Emulator MIPS

1 Submission

At the end of this tutorial, your submission file will have to respect the following architecture:

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- MyMiniMips.sln*
    |-- MyMiniMips*
        |-- Everything but bin/ and obj/
        |-- CPU.cs
        |-- ALU.cs
        |-- Instruction.cs
        |-- Program.cs
```

You obviously have to replace *login_x* by your own login. Any file that has an asterisk is mandatory. Do not forget to check the following things before submitting your work:

- The *AUTHORS* file follows the usual pattern: * login_x followed by a line break.
- Make sure you do not leave any `bin` or `obj` folder in your zip file.
- **The code MUST compile!**

2 Introduction

The processor, or CPU (Central Processing Unit) is the part of the computer that executes instructions given by programs.

In other words, it is the part that will handle program's data. Your computer's CPU must be an Intel x86/64.

2.1 Objectives

The aim of this practical is quite simple: create a MIPS 32 bit emulator.

- *Emulating* means *trying to mimic*. What we try to emulate here is the MIPS microprocessor behavior.
- A microprocessor is a processor in which the composing parts were miniaturized enough that they can fit a case.
- A machine instruction is what composes any program in your computer. It is the simplest command that the computer can understand and execute.

3 Lecture

3.1 MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a type of microprocessor that is easy to understand and to position.

It also is a type of microprocessor which belongs to the category of RISC (Reduced Instruction Set Computing) microprocessors.As you can guess, RISC microprocessors only have a reduced set of instructions.
Moreover, those instructions are very easy to decode, since they are in binary code of a maximum of 32 bits; and they have very simple behavior such as adding or multiplying.

To recap:

- A processor executes instructions.
- MIPS only includes few instructions.
- A machine instruction is written in binary code and must be decoded.

3.1.1 Registers

A register is a little memory zone that exists inside the CPU. It is the fastest memory in the computer since it is the closest to the CPU.

Those registers are used to store important or intermediate values the processor is working with.

Registers are very important parts of a microprocessor. That is because they are fairly fast, and because most machine instructions modify them.

MIPS owns 32 registers of 32 bits each.

MIPS only has three different types of registers:

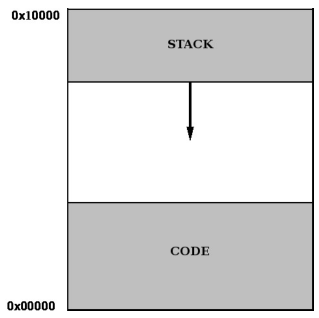
- The *$zero* register, which always contains the value 0.
- The 31st register, or *$ra*, which contains the function's return address. In other words, it contains the address of the program where the execution will continue after the current function ends.
- The 29th register, or $sp, which contains the stack's address.

To recap:

- Registers are small and fast memory parts.
- MIPS has 32 registers of 32 bits each.
- Important registers for this project are *$zero*, *$ra*, and *$sp*

3.2 Memory

We shall have a simplified look at the memory during this practical:



In the first addresses (near 0x00000) are the machine instructions. Above that (at the very top) is the stack, growing upside down.
Data is stored on the stack and their address (the top of the stack) is represented by the stack pointer *$sp*.

3.3 Machine instructions

Machine instructions are encoded in binary and stored in the memory just like any other type of data. That means they can not only be read, but also modified.
MIPS machine instructions match simple actions like adding, subtracting, or multiplying two registers and put the result in the first register specified.

Machine instructions can either modify the program flow (jump...), or command the processor to store a register on the stack, or load one, or even modify a stored value!

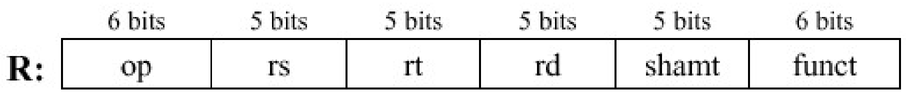
A MIPS instruction has a fixed size of 32 bits or 4 bytes (like an integer in C#) and can be split into several different fields.

Each field tells us something about an instruction and its nature.

MIPS has three type of instructions:

- *R-type* instructions: they mostly act on registers.
- *I-type* instructions: they only use constants.
- *J-type* instructions: they modify the program flow.

3.3.1 R-type instructions



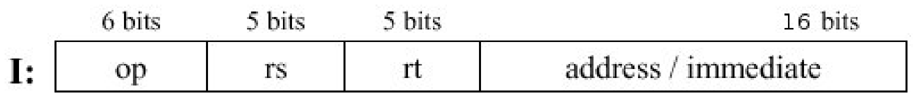
R-type instructions are composed as explained in the previous picture.
One can easily recognize an R-type instruction by its *op* field (or *opcode*): it is set to zero.
Then the *funct* field tells us about the action to execute. Indeed, *funct* gives us the action number.

There is a picture associating every number to its action in the links at the end of the subject.

Finally, the *Rs*, *Rt*and *Rd* fields respectively represent the source register, the destination register, and the action.

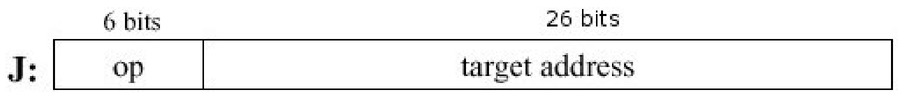
The field *shamt* will not be used in this practical, thus it will not be explained.

3.3.2 I-type instructions



I-type instructions are decomposed as explained in the previous picture.
The *opcode* represents the action to execute. *Rs* and *Rt* respectively represent the source and destination registers.
Finally, *address/immediate* represent a constant (an integer or a memory address) on which we want to execute some action.

3.3.3 J-type instructions



J-type instructions are decomposed as explained in the previous picture.
The *opcode* represents the action to execute. There are two kinds of J-type instructions:

- *Jmp*: allows you to 'jump' in the memory.
- *Jal*: does the same as *Jmp* but saves the address in $ra before jumping. This is the instruction we use to call a function.

4 Exercises

4.1 Part 1: the CPU

In this first part, you will have to code the CPU class. The CPU will be the link between all other objects.
The CPU contains:

- An access to the RAM that you can implement with an array of 64 KiB, which can be seen as 64 * 1024 bytes.
- 32 registers of 32 bits each.
- A side-value called the *Program Counter*, which indicates the address of the current instruction. It starts at 0 and increments each time an instruction is executed.

Moreover, you will implement the following constructor and method:

