Assembly Language Basics

CS 64: Computer Organization and Design Logic
Lecture #5
Winter 2019

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

Lecture Outline

- Talking to the OS
  - Std I/O
  - Exiting
- General view of instructions in MIPS
- Operand Use
- **.data** Directives and Basic Memory Use

Administrative Stuff

- How did Lab# 2 go?
  - Challenge level:
    HARD vs. **OK** vs. **EASY-PEASY**

- Remember, our office hours! ☺
  - Prof. Matni Th. 1 – 2:30 PM *SSMS 4409*
  - TA Bay-Yuan Fr. 11 AM – 1 PM *Trailer 936*
  - TA Shiyu Fr. 3 – 5 PM *Trailer 936*

Any Questions From Last Lecture?

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: 12
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes
(more on this later).

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

Ok. Where's My MIPS Computer???

- You're not getting one.
- Who needs hardware when "cutting edge" software can do the job?!?!?!?!
- We will be SIMULATING a MIPS processor using software on our Macs/Windows/Linux machines.
- Hence… ***SPIM***… **<u>The MIPS Emulator</u>**!
  - Something funny about that name…

Adding More Functionality

- Ok, so I know how to add 2 numbers in MIPS.
  - Wow
- What about: display results???? *Yes, that's kinda important…*
- What would this entail?
  - Engaging with Input / Output part of the computer
  - i.e. talking to devices

  Q: What usually handles this?

  A: the operating system
- So we need a way to tell
  the operating system to kick in

Talking to the OS

- We are going to be running on MIPS ***emulator*** called **SPIM**
  - Optionally, through a program called **QtSPIM** (GUI based)
  - *What is an emulator?*

- We're not actually running our commands on an actual MIPS (hardware) processor!!

  …we're letting software pretend it's hardware…

  …so, in other words… we're "faking it"

- Ok, so how might we print something onto std.out?

SPIM Routines

- MIPS features a `syscall` instruction, which triggers a ***software interrupt***, or ***exception***
- Outside of an emulator (i.e. in the real world), these instructions **pause the program** and tell the OS to go do something with I/O
- Inside the emulator, it tells the emulator to go *emulate* something with I/O

`syscall`

- So we have the OS/emulator's attention, but how does it know what we want?

- The OS/emulator has access to the CPU registers

- We put special values (codes) in the registers to indicate what we want
  - These are codes that can't be used for anything else, so they're understood to be just for `syscall`
  - So… is there a "code book"????

Yes! All CPUs come with manuals.
For us, we have the **MIPS Ref. Card**

(Finally) Printing an Integer

- For SPIM, if register **$v0** contains **1** and then we issue a **syscall**, then SPIM will *print whatever **integer** is stored in register **$a0** ← this is a specific rule using a specific code*
  - Note: $v0 is used for other stuff as well – more on that later…
  - When $v0=1, syscall is *expecting* an integer!

- Other values put into **$v0** indicate other types of I/O calls to **syscall**

  Examples:
  - $v0 = 3 means **double (or the mem address of one) in $a0**
  - $v0 = 4 means **string (or the mem address of one) in $a0**
  - $v0 = 5 means **get user input from std input and place in $v0**
  - We'll explore some of these later, but check **MIPS ref card** for all of them

(Finally) Printing an Integer

- Remember, the usual syntax to load immediate a value into a register is:

```
li <register>, <value>
```

Example: **li $v0, 1** # PUTS THE NUMBER 1 INTO REG. $v0

- **You can also move the value of one register into another too!**
- E.g. To make sure that the register **$a0** has the value of what you want to print out (let's say it's in another register), use the **move** command:

```
move <to register>, <from register>
```

Example: **move $a0, $t0** # PUTS THE VALUE IN REG. $t0 INTO REG. $a0

Ok… So About Those Registers
MIPS has 32 registers, each is 32 bits

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Used for data

Program Files for MIPS Assembly

- The files have to be text
- Typical file extension type is **.asm**
- To leave comments,
  use **#** at the start of the line

Augmenting with Printing

```
# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print the integer that’s in $t3
# to std.output
li $v0, 1
move $a0, $t3
syscall
```

What About Std In?

```
# Get an integer value from user
li $v0, 5
syscall
# Your new input int is in $v0
# You can move it around
# and do stuff with it
move $t0, $v0
sll $t0, $t0, 2  # Multiply it by 4
```

We're Not Quite Done Yet! Exiting an Assembly Program in SPIM

- If you are using SPIM, then you need to say ***when you are done as well***
  - Most HLL programs do this for you automatically

- How is this done?
  - Issue a `syscall` with a special value in **$v0 = 10** (decimal)

Augmenting with Exiting

```
.text     # We always have to have this starting line
# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print to std.output
li $v0, 1
move $a0, $t3
Syscall

# End program
li $v0, 10
syscall
```

Let's Run This Program Already!
Using SPIM

- We'll call it **simpleadd.asm**
- Run it on CSIL as: `$ spim –f simpleadd.asm`

DEMO !!!

