Number Systems and Arithmetic

or

Computers go to elementary school

Questions About Numbers

• How do you represent
  – negative numbers?
  – fractions?
  – really large numbers?
  – really small numbers?

• How do you
  – do arithmetic?
  – identify errors (e.g. overflow)?

• What is an ALU and what does it look like?
  – ALU = arithmetic logic unit

Introduction to Binary Numbers

Consider a 4-bit binary number

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Examples of binary arithmetic:

\[
\begin{array}{c}
3 + 2 = 5 \\
\hline
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 0 \\
\hline
& & 1 & 1 & 1
\end{array}
\]

\[
\begin{array}{c}
3 + 3 = 6 \\
\hline
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 1 \\
\hline
& & 1 & 1 & 1
\end{array}
\]

Negative Numbers?

• We would like a number system that provides
  – obvious representation of 0, 1, 2...
  – uses adder for addition
  – single value of 0
  – equal coverage of positive and negative numbers
  – easy detection of sign
  – easy negation
Some Alternatives

• Sign Magnitude -- MSB is sign bit, rest the same
  -1 == 1001
  -5 == 1101

• One’s complement -- flip all bits to negate
  -1 == 1110
  -5 == 1010

Two’s Complement Representation

• 2’s complement representation of negative numbers
  – Take the bitwise inverse and add 1
• Biggest 4-bit Binary Number: 7 Smallest 4-bit Binary Number: -8

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>1000</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Two’s Complement Arithmetic

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement Binary</th>
<th>Decimal</th>
<th>2’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

• Examples: 7 - 6 = 7 + (-6) = 1

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement Binary</th>
<th>Decimal</th>
<th>2’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111</td>
<td>3</td>
<td>1011</td>
</tr>
</tbody>
</table>

Some Things We Want To Know About Our Number System

• negation
• sign extension
  – +3 => 0011, 00000011, 0000000000000011
  – -3 => 1101, 11111101, 1111111111111101
• overflow detection

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>+ 0110</td>
<td>6</td>
</tr>
</tbody>
</table>
Overflow Detection

So how do we detect overflow?

Arithmetic -- The heart of instruction execution

Designing an Arithmetic Logic Unit

A One Bit ALU

• This 1-bit ALU will perform AND, OR, and ADD
A 32-bit ALU

1-bit ALU

32-bit ALU

Result

Operation

CarryIn

CarryOut

How About Subtraction?

• Keep in mind the following:
  – (A - B) is the same as: A + (-B)
  – 2’s Complement negate: Take the inverse of every bit and add 1

• Bit-wise inverse of B is !B:
  – A - B = A + (-B) = A + (!B + 1) = A + !B + 1

Overflow Detection Logic

• Carry into MSB ! = Carry out of MSB
  – For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]

Zero Detection Logic

• Zero Detection Logic is just one BIG NOR gate
  – Any non-zero input to the NOR gate will cause its output to be zero
Set-on-less-than

- Do a subtract
- use sign bit
  - route to bit 0 of result
  - all other bits zero

The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an N-bit RC adder: 2N-gate delay

MULTIPLY

- Paper and pencil example:
  Multicand \( \times 1011 \)
  \[ \text{Product} = ? \]

- \( m \) bits \( \times \) \( n \) bits = \( m+n \) bit product
- Binary makes it easy:
  - 0 => place 0 ( 0 \( \times \) multiplicand)
  - 1 => place multiplicand ( 1 \( \times \) multiplicand)
- we’ll look at a couple of versions of multiplication hardware
**MULTIPLY HARDWARE**

**Version 1**

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

**Multiply Algorithm**

**Version 1**

1. Test Multiplier
   - Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
4. 32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

**Observations on Multiply**

**Version 1**

- 1 clock per cycle => 100 clocks per multiply
  - Ratio of multiply to add 100:1
- 1/2 bits in multiplicand always 0
  - => 64-bit adder is wasted
- 0’s inserted in left of multiplicand as shifted
  - => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Wasted space (zeroes) in product register exactly matches meaningful bits of multiplier at all times. Combine?

**MULTIPLY HARDWARE**

**Version 2**

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)
Observations on Multiply
Version 2

- 2 steps per bit because Multiplier & Product combined
- 32-bit adder
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done.

Divide: Paper & Pencil

Quotient
Divisor 1000 1101010 Dividend

Remainder
- See how big a number can be subtracted, creating quotient bit on each step
  - Binary => 1 * divisor or 0 * divisor
- Dividend = Quotient x Divisor + Remainder

DIVIDE HARDWARE
Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

Divide Algorithm
Version 1

- Takes n+1 steps for n-bit Quotient & Rem.

Start
1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

Remainder
- Test Remainder
  - Remainder >= 0
  - Remainder < 0

2a. Shift the Quotient register to the left setting the new rightmost bit to 1.
2b. Restore the original value by adding the Divisor register to the Remainder register, and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

3. Shift the Divisor register right 1 bit.

3rd repetition?
- No: 33 repetitions
- Yes: 33 repetitions

Done
Divide Hardware Version 1

- Again, 64-bit adder is unnecessary.
- Quotient grows as remainder shrinks

Observations on Divide

Version 2

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree

Key Points

- Instruction Set drives the ALU design
- ALU performance, CPU clock speed driven by adder delay
- Multiplication and division take much longer than addition, requiring multiple addition steps.