MIPS Addressing

CS 64: Computer Organization and Design Logic
Lecture #8
Fall 2019

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

This Week on
“Didja Know Dat?!”

Xerox PARC (the research arm of the main company) invented the **first GUI** in the **early 1970s** and developed the Alto Computer to show it off, along with the **first mouse** input device AND the **first Ethernet** communication port, but Xerox thought it was all useless (how could those things sell copy machines??)

Steve Jobs and his *frenemy* **Bill Gates** took a tour of Xerox PARC in the early 80s, looking for new ideas. They were shown all of this and were told Xerox wouldn’t care much if anyone used the tech!

Right away, Jobs went on to invent the **Macintosh Computer** (which had the first ever commercial GUI-based OS + mouse) & Gates went on to develop **Windows OS** (which quickly overtook Mac OS sales).

Moral of the Story? Don’t be shortsighted like Xerox…

Administrative

- Lab 3 due today!

- Lab 4 posted later today – Lab on Friday!

Lecture Outline

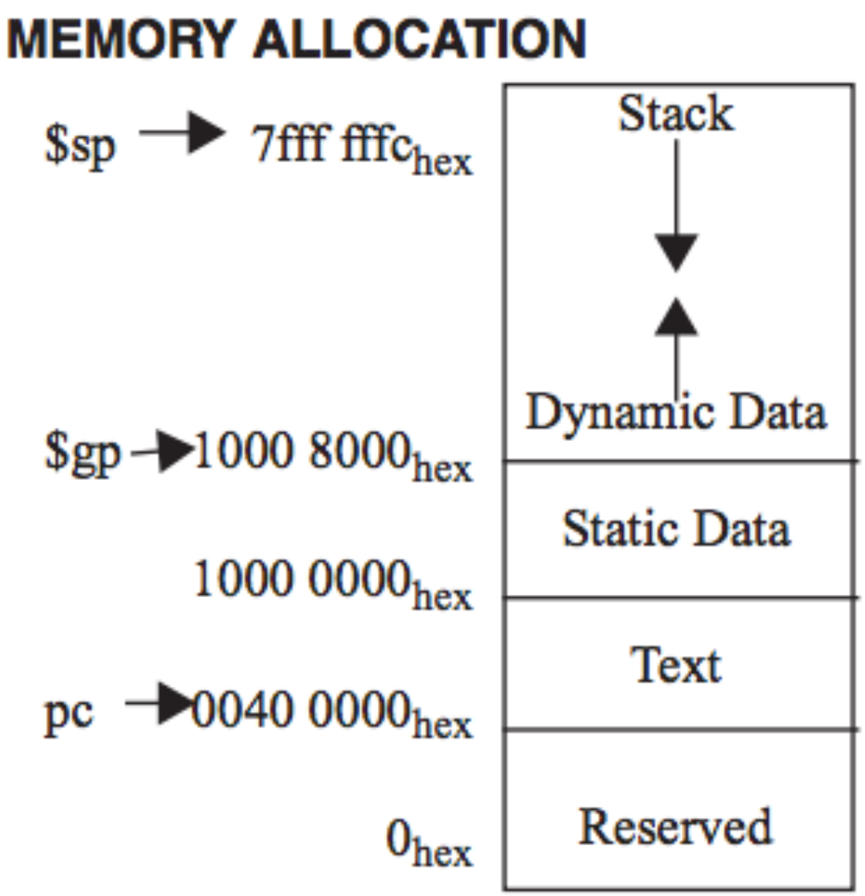
- MIPS Instructions
  - How they are represented

- Overview of Functions in MIPS

Any Questions From Last Lecture?

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.

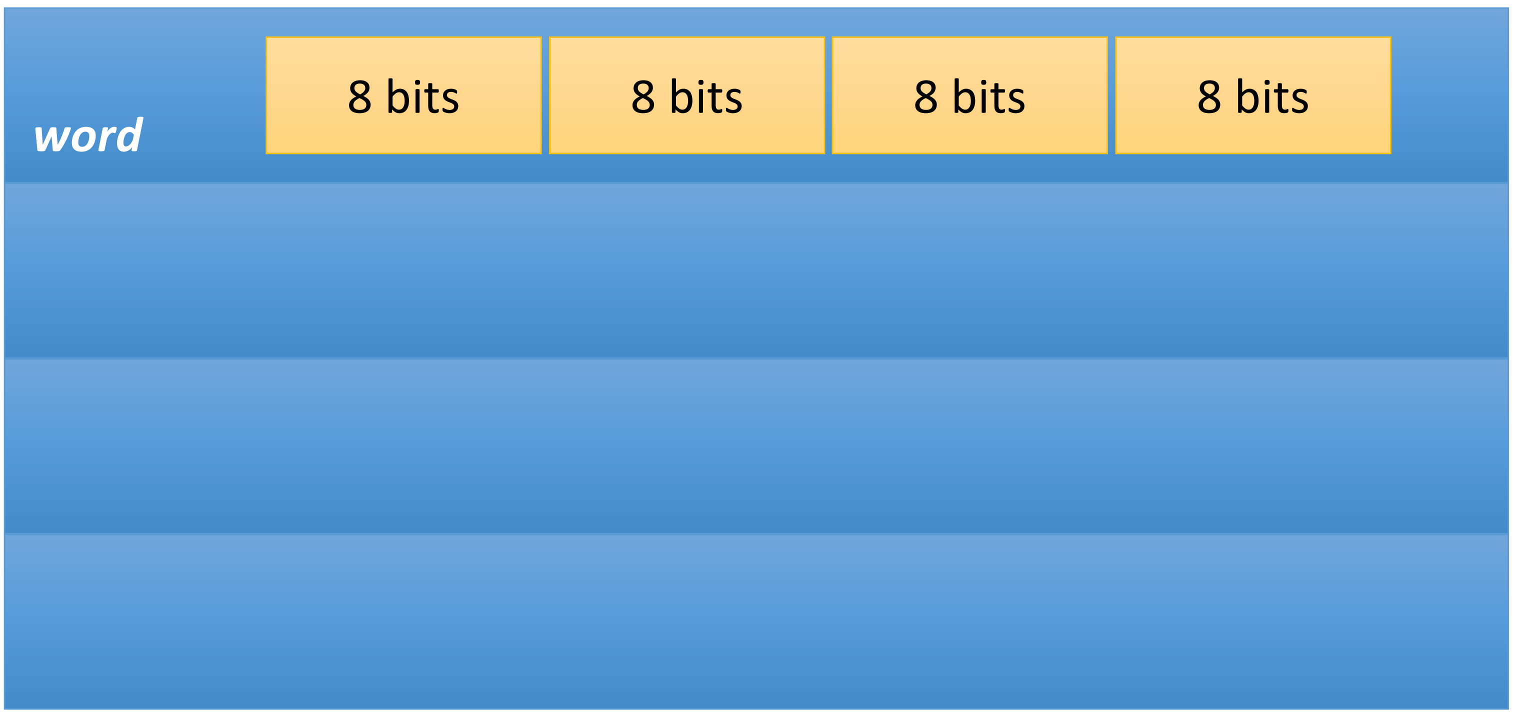
Although the address space is 32 bits, the top addresses from **0x80000000** to **0xFFFFFFFF** are not available to user programs. They are used mostly by the OS.

