ECE 4750 - Lab 2 Report

Sabrina Herman (sh997), Yoon Jae Oh (yo82), Benjamin Xing (byx2)

Section 1: Introduction

The purpose of this lab was to gain experience with topics taught in class such as ISAs, basic pipelined processor microarchitecture, microarchitectural techniques for handling data and control hazards, and interfacing processors and memories through designing two pipelined processor microarchitectures for the TinyRV2 instruction set architecture. The lab incorporates many important themes in computer architecture like analyzing processor performance through average latency and cycles per instruction (CPI), and therefore relates to lecture materials as well as industry practices. We designed a control unit and datapath with a split design pattern, building off of what we learned in lab 1. We used an incremental development design and testing methodology by beginning with a baseline design before moving on to an alternate design. These computer architecture themes are relevant not only to this class, but to how computer architecture, design, and verification are implemented in industry.

The lab taught us microarchitectural techniques for handling data and control hazards in pipelined designs through the RTL implementation of stalling, bypassing, and squashing. In the baseline design, instructions were stalled if a data hazard occurred (when an instruction earlier in the pipeline depended on a value that was later in the pipeline). In the alternative design, bypassing allowed an instruction with a data or control hazard to get the needed value forwarded to its decode stage by the older instruction. We worked with instruction set architecture both in the design and testing portion of the lab. We used the TinyRV2 ISA when we used the type of instruction and its values to set the control signals, and wrote Tiny RV2 assembly instructions in our tests. We worked with basic pipelined processor microarchitecture by designing components in the datapath, setting values in the control unit, and editing the parent module which together incorporated a pipeline design. We learned that in a pipelined design, it is important to hold values in registers and accurately pass information from stage to stage. In our design, we had to be careful to access the values of the correct stage. Understanding how the parent module interfaced between the datapath, the control unit, the memory, and the test source/sink was essential for completion of the lab, and taught us design patterns including message interfaces between the processor and memory, the control/datapath split, and pipelined control.

The alternative design is more compelling than the baseline design because it saves on the number of clock cycles needed to finish a set of instructions with dependencies. On average for functions with data dependencies, the alternative design is 13% more efficient (see table 5). The alternative design accomplishes this with a very modest increase in hardware area and energy requirements. With the overall cycles saved for executing programs, the alternative design will actually save energy overall because the extra power required for the added components is small. The alternative design accomplishes this by hardware bypassing data values from later stages to earlier stages in the pipeline. Due to the fact that dependencies can be common in instructions sequences that perform functions such as arithmetic operations (many sequences consist of a series of arithmetic instructions that depend on the previous instruction's output, ex. Fibonacci sequence), the fact that the alternative design saves on clock cycles for dependencies makes it a very compelling design. For instance, as shown in the evaluation data below, the number of cycles required for a general computation task like performing a convolutional filter on a small image is smaller for the bypassed design than the basic stalling design (7660 vs 9540 cycles, for a 20% decrease in CPI). A mix of different instructions is used, including arithmetic instructions like add, slli, and mul, along with memory instructions and branch instructions. As other examples show, however, the alternative design is only more compelling than the baseline design for instruction sequences with control or data dependencies. Otherwise, their performance is the same, as demonstrated by the vvadd optimized code below.

In general, reducing the number of cycles stalled will significantly decrease the CPI and increase the throughput for any microarchitecture. There are more approaches to resolving hazards without stalling that were not implemented in this lab. Data hazards could be solved by exposing data hazards in ISA, hardware scheduling, and hardware speculation. There are also more approaches to resolving control hazards, such as exposing in ISA, software predication, hardware speculation, and software hints. Bypassing was implemented in this lab and virtually all modern processors because it reduces stalling while demanding very few hardware and control logic additions, an ideal result for all possible computing scenarios.

Section 2: Baseline Design

A pipelined microarchitecture is divided into stages with each stage performing specific tasks. To implement stages, registers need to be added between stages to send information from one stage to the next on every clock cycle. Compared to a single-cycle processor, pipelining reduces the cycle time while still approximately achieving an average of one cycle per instruction. Compared to an FSM processor, pipelining has a smaller CPI while approximately achieving a similar cycle time (clock period) to the FSM. However, pipelining introduces various hazards that complicate the control logic. The baseline design is a 5-stage pipelined processor that utilizes stalling and squashing to handle data and control hazards. The stages in the pipeline are controlled by control signals which enable registers, set select signals of muxes, and set other valid/ready signals. Figure 1 displays the datapath for the baseline design.

