LAB 2 REPORT: PIPELINED PROCESSOR

Luke Puchalla [lcp77]
Eldor Bekpulatov [eb654]
Shivangi Gambhir [sg2439]

Luke Puchalla (RTL Architect, Design & Verification):

- Implemented functionality and wrote tests for the following instructions:
    - Jump instructions
    - Memory instructions
- Designed and wrote tests for different bypass paths for the alternate design
- Wrote the testing section of this report
- Contributed to the baseline implementation section

Eldor Bekpulatov (RTL Design & Verification):

- Implemented functionality and wrote tests for the following instructions:
    - Register-Immediate Arithmetic and Logical Instructions
- Helped design, troubleshoot, and carry out the test cases for alternate design
- Put in tables and charts depicting the performance of both the designs
- Wrote the introduction and baseline design sections of this report
- Contributed to the baseline implementation section

Shivangi Gambhir (RTL Design & Verification):

- Implemented functionality and wrote tests for the following instructions:
    - Register-Register Arithmetic and Logical Instructions
    - Branch instructions
    - Integrated Iterative multiplier with the pipelined processor
- Designed the fully bypassed pipelined processor for the alternative design
- Wrote the Alternative Design and Evaluation section of this report
- Created tables and charts depicting the performance of both the designs
- Contributed to the baseline implementation section

INTRODUCTION

In the second lab, we designed two implementations of 5 staged pipelined processors: a baseline design that makes use of stalling, and an alternate design that uses bypassing in order to avoid various hazards raised by pipelining. Some of the hazards raised in pipelining include data and control hazards. The data hazards are raised when an instruction is dependent on another instruction's completion. The baseline design solves data hazards by simply by (stalling) allowing the dependent instructions wait for the independent instructions to complete. Since, this method is clearly inefficient, the alternative design offers a performance optimization by allowing the data calculated in independent instructions to be forwarded down the pipeline for the dependent instructions to be executed without necessarily waiting for completion of the independent instruction. The control hazards are raised when the CPU realizes that it has loaded the wrong instructions and needs to start executing different instructions. Control hazards are primarily resolved with squashing (invalidating loaded instructions and loading new ones). Both designs incorporate very similar logic/signals to handle such hazards. This lab also requires integrating the multiplier module developed in the previous lab. This lab is part of a series of four labs.

The implementation and evaluation of both designs gave us a fair idea of how the CPI (Cycles per Instruction) got optimized and reduced. We saw an **improvement of about arund 50%** in the total number of cycles per instruction taken when a specific test case was run on both the base and alt design. The dataset for the test cases consisted of masking the lower order bits of the input operands and carrying out a multiplication operation to test the best case improvement. We observed an increase in the cycles per instruction using the alt design as more operations were done in a single cycle. With two additional muxes and a synthesizable combinational block in the alt design and an almost 2x improvement in CPI, it gives a good area-time tradeoff between the two. This increased performance comes at the cost of hardware (space) and energy usage.

BASELINE DESIGN

We developed the baseline design of 5 stage pipelined processors to implement stalling and squashing as means to avoid any control and data hazards. Use of stalling and squashing ensured the correctness of our program. This would not give us the CPI we wanted, but it was more important to have our design tested for correctness before considering performance.
Development of baseline design involved all three members to divide up instructions and having each be responsible for full implementation of the instruction. We decided to approach building our baseline design using incremental method. Each member would incrementally add instructions and fully implement them. Each member was responsible for including components in the datapath, incorporating its control signals properly and documenting new ones, and testing the generic and edge cases with multiple and no delays. Last requirement for full implementation of an instruction was that it shouldn't cause previously implemented instructions to fail. So, with each additional case, all previously implemented instructions would be tested. Some instructions would later have to have their test cases increased to adapt to the alternate design requirements.

We started working on our baseline design by adding register-register instructions, instructions that required two source registers. The reason being, they simply required minimal additional components to the datapath. **Figure 2.1** shows the initial datapath that was given to us. In order to complete all register-register instructions we needed to update the Arithmetic Logic Unit (ALU) with appropriate operations such as add, subtract, shift etc. and also add control signals that would correspond to the operations defined in the ALU. As an example, if the instruction was subtract, our control unit would have to decode that and set appropriate signals to enable both register to enter ALU and also signal ALU that the operation is subtraction. Once we implemented appropriate arithmetic and logical operations and included appropriate signals in our control logic, we were immediately able to write corresponding test cases and run a simulation on our design. Next, natural choice for follow up instructions were the register-immediate instructions. They required a completion of immediate generation unit, and a path to allow the data to flow from imm-generation unit to ALU. This was solved with inclusion of multiplexers in the Decode stage. **Figure 2.2** shows all the additional data paths and signals highlighted that enabled our pipelined diagram to execute all of the arithmetic and logical operations. After this, we targeted on to integrate the integrative multiplier we developed in Lab 1. A major part of modification was needed in the control unit where stalling signals had to be added to take care of the integration. Once we concluded all of our arithmetic and logical instructions were adequate, we moved on to implementing memory instructions and other control instructions such as branching, and jumping. These instructions required a little more attention to detail when implementing, because our control unit had to be modified to allow for squashing of instructions based on control hazards raised by branches and jumps. An additional path was implemented to allow the JAL instruction to be identified at the Decode stage. **Figure 2.3** highlights the additional components that allowed for execution of instructions such as multiplication, jumps and branches. At this point we had fully implemented baseline design, and tested each instruction independently. Lastly, we ran the benchmark and with the benchmark and our custom tests passing, we were confident with our baseline and moved onto implementing the alternate design.

ALTERNATIVE DESIGN

We developed the alternative design to implement hardware bypassing/forwarding by taking baseline design as the point of reference which we developed earlier. Hardware bypassing is a technique where the hardware is modified to allow values to be sent from an earlier instruction at the back of the pipeline to a later instruction which is present in the front of the pipeline before the earlier one has left the pipeline. This technique can be used to solve data hazards by optimizing the cycles per instruction as compared to only stalling which we have used for the baseline design. We can see a more real-time scenario, depicted in **Figure 3.1**, where a pipeline diagram indicating No Bypassing (Baseline design) and Full Bypassing (Alternative Design) is shown. In this figure, RAW dependencies are shown in Red and Blue colour and stalls and bypasses have been highlighted.

