CDA3101 Project 2: Pipeline Simulator
Due 7/7

I. Purpose

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. The secondary purpose of this project is to give you experience with writing C code.

II. Basic Description

Your job is to create a program, contained in a single C file called **proj2.c**, 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 program will be compiled and run on linprog with the following commands, where test.asm is an assembly file as described below.

```
$ gcc proj2.c -lm
$ ./a.out < test.asm
```

The –lm flag links the math library, which is included by default in proj2_start.c. You do not have to use any functions from this library, but you may if you choose to do so. You should not rely on any other 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. At the heart of the proj2_start.c file is the run() function. The run() function creates a state, represented by the stateType struct. This state 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 state, which represents current state of the pipeline (in other words, the state of the pipeline at the end of the previous cycle), and newState, 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:

1. Print the state.
2. Check to see if a halt instruction is entering its WB stage. If so, then we must be done. Print information about the execution and end the program.

3. Create newState, a copy of the current state. Any changes to the pipelined datapath are reflected in newState. **In general, while simulating the execution, state should only be read from and newState should only be written to. However, there a few important exceptions.**
4. 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).

You do *not* need to modify the instToInt, get_opcode, get_funct, get_immed, get_rs, get_rt, get_rd, get_shamt, or printState functions. These are provided for convenience. You will need to modify the run() instruction and possibly add support functions, depending on how you approach the problem. You may also modify the structs to include more information (control lines, for example) as well as the init_state function if you'd like.

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 .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)
bne	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	bne $0,$1,-2

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).

IV. 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. Within the proj2_start.c file, you will need to modify the run() function. The run() function is the heart of the program and contains the statements for executing a cycle within the pipelined datapath. These statements should read the current state of the pipelined datapath in order to update the new state of the pipelined datapath.

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

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 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 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.

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

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), 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.

V. General Notes/FAQs about the Project

You want to start by modifying the run() function. The first thing you should try to do is implement the steps that take place in the IF stage (notice the comments indicate where you should make modifications). There are only a couple of things you need to do: fill in the IFID pipeline register (read data from state and record changes in newState) and update PC. Altogether it's only three lines of code (because there are two IFID fields to be filled in). 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 stateType structs and pipeline register structs to see how they are constructed and hopefully this will point you to the right direction for the IF stage. After that, tackle the ID stage. But don't worry about hazards or anything like that, yet. 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.**

You are absolutely welcome to modify the code provided (for example, by adding control lines to the pipeline registers) but you do not have to. You can complete the project by solely modifying the run() function. Also, you are welcome to start from scratch if you'd like -- I only ask that you have the exact same output format for ease of grading.

- The number of branches statistic is the number of branch instructions executed, not the number of branches taken.

- The mispredicted branches statistic is the number of branch instructions for which the prediction was incorrect.

- Remember, addresses are multiples of 4 but array indeces are multiples of 1. You must account for this.

- Register writes MUST happen before Register reads -- this means you might have to be creative with your WB stage code placement.

In general, make sure you're paying attention to the little details in the writeup!

VI. Miscellaneous

Submissions may be made through Blackboard in the Assignments section. You must submit before 11:59 PM of July 7 to receive full credit. Late submissions will be accepted for 10% off each day they are late, up to two days. The first person to report any errors in the sample code or executable 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.