MIPS Assembler

Ayoub Bargach

Grenoble INP
https://github.com/bargacha/MIPS

November 23, 2017

Abstract

This documentation aim to explicit the way the MIPS assembly have been built. This project have been followed through my education in Phelma - Grenoble INP. A basic source files have been written by Francois Portet and Francois Cayre. The project goal is to build a complete assembler in C language. The biggest part is open source in my github. Due to lack of information about initial code license, formatting files have been deleted. I'm working on creating a more visual interface for education purposes. From structure to code, you will find a lot of explanation of how an assembler is built. A special thanks to all those who took time to help me when we was stucked.

I. Introduction

The goal of this computing project is to build an assembler for MIPS processors using C language. This assembler will be able to translate a human understandable language (Here assembler) to a machine understood language (binaries). This binaries are designed for MIPS 32 bits processors. As an input, the program takes an object file that is actually a text file. Depending on options, it will produce :

- An assembly list.
- A binary object.
- An object in ELF format.

We will **not** implement all the instruction set. Indeed, the purpose is to highlight the functioning of an assembler, not to build a brand new one (gcc do it well). This project have been built in 4 deliverables (or sprints). For each one, there is a description of sprint goals, functions that must be implemented and those who are effectively implemented.

II. Run and test

To run the assembler, use makefile to compile the program (**debug** if you want to print logs and **release** if not). You can use some options to explicit to the program what you want as a result. It is not mandatory to use them, by default, you will get an object ELF file.

```
./as-mips [Options ..] pathToFile
```

Table 1: *Options*

Option	Function
-l	To produce an assembly list
-b	To produce only a binary object
-r	For ELF release
-t	To run a test

Note : Each test is speciafied by an ID. Just fill the ID to generate the test. (Can be run either in debug and release) Please refer to appendix A to have a precise description on every test.

III. The project

i. What is MIPS ?

MIPS is a reduced instruction set computer (RISC) instruction set architecture. Designed by MIPS Technologies, it have been introduced in 1985. The philosophy of a RISC architecture is to offer a reduced -means in a single memory cycle- instruction set. It doesn't mean that there is less instruction. In fact, RISC instruction are sometimes bigger than CISC ones. RISC architectures have fixed-length instructions and simplified incoding that enable an improved code density. Also, it is based on load/store architecture that means that the result of calculations is made by specific instructions. ALU instructions mainly use general purpose registers that simplify compiler design. RISC processors solve generally an instruction in one cycle.

Like major RISC processors, MIPS are featured in a Harvard memory model that seperate physically the instruction memory from data memory that we distiguish by using directives .text and .data. Thus, changing data does not impact the code.

In this project, we aim to focus on a specific MIPS processor, the first made one. We will not be interessed in application-specific extensions such as MIPS MCU or MIPS DSP (For signal processing). The aim is to build a simple binary ELF object that can be run using MIPS I for instance. All extension that can be seen in other MIPS extension such as floating point instruction will not be handled.

The MIPS architecture is based on pipeline architecture. As a picture often speaks better than a thousand words, here a simplified drawing of the first commercialized MIPS processors R2000 :

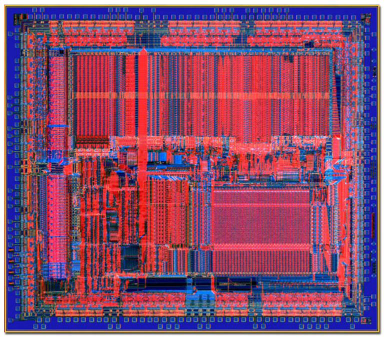

Figure 1: *Layout of MIPS R2000*

This pipeline is organised in several stages :

Instruction Fetch This stage retrieve the next instruction to decode using the program counter.

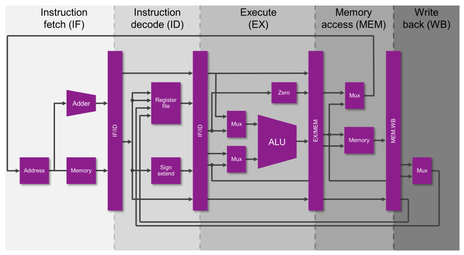

Figure 2: *Pipeline of MIPS R2000*

Instruction Decode The retrieved instruction is decoded following the type of the instruction.

Execute The instruction is executed. The execution depends on the decoded input. MUXs are used to manage the execution and control the ALU input.

