ECE260: Fundamentals of Computer Engineering

Arithmetic for Computers

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

YORK COLLEGE OF PENNSYLVANIA

Based on Computer Organization and Design, 5th Edition by Patterson & Hennessy

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Binary Integer Addition

- Benefit of 2's complement integer representation:
  - Same binary addition procedure will work for adding both signed and unsigned numbers
- If result is out of range, ***overflow*** occurs
  - Adding positive and negative operands
    - No overflow will occur
  - Adding two positive operands
    - Overflow occurred if sign bit of result is 1
  - Adding two negative operands
    - Overflow occurred if sign bit of result is 0

- Example: $7_{ten} + 6_{ten}$

(1)	*(1)*	*(0)*		*(Carries)*
0	1	1	1	
0	1	1	0	
1	1	0	1	

Grade school style!

- Example expanded to show carries inline

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

Binary Integer Subtraction

- Two options:
  - Subtract numbers directly grade school style
  - Negate 2nd operand and perform an addition
- If result is out of range, ***overflow*** occurs
  - Subtracting two positive or two negative operands
    - No overflow will occur
  - Subtracting positive from negative operand
    - Overflow occurred if sign bit of result is 0
  - Subtracting negative from positive operand
    - Overflow occurred if sign bit of result is 1

- Example: 7_{ten} - 6_{ten}
  - Grade school style

$$\begin{array}{lrcl} & 0000\ 0111_{two} & = & 7_{ten} \\ - & 0000\ 0110_{two} & = & 6_{ten} \\ \hline = & 0000\ 0001_{two} & = & 1_{ten} \end{array}$$

  - Negate 2nd operand and add

$$\begin{array}{lrcl} & 0000\ 0111_{two} & = & 7_{ten} \\ + & 1111\ 1010_{two} & = & -6_{ten} \\ \hline = & 0000\ 0001_{two} & = & 1_{ten} \end{array}$$

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Up to the programmer to address potential overflow issues
- Other languages (e.g., Fortran, Ada) will cause an ***exception*** if overflow occurs
  - Exception notifies programmer so that overflow can be handled
- In MIPS, overflow behavior is as follows:
  - Signed instructions raise exceptions (e.g. add, addi, sub)
  - Unsigned instructions do not raise exceptions (e.g. addu, addiu, subu)

- The following table summarizes the results that indicate overflow occurred

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- Examples:
  - Result = $Op_A + Op_B$
    IF ($Op_A \geq 0$ and $Op_B \geq 0$ and Result < 0)
    THEN overflow occurred
  - Result = $Op_A - Op_B$
    IF ($Op_A \geq 0$ and $Op_B < 0$ and Result < 0)
    THEN overflow occurred

Integer Multiplication

- Here's the classic grade school "Times Table"
  - At some point you probably memorized this

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

- Multiplying two numbers together looks something like this:

Multiplicand				A_3	A_2	A_1	A_0
Multiplier			x	B_3	B_2	B_1	B_0
				A_3B_0	A_2B_0	A_1B_0	A_0B_0
			A_3B_1	A_2B_1	A_1B_1	A_0B_1	
		A_3B_2	A_2B_2	A_1B_2	A_0B_2		
+	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
Product						RESULT	

- ***Note***: multiplying N-digit number by M-digit number gives (N+M)-digit result

Binary Integer Multiplication

- Once again, it's the same as grade school multiplication, only easier

- The "Times Table" is significantly smaller

×	0	1
0	0	0
1	0	1

but the process is exactly the same!

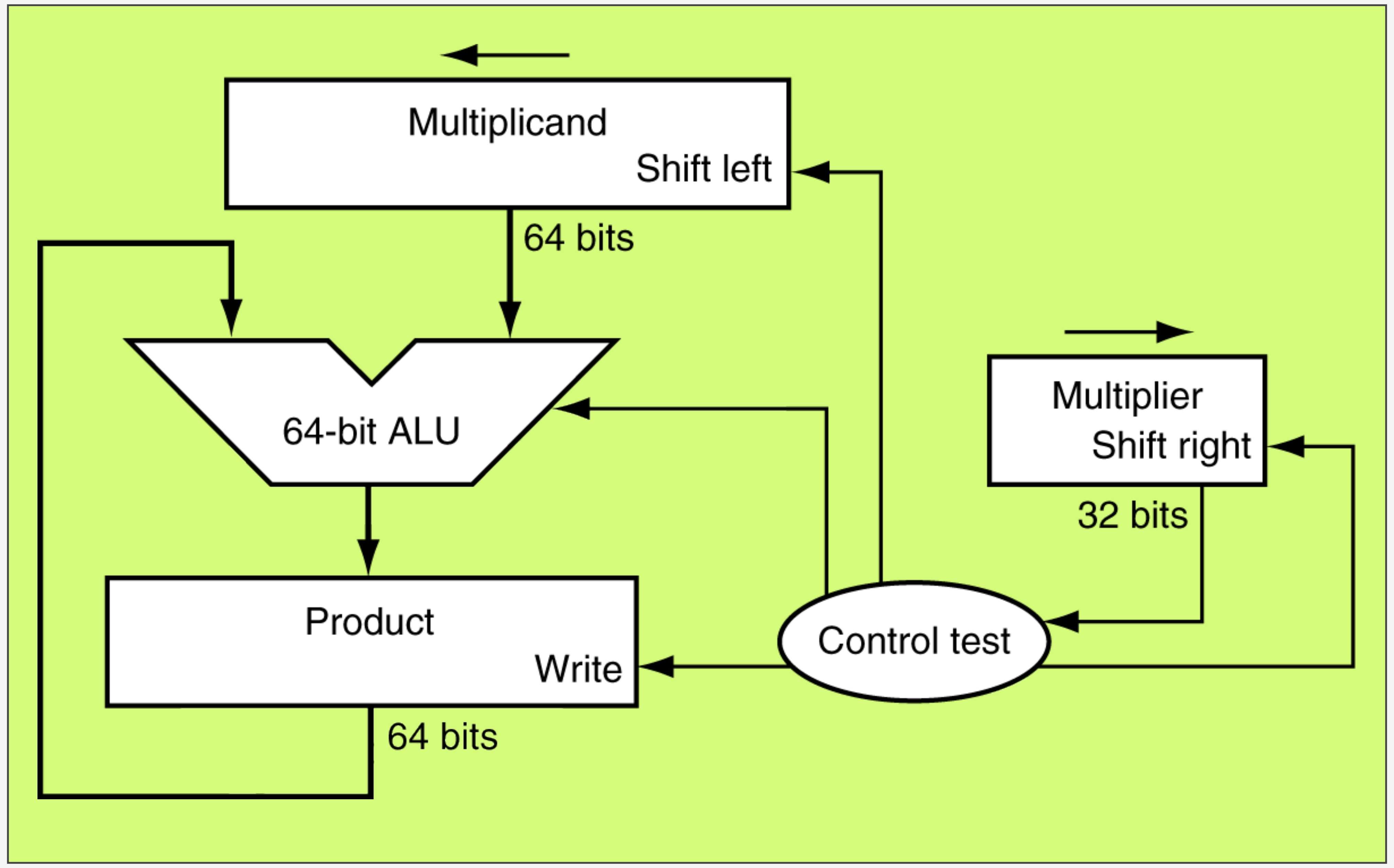
- Example of multiplying two numbers together:

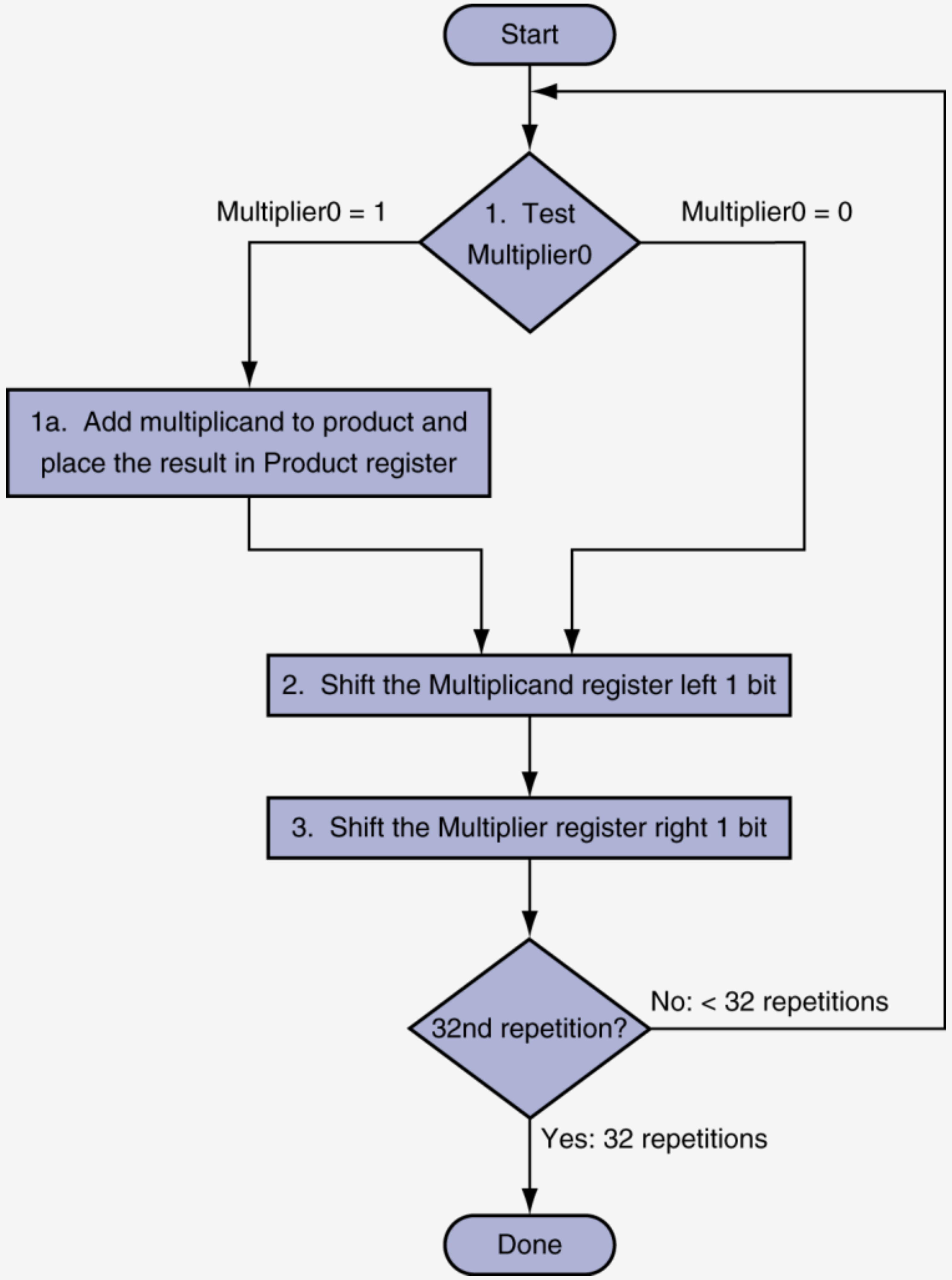
Multiplicand					1	0	0	0two
Multiplier				×	1	0	0	1two
					1	0	0	0
				0	0	0	0	
			0	0	0	0		
	+	1	0	0	0			
Product		1	0	0	1	0	0	0

- ***Note***: multiplying two 4-bit numbers together produces an 8-bit result

Multiplication Hardware & Algorithm

- Basic hardware for 32-bit architecture
  - 64-bit registers for multiplicand and product
  - 32-bit register for multiplier
  - 64-bit ALU to perform repeated additions

