ECE352 Final Project

Overview

The final project will involve converting a basic multicycle processor to a pipeline processor. On this package you will find:

- A verilog implementation of a multicycle processor
- An assembler for the processor
- Documentation describing the multicycle processor

Your *(tentative)* tasks over the next weeks will be to:

1. Add a performance counter that will count the number of clock cycles your processor took to execute a program.
2. Add a “nop” and a “stop” instruction. The stop instruction will signify the end of your program and will stop the performance counter.
3. Modify the processor to implement a 5-stage pipeline.
4. Enhance the processor to handle hazards automatically.

Guidelines

Make sure you are familiar with the multicycle processor and its implementation by reading the documentation in **multicycle_documentation**. Note this has some instructions for Quartus regarding modifying memory contents, which will not apply when using Modelsim.

The "initial memory" must have the program and data you want to use for testing; you can generate those using the assembler (in **multicycle_assembler**). Note that the assembler doesn't support the new instructions you're adding in the project, so you'll have to assemble them yourself and specify them as data bytes (or you can add support for them to the assembler). Also note that the assembler is picky with the syntax, all instructions should be indented otherwise they are interpreted as labels.

The **multicycle** directory contains the actual verilog source code files and Quartus project for the processor. You will be mostly using the Verilog for simulation under Modelsim.

Finally, in **ECE352_specific** you have the documentation about the tasks you need to perform in this project lab, as well as some test programs you'll have to demonstrate during the different weeks, and the dual-ported memory you'll have to use to enable pipelining.

Project Guide

This section will give you a rough guide for implementing your processor.

1. Add a “nop” instruction to your processor. The `nop` instruction takes no arguments and has the opcode `nop`. It has the following encoding:

   ```
   nop: 00001010
   ```

2. Add a 16-bit performance counter that will count the number of cycles your processor takes to execute a program. The counter will have the following behavior:

   a. It will be initialized to zero whenever you hit the processor reset button (or assert the reset signal in Modelsim).

   b. It will increment by one every clock cycle.

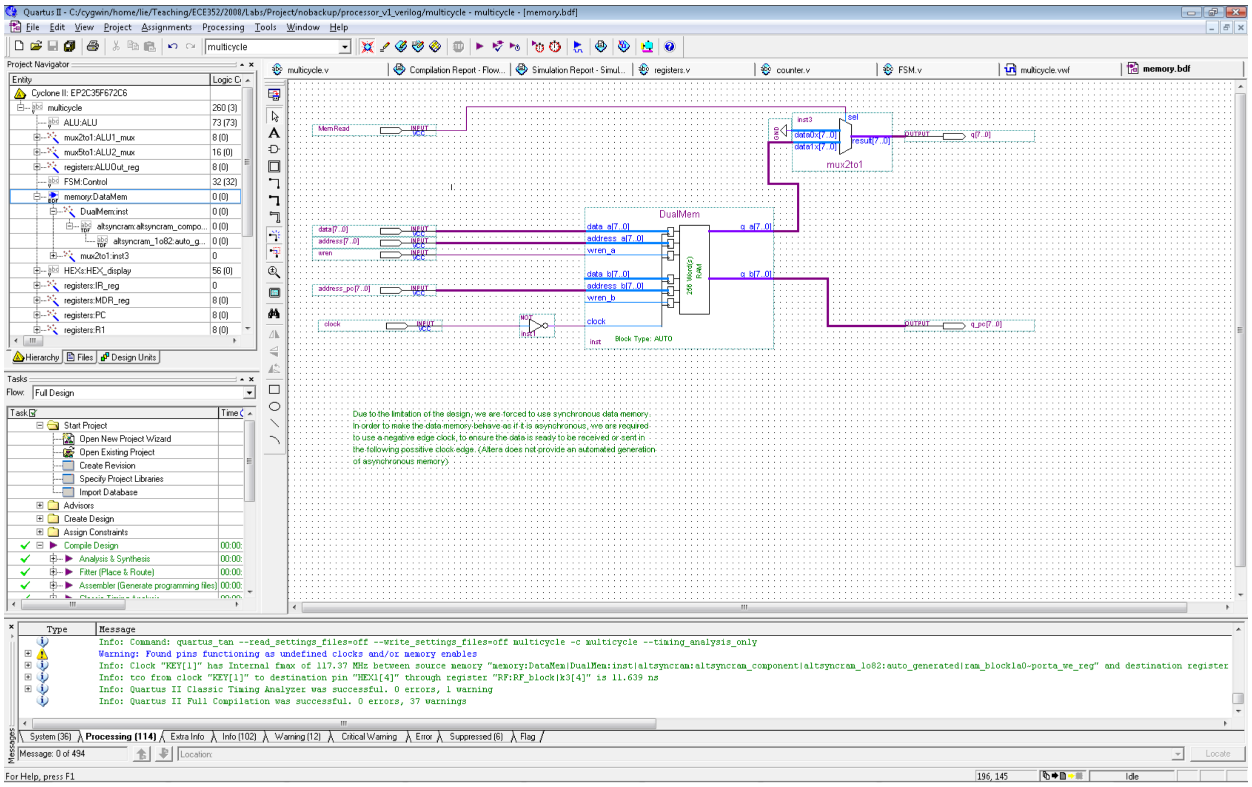
   c. Toggling `SW[2]`, will display the value of this counter on the HEX digit displays. Since you will be running under Modelsim, it’s enough to have the counter as a signal that can be visualized as a waveform.

   d. This counter will stop incrementing when your processor detects a “stop” instruction in the instruction decode stage.

