**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). In the course of this project, the encoding was modified even more as it was more easily changed than the code in “lnspipe.v”. 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

The pipeline described by the module “lnspipe.v” was built in three stages. Stage 1 is responsible for program flow and instruction decoding. Stage 2 contains the Register File as well as logic to perform sr and si. Implementing the shift logic in this module allowed for sr and si to be performed with no extra nop’s in between for interlock handling. Stage 3 contains the data memory (DMEM) and the ALU.

In stage 1, the program counter is changed based on what part of the program needs to be run next. This is implemented by a 4x1 multiplexor that is controlled from the stage one decoder in the “control” module. On every clock cycle, the current program location is latched on the output of the PC. The PC output is fed to the address input of the instruction memory (IMEM). The IMEM is a lookup table whose entries are the machine coded instructions of the program. These instructions are sent to the control to be decoded into control signals as well sent to the next stage to be used directly.

The instruction in stage 1 is latched into stage 2 on the clock cycle at the same time that stage one fetches the next instruction. The D, S, and T fields of the instruction hold the physical address of data inside the register file. The register file returns the data at the address given for S and T. when it receives a write enable signal(REGwe), it will write whatever it sees on its input(din) to the register at address D. The din input has 4 possible drivers that are selected between using another 4x1 multiplexor whose select input(REGinsel) is driven from the control module.

Stage 2 also contains the condition register. The register is implemented to latch its input value on a compare(co or cl). The output is sent directly to stage one to be used for branch decisions. Since the condition is only known when the instruction reaches stage 2, a nop after each compare is needed to avoid interlocks.

In stage 3, the ALU performs the operations specified in the instruction passed from stage 2 on the clock cycle. Whenever there is an ALU operation, there must always be a nop that follows since every ALU operation has the result stored in the register file. This is the same for loading from memory, but not for storing to memory. The ALU and the DMEM both are fed back to stage 2 to be written into the register file.

In order to implement the LNS, new logick was to be added for the log add instruction (al). all other log instruction can be implemented using existing circuitry. We did not find the need to add any more stages in order to implement the LNS. Instead, we determined what we needed to implement only the log add and what we already had implemented. By working with a few different configurations, we implemented the log add partially in stage 2 and in stage 3. In order to perform log add or subtract (which is really just log add with one negative operand), we needed to compute 1±2log(b)-log(a) with a lookup table, then add the result to log(a). This result is log(a)±log(b). We implemented the log(b)-log(a) part with a subtractor in stage 2. This was output to stage 3 where the lookup table latches the result on the clock cycle. In stage 3, the log add or lo subtract both look like an integer add with special cases, so the existing ALU is used to compute the final result, which is sent to the register file in stage 2.

On a power cycle, the processor begins running in an idle mode in which the Program Counter(PC) is initialized to 0. The instruction at memory location 0 will always be a system call, so the processor will halted initially. On a reset signal, the PC is set to 1, which marks the beginning of any program assembled with our encoding specification. From there, the program will increment as written.

# 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