UMEÅ UNIVERSITY
Department of Computing Science
Assignmen2

December 22, 2022

Computer Organization and Architecture 5dv193

Assignment 2 - MIPS Simulator user manual

Authors Hanna Littorin, Arvid Sundbom
Usernames c19hln, mai21asm

Graders
Oscar Kamf

Contents

1 Introduction

This document is a user manual for a program which simulates the internal control- and data path-structure of a single-cycle MIPS implementation, using an assembler supporting a subset of instructions of the `MIPS32` instruction set. The aim is to simulate the implementation as closely as practical in order to obtain a better understanding of the internal operation of a modern microprocessor.

Chapter 2 presents the set of supported instructions and register names which can be used in code to be processed by the assembler front-end to the simulator.

Chapter 3 contains information about how to set up the environment before running the simulator, as well as its capabilities and which commands can be given to the simulator, as well as the effect of these commands.

Finally, some example runs of the simulator are presented in Chapter 4.

2 Supported instructions and registers

The assembler program used here as a front-end to the simulator supports the following instructions: **add**, **sub**, **and**, **or**, **ori**, **nor**, **slt**, **lw**, **sw**, **beq**, **addi**, **sll**, **j**, **jr**,**nop**, **srl** and **sra**.

The full instruction names and their given format can be found in Table 1.

The registers supported by the assembler are **t0-t9**, **s0-s7**, **a0-a3**, **v0**, **v1 k0**, **k1**, **zero**, **at**, **gp**, **sp**, **fp** and **ra**. All registers are expected to start with a *$* character in the input given to the assembler. Immediates are expected to be in decimal format, and are interpreted as such.

It is theoretically possible to replace the front-end assembler for another implementation of a MIPS assembler, as long as the result in the form of hexadecimal instructions is identical. However, in order to implement such a change, some limited changes to the source code of the simulator would be required.

Instruction	Format
Add Word	add rd, rs, rt
Subtract Word	sub rd, rs, rt
Add Immediate Word	addi rt, rs, immediate
And	and rd, rs, rt
Or	or rd, rs, rt
Or Immediate	ori rt, rs, immediate
Not or	nor rd, rs, rt
Load Word	lw rt, offset(base)
Store Word	sw rt, offset(base)
Set on Less Than	slt rd, rs, rt
Branch if equal	beq rs, rt offset
Jump Register	jr rs
Jump	j target
No operation	nop
Shift word right logical	srl rd, rt, sa
Shift word right arithmetic	sra rd, rt, sa

Table 1: Supported instructions and corresponding format

3 User guide

The program is written in Rust, specifically Rust 1.65. Hence, before starting to use the program, it is expected that Rust and Cargo, the Rust build-tool, are installed on the system the simulator is to be run on.

With the required version of both Rust and Cargo installed, it is expected that all terminal-commands are executed from the project root folder, named `darken- assignment2` in this case.

Provided that you are in the correct location folder, you will then be able to run the program by executing the command:

```
cargo run input_file, encoded_output_file, listings_file
```

in the terminal, where the `input_file` is the file with the assembly-language instructions, the `encoded_output_file` and `listings_file` are files to which the encoded instructions and the assembly listing will be written, respectively. The output files need not be created manually before executing the program. If the encoded output file path or the listings file path are omitted when executing the above command, the instructions and listings file generated by the assembler will be output to files named $\texttt{default}_{i}nstructions.txtand\texttt{default}_{l}istings.\texttt{txt}, respectively.$

`Executing this command builds a binary with debug information and then executes the binary with the given input files. The binary will be located in folder target/debug after building the project. If increased program performance is desirable, the same`

command can be run with a --release flag, i.e. cargo run input_file, encoded_output_file, listings_file --release. This generates an optimized binary, sacrificing debug information for higher performance. In this case, the binary is located in target/release. Note that in this case, the compilation time required to produce the binary may be increased significantly.

The first output file (i.e. the second command-line argument) will be filled with encoded instructions in hexadecimal format and the second output file is filled with assembly listing information. The listing information contains the memory location of each instruction, the label names and corresponding memory addresses as well as any eventual error messages regarding individual instructions.

The assembler program is silent, meaning there will be nothing written to stdout or stderr, except in cases where there is no reasonable alternative. Therefore, any error messages regarding the input given to the program will be printed in the assembly listing file instead of to stdout or stderr. Only errors such as supplying to few parameters to the program or failing to open any of the given files will produce output to stderr. In the case of the actual simulator, outputs beyond that of the terminal graphical user interface are made to stderr only in cases when an error leading to the halting of the program occurs.

