ECE352 Final Project

Overview

The final project will involve converting a basic multicycle processor to a pipeline processor. On the course website you will find:

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

Your tasks over the next four 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. Optimize the performance of your processor. Bonus marks will be assigned to the 3 fastest designs in the class as indicated by the metric below.

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.

   b. It will increment by one every clock cycle.

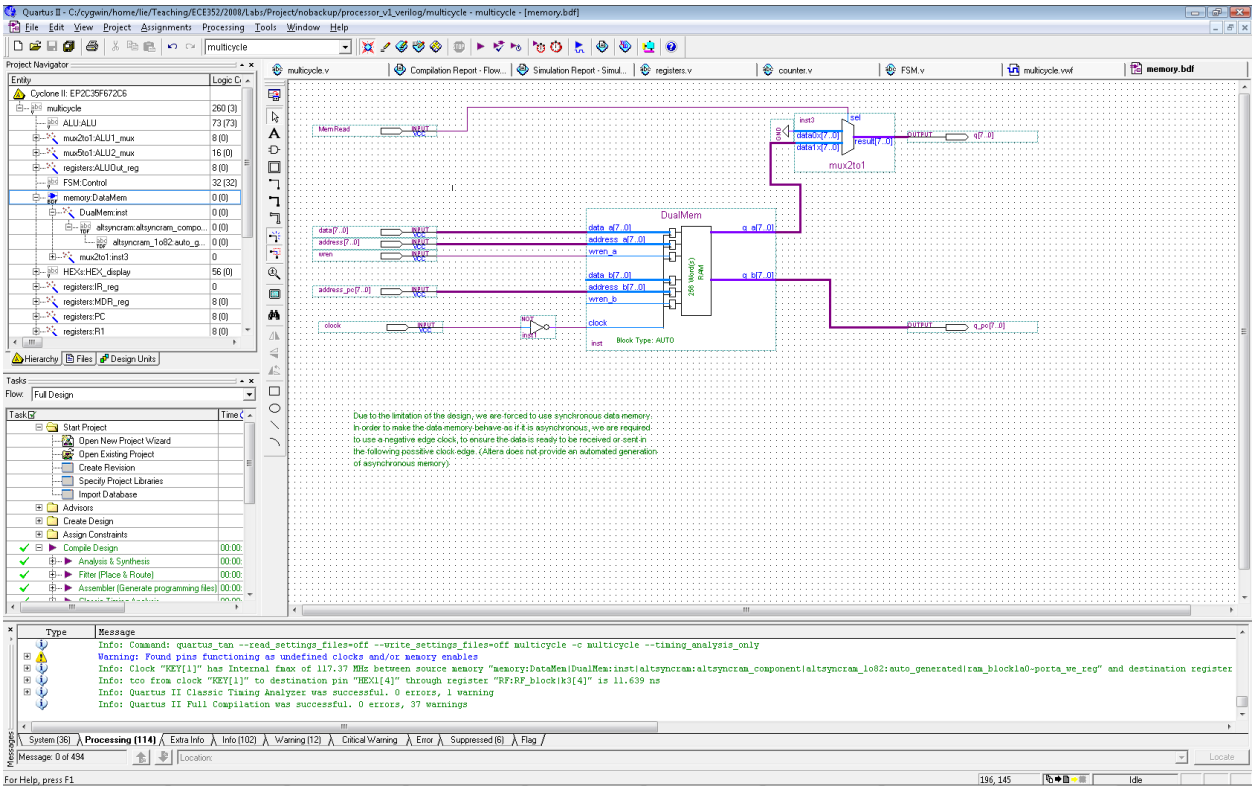
   c. Toggling `SW[2]`, will display the value of this counter on the HEX digit displays.

   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, download DualMem.zip from the course website and unzip it into your project directory and recompile your project. This will install the new dual ported memory in your project. By double clicking on the new memory module in your project hierarchy, you will be able to view a schematic of the new element:



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 source code for the assembler is available on the course website under the project section. The assembler must be built on a system that has GNU G++ installed. The necessary tools are installed on the ugsparc and ecf systems. In addition, the assembler may be built in the cygwin environment on any windows machine (including the ones in the lab). Note that cygwin is installed along with Quartus by

default. You may access a cygwin shell on any machine with Quartus installed by running the cygwin batch file. The instructions below apply to a home machine, the machines in the Bahen labs have a shortcut to the Altera cygwin shell in the start menu:

1. Go to the directory where you have Quartus installed. This is usually `C:\altera\<quartus version>\quartus`.
2. Go to `\bin\cygwin` in the quartus directory and run `Cygwin.bat`.
3. Go to the directory where the assembler source files are located and type "make". Note that to get to `C:\` in cygwin, you need to type "`cd /cygdrive/c`"

Grading Scheme

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. The TA's will test your processor at each stage with their own set of diagnostic programs as well as observing the correct operation of your diagnostic program.

Debugging Hints

Because of the size and complexity of this project, it is absolutely critical that you understand how to debug your system. The base processor design already externalizes many of the interesting control signals and states, so that you may view them on the board and in the Quartus simulator. To view other internal signals in the Quartus timing simulator you must make sure they are output pins in the top level module (i.e. `multicycle`) and make sure there is a path from the internal signal up to the top level module. To view the signal on the board, route it to one of the LED pins. Note that LEDG[2:6] are free and may be assigned new signals of your choice (they are currently tied to a constant value of 1).

You also have available to you a more powerful simulator called *ModelSim*. Martin Labrecque provides a basic tutorial on ModelSim here: http://www.eecg.utoronto.ca/~martinl/modelsim/ . ModelSim is

available on the lab machines, and can also be downloaded and installed for free from Altera on your home machine.

Bonus mark competition

Previously, you competed with implementations made by this year's and previous year's TAs to get your bonus marks. In this project, you will be pitted in a (friendly) competition against your peers! A bonus mark of 1% on your final grade will be assigned to the 5 fastest designs in the class. The fastest designs will be determined as follows:

1. The TAs will provide a set of test programs for your processor to execute.
2. Each program will end with a series of `nop`s followed by a stop instruction. There will be enough `nop`s to ensure that all instructions have completed before the `stop` instruction is fetched.
3. At this point the execution time for your processor will be determined by multiplying the value of the cycle performance counter and the theoretical minimum clock cycle of your processor, as reported by Quartus in the "Timing Analyzer->summary" section of the compilation report. For example, if your processor takes 123 cycles to execute a program at a frequency of 8ns, then your program took 8 x 123 = 984ns to execute. The smaller the number, the faster your processor.

Here are some suggestions for improving your processor performance. You are of course free to use any method you like, including taking advantage of on-chip structures in the FPGA:

1. Reduce instruction stalls by implementing forwarding.
2. Speculatively execute instructions after a branch.
3. Use a cache to minimize latency to memory.
4. Add additional pipeline stages to increase your frequency.

Additional Resources

The base multicycle processor used in this project is the exact same processor used in ECE243. As a result, you may find it helpful to also look over the ECE243 lab and documentation, though not all of it is relevant. The associated ECE243 lab can be found here: http://www-ug.eecg.toronto.edu/msl/nios_labs/lab7/ . Please note that the original assembler used in ECE243 does not support the nop and stop instructions. Please use the assembler posted on the ECE352 course website, which has been modified to accept those instructions.