EEC 483/581 Computer Architecture, Spring 2021
Project 3 & 4 – MIPS Simulator

1. Objectives

This project is designed to help students to understand the pipelining on a RISC architecture (MIPS), pipeline hazards, and hazards mitigation techniques including forwarding and stalling. Students will gain programming experience in C.

2. Goals

Your team (2 persons) will build a simulator of a 5-stage pipelined MIPS in C. This simulator will read a binary executable file containing a sequence of MIPS instructions and simulate their execution in stages in a pipelined MIPS processor.

Project 4 is the extension of Project 3 by including branch hazard. Note that the shaded parts in this document are Project 4 requirements. Due dates for Project 3 and 4 is **April 9** and **April 30**, respectively.

3. Specification

3.1 Input

Your simulator will read a binary executable file such as the Project 1/2 output. It is of size 1KB and consists of 512B of data segment followed by 512B of text segment (offset 0x200). Do not try to open this file as it causes an error. Assume big-endian. Assume input files do not contain any errors.

3.1.1 Registers

The user can use any of the 32 MIPS registers (0-31). Please note that $0 is read-only and should always contain the value 0. Your simulator will need to include a register file (e.g., an array of 32 integers) that maintains the values of all the 32 registers and updates the registers as the instructions are executed.

3.1.2 Data segment

The data segment is of 512B and the only form of data will be integer numbers (4 bytes long). I.e., there can be at maximum 128 integer data in the data segment. The data will not contain any other form of data such as string (ascii characters) or floating point numbers. When the program sees load or store instructions, the address will resolve to access one of these 128 integer data. Therefore, as execution proceeds, loads (lw) can read from the data segment and stores (sw) can write to the data segment.

3.1.3 Text segment

The text segment is of 512B, i.e., there can be at maximum 128 MIPS instructions because each MIPS instruction is 4 bytes long. However, due to the presence of branches and jumps, some instructions will be executed multiple times. The text segment will only contain the following MIPS instructions.

Instruc -tion	Syntax	MIPS instruction encoding (32 bits)					
	(R-type)	Opcode(6)	Rs(5)	Rt(5)	Rd(5)	Shamt(5)	Func(6)
ADD	add $1,$2,$3	000000				N/A	100000 (=32_{10})
SUB	sub $1,$2,$3	000000				N/A	100010 (=34_{10})
SLL	sll $1,$2,5	000000					000000 (=0)
SRL	srl $1,$2,5	000000					000010 (=2)
SLT	slt $1,$2,$3	000000				N/A	101010 (=42_{10})
HALT	halt	000000	00000	00000	00000	00000	001100 (=12_{10})
	(I-type)	Opcode(6)	Rs(5)	Rt(5)	Immediate (16)		
ADDI	addi $1,$2,45	001000					
LW	lw $1,100($2)	100011					
SW	sw $1,100($2)	101011					
BEQ	beq $1,$2,Label	000100					
BNE	bne $1,$2,Label	000101					

- The 'halt' instruction indicates the end of the program but note that it is not an actual MIPS instruction.
- ORI, LUI, and J are not supported in this project.
- Note that branch target will be contained in the program. In other words, it does not go beyond the 512B boundary. Since the text segment begins at 0x200, this means the targets will be in the range of 0x200 ~ 0x400.

3.1.4 Sample input

Consider an input file as follows (binary view, e.g., od –Ax –t x4):

```
000000 00000000 00000000 00000001 00000002
000010 00000000 00000000 00000000 00000000
*
000200 20010008 8c240000 8c250004 00020940
000210 00020942 0043082a 00000000 00000000
000220 00000000 00000000 00000000 00000000
*
000400
```

Note that this file begins with the data segment (512 bytes) followed by the text segment (512 bytes), i.e., the first instruction is "20010008" which indicates an ADDI instruction. And so on.

3.2 Output

Your simulator should show the followings for every cycle of program execution until it encounters an HALT instruction.

- The value of PC
- Data memory words (the first 16 only out of 128 words)
- Registers ($0 ~ $15 only)
- The values of all pipeline registers (including control signals)

