Flow Control & Memory Use in MIPS Assembly Language

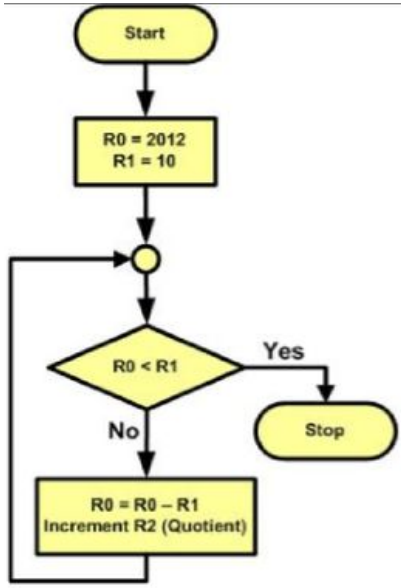
CS 64: Computer Organization and Design Logic

Lecture #6

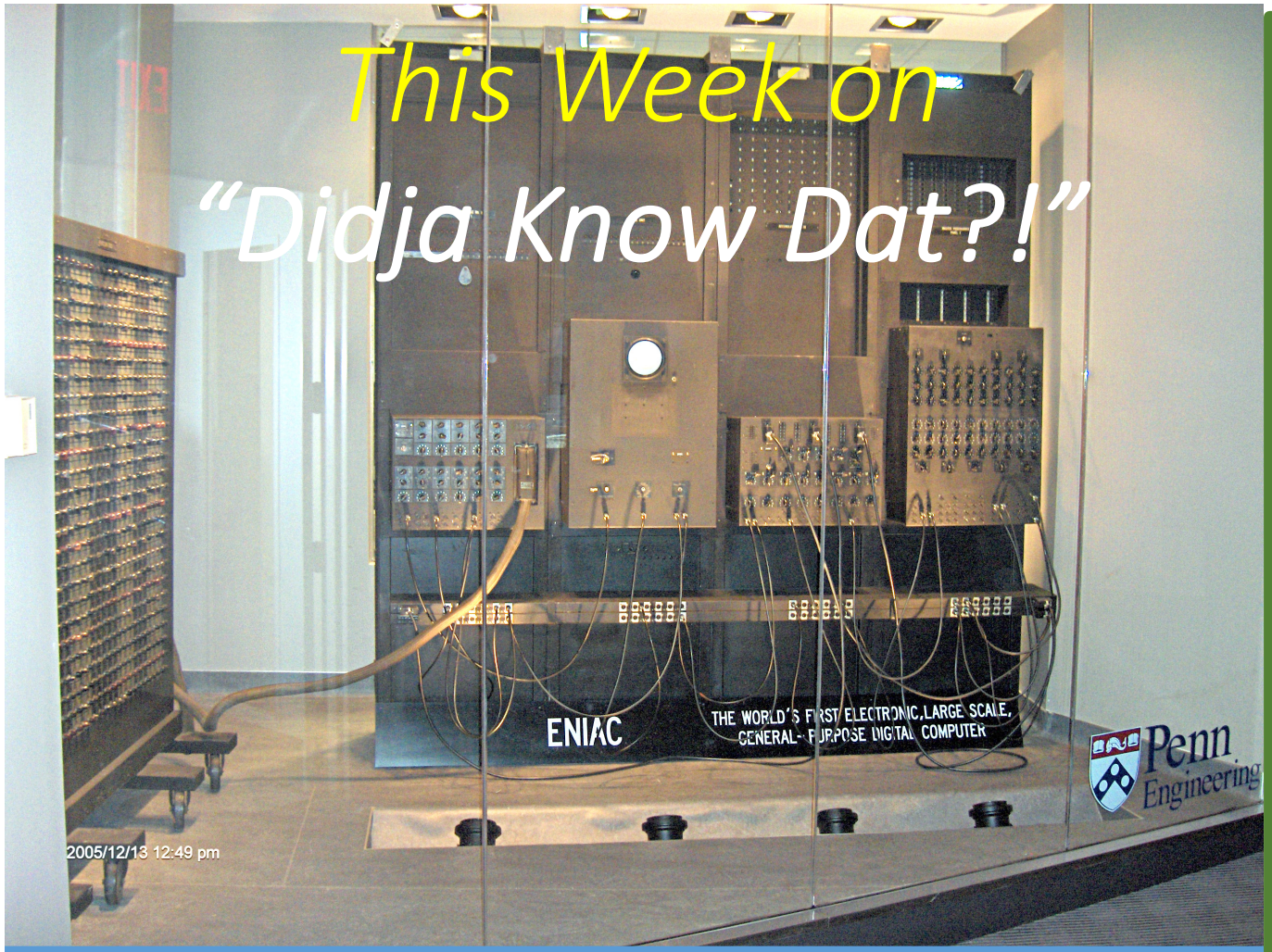
Fall 2019

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB



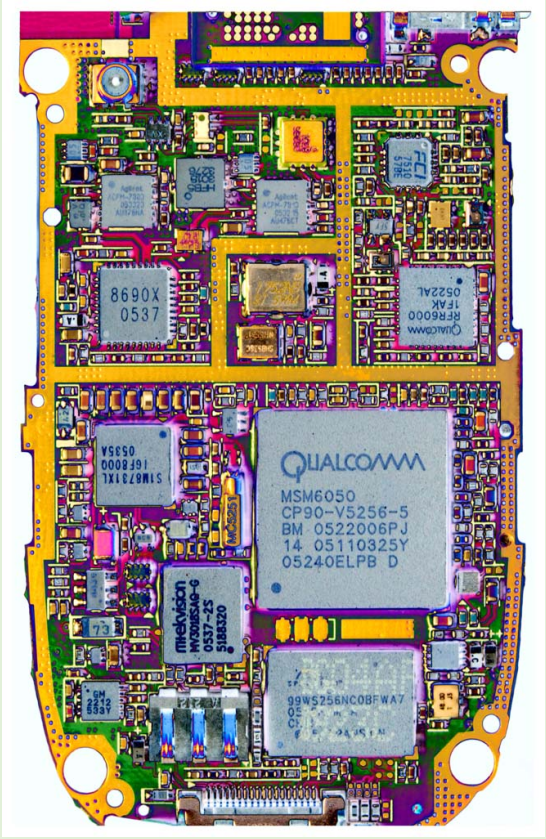
This Week on "Didja Know Dat?!"



One of the first *programmable* computers ever built for general and commercial purposes was the Electronic Numerical Integrator and Computer (ENIAC) in 1945.

It was 27 tons and took up 1800 square feet.
It used 160 kW of power (about 3000 light bulbs worth)
It cost $6.3 million in today's money to purchase.

Comparing today's cell phones (with dual CPUs), with ENIAC, we see they…



cost 17,000X less
are 40,000,000X smaller
use 400,000X less power
are 120,000X lighter
AND…
are 1,300X more powerful.

Lecture Outline

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

Any Questions From Last Lecture?

Bring Out Your MIPS Reference Cards!

Look for the following instructions:

- `nor`
- `addi`
- `beq`
- `move`

Tell me everything you can about them, based on what you see on the Ref Card!

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?

- 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

