Arithmetic Circuits

Didn’t I learn how to do addition in the second grade? UNC courses aren’t what they used to be...

Finally; time to build some serious functional blocks

We’ll need a lot of boxes

Reading: Study Chapter 3.
Review: 2's Complement

N bits

-2^{N-1} 2^{N-2} \ldots \ldots 2^3 2^2 2^1 2^0

“sign bit”

Range: \(-2^{N-1}\) to \(2^{N-1} - 1\)

“binary” point

8-bit 2’s complement example:
\[11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42\]

If we use a two’s-complement representation for signed integers, the same binary addition procedure will work for adding both signed and unsigned numbers.

By moving the implicit “binary” point, we can represent fractions too:
\[1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625\]
Binary Addition

Here’s an example of binary addition as one might do it by “hand”:

\[
\begin{array}{c}
\text{A: } 1101 \\
\text{B: } + 0101 \\
\hline
10010
\end{array}
\]

Adding two N-bit numbers produces an (N+1)-bit result.

Let’s start by building a block that adds one column:

Then we can cascade them to add two numbers of any size…
Designing a Full Adder: From Last Time

1) Start with a truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Ci</th>
<th>Co</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</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>

2) Write down eqns for the “1” outputs

\[
C_o = \overline{C_i}AB + C_i\overline{A}B + C_i\overline{A}B + C_iAB \\
S = \overline{C_i}AB + C_i\overline{A}B + C_i\overline{A}B + C_iAB
\]

3) Simplifying a bit

\[
C_o = C_i(A + B) + AB \\
S = C_i \oplus A \oplus B
\]

\[
C_o = C_i(A \oplus B) + AB \\
S = C_i \oplus (A \oplus B)
\]
For Those Who Prefer Logic Diagrams …

\[ C_o = C_i (A \oplus B) + AB \]
\[ S = C_i \oplus (A \oplus B) \]

- A little tricky, but only 5 gates/bit
Subtraction: $A - B = A + (-B)$

Using 2’s complement representation: $-B = \overline{B} + 1$

So let’s build an arithmetic unit that does both addition and subtraction. Operation selected by control input:
**Condition Codes**

Besides the sum, one often wants four other bits of information from an arithmetic unit:

- **Z (zero):** result is \( O \)
- **N (negative):** result is \( < O \)
- **C (carry):** indicates that add in the most significant position produced a carry, e.g., “\( 1 + (-1) \)” from last FA
- **V (overflow):** indicates that the answer has too many bits to be represented correctly by the result width, e.g., “\( (2^{i-1} - 1) + (2^{i-1} - 1) \)”

**Signed comparison:**
- **LT** \( N \oplus V \)
- **LE** \( Z + (N \oplus V) \)
- **EQ** \( Z \)
- **NE** \( \sim Z \)
- **GE** \( \sim (N \oplus V) \)
- **GT** \( \sim (Z + (N \oplus V)) \)

**Unsigned comparison:**
- **LTU** \( C \)
- **LEU** \( C + Z \)
- **GEU** \( \sim C \)
- **GTU** \( \sim (C + Z) \)
**$T_{PD}$ of Ripple-Carry Adder**

Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001.

$$t_{PD} = (t_{PD,XOR} + t_{PD,AND} + t_{PD,OR}) + (N-2)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

$\Theta(N)$ is read “order N” and tells us that the latency of our adder grows in proportion to the number of bits in the operands.
Faster Carry Logic

Let's see if we can improve the speed by rewriting the equations for $C_{OUT}$:

$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$

$$= AB + (A + B)C_{IN}$$

$$= G + P C_{IN} \quad \text{where } G = AB \text{ and } P = A + B$$

Actually, $P$ is usually defined as $P = A \oplus B$ which won't change $C_{OUT}$ but will allow us to express $S$ as a simple function of $P$ and $C_{IN}$: $S = P \oplus C_{IN}$

To generate the Carry of the $N^{th}$ bit:

$$C_N = G_{N-1} + P_{N-1}C_{N-1}$$

$$= G_{N-1} + P_{N-1} G_{N-2} + P_{N-1} P_{N-2}C_{N-2}$$

$$= G_{N-1} + P_{N-1} G_{N-2} + P_{N-1} P_{N-2} G_{N-3} + \ldots + P_{N-1} \ldots P_{0} C_{IN}$$

$C_N$ in only 3 (!) gate delays:
1 for $P/G$ generation, 1 for ANDs, 1 for final OR
N-Bit Addition in Constant Time?

If we had \((N+1)\)-input gates and didn’t mind a lot of loading on the \(P\) signals, the propagation delay of adder built using \(P/G\) equation to compute \(C_{IN}\) of each bit would be:

\[
4 \text{ gate delays} \approx \Theta(1)
\]

So if we had \((N+1)\)-input gates and didn’t mind a lot of loading on the \(P\) signals, the propagation delay of adder built:

\[
\text{N-Bit Addition in Constant Time?}
\]

Hierarchical carry-lookahead adders

Faster ripple-carry implementations

Of course, this is impractical when \(N\) is “large” (i.e., \(> 4\)) but it does lead to some interesting ideas:
Carry-Lookahead Adders (CLA)

We can build a hierarchical carry chain by generalizing our definition of the Carry Generate/Propagate (GP) Logic. We start by dividing our addend into two parts, a higher part, \( H \), and a lower part, \( L \). The GP function can be expressed as follows:

\[
G_{HL} = G_H + P_H G_L \\
P_{HL} = P_H P_L
\]

Generate a carry out if the high part generates one, or if the low part generates one and the high part propagates it. Propagate a carry if both the high and low parts propagate theirs.

