**Boolean Unit (The obvious way)**

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.

This is a straightforward, but not elegant design.
Cooler Bools

We can better leverage a MUX’s capabilities in our Boolean unit design, by connecting the bits to the select lines.

Why is this better?

While it might take a little logic to decode the truth table inputs, you only have to do it once, independent of the number of bits.

BTW, it also handles the MOV and MVN cases.
Decoding the Booleans and Others

It may seem a little tedious, but all the controls that we need can be derived from the ARM OpCode encodings.

The 'X's in the truth table are "don't cares" they provide flexibility in the implementation.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Code</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>Sub</th>
<th>Rsb</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0 0 0 0 0</td>
<td>0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>0 0 0 1</td>
<td>0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>0 0 1 0</td>
<td>X X X X 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSB</td>
<td>0 0 1 1</td>
<td>X X X 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>0 1 0 0</td>
<td>X X X 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td>0 1 0 1</td>
<td>X X X 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC</td>
<td>0 1 1 0</td>
<td>X X X 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSC</td>
<td>0 1 1 1</td>
<td>X X X 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>1 0 0 0</td>
<td>0 0 0 1 X X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEQ</td>
<td>1 0 0 1</td>
<td>0 1 1 0 X X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>1 0 1 0</td>
<td>X X X 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN</td>
<td>1 0 1 1</td>
<td>X X X 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORR</td>
<td>1 1 0 0</td>
<td>0 1 1 1 X X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>1 1 0 1</td>
<td>0 1 0 1 X X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>1 1 1 0</td>
<td>0 0 1 0 X X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVN</td>
<td>1 1 1 1</td>
<td>1 0 1 0 X X 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We give the "Math Center" of a computer a special name--the Arithmetic Logic Unit (ALU). For us, it just a big box of gates!
**Binary Multiplication**

The key to multiplication was memorizing a digit-by-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>

<table>
<thead>
<tr>
<th>×</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

You've got to be kidding... It can't be that easy
Suppose that you wanted to extend the ARM ISA to include a **nor** instruction like MIPS, how would the mux inputs of the BOOL functional block shown on the right be set?

A) $X, Y, Z = 1, W = 0$
B) $X = 0, Y, Z, W = 1$
C) $X = \text{NOT}(\text{OR}(A_i, B_i)), Y, Z, W = 0$
D) $X = 1, Y, Z, W = 0$
E) A NOR cannot be implemented with this functional block
Digit by digit = bit by bit

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

\[
\begin{array}{c|cc}
  \times & A_3 & A_2 & A_1 & A_0 \\
  \hline
  B_3 & A_3 B_0 & A_2 B_0 & A_1 B_0 & A_0 B_0 \\
  B_2 & A_3 B_1 & A_2 B_1 & A_1 B_1 & A_0 B_1 \\
  B_1 & A_3 B_2 & A_2 B_2 & A_1 B_2 & A_0 B_2 \\
  B_0 & A_3 B_3 & A_2 B_3 & A_1 B_3 & A_0 B_3 \\
\end{array}
\]

A_jB_i is a "partial product"

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

One can use this "Shift and Add" approach to write a multiply function in assembly language:

```assembly
; multiplies r0 and r1
mult:  mov     r3,#0           ; zero product
part:  tst     r1,#1           ; check if least significant bit=1
       addne   r3,r3,r0         ; add multiplicand to product
       mov     r0,r0,lsl #1     ; multiplicand *= 2
       movs    r1,r1,lsr #1     ; multiplier /= 2
       bne     part             ; continue while multiplier is not 0
       mov     r0,r3             ; copy product to return value
       bx      lr                ; return
```

Hum, maybe we could do something more clever.

```
<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0110 1010</td>
<td>0000 0000 0100 1000</td>
</tr>
<tr>
<td>0000 0000 0110 1011</td>
<td>0000 0000 1001 000_</td>
</tr>
<tr>
<td>0000 0000 0000 1010</td>
<td>0000 0000 0000 00__</td>
</tr>
<tr>
<td>0000 0000 0000 1011</td>
<td>0000 0010 0100 0___</td>
</tr>
<tr>
<td>0000 0000 0000 0100</td>
<td>0000 1001 000_ ___</td>
</tr>
<tr>
<td>0000 0000 0000 0010</td>
<td>0000 1011 1101 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0000</td>
<td>0000 1011 1101 0000</td>
</tr>
</tbody>
</table>
```
Multiplier Unit-Block

We introduce a new abstraction to aid in the construction of multipliers called the "Unsigned Multiplier unit-block." We did a similar thing last lecture when we converted our adder to an add/subtract unit.

