ECE-425: Chapter-3: Arithmetic for Computers

Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

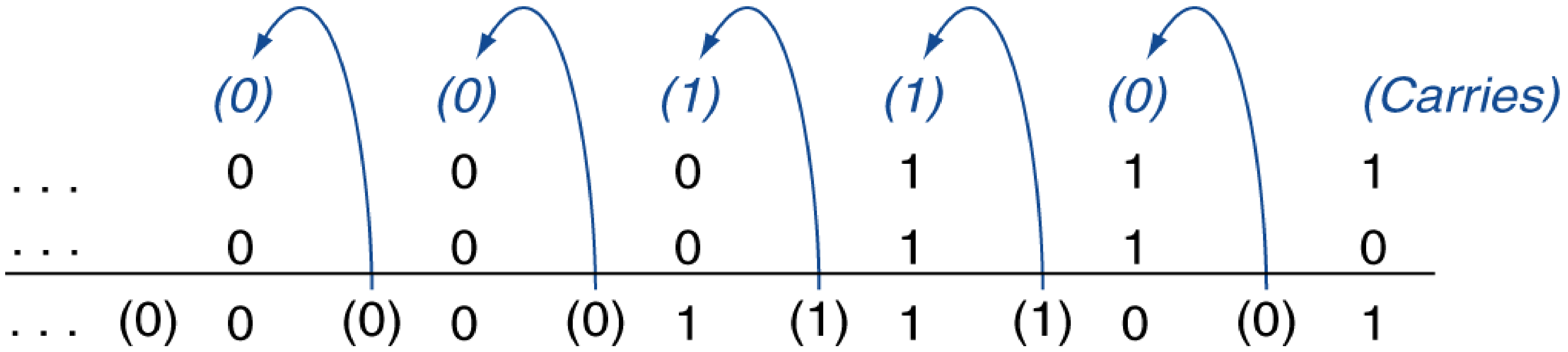
Office: 909-869-2594

Arithmetic for Computers

- **Operations on Integers**
  - Addition and Subtraction
  - Multiplication and Division
  - Dealing with Overflow
- **Floating-point real numbers**
  - Representation and Operations

Integer Addition

- Example: 7 + 6



Overflow: if result out of range

- Adding –ve and +ve operands, no overflow
- Adding two +ve operands
  - Overflow if result sign is 1
- Adding two –ve operands
  - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

```
+7:    0000 0000 … 0000 0111
–6:    1111 1111 … 1111 1010
---------------------------
+1:    0000 0000 … 0000 0001
```

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

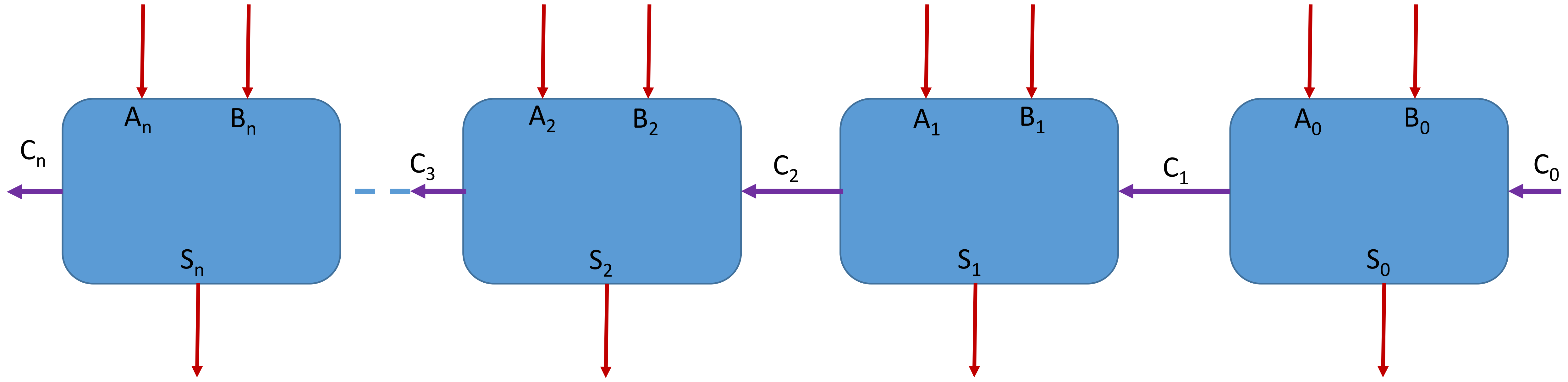
Dealing with Overflow

- **Some Languages (e.g., C) ignore overflow**
  - ✓ Use MIPS addu (Add unsigned), addui (Add immediate unsigned), subu (Subtract unsigned) instructions
- **Other languages (Fortran) require raising an exception**
  - ✓ Use MIPS add (Adding), addi (Add immediate), sub (Subtract) instructions

MIPS detects **overflow** with an exception, also called an interrupt on many computers.

MIPS includes a register called the exception program counter (EPC) to contain the address of the instruction that caused the exception. The instruction move from (**mfc0**) system control is used to copy EPC into a GPR so that MIPS software has the option of returning to the offending instruction via a jump register instruction

Ripple Carry Adder



- $c_2 = b_1c_1 + a_1c_1 + a_1b_1$
- $c_1 = b_0c_0 + a_0c_0 + a_0b_0$
- Substituting for $c_2 = a_1a_0b_0 + a_1a_0c_0 + a_1b_0c_0 + b_1a_0b_0 + b_1a_0c_0 + b_1b_0c_0 + a_1b_1$
- Continuing this to 32-bits yields as fast, but unreasonably expensive adder
- Assume all gate delays are the same regardless of fan-in