- We'll also run other arithmetic programs and explain them as we go along
  - TAKE NOTES!

MIPS System Services

Examples of what we'll be using in CS64: print_int, print_string, read_int, exit

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		
print_character	11	`$a0` = character	
read_character	12		character (in `$v0`)
open	13	`$a0` = filename,	file descriptor (in `$v0`)
		`$a1` = flags, `$a2` = mode	
read	14	`$a0` = file descriptor,	bytes read (in `$v0`)
		`$a1` = buffer, `$a2` = count	
write	15	`$a0` = file descriptor,	bytes written (in `$v0`)
		`$a1` = buffer, `$a2` = count	
close	16	`$a0` = file descriptor	0 (in `$v0`)
exit2	17	`$a0` = value	

stdout (services 1–4)

stdin (services 5–8)

File I/O (services 13–16)

Now Let's Make it a Full Program (almost)

- We need to tell the assembler (and its simulator) **which bits** should be placed **where** in memory

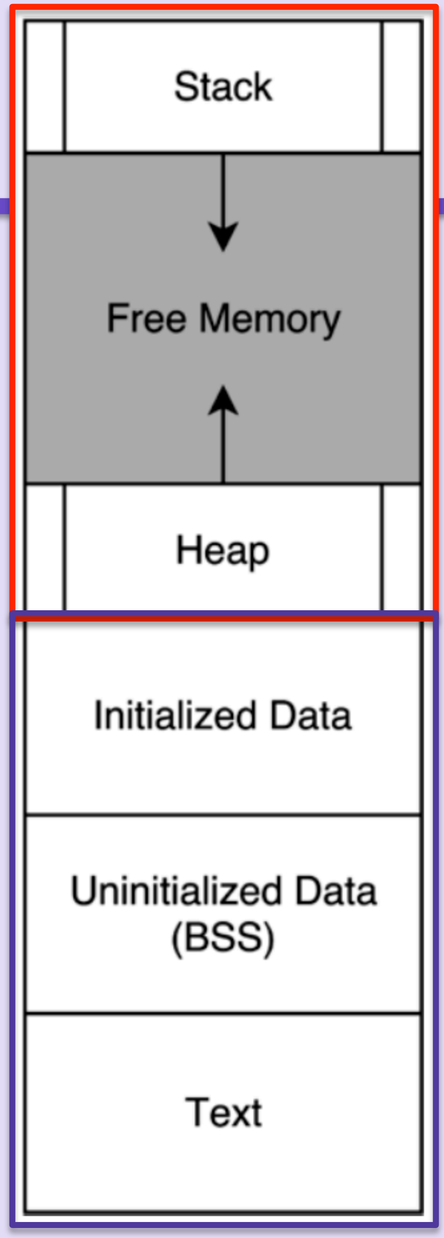
Allocated as program RUNs

Constants to be used in the program (like strings)

Allocated at program LOAD

mutable global variables

the text of the program



Marking the Code

- For the simulator, you'll need a **.text** directive to specify code

```
.text

# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print to standard output
li $v0, 1
move $a0, $t3
syscall

# End program
li $v0, 10
syscall
```

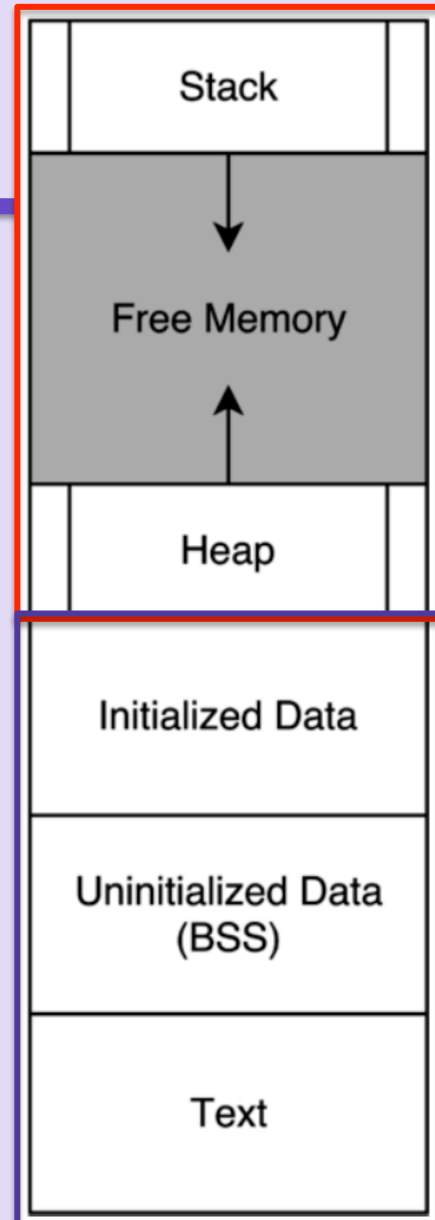
Allocated as program RUN

Constants to be used in the program (like strings)