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:
      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

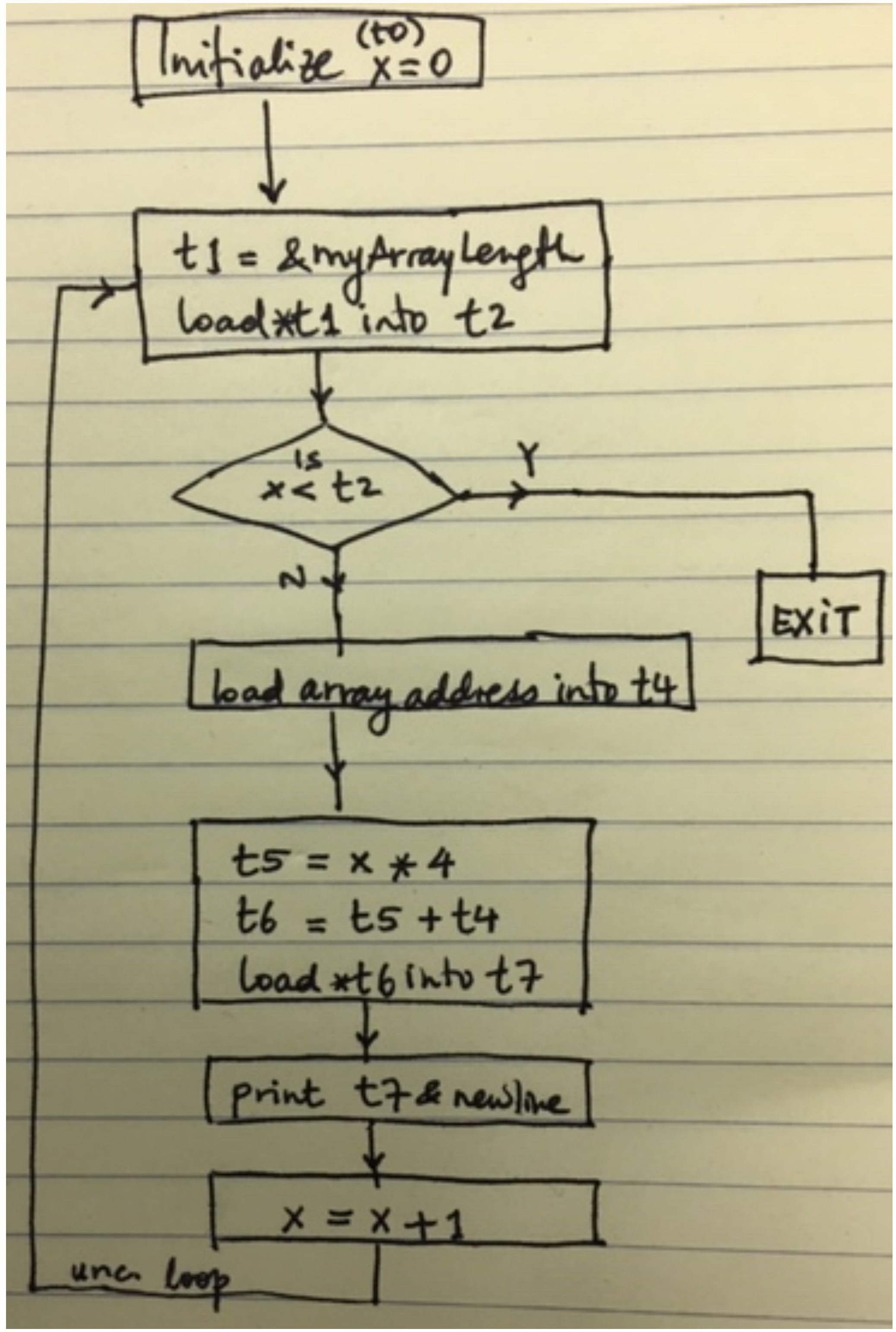
- Let's take a look at:
  - print_array1.asm
  - print_array2.asm
  - print_array3.asm

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

- Turn in Assignment #3
- Work on Assignment #4
  - Due on **Wednesday, 10/30, by 11:59:59 PM**

</LECTURE>