Carry-Lookahead Adders

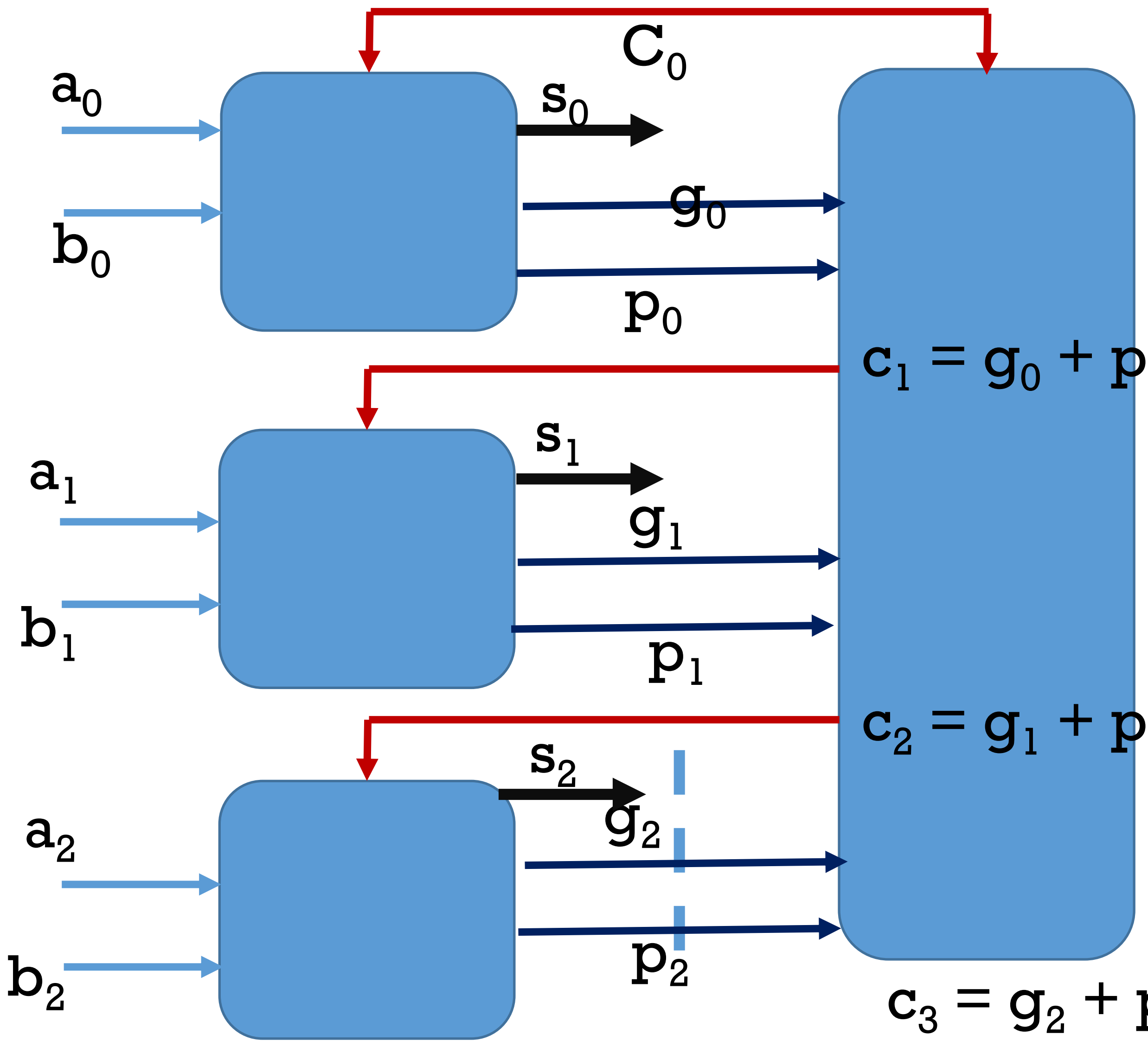
- The basic formula can be rewritten:
  - $c_{i+1} = b_i c_i + a_i c_i + a_i b_i$
  - $c_{i+1} = (b_i + a_i)c_i + a_i b_i$
- Applying it to c_2, we get:
  - $c_2 = (b_1 + a_1)(a_0 b_0 + (a_0 + b_0)c_0) + a_1 b_1$
- Define two "signals" or abstractions:
  - Generate: $g_i = a_i * b_i$
  - Propagate: $p_i = a_i + b_i$
- Redefine c_{i+1} as:
  - $c_{i+1} = g_i + p_i * c_i$
- So $c_{i+1} = 1$ if
  - $g_i = 1$ (generate) or
  - $p_i = 1$ and $c_i = 1$ (propagate)

Carry-Lookahead Adders

- The basic formula can be rewritten:
  - $c_{i+1} = b_i c_i + a_i c_i + a_i b_i$
  - $c_{i+1} = (b_i + a_i)c_i + a_i b_i$
- Applying it to c_2, we get:
  - $c_2 = (b_1 + a_1)(a_0 b_0 + (a_0 + b_0)c_0) + a_1 b_1$
- Define two "signals" or abstractions:
  - Generate: $g_i = a_i * b_i$
  - Propagate: $p_i = a_i + b_i$
- Redefine c_{i+1} as:
  - $c_{i+1} = g_i + p_i * c_i$
- So $c_{i+1} = 1$ if
  - $g_i = 1$ (generate) or
  - $p_i = 1$ and $c_i = 1$ (propagate)

- Our logic equations are simpler:
  - $c_1 = g_0 + p_0 c_0$
  - $c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$
  - $c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$
  - $c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

Carry-Lookahead Adders



$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1g_0 + p_1p_0c_0$

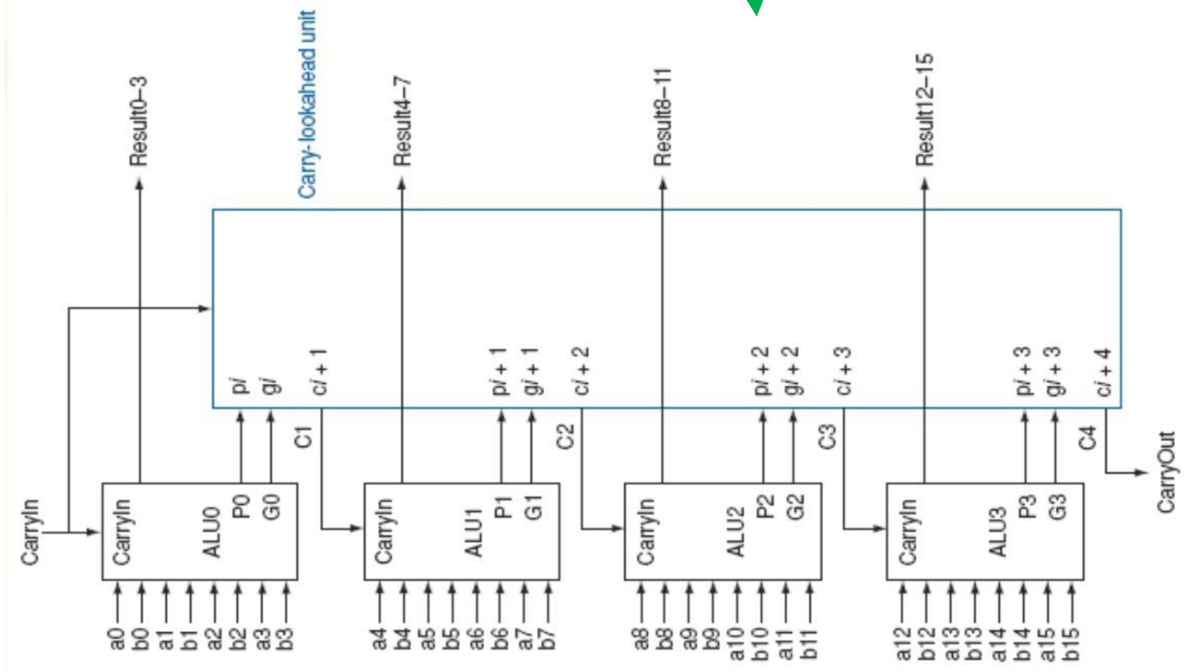
$c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

