MIPS Assembly: Use of Memory and Flow Control

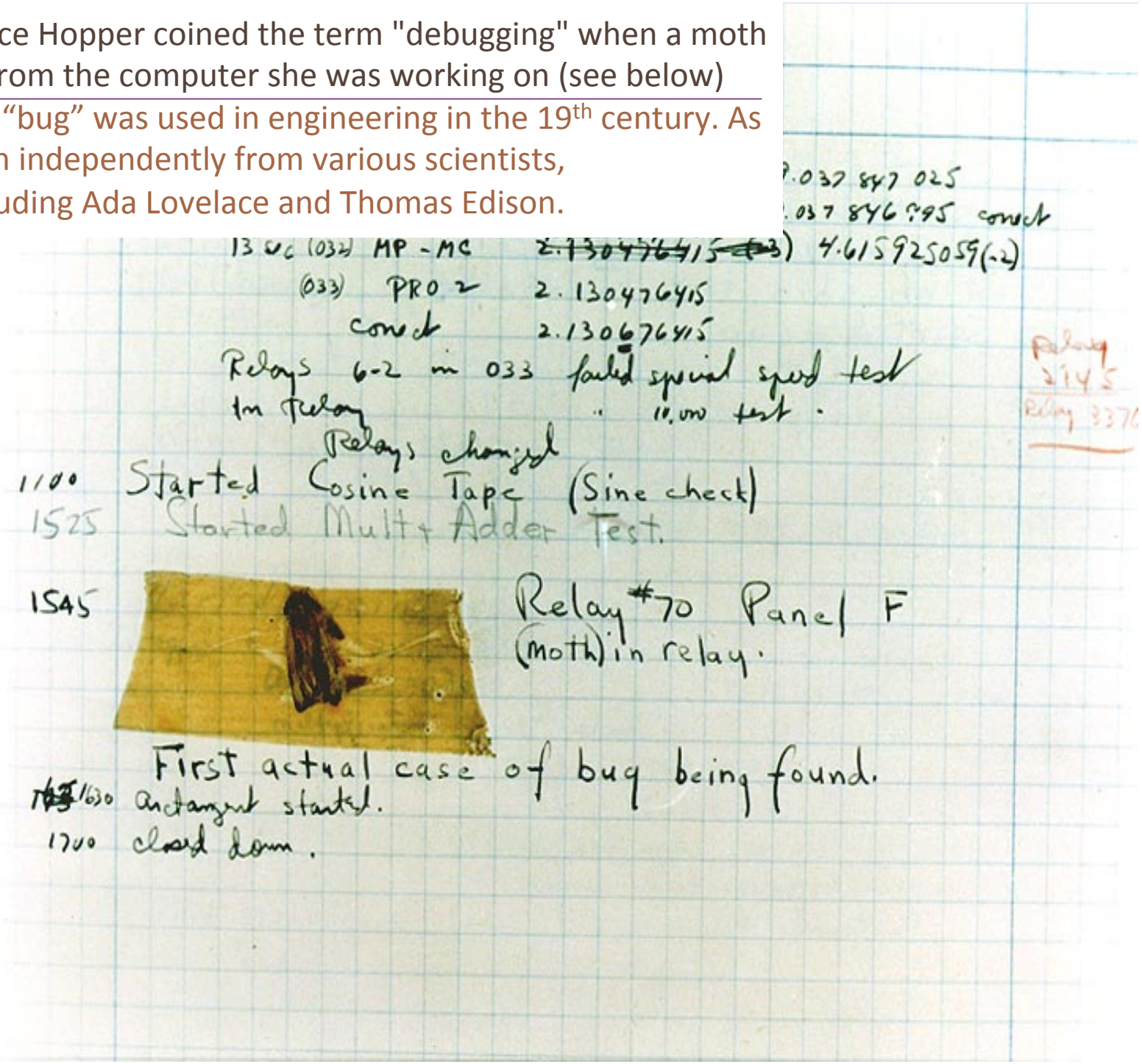
CS 64: Computer Organization and Design Logic
Lecture #5

Ziad Matni
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

- Operand Use
- **.data** Directives and Basic Memory Use
- Flow Control in Assembly
  - With Demos!
- Reading/Writing MIPS Memory

MIDTERM IS COMING!

