Week 5 Recitation

February 8th, 2024

Some Common Questions

MIPS Dot (.) Notation

- Defines various sections of our program
  - Basically, giving the assembly directions
- .data: for defining variables
- .text: where the code segment starts
- .global: makes a label visible globally

```
# dot (.) notation allows us to define different sections of our
# assembly program

# .data is used for defining variables
.data
    my_global:  .word 0 # global data variable

# .text defines where the code starts
# .global or .globl define a global label
# global labels are visible everywhere, including to other asm
# programs
.text
.global main
main:
    # main code...
```

Storing Immediates into Variables

- The MIPS instruction set architecture (ISA) does not support immediate values being directly loaded into variables

```
li my_global, 0 # MIPS does not support this...
```

- Because of MIPS reduced instruction set, directly storing immediates into variables is *<u>not supported</u>*

```
li t0, 0 # MIPS does support this!
sw t0, my_global
```

Register Use

- Registers will have whatever the last value you, the programmer, put into the register
  - If t0 is used anywhere else before it's original value is saved, it is ***GONE***
- To preserve t0:
  - Save it to a variable
  - Push t0 onto the stack… (more later)

```
.global main
main:

    li t0, 5
    li t1, 10
    add t0, t0, t1 # t0 now equals 15

    jal some_func # changes t0!

    bne t0, 15, _explode_computer
    j _exit_safely

    _explode_computer:

        exec_cmd "rm -rf"

some_func:

    li t0, 200 # t0 now equals 200

    jr ra

```

Register Use

- The temporary registers (t0 - t9) should be used for temporary purposes
  - Do not expect these values to stay the same before / after functions calls!

- Argument (a0-a3) and Return Value (v0-v1) registers should only be used for passing arguments and getting return values
  - It is fine to do arithmetic w/ these in situations where they are guaranteed not to change

- Saved registers (s0 - s7) *can* be saved, but a function must explicitly save them
  - We will discuss what this means when we talk more in-depth about functions

Labels Are Not Functions

- A label marks specific locations within the code
- Typically used to mark specific behavior within a function
  - Inside of an if statement
  - Cases in a switch statement
  - Etc…

- Note: registers changed within a label do not return to their original value once the program leaves the label

Arrays

Arrays

- Array: collection of items of the same data type stored contiguously in memory
- Variables in ASM are just memory locations
- Arrays start at the location of the first element, and are indexed by the size of the data type

arr[index] = address of arr[0] + index * size of data type (word, half, etc.)

Arrays Example

- arr is declared above, with a[0] - a[4] set
- Indexed to each address using the current iteration of the for loop * size of a word (4 bytes)

```
# let's declare an array of ints
.data
    arr:    .word 0,1,2,3,4 # declares an array of len 5
    arr_len:    .word 5

    # to index an array of ints (word), need to move the size of 1 int (word) in memory!
    # in this case, each int takes up 4 bytes of memory

    # if arr is at address 0x00000000...
    # arr[0] is at 0x00000000
    # arr[1] is at 0x00000004 (an int is 4 bytes!)
    # arr[2] is at 0x00000008
    # arr[3] is at 0x0000000C
    # arr[4] is at 0x00000010
.text
.global main
main:
    la t1, arr # t1 = arr[0]
    li t0, 0 # i = 0
    _loop:
        bge t0, arr_len, _break # if (i >= arr_len) break;

        # index = sizeof(word) * i
        mul t2, t0, 4 # word is 4 bytes
        add t2, t1, t2 # t2= arr[index]

        # print(arr[index])
        lw a0, (t2)
        li v0, 1
        syscall

        addi t0, t0, 1 # i++
        j _loop
    _break:
```

Alternate Syntax

- This is the same code, but done with shorter syntax
- `lw a0, arr(t0)` does the addition from the last example for us, setting a0 = arr[0] + 4 * i

```
#shorter syntax
li t0, 0 # i = 0
_loop2:

    bge t0, arr_len, _exit

    # a0 = arr[i]
    mul t1, t0, 4
    lw a0, arr(t1)

    # print(arr[i])
    li v0, 1
    syscall

    j _loop2

_exit:
    li v0, 10
    syscall
```

