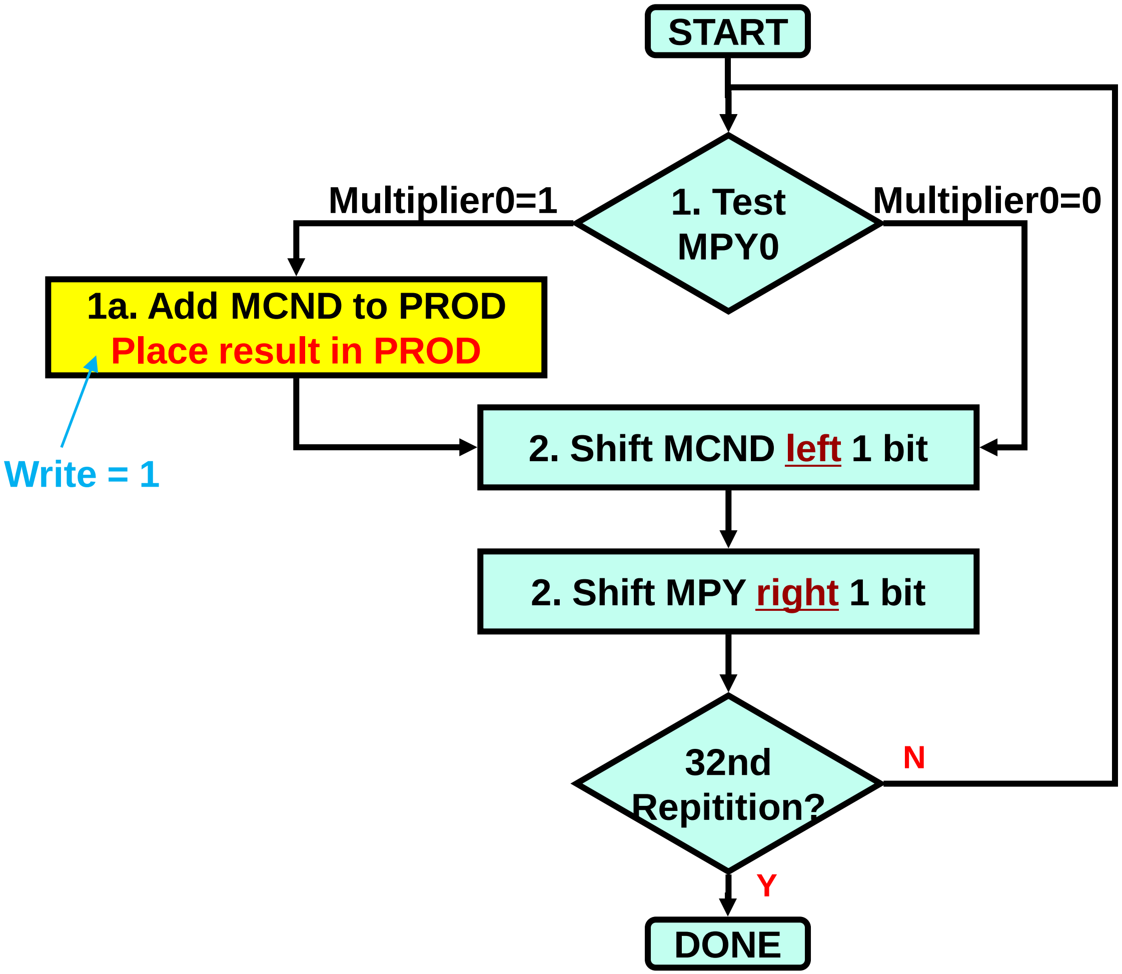
- Multiplicand shifts left
- Multiplier shifts right
- Sample LSB of multiplier to decide whether to add

```
          0110
x         0011
+         0110   add + shift
+        0110    add + shift
+       0000     shift
+      0000      shift
      00010010
```



Write control:
Decide whether the partial result will be written or not by checking the LSB of multiplier

Algorithm - 1st Cut Multiplier



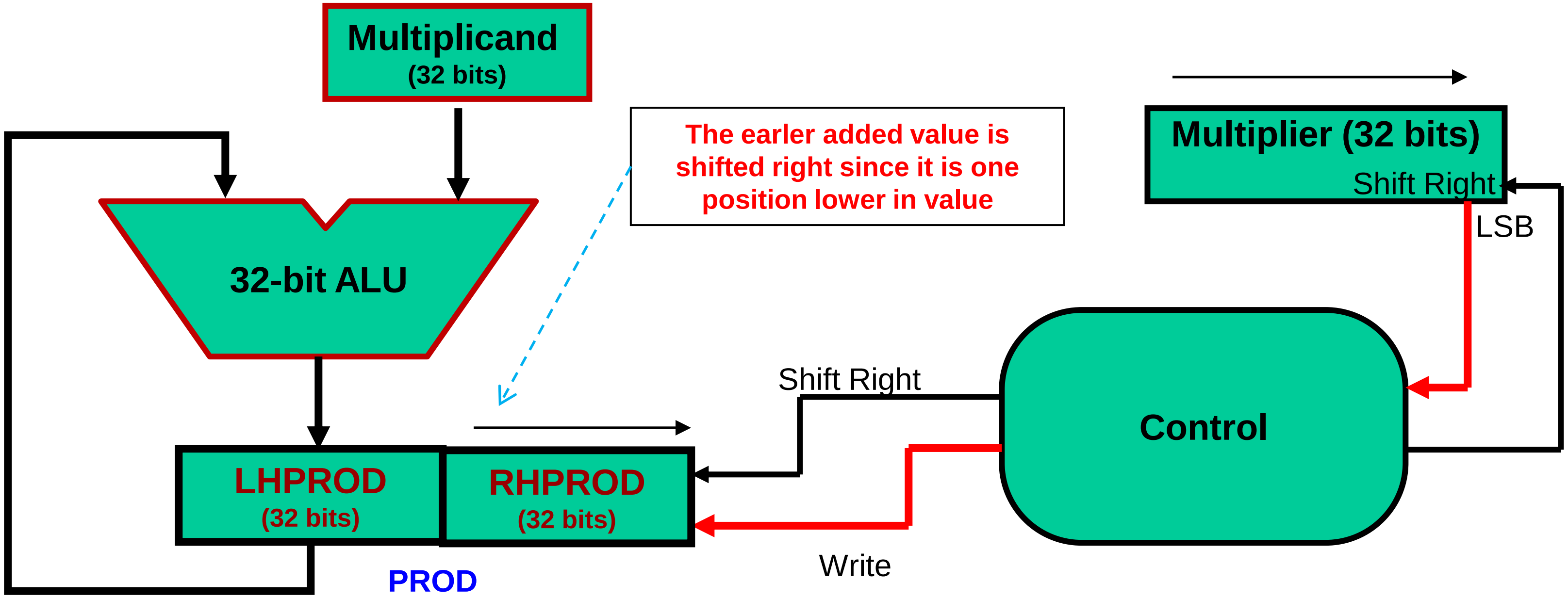

```
        0110
x       0011
+       0110   add + shift
+      0110    add + shift
+     0000     shift
+    0000      shift
    00010010
```

