ECE 437L Midterm Report

Course – ECE 437 L

David Larson
Pranjit Kalita

TA: John Skubic

October 16, 2015

1 Overview

Here in the midterm report, we compare between the single-cycle and pipelined processor designs implemented in labs 3-4 and 5-7 respectively. Philosophically, the reason for using a pipelined processor design over single cycle is to increment the clock speed, however there is no perceivable decrease in instruction latency (for all practical purposes the addition of additional hardware such as latches introduce delays that lead to an increase in instruction latency). Nevertheless, the throughput of the overall program increases, which is the rate of completion of instructions during the course of execution of the program. The deeper the pipeline the higher the clock speed. Introduction of the pipelined processor also could incur an increase in CPI due to the insertion of wasted clock cycles due to the introduction of noops and stalls, which occur due to missed prediction of branches, jumps, and/or during load/store operations in order to not let valuable instructions be overwritten. Finally, with regards to single cycle, since each instruction takes one clock cycle to finish, therefore the inefficiency roots from the fact that even the shortest latency instruction would have to wait for the longest latency instruction (load word) to finish, therefore in a program with very minimal loads the program would still take a long time to finish execution.

For the basis of comparison between the single cycle processor and pipelined processor, we compared between the clock frequencies (expected and determined to be higher for pipelined), average CPI (higher for pipeline), instruction latency (expected to be similar but larger in pipelined processor due to h/w delays), the performances of each designs in MIPS rating and the FPGA resources utilized by each design. The actual piece of assembly code that we tested on was the mergesort .asm file, because of its abundance of branch, jump and hazard introduction structure that could test all components of our forwarding logic as well as branch and jump stalls and corresponding noops.

2 Processor Design

(A) SINGLE CYCLE –

As attached in the figure 1 that follows, our single cycle processor design is essentially divided into the following parts – program counter (PC) combinational logic, the register file, ALU logic, instruction/data read and data write memory interfacing with the correct control signals, the Control Unit block, and the final writeback stage to the register that incorporates two control signals (lui and jal) to decide what needs to be written back to the register file.

The PC logic consists of an adder that increments PC by 4; it also contains the jump address and branch address calculation logic through a series of muxes. The register file is standard with several read and write ports, the ALU unit takes two operands and performs the requisite ALU op based on the opcode. The Control Unit determines what operation the ALU must execute, the destination register, whether there is a memory operation taking place, the immediate extension, and among other things whether there would be a branch or jump.

(B) PIPELINE PROCESSOR –

As attached in the figure 2 that follows, the pipelined processor is divided into the following five stages via four latches – fetching, decode, execute, memory and writeback. The corresponding four latches separating the five stages are fetch/decode, decode/execute, execute/memory and memory/writeback. Without going into too much low level implementation details, we will try to give a general idea of important critical stages of the pipeline unique to our implementation.

The branch address calculation logic takes place in the MEM stage, along with the jump address calculation. This means that should there be a misprediction for either branch or jump, we will have to introduce a series of noops (three to be precise) starting from the fetch/decode latch. Our forwarding logic was a little more involved as we went about the debugging stages, wherein we had to write separate logic for store words and load words followed by stores or loads.

Whenever we had a load/store followed by another load/store, we included a stall for each of the previous pipes. Also, we had to make the stall of the MEM latch dependent on the halt signal as it was causing an issue with the FPGA when the halt signal was not being generated. Other than these specific logic additions, we had independent pipe enabler logic for the different pipes of the pipeline. For eg: the fetch/decode stage enabler depended only on ihit, whereas the rest depended on ihit or dhit.

We calculated in the decode stage the extender and shift logic, which we propagated to the execute stage in order to provide the ALU with the correct set of operands. The one notable difference from the single cycle was the placement of the lui signal from the control signal during the writeback stage. Here we just overrode the ALU output with the calculated lui operation result, and wrote it directly to the register write port, moving on to the next program counter logic. We had initially implemented this in the writeback stage, but noticed that it created issues by overriding certain instructions.

As we continued with the debugging process and checking the processor with variable RAM latency, we had to make a few tweaks, such as making the flush logic dependent on the pipeline enabler signals, but nothing major needed to be changed as we shifted from zero to variable latency.

