Lab 6: Pipelined CPU Design in PyRTL

Assigned: *Wednesday, February 22nd, 2023*
Due: *Wednesday, March 1st, 2023*
Points: *100*

- MAY ONLY BE TURNED IN ON **GRADESCOPE as PYTHON files** (see below for details).
- There is NO MAKEUP for missed assignments.
- We strictly enforce the LATE POLICY for all assignments (see syllabus).

Goals for This Lab

By the time you have completed this work, you should be able to utilise **PyRTL** and Python to simulate pipelined CPU hardware for multiple instructions with hazard resolution using stalling and forwarding. A goal of this lab is also to develop the skill to understand and modify the code of a large hardware design.

Task

The skeleton code provides a pipelined CPU using PyRTL that can run the following 9 MIPS instructions.

ADD, AND, ADDI, LUI, ORI, SLT, LW, SW, BEQ

Your task is to implement forwarding logic that optimises the resolution of data hazards in the pipeline.

Provided Files

We have provided you with three sets of skeleton files (see the Canvas folder):

1. ucsbcs154lab6_skeleton.py - This is the skeleton file that contains the code for a pipelined CPU. You are required to start building your forwarding logic using the code in this file. You should read this thoroughly before implementing everything for this assignment.
2. add.s, branch.s, lw.s, sw.s- These are sample assembly programs that you can use to test your CPU. They should operate correctly on the skeleton, but exhibit no hazards. You must write more tests with hazards to test your final CPU.
3. mips_to_hex.sh - This script will assemble instructions from a MIPS test for you to load into instruction memory of the CPU before simulation begins.

Instructions

The following figure shows the implementation of the pipelined CPU present in the provided skeleton file ucsbcs154lab6_skeleton.py. This CPU implements the core instructions but **will not execute correctly in the presence of data hazards**! For this assignment, we first want to implement the forwarding/bypassing logic needed for correct and optimised execution of the instructions.

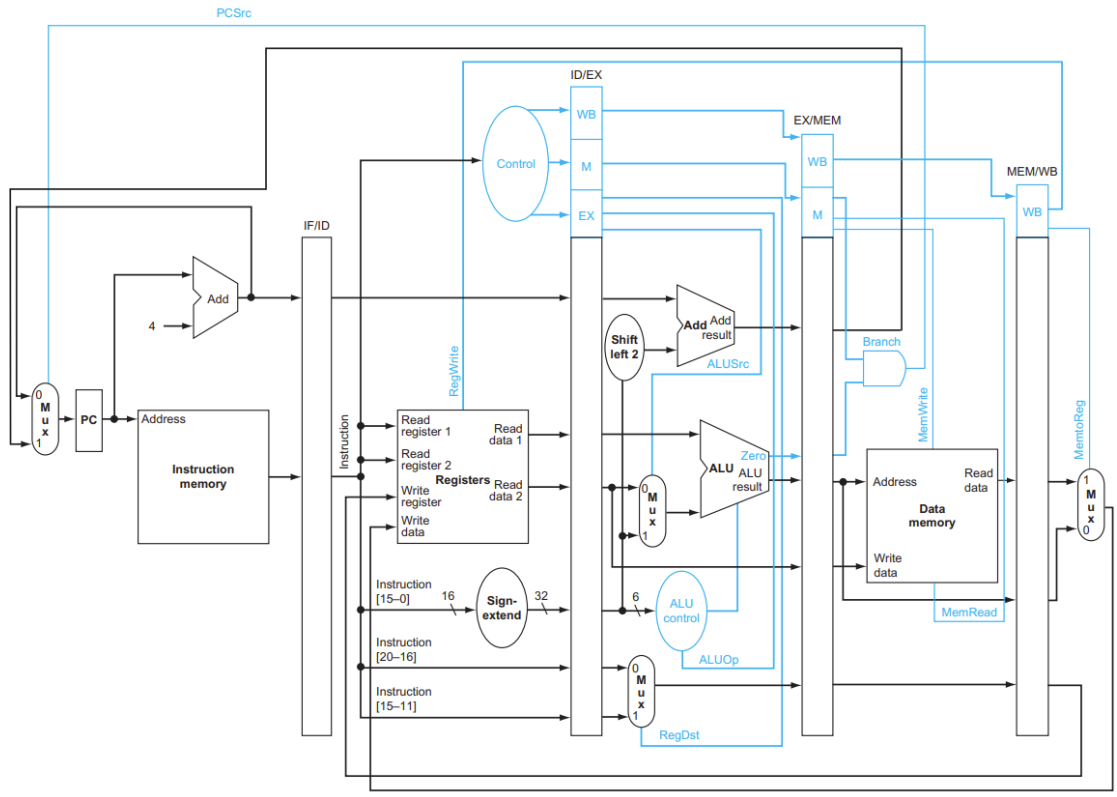

Figure 1: Skeleton pipelined CPU design with no stalling and no forwarding

Forwarding Logic

Figure 1 shows the skeleton CPU pipeline with no stalling and no forwarding. Figure 2 shows a complete, optimised pipeline design.

The Forwarding Unit is a performance optimisation beyond simply stalling the pipeline, which resolves data hazards via forwarding. This can be noted from the datapath presented in Figure 2, which is modified compared to Figure 1.

The skeleton file implements the pipelined CPU shown in Figure 1. You will implement the Forwarding logic, which includes the multiplexers to the inputs to the ALU, the pipelining of additional rs/rt/rd register numbers, and the Forwarding Unit itself.

