ECE 510 Fall 2020

Project 2: MIPS Simulator

Nick Name: Maxy **Date:** 2020/12/06
First Name: Maoqin **Last Name:** Zhu

1. Problem Statement

This assignment requires a simple 5 stage pipelined machine to be simulated. The simulator should be capable of implementing the MIPS architecture on a cycle by cycle basis. The simulator must be cycle accurate with respect to contents of the registers, but need not be faithful to other hardware details such as control signals. The output of the simulator, in addition to the register contents and latch values should include the utilization factor of each functional unit and the total time in cycles to execute a set of instructions. Implement the simulator according to the specifications described below in C++.

1.1 Instruction to be implemented

The simulator should implement the following instructions: add, sub, addi, mul, lw, sw, beq, lui, and, andi, or, ori, sll, srl, slti, and sltiu. Note that these instructions operate integer instructions only. The MIPS instruction format can be used for all instructions except mul. Assume the syntax for mul is mul $a,$b,$c, meaning that we multiply the contents of $b and $c, the least significant 32 bits of results are placed in register $a and the most significant 32-bits of the result will be stored in register $(a+1). For example, mul $t0, $t8, $t9 will store lower 32-bits of the product of $t8 * $t9 in register $t0 and the upper 32-bits of the product in register $t1. This is different from the mult instruction in MIPS. Assume the opcode and function code for mul to be same as that of mult.

1.2 Inputs to the simulator

1) MIPS machine code as a text file: Convert the assembly level instructions to machine level.
2) A query to the user to select between instruction or cycle mode

• Instruction mode: To observe execution of the program instruction by instruction
• Cycle mode: To observe execution of the program cycle by cycle
3) A query to the user to select the number of instructions or cycles (depending on the choice made in the previous query) to be executed.
4) After executing the number of instructions or cycles entered initially by the user, a third query to the user to choose to continue execution or not.
• If yes, Repeat from step 3
• If no, exit the execution and display the results

1.3 Memory, Registers and PC

The memory is one word wide and 2K bytes in size. There are physically separate instruction and data memories for the instruction and data. Data memory is initialized to 0 at the beginning of each simulation run. There is no cache in this machine.
There are 32 registers; register 0 is hardwired to 0. In addition, there is a Program Counter (PC). PC should start execution by fetching the instruction stored in the location to which it is initialized.

1.4 CPU

The pipelined MIPS processor has 5 stages: IF, ID, EX, MEM, WB. There are pipeline registers between the stages: IF/ID, ID/EX, EX/MEM, MEM/WB. Assume the pipeline registers to contain following latches:
• IF/ID : IR, NPC
• ID/EX : IR, NPC, A, B , Imm
• EX/MEM : IR, B, ALUOutput , cond
• MEM/WB : IR, ALUOutput, LMD

1.5 Output of the simulator

In addition to displaying the register contents and latch values after the execution of each cycle/instruction, it should output the following statistics
• Utilization of each stage. Utilization is the fraction of cycles for which the stage is doing useful work. Just waiting for a structural, control, or data hazard to clear in front of it does not constitute useful work.

• Total time (in CPU cycles) taken to execute the MIPS program on the simulated machine. (This is NOT the time taken to execute the simulation; it is the time taken by the machine being simulated.)

1.6 Dealing with branches

The processor does not implement branch prediction. When the ID stage detects a branch, it asks the IF stage to stop fetching and flushes the IF_ID latch (inserts NOP). When the EX stage resolves the branch, IF is allowed to resume instruction fetch depending on the branch outcome.

