RISC V Simulator

Installing

RARS is a Java program and should work on any machine that supports a Java Virtual Machine. If you need java, you can get it here. Download RARS from here or from the main RARS webpage: https://github.com/TheThirdOne/rars/releases. We will use this program throughout the course, so put it somewhere you can keep it long term.

On Windows and MacOS you should be able to double click the .jar file.

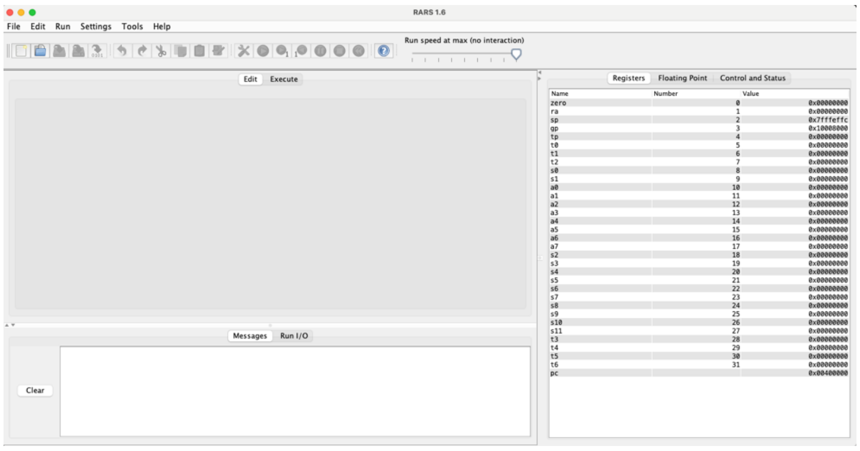
Sometimes you need to launch it from a command line. Example for running it from a linux command line:

```
> java -jar rars1_6.jar
```

Some help and info pages are available on the RARS github page.

Running Programs

Launch RARS. A window should appear named similar to "RARS 1.6" with three panes.

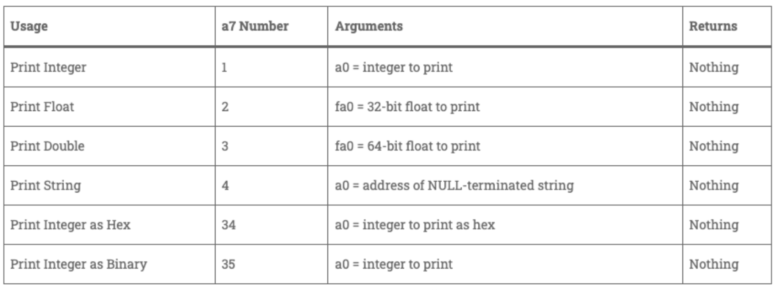


Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400000

1. The right-most pane shows the contents of the registers. At the bottom of the register list is the pc register, which tracks the address of the current instruction. The tabs at the top of the pane allow you to look at control and status registers. If an error occurs in your code, the simulator might automatically show you the values of the "control and status" registers. We will not use these for a while, so you should switch back to the general registers for now.
2. The large upper-left pane shows the contents of the currently loaded file. It should be blank when you start up RARS. There are two tabs at the top of the pane.
    1. The Edit tab allows you to edit the currently loaded assembly file.
    2. The Execute tab shows the assembled code and processor state as the code runs.
3. The bottom panel is the output console. The 'Messages' tab displays messages from the simulator. The 'Run I/O' tab shows any messages generated from the running program.

Download or cut-paste the raw file (https://raw.githubusercontent.com/TheThirdOne/rars/master/test/loop.s) to use it with RARS. Convert the loop.s program to a C program that does the same work as loop.s . You can execute the C program in your local machine. In the loop.s file, check https://github.com/TheThirdOne/rars/wiki/Environment-Calls to understand the ecall. For this one, ecall is just a "return " call in C.

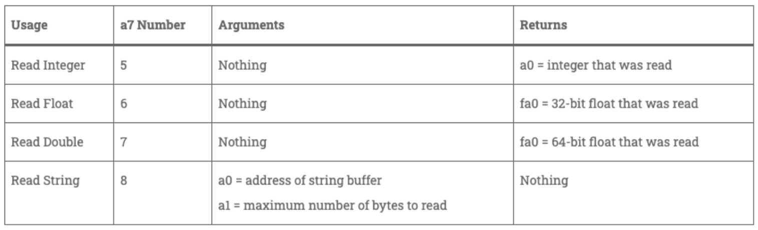
The key ones for printing are :



Usage	a7 Number	Arguments	Returns
Print Integer	1	a0 = integer to print	Nothing
Print Float	2	fa0 = 32-bit float to print	Nothing
Print Double	3	fa0 = 64-bit float to print	Nothing
Print String	4	a0 = address of NULL-terminated string	Nothing
Print Integer as Hex	34	a0 = integer to print as hex	Nothing
Print Integer as Binary	35	a0 = integer to print	Nothing

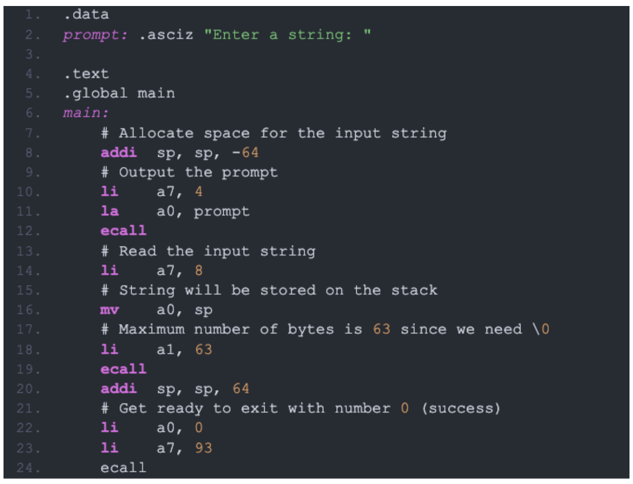
Printing System Calls

The ones for reading are :



Usage	a7 Number	Arguments	Returns
Read Integer	5	Nothing	a0 = integer that was read
Read Float	6	Nothing	fa0 = 32-bit float that was read
Read Double	7	Nothing	fa0 = 64-bit float that was read
Read String	8	a0 = address of string buffer a1 = maximum number of bytes to read	Nothing

The following example uses two of the system calls. The first system call is number 4, which prints a string to the screen. Then, the second system call waits for the user to enter a string, up to 63 bytes. The reason we use 63 bytes instead of 64 is because we need to store the NULL (0) terminator.



```
.data
prompt: .asciz "Enter a string: "

.text
.global main
main:
    # Allocate space for the input string
    addi  sp, sp, -64
    # Output the prompt
    li    a7, 4
    la    a0, prompt
    ecall
    # Read the input string
    li    a7, 8
    # String will be stored on the stack
    mv    a0, sp
    # Maximum number of bytes is 63 since we need \0
    li    a1, 63
    ecall
    addi  sp, sp, 64
    # Get ready to exit with number 0 (success)
    li    a0, 0
    li    a7, 93
    ecall
```

As I mentioned in class, https://godbolt.org/ can be of help to generate assembly code.