CDA3101 – Project 2

MIPS Pipeline Simulator

Submission:

1. Create a tar file that includes your program and any test files you may want the TAs to use.
2. Make an electronic submission of your tarball file to Canvas.

Due Date:

1. Oct 31, 2018, 11:50pm for all students

Late Penalty:

1. 10% per day (round up)
2. Maximum two days

1. Overview

The primary purpose of this project is to help you understand the pipelining process for a simple set of MIPS instructions. You will gain experience with basic pipelining principles, as well as the hazard control techniques of forwarding, stalling, and branch prediction.

2. Basic Description

Your job is to create a program, contained in a single C or C++ file called `proj2.c` or `proj2.cpp`, which takes as input a small simplified MIPS assembly program and prints to standard output the state of the pipelined datapath at the beginning of each clock cycle.

At the end of execution, you should print out some information about the instruction sequence just executed. Your submitted C/C++ program will be compiled and run on `linprog` with the following commands, where `test.asm` is an assembly file as described below.

```
$ g++ -o pmips proj2.cpp
$ gcc -o pmips proj2.c
$ ./pmips < test.asm
```

Of course, you need to use g++ if you have a C++ program, or gcc in case you have a C program. **If you use C, a partial program is provided. You are on your own if you use C++.** From this point on, all descriptions assume that the program is written in C. You should not rely on any special compilation flags or other input methods. If your program does not compile and execute as expected with the above commands, points will be deducted from your project grade.

Take a look at the provided `proj2_start.c` file. The `main`() function creates a state that represents the state of the pipeline as a whole at the beginning of a specific clock cycle. Notice that it contains pipeline register structs to record the values of the pipeline registers in that specific cock cycle. Note that, in a real pipeline, all stages are executed at once. We cannot do this because our code will be executed sequentially. To mimic this "parallel" execution, we have `current`, which represents current state of the pipeline (in other words, the state of the pipeline at the end of the previous cycle), and `next`, which should be used to represent the state of the pipeline after the current cycle has executed. The state is initialized and then we enter a while loop with the following steps:

i. Print the state.
ii. Check to see if a halt instruction is entering its WB stage. If so, then we must be done. Print execution information before the end of the program.

iii. Create `next`, a copy of the current state. Any changes to the pipelined datapath are reflected in `next`. In general, while simulating the execution, state should only be read from and `next` should only be written to. However, there a few important exceptions.

iv. Comments following indicate the general order in which steps should be implemented. Note that this order is not strict – for example, we assume that register writes (performed by instructions in their WB stage) must happen before register reads (performed by instructions in their ID stage).

There are many helper functions and macros that are provided for your convenience. You will need to add additional code in the `main`() function and more functions to support the simulation. The number of additional functions and their organization depends on how you approach the problem. You may also modify the structs to include more information (control lines, for example) as well as any other portion of the code as you like. You may even disregard the provided template to start your own from scratch.

3. III. Assembly Input File

Our simplified ISA has 8 registers, denoted `$0-$7`. As in the real MIPS ISA, the $0 is required to contain zero, it may not contain any other value. Any instructions that write to register $0 simply have no effect in the WB stage. Also imagine that we have two separate memory elements for instructions in data. Both elements contain only 16 slots. In other words, instruction memory only has room for 16 instructions and data memory can only hold 16 pieces of data at a time. Both instruction and data memory are word aligned, beginning at address 0. That is, the first instruction is stored at address 0, the second instruction is stored at address 4, the third instruction at address 8, etc. Note that data and instructions are stored in arrays, so the instruction at address 0 will be stored in `instrMem`[0], the instruction at address 4 will be stored in `instrMem`[1], etc. Make sure you keep this in mind while writing your simulator.

The input to your pipeline simulator will be a small assembly file containing limited MIPS assembly instructions. The format of the file is as follows (note the indented instructions are indented by a tab character and no space appear between instruction arguments):

```
.data
        .word w1,w2,w3,…,wn
.text
       <instruction 1>
       <instruction 2>
      …
       halt $0,$0,$0
```

The input to your pipeline simulator will be a small assembly file containing limited MIPS assembly instructions. The format of the file is as follows (note the indented instructions are indented by a tab character and no space appear between instruction arguments):

The `.data` section simply contains a `.word` directive, which places the 32-bit representations of the values `w1, w2, …, wn` in consecutive entries in data memory. The `.text` section contains the instructions to be executed in the pipelined datapath. Our modified MIPS assembly files always contain a halt instruction at the end of the file – this is purely to signify the end of execution in the pipelined datapath. The arguments to halt have no meaning. The supported instructions include:

Instruction	**Meaning**	**Example**

add	Add the contents of the rs and rt registers, and store the result in the rd register. opcode: 0, funct: 32	add $3,$1,$2
sub	Subtract the contents of the rt register from the rs register, and store the result in the rd register. opcode: 0, funct: 34	sub $3,$1,$2
lw	Add the immediate field to the contents of the rs register. The result is used as an address in data memory, whose contents are written to the rt register. opcode: 35	lw $2,4($1)
sw	The contents of the rt register are stored in data memory at the address computed by adding the contents of the rs register to the immediate field. opcode: 43	sw $2,4($1)
beq	If the contents of the rs and rt registers are not equal, then the next instruction to be executed is indicated by the immediate field using the following relation: branch target = (PC+4) + immed<<2. Otherwise, continue executing instructions sequentially. opcode: 4	beq $0,$1,-2

In addition, two auxiliary instructions: "`noop`" and "`halt`", shall also be supported for proper execution of the pipeline. You are provided with a few supportive functions for parsing the assembly file and initializing the processor. The `init_state`() function initializes the starting state of the processor by zeroing out all register values and memory fields, inserting into data memory the word values from the assembly file, and inserting into instruction memory the unsigned integer representation of the instructions from the assembly file (generated by the `instrToInt`() function).