1.7 Other remarks

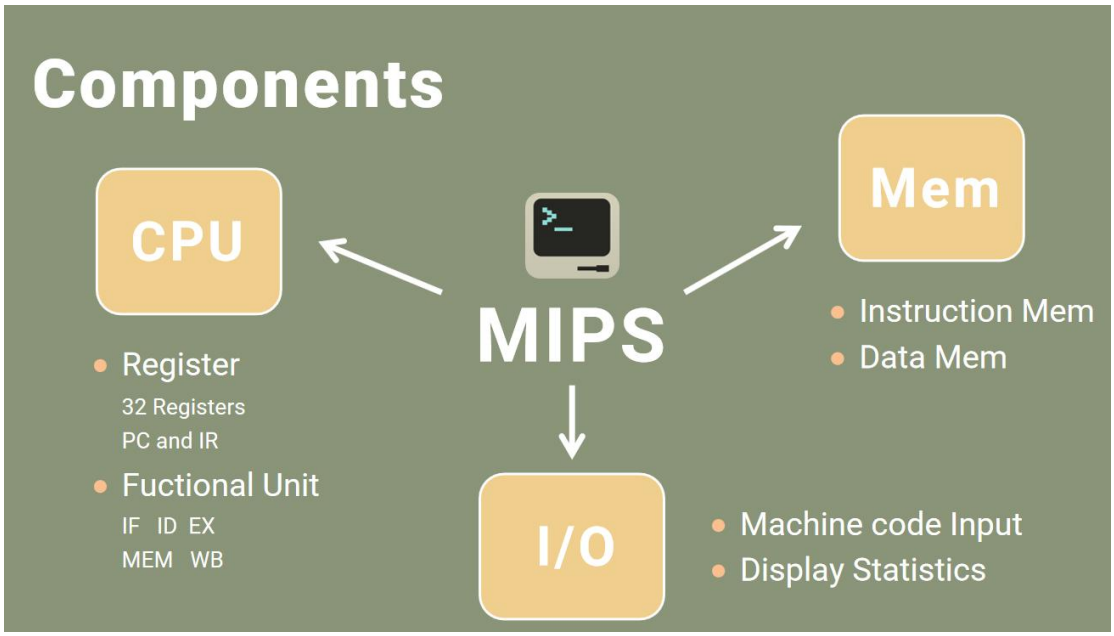
• No interrupts.
• Does not support out of order execution
• Does not support data forwarding
• Assume register writes are completed in the first half of clock cycle and register reads
are carried out in the second half.
• All data, structural and control hazards must be taken into account.
• Branches are resolved in the EX stage.

2. Way to Approach

In this section, I will talk about my approach for solving the problem. The steps I took to find the result will be explained as follow.

2.1 Separate module for MIPS

First of all, I summarize the components of my MIPS machine. It consists of three different module on balance.
The first is CPU module which includes different registers(32 registers, PC, IR, 4 pipeline registers) and 5 function units(IF, ID, EX, MEM, WB). The next is Memory module implementing instruction memory and data memory respectively. Finally, considering the format of input is loading a machine code text file, the last module is I/O module. And of course, all the calculating results and printed outputs belong to this module.



2.2 Create class in header file

Firstly, I create a CPU class in the header file “Cpu.h”. The pseudocode is shown below with some basic declarations. Variables include 8 control signals, latch values and so on. And things like FuctionUnit and hazard_Check are member functions.

Those variables and functions are defined in other cpp files. Please click the code folder to get more coding details.

Cpu Module Header File

```
Class cpu {
public:
    Registers[];   // 32 Regs, IR, PC
    Control unit;   // 8 control signals
    Latch value;   // for 4 pipeline registers
    Other flag signals;   // for each stage

    int FuctionUnit();   // 5 functions for 5 stages
    int hazard_check();   // at ID stage
    void EX_ALU(data1, data2, ALUOp);
}
```

And then I create instruction memory and data memory from the Memory class. The pseudocode is shown below with some basic declarations. Those objects can access all the attributes in class as needed. Please click the code folder to get more coding details.

Memory Module Header File

```
Class Memory{
public:
    int Mem[];    // IMem and DMem
    void loadMem();
    void printMem();
    void intMem();
    void charMem();
}
```

Finally, you can always include the “fstream” header when your program needs to read from a text file.

fstream Header

```
#include <fstream>
char ins[32];
ifstream infile;
infile.open(".../ Instruction/ code.txt");
for (int i = 0; i < Isize; ++i) {
    infile >> ins;
    IMem.charToMem(ins, i);
}
    infile.close();
```

2.3 Implement parallel execution

Here I will introduce the loop I use to implement parallel execution. 6 Functions can be called by cpu object to implement the 5 stage pipeline. What’s important is that the functions must be called in this specific order. Meanwhile, with the

control of some flag signals at each stage, MIPS machine can do parallel execution correctly. Please click the code folder to get more coding details.

