Binary Multipliers

The key trick of multiplication is memorizing a digit-to-digit table...
Everything else was just adding

<table>
<thead>
<tr>
<th>×</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
</tr>
</tbody>
</table>

You’ve got to be kidding… It can’t be that easy

Reading: Study Chapter 3.
Have We Forgotten Something?

Our ALU can add, subtract, shift, and perform Boolean functions. But, even rabbits know how to multiply…

But, it is a huge step in terms of logic… Including a multiplier unit in an ALU doubles the number of gates used.

A good (compact and high performance) multiplier can also be tricky to design. Here we will give an overview of some of the tricks used.
Binary Multiplication

Binary multiplication is implemented using the same basic longhand algorithm that you learned in grade school.

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
x & B_3 & B_2 & B_1 & B_0 \\
\end{array}
\]

\[
A_jB_i \text{ is a “partial product” } \rightarrow \begin{array}{cccc}
A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
\end{array} 
\]

\[
+ \begin{array}{cccc}
A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
\end{array}
\]

Multiplying N-digit number by M-digit number gives (N+M)-digit result

Easy part: forming partial products (just an AND gate since \(B_1\) is either 0 or 1)

Hard part: adding M, N-bit partial products
Sequential Multiplier

Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit M times:

Init: \( P \leftarrow 0 \), load A&B

Repeat M times {
  \( P \leftarrow P + (B_{\text{LSB}} == 1 ? A : 0) \)
  shift \( P/B \) right one bit
}

Done: \((N+M)\)-bit result in \(P/B\)
Simple Combinational Multiplier

\[ t_{PD} = 10 \times t_{PD,FA} \]

Not 16

\[ t_{PD} = (2 \times (N-1) + N) \times t_{PD,FA} \]

Components

\[ N \times HA \]

\[ N(N-1) \times FA \]

The Logic of a Half-Adder

\[ A \quad B \]

\[ CO \quad S \]

NB: this circuit only works for nonnegative operands
Carry-Save Combinational Multiplier

Observation: Rather than propagating the sums across each row, the carries can instead be forwarded onto the next column of the following row.

\[ t_{PD} = 8 \times t_{PD,FA} \]

\[ t_{PD} = (N+N) \times t_{PD,FA} \]

Components

- \( N \times FA \)
- \( N^2 \times FA \)

These Adders can be removed, and the AND gate outputs tied directly to the Carry inputs of the next stage.

This small improvement in performance hardly seems worth the effort, however, this design is easier to pipeline.
Higher-Radix Multiplication

Idea: If we could use, say, 2 bits of the multiplier in generating each partial product we would **halve the number of columns and halve the latency of the multiplier**!

Booth’s insight: rewrite 2*A and 3*A cases, leave 4A for next partial product to do!

\[
A_{N-1} A_{N-2} \ldots A_4 A_3 A_2 A_1 A_0 \times B_{M-1} B_{M-2} \ldots B_3 B_2 B_1 B_0
\]
Booth Recoding

Each bit can be considered to have the following weights:

- \( W(B_{2K+1}) = -2 \)
- \( W(B_{2K}) = 1 \)
- \( W(B_{2K-1}) = 1 \)

A “1” in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!
Booth Recoding

Logic surrounding each basic adder:

- Control lines (x2, Sub, Zero) are shared across each row
- Must handle the “+1” when Sub is 1 (extra half adders in a carry save array)

NOTE:
- Booth recoding can be used to implement signed multiplications
Bigger Multipliers

- Using the approaches described we can construct multipliers of arbitrary sizes, by considering every adder at the “bit” level
- We can also, build bigger multipliers using smaller ones

- Considering this problem at a higher-level leads to more “non-obvious” optimizations
Can We Multiply With Less?

- How many operations are needed to multiply 2, 2-digit numbers?
  - 4 multipliers
  - 4 Adders

- This technique generalizes
  - You can build an 8-bit multiplier using
    4 4-bit multipliers and 4 8-bit adders
  - $O(N^2 + N) = O(N^2)$
An $O(N^2)$ Multiplier In Logic

The functional blocks would look like

\[
\begin{array}{c}
\text{Product bits} \\
\text{HA} \quad \text{Add} \quad \text{Add} \\
\text{Mult} \quad \text{Mult} \quad \text{Mult} \\
B \quad C \quad A \quad D \quad B
\end{array}
\]
A Trick

- The two middle partial products can be computed using a single multiplier and other partial products
- \[ DA + CB = (C + D)(A + B) - (CA + DB) \]
- 3 multipliers
- 8 adders
- This can be applied recursively (i.e. applied within each partial product)
- Leads to \( O(N^{1.58}) \) adders
- This trick is becoming more popular as \( N \) grows. However, it is less regular, and the overhead of the extra adders is high for small \( N \)
Let's Try it By Hand

