Guide For ECSE425 Final Project - Pipelined Processor

Winter 2022

February 25, 2022

This guide is designed to provide you with some tips and hints for implementing the MIPS pipelined processor. These are suggestions for your implementation, you are not required to follow these suggestions.

1 Resources

Below are some good resources to learn the operation of a pipelined processor. They should give you a general idea of what each pipeline stage should do and what input/output signals may be needed.

- https://www.dcc.fc.up.pt/~ricroc/aulas/1920/ac/apontamentos/T07_mips_implementation.pdf
- https://www.scss.tcd.ie/Jeremy.Jones/vivio%205.1/dlx/printable.htm

2 Suggested Components

Below are some suggestions on how you might want to breakdown the project into different components. Note that these are just suggestions, you are free to implement the processor in any way that you see fit. The tips on input and output signals of each component DO NOT include all the signals you might need for your design.

2.1 Instruction/Data_Memory.vhd

- Adapted from 'memory.vhd' of the Cache project
- Read from 'program.txt' outputted by the Assembler
- Address of memory is defined by byte (integer range from 0 to RAM_SIZE-1), where RAM_SIZE is 32768 bytes
- MIPS processor instructions are always in words, so you need to concatenate 4 bytes together in the end to form the 32-bit instructions that you can pass on to later pipeline stages

You need to define the entity instruction_memory in a similar way as 'memory.vhd' from the Cache project, except the writedata and readdata signals are 32-bits instead of 8-bits, as we are reading from 'program.txt' line-by-line.

2.2 Fetch.vhd

- Fetch instructions from instruction_memory
- Update PC accordingly (+4 for basic instructions, specified address for branches and jumps)
- Some important inputs to the Fetch component may include *clock, reset, stall*
- Some important outputs may include *PC (or next address), instruction*

2.3 Decode.vhd

- Define register file here (32 registers, R0 is always zero)
- Decode instructions according to the MIPS Assembler specifications
- Sign-extension can be implemented here
- Handle hazard detection and forwarding
- Some important inputs to the Decode component may include *clock, current_PC, instruction*
- Some important outputs of the Decode component may include *read_data_1, read_data_2*

2.4 Execute.vhd

- Check the type of operation and execute various types of instructions including arithmetic, logical, transfer, shift, memory, and control flow (jumps and branching).
- Continue to deal with hazard detection and forwarding
- Some important inputs may include *instruction, rs_data, rt_data*
- Some important outputs may include *alu_result, updated_PC*

2.5 Memory.vhd

- Handle read/write to data memory (lw, sw, etc.)
- Prepare for write-back stage
- Continue to deal with hazard detection and forwarding

- Some important inputs may include *clock, instruction, rt_data_in (data needs to be written), address (coming from output of ALU)*
- Some imporant outputs may include *rt_data_out (read from memory)*

2.6 Write_Back.vhd

- Write back to register file (the number written back to the registers will be used in the Decode stage)
- Some important inputs may include *instruction, read_data, alu_result*
- Some important outputs may include *write_data (this will be used in Decode stage)*

2.7 Pipelined_Processor.vhd

- Connect all components together (initialize them first and then connect them to appropriate signals)
- Write register values and data memory to 'register_file.txt' and 'memory.txt', respectively

2.8 Testbench.tcl

Compile and run your code. Your testbench should run for 10,000 clock cycles, after which point the simulation should stop and the output should be written to the proper files. You may assume that we will not test your processor on a program with a duration of longer than 10,000 clock cycles. Give your clock a frequency of 1 GHz.

3 Guidelines on Unit Tests

These guidelines on component test are meant to provide you with some ideas of the types of intermediate results you could show. However, you are free to include additional intermediate results that you think are appropriate. You can consider constructing individual testbench for the pipeline stages, which makes your design more modular and easier to test.

3.1 Instruction Fetch

The testbench for the Fetch component should be able to test if an instruction is read correctly and if the PC is correctly updated. For example, you can assert an error evaluation where the simulator reports an error when the current instruction and PC signals do not match what you are expecting.

3.2 Instruction Decode

The testbench for the Decode stage should be able to test if the instruction is decoded correctly. Suppose you are testing the instruction *addi $1,$1,1*, you can show that the instruction type indeed evaluates to addi, and rt evaluates to 1, rs evaluates to 1, and immediate value evaluates to 1.

3.3 Execute

You need to show that the output of the ALU is indeed the correct value that you are expecting with your tests. For example, if you are testing 2×7, you can assert an error evaluation where the testbench outputs an error message if the ALU output does not evaluate to 14.

3.4 Memory and Write-back

You could show that the data read from memory is indeed what you are loading (lw) and storing (sw). Usually, if these two stages are correctly implemented, majority of your pipelined processor's functionality should be correct as well.