Simple Arithmetic [Arithm Notes]

- Number representations
- Signed numbers
  - Sign-magnitude, ones and twos complement
- Arithmetic
  - Addition, subtraction, negation, overflow
  - MIPS instructions
- Logic operations
  - MIPS instructions

Number Representations

C language data types:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>signed int</td>
<td>ordinary binary</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>text character</td>
<td>ASCII</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>real number</td>
<td>floating point (cover later)</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>real number</td>
<td>floating point (cover later)</td>
<td></td>
</tr>
</tbody>
</table>
Binary Numbers

Let's look at decimal representation of unsigned integers first

* 40321 string of symbols (digits)
* symbol (digit) is from the set \{0, 1, ..., 9\}
* value of 40321 is \(4 \times 10^4 + 0 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 1 \times 10^0\)

More generally, a string of digits with a base (or radix) \(b = 10\)

\[d_{n-1}d_{n-2}...d_0\] (digit \(d_k\) is from \{0, 1, ..., 9\}, for \(k = 0, 1, ..., n-1\))

represents the value \(d_{n-1} \times b^{n-1} + d_{n-2} \times b^{n-2} + ... + d_0 \times b^0\)

\[= \sum_{k=0}^{n-1} d_k \times b^k\]

Common Representations

<table>
<thead>
<tr>
<th>Representation</th>
<th>Base</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>2</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>Octal</td>
<td>8</td>
<td>{0, 1, 2, 3, 4, 5, 6, 7}</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16</td>
<td>{0, 1, ..., 9, A, B, C, D, E, F} or {0, 1, ..., 9, a, b, c, d, e, f}</td>
</tr>
</tbody>
</table>

Binary is a natural representation of signals and states in an electronic circuit because a signal/state will usually have one of TWO values.

Octal and Hexadecimal are useful because basically they provide shorter representations of binary numbers.

Example: instead of binary 1001010001010001 we can write hexadecimal 9451

Factoid: “0x” prefix is often used to indicate a hexadecimal number.
Simple Conversions

**Octal to binary:** expand each octal digit to a 3 bit string
307 =

**Hexadecimal to binary:** expand each digit to a 4 bit string
0xa3f =

**Binary to octal:** break up binary string to 3-bit chunks, starting from the right
Then convert chunks into octal digits
01 010 110 =

**Binary to hexadecimal:** break up binary string into 4-bit chunks, starting from the right. Then convert chunks into hexadecimal digits
101 0010 1100 0101 =

Binary Addition

Let's do decimal addition first

\[
\begin{array}{cccc}
1 & 9 & 8 & 5 \\
+ & 3 & 0 & 2 & 7 \\
\end{array}
\]

Addition is done by
- Going from right to left
- Adding in a digit position
- With “carries” to the next digit position
  (carry occurs if a digit addition at digit position yields a two digit number)

**Binary Addition:** same thing except digits are bits

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 1 \\
\end{array}
\]

Galen H. Sasaki EE 361 University of Hawaii
ASCII Representation of text characters by a binary string (within a byte)

See page 142 of textbook for ASCII conversion table (Figure 3.15)

Examples:
"a" = 97 (decimal)
"A" = 65 (decimal)

32-127 are mostly text characters

0-31 are used for control such as
10 = line-feed (LF)
13 = carriage-return (CR)
8 = backspace (BS)
4 = end-of-transmission (EOT): used to delimit a byte string
Signed Integer Representation

Binary numbers can represent unsigned integers, but what about signed integers.