1) Choose 2, 2 digit numbers to multiply \( ab \times cd \)

\( 42 \times 37 \)

2) Multiply \( p_1 = a \times c, \ p_2 = b \times d, \ p_3 = (c + d)(a + b) \)

\( p_1 = 4 \times 3 = 12, \ p_2 = 2 \times 7 = 14, \)

\( p_3 = (4+2)(3+7) = 60 \)

3) Find partial subtracted sum, \( SS = p_3 - (p_1 + p_2) \)

\( SS = 60 - (12 + 14) = 34 \)

4) Add to find product, \( p = 100*p_1 + 10*SS + p_2 \)

\( p = 1200 + 340 + 14 = 1554 = 42 \times 37 \)
An $O(N^{1.58})$ Multiplier In Logic

The functional blocks would look like

\[
\begin{array}{cccc}
\text{A} & \text{B} \\
\times & \text{C} & \text{D} \\
\text{DB} & \text{SS} & \text{CA}
\end{array}
\]

Where
\[
\text{SS} = (C+D)(A+B) - (CA+DB)
\]
Binary Division

• Division merely reverses the process
  – Rather than adding successively larger partial products, subtract successively smaller divisors
  – When multiplying, we knew which partial products to actually add (based on the whether the corresponding bit was a 0 or a 1)
  – In division, we have to try *both ways*

```
multiplication upside-down

P P P P P P P P
- D D D D
- D D D D
- D D D D
- D D D D
- D D D D
```

\[
\begin{align*}
P & = 0 \text{ or } 1? \\
Q_3 & = 0 \text{ or } 1? \\
Q_2 & = 0 \text{ or } 1? \\
Q_1 & = 0 \text{ or } 1? \\
Q_0 & = 0 \text{ or } 1? \\
R & = 0 \text{ or } 1?
\end{align*}
\]
Restoring Division

Start: Align MSBs of Divisor and Remainder, K = number of bits shifted, Quotient = 0

1. Subtract Divisor from the Remainder leave the result in the Remainder
2. Test Remainder:
   - If $\geq 0$, Shift Quotient left one bit set rightmost bit = 1
   - If $< 0$, Restore Remainder by adding Divisor
     - Shift Quotient left one bit set rightmost bit = 0
3. Shift Divisor right one bit
4. Repeat K+1 times
Division Example

Step 1:

R   D       Q
42 ÷ 7  =    6

Start:

Q =    0  =  00000000
R =   42  =  00101010
D = (7*8) =  00111000

Subtract:

R =    42  = 00101010
D = -(7*8) = 00111000
-14  = 11110001

Restore:

R =    42  = 00101010

Shifts:

Q = 00000000
D = 00011100

Step 2:

R   D       Q
42 ÷ 7  =    6

Start: Align bits of Divisor and Remainder. K = number of bits shifted. Quotient = 0

Q = 00000000
R = 00101010
D = (7*4) = 00011100

Subtract:

R = 42 = 00101010
D = -(7*4) = 00011100
R = 14 = 00001110

Shifts:

Q = 00000001
D = 00001110

Note: K = 3, so repeat 4 times
Division Example (cont)

Step 3:
\[
\begin{array}{c|c|c}
R & D & Q \\
42 & 7 & 6 \\
\end{array}
\]

\[
\begin{align*}
Q &= 1 = 00000001 \\
R &= 14 = 00001110 \\
D &= (7*2) = 00001110 \\
\end{align*}
\]

Subtract:
\[
\begin{align*}
R &= 14 = 00001110 \\
D &= -(7*2) = 00001110 \\
0 &= 00000000 \\
\end{align*}
\]

No Restore
Shifts:
\[
\begin{align*}
Q &= 00000011 \\
D &= 00000111 \\
\end{align*}
\]

Step 4:
\[
\begin{array}{c|c|c}
R & D & Q \\
42 & 7 & 6 \\
\end{array}
\]

\[
\begin{align*}
Q &= 3 = 00000011 \\
R &= 0 = 00000000 \\
D &= 7 = 00000111 \\
\end{align*}
\]

Subtract:
\[
\begin{align*}
R &= 0 = 00000000 \\
D &= -7 = 00000111 \\
-7 &= 11111001 \\
\end{align*}
\]

Restore:
\[
\begin{align*}
R &= 0 = 00000000 \\
\end{align*}
\]

Shifts:
\[
\begin{align*}
Q &= 00000110 \\
D &= 00000011 \\
R &= 00000000 \\
\end{align*}
\]
Next Time

• We dive into floating point arithmetic