Arithmetic

- Where we’ve been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    Instruction Set Architecture
    Assembly Language and Machine Language
- What’s up ahead:
  - Implementing the Architecture

Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: 0...2\(^n-1\)
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?
### Possible Representations

<table>
<thead>
<tr>
<th></th>
<th>Sign Magnitude</th>
<th>One's Complement</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
<td></td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
<td></td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
<td></td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
<td></td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
<td></td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
<td></td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
<td></td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
<td></td>
</tr>
</tbody>
</table>

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

### MIPS

- 32 bit signed numbers:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000 = 0_{ten}</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0001 = +1_{ten}</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0010 = +2_{ten}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1110 = +2,147,483,646_{ten}</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111 = +2,147,483,647_{ten}</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0010 = -2,147,483,646_{ten}</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0001 = -2,147,483,647_{ten}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1101 = -3_{ten}</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1110 = -2_{ten}</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111 = -1_{ten}</td>
<td></td>
</tr>
</tbody>
</table>

\(\text{maxint}\)
\(\text{minint}\)
Negating a two’s complement number: invert all bits and add 1
– remember: “negate” and “invert” are quite different!

Converting n bit numbers into numbers with more than n bits:
– MIPS 16 bit immediate gets converted to 32 bits for arithmetic
– copy the most significant bit (the sign bit) into the other bits
 0010  \rightarrow  0000 0010
 1010  \rightarrow  1111 1010
– "sign extension" (lbu vs. lb)

Two’s Complement Operations

Just like in grade school (carry/borrow 1s)

\[
\begin{array}{ccc}
0111 & 0111 & 0110 \\
+ 0110 & - 0110 & - 0101
\end{array}
\]

Two’s complement operations easy
– subtraction using addition of negative numbers

\[
\begin{array}{c}
0111 \\
+ 1010
\end{array}
\]

Overflow (result too large for finite computer word):
– e.g., adding two n-bit numbers does not yield an n-bit number

\[
\begin{array}{c}
0111 \\
+ 0001 \\
1000
\end{array}
\]

note that overflow term is somewhat misleading,
it does not mean a carry “overflowed”
Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations $A + B$, and $A - B$
  - Can overflow occur if $B$ is 0?
  - Can overflow occur if $A$ is 0?

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don’t always want to detect overflow
  - new MIPS instructions: `addu`, `addiu`, `subu`

  note: `addiu` still sign-extends!
  note: `sltu`, `sltiu` for unsigned comparisons
Review: Boolean Algebra & Gates

- Problem: Consider a logic function with three inputs: A, B, and C.
  
  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

- Show the truth table for these three functions.
- Show the Boolean equations for these three functions.
- Show an implementation consisting of inverters, AND, and OR gates.

An ALU (arithmetic logic unit)

- Let's build an ALU to support the andi and ori instructions
  - we'll just build a 1 bit ALU, and use 32 of them

- Possible Implementation (sum-of-products):
Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

```
\begin{center}
\begin{tikzpicture}
  \node at (0,0) {S};
  \node at (-1,0) {A};
  \node at (1,0) {B};
  \node at (0,-1) {C};
  \draw (A) -- (0,0.5);
  \draw (B) -- (0,-0.5);
  \draw (0,0) -- (A) -- (0,0.5);
  \draw (0,0) -- (B) -- (0,-0.5);
\end{tikzpicture}
\end{center}
```

note: we call this a 2-input mux even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

- Not easy to decide the “best” way to build something
  - Don’t want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - For our purposes, ease of comprehension is important
- Let’s look at a 1-bit ALU for addition:

```
c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}
sum = a \oplus b \oplus c_{in}
```

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?
Arithmetic Logic Unit (ALU)

ALU operation (2-bit):

00 = and
01 = or
10 = add
What about subtraction \((a - b)\) ?

- Two's complement approach: just negate \(b\) and add.
- How do we negate?

**ALU operation (3-bit):**

<table>
<thead>
<tr>
<th>Binvert</th>
<th>Operation</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>and</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>or</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>sub</td>
</tr>
</tbody>
</table>