Umeå University

Computer Science Department, 2012/2013

Computer Organization and Architecture
Assignment 1

Authors :
Rémi Destigny (ens12rdy@cs.umu.se)
Paul Laturaze (ens12ple@cs.umu.se)
Isaline Laurent (ens12ilt@cs.umu.se)

January 14, 2013

Introduction

MIPS instructions are represented as 32-bits numbers. Those numbers are split in fields, which indicate what the instruction is supposed to do. The common point among all instructions is the *opcode* field which is used to find out the instruction familly it belongs to. This field is always stored in the first 6 bits. All instructions comply with a format, which determines which fields are used, and how. There is several format, each corresponding to a part of this report.

The aim of this program is to analyze a file containing MIPS instructions in either hexadecimal or decimal representations. In output it must provide for each instruction the following information :

- The number analyzed, from the input file.
- The format of the instruction.
- The decomposed representation in decimal.
- The decomposed representation in hexadecimal.
- The decomposed representation in mnemonic format.

Contents

1 Building the solution

The solution has been implemented in Java, version 1.6.

Source Files are in the sub-repository named "src". Compiled classes are supposed to go into "bin". You can find input and output examples in "data".

In order to build the solution, go in the root repository of the project. Then type the following instruction in a shell :

```
javac -cp src/ -d bin/ src/Main/Dissasembler.java
```

2 User manual

2.1 Running the program

Once building is done, the program can be run using the following command :

```
java -cp bin/ Main.Dissasembler <Input file name> <output file name>
```

`Input file name` is a path to a file where each line represent a MIPS instruction encoded in a decimal or hexadecimal value. The `output file` is an HTML page and can be viewed using any internet browser.
The input file must contain on each line something like :

- `0x2931fb2e` wich correspond to the following mnemonic : `slti $s1, $t1, -1234`
- `556860626` wich correspond to the following mnemonic: `addi $s1, $t1, 1234`

The list of the recognized instructions is given in chapter 3 of this report.

2.2 Understanding the output

The html page generated offers an array with 5 columns :

- Value : The hexadecimal value given as input on this line. If the value was in decimal, it will be automatically converted.
- Format : The MIPS format of the instruction, can be R, I or J for the common MIPS instruction. Others possible values are C for coprecessor instructions, BC for copressor branch instructions, E for the eret instruction and IRQ for syscall and break.
- Mnemonic : Display the instruction mnemonic as it should appear in a MIPS assembly source code.
- Decimal decomposition : The decimal value of each field according to the format of the instruction.

- Hexadecimal decomposition : The hexadecimal value of each field according to the format of the instruction.

Each line in the html output refers to a line in the input file

3 Handled MIPS instructions

3.1 Classical formats

R format

There is two possible decompositions for an instruction in R-format. The first one is the following :

opcode (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	shamt (5 bits)	function (6 bits)

Table 1: R-format first representation

The second one is described below :

opcode (6 bits)	rs (5 bits)	rt (5 bits)	0 (10 bits)	function (6 bits)

Table 2: R-format second representation

The second representation corresponds to mnemonic representations which only use registers *rs* and *rt*. From mnemonic representations following recurrent display formats can be extracted :

	Fields to display	**Operation code** and **Function code**	
function_name	rd	**Opcode** 0	**function_code** 16, 18
function_name	rd rs	**Opcode** 28	**function_code** 32, 33
function_name	rd rs rt	**Opcode** 0 28	**function_code** 10, 11, 32, 33, 34, 35, 36, 37, 38, 39, 42, 43 2
function_name	rd rt imm	**Opcode** 0	**function_code** 0, 2, 3
function_name	rd rt rs	**Opcode** 0	**function_code** 4, 6, 7
function_name	rs	**Opcode** 0	**function_code** 8, 17, 19
function_name	rs rd	**Opcode** 0	**function_code** 9
function_name	rs rt	**Opcode** 0 28	**function_code** 24, 25, 26, 27, 48, 49, 50, 51, 52, 54 0, 1, 4, 5

Table 3: Reccurent display formats

To determine a mnemonic representation from hexadecimal or decimal value, two fields are important : the *opcode* field and the *function* field. For R-format instruction the *opcode* field can take two values : 0 or 28 (in decimal). Then *function* field is used to get the corresponding mnemonic representation.

