Chapter 3
Arithmetic for Computers
Arithmetic

• Where we've been:
  – Abstractions:
    Instruction Set Architecture
    Assembly Language and Machine Language
• What's up ahead:
  – Implementing the Architecture
• 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>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>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
</tr>
</tbody>
</table>

- **Issues:** balance, number of zeros, ease of operations
32 bit signed numbers:

0000 0000 0000 0000 0000 0000 0000 0000\_\text{two} = 0\_\text{ten}
0000 0000 0000 0000 0000 0000 0000 0001\_\text{two} = + 1\_\text{ten}
0000 0000 0000 0000 0000 0000 0000 0010\_\text{two} = + 2\_\text{ten}
\ldots
0111 1111 1111 1111 1111 1111 1111 1110\_\text{two} = + 2,147,483,646\_\text{ten}
0111 1111 1111 1111 1111 1111 1111 1111\_\text{two} = + 2,147,483,647\_\text{ten}
1000 0000 0000 0000 0000 0000 0000 0000\_\text{two} = - 2,147,483,648\_\text{ten}
1000 0000 0000 0000 0000 0000 0000 0001\_\text{two} = - 2,147,483,647\_\text{ten}
1000 0000 0000 0000 0000 0000 0000 0010\_\text{two} = - 2,147,483,646\_\text{ten}
\ldots
1111 1111 1111 1111 1111 1111 1111 1101\_\text{two} = - 3\_\text{ten}
1111 1111 1111 1111 1111 1111 1111 1110\_\text{two} = - 2\_\text{ten}
1111 1111 1111 1111 1111 1111 1111 1111\_\text{two} = - 1\_\text{ten}
Two's Complement Operations

• 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)
2’s complement

16-bit binary

Negate number
(invert & add 1)

0000 0000 0000 0010\textsuperscript{two}

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
+ \quad \begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

1111 1111 1111 1110\textsuperscript{two}
2’s complement

\[
\begin{array}{c}
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0010_{\text{two}} \\
1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1101_{\text{two}} \\
\hline
1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_{\text{two}}
\end{array}
\]
Sign extension

16-bit

32-bit

16-bit

32-bit
## Two’s complement

<table>
<thead>
<tr>
<th>Number</th>
<th>binary</th>
<th>2’s complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000 0101</td>
<td>1111 1011</td>
</tr>
<tr>
<td>12</td>
<td>0000 1100</td>
<td>1111 0100</td>
</tr>
<tr>
<td>16</td>
<td>0001 0000</td>
<td>1111 0000</td>
</tr>
<tr>
<td>37</td>
<td>0010 0101</td>
<td>1101 1011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Addition & Subtraction

- 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
    
    \[
    \begin{array}{c}
    0111 \\
    + 0001 \text{ \quad note that overflow term is somewhat misleading,} \\
    \cline{3-4}
    1000 \text{ \quad it does not mean a carry “overflowed”} \\
    \end{array}
    \]
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
## Overflow

<table>
<thead>
<tr>
<th>Operation</th>
<th>A</th>
<th>B</th>
<th>Result indicating overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B$</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$&lt; 0$</td>
<td>$&lt; 0$</td>
<td>$\geq 0$</td>
</tr>
<tr>
<td>$A - B$</td>
<td>$\geq 0$</td>
<td>$&lt; 0$</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$A - B$</td>
<td>$&lt; 0$</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
</tr>
</tbody>
</table>
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
• 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):

```
operation

<table>
<thead>
<tr>
<th>op</th>
<th>a</th>
<th>b</th>
<th>res</th>
</tr>
</thead>
</table>
```

```
```

```
```

```
```
Review: The Multiplexor

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

\[ S \]

\[ A \quad 0 \quad B \quad 1 \quad C \]

*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:

  \[ \text{cout} = a \cdot b + a \cdot \text{cin} + b \cdot \text{cin} \]
  \[ \text{sum} = a \oplus b \oplus \text{cin} \]

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?
Building a 32 bit ALU

![ALU Diagram]
What about subtraction \((a - b)\) ?

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

- A very clever solution:
Tailoring the ALU to the MIPS

• Need to support the set-on-less-than instruction (slt)
  – remember: slt is an arithmetic instruction
  – produces a 1 if rs < rt and 0 otherwise
  – use subtraction: \((a-b) < 0\) implies \(a < b\)
• Need to support test for equality (beq $t5, $t6, $t7)
  – use subtraction: \((a-b) = 0\) implies \(a = b\)
Supporting slt

- Can we figure out the idea?
Test for equality

- Notice control lines:
  - 000 = and
  - 001 = or
  - 010 = add
  - 110 = subtract
  - 111 = slt

- Note: zero is a 1 when the result is zero!
Conclusion

• We can build an ALU to support the MIPS instruction set
  – key idea: use multiplexor to select the output we want
  – we can efficiently perform subtraction using two’s complement
  – we can replicate a 1-bit ALU to produce a 32-bit ALU

• Important points about hardware
  – all of the gates are always working
  – the speed of a gate is affected by the number of inputs to the gate
  – the speed of a circuit is affected by the number of gates in series
    (on the “critical path” or the “deepest level of logic”)

• Our primary focus: comprehension, however,
  – Clever changes to organization can improve performance
    (similar to using better algorithms in software)
  – we’ll look at two examples for addition and multiplication
Problem: ripple carry adder is slow

• Is a 32-bit ALU as fast as a 1-bit ALU?
• Is there more than one way to do addition?
  – two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[
\begin{align*}
c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
c_4 &= b_3c_3 + a_3c_3 + a_3b_3
\end{align*}
\]

Not feasible! Why?

\[
c_2 = b_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1b_1
\]
Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? \( g_i = a_i b_i \)
  - When would we propagate the carry? \( p_i = a_i + b_i \)
- Did we get rid of the ripple?

\[
\begin{align*}
c_1 &= g_0 + p_0 c_0 \\
c_2 &= g_1 + p_1 c_1 \\
c_3 &= g_2 + p_2 c_2 \\
c_4 &= g_3 + p_3 c_3 \\
\end{align*}
\]

Feasible! Why?
Use principle to build bigger adders

- Can’t build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!
Multiplication

• More complicated than addition
  – accomplished via shifting and addition
• More time and more area
• Let's look at 3 versions based on grade school algorithm

\[
\begin{array}{c}
0010 \quad \text{(multiplicand)} \\
\underline{\times \ 1011} \quad \text{(multiplier)}
\end{array}
\]

• Negative numbers: convert and multiply
  – there are better techniques, we won’t look at them
Multiplication: Implementation

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to product and place the result in Product register
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit

32nd repetition?
- No: < 32 repetitions
- Yes: 32 repetitions

Done
1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
   - Multiplier0 = 0

2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit

32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

Done
Control
test
Write
32 bits
64 bits
Shift right
Product
Multiplicand
32-bit ALU

Start

1. Test Product0
   - Product0 = 1
   - Product0 = 0

   1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

Done