Memory access Give access to memory in order to load or store data in registers.

Write back Upload the value of some register depending of the execution results.

This pipeline is the same in many RISC processors. If you are interested on how a processor is designed, here a VHDL description of an ARM processor with the same number of stages (Of course, the instruction set is completely different !) : https://github.com/bargacha/ARM-Processor

Ideally, each instruction is made in a cycle without hazards. However, when instructions depends on older ones or when branch instructions are triggered, it may have some delay.

The MIPS processor have a 4Go memory adressable by bytes. It means that incrementing Program Counter (PC) is the same as jumping 8 bits. In addition, all MIPS processors are *big endian* : The high byte first, followed by the lower bytes.

ii. All features

All features are recorded in this section. In order to manage correctly the project, I started by pinpoint the subject and describe what will be implemented in our basic assembler project. The main goal is to :

> **Implement a basic MIPS assembler. As a input, it will get a standarized MIPS assembler file and should produce as an output an ELF binary file that can be executed using QEMU, an emulator for many basics processors.**

Features are :

- An assembler for MIPS 32 bits microprocessor.
- Manage 32 General Purpose Registers with support of conventional notations ($sp means register $29).
- Main instructions are supported. The standard provided by MIPS Technologies will be followed. Please refer to InstSet.txt to see wich ones are managed. **By following the syntax, you can add instructions directly in this file !** Also, pseudo-instructions are implemented in the same file.
- Comments are managed.
- Main directives are managed : .text, .data, .bss, .set (only noreorder), .word ...
- All adressing modes will be implemented.
- Can take options that define wich input will be produced. (See section 'Run and Test')
- Can produce an assembly list that explicit each line of the source code.
- Can produce a binary object.
- Can produce an ELF binary object understood by in many OS.
- Using -t option can print some testing logs. No output file are produced in this case.

In addition, at the time of writing, I plan to offer some basic optimisation examples to reduce hazards in pipeline processors. To do so, please add the option -o.

iii. Program structure and delivrables

This document is organised according to our professors intructions. Indeed, this project have been made in my college context. It means that code is built around deliverables to provide each 3 weeks. Here a description of each deliverable :

Deliverable 1 Lexical analysis. Understand the signification of each word and build a chain of lexemes.

Deliverable 2 Syntaxic analysis. All instructions are fetched and decoded.

Deliverable 3 Syntaxic analysis. The instructions syntax is verified and relocation inputs are generated.

Deliverable 4 The outputs are generated. (Assembly list, binary object and ELF binary object)

The program structure is :

main.c The main code.

lex.c Lexical analysis.

syn.c Syntaxic analysis.

eval.c Used in syntaxic analysis to evaluate relocation informations.

print.c To generate outputs.

functions.c Some useful functions.

For more information, please refer to the head of each source where a complete description is provided. You can also use Doxygen.

iv. Project management tools

In order to keep the project organised and be able to work with other collaborators, all sources will be shared in my github. To get initial sources, here the college link : http://tdinfo.phelma.grenoble-inp.fr/

In addition, a project management tool will be used : Odoo. It is an open source tool based on kanban models. There is many tools for collaborative work. In order to reduce costs, I took a free AWS server and I implemented a basic configuration provided by http://www.pragtech.co.in/ based on ubuntu.

Finally, this report is written using LateX. The sources are also available in my github.

The program will be tested for ubuntu 16.04 and centos 6 standard configuration.

IV. Delivrable 1

i. Goals and features

In this first deliverable, we aim to produce a complete lexical analysis. The main function is :

```
void lex_standardise( char* in,
char* out )
```

It is responsible of the input shape and other upstream processing. It cleans redondant spaces and help to detect comments by puting a space before #. The following function uses a simple strtok function to split a line into different word. It is mandatory to have a fully functionable function !

This function do some adjustments :

- Add a space before and after chars ',', ')', '(' and '#'
- No space before and only one after ':'
- No space after and only one before '-'

The next step concerns the construction of lexemes chain. A chain is a coding method in C to structure data. In our case, it will have this structure :

```
typedef struct chain_t {
  struct chain_t *next;
  union {
    struct chain_t *bottom;
    lex bottom_lex;
    inst bottom_ins;
    symbol sym;
    code c;
  }this;
} *chain;
```

In order to symplify the code, We use the same structure for all types of chains : symbol tables, code chain ... This able us to use chain elements of different types and combine them. For this purpose, we use the directive 'union' so we can manage different types of variables and reducing by the way the structure length.

