CS61C
Characters and Floating Point

Lecture 8

February 12, 1999

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

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

- 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
Review 2/2 : 12 new instructions

° Arithmetic:
  • No overflow (Unsigned): addu, subu, addiu
  • May overflow (2’s comp.): add, sub, addi
  • Handle overflow exception: EPC register has address of instruction and mfc0 to copy EPC

° Compare:
  • Unsigned (0 to 2^{N-1}): sltu, sltiu
  • 2’s comp. (-2^{N-1} to 2^{N-1}-1): slt, slti

° Logical operations (0 to 2^{N-1}):
  and, or, andi, ori, sll, srl
Overview

° How Represent Characters?
° How Represent Strings?
° Administrivia, “Computers in the News”
° What about Fractions, Large Numbers?
° Conclusion
Beyond Integers (Fig. 3-15, page 142)

8-bit bytes represent characters, nearly every computer uses American Standard Code for Information Interchange (ASCII)

<table>
<thead>
<tr>
<th>No.</th>
<th>char</th>
<th>No.</th>
<th>char</th>
<th>No.</th>
<th>char</th>
<th>No.</th>
<th>char</th>
<th>No.</th>
<th>char</th>
<th>No.</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48</td>
<td>0</td>
<td>64</td>
<td>@</td>
<td>80</td>
<td>P</td>
<td>96</td>
<td>`</td>
<td>112</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
<td>65</td>
<td>A</td>
<td>81</td>
<td>Q</td>
<td>97</td>
<td>a</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
<td>66</td>
<td>B</td>
<td>82</td>
<td>R</td>
<td>98</td>
<td>b</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
<td>67</td>
<td>C</td>
<td>83</td>
<td>S</td>
<td>99</td>
<td>c</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>79</td>
<td>O</td>
<td>95</td>
<td>_</td>
<td>111</td>
<td>o</td>
<td>127</td>
<td>DEL</td>
</tr>
</tbody>
</table>

- Uppercase + 32 = Lowercase (e.g., B+32=b)
- tab=9, carriage return=13, backspace=8, Null=0
Instruction Support for Characters

- MIPS (and most other instruction sets) include 2 instructions to move bytes:
  - **Load byte** ($lb$) loads a byte from memory, placing it in rightmost 8 bits of a register
  - **Store byte** ($sb$) takes a byte from rightmost 8 bits of register and writes it to memory

- Declares byte variables in C as “char”

- Assume $x, y$ are declared `char`, $y$ in memory at $0 ($sp$) and $x$ at $4 ($gp$). What is MIPS code for $x = y$; ?

  ```
  lb $t0, 0($sp)       # Read byte y
  sb $t0, 4($gp)       # Write byte x
  ```
Strings

° Characters normally combined into strings, which have variable length
  • e.g., “Cal”, “U.C.B.”, “U.C. Berkeley”

° How represent a variable length string?
  1) 1st position of string reserved for length of string (Pascal)
  2) an accompanying variable has the length of string (as in a structure)
  3) last position of string is indicated by a character used to mark end of string (C)

° C uses 0 (Null in ASCII) to mark end of string
**Example String**

- **How many bytes to represent string “Popa”?**
- **What are values of the bytes for “Popa”?**

<table>
<thead>
<tr>
<th>No. char</th>
<th>No. char</th>
<th>No. char</th>
<th>No. char</th>
<th>No. char</th>
<th>No. char</th>
<th>No. char</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48 0</td>
<td>64 @</td>
<td>80 P</td>
<td>96 `</td>
<td>112 p</td>
<td></td>
</tr>
<tr>
<td>33 !</td>
<td>49 1</td>
<td>65 A</td>
<td>81 Q</td>
<td>97 a</td>
<td>113 q</td>
<td></td>
</tr>
<tr>
<td>34 &quot;</td>
<td>50 2</td>
<td>66 B</td>
<td>82 R</td>
<td>98 b</td>
<td>114 r</td>
<td></td>
</tr>
<tr>
<td>35 #</td>
<td>51 3</td>
<td>67 C</td>
<td>83 S</td>
<td>99 c</td>
<td>115 s</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>47 /</td>
<td>63 ?</td>
<td>79 O</td>
<td>95 _</td>
<td>111 o</td>
<td>127 DEL</td>
<td></td>
</tr>
</tbody>
</table>

