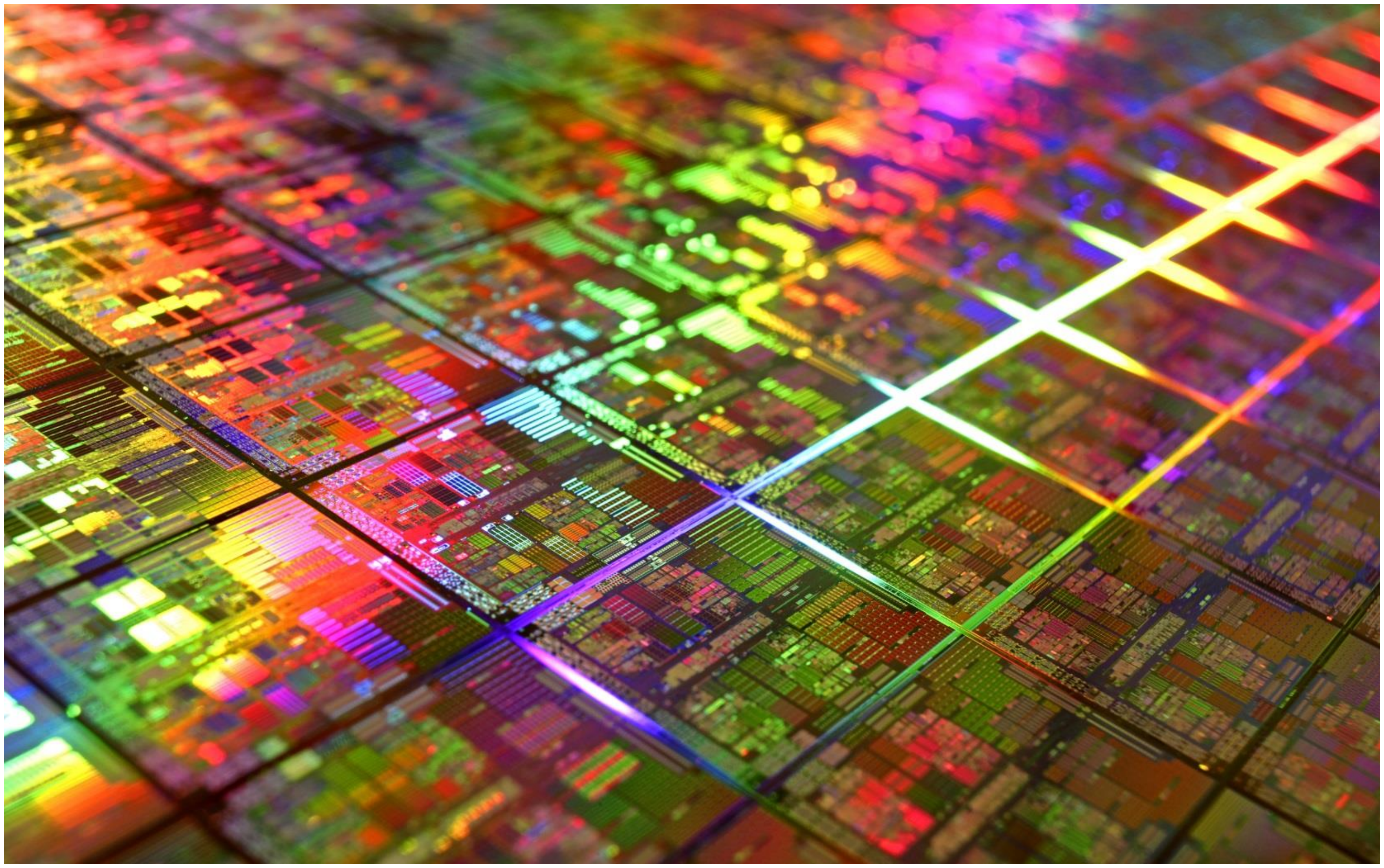
ECE/CS 552
Intro to Computer Architecture
Final Report: Five Stage Pipelined Processor



Jake Truelove and Emad Sadeghi

1 Introduction

This five stage pipelined processor was built as a final project for ECE/CS 552 Introduction to Computer Architecture. The processor was built in two phases, during the first phase a fully functional five stage pipelined processor was developed and tested with a total of 15 instructions, of those, 8 were arithmetic instructions, 4 were load/store instructions, and 3 were control instructions. During the second phase a direct mapped instruction cache was added which interfaced with a unified memory via a 2 word bus. Each block in the I-Cache was 4 words long therefore, an additional state machine was added to control the interaction of the I-Cache with the unified memory. During the second phase of the project full data forwarding was also implemented into the processor. This was not a required aspect of the project, it was instead an extra feature. Adding this extra feature to our processor design has had great improvement in the number of cycles per instructions of several test programs; for instance, in the LwStall and Branch, we calculated about 30% improvement (with CPI as our relative metric) compared to the processor without data forwarding. Therefore, the implemented full data forwarding feature has certainly optimized the processor to a good extent especially for the test programs which included many data hazards. More thorough analysis of the CPI improvement between both processors can be seen in Table 1.