Your simulator should output as follows: (**note that an arrow sign and the instructions on the right is just for your information; you do not have to print this**):

```
PC = 200
DM          0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0
RegFile     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
IF/ID (pc4, inst)                                204 20010008          →addi $1,$0,8 in IF
ID/EX (pc4, rd1, rd2, extend, rt, rd, ctrl)      0 0 0 0 0 0 0
EX/MEM (btgt, zero, ALUOut, rd2, RegRd, ctrl) 0 0 0 0 0 0
MEM/WB (memout, ALUOut, RegRd, ctrl)      0 0 0 0
-------------------------------------------------------------------------------------
PC = 204
DM              0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0
RegFile         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
IF/ID (pc4, inst)                                208 8c240000          →lw $4,0($1) in IF
ID/EX (pc4, rd1, rd2, extend, rt, rd, ctrl)      204 0 0 8 1 0 180     →addi $1,$0,8 in ID
EX/MEM (btgt, zero, ALUOut, rd2, RegRd, ctrl) 0 0 0 0 0 0
MEM/WB (memout, ALUOut, RegRd, ctrl)      0 0 0 0
-------------------------------------------------------------------------------------
PC = 208
DM              0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0
RegFile         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
IF/ID (pc4, inst)                                20c 8c250004          →lw $5,4($1) in IF
ID/EX (pc4, rd1, rd2, extend, rt, rd, ctrl)      208 0 0 0 1 4 2c0     →lw $4,0($1) in ID
EX/MEM (btgt, zero, ALUOut, rd2, RegRd, ctrl) 204 0 8 0 1 180       →addi $1,$0,8 in EX
MEM/WB (memout, ALUOut, RegRd, ctrl)      0 0 0 0
-------------------------------------------------------------------------------------
```

Note that the content of the pipeline registers and control signals (ctrl) are explained in detail later in this document.

3.3 Testing

A few sample test binary files will also be provided about a week before the project is due. Make use of the sample executable to help verify your output.

You can test your design using your own programs. The quality of your simulator will be determined by how much and how varied the tests are. For example, testing if the simulator works for BNE for both the equal and not equal cases, and forward and backward branching, and all combinations of these, would get a better grade than just testing for one case of BNE.

To confirm your assembler generates the correct machine code, you can get some help from, e.g., http://www.kurtm.net/mipsasm/. Just type your assembly program such as “add $1, $2, $3”, it will output machine code for you. It allows comments as well as labels.

4. Suggested Approach

4.1 Pipeline datapth

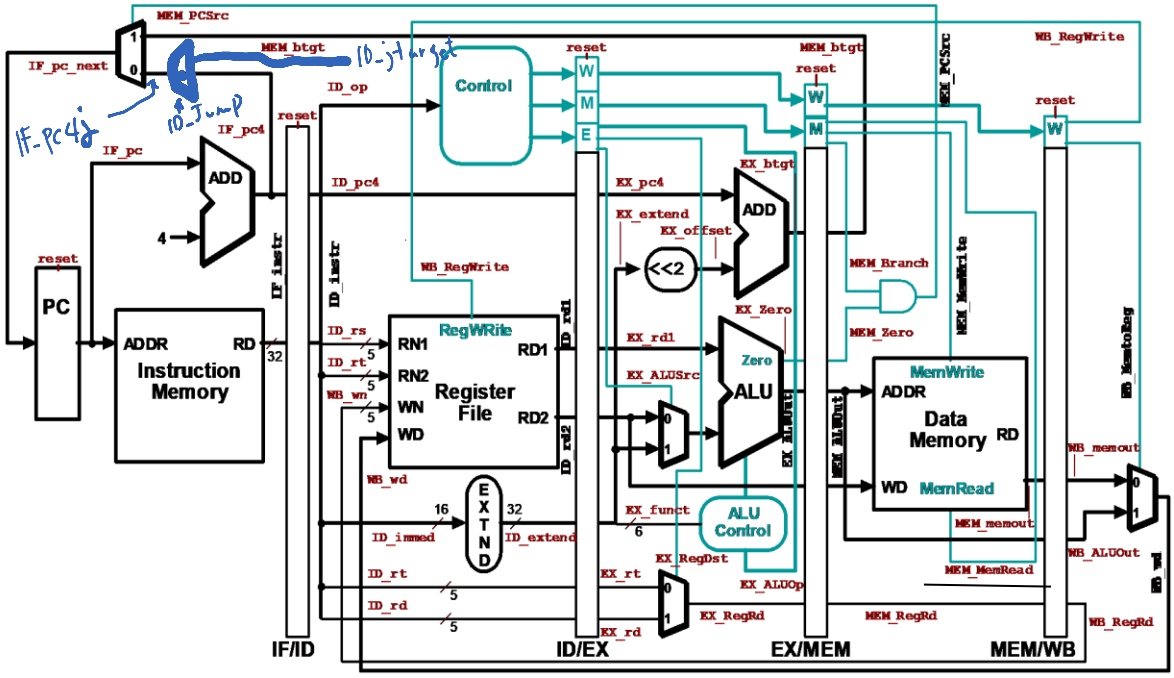
The processor pipeline has the following 5 stages: Instruction Fetch (IF), Register fetch/ instruction decode (ID), ALU/Execute (EX), Memory (MEM), and Write back (WB). You may want name the variables in your simulator as shown in the following figure. For example, “EX_RegRd” and “MEM_RegRd” refer to the destination register number in the EX and the MEM stage, respectively. **Note that you may use an integer variable (32-bit) for most of the signals for your convenience even though they are not 32-bit data.**

Please note the use of labels in the branch (beq and bne) instructions. You will need to calculate the branch target based on the following relationship.