Parallel Execution Loop

```
while (current<input) {
        object.WB();
        object.MEM();
        object.EX();
        object.hazard_check;
        object.ID();
        object.IF();
}
```

2.4 Hazard Analysis

Firstly, I'm spelling out the hazard check process. Actually, we can detect branches only by opcode. For example, we transfer the binary to integer, the BEQ instruction opcode is 4. As for data hazard, I declare a regBusy array at first. And check if "rs" and "rt" is busy or not. Please click the code folder to get more coding details.

Hazard Check Process

```
// Control hazard
op <- binary2int(IR,31,26)
if (op=branch) {
      ......          //deal with branch
}
// Data hazard
rs <- binary2int(IR,25,21)
rt <- binary2int(IR,20,16)
if ( regBusy[rs] or regBusy[rt] ) {
      ......
}
```

And then we can declare a variable called NOP at beginning. When the hazard is detected. NOP can be inserted to ask IF and ID function to stop fetching and just wait. Please click the code folder to get more coding details.

NOP Insert

```
hazard_check() {
nop <- 0;
if (hazards) {nop<-1}
}

ID() {
if (!nop) { ..... }
}

IF() {
if (!nop) { ..... }
}
```

2.5 Calculate utilization and total time

To calculate the utilization of each stage and record total CPU cycles, some basic variables could be declared firstly. And then, every time certain function unit is been called, we count at the same time. Please click the code folder to get more coding details.

Utilization and Total Time

```
while (current<input) {
        u5+=object.WB();
        u4+=object.MEM();
        u3+=object.EX();
        object.hazard_check;
        u2+=object.ID();
        u1+=object.IF();
```

```
        Usum++;
}
Utilization = (double) ui / Usum * 100.0;
Total Time = Usum;
```

3. Problems

In this section, I will talk about problems I faced and my solution to them.

3.1 How to implement array operation?

Since several necessary array operations need to be done in this project, but we do not have existing functions or methods to use. Thus how to deal with that? Firstly, I discussed and collaborated with my class mates William and Zijie, we figured out some necessary operations and tried to create a tool box file helping implement related module. By searching on Internet we get to know some basic functions coded in C++. Then picked up what we need and gathered in a header file named Methods. The functions in the header file will be define in cpp file specifically. Please click the code folder to get more coding details.

Methods for Array Operation

```
void addArray(int a[], int b[], int c[], int len);
void subArray(int a[], int b[], int c[], int len);
void andArray(int a[], int b[], int c[], int len);
void orArray(int a[], int b[], int c[], int len);
void hexArray(int a[],int size);
void printArray(int a[], int begin, int end);
void copyArray(int a[], int b[], int begin1, int begin2, int len);
long long array2Int(int a[], int begin, int end);
void shiftL(int a[], int b[], int len, int shamt);
void shiftRL(int a[], int b[], int len, int shamt);
void ALUOpS(int op);
```

4. Analysis

In this section, I'm showing you how my program goes correctly. Some necessary screenshots will be given as follow. And also these calculating result such as utilization and total cycle can be found in other folder. Any question, please reach me out.

Step1: select the number of instruction you wanna load into IMem(34 in this project) and select the number of register you wanna display on screen(16 is enough in this project)

```
Welcome, how many instructions you wanna load into Instruction Memory?(from 1 to 34):
34

IMem:

MemAdress                  Binary                  Hex
0x00000000:     00111100000010010001001000110100  0x3c091234
0x00000004:     00110101001010010101011001111000  0x35295678
0x00000008:     00111100000010100000000000000000  0x3c0a0000
0x0000000c:     00110101010010100000000000000000  0x354a0000
0x00000010:     10101101010010010000000000000000  0xad490000
0x00000014:     00110001001010010000000000000000  0x31290000
0x00000018:     00111100000010011010101111001101  0x3c09abcd
0x0000001c:     00110101001010011110111100010010  0x3529ef12
0x00000020:     10101101010010010000000000000100  0xad490004
0x00000024:     00110001001010010000000000000000  0x31290000
0x00000028:     00111100000010010001000100100010  0x3c091122
0x0000002c:     00110101001010010011001101000100  0x35293344
0x00000030:     10101101010010010000000000001000  0xad490008
0x00000034:     00010001101011100000000000000011  0x11ae0003
0x00000038:     00110101111011111111111111111111  0x35efffff
0x0000003c:     00110101000010001101101010101101  0x3508daad
0x00000040:     00110101000010001110101010101110  0x3508eaae
0x00000044:     10001101010010110000000000000000  0x8d4b0000
0x00000048:     10001101010011000000000000000100  0x8d4c0004
0x0000004c:     10001101010011010000000000001000  0x8d4d0008
0x00000050:     00000001011011000111000000100000  0x016c7020
0x00000054:     10101101010011100000000000001100  0xad4e000c
0x00000058:     00000001100011010100000000100000  0x018d4020
0x0000005c:     00000001000011000111100000100010  0x010c7822
0x00000060:     00000001111010000100100000100000  0x01e84820
0x00000064:     00000000000000000101000000100000  0x00005020
0x00000068:     00000001010010100101100000100000  0x014a5820
0x0000006c:     00110101010010101010101001110110  0x354aaa76
0x00000070:     00000001001010100100100000100101  0x012a4825
0x00000074:     00000001011010110101000000100000  0x016b5020
0x00000078:     00110101010010100000000000000101  0x354a0005
0x0000007c:     00111100000010100000000000001010  0x3c0a000a
0x00000080:     00000001010000000101100000100000  0x01405820
0x00000084:     00000001010010110100000000011000  0x014b4018

Load instructions successfully!

How many registers you wanna display on screen?(from 1 to 32):
16
```