I format

Instructions in I-format correspond to the following representation :

opcode (6 bits)	rs (5 bits)	rt (5 bits)	imm (16 bits)

Table 4: I-format representation

From mnemonic representation, following recurrent formats can be extracted :

	Fields to display	**Operation code** and **rt value**
function_name	rs imm	**Opcode** **rt value** 1 8, 9, 10, 11, 12, 14
function_name	rs label	**Opcode** **rt value** 1 0, 1, 16, 17 6, 7
function_name	rs rt imm	**Opcode** **rt value** 9
function_name	rs rt label	**Opcode** **rt value** 5
function_name	rt addr	**Opcode** **rt value** 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 46, 48, 56
function_name	rt imm	**Opcode** **rt value** 8
function_name	rt rs imm	**Opcode** **rt value** 8, 10, 11, 12, 13, 14

Table 5: Recurrent mnemonic format for instruction in I-format

To determine a mnemonic representation from its hexadecimal or decimal value, the most important field is the *opcode* field which can take the following values : 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 46, 48, 56. If the *opcode* is equal to 1 then the *rt* value determines which function has to be displayed. That value can be : 0, 1, 8, 9, 10, 11, 12, 14, 16, 17.

J format

Instruction in J-format have the following representation :

opcode (6 bits)	target (26 bits)

Table 6: J-format representation

Mnemonic representation for that type of instruction is described below :

	Fields to display	**Operation code**
function_name	target	2, 3

Table 7: Mnemonic representation for J-format instructions

To determine the mnemonic representation from its hexadecimal or decimal value, the only important field is the *opcode* field which can take one of the following value : 2, 3.

3.2 Custom formats

Some instructions have a special format which does not match R,I or J. These instructions are the following : `bc1t`, `bc1f`, `mtc0`, `mtc1`, `mfc0`, `mfc1`, `eret`, `syscall`, `break` and `nop`.

Some of these operations involve the coprocessor, which are identified with C-format. For those which branch on the coprocessor, BC-format is used. Interruption as `syscall` and `break` are in IRQ-format. `eret` and `nop` have their own format, respectively E-format and NOP-format.

C format

The C-format has the following structure :

opcode (6 bits)	format_code (5 bits)	rt (5 bits)	rd or fs (5 bits)	0 (11 bits)

Table 8: Description of C-format

The mnemonic representation depends on the value of *opcode* field and on the value of the *format_code* field, which respectively take value : 16 or 17 and 0 or 4.

BC format

The BC-format has the following structure :

opcode (17) (6 bits)	8 (5 bits)	cc (3 bits)	flag (2 bits)	offset (16 bits)

Table 9: Description of BC-format

The mnemonic representation depends on the value of the *flag* field, which take value : 0 or 1.

IRQ format

The IRQ-format has the following structure :

opcode (0) (6 bits)	code (20 bits)	function_code (6 bits)

Table 10: Description of IRQ-format

The mnemonic representation depends on the value of the *function_code* field, which take value : 12 or 13.

E format

The E-format has the following structure :

opcode (6 bits)	flag (1) (1 bits)	code (0) (19 bits)	function_code (24) (6 bits)

Table 11: Description of C-format

The mnemonic representation for this format only match `eret` instruction.

NOP format

The `nop` instruction is treated as a special instruction. `nop` has the following structure :

0 (6 bits)	0 (5 bits)	0 (5 bits)	0 (5 bits)	0 (5 bits)	0 (6 bits)

Table 12: nop representation

That representation is the same as `sll` with *rs*, *rd*, *rt* and *shamt* set to 0. So to differentiate these two instructions, an attempt to match `nop` representation is done before trying to match other representations.

Conclusion

To conclude on this project, we offer a MIPS disassembler able to understand most of the MIPS instruction set and to produce a clean output. Thanks to the code architecture, adding the missing instructions to support the whole set would be really quick and would require a minimum amount of code.