Though some of the pipelined processor was given to us, we still needed to make changes to the datapath and control unit to make this processor fully-functional for all instructions. The datapath needed more muxes and registers to handle more complicated instructions. For example, a register needed to be added to hold data being sent to memory, and a mux needed to be added to send the PC to the ALU for jump and link instructions. The control unit needed additions to the control signal table for every new instruction added and some instructions required new control signals to be defined.

Stalling and squashing are good approaches to handling hazards in the baseline design, as they are simpler to implement than bypassing. This is because bypassing requires passing of data backwards in the pipeline to other stages. Resolutions of hazards that went beyond hardware were not implemented as they were not realistic for a lab in which the students design only the hardware of the processor. Examples of software solutions include exposing data and control hazards to the ISA, control hazard software predication, and control hazard software hints, all of which would require going beyond designing hardware and looking into the role of the programmer or compiler to implement. Another approach for resolving data hazards is hardware rescheduling, but this task would also be difficult to implement as there would need to be complicated logic and hardware to be able to detect the hazard, determine the order in which to execute instructions, and then reorder the instructions going into the pipeline. Therefore, hardware stalling, hardware bypassing/forwarding, and hardware speculation were the optimal solutions considered for this lab. Of those solutions, hardware speculation is the only one that can handle control hazards, so this was implemented in both the baseline and alternative design. Hardware stalling is simpler than bypassing, so stalling makes for a good baseline design, and bypassing makes for a good alternative design where both bypasing and some stalling are needed.

Hardware stalling is the simplest hardware solution because it relies on updating the control signals in our control unit based on our datapath's status signals, therefore integrating well with our datapath/control unit split design. To implement stalling, our control unit first checks for data hazards, namely read-after-write data hazards, by looking at the register being read and the address being written to in a later stage. If the control unit finds that an instruction is trying to read a register that any instruction later in the pipeline is trying to write, it notifies the datapath to stall (by disabling registers). The control unit will detect stall if there is a valid instruction in the stage originating the stall and if a hazard was detected.

Hardware speculation was used for the branching and jumping instructions. We implemented this by writing the appropriate squashing signals in the D and M stages for jumps and branches, respectively. These squashes were similar to stall signals in that a stage originated a squash if a control hazard was detected in any of the earlier stages. The main difference between squashing and stalling is that a stage will stall itself, but an instruction will not squash itself.

Section 3: Alternative Design

Our baseline design handles data hazards with stalling, which is not the optimized solution. With hardware stalling, throughput is lowered because less instructions are being executed per cycle (if there are data dependencies). The reason for stalling was that in some cases - specifically data hazards involving read-after-write (RAW) operations - a later instruction needed a data value from an earlier one, and therefore needed to wait for the earlier instruction to finish writing back to a register before it could continue in the pipeline. This waiting causes wasted cycles, as the stages of the pipeline are holding onto instructions for multiple cycles instead of just one cycle.

The optimization of the alternative design was to use bypassing instead of stalling whenever possible. Bypassing solves the problem of having to wait for a computed value to be written back to the register file before it can be used by a later instruction as it allows values to be sent back to the end of the decode (D) stage from any of the later pipeline stages. Eliminating the holding of instructions for multiple cycles in a stage significantly increases the throughput of the processor while requiring only a small increase in hardware area and power requirements, namely a couple extra multiplexers and some new control logic. The only case in which stalling is still necessary for a data hazard is in the case of the load word instruction, since the new data enters the processor only in the memory (M) stage, not the execute (X) stage. As a result, the earliest the data can be bypassed back to the D stage is at the end of the M stage as opposed to the X stage which is in most arithmetic instructions; thus if the instruction immediately following the lw instruction has a dependency on the result of the load, it would need to stall in the D stage for one cycle to wait for the load value to be forwarded. This example helps demonstrate why bypassing was implemented in the alternative design, after stalling was implemented in the baseline design, since this example requires both bypassing and stalling.

To add the bypassing functionality, we once again structured our approach around the control/datapath split design principle. Since the additions to the datapath were fairly simple, we chose to implement them first. As mentioned, three bypass paths were implemented: the end of X to the end of D, the end of M to the end of D, and the beginning of W to the end of D. The sources of these bypass paths were chosen specifically to take the value from just before the staging registers in the case of X and M, and just before writeback for W. This is because the logic can be viewed as combinational within the X and M stages (though in practice, memory accesses and the multiplier are not actually combinational) and we need to bypass the resulting data of each stage. For the bypass from W we need to take the value from before writeback since writeback is the time-consuming part of W. Two 4-input bypass multiplexers were added in the D stage before the operand select muxes. The bypass paths from X, M, and W, along with the input from the register

file, make up the inputs to each mux. The outputs of the bypass muxes connect to the inputs of the original operand select muxes as before.

