MIPS Assembly: More on Memory

CS 64: Computer Organization and Design Logic
Lecture #7

Ziad Matni
Dept. of Computer Science, UCSB

Looking for Mandarin bilinguals!

If you are a native Mandarin speaker, please consider participating in our study. This study looks at perceptions of Mandarin speakers. You will be entered into a lottery where you can earn $100 VISA Card.

Please contact **ahansia**@umail.ucsb.edu if interested.

Administrative

- SPIM is now installed properly on CSIL
- You can also install SPIM on your computer (this separate from QtSpim)
  - You can use your computer to check SPIM runs
  - The tester programs MAY NOT run on your computer
  - You have to run the tester programs on CSIL
- Your midterm exam is next week on **Thurs. 2/15**
  - More on this in the next class…

Lecture Outline

- More Assembly Programming
- MIPS Memory Use
- MIPS Addressing Conventions
- MIPS Instruction Representations

More Branches in MIPS

Last time, we looked at **else_if.asm**

Now, let's take a look at another problem:

nested_if.asm

nested_if.asm

```
if (x < 6)
{
   if (x > 1)
      print("x is in (1,6)\n”);
   else
      print("x <= 1\n”);
}
else
   print("x >= 6\n”);
```

Let's Plan It Out

```
if (x < 6)
{
    if (x > 1)
        print("x is in (1,6)\n”);
    else
        print("x <= 1\n”);
}
else
    print("x >= 6\n”);
```

```
.data: # setup the strings to be used
.text
main:
    # setup var x in a reg (e.g. $t0)
    # if x < 6 (set-less-than) then branch to less_than_six
    # fall through to the main else
    # print “x >= 6\n”
    # jump to exit

less-than-six:
    # if x > 1 then branch to greater_than_one
    # fall through to the inner else
    # print “x <= 1\n”
    # jump to exit

greater_than_one:
    # print “x is in (1,6)\n”
    # NO need to jump to exit!

exit:
    # end program
```

```
# C code:
# if (x < 6) {
#   if (x > 1)
#     print("x is in (1, 6)\n")
#   else
#     print("x <= 1\n")    }
# else
#   print("x >= 6\n")

.data
x_in_1_6: .asciiz "x is in (1, 6)\n"
x_le_1:   .asciiz "x <= 1\n"
x_ge_6:   .asciiz "x >= 6\n"

.text
main:
     # t0: x
     # initialize our value of x
     li $t0, 7

     # check < 6
     li $t1, 6
     slt $t2, $t0, $t1
     bne $t2, $zero, less_than_6

     # fall through to else of < 6
     li $v0, 4
     la $a0, x_ge_6
     syscall
     j main_exit
```

```
.data: # setup the strings to be used
.text
main:   # setup var x in a reg (e.g. $t0)
        # if x < 6 (set-less-than) then branch to less_than_six
        # fall through to the main else
        # print "x >= 6\n" and jump to exit
less-than-six:
        # if x > 1 then branch to greater_than_one
        # fall through to the inner else
        # print "x <= 1\n" and jump to exit
greater_than_one:
        # print "x is in (1,6)\n"
        # NO need to jump to exit!
exit:   # end program
```

```
less_than_6:
     # check x > 1 (or equivalently, 1 < x)
     li $t1, 1
     slt $t2, $t1, $t0
     bne $t2, $zero, greater_than_1

     # fall through to else of x > 1
     li $v0, 4
     la $a0, x_le_1
     syscall
     j main_exit

greater_than_1:
     # true branch of x > 1
     li $v0, 4
     la $a0, x_in_1_6
     syscall
     # could jump to main_exit,
     # but this is what we will
     # fall through to anyways!

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

More Nested if-else Examples

- See the file **nested_else_if.asm** in the demo directory for another example