Step2: We should select the mode. Here we choose the first one instruction mode at first. And then select the number of instruction.

What I wanna point out is that there are several error detection in my program,

so as you can see, here if we input a 40, It will go like invalid input. Because we only load 34 instructions at beginning. So we have to input again and just execute 18 instructions this time.

```
Load instructions successfully!

How many registers you wanna display on screen?(from 1 to 32):
16
Which mode? instruction mode(1) or cycle mode(2):
1
Select the number of instructions:
40
Invalid input! You loaded 34 instructions into memory at beginning. Please select again:
18
```

And then as you can see the process stops until instruction 18 writes back successfully.

```
Cycle 38
Instruction 18 current stage: WB        (for Instruction 18, RegWrite in MEM/WB pipeline register is 1) Writing finished! Thus, R11 is available!
Instruction 19 current stage: MEM
Instruction 20 current stage: EX        (R13 is being used!)
Instruction 21 hazard checking...       (rs: R11 & rt: R12) Currently R11 is available and R12 is unavailable!
Data Hazard!

DMem:
MemAdress                   Binary                  Hex
0x00000000:     00010010001101000101011001111000  0x12345678
0x00000004:     10101011110011011110111100010010  0xabcdef12
0x00000008:     00010001001000100011001101000100  0x11223344
0x0000000c:     00000000000000000000000000000000  0x00000000

Registers:
R               Binary                  Hex
00:  00000000000000000000000000000000  0x00000000
01:  00000000000000000000000000000000  0x00000000
02:  00000000000000000000000000000000  0x00000000
03:  00000000000000000000000000000000  0x00000000
04:  00000000000000000000000000000000  0x00000000
05:  00000000000000000000000000000000  0x00000000
06:  00000000000000000000000000000000  0x00000000
07:  00000000000000000000000000000000  0x00000000
08:  00000000000000000000000000000000  0x00000000
09:  00010001001000100011001101000100  0x11223344
10:  00000000000000000000000000000000  0x00000000
11:  00010010001101000101011001111000  0x12345678
12:  00000000000000000000000000000000  0x00000000
13:  00000000000000000000000000000000  0x00000000
14:  00000000000000000000000000000000  0x00000000
15:  00000000000000000000000000000000  0x00000000

PC:             84
IR:             00000001011011000111000000100000
Immediate:      00000000000000000000000000001000
ReadData1:      00000000000000000000000000000000
ReadData2:      00000000000000000000000000000000
ALUResult:      00000000000000000000000000001000
```

But what I wanna point out is here.

As mentioned before, some data hazards can be detected correctly and we just insert NOPs to deal with it. It just stops fetching and waits until the 12th writes back.