Sequential Multiplier - 2nd Version

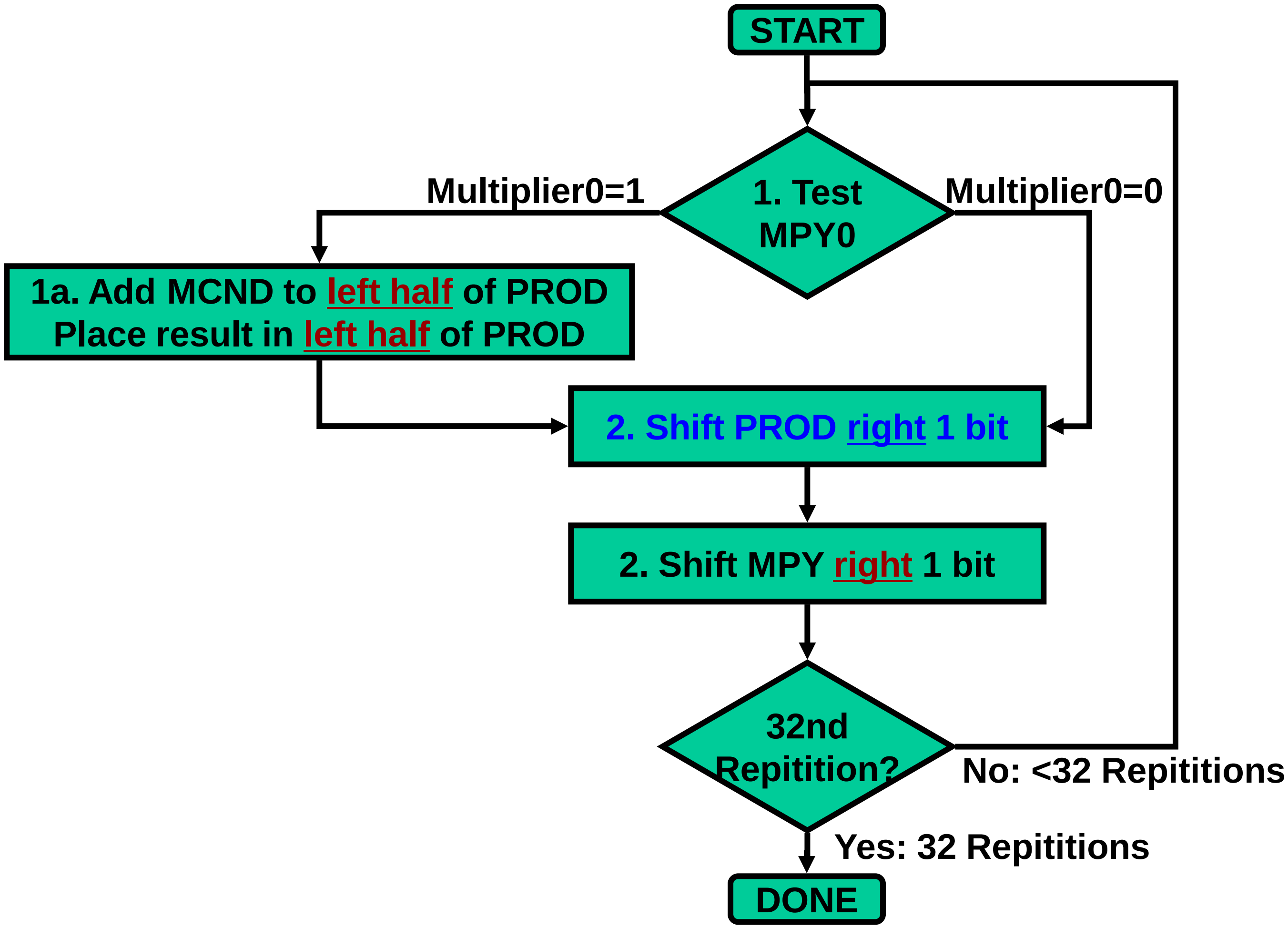
연산을 하고 나서 product의 자릿수를 하나씩 내린다 (shift right) .

- Observation: we're only adding 32 bits at a time
- Clever idea: Why not...
  - Hold the multiplicand still and…
  - Shift the product right!

```
   0110     (6)
x  0011     (3)
   00000000 current product
   00110000 add & shift rht
   01001000 add & shift rht
   00100100 no add & shift rht
   00010010 no add & shift rht
```



