My MIPS

Version 1
10 December 2012

Epita systems/security laboratory 2013 <contact@lse.epita.fr>

Copyright

This document is for internal use only at EPITA < http://www.epita.fr/>.

Copyright © 2012/2013 LSE < contact@lse.epita.fr>.

Copying is allowed only under these conditions:

- You must have downloaded your copy from the LSE website < https://www.lse.epita.fr/recrutement/>.
- You must make sure that you have the latest version of this document.
- It is your responsibility to make sure that this document stays out of reach of students or individuals outside your class.

Contents

Obligations

- ▷ Read the *entire* subject
- ▷ Follow the rules
- ▷ Respect the submission

Submission

Project managers:	Nicolas Hureau < kalenz@lse.epita.fr> Pierre-Marie de Rodat < pmderodat@lse.epita.fr>
Project markup:	**[MIPS]**
Developers:	1
Submission method:	Git ACU
Submission deadline:	23/12/2011 at 11h42
Project duration:	2 weeks
Newsgroup:	iit.labos.lse.recrutement
Architecture/OS :	x86_64 Linux
Language(s):	C++
Compiler:	**g++**
Compiler options:	**-Wall -Wextra -Werror**
Authorized includes:	All

Instructions

The following instructions are fundamental:

Respect them, otherwise your mark may be multiplied by 0.

They are clear, non-ambiguous and each of them has a precise objective.

They are non-negotiable and you must follow them carefully.

Do not dither to ask if you do not understand any of the rules.

- **Instruction 0:** A file **AUTHORS** must be present in the root directory of your submission. This file must contain the login of each member of your group, leader first, in the following format : a star *****, a space , then your login (ex: login_x) followed by a newline.

  Example:

  ```
  42sh$ cat -e AUTHORS
  * login_x$
  42sh$
  ```

  If this file is not present you will get a non-negotiable mark of 0.

- **Instruction 1:** Respect carefully the output samples' format.
  Examples are indented: **$** symbolizes the prompt, use it as a landmark.

- **Instruction 2:** You have to **automagically** produce a source code using the coding style when it exists for the language you are using. In any case, your code must be clean, readable, never exceeding 80 columns.

- **Instruction 3:** File and directory permissions are mandatory and stand for the whole project: main directory, main directory's files, subdirectories, subdirectories' files, etc.

- **Instruction 4:** When an executable is requested, you must only provide its source code. It will be compiled by us.

- **Instruction 5:** If a file named **configure** is present in the root directory of your submission, it will be executed before processing the compilation. This file must have the execution permissions.

- **Instruction 6:** Your submission must respect the following directory tree, where **login_x** has to be replaced by the login of the leader of your group :

  login_x-my_mips/AUTHORS *
  login_x-my_mips/README *
  login_x-my_mips/TODO *
  login_x-my_mips/src/ *

  The following files are mandatory:

TODO	describes the tasks to complete. Need to be updated regularly.
AUTHORS	contains the authors of the project. Must be EPITA style compliant.
README	describes the project and the used algorithms in a proper english. Also explains how to use your project.

  Your test-suite will be executed during your oral examination (if any) by an assistant.

▷ **Instruction 7**

The root directory of your submission must contain a file named **`Makefile`** with the following **mandatory** rules:

`all`	compiles your project with the correct compiling options.
`clean`	removes all temporary and compiler-generated files.
`distclean`	follows the behavior of **clean** and also removes binaries and libraries.

Your project will be tested by executing **`( [ -x ./configure ] && ./configure ); make`**, then launching the resulting executable file.

▷ **Instruction 8** A *non-clean* tarball is a tarball containing forbidden files: **`*~`**, **`*.o`**, **`*.a`**, **`*.so`**, **`*#*`**, **`*core`**, **`*.log`**, binaries, etc.

A *non-clean* tarball is also a tarball containing files with inappropriate permissions.

A *non-clean* tarball will automatically remove two points from your final mark.

▷ **Instruction 9** Your work has to be submitted on time. Any late submission, even one second late, will result in a non-negotiable mark of 0 in the best case scenario.

▷ **Instruction 10:** Functions or commands not explicitly allowed are forbidden. Any abuse will result in a non-negotiable mark of -21.