- **80, 111, 112, 97, 0**
Sign Extension and Load Byte

- MIPS automatically extends “sign” of byte for load byte (lb) instruction

 Naturally don’t want sign extension; hence another instruction: load byte unsigned (lbu)

• Almost always use lbu instead of lb
Strings in C: Example

° String simply an array of char

```c
void strcpy (char x[], char y[])
{
    int i = 0; /* declare, initialize i*/

    while ((x[i] = y[i]) != '\0') /* 0 */
        i = i + 1; /* copy and test byte */
}
```

° a leaf function (no calls), so i maps to $t0

```assembly
strcpy:
    add $t0, $zero, $zero    # i = 0 + 0
L1:   add $t1, $a1, $t0    # & y[i] in $t1
    lbu $t2, 0($t1)        # $t2 = y[i]
    add $t3, $a0, $t0     # & x[i] in $t3
    sb $t2, 0($t3)         # x[i] = y[i]
    add $t0, $t0, 1       # i = i + 1
    bne $t2, $zero, L1    # y[i]!=0, goto L1
    jr $ra                # return
```

i*4?
Strings in C: Example using pointers

° String simply an array of char
void strcpy2 (char *px, char *py) {

while ((*px++ = *py++) != '\0') /* 0 */
; /* copy and test byte */
}

strcpy2:
L1: lbu $t2, 0($a1)  # $t2 = y[i]
     add $a1,$a1,1   # py++
     sb $t2, 0($a0)  # x[i] = y[i]
     add $a0,$a0,1   # px++
     bne $t2,$zero,L1 # y[i]! =0, goto L1
     jr $ra  # return

° Again, ideally compiler optimizes code for you

° How see assembly? "(g)cc -S foo.c" => foo.s
What about non-Roman Alphabet?

° **Unicode**, universal encoding of the characters of most human languages
  
  • Java uses Unicode
  
  • needs 16 bits to represent a character
  
  • 16-bits called **halfwords** in MIPS

° **MIPS support for halfwords**
  
  • **Load halfword (unsigned)** ($lh, lhu$) loads 16 bits from memory, places in rightmost 16 bits of register; left half sign extend or zero
  
  • **Store halfword** ($sh$) takes rightmost 16 bits of register and writes it to memory

° We’ll skip $lh, lhu, sh$ in 61c MIPS subset
Administrivia

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

° **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

° **Midterm conflict time:** Mon 3/15 6-9PM

° **Course workload**
  • Trying to “front load” the course
  • 4/6 projects before Spring break
  • Fewer hours/week after Spring break
“Computers in the News”

“Price War Between Advanced Micro and Intel Ravages Chip Stocks”, NY Times 2/8/99

- Intel reduced price of fastest Celeron, 400-MHz, to $133 from $158. Advanced Micro in turn lowered price of its most powerful chip, the 400-megahertz K6-2, to $134 from $157.

- Intel stock drops 8% in 2 days, AMD drops 20%, Set off 2-day rout of technology-laden Nasdaq

- Technical: same instruction set abstraction, so AMD binary-compatible with Intel, head-to-head

  - Intel announce, AMD catchup, Intel announce next

- Why? Intel never as aggressive but “... now no one, including Intel, can ignore the low end because that's where all the growth is.”
ASCII v. Binary

° Why not ASCII computers vs. binary computers?
  • Harder to build hardware for add, subtract, multiply, divide
  • Memory space to store numbers

° How many bytes to represent 1 billion?

° ASCII: '1000000000' => 11 bytes

° Binary: 0011 1011 1001 1010 1000 0000 0000 0000
  => 4 bytes

° up to 11/4 or almost 3X expansion of data size
Other numbers

What can be represented in N bits?