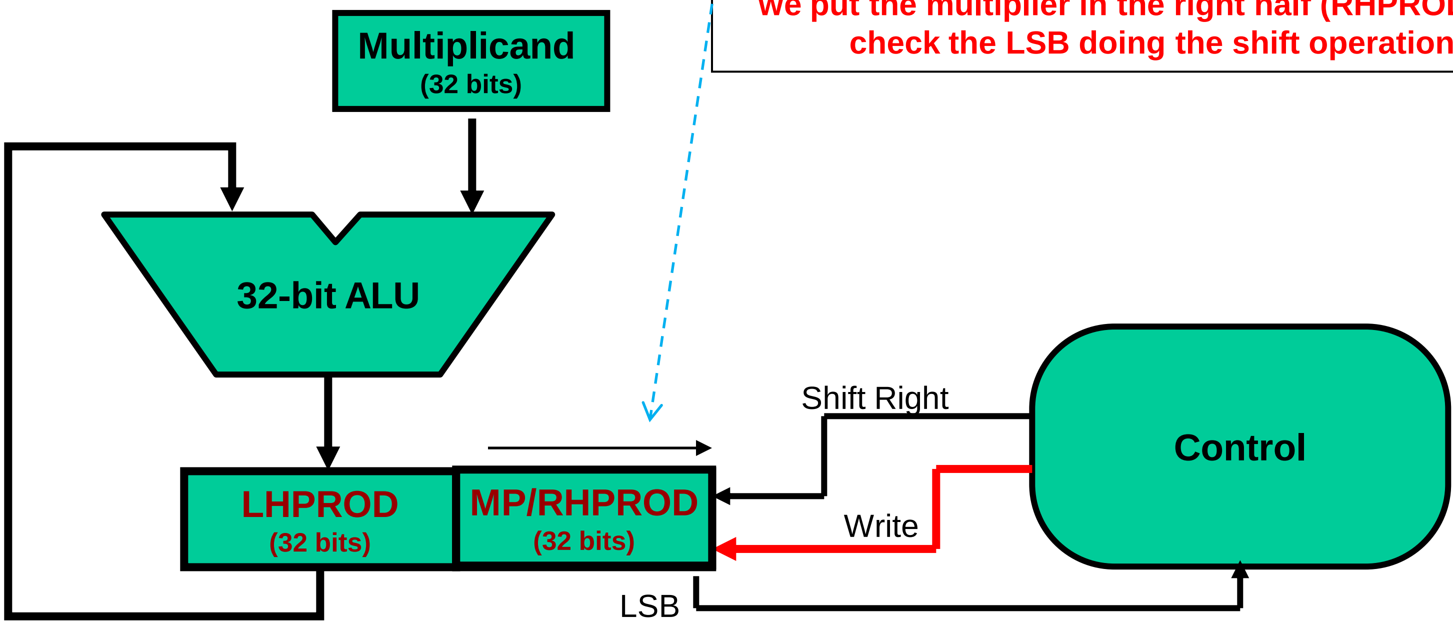
Algorithm - 2nd Version Multiplier



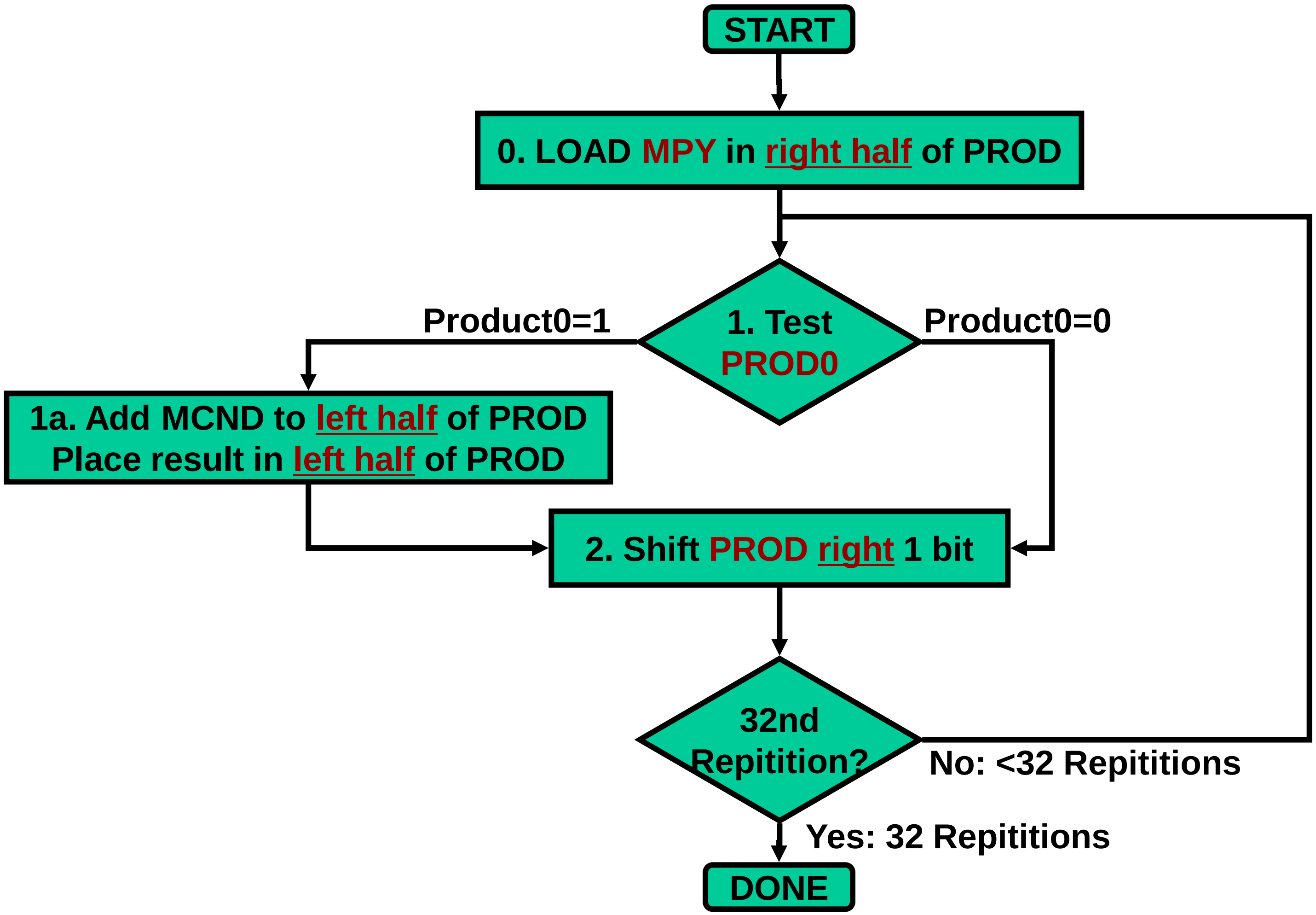
Sequential Multiplier - 3nd Version

- **Observation: we can store the multiplier and product in the same register!**
  - **As multiplier shifts out….**
  - **Product shifts in**

Since the right half of the product (PROD) disappears, we put the multiplier in the right half (RHPROD) and check the LSB doing the shift operation



Algorithm - 3rd Version Multiplier



Outline - Multiplication and Division

- **Multiplication**
  - **Review: Shift & Add Multiplication**
  - **Review: Booth's Algorithm**
  - **Combinational Multiplication**
  - **MIPS Multiplication Instructions**
- **Division**
- **Summary**

Signed Multiplication with Booth's Algorithm

- **Originally proposed to reduce addition steps**
- **Bonus: works for two's complement numbers**
- **Uses shifting, addition, and subtraction**

Booth's Algorithm

- Observation: if we can both add and subtract, there are multiple ways to create a product
- Example: multiply 2_{ten} by 6_{ten} (0010_{two} X 0110_{two})
  - 2 X 6 = 2 X (2 + 4) = (2 X 2) + (2 X 4) OR
  - 2 X 6 = 2 X (-2 + 8) = (2 X -2) + (2 X 8)

Regular Algorithm

```
        0010
x       0110
+       0000   shift
+      0010    add + shift
+     0010     add + shift
+    0000      shift
    00001100
```

Booth's Algorithm

```
        0010
x       0110      (= 1000 - 0010)
        0000   shift
-      0010    sub + shift
      0000     shift
+    0010      add + shift
    00001100
```

Booth's Algorithm Continued

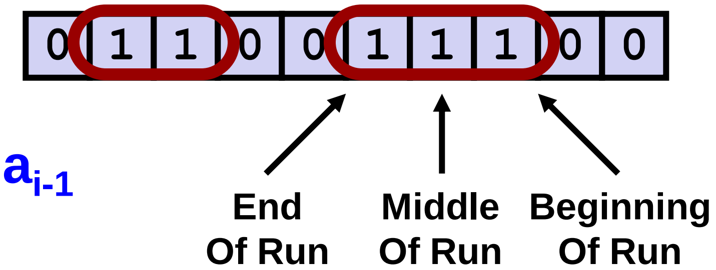
- **Question:**
  - **How do we know when to subtract?**
  - **When do we know when to add?**
- **Answer: look for "runs of 1s" in multiplier**
  - **Example:** `00`<u>`111`</u>`00`<u>`11`</u>
  - **Working from Right to Left, any "run of 1's" is equal to:**

    - value of first digit that's one

    +value of first digit that's zero
  - **Example :** `00`<u>`111`</u>`00`<u>`11`</u>
    - **First run: -1 + 4 = 3**
    - **Second run: -16 + 128 = 112**
    - **Total: (-1 + 4) + (-16 + 128) = 3 + 112 = 115**

