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

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.

3 Implementation

The input file is processed in two steps : Parsing the file and decoding the data. The parsing step is fairly simple, for each line, we detect whether the number is in a decimal representation or an hexadecimal one. Once this is done, the value is converted to a binary string and added to a list for later computation.

The computation part was a bit tricky, we wanted a solution with as few code duplication as possible but still able to handle new instructions if some were to be added. Since the ultimate goal of this project was to give a mnemonic representation of the instruction, all the handled instructions were sorted by representation and a Java class was created for each representation. As most of the informations and display functions are the same from an instruction to another, the inheritance mechanism was used to avoid code duplication. An abstract class called *Instruction* was built with the prototypes for the basic functions we wanted : printing the mnemonic, the decimal and hexadecimal decomposition and the format of the instruction. Another set of abstract classes was used for the most common instruction format : *RegisterInstruction*, *JumpInstruction* and *ImmediateInstruction* respectively for the R, J and I formats. These class were used to handle format specific representation which were common to all members of the instruction set. For instance the *RegisterInstruction* class managed the decimal and hexadecimal decomposition of the instruction because it was the

same for all the R format instructions regardless of the mnemonic output format.

In order to sort the instructions, every representation class contains an array of operation code and/or an array of function code linked to the name of the corresponding instruction. Each binary string goes through a check against every class operation code array and then if needed every function code too which allows the program to instantiate the class corresponding to this instruction representation. Every representation class implements a *printMnemonic()* function which print the mnemonic using the information from the binary string and according to the class representation pattern. All the informations stored in the classes are then gathered and put in an HTML array which is a convenient way to display the data without requiring our program to use any external GUI library.

4 Handled MIPS instructions

4.1 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.

4.2 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.