```
char str
if (x < 11)
  if (x > 5)
     str = "(5, 11)\n";
else if (x == 0)
  str = "0\n";
else
  str = "< 11\n";
else
  str = "x >= 11\n";

print(str);
```

Addressing Memory 1

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

Example: `lw $t0, 0x0000400A` (← not a real address)

Example: `lw $t0, 0x0000400A($s0)` (← not a real address)

- A **word** is a natural unit of data used in processor designs
  - **Size** of a word a varies; depends on the processor ***architecture***
- Recall: 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…

Addressing Memory 2

- 1 word = 32 bits (in MIPS)
  - Which is 4 bytes, represented with 8 hex

- MIPS addresses sequential memory addresses, but not in **words**
  - Addresses are rather addressed in **Bytes** instead
  - So, new words (32-bits) are assigned every 4 Bytes

- MIPS' **alignment restriction**:
  MIPS words ***must*** start at addresses that are multiples of 4
- So, which of these are ***illegal*** MIPS addresses?

0x000010A2

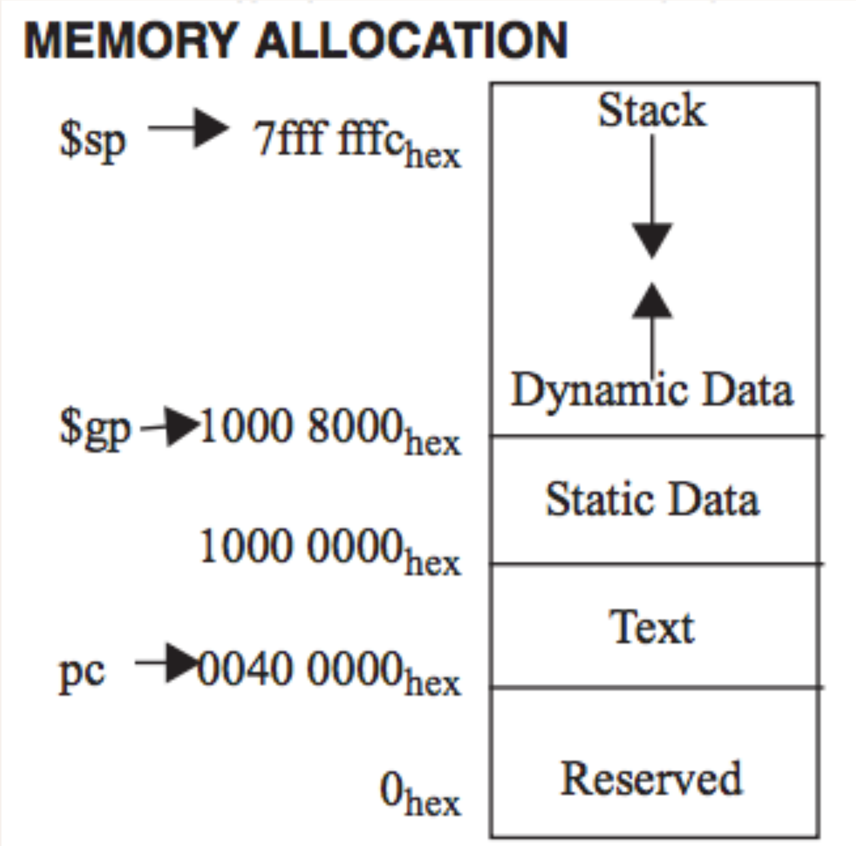
0x00012234

0x004ACDEE

0x0001500C

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.

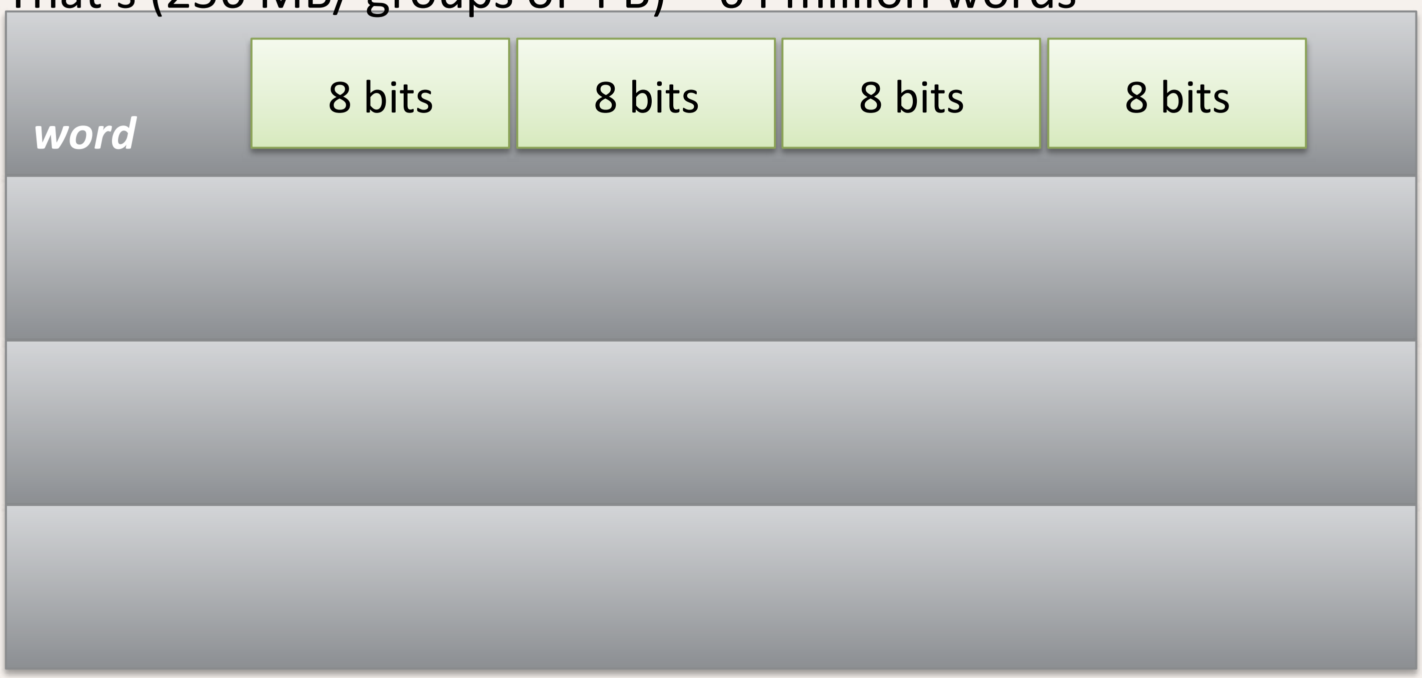
Mapping MIPS Memory

(say that 10 times fast!)

- Imagine computer memory as a big array of words
- Size of computer memory is (remember, it's 32-bit memory):

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

