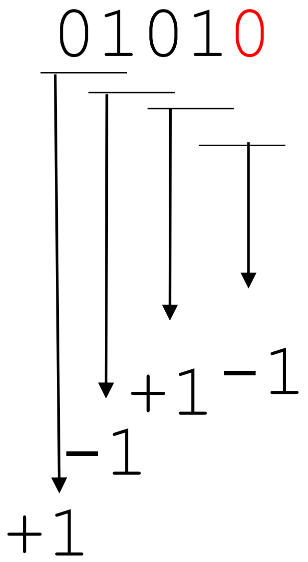
Booth's Algorithm

- Treats positive and negative multipliers uniformly.
- Rewrites multiplier in terms of sums and differences.
- Convert code according to next bit at right
- 0 to 1 $\Rightarrow$ +1
- 1 to 0 $\Rightarrow$ –1
- Otherwise, 0
- Right of lsb is "nothing", *i.e.*, equal to 0

Booth’s Algorithm



Steps in obtaining Booth’s equivalent of a binary number

1. append 0 at the LSb side
2. pair 2 bits starting at LSb
3. 00 → 0; 01→ +1; 10→-1;11→0

0*m = 0

1*m = m

-1*M = 2’s complement(m)

Booth's Algorithm

```
0100
0101
```

step 1

```
01010
+1-1+1-1
```

step 2

```
       0100
 x  +1-1+1-1
 -----------
   11111100
   0000100
   111100
   00100
 -----------
   00010100
```

step 3

Example: +4 * +5

Steps:

1.) represent both multiplicand (m) and multiplier (n) using 2's complement format

2.) convert multiplier to its Booth's equivalent

- append 0 at the LSb side
- pair 2 bits starting at LSb
- 00 → 0; 01→ +1; 10→-1;11→0

3.) Multiply using pencil-and-paper method ignoring extra steps if multiplier is negative

Booth's Algorithm

```
0100 ┐
1011 ┘ step 1

10110 ┐
-1+1 0-1 ┘ step 2
```

```
      0100
x  -1+10-1
-----------
   11111100
   0000000    step 3
   000100
   11100
-----------
   11101100
```

Example: +4 * -5

Steps:

1.) represent both multiplicand (m) and multiplier (n) using 2's complement format

2.) convert multiplier to its Booth's equivalent

- append 0 at the LSb side
- pair 2 bits starting at LSb
- 00 → 0; 01→ +1; 10→-1;11→0

3.) Multiply using pencil-and-paper method ignoring extra steps if multiplier is negative

Try

Try: +13*-6 (using Booth's algorithm)

```
   01101
x  11010
--------
```

```
 01101 }
 11010 }  step 1

110100 }
0-1+1-10 } step 2
```

```
        01101
x  0-1+1-10
-----------
   0000000000
   111110011
   00001101      step 3
   1110011
   000000
-----------
   1110110010
```

Extended Booth's Algorithm

- Also known as fast multiplication or bit-pair recording
- Bit-Pair Recording – reduces to half the number of summands The number of summands is reduced by pairing multiplier bits

Bit-pair recording:

0 0 0 => 0

0 0 1 => +1

0 1 0 => +1

0 1 1 => +2

1 0 0 => -2

1 0 1 => -1

1 1 0 => -1

1 1 1 => 0

Extended Booth's Algorithm

01010

+1

+1

Steps in obtaining Booth's equivalent of a binary number

1. append 0 at the LSb side
2. if odd number of bits, sign-extend
3. bit-pair starting at LSb

0*m = 0

1*m = m

-1*m = 2's complement(m)

+2*m = m0

-2*m = [2's complement(m)]0

Bit-pair recording:

0 0 0 => 0

0 0 1 => +1

0 1 0 => +1

0 1 1 => +2

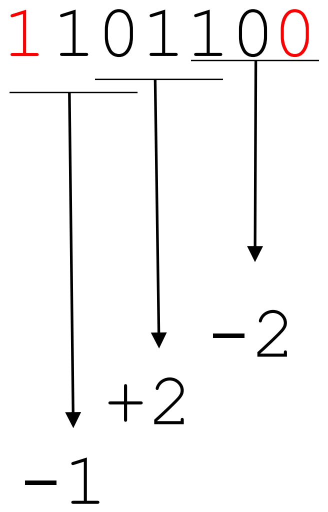
1 0 0 => -2

1 0 1 => -1

1 1 0 => -1

1 1 1 => 0

Extended Booth's Algorithm



Steps in obtaining Extended Booth's equivalent of a binary number

1. append 0 at the LSb side
2. if odd number of bits, sign-extend
3. bit-pair starting at LSb

0*m = 0
1*m = m
-1*m = 2's complement(m)
+2*m = m0
-2*m = [2's complement(m)]0

Bit-pair recording:
0 0 0 => 0
0 0 1 => +1
0 1 0 => +1
0 1 1 => +2
1 0 0 => -2
1 0 1 => -1
1 1 0 => -1
1 1 1 => 0

Extended Booth's Algorithm

```
0100
0101    step 1

01010
+1+1    step 2
```

```
        0100
x       +1+1
-----------
   00000100
   000100        step 3
   --------
   00010100
```

Example: +4 * +5

Steps:

1.) represent both multiplicand (m) and multiplier (n) using 2's complement format

2.) convert multiplier to Extended Booth's equivalent

3.) Multiply using pencil-and-paper method ignoring extra steps if multiplier is negative. Since each bit-pair is equivalent to 2 bits, skip two after the initial intermediate product

Extended Booth's Algorithm

```
 0100
 1011     step 1

10110
-1-1      step 2

        0100
x      -1-1
   -----------
     11111100
     111100       step 3
     ---------
     11101100
```

Example: +4 * -5

Steps:

1.) represent both multiplicand (m) and multiplier (n) using 2's complement format

2.) convert multiplier to Extended Booth's equivalent

3.) Multiply using pencil-and-paper method ignoring extra steps if multiplier is negative. Since each bit-pair is equivalent to 2 bits, skip two after the initial intermediate product

Try

Try: +13*-6 (using Extended Booth's algorithm)

```
  01101
x 11010
-------
```

```
 01101   } step 1
 11010
1110100  } step 2
0-1-2
```

```
      01101
x     0-1-2
-----------
1111100110
11110011        step 3
000000
-----------
1110110010
```