

#6 : MIPS Programming II

Computer Architecture 2021/2022

Ricardo Rocha

Computer Science Department, Faculty of Sciences, University of Porto

MARS – MIPS Simulator

Main functionalities:

- Edit programs (**assembly**)
- Compile (**assembler**)
- Run and/or execute step by step
- See the memory contents and the values in the set of registers

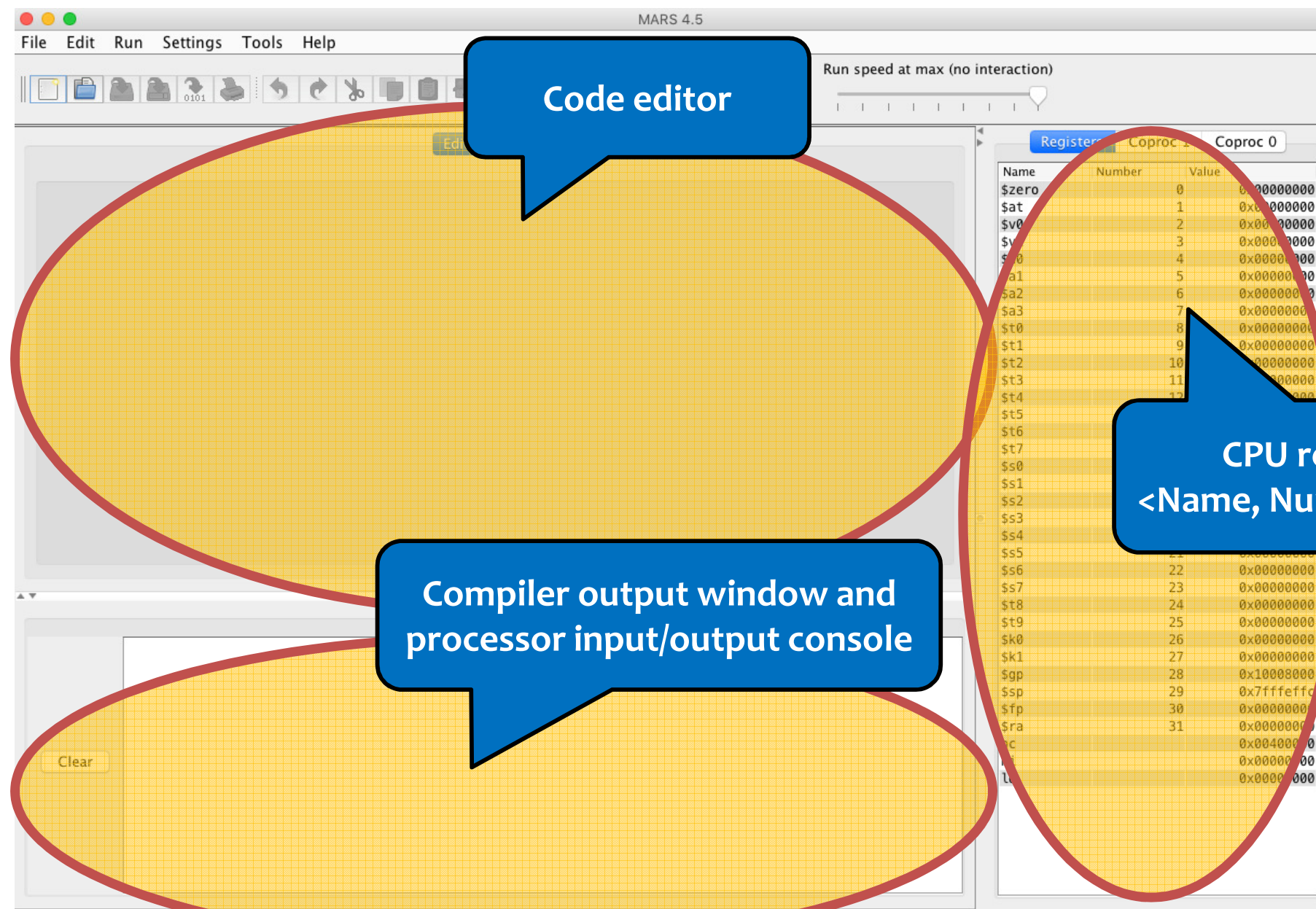
Download Mars4_5.jar:

- <http://www.softpedia.com>

Command to execute:

- `java -jar Mars4_5.jar`

MARS – MIPS Simulator



MARS – MIPS Simulator

The screenshot shows the MARS MIPS Simulator interface. The main window is divided into several sections. At the top is a menu bar with 'File', 'Edit', 'Run', 'Settings', 'Tools', and 'Help'. Below the menu bar is a toolbar with various icons. The main area is split into two panes: 'Text Segment' and 'Data Segment'. Both panes are currently empty and highlighted with red ovals. A blue callout box points to the 'Text Segment' pane with the text 'Text segment preview (binary code and source code)'. Another blue callout box points to the 'Data Segment' pane with the text 'Data segment preview (memory contents)'. To the right of the main panes is a register window titled 'Proc 1 Coproc 0'. It contains a table with columns 'Name', 'Number', and 'Value'. The table lists various MIPS registers and their current values. At the bottom of the window is a 'Mars Messages' pane with a 'Clear' button.

