**EE480 Assignment 4: LNS Implementation**

TEAM32: Parker Householder, Ryan Parsley & CJ Vanderpool

Computer Engineering & Computer Science

Lexington, KY 40504

[paho224@g.uky.edu](mailto:paho224@g.uky.edu), [ryan.parsley@uky.edu](mailto:ryan.parsley@uky.edu), [cjva226@g.uky.edu](mailto:cjva226@g.uky.edu)

*Abstract*—Our goal in this project was to build a pipelined implementation of the of the multi-cycle design that we created in Assignment 2 while also incorporating LNS.

# AIK Specification

We first started this project by selecting which Logick instruction set we wanted to use for the project. After comparing between our files, we decided to go with CJ’s specification from Assignment 3 (A slightly modified version of Dr. Dietz’ ISA). This specification can be viewed under the file “logick.aik”.

# Definitions

After selecting and modifying our Logick.aik file, we started our pipelined implementation by first writing all of the define statements for our op codes like so:

**`define OPalad 3'b101**

As you can see we followed the format of OP<instruction name> followed by the 3-bit op code specified in our instruction set. Our next bit (the 4th bit) determines whether we were using the log instruction or non-log instruction. For the example above, the 4th bit would be 1 for “al” and 0 for “ad”. In other words when differentiating between log and non-log instructions, we used 1 as the 4th bit for log and 0 for non-log. We did this for most of the instructions except for si, li, jr, br, and sy which were all defined uniquely.

With all of our op codes defined, we then added our size related definitions for our processor to reference. OPCODE, LOGBIT, DEST, SRC, TSRC, and IMMED were all added based upon the same specified locations in our instruction set while REGSIZE, MEMSIZE, HALFWORD, and WORD were all added as constraints for our processor.

# Decode and ALU modules

Our decode module was built for the processor to be able to decode the current instruction and set the appropriate op code. We built this module with 2 inputs (ir and opin) and 2 outputs (regdst and opout) where if opin equals the op code for li, we specifically set regdst to 0 and our opout to to our `OPnop (16’b0). If it doesn’t, we then wrote a case statement to handle all of our extended op code instructions if the op code equals OPsy. Finally, for the decode module, if none of the other conditions apply, then we simply set the opout to equal the opin.

Our ALU module was built for the processor to handle all of our “ALU operations”. It works by simply taking in an op code and two inputs and then doing a case statement on the op code to set the appropriate output result while also setting the appropriate conditionals if necessary. For example, for the add instruction (ad) we wrote the following line:

**`OPad: begin result = in1 + in2; end**

# Processor module

# Testing

To test our processor, we added the given test bench to our Verilog file and wrote the assembly code seen in our “assembly.aik” file. Using our assembly code and our chosen AIK specification, we used Dr. Dietz’s assembler web interface to generate a corresponding .text segment (refer to the file “VMEM1.vmem”). After generating our VMEM1 file, we then used Dr. Dietz’s Verilog simulator by plugging in our source code (seen in “logick.v”), VMEM1, and “@0 \n 0” for VMEM0.

It was noted that Dr. Dietz mentioned the coverage analysis may be a little inaccurate so we instead decided to just look mainly at the Trace Browser section. This helped quite a bit as when we initially tested it we had several problems. We noticed that the ir was never actually getting set at first and the pc was never getting incremented. Eventually when we fixed these issues everything else seemed to be running fine after that. To ensure ourselves of this, we scrolled through checking to see if the pc count matched what should be in the ir based upon our assembly code.

Just as an example, when our newpc was at decimal value 1, the ir was set to 1000111100000001. This should indicate that the op code was a 4-bit value 8 in hexadecimal and that the register whose decimal value is 15 was used. Based on our Verilog define statements and AIK specifications, that instruction should be an li instruction on register u0. Knowing this, the last 8 bits represent a decimal value 1, so the full instruction should be li $u0, 1 which was correct. We repeated this process several times.

# Issues

Our