More on MIPS Programming and Memory

CS 64: Computer Organization and Design Logic
Lecture #5

Ziad Matni
Dept. of Computer Science, UCSB
Administrative

• Assignment #2: Due Monday 1/30
• Assignment #3: Due Friday 2/3

• Midterm #1 is on Feb. 2nd
  – Material included is: all lectures, all readings (see syllabus), all assignments turned in before 2/2.
  – Will do a review on the lesson before the midterm.
  – If you need to contact DSP, please do so ASAP
Lecture Outline

• Operand Use

• `.data` Directives

• Control logic programs
  – With Demo!

• MIPS Memory
A Note About Operands

• Operands in arithmetic instructions are limited and are done in a certain order
  – Arithmetic operations always happen in the registers

• Example: \( f = (g + h) - (i + j) \)
  – The order is prescribed by the parentheses
  – Let’s say, \( f, g, h, i, j \) are assigned to registers \( \$s0, \$s1, \$s2, \$s3, \$s4 \) respectively
  – What would the MIPS assembly code look like?
Example 1

\[ f = (g + h) - (i + j) \]

i.e. \[ s0 = (s1 + s2) - (s3 + s4) \]

- add \( t0 \), \( s1 \), \( s2 \)
- add \( t1 \), \( s3 \), \( s4 \)
- sub \( s0 \), \( t0 \), \( t1 \)
Example 2

\[ f = g \times h - i \]

i.e. \( s0 = (s1 \times s2) - s3 \)

```
mult $s1, $s2
mflo $t0
sub $s0, $t0, $s3
```
What’s the Difference Between...

- **add** and **addu** and **addi** and **addiu**
  - **add**: add what’s in 2 registers & put in another
  - **addu**: same as **add**, but only w/ *unsigned* numbers
  - **addi**: add a number to what’s in a register & put in another
  - **addiu**: same as **addi**, but only w/ *unsigned* numbers

- **Syntax:**
  
  - add $rd, $rs, $rt (R-Type)
  - addu $rd, $rs, $rt (R-Type)
  - addi $rd, $rs, immediate (I-Type)
  - addiu $rd, $rs, immediate (I-Type)
Global Variables

• Typically, global variables are placed directly in memory and **not** registers
  – Why might this be?
    • Not enough registers...
      esp. if there are multiple large GVs

• Can use the `.data` directive
  – Declares variable names used in program
  – Storage is allocated in main memory (RAM)
.data Declaration Types

| var1: .byte 9            | # declare a single byte |
| var2: .half 63           | # declare a 16-bit half-word |
| var3: .word 9433         | # declare a 32-bit word |
| num1: .float 3.14        | # declare 32-bit floating point number |
| num2: .double 6.28       | # declare 64-bit floating pointer number |
| str1: .ascii "Text"      | # declare a string of chars |
| str3: .asciiz "Text"     | # declare a null-terminated string |
| str2: .space 5           | # reserve 5 bytes of space |
.data
name: .ascii "Jimbo Jones is: 
rtm: .ascii "\n"

.text
main:
    li $v0, 4
    la $a0, name
    syscall

    li $v0, 1
    li $a0, 13
    syscall

    li $v0, 4
    la $a0, rtm
    syscall

    li $v0, 10
    syscall
Conditionals

• What if we wanted to do:
  
  ```
  if (x == 0) { printf("x is zero"); } 
  ```
  – Can we write this in assembly with what we know?
    • No...

• What do we need to implement this?
  – A way to compare numbers
  – A way to conditionally execute code
Relevant Instructions in MIPS

*for use with conditionals*

- Comparing numbers:
  - `set-less-than (slt)`

- Conditional execution:
  - `branch-on-equal (beq)`
  - `branch-on-not-equal (bne)`
if (x == 0) { printf(“x is zero”); }

.data
x_is_zero: .asciiz “x is zero”

.text
bne $t0, $zero, after_print
li $v0, 4
la $a0, x_is_zero
syscall

after_print:
li $v0, 10
syscall

Create a constant string called “x_is_zero”
If $t0 != 0 go to “after_print”
(otherside) prepare to print a string...
...and that string is inside of “x_is_zero”
End the program
Loops

• How might we translate the following to assembly?
  
  ```
  sum = 0;
  while (n != 0) {
    sum = sum + n;
    n--;
  }
  printf(sum);
  ```
sum = 0;
while (n != 0) { sum += n; n--; }
Let’s Run More Programs!!
Using SPIM

• More!!
• This time exploring conditional logic and loops

These assembly code programs are made available to you via the webpage
Larger Data Structures

• Recall: registers vs. memory
  – Where would data structures, arrays, etc. go?
  – Which is faster to access? Why?

• Some data structures have to be stored in memory
  – So we need instructions that “shuttle” data to/from the CPU and computer memory (RAM)
Accessing Memory

• Two base instructions:
  – load-word (lw) from memory to registers
  – store-word (sw) from registers to memory

• MIPS lacks instructions that do more with memory than access it (e.g., retrieve something from memory and then add)
  – Operations are done step-by-step
  – Mark of RISC architecture
.data
num1: .word 42
num2: .word 7
num3: .space 1

.text
main:
    lw $t0, num1
    lw $t1, num2
    add $t2, $t0, $t1
    sw $t2, num3

    li $v0, 1
    lw $a0, num3
    syscall

    li $v0, 10
    syscall
Addressing Memory

- If you’re not using the `.data` declarations, then you need *starting addresses* of the data in memory with `lw` and `sw` instructions.

  Example: `lw $t0, 0x0000400A`  (← not a real address)
  Example: `lw $t0, 0x0000400A($s0)`  (← not a real address)

- 1 word = 32 bits (in MIPS)
  - So, in a 32-bit unit of memory, that’s 4 bytes
  - Represented with 8 hexadecimals 8 x 4 bits = 32 bits... checks out...

- MIPS addresses sequential memory addresses, but not in “words”
  - Addresses are in Bytes instead
  - MIPS words *must* start at addresses that are multiples of 4
  - Called an *alignment restriction*
YOUR TO-DOs

• Thursday (1/26): more on MIPS memory
  – Finish reading 2.5, 2.6, 2.8

• Assignment #2: Due Monday 1/30
• Assignment #2: Due Friday 2/3