- **Tuesday, 4/24** in this classroom
- **Starts at 11:00 AM **SHARP****
  - Please start arriving 5-10 minutes before class
- **I may ask you to change seats**
- Please bring your UCSB IDs with you

- **Closed book**: no calculators, no phones, no computers
- ***Only*** the MIPS Reference Card is allowed
- **You will write your answers on the exam sheet itself.**

What's on the Midterm?? 1/2

- Data Representation
  - Convert bin ←→ hex ←→ decimal ←→ bin
  - Signed and unsigned binaries
- Logic and Arithmetic
  - Binary addition, subtraction
    - Carry and Overflow
  - Bitwise AND, OR, NOT, XOR
  - General rules of AND, OR, XOR, using NOR as NOT
- All demos done in class
- Lab assignments 1 and 2

What's on the Midterm?? 2/2

Assembly

- Core components of a CPU
  - How instructions work
- Registers ($t, $s, $a, $v)
- Arithmetic in assembly (add, subtract, multiply, divide)
  - What's the difference between add, addi, addu, addui, etc…
- Conditionals and loops in assembly
- Conversion to and from Assembly and C/C++
- syscall and its various uses (printing output, taking input, ending program)
- **.data** and **.text** declarations
- Memory in MIPS
- Big Endian vs Little Endian
- R-type and I-type instructions
- Pseudo instructions

Depending on what we cover on Thursday

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

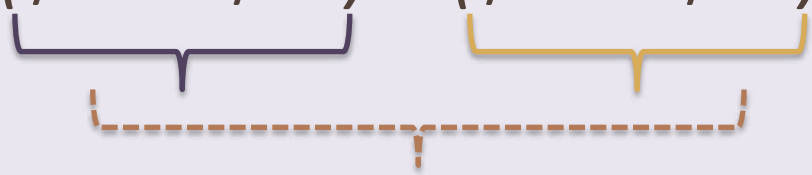
Syntax for "add"

```
add rd, rs, rt
```

destination, source1, source2

f = (g + h) – (i + j)

i.e. $s0 = ($s1 + $s2) – ($s3 + $s4)

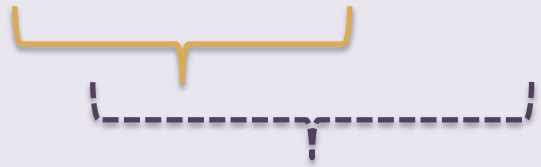


```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

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
```

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)
```

This is a 16-bit number

Global Variables

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

- Can use the **.data** directive
  - Declares variable names used in 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
```

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.

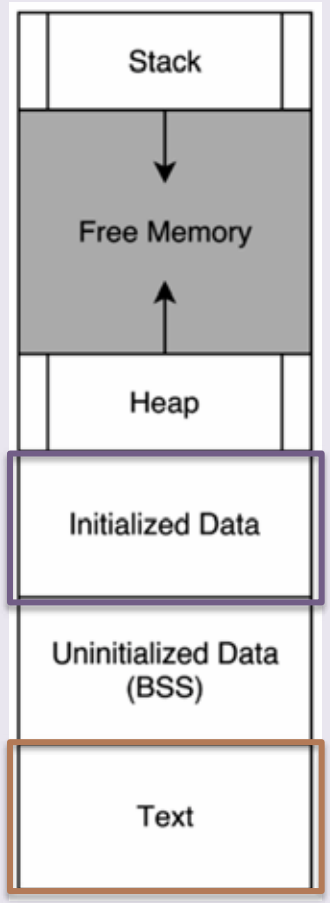
Example 3

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? →

What goes in here? →



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… 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 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 inside of "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

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

YOUR TO-DOs

- Finish assignment/Lab #2
  - Assignment due on FRIDAY

</LECTURE>