Basic Arithmetic and the ALU

- Number representations: 2’s complement, unsigned
- Addition/Subtraction
- Add/Sub ALU
  - Full adder, ripple carry, subtraction
  - Carry-lookahead addition
- Logical operations
  - and, or, xor, nor, shifts
- Overflow

Basic Arithmetic and the ALU

- Covered later in the semester:
  - Integer multiplication, division
  - Floating point arithmetic
- These are not crucial for the project

Background

- Recall
  - n bits enables $2^n$ unique combinations
- Notation: $b_{31} b_{30} \ldots b_1 b_0$
- No inherent meaning
  - $f(b_{31} \ldots b_0) \Rightarrow$ integer value
  - $f(b_{31} \ldots b_0) \Rightarrow$ control signals

Unsigned Integers

- $f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$
- Treat as normal binary number
  - E.g. 0...01101010101
    - $= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    - $= 128 + 64 + 16 + 4 + 1 = 213$
- Max $f(111 \ldots 11) = 2^{32} - 1 = 4,294,967,295$
- Min $f(000 \ldots 00) = 0$
- Range $[0, 2^{32} - 1]$ => # values $(2^{32} - 1) - 0 + 1 = 2^{32}$
Signed Integers

- 2’s complement
  \[ f(b_{31} \ldots b_0) = -b_{31} \times 2^{31} + \ldots b_1 \times 2^1 + b_0 \times 2^0 \]
- Max \( f(0111 \ldots 11) = 2^{31} - 1 = 2147483647 \)
- Min \( f(100 \ldots 00) = -2^{31} = -2147483648 \) (asymmetric)
- Range \([-2^{31}, 2^{31}-1]\) => # values \(2^{32}\)
- Invert bits and add one: e.g. \(-6\)
  
  \[ -000\ldots0110 => 111\ldots1001 + 1 => 111\ldots1010 \]

Why 2’s Complement

- Why not use sign-magnitude?
- 2’s complement makes hardware simpler
- Just like humans don’t work with Roman numerals
- Representation affects ease of calculation, not correctness of answer

Addition and Subtraction

- 4-bit unsigned example

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
<th>c_out</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- 4-bit 2’s complement – ignoring overflow

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
<th>c_out</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Subtraction

- \( A - B = A + 2\’s \) complement of \( B \)
- E.g. \( 3 - 2 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c_in</th>
<th>s</th>
<th>c_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Full Adder

- Full adder \((a,b,c_in) => (c_out, s)\)
- \(c_in\) = two or more of \((a, b, c_in)\)
- \(s\) = exactly one or three of \((a,b,c_in)\)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c_in</th>
<th>c_out</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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</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>

Ripple-carry Adder

- Just concatenate the full adders

<table>
<thead>
<tr>
<th>a_0</th>
<th>a_1</th>
<th>b_0</th>
<th>b_1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ripple-carry Subtractor
- $A - B = A + (-B) \Rightarrow$ invert $B$ and set $c_{in}$ to 1

```
\hspace{2cm}
\begin{array}{cccc}
\text{Full Adder} & \text{Full Adder} & \text{Full Adder} & \text{Full Adder} \\
\hspace{1cm} a_0 b_0 & \hspace{1cm} a_1 b_1 & \hspace{1cm} a_2 b_2 & \hspace{1cm} a_3 b_3 \\
\end{array}
```

Combined Ripple-carry Adder/Subtractor
- Control = 1 $\Rightarrow$ subtract
- XOR $B$ with control and set $c_{in}$ to control

```
\hspace{2cm}
\begin{array}{cccc}
\text{Full Adder} & \text{Full Adder} & \text{Full Adder} & \text{Full Adder} \\
\hspace{1cm} a_0 b_0 & \hspace{1cm} a_1 b_1 & \hspace{1cm} a_2 b_2 & \hspace{1cm} a_3 b_3 \\
\end{array}
```

Carry Lookahead
- The above ALU is too slow
  - Gate delays for add = $32 \times FA + XOR \approx 64$
- Theoretically, in parallel
  - $S_{sum0} = f(c_{in}, a_0, b_0)$
  - $S_{sum} = f(c_{in}, a_0...a_2, b_0...b_0)$
  - $S_{sum31} = f(c_{in}, a_0...a_31, b_0...b_0)$
- Any boolean function in two levels, right?
  - Wrong! Too much fan-in!

```
\hspace{2cm}
\begin{array}{cccc}
\text{Full Adder} & \text{Full Adder} & \text{Full Adder} & \text{Full Adder} \\
\hspace{1cm} a_0 b_0 & \hspace{1cm} a_1 b_1 & \hspace{1cm} a_2 b_2 & \hspace{1cm} a_3 b_3 \\
\end{array}
```

Carry Lookahead
- Need compromise
  - Build tree so delay is $O(\log_2 n)$ for $n$ bits
  - E.g. $2 \times 5$ gate delays for 32 bits
- We will consider basic concept with
  - 4 bits
  - 16 bits
- Warning: a little convoluted

```
\hspace{2cm}
\begin{array}{cccc}
\text{Full Adder} & \text{Full Adder} & \text{Full Adder} & \text{Full Adder} \\
\hspace{1cm} a_0 b_0 & \hspace{1cm} a_1 b_1 & \hspace{1cm} a_2 b_2 & \hspace{1cm} a_3 b_3 \\
\end{array}
```

