Overview

- Recall basic positional notation
- Binary Addition
  - Full Adder (Boolean Logic Revisited)
  - Ripple Carry
  - Carry-select adder
  - Carry lookahead adder
- Binary Number Representation
  - Sign & Magnitude, Ones Complement, Twos Complement

Manipulating representations of numbers

- Example (from 2nd grade)

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
5 & 3 & 9 & 6 \\
\hline
1 & 0 & 1 & 2 & 1 \\
\end{array}
\]

- Sequence of decimal digits (radix 10)
- Position represents significance (most -> least)
- Carry into next position
- 3-to-2 conversion at each step
- Results may be one digit longer, but assumed you could “make room” for it

Positional Notation

- Sequence of digits: \( D_{k-1} D_{k-2} \ldots D_0 \)
  represents the value
  \[ D_{k-1} B^{k-1} + D_{k-2} B^{k-2} + \ldots + D_0 B^0 \]
  where \( D_i \in \{ 0, \ldots, B-1 \} \)
- \( B \) is the “base” or “radix” of the number system
- Example: \( 2004_{10} \)
- Can convert from any radix to any other
  - \( 1101_2 = 13_{10} = 0D_{16} \)
  - \( 1CEB_{16} = 1\cdot16^3 + 12\cdot16^2 + 14\cdot16^1 + 8\cdot16^0 = 7400_{10} \)
  - \( 436_8 = 4\cdot8^2 + 3\cdot8^1 + 6\cdot8^0 = 286_{10} \)
Computer Number Systems

- We all take positional notation for granted
  - \( D_{k-1} D_{k-2} \ldots D_0 \) represents \( D_{k-1}B^{k-1} + D_{k-2}B^{k-2} + \ldots + D_0 B^0 \)
    where \( B \in \{0, \ldots, B-1\} \)
- We all understand how to compare, add, subtract these numbers
  - Add each position, write down the position bit and possibly carry to the next position
- Computers represent finite number systems
  - Generally radix 2
- How do they efficiently compare, add, sub?
  - How do we reduce it to networks of gates and FFs?
- Where does it break down?
  - Manipulation of finite representations doesn’t behave like same operation on conceptual numbers

Unsigned Numbers - Addition

Example: \( 3 + 2 = 5 \)
Unsigned binary addition
Is just addition, base 2
Add the bits in each position and carry

\[
\begin{array}{c}
+0 \quad 0 \quad 0 \quad 1 \\
+0 \quad 0 \quad 1 \\
\hline
0 \quad 1 \quad 0 \quad 1
\end{array}
\]

How do we build a combinational logic circuit to perform addition?
=> Start with a truth table and go from there

Binary Addition: Half Adder

\[
\begin{array}{c|c|c|c|c}
A_i & B_i & \text{Sum} & \text{Carry} \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
A_i & B_i & \text{Sum} = A_i \oplus B_i \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

Full-Adder (derivation)

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
A & B & C_i & \text{Carry Out} & S \\
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
A & B & C_i & \text{Carry Out} & S \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
S = C_i \text{xor} A \text{xor} B \\
C_O = B C_i + A C_i + A B = C_i (A + B) + A B
\]
Full Adder

A\rightarrow A\oplus B\oplus Cl S \rightarrow Co

B\rightarrow A\oplus B\oplus Cl S \rightarrow Co

Cl\rightarrow A\oplus B\oplus Cl S \rightarrow Co

Now we can connect them up to do multiple bits...

Ripple Carry

A3 B3 S3 C3
A2 B2 S2 C2
A1 B1 S1 C1
A0 B0 S0

Full Adder from Half Adders (little aside)

Standard Approach: 6 Gates

A\oplus B \rightarrow (A\oplus B)\oplus Cl \rightarrow S

Alternative Implementation: 5 Gates

A \rightarrow Half Adder S \rightarrow (A\oplus B)\oplus Cl \rightarrow S

B \rightarrow Half Adder S \rightarrow (A\oplus B)\oplus Cl \rightarrow S

Cl \rightarrow (A\oplus B)\oplus Cl \rightarrow S

A \oplus B \oplus Cl = A \oplus B + Cl = A + Cl

Delay in the Ripple Carry Adder

Critical delay: the propagation of carry from low to high order stages

Final sum and carry

\text{two gate delays to compute } Co

A_0 B_0 \rightarrow S_0@2 C_0@4
A_1 B_1 \rightarrow S_1@3 C_1@6
A_2 B_2 \rightarrow S_2@5 C_2@8
A_3 B_3 \rightarrow S_3@7 C_3@10

