More on Instructions in Assembly

Functions

CS 64: Computer Organization and Design Logic
Lecture #8
Fall 2018

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- Mistakes on the practice exam questions
  - See fixes on Piazza
- Reminder that your midterm exam is on **WEDNESDAY!**
- **NO LAB THIS WEEK!**
- *Next week*: Finish off functions and MIPS calling conventions + 1 last assembly lab
- *Week after*: Digital Design!!!

Midterm? What Midterm?

What's on It?

- Everything we've done so far, incl. today's lesson

What to Bring?

- Your pencil(s), eraser, MIPS Ref. Card
- Your UCSB ID Card
- THAT'S ALL!

PLEASE ARRIVE 5 – 10 MINUTES EARLY!

I may choose to re-seat some people

Lecture Outline

- More on Instructions in MIPS
- Functions in MIPS

Exercises

- Using your MIPS Reference Card, write the 32 bit instruction (using the I-Type format and decimal numbers for all the fields) for the following:

```
addi $t3, $t2, -42      0x214BFFD6
andi $a0, $a3, 1        0x30E40001
slti $t8, $t8, 14       0x2B18000E
```

Aside: The Use of **add** vs **addu**

- Which one is for signed and which one is for unsigned numbers?
- Which one triggers an overflow bit and which one doesn't?

`srl` vs `sra`
Shift-Right Logic vs Arithmetic

- `srl` replaces the "lost" MSBs with 0s
- `sra` replaces the "lost" MSBs with *either* 0s (if number is +ve) *or* 1s (if number is –ve)

IMPLICATIONS:

- `srl` should NOT be used with negative numbers
  - That is, **unsigned** use only
- `sra` should be used with **signed** numbers
  - Can also be used with unsigned, but there's srl for that…

sra vs srl Exercise

DEMO!
srlsra.asm

- Is **sra (-19) >> 2** the same as **-(srl (19) >> 2)**?

```
19 =   0000000000000000 0000000000010011
-19 =  1111111111111111 1111111111101101

sra(-19) >> 2 = 1111111111111111 1111111111111011   (01)
              = -5                                   goes away

-srl(19) >> 2 = -(0000000000000000 0000000000000100) (11)
              = -4                                   goes away
```

Remember: **sra** vs **srl** Exercise

- `srl` replaces the "lost" MSBs with 0s
- `sra` replaces the "lost" MSBs with *either* 0s (if number is +ve) *or* 1s (if number is –ve)

EXAMPLE:

```
addi $t0, $zero, 12
addi $t1, $zero, -12

srl $s0, $t0, 1
sra $s1, $t0, 1
srl $s0, $t1, 1
sra $s1, $t1, 1
```

DEMO!
shiftDemo.asm

Functions in Assembly

Functions

- Up until this point, we have not discussed **functions**
- Why not?
  - If you want to do functions, you need to use **the stack**
  - Memory management is a must for the call stack ...
    though we can make *some* progress without it
- Think of recursion…
  - How many variables are we going to need ahead of time?
  - What memory do we end up using in recursive functions?
  - We don't always know…

Implementing Functions

What capabilities do we need for functions?

1. Ability to execute code elsewhere
   - Branches and jumps

2. Way to pass arguments in and out of the func.
   - There a way (aka convention) to do that that we'll learn about
   - We'll use the registers to do function I/O

Jumping to Code

- We have ways to jump unconditionally to code (`j` instruction)

```
void foo() {
  bar();
  baz();
}
```

```
void bar() {
  ...
}
```

```
void baz() {
  ...
}
```

- But what about ***jumping back***?
  - That is, after you're done with a function?
  - We'll need a way to *save* where we were (so we can "jump" back)

- **Q**: What do need so that we can do this on MIPS?
  - **A**: A way to store the program counter (**$PC**)

  (to tell us where the *next* instruction is so that we know *where* to return!)

Calling Functions on MIPS

- Two crucial instructions: `jal` and `jr`
- One specialized register: **`$ra`**

- `jal` (**jump-and-link**)
  - Simultaneously **jump to an address**, and **store the location of the next instruction** in register `$ra`

- `jr` (**jump-register**)
  - **Jump to the address stored in a register**, often `$ra`

Simple Call Example

- See program: **simple_call.asm**

```
# Calls a function (test) which immediately returns
.text
test:  # return to whoever made the call
       jr $ra

main:  # do stuff…
       # then call the test function
       jal test

exit:  # exit
       li $v0, 10
       syscall
```

*Note: SPIM always starts execution at the line labeled "**main**"*

Passing and Returning Values

- We want to be able to call arbitrary functions without knowing the implementation details
- So, we need to know our pre-/post-conditions
- Q: How might we achieve this in MIPS?
  - A: We designate specific registers for **arguments** and **return values**

Passing and Returning Values in MIPS

- Registers **`$a0`** thru **`$a3`**
  - **Argument registers**, for passing function arguments
- Registers **`$v0`** and **`$v1`**
  - **Return registers**, for passing return values
- What if we want to pass >4 args?
  - There are ways around that…
    but we won't discuss them in CS64…!

Passing and Returning Values in MIPS

Demo: *print_ints.asm*

- Illustrates the use of a printing sub-routine (i.e. like a simple function)

Passing and Returning Values in MIPS

Demo: *print_ints.asm*

- Illustrates the use of a printing sub-routine
  (i.e. like a simple function)

- How would you write this function in C++?

```
void print_ints(int a0, int a1)
{
    cout << a0 << endl << a1 << endl;
}
```

Passing and Returning Values in MIPS

Demo: *add_ints.asm*

– Illustrates the use of an adding sub-routine
(i.e. like a simple function that returns a value)

Passing and Returning Values in MIPS

Demo: *add_ints.asm*

– Illustrates the use of an adding sub-routine
(i.e. like a simple function that returns a value)

- How would you write this function in C++?

```
int add_ints(int a0, int a1)
{
    v0 = a0 + a1;
    return (v0);
}
```

Function Calls Within Functions…

Given what we've said so far…

- What about this code (→) makes our previously discussed setup *break*?
  - ANS: You would need

    multiple copies of $ra

```
void foo() {
  bar();
}
void bar() {
  baz();
}
void baz() {}
```

- You'd have to copy the value of `$ra` to *another* register (or to memory) before calling another function

- Danger: You could run out of registers!

That's Why We Need…

… A set of agreed-upon rules (i.e. a convention) on how to deal with functions in assembly…

How to call them and
what we expect them and the memory to do once we call them

"The MIPS Calling Convention"

(coming soon to a CS64 lecture near you!)

YOUR TO-DOs

- Study for the midterm on Wednesday!
- No lab this week! No assignment either!

</LECTURE>