We can add "much" faster

Using more gates we can speed up adding considerably if we add 2 "free" extra outputs from our adder

- $P$, Propagate, means the carry-out depends entirely on the carry-in
- $G$, generates a carry-out regardless of the carry-in

<table>
<thead>
<tr>
<th>$C_i$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C_o$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
CARRY-SKIP ADDERS

If all full adders in a contiguous block have their Propagate true, then the incoming carry-in can "skip" over the entire block!

Requires extra AND gates and a MUX, but reduces the worst case add-time
**Full Carry-Lookahead**

The fastest adders use full carry look-ahead.

- Given the Ps and Gs of a block, one can simultaneously compute the carry-ins for all bits as well as the block using the 3-level SOP methods discussed last lecture.
- Results in an $\Theta(\log_2(N))$, $T_{pd}$, like an N-input AND gate, using $\approx 2x$ more gates.
AN ARITHMETIC LOGIC UNIT

So they all rolled over and one fell out!

- Shifts of shifts
- Boolean logic
- An ALU
Shifting Logic

Shifting is a common operation that is applied to groups of bits. Shifting is used for alignment, selecting parts of a word, as well as for arithmetic operations.

- $X << 1$ is approx the same as $2 \times X$
- $X >> 1$ can be the same as $X/2$

For example:

$X = 00010100_2 = 20_{10}$

Left Shift:

$(X << 1) = 00101000_2 = 40_{10}$

Right Shift:

$(X >> 1) = 00010100_2 = 10_{10}$

Signed or "Arithmetic" Right Shift:

$(-X >> 1) = (11101100_2 >> 1) = 11110110_2 = -10_{10}$
More Shifting

Using the same basic idea we can build left shifters of arbitrary shift amounts using muxes.

Each shift amount requires its own set of muxes.

Hum, maybe we could do something more clever.
If we connect our "shift-left-two" shifter to the output of our "shift-left-one" we can shift by 0, 1, 2, or 3 bits.

And, if we add one more "shift-left-four" shifter we can do any shift up to 7 bits!

So, let's put a box around it and call it a new functional block.
Adding a Twist

It would be straightforward to construct a "right barrel shifter" unit. However, a simple trick that enables a "left barrel shifter" to do both.

![Diagram of Left Barrel Shifter]
**Rotate AND Arithmetic Shifts**

The basics are the same for logical and arithmetic shifts except instead of shifting in zeros for the vacated bits on arithmetic shifts, you shift in copies of \( X_0 \). For rotates, you shift in the bits from the other end.

This adds two control lines, ASR, and ROT, which are shared amongst all of the LFTx units.

\[ X_7 \quad S_7 \]
\[ X_6 \quad S_6 \]
\[ X_5 \quad S_5 \]
\[ X_4 \quad S_4 \]
\[ X_3 \quad S_3 \]
\[ X_2 \quad S_2 \]
\[ X_1 \quad S_1 \]
\[ X_0 \quad S_0 \]

The ASR MUX is just an "AND" gate!
Bitwise Logical Operations

We need to perform logical operations, or Booleans, on groups of bits. Which ones?

**ANDing** is used for "masking" off groups of bits.

ex. \(10101110 \& 00001111 = 00001110\) (mask selects last 4 bits)

**ORing** is used for "setting" groups of bits.

ex. \(10101110 \mid 00001111 = 10101111\) (1’s set last 4 bits)

**EORing** is used for "complementing" groups of bits.

ex. \(10101110 \^ 00001111 = 10100001\) (complement last 4 bits)

**BICing** is used to "clear" groups of bits (BIC = bit clear).

ex. \(10101110 \& \neg(00001111) = 01010000\) (1’s clear)
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.

Which ever way makes the most sense to you. Let’s get a box around it!

I should pay more attention to those muxes.
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>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>EOR</td>
<td>000011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>000100</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RSB</td>
<td>000111</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>010000</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ADC</td>
<td>010011</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SBC</td>
<td>010100</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RSC</td>
<td>010111</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TST</td>
<td>100000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>TEQ</td>
<td>100011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>CMP</td>
<td>100100</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CMN</td>
<td>100111</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ORR</td>
<td>110000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>MOV</td>
<td>110011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>BIC</td>
<td>110100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>MVN</td>
<td>110111</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>
AN ALU, AT LAST

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!