Example: +12, -513

Three representations:
- sign-magnitude
- ones complement
- twos complement

Sign-Magnitude

Our usual notation: +12

Sign-magnitude: 0 1100

sign bit: 0 = “+”, and 1 = “-”

binary representation of the magnitude

What are the 5 bit representations (including the sign bit) of the values:

+8, -9, +3, -7
Arithmetic

Addition: \( X + Y \)
Negation: \( -Y \)
Subtraction: \( X - Y \) or \( X + (-Y) \), so we only need to look at addition and negation

Negation for sign-magnitude: easy, flip sign bit

\[
\begin{align*}
00011 & = +3 \\
10011 & = -3
\end{align*}
\]

Addition: more complicated

if \( \text{signs are the same} \) then

magnitude = \(|X| + |Y|\)

sign = sign of \( X \) (or sign of \( Y \))

if \( \text{signs are different} \) then

magnitude = larger magnitude - smaller magnitude

sign = sign of number with larger magnitude

Ones Complement

Representation of positive integer \(+3\): \( 00011 \)

sign bit

magnitude in binary

Representation of negative integer \(-3\): \( 11100 \) complement bits of the ones representation of \(+3\). Note, first bit indicates negative sign.

What value does the following represent?

\[
\begin{align*}
11010 \\
10011
\end{align*}
\]
Twos Complement

Sign-magnitude: negation is easy, but addition is a pain.

We would a representation where addition is easy, like ordinary binary addition.

Twos complement

1. Positive numbers are represented in ordinary binary, with sign bit 0

   Example: 001010 = +6

   sign bit

2. We’ll get to negative numbers in a moment.

3. Suppose we have an m-bit representation. To negate,

   * complement all bits
   * add 1, using ordinary binary addition
   * truncate to m bits (the least significant m bits).

4. Negative numbers

   * ordinary binary representation of magnitude
   * negate

   4-bit representation of -7 0111 (+7)
   1000 (negate)
   + 1
   1001 (-7)

5. Addition operation: for m-bit representation

   * ordinary binary addition
   * truncate to m bits

   0110 (+6)
   + 1101 (-3)
   10011
   0011 (+3)
Why Does This Work?

Adding a number with its negative should give 0.

\[
\begin{array}{c}
(+6) \\
\text{+ (-6)} \\
\hline
0
\end{array}
\quad
\begin{array}{c}
0110 \quad (+6) \\
1010 \quad (-6) \\
\hline
\text{zero}
\end{array}
\]

\[
\begin{array}{c}
0110 \quad (+6) \\
\text{+ 0001} \\
\hline
1001 \\
\text{+ 0001} \\
\hline
0110 \\
\text{+ 0001} \\
\hline
1111 \\
\text{+ 0001} \\
\hline
10000
\end{array}
\]

More generally,

\[
\begin{array}{c}
b_{m-1} \ b_{m-2} \ldots \ b_0 \\
\text{+} \\
\hline
\text{negative of}
\end{array}
\quad
\begin{array}{c}
b_{m-1} \ b_{m-2} \ldots \ b_0 \\
\text{+} \\
\hline
1 \ 1 \ldots \ 1 \\
\text{+ 1} \\
\hline
1 \ 0 \ 0 \ldots \ 0
\end{array}
\]

Galen H. Sasaki  
EE 361 University of Hawaii
Another Interpretation of Twos Complement

Here’s another view of twos complement.

The representation of a negative number $-M$ with $n$ bits is the binary representation of $2^n - M$.

Example: $-M = -9$, $n = 5$

$2^5 - 9 = 32 - 9 = 23$

Binary representation of 23 is 10111

Why Does This Work?

Using the “wheel” to compute $X + Y$

Start at $X$ and go clockwise $Y$ steps to get $X + Y$.

Galen H. Sasaki  
EE 361 University of Hawaii
Why Does This Work?