4. Suggested Development Approach

You are provided with a partial proj2.c file named `proj2_start.c` – you do not need to use this file, it is simply some code to get you started but you are welcome to take a different approach. Your output must be formatted in the same way as the `printState` function, however. Your code should read the current state of the pipelined datapath, execute the instructions one cycle at a time, and update the new state of the pipelined datapath and control accordingly.

Start with the goal of simply correctly pipelining instructions which contain no hazards. Once you have accomplished this task, add support for data hazards. You'll notice that the only way to move values into the registers is with the load word instruction, so even the simplest nontrivial programs are likely to have data hazards – it is recommended that you start with pipelining simple add/sub sequences with hardcoded register values, then add support for load word.

Data Hazards

For this project, you shall resolve data hazards by forwarding. You can start 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. You shall also introduce a stall cycle when a load word instruction is immediately followed by another instruction which reads load word's destination register. You can implement this by checking the conditions discussed in class and inserting a NOOP in place of the subsequent instruction. NOOP's are characterized by zeroed out pipeline registers and the machine instruction is simply a 32-bit number 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.

Even with the limited types of instructions, there are many other scenarios of data hazards. You need to resolve all data hazards via either forwarding for stalling in a pipelined sequence of instructions. Your execution report at the end of the program should summarize the total number of execution cycles, the total number of forwarding steps, the total number of stalls, the total number of branches and the number of branches taken.

Control and Data Hazards for BEQ instructions

For control hazards, we assume the pipeline implementation has refined the datapath and control by moving the comparison and target calculation of BEQ instructions to the ID stage, as we have covered in class. With this refinement to the short 5-stage pipeline, the benefit of dynamic branch prediction is limited. Thus we take a simple static scheme to predict every branch instruction will not be taken, i.e., the PC will be updated and written to the IFID pipeline register with the address of the next instruction, just like all other instructions. When an instruction is decoded and determined to be taken instead (at the end of the ID stage), we will insert a NOOP behind the branch instruction (i.e. zero-out the IFID pipeline register), and write back to PC the new branch target calculated in the ID stage.

This refinement to BEQ instructions mandates the input operands to be available at the ID stage. Similar to the hazard detection and forwarding logic that is needed to other instructions, you shall provide another set of hazard detection/forwarding logic in the ID logic for BEQ instructions. You shall resolve the data hazards via forwarding whenever possible. Otherwise, you add stall cycles to the pipeline. The occurrence of forwarding and stalling shall be included as part of the final execution report.

Handling NOOPS and Halts

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 for data hazards), 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, i.e., no need to fetch the instruction again.

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 with an opcode of 63. Its sole purpose is to stop the simulator at the end of its WB stage.

5. General Notes/FAQs about the Project

If you find it possible to leverage the provided template, there are not many lines of code to write. But there are many test cases to think through and test your code correctly with. So be warned that it takes a lot of thinking and debugging to get your code right.

Another point: addresses are word-aligned to mimic real MIPS addresses. So, addresses are multiples of 4 (0, 4, 8, 12, etc.), but their entries into the instruction memory and data memory arrays are sequential (0, 1, 2, 3, etc.). Keep this in mind when creating your simulator.

Take the time to look at the `mipsState_t` and pipeline register structs to see how they are constructed and hopefully this will point you to the right direction for the IF stage. You may start without worrying about hazards or anything like that. Just try to move data through the datapath. Slowly build up basic functionality until you can test the first test file provided (use the sample executable on `linprog` for verification). Only when that looks good should you move on to adding additional functionality. Take it test case by test case.

Again, you are absolutely welcome to modify the code provided (for example, by adding control lines to the pipeline registers), or start from scratch as you'd like. You are required to produce the exact same output format for ease of grading. Of course, you report should report the same number of cycles, stalls, forwards, branches, and branches taken, as well as correct values for the memory words and registers. In general, make sure you're paying attention to the little details in the writeup.

6. Breakdown of points

The grade breakdown for the project is as follows:

Points	Requirements
5	Conformant to the required compilation and execution commands.
5	Compile, execute and interpret instructions
10	Handle code sequences without any data hazards
15	Handle avoidable data hazards to non-BEQ instructions by forwarding
15	Handle unavoidable data hazards to non-BEQ instructions by stalling
10	Handle avoidable data hazards to BEQ by forwarding
10	Handle unavoidable data hazards to BEQ by stalling
10	Handle control hazards from BEQ by stalling
10	Handle BEQ instructions when the branch is not taken
10	Report the number of instructions and total clock cycles correctly

7. Miscellaneous

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 for those involved.