▷ **Instruction 11:** Cheating, exchanges of source code, tests, test frameworks or coding style checking tools are severely punished with a non-negotiable mark of -42.

▷ **Advice:** Do not wait until the last minute to start the project.

Introduction

Are you able to work on internal projects of the LSE? It is what this subject is here to assess.

This year, the subject will approach a CPU instruction set known as MIPS. To be more precise you have to develop a MIPS userland emulator. It will be capable of running MIPS program which uses a certain set of syscalls described later.

If you have any question, do not hesitate to ask us either on the news, or in the IRC channel `#lse-recrut` @ irc.epita.fr.

Have fun and impress us!

1 Brief MIPS overview

1.1 History

MIPS is a RISC[1] ISA[2] designed from the ground up to be simple with pipelining in mind. Indeed MIPS stands for Microprocessor without Interlocked Pipeline Stages. MIPS is primarily used in the embedded world. You may find MIPS systems in routers as well as game consoles (Playstation 1 & 2, PSP).

MIPS Technlogies have published several versions of the MIPS ISA, MIPS-I through MIPS-V, MIPS32, MIPS64 and the very recent MIPS R5[3]. They are all backward compatible. In this subject, we will focus on the MIPS-I.

1.2 MIPS-I

MIPS-I is a 32-bit ISA which defines a basic instruction set in which all instructions are 32-bits long. Every operations work on registers and/or immediates, the only way to interact with the memory is via dedicated instructions to load from and store to it.

Registers

There are 32 general purpose registers: $0 to $31. Two are special to the hardware:

- $0 always is 0
- $31 is called the link register. Some branch instructions may store the return address in it

There are two special registers used in multiplications and divisions : HI and LO. And of course there is PC, the program counter, which is not directly accessible.

Although there is no hardware requirement, there are conventional names and uses for the GPRs. You may find such conventions in the manual.

Instruction types

There are three instruction types in MIPS, which makes it very easy to decode and encode:

- R: pure register instruction
- I: Immediate instructions
- J: Jump instructions

Memory

As mentioned above, the only way to interact with the memory is by using load and store instructions. There is only one data-addressing mode: all loads and stores define the memory location with a single base register value modified by a 16-bit signed displacement. Furthermore, memory operations must be word-aligned.

[1] Reduced Instruction Set Computer
[2] Instruction Set Architecture
[3] http://www.mips.com/news-events/newsroom/newsindex/index.dot?id=79069

Delay slot

Some RISC and DSP architectures have a particularity called the delay slot. The delay slot is an instruction that is executed independently of the last instruction result. This feature is a side-effect of the one instruction per cycle goal. For example, when having a branch instruction, you do not know what is the next instruction to be executed and therefore the pipeline will stall (it will be filled with bubbles). To avoid this problem some ISAs specify that the next instruction is always executed, wasting one less cycle. Nowadays, CPUs have out-of-order execution, achieving the same goal at runtime.

Branch delay slots As stated above, branch delay slots are when you encounter a branch or jump instruction, you always execute the instruction right after, even if you take the branch. A small example to illustrate:

```
    ori $a0, $0, 0
    b print_int
    addi $a0, $a0, 1

print_int:
    ori $v0, $0, 1
    syscall
```

```
$ ./my_mips print_a0.exe
1
```

Load delay slots Load delay slots are a different kind of delay slots. When you load the content of a register from memory, it may take time, and MIPS designers allocated a delay slot, meaning the next instruction executed does not see the result of the load, it is not garanteed that you can use the content of the register.

1.3 Your implementation

Of course, this was a very brief overview, and it is far from enough to understand how to implement your own MIPS emulator. But the rest of the research work is up to you. Your job to complete this subject and achieve a good mark is to have your emulator MIPS-I compliant. But, in order to have as many finished projects at the end of these two weeks as possible, there will be some tradoffs.