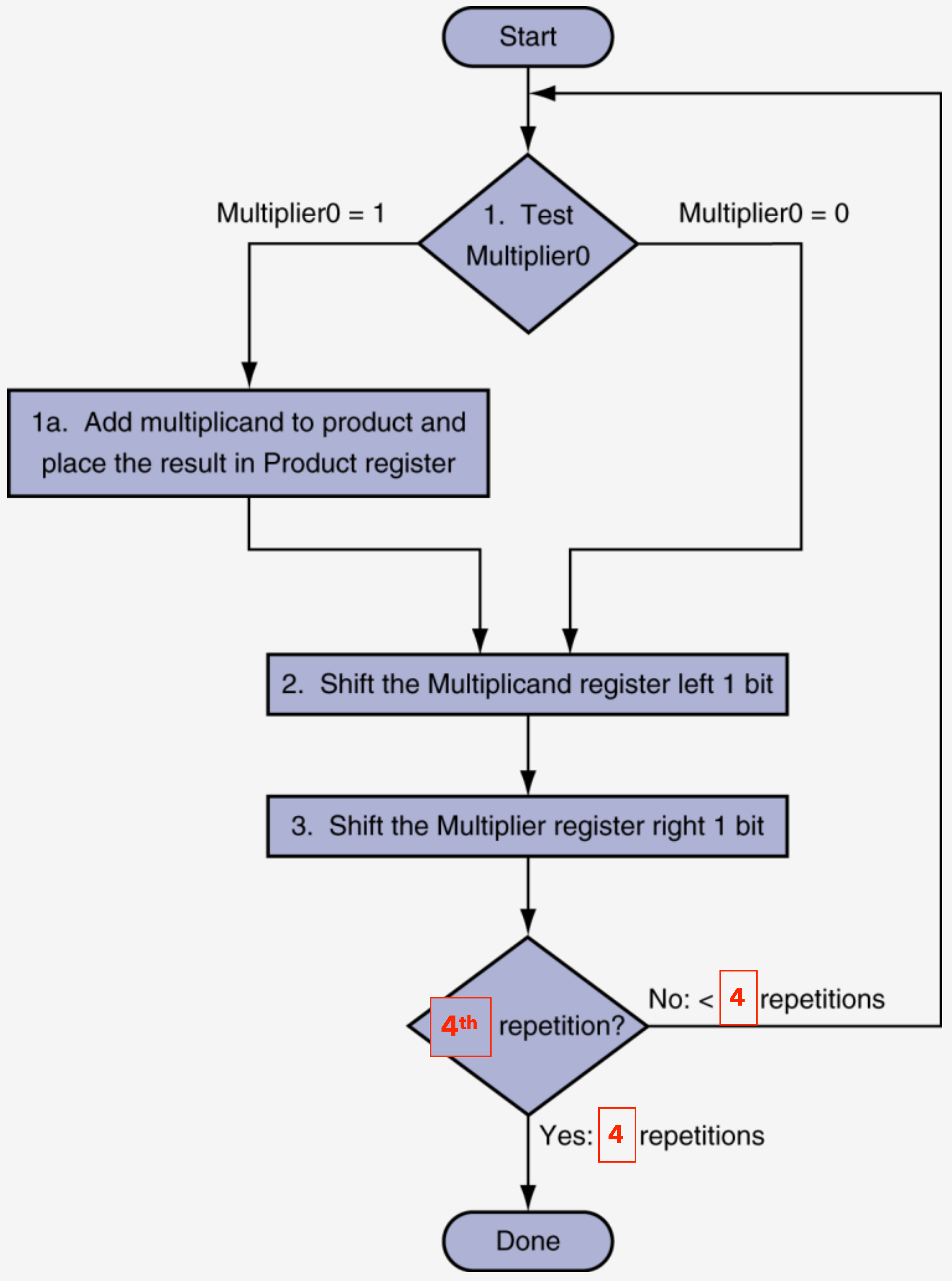




Multiplication Example

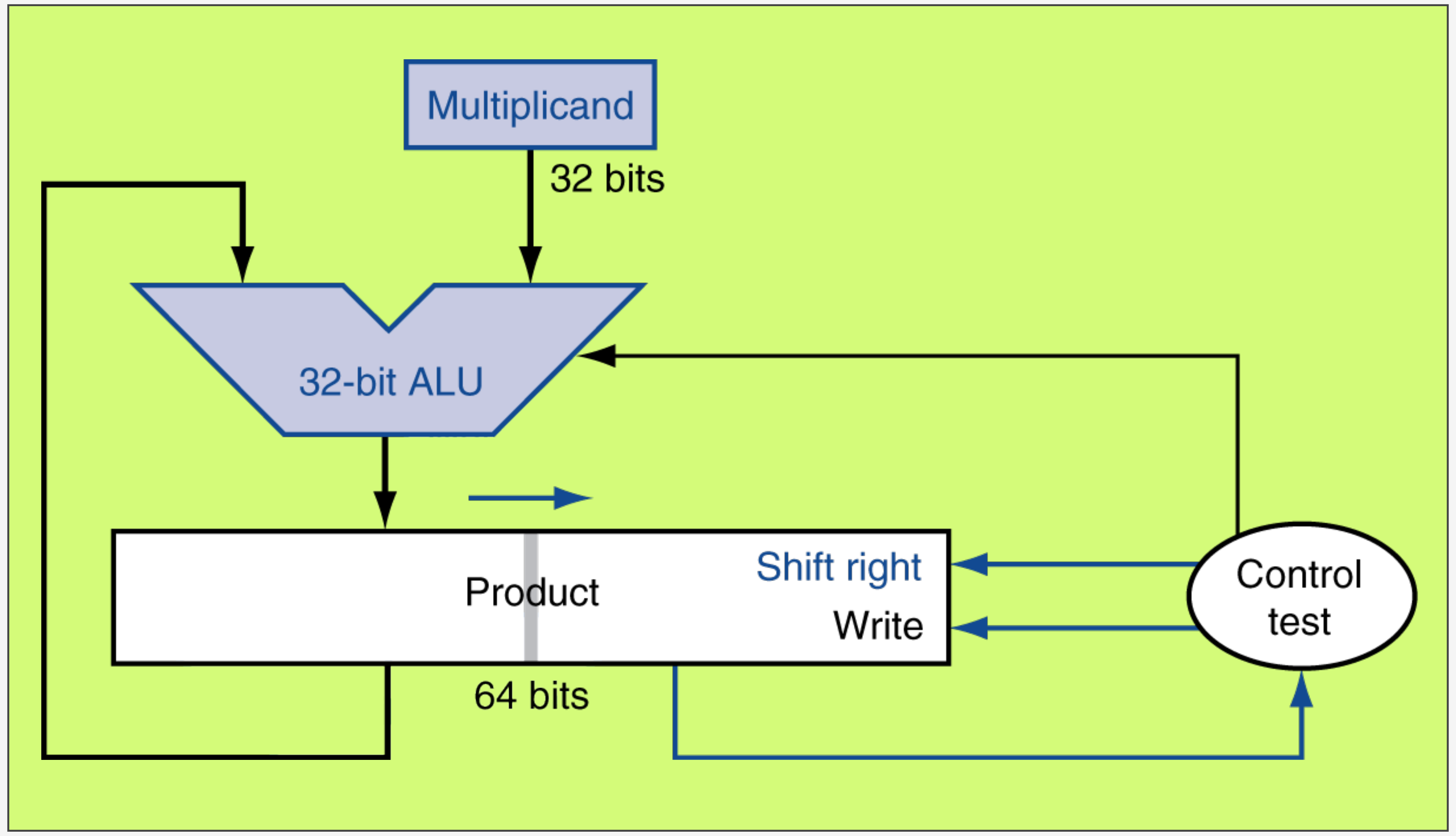
- Multiplication example using basic hardware and 4-bit inputs
  - 4-bit example requires only 4 iterations, not 32
  - Initialize Product register to 0
  - Example: $2_{ten} \times 3_{ten}$

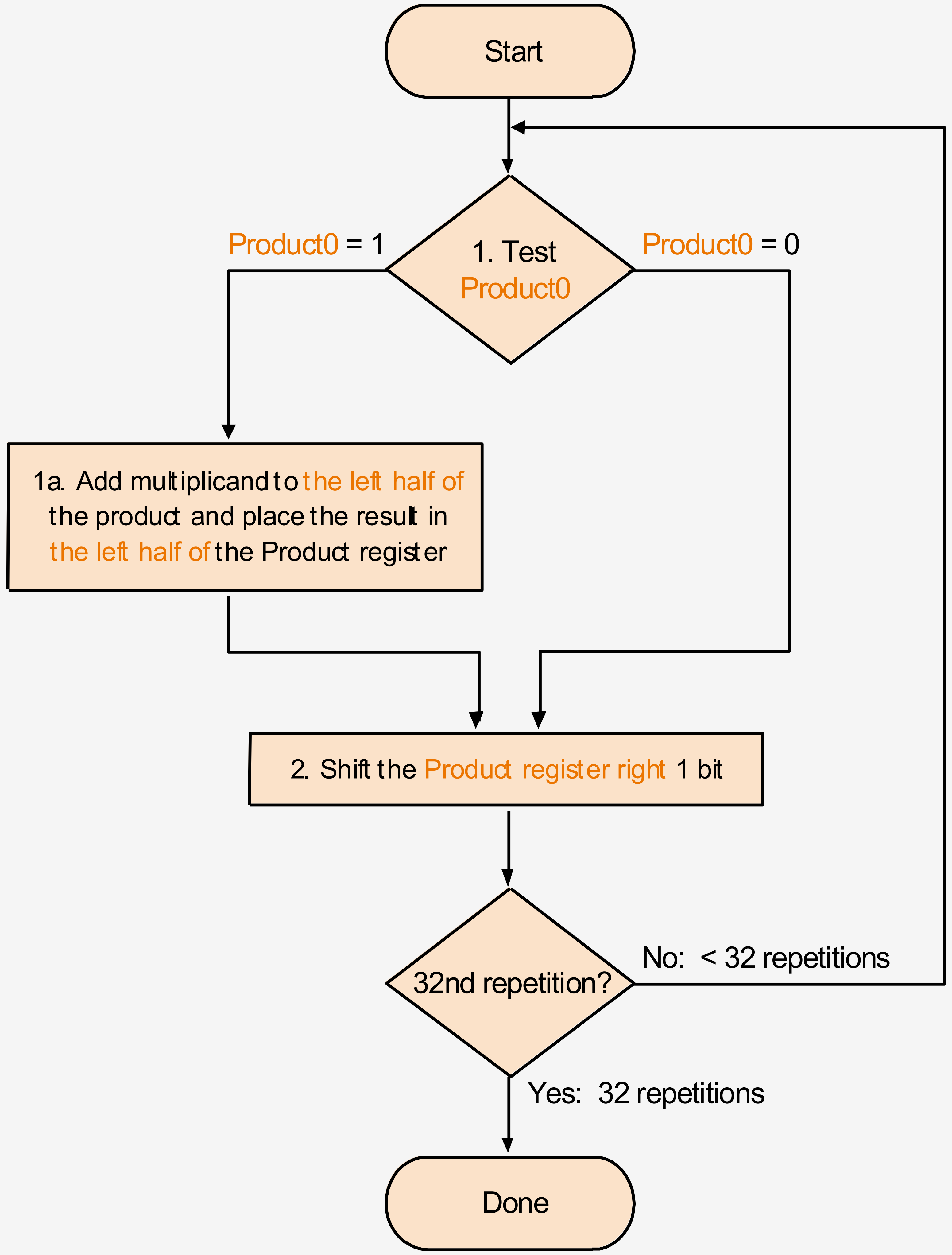
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110



Optimized (for size) Multiplication Hardware

- Reduced hardware requirements
  - Multiplicand register and ALU now 32-bit
  - Multiplier no longer has dedicated register
    - Right half of Product register is initialized with multiplier, left half initialized to zero