The new control logic is slightly more involved as it requires not only the bypass logic to be implemented but also changes to the stalling logic. We approached this problem by first categorizing all possible hazards that could occur in our microarchitecture. For the TinyRV2 instruction set, only RAW and control hazards could occur (structural and WAW/WAR hazards do not arise in this microarchitecture). Moreover, since control hazards can only be resolved through stalling or hardware speculation, our bypass paths would not affect how we were already handling control hazards.Bypassing values meant we would need to change the stalling logic in the D and X stages. Originating stall (ostall) hazards for load dependencies were created for both source operands, which originate stalls if an instruction depends on a load instruction. The alternative processor then only stalls if there is a load use dependency. We also added bypassing signals to check whether any of the source operands of the instruction in the D stage are the same as a write address of an instruction in the X, M, or W stages; if so, we allow the bypass muxes to pass the corresponding bypass value through. The only exception to this is if a lw instruction is in the X stage, in which case the bypass will have to occur in the M stage after stalling for one cycle. Since bypassing handles all RAW data hazards that were handled in the baseline design through stalling, we simply deleted the extra ostall signals (that were in the baseline) in the alternative design.

Bypassing was chosen as the method of dealing with hazards for the alternative design because it is an improved design over stalling due to the amount of cycles it saves when there are dependencies between instructions. Unlike stalling, bypassing allows values to be forwarded from the X, M, or W stage to the instruction that requires the necessary value(s). This saves on clock cycles because the instruction does not need to stall until the previous instructions it depends on are finished. For example, Table 1 displays a simple example set of instructions that shows how bypassing saves cycles as opposed to stalling. As can be seen in the table, the processor with stalling would take 9 cycles to execute the assembly code whereas the processor with bypassing would only take 3 cycles to execute (both calculations exclude warm-up time of pipeline). Bypassing reduced the number clock cycles by 3 times the number of stalling to execute the same set of instructions.

Section 4: Testing Strategy

We used incremental and test driven methodologies for completing both the baseline and the alternative design. That is, we verified our implementation of each instruction before continuing to the next one by running the provided basic test and a few simple directed value tests we write during the development process. To ensure that the tests themselves were valid, we check their correctness in the functional-level representation since the functional level does not require a microarchitecture to compute the results (so it is guaranteed to be correct). After fully implementing the datapath and control modules, we comprehensively tested each instruction by adding directed test cases for data hazard resolution and data edge cases. We also added random value testing as well as random delay testing to ensure that the processor could handle all cases and the uncertainty of real-world computation. We combined these directed and random tests into unit test suites for each instruction that not only verified the correctness of the processor in isolated tests but also in sequences of instructions. Throughout the testing process, we use the line trace outputs of the unit tests to locate errors and debug faulty connections and control logic in our modules.

Directed testing was essential to ensuring that our instructions would respond appropriately to read-after-write (RAW) data dependency hazards. Specifically, we added Python assembly-generation test functions for destination-dependent and source-dependent hazards as well as when the source and destination are the same. To make sure that the microarchitecture would either stall or bypass to handle these cases, we varied the number of nops between consecutive instructions from 5 to 0 to test whether, for example, an add instruction in the baseline design would stall until both source registers have been written by the preceding csrr instruction. We also wrote directed value tests to check that the implementation correctly handled edge cases, especially those instructions in which signed/unsigned representations matter. In all of the register-register and register-immediate instructions, for example, we add test cases for small positive, large positive, small negative, large negative, and zero data values for both source operands. Likewise, for memory instructions we test these edge cases for both the data value being stored/loaded and the memory address in which to do so. For the jump and branch tests, our directed tests check that the PC is loaded with the correct value by keeping track of a register in which bits are set by addi instructions at various labeled points; some of these points should be skipped if the control flow is correct. These tests allowed us to verify that the jump or branch instruction performs correctly when we add new labels the processor should jump to. Within those tests, we verify branching/jumping to both forward and backward addresses correctly. The jump and branch test cases also test that the squash signals work correctly by checking that the right set of instructions are executed after a jump/branch instruction.