FPU

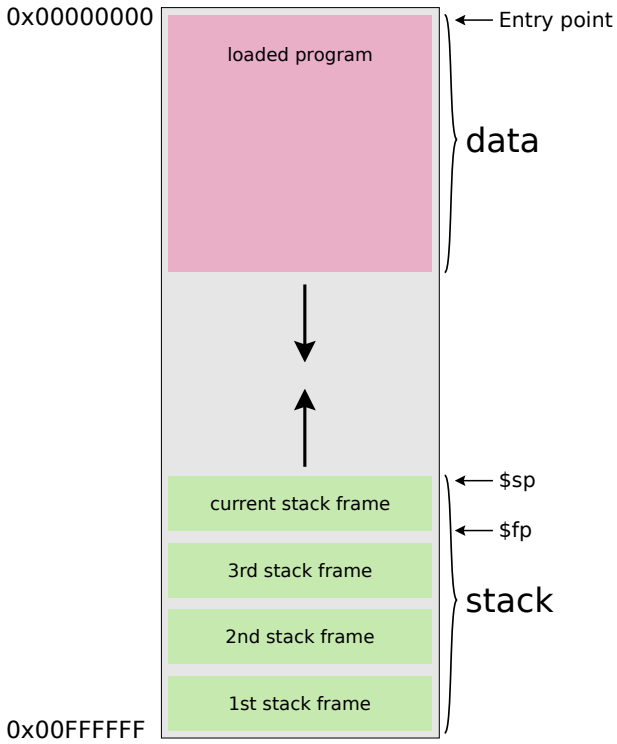
As it is an optional coprocessor (COP1), you do not have to implement the floating-point unit specific instructions. Unless you want to do it as a bonus.

Traps

To simplify your life, instead of trapping, instructions will write 1 to `$k0` (`$26`). Otherwise they will write 0. (In a sense this behavior can be considered as of the trap handler).

Memory

You should use the memory model described below in order to be able to compile C programs and run them on your emulator. In the graphic, memory ends in `0x00ffffff` (16MB memory), however, you may very well want it to end in `0x0000ffff` (65KB memory) or `0xffffffff` (4GB memory, requires more work).



2 Step 1: Hello World!

2.1 First instructions

There are many kind of instructions: do not implement everything before testing! Instead, look at instructions used in examples below and focus on them first.

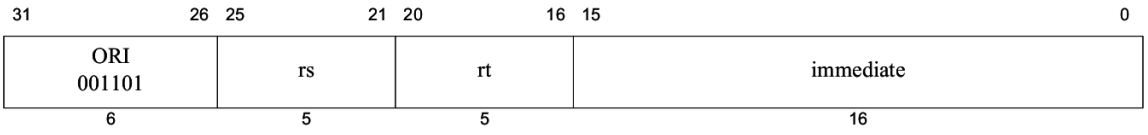
At the end of this step, your emulator should be able to run this simple "Hello World!" piece of code:

```
main:
    # print_string("Hello World!\n");
    ori $a0, $0, hello
    ori $v0, $0, 4
    syscall

    # exit();
    ori $v0, $0, 10
    syscall

hello:
    .asciiz "Hello World!\n"
```

As you can see, you only need to implement two instructions: `ori` and `syscall`. First thing to do when trying to implement an instruction is to look at the MIPS manual. First, let's take a look at what it says about `ori`:



Format: `ORI rt, rs, immediate` **MIPS32**

Purpose: Or Immediate

To do a bitwise logical OR with a constant

Description: `GPR[rt] ← GPR[rs] or immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

None

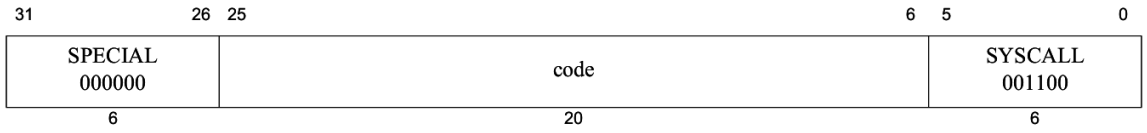
Operation:

```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Exceptions:

None

This description is quite self-explanatory. Let's take a look at `syscall` now:



Format: `SYSCALL` **MIPS32**

Purpose: System Call

To cause a System Call exception

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