Allocated at program LOAD

mutable global variables

the text of the program



List of all Core Instructions in MIPS

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT
Add	add	R
Add Immediate	addi	I
Add Imm. Unsigned	addiu	I
Add Unsigned	addu	R
And	and	R
And Immediate	andi	I
Branch On Equal	beq	I
Branch On Not Equal	bne	I
Jump	j	J
Jump And Link	jal	J
Jump Register	jr	R
Load Byte Unsigned	lbu	I
Load Halfword Unsigned	lhu	I
Load Linked	ll	I

"R"

Arithmetic

Branching

Load Upper Imm.	lui	I
Load Word	lw	I
Nor	nor	R
Or	or	R
Or Immediate	ori	I
Set Less Than	slt	R
Set Less Than Imm.	slti	I
Set Less Than Imm. Unsigned	sltiu	I
Set Less Than Unsig.	sltu	R
Shift Left Logical	sll	R
Shift Right Logical	srl	R
Store Byte	sb	I
Store Conditional	sc	I
Store Halfword	sh	I
Store Word	sw	I
Subtract	sub	R
Subtract Unsigned	subu	R

R-Type Syntax

<op>   <rd>, <rs>, <rt>

op : operation

rd : register destination

rs : register source

rt : register target

Examples:

```
add $s0, $t0, $t2
```

Add ($t0 + $t2) then store in reg. $s0

```
sub $t3, $t4, $t5
```

Subtract ($t4 – $t5) then store in reg. $t3

List of all Core Instructions in MIPS

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT
Add	add	R
Add Immediate	addi	I
Add Imm. Unsigned	addiu	I
Add Unsigned	addu	R
And	and	R
And Immediate	andi	I
Branch On Equal	beq	I
Branch On Not Equal	bne	I
Jump	j	J
Jump And Link	jal	J
Jump Register	jr	R
Load Byte Unsigned	lbu	I
Load Halfword Unsigned	lhu	I
Load Linked	ll	I
Load Upper Imm.	lui	I
Load Word	lw	I
Nor	nor	R
Or	or	R
Or Immediate	ori	I
Set Less Than	slt	R
Set Less Than Imm.	slti	I
Set Less Than Imm. Unsigned	sltiu	I
Set Less Than Unsig.	sltu	R
Shift Left Logical	sll	R
Shift Right Logical	srl	R
Store Byte	sb	I
Store Conditional	sc	I
Store Halfword	sh	I
Store Word	sw	I
Subtract	sub	R
Subtract Unsigned	subu	R

"I"

Arithmetic

Branching

Memory

Not for CS64

I-Type Syntax

<op> <rs>, <rt>, immed

op : operation

rs : register source

rt : register target

Examples:

```
addi $s0, $t0, 33
```

Add ($t0 + 33) then store in reg. $s0

```
ori $t3, $t4, 0
```

Logic OR ($t4 with 0) then store in reg. $t3

Note: this last one has the effect of just moving $t4 value into $t3

List of the Arithmetic Core Instructions in MIPS

Mostly used in CS64

You are not responsible for the rest of them

NAME, MNEMONIC		FOR-MAT
Branch On FP True	bc1t	FI
Branch On FP False	bc1f	FI
Divide	div	R
Divide Unsigned	divu	R
FP Add Single	add.s	FR
FP Add Double	add.d	FR
FP Compare Single	c.*x*.s*	FR
FP Compare Double	c.*x*.d*	FR
* (*x* is eq, lt, or le) (*op* is =		
FP Divide Single	div.s	FR
FP Divide Double	div.d	FR
FP Multiply Single	mul.s	FR
FP Multiply Double	mul.d	FR
FP Subtract Single	sub.s	FR
FP Subtract Double	sub.d	FR
Load FP Single	lwc1	I
Load FP Double	ldc1	I
Move From Hi	mfhi	R
Move From Lo	mflo	R
Move From Control	mfc0	R
Multiply	mult	R
Multiply Unsigned	multu	R
Shift Right Arith.	sra	R
Store FP Single	swc1	I
Store FP Double	sdc1	I

The **move** Instruction…

… is suspicious…

- The move instruction does not actually show up in SPIM!

