ECE 4750 Lab 2: Pipelined Processor

Carly Swetz (cms464), Raghav Kumar (rk524), Kyle Betts (kcb82)

1. Introduction

The purpose of this lab was to design two pipelined processors for the Tiny RV2 instruction set architecture. Once implemented, the processors will be able to execute straightforward C programs that do not need to deal with system calls. Because the two processors are pipelined, several transactions are able to execute at the same time. In order to create the pipelined microarchitecture, we divided the processor into different stages that each perform a certain task. A pipelined processor achieves an average of one cycle per instruction while having a short cycle time than a single-cycle processor. Furthermore, in comparison to an FSM processor, pipelining has a similar cycle time, but reduces the CPI. Yet, the tradeoff to pipelining is that more complicated control logic is required due to numerous hazards. In this lab, our two pipelined processor designs deal with these hazards with stalling and/or bypassing.

First, we completed the baseline design which is a five-stage pipelined processor that stalls when there are data hazards and squashes subsequent instructions if a branch or jump is taken. For the multiply instruction, we include our iterative multiplier from Lab 1. After completing the baseline design, we moved onto the alternative design. The alternative design is almost identical to the baseline, yet it is capable of bypassing, which removes the need to stall in some, but not all, cases. Both implementations of the processor utilize the val/rdy microprotocol for testing and memory access in order for the processor to be latency insensitive.

Overall, the alternative design of the pipelined processor performed better. Our evaluation results show that for almost all of the benchmarks provided, the five-stage bypassing pipelined processor had a lower number of cycles executed. The only outlying benchmark is the optimized version of the vector-vector add program in which both processor designs had the same number of cycles executed. The improvement in the number of cycles varies for each different benchmark and will be discussed in more detail in the **Evaluation** section of our report. Although our alternative design has better performance on most programs, tradeoffs do exist. Because our alternative design includes more hardware, there is likely to be a higher cycle time, more area taken up, and ultimately, more energy required. However, these downsides are small compared to the drastic increase in performance for most programs. Therefore, the fully bypassing pipelined processor is the better design option.

2. Baseline Design

As previously mentioned, the baseline design is a five-stage stalling processor for the TinyRV2 ISA. The subset of Tiny RV2 instructions our processor is able to handle are below.

- CSR: **csrr, csrw**
- Reg-Reg: **add, sub, mul, and, or, xor, slt, sltu, sra, srl, sll**
- Reg-Imm: **addi, ori, andi, xori, slti, sltiu, srai, srli, slli, lui, auipc**
- Memory: **lw, sw**
- Jump: **jal, jalr**
- Branch: **bne, bew, blt, bltu, bge**

The design is broken down into three files: a datapath module, a control unit module, and a parent module that connects the two together. This breakdown makes use of the *modular design principle*, and more specifically the *control/datapath split design pattern*. The pipelined processor is broken down into five stages: F - fetch instructions, increment PC; D - decode instructions, read register operands, handle jumps; X - arithmetic operations, address generation, branch comparison; M - access data memory; and W - write register file. Our processor design also includes bypass queues that enables messages to be buffered when the val/rdy interface is not ready. For instruction memory requests, we were also given a special drop unit that allows the processor to "cancel" memory requests when we need to squash instructions at the beginning of the pipeline.

The final datapath for our baseline design can be seen in **Figure 1**. The control and status signals used to communicate between the datapath and control units are shown in blue. To start off, we were provided with the datapath shown in **Figure 2** and the implementations of three basic instructions (**add, lw, bne**). We were also given the **csrr** and **csrw** instructions that enable the processor to communicate with the test source and sink. To complete the necessary processor for this lab, we used an incremental development design methodology. We implemented one instruction at a time by adding the necessary components to the datapath and control signals and logic to the control unit.

We started off by implementing the register-register arithmetic instructions. We were already provided with the **add** instruction implementation as well as an initial ALU. For the remainder of these instructions, except **mul**, all we needed to do was add the corresponding functions to the ALU. In addition, we needed to update the control signal table in the control unit with the correct ALU functions for each instruction. For the **mul** instruction, we imported our variable-latency integer multiplier that we completed in Lab 1. The processor sends the request to the multiplier in the D stage and waits for the response in the X stage. Because our multiplier takes more than 1 cycle and instructions in the pipeline may have to stall, we use val/rdy signals to know when to send data to the multiplier and when to receive the result. Lastly, to choose the correct result among the ALU, multiplier, and eventually PC+4, we added the **ex_result_sel_mux_X**.

