MIPS SIMULATOR

Andrea Fernandez Buitrago
AF1312@ic.ac.uk

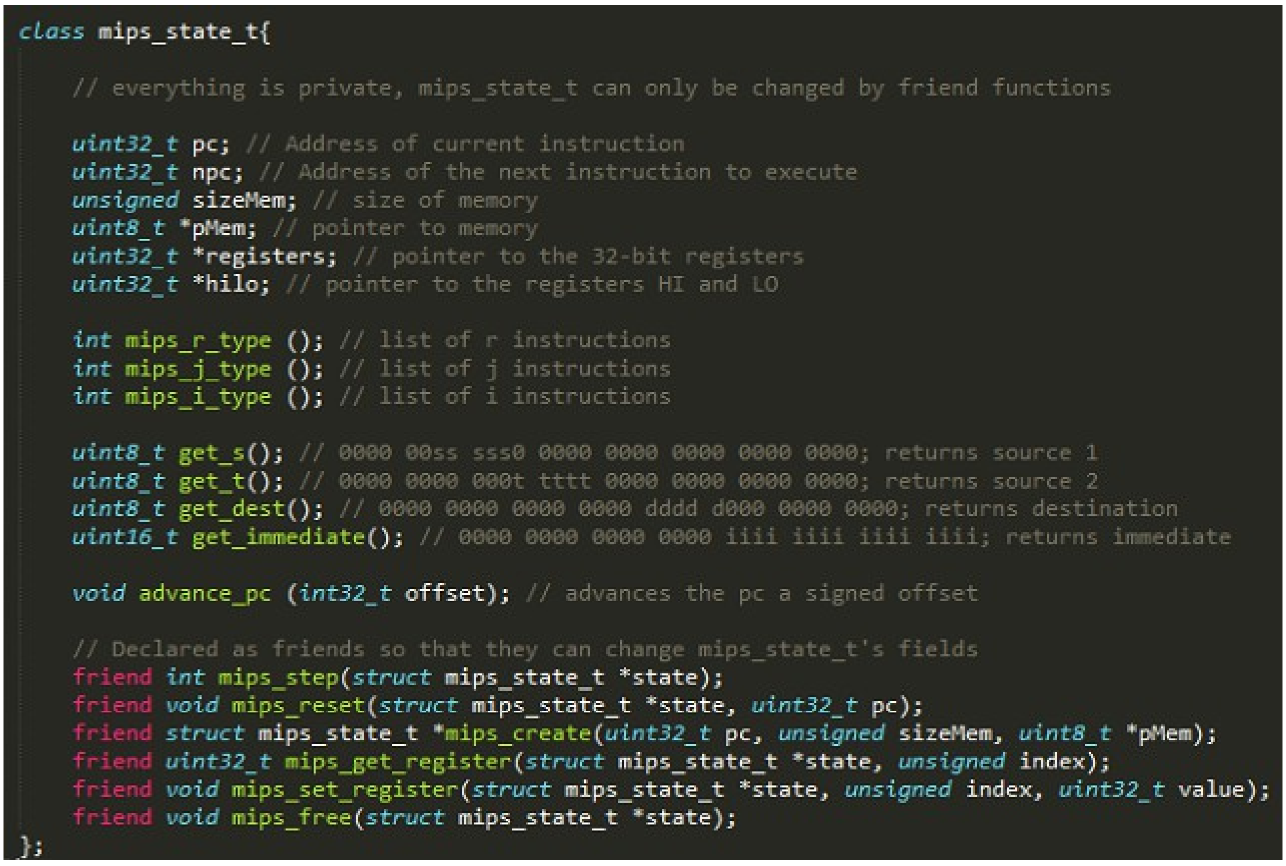
The purpose of this coursework was to create a library that implements a MIPS-1 simulator. We're given some functions that have already been declared, which we have to define, as well as some tests to try our simulator on.

For this coursework I will be using C++, which may give some problems if compiling with GCC (even using extern "C"). As a result of this, I have included the makefile that gives me no problems when compiling the provided tests.

Files included are:

- *mips.h*
- *mips_simulator.cpp* – Implementation of mips.h
- *implementedfunctions.csv*
- *binfile.cpp* – Creates the binary files for the extra tests.
- *Inputs* folder - Contains extra tests and a makefile.

