Memory Topics in MIPS: Addressing, Global Vars & Arrays

CS 64: Computer Organization and Design Logic
Lecture #7
Winter 2019

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

This Week on *“Didja Know Dat?!”*

Steve Wozniak and Steve Job’s first commercial venture was the **Apple 1** in **1976** using an **8-bit** **MOS 6502 CPU**. It was built for $500 and initially **sold for $666.66** because Wozniak “*liked repeating digits*” (about $2900 in today’s dollars). Keyboard and TV not included. They sold about 200 of them in 10 months, thus assuring the continuation of their company.

Previously, the only other popular “personal” computer was the Altair 8800, which you had to operate with switches!

Administrative

- Lab 4 starts Thursday (Due Monday)
- Midterm Exam on Feb. 5th (Next Week Tue.)

What's on the Midterm?

What's on It?

- Everything we've done so far from start to end of this week

What Should I Bring?

- Your pencil(s), eraser, MIPS Ref. Card
- THAT'S ALL!

What Else Should I Do?

- Come to the classroom 5-10 minutes EARLY
- I will have some of you re-seated
- Bring your UCSB ID

Lecture Outline

- 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 ________________
       blt _________________________
       li $t1, 1
exit:  ________________________
       ________________________
```

In C++, the code would be:

```
if (t0 >= 5)
   t1 = 1;
else
   t1 = 0;
```

Pop Quiz Answers!

- 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, 5          # something to compare!
       blt $t0, $t2, exit
       li $t1, 1;
exit:  li $v0, 10
       syscall
```