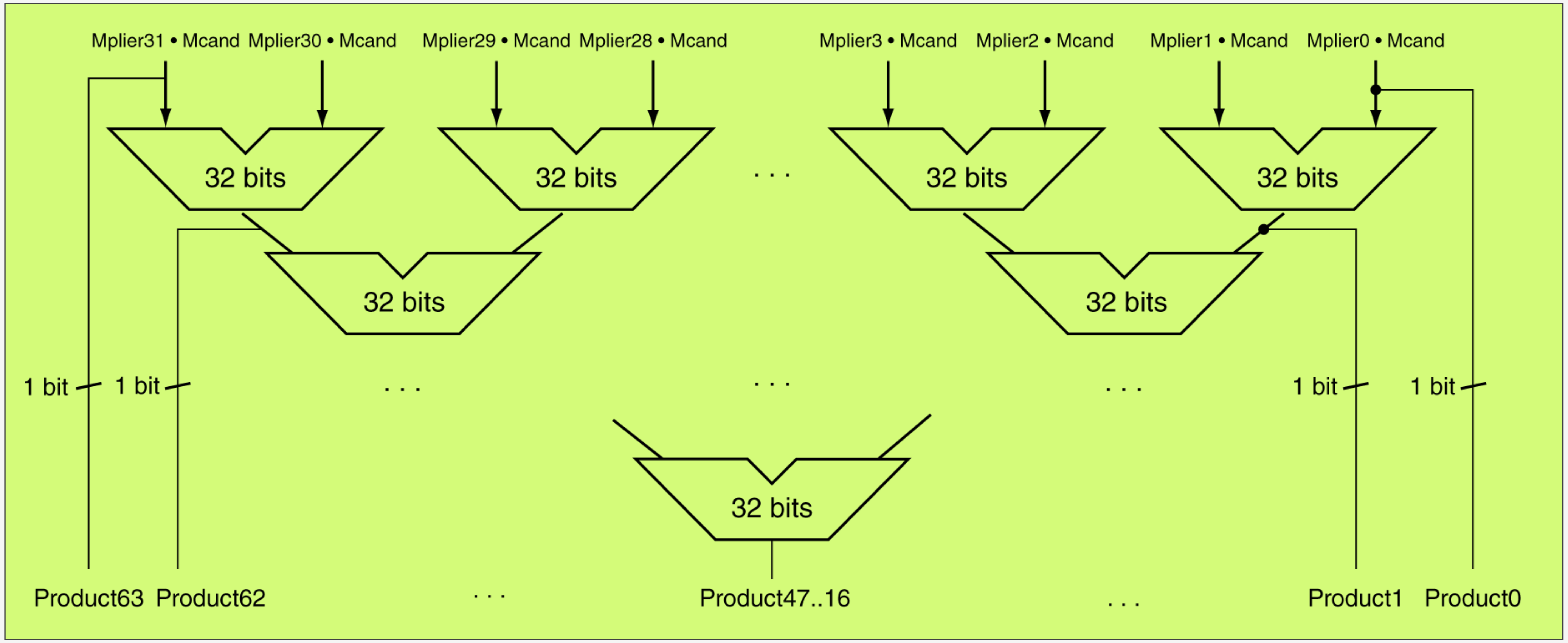


Multiplication Example #2

- Be sure to try out the previous multiplication example using the optimized hardware!
  - Example: $2_{ten} \times 3_{ten}$

A Faster Multiplier

- Uses multiple adders in a tree structure
  - Requires more silicon but can be pipelined to perform much faster
    - Cost/performance tradeoff



Signed Multiplication

- Recall from grade school arithmetic that the Product is negative if the signs of the Multiplicand and the Multiplier differ

```
positive × positive = positive
negative × negative = positive
positive × negative = negative
```

- Thus, in hardware:
  - Perform the multiplication algorithm for 31 iterations (not 32) (this ignores the sign bit)
  - If the original sign bits differed, then negate the result
  - Be sure to do sign extension during computation when shifting multiplier to the right

Binary Integer Division

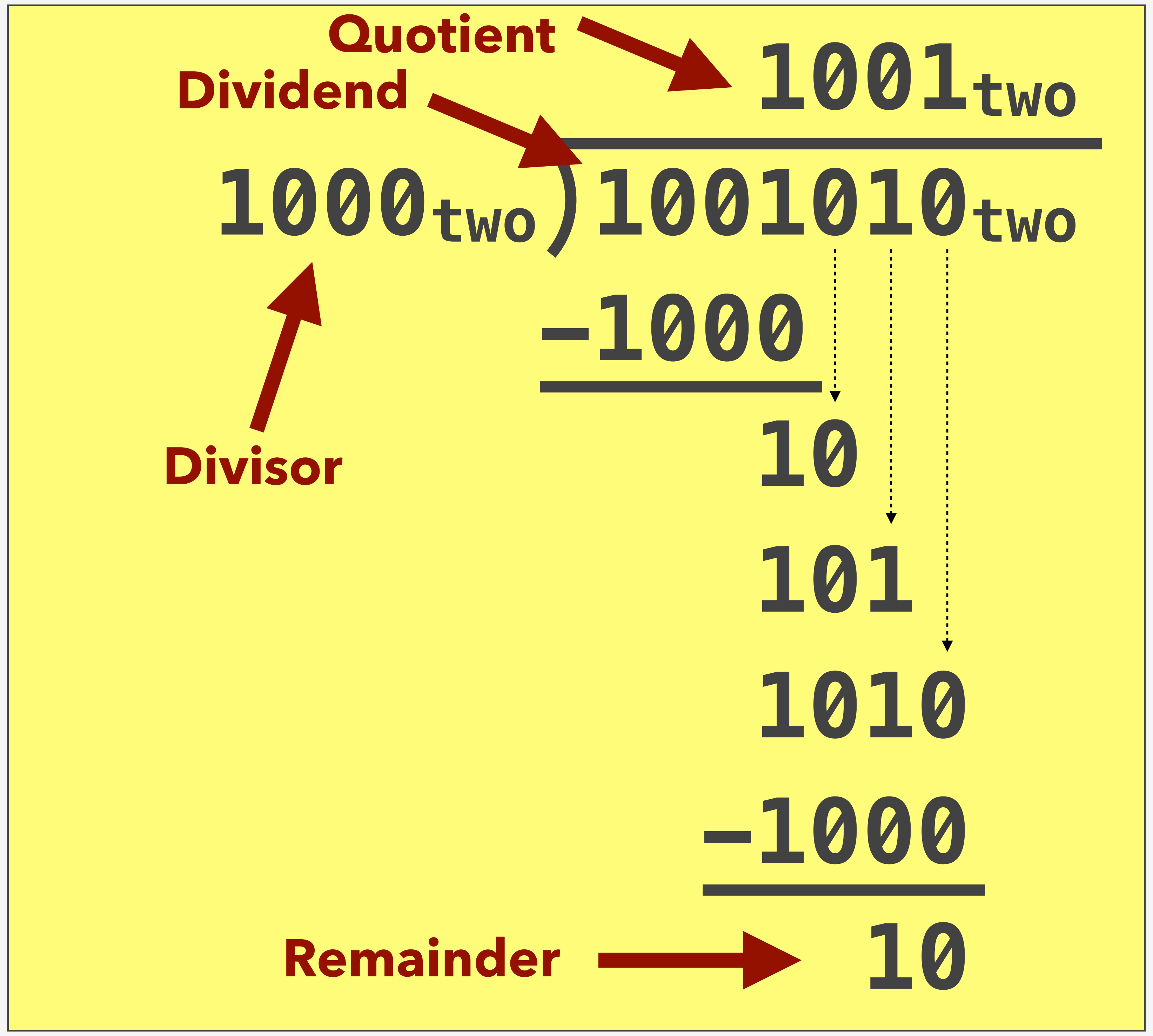
- The grade school long-division algorithm works for binary integer division
  - First, ensure that the divisor is not 0
  - IF divisor ≤ dividend THEN

    place a 1 in the quotient and subtract the divisor from the dividend

    ELSE

    place a 0 in the quotient and expand the dividend to include the next bit
  - When dividend is exhausted, whatever is left over is the remainder

- Example of long-division on binary numbers:



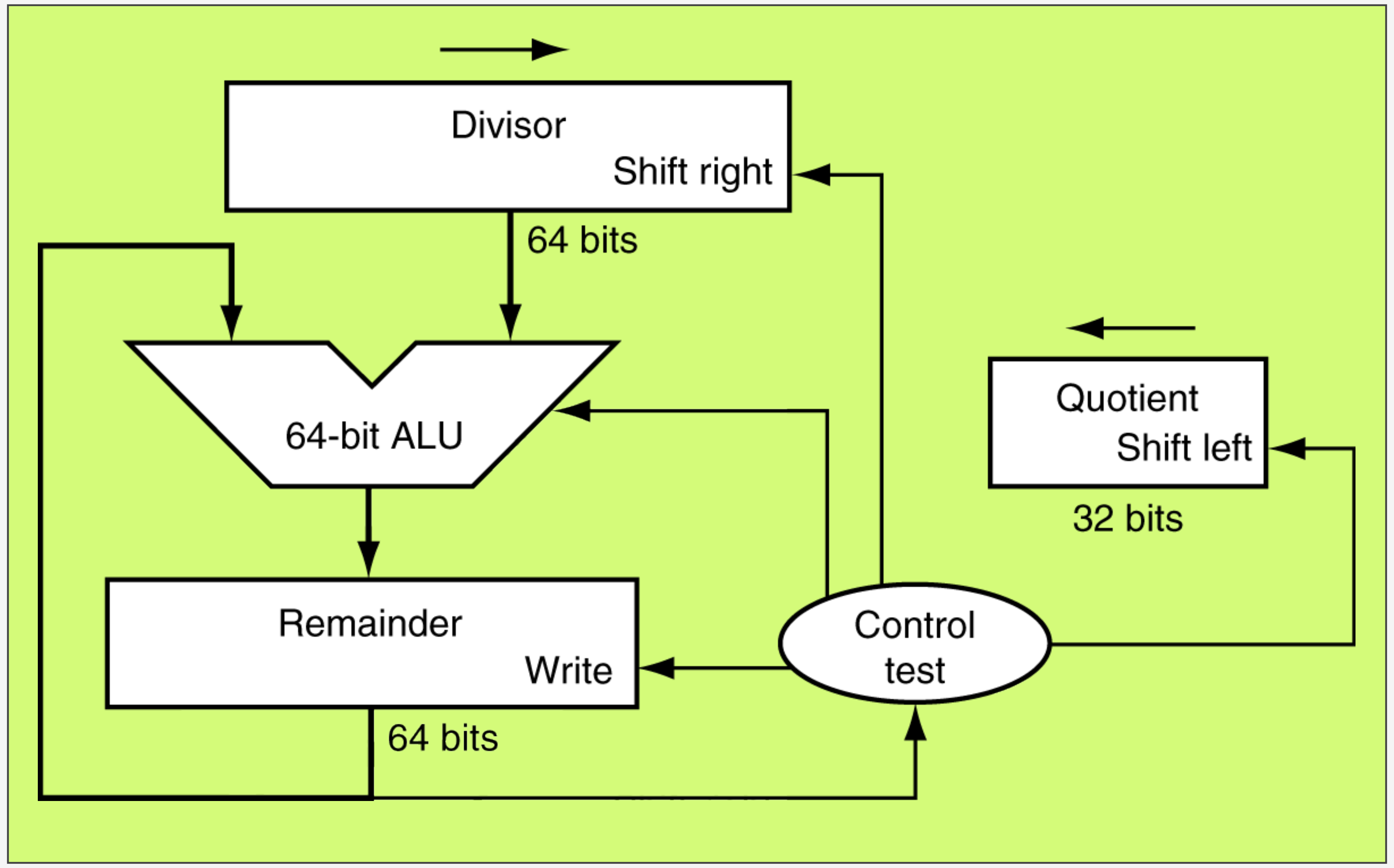
  - Note: dividing n-bit operands yields an n-bit quotient and an n-bit remainder

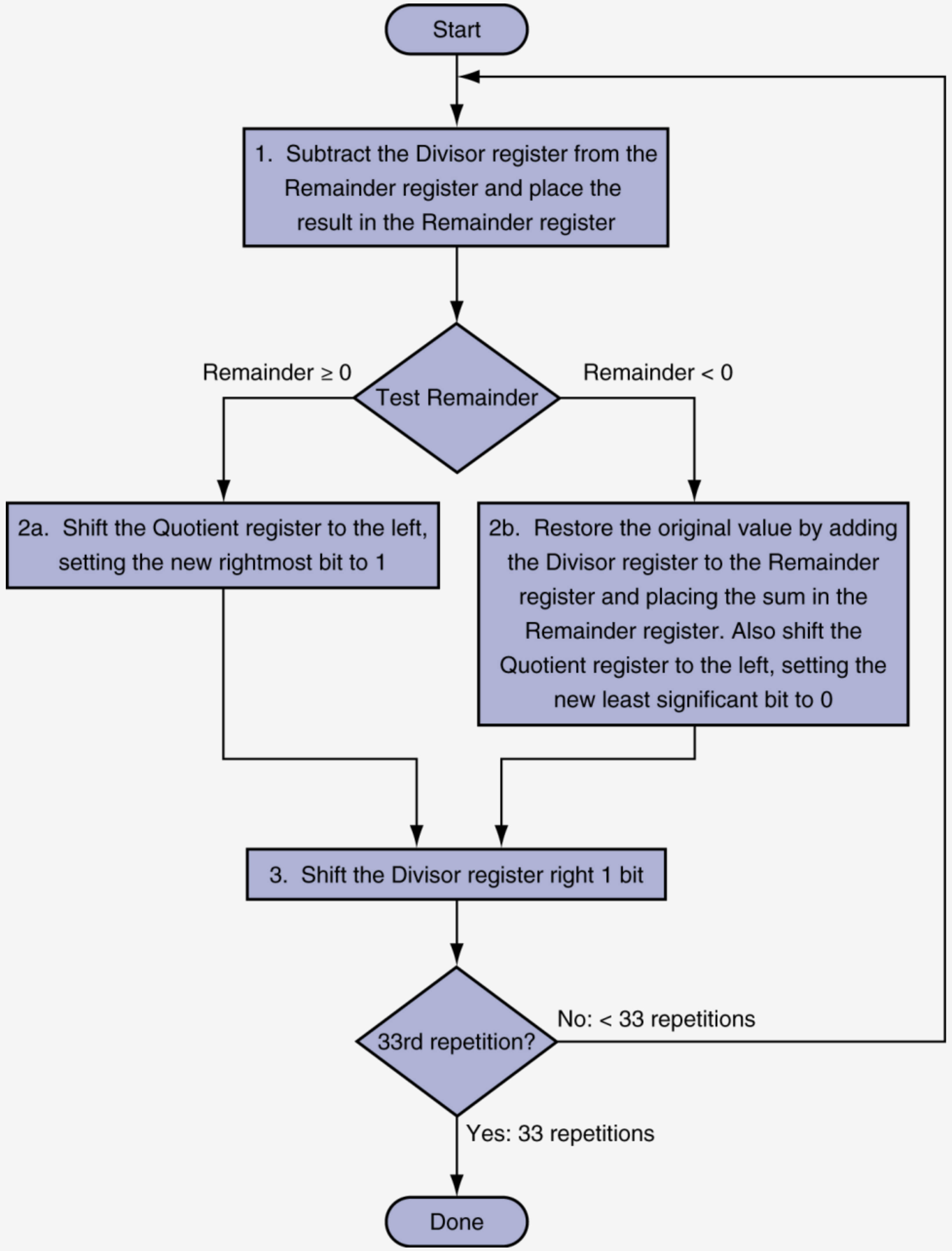
Binary Integer Division

- Several different approaches to perform long-division in hardware
  - ***Restoring division*** subtracts the divisor from the dividend without comparing them
    - If the result of the subtraction is < 0, then the dividend was smaller than the divisor
      - Insert a 0 into the quotient
      - Restore the dividend to its previous state by adding the divisor back in
    - If the result of the subtraction is ≥ 0, then the dividend was larger than or equal to the divisor
      - Insert a 1 into the quotient

Division Hardware & Algorithm

- Basic hardware for 32-bit architecture
  - 64-bit registers for divisor and remainder
  - 32-bit register for quotient
  - 64-bit ALU to perform repeated sub/add ops

