# Computer Organization

Multiplication of signed operands using Booth's Algorithm- Illustration

# Points to remember

- When using Booth's Algorithm:
  - You will need twice as many bits in your **product** as you have in your original two **operands**.
  - The **leftmost bit** of your operands (both your multiplicand and multiplier) is a SIGN bit, and cannot be used as part of the value.

# To begin

- Decide which operand will be the **multiplier** and which will be the **multiplicand**
- Convert operands to **two's complement** representation using X bits
  - X must be at least one more bit than is required for the binary representation of the numerically larger operand
- Begin with a product that consists of the multiplier with an additional X leading zero bits

# Example

- consider an example of multiplying **2 x (-5)**
  - The numerically larger operand (5) would require 3 bits to represent in binary (101). So we must use AT LEAST 4 bits to represent the operands, 1 bit to allow for the sign bit.
- Let's use 5-bit 2's complement:
  - -5 is 11011 (multiplier)
  - 2 is 00010 (multiplicand)
    - 1’s complement of 5 =11010
    - 2’s complement of 5= 11010 +1 = 11011

# Beginning Product

- The multiplier is:

  **11011**

- Add 5 leading zeros to the **multiplier** to get the **beginning product:**

  **00000 11011**

# Step 1 for each pass

- Use the **LSB** (least significant bit) and the **previous LSB** to determine the arithmetic action.
  - If it is the FIRST pass, use **0** as the previous LSB.
- Possible arithmetic actions:
  - **00** → no arithmetic operation
  - **01** → add multiplicand to left half of product
  - **10** → subtract multiplicand from left half of product
  - **11** → no arithmetic operation

# Step 2 for each pass

- Perform an **arithmetic right shift** (ASR) on the entire product.

- NOTE: For X-bit operands, Booth's algorithm requires X passes.

# Example

- Let's continue with our example of multiplying **(-5) x 2**
- Remember:
  - -5 is 11011 (multiplier)
  - 2 is 00010 (multiplicand)

- And we added 5 leading zeros to the **multiplier** to get the **beginning product:**

**00000 11011**

# Example continued

- Initial Product and previous LSB

**00000 11011 0**

(Note: Since this is the first pass, we use 0 for the previous LSB)

- Pass 1, Step 1: Examine the last 2 bits

**00000 11011 0**

The last two bits are 10, so we need to:

subtract the **multiplicand** from left half of product

# Example: Pass 1 continued

- Pass 1, Step 1: Arithmetic action

(1) **00000** (left half of product)

**-** (mulitplicand)

**00010**

**1111** (uses 2's complement)

**0**

- Place result into **left half** of product

**11110 11011 0**

# Example: Pass 1 continued

- Pass 1, Step 2: ASR (arithmetic shift right)
  - Before ASR

    **11110 110111 0**

  - After ASR

    **11111 01101 1**

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 1 is complete.

# Example: Pass 2

- Current Product and previous LSB

  **11111 01101 1**

- Pass 2, Step 1: Examine the last 2 bits

  **11111 01101 1**

  The last two bits are 11, so we do NOT need to perform an arithmetic action --

  just proceed to step 2.

# Example: Pass 2 continued

- Pass 2, Step 2: ASR (arithmetic shift right)
  - Before ASR

    **11111 01101 1**

  - After ASR

    **11111 10110 1**

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 2 is complete.

# Example: Pass 3

- Current Product and previous LSB

  **11111 10110 1**

- Pass 3, Step 1: Examine the last 2 bits

  **11111 10110 1**

  The last two bits are 01, so we need to:

  add the **multiplicand** to the left half of the product

# Example: Pass 3 continued

- Pass 3, Step 1: Arithmetic action

~~(1)~~ **11111** (left half of product)

**+00010** (mulitplicand)

**00001** (drop the leftmost carry)

- Place result into **left half** of product

**00001 10110 1**

# Example: Pass 3 continued

- Pass 3, Step 2: ASR (arithmetic shift right)
  - Before ASR

    **00001 10110 1**

  - After ASR

    **00000 11011 0**

  (left-most bit was 0, so a 0 was shifted in on the left)

- Pass 3 is complete.