Using the "wheel" to compute \( X - Y \)

- Start at \( X \)
- Go counter clockwise \( Y \) steps
- \( X - Y \)

Using the "wheel" to compute \( X - Y \)

- Start at \( X \)
- Go clockwise to \( X - Y \) the long way
- How many steps?
  - \( 2^3 \) steps has us circumventing the wheel and back at \( X \).
- We want to stop short by \( Y \) steps, so the number of steps should be \( 2^3 - Y \).
Example

5-bit representations

<table>
<thead>
<tr>
<th></th>
<th>binary</th>
<th>sign-magnitude</th>
<th>ones complement</th>
<th>twos complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Value for the representation

<table>
<thead>
<tr>
<th></th>
<th>binary</th>
<th>sign-magnitude</th>
<th>ones complement</th>
<th>twos complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comparisons

Arithmetic

Sign-magnitude: negation is easy addition is hard
Twos complement: negation is harder addition is easy
Ones complement: nobody uses this too much

Range of numbers with m bits

Sign-magnitude: \{ \pm 0, \pm 1, \pm 2, \ldots, \pm 2^{m-1} - 1 \}
Ones complement: \{ \pm 0, \pm 1, \pm 2, \ldots, \pm 2^{m-1} - 1 \}
Twos complement: \{ \pm 0, \pm 1, \pm 2, \ldots, \pm 2^{m-1} - 1, -2^{m-1} \}

Example: m = 3
Sign magnitude and ones complement: \{-3, -2, -1, 0, 1, 2, 3\}
Twos complement: \{-4, -3, -2, -1, 0, 1, 2, 3\}
Overflow

Overflow occurs when you have m bits, but the number you want to represent needs more.

Example: Can you represent -7 with 3 bits?: {-4, -3, -2, -1, 0, 1, 2, 3}

Overflow occurs after some arithmetic operation

Negation:

Ones complement and sign magnitude: no overflow
Twos complement: for one number, the most negative number.
For example: 1000 (-4) cannot be negated since there is no 4-bit representation of +4

Addition:

Overflow can sometimes occur when the addition creates a number with larger magnitude
Thus, no overflow if the numbers are of different sign.
But overflow can possibly occur if the numbers are of the same sign.

Overflow Exceptions

MIPS processor checks when overflow occurs during its additions and subtractions

If an overflow occurs then the MIPS does something to alert the user that a possible error occurred. This something is called an exception or interrupt.

When is overflow bad? When it occurs from data. Then you want an exception.

\texttt{add, sub, slt, addi, subi, slti} lead to exceptions upon overflow.

When is overflow not bad? When it occurs from addresses. Then you want to ignore overflow.

\texttt{addu, subu, sltu, addiu, subiu, sltiu} ignore overflows.
Logic Operations

### AND

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X \cdot Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### OR

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X + Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Bit-wise operations between two strings of bits of equal length
- Pairs of bits from each bit position are AND’d or OR’d together

### Bit-wise AND

- **X** = 001011
- **Y** = 101101

**X \cdot Y** = 001001

### Bit-wise OR

- **X** = 001011
- **Y** = 101101

**X + Y** = 101111

---

Logic Operation Instructions

- **and $r1, r2, r3$**
  - Similar to add or sub but bit-wise AND operation
  - Of course, overflow doesn’t make sense here.
  - The format is R type

- **andi $r1, r2, constant$**
  - Similar to addi or subi but bit-wise AND operation
  - The format is I type with a 16-bit constant.
  - The constant is zero-extended to 32 bits, which means the first 16 bits are set to zero.

- **or and ori** are similar to **and** and **andi** except with a bit-wise OR.
Applications

and is used to clear bits

Data = 00010011011
Mask = 10110110001 0 => clear and 1 => leave alone
Result = 00010010001

Application: check if the value of a particular bit

Data = 000100b1011
Mask = 00000010000
Result = 000000b0000 Result = 0 if and only if b = 0

or is used to set bits

Data = 00010011011
Mask = 10110110001 1 => set to 1 and 0 => leave alone
Result = 10110111011

Review

• Representation of C language data types
  – Char, unsigned int, and int
  – Binary and twos complement arithmetic
• Logic operations and what they’re used for
• New instructions
  – addiu, subiu, sliu, and, andi, or, ori