$A_k$ are bits of the Multiplicand and $B_i$ are bits of the Multiplier.

The $P_{ik}$ inputs and outputs represent "partial products" which are partial results from adding together shifted instances of the Multiplicand. The initial $P_{0,k}$ is zero.
Simple Combinational Multiplier

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

not 16

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

Components

\( N \times HA \)

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

To determine the timing specification of a composite combinational circuit we find the worst-case path for every output to any input.

Is this faster than our assembly code?

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

Observation: Rather than propagating the carries to the next adder in each row, they can instead be forwarded to the next column of the following row.

\[ t_{PD} = 8 * t_{PD} \]
\[ t_{PD} = (N+N) * t_{PD} \]

Components

- \( N \times HA \)
- \( N^2 \times FA \)
**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 rows 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!

\[
\begin{align*}
B_{k+1,k} * A &= 0 * A \iff 0 \\
&= 1 * A \iff A \\
&= 2 * A \iff 2A \text{ or } 4A - 2A \\
&= 3 * A \iff 4A - A
\end{align*}
\]
**Booth Recoding of Multiplier**

A "1" in this bit means the previous stage needed to add $4\times A$. Since this stage is shifted by 2 bits with respect to the previous stage, adding $4\times A$ in the previous stage is like adding $A$ in this stage!

<table>
<thead>
<tr>
<th>$B_{2K+1}$</th>
<th>$B_{2K}$</th>
<th>$B_{2K-1}$</th>
<th>action</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>add 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add $A$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>add $A$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>add $2\times A$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sub $2\times A$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sub $A$</td>
<td>$-2\times A + A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sub $A$</td>
<td>$-A + A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>add 0</td>
<td></td>
</tr>
</tbody>
</table>

Hey, isn’t that a negative number?

Yep! Booth recoding works for 2-Complement integers, now we can build a signed multiplier.

<table>
<thead>
<tr>
<th>$-89 = 10100111.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$= -1 \times 2^0$</td>
</tr>
<tr>
<td>$+ 2 \times 2^2$</td>
</tr>
<tr>
<td>$+ (-2) \times 2^4$</td>
</tr>
<tr>
<td>$+ (-1) \times 2^6$</td>
</tr>
</tbody>
</table>

Hey, isn’t that a negative number?

Yep! Booth recoding works for 2-Complement integers, now we can build a signed multiplier.

From previous bit pair

Current bit pair

An encoding where each bit has the following weights:

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

$W(B_{2K}) = 1 \times 2^{2K}$

$W(B_{2K-1}) = 1 \times 2^{2K}$
Booth Multiplier Unit Block

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

<table>
<thead>
<tr>
<th>$B_{2K+1}$</th>
<th>$B_{2K}$</th>
<th>$B_{2K-1}$</th>
<th>x2</th>
<th>Sub</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>
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.

Consider 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
  - $O(N^2 + N) = O(N^2)$
$O(N^2)$ MULTIPLIER LOGIC

The functional blocks look like

```
AB
x
CD
```

```
DB
DA
CB
CA
```

```
Mult  Mult  Mult  Mult

Add   Add   Add   Add

HA    Add   Add   B
```

Product bits
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 digits: \( p1 = a \times c, \quad p2 = b \times d, \quad p3 = (c + d)(a + b) \)

\[
p1 = 4 \times 3 = 12, \quad p2 = 2 \times 7 = 14, \quad p3 = (4 + 2)(3 + 7) = 60
\]

3) Compute partial subtracted sum, \( SS = p3 - (p1 + p2) \)

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

4) Add as follows: \( p = 100 \times p1 + 10 \times SS + p2 \)

\[
P = 1200 + 340 + 14 = 1554 = 42 \times 37
\]
An $O(N^{1.58})$ Multiplier

The functional blocks would look like:

$$AB \times CD \rightarrow DB$$

Where

$$SS = (C+D)(A+B) - (CA+DB)$$

Note: Adders with a bubble on one of their inputs becomes a subtractor in this notation.