# ECE260: Fundamentals of Computer Engineering

## Arithmetic for Computers

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

YORK COLLEGE OF PENNSYLVANIA

# 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

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

  - Negate 2nd operand and add

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

# 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 |
|---|---|---|---|
| $A + B$ | $\geq 0$ | $\geq 0$ | $< 0$ |
| $A + B$ | $< 0$ | $< 0$ | $\geq 0$ |
| $A - B$ | $\geq 0$ | $< 0$ | $< 0$ |
| $A - B$ | $< 0$ | $\geq 0$ | $\geq 0$ |

- 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!

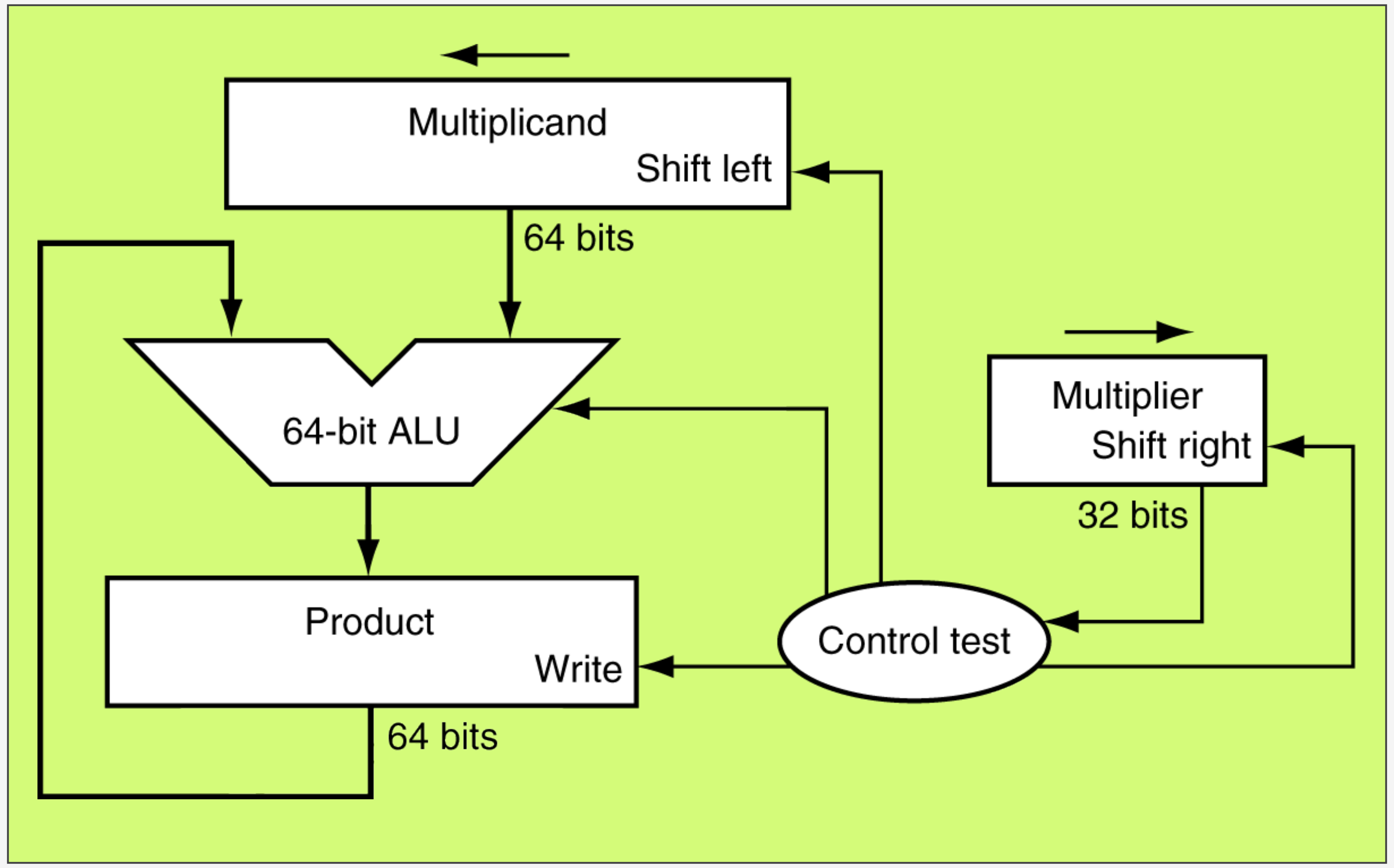
- Example of multiplying two numbers together:

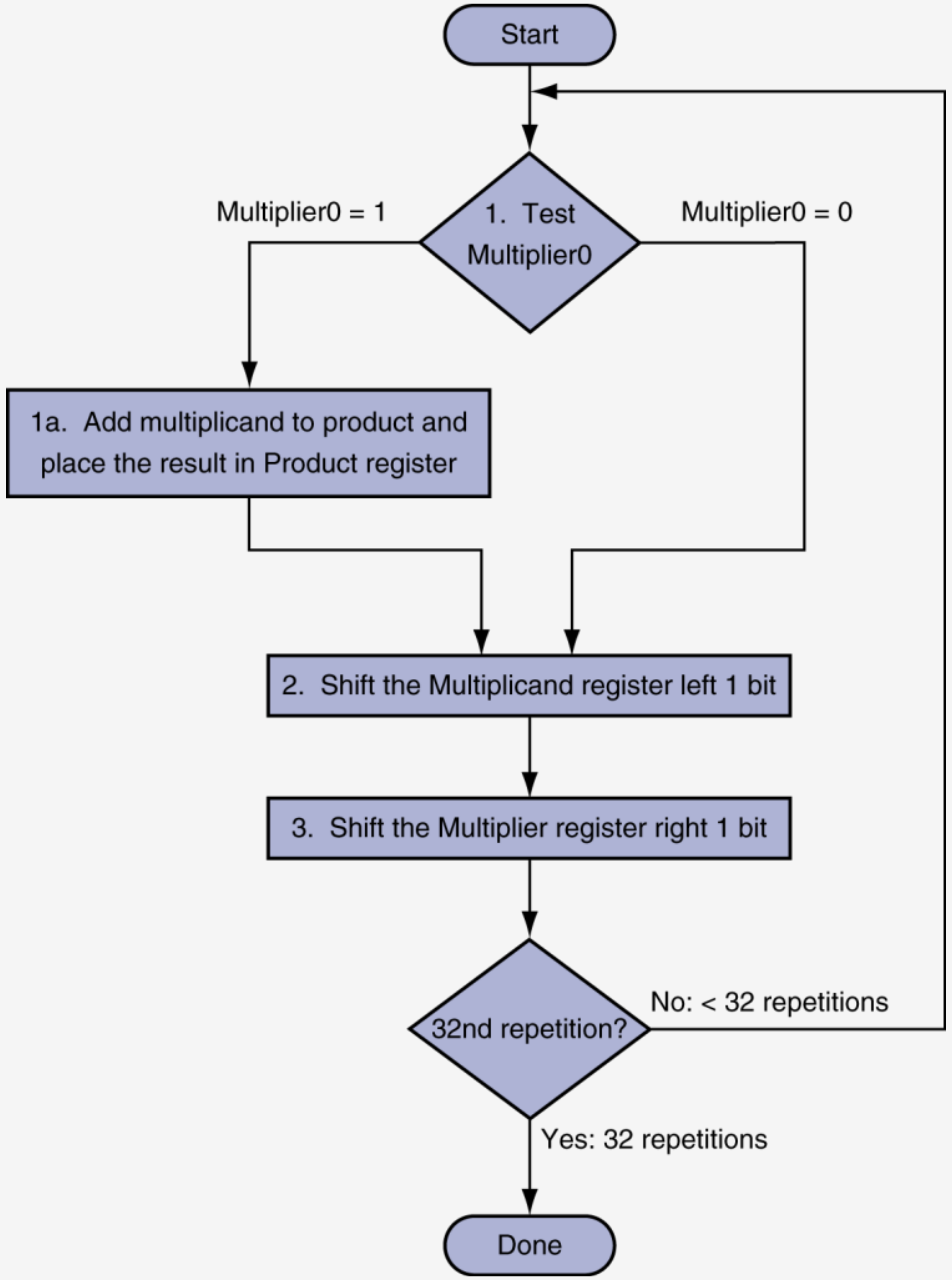
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Multiplicand | | | | | 1 | 0 | 0 | $0_{two}$ |
| Multiplier | | | | × | 1 | 0 | 0 | $1_{two}$ |
| | | | | | 1 | 0 | 0 | 0 |
| | | | | 0 | 0 | 0 | 0 | |
| | | | 0 | 0 | 0 | 0 | | |
| | + | 1 | 0 | 0 | 0 | | | |
| Product | | 1 | 0 | 0 | 1 | 0 | 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