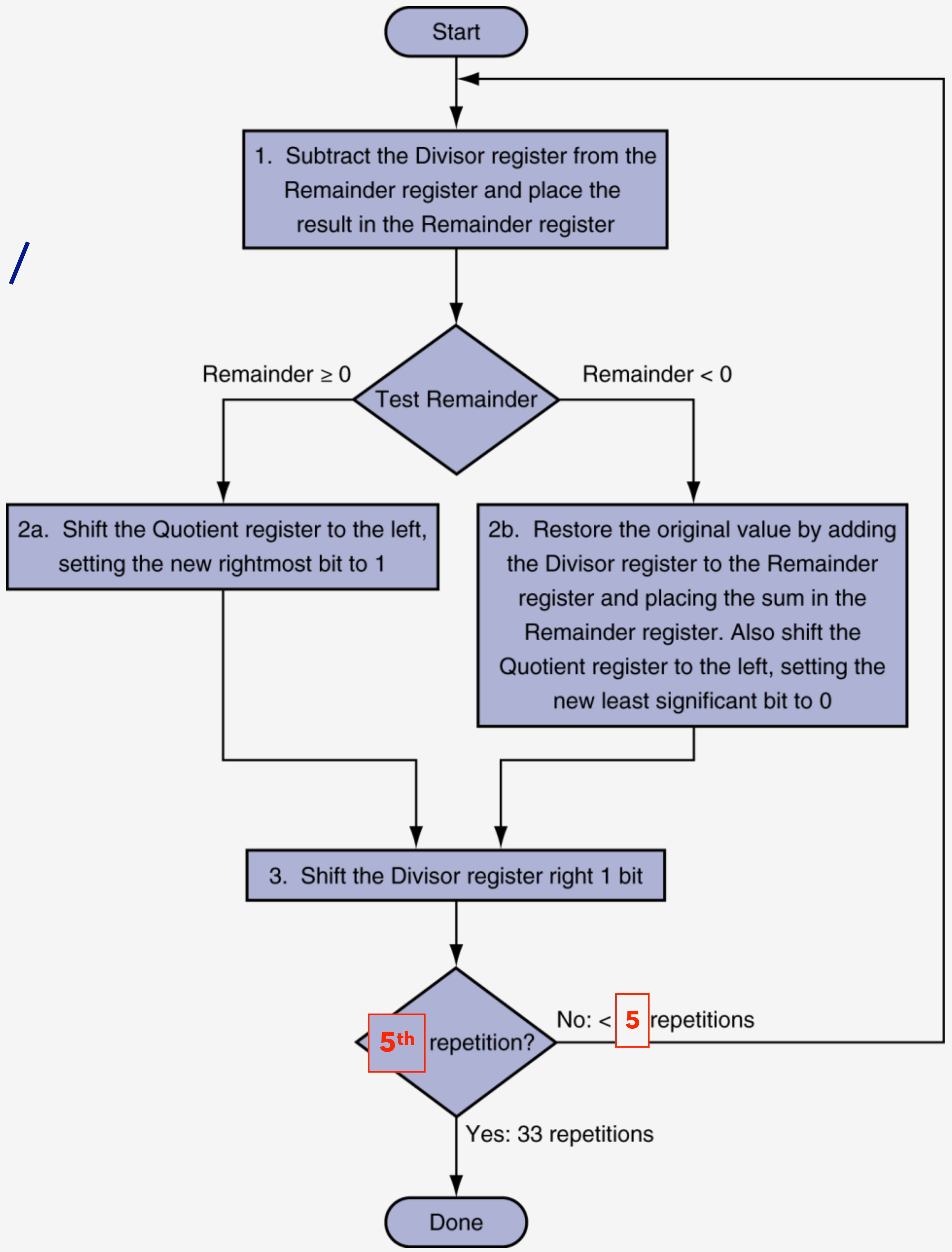




Division Example

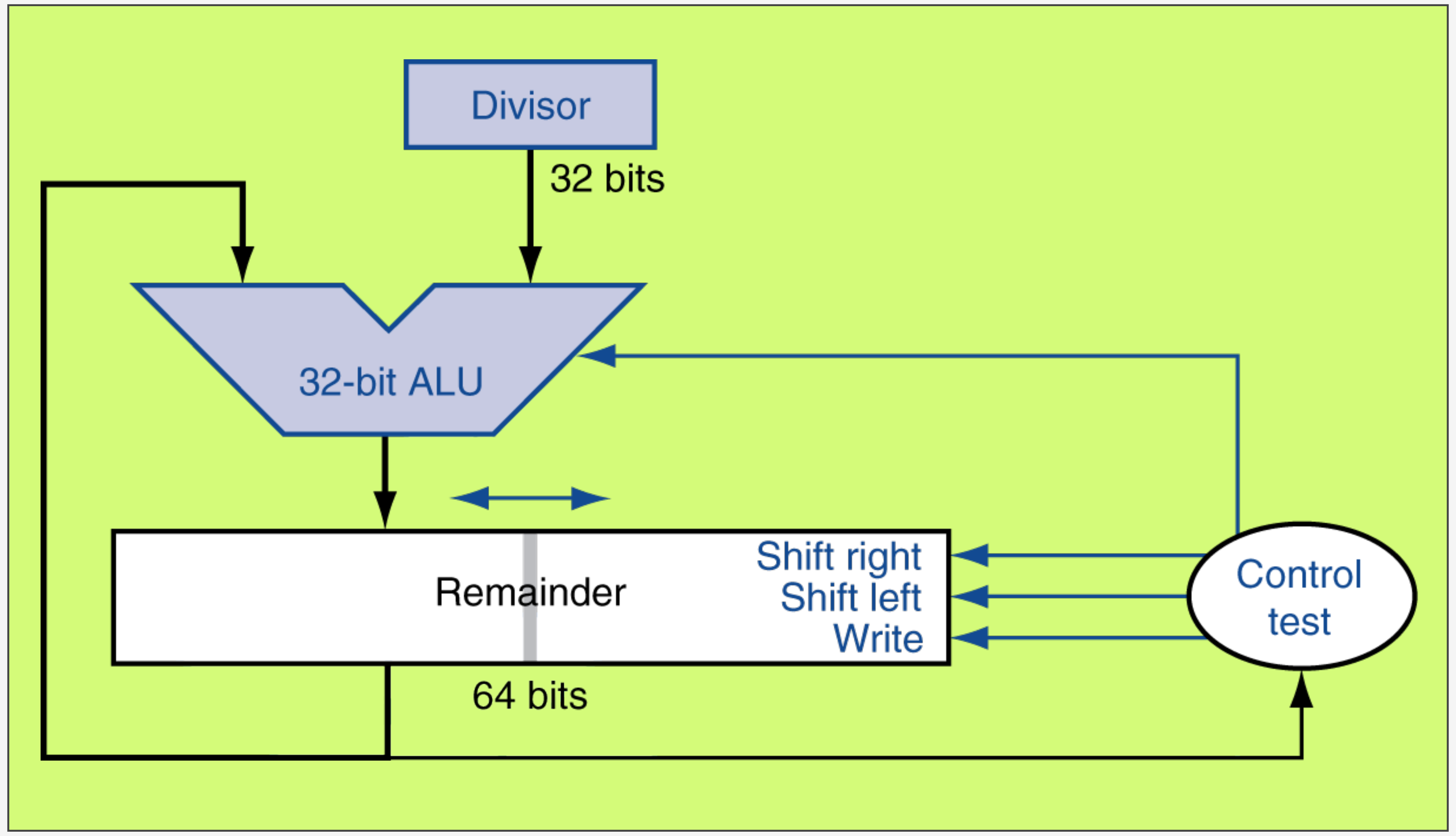
- **Division example using basic hardware and 4-bit inputs**
  - 4-bit example requires ***5 iterations*** (one more than word size)
  - Initialize Quotient register to 0 / Remainder register to dividend / and place divisor in top half of Divisor register
  - Example: $7_{ten} \div 2_{ten}$