Addr(Label) = Addr(branch_instruction) + 4 + immediate(16 bits)*4

The logic for "BNE" is not included in the figure and can be added easily. I.e., find in the figure that MEM_PCSrc = MEM_Branch & MEM_Zero. It can be extended to incorporate BNE:

MEM_PCSrc = (MEM_Branch && MEM_Zero) || (MEM_BranchNE && !MEM_Zero)



4.2 Pipeline control

Control signals are generated in the ID stage and move to next stages along with the instruction as discussed in class. For convenience, you can move all 11 control signals (ctrl) including BranchNE and Jump.

	RegDst	ALU Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op1	ALU Op0	Branch NE	Jump	
R-format	1	0	1	1	0	0	0	1	0	0	0	0x588 or 1416
lw	0	1	0	1	1	0	0	0	0	0	0	0x2c0 or 704
sw	x	1	x	0	0	1	0	0	0	0	0	0x220 or 544
beq	x	0	x	0	0	0	1	0	1	0	0	0x014 or 20
addi	0	1	1	1	0	0	0	0	0	0	0	0x380 or 896
bne	x	0	x	0	0	0	0	0	1	1	0	0x006 or 6

Note that you may use an integer variable (32-bit) for each of the control signals for your convenience even though they are just a 1-bit data. Whenever necessary, you can extract a particular control signal by using bitwise "shift (>>)" and "and (&)" operations. I.e.,

```
EX_RegDst = (EX_ctrl >> 10) & 1;
EX_ALUSrc = (EX_ctrl >> 9) & 1;
```

```
WB_MemtoReg = (WB_ctrl >> 8) & 1;
```

For example, MemtoReg is bit 8 in the control signal. Given the "lw" instruction, the 11-bit control signal is 01**0** 1100 0000 (0x2c0 0r 704) and bit 8 (MemtoReg) is 0. To obtain bit 8 (MemtoReg), first, shift the 11-bit control data 8 bits to the right. Thus, (**010** 1100 0000 >> 8) = 000 0000 0**010**. Then, this is ANDed with 1, thus, 0000 0000 0010 & 1 = 0. I.e., MemtoReg is obtained by (ctrl>>8) & 1.

4.3 Pipeline registers

Pipeline registers (such as IF/ID and ID/EX in the figure) are to save the computed values necessary for the next stage and each pipeline register includes a number of components. To conform to the naming conventions, it is advised to name the pipeline registers (their components) like IFID_pc4 (a part of IF/ID pipeline register) and IDEX_rd1 (a part of ID/EX pipeline register).