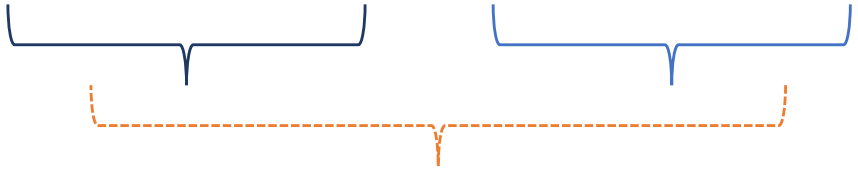
Syntax for "add"

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

destination, **source1**, **source2**

f = (g + h) – (i + j)

i.e. $s0 = ($s1 + $s2) – ($s3 + $s4)

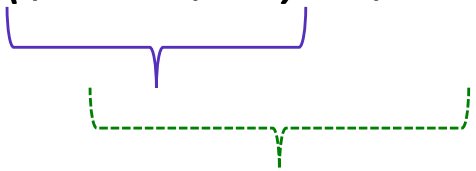


```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

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
```

Recap: The **mult** instruction

- To multiply 2 integers together:

```
li $t0, 5
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

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`

- **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, LilSebastian`

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

Example
What does this do?

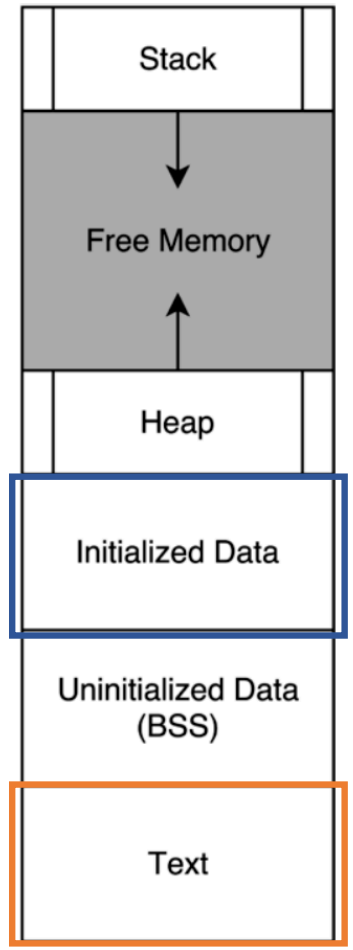
```
.data
name: .asciiz “Jimbo Jones is ”
rtn: .asciiz “ years old.\n”

.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? →

What goes in here? →

MIPS Peculiarity: NOR used as 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

YOUR TO-DOs

- Do readings!
  - Check syllabus for details!
- Review ALL the demo codes
  - Available via the class website
- Turn in Assignment #2 today!
- Work on Assignment #3
  - Due on **Wednesday, 10/23, by 11:59:59 PM**


</LECTURE>