2 Design Methodology

It was important for us to fully understand what we were trying to build before we began writing any code. We started this project by drawing block diagrams of how instructions would flow through the five cycle data path, our diagrams included all signals that we thought we would need to accomplish each instruction. Once we felt we understood the structure of what we were trying to build and how the actual hardware would accomplish the task, we set out to code everything in Verilog.

After some debate, we decided to split our pipeline into several modules as follows: Datapath, Control, ALU, Register File, Instruction Memory, Data Memory, and Full Processor, I-Cache Unified Memory, Control-Mem State Machine. We decided to combine all pipes into one data path because we thought it would be easier to implement in such a manner. We thought that trying to get every instruction working for one pipe of the pipeline would be more difficult than getting one instruction working in five stages. We believed that by building our data path one instruction at a time we would face less design errors and complications.

We built the processor by first designing the ALU. Once we had this module built and tested every instruction, we began designing the Datapath in conjunction with the control unit. We decided on a method of development in which one instruction was implemented at a time, with its necessary control signals, through all 5 stages of the pipelined Datapath. After implementing an instruction, the instruction was fully tested for correct function before moving on to implement the next instruction. By designing in this manner, we were able to keep the design process relatively simple by focusing on one specific well designed task at a time. Once each instruction was implemented, we tested the processor with more complete programs.

After verification of a fully functional five stage pipelined processor with all 15 instructions fully functional, the team added a direct mapped 8 page I-Cache with 4 word pages (an instruction is a single word). This I-Cache has one read and one write port; each of which are 4 words. The I-Cache interfaces with a Unified Memory module, this module features a 2 word read port. Due to the mismatch in size of the I-Cache write port and Unified Memory read port, a state machine was developed in order to control the read and writes between the two modules.

Once a fully functional I-Cache was developed and tested, full data forwarding was implemented for the processor. Data forwarding was added in stages as well, starting with ALU instructions. After ALU data forwarding was implemented and tested, similar logic was used to add data forwarding to load and store instructions as well as control instructions.

3 Methodology for Testing Verification

When designing complicated computer architecture such as a five stage pipelined processor with an I-Cache and data forwarding one of the most important parts of the development process is incremental testing and validation. When designing this processor we followed a simple procedure: divide the problem into smaller pieces, next design a solution to a single small piece of the problem, and finally test the design. As mentioned above, we started building the processor by building the ALU unit. After implementing the functionality for every instruction into the ALU we created a test bench that fed the inputs with data and signaled various commands to the ALU. Since the ALU is fairly simple with few signals, it was fairly easy to verify correctness simply by looking at the waveform of the signals. After adding many operations to the ALU test bench and verifying the correctness of every single one we began testing our implementation of the datapath.

Testing the datapath was done in two phases. The first testing was done in conjunction with development. After adding functionality for each new instruction, we thoroughly tested the instruction. We created a test bench for each instruction in which the opcode with operands were inputted into the processor. We then examined the waveform checking important aspects of the design such as control signals, number of clock cycles, stalls due to data hazards, and output correctness. After verifying the correctness of the single instruction, we began implementation of the next instruction. By beginning implementation of the datapath with LLB and LHB instructions, we could use actual register values when testing each of the next instructions that we implemented. After implementing and testing several of the instructions individually, we began making slightly more complicated test benches with several instructions. Eventually we had every instruction built into the processor and tested individually, at this point we began the second phase of testing.

The second phase of testing was done at a much higher level than the first phase. Instead of relying mostly on waveforms, as in the first phase, we built a test bench that outputted any change in register values, condition flags, PC value, instruction in the decode stage, and clock number to a text file. At the end of the text file there was also a print out of the final values of each register and a short confirmation or denial of correctness. This test bench was used in conjunction with full programs containing many instructions in order to show very quickly that the processor preformed correctly. By examining the statement at the end of the file confirming or denying the functionality, it was very easy to pass a test and move on without looking at any waveforms. If the test failed we could examine the register outputs in the same text file and see exactly where the error was. Once we found which register was wrong and during what clock cycle it was wrong at, we could examine the waveform at a particular point and figure out why the instruction was not preforming correctly very quickly.

4 Test benches

The test bench files are named "*test name*"_tb.sv, and can be used as a way of verifying the functionality of our code. They are attached in the zip file submitted. We have converted each test's assembly code instructions to its corresponding binary. To run the simulation, one needs to rename the instruction hex file to be read into memory in unifed_mem.v. Once that is done, compilation and simulation of the test bench can be proceeded. The test bench redirects the monitor output to a text file, output_"*test name*".txt in the same directory as the test bench "*test name*"_tb.sv file. In each cycle, the outputs to monitor are the PC, cycle #, any N, Z, V value that was changed in that cycle, and any register value that was changed during the cycle.