- It is a ***pseudo-instruction***
- It's easy for us to use, but it's actually a "macro" of another actual instruction

ORIGINAL: `move $a0, $t3`

ACTUAL: `addu $a0, $zero, $t3`

```
# what's addu? what's $zero?
```

Why Pseudocodes?
And what's this `$zero`??

- **$zero**
  - Specified like a normal register,
    but does not behave like a normal register
  - Writes to $zero are not saved
  - Reads from $zero always return 0 value

- Why have **`move`** as a **pseudo-instruction** instead of as an actual instruction?
  - It's one less instruction to worry about
  - One design goal of RISC is to cut out redundancy
  - **`move`** isn't the only one! **`li`** is another one too!

List of all PsuedoInstructions in MIPS
That You Are Allowed to Use in CS64!!!

PSEUDOINSTRUCTION SET

NAME	MNEMONIC
Branch Less Than	blt
Branch Greater Than	bgt
Branch Less Than or Equal	ble
Branch Greater Than or Equal	bge
Load Immediate	li
Move	move

plus this one → **Load Address** **la**

ALL OF THIS AND MORE IS ON YOUR HANDY "MIPS REFERENCE CARD" FOUND ON THE CLASS WEBSITE

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

Syntax for "add"

`add rd, rs, rt`

destination, source1, source2

$f = (g + h) - (i + j)$

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

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Example 2

*f = g * h - i*

*i.e. $s0 = ($s1 * $s2) – $s3*

```
mult $s1, $s2
mflo $t0
# mflo directs where the answer of the mult should go
sub $s0, $t0, $s3
```

The **mult** instruction

- To multiply 2 integers together:

```
li $t0, 5
mult $t1, $t0
mflo $t2
```

- **`mult`** cannot be used with an 'immediate'
- So first, we load our multiplier into a register ($t0)
- Then we multiply this with out multiplicand ($t1)
- And we finally put the result in the final reg ($t2) using the **`mflo`** instruction

Global Variables, Arrays, and Strings

- Typically, global variables are placed directly in memory and **not** registers
  - Why might this be?
    - Ans: Not enough registers…
      esp. if there are multiple variables

- What do you think we do with arrays? Why?
- What do you think we do with strings? Why?

- We use the **.data** directive
  - To declare variables, their values, and their names used in the program
  - Storage is allocated in main memory (RAM)

.data Declaration Types

w/ Examples

```
var1:    .byte 9          # declare a single byte with value 9
var2:    .half 63         # declare a 16-bit half-word w/ val. 63
var3:    .word 9433       # declare a 32-bit word w/ val. 9433
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 (useful for arrays)
```

These are now reserved in memory and we can call them up by loading their memory address into the appropriate registers.

Highlighted ones are the ones most commonly used in this class.

`li` *vs* `la`

Very Important!

ATTN: Newbies!!!
Common Mistake!

- li Load Immediate
  - Use this when you want to put an integer value into a register
  - Example: `li $t0, 42`

- la Load Address
  - Use this when you want to put an address value into a register
  - Example: `la $t0, myLittlePony`

    where "myLittlePony" is a pre-defined label for something in memory (defined under the .data directive).

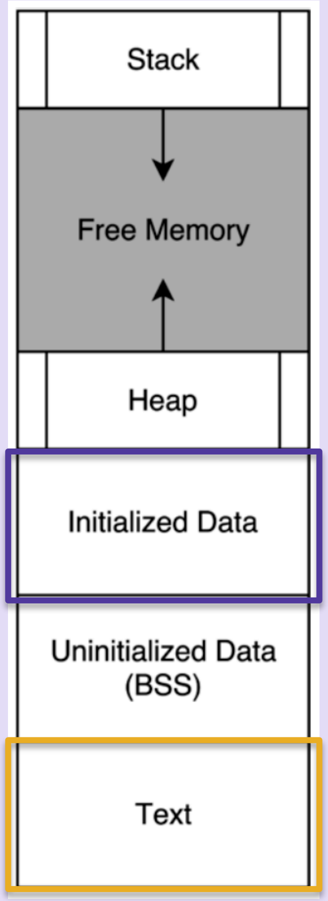
Example

What does this do?

```
.data
name: .asciiz "Jimbo Jones is "
rtn: .asciiz " years old.\n"

.text
main:
    li $v0, 4
    la $a0, name     # la = load memory address
    syscall

    li $v0, 1
    li $a0, 15
    syscall

    li $v0, 4
    la $a0, rtn
    syscall

    li $v0, 10
    syscall
```

What goes in here? → (Initialized Data)

What goes in here? → (Text)



YOUR TO-DOs

- Review ALL the demo codes
  - Available via the class website
- Assignment #3
  - Due Monday!

</LECTURE>