Carry Lookahead
- Define: $g_i = a_i \times b_i$ \# carry generate
- $p_i = a_i + b_i$ \# carry propagate
- Recall: $c_{i+1} = a_i \times b_i + a_i \times c_i + b_i \times c_i$
  - $= a_i \times (b_i + (a_i + b_i)) \times c_i$
  - $= g_i + p_i \times c_i$

```
\hspace{2cm}
\begin{array}{cccc}
\text{Full Adder} & \text{Full Adder} & \text{Full Adder} & \text{Full Adder} \\
\hspace{1cm} a_0 b_0 & \hspace{1cm} a_1 b_1 & \hspace{1cm} a_2 b_2 & \hspace{1cm} a_3 b_3 \\
\end{array}
```

Carry Lookahead
- Therefore
- $c_1 = g_0 + p_0 \times c_0$
- $c_2 = g_1 + p_1 \times c_1$
- $c_3 = g_2 + p_2 \times g_1 + p_2 \times p_1 \times c_1$
- $c_4 = g_3 + p_3 \times g_2 + p_3 \times p_2 \times g_1 + p_3 \times p_2 \times p_1 \times g_0 + p_3 \times p_2 \times p_1 \times p_0 \times c_0$
- Uses one level to form $p_i$ and $g_i$, two levels for carry
- But, this needs $n+1$ fanin at the OR and the rightmost AND
Hierarchical Carry Lookahead Basics

Fill in the holes in the G’s and P’s

\[ G_i = G_{i+1} + P_{i+1} \cdot G_i \] (assume \( i < j + 1 < k \))

\[ P_{i,k} = P_{i,j} \cdot P_{j+1,k} \]

\[ G_{12,15} = G_{12,11} + P_{12,11} \cdot G_{8,11} \]

\[ P_{12,15} = P_{8,15} \cdot P_{12,15} \]

CLA: Compute G’s and P’s

CLA: Compute Carries

Hierarchical CLA for 16 bits

Build 16-bit adder from four 4-bit adders

Figure out G and P for 4 bits together

\[ G_{0,3} = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \]

\[ P_{0,3} = p_3 \cdot p_2 \cdot p_1 \cdot p_0 \] (Notation a little different from the book)

\[ G_{4,7} = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 \]

\[ P_{4,7} = p_7 \cdot p_6 \cdot p_5 \cdot p_4 \]

\[ G_{8,15} = g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12} \]

\[ P_{8,15} = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12} \]
Other Adders: Carry Select
• Two adds in parallel; with and without cin
  – When Cin is done, select correct result

Other Adders: Carry Save
A + B => S
Save carries A + B => S, C_{out}
Use Cin, A + B + C => S1, S2 (3# to 2# in parallel)
Used in combinational multipliers by building a
Wallace Tree

Adding Up Many Bits

Logical Operations
• Bitwise AND, OR, XOR, NOR
  – Implement w/ 32 gates in parallel
• Shifts and rotates
  – rol => rotate left (MSB->LSB)
  – ror => rotate right (LSB->MSB)
  – sll => shift left logical (0->LSB)
  – srl => shift right logical (0->LSB)
  – sra => shift right arithmetic (old MSB->new MSB)

Shifter
E.g., Shift left logical for d<7:0> and shamt<2:0>
Using 2-1 Muxes called Mux(select, in0, in1)
stage0<7:0> = Mux(shamt<0>, d<7:0>, 0 || d<7:1>)
stage1<7:0> = Mux(shamt<1>, stage0<7:0>, 00 || stage0<6:2>)
dout<7:0) = Mux(shamt<2>, stage1<7:0>, 0000 || stage1<3:0>)
For Barrel shifter used wider muxes
**All Together**

![Diagram](image)

**Overflow**

- With n bits only $2^n$ combinations
- Unsigned $[0, 2^n-1]$, 2's complement $[-2^{n-1}, 2^{n-1}-1]$
- Unsigned Add
  
  \[5 + 6 > 7; 101 + 110 \Rightarrow 1011\]
  \[f(3:0) = a(2:0) + b(2:0) \Rightarrow \text{overflow} = f(3)\]
  Carryout from MSB

**Addition Overflow**

- More involved for 2's complement
  
  \[-1 + -1 = -2:\]
  
  \[111 + 111 = 1110\]
  
  \[110 = -2 \text{ is correct}\]
- Can't just use carry-out to signal overflow

**Subtraction Overflow**

- No overflow on a-b if signs are the same
- Neg – pos $\Rightarrow$ neg ;; overflow otherwise
- Pos – neg $\Rightarrow$ pos ;; overflow otherwise

Overflow = \[f(2) * \neg(a2) \neg(b2) + \neg(f(2)) * a(2) * \neg(b2)\]
What to do on Overflow?
- Ignore ! (C language semantics)
  - What about Java? (try/catch?)
- Flag – condition code
- Sticky flag – e.g. for floating point
  - Otherwise gets in the way of fast hardware
- Trap – possibly maskable
  - MIPS has e.g. add that traps, addu that does not

Zero and Negative
- Zero = ~[f(2) + f(1) + f(0)]
- Negative = f(2) (sign bit)

Zero and Negative
- May also want correct answer even on overflow
- Negative = (a < b) = (a – b) < 0 even if overflow
- E.g. is –4 < 2?
  100 – 010 = 1010 (-4 – 2 = -6): Overflow!
- Work it out: negative = f(2) XOR overflow

Summary
- Binary representations, signed/unsigned
- Arithmetic
  - Full adder, ripple-carry, carry lookahead
  - Carry-select, Carry-save
  - Overflow, negative
  - More (multiply/divide/FP) later
- Logical
  - Shift, and, or