Hierarchical building block

P/G generation

1st level of lookahead
8-bit CLA (GP Generation)

We can build a tree of GP units to compute the generate and propagate logic for any sized adder. For a $2^N$-bit adder, we need $2^N-1$ GP units.

$$C = G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4 + \ldots + P_7 \ldots P_0 C_{IN}$$
8-bit CLA (Carry Generation)

Now, given the value of the carry-in of the least-significant bit, we can generate the carries for every adder.

\[ c_j = G_{j-i} + P_{j-i}c_i \]

Notice that the inputs on the right of each C blocks are the same as the inputs on the left of each corresponding GP block.
8-Bit CLA (Complete)

Notice that we don’t need the carry-out output of the adder any more.

\[ t_{PD} = \Theta(\log(N)) \]
Carry-Skip Adders

Idea: full P/G equations are complicated, but P by itself is simple. So just use P to “skip” carry across a block of ripple-carry adders:

(A) Carries ripple simultaneously through each block; if block generates a carry, it appears on carry-out of block (similar to G). If carry-in is 0 at start of operation, no spurious carry-outs will be generated.

(B) If carry-in and \( P_{\text{BLOCK}} \) are both true, carry skips to next block

(C) Carry ripples through final block. \( t_{PD} = 2*[K+(N/K-2)+K] \)

With variable size blocks \( t_{PD} \rightarrow \Theta(\sqrt{N}) \)
Carry-Select Adders

Idea: do two additions, one assuming carry-in is 0, the other assuming carry-in is 1. Use MUX to select correct answer when correct carry-in is known.

Left blocks can be bigger – more ripple time time while waiting for select

With one stage: 50% more cost, but twice as fast as ripple-carry
With multiple (variable-size) blocks: \( t_{PD} \rightarrow \Theta(\sqrt{N}) \)
Adder Summary

Adding is not only a common, but it is also tends to be one of the most time-critical of operations. As a result, a wide range of adder architectures have been developed that allow a designer to tradeoff complexity (in terms of the number of gates) for performance.

Smaller / Slower  Bigger / Faster
Ripple  Carry  Carry  Select  Carry  Lookahead

A this point we’ll define a high-level functional unit for an adder, and specify the details of the implementation as necessary.
Shifting is a common operation that is applied to groups of bits. Shifting can be used for alignment, as well as for arithmetic operations.

\[ X << 1 \text{ is approx the same as } 2^X \]
\[ X >> 1 \text{ can be the same as } X/2 \]

For example:

\[ X = 20_{10} = 00010100_2 \]

Left Shift:

\[ (X << 1) = 00101000_2 = 40_{10} \]

Right Shift:

\[ (X >> 1) = 00010100_2 = 10_{10} \]

Signed or “Arithmetic” Right Shift:

\[ (-X >> 1) = (11101100_2 >> 1) = 1110110_2 = -10_{10} \]
More Shifting

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

Each shift amount requires its own set of muxes.

Hum, maybe we could do something more clever.
Barrel Shifting

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.

Barrel Shifter

$$A \xrightarrow{\log_2(N) \text{ bits}} Y$$

Left Barrel Shifter

A

log$_2$(N) bits

Y

N-bits

Left Barrel Shifter

A

log$_2$(N) bits

Y

N-bits

N-bits

log$_2$(N) bits

N-bits

N-bits

log$_2$(N) bits

N-bits

N-bits

log$_2$(N) bits

N-bits

N-bits

log$_2$(N) bits

N-bits

N-bits
Barrel Shifting with a Twist

At this point it would be straightforward to construct a “Right barrel shifter” unit. However, a simple trick that enables a left shifter to do both.
Boolean Operations

It will also be useful to perform logical operations on groups of bits. Which ones?

ANDing is useful for “masking” off groups of bits.
   ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for “clearing” groups of bits.
   ex. 10101110 & 00001111 = 00001110 (0’s clear first 4 bits)

ORing is useful for “setting” groups of bits.
   ex. 10101110 | 00001111 = 10101111 (1’s set last 4 bits)

XORing is useful for “complementing” groups of bits.
   ex. 10101110 ^ 00001111 = 10100001 (1’s complement last 4 bits)

NORing is useful.. Uhm, because John Hennessy says it is!
   ex. 10101110 # 00001111 = 01010000 (0’s complement, 1’s clear)
Boolean Unit (The book’s 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 too elegant of a 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?

1) 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.

2) It is trivial to extend this module to support any 2-bit logical function. (How about NAND, John? Actually A & /B might be more useful)

I should pay more attention to those muxes.

Which ever way makes the most sense to you. Let’s get a box around it!
An ALU, at Last

Now we’re ready for a big one! An Arithmetic Logic Unit.