MIPS Assembly:
Use of Memory and Flow Control

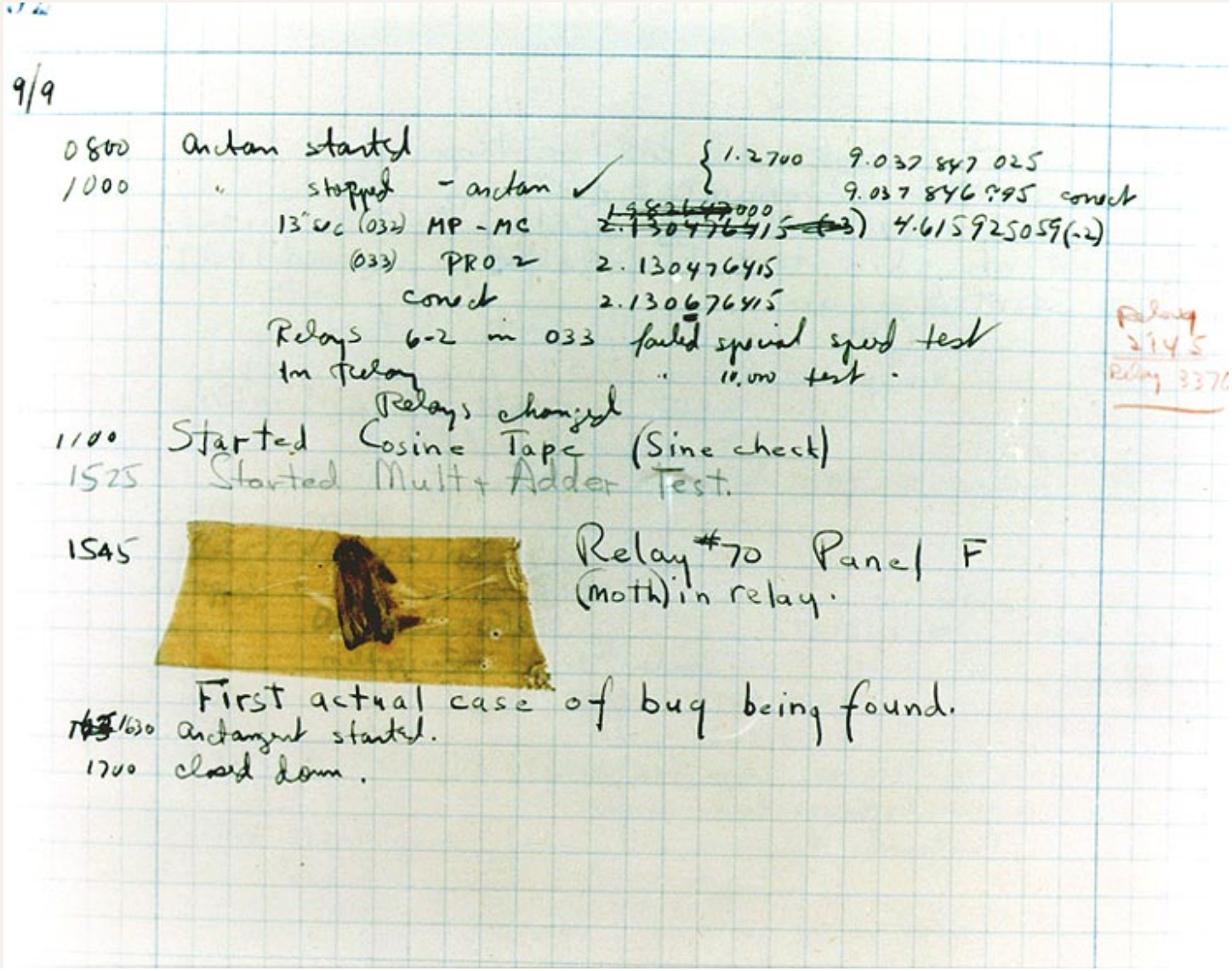
CS 64: Computer Organization and Design Logic
Lecture #5

Ziad Matni
Dept. of Computer Science, UCSB

This Week on "Didja Know Dat?!"

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.



1/31/18

Lecture Outline

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

MIPS Peculiarity:
NOR used a 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!

More Demos!

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

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)

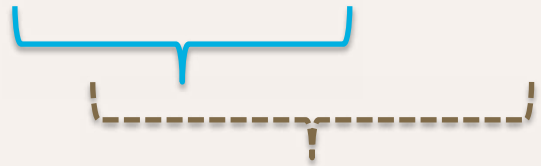
i.e. $s0 = ($s1 + $s2) – ($s3 + $s4)

```
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 GVs

- 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
var2:   .half 63         # declare a 16-bit half-word
var3:   .word 9433       # declare a 32-bit word
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

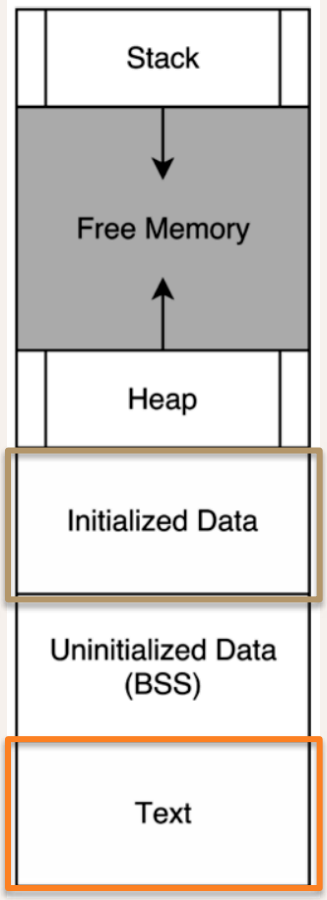
Example 3

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…

- 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`)

- Conditional execution:

  branch-on-equal (beq)

  branch-on-not-equal (bne)

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
```

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 to assembly?

```
n = 3;
sum = 0;
while (n != 0)
{
    sum = sum + n;
    n--;
}
printf(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
   la $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

- Assignment/Lab #3
  - Will post online on WEDNESDAY
  - Your lab is on THURSDAY
  - Assignment will be due on FRIDAY

</LECTURE>