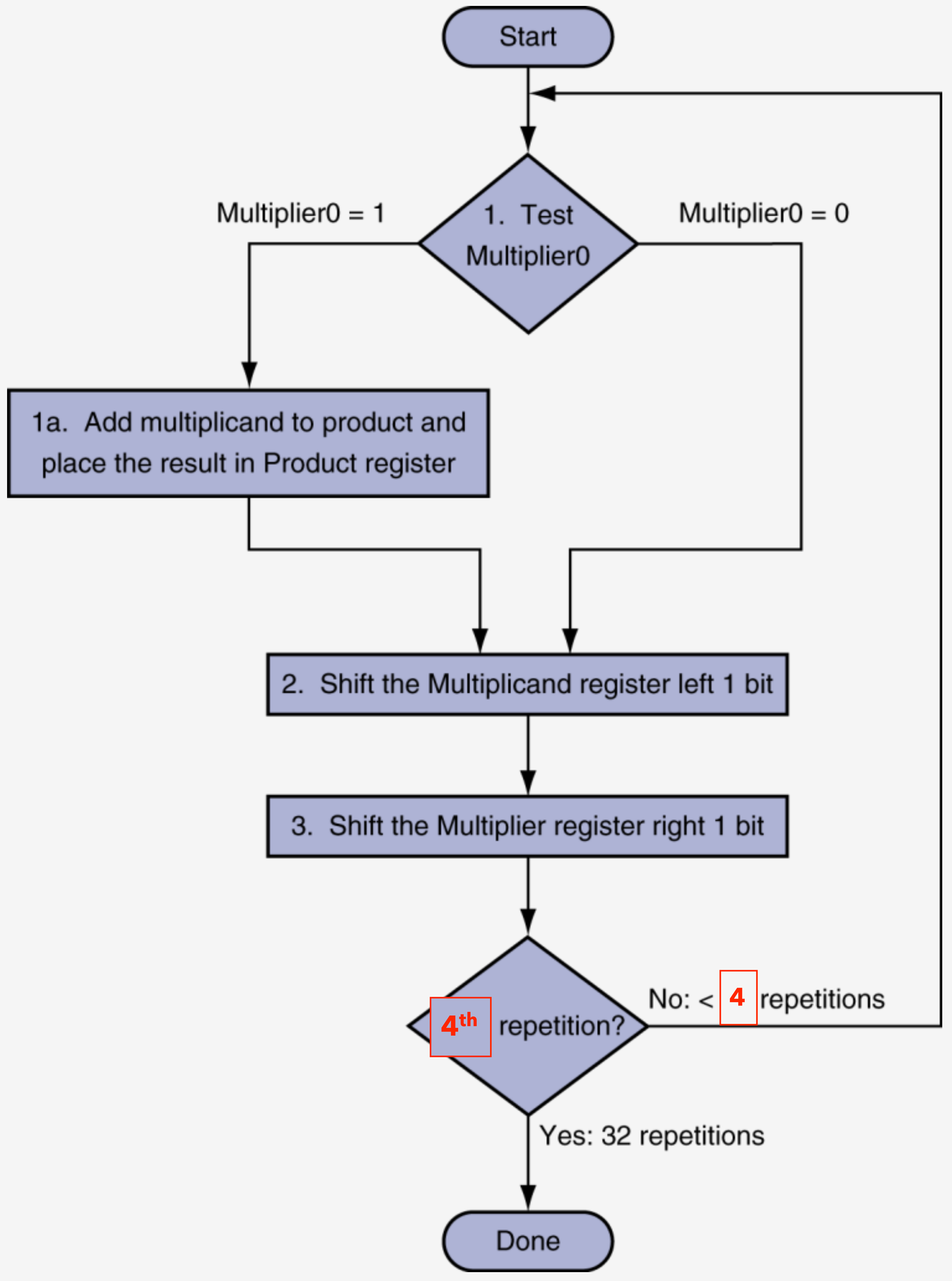


# Multiplication Example

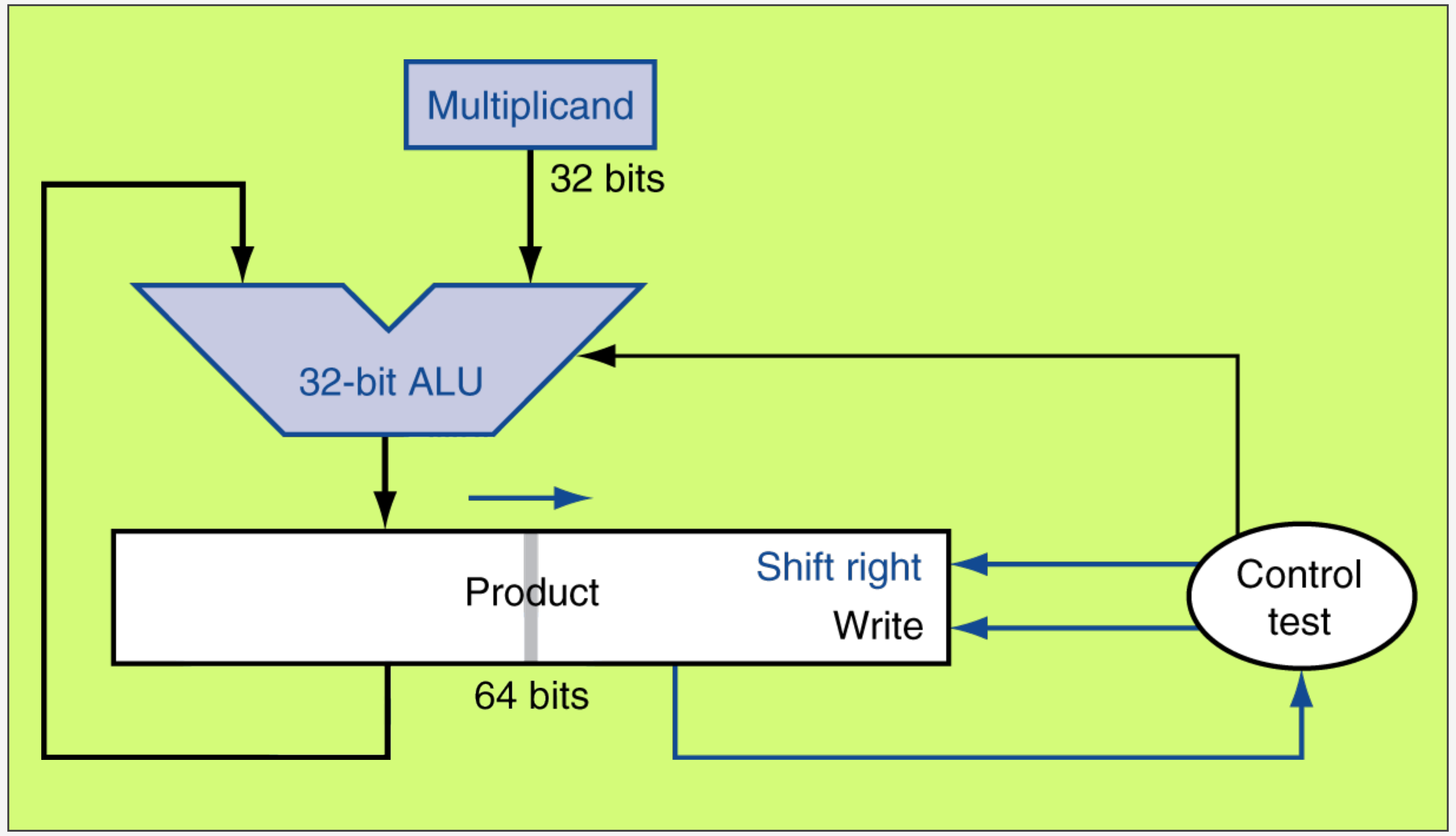
- Multiplication example using basic hardware and 4-bit inputs
  - 4-bit example requires only 4 iterations, not 32
  - Initialize Product register to 0
  - Example: $2_{ten} \times 3_{ten}$