We also added random testing to ensure not only that the implementation is correct across general use cases but also to simulate what the processor might see in a real computing scenario. For instance, random testing is often useful for covering edge cases or other special cases we may have forgotten to check in our directed testing. Moreover, random testing sometimes exposes errors in our microarchitectural implementation, such as when we forgot to restrict ALU shift operations to the 5 least-significant bits in the second source operand. We implemented random value testing using Python's built-in pseudorandom number generation

functions for the operand values and checking the output of the processor against the expected value that we determine as a function of the inputs. In the case of branch and jump instructions, we use random testing to insert a variable number of nops between control flow instructions to test whether the branch or jump logic is affected. To inject even more uncertainty into the tests, we also add random delay testing. These tests simulate real-world delays in hardware, such as when requesting data from source or memory. Since data memory accesses, for example, can take an unpredictable amount of time to complete due to the possibility of cache misses, random delay tests ensure that our processor will simply stall to wait for data to arrive.

Finally, we combine individual directed tests and random tests into unit tests to take advantage of the modularity of the processor. Since the processor functionality can be viewed as a hierarchy of instruction classes and operation types, the majority of our unit tests execute at the level of groups of instructions of the same class (i.e. register-register, register-immediate, memory, etc.). These unit tests verify that we are not missing any major components in our datapath since all the instructions in a class of instructions would fail if an essential datapath component was missing. Moreover, unit tests verify the implementation of more complicated instructions, such as branch instructions that depend on the correctness of the fundamental instructions like the addi instruction. They also give us the freedom to ascend another hierarchical level and test the processor as a whole, combining instructions from multiple classes and varying or eliminating separating nops to ensure that data hazards are still handled correctly.

Section 5: Evaluation

Our alternative design has a higher cycle time because of a longer critical path due to bypassing. In a pipelined processor, the cycle time is determined by the slowest stage in the pipeline, and how much time that stage requires is determined by looking at the longest datapath from one register to another. In our alternative design, we added bypassing which created a longer critical path. An example of a long bypassing critical path is if there is a RAW data hazard in the X and D stage. This is a longer critical path because the result of our execution stage needs to be forwarded to the start of the decode stage, and therefore the cycle time is based on the time for the X stage instruction to finish and then the decode instruction to finish. This is longer than our baseline design where values are only dependent on data from its own stage.

Though our cycle time is slightly longer, this tradeoff is well worth it for the alternative design. This is because the energy required for the alternative design will be much less than the baseline design. In the baseline design, a lot of energy is wasted on stalled pipeline stages where instructions take longer to execute and resources within the pipeline are wasted waiting to resolve a hazard. In the alternative design, the only stall occurs for a load word instruction, and all other RAW data hazards use bypassing, which helps prevent wasted time due to stalling. In preventing stalling, we increase the number of instructions able to execute per cycle. Increasing CPI so significantly is well worth the small increase in cycle time, as our execution time will be less for the case of high CPI and slightly higher cycle time.

Our area will increase slightly for the alternative design because more hardware is required for bypassing. Specifically, wires and muxes are required to send the data back to an early stage and then choose between values coming in from the register file or from later stages. Note, the addition of 2 muxes and some more wired connections within our datapath and from the control unit to the datapath is not that much more hardware than baseline design, so the tradeoff of adding more area for our design is well worth it.

Our evaluation of the provided benchmarks demonstrates when the alternative design has a higher CPI than our baseline design. As discussed previously, the alternative design will have a higher CPI when there are RAW data dependencies. The first benchmark, an unoptimized vector-vector add function, demonstrates this fact. In Table 3: Evaluation Data, it is evident that the baseline design has a CPI of 9.33 while the alternative design has a CPI of 8.67. The reason for this is due to the data dependencies in the assembly code for this benchmark (see Code Snippet 1). These dependencies are also present for bsearch and mfilt (see code snippets 4 and 5) which our evaluation determines have lower CPI for the alternative design. There are no RAW data dependencies in the optimized version of vector-vector add, and as a result the CPI for both designs is the same. Though there are RAW dependencies in the mult benchmark, our design stalls for multiplication in a unique way which does not involve holding the operand values in registers (See Figures 1 and 2 the Baseline/Alternative Datapaths and notice that there are no registers for multiplication unit). Our multiplication unit waits until the operands are valid, and then will immediately begin executing. Therefore the entire stage will not have to be stalled. The multiplication unit waits long enough for the operand(s) to be written back and then finishes execution stage. Therefore, with these specific assembly instructions for complex multiplication, there is no need to stall/bypass as the dependencies will be resolved within our datapath design.