There are multiple types of data hazards to consider while implementing the Forwarding logic. You should write several test cases to expose each type of hazard and then ensure that your final pipeline implementation correctly stalls and forwards to ensure that instructions' dependencies are maintained.

Implement the necessary forwarding logic, including the Forwarding Unit, in ucsbcs154lab6_forward.py using the skeleton pipelined CPU from ucsbcs154lab6_skeleton.py provided to you. You will submit your file to Gradescope.

Figure 2 shows the Forwarding Unit as well as the Hazard Detection Unit. Note that this figure does not have all the details from the full datapath, such as the branch and sign extension hardware. You do not need to implement the Hazard Detection Unit.

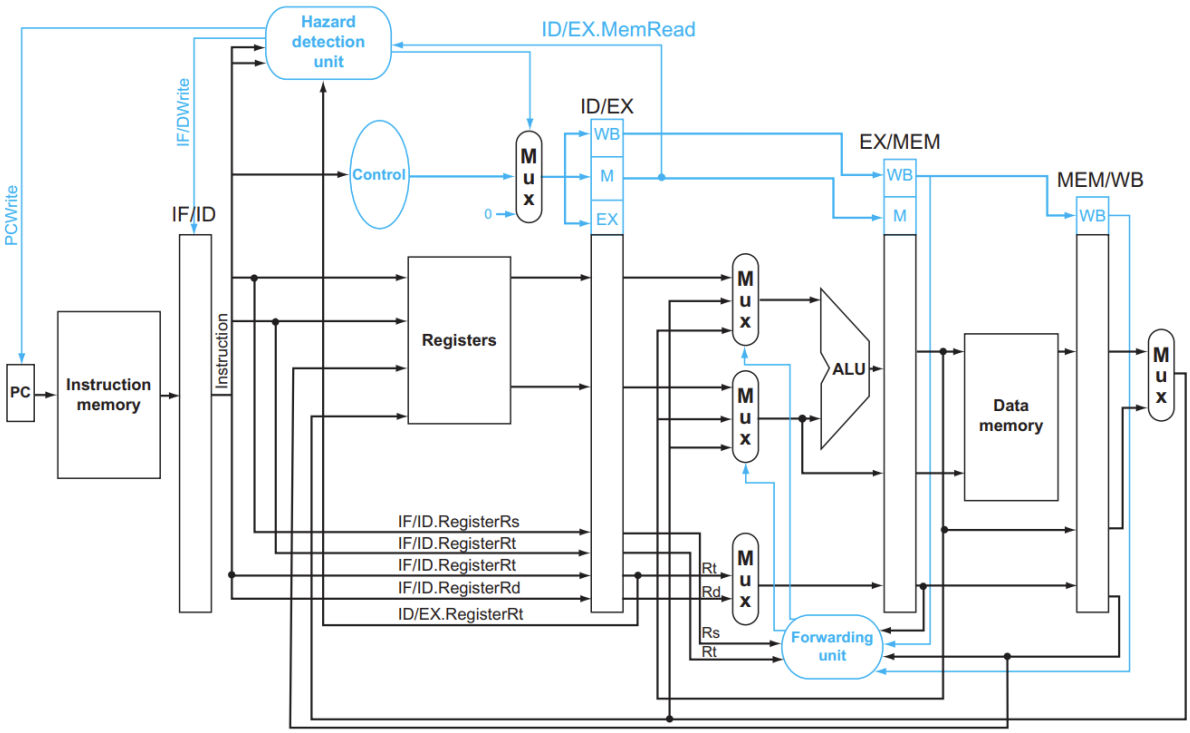

Figure 2: Final CPU pipeline design with both stalling and forwarding

Design Note

Unlike the last lab, the **i_mem** memory is *BYTE* ADDRESSABLE instead of WORD ADDRESSABLE. This means that when the PC increments, it *increments by 4* instead of 1. It also means that BEQ requires left shifting by 2. Note that MIPS uses byte-addressable memory. However, **d_mem** is still *WORD* ADDRESSABLE.

Test your Design!

How do you test your PyRTL CPU design? Well, we just need to write some sample MIPS programs and run them on your CPU, just like you did in the last lab!

1. Write a MIPS program
2. Assemble the MIPS program (using mips_to_hex.sh)
3. Put the assembled instructions (machine code) in hex format into i_mem_init.txt
4. python ucsbcs154lab6_forward.py

We've provided you with some sample test cases in add.s, branch.s, lw.s, sw.s. You can assemble each test into an i_mem_init.txt file using mips_to_hex.sh.

Files to Submit

You need to submit one file for this lab.

- ucsbcs154lab6_forward.py : Implement the forwarding logic in this file using the skeleton code from the ucsbcs154lab6_skeleton.py file.

Please keep your eyes open for any Piazza announcements in case of any updates to the submission requirements.

Autograder

Unlike previous labs, we will NOT release an autograder to verify your CPU. Part of this assignment is to learn how to test your code! We WILL allow you to share **test code** (MIPS assembly) with each other on Piazza. While this is optional, you may write your own comprehensive test suite in MIPS and share it on Piazza under our stickied post, and you may also download your classmates' tests if they've uploaded them.

We (the instructors) will not be responsible for debugging any mistakes in these test suites, nor will we endorse any of them. If you think a test suite has a bug or issue, you should comment under the author's submission with your concerns. Lastly, passing these tests is not a guarantee of any score on the assignment or from the autograder.