For lexemes chain, here a simple drawing of how everything is working. First, the basic element of a chain :

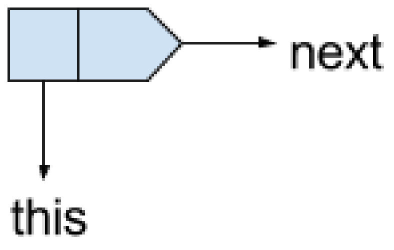

Figure 3: *Basic element of the chain*

Secondly, the lexemes chain structure. Red arrow means that it points using "struct chain_t *bottom;". To express the end of a chain, we use NULL operator. Black arrow contains a lexeme in this case using "lex bottom_lex;".

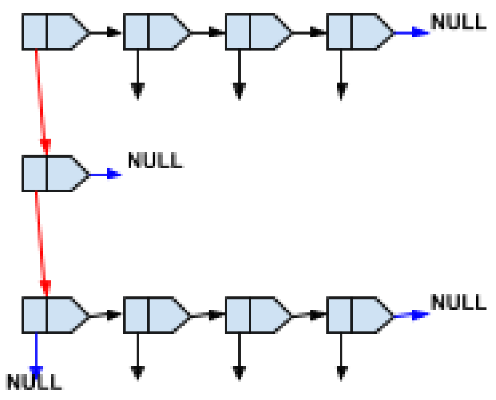

Figure 4: *Lexemes chain*

To recognize lexemes, we use a FSM (finite state machine). Here another drawing that explicit the general function. For more information, the code have been commented to specify each step.

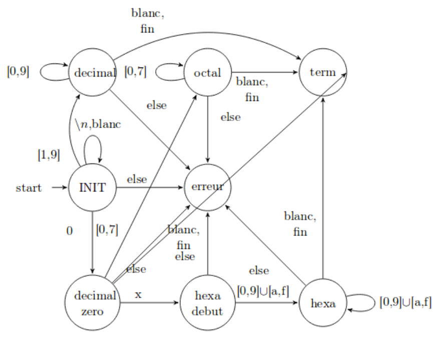

Figure 5: *FSM - Credits to Francois Portet*

In the end of this deliverable, we must have a clean collection of lexemes. It means that sometimes, we may have some errors. For example, the lexeme '0R6155' is not legal. Indeed, this syntax means that we expect an octal number and the char 'R' is not expected. However, the program understand some basic deviations such as '099' that can be translated to a Decimal number.

Thus, when a lexeme is not legal, a error is raised by a specific sentence : 'Lexical error'.

ii. Examples and test

To try lex_standardise, run test 1 using file miam.s for instance :

```
./as-mips -t 1 tests/miam.s
```

It will print you the input and the standardised output. Normally, this function manage all basic cases. Of course, there is no further verifications in lexical analysis. It means that this function do not raise errors. It just bring an easy readable string that can be used after for further analysis.

To display lexeme chain, **run test 2**. It will print the recognized lexemes and their types.

To conclude :

- A chain of lexemes is generated, it contains all relevant lexemes.
- We manage bad written source code by standardising it before further analysis.
- Comments and punctuations are ignored.
- FSM manages decimal, octal, hexa, labels, registers, directives and symbols.
- You can find all lexeme and chain functions in the file functions.c
- **This step is fully functional.**

V. Delivrable 2

i. Goals and features

In this deliverable, we start by generating the instruction set. The instruction set is the list of all instructions that are taken into account by our program. In MIPS context, there 3 kinds of instructions :

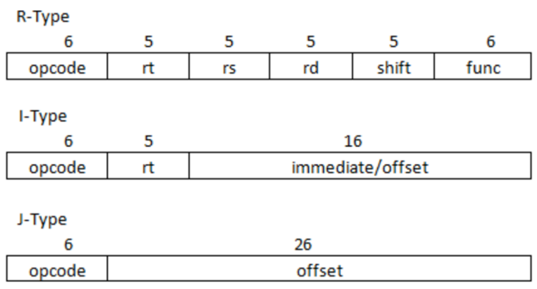

Figure 6: *Instruction types from researchgate.net*

Also, we note that for I-type instructions, there is 2 ways to code it in MIPS assembly code (depending on the instruction), for example :

- For ADDI instruction : ADDI \$rt, \$rs, immediate
- For Load instruction : LW \$rt, offset(\$rs)

For this reason, we have in fact 4 type of grammatical sentences in MIPS Assembly language : R, I (for ADDI), J and IB (for LW). I and IB is a distinction who have nothing to do with the binary decoding. It will just help us to optimize the code.

