Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Computer Architecture and Assembly Lab
Fall 2018

Lab 3
C Memory Management and Introduction to RISC-V

Instructions

Go over the lab manual and then answer all the questions in the Exercises section below. Push everything in the lab3 folder of your Github repository and zip the folder in order to upload to Sakai.

C Memory Management

We will discuss the following questions:

- How does the C compiler determine where to put code and data in the machine's memory?
- How can we create dynamically sized objects?
  - e.g., an array of variable size depending on requirements.

Overview

Code

- Loaded when program starts
- Does not change

Static Data

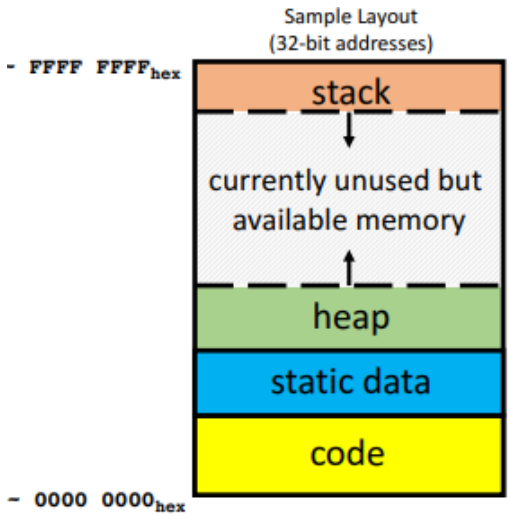
- Loaded when program starts
- Can be modified
- Size is fixed

Stack

- Local variables and arguments inside functions
- Allocated when function is called
- Stack usually grows downward

Heap

- Space for dynamic data
- Allocated and freed by program as needed



Stack

- Every time a function is called, a new frame is allocated
- When the function returns, the frame is deallocated
- Stack frame contains
  - Function arguments

- Local variables
- Return address (who called me?)

- Stack uses contiguous blocks of memory
  - Stack pointer indicates current level of stack
- Stack management is transparent to C programmer
  - More details when we program in Assembly

Managing the Heap

C functions for heap management
• malloc() allocate a block of uninitialized memory
• calloc() allocate a block of zeroed memory
• free() free previously allocated block of memory
• realloc() change size of previously allocated block
    * previously allocated contents might move
Exact definitions can be found by typing in a terminal
    man malloc

Observations

• Code, Static storage are easy:
    − they never grow or shrink
    − taken care of by OS
• Stack space is relatively easy:
    − stack frames are created and destroyed in last-in, first-out (LIFO) order
    − transparent to programmer
    − but don't hold onto it (with pointer) after function returns!
• Managing the heap is the programmer's task:
    − memory can be allocated/deallocated at any time
    − how to tell/ensure that a block of memory is no longer used anywhere in the program?
    − requires planning before coding …

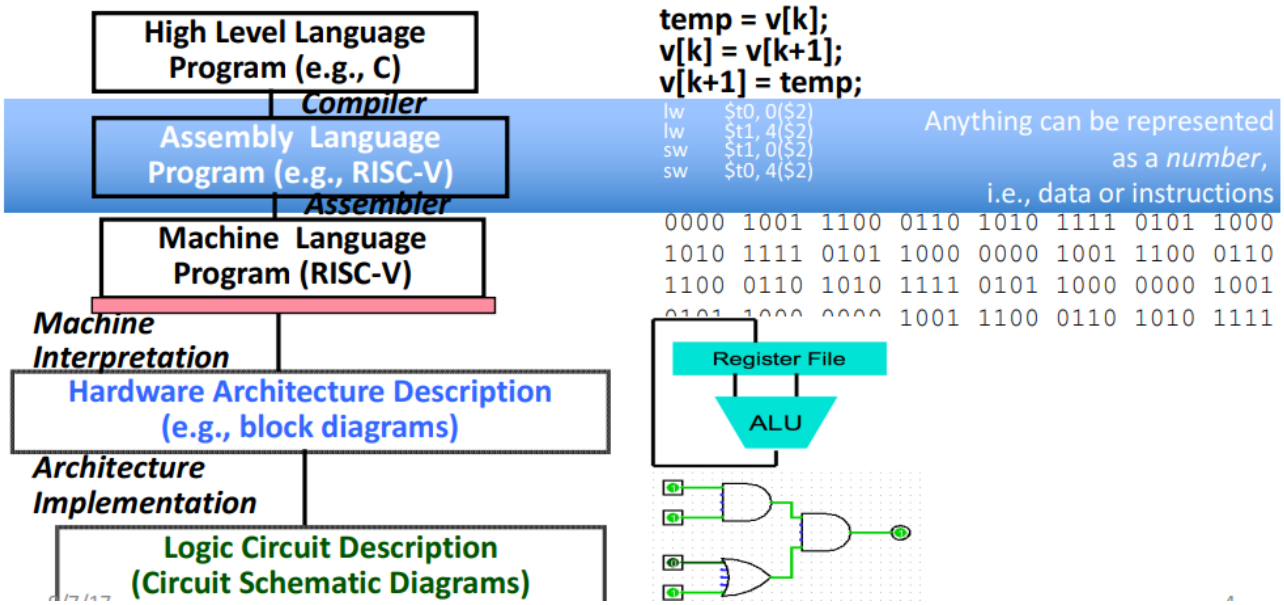
Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - De-allocated stack or heap variable
  - Out-of-bounds reference to array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer returned by malloc/calloc
- Memory leaks