First of all, we define the structure we will be using. I have chosen to use a class instead of a simple structure, so that all functions and members of mips_state_t are by defect private. This impedes the user from manually varying them and using them in non-permitted ways. The functions that are allowed to change these values (mips_create, mips_reset...) are declared as *friends* so that they will have access to them.



```
class mips_state_t{

    // everything is private, mips_state_t can only be changed by friend functions

    uint32_t pc; // Address of current instruction
    uint32_t npc; // Address of the next instruction to execute
    unsigned sizeMem; // size of memory
    uint8_t *pMem; // pointer to memory
    uint32_t *registers; // pointer to the 32-bit registers
    uint32_t *hilo; // pointer to the registers HI and LO

    int mips_r_type (); // list of r instructions
    int mips_j_type (); // list of j instructions
    int mips_i_type (); // list of i instructions

    uint8_t get_s(); // 0000 00ss sss0 0000 0000 0000 0000 0000; returns source 1
    uint8_t get_t(); // 0000 0000 000t tttt 0000 0000 0000 0000; returns source 2
    uint8_t get_dest(); // 0000 0000 0000 0000 dddd d000 0000 0000; returns destination
    uint16_t get_immediate(); // 0000 0000 0000 0000 iiii iiii iiii iiii; returns immediate

    void advance_pc (int32_t offset); // advances the pc a signed offset

    // Declared as friends so that they can change mips_state_t's fields
    friend int mips_step(struct mips_state_t *state);
    friend void mips_reset(struct mips_state_t *state, uint32_t pc);
    friend struct mips_state_t *mips_create(uint32_t pc, unsigned sizeMem, uint8_t *pMem);
    friend uint32_t mips_get_register(struct mips_state_t *state, unsigned index);
    friend void mips_set_register(struct mips_state_t *state, unsigned index, uint32_t value);
    friend void mips_free(struct mips_state_t *state);
};
```

When creating an instance of the class, I allocate all memory dinamically so that it isn't destroyed when its scope ends. This is done for the object itself but also for the 32 registers and registers HI and LO (where the results of multiplication and division are stored). The memory is then deallocated in mips_free(). Two program counters are used, one for the instruction that is being currently executed (pc) and one for the next one to execute (npc). This is used for branches and jumps, where we execute the next instruction before actually jumping to the address.

In mips_step is where everything is executed. The first thing to check before we execute anything is if the pc is out of memory bounds, and return an exception if it is. From the byte-addressable memory we then get the opcode, with which we can classify instructions into three different types: R-type, I-type and J-type. For easier readability, I kept the code in mips_step to a minimum, calling instead one of 3 functions inside it (there's one for each type of instruction). This keeps them separated and makes it easier to debug. Mips_step then returns whatever these functions return (0 if no error, non-zero if error).

The three functions we have mentioned are all member functions of the class mips_state_t, so the user doesn't have access to them. Inside each of these instructions, the first step to be taken is to get the information that is most used from memory- in the case of R-type instructions, that would be source1, source2 and a destination. For further readability, these are read in separate member functions of mips_state_t (get_s, get_t, get_dest and get_immediate).

Inside these functions there are a list of if's, one for each instruction. Inside of these different code is executed, and the pc is advanced in each, as not all functions advance it in the same way (for example, jumps and branches). Because of this, a function called advance_pc has been defined, which advances the pc a signed offset. The offset is signed mainly for branches, which can advance the pc forwards (a positive offset) or backwards (a negative offset).

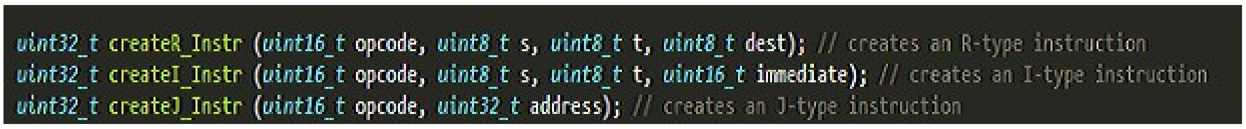
After pc has been advanced inside the if, I return 0 if there has been no error. This way, there is no need to check all ifs if the correct one has already been found, and ensures that if we have reached the end of the function, no if has been executed. If this happens, the instruction hasn't been recognized, and so an error should be returned. Before returning, the pc is advanced to the next instruction so that we are not permanently stuck in the same undefined instruction.