Implementing Booth’s Algorithm

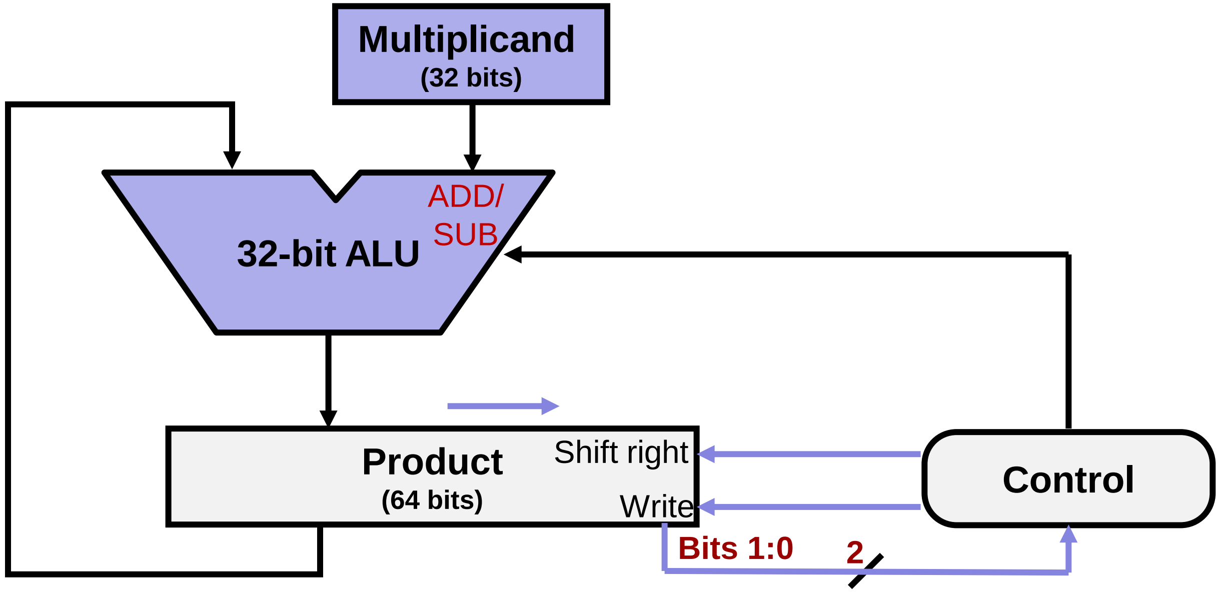
- **Scan multiplier bits from right to left**
- **Recognize the beginning and in of a run looking at only 2 bits at a time**
  - **“Current” bit: a_i**
  - **Bit to right of “current” bit: a_{i-1}**



Bit a_i	Bit a_{i-1}	Explanation
1	0	Begin Run of 1’s
1	1	Middle of Run of 1’s
0	1	End of Run
0	0	Middle of Run of 0’s

Implementing Booth's Algorithm

- Key idea: test 2 bits of multiplier at once
  - 10 - subtract (beginning of run of 1's)
  - 01 - add (end of run of 1's)
  - 00, 11 - do nothing (middle of run of 0's or 1's)



Booth's Algorithm Example

Multiply 4 X -9

```
   00100
x  10111
```

Remember

```
 4 = 000100
-4 = 111100
```

-9

```
  000000101110
 +111100              (sub 4 / add -4)
  111100101110
  111110010111        (shift after add)
  111111001011        (shift w/ no add)
  111111100101        (shift w/ no add)
 +000100              (add +4)
  000011100101
  000001110010        (shift after add)
 +111100              (sub 4 / add -4)
  111101110010
  111110111001        (shift after add)
```

Drop leftmost & rightmost bit

```
1111011100 = -(000100011 + 1)
           = -(000100100)
           = -36 = 4 X -9!
```

Outline - Multiplication and Division

- **Multiplication**
  - **Review: Shift & Add Multiplication**
  - **Review: Booth's Algorithm**
  - **Combinational Multiplication**
  - **MIPS Multiplication Instructions**
- **Division**
- **Summary**

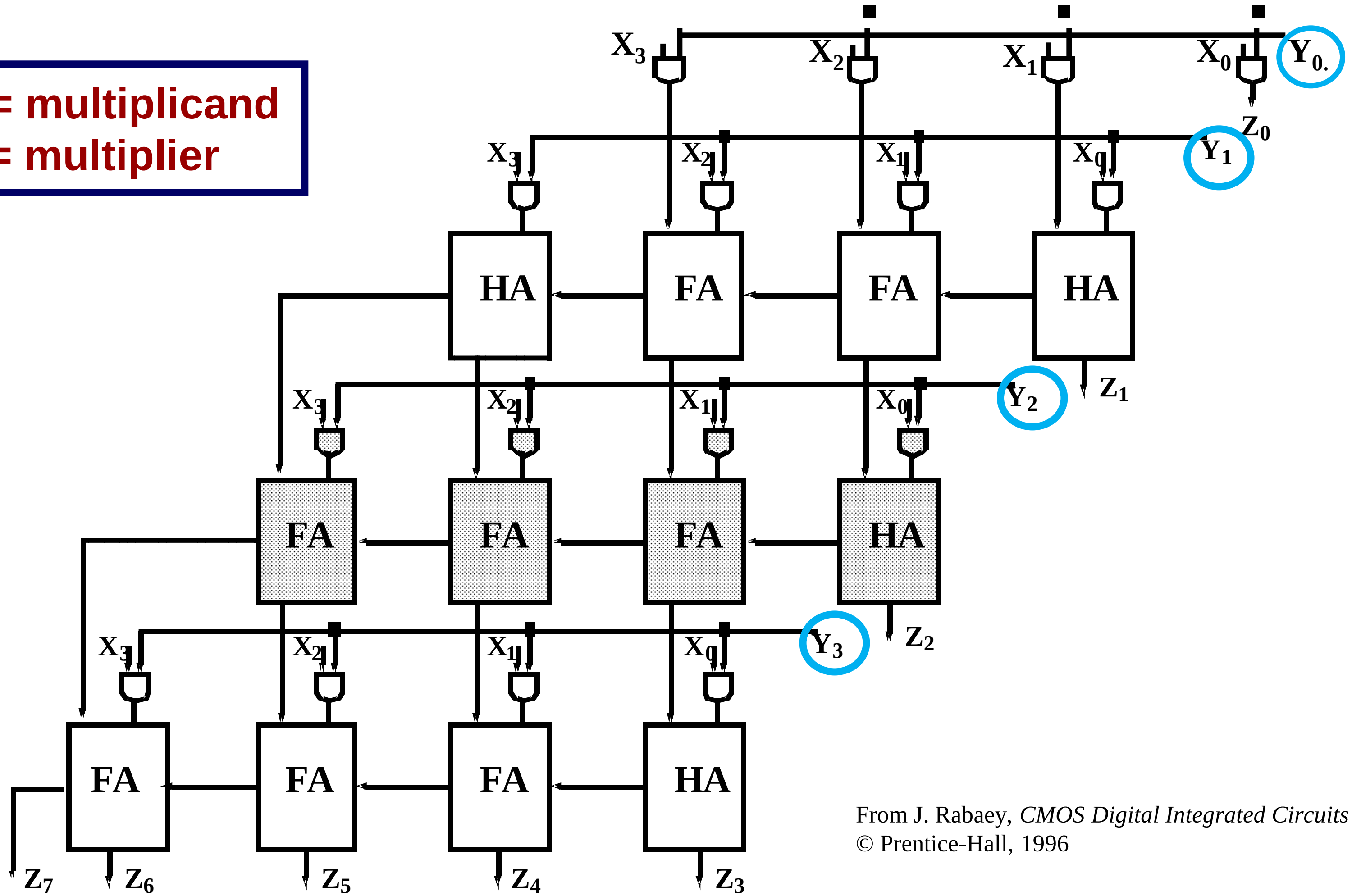
Combinational Multipliers

- Goal: make multiplication faster
- General approach
  - Use AND gates to generate partial products
  - Sum partial products with adders

