CDA 3101: Spring 2020
Project 2 - Pipeline Simulator

Total Points: 100
Due: Friday, 03/13/2020, 11:59:00 PM

1 Objective

The objective for this assignment is to make sure

- You are familiar with the pipelining process on a MIPS processor.
- You gain some experience with the principles of pipelining and hazard mitigation techniques including stalling, forwarding and branch prediction.
- You are comfortable working with C, including I/O, parsing and memory management.

2 Specifications

Fir this project, you will be creating a MIPS 5-stage pipeline simulator. Please write a C program that will read in a small number of machine instructions an simulate their execution, in stages, in a pipelined MIPS processor.

2.1 Input

Your input will be in the form of MIPS machine instructions (similar to the output of the first program). These instructions will formatted as a SIGNED 32 bit integer and redirected into your program through standard input. Please do not try to open the file in your program.

You input file will contain both instructions and data. The instructions and data will be separated by a blank line. You may assume the following:

- Every line of the file (except for the blank line separating instructions and data) is either a 32-bit instruction or a data word.
- The only form of data will be 32-bit numbers (words). The input will not contain any other form of data like formatted free space (.space), strings (.asciiz), etc.
- There are no more than 100 instructions and 32 data words.
- Input files do not contain any errors.
- The user could be using any of the 32 programmer visible MIPS registers.
- The data words are in addresses reachable by the program. The data segment immediately follows the text segment and the addresses on the load and store instructions will resolve correctly.
- Branch and jump targets will be contained in the program.

- Due to the presence of branches and jumps, some lines in the program might execute multiple times. So, even if the number of instructions are finite, the program might run for an arbitrary number of cycles. However, every test program is guaranteed to halt eventually. You will not get infinite loops.

2.1.1 Registers

We will allow the use of all the 32 programmer visible MIPS registers (0-31). Please note that register 0 is read-only and should always contain the value 0. We will also not be using registers 26-31 since we are not testing for function calls or kernel operations. You can keep these constant or 0 them out.

You will need to implement a register file that maintains the values in all the 32 registers and updates the registers as the instructions are executed.

2.1.2 Data Segment

The data segment should immediately follow the text segment, but you can assume the text and data segments are kept separate. Your program should be able to support a maximum of 32 data words. When the program sees load or store instructions, the addresses will resolve to one of these 32 data words. The data segment should be populated at the before the beginning of the first cycle. As execution proceeds, loads can read from the data segment and stores can write to the data segment.

2.1.3 Instructions

Our input file will only contain the following instructions

Instruction	Example	Meaning	Opcode/Funct Code
add	17387552	add $t2,$t0,$t1	32
sub	17387554	sub $t2,$t0,$t1	34
lw	-1912078332	lw $t0, 4($s0)	35
sw	-1375207420	sw $t0, 4($s0)	43
sll	606528	sll $t0,$t1,5	0
andi	839974927	andi $s1,$s0,15	12
ori	889782287	ori $t1,$t0,15	13
bne	352911362	bne $t0,$t1,2	5
noop	0	noop	0
halt	1	halt	1

Here, we define the HALT instruction as just 1. The HALT instruction is made up and signifies the end of the program. It is not an actual instruction, it is just a signal for us. When the simulator runs into the halt instruction (funct field 1), it has to stop executing.

2.1.4 Sample Input

You will get a 32 bit signed integer for each machine instruction. The last instruction will be the HALT instruction, which will just be encoded as 1. Following that, You will have the blank like that separates the text and data segments. The rest of the lines on the files will be the data words used in the program (maximum 32). You need to keep track of the starting address of the data segment. In the following sample input file, instructions take up lines 0 - 5 (bytes 0-23), which means the data segment begins at memory address 24.

Shown below is a sample input file.

```
873463832
-1912012800
-1911947260
19552288
-1375010808
1

15
25
```

3 Output

Your output should consist of the following for EVERY CYCLE.

- The value in the PC
- All the data memory words
- The entire register file
- The values in each pipeline register

There is a sample output file on the course website. It is too unwieldy to attach here. Please make sure your output is formatted exactly as it is there.

4 Suggested Approach

This is a suggested development approach for this project. Please note that you are not required to do this EXACTLY this way, but it will help.

1. First, set up some form of structure/array to read in the machine instructions and store them.
2. Set up a structure to contain the interpretation of the instruction into its component parts (instruction type, instruction, rs, rt and rd registers, immediate fields, etc.). Have an array of these structures, one for each instruction.
3. Set up an array/structure for the register file and the data memory.
4. Set up a structure for each of the pipleine registers (IF/ID, ID/EX, EX/MEM and MEM/WB). Make sure yu are saving all the information you need (and some additional information if required).
5. Write some helper functions for extracting the opcode, register numbers, immediate fields, funct field, etc. You can do this by bit shifting.
6. Write a helper function which could print a state structure, formatted like the output.
7. First implement the pipeline for a program that has no hazards.
8. Add support for implementing data hazard mitigation through stalling. You can stall by inserting NOOP instructions and moving the PC back to its old value.
9. Add support for data hazard mitigation using forwarding.
10. Add support for branch hazards by stalling.
11. Add support for branch hazards using branch prediction.