\text{late arriving signal}

@0 \rightarrow A \rightarrow 0 \rightarrow @1 \rightarrow @N+1 \rightarrow Co

@0 \rightarrow B \rightarrow 0 \rightarrow @1 \rightarrow @N+1 \rightarrow Co

@0 \rightarrow Cl \rightarrow 0 \rightarrow @1 \rightarrow @N+1 \rightarrow Co

4 stage adder
Ripple Carry Timing

Critical delay: the propagation of carry from low to high order stages

1111 + 0001 worst case addition

T0: Inputs to the adder are valid
T2: Stage 0 carry out (C1)
T4: Stage 1 carry out (C2)
T6: Stage 2 carry out (C3)
T8: Stage 3 carry out (C4)

2 delays to compute sum but last carry not ready until 6 delays later

Recall: Virtex-E CLB

CLB = 4 logic cells (LC) in two slices
LC: 4-input function generator, carry logic, storage ele’t
80 x 120 CLB array on 2000E

Adders (cont.)

Ripple Adder

Ripple adder is inherently slow because, in general s7 must wait for c7 which must wait for c6 …

\[ T \propto n, \quad \text{Cost} \propto n \]

How do we make it faster, perhaps with more cost?

Classic approach: Carry Look-Ahead

Or use a MUX !!!

Carry Select Adder

\[ T = \frac{T_{\text{ripple_adder}}}{2} + T_{\text{MUX}} \]

\[ \text{COST} = 1.5 \times \text{COST}_{\text{ripple_adder}} + (n+1) \times \text{COST}_{\text{MUX}} \]
**Extended Carry Select Adder**

- What is the optimal # of blocks and # of bits/block?
  - If # blocks too large delay dominated by total mux delay
  - If # blocks too small delay dominated by adder delay per block

\[ T \propto \sqrt{N}, \quad \text{Cost} \approx 2 \times \text{ripple} + \text{muxes} \]

**Carry Select Adder Performance**

- Compare to ripple adder delay:
  \[ T_{\text{total}} = 2 \sqrt{N} T_{\text{FA}} - T_{\text{FA}}, \text{assuming } T_{\text{FA}} = T_{\text{MUX}} \]
  For ripple adder \[ T_{\text{total}} = N T_{\text{FA}} \]
  “cross-over” at \( N=3 \), Carry select faster for any value of \( N>3 \).

- Is \( \sqrt{N} \) really the optimum?
  - From right to left increase size of each block to better match delays
  - Ex: 64-bit adder, use block sizes \([12 \, 11 \, 10 \, 9 \, 8 \, 7 \, 7]\)

- How about recursively defined carry select?

**Announcements**

- Reading Katz 5.6 and Appendix A
- Mid III will just stay put in final slot – no more fussing with it.
- Project demo at Lab Lecture friday
- Don’t hedge on lab workload reporting
  - It matters to us and is NOT a negative in your grade
- Lab5 | CP1 | CP2 crunch
  - It should lighten
  - Don’t hesitate to get guidance on the specifics of your approach from the TAs. They are there to help.
  - Lab5 solution walk-thru Friday after lab lecture

**What really happens with the carries**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cout</th>
<th>S</th>
<th>Carry action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Cin</td>
<td>kill</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Cin</td>
<td>~Cin</td>
<td>Propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Cin</td>
<td>~Cin</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Cin</td>
<td>1</td>
<td>generate</td>
</tr>
</tbody>
</table>

\[ \text{Carry Generate } G_i = A_i \, B_i \quad \text{must generate carry when } A = B = 1 \]
\[ \text{Carry Propagate } P_i = A_i \oplus B_i \quad \text{carry in will equal carry out here} \]