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001



Optimized (for size) Division Hardware

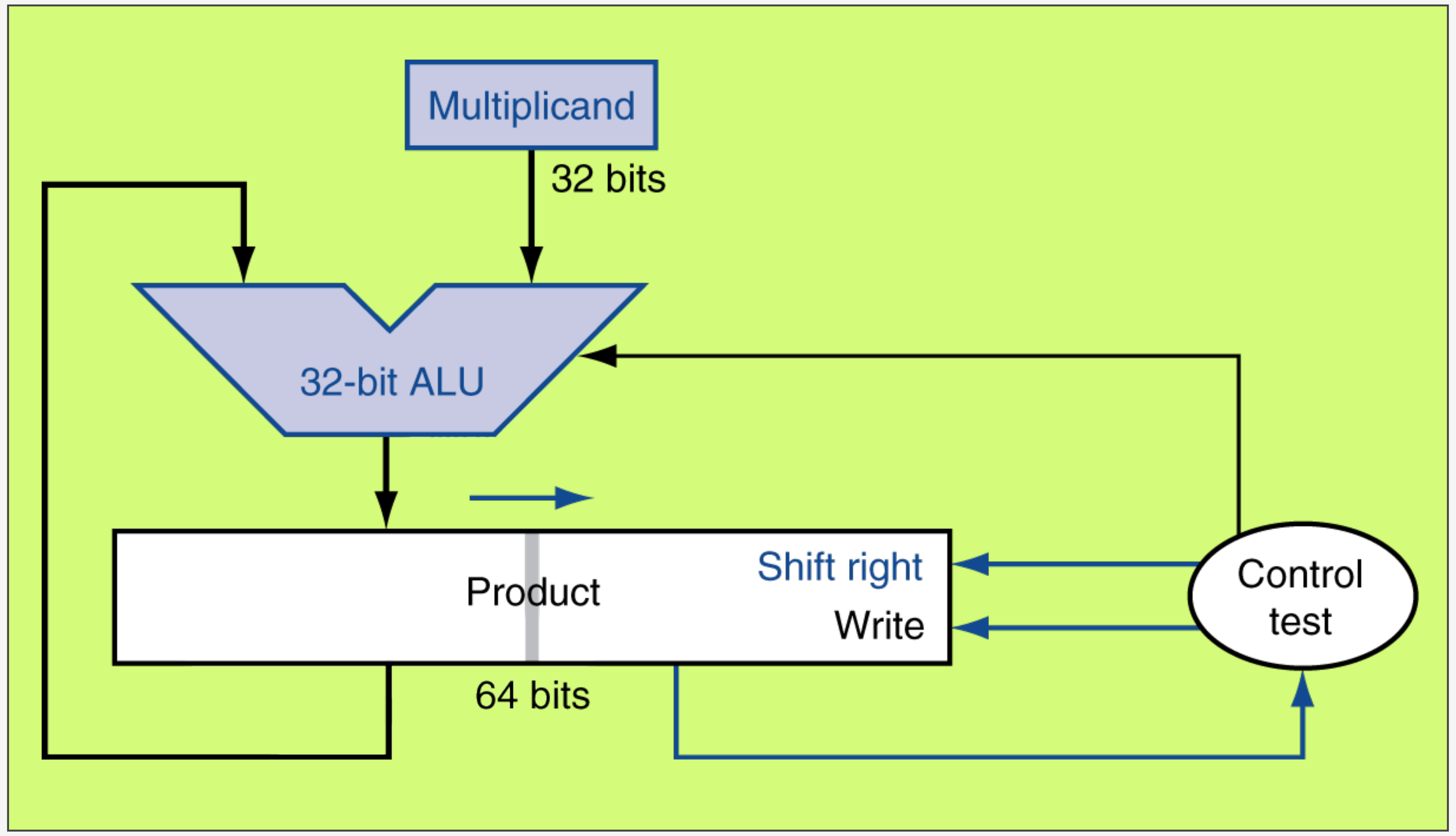
- Reduced hardware requirements
  - Divisor register and ALU now 32-bit
  - Quotient no longer has dedicated register
    - Right half of Remainder register is initialized with dividend, and left half to 0

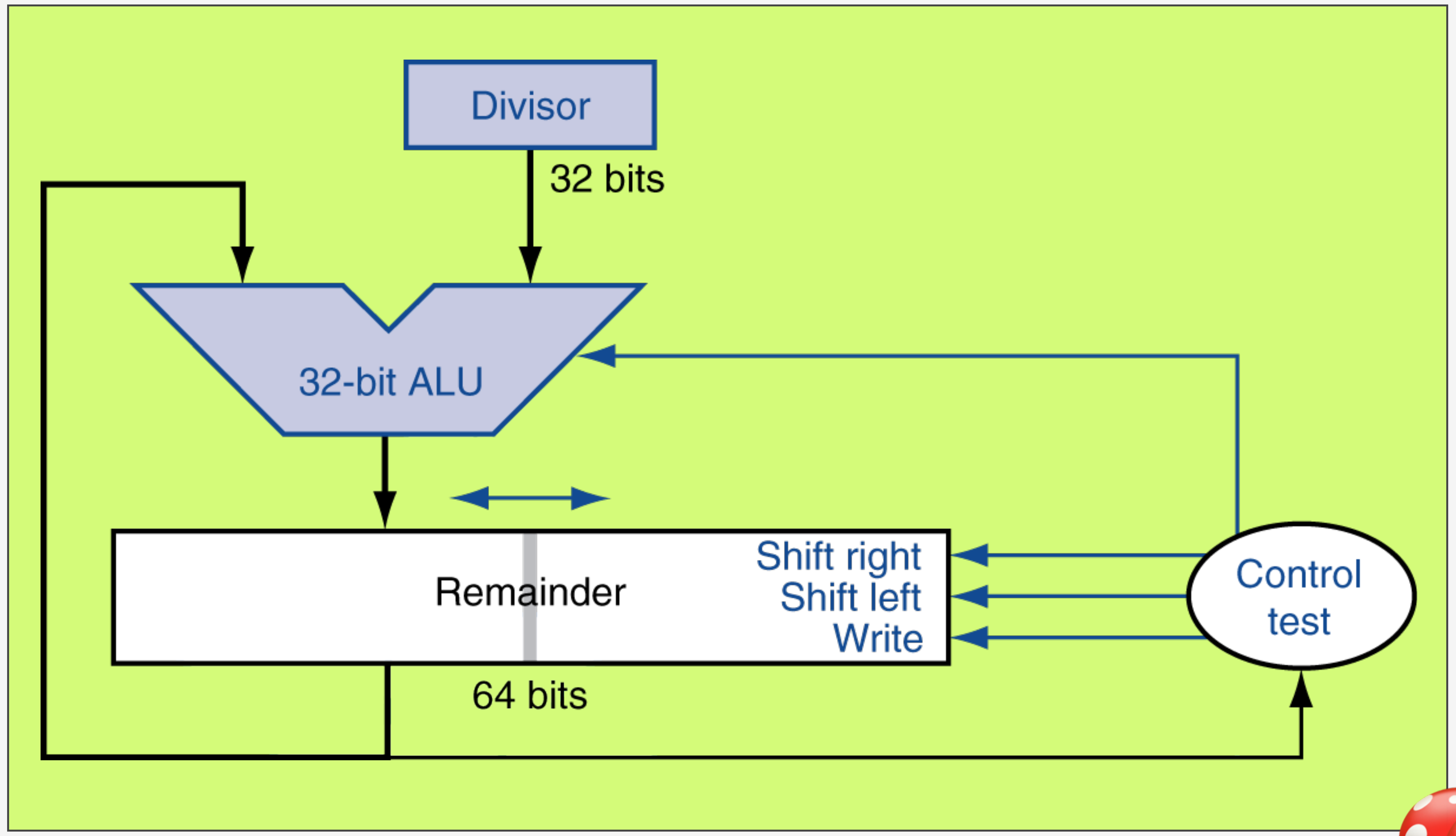


- When division operation is complete:
  - Left half of Remainder register contains the 32-bit remainder value
  - Right half of the Remainder register contains the 32-bit quotient value
- Only the “shift left” operation is used for the division operation
  - Well, then why is there a “right shift” input?!??!??!?!