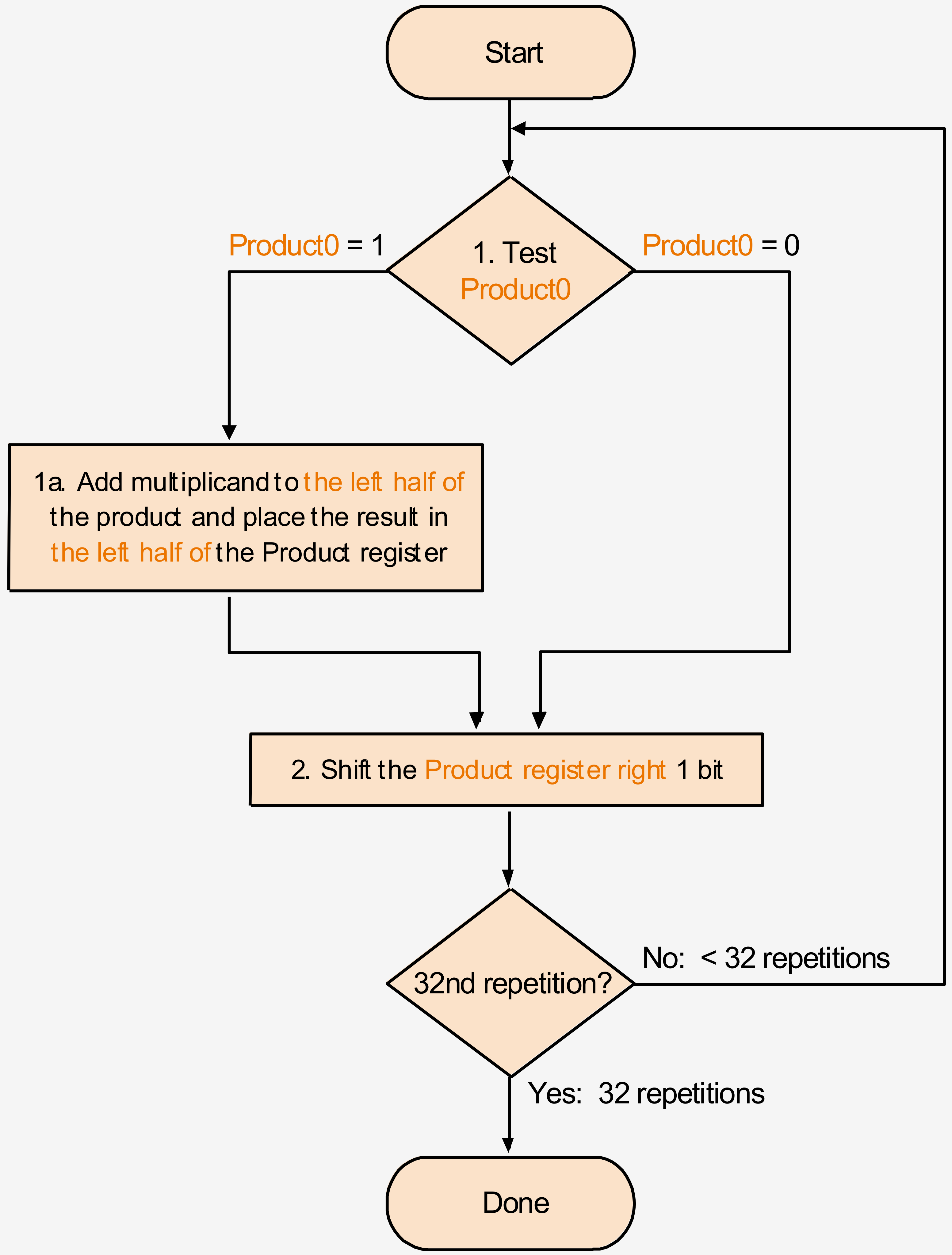
| Iteration | Step | Multiplier | Multiplicand | Product |
|---|---|---|---|---|
| 0 | Initial values | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a: 1 ⇒ Prod = Prod + Mcand | 0011 | 0000 0010 | 0000 0010 |
| | 2: Shift left Multiplicand | 0011 | 0000 0100 | 0000 0010 |
| | 3: Shift right Multiplier | 0001 | 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