Second, we completed the register-immediate instructions. We were provided with the immediate generator unit that already included the I-type and B-type immediates. Hence, for these instructions we needed to set the immediate to type I and the mux select for the second operand to choose the immediate input. For **addi, ori, andi, xori, slti, sltiu, srai, srli,** and **slli**, all remaining to do was assign the corresponding ALU function. For **lui**, we set the ALU function to **cp1**, which passes through the value of the second operand. Lastly, we had to instantiate a mux to select the correct first operand which we set to the PC for the **auipic** instruction.

Next, we needed to implement the remaining memory instruction, **sw**. First, we added the register named **dmem_write_data_reg_X** whose input and output is the value to be stored to memory. We also needed to create the new S-type immediate to calculate the correct memory address to store the value to. Then, because our processor uses memory messages to interact with memory, we needed to update the message for when there is a **sw** instruction. In order to differentiate between a load and store message, we use the **dmemreq_type_X** signal from the control unit and set the data bits when the message is a store.

Then, we moved on to the jump instructions, which were certainly more involved. The first thing we added was the J-type immediate to the immediate generator unit so we could correctly calculate the new address for **jalr**. Then, inside the datapath, we had to create new wires to connect the jump target addresses back to the **pc_select_mux_F**. For **jal**, in the D stage, the jump target address is calculated by the existing adder. For **jalr**, we calculate the target address inside the ALU. We then added logic to decide when the processor should redirect the PC and which target address it should select. In order to complete this logic, we added two control signals to the control signal table to signify if the instruction was a **jal** or **jalr**. Next, we instantiated a **pc_reg_X** and **+4** component to calculate PC+4 to store in the destination register. Finally, we introduced new squashing logic in the D stage since jump instructions go to a new PC unconditionally.

Lastly, we implemented the remaining branch instructions. To do this, we first created the less-than and less-than unsigned conditions for our ALU to output for branch instructions. Once we had these, we added to the existing **bne** logic in the control unit module. We created parameters for each of the remaining branch types and if the corresponding condition was set, then our processor would decide to take the branch in the X stage. As a result, the processor knows it needs to redirect the PC and the PC select mux chooses the branch target address.

Our final control signal table for the baseline design can be seen in **Figure 3**, which shows the additional signals we needed to add when implementing all of the remaining instructions. Overall, this design is a good baseline for comparison because of the latency insensitive interface for our memory system which allows for easier composition with memories and networks in later labs. Also, because it just uses stalling and squashing, there is room for improvement in the performance by incorporating bypassing like we do in the alternative design or other techniques to exploit instruction-level parallelism.

3. Alternative Design

For our alternate design we added complete bypassing to our five stage pipelined processor. Our complete bypassing enabled us to provide a value that is to be written to the register file from an earlier issued instruction to a later issued instruction that needs the value in decode. The later issued instruction would typically have to wait in deconde for the value to be written to the register file, before it could read the same value from the register file for itself. Now with our bypassing, as soon as the value is calculated it can be bypassed to the later issued instruction reducing the need to wait. Our final datapath for the alternative design can be seen in **Figure 4**.

As can be seen from the datapath, the key to our bypassing is the addition of two muxes in the decode stage right after the register file. There is one mux for each operand. The muxes are used to select where the value of each operand should be selected from. From the end of the alu stage there is a wire back to each mux. If the mux selects this value it means that the respective operand is the same register as the register that the instruction in execute is going to write to. An example of this would be back to back ADD instructions, where the second instruction has an operand the same as the destination register of the first instruction. In a similar manner, there are also wires from the end of the memory stage and the end of the writeback stage to each of the new muxes in the decode stage. These wires could pass back a calculation from the ALU or also a LW value retrieved from memory. Which stage the value is passed from depends on the sequence of instructions and spacing between dependencies. For instance for the same two ADD instructions described earlier, if there is instead a NOP between the two, the value of the first ADD would be passed from the memory stage since that is where the instruction would be located when the second ADD enters the decode stage. If there were two NOPs between the ADDs, the value would be passed from the writeback stage due to similar logic.