All generates and propagates in parallel at first stage. No ripple.
**Carry Kill / Prop / Gen example**

```
<table>
<thead>
<tr>
<th>K</th>
<th>P</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Carry Look Ahead Logic**

- Carry Generate \( G_i = A_i \cdot B_i \) must generate carry when \( A = B = 1 \)
- Carry Propagate \( P_i = A_i \oplus B_i \) carry in will equal carry out here

Sum and Carry can be re-expressed in terms of generate/propagate:

- \( S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \)
- \( C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i \)

\( C_{i+1} = A_i \cdot B_i + C_i \cdot (A_i + B_i) \)
\( = A_i \cdot B_i + C_i \cdot (A_i \oplus B_i) \)
\( = G_i + C_i \cdot P_i \)

**All Carries in Parallel**

Re-express the carry logic for each of the bits:

- \( C_1 = G_0 + P_0 \cdot C_0 \)
- \( C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \)
- \( C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0 \)
- \( C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0 \)

Each of the carry equations can be implemented in a two-level logic network

Variables are the adder inputs and carry in to stage 0!

**CLA Implementation**

Adder with Propagate and Generate Outputs

Increasingly complex logic
How do we extend this to larger adders?

- Faster carry propagation
  - 4 bits at a time
- But still linear
- Can we get to log?
- Compute propagate and generate for each adder BLOCK

Cascaded Carry Lookahead

4-bit adders with internal carry lookahead
second level carry lookahead unit, extends lookahead to 16 bits
One more level to 64 bits

Trade-offs in combinational logic design

- Time vs. Space Trade-offs
  Doing things fast requires more logic and thus more space
  Example: carry lookahead logic
- Simple with lots of gates vs complex with fewer

Arithmetic Logic Units
Critical component of processor datapath
Inner-most “loop” of most computer instructions

So what about subtraction?

- Develop subtraction circuit using the same process
  - Truth table for each bit slice
  - Borrow in from slice of lesser significance
  - Borrow out to slice of greater significance
  - Very much like carry chain
- Homework exercise
Finite representation?

- What happens when $A + B > 2^N - 1$?
  - Overflow
  - Detect?
    » Carry out

- What happens when $A - B < 0$?
  - Negative numbers?
  - Borrow out?

Number Systems

- Desirable properties:
  - Efficient encoding ($2^n$ bit patterns. How many numbers?)
  - Positive and negative
    » Closure (almost) under addition and subtraction
      - Except when overflow
    » Representation of positive numbers same in most systems
    » Major differences are in how negative numbers are represented
  - Efficient operations
    » Comparison: $=, <, >$
    » Addition, Subtraction
    » Detection of overflow
  - Algebraic properties?
    » Closure under negation?
    » $A = B$ iff $A - B = 0$

- Three Major schemes:
  - sign and magnitude
  - ones complement
  - twos complement (excess notation)

Which one did you learn in 2nd grade?

Sign and Magnitude

- High order bit is sign: 0 = positive (or zero), 1 = negative
- Remaining low order bits is the magnitude: 0 (000) thru 7 (111)
- Number range for $n$ bits = $\pm 2^{n-1} - 1$

Operations: $=, <, >, +, -$ ???

Ones Complement

- Bit manipulation:
  simply complement each of the bits
  $0111 \rightarrow 1000$

Algebraically ...

$N$ is positive number, then $\overline{N}$ is its negative 1's complement

$\overline{N} = (2^n - 1) - N$

Example: 1's complement of 7

- $-7 \rightarrow -0111$
- $-7 \rightarrow \overline{0001}$
- $-7 \rightarrow 1111$

-7 in 1's comp.
Ones Complement on the number wheel

Subtraction implemented by addition & 1's complement
- Closure under negation: If A can be represented, so can -A
- Still two representations of 0!
- If A = B then is A - B == 0?
- Addition is almost clockwise advance, like unsigned

Twos Complement number wheel

Easy to determine sign (0?)
- Only one representation for 0
- Addition and subtraction just as in unsigned case
- Simple comparison: A < B iff A - B < 0
- One more negative number than positive number
- One number has no additive inverse

Twos Complement (algebraically)

\[ N^* = 2^n - N \]

Example: Twos complement of 7
\[ 2^4 = 10000 \]
sub 7 = 0111
1001 = repr. of -7

Example: Twos complement of -7
\[ 2^4 = 10000 \]
sub -7 = 1001
0111 = repr. of 7

Bit manipulation:
- Twos complement: take bitwise complement and add one
  - 0111 -> 1000 + 1 -> 1001 (representation of -7)
  - 1001 -> 0110 + 1 -> 0111 (representation of 7)

How is addition performed in each number system?

- Operands may be positive or negative
Sign Magnitude Addition

Operand have same sign: unsigned addition of magnitudes

Result sign bit is the same as the operands' sign

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>-4</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>0011</td>
<td>+(-3)</td>
<td>1011</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1111</td>
</tr>
</tbody>
</table>

Operand have different signs:
subtract smaller from larger and keep sign of the larger

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>-4</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>1011</td>
<td>+3</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1001</td>
</tr>
</tbody>
</table>

Ones complement addition

Perform unsigned addition, then add in the end-around carry

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>-4</th>
<th>1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>0011</td>
<td>+(-3)</td>
<td>1100</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1011</td>
</tr>
</tbody>
</table>

End around carry

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>-4</th>
<th>1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>1100</td>
<td>+3</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>-1</td>
<td>1110</td>
</tr>
</tbody>
</table>

End around carry

|   | 0001 |

Why does end-around carry work?

Recall: \( \overline{N} = (2^n - 1) - N \)

End-around carry work is equivalent to subtracting \( 2^n \) and adding 1

\[
M - N = M + \overline{N} = M + (2^n - 1 - N) = (M - N) + 2^n - 1 \quad \text{(when } M > N)\
\]

\[-M + (-N) = \overline{M} + \overline{N} = (2^n - M - 1) + (2^n - N - 1) = 2^n + [2^n - 1 - (M + N)] - 1 \quad \text{if } M + N < 2^{n-1}\
\]

after end around carry:

\[
= 2^n - 1 - (M + N)\
\]

this is the correct form for representing \(-(M + N)\) in 1's complement.
Twos Complement Addition

Perform unsigned addition and

\[ \begin{align*}
4 & \quad 0100 \\
+3 & \quad 0011 \\
\hline
7 & \quad 1101
\end{align*} \]

Discard the carry out.

\[ \begin{align*}
7 & \quad 0111 \\
-7 & \quad 1111 \\
\hline
0 & \quad 0000
\end{align*} \]

Overflow?

\[ \begin{align*}
4 & \quad 0100 \\
-3 & \quad 1101 \\
\hline
1 & \quad 10001
\end{align*} \]

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems.

2s Comp: ignore the carry out

\(-M + N\) when \(N > M\):

\[ M^* + N = (2^n - M) + N = 2^n + (N - M) \]

Ignoring carry-out is just like subtracting \(2^n\)

\(-M + -N\) where \(N + M \leq 2^{n-1}\)

\[-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2^n \]

After ignoring the carry, this is just the right twos compl. representation for \(-(M + N)!\)

2s Complement Overflow

How can you tell an overflow occurred?

Add two positive numbers to get a negative number

or two negative numbers to get a positive number
### 2s comp. Overflow Detection

<table>
<thead>
<tr>
<th>5</th>
<th>0111</th>
<th>-7</th>
<th>1000</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011</td>
<td>-2</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td>1000</td>
<td>7</td>
<td>1011</td>
<td>1111</td>
</tr>
</tbody>
</table>

**Overflow**

<table>
<thead>
<tr>
<th>5</th>
<th>0000</th>
<th>-3</th>
<th>1111</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0010</td>
<td>-5</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-8</td>
<td>1100</td>
<td></td>
</tr>
</tbody>
</table>

**No overflow**

Overflow occurs when carry in to sign does not equal carry out

---

### 2s Complement Adder/Subtractor

\[
A - B = A + (-B) = A + \bar{B} + 1
\]

---

### Summary

- **Circuit design for unsigned addition**
  - Full adder per bit slice
  - Delay limited by Carry Propagation
    - Ripple is algorithmically slow, but wires are short
- **Carry select**
  - Simple, resource-intensive
  - Excellent layout
- **Carry look-ahead**
  - Excellent asymptotic behavior
  - Great at the board level, but wire length effects are significant on chip
- **Digital number systems**
  - How to represent negative numbers
  - Simple operations
  - Clean algorithmic properties
- **2s complement is most widely used**
  - Circuit for unsigned arithmetic
  - Subtract by complement and carry in
  - Overflow when cin xor cout of sign-bit is 1