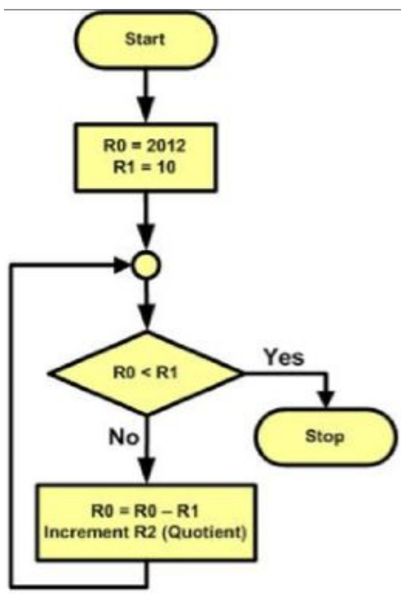
Accessing Memory in MIPS

CS 64: Computer Organization and Design Logic

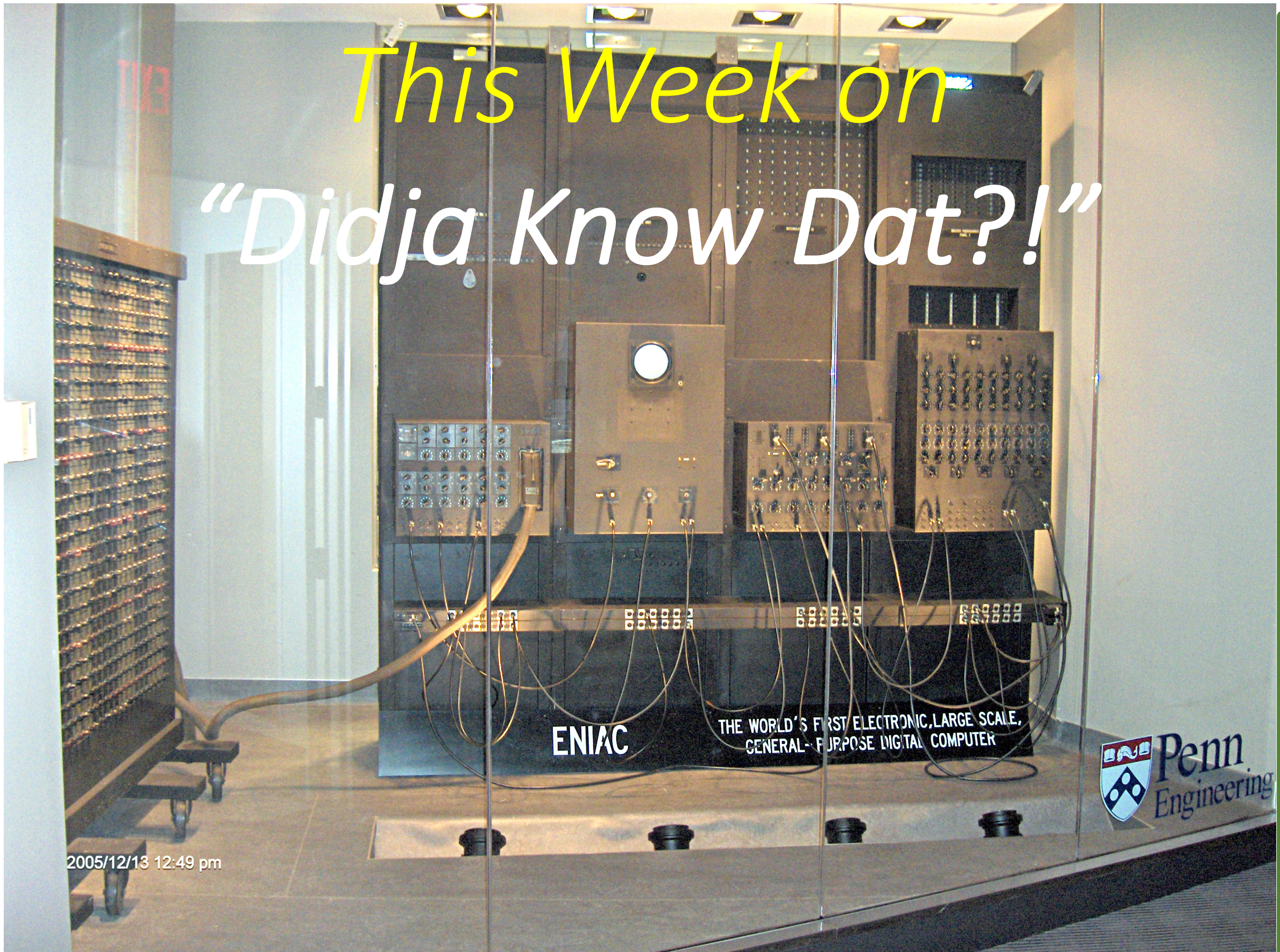
Lecture #7

Winter 2020

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



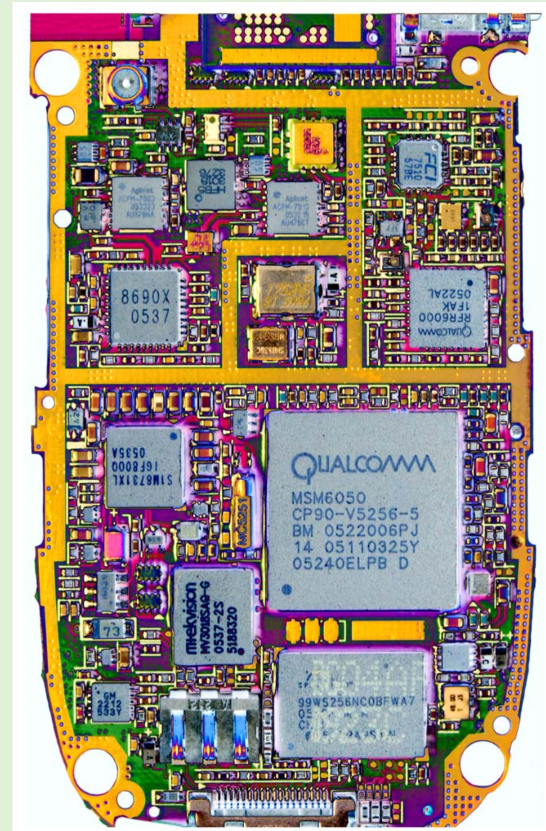
This Week on "Didja Know Dat?!"



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

- Loop Instructions
- Addressing MIPS Memory
- Global Variables
- Arrays

Any Questions From Last Lecture?

Pop Quiz!

- **You have 5 minutes to fill in the missing code. You can use your MIPS Reference Card.**
- **Fill in the 4 blank spaces :**

```