- Unsigned: 0 to $2^N - 1$
- 2s Complement: $-2^{(N-1)}$ to $2^{(N-1)} - 1$
- ASCII: $-10^{(N/8-2)} - 1$ to $10^{(N/8-1)} - 1$

But, what about?

- Very large numbers? (seconds/century)
  $3,155,760,000_{\text{ten}} (3.15576_{\text{ten}} \times 10^9)$
- Very small numbers? (secs/ nanosecond)
  $0.000000001_{\text{ten}} (1.0_{\text{ten}} \times 10^{-9})$
- Rationals: $2/3$ (0.666666666. . .)
- Irrationals: $2^{1/2}$ (1.414213562373. . .)
- Transcendentals: $e$ (2.718...), $\pi$ (3.141...)
Recall Scientific Notation

(sign, magnitude) \[ \text{Mantissa} \]

\[ 6.02 \times 10^{23} \]

\text{decimal point} \hspace{1cm} \text{radix (base)}

° Normal form: no leadings 0s (1 digit to left of decimal point)

° Alternatives to represent 1/1,000,000,000,000

• Normalized: \[ 1.0 \times 10^{-9} \]
  • Not normalized: \[ 0.1 \times 10^{-8}, 10.0 \times 10^{-10} \]
Scientific Notation for Binary Numbers

(sign, magnitude) (sign, magnitude)

Mantissa exponent

1.0\text{two} \times 2^{-1}

“binary point” radix (base)

° Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers

• Declare such variable in C as float

° Normal format: \(1.\text{xxxxxx}_\text{two} \times 2^{\text{yyyy}_\text{two}}\)

• Simplifies data exchange, increases accuracy
Floating Point Number Representation

- Multiple of Word Size (32 bits)

\[
\begin{array}{c|cc|c}
S & \text{Exponent} & \text{Significand} \\
\hline
1 \text{ bit} & 8 \text{ bits} & 23 \text{ bits} \\
\end{array}
\]

- Roughly \((-1)^S \times F \times 2^{\text{Exponent}}\) : details soon

- Represent numbers as small as \(2.0 \times 10^{-38}\) to as large as \(2.0 \times 10^{38}\)
Floating Point Number Representation

° What if result too large? (> 2.0\times10^{38} )
  • **Overflow**!
  • Overflow => Exponent larger than represented in 8-bit Exponent field

° What if result too small? (>0, < 2.0\times10^{-38} )
  • **Underflow**!
  • Overflow => Negative exponent larger than represented in 8-bit Exponent field

° How reduce chances of overflow or underflow?
Double Precision Fl. Pt. Representation

° Next Multiple of Word Size (64 bits)

<table>
<thead>
<tr>
<th></th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>31 30</td>
<td>20 19 0</td>
</tr>
</tbody>
</table>

1 bit 11 bits 20 bits

Significand (cont’d)

32 bits

° Double Precision (vs. Single Precision)

- C variable declared as `double`
- Represent numbers almost as small as \(2.0 \times 10^{-308}\) to almost as large as \(2.0 \times 10^{308}\)
- But primary advantage greater accuracy due to larger significand
MIPS follows IEEE 754 Floating Point Standard

° To pack more bits, leading 1 implicit for normalized numbers
  • 1 + 23 bits single, 1 + 52 bits double
  • 0 has no leading 1, so reserve exponent value 0 just for number 0
  • Represents \((-1)^S \times (1 + \text{Significand}) \times 2^{\text{Exponent}}\)
    where \(0 < \text{Significand} < 1\)

° If number significand bits left-to-right \(s_1, s_2, s_3, \ldots\) then value is
  \((-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + \ldots) \times 2^{\text{Exponent}}\)
Representing Exponent

° Want compare Fl.Pt. numbers as if integers, to help in sort
  • Sign first part of number
  • Exponent next, so big exponent => bigger
    $1.1 \times 10^{20} > 1.9 \times 10^{10}$