3.1 Input file format

The input file should contain MIPS assembly code, where each line should conform to the following format:

- An optional label name, which must begin on position one on the line and end with a colon. Labels are not allowed to contain any whitespace.
- An optional instruction, of the ones listed in Table 1 on the given format. Must begin with a blank space, regardless of having a label before it or not.
- An optional comment, which should start with the character #.

This means that lines do not necessarily have to contain any instructions. A line could, for example, be made up of just a label or just a comment. Lines are also allowed to be empty.

Note that a Jump instruction is expected to be followed by a label name in this assembler. Hence a decimal representation of the address will result in an error message in the listings file and the program will exit, given that the address is not actually used precisely as a label, being introduced at the beginning of a line, followed by a comma.

3.2 Encoded instruction output file

The encoded instruction output file contains the hexadecimal strings which represent the machine-code version of the instructions in the input file. The representations

are written on a separate line for each instruction. Neither labels nor comments are written in this file, so the output generated to this file is not particularly human-readable. In order to signify that the written strings are in hexadecimal format, each string begins with the characters '0x'. For example, the instruction add $t0 $t1 $t2 would generate the string '0x012a4020'. As can be seen from this example, the alphabetic characters 'a'-'f' are represented in lower case in the generated output.

3.3 Listing output file

The listing output file can be viewed as a combination of the input file and the encoded output file. The listing file contains both the hexadecimal strings corresponding to each instruction in the input file as well as the labels, instructions and comments in the input file. Furthermore, each row containing an instruction also contains the address of that instruction in memory, assuming the first instruction is placed at address 0. Since each instruction is the same size, 32 bits, the address of each instruction is 4 bytes larger than that of the previous instruction address.

If a label is written on the row preceding an instruction, it will be generated on the line preceding the instruction in the assembly listing output file.

Specifically, the assembly listing is generated in a fixed order of columns for each row. The order of these columns are as follows:

1. the address in memory of the instruction
2. the hexadecimal string representing the encoded instruction
3. the label corresponding to the instruction
4. the mnemonic representation of the instruction along with its arguments
5. the comment on the line

Since labels, comments and even instructions are optional for each line of the input, not every line in the listing output will contain each of the items listed above. If an item from the list above is not to be generated in the listing output, it is simply replaced by whitespace, in order to keep the columns somewhat neatly organized in the output.

For example, if the line
label: add $t0 $t1 $t2 comment
is the first line in the input, then the first line in the listing output file will be the following:
0x00000000 0x012a4020 label: add $t0 $t1 $t2 # comment

Finally, the listing output file also contains information from the symbol table which is generated during the first pass of the assembler. Beneath the output whose format

is specified above, each label and the address it corresponds to in the symbol table is printed on the final line of the listing output.

3.4 Running the simulator

Once the code is assembled, provided there was no errors found in the code, a command line interface will be shown, which should look like the one in Figure 1. If an error has occurred in the code, the information about the error will be displayed in the listings_file provided and the instruction memory displayed on the upper part of the terminal window will be empty.

