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

Computer Architecture and Assembly Lab
Spring 2021

Lab 3
C heap functions and Introduction to RISC-V

Goals

1. Allocate heap memory using malloc() free()
2. Write RISC-V assembly code

The stack, the heap and static

- stack: local variable storage (automatic, continuous memory)
- heap: dynamic storage (large pool of memory, not allocated in contiguous order)
- static: global variable storage, permanent for the entire run of the program (.data segment, part of the executable file)

```
[uty@u ~]$ ps -A | grep lab3
 157926 pts/3    00:00:00 lab3
[uty@u ~]$ cat /proc/157926/maps
558836ad1000-558836ad2000 r--p 00000000 08:07 5736320                    /home/uty/prjs/testlab/lab3
558836ad2000-558836ad3000 r-xp 00001000 08:07 5736320                    /home/uty/prjs/testlab/lab3
558836ad3000-558836ad4000 r--p 00002000 08:07 5736320                    /home/uty/prjs/testlab/lab3
558836ad4000-558836ad5000 r--p 00002000 08:07 5736320                    /home/uty/prjs/testlab/lab3
558836ad5000-558836ad6000 rw-p 00003000 08:07 5736320         .data      /home/uty/prjs/testlab/lab3
5588370db000-5588370fc000 rw-p 00000000 00:00 0                          [heap]
7f03f94a7000-7f03f94a9000 rw-p 00000000 00:00 0
7f03f94a9000-7f03f94cf000 r--p 00000000 08:07 4721411                    /usr/lib/libc-2.32.so
7f03f94cf000-7f03f961c000 r-xp 00026000 08:07 4721411                    /usr/lib/libc-2.32.so
7f03f961c000-7f03f9668000 r--p 00173000 08:07 4721411                    /usr/lib/libc-2.32.so
7f03f9668000-7f03f966b000 r--p 001be000 08:07 4721411                    /usr/lib/libc-2.32.so
7f03f966b000-7f03f966e000 rw-p 001c1000 08:07 4721411                    /usr/lib/libc-2.32.so
7f03f966e000-7f03f9674000 rw-p 00000000 00:00 0
7f03f96a8000-7f03f96aa000 r--p 00000000 08:07 4721393                    /usr/lib/ld-2.32.so
7f03f96aa000-7f03f96cb000 r-xp 00002000 08:07 4721393                    /usr/lib/ld-2.32.so
7f03f96cb000-7f03f96d4000 r--p 00023000 08:07 4721393                    /usr/lib/ld-2.32.so
7f03f96d4000-7f03f96d5000 r--p 0002b000 08:07 4721393                    /usr/lib/ld-2.32.so
7f03f96d5000-7f03f96d7000 rw-p 0002c000 08:07 4721393                    /usr/lib/ld-2.32.so
