Number Systems and Arithmetic

or

Computers go to elementary school

What do all those bits mean now?

bits (011011011100010 ....01)

instruction

R-format I-format...

number
data

text chars

integer

floating point

signed

unsigned

single precision
double precision

Questions About Numbers

• How do you represent
  –
  –
  –
  –

• How do you
  –
  –

• What is an ALU and what does it look like?
  –

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:

3 + 2 = 5

0011
+ 0010
1101

3 + 3 = 6

0011
+ 0011
1110
Negative Numbers?

- We would like a number system that provides
  - uses adder for addition
  - equal coverage of positive and negative numbers
  - 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

- Examples: 7 - 6 = 7 + (-6) = 1
- 3 - 5 = 3 + (-5) = -2

<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>
Some Things We Want To Know About Our Number System

- negation
- sign extension
  - \(+3\) \(\Rightarrow\) 0011, 00000011, 0000000000000011
  - \(-3\) \(\Rightarrow\) 1101, 11111101, 1111111111111101
- overflow detection
  0101 5
  \(+\) 0110 6
  0111 0
  0011 \(+\) 1010 1
  11001011 \(+\) 0111 1
  11001110 \(+\) 1010 1

Overflow Detection

So how do we detect overflow?

Arithmetic -- The heart of instruction execution

Designing an Arithmetic Logic Unit

- ALU Control Lines (ALUop) Function
  - 000 And
  - 001 Or
  - 010 Add
  - 110 Subtract
  - 111 Set-on-less-than
A One Bit ALU

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

\[
\begin{array}{c}
\text{CarryIn} \\
\text{Result} \\
\text{CarryOut}
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
1
\end{array}
\]

A One-bit Full Adder

- This is also called a (3, 2) adder
- Half Adder: No CarryIn
- Truth Table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logic Equation for CarryOut

\[
\text{CarryOut} = \left( \neg A \land B \land \text{CarryIn} \right) \lor \left( A \land \neg B \land \text{CarryIn} \right) \lor \left( A \land B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \text{CarryIn} \right)
\]

Logic Equation for Sum

\[
\text{Sum} = \left( \neg A \land \neg B \land \text{CarryIn} \right) \lor \left( A \land \neg B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \text{CarryIn} \right)
\]

\[
\text{CarryOut} = B \land \text{CarryIn} \lor A \land \text{CarryIn} \lor A \land B
\]

\[
\text{CarryOut} = \left( \neg A \land B \land \text{CarryIn} \right) \lor \left( A \land \neg B \land \text{CarryIn} \right) \lor \left( A \land B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \text{CarryIn} \right)
\]

\[
\text{Sum} = \left( \neg A \land \neg B \land \text{CarryIn} \right) \lor \left( A \land \neg B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \text{CarryIn} \right)
\]

\[
\text{CarryOut} = B \land \text{CarryIn} \land A \land \text{CarryIn} \land A \land B
\]

\[
\text{CarryOut} = \left( \neg A \land B \land \text{CarryIn} \right) \lor \left( A \land \neg B \land \text{CarryIn} \right) \lor \left( A \land B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \text{CarryIn} \right)
\]

\[
\text{Sum} = \left( \neg A \land \neg B \land \text{CarryIn} \right) \lor \left( A \land \neg B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \neg \text{CarryIn} \right) \lor \left( A \land B \land \text{CarryIn} \right)
\]

\[
\text{CarryOut} = B \land \text{CarryIn} \land A \land \text{CarryIn} \land A \land B
\]
A 32-bit ALU

1-bit ALU

32-bit ALU

Result

Operation

CarryIn

CarryOut

A 32-bit ALU

1-bit ALU

32-bit ALU

Result

Operation

CarryIn

CarryOut

A 32-bit ALU

1-bit ALU

32-bit ALU

Result

Operation

CarryIn

CarryOut

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

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 \(!B\)
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

Problem: ripple carry adder is slow

- 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?

\[
\begin{align*}
c_1 &= b_1c_0 + a_0c_0 + a_0b_0 \\
c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
c_4 &= b_3c_3 + a_3c_3 + a_3b_3
\end{align*}
\]

Not feasible! Why?
Carry Lookahead Adders

- We'll define two new terms, based on the relationship between Cin and Cout
  - **Generate** Carry at Bit i: \( g_i = A_i \land B_i \)
  - **Propagate** Carry via Bit i: \( p_i = A_i \lor B_i \)

Carry Lookahead (Continued)

- Using the two new terms we just defined:
  - Generate Carry at Bit i: \( g_i = A_i \land B_i \)
  - Propagate Carry via Bit i: \( p_i = A_i \lor B_i \)
- We can rewrite:
  - \( Cin_{1} = g_0 \mid (p_0 \land Cin_0) \)
  - \( Cin_{2} = g_1 \mid (p_1 \land g_0) \mid (p_1 \land p_0 \land Cin_0) \)
  - \( Cin_{3} = g_2 \mid (p_2 \land g_1) \mid (p_2 \land p_1 \land g_0) \mid (p_2 \land p_1 \land p_0 \land Cin_0) \)
- Carry going into bit 3 is 1 if:
  - We generate a carry at bit 2 (\( g_2 \))
  - Or we generate a carry at bit 1 (\( g_1 \)) and bit 2 allows it to propagate (\( p_2 \land g_1 \))
  - Or we generate a carry at bit 0 (\( g_0 \)) and bit 1 as well as bit 2 allows it to propagate (\( p_2 \land p_1 \land g_0 \))
  - Or we have a carry input at bit 0 (\( Cin_0 \)) and bit 0, 1, and 2 all allow it to propagate (\( p_2 \land p_1 \land p_0 \land Cin_0 \))

A Partial Carry Lookahead Adder

- It is very expensive to build a “full” carry lookahead adder
  - Just imagine the length of the equation for \( Cin_{31} \)
- Common practices:
  - Connect several N-bit Lookahead Adders to form a big adder
  - Example: connect four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder

Hierarchical Carry-Lookahead Adders

- \( P_0 = p_0 \land p_1 \land p_2 \land p_3 \)
- Worst-case delay??
MULTIPLY

• Paper and pencil example:
  Multiplicand  1000
  Multiplier x 1001
  1000
  0000
  0000
  1000
  Product  1001000

• m bits x n bits = m+n bit product
• Binary makes it easy:
  – 0 => place 0 (0 x multiplicand)
  – 1 => place multiplicand (1 x 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
   a. 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
   - 32nd repetition?

Started
Multiplier0 = 0
Multiplier0 = 1

No: < 32 repetitions
Yes: 32 repetitions

Done

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 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 
  (9-bit Multiplier reg)

Observations on Multiply
Version 3

- 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 (leave out the sign bit, run for 31 steps)

Key Points

- Instruction Set drives the ALU design
- ALU performance, CPU clock speed driven by adder delay
- Multiply is expensive