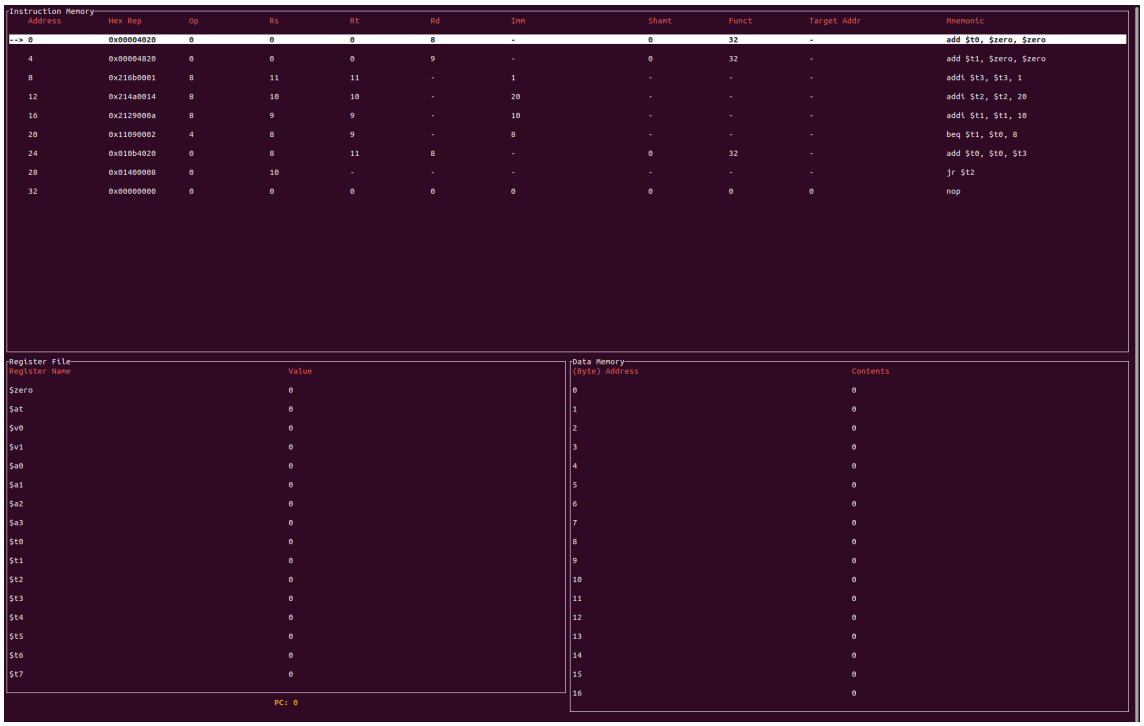

Figure 1: Simulator on startup.

The simulator is a command line interface displaying the instruction memory, registers and data memory locations for the addresses 0-999. The data memory is shown in byte address format, and the value of each memory location is displayed as a value. Both register file and data memory supports a scrolling functionality, which is controlled with the keyboard. To scroll up and down the data memory use arrow up and arrow down on your keyboard. The register file can be scrolled up and down using the 'j' and 'k' keys, respectively.

The upper half of the interface displays the contents of the instruction memory, with the next instruction to be executed highlighted. For each instruction, the address, hexadecimal representation, constituent fields and mnemonic representations are displayed. A column for each constituent field is always presented, with a '_' character presented for fields not present for a given instruction.

In the case of the instructions 'j', 'beq' and 'jr', the value shown in the 'Target Addr' field is the numerical address corresponding to the target label, presented in the selected presentation mode for numerical values.

The lower half of the interface is split into two vertical columns. The left column shows the name of each of the registers along with the contents of the register, presenting each register on a separate row. The bottom of this column contains the value of the PC register, corresponding to the instruction highlighted in the top half of the interface.

The right column contains the byte-addressable data memory, with the address of each byte presented in the left half of the column, and the contents of each address presented in the right half.

The simulator supports switching between different representation formats for *most* numerical values shown in the interface. By default, values are presented in a decimal format. However, pressing the 'h' button will switch to presenting values in a hexadecimal format, and pressing the 'b' key will present values in a binary format. The 'd' key can be pressed in order to switch back to a decimal representation.

There are two options for executing a program with the simulator. Firstly, the simulator supports "stepping" through the instruction memory, i.e. executing instructions one-by-one, waiting for user input between each instruction. In order to step through the instructions one-by-one, you can press the 's'-button, which will execute the highlighted instruction, as well as any effects on the values stored in the registers or data memory. Secondly, it is possible to let the simulator execute instructions without waiting for user input. This execution mode can be enabled by pressing the 'g' key. In this mode, the simulator will execute the instructions until reaching either an 'exit' directive or the end of the occupied addresses of the instruction memory, which is assumed to be filled sequentially, with no empty slots between instructions.

At any point, the state of the program loaded into the simulator can be reset by pressing the 'r' key. This will reset both the values of all registers, including the PC register, as well as the values stored in all locations of the data memory.

Finally, the 'q' key can be pressed in order to exit the simulator.

4 Example runs

In this section, some example runs of the simulator are presented. The purpose of these example runs is to give a clear view of how the program is actually used, to obtain a better understanding of how the numbers are presented and should be presented and so on.

To see examples of input and output to the assembler, we refer to Appendix A.

Figure 2 presents an example simulation which uses the load word and store word instructions,

displaying all fields in hexadecimal format. It highlights the end of the program, which is at the nop instruction, and the PC value is 28. In this example, we scrolled down to visualize the registers $t0-$t3 as well as to word address 0x4, which was utilized in the program. As we can see, after this run, the number 1000 has been successfully added to register $t3, stored in memory and then loaded back to register $t0. As seen in the datamemory, it is stored on word address 4, on byte address 6 and 7.