In C++, the code would be:

```
if (t0 >= 5)
   t1 = 1;
else
   t1 = 0;
```

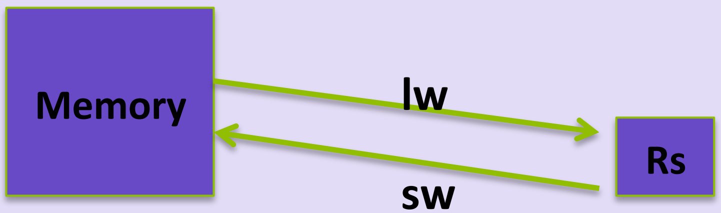
Example 4

```
.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…

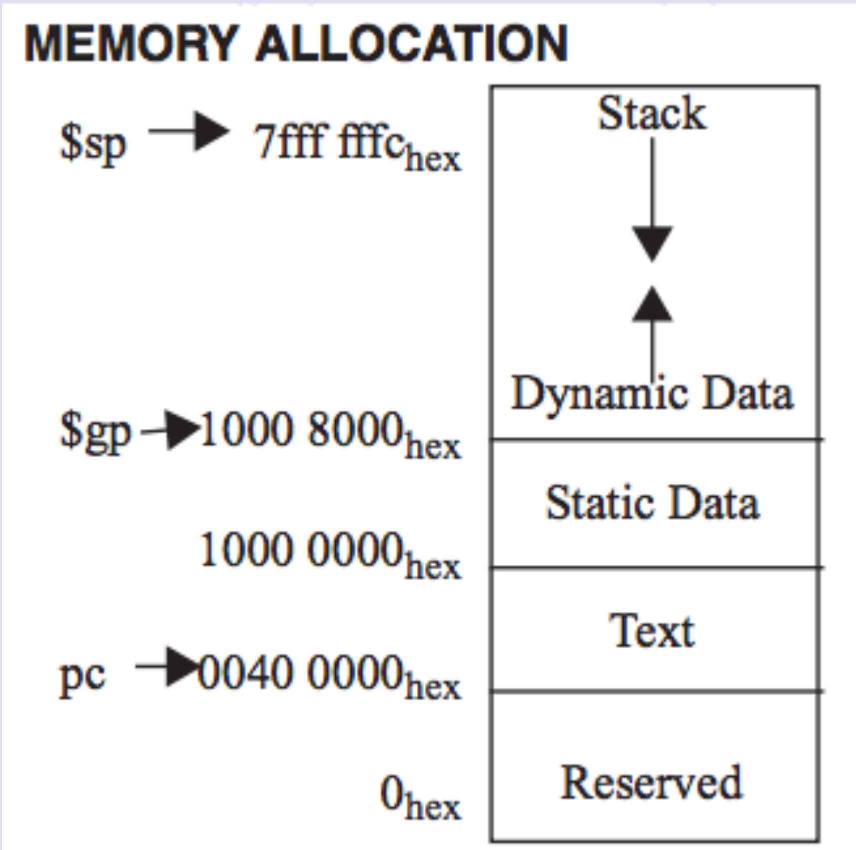
Example: `lw $t0, 16($s0)`

- 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***

Memory Allocation Map

How much memory does a programmer get to directly use in MIPS?



This is found on your ***MIPS Reference Card***

NOTE:

Not all memory addresses can be accessed by the programmer.

Although the address space is 32 bits, the top addresses from **0x80000000** to **0xFFFFFFFF** are not available to user programs. They are used mostly by the OS.

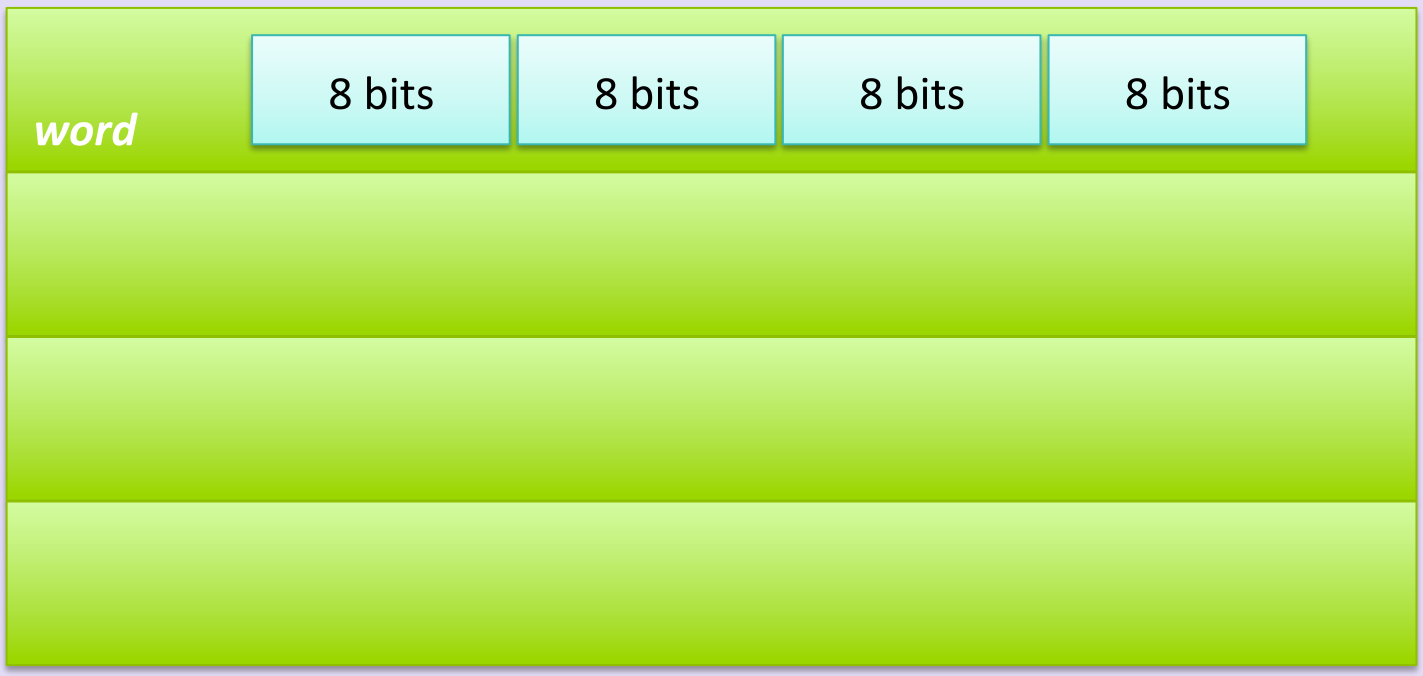
Mapping MIPS Memory

(say that 10 times fast!)

- Imagine computer memory like a big array of words
- Size of computer memory is:

  2^{32} = 4 Gbits, or 512 MBytes (MB)

  – We only get to use 2 Gbits, or 256 MB
  – That's (256 MB/ groups of 4 B) = 64 million words



MIPS Computer Memory Addressing Conventions

	1A	80	C5	29
	0x0000	0x0001	0x0002	0x0003
	52	00	37	EE
A	0x0004	0x0005	0x0006	0x0007
→	B1	11	1A	A5
	0x0008	0x0009	0x000A	0x000B

MIPS Computer Memory Addressing Conventions

or…

1A	80	C5	29
0x0003	0x0002	0x0001	0x0000
52	00	37	EE
0x0007	0x0006	0x0005	0x0004
B1	11	1A	A5
0x000B	0x000A	0x0009	0x0008

B

←

A Tale of 2 Conventions…

BIG END (MSByte) gets addressed first

← BIG ENDIAN

1A	80	C5	29
0x0000	0x0001	0x0002	0x0003
52	00	37	EE
0x0004	0x0005	0x0006	0x0007
B1	11	1A	
0x0008	0x0009	0x000A	

LITTLE END (LSByte) gets addressed first

LITTLE ENDIAN →

1A	80	C5	29
0x0003	0x0002	0x0001	0x0000
52	00	37	EE
0x0007	0x0006	0x0005	0x0004
B1	11	1A	A5
0x000B	0x000A	0x0009	0x0008

The Use of Big Endian vs. Little Endian

Origin: Jonathan Swift (author) in "Gulliver's Travels".

Some people preferred to eat their hard boiled eggs from the "little end" first (thus, little endians), while others prefer to eat from the "big end" (i.e. big endians).

- MIPS users typically go with Big Endian convention
  - MIPS allows you to program "endian-ness"
- Most Intel processors go with Little Endian…
- It's just a convention – it makes no difference to a CPU!

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:                    $t0 = &myVariable
    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??
```