Array Multiplier

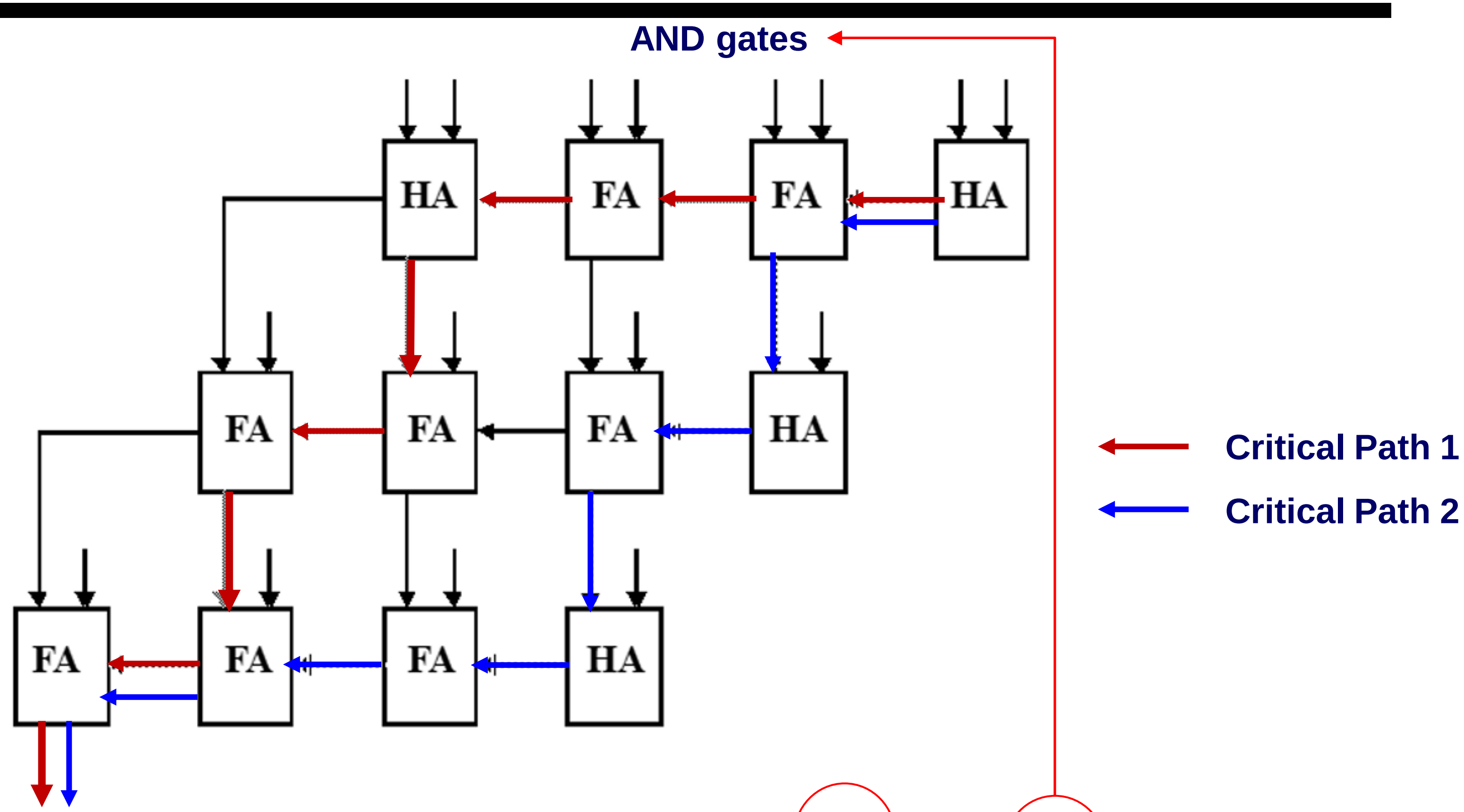
$$\begin{array}{r} X_3\ X_2\ X_1\ X_0 \\ \times \quad Y_3\ Y_2\ Y_1\ Y_0 \\ \hline Z_7\ Z_6\ Z_5\ Z_4\ Z_3\ Z_2\ Z_1\ Z_0 \end{array}$$

X = multiplicand
Y = multiplier



From J. Rabaey, *CMOS Digital Integrated Circuits*
© Prentice-Hall, 1996

Array Multiplier - Critical Paths



$$T_{mult} \approx [(M\text{-}1)+(N\text{-}2)]t_{carry} + (N\text{-}1)t_{sum} + t_{and}$$

From J. Rabaey, *CMOS Digital Integrated Circuits*

Outline - Multiplication and Division

- Multiplication
  - Review: Shift & Add Multiplication
  - Review: Booth’s Algorithm
  - Combinational Multiplication
  - MIPS Multiplication Instructions
- Division
- Summary

Multiply Instructions in MIPS

- MIPS adds new registers for product result:
  - `Hi` - upper 32 bits of product
  - `Lo` - lower 32 bits of product
- MIPS multiply instructions
  - `mult  $s0, $s1`
  - `multu $s0, $s1`
- Accessing `Hi`, `Lo` registers
  - `mfhi $s1`
  - `mflo $s1`

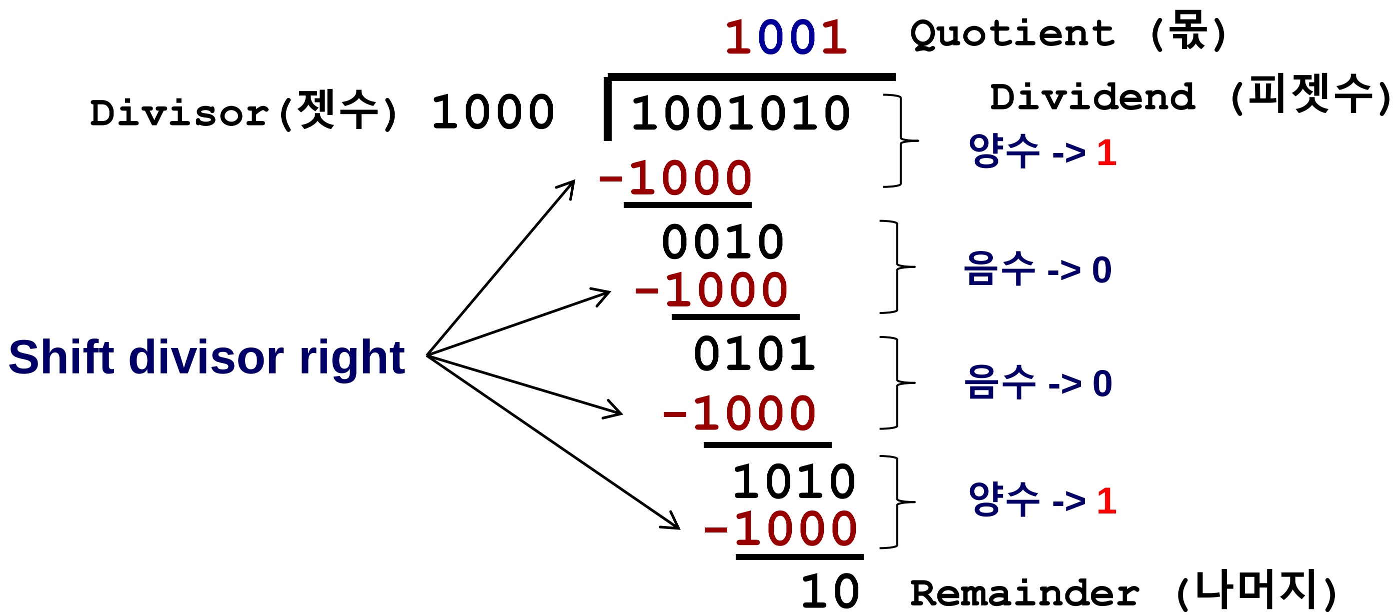
Outline - Multiplication and Division

- Multiplication
  - Review: Shift & Add Multiplication
  - Review: Booth's Algorithm
  - Combinational Multiplication
  - MIPS Multiplication Instructions
- Division
  - Division Algorithms
  - MIPS Division Instructions
- Summary

Division Overview

1st version H/W => next page

- Grammar school algorithm: long division
  - Subtract shifted divisor from dividend when it “fits”
  - Quotient bit: 1 or 0
- Question: how can hardware tell “when it fits?”



