

EEEE.521 Final Report Lab 12: ixwRISC Multicore

To: Dr. Patru

From: Isaac Witlin

Teaching Assistant: Jared Loewenthal

Date: Demonstrated: 12/8/17; Due: 12/11/17

Re: Final Report Lab 12

1 Specifications of the ixwRISC Processor

Specification:	ixwRISC:
I/O-P Mapping	Separate
Program and Data Memory Organization	Von-Neumann
Word Size	12-bit
Cache	One-Way Set Associative

Table 1: Specifications of the ixwRISC Processor

2 Non-pipelined Version: Lab 2 & 3

Taking the example code provided for Labs 2 and 3 as a baseline, The example was modified to support the specific operation of a 12-bit RISC with a Von-Neuman Architecture, 4-bit op-code, 4-bit Ri, and 4-bit Rj. Significant modification was required to implement the load and store functions as the example used a Harvard architecture where as the ixwRISC is Von-Neuman. Attention needed to be given to the clock cycles and whether its possible to read or write in a specific stage, making sure the program counter is incremented in the proper stages. Shift Right Arithmetic and Rotate Right Through Carry were replaced with behavioral logic because It difficult to understand the manually implemented logic provided.

3 Pipeline Version: Lab 6 & 7

Similar to the non-pipelined version, a hybrid between the structure provided in the Lab 6 example code and the code developed in Lab 3 was used to create a working pipeline ixwRISC processor. Much time was spent making sure the program counter was working properly with the modified instructions and the change in memory architecture. In Lab 6 a speed-up of 3.2x was observed when comparing it to the non pipelined version (2110ns to 662ns). Graph depicted in Section 7.

4 The Assembler: Lab 4 & 5

Python was used to create a rudimentary assembler for all ixwRISC521 instructions because Python has been known to easily manipulate strings. Using enums, dictionaries and line splitting the program can take in a text file and output memory initialization file. The pycharm IDE was used to create this program.

5 Cache: Lab 9

The cache architecture for the ixwRISC processor is one-way set associative with a 5-bit key 4-bit group and 3-bit address (16 blocks, 8 words in each block, a total of 128 memory locations in the cache). Extreme difficulty was found with stepping through, and understanding the provided code in lab 9, but this was mostly because it needed to be fully understood to correct and modify it. Before adapting the example code to the ixwRISC specification a hold variable that forces the cache to wait one more clock cycle before setting the done flag to high was added. This was done because it was evident that the done flag was set too early in some cases and would output the wrong data value for a third of a clock cycle before correcting itself. The testbench provided included tests that were only valid for two-way set associative caches, and caused a bit of confusion when checking for errors.

In the end the cache was adapted to the ixwRISC specifications and the code worked for Lab9. In lab 11 it was discovered that for some reason in simulation only the cache would fail and read the wrong values out of memory, but only once, duplicating the failed code segment fixed the issue in simulation. The same issue was evident in Lab 12. In addition, lab 12 has another cache related error that replicated itself in hardware as well, this issue was also able to be resolved by duplicating the effected code segment. More details on these issues are provided in the reports for Lab 11 and 12 submitted to the TA.

6 Multicore Handshake Protocol: Lab 12

To add more cores to the design much thought was put into how the cores will communicate with the CPU, eventually it was decided it was best to provide one master core and three slave cores, the master core will have an output hardwired to all the inputs of the slave cores, a three-bit data_ready output, and a three-bit data_send input (each tied to an I/O peripheral). Each bit of the data_ready and data_send registers were hardwired to the input and output of each slave core. Data coming into the master core is controlled by the 3-bit data_ready output the acts as a select line for a decoder that chooses which core the data_in the I/O peripheral reads. Data_ready and Data_send change their roles depending on who is sending and receiving, since the master core can only send or receive at any given time, the need for more control signals was unnecessary.

There is one master core and three slave cores. The data_ready signal is a three-bit signal that communicates to each core. (100) Sends a ready signal to slave core 1, (010) Sends a ready signal to slave core 2, (001) sends a ready signal to slave core 3. When the ready signal changes so do the master cores data input peripheral, it points to the core the ready signal was sent too. When a slave core receives a ready signal, it begins recording the word transmittined, data_send (either 100, 010, or 001 depending on which core is communicating) is then used by the slave core to tell the master core it is ready for the next value. Doing this would change the data_ready signal to low and the next data value will be loaded from memory. This process is then repeated. The role of data_ready and data_send flip when the slave transmits back to the master. This process is performed mostly through the developed assembly code. Each slave core will continue to be in a transmit state for the master core in case the data needs to be re-transmitted.