4.4 Simulation program

This is a suggested skeleton of the MIPS simulator.
(I have never tested this pseudo-code. Please use it at your own risk.)

```
main() {
        initialize();
        PC = 512; // equivalent to 0x200
        cycles_to_halt = 1000; //maximum number of cycles to execute

        while (true) {
                read_pipeline_registers();      // beginning of a cycle
                carryout_operations();          // during the cycle
                update_pc_reg_mem();            // end of the cycle (rising edge of the clock)
                                                // make sure PC, registers, and memory are updated in this function only
                update_pipeline_registers();    // end of the cycle (rising edge of the clock)
                                                // make sure pipeline registers are updated in this function only
                print_results();

                // The instruction word "0000 0000 0000 1100" (or 12) indicates HALT.
                // It does not stop immediately because there are four proceeding instructions in the pipeline
                // that have to be completed their execution. Thus, it needs to run 4 more cycles.
                if (IF_inst = 12) cycles_to_halt = 4;
                if (cycles_to_halt > 0) cycles_to_halt --;
                if (cycles_to_halt == 0) break;
        }
}

initialize() {
        // Set up and initialize the register file array (e.g., register[32]) and the memory array (e.g. memory[256]).
        // your code

        // Copy the contents in the input binary file into the memory array.
        // Note that it is 128 data words and 128 instruction words, totaling 256 items in the memory array.
        // The data segment begins at memory[0] while the text segment begins at memory[128].
        // you code

        // initialize all pipeline registers
        // IFID_pc4, IFID_inst
```

```
	// IDEX_pc4, IDEX_rd1, IDEX_rd2, IDEX_extend, IDEX_rt, IDEX_rd, IDEX_ctrl
	// EXMEM_btgt, EXMEM_zero, EXMEM_ALUOut, EXMEM_rd2, EXMEM_RegRd, EXMEM_ctrl
	// MEMWB_memout, MEMWB_ALUOut, MEMWB_RegRd, MEMWB_ctrl
	// your code
}

read_pipeline_registers() {
	// Read pipeline registers to set stage signals which comes directly from the pipeline registers, e.g., takes no
	// time. Note that it is intentional that the pipeline register names and stage signal names are similar. Refer a
	// figure in Section 4.1 of this document for signal names.

	// ID stage – ID_pc4, ID_inst
	// EX stage – EX_pc4, EX_rd1, EX_rd2, EX_extend, EX_rt, EX_rd, EX_ctrl
	// MEM stage – MEM_btgt, MEM_zero, MEM_ALUOut, MEM_rd2, MEM_RegRd, MEM_ctrl
	// WB stage – WB_memout, WB_ALUOut, WB_RegRd, WB_ctrl
	// your code
}

carryout_operations() {
	// Carry out the operations in each stage and produce signals accordingly.

	// IF stage – IF_inst, IF_pc, IF_pc4, and IF_pc_next must be updated
	// your code

	// ID stage – ID_op, ID_rs, ID_rt, ID_rd, ID_immed, ID_extend, ID_rd1, ID_rd2, and ID_ctrl must be
	// updated
	// your code

	// EX stage – EX_offset, EX_btgt, EX_ALUOut, EX_Zero, EX_funct, and EX_RegRd must be updated
		// As an example, EX_ALUOut can be updated as follows.
		// Refer a table in Section 4.2 of this document for the content of "ctrl" signals.
		// Refer a table in Section 3.1.3 of this document for the function code.
		// if (EX_ctrl==1416) {			// R-type
		//	if (EX_funct==32)		EX_ALUOut = ; // add
		//	else if (EX_funct==34)		EX_ALUOut = ; // sub
		//	else if (EX_funct==0)		EX_ALUOut = ; // sll
		//	else if (EX_funct==2)		EX_ALUOut = ; // srl
		//	else if (EX_funct==42)		EX_ALUOut = ; // slt
		//	else if (EX_funct==12)		EX_ALUOut = 0; // halt
		//}
		//else if ((EX_ctrl==20) || (EX_ctrl==6))			EX_ALUOut = 0; //beq or bne
		//else if ((EX_ctrl==704) || (EX_ctrl==544) || (EX_ctrl==896)) EX_ALUOut = ; //lw or sw or addi
		// your code

	// MEM stage – MEM_PCSrc and MEM_memout must be updated
		// when accessing memory, be sure to divide the memory address by four
		// for example, if (MEM_MemRead) MEM_memout = memory[MEM_ALUOut/4];
		// your code

	// WB stage – WB_wd must be updated;
		// your code
}

update_pc_reg_mem() {
	// Update register/memory based on signals as a result of carryout_operations()

	// IF stage – update "PC"
		// your code

	// ID stage – nothing to update
	// EX stage – nothing to update
```

```
        // MEM stage – update memory if appropriate (MEM_MemWrite==1)
                // your code

        // WB stage – update the destination "register" if appropriate (WB_RegWrite==1)
                // your code
}

update_pipeline_registers() {
        // Update the pipeline registers
        // ID stage – ID_pc4, ID_inst
        // EX stage – EX_pc4, EX_rd1, EX_rd2, EX_extend, EX_rt, EX_rd, EX_ctrl
        // MEM stage – MEM_btgt, MEM_zero, MEM_ALUOut, MEM_rd2, MEM_RegRd, MEM_ctrl
        // WB stage – WB_memout, WB_ALUOut, WB_RegRd, WB_ctrl
        // your code
}

print_results() {
        // print the content of PC, data memory, register file and pipeline registers as follows
        // (use hexadecimal for PC and pipeline registers; use decimal for memory and register contents)
        //PC = 208
        //DM                0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0
        //RegFile           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        //IF/ID (pc4, inst)                              20c 8c250004
        //ID/EX (pc4, rd1, rd2, extend, rt, rd, ctrl)    208 0 0 0 1 4 2c0
        //EX/MEM (btgt, zero, ALUOut, rd2, RegRd, ctrl) 204 0 8 0 1 180
        //MEM/WB (memout, ALUOut, RegRd, ctrl)      0 0 0 0
}
```

4.5 Forwarding and Stalling (Not included in this project)

[redacted]

[redacted]

4.6 Branch hazards

As for branch instructions, our implementation should be based on **"predict not-taken" approach**, which assumes the branch would not be taken and continues to accept the subsequent instructions. If the prediction is not correct, the architecture has to eliminate or flush those subsequent instructions from the pipeline.