Carry-Lookahead Adders

16-bit adder performance

$= T_{add} + \max(P_i, G_i) = 2 + 2 + 1 = 5$

- How much better (16-bit adder)?
  - Ripple-carry: $16 * T_{add} = 16 * 2 = 32$ gate delays
  - Carry-lookahead: $T_{add} + \max(p_i, g_i) = 2 + 2 = 4$
  - Much better, but still too profligate
- What if we apply another level of this abstraction?
  - Use the four-bit adder on the previous slide as a building block
  - Define P and G signals
    - $P_0 = p_3p_2p_1p_0$
    - $G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$
    - Similarly for $P_1 - P_3$ and $G_1 - G_3$
  - Derive equations for $C_1 - C_4$
    - $C_1 = G_0 + P_0 C_0$
    - $C_2 = G_1 + P_1G_0 + P_1P_0C_0$, etc.



Arithmetic for Multimedia

- **Graphics and media processing operates on vectors of 8-bit and 16-bit data**
  - ❑ **Use 64-bit adder, with partitioned carry chain**
    - ✓ **Operate on 8x8-bit, 4x16-bit, or 2x32-bit vectors**
  - ❑ **SIMD (Single-instruction, multiple-data)**

Saturation Arithmetic: a version of arithmetic in which all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value.

Instruction Category	**Operands**
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

- Saturating operations

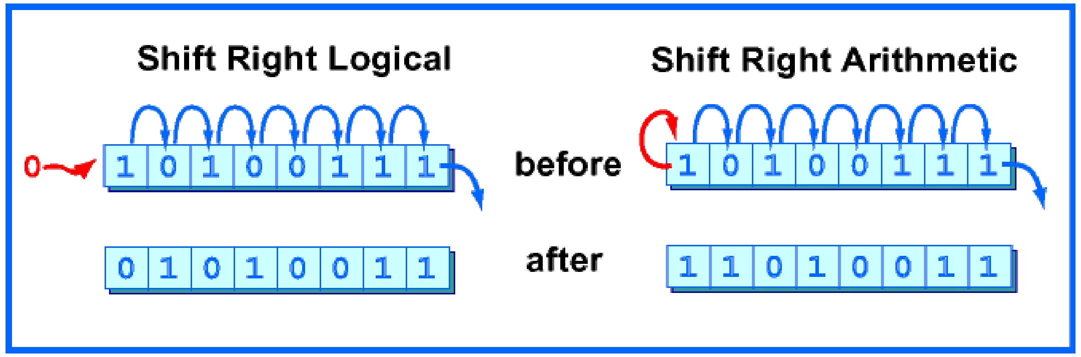
  On overflow, result is largest representable value

  c.f. 2s-complement modulo arithmetic

  E.g., clipping in audio, saturation in video

Shifters

- **Two Kinds:**
- **Logical: value shifted in is always "0"**
- **Arithmetic: sign-extend on right shifts**



Arithmetic Shift	Logical Shift
1) Sign bit is preserved	1) Sign bit is not preserved
2) In left arithmetic shift, 0's are shifted to left keeping sign bit.	2) In left logical shift 0's are replaced by discarded bits.
3) In right arithmetic sign bit is shifted to the right keeping sign bit as is.	3) In right logical shift inserts value 0 to shifted bits
4) Efficient way to perform multiplication (shifting left n bits) and division (shift right n-bits) of signed integers using power of 2	4) Just performs multiplication operation by shifting left.