main: # assume $t0 has been declared earlier (not here)
        li $t1, 0
        li  $t2, -77            # something to compare!
        blt $t0, $t2, exit
        li $t1, 1
exit:   li $v0, 10
        syscall
```

In C++, the code would be:

```
if (t0 >= -77)
    t1 = 1;
else
    t1 = 0;
```

.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`

Very Important!

ATTN: Newbies!!!
Common Mistake!

- **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`

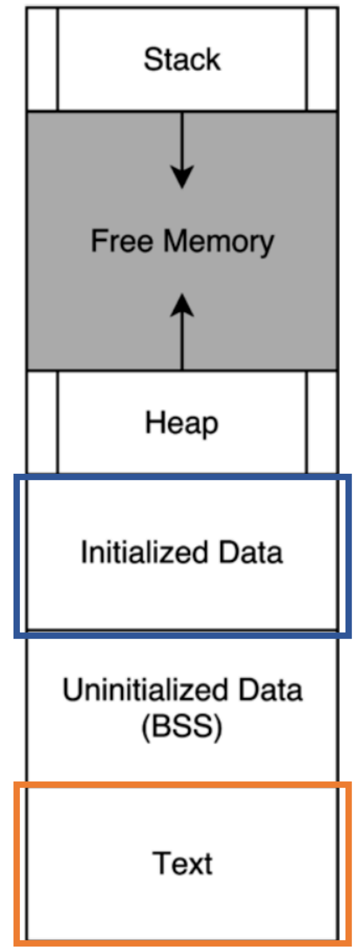
    where "LilSebastian" is a pre-defined label for something in memory (defined under the **.data** directive).