```
SignalException(SystemCall)
```

Exceptions:

System Call

As you may have noticed there is an information you need that is missing. Which `syscall` is called? It depends on the system you run on. There are a few already existing MIPS userland emulators, so to ease your development and the testing of your software, we will use their `syscall` convention. They pass the syscall number in the register `$v0` (`r2`). A few of their specific numbers may be found in the following table:

Name	Number	Arguments	Result
`print_int`	1	`$a0` = integer to print	
`print_string`	4	`$a0` = adress of null-terminated string to print	
`read_int`	5		`$v0` = integer read
`read_string`	8	`$a0` = address of input buffer `$a1` = maximum number of characters to read	
`exit`	10		

Once you have implemented `ori` and `syscall`, you should be able to run the following commands and have the same results:

```
$ mipsel-elf-as hello.s -o hello.o
$ mipsel-elf-objcopy -O binary hello.o hello.exe
$ ./my_mips hello.exe
Hello World!
$ echo $?
0
```

For the following steps, you will only be provided the instructions to implement, the rest of the research work is up to you. One thing though: you really should implement the 5 syscalls provided above, as they may be very useful to test your emulator.

2.2 Tests

So that you do not waste time, we provide a test tarball ([4]) which has a few tests. Be careful though, these are very basic tests, you should have more elaborate ones.

C

Once your emulator implements basic features (like aligned memory access), you can start to compile C programs and run them on it. This is very interesting since C makes writing complex programs really easier.

In the `c` directory of the tarball, we put an C program example (`my_program.c`) that computes the factorial of a number.

```
$ make && ./my_program
fact(6) = 720
```

If everything went well, you should get this output for our C program example. If not, go add missing instructions implementation and fix your code!

[4] http://lse.epita.fr/my_mips-tests.tar.gz

3 Step 2: Debug

When one develops a program, there always comes a day when having debug capabilities is very handy. In an emulator, this statement is also true, and perhaps more than in some others cases, because you need to be able to debug your program on your machine, which is possible using for example `gdb`, *and* you also need to be able to debug the CPU you emulate. We will point you to a few facilities that will be useful. Some are trivial, some are not. Of course you are encouraged to implement more if you have the time!

3.1 Printing of executed instruction

This is one of the trivial, yet vital way of debugging your emulator. For example let's take a basic Hello World:

```
$ ./my_mips -l 4 test/hello_world.exe
[my_mips] Executing pc = 0x00000000: 0x34040064: ori r4, r0, 100 (cpu.cc:54)
[my_mips] Executing pc = 0x00000004: 0x34020004: ori r2, r0, 4 (cpu.cc:54)
[my_mips] Executing pc = 0x00000008: 0x0000000c: syscall (cpu.cc:54)
Hello World!
[my_mips] Executing pc = 0x0000000c: 0x3402000a: ori r2, r0, 10 (cpu.cc:54)
[my_mips] Executing pc = 0x00000010: 0x0000000c: syscall (cpu.cc:54)
```

As you can see there are three important informations here:

- PC
- Executed instruction in hexadecimal form
- Same instruction disassembled

These informations may help us on a few levels when writing the emulator:

- To see if we decode an instruction correctly
- To see rapidly where an error occurs in the program
- To show which registers or memory locations may be of importance to debug
- ...

3.2 Registers state

Another trivial way of debugging is to print the registers content.

```
[my_mips] +++ Processor registers +++ (cpu.cc:64)
[my_mips] pc = 0x0000000c (cpu.cc:66)
[my_mips] hi = 0x00000000 (cpu.cc:67)
[my_mips] lo = 0x00000000 (cpu.cc:68)
[my_mips] r0 = 0x00000000 (cpu.cc:71)
[my_mips] r1 = 0x00000000 (cpu.cc:71)
[my_mips] r2 = 0x0000000a (cpu.cc:71)
[my_mips] r3 = 0x00000000 (cpu.cc:71)
[my_mips] r4 = 0x00000064 (cpu.cc:71)
[my_mips] r5 = 0x00000000 (cpu.cc:71)
[my_mips] r6 = 0x00000000 (cpu.cc:71)
```