   You must also add this stop instruction to your processor. The stop instruction takes no arguments and has the opcode `stop`. It has the following encoding:

   ```
   stop: 00000001
   ```

3. Convert the single memory into a dual-ported memory. This memory will have a read-only port for the Instruction Fetch (IF) stage and read-write port for executing loads and stores in the EX stage. To do this, unzip DualMem.zip into your project directory and recompile your project. This will install the new dual ported memory in your project. If you open your project in Quartus, by double clicking on the new memory module in your project hierarchy you will be able to view a schematic of the new element:



Due to the limitation of the design, we are forced to use synchronous data memory. In order to make the data memory behave as if it is asynchronous, we are required to use a negative edge clock, to ensure the data is ready to be received or sent in the following possitive clock edge. (Altera does not provide an automated generation of asynchronous memory)

```
Info: Command: quartus_tan --read_settings_files=off --write_settings_files=off multicycle -c multicycle --timing_analysis_only
Warning: Found pins functioning as undefined clocks and/or memory enables
Info: Clock "KEY[1]" has Internal fmax of 117.37 MHz between source memory "memory:DataMem|DualMem:inst|altsyncram:altsyncram_component|altsyncram_1o82:auto_generated|ram_block1a0~porta_we_reg" and destination register
Info: tco from clock "KEY[1]" to destination pin "HEX1[4]" through register "RF:RF_block|k3[4]" is 11.639 ns
Info: Quartus II Classic Timing Analyzer was successful. 0 errors, 1 warning
Info: Quartus II Full Compilation was successful. 0 errors, 37 warnings
```

Notice that the new memory has some new signals:

- The `address_pc` port takes an address to be read.
- The `q_pc` output port will hold the value of memory at the address given by `address_pc`.
- The remaining signals have the same behavior as before. This memory is capable of performing operations on both ports simultaneously in the same cycle.

Now go back to the top level `multicycle.v` and edit the verilog to use these new signals. How should you hook up these new signals?

4. Replicate the instruction register so that there is one for the Register Fetch (RF), Execute (EX), and Write Back (WB) stages. The control signals for each stage should be determined only by the value of its local instruction register. You should also:

   a. Create separate adders for incrementing the program counter after every instruction. The PC increment cannot use the adder in the EX stage.

   b. Instructions that do not use all the stages must set control signals to do nothing when they reach an unused stage. *Hint: you can reuse the control signals that the nop instruction will set for the unused stages of other instructions.*

   c. Branches are tricky because you don't know whether to take the branch or not until the previous ALU operation has completed. Delay the computation of the new PC after a branch until the branch enters the EX stage and all proceeding instructions have gotten past the EX stage. At this point examine the N or Z bit and then write the result directly into the PC register at the end of the EX stage. Note this allows you to reuse the ALU in the EX stage. *Hint: Be careful not to clobber the N and Z registers when there is no ALU operation, they must store the results of the most recent ALU operation for branches to work!*

   Your processor should be able to properly execute any program that:

   - Does not have any data hazards

   - Follows every branch with a 3 `nop`'s (to avoid the control hazard). Note that these are known as "branch delay slots".

5. Enhance the processor to handle control hazards. Detect the presence of a branch in the ID stage and stop incrementing the PC until the branch resolves. Your processor should not need programs to contain branch delay slots to execute properly. *Hint: Create a state machine for the ID stage that remembers whether a branch has been fetched or not, and whether it has resolved or not.*

6. Enhance the processor to handle data hazards. To do this:

   a. Enhance the ID stage to check the later instruction registers in the RF, EX and WB stages for a conflict.

   b. When a conflict is detected, prevent the program counter from incrementing and stall the current instruction from proceeding to the RF stage. Insert `nop`'s into the pipeline until the source of the conflict has completed it's WB stage.

   At this point your processor should be able to execute programs that contain control and data hazards properly.

Building the assembler

The assembler can be built by running **make** on the **multicycle_assembler** directory inside the distribution, given a working C++ compiler (it is a single C++ file). For your convenience, a prebuilt executable for Windows is included (**asm.exe**)

Grading Scheme *(tentative)*

To attain full marks, you must complete all 6 components in the previous section. This project is fairly complicated and the processor design is considerably larger than any of the designs you have worked previously in the labs. To help keep you on track, we will have two check-points which you must meet. The project will be graded out of 16. The checkpoints will be worth 2 marks each and the final project will be worth 12 marks. Note that these checkpoints are *minimum* progress requirements, you are strongly advised to try and *stay ahead of them!* Grades are assigned as follows:

1. Checkpoint #1: Have steps 1-3 completed [2 marks + 2 marks if completed by end of 1st lab]

2. Checkpoint #2: Have step 4 completed. [4 marks + 2 marks if completed by end of 2nd lab]

3. Completed project: Have all step 5 [3 marks] and step 6 [3 marks] completed by end of project.

You are required to have at least one diagnostic program that demonstrates and tests the functionality required in each checkpoint. We suggest having several such programs that test all possible cases. A few diagnostic programs are available under **ECE352_specific/TestPrograms**, these will be used to test your modifications.