Booth's Algorithm Example

CS440

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 both 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

- In the week by week, there is an example of multiplying **2 x (-5)**
- For our example, let's reverse the operation, and multiply (**-5) x 2**
  - 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, to allow for the sign bit.
- Let's use 5-bit 2's complement:
  - -5 is 11011 (multiplier)
  - 2 is 00010 (multiplicand)

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)
    -00010     (mulitplicand)
    ------
     11110     (uses a phantom borrow)
```

- 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 11011 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    (uses a phantom borrow)
```

- 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`