Multiplication

- ❑ **Start with Long-multiplication approach**

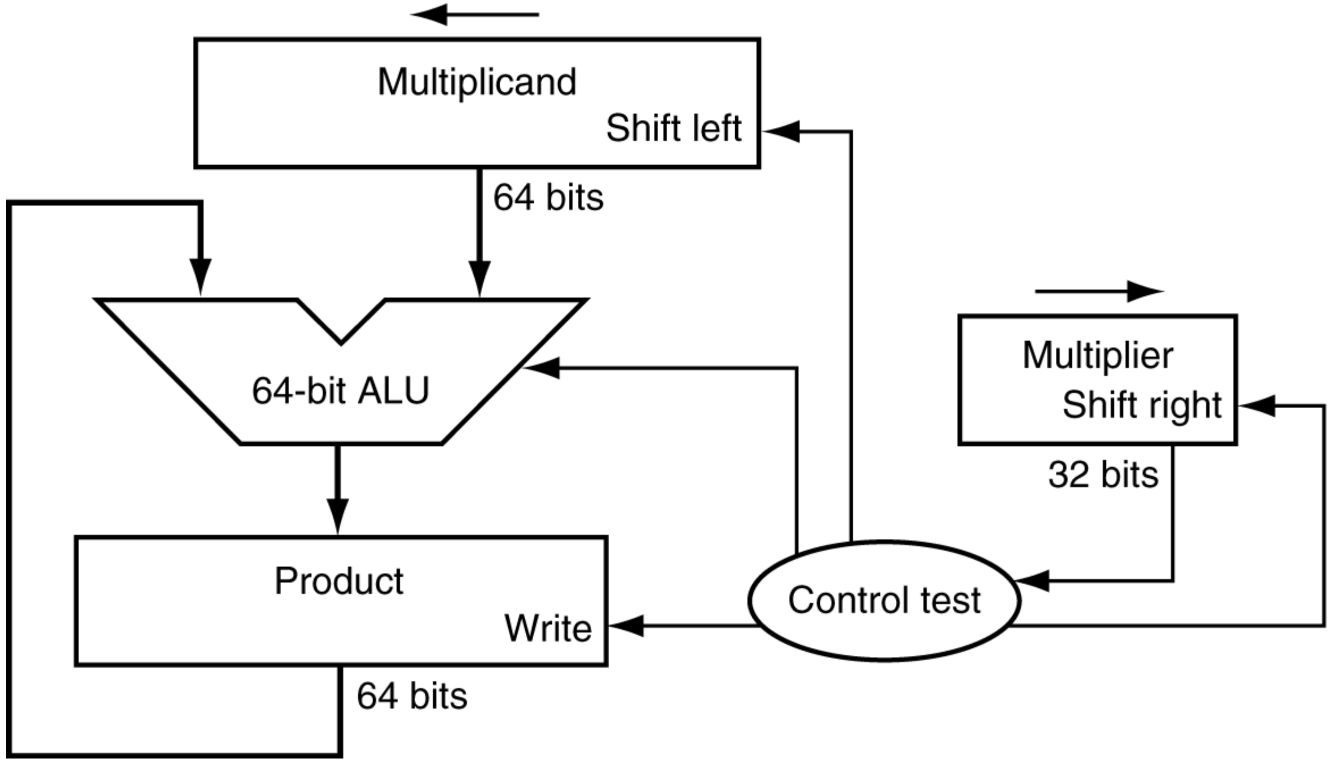
multiplicand

multiplier

product

```
      1000
   × 1001
   ---------
      1000
     0000
    0000
   1000
   ---------
   1001000
```