The choice to implement bypassing from every stage was intriguing since the bypassing added very little logic, just two muxes and some wires. With this little addition, we were able to remove a lot of data hazard stalling logic due to dependencies. The only remaining data hazard stalling logic is if there is a dependency between a LW and an instruction immediately after because the value of LW will not be determined until the instruction is in memory, causing the dependent instruction to stall in deconde until the value is determined. We were able to remove all other stalling logic which dramatically reduced our CPI as can be seen from the evaluation part at the expense of only a small increase in cycle time potentially.

4. Testing Strategy

Due to the complexity and extensiveness of this lab, we strictly adhered to an incremental, test driven strategy and wrote our tests in a way that would reliably test all individual instructions. From a high level perspective, the test suite was divided into seven broad categories, each tackling one of register-register, register-immediate, branch, jump, csr and memory instructions and an additional category testing the ALU functions. For all the instructions other than SW (store word), JAL (Jump and Link) and JALR (Jump and Link Register) we were provided with templates that tested out the instructions under various stress conditions. Using nops, one could individually test different bypass paths before finally testing the instruction itself. The templates also enabled us to test our instructions in scenarios such as when either of the source registers are the same as the destination register which were important edge cases from the processor's standpoint. The real challenge in the testing for this lab was to write new, assembly code generating templates that were parameterized which would enable them to generate different code sequences for the same instruction. Since we weren't provided any templates for sw, jal and jalr, as the verification engineer most of my time went in writing these templates. The template for **sw** ended up being slightly more trivial because we were provided with a working load instruction. The main idea with sw was to store an arbitrary value at an arbitrary memory location and then leverage the load instruction to load the value from the same base and offset. By inserting both the store and the load in the same template, we were able to parametrize the base and the offset since both the memory instructions have a similar method of calculating the base address. Using the load instruction test format, we made similar tests that stressed various bypass paths before finally testing the store instruction itself.

The **jal** and **jalr** instructions were far more complex to write templates for. This was due to the fact that each successive test depended on the previous one for the correct result of the link address and the jump address. For each new test case (a new function in the inst_jal or inst_jalr file), the instruction memory would start from 0x200 but the next test (in the same function) would start from a different memory address that depended on the ending of the first test. In addition, the start of the new test would also vary with the number of nops one decided to test in the preceding instruction. To tackle this variability, we came up with a small formula that calculated the result of each test (the jump and the link addresses) and used it in the jal and jalr test files. Using that framework, one could just vary the number of nops for each test and subsequently add a new test case with minimal effort. This went a long way in speeding up our testing process because manually calculating the link and jump addresses would not be feasible. Regarding the structure of the jump tests and templates, we wrote 3 templates for the jal instruction and 6 templates for the jalr instruction. The templates varied in the number of jump instructions they had (1,2 and 3 for both jal and jalr) while jalr also had 3 more templates to enable jumping with offset. Using these templates, we ran 4 different types of tests for jal (1,2,3 jumps and random) and 7 different types of tests for jalr. The three jump case also tested a backward jump which we thought was a great corner case to check for. By using the three jump case in our random tests, we ensured that the processor was being stressed and tested adequately.. In the random test it was imperative to limit the maximum number of loops or the maximum number of nops to avoid the processor from hitting the 10K cycle limit which is why we chose to limit the loops to around 20 for jalr and 25 for jal. We also took an extra measure and tested all our instructions with a random set of source, sink and delays. In the ***ProcBaseRTL_ *_test.py*** files, we added a small loop after each instruction to achieve this and are thus confident that our processor can handle a variety of delay scenarios.

Finally, we tested our ALU's functionality using a variety of input operands. All instructions were tested for negative and positive operands (where applicable) and various combinations of the two input operands were brainstormed. In the end, we are convinced that our ALU testing suite tests the implementation rigorously, building our confidence in the functionality of the processor as a whole.