° Negative Exponent?
  • 2’s comp? $1.0 \times 2^{-1}$ v. $1.0 \times 2^{+1}$ (1/2 v. 2)

<table>
<thead>
<tr>
<th></th>
<th>1111 1111</th>
<th>000 0000 0000 0000 0000 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>0</td>
<td>000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0000 0001 000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

• This notation using integer compare of
  1/2 v. 2 makes $1/2 > 2$!
Representing Exponent

° Instead, pick notation 0000 0000 is most negative, and 1111 1111 is most positive
  • $1.0 \times 2^{-1}$ v. $1.0 \times 2^{+1}$ ($1/2$ v. $2$)

| 1/2 | 0 0111 1110 | 000 0000 0000 0000 0000 0000 0000 |
| 2   | 0 1000 0000 | 000 0000 0000 0000 0000 0000 0000 |

° Called **Biased Notation**, where bias is number subtract to get real number
  • IEEE 754 uses bias of 127 for single prec.:
    $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  • 1023 is bias for double precision
Example: Converting Decimal to Fl. Pt.

Show MIPS representation of -0.75 (show exponent in decimal to simplify)

-0.75 = -3/4 = -3/2^2

-11_{two}/2^2 = -0.11_{two}

Normalized to -1.1_{two} x 2^{-1}

(-1)^S x (1 + Significand) x 2^{(Exponent-127)}

(-1)^1 x (1 + .100 0000 ... 0000) x 2^{(126-127)}

\[ \begin{array}{cccccccccccccccc}
1 & 0111 & 1110 & 100 & 0000 & 0000 & 0000 & 0000 & 0000 \\
\end{array} \]
Example: Converting Fl. Pt. to Decimal

| 0 | 0110 1000 | 101 0101 0100 0011 0100 0010 |

° Sign: 0 => positive

° Exponent:
  • 0110 1000\text{two} = 104\text{ten}
  • Bias adjustment: 104 - 127 = -13

° Significand:
  • \(1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}\)
    = 1+ (5,587,778/2^{23})
    = 1+ (5,587,778/8,388,608) = 1.0 + 0.666115

° Represents: \(1.666115\text{ten} \times 2^{-13} \sim 2.034 \times 10^{-4}\)
Continuing Example: Binary to ???

| 0011 0100 0101 0101 0100 0011 0100 0010 |

Convert 2’s Comp. Binary to Integer:

\[2^{29} + 2^{28} + 2^{26} + 2^{22} + 2^{20} + 2^{18} + 2^{16} + 2^{14} + 2^9 + 2^8 + 2^6 + 2^1\]

\[= 878,003,010_{ten}\]

Convert Binary to Instruction:

<table>
<thead>
<tr>
<th>0011 0100 0101 0101 0100 0011 0100 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011 0100 0101 0101 0100 0011 0100 0010</td>
</tr>
</tbody>
</table>

| 13 2 21 17218 |

ori $s5, $v0, 17218

Convert Binary to ASCII:

| 0011 0100 0101 0101 0100 0011 0100 0010 |

| 4 U C B |

ori $s5, $v0, 17218
Big Idea: Type not associated with Data

What does bit pattern mean:

- $2.034 \times 10^{-4}$? 878,003,010? “4UCB”? ori $s5, \$v0, 17218$?

Data can be anything; operation of instruction that accesses operand determines its type!

- Side-effect of stored program concept: instructions stored as numbers

Power/danger of unrestricted addresses/pointers: use ASCII as Fl. Pt., instructions as data, integers as instructions, ... (Leads to security holes in programs)
“And in Conclusion...” 1/1

° **Big Idea:** Instructions determine meaning of data; nothing inherent inside the data

° **Characters:** ASCII takes one byte
  • MIPS support for characters: `lbu, sb`

° **C strings:** Null terminated array of bytes

° **Floating Point Data:** approximate representation of very large or very small numbers in 32-bits or 64-bits
  • IEEE 754 Floating Point Standard
  • Driven by Berkeley’s Professor Kahan

° **Next time:** Fl. Pt. Ops, Multiply, Divide