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? →



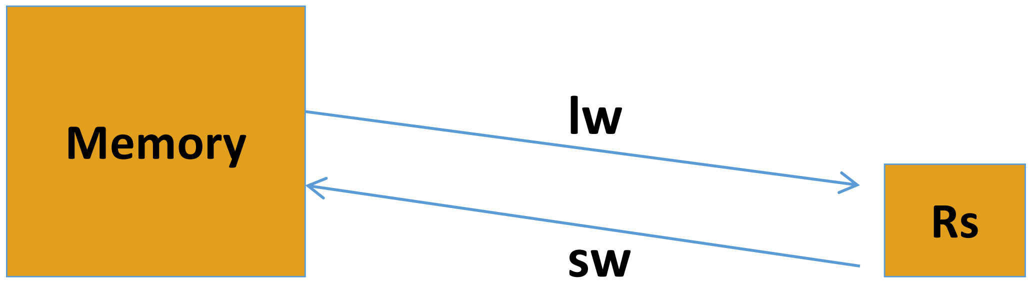
Larger Data Structures

- 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)

Accessing Memory

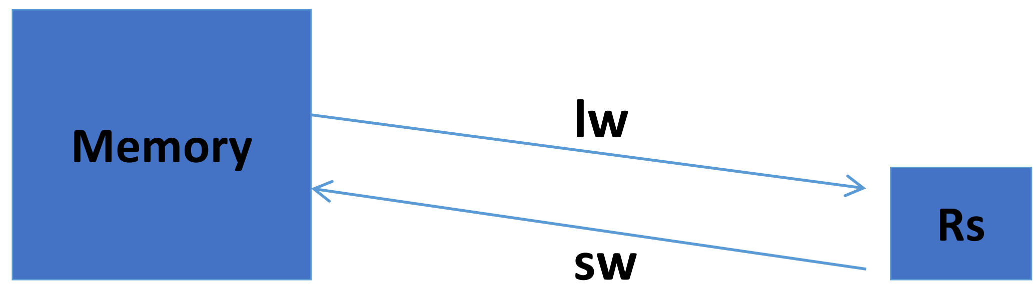
- Two base instructions:
  - load-word (**lw**) from memory to registers
  - store-word (**sw**) from registers to memory



- MIPS lacks instructions that do more with memory than access it (e.g., retrieve something from memory and then add)
  - Operations are done step-by-step
  - Mark of RISC architecture

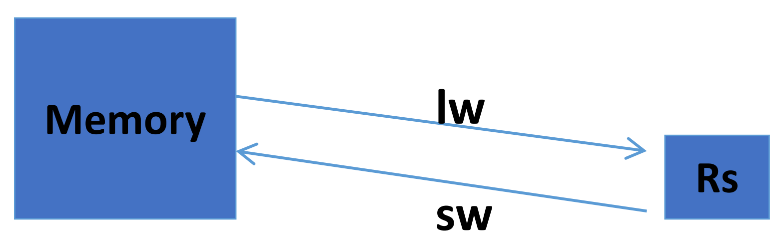
Example 4
What does this do?

```
.data
num1: .word 42
num2: .word 7
num3: .space 1

.text
main:
    lw $t0, num1
    lw $t1, num2
    add $t2, $t0, $t1
    sw $t2, num3

    li $v0, 1
    lw $a0, num3
    syscall

    li $v0, 10
    syscall
```



Example 4

```
.data
num1: .word 42   # define 32b w/ value = 42
num2: .word 7          # define 32b w/ value = 7
num3: .space 1   # define one (1) 32b space

.text
main:
     lw $t0, num1          # load what’s in num1 (42) into $t0
     lw $t1, num2          # load what’s in num2 (7) into $t1
     add $t2, $t0, $t1     # ($t0 + $t1) → $t2
     sw $t2, num3       # load what’s in $t2 (49) into num3 space

     li $v0, 1
     lw $a0, num3     # put the number you want to print in $a0
     syscall          # print integer

     li $v0, 10       # exit
     syscall
```



Addressing Memory

- If you're not using the **.data** declarations, then you need *starting addresses* of the data in memory with ***lw*** and ***sw*** instructions

Example: `lw $t0, 0x0000400A` ← not a real address, just looks like one...