Keeping in mind the concepts of bypassing, we have implemented bypass paths in the datapath controlled by bypass signals from the control unit.
We followed an incremental approach for designing the datapath and control unit for alternative design starting with partial bypassing i.e. a single bypass path from the X stage to D stage for rs1 and rs2 as indicated in **Figure 3.2**
For the datapath, we added a 2:1 multiplexer on both rs1 and rs2 paths controlled by select signals depending on bypass signals from control unit. This mux selects between data decoded in the D stage and data bypassed from X to D stage. These bypass signals are dependent on the read/write address, instruction validity and read/write operation.
After testing a test scenario, we noticed an expected change b/w baseline design and alternative design as can be seen in the traces in **Figure 3.3** and **Figure 3.4** respectively. After thoroughly verifying a single bypass path, we extending our logic to a fully bypassed processor.
Also, note that no stalling signal is required to be generated from X to D stage since our test case is a non-load use.

Figure 3.5 shows a fully bypassed datapath consisting of two 4:1 multiplexers controlled by the same bypass select signals. The muxes choose between bypass data from X,M,W stage and data decoded from message from memory in D stage. Individual bypass signals are written very similar to the one written for a partial bypassed processor. However, there is a slight modification for the X stage in terms of stalling and bypass signal. Consider the scenario in **Figure 3.6**. In this case, we see the need of stalling signal in X when load word is used. This is the only scenario where we need to stall in a fully bypassed structure.

One important thing to notice here is that when back to back instructions are implemented as in **Figure 3.7**, there will bypass signals set from both M to D and X to D like in this case. In such an ambiguous scenario, we must make sure that we consider the bypass signal of the latest stage i.e. X to D as it has the most fresh data. Such a logic has been incorporated inside the control unit which takes care of such scenarios and gives preference to the stage later in the pipeline.

Theoretically, we notice that a fully bypassed processor helps optimize the cycles per instruction as opposed to a non-bypassed structure as we will be wasting a lot of cycles just waiting in a particular stage. After simulating both the designs i.e. the baseline and alternative, we can see this theory fall correctly. This can be realised most in terms of RAW i.e. Read After Write data hazards. To implement this, we just need two multiplexers in the datapath and a few case statements in the control unit for the bypassing logic. If we compare the hardware requirement, complexity of the bypassing logic and CPI optimization, it is a win win situation for a fully bypassed pipelined processor.

TESTING STRATEGY

In this lab, we chose to utilize a workflow in which team members would implement certain instructions, and then write the tests for those instructions as well. We felt this was a more logical approach to completing this lab, since whoever implemented a specific instruction would have a better idea of what to test for (edge cases, testing all parts of the hardware, etc.). This also allowed all three members to be involved in the implementation of the processor.

Our testing strategy consists of unit test files (for different units/groups of test, as specified in **Figure 4.1**). Within the unit test files, we wrote directed tests for different instructions within or aspects about that "unit." For nearly all instructions, we wrote value, random value, and random delay tests. Our goal was to test as many instances of instruction sequences as possible, and to fully test all pathways and control functions in the processor. This means testing different types of hazards that should be stalled in the baseline but often times forwarded using bypass logic in the alternate design. Upon completion of our thorough testbed, we are confident in the functionality of both our baseline and alternate designs.

There is a huge variety of directed tests to be discussed. First, there are the source and destination "dep" tests (as labeled in the files). These tests vary the number of "nops" between different parts of the assembly test code (e.g. between loading a value into a register using `csrr` and using it in the instruction-under-test, or between the instruction-under-test and checking the value with `csrw`). The purpose of these tests is to check if the stalling logic is working correctly in the control unit, by testing instances where the pipeline should and should not stall, and stall from different stages. We wrote value tests, to explicitly check the fidelity of the datapath module moving data through the pipeline. These tests are essentially the same as the "dep" tests but with no "nops." Finally, we wrote random value and random delay tests for nearly all units.

Now, we'll examine some specific test cases for certain instructions, starting with the jump instructions. For JAL, we tested the basic functionality, and its functionality when it is embedded in a real instruction sequence, as tested with a complicated multi-jump test. Here, we also notice an obvious corner case of a branch followed by a JAL. This sequence is special because the branch is "decided" at the end of the X stage, but the JAL is resolved at the end of the D stage. If the branch is taken, the JAL should not cause the program to jump, even though it is resolved at the same time as the branch. Testing and verifying this case was critical for our confidence in the PC mux select logic. Similar tests were also written for JALR. We also have to verify the correct handling of control hazards (i.e. are programs executing in the right order, and is the squashing logic correct). The above branch-followed-by-JAL test is an example of this. Inherently, the branch tests all verify this ability. Finally, we need to address pipeline hazards, specifically RAW data hazards. One example of such a case is a load-word followed by a store-word whose address register is the same as the load-word's destination register. This tests the load-use stalling logic. We also already tested for these kinds of hazards in the value tests that included zero nops (for example, a sequence of add instructions with RAW hazards).

After implementing the alternate design, we would expect our baseline tests to pass. However, since we want to ensure that the bypass logic is working correctly, we created tests that purposely failed at the end of the test so that we could manually monitor the trace and confirm that (when necessary) data was bypassed and the pipeline was NOT stalled. We wrote tests for all bypass paths (X to D, M to D, and W to D).

In conclusion, we have written a variety of directed tests (for specific instructions, and specific common control hazards and datapath hazards) within larger unit test files. These files test all datapath units with random values and random delays, while simultaneously testing the control unit's ability to handle hazards. The rr and rimm tests specifically test the ALU functionality, as well as control logic. The branch tests verify the control logic as well as the branch flags of the ALU. The mem tests verify control logic with load and store, and test the correct functioning of the data memory interface in the processor. The jump tests verify control logic and components of the datapath. Finally, we designed tests for each bypass path after completing the alternate design, and manually tested and checked for the correct number of nops (if any) in such cases. With our testbed, we are confident in the correctness of our designs.