Text segment preview
(binary code and source code)

Data segment preview
(memory contents)

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
	21	0x00000000
	22	0x00000000
	23	0x00000000
	24	0x00000000
	25	0x00000000
	26	0x00000000
	27	0x00000000
	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

MARS – MIPS Simulator

The screenshot shows the MARS MIPS Simulator window. The title bar reads "mips1.asm* - MARS 4.5". The menu bar includes "File", "Edit", "Run", "Settings", "Tools", and "Help". The toolbar contains icons for file operations, editing, and execution. A blue callout bubble points to the "Run" menu, containing the text "Compile (Run -> Assemble)". The main text area shows assembly code on a grid background, with lines 2 through 13 visible. A yellow oval highlights the code area. A second blue callout bubble points to the "File" menu, containing the text "Create new program (File -> New), edit and save (File -> Save)". The status bar at the bottom left shows "Line: 13 Column: 9" and a "Show Line Numbers" checkbox. The right panel shows the "Registers" tab with a table of MIPS registers and their values.

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Execute

asm*

2 `li $t1, 0`
3 `lw $t2, _X`
4
5 `add $t1, $t1, $t2`
6 `sw $t1, _X`
7
8 `addiu $v0, $zero, 10`
9 `syscall`
10
11
12
13

Line: 13 Column: 9 Show Line Numbers

Clear

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

MARS – MIPS Simulator

The screenshot shows the MARS MIPS Simulator interface. At the top, the title bar indicates the file path and version: `/Users/joaosoares/Dropbox/[Documents]/[Faculdade]/[docente]/19-20/AC/mips1.asm - MARS 4.5`. The menu bar includes **File**, **Edit**, **Run**, **Settings**, **Tools**, and **Help**. Below the menu bar, a status bar shows **Run speed at max (no interaction)**. A blue callout bubble points to the **Run** menu, containing the text: **Run program (Run -> Go) ou execute step by step (Run -> Step)**.

The main window is divided into several panes. The **Text Segment** pane on the left displays assembly code with columns for **Bkpt**, **Address**, **Code**, **Basic**, and **Source**. A blue callout bubble points to this pane, containing the text: **Lines <Address, Binary code, Source code>**. Below this, the **Address** pane shows a table of memory addresses and their corresponding values. A blue callout bubble points to this table, containing the text: **1 word pairs <Address, Value>**.

The **Registers** pane on the right shows a table of registers and their values. The table has columns for **Name**, **Number**, and **Value**. The registers listed are `$zero` through `$lo`.

At the bottom, the **Assembler** pane shows the output of the assembly process, including the message: **Assemble: operation completed successfully.** A **Clear** button is located below the assembler output.

MARS – MIPS Simulator

The screenshot shows the MARS MIPS Simulator interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons. The main window displays assembly code with annotations. A blue callout points to the instruction at address 0x10010005, identifying it as the next instruction to be executed. Another blue callout points to the \$t1 register in the assembly code, identifying it as the last modified register. A third blue callout points to the memory location 0x10010005 in the Data Segment, identifying it as the last modified memory position. The right side of the window shows the Registers window, which lists registers and their values. The bottom of the window shows the Mars Messages and Run I/O sections.

Program counter (next instruction to be executed)
<Address, Binary code, Source code>

Last modified register
<Name, Number, Value>

Last modified memory position
<Address, Value>

Name	Number	Value
\$zero	0	0x00000000
\$t1	1	0x10010005
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000005	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Name	Number	Value
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10003000
\$sp	29	0x7ffffcfc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400018
hi		0x00000000
lo		0x00000000

Procedure Calls

The execution of a procedure call happens when one procedure (the **caller**) invokes another procedure (the **callee**). In general, the execution of a procedure call follows six steps:

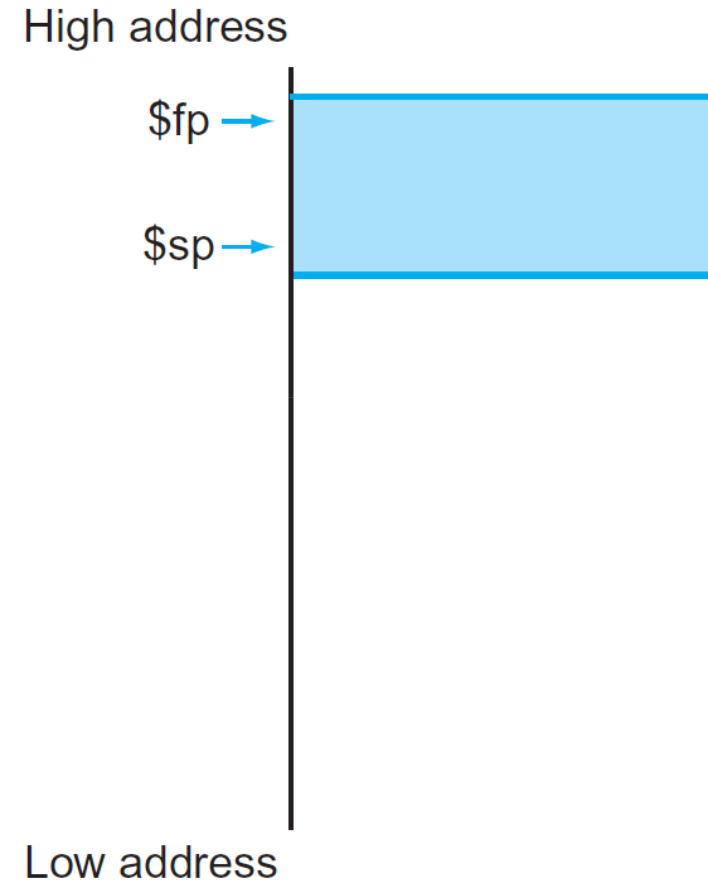
- Put arguments in a place where the callee can access them
- Transfer control to the callee
- Acquire storage resources needed for callee execution
- Perform callee's operations
- Put results in a place where the caller can access them
- Return control to the caller's next instruction