```
[my_mips] r7 = 0x00000000 (cpu.cc:71)
[my_mips] r8 = 0x00000000 (cpu.cc:71)
[my_mips] r9 = 0x00000000 (cpu.cc:71)
[my_mips] r10 = 0x00000000 (cpu.cc:71)
[my_mips] r11 = 0x00000000 (cpu.cc:71)
[my_mips] r12 = 0x00000000 (cpu.cc:71)
[my_mips] r13 = 0x00000000 (cpu.cc:71)
[my_mips] r14 = 0x00000000 (cpu.cc:71)
[my_mips] r15 = 0x00000000 (cpu.cc:71)
[my_mips] r16 = 0x00000000 (cpu.cc:71)
[my_mips] r17 = 0x00000000 (cpu.cc:71)
[my_mips] r18 = 0x00000000 (cpu.cc:71)
[my_mips] r19 = 0x00000000 (cpu.cc:71)
[my_mips] r20 = 0x00000000 (cpu.cc:71)
[my_mips] r21 = 0x00000000 (cpu.cc:71)
[my_mips] r22 = 0x00000000 (cpu.cc:71)
[my_mips] r23 = 0x00000000 (cpu.cc:71)
[my_mips] r24 = 0x00000000 (cpu.cc:71)
[my_mips] r25 = 0x00000000 (cpu.cc:71)
[my_mips] r26 = 0x00000000 (cpu.cc:71)
[my_mips] r27 = 0x00000000 (cpu.cc:71)
[my_mips] r28 = 0x00000000 (cpu.cc:71)
[my_mips] r29 = 0x00000000 (cpu.cc:71)
[my_mips] r30 = 0x00000000 (cpu.cc:71)
[my_mips] r31 = 0x00000000 (cpu.cc:71)
[my_mips] Executing pc = 0x0000000c: 0x0000000c: syscall (cpu.cc:54)
```

`syscall` uses the content of register v0 (also known as r2) to determine which function to call. Therefore, being able to have a quick look at its value is useful.

3.3 Integrated debugger

A more advanced and powerful way of debugging the code running in your emulator is through the use of an integrated debugger. Even though it may be very helpful, you should consider this a bonus and only implement it if you are absolutely sure you will have the time to finish the other steps.

As an example is worth a thousand words:

```
$ ./my_mips -d test/hello_world.exe
dbg> registers
pc = 0x00000000
hi = 0x00000000
[...]
r31 = 0x00000000
dbg> step
Executing pc = 0x00000000: 0x34040064: ori r4, r0, 100
dbg> print 0x00000000
0x34040064
dbg> break 0x0000000c
dbg> continue
Executing pc = 0x00000004: 0x34020004: ori r2, r0, 4
Executing pc = 0x00000008: 0x0000000c: syscall
Hello World!
dbg> exit
```

4 Step 3: Arithmetic and logic

You have to implement the following instructions.

4.1 Arithmetic

- `add`
- `addu`
- `addi`
- `addiu`
- `div`
- `divu`
- `mult`
- `multu`
- `sub`
- `subu`

4.2 Logic

- `and`
- `andi`
- `nor`
- `or`
- `ori`
- `slt`
- `sltu`
- `slti`
- `sltui`
- `xor`
- `xori`

4.3 Shifts

- `sll`
- `sllv`
- `srl`
- `srlv`
- `sra`
- `srav`

5 Step 4: Data processing

You have to implement the following instructions.

5.1 Loads and stores

- `lb`
- `lbu`
- `lh`
- `lhu`
- `lw`
- `lwl`
- `lwr`
- `sb`
- `sh`
- `sw`
- `swl`
- `swr`
- `lui`

5.2 Special registers

- `mfhi`
- `mflo`
- `mthi`
- `mtlo`

6 Step 5: Jumps and branches

You have to implement the following instructions.

6.1 Branches

- `beq`
- `bgez`
- `bgezal`
- `bgtz`
- `blez`
- `bltz`
- `bltzal`
- `bne`

6.2 Jumps

- `j`
- `jal`
- `jalr`
- `jr`

7 Step 6: Syscalls

You should already have implemented 5 syscalls. Now you are about to implement a few others:

Name	Number	Arguments	Result
`print_int`	1	`$a0` = integer to print	
`print_string`	4	`$a0` = adress of null-terminated string to print	
`read_int`	5		`$v0` = integer read
`read_string`	8	`$a0` = address of input buffer `$a1` = maximum number of characters to read	
`exit`	10		
`print_char`	11	`$a0` = character to print	
`read_char`	12		`$v0` = character read
`open`	13	`$a0` = address of null-terminated filename `$a1` = flags (0 = read, 1 = write) `$a2` = mode (ignore)	`$v0` = file descriptor
`read`	14	`$a0` = file descriptor `$a1` = address of input buffer `$a2` = maximum number of characters to read	`$v0` = characters read
`write`	15	`$a0` = file descriptor `$a1` = address of output buffer `$a2` = number of characters to write	`$v0` = characters written
`close`	16	`$a0` = file descriptor	
`exit2`	17	`$a0` = termination result	

These syscalls are used by SPIM and many other emulators (we try to be compatible here).

To be a complete MIPS-I compliant userland emulator, you also need to implement the `break` instruction.

8 Bonus

These are ideas for bonuses. You are not expected to implement all of them, choose what you want to do! Also you may implement only part of a bonus (as long as it is interesting and you understand what is missing) and then skip to the next.

One side-effect of implementing bonuses is that we will be far more encline to answer questions than if asked about mandatory parts. If you do not understand something, ask!

8.1 ELF loading

Instead of "flattening" the binaries with `objcopy` before loading them into the emulator, loading ELFs directly could be very handy. This bonus is quite easy: you just have to parse the ELF files header and their program header table, then to copy `PT_LOAD` segments content to your emulated memory space and jump to the program entry point!

For more details about ELF structure and loading, read `man 5 elf`. To inspect the content of an ELF file, use the `readelf(1)` command.

8.2 JIT compilation

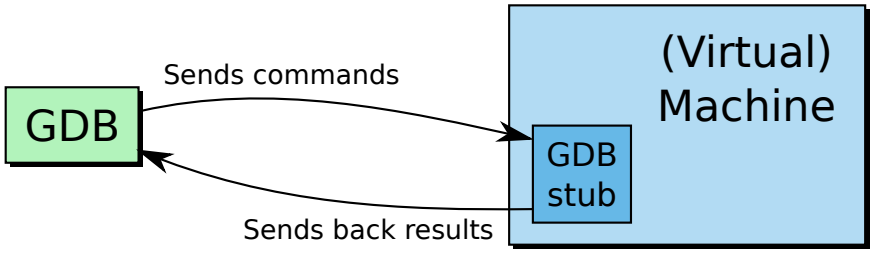
In our emulator, we interpret MIPS bytecode, therefore it can not be as fast as native code. Interpretation of code is quite slow, and there is a few techniques to improve execution speed. The most widely used is JIT compilation.

JIT compilation relies on converting bytecode into native code at runtime, and then executing this native code.

8.3 GDB stub

An more advanced and powerful way to debug emulated programs is to use a GDB that is compiled to debug MIPS programs as your debugger. To do so, your emulator must be able to talk with GDB, for instance through a BSD socket. GDB send instructions to the *GDB stub* (the debugging part of your emulator), and the GDB stub sends back results to GDB. `qemu`'s `-gdb` option uses this tecnics.

This way, your emulator users will be able to use the friendly GDB and all its features (commands, DWARF interpretation, ...) to debug their programs! When your emulator is ready, your GDB stub must be listening on `localhost:1234` (for instance). Launch GDB and execute the command `target remote localhost:1234`. GDB will report you any error that occurs if you do not handle correctly the protocol.



Because it has been generally used to debug kernels on physical machines using the serial link, the protocol involved in the comunication with the GDB stub is called the Remote Serial

Protocol. This protocol is quite simple[5], but there are many kind of instructions. Hopefully, you do not have to implement everything to have a working GDB stub. Just try to use GDB with it and implement features step by step.

8.4 MIPS32

Be MIPS32 compliant!

8.5 FPU

Implement the floating-point unit instructions.

8.6 Game system

Implement your own game system on top of your emulator, add a way of displaying graphics, and another of getting inputs.

8.7 Free porn

If you have an idea, and it is related to the subject, go for it!

8.8 CTF

Do not forget the CTF[6] ;)

[5] http://www.it.uom.gr/teaching/gcc_manuals/onlinedocs/gdb_33.html
[6] http://ctf.lse.epita.fr/