EVALUATION

As we know from the lectures that potential speedup is proportional to the number of pipeline stages. As the number of pipeline stages increases, the stalling logic gets worse and the processor has to wait for a very long time. This is where we can take advantage of bypassing where the processor doesn't have to wait and we can get the data as soon as it's ready in any intermediate stage.

The alternative design is better than the baseline design in terms of performance as the cycles per instruction is very well optimized for the former one. An analysis of the same is tabulated in **Table 5.1**.

Figure 5.1 pictorially represents a comparison of CPI across designs i.e. functional level, baseline design and alternative design where it can be clearly seen that the alternative design always falls below the functional and baseline showing a CPI optimization for every test case.

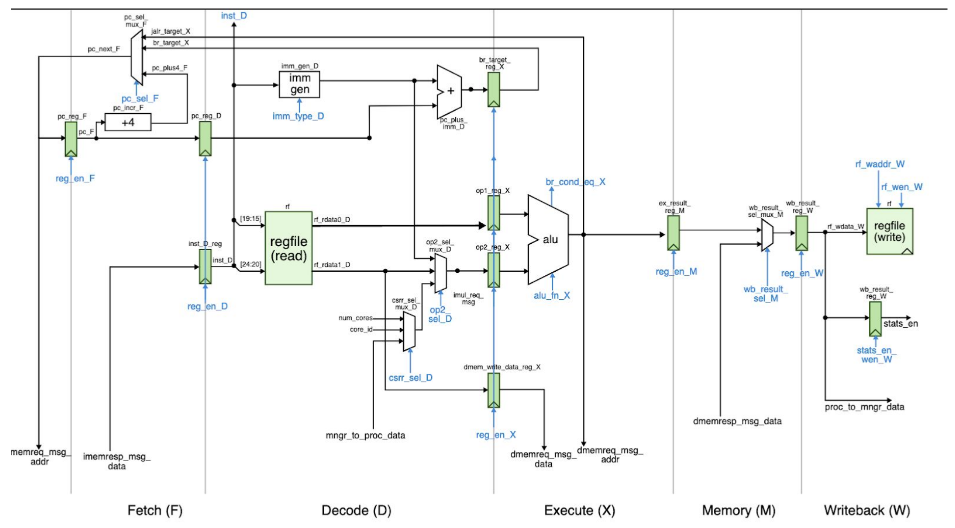
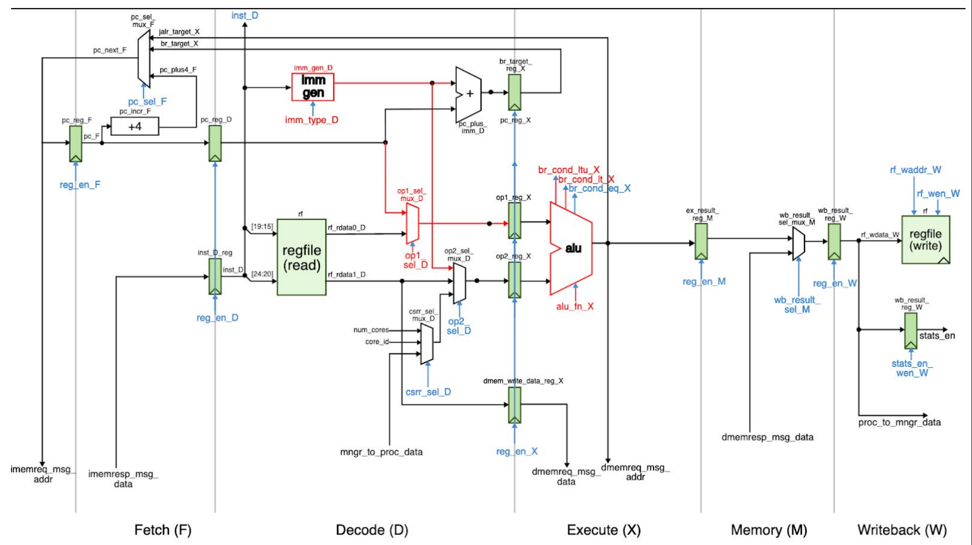
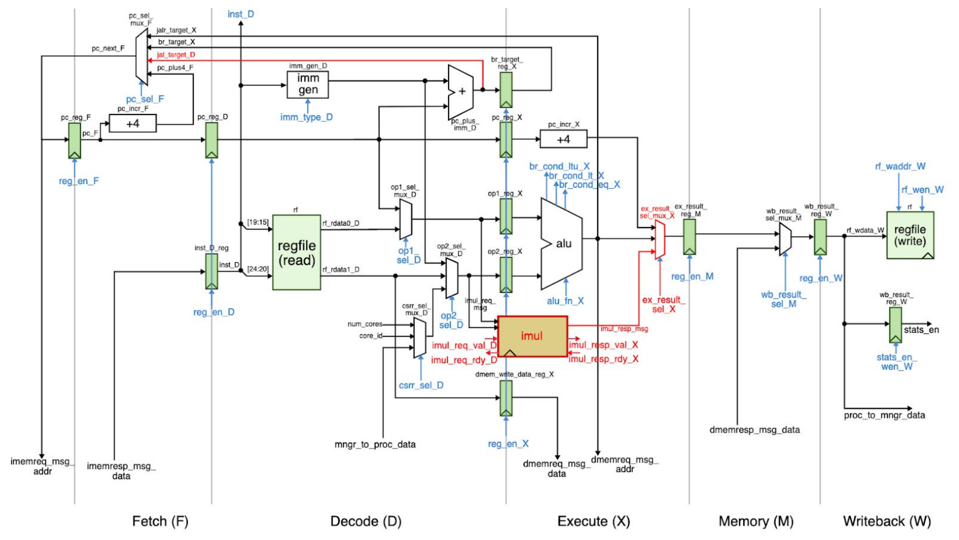
We need to account for the ideal case considered all this while where the latency is zero. In a more realistic case, the memory latency is non-zero. We captured a real scenario for both the baseline and alternative design and have captured the trend in **Figure 5.2**

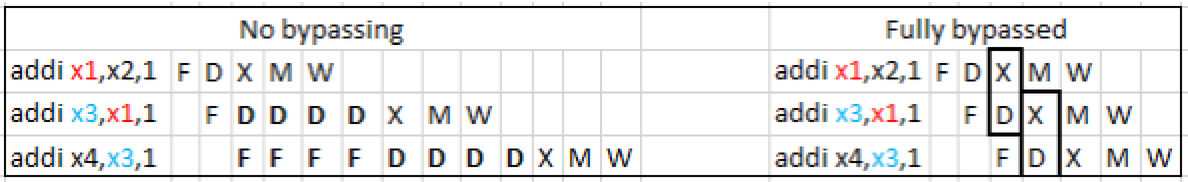
However, we did modify the datapath in the alternative design to include two new 4:1 multiplexers and the logic is modified in the control unit to take care of forwarding. If a trade off is taken b/w the cycle time affected, hardware added, energy spent to form a new design and CPI optimization, we noticed that the performance benefits are more dominant which will be explained further.