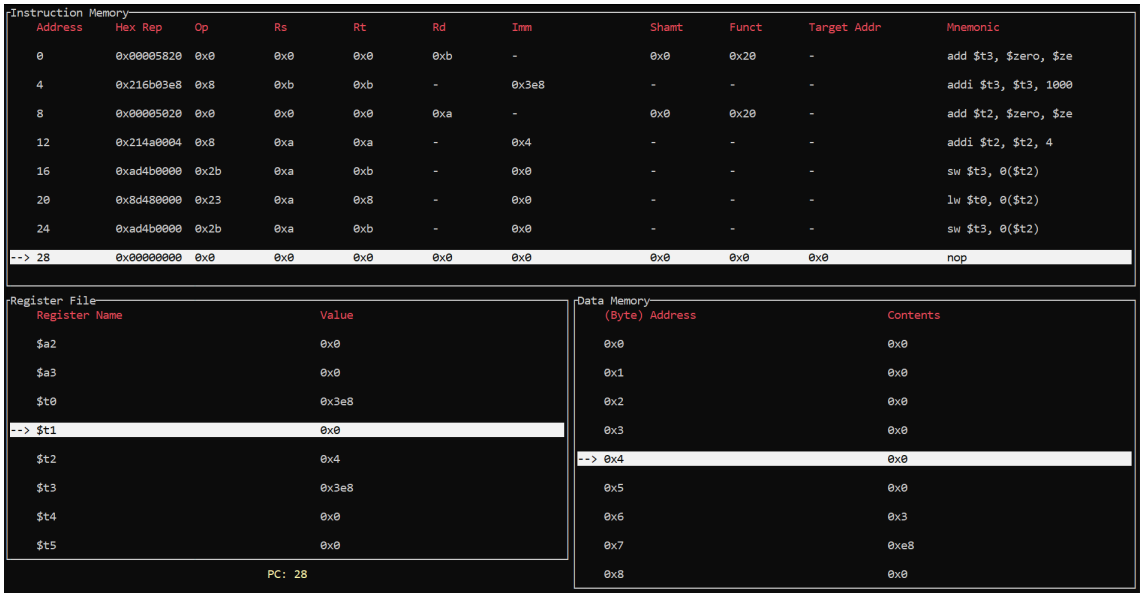

Figure 2: Simulator program using lw and sw instructions.

Figure 3 shows the simulator when loaded a beq instruction into memory, this time displaying all fields in decimal format. As seen, we've reset the registers $t0-$t1 and then added 1 to them twice. Hence, they are both storing 2 in each register. After the beq instruction, it should then branch to address PC + 4 + imm. Where PC on is 24 in this case and imm is -20. Keep in mind that the original instruction used labels, and not decimal values for where the MIPS should branch if the two registers are equal. If wanting to double check addresses of the labels that was put into the assembler on starting the simulator, we refer to check the file that will be filled with listings information. See assembler expected input and generated output under 3.

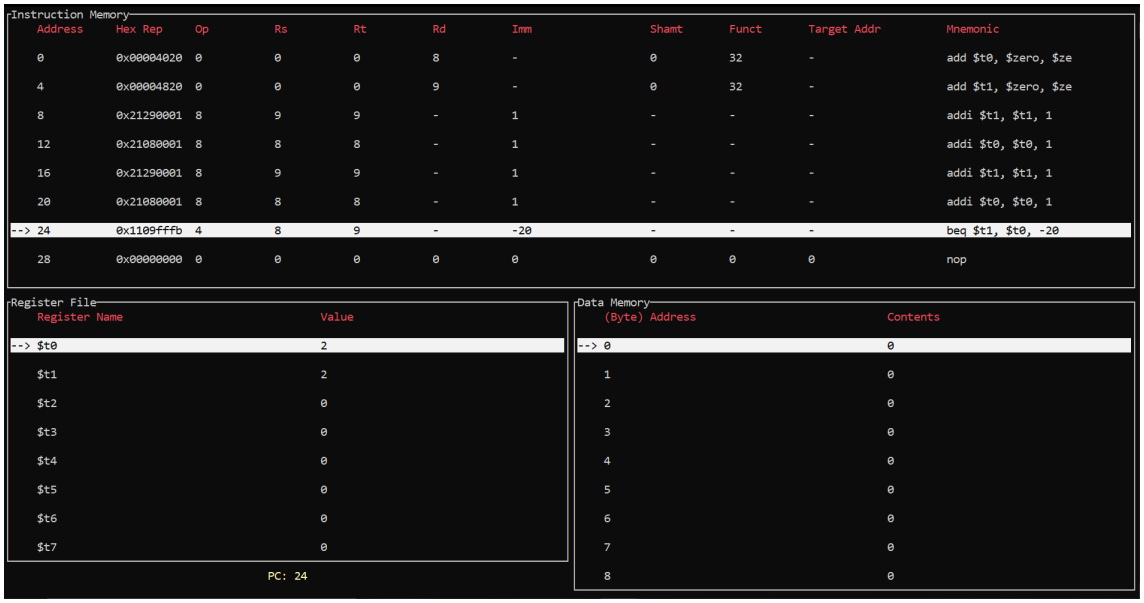

Figure 3: Simulator program using beq instruction.

