Digital Arithmetic

Tyler Bletsch
Duke University

Slides are derived from work by Andrew Hilton (Duke)
Last Time in ECE 550….

• Who can remind us what we talked about last time?
  • Numbers
    • One hot
    • Binary
    • Hex
  • Digital Logic
    • Sum of products
    • Encoders
    • Decoders
  • Binary Numbers and Math
    • Overflow
Designing a 1-bit adder

- What boolean function describes the low bit?
  - XOR

- What boolean function describes the high bit?
  - AND

\[
\begin{align*}
0 + 0 &= 00 \\
0 + 1 &= 01 \\
1 + 0 &= 01 \\
1 + 1 &= 10
\end{align*}
\]
Designing a 1-bit adder

• Remember how we did binary addition:
  • Add the **two bits**
  • Do we have a **carry-in** for this bit?
  • Do we have to **carry-out** to the next bit?

\[
\begin{array}{c}
01101100 \\
+00101100 \\
\hline
10011001
\end{array}
\]
Designing a 1-bit adder

- So we’ll need to add three bits (including carry-in)
- Two-bit output is the **carry-out** and the **sum**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>C&lt;sub&gt;in&lt;/sub&gt;</th>
<th>a + b + C&lt;sub&gt;in&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>
A 1-bit Full Adder

\[
\begin{array}{c|c|c|c}
\text{a} & \text{b} & \text{Cin} & \text{Sum} \\
\hline
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[01101100 + 00101100 = 10011001\]
Ripple Carry

- Full Adder = Add 1 Bit
  - Can chain together to add many bits
  - Upside: Simple
  - Downside?
    - Slow. Let’s see why.
Full adder delay

- Cout depends on Cin
  - 2 "gate delays" through full adder for carry
Ripple Carry

- Carries form a chain
  - Need CO of bit N is CI of bit N+1
- For few bits (e.g., 4) no big deal
  - For realistic numbers of bits (e.g., 32, 64), slow
Adding

- Adding is important
  - Want to fit add in single clock cycle
    - (More on clocking soon)
    - Why? Add is ubiquitous

- Ripple Carry is slow
  - Maybe can do better?
  - But seems like Cin always depends on prev Cout
  - ...and Cout always depends on Cin...
Hardware != Software

• If this were software, we’d be out of luck
  • But hardware is different
  • Parallelism: can do many things at once
  • Speculation: can guess
Carry Select

- Do three things at once (32 gates)
  - Add low 16 bits
  - Add high 16 bits assuming CI = 0
  - Add high 16 bits assuming CI = 1
- Then pick correct assumption for high bits (2—3 gates)
• Could apply same idea again
  • Replace 16-bit RC adders with 16-bit CS adders
    • Reduce delay for 16 bit add from 32 to 18
    • Total 32 bit adder delay = 20
• So... just go nuts with this right?
Tradeoffs

- Tradeoffs in doing this
  - Power and Area ($\sim$ number of gates)
    - Roughly double every "level" of carry select we use
  - Less return on increase each time
    - Adding more mux delays
  - Wire delays increase with area
    - Not easy to count in slides
    - But will eat into real performance

- Fancier adders exist:
  - Carry-lookahead, conditional sum adder, carry-skip adder, carry-complete adder, etc...
Recall: Subtraction

- 2’s complement makes subtraction easy:
  - Remember: \( A - B = A + (-B) \)
  - And: \( -B = \sim B + 1 \)
    - that means flip bits (“not”)
  - So we just flip the bits and start with \( \text{CI} = 1 \)
  - Fortunate for us: **makes circuits easy**

\[
\begin{align*}
1 \\
0110101 & \quad \rightarrow \quad 0110101 \\
-1010010 & \quad + \quad 0101101 \\
\end{align*}
\]
32-bit Adder/subtractor

- Inputs: A, B, Add/Sub (0=Add, 1 = Sub)
- Outputs: Sum, Cout, Ovf (Overflow)
• By the way:
  • That thing has about 3,000 transistors
  • Aren’t you glad we have abstraction?
Arithmetic Logic Unit (ALU)

- ALUs do a variety of math/logic
  - Add
  - Subtract
  - Bit-wise operations: And, Or, Xor, Not
  - Shift (left or right)

- Take two inputs (A,B) + operation (add, shift..)
  - Do a variety in parallel, then mux based on op
Bit-wise operations: SHIFT

- **Left shift (<<)**
  - Moves left, bringing in 0s at right, excess bits “fall off”
  - $10010001 << 2 = 01000100$
  - $x << k$ corresponds to $x \times 2^k$

- **Logical (or unsigned) right shift (>>)**
  - Moves bits right, bringing in 0s at left, excess bits “fall off”
  - $10010001 >> 3 = 00010010$
  - $x >> k$ corresponds to $x / 2^k$ for unsigned $x$

- **Arithmetic (or signed) right shift (>>>)**
  - Moves bits right, bringing in (sign bit) at left
  - $10010001 >> 3 = 11110010$
  - $x >>> k$ corresponds to $x / 2^k$ for signed $x$
Shift: Implementation…?