Area:
The extra hardware required for the alternative design is just a pair of two 4:1 multiplexers in the case of our five stage pipelined design since we have to bypass the X,M,W stage. Therefore, there is not much increase in the area compared to the baseline design.
As we know from the lectures that potential speedup is proportional to the number of pipeline stages. As the number of pipeline stages increases, the stalling logic gets worse like discussed above. But the overhead for bypassing is not much, as we will still require a pair of muxes but of different sizes. i.e. for a N stage pipeline we require N:1 muxes which still doesn't affect the area significantly.
Energy:
The alternative design required minor updates in the datapath and a bypassing logic in control unit. The bypassing logic or signals can be derived from the stalling signals by taking them as point of reference which helped us in an easy transition.
In terms of man hours, the alternative design required an additional six hours after the thorough development and testing of the baseline design. This means that on thorough implementation and testing of the baseline design, we didn't face any glitches while implementing the alternative design which optimizes the baseline design drastically.
Cycle time:
In terms of cycle time, we must notice that there is a change in the critical path as compared to our baseline design i.e. with the addition of a multiplexer. Even though only one component is added, the effect of having a change in the critical path can be realized at SOC level. If we think of our architecture, we require the clock only for this architecture at this level. Therefore, the clock frequency doesn't play a big role. However, at the chip level, the same clock is going to all the components where a small change in the critical path like this might cost a lot. Therefore, we must be really careful in analyzing the critical path when implementing a bypass design.
There can be a couple of scenarios we need to analyse for bypassing.
In the first scenario, Consider the five stage pipelined architecture is converted into a six stage pipelined architecture with X stage split into two X0 and X1. In this case, if the baseline design is considered, we will have to stall an extra stage which will worsen the CPI whereas in the case of bypassing, we directly it after both the stages.
Affect on :
Area : No affect in area because the output is taken after X0 and X1 completes , therefore we still require a pair of 4:1 muxes
Energy: The man hours spent for designing a fully bypassed architecture is minimal as the bypassing logic doesn't change
Cycle time: Again, the cycle time is something we must be careful about while applying any modification to the design as the number of pipeline stages proportionally increases throughput but it also affects critical time as the bypass path becomes longer and the control logic also holds some delay.

In the second scenario, let's consider a six stage pipelined architecture to speed up the performance without splitting the X stage. Say an additional Y stage is added to the pipeline. To compare it with the baseline design, we will still need additional stalling which will worsen the performance. In this case, there will be modifications required for bypassing logic which we will evaluate in terms of the following:
Area: In terms of area, we now need to replace the pair of 4:1 multiplexers by 8:1 multiplexers as we have five inputs to select from now.
Energy: There will be insignificant amount of man hours required to implement the new bypassing logic as it is just an extension of the basic five stage pipelined one
Cycle time: There is no significant change because of the datapath as we are just replacing multiplexers not adding any new hardware. However if we repeat the same argument given above, the number of pipelined stages is proportionate to the size of bypass paths and bypassing logic complexity in control unit. Therefore, we must see the difference it will make at chip level and take a reasonable trade-off.
To summarize and provide a closed discussion, we would like to highlight that we a significant improvement in the CPI b/w the alternative design and baseline design. An improvement of almost double can be seen which is a dominant factor when compared with the expense of area, energy and cycle time. Even though it doesn't really resolve the problem on control hazards, it significantly improves and provides a good resolution to the data hazards.

APPENDIX

 pc_sel_mux_F jalr_target_X br_target_X pc_next_F pc_plus4_F inst_D imm_gen_D imm gen imm_type_D br_target_reg_X pc_plus_imm_D pc_sel_F pc_incr_F +4 pc_reg_F pc_F pc_reg_D reg_en_F op1_reg_X br_cond_eq_X alu rf [19:15] rf_rdata0_D regfile (read) inst_D_reg inst_D [24:20] rf_rdata1_D op2_sel_mux_D op2_reg_X alu_fn_X imul_req_msg csrr_sel_mux_D num_cores core_id op2_sel_D csrr_sel_D reg_en_D dmem_write_data_reg_X reg_en_X ex_result_reg_M reg_en_M wb_result_sel_mux_M wb_result_reg_W reg_en_W wb_result_sel_M rf_waddr_W rf_wen_W rf_wdata_W regfile (write) stats_en stats_en_wen_W proc_to_mngr_data dmemresp_msg_data memreq_msg_addr imemresp_msg_data mngr_to_proc_data dmemreq_msg_data dmemreq_msg_addr Fetch (F) Decode (D) Execute (X) Memory (M) Writeback (W)	Figure 2.1
 imm_gen_D imm gen imm_type_D op1_sel_mux_D op1_sel_D br_cond_ltu_X br_cond_lt_X br_cond_eq_X alu alu_fn_X pc_reg_X regfile (read) regfile (write) Fetch (F) Decode (D) Execute (X) Memory (M) Writeback (W)	Figure 2.2
 jal_target_D pc_incr_X +4 br_cond_ltu_X br_cond_lt_X br_cond_eq_X ex_result_sel_mux_X ex_result_sel_X imul imul_req_val_D imul_req_rdy_D imul_resp_msg imul_resp_val_X imul_resp_rdy_X regfile (read) regfile (write) Fetch (F) Decode (D) Execute (X) Memory (M) Writeback (W)	Figure 2.3