```
public CPU(string fileName);
public void LoadFile(string fileName);
```

The file contains MIPS instructions under binary form. Thus, you have to copy the file's content in the RAM starting from the address *0x000000*.
The constructor will initialize attributes and call the *LoadFile* function.

Hint: A byte array would be extremely useful...

4.2 Part 2: the instructions

You have to create and complete the Instruction class as follows:

```
public class Instruction {
     int Op    {get; private set; }
     int Rs    {get; private set; }
     int Rt    {get; private set; }
     int Rd    {get; private set; }
     int Funct {get; private set; }
     int Imm   {get; private set; }
     int Addr  {get; private set; }

     public Instruction(int ins);
}
```

The instruction type does not matter here. You have to consider every possible option and fill the fields accordingly. The future use of the class will simply be different when we will use it later.

```
// 0x00004020
// op: 0x00
// rs: 0x00
// rt: 0x00
// rd: 0x08
// funct: 0x20
// imm: 0x4020
// addr: 0x4020
```

4.3 Part 3 : ALU

The ALU is your computer's physical component that handles mathematical operations.

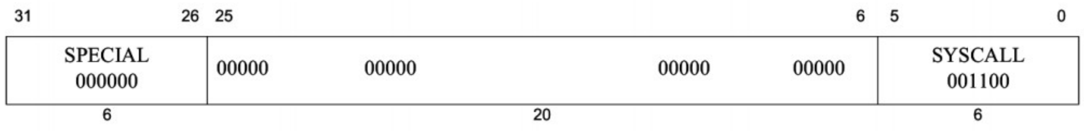
You have to implement the following class:

```
public class ALU {
          public ALU(CPU cpu);
     public void Exec(Instruction i);
}
```

Please note that you will have to implement the *Exec* class throughout this part.

4.3.1 The syscalls

First, you will have to implement the syscall instruction, which is a bit special:



This instruction allows the user to call system functions. For instance, print an integer in the console:

```
addi     $a0, $t0, 0 # move value of t0 into a0
addi     $v0, $0,  1 # displays value in a0
syscall
```

When the *syscall* instruction is executed (on a MIPS architecture), the system takes control of the program for a moment, looks at the register $v0, and executes the function asociated with the number it read.
Here, for example, if the value is *1*, then the function will print the value contained in *$a0*.

Now, you must implement the following function:

```
public void Exec(Instruction i);
```

If it takes a *syscall* instruction, here are the actions that will be executed in function of the $v0 contained value.

- 1: Print the integer contained by *$a0*.
- 4: Print the string address contained in *$a0*, followed by the null value.
- 5: Read an integer and put it in *$v0*.
- 10: Exit.
- 11: Print the character contained in *$a0*.

Please complete the Exec function accordingly:

```
public void Exec(Instruction i);
```

4.3.2 R-type instructions

In this function, you have to be able to handle the following instructions:

- ***add:*** ***Opcode*** 0x20, R[rd] = R[rs] + R[rt]
- ***addu:*** ***Opcode*** 0x21, R[rd] = R[rs] + R[rt] (unsigned addition)
- ***sub:*** ***Opcode*** 0x22, R[rd] = R[rs] - R[rt]

Please do not forget that the *opcode* for *r-type* instructions is their *funct* field.
Please complete the Exec function accordingly:

```
public void Exec(Instruction i);
```

4.3.3 I-type instructions

In this part, you have to be able to handle the following instructions:

- ***addi:*** ***Opcode*** 0x08, R[rd] = R[rs] + SignExtImm
- ***addiu:*** ***Opcode*** 0x21, R[rd] = R[rs] + SignExtImm (non-signed addition)
- ***ori:*** ***Opcode*** 0x22, R[rd] = R[rs] | ZeroExtImm
- ***beq:*** ***Opcode*** 0x04, if (R[rs] == R[rt]) PC = PC + 4 + BranchAddr
- ***bne:*** ***Opcode*** 0x05, if (R[rs] != R[rt]) PC = PC + 4 + BranchAddr

Please complete the Exec function in consequence:

```
public void Exec(Instruction i);
```

Please note that the immediate that an I-type instruction takes must be either *SignExtended* or *ZeroExtended*. Indeed, we have to extend the immediate since it is coded on 28 bits.

A simple *cast* will make a *ZeroExtension*. For sign extensions however, you have to copy the sign-bit (the 28th) and paste it on the 6 remaining bits.

4.3.4 J-type instructions

Finally, you have to handle the following instructions:

- ***j*** **:** ***Opcode*** 0x02, PC = JumpAddr
- ***jal*** **:** ***Opcode*** 0x03, R[31] = PC + 8; PC = JumpAddr

Warning: Due to the fact that your RAM is only on 64KiB, you must only take the 16 first bits of the address. The remaining bits *MUST* be ignored!

4.4 Part 5: Debug

Your function *Exec* is now almost done! To make sure you implemented every step successfully, we ask you to code a debugger. It will simply print every instructions you execute under the following format:

```
[my_minimips] Executing pc = 0x00000000: 0x34040064: ori r4, r0, 100
[my_minimips] Executing pc = 0x00000004: 0x34020004: ori r2, r0, 4
[my_minimips] Executing pc = 0x00000008: 0x0000000c: syscall
Hello World!
[my_minimips] Executing pc = 0x00000004: 0x3402000a: ori r2, r0, 10
[my_minimips] Executing pc = 0x00000008: 0x0000000c: syscall
```

4.5 Part 6: Run

Now you have to test your emulator. This part will consist in linking every part you implemented before.

You have to implement the following function:

```
public void Run();
```

This function will find the current instruction in the CPU class (using the PC). Then it will decode it via the Instruction class, and it will give it to the ALU so that the instruction gets executed.
Finally, do not forget to increment the PC by four, and go back to the beginning of the cycle.

4.6 Bonus

Here is a non-exhaustive list of instructions and/or ideas you can implement:

- FPU: a Floating Point Unit that allows the handling of float numbers, with the corresponding set of instructions.
- JIT compilation
- ...

5 Documents

http://inst.eecs.berkeley.edu/c̃s61c/resources/MIPS_Green_Sheet.pdf
http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

The bullshit is strong with this one