ECE 550: Fundamentals of Computer Systems and Engineering

Digital Arithmetic
Admin

• Homework
  • Homework 1

• Reading:
  • Chapter 3
Last Time in ECE 550…. 

- Who can remind us what we talked about last time?
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
Implementing Addition

- First, one bit addition.
  - Three inputs: Carry In (CI), A, B
  - Two outputs Carry Out (CO), Sum (S)

- Go around room for truth table:

<table>
<thead>
<tr>
<th>CI</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Implementing Addition

- First, one bit addition.
  - Three inputs: Carry In (CI), A, B
  - Two outputs Carry Out (CO), Sum (S)
- Go around room for truth table:

<table>
<thead>
<tr>
<th>CI</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Half Adder

- Ignore CI for a second (assume is 0)
  - Can simplify a lot and build “half adder”
    - Formula for S?
    - Formula for CO?

<table>
<thead>
<tr>
<th>CI</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Half Adder

- Ignore CI for a second (assume is 0)
  - Can simplify a lot and build “half adder”
    - Formula for S?  \( A \ XOR \ B \)
    - Formula for CO?  \( A \ AND \ B \)

<table>
<thead>
<tr>
<th>CI</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Half Adder

- Half adder:
  - 1 XOR and 1 AND
  - Can anyone guess why its called a **half** adder?
Implementing Addition

- Re-visit Truth table, but..
  - Use Half-Sum and Half-CO (results of Half-Adder)
- Go around room for truth table:

<table>
<thead>
<tr>
<th>CI</th>
<th>Half-Sum</th>
<th>Half-CO</th>
<th>S</th>
<th>CO</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>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td>1</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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Implementing Addition

- Re-visit Truth table, but..
  - Use Half-Sum and Half-CO (results of Half-Adder)
- Go around room for truth table:

<table>
<thead>
<tr>
<th>CI</th>
<th>Half-Sum</th>
<th>Half-CO</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ECE 550 (Hilton): Digital Arithmetic
Implementing Addition

- Re-visit Truth table, but..
  - Use Half-Sum and Half-CO (results of Half-Adder)
- Go around room for truth table:

<table>
<thead>
<tr>
<th>CI</th>
<th>Half-Sum</th>
<th>Half-CO</th>
<th>S</th>
<th>CO</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>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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Implementing Addition

- Formulas:
  - Sum?
  - CO?

<table>
<thead>
<tr>
<th>CI</th>
<th>Half-Sum</th>
<th>Half-CO</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Implementing Addition

- Formulas:
  - Sum? \( \text{CI xor Half-Sum} \)
  - CO? \( (\text{CI and Half-Sum}) \text{ OR Half-CO} \)

<table>
<thead>
<tr>
<th>CI</th>
<th>Half-Sum</th>
<th>Half-CO</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Implementing Addition

- **Formulas:**
  - **Sum?** [CI xor Half-Sum]
  - **CO?** (CI and Half-Sum) OR Half-CO

<table>
<thead>
<tr>
<th>CI</th>
<th>Half-Sum</th>
<th>Half-CO</th>
<th>S</th>
<th>CO</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Full Adder

- Full Adder
- 2 Half Adders + an OR Gate
Ripple Carry

- Full Adder = Add 1 Bit
  - Can chain together to add many bits
  - Upside: Simple
  - Downside?
**Ripple Carry**

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

- 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
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 (~= 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: recitation
  - Can do even better
Recall: Subtraction

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

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

- Inputs: A, B, Add/Sub (0=Add, 1 = Sub)
- Outputs: Sum, Cout, Ovf (Overflow)
32-bit Adder/subtractor

- 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
Let's simplify

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

- Simpler problem: 8-bit number shifted by 2 bit number (new muxes selected by 2\text{nd} bit)
Now shifted by 3-bit number

- Full problem: 8-bit number shifted by 3 bit number (new muxes selected by 3\textsuperscript{rd} bit)
Now shifted by 3-bit number

- Shifter in action: shift by 000
Now shifted by 3-bit number

- Shifter in action: shift by 010
Now shifted by 3-bit number

- Shifter in action: shift by 011
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - $\pi = 3.145...$
  - $\frac{1}{2} = 0.5$
- 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:
  \[ 6.02 \times 10^{23} \]
- 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:
• \[ +/- \ X.YYYYYY \times 2^{+/-N} \]

• 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...
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
  - Eights
  - ....
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

\[
\begin{array}{cccccccccccccccc}
31 & 30 & 23 & 22 & & & & & & & & & & & & 0 \\
0 & 1000 & 0001 & 011 & 0100 & 0000 & 0000 & 0000 & 0000
\end{array}
\]
Floating Point Representation

Example:
What floating-point number is:
0xC1580000?
Answer

What floating-point number is 0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

Sign = 1 which is negative
Exponent = (128+2)-127 = 3
Mantissa = 1.1011
-1.1011x2^3 = -1101.1 = -13.5
Trick question

- How do you represent 0.0?
  - Why is this a trick question?
Trick question

• How do you represent 0.0?
  • Why is this a trick question?
  • 0.0 = 000000000
  • But need 1.XXXX representation?
Trick question

- 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: +/- ∞
    1/0 = +∞
    -1/0 = -∞
  • 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++, ...

\[
\begin{array}{ccc}
S & \text{Exp} & \text{Mantissa} \\
1 & 11\text{-bit} & 52 - \text{bit}
\end{array}
\]
## Danger: floats cannot hold all ints!

- Many programmers think:
  - Floats can represent all ints
  - NOT true

- First summer internship I had:
  - Need some floats and some ints: just use floats!
  - Bug in their code!
  - Other developers shocked as I demonstrated problem...

- Doubles can represent all 32-bit ints
- (but not all 64-bit ints)

<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>

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8-bit</td>
<td>23-bit</td>
</tr>
</tbody>
</table>

---

**ECE 550 (Hilton): Digital Arithmetic**
Wrap Up

- Implementation of Math
  - Addition/Subtraction
  - Shifting
- Floating Point Numbers
  - IEEE representation
  - Denormalized Numbers
- Next Time:
  - Storage
  - Clocking