CS61C
Negative Numbers and Logical Operations

Lecture 7

February 10, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html
Review 1/2

- **MIPS assembly language instructions mapped to numbers in 3 formats**

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>R</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td>address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Op field determines format

- **Binary => Decimal => Assembly => Symbolic Assembly => C**

- Reverse Engineering or Disassembly
- Its hard to do, therefore people like shipping binary machine language more than assembly or C
Review 2/2

• Programming language model of memory allocation and pointers
  • Allocate in stack vs. heap vs. global areas
  • Arguments passed call by value vs. call by reference
  • Pointer in C is HLL version of machine address
Numbers: Review

° Number Base B => B symbols per digit:
  • Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  • Base 2 (Binary): 0, 1

° Number representation: \(d_{31}d_{30} \ldots d_2d_1d_0\)
  • \(d_{31} \times B^{31} + d_{30} \times B^{30} + \ldots + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0\)
  • One billion (1,000,000,000\_ten\) is
    \[
    \begin{align*}
    &0011 \ 1011 \ 1001 \ 1010 \ 1100 \ 1010 \ 0000 \ 0000 \\
    &= 1 \times 2^{29} + 1 \times 2^{28} + 1 \times 2^{27} + 1 \times 2^{25} + 1 \times 2^{24} + 1 \times 2^{23} + 1 \times 2^{20} \\
    &+ 1 \times 2^{19} + 1 \times 2^{17} + 1 \times 2^{15} + 1 \times 2^{14} + 1 \times 2^{11} + 1 \times 2^9 \\
    &= 536,870,912 + 268,435,456 + 134,217,728 \\
    &+ 33,554,432 + 16,777,216 + 8,388,608 + 1,048,576 \\
    &+ 524,288 + 131,072 + 32,768 + 16,384 + 2,048 + 512
    \end{align*}
    \]
Overview

° What if Numbers too Big?
° How Represent Negative Numbers?
° What if Result doesn’t fit in register?
° More Compact Notation than Binary?
° Administrivia, “What’s this stuff good for”
° Shift Instructions
° And/Or Instructions
° Conclusion
What if too big?

○ Binary bit patterns above are simply **representatives** of numbers

○ Numbers really have an infinite number of digits
  • with almost all being zero except for a few of the rightmost digits
  • Just don’t normally show leading zeros

○ If result of add (or any other arithmetic operation), cannot be represented by these rightmost hardware bits, **overflow** is said to have occurred

○ Up to Compiler and OS what to do
How avoid overflow, allow it sometimes?

° Some languages detect overflow (Ada), some don’t (C)

° MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  • add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow
  • add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do not cause exceptions on overflow

° Compiler selects appropriate arithmetic
  • MIPS C compilers produce addu, addiu, subu
What if Overflow Detected?

° An “exception” (or “interrupt”) occurs
  • Address of the instruction that overflowed is saved in a register
  • Computer jumps to predefined address to invoke appropriate routine for that exception
  • Like an unplanned hardware function call

° Operating system decides what to do
  • In some situations program continues after corrective code is executed

° MIPS support: exception program counter (EPC) contains address of that instruction
  • move from system control (mfco) to copy EPC
How Represent Negative Numbers?

° Obvious solution: add a separate sign!
  • sign represented in a single bit!
  • representation called **sign and magnitude**

° Shortcomings of sign and magnitude
  • Where to put the sign bit: right? left?
  • Separate sign bit means it has both a positive and negative zero, lead to problems for inattentive programmers: \( +0 = -0 \)?
  • Adder may need extra step size don’t know sign in advance

° Thus sign and magnitude was abandoned
Search for Negative Number Representation

° Obvious solution didn’t work, find another

° What is result for unsigned numbers if tried to subtract large number from a small one?
  • Would try to borrow from string of leading 0s, so result would have a string of leading 1s
  • With no obvious better alternative, pick representation that made the hardware simple: leading 0s => positive, leading 1s => negative
    • 000000...xxx is >=0, 111111...xxx is < 0