`Dividend = Quotient X Divisor + Remainder`

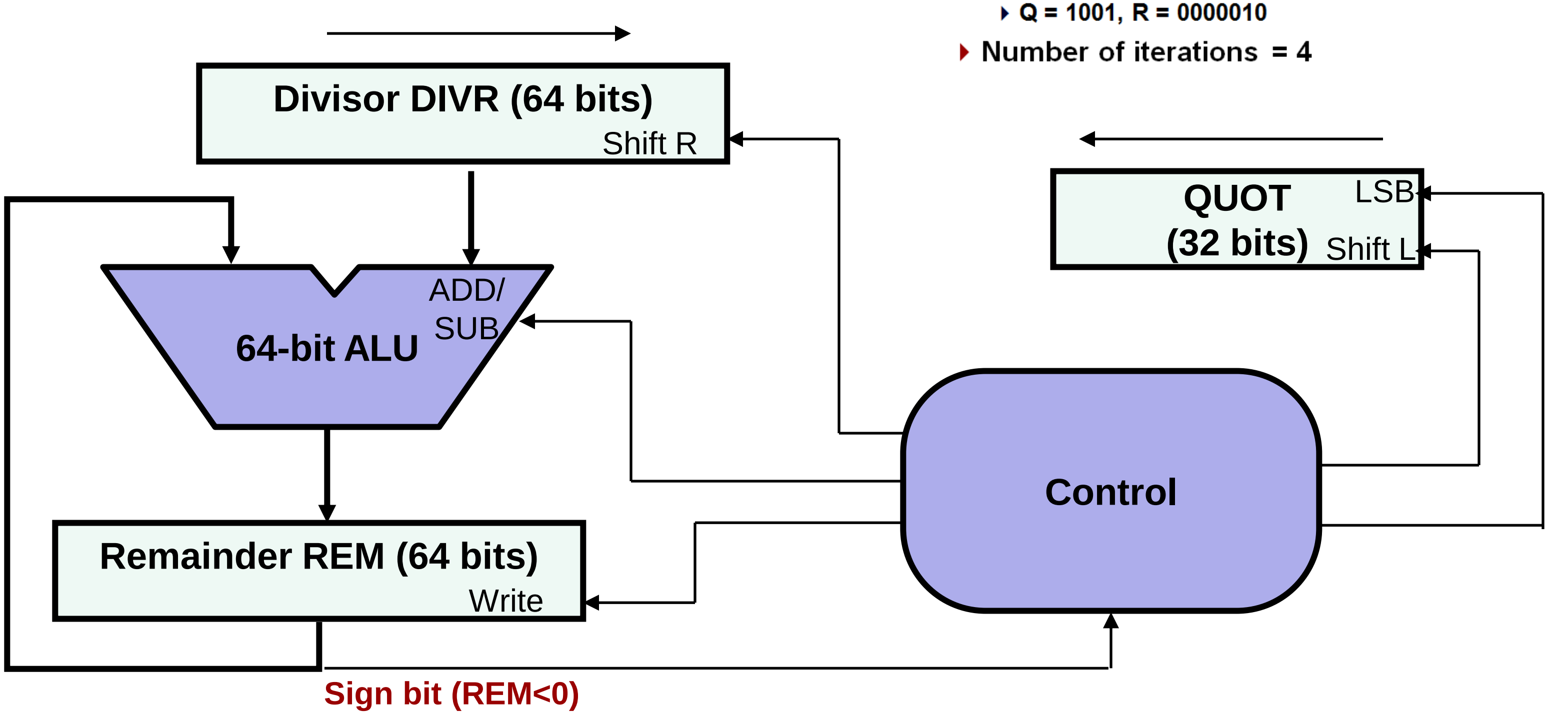
Compuation Process

- **1001010/1000 = ?**
- **1001010 – 1000000 = 0001010 => Q = 0001**
- **0001010 – 0100000 = Neg (recover) => Q = 0010**
- **0001010 – 0010000 = Neg (recover) => Q = 0100**
- **0001010 – 0001000 = 0000010 => Q = 1001**
  - **Q = 1001, R = 0000010**
- **Number of iterations = 4**