# Example: Pass 4

- Current Product and previous LSB

  **00000 11011 0**

- Pass 4, Step 1: Examine the last 2 bits

  **00000 11011 0**

The last two bits are 10, so we need to:

subtract the **multiplicand** from the left half of the product

# Example: Pass 4 continued

- Pass 4, Step 1: Arithmetic action

(1) **00000** (left half of product)

**- 00010** (mulitplicand)

**11110** (use two's complement arithmetic)

- Place result into **left half** of product

**11110 11011 0**

# Example: Pass 4 continued

- Pass 4, Step 2: ASR (arithmetic shift right)
  - Before ASR

    **11110 11011 0**

  - After ASR

    **11111 01101 1**

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 4 is complete.

# Example: Pass 5

- Current Product and previous LSB

  **11111 01101 1**

- Pass 5, Step 1: Examine the last 2 bits

  **11111 01101 1**

  The last two bits are 11, so we do NOT need to perform an arithmetic action --

  just proceed to step 2.

# Example: Pass 5 continued

- Pass 5, Step 2: ASR (arithmetic shift right)
  - Before ASR

    **11111 01101 1**

  - After ASR

    **11111 10110 1**

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 5 is complete.

# Final Product

- We have completed 5 passes on the 5-bit operands, so we are done.
- Dropping the previous LSB, the resulting **final product** is:

**11111 10110**

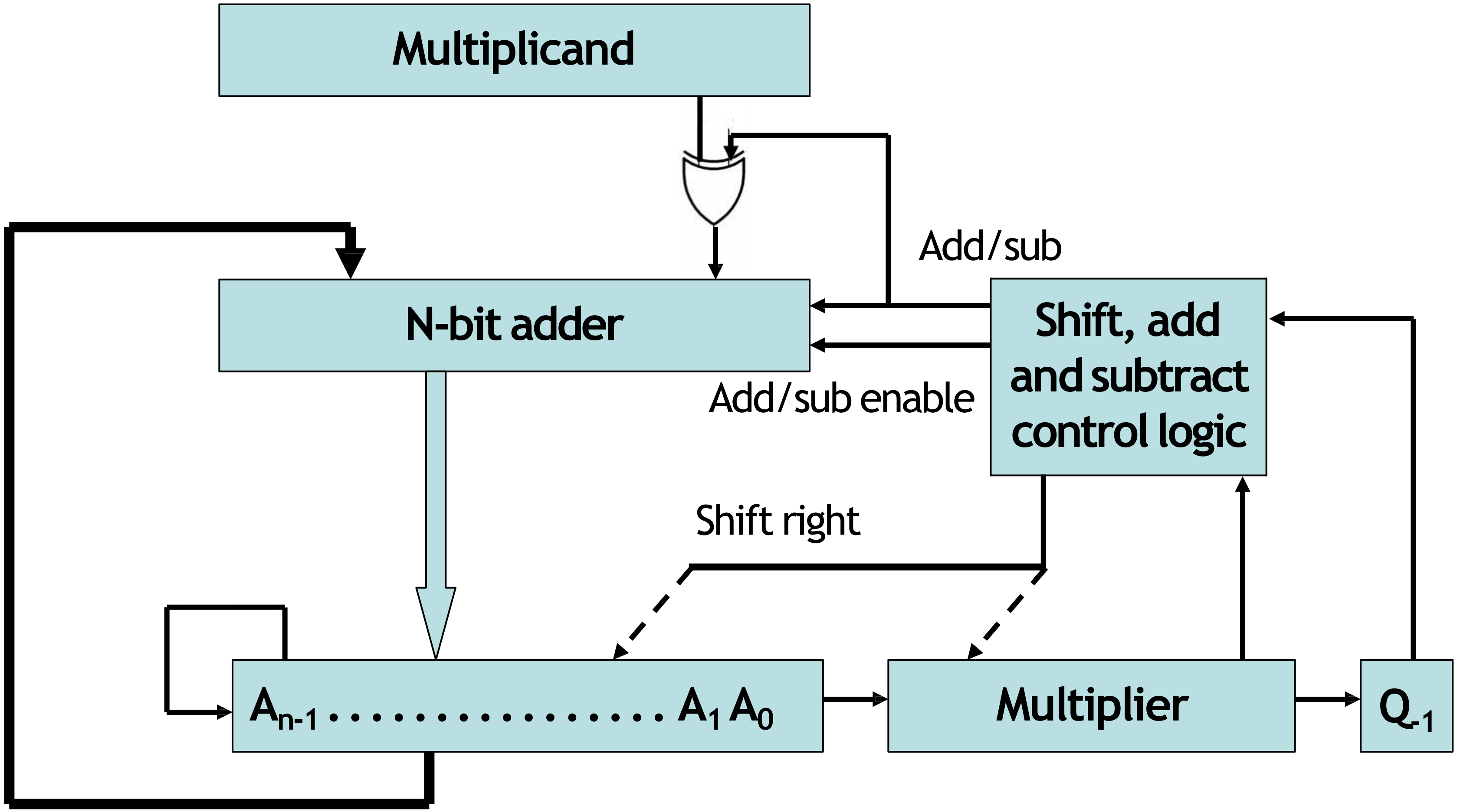
# Verification

- To confirm we have the correct answer, convert the 2's complement **final product** back to decimal.
- Final product: **11111 10110**
- Decimal value: **-10**

which is the CORRECT product of:

**(-5) x 2**

# Hardware Implementation



# Multiplication

**Booth's Algorithm:**

Multiply 14 times -5 using 5-bit numbers (10-bit result).

14 in binary: 01110

-14 in binary: 10010 (so we can add when we need to subtract the multiplicand)

-5 in binary: 11011

Expected result: -70 in binary: 11101 11010

# Multiplication

| Step | Multiplicand | Action | Multiplier<br><br>upper 5-bits 0,<br>lower 5-bits multiplier,<br>1 “Booth bit” initially 0 |
|---|---|---|---|
| 0 | 01110 | Initialization | 00000 11011 0 |
| 1 | 01110 | 10: Subtract Multiplicand | 00000+10010=10010<br><br>10010 11011 0 |
| | | Shift Right Arithmetic | 11001 01101 1 |
| 2 | 01110 | 11: No-op | 11001 01101 1 |
| | | Shift Right Arithmetic | 11100 10110 1 |
| 3 | 01110 | 01: Add Multiplicand | 11100+01110=01010<br>(Carry ignored because adding a positive and negative number cannot overflow.)<br><br>01010 10110 1 |
| | | Shift Right Arithmetic | 00101 01011 0 |
| 4 | 01110 | 10: Subtract Multiplicand | 00101+10010=10111<br><br>10111 01011 0 |
| | | Shift Right Arithmetic | 11011 10101 1 |
| 5 | 01110 | 11: No-op | 11011 10101 1 |
| | | Shift Right Arithmetic | 11101 11010 1 |

# Modified Booth Multiplier

# Booth Recoding: Advantages and Disadvantages

Depends on the architecture

Potential advantage: might reduce the # of 1's in multiplier

In the multipliers that we have seen so far:

Doesn't save speed (still have to wait for the critical path, e.g., the shift-add delay in sequential multiplier)

Disadvantage: Increases area , recoding circuitry AND subtraction

tusharkute.com

# Modified Booth Multiplier: Idea (cont.)

- Can encode the digits by looking at three bits at a time
- Booth recoding table:

| i+1 | i | i-1 | add |
|---|---|---|---|
| 0 | 0 | 0 | 0*M |
| 0 | 0 | 1 | 1*M |
| 0 | 1 | 0 | 1*M |
| 0 | 1 | 1 | 2*M |
| 1 | 0 | 0 | –2*M |
| 1 | 0 | 1 | –1*M |
| 1 | 1 | 0 | –1*M |
| 1 | 1 | 1 | 0*M |

  – Must be able to add *multiplicand* times –2, -1, 0, 1 and 2
  – Since Booth recoding got rid of 3's, generating partial products is not that hard (shifting and negating)

$-9 \times -13 = 117$

M = -9 => 110111      9 -> 001001, -9 -> 110111

Q = -13 => 110011      13 -> 001101, -13 -> 110011

Initial setting:

Accumulator Register A = 000000

Register Q (Multiplier) = 110011 -> (-13)

Register M (Multiplicand) = 110111 -> (-9)

| | A | Q | $Q_{-1}$ |
|---|---|---|---|
| I | 000000 | 110011 | 0 |
| A = A-M | 001001 | | |
| | 001001 | 110011 | 0 |
| ASR | 000100 | 111001 | 1 |
| ASR | 000010 | 011100 | 1 |
| II | 110111 | | |
| A = A+M | 111001 | 011100 | 1 |
| ASR | 111100 | 101110 | 0 |
| ASR | 111110 | 010111 | 0 |
| III | 001001 | | |
| A = A-M | 000111 | 010111 | 0 |
| ASR | 000011 | 101011 | 1 |
| ASR | $\boxed{000001}$ | $\boxed{110101}$ | 1 |

Drop $Q_{-1}$ (previous LSB)

Product = $\boxed{000001 \quad 110101}$ = 117

# Reference

- Computer Organization, Designing for Performance by *William Stallings*