What's more, since the 14th instruction is BEQ, so you can see my program detects control hazard correctly too. And it goes from 14th instruction straightly to 18th instruction.

```
Cycle 30
Instruction 12 current stage: EX        (R9 is being used!)
Instruction 13 hazard checking...       (rs: R10 & rt: R9) Currently R10 is available and R9 is unavailable!
Data Hazard!

Cycle 31
Instruction 12 current stage: MEM
Instruction 13 hazard checking...       (rs: R10 & rt: R9) Currently R10 is available and R9 is unavailable!
Data Hazard!

Cycle 32
Instruction 12 current stage: WB        (for Instruction 12, RegWrite in MEM/WB pipeline register is 1) Writing finished! Thus, R9 is available!
Instruction 13 hazard checking...       (rs: R10 & rt: R9) Currently R10 is available and R9 is available!
No hazards!
Instruction 13 current stage: ID
Instruction 14 current stage: IF
```

```
Cycle 33
Instruction 13 current stage: EX
Instruction 14 hazard checking...       (rs: R13 & rt: R14) Currently R13 is available and R14 is available!
Control Hazard!
Instruction 14 current stage: ID

Cycle 34
Instruction 13 current stage: MEM
Instruction 14 current stage: EX
Instruction 14 hazard checking...       (rs: R13 & rt: R14) Currently R13 is available and R14 is available!
No hazards!
Instruction 18 current stage: IF

Cycle 35
Instruction 13 current stage: WB        (for Instruction 13, RegWrite in MEM/WB pipeline register is 0)
Instruction 14 current stage: MEM
Instruction 18 hazard checking...       (rs: R10 & rt: R11) Currently R10 is available and R11 is available!
No hazards!
Instruction 18 current stage: ID
Instruction 19 current stage: IF
```

Step3: What's my program different is here. Since this process is not been canceled yet. Now we have finished 18 instructions in total. If we continue this process and select 16, we will execute all the 34 instructions.

```
Control Signals:
PCSrc:          0
RegWrite:       1
ALUSrc:         1
ALUOp:          lw
RegDst:         0
MemWrite:       0
MemRead:        1
MemToReg:       1

Excellent! Instructions have been executed successfully!

CPU has executed 18 instruction(s) in total. Continue this process or not?(y/n)
y
Select the number of instructions:
16
```

```
Cycle 73
Instruction 34 current stage: WB       (for Instruction 34, RegWrite in MEM/WB pipeline register is 1) Writing finished! Thus, R8 is available!
No hazards!

DMem:
MemAdress                  Binary                    Hex
0x00000000:     00010010001101000101011001111000  0x12345678
0x00000004:     10101011110011011110111100010010  0xabcdef12
0x00000008:     00010001001000100011001101000100  0x11223344
0x0000000c:     10111110000000100100010110001010  0xbe02458a

Registers:
R               Binary                  Hex
00:  00000000000000000000000000000000  0x00000000
01:  00000000000000000000000000000000  0x00000000
02:  00000000000000000000000000000000  0x00000000
03:  00000000000000000000000000000000  0x00000000
04:  00000000000000000000000000000000  0x00000000
05:  00000000000000000000000000000000  0x00000000
06:  00000000000000000000000000000000  0x00000000
07:  00000000000000000000000000000000  0x00000000
08:  00000000000000000000000000000000  0x00000000
09:  00000000000000000000000001100100  0x00000064
10:  00000000000010100000000000000000  0x000a0000
11:  00000000000010100000000000000000  0x000a0000
12:  10101011110011011110111100010010  0xabcdef12
13:  00010001001000100011001101000100  0x11223344
14:  10111110000000100100010110001010  0xbe02458a
15:  00010001001000100011001101000100  0x11223344

PC:             140
IR:             00000000000000000000000000000000
Immediate:      00000000000000000000010000000011000
ReadData1:      00000000000010100000000000000000
ReadData2:      00000000000010100000000000000000
ALUResult:      00000000000000000000000000000000

Control Signals:
PCSrc:          0
RegWrite:       1
ALUSrc:         0
ALUOp:          mul
RegDst:         1
MemWrite:       0
MemRead:        0
MemToReg:       0
```