7ffd36f52000-7ffd36f73000 rw-p 00000000 00:00 0                          [stack]
7ffd36ff6000-7ffd36ff9000 r--p 00000000 00:00 0                          [vvar]
7ffd36ff9000-7ffd36ffa000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Heap vs. stack, key difference

- Stack is a linear data structure whereas Heap is a hierarchical data structure.
- Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed.
- Stack accesses local variables only while Heap allows you to access variables globally.
- Stack variables can't be resized whereas Heap variables can be resized.
- Stack memory is allocated in a contiguous block whereas Heap memory is allocated in any random order.
- Stack doesn't require to de-allocate variables whereas in Heap de-allocation is needed.
- Stack allocation and deallocation are done by compiler instructions whereas Heap allocation and deallocation is done by the programmer.

Libc heap functions

- **void* malloc (size_t size);**
  - Allocate memory block

- **void free (void* ptr);**
  - Deallocate memory block

- **void* calloc (size_t num, size_t size);**
  - Allocate and zero-initialize array

- **void* realloc (void* ptr, size_t size);**
  - Reallocate memory block (realloc() = free() then malloc())

Exercise 1 [10 pts]: Memory Allocation in C

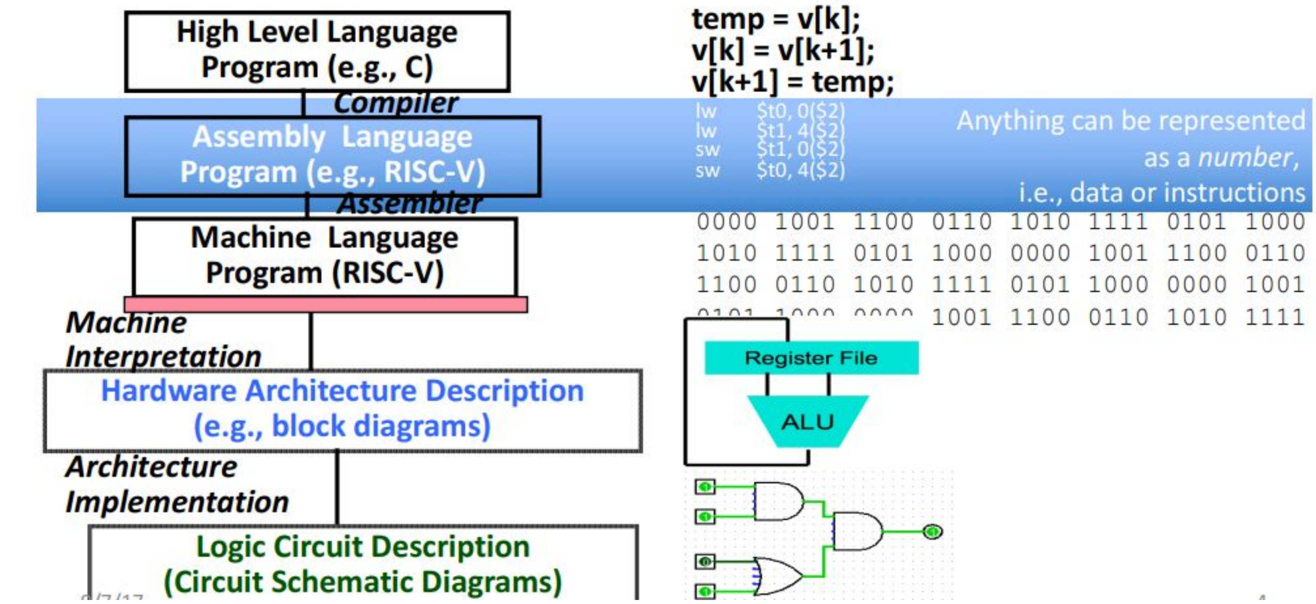
E1.1. [2 pts] Where in memory would a static integer (global) variable be stored? Explain why.

E1.2 [8 pts] Write a C program which dynamically allocates memory to define a float array with 10 entries by using the malloc() function and prints each element out in a for loop. Do not forget to free the memory in the end of your program.

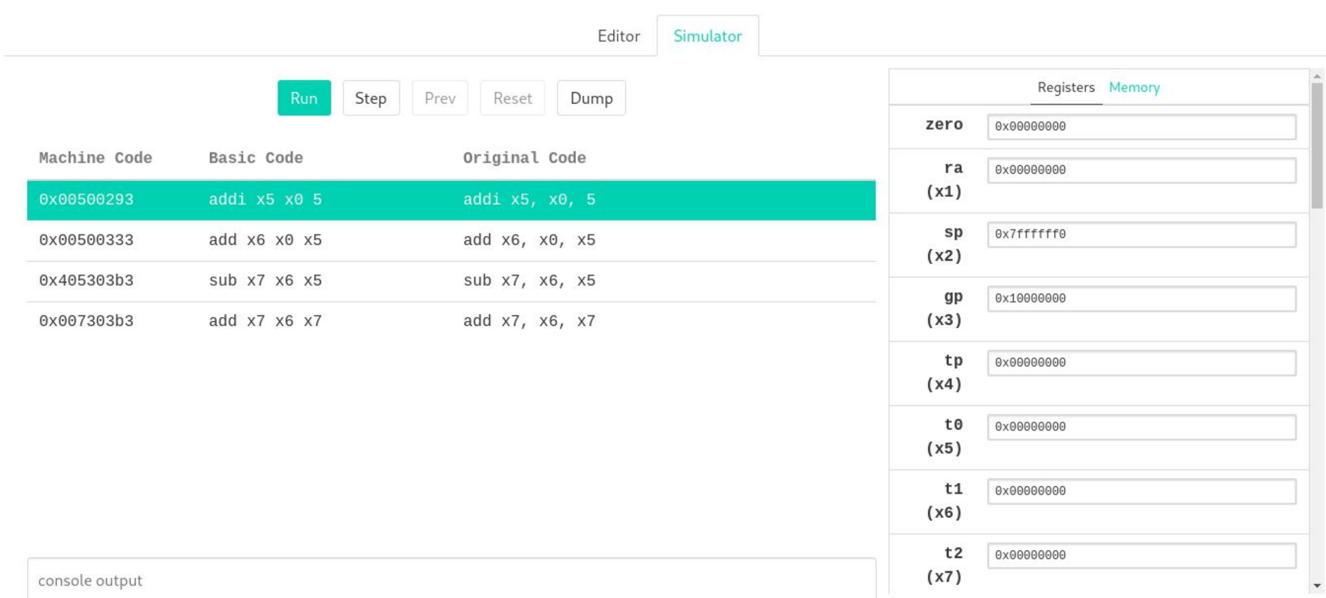
Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Introduction to RISC-V and the Venus Simulator

Levels of representation/interpretation



We use the **Venus simulator** [(https://www.kvakil.me/venus/)](https://www.kvakil.me/venus/) to execute RISC-V assembly programs in a browser with JavaScript enabled.



Exercise 2 [30 pts]: A Minimal Assembler Program Tutorial
(How to Start Everything Off with Loading Constants)

Start by pasting the following minimal.s assembly program into Venus:

```
# As minimal RISC-V assembly language example (minimal.s)
addi x5, x0, 5
add x6, x0, x5
sub x7, x6, x5
add x7, x6, x7
```

E2.1. [8 pts] Switch to the Simulator and run your program with the Run step, or step through the code with the Step button. You can also clear the registers and the program counter by pressing Reset. Please indicate how the registers x5, x6, and x7 change after running each line.

E2.2. [2 pts] Try to use larger constants in your program. What is the largest immediate constant you can use with the ALU operations?

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

After you finish the E2.1 and E2.2 parts, please extend your program with a handful of more instructions to explore the functionality of the Venus simulator. You can refer to the instructions in the **RISC-V Instruction Set Manual** (https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf).

Locate all integer ALU instructions of RISC-V (below) and explore them using the simulator.

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

E2.3. [10 pts] Assume you need to load the 0xdeadbeef immediate constant from the memory address 0x10000018 as shown in the following screenshot. Please write a RISC-V program and explain how you achieve it.

Registers Memory

Address	+0	+1	+2	+3
0x10000018	de	ad	be	ef

E2.4. [10 pts] Assume we have an array long int* arr = {3,7,6,4,5}. The values in arr store in the memory of 0(x21), 8(x21),…, 32(x21). Convert the following RISC-V code into a C program.
(Name the variable yourself if needed. Notice, on 64-bit system, sizeof(int) is 4, sizeof(long int) is 8)

```
ld x5, 16(x21)
addi x6, x0, 3
sll x5, x5, x6
sd x5, 8(x21)
```

Exercise 3 [35 pts]: Environment Call

The ECALL instruction is used to make a service request to the execution environment, which might require privileged access. The EEI (execution environment interface) will define how parameters for the service request are passed, but usually these will be in the registers. The ECALL is equivalent to int 0x2e, int 0x80, sysenter and syscall in x86 systems.

Venus contains a simulation of low level operating system functions. The functions in Venus have been inspired by the MIPS simulator MARS. Arguments to the system function are passed via the normal argument register x10 and x11, where x10 contains the function code. Explore io.s to print different kinds of information to the output console. You can use this feature to help you debug your code. For more information about Venus' ECALL, please refer to the wiki page: https://github.com/kvakil/venus/wiki/Environmental-Calls

The following environmental calls are currently supported.

ID (`a0`)	Name	Description
1	print_int	prints integer in `a1`
4	print_string	prints the null-terminated string whose address is in `a1`
9	sbrk	allocates `a1` bytes on the heap, returns pointer to start in `a0`
10	exit	ends the program
11	print_character	prints ASCII character in `a1`
17	exit2	ends the program with return code in `a1`

```
# Ecall example (io.s)
addi x10, x0, 1
addi x11, x0, 37
ecall
```

E3.1. [8 pts] What do you see from the console in the simulator when you run the io.s program? Please explain the purpose of each line in the program.

E3.2 [12 pts] Please convert the following C program into the RISC-V assembly code equivalent.

```
#include <stdio.h>

int main()
{
        int x = 30, y = 17;

        printf("x + y = ");
        printf("%d\n", x + y);

        return 0;
}
```

E3.3 [15 pts] Please convert the following C program into the RISC-V assembly code.

```
#include <stdio.h>

int main()
{
	int a = 23, b = 26, c = 5;

	if (a < b)
	{
		printf("%d\n", c*2);
	}
	else
	{
		a = a * 4;
	}

	b = b * 8;

	printf("%d\n", c);

	return 0;
}
```

Exercise 4 [25 pts]: 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:

For more information, check: https://github.com/kvakil/venus/wiki/Assembler-Directives

```
.text
main:
addi x10, x0, 4
la x11, hello
ecall
.data
hello:
.asciiz "Hello"
```

E4.1. [5 pts] What happens when you add the following instruction at the end of your program and run the program? Please explain why.

```
j main
```

E4.2 [20 pts] Please convert the following C problem into the RISC-V assembly code.

```
#include <stdio.h>

int exercise4 (int a)
{
	int res = 0;
	for (int i = 0; i < a; i++)
	{
		res += i;
	}
	return res;
}

int main (void)
{
	int a = 10;
	int b;

	b = exercise4(a);
	printf("%d\n", b);
}
```