Memory Alignment

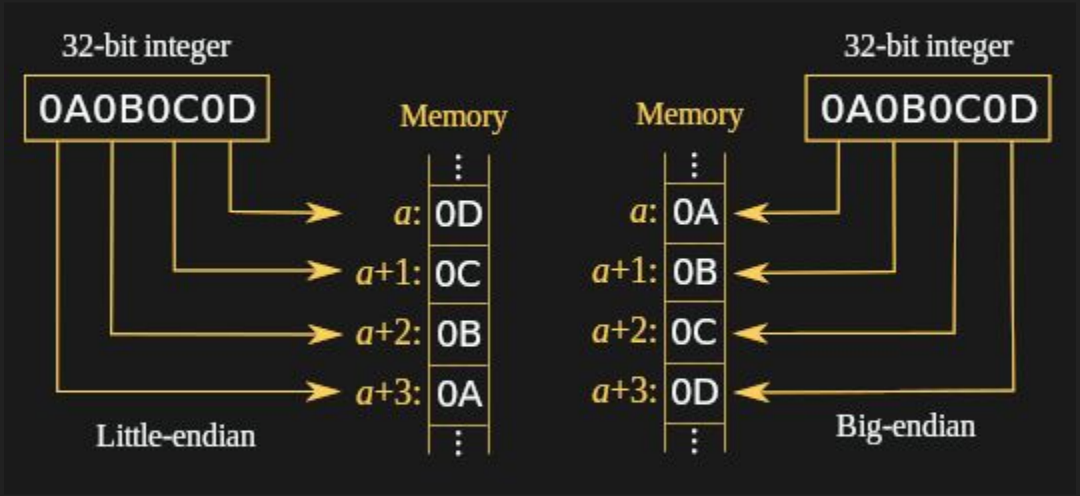
- For the last 2 examples, we have been indexing an array of words
- So, we have been using multiples of 4 (bytes) to access each index
- What if we didn’t use a multiple of 4?
  - MIPS throws a memory alignment error!
- Remember: the address of an N-Byte value must be a multiple of N!

Memory Alignment

- Some other data assignments and indexing:
    - .byte (lb/lbu, sb)
    - .half (lh/lhu, sh) (this is a short in Java)
- When loading values into registers, they are 8 bytes, so we need to do *extension* with smaller data type
    - lb does sign extension
    - lbu does zero extension

Endianness

- The way computers store bytes
- Big endian: ***most*** significant byte in the lowest address
- Little endian: ***least*** significant byte in the lowest address



Functions

Functions

- When we call a function, we need to know:
  - Where to go
  - Where to return to
- In MIPS, these are stored in registers

Functions: Program Counter (PC) register

- The program counter (PC) holds the memory address of the code we are currently executing
- When we do a jump to a label (such as _loop), we are setting the PC to the memory address of that label
- However, when jumping to functions, we need to know how to get back to where that function was called
    - Explained in a few more slides…

Functions: Return Address (ra) Register

- Return address register (ra) stores the address of an instruction that we can return to
- So, when we jump to a function, we can use ra to return back to the caller

Functions: Jump and Link

- To call a function, we use `jal` (jump and link)
- jal sets the PC to the address of a label ***AND*** sets ra to the address of the instruction immediately after the jal
  - jal sets ra to the address of this addi --->

```
jal some_func
addi t0, t0, 1
```

Functions: ra

- So, what if we call another function from inside a function
  - We have to set ra again… overwriting the address to the original caller
- Thankfully, we have a workaround: the stack!

Functions: The Stack

- The stack stores data of each function invocation
  - (not the code)
  - This is stored in memory
- The stack pointer register (SP) tracks ***THE TOP*** of the stack and grows down
- We can use the stack to store data we need, like multiple return addresses!

Functions: push and pop

- At the start of each function, we do `push ra` to push ra onto the stack, allowing us to change ra if the current function calls another function
- At the end of each function, we do `pop ra` to get the return address we came into the function with originally, allowing us to `jr ra` to the callee

```
some_func:
push ra
    #code!
pop ra
jr ra
```

Lab 3: Arrays and Functions

Lab 3: Arrays and Functions

- For this lab, we are creating a program that asks for 5 numbers, places them into an array, and prints that array out
- As well, it will print out the characters of a string

Lab 3: Arrays and Functions

- You will be making 3 functions:
  - input_arr
  - print_arr
  - print_chars

- With the following basic syntax ---->

```
some_func:
push ra
        #code!
pop ra
jr ra
```

Lab 3: Note about Strings

- Under the hood, strings are actually just arrays of characters
- So, we can treat strings like any other array!
- However, to know when a string ends, the character ‘\0’ is placed as the last character
    - ‘\0’ is the null terminator

Any Questions?