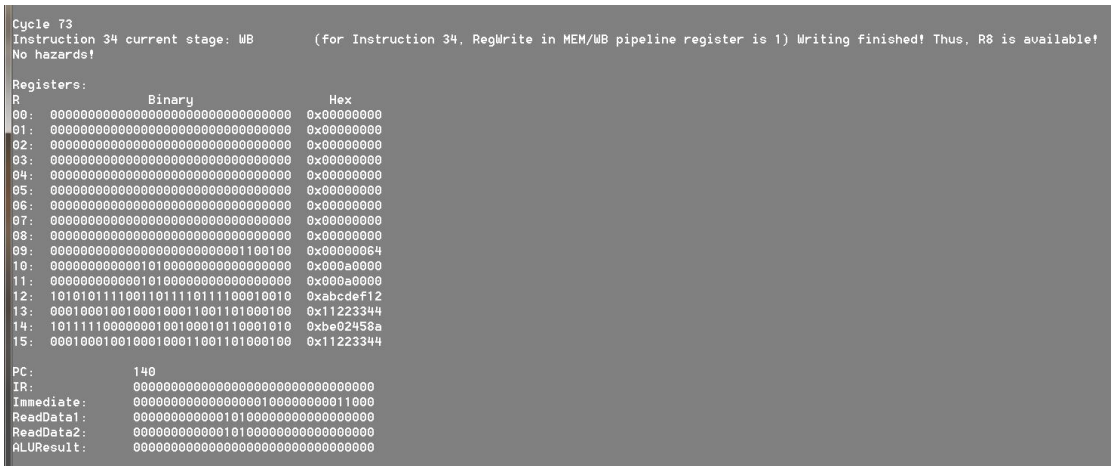
As you can see, the 34th instruction writes back correctly. If we can exit, the calculating result output will be printed on screen as follow.

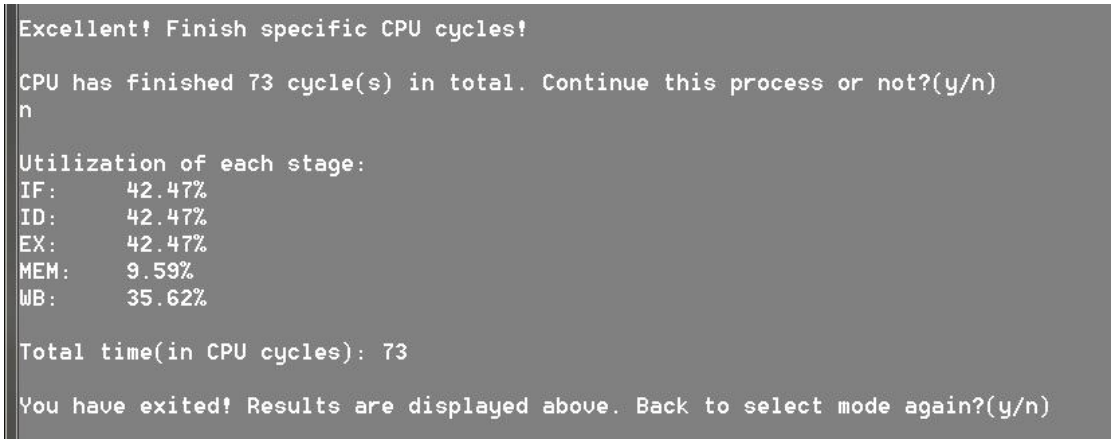
```
CPU has executed 34 instruction(s) in total. Continue this process or not?(y/n)
n

Utilization of each stage:
IF:      42.47%
ID:      42.47%
EX:      42.47%
MEM:     9.59%
WB:      35.62%

Total time(in CPU cycles): 73
```

Step4: What's more, we can still back to select the mode and start a new process again. We select mode 2 this time and give a 73.

```
You have exited! Results are displayed above. Back to select mode again?(y/n)
y
Which mode? instruction mode(1) or cycle mode(2):
2
Select the number of cycles:
73
```

As you can see, the program stops at Cycle 73 accurately. Also the calculating results like utilization and total time could be printed on screen if we exit.





5. Conclusion

This project assignment actually gives me an experience in C++ by designing a MIPS simulator.

Firstly, I'm more familiar with the architecture of computer. With some basic concepts that CPU, memory, I/O, I gradually could know how computer works and even how it works more efficiently.

And then, when I face some confusing points, thanks to help from my classmates and friends. Collaboration and discussion are significant whatever in study life or at work place. I think learning from others is something super helpful.

What's more, after finishing this project, I find that I'm really more familiar with C++ coding language. I would say maybe I'm still using it in the rest of my graduate education.

On balance, by using what I learned in live lecture, this C++ project is a great project experience for me, which helps me know what to learn next.