No bypassing															Fully bypassed							
addi x1,x2,1	F	D	X	M	W										addi x1,x2,1	F	D	X	M	W		
addi x3,x1,1		F	D	D	D	D	X	M	W						addi x3,x1,1		F	D	X	M	W	
addi x4,x3,1			F	F	F	F	D	D	D	D	X	M	W		addi x4,x3,1			F	D	X	M	W

Figure 3.1

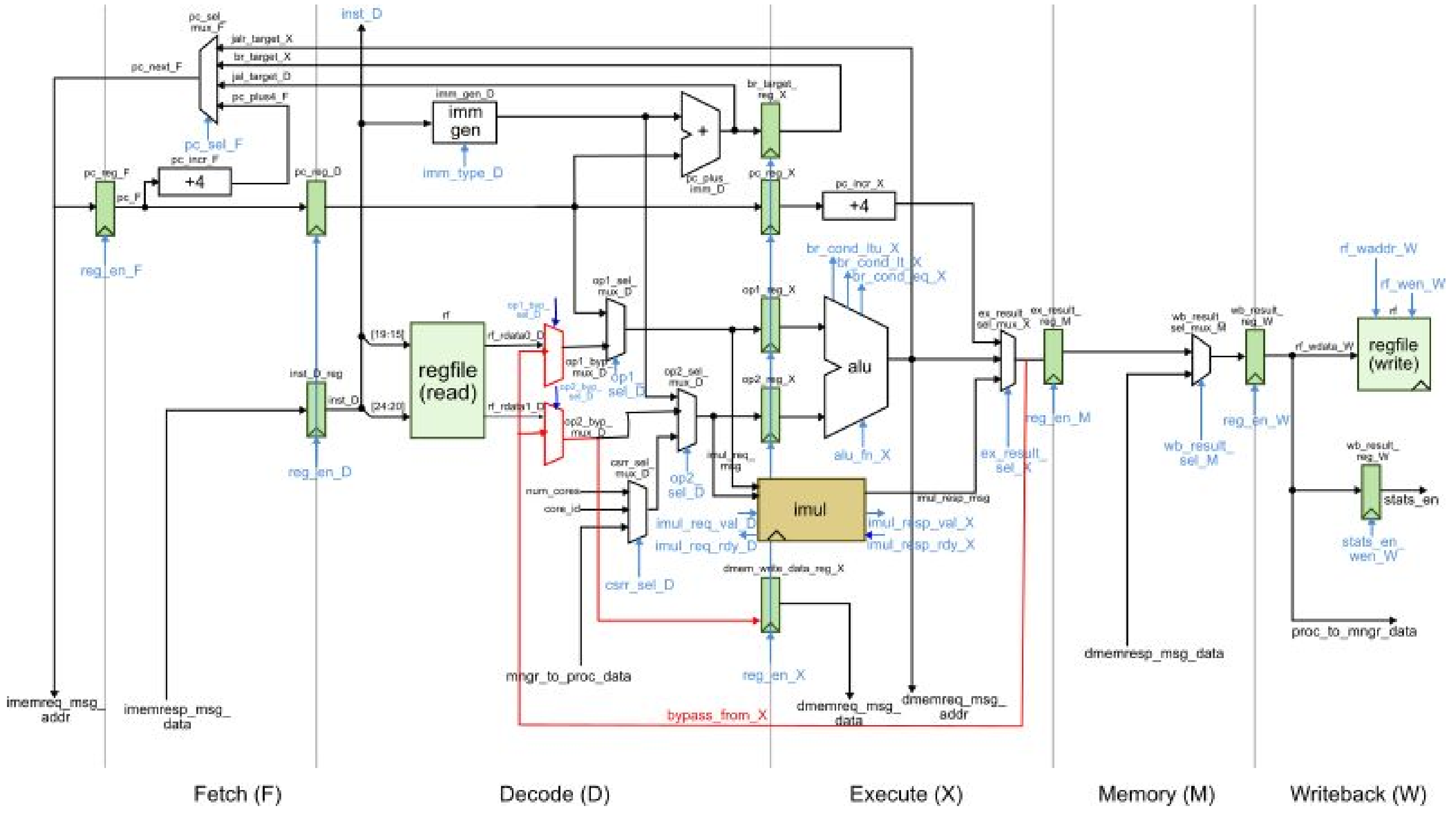
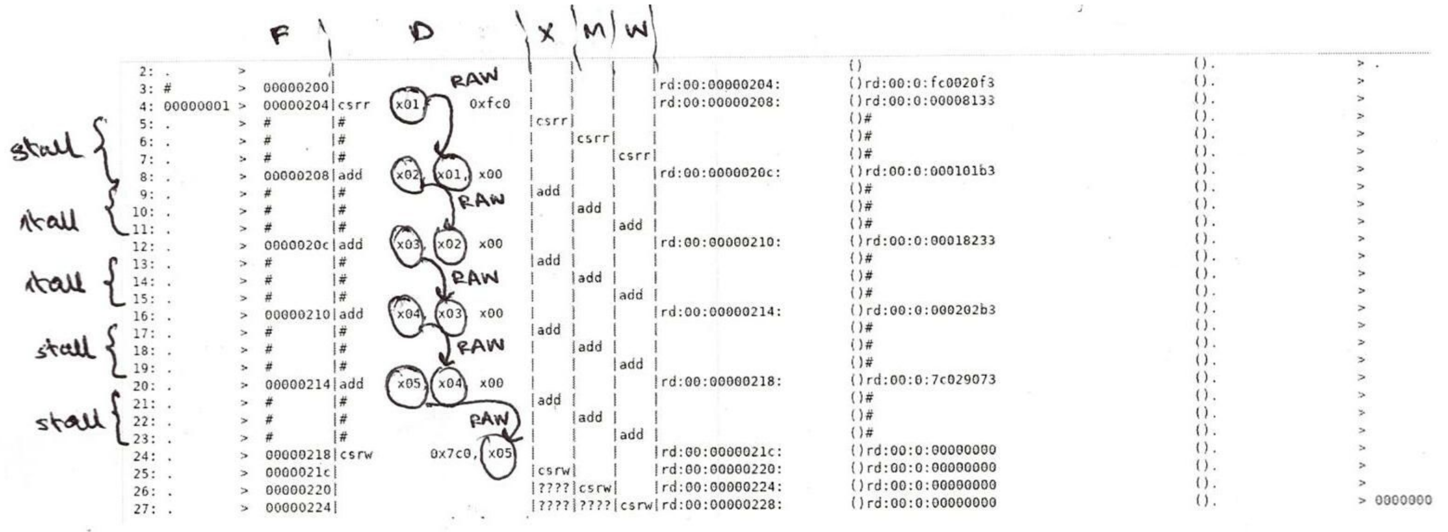
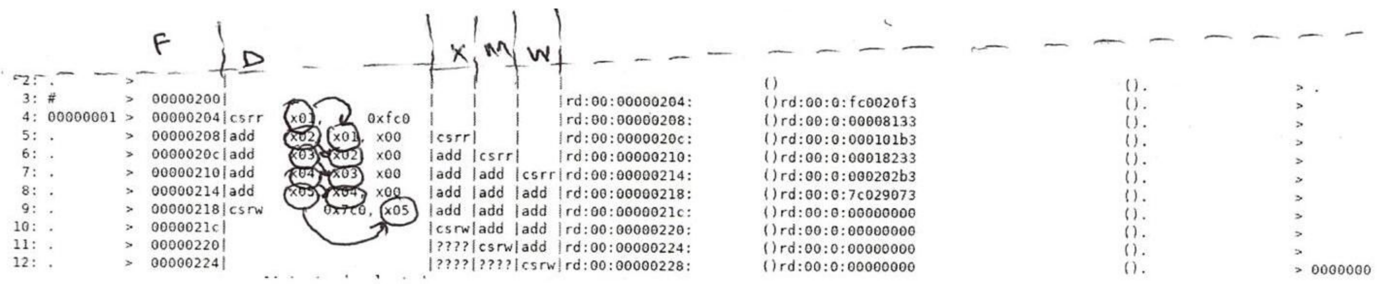