Also included in our evaluation of memory latency on CPI. The relationship between memory latency and CPI appears to be linear in both graph 1 and graph 2, and the type of design (baseline or alternative) does not matter when considering the type of relationship between memory latency and CPI. This makes sense considering that a longer memory latency will increase cycle time, not CPI. However, as the memory latency increased, the difference in CPI between the baseline design and the alternative design decreased.

Section 6: Additional Diagrams

Figure 1: Baseline Datapath

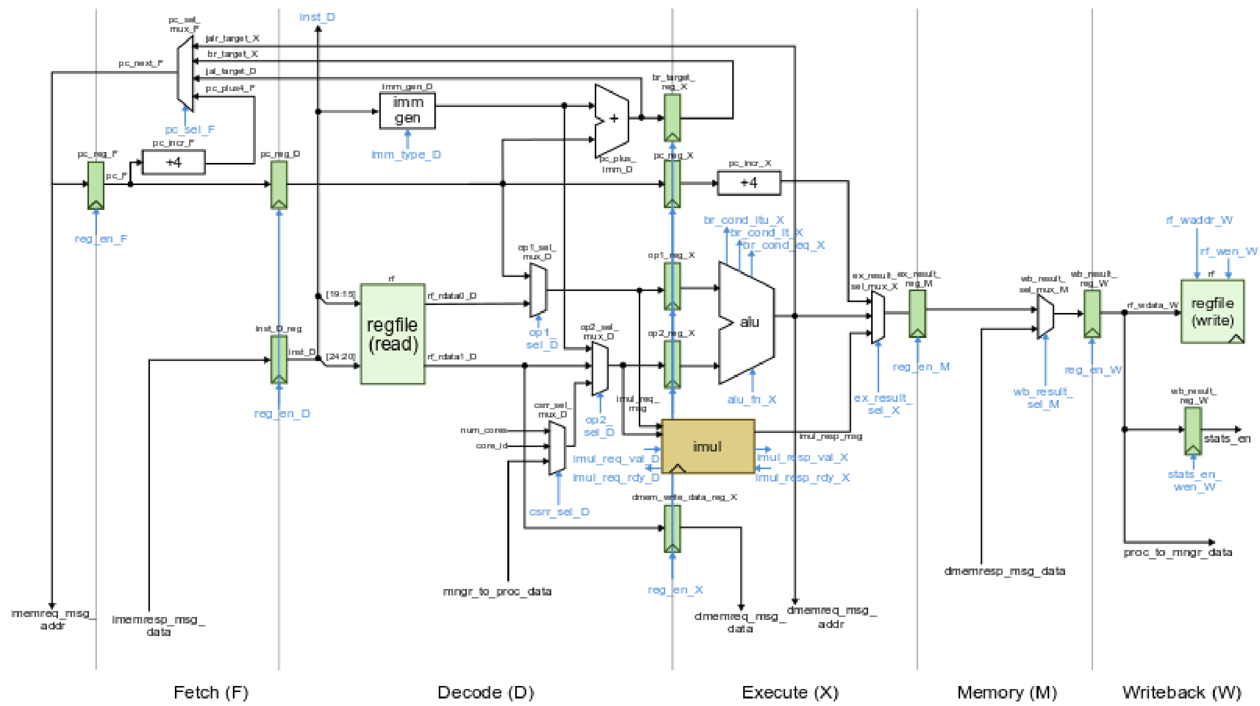


Figure 2: Alternative Datapath

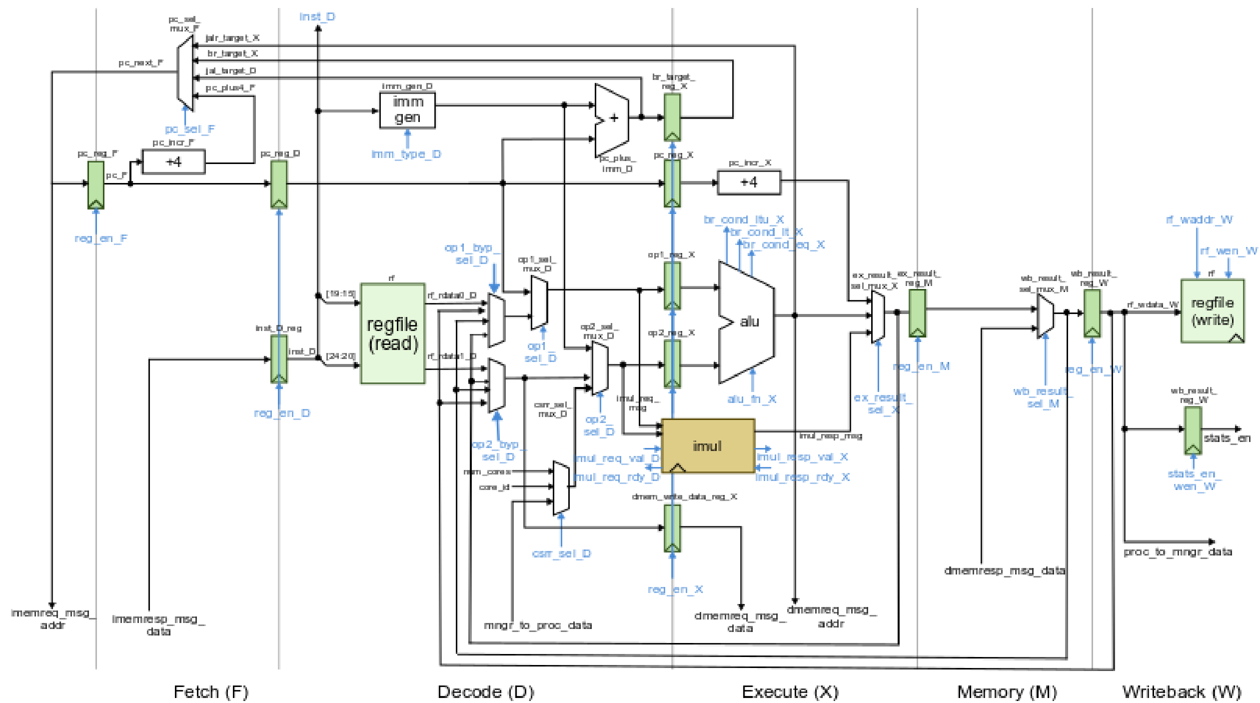


Table 1: Example Assembly Code

Example Assembly:	Stalling (Baseline):													
Add x3, x2, x1	Add x3, x2, x1	F	D	X	M	W								
Add x4, x4, x3	Add x4, x4, x3		F	D	D	D	D	X	M	W				
Sub x5, x4, x1	Sub x5, x4, x1			F	F	F	F	D	D	D	D	X	M	W
	Bypassing (Alternate)													
	Add x3, x2, x1	F	D	X	M	W								
	Add x4, x4, x3		F	D	X	M	W							
	Sub x5, x4, x1			F	D	X	M	W						

Table 2: Role and Task Table

Lab	sh997	byx2	yo82
Lab 1	RTL	Verification	RTL (architect)
Lab 2	Verification	RTL(architect)	RTL
Lab 3			
Lab 4			
Lab 5			

	sh997	byx2	yo82
Tasks	**Baseline Design:** -Made changes to datapath to include new muxes and registers needed -Added lines in control signal table for branching and jumping -Added logic for jump instruction squashing -Added to ALU for less than and unsigned less than condition indicators **Alternative Design:** -Designed new datapath for alternative design, referencing lab material and lecture notes for optimal solution **Testing:** Wrote tests for branching and jumping, as well as some additional tests to test stalling, squashing and bypassing **Report:**	**Baseline Design:** -Implemented parts of control signal table -Added new operations to ALU module -Helped with stalling and squashing, PC redirect signals **Alternative Design:** -Implemented datapath changes including bypass connections and muxes in the dpath module and parent module -Modified stalling signals to account for new bypassing functionality **Testing:** -Wrote test cases for register-immediate instructions, helped with datapath component testing **Report:** -Wrote parts of testing section and alternative	**Baseline Design:** -Created missing muxes and registers as needed -Added to control table in control unit -Helped with signals for stalling and squashing **Alternative Design:** -Created and changed bypass signals -Implemented datapath changes **Testing:** -Wrote test cases for all register to register instructions, memory instructions, created the store word test template, and worked on the datapath components test cases **Report:** -Introduction, parts of alternative design and testing sections

	Wrote large parts of Baseline and Alternative design sections, as well as the evaluation section	design section	

