CSE141L Lab #6 Report

Written 2011-03-11
T. O'Neil (A07716380) and I. Yulaev (A06955201)
CSE141L, Winter 2011
Taught by Prof. Swanson
University of California, San Diego

0 Table of Contents

0.1 Table of Figures

0.2 Table of Tables

1 Overview

This report constitutes our [Tyler O'Neil and Ivan Yulaev's] submission for lab assignment #6, for the CSE141L course. In this lab, we improve our SH17-compliant processor implementation from lab #5, by adding pipelining for the slowest instructions so as to reduce the critical path, and also by enabling our processor to execute several instructions per cycle, i.e. Superscalar execution. We show a clock frequency increase of **43.9%**, and an average reduction in cycle count for executing the SuperGarbage programs of **11.3%,** without modification of the assembly code being executed. The net speedup, as measured by the decrease in real time to execute a program from the SuperGarbage application suite, is **60.2%.**

1.1 Terminology

The terms P&R, "place-and-route", and "place and route" all refer to the "Place and Route" process of the ISE design flow.

1.2 Note to Users

For TAs grading this report, the below notes may be helpful.

1.2.1 Building and Running Our Project

There should be little out-of-the-ordinary as far as running our project. The top-level simulation testbench, for running SuperGarbage apps, is top_tb.v; iSim simulations should be run using this file as the simulation top-level. The top-level synthesizable Verilog file is core.v; synthesis and implementation should be run on this file. **The ISE project itself is the .xise file in the ise/ directory.**

It should NOT be necessary to change the locations of any files, in order to get the design to simulate and synthesize correctly. It also should not be necessary to re-assemble any COE files. **It is absolutely necessary to re-generate the memory core files in Xilinx ISE; this should be done by running the "Manage Cores" operation on each imem/dmem core and pointing the Xilinx Coregen tool to the appropriate COE file after extraction from the ZIP archive**.

Some COE files may have be to manually modified, f.ex. to change the SuperGarbage app that they load. This is described in the following section.

1.2.2 Note on time step / breaks for simulation

The top_tb.v testbench may pause before all simulation output has completed. Please run the simulator for longer (at least 300,000ns) until it has finished running and has dumped all output to the processor i/o channels.

1.2.3 Switching SuperGarbage apps (question #5)

To switch between different (numbered) SuperGarbage apps, only one file needs to be modified. The source/assembly/test1_i.coe file needs to have the last nibble of the first data line (0x0e000 by default) changed; 0 corresponds to app0, 1 to app1, etc.