Figure 3.2



```
 2: .         >          |                          |    |    |    |                      ()                            ().          > .
 3: #         > 00000200|                          |    |    |    |rd:00:00000204:      ()rd:00:0:fc0020f3            ().          >
 4: 00000001  > 00000204|csrr    x01    0xfc0      |    |    |    |rd:00:00000208:      ()rd:00:0:00008133            ().          >
 5: .         > #        |#                        |csrr|    |    |                      ()#                           ().          >
 6: .         > #        |#                        |    |csrr|    |                      ()#                           ().          >
 7: .         > #        |#                        |    |    |csrr|                      ()#                           ().          >
 8: .         > 00000208|add     x02, x01, x00     |    |    |    |rd:00:0000020c:      ()rd:00:0:000101b3            ().          >
 9: .         > #        |#                        |add |    |    |                      ()#                           ().          >
10: .         > #        |#                        |    |add |    |                      ()#                           ().          >
11: .         > #        |#                        |    |    |add |                      ()#                           ().          >
12: .         > 0000020c|add     x03, x02  x00     |    |    |    |rd:00:00000210:      ()rd:00:0:00018233            ().          >
13: .         > #        |#                        |add |    |    |                      ()#                           ().          >
14: .         > #        |#                        |    |add |    |                      ()#                           ().          >
15: .         > #        |#                        |    |    |add |                      ()#                           ().          >
16: .         > 00000210|add     x04, x03, x00     |    |    |    |rd:00:00000214:      ()rd:00:0:000202b3            ().          >
17: .         > #        |#                        |add |    |    |                      ()#                           ().          >
18: .         > #        |#                        |    |add |    |                      ()#                           ().          >
19: .         > #        |#                        |    |    |add |                      ()#                           ().          >
20: .         > 00000214|add     x05, x04, x00     |    |    |    |rd:00:00000218:      ()rd:00:0:7c029073            ().          >
21: .         > #        |#                        |add |    |    |                      ()#                           ().          >
22: .         > #        |#                        |    |add |    |                      ()#                           ().          >
23: .         > #        |#                        |    |    |add |                      ()#                           ().          >
24: .         > 00000218|csrw         0x7c0, x05   |    |    |    |rd:00:0000021c:      ()rd:00:0:00000000            ().          >
25: .         > 0000021c|                          |csrw|    |    |rd:00:00000220:      ()rd:00:0:00000000            ().          >
26: .         > 00000220|                          |????|csrw|    |rd:00:00000224:      ()rd:00:0:00000000            ().          >
27: .         > 00000224|                          |????|????|csrw|rd:00:00000228:      ()rd:00:0:00000000            ().          > 0000000
```

Figure 3.3



```
  2: .          >
  3: #          >  00000200|                             |    |    |    |rd:00:00000204:     ()                        ().       > .
  4: 00000001 > 00000204|csrr   x01,       0xfc0  |    |    |    |rd:00:00000208:     ()rd:00:0:fc0020f3      ().       >
  5: .          >  00000208|add    x02, x01,  x00    |csrr|    |    |rd:00:0000020c:     ()rd:00:0:00008133      ().       >
  6: .          >  0000020c|add    x03, x02,  x00    |add |csrr|    |rd:00:00000210:     ()rd:00:0:000101b3      ().       >
  7: .          >  00000210|add    x04, x03,  x00    |add |add |csrr|rd:00:00000214:     ()rd:00:0:00018233      ().       >
  8: .          >  00000214|add    x05, x04,  x00    |add |add |add |rd:00:00000218:     ()rd:00:0:000202b3      ().       >
  9: .          >  00000218|csrw       0x7c0, x05    |add |add |add |rd:00:0000021c:     ()rd:00:0:7c029073      ().       >
 10: .          >  0000021c|                         |csrw|add |add |rd:00:00000220:     ()rd:00:0:00000000      ().       >
 11: .          >  00000220|                         |????|csrw|add |rd:00:00000224:     ()rd:00:0:00000000      ().       >
 12: .          >  00000224|                         |????|????|csrw|rd:00:00000228:     ()rd:00:0:00000000      ().       > 0000000
```