5. Evaluation

Just looking at Baseline stats and the nature of the algorithms, we understand a few important things about how the benchmark programs run on the processor. The results of our baseline tests are in **Figure 5** and there is a significant decrease in the cycles per instructions as we go from the vvadd-unopt to the vvadd-opt. This is because vvadd-opt unwraps the for-loop that vvadd-unopt relies on to loop over the elements of the array and do the addition. This greatly reduces the number of instructions squashed after a branch not equal (there are essentially almost no branch related squashes in the optimized version because there is only a single bne execution) and thus saves the processor a lot of cycles. This kind of optimization is often used by compilers to reduce the total number of cycles as un-looping a loop almost always reduces the number of squashes by reducing the number of branch instructions. The improvement will be more drastic with the size of the two arrays. In our particular evaluation benchmark, the vvadd was only done on arrays with a length of 4 which is pretty trivial. With increasing array lengths, the difference in CPI would become much larger courtesy the higher number of squashes in the unoptimized version. The CPI drastically falls from 2.22 to 1.34 when the vvadd-unopt is run on the the alternate version and this is primarily due to the alternate design's advantageous bypass paths. As one can see from the vvadd_unopt assembly code, the branch (bne) instruction in the end has a data dependency with the addi just above it. This would cause the baseline pipeline to stall when the branch is the D stage and the addi is in the X stage and the pipeline would stay stalled until the addi instruction completes writeback. This would increase the cycle count by 3 cycles in every loop and the problem is compounded by the fact that the number of stalling cycles will increase linearly with the size of the array. On the other hand, in the alternative design, such data dependencies would be resolved using the bypass path in the same cycle because the X stage would be able to forward the value of the x5 register back to the branch instruction in the D stage. Thus we get rid of those inefficient stalls. It should be noted however that the data hazard between the second lw and the first add in vvadd-unopt would give rise to a stall in both the design since bypassing does not eliminate memory access induced stalls. The fact that both the baseline and alternative implementation gave a CPI of 1.11 for vvadd-opt tells us that there weren't many data hazards and hence stalls in vvadd-opt to begin with. Since the baseline and the alternative design differ in their handling of data hazards (stalls vs forwarding), the absence of data hazards means that the vvadd-opt gives the same performance on both the designs.

The cmult algorithm does not show vast improvement in CPI between the two designs primarily because the data hazards in the program assembly code are due to the load and the mul instructions. We know that load-use dependencies cannot be resolved using bypassing and that is exactly what this benchmark aims to demonstrate. The marginal improvement in CPI most definitely comes from the couple of non-memory data hazards such as the last mul and the first add which the alternative design is able to resolve without a stall.

Next looking at the Bsearch algorithm we see a large performance increase in the alternate design. There are lots of dependencies again which is where our alternate design excels. The dependencies between the ADD/LW and ADD/SW both result in 3 fewer cycles each iteration, which really adds up. Finally, in the MFilt algorithm, we see that the alternate design is once again superior. Once again the repetition of dependencies adds up and results in a much lower CPI using bypassing.