° This representation called two’s complement
## Two’s Complement

<table>
<thead>
<tr>
<th>Binary</th>
<th>Two's Complement</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 ... 0000 0000 0000 0000</td>
<td>0&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>0&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>0000 ... 0000 0000 0000 0001</td>
<td>1&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>1&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>0000 ... 0000 0000 0000 0010</td>
<td>2&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>2&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>0111 ... 1111 1111 1111 1101</td>
<td>2,147,483,645&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>2,147,483,645&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>0111 ... 1111 1111 1111 1110</td>
<td>2,147,483,646&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>2,147,483,646&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>0111 ... 1111 1111 1111 1111</td>
<td>2,147,483,647&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>2,147,483,647&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>1000 ... 0000 0000 0000 0000</td>
<td>-2,147,483,648&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>-2,147,483,648&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>1000 ... 0000 0000 0000 0001</td>
<td>-2,147,483,647&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>-2,147,483,647&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>1000 ... 0000 0000 0000 0010</td>
<td>-2,147,483,646&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>-2,147,483,646&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>1111 ... 1111 1111 1111 1101</td>
<td>-3&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>-3&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>1111 ... 1111 1111 1111 1110</td>
<td>-2&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>-2&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
<tr>
<td>1111 ... 1111 1111 1111 1111</td>
<td>-1&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>-1&lt;sub&gt;ten&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

° One zero, 1st bit => >=0 or <0, called **sign bit**

• but one negative with no positive –2,147,483,648<sub>ten</sub>
Two’s Complement Formula, Example

- Recognizing role of sign bit, can represent positive and negative numbers in terms of the bit value times a power of 2:
  \[ d_{31} \times -2^{31} + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0 \]

- Example

  \[ \text{1111 1111 1111 1111 1111 1111 1111 1100}_{\text{two}} \]

  \[ = 1 \times -2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \]

  \[ = -2^{31} + 2^{30} + 2^{29} + \ldots + 2^2 + 0 + 0 \]

  \[ = -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}} \]

  \[ = -4_{\text{ten}} \]
Overflow for Two’s Complement Numbers?

° Adding (or subtracting) 2 32-bit numbers can yield a result that needs 33 bits

• sign bit set with \textbf{value} of result instead of proper \textbf{sign} of result

• since need just 1 extra bit, only sign bit can be wrong

<table>
<thead>
<tr>
<th>Op</th>
<th>A</th>
<th>B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>&lt;0</td>
</tr>
<tr>
<td>A + B</td>
<td>&lt;0</td>
<td>&lt;0</td>
<td>&gt;=0</td>
</tr>
<tr>
<td>A - B</td>
<td>&gt;=0</td>
<td>&lt;0</td>
<td>&lt;0</td>
</tr>
<tr>
<td>A - B</td>
<td>&lt;0</td>
<td>&gt;=0</td>
<td>&gt;=0</td>
</tr>
</tbody>
</table>

° Adding operands with different signs, (subtracting with same signs) overflow cannot occur
Signed v. Unsigned Comparisons

° Note: memory addresses naturally start at 0 and continue to the largest address
  • That is, negative addresses make no sense

° C makes distinction in declaration
  • integer (int) can be positive or negative
  • unsigned integers (unsigned int) only positive

° Thus MIPS needs two styles of compare
  • Set on less than (slt) and set on less than immediate (slti) work with signed integers
  • Set on less than unsigned (sltu) and set on less than immediate unsigned (sltiu)
Example: Signed v. Unsigned Comparisons

° $s_0$ has
1111 1111 1111 1111 1111 1111 1111 1100\text{two}

° $s_1$ has
0011 1011 1001 1010 1000 1010 0000 0000\text{two}

° What are $t_0$, $t_1$ after
\begin{align*}
\text{slt} & \quad t_0, s_0, s_1 \# \text{ signed compare} \\
\text{sltu} & \quad t_1, s_0, s_1 \# \text{ unsigned compare}
\end{align*}

° $t_0$: -4_{\text{ten}} < 1,000,000,000_{\text{ten}}? \\
° $t_1$: 4,294,967,292_{\text{ten}} < 1,000,000,000_{\text{ten}}?
Administrivia