Length of product is the sum of operand lengths



Add-Shift Method

AQ: holds final result
Size of Q define the number of adding and shifting

11 X 13 = 143
M X Q = AQ

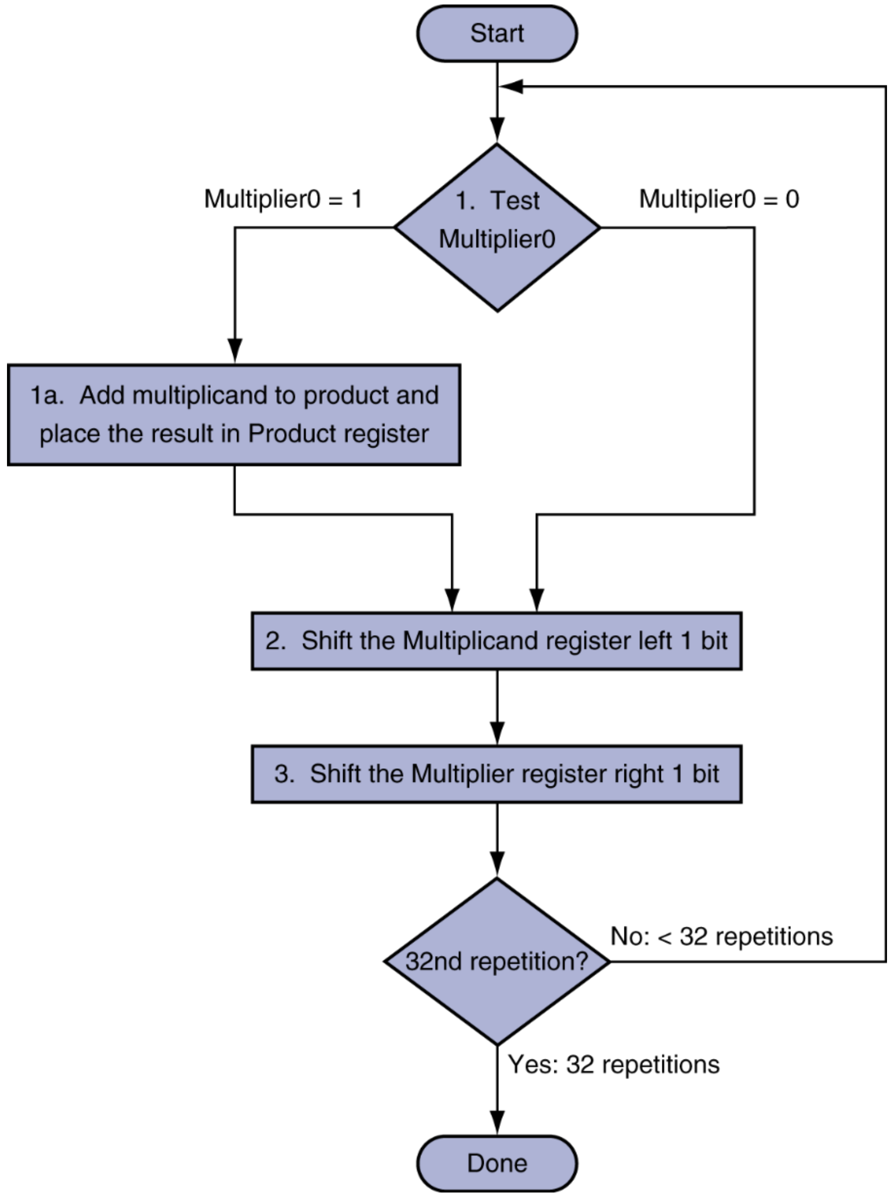
Go for addition=1

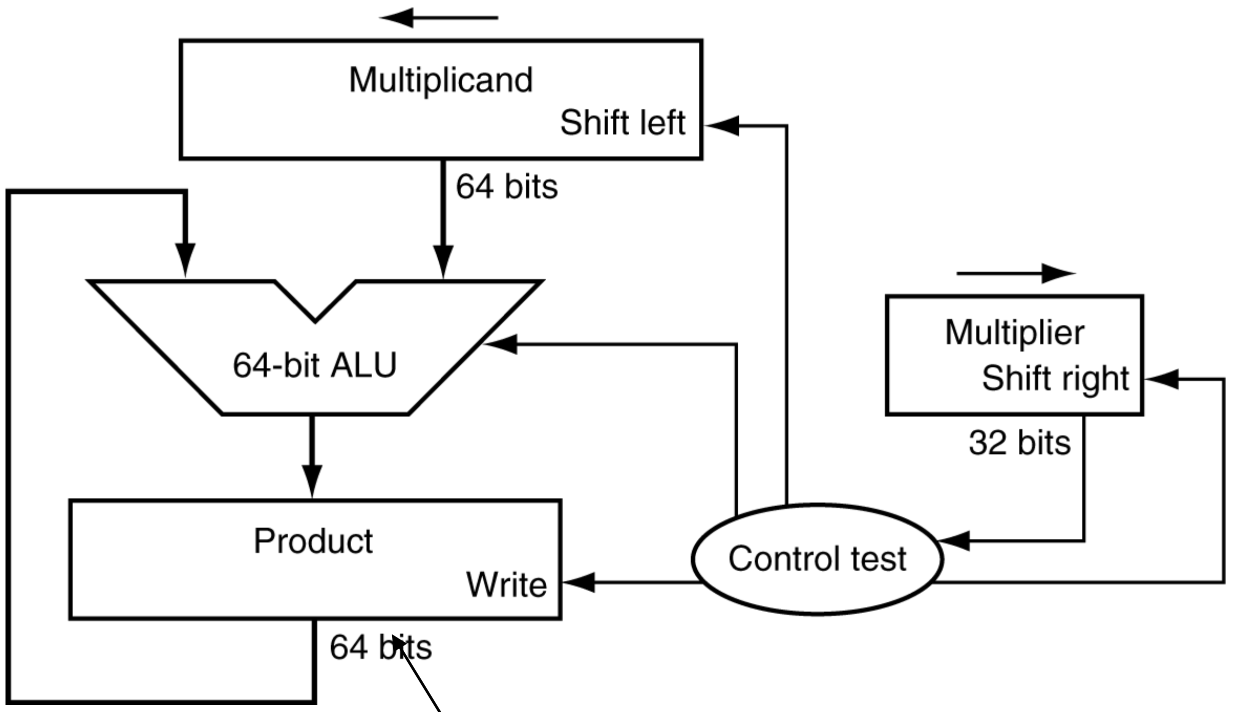
Go for shifting =0

Shift direction →

M	C	A	Q	Operation
1011	0	0000	1101	Initialization
	0	1011	1101	First Cycle: Add M with A A = A+M
	0	0101	1110	Shift-Right CAQ
	0	0010	1111	Second cycle: Shift right CAQ
	0	1101	1111	Third cycle: A = A+M
	0	0110	1111	Shift right CAQ
	1	0001	1111	Fourth Cycle: A = A+M
	0	1000	1111	Shift-Right CAQ

Multiplication Hardware

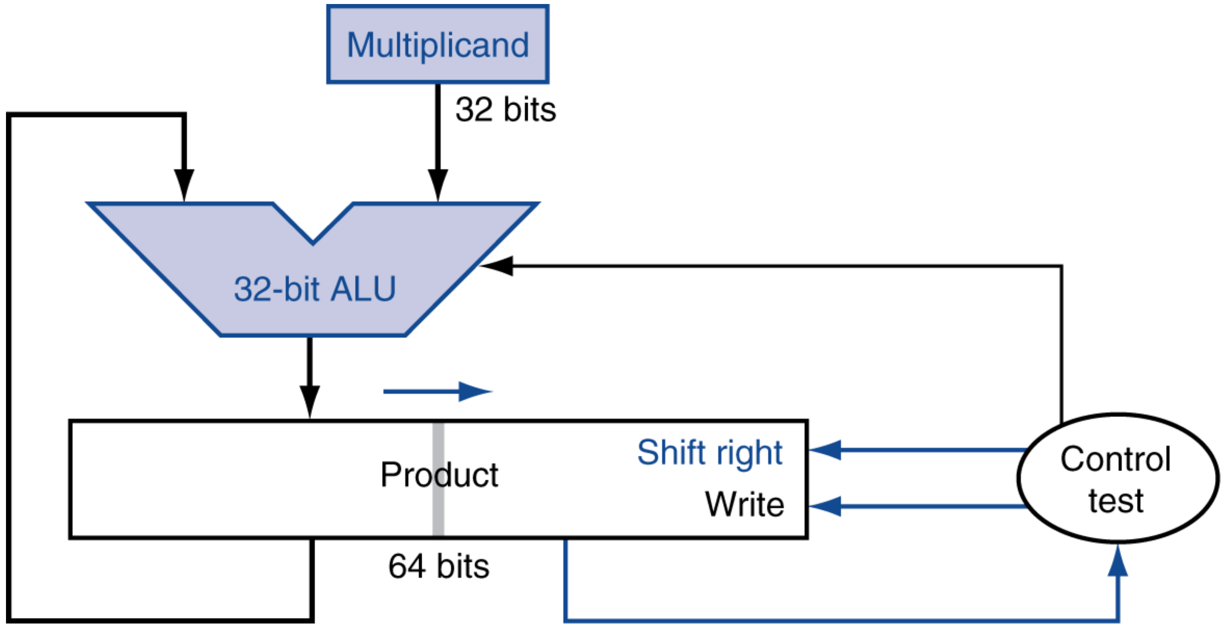

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



Initially 0

Optimized Multiplier

- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low

Booth's Algorithm for Binary Multiplication Example

We will use 5-bits

2_{10} * -4_{10}

2 in base 2 is 00010

-4 in base 2 is 11100

-2 in base2 is 11110

Operations:

00 :	Do Nothing to product register
01 :	Add multiplicand to upper Product register
10 :	Subtract Multiplicand from upper Product register
11 :	Do Nothing to product register

Steps:

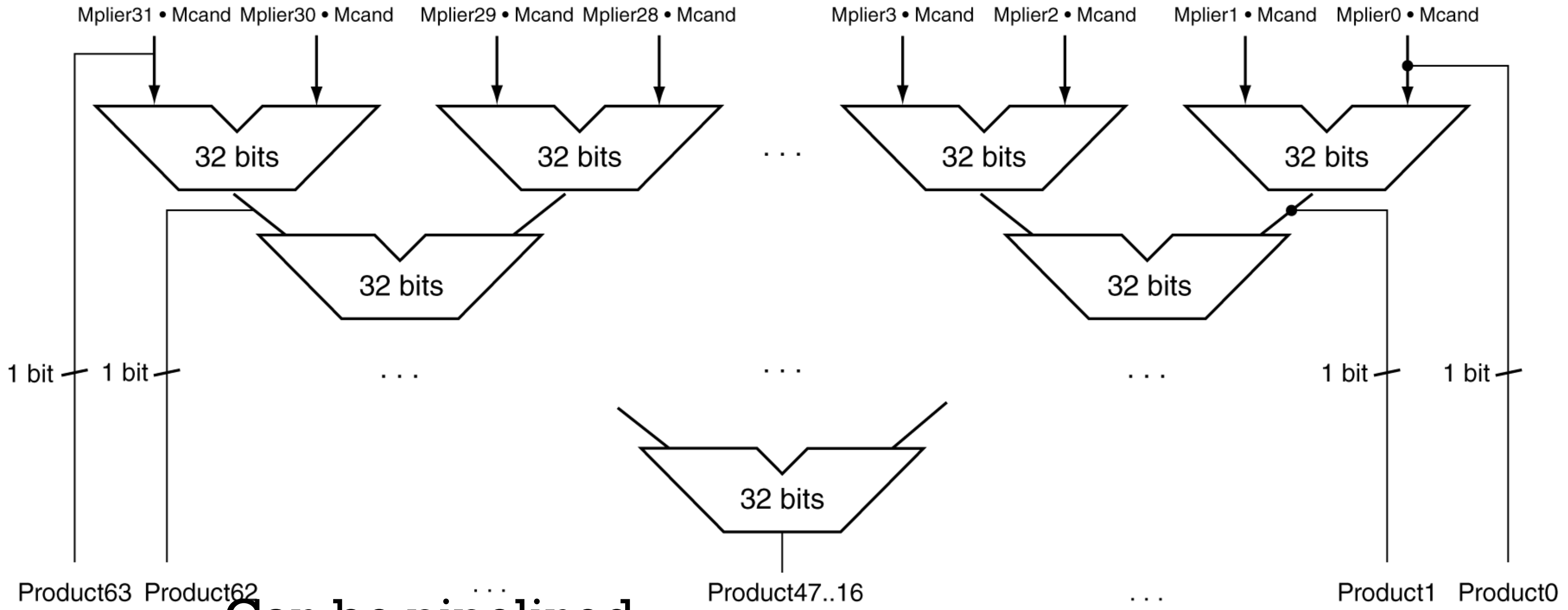
1. Populate the product register with a zero value in the upper half and the multiplier in the lower half. Populate the multiplicand. Populate the Previous Bit with 0 since this is our first step.
2. Populate the previous bit and check the least significant bit of the product register. Perform the correct "operation".
3. Perform an arithmetic right shift of the product register (in other words make sure you keep correct 'sign')
4. Repeat step 2 and 3 until the number of repetitions is greater than or equal to the number of bits. (Remember, we are using 5-bits for our example).

Booth's Algorithm for Binary Multiplication Example

Rep	Product Register	Multiplicand	Current Bit	Previous Bit	Operation on Product Register then shift right
0	00000 11100	00010	0	0	Do Nothing
1	00000 01110	00010	0	0	Do Nothing
2	00000 00111	00010	1	0	Subtract multiplicand from upper product register
2a	11110 00111	New Product but still need to shift right keeping the correct "signed" Bit			
3	11111 00011	00010	1	1	Do Nothing
4	11111 10001	00010	1	1	Do Nothing
5	11111 11000	00010	0	1	Add multiplicand to upper Product Register
5a	00001 $\boxed{11000}$	Last Iteration (#bits = reps) then the answer is in lower half of the register, -8_{10}.			

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

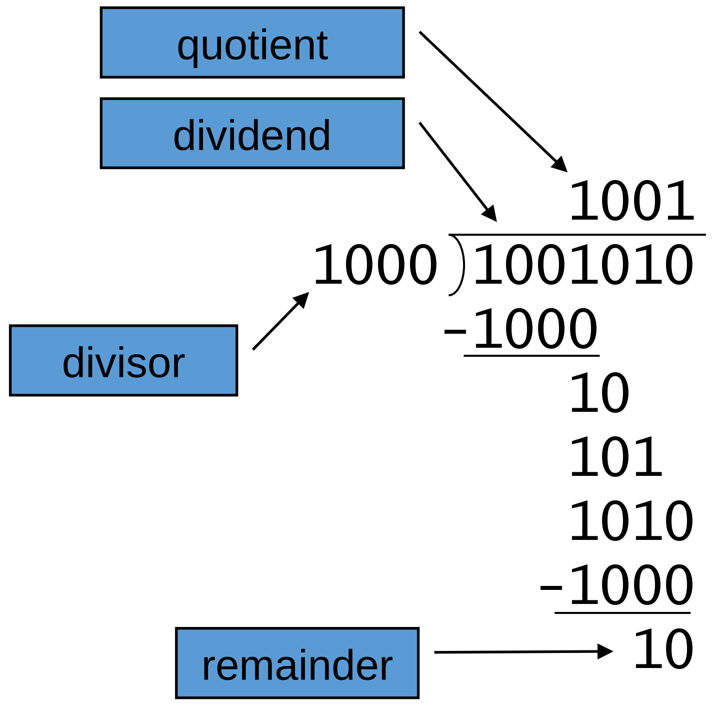


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI/LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - mul rd, rs, rt
    - Least-significant 32 bits of product –> rd