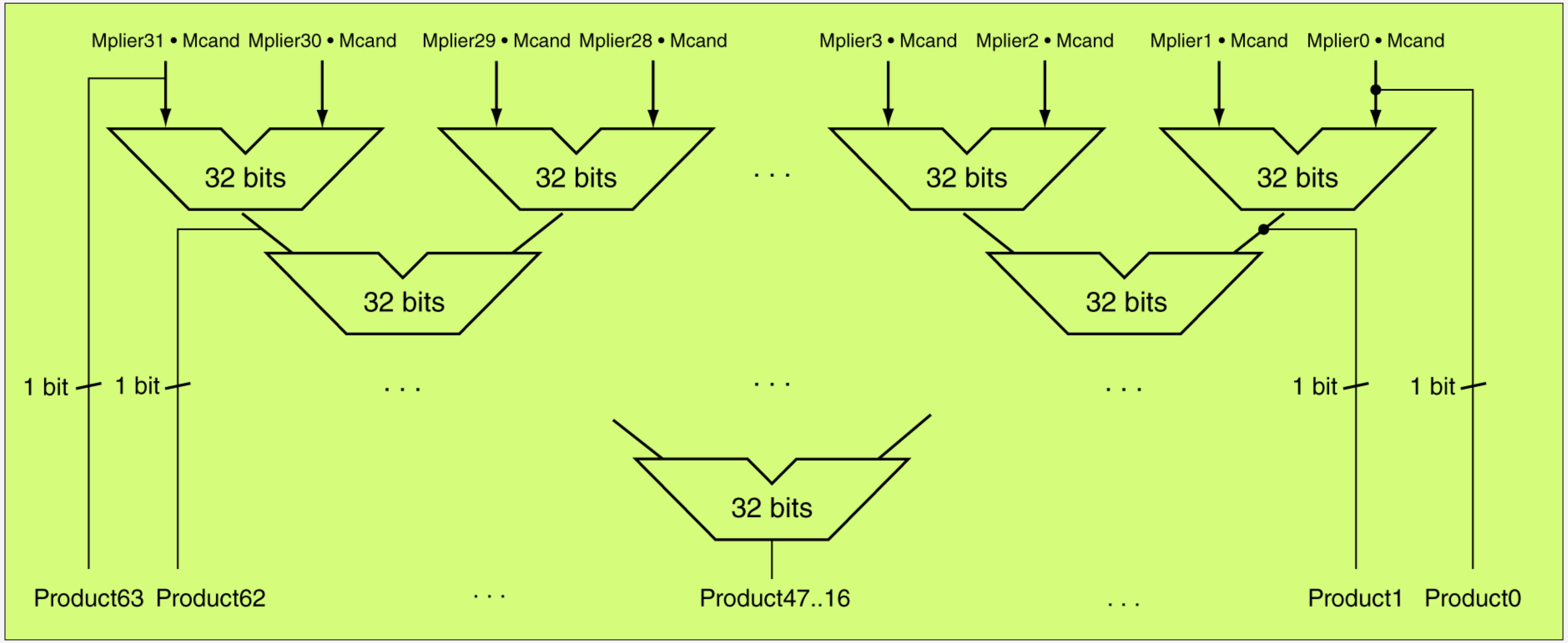



# Multiplication Example #2

- 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 signs differed, then negate the result
  - Be sure to do sign extension for right shifts