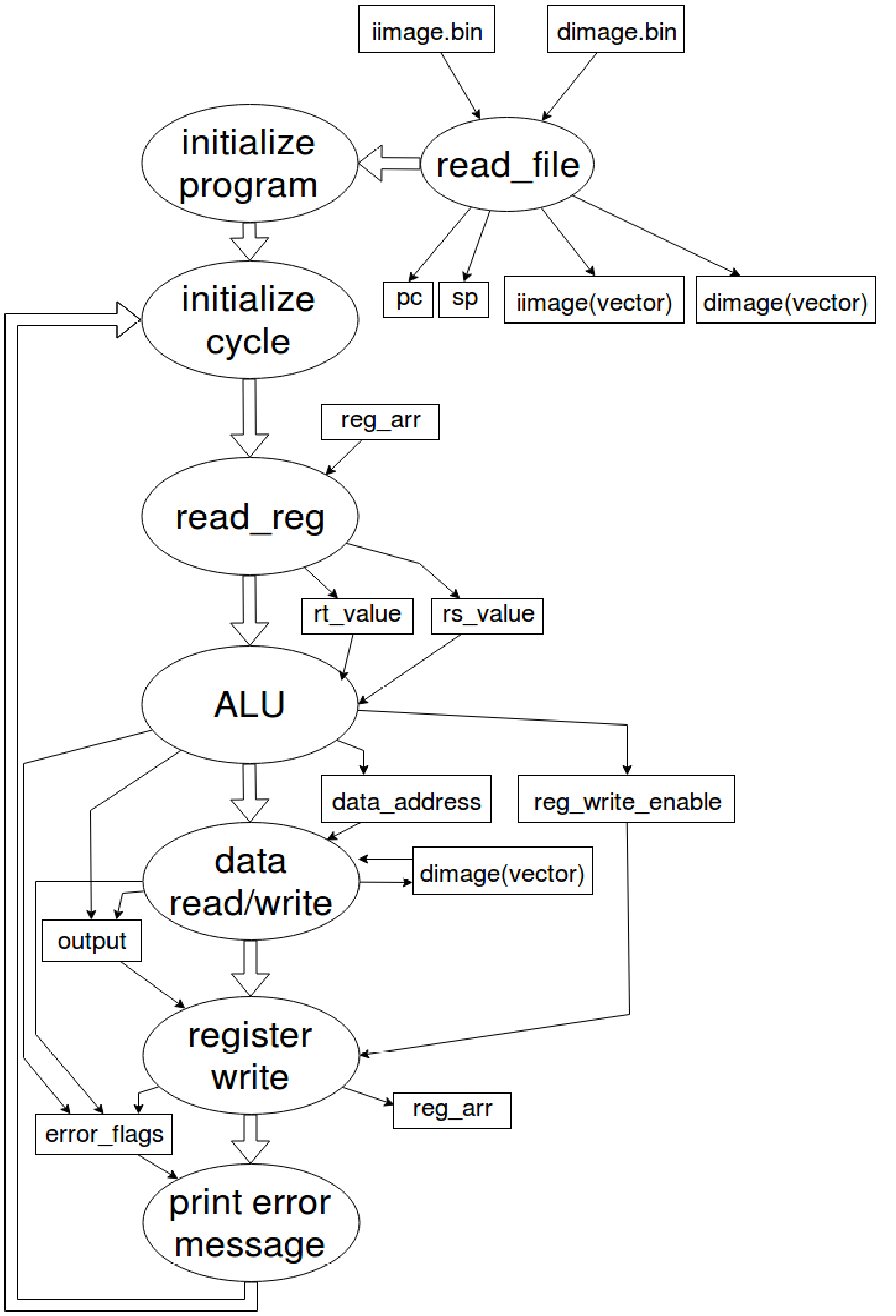
Computer Architecture Project 1 Report

104062321 劉樂永

一、流程圖



流程圖中，粗箭頭代表程式執行的順序，細箭頭代表 input/output。以下會詳細說明其中流程。

二、細節說明

程式架構是仿照課本上的實際線路圖，分成不同 module 來實做。

為了讓 c++ function 表現出類似 verilog module 的樣子，實際的作法是將所有 function 都設定成無回傳值（void），而讀入的 value 就作為 input，讀入的 reference 就作為 output，符合 input 無法修改，output 可以修改的特性。如此就可以讓程式很接近實際的硬體線路，比較容易增減功能和除錯。

但是 c++和 verilog 還是有最明顯的差別，就是程式碼的執行順序。因此也運用此特性，省略掉一些硬體上的麻煩。例如兩個 function 可以先後寫入同一個變數，而不用像兩個硬體 module 就必須有兩個不同 output。

以下分別說明每一個 function 或程式區段。

1. read_file

Read two binary files and store them in vectors. Because data memory can be loaded and stored by byte, I use vector<unsigned char> to store it, and store instructions by vector<uint32_t>.

