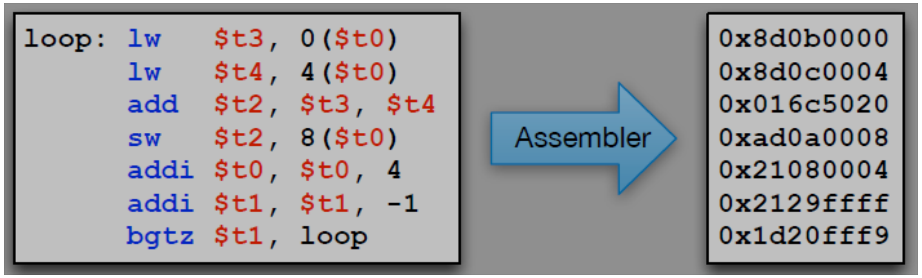


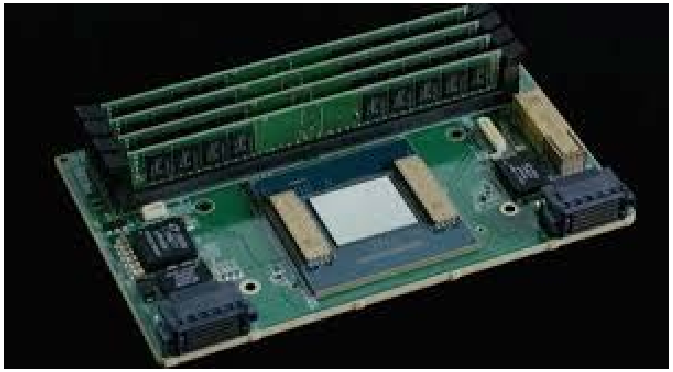
Basic Programs in MIPS Assembly Language

CS 64: Computer Organization and Design Logic

Lecture #5

Fall 2019

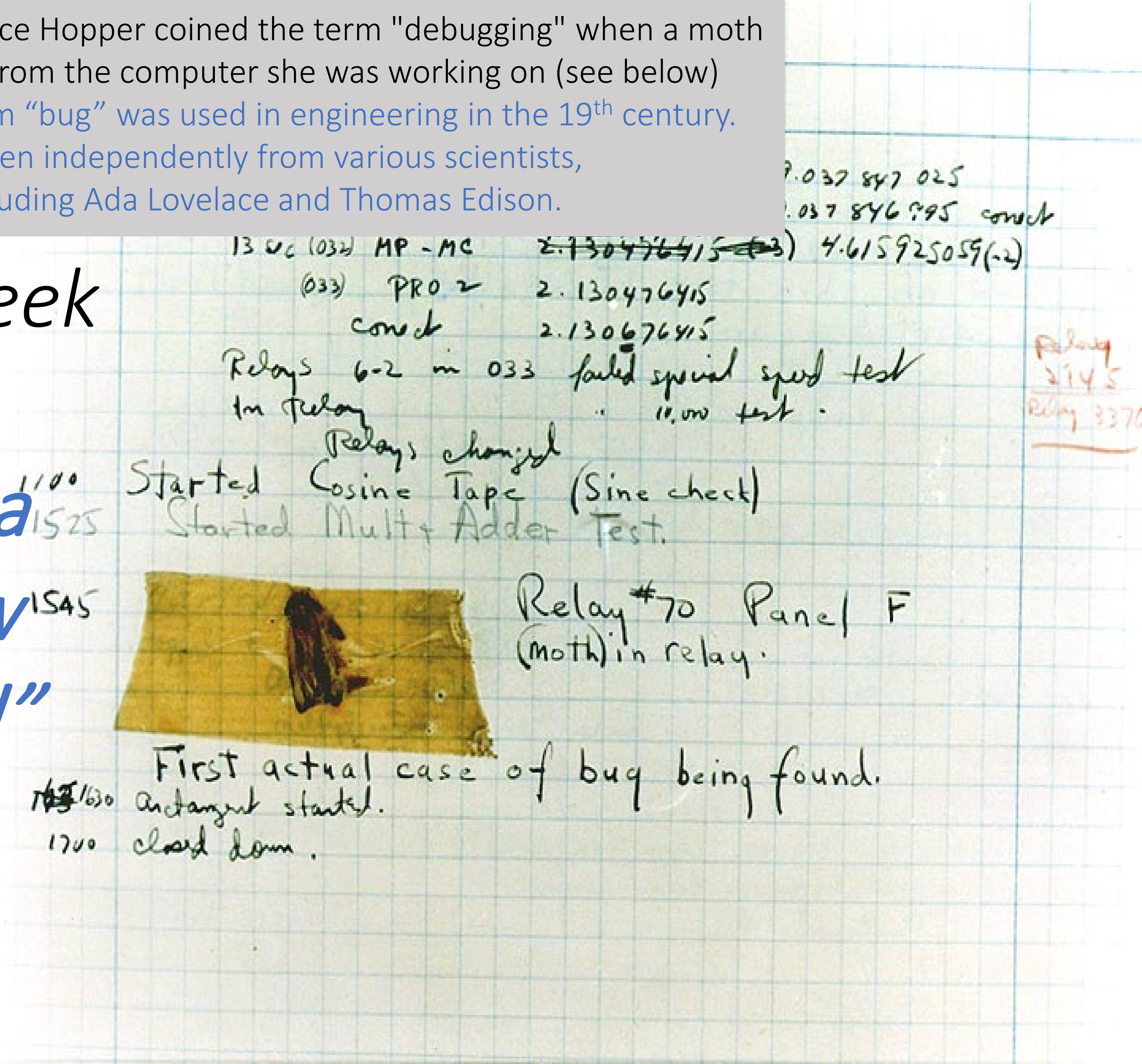
Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB



Legend: Adm. Grace Hopper coined the term "debugging" when a moth was removed from the computer she was working on (see below)

Reality: The term "bug" was used in engineering in the 19th century. As seen independently from various scientists, including Ada Lovelace and Thomas Edison.