4.1 Data Hazards

In our implementation, we will resolve data hazards by forwarding. You may want to simulate a forwarding unit in the EX stage which checks for the data hazard conditions we discussed in class and performs forwarding from the appropriate location when necessary. The only stall required for data hazards will occur when the load word instruction is immediately followed by another instruction which reads load words destination register. You can implement this by checking the conditions discussed in class and inserting a NOOP in place of the subsequent instruction. NOOPs are characterized by zeroed out pipeline registers and the decimal representation of a NOOP instruction is simply 0. Note that we assume that, in a given clock cycle, register writes take place before register reads. Therefore, any possible data hazards between an instruction in the WB stage and an instruction in the ID stage are already resolved.

4.2 Control Hazards

For branch hazards, we will be implementing a simple version of the 2-bit Branch Prediction Buffer which has an entry for every slot in the instruction memory. Every instruction is initially considered weakly not taken. When an instruction is decoded in the ID stage, we check the BPB. If the BPB entry indicates that we should predict taken (i.e. the entry is 2 or 3), we will insert a NOOP behind the branch instruction (i.e. zero-out the IFID pipeline register), and write back to PC the branch target calculated in the ID stage. If the BPB entry indicates that we should not predict taken (i.e. the entry is 1 or 0), we will continue execution as normal. In the EX stage, we compute the branch decision. If the decision was correct, we simply need to make sure this is reflected in the BPB (ensure that the entry is either strongly taken or strongly not taken, depending on the situation). Otherwise, we flush all subsequent instructions executed (i.e. zero out the IFID, IDEX pipeline registers) and write the correct instruction address to PC, as well as update the BPB. Note that the IDEX pipeline register has space for both PC+4 and the branchTarget so this information is easily accessible.

When we make an incorrect decision, we either update the entry to a weakly taken position from a strongly taken position or we update the entry from one weakly taken position to another. This is easily done with the following rules: if we predicted not taken, but we should have taken, then bpb[inst]+=1. If we predicted taken, but we should have not taken, then bpb[inst]-=1.

State	**Binary**	**Decimal**
Strongly Taken	11	3
Weakly Taken	10	2
Weakly Not Taken	01	1
Strongly Not Taken	00	0

4.3 NOOOP and HALT

The NOOP instruction in MIPS, which has the decimal value 0, is actually the instruction the shift-left-logical instruction sll $0, $0, 0. However, because $0 is hardcoded to always contain the value 0, this instruction has no significant effect as it moves through the pipeline. The easiest way to implement a NOOP is to simply treat it like the sll instruction that it is. If your pipeline is constructed correctly, it will move through the processor with no effect.

When inserting a NOOP into the pipeline after the IF stage (as is the case for stalling and flushing), the behavior is different. In this case, we should simply zero-out all control signals but leave any data that has been already fetched/computed.

The halt instruction is a made-up instruction for this project that simply tells us when to stop execution. We will characterize the halt instruction as being an instruction whose control lines are all set to 0.

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 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.

6 Submission and Grading

You are required to turn in the following:

- You C program, called "proj2_LastName.c"
- A suite of test cases that you have generated, beyond the test cases provided, that demonstrate your rigorous testing of your solution.
- A README file listing all the instructions your program can handle, and all known issues with your program (parts not implemented, parts implemented with issues, etc).

Please tar your C program and your test cases into one tarball. Please make sure your tarball does not contain any other files, especially executables.

Submissions may be made through Canvas in the Assignments section. You must submit before 11:59:00 PM on March 13 to receive full credit. Late submissions will be accepted for 10% off on March 14. The first person to report any errors in the provided materials will be given 5% extra credit. 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 `linprog` with the following commands, where `test.in` is a MIPS executable, as described below:

```
$ gcc proj2.c
$ ./a.out < test.in
```

You should not rely on any special compilation flags or other input methods. The grade breakdown for the project is as follows:

- Reading and parsing - 10 points
- Setting up the required pipeline registers - 10 points
- Populating the data memory and maintaining the register file - 10 points
- Handling NOOP and HALT - 10 points

- Handling a program without hazards - 10 points
- Handling Data Hazards through forwarding - 10 points
- Stalling when forwarding is not possible - 10 points
- Handling branches by assuming branch is not taken and correcting later - 10 points
- Handling branches using prediction - 10 points
- Quality of the test suite - 10 points

Please do NOT write header files / break your code into several files. This makes it harder to run through plagiarism detection software.

Programs that do not compile will automatically receive a grade of 0.

Prgrams that crash with a segmentation fault will receive a 10% penalty for every test case that crashes.

Students are responsible for making sure that the right file(s) are turned in to Canvas. Please note that we will extend the homework only for documented absences excused by the FSU absence policy. Students will not be given extensions if they accidentally delete or tar over their files.