All variables are by default unsigned, since there are equivalent functions that work with the signed or the unsigned values. The immediate is used as a signed value by default since it is the only value that is always used in its signed form. This makes a difference, unlike with other variables, because the offset is a 16 bit variable. If signed, we would have to sign extend when dealing with the 32 bit registers. If unsigned, no sign extension would occur.

The resulting code passes all five tests provided, so additional tests were made to test all remaining instructions. All MIPS-I instructions specified in the assignment have been implemented, and an amount of them are tested in the testing files included.

To test the remaining instructions, I used the same template as the previous tests. The instructions to execute are saved in a binary file, and then run through in a tx_driver.c file. The results from the simulation are compared with the correct results, and if they don't match then an error message is printed. The code is run a number of times with random numbers, to try as many possibilities as possible.

The binary files are written by binfile.cpp. In this file, arrays of instructions are created and then written to their respective binary files. The hexadecimal instructions aren't created manually, but using functions that take as arguments data such as *source1*, *source2*...



```
uint32_t createR_Instr (uint16_t opcode, uint8_t s, uint8_t t, uint8_t dest); // creates an R-type instruction
uint32_t createI_Instr (uint16_t opcode, uint8_t s, uint8_t t, uint16_t immediate); // creates an I-type instruction
uint32_t createJ_Instr (uint16_t opcode, uint32_t address); // creates an J-type instruction
```

This was decided upon after realising that a lot of time was being spent creating the hexadecimal for the instruction instead of actually testing the instruction. Manually doing the hexadecimal works fine when doing small tests, but not if you want to test a great range of instructions, and change them easily if it is required.

An early form of testing was this one of manually creating the opcodes as well as manually saving them into memory, just running one instruction at a time. An array of 8 bit unsigned ints was created, and the hexadecimal stored into memory. The registers were then manually set, and mips_step was called, and the result printed.

While all instructions implemented have been tested using this method, this is not as effective as the first method mentioned. On the other hand, it is easier and faster to implement and to check, unlike using tx_driver.c, the second one taking a much longer amount of time (since the test has to go through debugging as well).

For this reason, not all instructions have been tested using tx_driver, and so may not hold against all cases if tried extensively. Here follows a list of instructions that have only been manually tested:

BLTZ
BGTZ
BGEZAL
BGTZ
J
JAL
LWR

All other instructions have been tested using tx_driver files.

As a final note, the explanation for LWL and LWR is not easily done in comments in the code, so they are then explained below. All other instructions implemented can be fairly easily explained and are explained in comments.

LWL

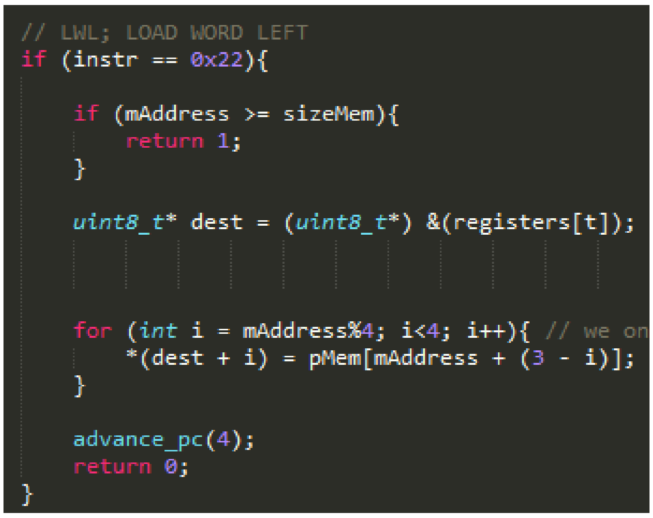
The amount of bytes that are loaded depends on the address that is given to us. If the byte the address refers to is in the middle of the word, at the beginning or at the end, the result is different. To know the address relative to the word, we just need to use %4 (since a word has 4 bytes).

We also need to make the register we're loading into byte addressable. This can be done by casting. The address of the register has type uint32_t*, but we can make it byte addressable by casting it into uint8_t*. It is now byte addressable, but this address is little-endian by default (in contrast to our memory, which is big endian). This means the r_ptr[0] is the rightmost byte of the register.