- you allocated something but forgot to free it later

Intro to RISC-V and Venus simulator

Levels of representation/interpretation



The lab section uses the Venus RISC-V simulator, which can execute in a browser with JavaScript enabled. We will explore basic machine instructions on the RISC-V instruction set.

After the lab you will have a good overview of RISC-V instructions (RV32I). You will be able to simulate small to medium size programs on a RISC-V instruction set simulator.

A Minimal Assembler Program

(And How to Start Everything Off with Loading Constants)

Start by pasting the following minimal.s assembler program into Venus, in the Editor pane:

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

```
# As minimal RISC-V assembler example
addi x1, x0, 2
addi x2, x0, 3
add x3, x1, x2
```

Switch to the Simulator and step through the code with the `Step` button. Watch how the register `x1`, `x2`, and `x3` change. Adding immediate values to register `x0`, which is always 0, is one way to load constants (immediate values) into registers. Loading immediate values is so basic to get a program started that RISC-V defines a pseudo instruction, `li`, as a shortcut. Enter the following code into the editor and switch to the simulation pane.

```
# Use of pseudo instructions to load immediate values
li x1, 2
li x2, 3
add x3, x1, x2
```

You will notice that your code is listed under Original Code, but the real RISC-V instructions are listed under Basic Code. Those instructions are the very same as you entered with minimal.s.

RISC-V immediate values for ALU operations are 12-bit wide. Try to use larger constants in your program (you can also use the 0xabcd notion for hexadecimal values). What happens when the constant does not fit into sign extended 12 bits?

Lookup the `lui` instruction in the The RISC-V Instruction Set Manual.
Why is immediate loading so fundamental?

Step, Breakpoint, and Run

Extend you program with a handful of more instructions to explore the functions of the Venus simulator. You can clear the registers and the program counter by pressing `Reset`. Step through your program with `Step` or run you program to completion with `Run`.
Another important concept is a breakpoint. You can set a breakpoint by clicking into the instruction. A breakpoint is marked by coloring it red. Another click on the instruction will clear the breakpoint.

With a breakpoint you can run the program until it reaches the breakpoint. There you might explore some register values.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Computing with ALU Instructions

A computer can compute. "Of course!", you will say. However, how do you compute on a RISC processor?

You compute with just a handful of instructions in the arithmetic logic unit (ALU). Operations for ALU instructions are provided in registers and the result is put into a register as well.

Locate all integer ALU instructions of RISC-V and explore them in the simulator.

Interlude: Talking to the World

Venus contains a simulation of low level operating system functions. The functions in Venus have been inspired by the MIPS simulator MARS, which itself has been inspired by SPIM.

System functions in RISC-V are invoked with the `ecall` instruction, where `ecall` stands for environment call. However, the concrete semantics of those functions is operating system dependent. Arguments to the system function are passed via the normal argument register `a0` and `a1`, where `a0` contains the function code. Explore io.s to print an integer value. You can use this simple print function for `printf` debugging.

