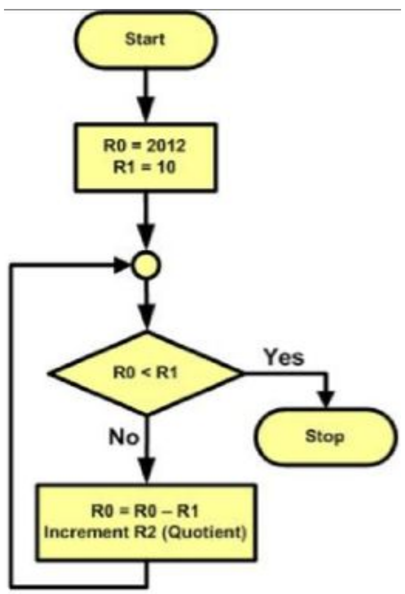
Flow Control in MIPS Assembly Language

CS 64: Computer Organization and Design Logic

Lecture #6

Winter 2020

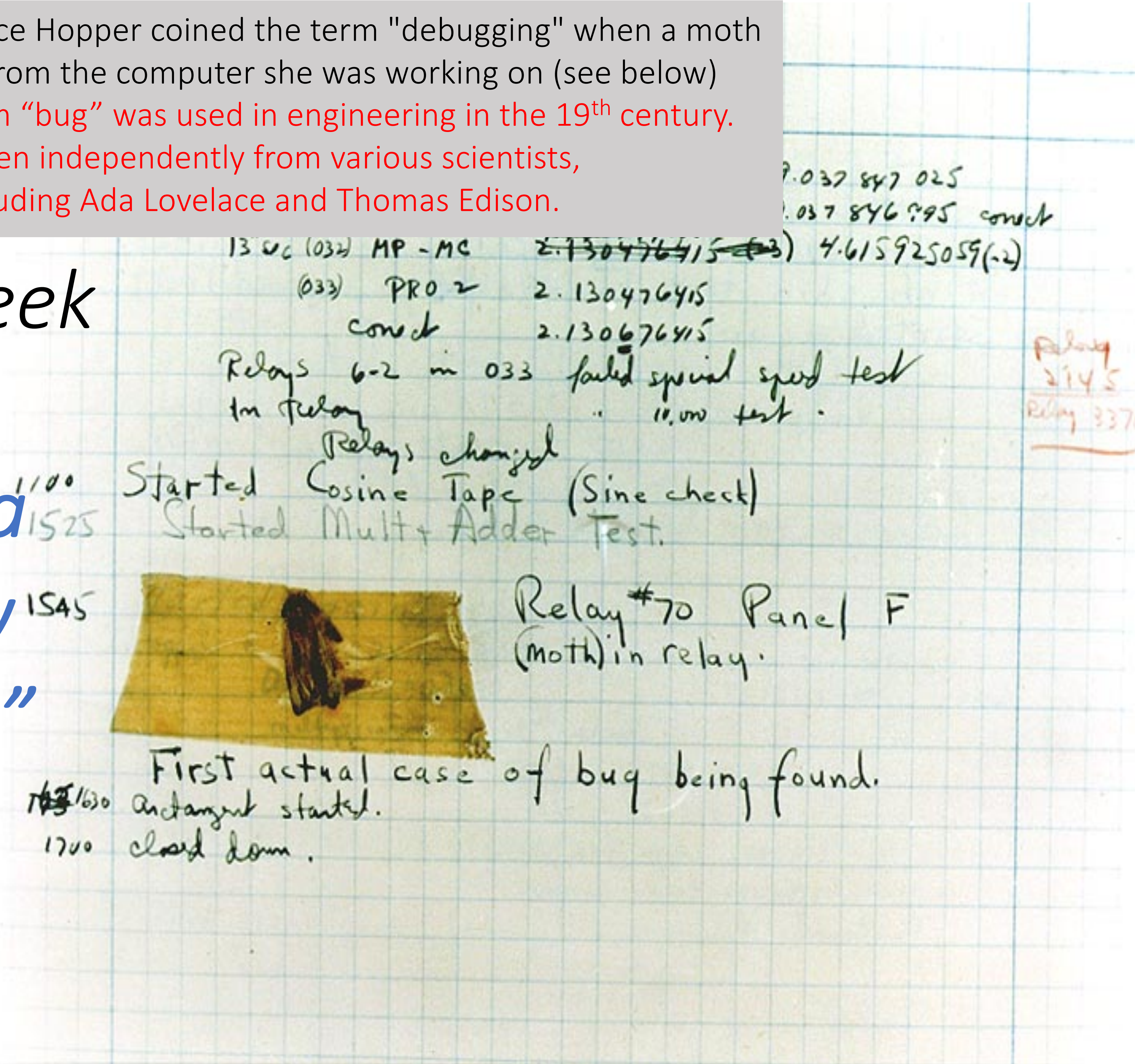
Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB



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?!”



Administrative

- How did lab03 go?

Lecture Outline

- Operand Use
- Flow Control – branching and conditionals

Any Questions From Last Lecture?

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
li $t1, 7
mult $t1, $t0
mflo $t2
```

- **`mult`** cannot be used with an ‘immediate’ value
- 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

MIPS Peculiarity: **NOR used as NOT**

- MIPS does not have NOT
- How to make a NOT function using ***NOR*** instead
- Recall: NOR = NOT OR
- Truth-Table:

A	B	A NOR B
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

Note that:

0 NOR x = NOT x

- So, in the absence of a NOT function, use a NOR with a 0 as one of the inputs!

Conditionals

- What if we wanted to do:

```
if (x == 0) { cout << “x is zero”; }
```

  - Can we write this in assembly with what we know?
    - No… we haven’t covered **if-else** (aka *branching*)

- 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 branching conditionals

- Comparing numbers:

  set-less-than (`slt`)

  - Set some register (i.e. make it "1") if a less-than comparison of some other registers is true

- Conditional execution:

  branch-on-equal (beq)
  branch-on-not-equal (bne)

  - "Go to" some other place in the code (i.e. jump)

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
```

Note the flow

Create a constant string called "**x_is_zero**"

If **$t0 != 0** go to the block labeled as "***after_print***"

(otherwise) prepare to print a string…

…and that string is located at memory address, labeled as "x_is_zero"

End the program

Loops

- How might we translate the following C++ to assembly?

```
n = 3;
sum = 0;
while (n != 0)
{
    sum += n;
    n--;
}
cout << sum;
```

n = 3; sum = 0;
while (n != 0) { sum += n; n--; }

```
.text
main:
   li $t0, 3    # n
   li $t1, 0    # running sum
loop:
   beq $t0, $zero, loop_exit
   addu $t1, $t1, $t0
   addi $t0, $t0, -1
   j loop

loop_exit:
   li $v0, 1
   move $a0, $t1
   syscall

   li $v0, 10
   syscall
```

Set up the variables in $t0, $t1

If **$t0 == 0** go to “loop_exit”

(otherwise) make $t1 the (unsigned) sum of $t1 and $t0 (i.e. **sum += n**)

decrement $t0 (i.e. **n--**)

jump to the code labeled “loop” (i.e. **repeat loop**)

prepare to print out an integer, which is inside the $t1 reg. (i.e. **print sum**)

end the program

Let’s Run More Programs!!
Using SPIM

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

DEMO !!!

These assembly code programs are made available to you via the class webpage

More Branching Examples

```
int y;
if (x == 5)
{
      y = 8;
}
else if (x < 7)
{
      y = x + x;
}
else
{
      y = -1;
}
print(y)
```

```
.text
main:   # t0: x and t1: y
     li $t0, 5      # example
     li $t2, 5      # what's
this?
     beq $t0, $t2, equal_5

     # check if less than 7
     li $t2, 7
     slt $t3, $t0, $t2
     bne $t3, $zero, less_than_7

    # fall through to final else
     li $t1, -1
     j after_branches

equal_5:
     li $t1, 8
     j after_branches
```

```
less_than_7:
     add $t1, $t0, $t0
 # could jump to after_branches,
 # but this is what we will fall
 # through to anyways

after_branches:
# print out the value in y ($t1)
     li $v0, 1
     move $a0, $t1
     syscall

     # exit the program
     li $v0, 10
     syscall
```

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)
  - We'll see how arrays are done in assembly…

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)