Other than the aforementioned specific implementation details, the rest of the pipeline and forwarding logic was standard for our implementation.

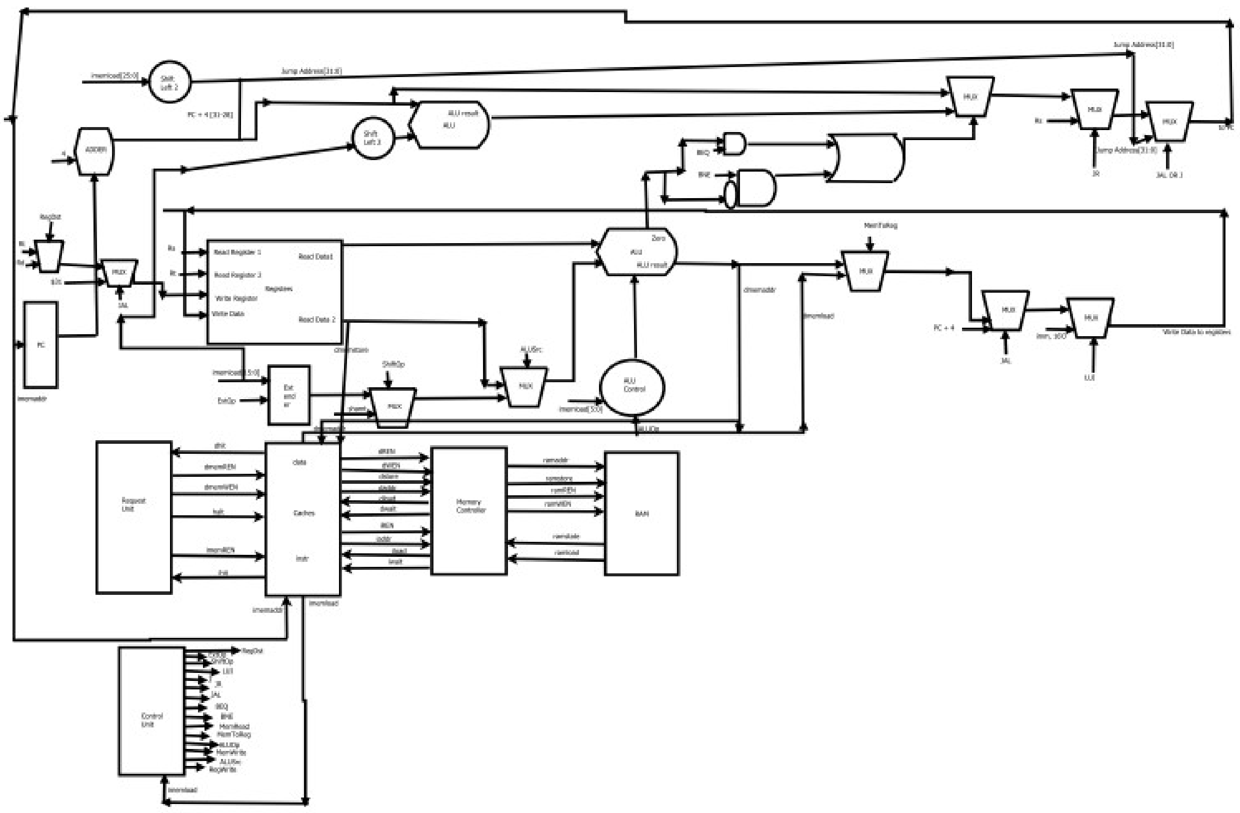

Figure 1 - Single Cycle RTL

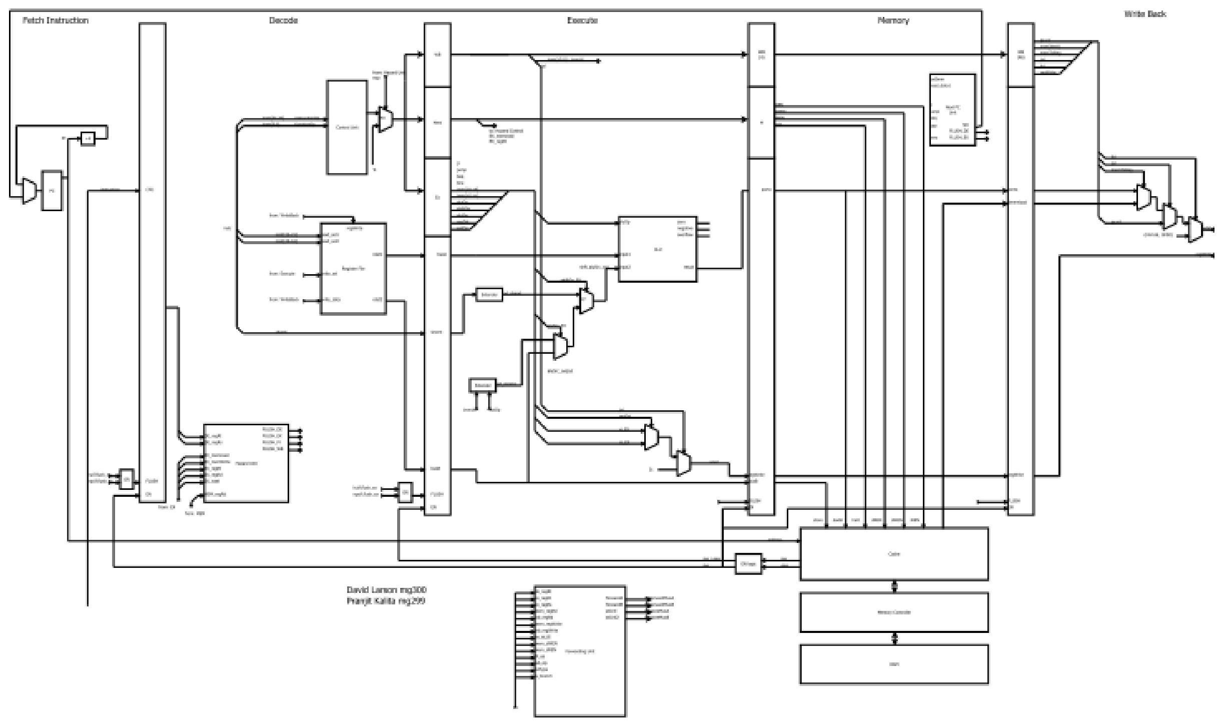

Figure 1 - Pipelined Processor RTL

3 Results

	Single Cycle	Pipelined
Frequency	32.9 MHz	57.11 MHz
Average CPI (CPU Cycles / # of instructions)	1	1.78
Latency (5 * Period)	30.39 ns	87.5 ns
Performance (MIPS) (Freq / CPI * 10^6)	32.9	32.08
FPGA Resources	Total combinational: 2872 Logic registers: 1311	Total combinational: 3149 Logic registers: 1687

4 Conclusions

As expected, with the pipelined implementation, the average CPI increased during mergesort, primarily due to the numerous branch and jump conditions, and the prevalence of hazards. Thus, the introduction of noops and stalls would account for the increase in CPI compared to its single cycle counterpart. However, we also see an increase in the clock speed, characterized in the table above by the maximum frequency arrived at for the pipelined processor. Increasing clock speed was the basic reason for choosing pipelining over single cycle as an improved processor design technology.

The interplay between the increase in clock speed and increase in the average CPI is characterized by the fact that despite the fact that our clock speed did not increase by 5 times compared to single cycle as is suggested by the ideal case to offset the 78% increase in average latency, we were still able to arrive at a performance (measured by MIPS rating) almost same to the single cycle. The probable causes for not arriving at the expected ~5 times increase in clock speed could be the limitations of our design, the lack of a one- or two-bit counter based branch prediction logic in the ID stage in order to reduce the penalties due to inaccurate branch prediction; rather we imposed a heavy penalty by resolving branches in the MEM stage by introducing 3 bubbles into the pipeline. Also, the significant increase in instruction latency could also be attributed to the less-than-expected increase in clock speed, coupled with additional hardware-induced delays such as those due to latches, muxes, etc. All in all, we can see to a certain extent the usefulness in using pipeline over single cycle through an increase in clock speed, and the almost similar MIPS performance rating, but there is no doubt that this design could be much improved upon. Lastly, the lesser the number of branches and hazards, the higher the performance rating of our pipelined processor. Therefore, we are currently limited by our rather less-than-efficient branch prediction logic.