Table 3: Evaluation Data

Benchmarks	Baseline	Alternative
vvadd-unopt : Element-wise vector-vector add (unoptimized)	[passed]: vvadd-unopt num_cycles = 8455 num_insts = 906 CPI = 9.33	[passed]: vvadd-unopt num_cycles = 7855 num_insts = 906 CPI = 8.67
vvadd-opt : Element-wise vector-vector add (optimized)	[passed]: vvadd-opt num_cycles = 4280 num_insts = 531 CPI = 8.06	[passed]: vvadd-opt num_cycles = 4280 num_insts = 531 CPI = 8.06
cmult : Element-wise complex multiplication	[passed]: cmult num_cycles = 9066 num_insts = 1706 CPI = 5.31	[passed]: cmult num_cycles = 9066 num_insts = 1706 CPI = 5.31
bsearch : Binary search in a linear array of key/value pairs	[passed]: bsearch num_cycles = 11845 num_insts = 1526 CPI = 7.76	[passed]: bsearch num_cycles = 10195 num_insts = 1526 CPI = 6.68
mfilt : Masked convolution on a small image	[passed]: mfilt num_cycles = 9540 num_insts = 1349 CPI = 7.07	[passed]: mfilt num_cycles = 7660 num_insts = 1349 CPI = 5.68

Table 4: Effect of Memory Latency

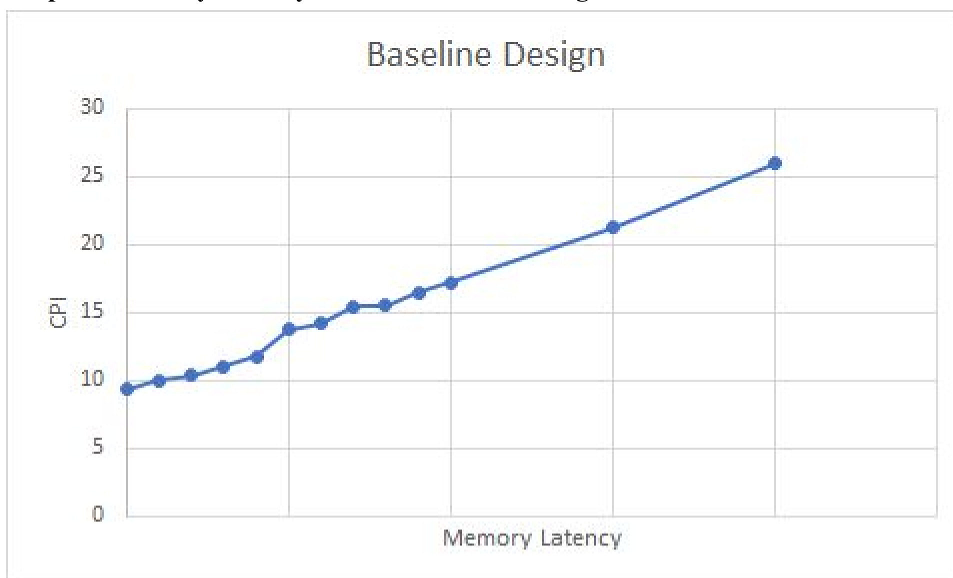
mem latency	Base CPI	Alt CPI	Percent difference
0	9.33	8.67	7.073955
1	9.99	9.33	6.606607
2	10.33	9.66	6.485963
3	10.99	10.33	6.00546
4	11.76	11.11	5.527211
5	13.77	13.33	3.195352

6	14.22	13.78	3.094233
7	15.43	14.99	2.851588
8	15.55	15.11	2.829582
9	16.46	16.02	2.673147
10	17.22	16.78	2.555168
15	21.23	21	1.083373
20	25.95	25.51	1.695568

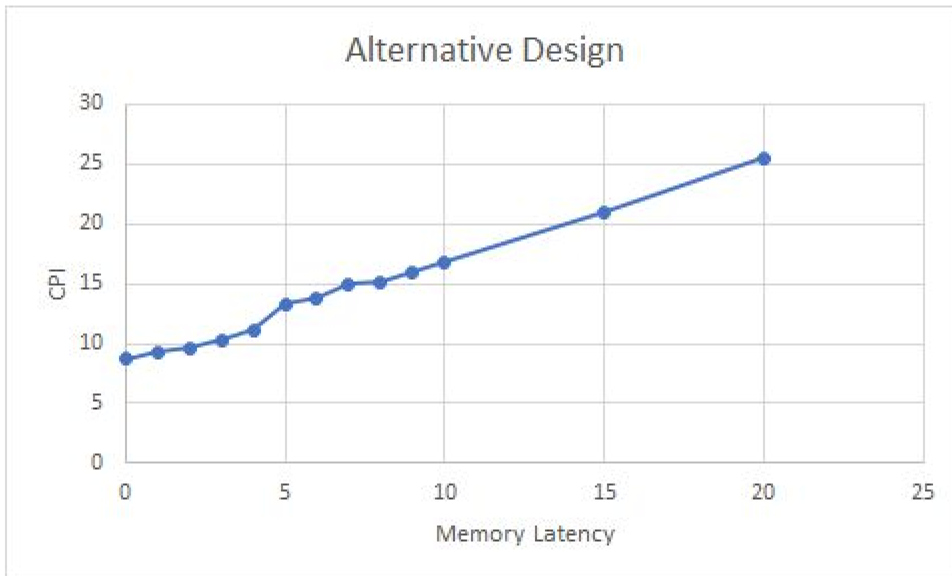
Table 5: Difference in CPI for Alternative and Baseline Design for Cases of Data Dependencies