Figure 3.4

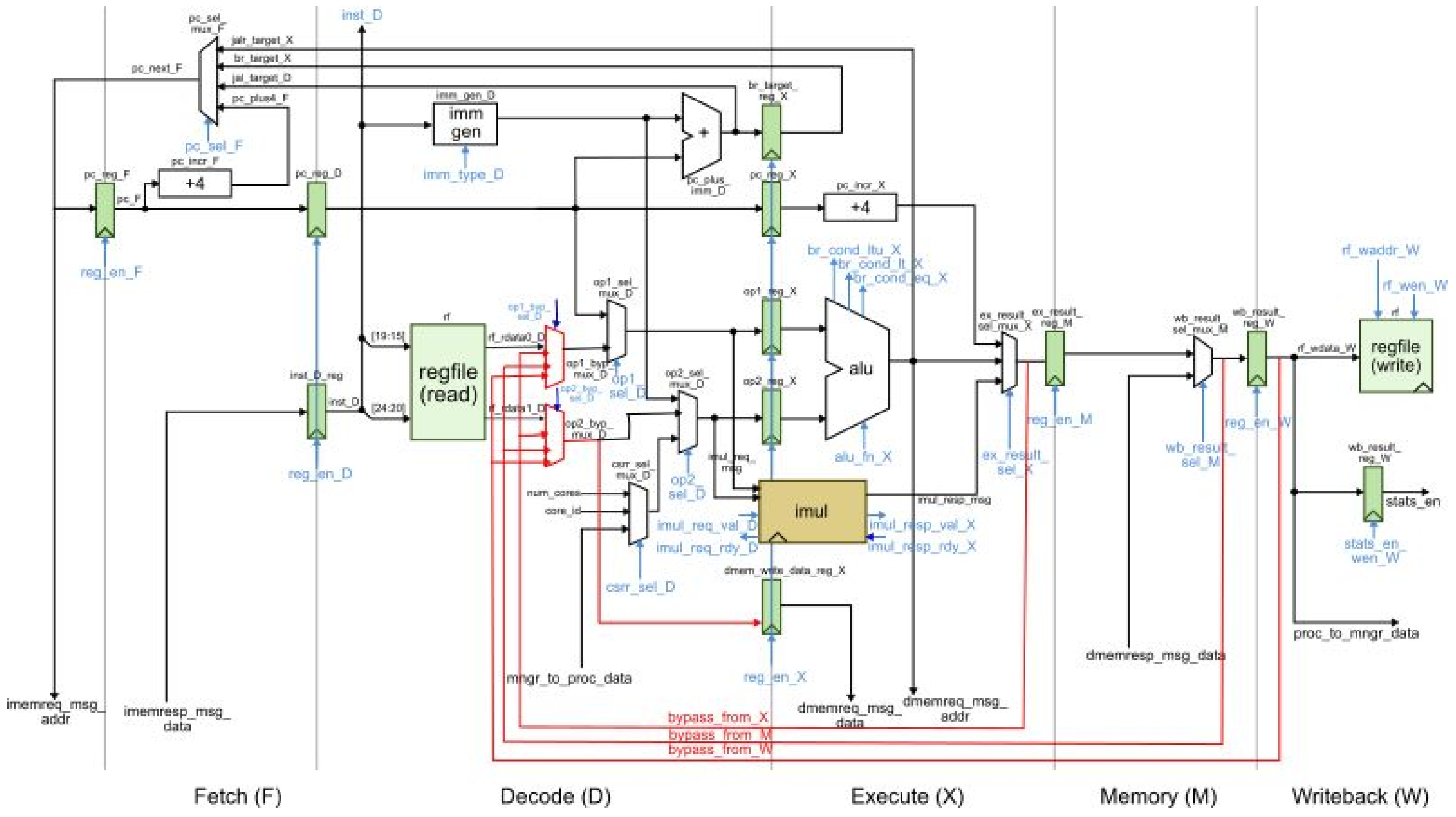
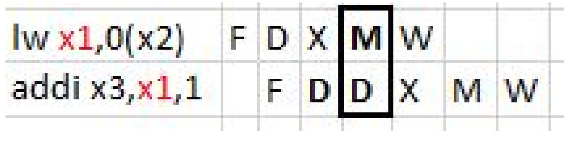
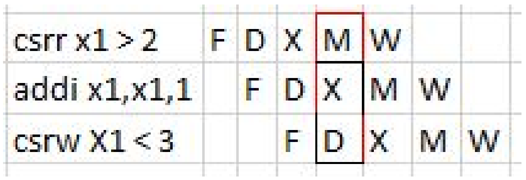


Figure 3.5



lw x1,0(x2)	F	D	X	M	W		
addi x3,x1,1		F	D	D	X	M	W

Figure 3.6

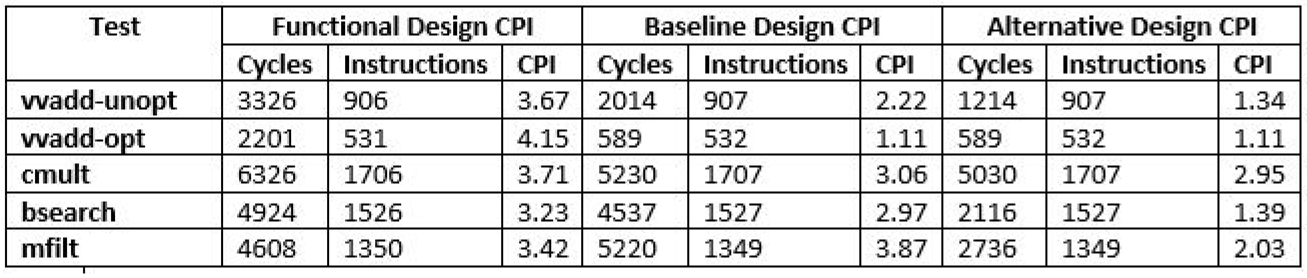


csrr x1 > 2	F	D	X	M	W		
addi x1,x1,1		F	D	X	M	W	
csrw X1 < 3			F	D	X	M	W

Figure 3.7

Instruction Groups		Applicable Tests
Register-register (rr, in the report)		“Dep” tests, destination=source tests, value tests, random value tests, random delay tests
Register-immediate (rimm, in the report)		“Dep” tests, destination=source tests, value tests, random value tests, random delay tests
Memory (l (mem, in the report)		“Dep” tests, value tests, random value tests, random delay tests
Branch		“Dep” tests, random branch tests, random delay tests
Jump (jal, jalr)		Multi-jump tests, branch-then-jump tests, jump-then-branch tests
Alternate design	X->D stage	Explicit tests
	M->D stage	Explicit tests
	W->D stage	Explicit tests