Figure 1: Baseline Design: Five-Stage Stalling Processor Datapath

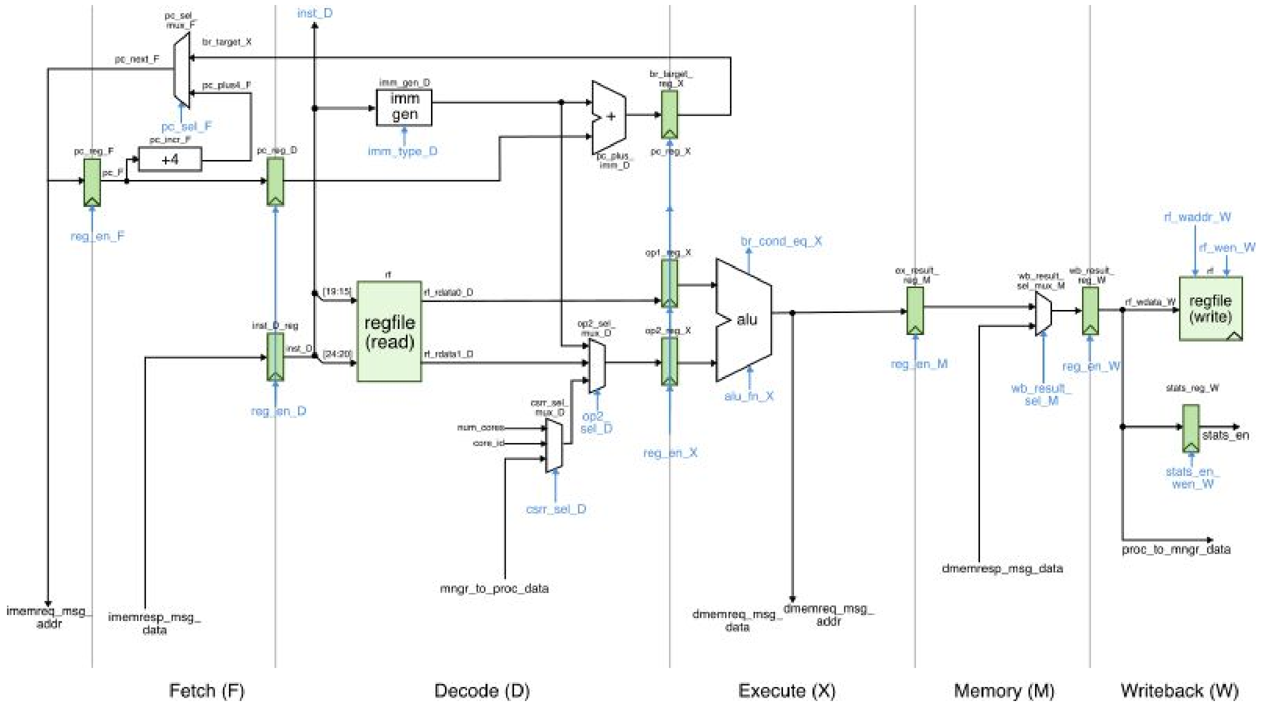