7 Benchmarks

7.1 Lab 2 vs Lab 6

The pipelined an non-pipelined manipulation only instruction labs were bench-marked as they shared the same memory initialization file. Results are shown in Figure 1. The time was recorded directly from the simulated waveform from the implemented test bench.

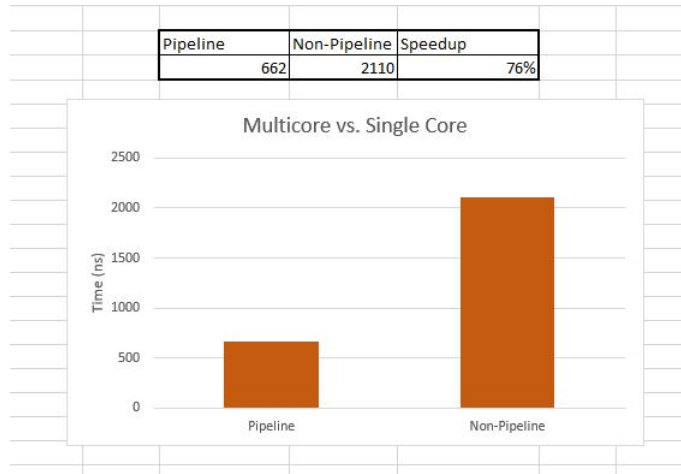


Figure 1: Benchmark for Pipeline vs Non-Pipeline

7.2 Lab 11 vs Lab 12

After multicore was implemented in Lab 12, it was bench-marked in Figure 2 with the code in Lab 11, both Labs input two 4x4 matrices, add and transpose the result. The matrices are input via switches and push buttons on the FPGA board, and the result matrices is output by LEDs on the FPGA board. The difference between lab 11 and 12 is that in lab 12, a master core received the data from the user via switches and push buttons, then transmits parts of the matrices to three different slave cores that perform their part of the addition and then the slave cores transmit back to the master core (transposing in the process) so the master core can display the result on the LEDs. The clock cycles were recorded by a performance counter I/O peripheral in both labs 11 and 12 that started after the data was input, and ended once the result is displayed on the LEDs. The peripheral result is shifted right 4 times so the value is readable on the same 8 LEDs the output is displayed on, this number can be multiplied by 16 to get a reasonably accurate clock cycle count.

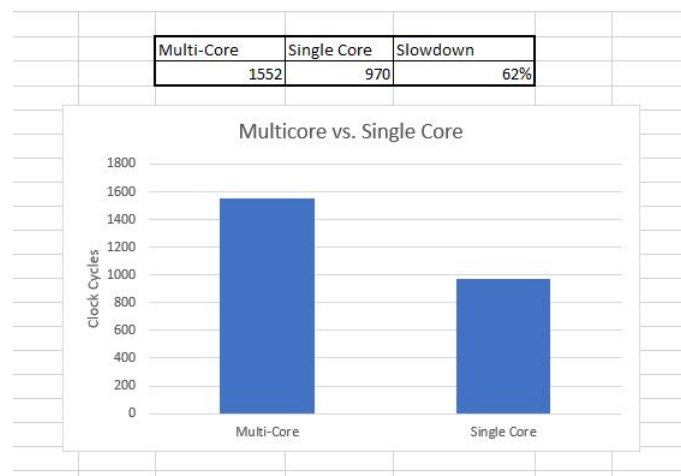


Figure 2: Benchmark for Multi-Core vs Single-Core

8 Discussion

The slowdown shown in Figure 2 is likely because the master core was only able to transmit or receive from one core at a time. The master core could not send and receive from the same core, or send to multiple cores. While this implementation is possible, time does not permit such a design to be implemented.

9 Appendix

9.1 Equipment/Software Used

Item	Tool
Verilog IDE	Quartus Prime Lite Edition 15.1.0
Simulator	ModelSim ALTERA STARTER EDITION 10.4b
Python IDE	PyCharm Community Edition 2017.2.3
FPGA	ALTERA Cyclone IV: EP4CE22I7C6N

Table 9.1 Equipment Used

9.2 References

[1] Lab Handouts and Presentation by Dr. Patru