CPI Base	CPI Alt	Percent Difference
9.33	8.67	7.073955
7.76	6.68	13.91753
7.07	5.68	19.66054
	average =	13.55067

Graph 1: Memory Latency Effect on Baseline Design CPI



Graph 2: Memory Latency Effect on Alternative Design CPI



Code Snippet 1: vvadd unoptimized (RAW dependencies highlighted)

```
loop:
            lw     x6, 0(x2)
            lw     x7, 0(x3)
            add    x8, x6, x7
            sw     x8, 0(x4)
            addi   x2, x2, 4
            addi   x3, x3, 4
            addi   x4, x4, 4
            addi   x5, x5, -1
            bne    x5, x0, loop
```

Code Snippet 2: vvadd optimized (with some RAW dependencies highlighted)

```
loop:
    lw     x6,    0(x2)
    lw     x7,    4(x2)
    lw     x8,    8(x2)
    lw     x9,   12(x2)
    lw     x10,   0(x3)
    lw     x11,   4(x3)
    lw     x12,   8(x3)
    lw     x13,  12(x3)
    add    x6, x6, x10
    add    x7, x7, x11
    add    x8, x8, x12
    add    x9, x9, x13
    sw     x6,    0(x4)
    sw     x7,    4(x4)
    sw     x8,    8(x4)
    sw     x9,   12(x4)
    addi   x5, x5, -4
    addi   x2, x2, 16
    addi   x3, x3, 16
    addi   x4, x4, 16
    bne    x5, x0, loop
```

Code Snippet 3: cmult (with some RAW dependencies highlighted)

```
loop:
    lw    x6, 0(x2)    # src0_real
    lw    x8, 0(x3)    # src1_real
    lw    x7, 4(x2)    # src0_imag
    lw    x9, 4(x3)    # src1_imag
    mul   x10, x6, x8  # real * real
    mul   x11, x7, x9  # imag * imag
    mul   x12, x7, x8  # imag * real
    mul   x13, x6, x9  # real * imag
    sub   x14, x10,x11 # dest_real
    add   x15, x12,x13 # dest_imag
    addi  x5, x5, 2
    addi  x2, x2, 8
    addi  x3, x3, 8
    sw    x14, 0(x4)
    sw    x15, 4(x4)
    addi  x4, x4, 8
    bne   x5, x1, loop
```

Code Snippet 4: bsearch (with some RAW dependencies highlighted)

```
loop:
    one:
    slli  x24, x11, 2    # idx_mid in pointer form
    add   x16, x4, x24   # idx_mid pointer in dict_keys
    lw    x17, 0(x16)    # midkey = dict_keys[idx_mid]

    bne   x9, x17, two   #  if ( key == midkey ) goto two:

    # if block starts
    add   x16, x5, x24   # idx_mid pointer in dict_values
    lw    x18, 0(x16)    # dict_values[idx_mid]
    add   x15, x2, x25   # i pointer in srch_values
    sw    x18, 0(x15)    # srch_values[i] = dict_values[idx_mid]
    addi  x13, x0, 1     # done = true
    # if block ends
```

Code Snippet 4: mfilt (with some RAW dependencies highlighted)

```
    mul   x11, x8, x9   # ridx*ncols
    add   x11, x11, x10 # ridx*ncols + cidx
    slli  x11, x11, 2   # ridx*ncols + cidx (pointer)
    add   x12, x5, x11  # ridx*ncols + cidx (pointer) for mask
    lw    x12, 0(x12)   # mask[ridx*ncols + cidx]

    # If block
    # if ( !mask[ridx*ncols + cidx] ) goto two:
    beq   x12, x0, two

    add   x12, x6, x11  # ridx*ncols + cidx (pointer) for src
    lw    x13, 0(x12)   # src[ridx*ncols + cidx]
    mul   x13, x13, x24 # src[ridx*ncols + cidx] * coeff0
    add   x23, x13, x0  # out = src[ridx*ncols + cidx] * coeff0
```