Combined Hardware for Multiplication/Division

- Same hardware can be used for both multiplication and division algorithms
  - Earlier, reduced hardware requirement for each unit when they were optimized for size
  - Now, reusing same hardware for both MUL and DIV further reduces hardware requirements





A Faster Divider

- Multiplication can be parallelized
  - Additional hardware resources can be used to perform multiplication faster
  - Sacrifice size and cost for better performance
- Division cannot be parallelized like multiplication
  - Dividers are slow _ _ ___🐌
    - Only produce a single bit for the quotient on each iteration
  - Other, faster division algorithms do exist, but we won't cover them here

Signed Division

- Recall from grade school arithmetic that the Quotient is negative if the signs of the Dividend and the Divisor differ

```
positive ÷ positive = positive
negative ÷ negative = positive
positive ÷ negative = negative
```

- Thus, in hardware:
  - Perform the division algorithm
  - If the sign bits of the dividend and the divisor differ, then negate the quotient
  - Set the sign of the remainder to be the same as the dividend

Floating-Point Numbers

- Floating point numbers are numbers that have “floating” decimal points
- Used to represent non-integer numbers, including:
  - Real numbers
    - Examples: 99.2 3.14159265359
  - Very small numbers such as fractions
    - Examples: 0.00187 -0.1211
  - Very large numbers that cannot be represented using using the provided word size
    - Examples: 987.02×10^{9} -0.002×10^{-4}
- In many programming languages, declare floating point numbers as using ***float*** or ***double*** keyword
  - Two different floating-point representations: ***single-precision*** and ***double-precision***

Single-Precision Floating-Point Representation

- Represented using a single 32-bit word
  - Sign bit (s) specifies sign of the floating-point value
    - 0 indicates a positive / 1 indicates a negative
  - Includes 8-bit exponent and 23-bit fraction
  - Value of floating-point number is computed as: with a **bias of 127** for single-precision values

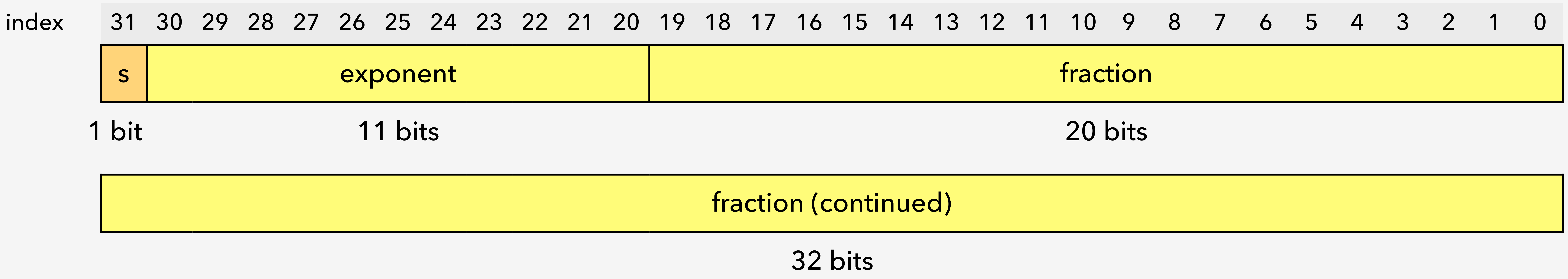
$$x = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{exponent - bias})}$$

index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	s	exponent								fraction																						
	1 bit	8 bits								23 bits																						

Double-Precision Floating-Point Representation

- Represented using a ***TWO*** 32-bit words (totaling 64-bits)
  - Sign bit (s) specifies sign of the floating-point value
    - 0 indicates a positive / 1 indicates a negative
  - Includes 11-bit exponent and 52-bit fraction
  - Value of floating-point number is computed as: with a **<u>bias of 1023</u>** for double-precision values

$$x = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{exponent - bias})}$$



Bias? 😕 What the heck is that?

- Floating-point numbers can be negative AND also have a negative exponent
  - Example: -0.002×10^{-4}
- The sign bit of a floating-point number indicates the sign of the value, not the exponent
- How about embedding a second sign bit in the exponent (bit 30) and storing as 2's complement?
  - Meh .. it would work, but it would make comparing floating-point values difficult
    - Direct comparison of binary floating-point values would not be possible since negative numbers would "appear" larger than positive numbers
- Instead, bias the exponent value by the largest positive value and adjust exponent when interpreting value of floating-point number
  - Enables DIRECT comparison of binary floating-point numbers

Single-Precision Range

- Exponents 0000_0000_{two} and 1111_1111_{two} are reserved
  - 0_{ten} with a fraction of 0_{ten} indicates the value zero
  - 255_{ten} with a fraction of 0_{ten} indicates the value ∞
  - 255_{ten} with a nonzero fraction indicates the value NaN (Not a Number)
- Smallest value in single-precision range:
  - Binary exponent value: 0000_0001_{two}
    Actual exponent after biasing: $1_{ten} - 127_{ten} = -126_{ten}$
  - Binary fraction value: $000\_0000\_0000\_0000\_0000\_0000_{two}$
    Significand: $1_{ten} + 0_{ten} = 1_{ten}$
  - Final value: $\pm \text{significand} \times 2^{\text{actual\_exponent}}$
    $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Single-Precision Range (continued)

- **Largest value in single-precision range:**
  - Binary exponent value: 1111_1110_{two}
    Actual exponent after biasing: $254_{ten} - 127_{ten} = +127_{ten}$
  - Binary fraction value: $111\_1111\_1111\_1111\_1111\_1111_{two}$
    Fraction value: $\approx 1_{ten}$
    Significand: $1_{ten} + 1_{ten} \approx 2_{ten}$
  - Final value: $\pm \text{significand} \times 2^{\text{actual\_exponent}}$
    $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents $000\_0000\_0000_{two}$ and $111\_1111\_1111_{two}$ are reserved
  - 0_{ten} with a fraction of 0_{ten} indicates the value zero
  - 2047_{ten} with a fraction of 0_{ten} indicates the value ∞
  - 2047_{ten} with a nonzero fraction indicates the value NaN (Not a Number)
- Smallest value in double-precision range:
  - Binary exponent value: $000\_0000\_0001_{two}$
    Actual exponent after biasing: $1_{ten} - 1023_{ten} = -1022_{ten}$
  - Binary fraction value: $000\_0000\_0000 \ldots \ldots 0000\_0000\_0000_{two}$ (52 bits of zeros)
    Significand: $1_{ten} + 0_{ten} = 1_{ten}$
  - Final value: $\pm$ significand $\times 2^{actual\_exponent}$
    $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

Double-Precision Range (continued)

- **Largest value in double-precision range:**
  - Binary exponent value: $111\_1111\_1110_{two}$
    Actual exponent after biasing: $2046_{ten} - 1023_{ten} = +1023_{ten}$
  - Binary fraction value: $111\_1111\_1111 \ldots \ldots 1111\_1111\_1111_{two}$ (52 bits of zeros)
    Fraction value: $\approx 1_{ten}$
    Significand: $1_{ten} + 1_{ten} \approx 2_{ten}$
  - Final value: $\pm$ significand $\times 2^{actual\_exponent}$
    $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Single-precision
  - Approximately 2^{-23}
  - Can represent a value x and $(x + 2^{-23})$, but not the numbers in between
- Double-precision
  - Approximately 2^{-52}
  - Can represent a value x and $(x + 2^{-52})$, but not the numbers in between

Floating-Point Example

- What number is represented by the single-precision floating-point value?

index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 bit	8 bits								23 bits																						

$$x = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{exponent - bias})}$$

- Sign bit: 1 (negative number)
  Exponent: $1000\_0001_{two} = 129_{ten}$
  Fraction: $010\_0000\_0000\_0000\_0000\_0000_{two}$
  $(0 \times 2^{-1}) + (1 \times 2^{-2}) = 0 + (1 \times ¼) = ¼ = .25$

$x = (-1)^s \times (1 + \text{fraction}) \times 2^{(\text{exponent - bias})}$
$= (-1)^1 \times (1 + .25) \times 2^{(129 - 127)}$
$= -1 \times 1.25 \times 2^2$
$= -5.0$