Example: `lw $t0, 16($s0)`

- 1 word = 32 bits (in MIPS)
  - So, in a 32-bit unit of memory, that's 4 bytes
  - Represented with 8 hexadecimals 8 x 4 bits = 32 bits... checks out...

- MIPS addresses sequential memory addresses, but not in "words"
  - Addresses are in Bytes instead
  - MIPS words *must* start at addresses that are multiples of 4
  - Called an ***alignment restriction***

Global Variables

Recall:

- Typically, global variables are placed directly in memory, not registers

- **lw** and **sw** for **load word** and **save word**
  - **lw ≠ la ≠ move !!!**
  - Syntax:

    lw ***register_destination***, **N**(***register_with_address***)

    Where **N = offset of address in bytes**

- Let's take a look at: ***access_global.asm***

access_global.asm

Load Address (la) and Load Word (lw)

```
.data
myVariable: .word 42
.text
main:
      la $t0, myVariable       ← WHAT'S IN $t0??
      lw $t1, 0($t0)           ← WHAT DID WE DO HERE??

      li $v0, 1
      move $a0, $t1
      syscall                  ← WHAT SHOULD WE SEE HERE??
```

$t0 = &myVariable

access_global.asm

Store Word (sw) (…continuing from last page…)

```
li $t1, 5
sw $t1, 0($t0)          ← WHAT'S IN $t0 AGAIN??

li $t1, 0
lw $t1, 0($t0)          ← WHAT DID WE DO HERE??

li $v0, 1
move $a0, $t1
syscall                 ← WHAT SHOULD WE SEE HERE??
```

Arrays

- Question:

As far as memory is concerned, what is the *major difference* between an **array** and a **global variable**?