Figure 4.1



Test	Functional Design CPI			Baseline Design CPI			Alternative Design CPI		
	Cycles	Instructions	CPI	Cycles	Instructions	CPI	Cycles	Instructions	CPI
vvadd-unopt	3326	906	3.67	2014	907	2.22	1214	907	1.34
vvadd-opt	2201	531	4.15	589	532	1.11	589	532	1.11
cmult	6326	1706	3.71	5230	1707	3.06	5030	1707	2.95
bsearch	4924	1526	3.23	4537	1527	2.97	2116	1527	1.39
mfilt	4608	1350	3.42	5220	1349	3.87	2736	1349	2.03

Table 5.1

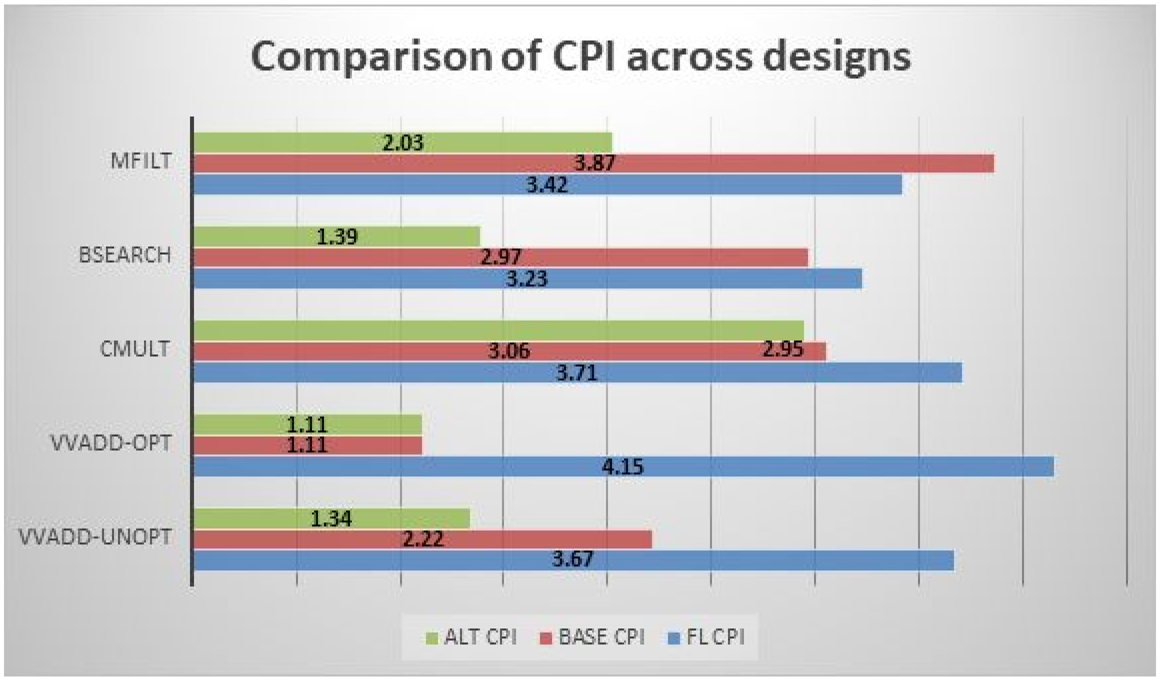


Figure 5.1

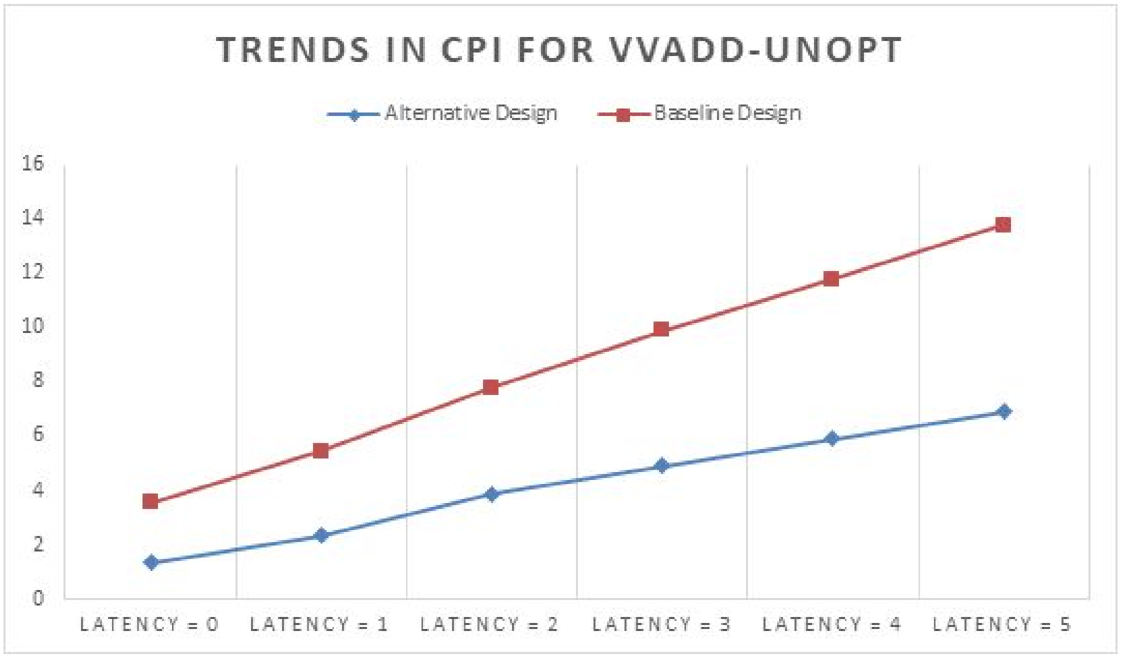


Figure 5.2