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

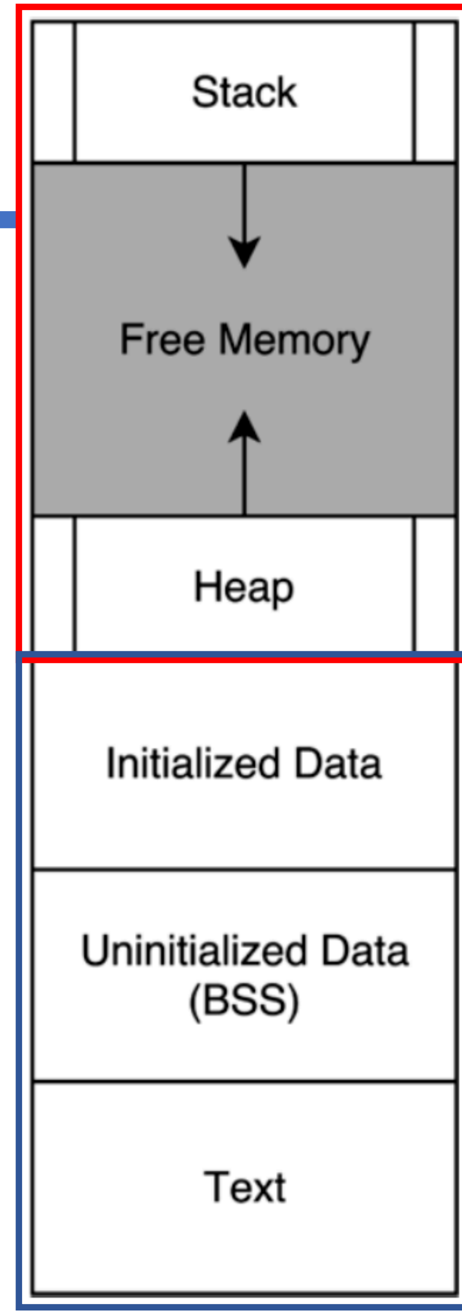
Allocated as program RUNs

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
```

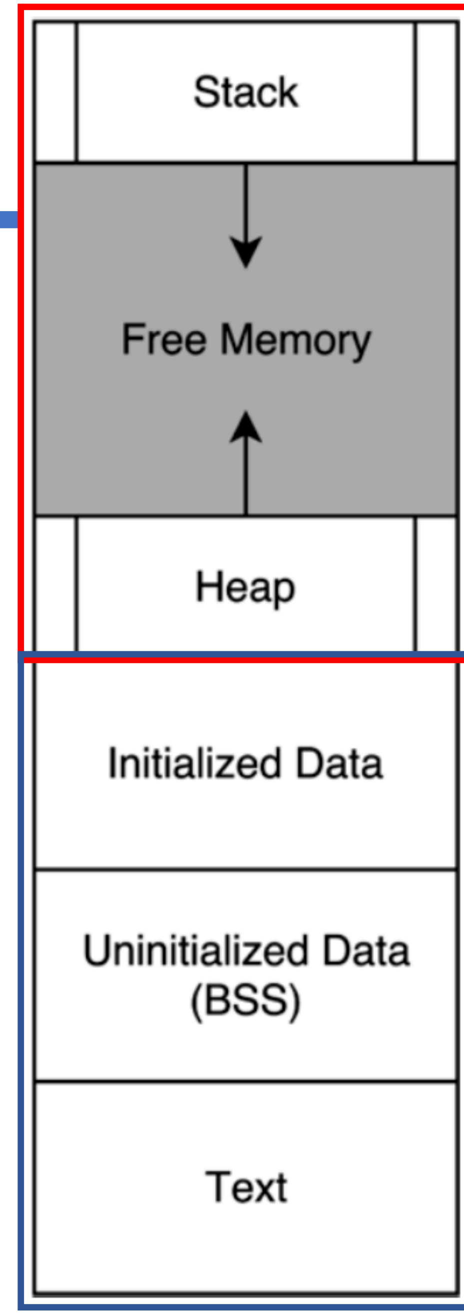
Allocated as program RUNs

Constants to be used in the program (like strings)

Allocated at program LOAD

mutable global variables

the text of the program



.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.

YOUR TO-DOs

- Do readings!
  - Check syllabus for details!
- Review ALL the demo codes
  - Available via the class website
- Work on Assignment #3

</LECTURE>