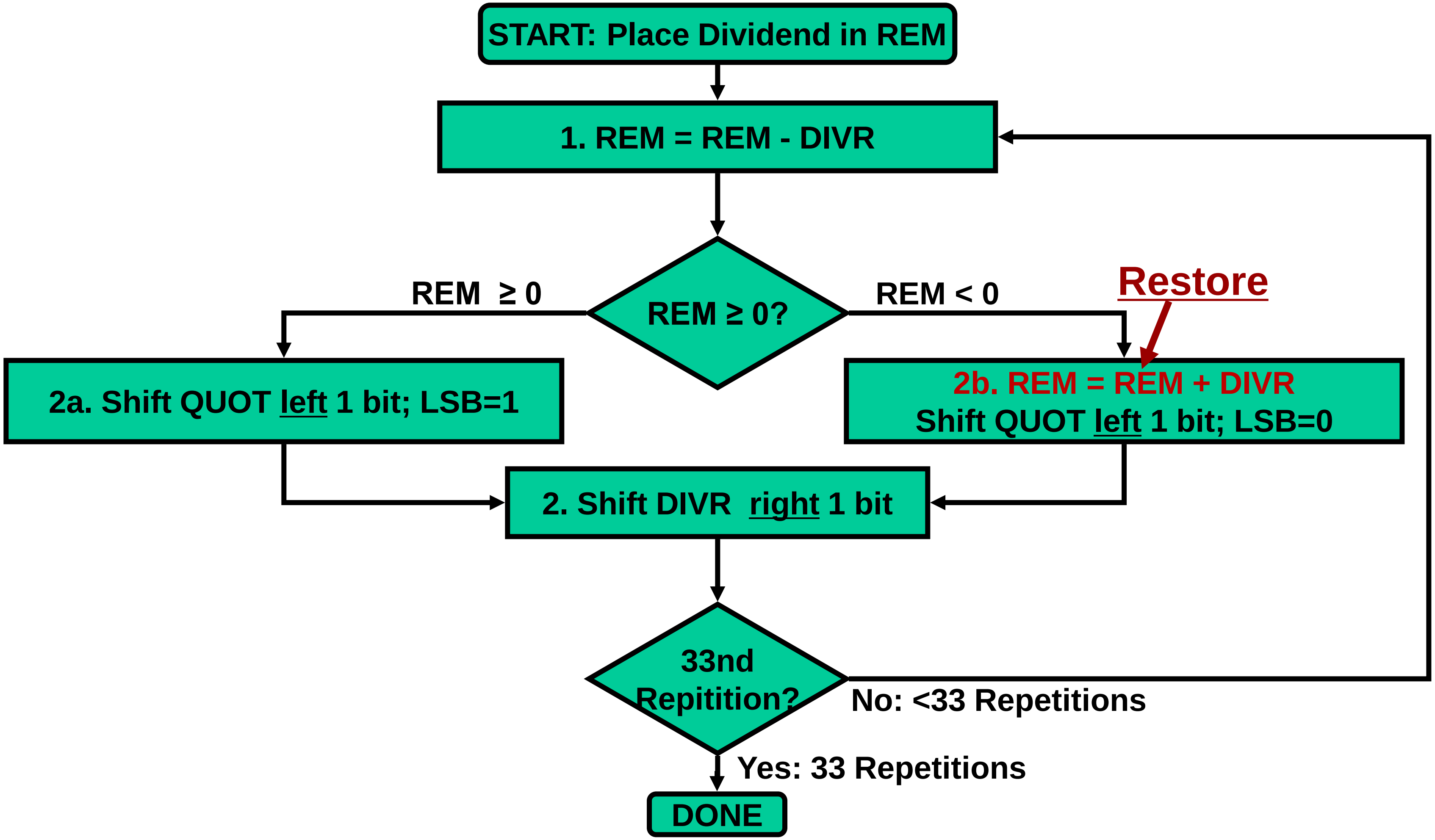
Division Hardware - 1st Version

- 1001010/1000 = ?
- 1001010 – 1000000 = 0001010 => Q = 0001
- 0001010 – 0100000 = 음수 (recover) => Q = 0010
- 0001010 – 0010000 = 음수 (recover) => Q = 0100
- 0001010 – 0001000 = 0000010 => Q = 1001
  - Q = 1001, R = 0000010
- Number of iterations = 4

- **Shift register moves divisor (DIVR) to right**
- **ALU subtracts DIVR, then <u>restores</u> (adds back) if REM < 0 (i.e. divisor was "too big")**



Division Algorithm - First Version



Divide 1st Version - Observations

- **We only subtract 32 bits in each iteration**
  - **Idea: Instead of shifting divisor to right,**
    shift remainder to left
- **First step cannot produce a 1 in quotient bit**
  - **Switch order to shift first, then subtract**
  - **Save 1 iteration**

Computation Process

- 10010100/00001000 = ?

substract

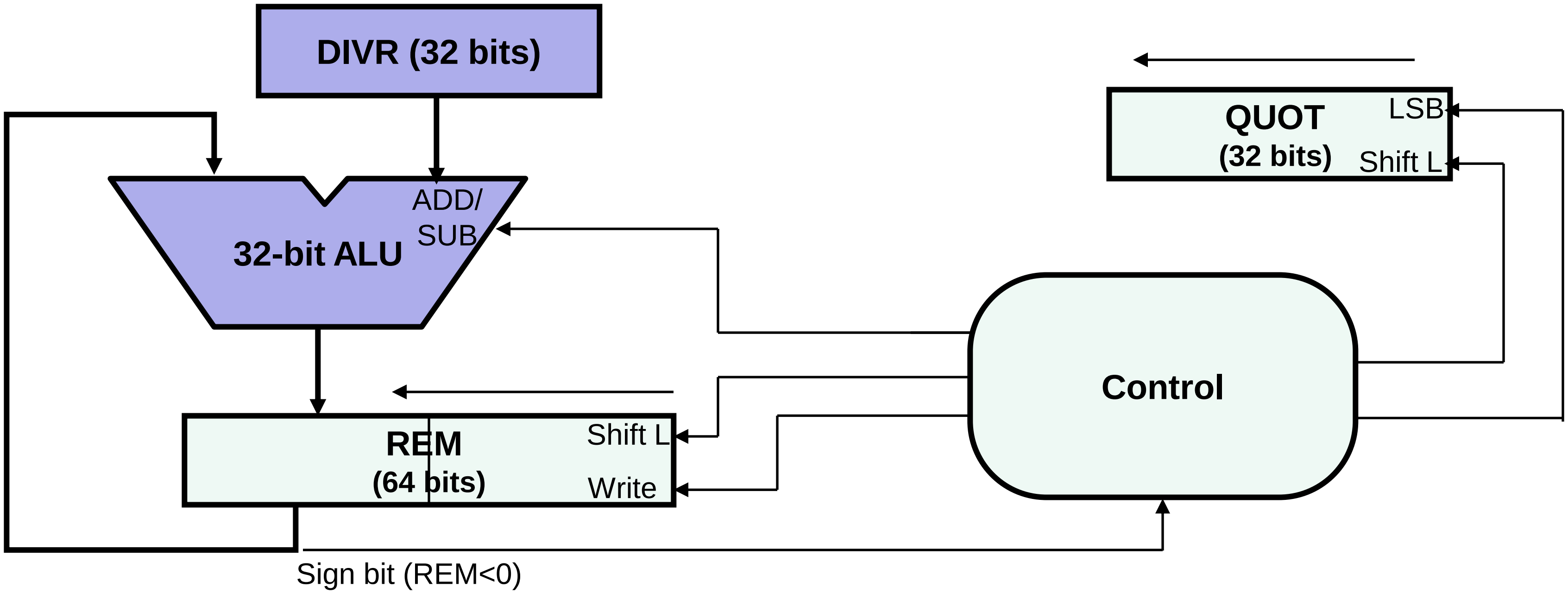
- ~~0000000010010100 00001000 = Neg (recover) => Q = 00000000~~
- 0000000100101000 – 00001000 = Neg (recover) => Q = 00000000
- 0000001001010000 – 00001000 = Neg (recover) => Q = 00000000
- 0000010010100000 – 00001000 = Neg (recover) => Q = 00000000
- 0000100101000000 – 00001000 = Pos => Q = 00000001
- 0000001010000000 – 00001000 = Neg (recover) => Q = 00000010
- 0000010100000000 – 00001000 = Neg (recover) => Q = 00000100
- 0000101000000000 – 00001000 = Pos => Q = 00001001
- 0000010000000000

remainder

First step: Always negative, so switch the order to shift first, then substract => To save one iteration

