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)

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

- 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

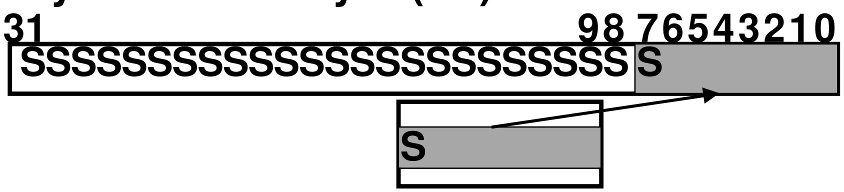
- How many bytes to represent string "Popa"?
- What are values of the bytes for "Popa"?

No. char	No. char	No. char	No. char	No. char	No. char
32	48 0	64 @	$\boxed{\text{80 P}}$	96 `	$\boxed{\text{112 p}}$
33 !	49 1	65 A	81 Q	$\boxed{\text{97 a}}$	113 q
34 "	50 2	66 B	82 R	98 b	114 r
35 #	51 3	67 C	83 S	99 c	115 s
...	...	...	...	...	...
47 /	63 ?	79 O	95 _	$\boxed{\text{111 o}}$	127 DEL

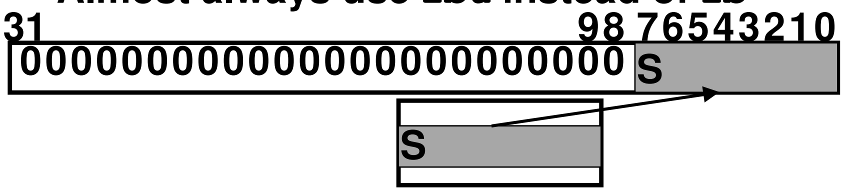
- 80, 111, 112, 97, 0

Sign Extension and Load Byte

- MIPS automatically extends "sign" of byte for load byte (`lb`) instruction



- Normally 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

```
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`

```
strcpy:
      add  $t0,$zero,$zero   # i = 0 + 0
   L1: add  $t1,$a1,$t0   # & y[i] in $t1
      lbu  $t2, 0($t1)   # $t2 = y[i]
i*4?  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
```


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++
sbu? 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?

- <u>Unicode</u>, universal encoding of the characters of most human languages
  - Java uses Unicode
  - needs 16 bits to represent a character
  - 16-bits called <u>halfwords</u> in MIPS
- MIPS support for halfwords
  - <u>Load halfword (unsigned)</u> (`lh`,`lhu`) loads 16 bits from memory, places in rightmost 16 bits of register; left half sign extend or zero
  - <u>Store halfword</u> (`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) $3{,}155{,}760{,}000_{ten}$ ($3.15576_{ten} \times 10^9$)
  - Very small numbers? (secs/ nanosecond) 0.000000001_{ten} ($1.0_{ten} \times 10^{-9}$)
  - Rationals 2/3 (0.666666666. . .)
  - Irrationals $2^{1/2}$ (1.414213562373. . .)
  - Transcendentals e (2.718...), π (3.141...)

Recall Scientific Notation

(sign, magnitude) *Mantissa* → 6.02×10^{23} ← *exponent* (sign, magnitude)

decimal point · *radix (base)*

- Normal form: no leadings 0s (1 digit to left of decimal point)
- Alternatives to represent 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation for Binary Numbers

(sign, magnitude) *Mantissa* → $1.0_{two} \times 2^{-1}$ ← *exponent* (sign, magnitude)

"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: $1.xxxxxxxxxx_{two} * 2^{yyyy_{two}}$
  - Simplifies data exchange, increases accuracy

Floating Point Number Representation

- Multiple of Word Size (32 bits)

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- Roughly $(-1)^S \times F \times 2^{Exponent}$: details soon
- Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Floating Point Number Representation

- What if result too large? ($> 2.0 \times 10^{38}$)
  - Overflow!
  - Overflow => Exponent larger than represented in 8-bit Exponent field
- What if result too small? (>0, $< 2.0 \times 10^{-38}$)
  - Underflow!
  - Overflow => Negatie 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)

31	30 … 20	19 … 0
S	Exponent	Significand
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 2.0×10^{-308} to almost as large as 2.0×10^{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 < Significand < 1
- If number signficand bits left-to-right s1, s2, s3, ... then value is

  $(-1)^S \times (1+(s1 \times 2^{-1})+(s2 \times 2^{-2})+(s3 \times 2^{-3})+\ldots) \times 2^{\text{Exponent}}$

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? 1.0×2^{-1} v. $1.0 \times 2^{+1}$ (1/2 v. 2)

	S	Exponent	Significand
1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

  - 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
  - 1.0×2^{-1} v. $1.0 \times 2^{+1}$ (1/2 v. 2)

	S	Exponent	Significand
1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 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 $-1.1_{two} \times 2^{-1}$
  - $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - $(-1)^1 \times (1 + .100\ 0000\ ...\ 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000

Example: Converting Fl. Pt. to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- **Sign: 0 => positive**
- **Exponent:**
  - $0110\ 1000_{two} = 104_{ten}$
  - 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: $1.666115_{ten}*2^{-13} \sim 2.034*10^{-4}$**

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:**

0011 01	00 010	1 0101	0100 0011 0100 0010
13	2	21	17218

```
ori $s5, $v0, 17218
```

- **Convert Binary to ASCII:**

0011 0100	0101 0101	0100 0011	0100 0010
4	U	C	B

Big Idea: Type not associated with Data

0011 0100 0101 0101 0100 0011 0100 0010

- **What does bit pattern mean:**
  - $2.034*10^{-4}$? 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