This Week on *"Didja Know Dat?!"*



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 Mo. 10 – 11:30 AM *SSMS 4409*
  - TA Charlie Tu. 3 – 5 PM *Trailer 936*
  - TA Kunlong Tu. 5 – 7 PM *Trailer 936*

MIPS Reference Card

UCSB CS64 F19 Course Information Lecture Notes Labs Calendar

CS64, Fall 2019

Prof. Ziad Matni

Course Information

- Calendar
- Syllabus
- Demo code used in lecture
- Class grades are on Gauchospace
- List of Readings for Class
- MIPS Reference Card PDF Link
- MIPS Calling Convention

Please have this with you in lectures!

Any Questions From Last Lecture?

What We’ve Seen So Far…

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

# Q: But how do we ***print***?!?!
# A: By “talking” with the OS!!!
```

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**

`syscall` Interaction Setup

You will need:

- System call code
  - Usually placed in $v0
- Argument
  - Usually placed in $a0

(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 (copy) the value of one register into another too!**

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

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

To make sure that the register **$a0** has the value of what you want to print out (let's say it's in another register, like **$t0**), use the **move** command:

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

Used for data

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

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 now 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`

- 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

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

stdin

File I/O

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

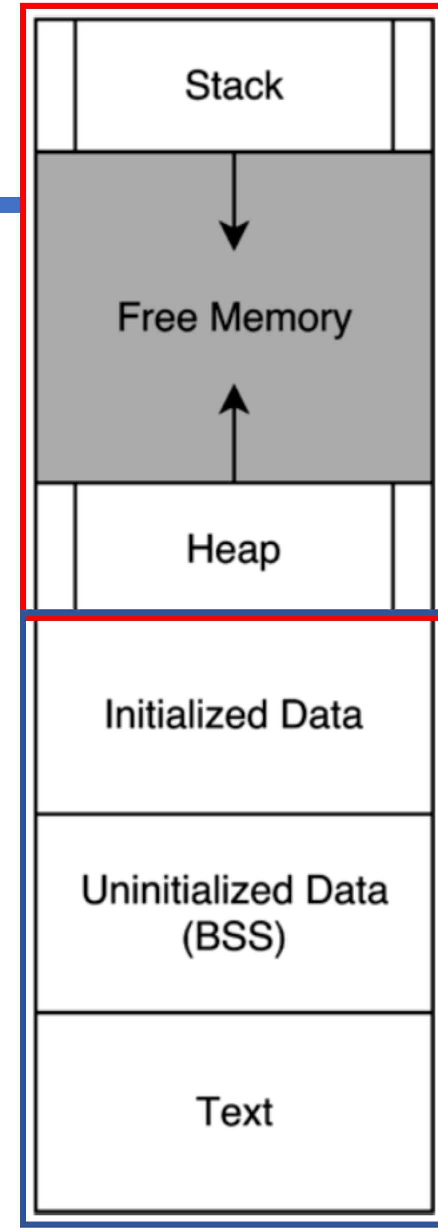
Allocated as program RUNs

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
```

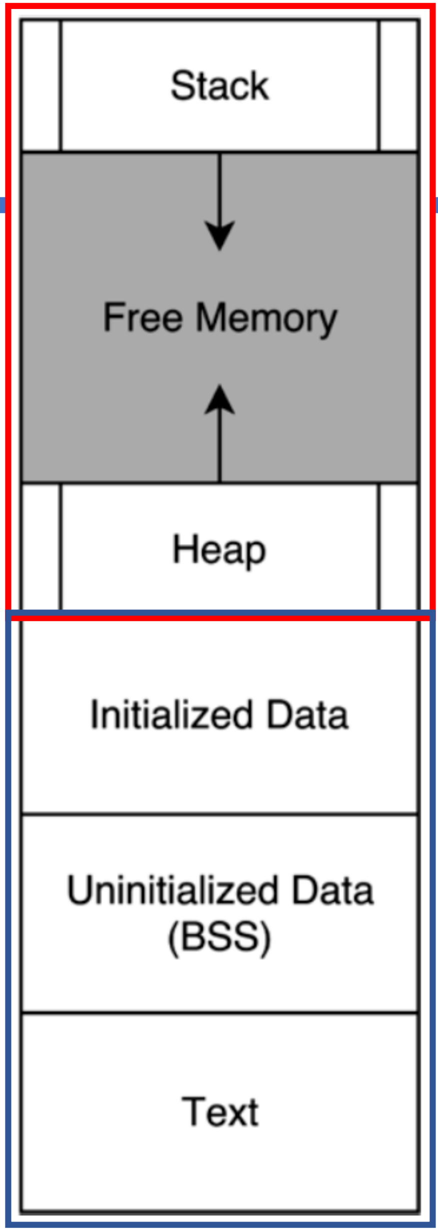
Allocated as program RUN

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

"R"

Arithmetic
Branching

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

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

"I"

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

Arithmetic	
Branching	
Memory	
Not for CS64	

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-Type Syntax

<op>   <rt>, <rs>, 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

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

Mostly used in CS64

You are not responsible for the rest of them

YOUR TO-DOs

- Do readings!
  - Check syllabus for details!
- Review ALL the demo codes
  - Available via the class website
- Work on Assignment #2
  - You have to submit it into ***Gradescope as 2 parts***
  - Due on **Wednesday, 10/16, by 11:59:59 PM**

</LECTURE>