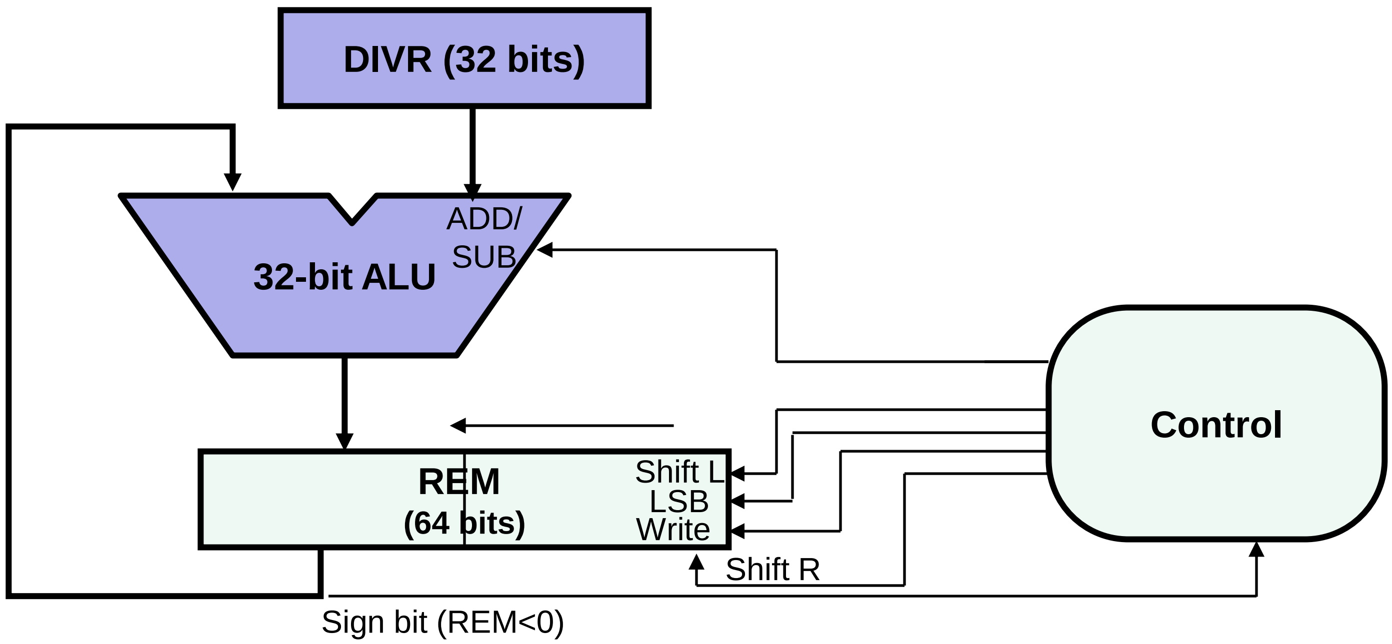
Divide Hardware - 2nd Version

- Divisor Holds Still
- Dividend/Remainder Shifts Left
- End Result: Remainder in upper half of register

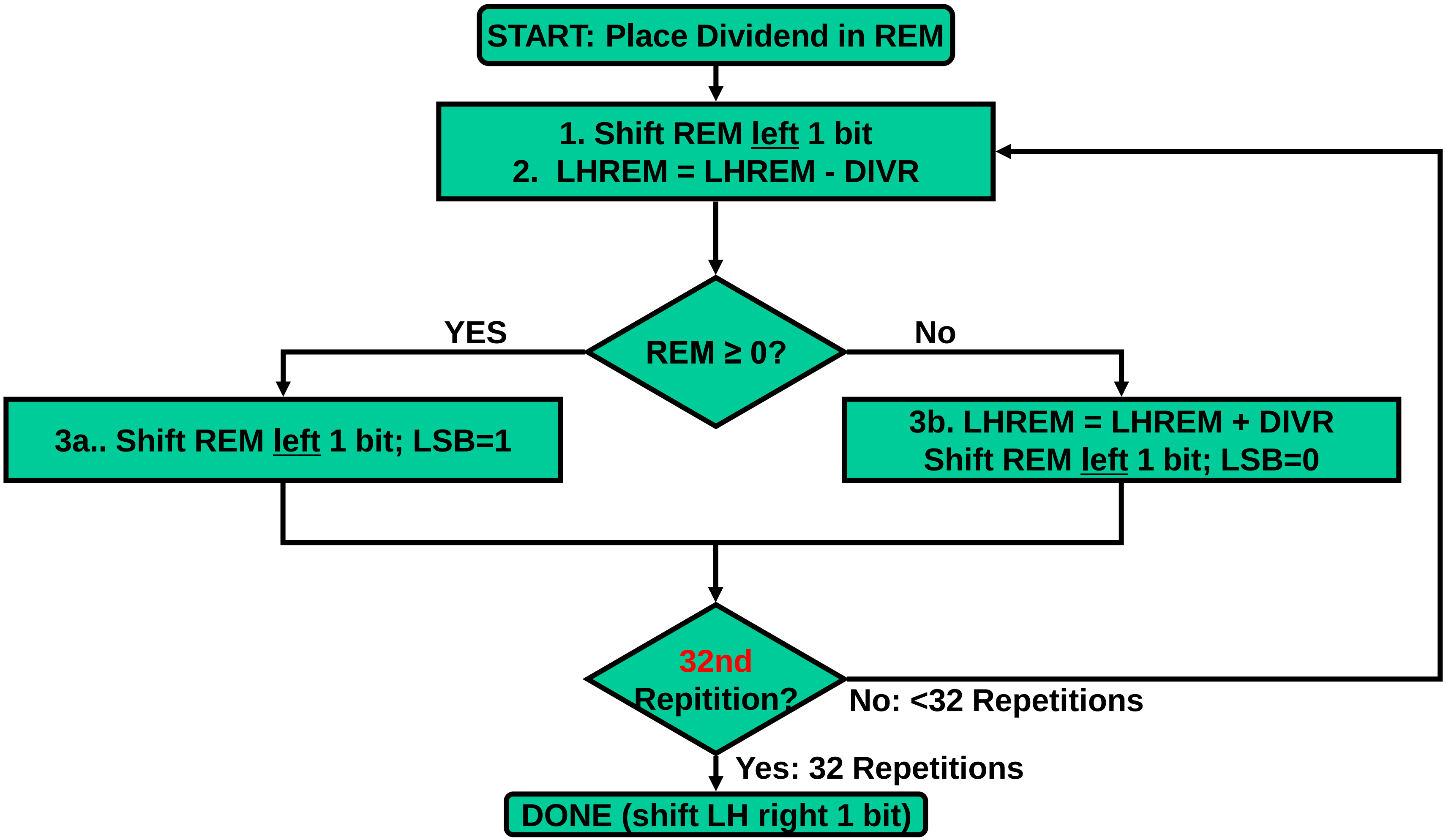


Divide Hardware - 3rd Version

- **Combine quotient with remainder register**



Divide Algorithm - 3rd Version



Dividing Signed Numbers

- **Check sign of divisor, dividend**
- **Negate quotient if signs of operands are opposite**
- **Make <u>remainder sign</u> match dividend (if nonzero)**

Outline - Multiplication and Division

- **Multiplication**
  - **Review: Shift & Add Multiplication**
  - **Review: Booth's Algorithm**
  - **Combinational Multiplication**
  - **MIPS Multiplication Instructions**
- **Division**
  - **Division Algorithms**
  - **MIPS Division Instructions**
- **Summary**

Divide Instructions in MIPS

- **Divide Instructions**
  - `div  $s2, $s3 # Lo = $s2 / $s3; Hi = $s2 % $s3`
  - `divu $s2, $s3 # Lo = $s2 / $s3; Hi = $s2 % $s3`
- **Results in Lo, Hi registers**
  - `Hi`: remainder
  - `Lo`: quotient
- **Divide pseudoinstructions**
  - `div  $s3, $s2, $s1         # $s3 = $s2 / $s1`
  - `divu $s3, $s2, $s1`

```
div  $s3, $s2, $s1

div  $s2, $s1
mflo $s3
```

- **<u>Software</u> must check for overflow, divide-by-zero**

Outline - Multiplication and Division

- Multiplication
  - Review: Shift & Add Multiplication
  - Review: Booth's Algorithm
  - Combinational Multiplication
  - MIPS Multiplication Instructions
- Division
  - Division Algorithms
  - MIPS Division Instructions
- Summary

Summary - Multiplication and Division

- Multiplication
  - Sequential multipliers - efficient but slow
  - Combinational multipliers - fast but expensive
- Division is more complex and problematic
  - What about divide by zero?
  - Restore step needed to undo unwanted subtractions
  - ~~Nonrestoring division: combine restore w/ next subtract~~
- Take a Computer Arithmetic course for more details
- Coming Up: Floating Point