- Suppose an 8-bit number
  \[ b_7b_6b_5b_4b_3b_2b_1b_0 \]
  Shifted left by a 3 bit number
  \[ s_2s_1s_0 \]

- Option 1: Truth Table?
  - 2048 rows? Not appealing

...but you can do it. Truth table gives this expression for output bit 0:
Let’s simplify

- Simpler problem: 8-bit number shifted by 1 bit number (shift amount selects each mux)

```
b_7     out_7
b_6     out_6
b_5     out_5
b_4     out_4
b_3     out_3
b_2     out_2
b_1     out_1
b_0     out_0
0        out_0
```
Let’s simplify

- Simpler problem: 8-bit number shifted by 2 bit number

```
<table>
<thead>
<tr>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>out7</th>
<th>out6</th>
<th>out5</th>
<th>out4</th>
<th>out3</th>
<th>out2</th>
<th>out1</th>
<th>out0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Now shifted by 3-bit number

- Full problem: 8-bit number shifted by 3 bit number
Now shifted by 3-bit number

- Shifter in action: shift by 000 (all muxes have S=0)
Now shifted by 3-bit number

- Shifter in action: shift by 010
- From L to R: $S = 0, 1, 0$
Now shifted by 3-bit number

- Shifter in action: shift by 011
  - From L to R: S = 1, 1, 0 (reverse of shift amount)
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - \( \pi = 3.14159265358965\ldots \)
  - \( \frac{1}{2} = 0.5 \)
- How could we represent these sorts of numbers?
  - Fixed Point
  - Rational
  - Floating Point (IEEE Single Precision)
Floating Point

• Think about scientific notation for a second:

• For example:
  6.02 * 10^{23}

• Real number, but comprised of ints:
  • 6  generally only 1 digit here
  • 2  any number here
  • 10 always 10 (base we work in)
  • 23 can be positive or negative

• Can we do something like this in binary?
Floating Point

• How about:
  • +/- X.YYYYYY * 2^{+/-N}

• Big numbers: large positive N
• Small numbers (<1): negative N
• Numbers near 0: small N

• This is “floating point”: most common way
IEEE single precision floating point

- Specific format called IEEE single precision:
  - +/- 1.YYYYY * 2^{(N-127)}
  - “float” in Java, C, C++, ...

- Assume X is always 1 (save a bit)
- 1 sign bit (+ = 0, 1 = -)
- 8 bit biased exponent (do N-127)
- Implicit 1 before binary point
- 23-bit mantissa (YYYYY)
Binary fractions

1. YYYY has a binary point
   - Like a decimal point but in binary
   - After a decimal point, you have
     - tenths
     - hundredths
     - thousandths
     - ...

So after a binary point you have...
- Halves
- Quarters
- Eighths
- ...

Floating point example

- Binary fraction example:
  \[101.101 = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5.625\]

- For floating point, needs normalization:
  \[1.01101 \times 2^2\]

- Sign is +, which = 0

- Exponent = 127 + 2 = 129 = 1000 0001

- Mantissa = 1.011 0100 0000 0000 0000 0000

\[
\begin{array}{cccccc}
31 & 30 & 23 & 22 & \cdots & 0 \\
0 & 1000 & 0001 & 011 & 0100 & 0000 0000 0000 0000
\end{array}
\]
Example:
What floating-point number is: 
0xC1580000?
What floating-point number is 0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

\[ X = \begin{array}{cccccc}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

Sign = 1 which is negative

Exponent = (128+2)-127 = 3

Mantissa = 1.1011

\[-1.1011 \times 2^3 = -1101.1 = -13.5\]
How do you represent 0.0?

- Why is this a trick question?
  - 0.0 = 000000000
  - But need 1.XXXXX representation?

Exponent of 0 is denormalized

- Implicit 0. instead of 1. in mantissa
  - Allows 0000....0000 to be 0
  - Helps with very small numbers near 0

Results in +/- 0 in FP (but they are “equal”)
Other weird FP numbers

- Exponent = 1111 1111 also not standard
  - All 0 mantissa: +/- \infty
    - \(1/0 = +\infty\)
    - \(-1/0 = -\infty\)
  - Non zero mantissa: Not a Number (NaN)
    - \(\sqrt{-42} = \text{NaN}\)
Floating Point Representation

- Double Precision Floating point:

64-bit representation:
  - 1-bit sign
  - 11-bit (biased) exponent
  - 52-bit fraction (with implicit 1).

- “double” in Java, C, C++, ...

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bit</td>
<td>52-bit</td>
</tr>
</tbody>
</table>
Danger: floats cannot hold all ints!

- Many programmers think:
  - Floats can represent all ints
  - NOT true
- Doubles can represent all 32-bit ints
  (but not all 64-bit ints)
Wrap Up

• Implementation of Math
  • Addition/Subtraction
  • Shifting

• Floating Point Numbers
  • IEEE representation
  • Denormalized Numbers

• Next Time:
  • Storage
  • Clocking