**Lab Assignment 3: Writing Pipelined MIPS simulator**

50 points (5% of entire grade)

Due: 11:59pm, Monday May 3rd, 2021

The goal of this assignment is to understand how pipelined MIPS datapath is performed. This pipelined simulator will not only model the functional behavior of each instruction, but also enable pipelined instruction processing. In lab2, you finish every instruction in 1 cycle. But for lab3, you are asked to implement a 5 – stage pipeline.

**What you should do**

Your job is to implement the pipe\_cycle() function in pipe.c. The pipe.c file is part of code template you must use, and details about the template will be explained later. The pipe\_cycle() function should be able to simulate the pipeline of the following instruction:

|  |  |
| --- | --- |
| R-Type | add, addu, sub, subu |
| I-Type | lui, ori, addi, addiu, lw, sw, bgtz |
| J-Type | j |

When there is no instruction to process, the pipe\_cycle() function that you write to set the global variable RUN BIT to 0 so that the program can terminate. In order to get the correct result, you must make sure all the stages finish its task before you halt the whole simulation. Your simulator should simulate each stage precisely by updating the proper pipeline register between two stages. Otherwise you cannot get the correct result.

As in the lab2, each MIPS instruction you are supposed to implement belongs one of MIPS instruction types: R-Type, I-Type, and J-Type. Refer to the MIPS reference data or lecture slides for details of each instruction type. While the table has many instructions, there are actually only a few unique instruction behaviors with a number of minor variations. And you need to know that not every instruction needs to perform all the 5 stages (fetch, decode, execute, mem, wb).

The simulator will execute the input program in the manner of one instruction at a time. After each instruction, the simulator will update the pipeline register. The simulator will change the memory in mem stage and change the register in the wb stage. The simulator is partitioned into two main sections: the (1) shell and the (2) pipeline routine. Again, your job is to implement the pipeline routine.

**The Simulation Routine**

To enable pipelined execution, you must implement the pipeline registers (by defining your own data structures) between different pipeline stages and assign values to these pipeline registers as instructions pass through the pipeline. Every time after an instruction finishes the Writeback stage, CURRENT STATE should be modified accordingly to reflect the effects of this instruction. Use good coding practices – for example, define a data structure like below to hold all pipeline registers between the fetch and decode stage (similarly for other stages).

typedef struct Pipe\_Reg\_IFtoDE {

//pipeline register 1

// ...

//pipeline register n

} Pipe\_Reg\_IFtoDE;

In order to get the correct result, you must make sure all the stages finish its task before you halt the whole simulation.

In the pipe.c file, you should implement the 5 pipeline stages using 5 functions that are called by the pipe\_cycle function. **You are only supposed to implement the logic of these five functions, editing the pipe.h file. Other changes are not permitted,** as TA will compile and run your submission. Also, you should keep the 5 pipe\_stage\_\* function calls in tact in the pipe cycle function.

void pipe\_cycle()

{

pipe\_stage\_wb();

pipe\_stage\_mem();

pipe\_stage\_execute();

pipe\_stage\_decode();

pipe\_stage\_fetch();

}

**Steps to this Lab**

We suggest that you should develop the pipeline simulator as follows. Implement the 5 stages of the pipeline in the pipe.c file without worrying about any data or control dependencies, as well as advanced technologies such as data forwarding. Remember that in each cycle the simulator runs each stage once. Try to finish as much as possible since partial credits will be given upon the completion of your source code.

**Lab Files**

The lab3.zip contains all code template for this assignment. You can compile the simulator with the provided Makefile. You will be provided with the same input files for testing your implementation. To test whether your implemented simulator supports all ISAs you are supposed to implement, you should create your own test files, which should be in hexadecimal formats. The simulator won’t accept files in other formats.

The source code for lab3 contain 4 files. We provide 2 files (shell.c, shell.h) that already implement the shell and other 2 files (pipe.h and pipe.c) where you will implement the pipelined simulator. Note: shell.c and shell.h have some modify from lab2 and you can only write your code on pipe.h and pipe.c

**Submission Guideline:**

* All program submissions should be made to Canvas.
* One copy of the code needs to be submitted for one group
* Write down team member’s name at the top of the pipe.c file.
* Document your code well so it is easily readable.
* You need to any additional document or test files/cases that are used in your implementation.

**Useful Tips:**

* Please understand how pipeline control works first.
* Please refer to MIPS green sheet. <https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf>.
* Use the MARS simulator to generate corresponding machine code for an individual instruction or a MIPS program.
* Use the MARS simulator also to convert decimal to binary or hexadecimal, or vice versa.
* Again, you are only allowed to modify pipe.c and pipe.h and leave the other files as it.
* Ask TA for help if you feel confused!