° Readings: (4.1, 4.2, 4.3) 3.7, 4.8 (skip HW)

° 3rd homework: Due Tonight 7PM

° 4th homework: Due 2/17 7PM
  • Exercises 3.21, 4.3, 4.7, 4.14, 4.15, 4.31

° 2nd project: MIPS Disassembler
  Due Wed. 2/17 7PM

° Book is a valuable reference!
  • Appendix A (as know more, easier to refer)
  • Back inside cover has useful MIPS summary: instructions, descriptions, definitions, formats, opcodes, examples
Remote Diagnosis:
“NeoRest ExII,” a high-tech toilet features microprocessor-controlled seat warmers, automatic lid openers, air deodorizers, water sprays and blow-dryers that do away with the need for toilet tissue. About 25 percent of new homes in Japan have a “washlet,” as these toilets are called. Toto's engineers are now working on a model that analyzes urine to determine blood-sugar levels in diabetics and then automatically sends a daily report, by modem, to the user's physician.

One Digital Day, 1998
www.intel.com/onedigitalday
Two’s complement shortcut: Negation

- Invert every 0 to 1 and every 1 to 0, then add 1 to the result
  - Sum of number and its inverted representation must be $111\ldots111_{two}$
  - $111\ldots111_{two} = -1_{ten}$
  - Let $x'$ mean the inverted representation of $x$
  - Then $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$

- Example: -4 to +4 to -4

x : \ 1111 1111 1111 1111 1111 1111 1111 1100 \_{two}
x' : 0000 0000 0000 0000 0000 0000 0000 0011 \_{two}
+1 : 0000 0000 0000 0000 0000 0000 0000 0100 \_{two}
\(): 1111 1111 1111 1111 1111 1111 1111 1011 \_{two}
+1 : 1111 1111 1111 1111 1111 1111 1111 1100 \_{two}
Two’s complement shortcut: Sign extension

- Convert two’s complement number represented in n bits to more than n bits
  - e.g., 16-bit immediate field converted to 32 bits before adding to 32-bit register in addi

- Simply replicate the most significant bit (sign bit) of smaller quantity to fill new bits
  - 2’s comp. positive number has infinite 0s to left
  - 2’s comp. negative number has infinite 1s to left
  - Bit representation hides most leading bits; sign extension restores some of them
  - 16-bit \(-4_{\text{ten}}\) to 32-bit:\n    \[
    \begin{array}{cccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]
    \[
    \begin{array}{cccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]
    \[
    \begin{array}{cccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]
    \[
    \begin{array}{cccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]
More Compact Representation v. Binary?

- Shorten numbers by using higher base than binary that converts easily into binary
  - almost all computer data sizes are multiples of 4, so use **hexadecimal** (base 16) numbers
  - base 16 a power of 2, convert by replacing group of 4 binary digits by 1 hex digit; and vice versa

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>b</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>e</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>f</td>
</tr>
</tbody>
</table>

- **Example**: from before, $1\ 000\ 000\ 000_{\text{ten}}$ is
  - $0011\ 1011\ 1001\ 1010\ 1100\ 1010\ 0000\ 0000_{\text{two}}$
  - C uses notation **0x3b9aca00**
Logical Operations

° Operations on less than full words
  • Fields of bits or individual bits

° Think of word as 32 bits vs. 2’s comp. integers or unsigned integers

° Need to extract bits from a word, insert bits into a word

° Extracting via Shift instructions
  • C operators: << (shift left), >> (shift right)

° Inserting via And/Or instructions
  • C operators: & (bitwise AND), | (bitwise OR)
Shift Instructions

° Move all the bits in a word to the left or right, filling the emptied bits with 0s

° Before and after shift left 8 of $s0 ($16):

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
& & & & & & & 1101 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
& & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 1101 \\
\end{array}
\]

° MIPS instructions

• shift left logical (sll) and shift right logical (srl)

\[
sll \quad s0,s0,8 \quad \# \quad s0 = s0 \ll 8 \text{ bits}
\]