Assembler Directives

Beside instructions in assembler format, an assembler also accepts so-called assembler directives. The code start is usually marked with `.text`. You can initialize data in the data segment with `.data`. Each assembler instruction can start with a label such as `main`: or `loop:`. This label can then be used as destination for a branch instruction. Also, data can be addressed by using a label. See below some examples:

```
.text
main:
la a1, hello
loop: li, x3, 123
... more code
.data
```

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

```
hello:
.asciiz "Hello"
```

Add a `main:` label to the start of your program and add following instruction at the end of your program:

```
j main
```

What happens when you step through your code? What happens when you press `Run`?

A RISC Machine is also Called a Load/Store Architecture

Operands for ALU instructions are always taken from registers and the result is also put into registers in a RISC machine. How can we then take operands from the memory or write results into memory?

We need to use load and store instructions. That's why a RISC machine is also called a load/store architecture.

Use a store instruction to store `0xdeadbeef` into the memory. The simulator can display memory. Click on the Memory field, scroll down and Jump to Data. The simulator choses to start the data segment at `0x1000000`. Now write into that location the `0xdeadbeef`.
How do you get that address into a register at first place? Remember immediate values?

After storing that value into the main memory also read it back into a register.

We can Say Hello World to the World

As a final exercise we will say "Hello World".

A list of implemented `ecall` functions can be found in the Environmental Calls section in the Venus Documentation.
You can print strings that are allocated in the static data segment. Explore hello.s.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Exercises

1. Match the items on the left with the memory segment in which they are stored. Answers may be used more than once, and more than one answer may be required.

1. Static variables 2. Local variables 3. Global variables 4. Constants 5. Machine Instructions 6. malloc() 7. String Literals	A. Code B. Static C. Heap D. Stack

2. Write the code necessary to properly allocate memory (on the heap) in the following scenarios

a). An array arr of k integers

b) A string str of length p

c) An n × m matrix mat of integers initialized to zeros

3. Assume we have an array in memory that contains int* arr = {1,2,3,4,5,6,0}. Let the value of arr be a multiple of 4 and stored in register s0. What do the snippets of RISC-V code do?

a) lw t0, 12(s0)

b) slli t1, t0, 2
add t2, s0, t1
lw t3, 0(t2)
addi t3, t3, 1
sw t3, 0(t2)

c) lw t0, 0(s0)
xori t0, t0, 0xFFF
addi t0, t0, 1

4. What are the instructions to branch to label on each of the following conditions? The only branch instructions you may use are beq and bne.

s0 < s1	s0 <= s1	s0 > 1

5. Open the files lab3_ex5_c.c and lab3_ex5_assembly.s. The assembly code provided (.s file) is a translation of the given C program into RISC-V. Your task is to find/explain the following components of this assembly file.

a. The register representing the variable k.
b. The registers acting as pointers to the source and dest arrays.
c. The assembly code for the loop found in the C code.
d. How the pointers are manipulated in the assembly code.

6. Translating between C and RISC-V
Translate between the C and RISC-V code. You may want to use the RISC-V Reference Card for more information on the instruction set and syntax. In all of the C examples, we show you how the different variables map to registers – you don't have to worry about the stack or any memory-related issues. You may assume all registers are initialized to zero

C	RISC-V

// s0 -> a, s1 -> b
// s2 -> c, s3 -> z
int a = 4, b = 5, c = 6, z;
z = a + b + c + 10;	

// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;	

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
a = 0;
} else {
b = a - 1;
}	

	addi s0, x0, 0
	addi s1, x0, 1
	addi t0, x0, 30
	loop:
	beq s0, t0, exit
	add s1, s1, s1
	addi s0, s0, 1
	jal x0, loop
	exit:

// s0 -> n, s1 -> sum
// assume n > 0 to start
int sum;
for(sum=0;n>0;sum+=n--);	

7. Implement a function factorial in RISC-V that has a single integer parameter n and returns n!. A stub of this function can be found in the file factorial.s. You will only need to add instructions under the factorial label, and the argument that is passed into the function is configured to be located at the label n. You may solve this problem using either recursion or iteration.