Figure 2: Initial Baseline Design Provided to Students

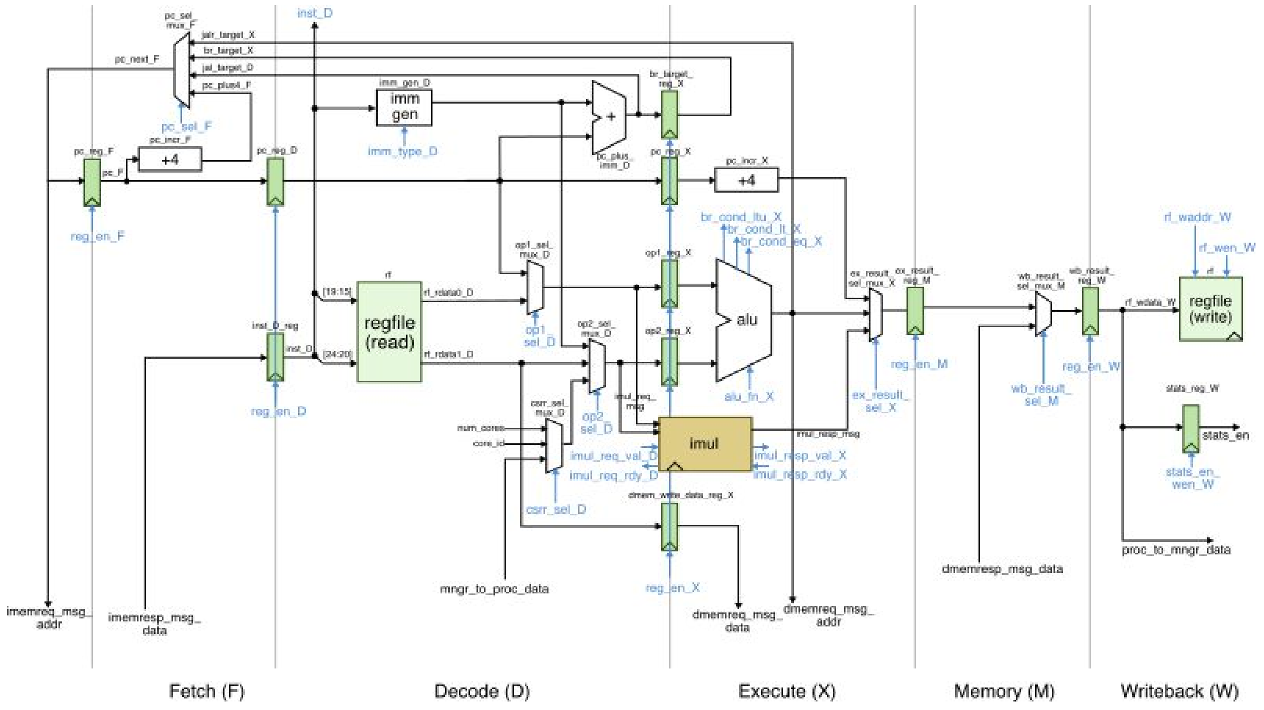
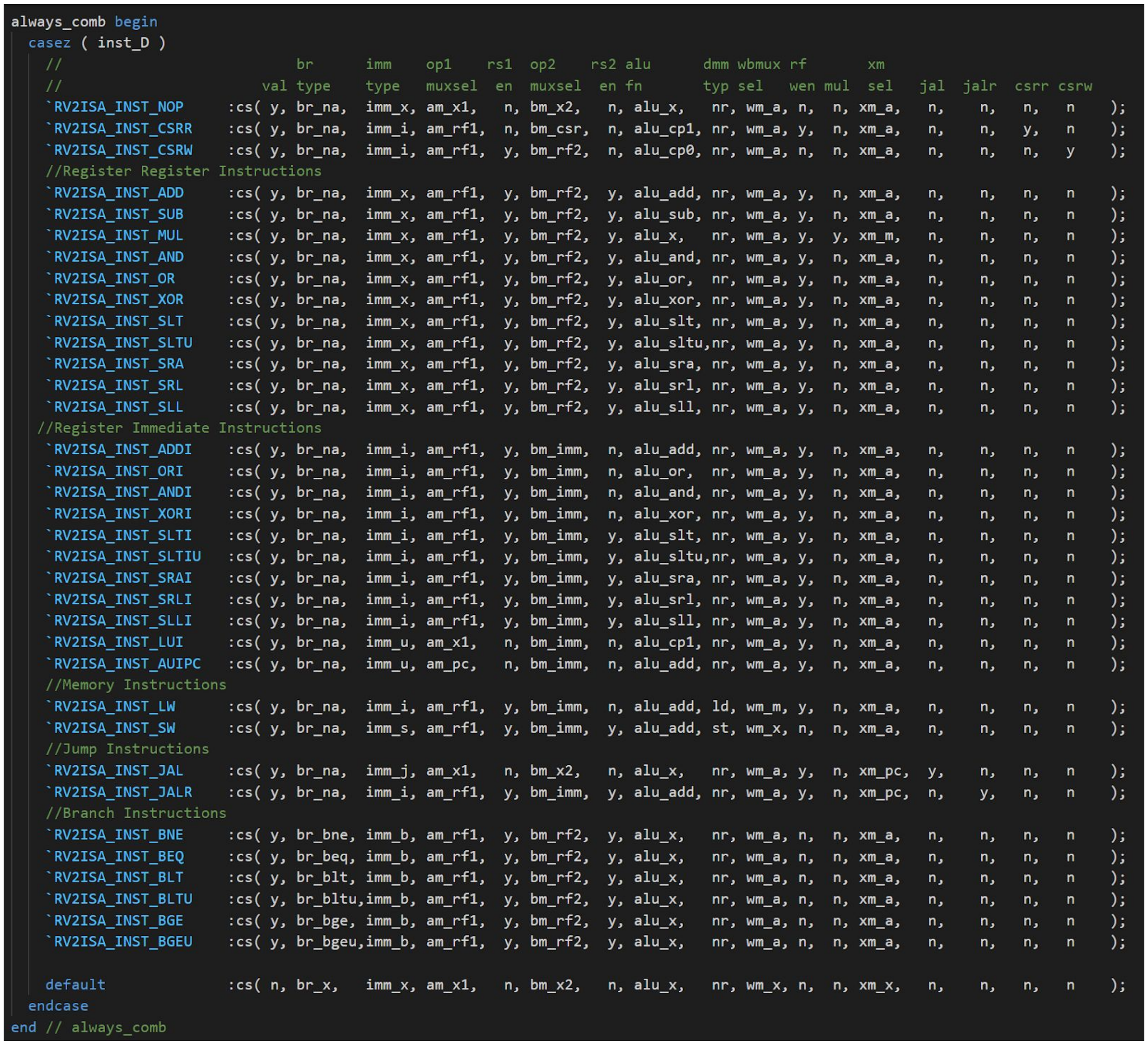