5 Full Data Forwarding Optimization

Speed of a processor is one of the main characteristics that drives the sales and success of individual processors in industry. After building our five stage pipeline processor, we felt that its performance was less than adequate even if it was simply built for educational purposes. We wanted to add something that would drastically improve the performance of our processor and thought that the best way to do this was to add full data forwarding to the processor. Several of the test programs that we ran had CPI of close to 6 on our initial design. A CPI of 6 is very bad, and after some examination of the tests we found that much of the wasted clock cycles were coming from data hazards which caused following instructions to wait and do nothing. By implementing full data forwarding at every pipeline from any instruction that was in the process of writing to a register we were able to improve CPI of several programs drastically. For example, the LwStall had a CPI of 5.8 when running on our first processor implementation due to a large number of its instructions needing to stall from data hazards. When running that same program on our improved processor with data forwarding we were able to get a CPI of 4.0! That is a 30.7% improvement in performance. An improvement of 30% is equivalent to the improvements made in one or two processor generations in industry, something that could take a company like *Intel* or *Nvidia* several years. Obviously in industry, processors are thousands of times more complicated and many times more difficult to innovate on, however we still believe that our data forwarding implementation accomplished exactly what we were hoping it to, improve speed and therefore performance. We knew that if we could implement data forwarding correctly we would improve our processors speed, however we never thought our optimization would improve the design by 30%!

Table 1: CPI Analysis of the Processor pre/post Optimization

Test Program	**Number of Cycles per Program**		**# of Instructions/ Program**	**I-Cache CPI**	**I-Cache with Data Forwarding CPI**	**Relative Improvement (% CPI)**
	I-Cache	**I-Cache with Data Forwarding**				
BasicOp	76	65	17	4.411	3.764	14.668 %
Branch	97	72	17	5.647	4.176	26.045 %
Control	90	63	16	5.562	3.875	30.331 %
Data Dependence	59	43	11	5.272	3.811	27.712 %
LwStall	53	37	9	5.778	4.000	30.772 %
Loop	2600	2592	1542	1.685	1.680	0.2974 %
Average CPI Over All Test Programs				*1.841	*1.776	*3.531 %

* These values are distorted since the Loop test program is the dominating test program due to the large number of Instructions executed by it.

6 Accomplishments

We have successfully built a multi-cycle five stage pipelined processor with data hazard avoidance, delayed branching, direct mapped I-cache, and full data forwarding. After many tests and alterations, we now pass every test bench that has been provided to us. This can be proven by running each of our six test benches for every separate test provided. Each test bench produces a txt file that tracks the completion of each program. The txt file shows the progression of each clock cycle by showing the instruction that is currently entering the IFID stage and the PC value at each cycle. It also prints any change to a register value at the clock cycle for which it changes. By showing each of these pieces of information in chronological order it is very easy to understand how the program is executing in our pipeline at a high level. We have also included files of each test that show what the value of every register is after every cycle. To conclude, we first implemented and tested, quite extensively, a simple processor with the mentioned 15 instructions with data hazards avoidance. After which, we added a direct mapped I-cache to our design. Finally, to optimize our design, we added full data forwarding feature; thereby, improving our CPI performance by up to 30% for the test programs given.

Appendix

Important Note 1: The summary sheet is included in the zip file, we do not include a single excel file with the print out of every register after every clock cycle in one document, instead they are separated into files named "output_allReg_testname.txt" for each respective test.

Important Note 2: In order for simulation to work properly it is best to change all files to System Verilog format in properties.

Stage 1 (without I-cache): Area of Processor: 1761.001264

Stage 2 (with I-cache): Area of Processor: 2725.909256

Included in Zip:

Area Report:

1. Full_processor_area.txt <-- The synthesized area of processor

Verilog Files:

2. Full_processor.v
3. Control.v – in properties change to System Verilog
4. Instr_mem.v
5. reg_file.v
6. reg_file_tb.v
7. Reg16.v
8. datapath.sv
9. ALU.v
10. Cntrl_Mem_SM.sv
11. UnifiedMem.sv
12. cache.v
13. datamem.v
14. Project.mpf

Testbench Files:

15. BasicOp_tb.sv
16. Loop_tb.sv
17. Control_tb.sv
18. DataDependence_tb.sv
19. LwStall_tb.sv
20. Branch_tb.sv
21. ALU_tb.sv

Output Files:

22. output_basicOp.txt
23. output_branch.txt

24. output_control.txt
25. output_DataDependence.txt
26. output_Loop.txt
27. output_Lw_Stall.txt