  – We only get to use ½ of it, or 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	← *byte addresses*
52	00	37	EE	
0x0004	0x0005	0x0006	0x0007	
B1	11	1A	A5	
0x0008	0x0009	0x000A	0x000B	

Type A →

MIPS Computer Memory Addressing Conventions

or…

Type B ←

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

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
2	00	37	EE
007	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".

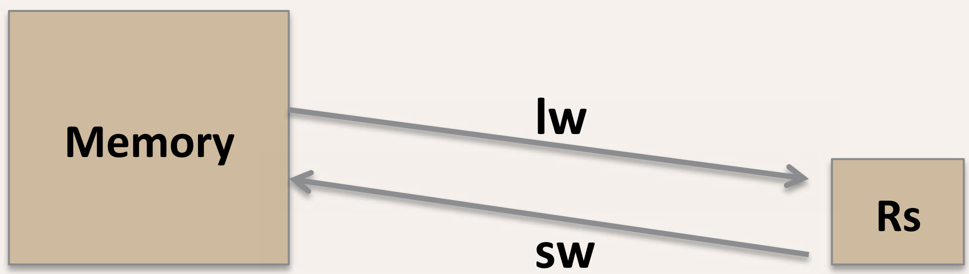
Some people preferred to eat their hard boiled eggs from the "little end" first (thus, little endians), while others prefer to eat from the "big end" (i.e. big endians).

- 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!

Accessing Memory

RECALL THAT:

- 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

Global Variables

Recall:

- Typically, global variables are placed directly in memory, not registers
- Let's take a look at: ***access_global.asm***

access_global.asm

Load Address (la) and Load Word (lw)

```
.data
myVariable: .word 42

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)

```
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");
}
```

```
# 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.

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 TO-DOs

- Assignment/Lab #4
  - Will post online on WEDNESDAY
  - Your lab is on THURSDAY
  - Assignment will be due on FRIDAY

</LECTURE>