Division



n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

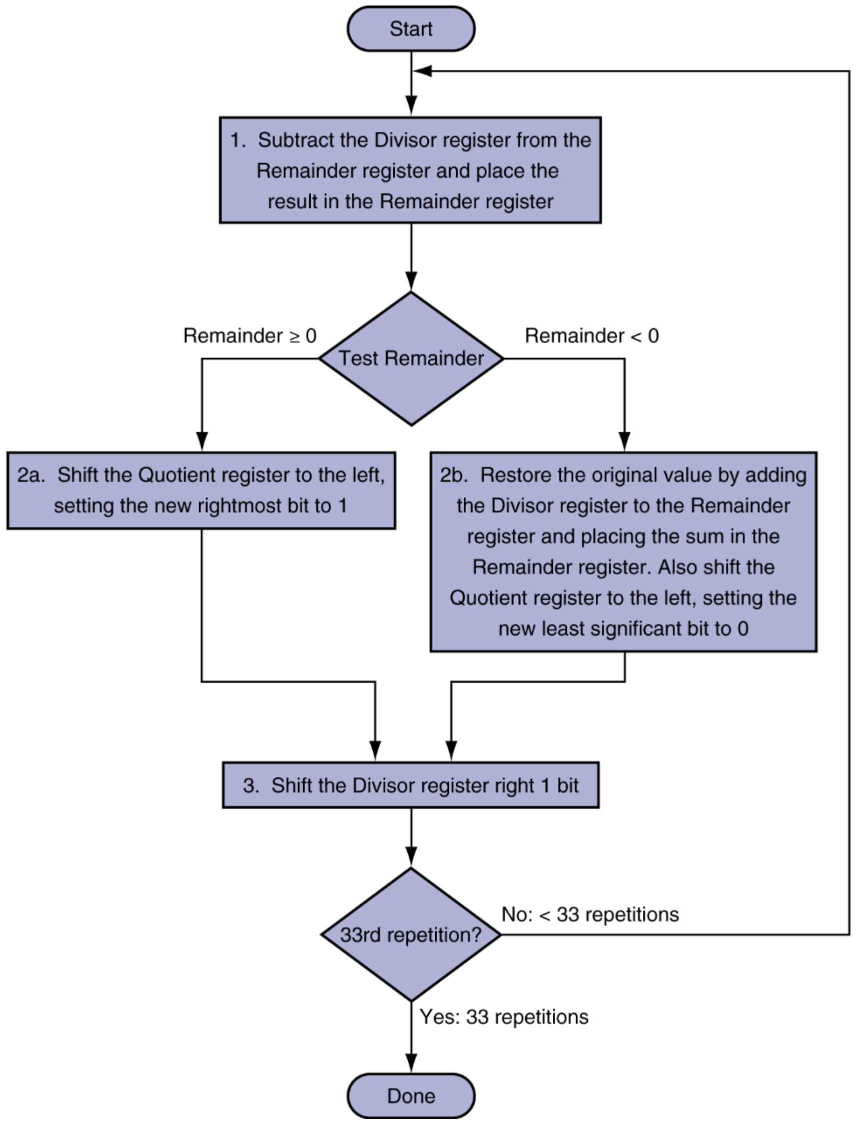
Dividend = Quotient X Divisor + Remainder

Binary Arithmetic Division

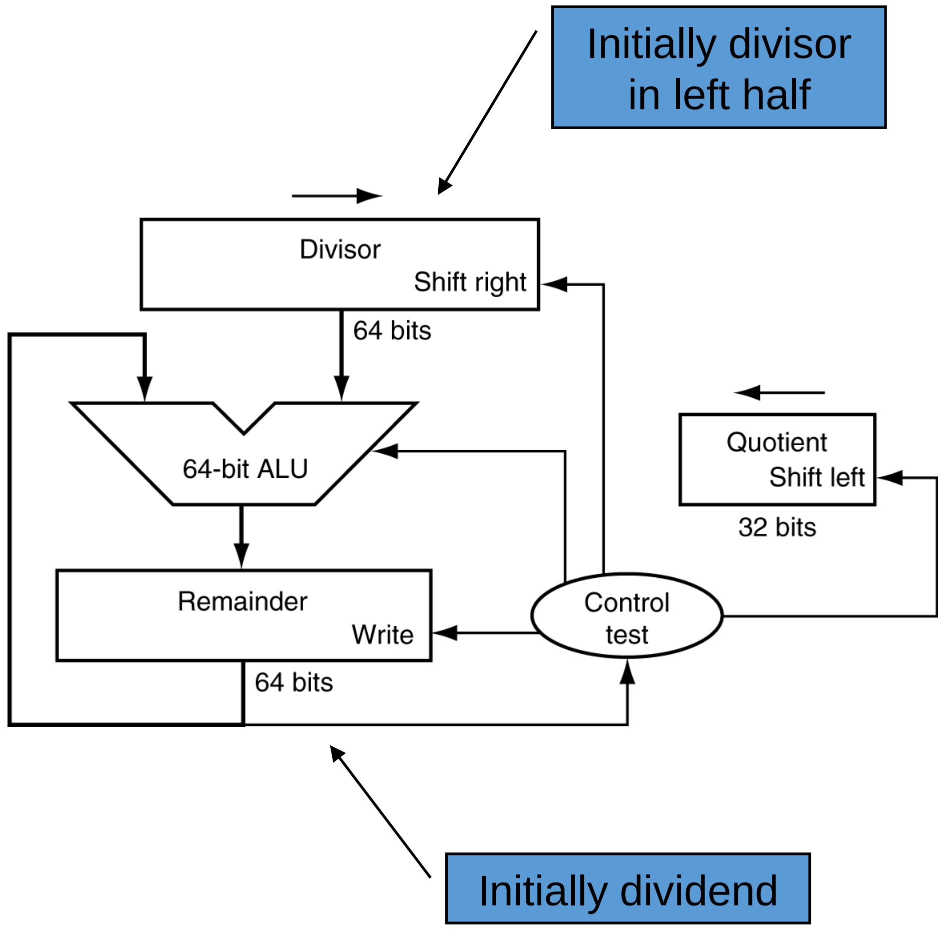
- Divide $(1001110)_2$ by $(100)_2$

```
100)  1001110 ( 10011. 1)  ➔ 19.5
     −100
     ---------
      000111
         −100
     ---------
           0110
        − 0100
     ---------
           0100
        − 0100
     ---------
           0000
```

