

MIPS Assembler

AYOUB BARGACH

Grenoble INP

<https://github.com/bargacha/MIPS>

November 15, 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 Franois Portet and Franois Cayre. The project goal is to build a complete assembler in C language. All is open source in my github. You will find there all the information step by step in how all of this program is written. From structure to code, you will find a lot of explanation of how an assembler is built. A special thanks to all the professor team who takes time to help us when we are in need.

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 understood language (Here assembler) to a machine understood language (binaries). This binaries are designed for MIPS 32 bits processors. As an input, the program take 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 assembler function, not to build a brand new one (gcc do it very well). To be aware about what this project manage, please refer to sprint 2 section. In fact, this project have been built in 4 deliverables (or sprints). For each one, there is a description of sprint goals and functions that have been implemented.

II. RUN AND TEST

Before we start, use makefile to compile the program (**debug** if you want to print logs and

release if not). In addition, 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 launch tests

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 instructions are sometimes bigger than CISC ones. RISC architectures have fixed-length instructions and simplified encoding 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 1 cycle.

Like in major RISC processors, MIPS are featured in a Harvard memory model that separates physically the instruction memory from data memory. Changing data does not impact the code.

In this project, we aim to focus on a specific MIPS processor, the first one. We will not be interested in Application-specific extensions such as MIPS MCU or MIPS DSP (For signal processing). The aim is to build a simple assembler that can be run in MIPS I for instance. All extension that can be seen in other MIPS extensions 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 :

This pipeline is organised in several stages :

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

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 an information in registers.

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

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 and follows each stage. However, when instructions depend on older ones or when branch instructions are launched, we may have some delay.

The MIPS processor has a 4Go memory addressable by bytes. That means that when increment the Program Counter (PC), we jump 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 assembly project. The main goal is to :

Implement a basic MIPS assembler. As an input, it will get a standardized MIPS assembler file and should produce as an output a ELF binary file that can be executed using QEMU, an emulator for many basic 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 proposed by MIPS Technologies will be followed. Please refer to InstSet.txt to see which ones. **In some conditions, you can add instructions directly in this file !** Also, pseudo-instructions are implemented.

- Comments are managed.
- Main directives are managed : .text, .data, .bss, .set (only noreorder), .word ...
- All addressing 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.
- Can produce a binary object.
- Can produce an ELF binary object used in many OS.
- Using -t option can print some testing logs. No output file are produced.

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

iii. Program structure and deliverables

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 our deliverables to provide each week. Here a description of each deliverable.

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

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

Deliverable 3 Syntactic 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 Syntactic analysis.

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

functions.c Some useful functions.

For more information, please refer to the head of each source where a complete description is provided.

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 model. 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/> base 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, the aim is to produce a complete lexical analysis. The central function is :

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

It is responsible of the correct shape input and some upstream processing. It cleans redondant spaces and help to detect comments for instance by puting a space before. Indeed, 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 adjustment :

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

The next step concern the construction the chain of lexemes. 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_inst;
        symbol sym;
        code c;
    } this;
} *chain;
```

In order to simplify the code, We use the same structure for all types of chains. This able us to user chain elements of different natures and combine chain between them. For this purpose, we use there 'union' directive. So we can manage different types of variables and reducing by the same way the structure lenght in bytes.

For lexemes chain, here a simple drawing of how everything is working :

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

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 raise specified bad the sentence : 'Lexical error'.

ii. Examples and test

To try lex_standardise, run test 1 with the file miam.s :

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

It will print you the input and the standi-rized 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 error. It just bring an easy readable string that can be used faster after.

To display lexeme chain, run test 2. It will print the recognized lexeme and type.

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 managed by our program. In MIPS context, there 3 kinds of instructions :

Also, we note that for instruction of I-type, 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, decrive 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 construct 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/de-caling ASCII value and calculate it modulo 1000.

Thus, for **ADD**, Hash function will calculate : $(A + D \ll 1 + D \ll 2) \% 1000$. 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 certainly 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 syntactic 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;
}
```

VI. DELIVRABLE 3

- Goals and features
- Examples and test

VII. DELIVRABLE 4

- Goals and features
- 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