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

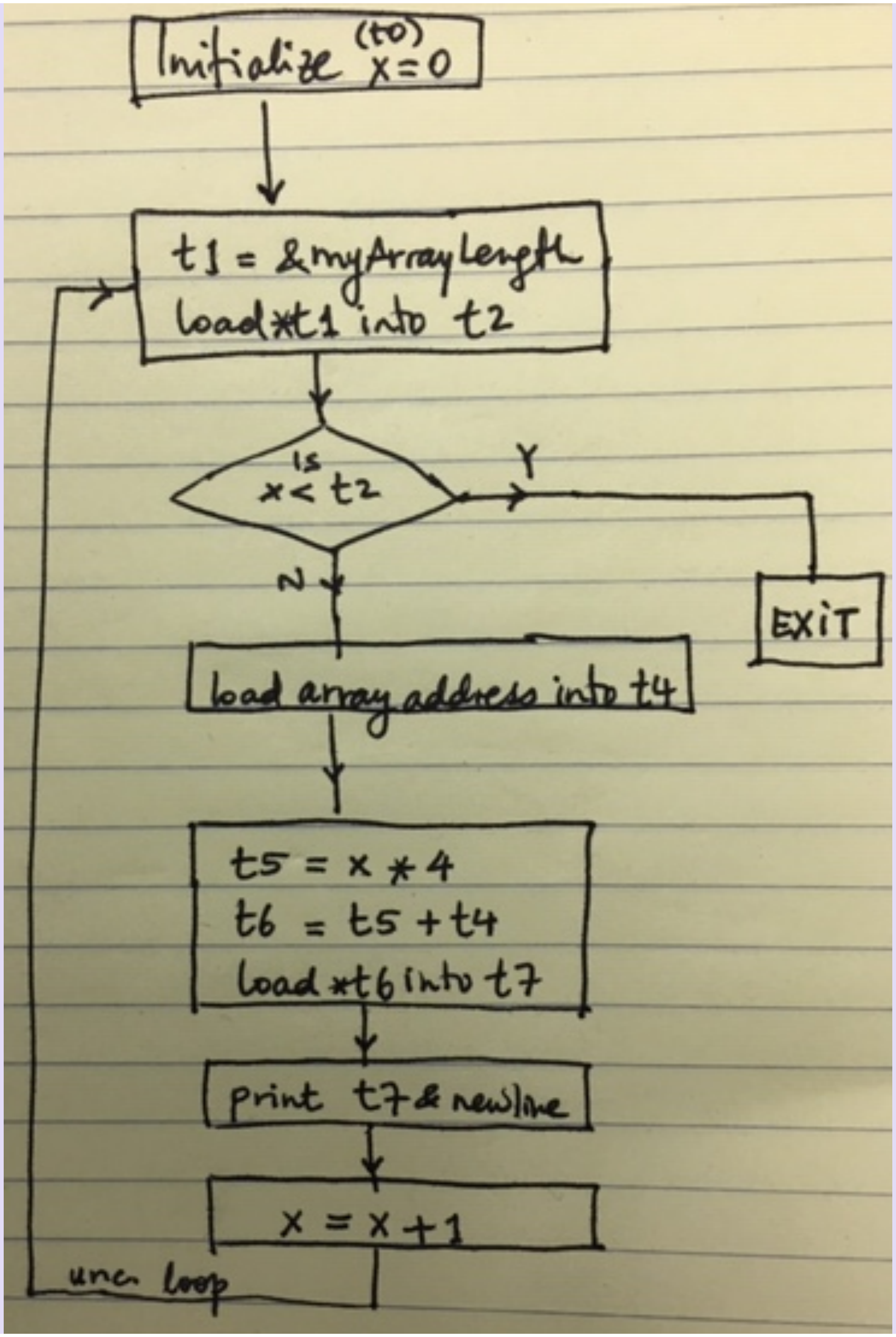
- Let's take a look at:
  - print_array1.asm
  - print_array2.asm
  - print_array3.asm

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

- Review ALL the demo codes
  - Available via the class website
- Lab 4 on Thursday!

</LECTURE>