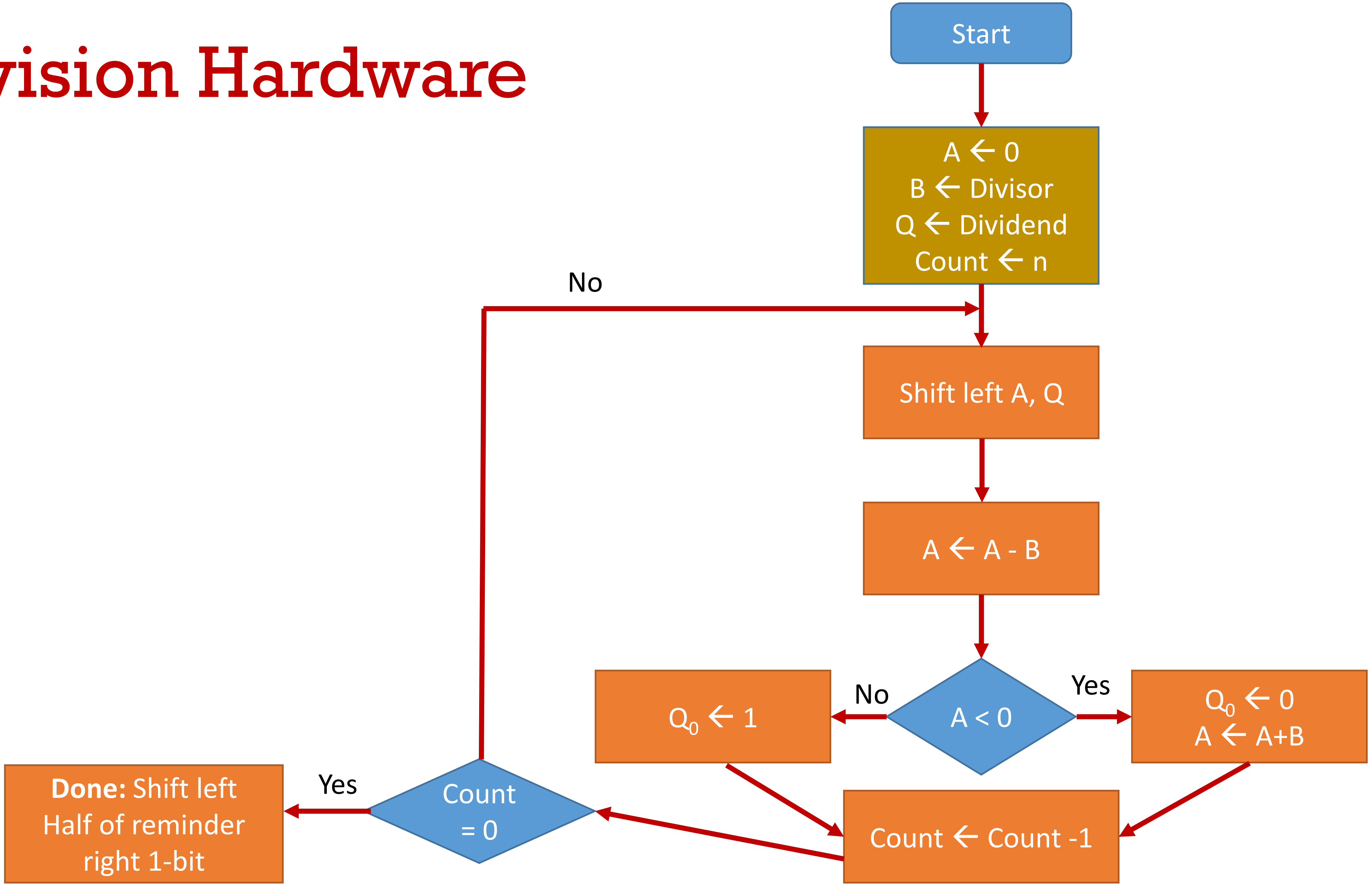
Division Hardware



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Division Hardware



Division Hardware (Restoring Division method)

Example :

Dividend = 1010

Divisor = 0011 =>B =>00011

B+1 = 11101

n = 4

```
Operation:     A         Q
Initially  :   00000     1010
Shift      :   00001     010■
Subtract   :   11101
              ------
               11110
Set Q0                  0
Restore A      00011+
              ------
              100001     0100
                                  Cycle 1

Left shift    00010     100■
Subtract      11101
             ------
              11111
Set Q0                        0
Restore       00011     100
             ------
            1 00010
                                  Cycle 2

Shift          00101    000■
Subtract       11101
              ------
              100010
Set Q0                          1

               00010    000 1
                 A        Q
                                  Cycle 3
```

Division Hardware

Example :

Dividend = 1010

Divisor = 0011 =>B =>00011

$\overline{B}$+1 = 11101

n = 4

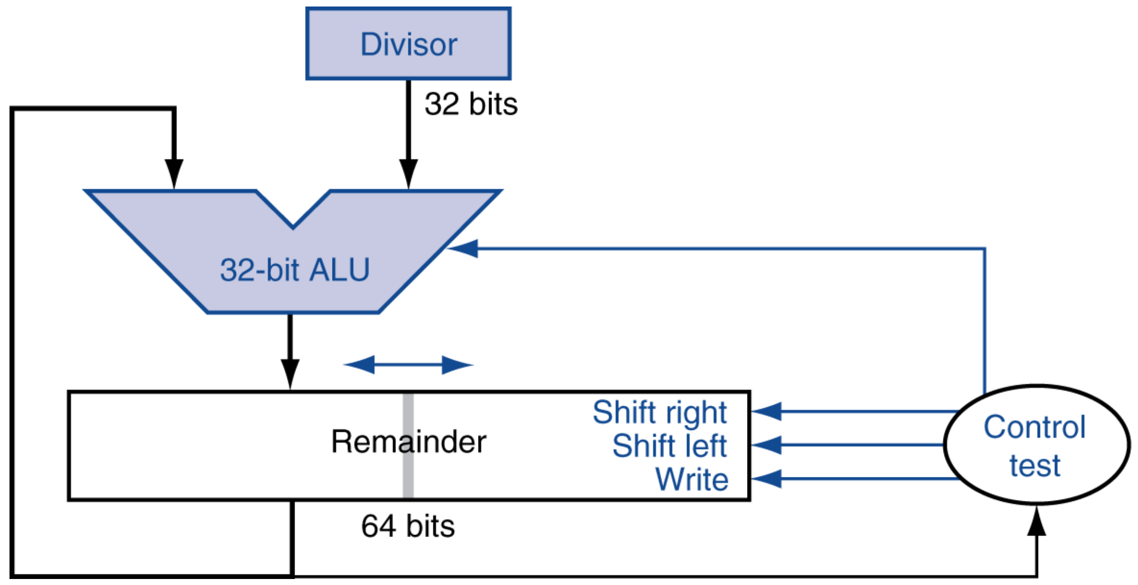
Solution:

R = 00001

Q = 0011

$$\left.\begin{array}{lll} & 00010 & 0001 \\ \text{Shift} & 00100 & 001\boxed{} \\ \text{Subtract} & \underline{11101} & \\ & \cancel{00001} & \\ \text{Set Q0} & \underbrace{00001}_{\text{Remainder}} & \underbrace{001\boxed{1}}_{\text{Quotient}} \end{array}\right\} \text{Cycle 4}$$

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

MIPS Division

- Use HI/LO registers for result
    - HI: 32-bit remainder
    - LO: 32-bit quotient
- Instructions
    - div rs, rt / divu rs, rt
    - No overflow or divide-by-0 checking
        - Software must perform checks if required
    - Use mfhi, mflo to access result

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

normalized

not normalized

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
    - Portability issues for scientific code
- Now almost universally adopted
- Two representations
    - Single precision (32-bit)
    - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- https://www.mips.com/
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html
- Professor El-Naga ECE-425 notes