The architecture gets to know about the outcome of the prediction when the branch instruction is in the MEM stage when the Zero flag is available. Accordingly, MEM_PCSrc=1 and it selects the branch target address (MEM_btgt) as the next PC.

If the prediction was not correct and the branch has to be taken, the three subsequent instructions in IF, ID and EX stage must be eliminated or flushed out. Our approach is to nullify the effect of those instructions by zeroing MemRead, MemWrite, and RegWrite control signals.

Based on the skeleton of the code in Section 4.4 of this document, it can be implemented as follows. Note that zeroing-out all control signals but leaving any data that has been already fetched/computed will have no effect on the CPU state. Likewise, the instruction of decimal value of 0 is a NOP instruction in MIPS, which is actually the shift-left-logical instruction sll $0, $0, 0. Since it shifts 0 bit, this instruction has no effect and thus, no operation or NOP.

```
update_pipeline_registers() {
        // Update the pipeline registers
        // ID stage – ID_pc4, ID_inst
        // EX stage – EX_pc4, EX_rd1, EX_rd2, EX_extend, EX_rt, EX_rt, EX_ctrl
        // MEM stage – MEM_btgt, MEM_zero, MEM_ALUOut, MEM_rd2, MEM_RegRd, MEM_ctrl
        // WB stage – WB_memout, WB_ALUOut, WB_RegRd, WB_ctrl
        // your code

        // This is added to implement the branch hazard
        if (MEM_PCSrc==1) {         // if branch had to be taken
                MEM_ctrl = 0;       // MEM_ctrl is set to zero even if it was set differently earlier
                                    // this mean MemWrite and RegWrite are zero; i.e., no effect
                EX_ctrl = 0;        // EX_ctrl is set to zero even if it was set differently earlier
                                    // this mean MemWrite and RegWrite are zero; i.e., no effect
                ID_inst = 0;        // There is no control signal at the ID stage; thus just nullify instruction itself.
                                    // The instruction word "0" is "sll $0,$0,0" indicating NOP (no operation)
        }
}
```

5. Testing

Part of the process of developing a good software artifact is thoroughly testing the artifact. The sample test cases, while extensive, do not cover all the possible combination of instructions your program could encounter.

For this project, you are required to develop your own suite of test cases. You can write anywhere between 1 (very long) and 10 (somewhat small) test programs, using the reduced instruction set for this project) to ensure the quality of your simulator. The quality of the test suite will be determined by how many of our reduced set of instructions are tested and how varied the tests are. For example, testing if the simulator works for BEQ for both the equal and not equal cases, and forward and backward branching, and all combinations of these, would get a better grade than just testing for one case of BEQ.

6. Submission and Grading

Each group submits one copy of the source code: simulator.c.
Create a folder and name it as your group name, e.g., lastname1_lastname2 (all lower case).
Copy all your source code to the above folder (clean your source code folder and remove all binary files).
Create a README file (in text format only) and list CLEARLY all the instructions and directives your program cannot handle and all known issues with your program (those that are not implemented, those that are implemented but not work correctly, etc.). If you use a suite of test cases that you have generated, beyond the test cases provided, explain it in the README file.

Log in grail, go to the parent director of the folder you created, and run

```
turnin -c eec483y -p proj3 lastname1_lastname2
```

If there is no error message, your submission is successful.
Your most recent submission will always automatically overwrite the previous one.

The grail computer can be reached using ssh protocol through a terminal emulator such as PuTTY (www.putty.org) as either grail.eecs.csuohio.edu, eecs.csuohio.edu, or via IP address 137.148.204.40.

You can verify the status of your projects by typing 'turnin -l' and then type either eec483y or eec581y to see the status of your projects. Status is either off (-), on (alternate) (+) or on (current) (=). You can find additional information about turnin by typing 'turnin -help' at the Linux command prompt.

You must submit before 11:59 PM on April 9 (April 30 for Project 4) to get full credit. Late submissions will be accepted for 10% off per day up to three additional days. Automatic plagiarism detection software will be used on all submissions. Any cases detected will result in a grade of 0, reduction of the overall grade by one letter grade, and a report will be sent to Academic Affairs.

Your submitted C program will be compiled and run on the grail server with the following commands, where test.asm is an assembly file, as described below:

```
$ gcc [-std=c99] simulator.c
$ ./a.out test.out
```

You should not rely on any special compilation flags or other input methods.