The list of this instructions are set in the file "instSet.txt". The file can be modified to add other primary instructions (To add for example floating points instructions). The file have been designed as follows :

```
OP OPCODE TYPE OPERAND SPECIAL
SRL 000010 0 0111 rs=00001
```

- OP is the name of operation
- OPCODE, the code translate in binary
- TYPE, decribe the type defined in global.h by "enum {R, I, J, IB};"
- OPERAND, expected operands. For R-type, 0111 means 0 rs 1 rt 1 rd 1 sa. So we need rt, rd and sa to execute the instruction. If not the assembler fail !
- SPECIAL, for future purposes. If an operand is not defined, you can set it (rs=00001). It is very useful to contruct pseudo-instructions.

Using this simple implementation, we build the instSet table using the structure inst.

```
typedef struct inst_t {
  char name[16];
  char op[16];
  int type;
  char operand[8];
  char special[STRLEN];
} *inst;
```

For performance purposes, we use a simple hash function knowing that our table does not exceed hundred lines. Furthermore, the instruction ID will be built by concatenating/decaling ASCII value and calculate it modulo 1000.

Thus, for ADD, Hash function will calculate : $(A + D << 1 + D << 2)\%1000$. Also, the program analyse automatically if there is any collision. If there is one, he raises an error.

In fact, we add all instructions included pseudo-instruction in instSet.txt. If you had more instructions, be aware that you will have to change certainely the hash function. But before, try changing modulo int.

Note : Please respect the instSet syntax in order to avoid any problem. a single double space may occur in unpredication errors.

The next stage is to generate the symbol table. We use the same structure as lexeme chain. we will get something that is near to this illustration :

After that, we range through all lines of lexeme chain to analyse it, in other word, to make the syntaxic analysis. Each step is described in syn.c. The "main" function is fetch(). it uses different functions from decodeInstruction() to decodeDirective(). We "fetch" each line and generate the next elements of code chain, symbol chain and relocation chain. In the end of this process, we will run a final function named eval. This function will make the final links between symtab and undefined yet relocations. (But all of this will be made in the third deliverable)

ii. Examples and test

To print all the instruction tab, please run test 3. It will also print you the ID of the symbol. This test is useful is you want to add instructions. You can also update hash function in order to make it more robust :

```
int hash( char * s, int modulo ) {
  int i =0;
  int result = 0;

  while ( s[i] != '\0' ) {
    result = result + ( s[i] << i);
    i++;
  }

  return result % modulo;
}
```

To conclude :

- All instruction types are managed : R, I and J.
- A "virtual" type have been added, IB. It manages I instruction of this shape : LW $6, 200($7).

VI. Delivrable 3

i. Goals and features

In this part, we aim to manage all the final functionalities. We must properly generate the symbol Table, so it had to be optimized and robust. Thus, if the symbol table is changed, the relocation table must be also changed.

Relocation entries should be generated. Final verifications and decode will be made, especially for relocations. For instance, a BEQ instruction need a decoded offset. In a first sight, the instruction is decoded without adding any offset. So, if no related symbol is found after, that means that the instruction will keep an offset of zero.

ii. Examples and test

VII. Delivrable 4

i. Goals and features

In the end of this step, the program wil be able to generate a fully functional ELF object file. As it have been specified in former section, there is 3 types of different outputs :

List mode In this mode, we generate the assembly list. It explicit each line of source code with the equivalent decode binary code. You will also find symbols and relocations tables.

Object mode Before each section, we put the lengh of input code. For section .bss, only the lengh will be added.

ELF mode An understandable object file, used by many major OS, focusing on a minimal implementation.

ii. Examples and test

References

[Wikipedia, 2017] Reduced instruction set computer

[Wikipedia, 2017] MIPS architecture

[Wikipedia, 2017]

A. All test functions by ID

Table 2: *Tests by ID*

ID	Description
1	Compare input and output of lex_standarise function
2	Dump the lexeme chain (to validate the first deliverable)
3	Print instruction table