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
0e000,
0b08a,
0b100,
0b180,
0e001,
0088a,
0020a,
0028a,
123ff,
12fff,
12fff,
0320b,
```

Then, just re-generate the m_imem ROM using ISE. The simulation should load and run the appropriate SuperGarbage app the next time it is simulated.

1.2.4 Files Included with this Report

The following list enumerates the directory structure of the auxiliary files included with this report

Table 1 - List of files included with this report

ise/	Contains the Xilinx ISE project folder, including generated files such as the imem and dmem cores.
Report/	Contains this report
Source/	Contains all source code for our project
Source/assembly	Contains assembly source code and generated COE files, including the assembler itself
Source/microcode	Contains the microcode source, and microcode assembler (uc_assembler.pl)
Source/primitives	Contains more Verilog source
Source/tb	Contains test bench files (typically non-synthesizable)

1.3 Contact

For questions or comments regarding the program, contact Tyler O'Neil the.tyler.oneil@gmail.com or Ivan Yulaev iyulaev@ucsd.edu.

2 SH17 Processor Implementation Improvement

2.1 Features Added

In this lab assignment, we add several features to our SH17 processor. We add

1. Additional pipelining for the mul/muli instruction. This decreases our cycle time significantly and results in an almost "free" performance gain.
2. Superscalar dual-issue execution – up to two instructions can be executed in parallel on each clock cycle, subject to some restrictions (described in section 2.2.2).
3. Moved the pipeline registers in order to more evenly balance our two stages.

Neither of these functions requires re-writing our reference programs, thus, we will measure the performance of our reference programs before and after adding these improvements, to determine the impact our improvements have upon our design. This simplifies the before/after comparison of processor changes; if we were to implement a "multi-core" CPU it is almost certain that we would have to re-write our benchmark software to see any performance gain.

2.2 Feature Implementation

2.2.1 Selective Instruction Pipelining

Pipelining the mul and muli instructions was very straight-forward. Inside of the ALU, we register the port 1 and 2 inputs, and send the register outputs to the multiplier. We also add a dedicated "multiply" output on the ALU and a corresponding input on the register file. Finally, we modify our microcode so that the multiply instructions result in a single stall, and so that the multiply input in selected on the ALU during the second multiply cycle. The illustration on the following page highlights the portions of our design that changed for the selective instruction pipelining. The impact on the rest of the design was quite minimal. Aside from the reduction in minimum clock period, the other instructions in our ISA did not have their execution affected at all.

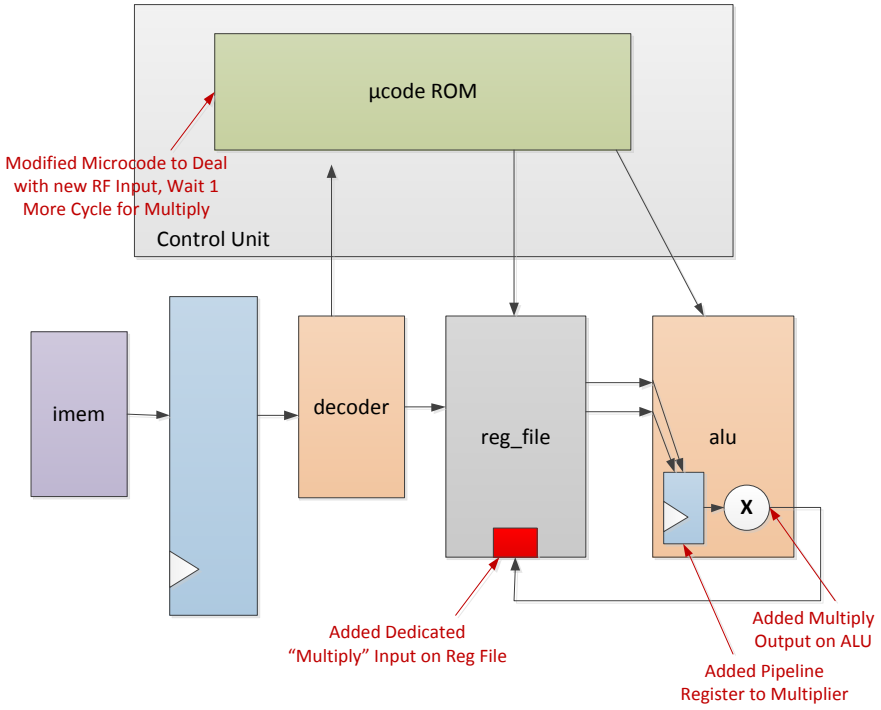

Figure 1 - Design Changes for Selective Pipelining Improvement

2.2.2 Superscalar Execution

The addition of superscalar dual-issue execution was quite involved, especially in comparison with the pipelining we described in section 2.2.1. We first move the decoder to before the pipeline register; this allows us to slightly decrease the clock period, while simplifying the implementation of the most critical part of our superscalar processor, the hazard detection unit. We added a duplicate imem, decoder, and ALU, and also doubled the width of our register file IO so that we could perform 4 register reads and two register writes on every cycle.