• Register Format, using shamt (shift amount):

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>
Extracting a field of bits

° Extract bit field from bit 9 (left bit no.) to bit 2 (size=8 bits) of register $s1$, place in rightmost part of register $s0$

```
31  9876543210
```

° Shift field as far left as possible (31-bit no.) and then as far right as possible (32-size)

```
00000000000000000000000000000000
```

° `sll $s0,$s1,22 #8bits to left end (31-9)`
° `srl $s0,$s0,24 #8bits to right end (32-8)`
And instruction

° AND: bit-by-bit operation leaves a 1 in the result only if both bits of the operands are 1. For example, if registers $t1$ and $t2$

- $0000 0000 0000 0000 0000 1101 0000 0000_{two}$
- $0000 0000 0000 0000 0011 1100 0000 0000_{two}$

° After executing MIPS instruction

- `and $t0,$t1,$t2` # $t0 = $t1 & $t2$

° Value of register $t0$

- $0000 0000 0000 0000 0000 1100 0000 0000_{two}$

° AND can force 0s where 0 in the bit pattern

- Called a “mask” since mask “hides” bits
Or instruction

- OR: bit-by-bit operation leaves a 1 in the result if *either* bit of the operands is 1. For example, if registers $t_1$ and $t_2$

```
• 0000 0000 0000 0000 0000 1101 0000 0000
• 0000 0000 0000 0000 0011 1100 0000 0000
```

- After executing MIPS instruction

```
or $t_0,$t_1,$t_2 # $t_0 = $t_1 \& $t_2
```

- Value of register $t_0$

```
• 0000 0000 0000 0000 0011 1101 0000 0000
```

- OR can force 1s where 1 in the bit pattern

  - If 0s in field of 1 operand, can insert new value
Inserting a field of bits (almost OK)

- Insert bit field into bit 9 (left bit no.) to bit 2 (size=8 bits) of register $s1$ from rightmost part of register $s0$ (rest is 0)

```
31 9876543210
000000000000000000000000
```

- Mask out field, shift left field 2, OR in field

```
0000000000000000000000
0000000000000000000000
```

- andi $s1,$s1,0xfc03 # mask out 9-2
- sll $t0,$s0,2 # field left 2
- or $s1,$s1,$t0 # OR in field
Sign Extension of Immediates

- `addi` and `slti`: deal with signed numbers, so immediates sign extended

- Branch and data transfer address fields are sign extended too

- `addiu` and `sltiu` also sign extend!
  - `addiu` really to avoid overflow; `sltiu` why?

- `andi` and `ori` work with unsigned integers, so immediates padded with leading 0s
  - `andi` won’t work as mask in upper 16 bits

```
addiu $t1,$zero,0xfc03  #32b mask in $t1
and        $s1,$s1,$t1   # mask out 9-2
sll        $t0,$s0,2     # field left 2
or         $s1,$s1,$t0   # OR in field
```
Summary: 12 new instructions (with formats)

- **Unsigned (no overflow) arithmetic:**
  - `addu (R)`, `subu (R)`, `addiu (I)`

- **Unsigned compare:**
  - `sltu (R)`, `sltiu (I)`

- **Logical operations:**
  - `and (R)`, `or (R)`, `andi (I)`, `ori (I)`, `sll (R)`, `srl (R)`

- **Handle overflow exception:**
  - `EPC` register and `mfcsr0 (R)`
Example: show C, assembly, machine

Convert C code: Bit Fields in C

```c
struct {
    unsigned int ready: 1; /* bit 31 */
    unsigned int enable: 1; /* bit 30 */
} rec; /* $S0 */
rec.enable = 1;
rec.ready = 0;
printf("%d %d", rec.enable, rec.ready);
...
```
“And in Conclusion...” 1/1

- Handling case when number is too big for representation (overflow)
- Representing negative numbers (2’s complement)
- Comparing signed and unsigned integers
- Manipulating bits within registers: shift and logical instructions
- Next time: characters, floating point numbers