Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

\[01010010 \text{ (multiplicand)} \times 01101101 \text{ (multiplier)}\]

- Negative numbers: convert and multiply
- Use other better techniques like Booth’s encoding

Multiplication Example

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Original algorithm</th>
<th>Multiplier0</th>
<th>Multiplier1</th>
<th>Product0</th>
<th>Product1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>10010010</td>
<td>01010010</td>
<td>01101101</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>2</td>
<td>10010010</td>
<td>01010010</td>
<td>01101101</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>4</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>
Signed Multiplication

- Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]
- Let F = 0 (shift flag)
- Let result A[n-1:0] = 0…0
- For n-1 steps do
  - F<= F .or. (M[n-1] .and. Q[0]) /* determine shift bit */
  - Shift A and Q with F, i.e.,
- Do the correction step
  - Shift A and Q while retaining A[n-1]
- This works in all cases excepts when both operands are 10..00

Signed Multiplication

- Numbers can be represented using three symbols, 1, 0, and -1
  - One representation is 1 1 1 1 1 1 1 1
  - Another possible one 0 0 0 0 0 0 0 1
  - Another example +14
    - One representation is 0 0 0 0 1 1 1 0
    - Another possible one 0 0 0 1 0 0 1 0
- We do not explicitly store the sequence
  - Look for transition from previous bit to next bit
   - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
- Multiplication by 1, 0, and -1 can be easily done
  - Add all partial results to get the final answer

Booth's Encoding

- Convert a binary string in Booth's encoded string
- Multiply by two bits at a time
  - For n bit by n-bit multiplication, n/2 partial product
  - Partial products are signed and obtained by multiplying the multiplicand by 0, +1, -1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
  - Example, multiply 0111 (+7) by 1010 (-6)
    - Booths encoding of 1010 is -1 +1 -1 0
    - With 2-bit groupings, multiplication needs to be carried by -1 and -2
      - 1 1 1 1 0 0 1 0 (multiplication by -3)
      - 1 1 1 0 1 1 0 0 (multiplication by -1 and shift by 2 positions)
- Add the two partial products to get 11010110 (-42) as result

Booth's Encoding

- n-bit carry-save adder take 1FA time for any n
- For n x n bit multiplication, n or n/2 (for 2 bit at time Booth's encoding) partial products can be generated
- For n partial products n/2 n-bit carry save adders can be used
- This yields 2n/3 partial results
- Repeat this operation until only two partial results are remaining
  - Add them using an appropriate size adder to obtain 2n bit result
  - For n=32, you need 30 carry save adders in eight stages taking 8T time where T is time for one-bit full adder
  - Then you need one carry-propagate or carry-look-ahead adder
Division

- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

```
0011  0010 0010
(Dividend)
```

- Negative numbers: Even more difficult
- There are better techniques, we won’t look at them

Division, First Version

Division, Second Version

Division, Final Version

Restoring Division

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Divide algorithm</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0000 0111</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1. Rem = Rem - Div</td>
<td>1110 1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2. Rem = Rem - Div</td>
<td>1111 1110</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>3a. Rem ≥ 0 ⇒ all R, R0 = 0</td>
<td>1111 1000</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2. Rem = Rem - Div</td>
<td>0001 0001</td>
</tr>
<tr>
<td>5</td>
<td>0010</td>
<td>3a. Rem ≥ 0 ⇒ all R, R0 = 1</td>
<td>1011 0011</td>
</tr>
<tr>
<td>Done</td>
<td>0010</td>
<td>Shift left half of Rem right 1</td>
<td>0011 0011</td>
</tr>
</tbody>
</table>

Non-Restoring Division

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Divide algorithm</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1. Rem = Rem - Div</td>
<td>1110 1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2. Rem &lt; 0 ⇒ all R, R0 = 0</td>
<td>1101 1100</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>3b. Rem = Rem + Div</td>
<td>1111 1100</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2a. Rem &gt; 0 ⇒ all R, R0 = 1</td>
<td>1111 1000</td>
</tr>
<tr>
<td>5</td>
<td>0010</td>
<td>3a. Rem = Rem - Div</td>
<td>0001 1000</td>
</tr>
<tr>
<td>6</td>
<td>0010</td>
<td>2a. Rem &gt; 0 ⇒ all R, R0 = 1</td>
<td>0010 0001</td>
</tr>
<tr>
<td>Done</td>
<td>0010</td>
<td>Shift left half of Rem right 1</td>
<td>0001 0011</td>
</tr>
</tbody>
</table>