By doing the above, we have effectively added a second execution pipeline. The first execution pipeline, which remains connected to the IO channels and dmem, is termed the **master execution unit.** The second pipeline, which contains its own imem, decoder, and ALU, but lacks a dmem and IO channels, is termed the **slave execution unit**. The **master execution unit** always fetches the instruction at instruction memory address PC; the **slave execution unit** fetches the instruction at PC+1. Finally, we add a hazard detection unit, which will determine whether the master and/or the slave execute on a particular cycle.

The hazard detection unit is a block of logic that examines the instructions and their operands, and decides whether it is safe to execute the master and/or the slave. It follows a fairly complicated set of rules. Most of these rules stem from the WAW/WAR/RAW (potentially false) data dependencies, although some also come from the reduced function of the **slave execution unit**, as it cannot perform memory nor IO operations, nor is it connected to the PC generator logic block.

The rules that the hazard detection unit uses to determine whether the master and the slave are allowed to execute are (evaluated in order):

1. Slave never executes the cycle after a branch
2. Slave never executes when master has callpush, callpop - managing collisions was too complicated here
3. Slave never executes when master has halt - no reason to right?
4. Slave never executes when master has a branch
5. Slave does not execute if one of its operands is the destination for the master
6. Master does not execute if master and slave have same destination register, UNLESS the slave instruction is set - then the slave does not execute, but the master does. This was a side-effect management issue.
7. Slave can only execute comparisons, arithmetic and logical instructions (excluding mul, muli), mov, and set. It cannot do branches, memory accesses, or channel access.
8. If none of the above apply, then master executes and slave does not.

On a high level, our processor can now be described by the following block diagram:

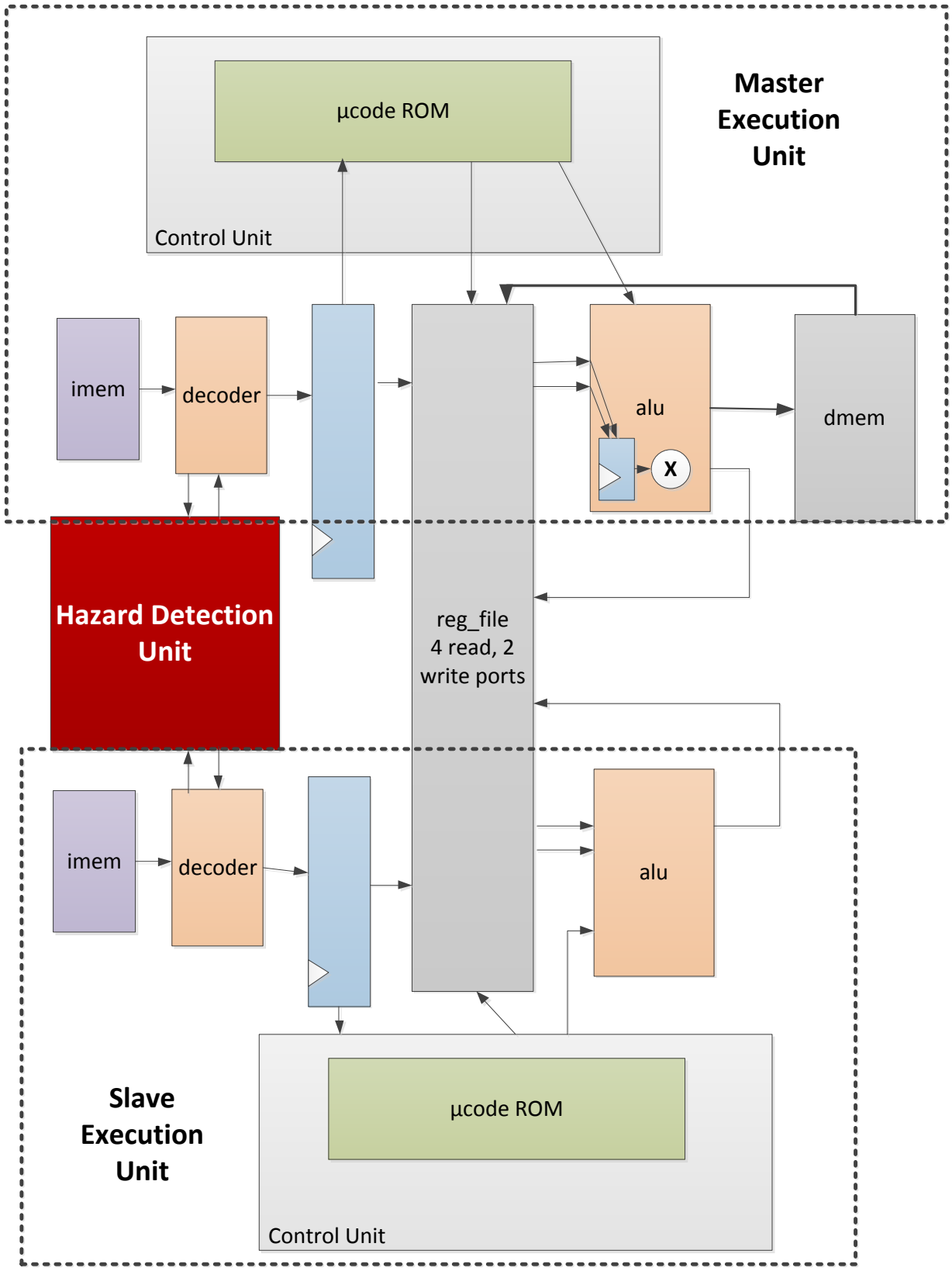

Figure 2 - Dual-Issue Processor Implementation Block Diagram

The master execution unit, with the exception of moving the decoder before the pipeline register, is essentially unchanged. However, we do modify the register file to increase the read/write port capability, and we add the slave execution unit. The hazard detection unit (block in **RED**) observes the instructions and their operands, and selectively replaces instructions with nops before they move down the pipeline, as necessary. It also determines whether to increment the PC by 1 or 2 for each cycle; if the slave does not execute its instruction, we must allow the master to execute it on the next cycle and so we cannot skip forward by 2 instructions in this case.