When doing LWL, there are 4 cases possible:

```
case address%4 =0:
r_ptr[0] = mem[address ]
r_ptr[1] = mem[address +1]
r_ptr[2] = mem[address +2]
r_ptr[3] = mem[address +3]

case address%4 =1:
r_ptr[1] = mem[address ]
r_ptr[2] = mem[address +1 ]
r_ptr[3] = mem[address +2]

case address%4 =2:
r_ptr[2] = mem[address ]
r_ptr[3] = mem[address +1]

case address%4 =3:
r_ptr[3] = mem[address]
```

This can be simplified into a for loop that goes from address%4 to 4 (4 excluded), which is the following:



```
// LWL; LOAD WORD LEFT
if (instr == 0x22){

    if (mAddress >= sizeMem){
        return 1;
    }

    uint8_t* dest = (uint8_t*) &(registers[t]);


    for (int i = mAddress%4; i<4; i++){ // we on
        *(dest + i) = pMem[mAddress + (3 - i)];
    }

    advance_pc(4);
    return 0;
}
```

A similar reasoning was followed for LWR.

LWR

LWR follows the same logic as above, but the possible cases are different:

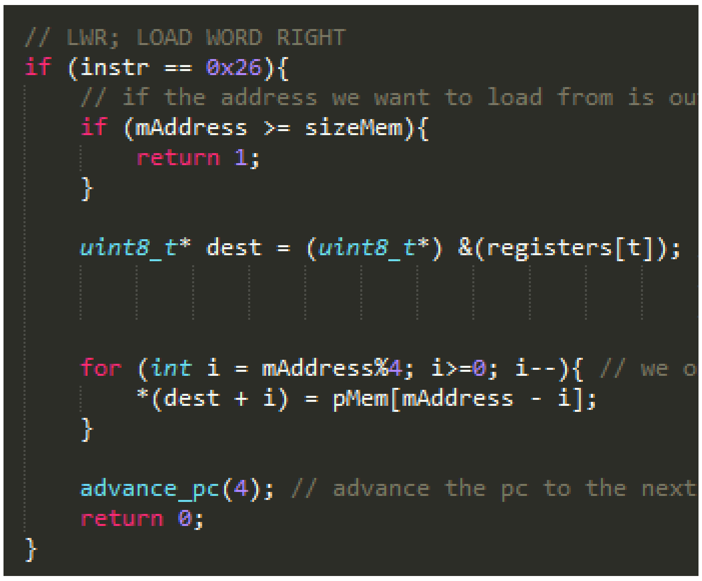
case address%4 =0:
r_ptr[0] = mem[address]

case address%4 =1:
r_ptr[0] = mem[address]
r_ptr[1] = mem[address -1]

case address%4 =2:
r_ptr[0] = mem[address]
r_ptr[1] = mem[address – 1]
r_ptr[2] = mem[address -2]

case address%4 =3:
r_ptr[0] = mem[address]
r_ptr[1] = mem[address – 1]
r_ptr[2] = mem[address -2]
r_ptr[3] = mem[address -3]

This can be simplified into a for loop that goes from 0 to address%4, which is the following:



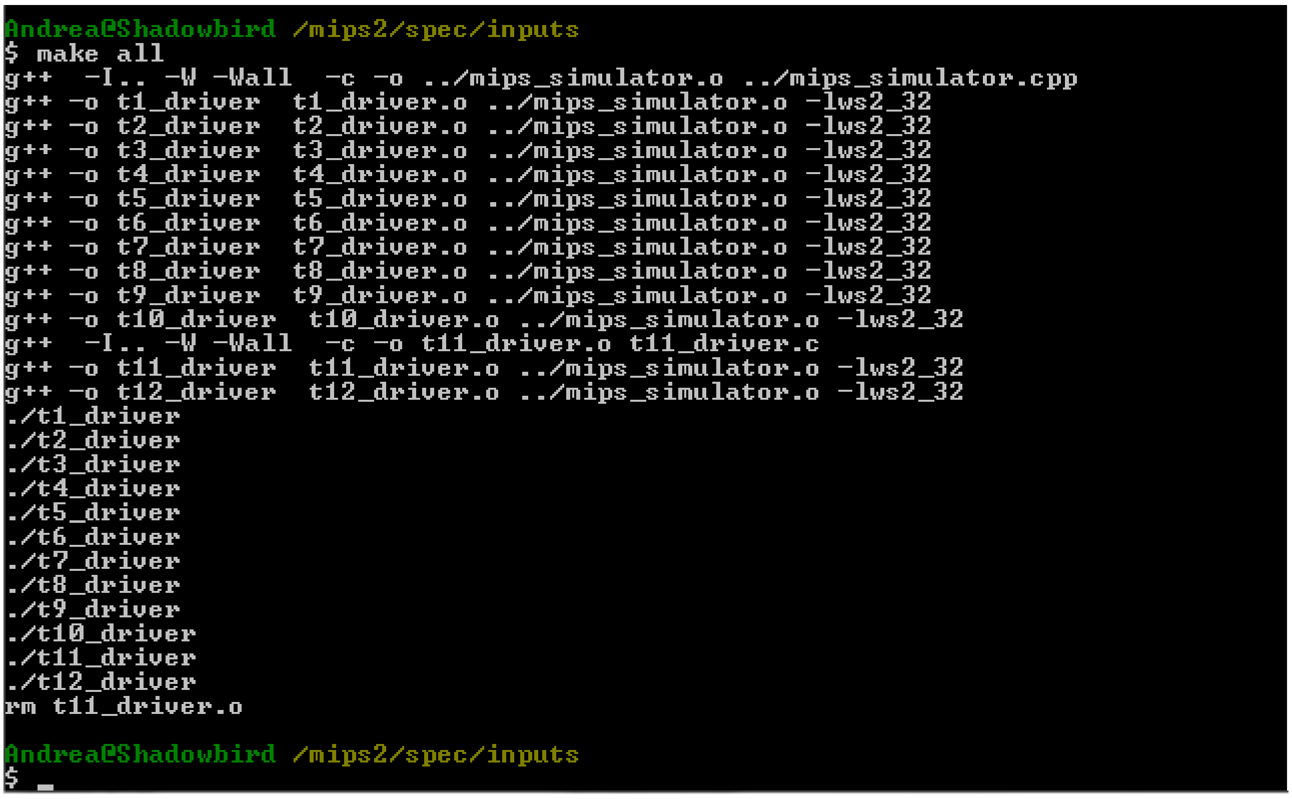
```
// LWR; LOAD WORD RIGHT
if (instr == 0x26){
    // if the address we want to load from is ou
    if (mAddress >= sizeMem){
        return 1;
    }

    uint8_t* dest = (uint8_t*) &(registers[t]);


    for (int i = mAddress%4; i>=0; i--){ // we o
        *(dest + i) = pMem[mAddress - i];
    }

    advance_pc(4); // advance the pc to the next
    return 0;
}
```

The explanations for both LWR and LWL were taken from:

http://math-atlas.sourceforge.net/devel/assembly/mips-iv.pdf

As well as the detailed descriptions of all instructions. The aforementioned pdf is the MIPS-VI complete instruction set, so it includes many instructions that we do not require, but still details those of MIPS-I.

All code was compiled in Windows using Cygwin.



```
Andrea@Shadowbird /mips2/spec/inputs
$ make all
g++  -I.. -W -Wall  -c -o ../mips_simulator.o ../mips_simulator.cpp
g++ -o t1_driver  t1_driver.o ../mips_simulator.o -lws2_32
g++ -o t2_driver  t2_driver.o ../mips_simulator.o -lws2_32
g++ -o t3_driver  t3_driver.o ../mips_simulator.o -lws2_32
g++ -o t4_driver  t4_driver.o ../mips_simulator.o -lws2_32
g++ -o t5_driver  t5_driver.o ../mips_simulator.o -lws2_32
g++ -o t6_driver  t6_driver.o ../mips_simulator.o -lws2_32
g++ -o t7_driver  t7_driver.o ../mips_simulator.o -lws2_32
g++ -o t8_driver  t8_driver.o ../mips_simulator.o -lws2_32
g++ -o t9_driver  t9_driver.o ../mips_simulator.o -lws2_32
g++ -o t10_driver  t10_driver.o ../mips_simulator.o -lws2_32
g++  -I.. -W -Wall  -c -o t11_driver.o t11_driver.c
g++ -o t11_driver  t11_driver.o ../mips_simulator.o -lws2_32
g++ -o t12_driver  t12_driver.o ../mips_simulator.o -lws2_32
./t1_driver
./t2_driver
./t3_driver
./t4_driver
./t5_driver
./t6_driver
./t7_driver
./t8_driver
./t9_driver
./t10_driver
./t11_driver
./t12_driver
rm t11_driver.o

Andrea@Shadowbird /mips2/spec/inputs
$ _
```