A Example input and output assembler

Two example runs will be presented, where the input to the first run is a simple file, containing only a few simple instructions, along with some labels and comments. The second example is slightly more advanced, consisting of a larger number of input lines.

The input file for the first example has the following contents:

```
        add $t0, $t1, $t2
label:  sub $t0, $t1, $t2
        add $t0, $t1, $t2
        j label # This is a comment
```

Assuming the file containing this input is named input1.txt, the assembler can be run by executing the following command in the root directory of the project:

```
cargo run input1.txt instruction_output.txt listing_output.txt
```

This generates the encoded instructions as output to the file instruction_output.txt and the listing output to the file listing_output.txt. After running the program, the following content of the instruction output file is generated:

```
0x012a4020
0x012a4022
0x012a4020
0x08000001
```

Furthermore, the following is the contents of the generated assembly listing file:

```
0x00000000     0x012a4020                          add $t0, $t1, $t2
0x00000004     0x012a4022                  label:  sub $t0, $t1, $t2
0x00000008     0x012a4020                          add $t0, $t1, $t2
0x0000000c     0x08000001                          j label # This is a comment

Symbols
label     0x00000004
```

As can be seen from these two files, the output file containing encoded instructions consists entirely of hexadecimal strings and is not very easy for humans to read. On the other hand, the listing file consists of both the contents of the input file and the encoded instructions, along with additional useful information.

The input file for the second run of the assembler is slightly longer, but follows the same basic structure as the previous input:

```
# Test program for assignment 1 - a MIPS Assembler
    nor $t1, $zero, $zero
```

```
    sub $t1, $zero, $t1
    add $t2, $t1, $t1
    add $t3, $t2, $t1
    and $t4, $t2, $t3
    or  $t4, $t2, $t1
    slt $t5, $t1, $t2
    add $t5, $t3, $t1
    sw  $t2, 4($t5)
    lw  $t4, 4($t5)
    nop

label: nop # sub $t4, $t3, $t3
    nop
    nop
    sub $t4, $t3, $t3
    beq $t4, $zero, label
    nop
    jr $t2
```

This input is both longer than the previous input and makes use of additional instructions that were not present in the previous example. Despite this apparent increase in complexity of the input, the output files for this example both follow the same basic structure as the previous ones, and therefore contain similar contents. The program can be run on this input by executing the same command as in the first example run, but changing the input file argument to refer to the file containing this input instead.

Running the program with the above input results in the following encoded output:

```
0x00004827
0x00094822
0x01295020
0x01495820
0x014b6024
0x01496025
0x012a682a
0x01696820
0xadaa0004
0x8dac0004
0x00000000
0x00000000
0x00000000
0x00000000
0x016b6022
0x1180fffb
0x00000000
```

```
0x01400008
```

As the previous case, this file contains only hexadecimal strings representing machine-understanda
instructions, and is not easy for human readers to understand.

The more interesting case of the assembly listing file contains the following:

```
                                        # Test program for assignment 1 - a MIPS Assembler
0x00000000      0x00004827               nor $t1, $zero, $zero
0x00000004      0x00094822               sub $t1, $zero, $t1
0x00000008      0x01295020               add $t2, $t1, $t1
0x0000000c      0x01495820               add $t3, $t2, $t1
0x00000010      0x014b6024               and $t4, $t2, $t3
0x00000014      0x01496025               or  $t4, $t2, $t1
0x00000018      0x012a682a               slt $t5, $t1, $t2
0x0000001c      0x01696820               add $t5, $t3, $t1
0x00000020      0xadaa0004               sw  $t2, 4($t5)
0x00000024      0x8dac0004               lw  $t4, 4($t5)
0x00000028      0x00000000               nop

0x0000002c      0x00000000              label: nop # sub $t4, $t3, $t3
0x00000030      0x00000000               nop
0x00000034      0x00000000               nop
0x00000038      0x016b6022               sub $t4, $t3, $t3
0x0000003c      0x1180fffb               beq $t4, $zero, label
0x00000040      0x00000000               nop

0x00000044      0x01400008               jr $t2

Symbols
label     0x0000002c
```

Although the supplied input was arguably more complex for this second run of the assembler,
the listing file follows the same general output format as in the first case, with
the same columns in the same order.