All data here is in original binary form. Both vector will be "push_back" to their max size in spec (1024 and 256) and set all elements to 0.

2. initialize program (not a function)

In the beginning of program, create a vector(reg_arr) to store all registers, initialize them to zero, then use information from "read_file" function to set original PC and SP value.

Here I set a flag "HI_LO_w_enable" to "true", which means HI and LO are writable, then mult/multu will not trigger "overwrite HI-LO" error.

I open two .rpt files here by std::ofstream. And I use a stringstream, storing output to snapshot.rpt for convenience, but it is unnecessary in this project.

Then, there is a while(1) loop, which repeats executing instructions until a halt instruction or an error to halt.

3. initialize cycle (not a function)

In the beginning of every cycle, set all 5 error flags to false, and use PC as index to load instruction (binary) from I_memory vector.

4. inst_decoder

It decodes instruction from binary to a class. This class describes a instruction by its members: name, opcode, rs, rt, rd, C(shamt, immediate, address), and funct. And rs, rt, rd has another class describes their name($s2) and index(18) in vector.

This function is a member function of the "Instruction class".

This function only use simple "switch" statement.

After this function, if a halt instruction is decoded, the while loop will be break and end the program (after close two std::ofstream).

5. read_reg

It is a member function of class "Instruction".

It loads the value in rt, rs from reg_arr vector, no matter what the instruction is. Even if this instruction do not need to use the value in rt/rs, it will still load it, because it unnecessary to check the kind of instruction in this function. It is ALU's job. (or the control signal unit in hardware)

6. ALU

It is a member function of class "Instruction".

It checks the name of instruction and do the corresponding calculate. The result is the "to_write" output, which means the value to overwrite to somewhere (data memory or registers); and the "data_addr" output, which means the data address should be check in next function.

It modify a "w_enable" flag, a "HI_LO_w_enable" flag and two error flags.

"w_enable" is used to control whether or not to overwrite the register, it is decided by the name of instruction.

"HI_LO_w_enable" is to detect overwrite HI_LO error. If the instruction is mult or multu, it will be set to false, if mfhi or mflo, it will be set to true.

The first error flag is number overflow, which will be detected while calculating by checking the first bit of the number to calculate (&0x80000000).

The second error flag is overwrite HI-LO registers. If the "HI_LO_w_enable" flag is false and the instruction is mult or multu, it will be set to true.

In this function, PC is also modified. It is different from the real block diagram but it is easier to implement, because in software, every value can be modified directly, including PC. After ALU calculate bgtz, beq, jal, etc., it is a good place to set next PC directly, and it doesn't disturb any other functions so it wouldn't make the architecture of program more complex.

7. data_rw

It is a member function of class "Instruction".

It read or write the data memory according to the name of instruction.

If the instruction needs to read or write the memory, this function will use the "data_addr" passed from ALU.

To write, it stores the data in "to_write", which is output of ALU, into the "data_addr" with correct size (word, half word, or byte). To read, it loads data from "data_addr" (also with right size) and overwrite the "to_write". This part can not be done by hardware. In hardware, I should make a new output, assigned by the output of ALU or the data loaded from memory, depends on the instruction. But it is a simulator, so I can avoid this by the characteristic of software program, only remembering the condition of real hardware.

If any part of data address to read/write is out of the range of memory, it sets the error flag of "memory address overflow". If the data address modulo the length of unit (4 for word, 2 for half word) is not 0 (e.g. %4!=0), it sets the error flag of "data misaligned".

8. write_reg

It is a member function of class "Instruction".

It writes register with these cases, and print to snapshot.rpt if there is difference.

Case1: mult/multu

Write HI and LO register directly.

Case2: w_enable==false

It means this instruction doesn't need to write register, so do nothing.

Case3: w_enable==true

First, get the correct register number to write. It can be done by only using if/else to check the type (R type, I type or jal). (A big problem in hardware but so simple in software.)

Second, check whether the register to write is $zero register. If it is and the instruction is not NOP, set a error flag to true. Then return this function.

Last, overwrite the value in reg_arr.

Because ALU has determined whether this instruction needs to write register, the write_reg function only needs to check the flag passed from ALU. It reduces lots of code in program and lots of cost in hardware.

9. print error message (not a function)

There are 5 error flags will be modified in the three functions above. In the end of a cycle, these flags will be check in order to gurantee they will be output correctly. Then the cycle will repeat from 3 if there is no error which needs to halt.