In terms of the execution of our processor, in the limiting case where the slave is never allowed to execute by the hazard detection unit, the processor function is essentially unchanged from our prior implementation. However, on those cycles when the slave does execute, we advance two instructions down the pipeline rather than 1. Also, the superscalar implementation will skip some instructions that will have no effect. Thus, the effective CPI of our processor will at worst be unchanged, and mostly likely will increase, especially if the compiler or assembler writer is aware of the dual-issue feature of our processor implementation.

2.3 Evaluation of Feature Addition

Implementing the mul/muli pipelining was quite trivial. Design and testing took roughly 1 hour; the work was straight-forward and implementation was simple. We saw a 32.5% reduction in minimum clock period as a result of this change. Furthermore, since none of our reference programs use the mul/muli instructions, the improvement in pipelining results in a 32.5% speed-up for their execution. The mul/muli pipelining was most definitely the low-hanging fruit for improving the performance of our processor with minimal effort on our part.

Implementing the dual-issue execution was quite a bit more involved, although considerably simpler than we had anticipated. The most difficult part was deciding on the rule set that we would use for our hazard detection unit, to determine whether we could execute one or two instructions on a given cycle. Once we did this, implementation was fairly straight-forward. To be sure, there was a good deal of rework required on our existing design, to fit in the slave execution unit and to translate the hazard detection rules into Verilog. On the whole implementation went rather smoothly and we were able to get Fibonacci and Supergarbage running on our dual-issue CPU without too much debugging.

The speed-up of the superscalar implementation is 3.8% from the increases pipelining of our design (due to decrease in clock period). Measuring the change in CPI due to the dual-issue architecture is somewhat more difficult; we decided to simply look at the cycle count for executing the Supergarbage example applications. *Note that this was done **without** re-writing the Supergarbage virtual machine to better take advantage of our dual-issue microarchitecture; sizable gains in performance could no doubt be extracted from a re-write of our assembly code.*

Our performance result for the Supergarbage benchmark applications was as below; the first “speedup” row indicates the speedup from improvement in CPI due to dual-issue execution, while the second is the total speed-up taking into account the decrease in clock period due to more aggressive pipelining.

Table 2 – Results of Performance Gain from Processor Modification

	Benchmark 0	**Benchmark 1**	**Benchmark 2**
# of cycles	454	5851	14341
Dyn. I Count	401	4809	11768
CPI	1.132169576	1.216677064	1.21864378
Speedup	1.107929515	1.115706717	1.114845548
Total Speedup	1.594778029	1.605972703	1.604733118

2.3.1 Impact of ISA on Processor Improvement Implementation

Our ISA was fairly easy to work with in making the improvements we desired. The instructions could be easily pipelined, and identifying data dependencies was relatively simple. The only problematic instruction was the **set** instruction; this instruction modifies the ri register, but does not overwrite it. If we have two consecutive instructions modifying the ri register, f.ex. a mov and then a set, we must execute both of them, in series. Thus, identifying data dependencies with the ri register is complicated somewhat, due to the function of the set instruction.