Procedure Calls

The bookkeeping associated with procedure calls is done in the **stack segment** around blocks of memory called **procedure frames**.

By historical precedent, the **stack grows from higher addresses to lower addresses**. This convention means that you push values onto the stack by subtracting from the stack pointer.

- Register **\$fp (frame pointer)** points to the **base of the current procedure frame** and offers a stable base register as it does not change in a procedure
- Register **\$sp (stack pointer)** points to the **top of the current procedure frame** and can change within a procedure



Preserved or Not Preserved

What is preserved across a procedure call?

- **\$sp is preserved** by the callee by adding exactly the same amount that was subtracted from it
- **Stack above \$sp is preserved** by making sure the callee does not write above \$sp, i.e., the caller will get the same data back on a load from the stack as it was stored there
- **Other registers can be preserved** by saving them on the stack (if they are used) and restoring them from there, specially **registers \$s0–\$s7** and **register \$ra**

Preserved	Not preserved
Saved registers: \$s0–\$s7	Temporary registers: \$t0–\$t9
Stack pointer register: \$sp	Argument registers: \$a0–\$a3
Return address register: \$ra	Return value registers: \$v0–\$v1
Stack above the stack pointer	Stack below the stack pointer

Procedure Call Support

MIPS conventions for procedure calling:

- **\$a0 – \$a3 registers** are used to **pass the first 4 arguments to the callee**
- **\$v0 – \$v1 registers** are used to **return values to the caller**
- **\$t0 – \$t9 registers** are used to **hold temporary values** that can be overwritten by the callee
- **\$s0 – \$s7 registers** are used to **hold long-lived values** that should be preserved across calls
- **\$sp register** is the pointer to the **current top location in the stack**
- **\$ra register** is the return address to the **caller's next instruction**
- **jump-and-link instruction (jal)** jumps to an address and simultaneously saves the address of the following instruction (**PC + 4**) in register **\$ra**
- **jump register instruction (jr)** jumps to the address stored in register **\$ra**

Caller Side

Save not preserved registers

- If the caller expects to use not preserved registers (\$t0 – \$t9, \$a0 – \$a3 and \$v0 – \$v1) after the call, save its values before the call in the current procedure frame

Pass arguments

- The first 4 arguments are put in registers \$a0 – \$a3
- Additional arguments are pushed on the stack and appear at the beginning of the procedure frame (register \$fp points to the base of the procedure frame)

Transfer control to the callee

- Execute a jal instruction to jump to the callee's first instruction and save the return address in \$ra

Callee Side

Allocate memory (and update stack pointer)

- Add a new procedure frame by subtracting the required size from `$sp`

Save preserved registers (and update frame pointer)

- If the callee expects to alter preserved registers (`$fp`, `$ra` and `$s0 – $s7`), save its values in the new procedure frame before altering them (`$fp` only needs to be saved if the frame's size is not zero; `$ra` only needs to be saved if the callee itself makes a call)
- Update `$fp` by adding the new frame's size minus 4 to `$sp`

Put results and return control to the caller

- If the callee returns something, put the result(s) in `$v0 – $v1`
- Restore all callee-saved registers (`$fp`, `$ra` and `$s0 – $s7`)
- Pop the procedure frame by adding its size to `$sp`
- Execute a `jr` instruction to return by jumping to the address in `$ra`

Simple Procedure Call

```
int proc (int arg1, int arg2) {    // arguments in $a0 and $a1
    int r = ...;                  // r in $s0, need to save $s0 on stack
    return r;                     // return value in $v0
}

-----

_main: ...
    li    $a0, ...                # put argument $a0
    li    $a1, ...                # put argument $a1
    jal   _proc                   # jump and link
    ...

_proc: addiu $sp, $sp, -4          # adjust stack pointer
    sw     $s0, 0($sp)            # save $s0
    ...                           # return value in $v0
    lw     $s0, 0($sp)            # restore $s0
    addiu  $sp, $sp, 4            # restore stack pointer
    jr     $ra                    # return
```