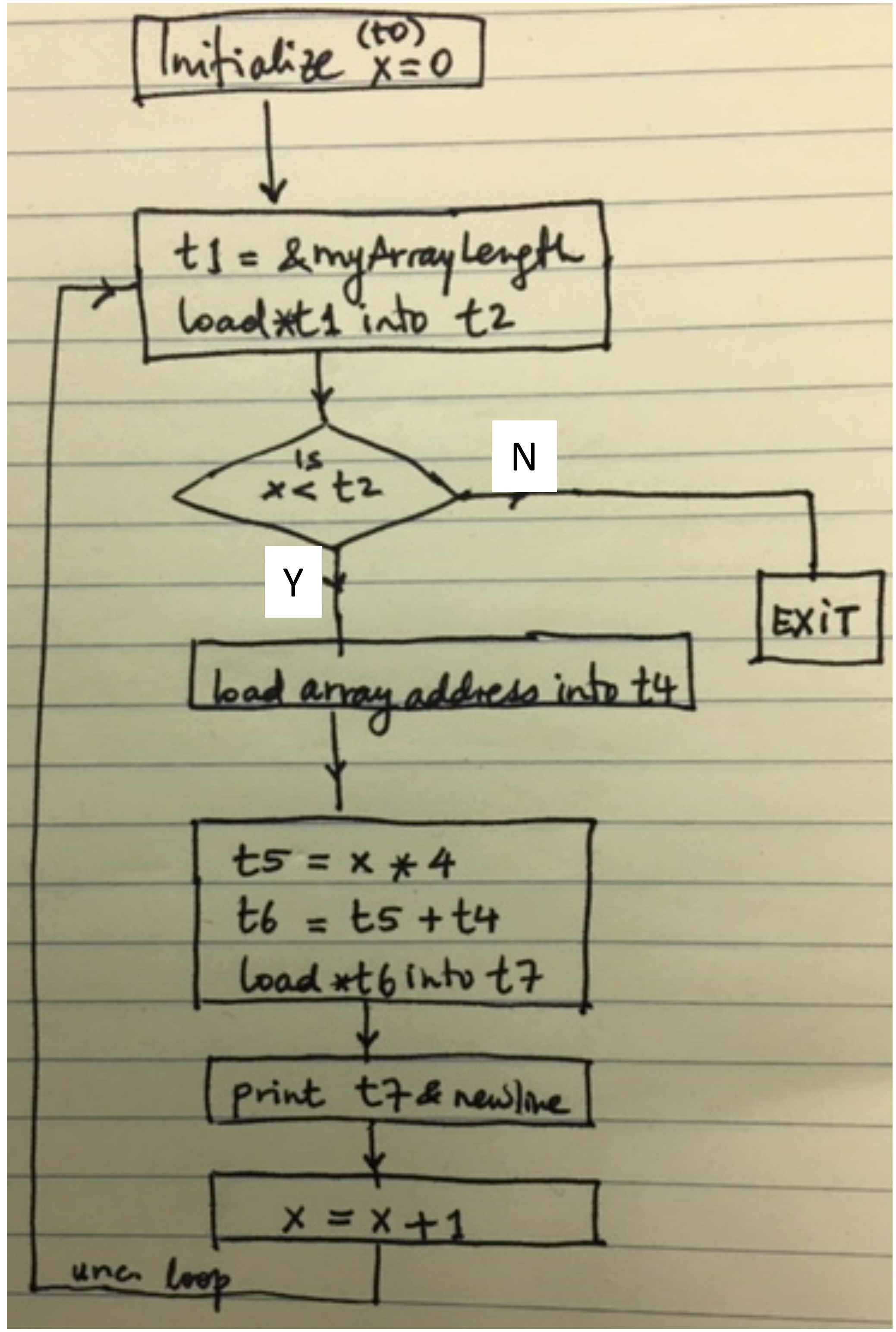
  - Arrays contain multiple elements

- Let’s take a look at:
  - print_array1.asm
  - print_array2.asm
  - print_array3.asm

print_array1.asm

```
int myArray[] = {5, 32, 87, 95, 286, 386};
int myArrayLength = 6;
int x;

for (x = 0; x < myArrayLength; x++)
{
    print(myArray[x]);
    print("\n");
}
```

Flow Chart for **print_array1**



```
# C code:
# int myArray[] =
#     {5, 32, 87, 95, 286, 386}
# int myArrayLength = 6
# for (x = 0; x < myArrayLength; x++) {
#   print(myArray[x])
#   print("\n") }
.data
newline: .asciiz "\n"
myArray: .word 5 32 87 95 286 386
myArrayLength: .word 6

.text
main:
        # t0: x
        # initialize x
        li $t0, 0
loop:
        # get myArrayLength, put result in $t2
        # $t1 = &myArrayLength
        la $t1, myArrayLength
        lw $t2, 0($t1)

        # see if x < myArrayLength
        # put result in $t3
        slt $t3, $t0, $t2
        # jump out if not true
        beq $t3, $zero, end_main

        # get the base of myArray
        la $t4, myArray

        # figure out where in the array we need
        # to read from. This is going to be the array
        # address + (index << 2). The shift is a
        # multiplication by four to index bytes
        # as opposed to words.
        # Ultimately, the result is put in $t7
        sll $t5, $t0, 2
        add $t6, $t5, $t4
        lw $t7, 0($t6)

        # print it out, with a newline
        li $v0, 1
        move $a0, $t7
        syscall
        li $v0, 4
        la $a0, newline
        syscall

        # increment index
        addi $t0, $t0, 1

        # restart loop
        j loop

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

print_array2.asm

- Same as print_array1.asm, ***except that*** in the assembly code, we lift redundant computation out of the loop.
- This is the sort of thing a decent compiler (**clang** or **gcc** or **g++**, for example) will do with a HLL program
- Your homework: **Go through this assembly code!**

print_array3.asm

```
int myArray[]
    = {5, 32, 87, 95, 286, 386};
int myArrayLength = 6;
int* p;

for (p = myArray; p < myArray + myArrayLength; p++)
{
    print(*p);
    print("\n");
}
```

Your homework: **Go through this assembly code!**

YOUR TO-DOs

- Do readings!
  - Check syllabus for details!
- Review ALL the demo codes
  - Available via the class website
- Assignment #4 for next lab!

</LECTURE>