Figure 3: Baseline Design: Control Signal Table



```
always_comb begin
  casez ( inst_D )
    //                           br      imm    op1    rs1  op2    rs2 alu      dmm wbmux rf       xm
    //                       val type    type   muxsel  en  muxsel  en fn       typ sel   wen mul  sel   jal  jalr  csrr csrw
    `RV2ISA_INST_NOP     :cs( y, br_na,  imm_x, am_x1,   n, bm_x2,   n, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_CSRR    :cs( y, br_na,  imm_i, am_rf1,  n, bm_csr,  n, alu_cp1, nr, wm_a, y,  n, xm_a,   n,    n,   y,   n    );
    `RV2ISA_INST_CSRW    :cs( y, br_na,  imm_i, am_rf1,  y, bm_rf2,  n, alu_cp0, nr, wm_a, n,  n, xm_a,   n,    n,   n,   y    );
    //Register Register Instructions
    `RV2ISA_INST_ADD     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_add, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SUB     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_sub, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_MUL     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, y,  y, xm_m,   n,    n,   n,   n    );
    `RV2ISA_INST_AND     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_and, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_OR      :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_or,  nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_XOR     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_xor, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SLT     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_slt, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SLTU    :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_sltu,nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SRA     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_sra, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SRL     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_srl, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SLL     :cs( y, br_na,  imm_x, am_rf1,  y, bm_rf2,  y, alu_sll, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
   //Register Immediate Instructions
    `RV2ISA_INST_ADDI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  n, alu_add, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_ORI     :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  n, alu_or,  nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_ANDI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  n, alu_and, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_XORI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  n, alu_xor, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SLTI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  n, alu_slt, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SLTIU   :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  y, alu_sltu,nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SRAI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  y, alu_sra, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SRLI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  y, alu_srl, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SLLI    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  y, alu_sll, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_LUI     :cs( y, br_na,  imm_u, am_x1,   n, bm_imm,  n, alu_cp1, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_AUIPC   :cs( y, br_na,  imm_u, am_pc,   n, bm_imm,  n, alu_add, nr, wm_a, y,  n, xm_a,   n,    n,   n,   n    );
    //Memory Instructions
    `RV2ISA_INST_LW      :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  n, alu_add, ld, wm_m, y,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_SW      :cs( y, br_na,  imm_s, am_rf1,  y, bm_imm,  y, alu_add, st, wm_x, n,  n, xm_a,   n,    n,   n,   n    );
    //Jump Instructions
    `RV2ISA_INST_JAL     :cs( y, br_na,  imm_j, am_x1,   n, bm_x2,   n, alu_x,   nr, wm_a, y,  n, xm_pc,  y,    n,   n,   n    );
    `RV2ISA_INST_JALR    :cs( y, br_na,  imm_i, am_rf1,  y, bm_imm,  y, alu_add, nr, wm_a, y,  n, xm_pc,  n,    y,   n,   n    );
    //Branch Instructions
    `RV2ISA_INST_BNE     :cs( y, br_bne, imm_b, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_BEQ     :cs( y, br_beq, imm_b, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_BLT     :cs( y, br_blt, imm_b, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_BLTU    :cs( y, br_bltu,imm_b, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_BGE     :cs( y, br_bge, imm_b, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );
    `RV2ISA_INST_BGEU    :cs( y, br_bgeu,imm_b, am_rf1,  y, bm_rf2,  y, alu_x,   nr, wm_a, n,  n, xm_a,   n,    n,   n,   n    );

    default              :cs( n, br_x,   imm_x, am_x1,   n, bm_x2,   n, alu_x,   nr, wm_x, n,  n, xm_x,   n,    n,   n,   n    );
  endcase
end // always_comb
```

Figure 4: Alternative Design: Five-Stage Bypassing Processor Datapath

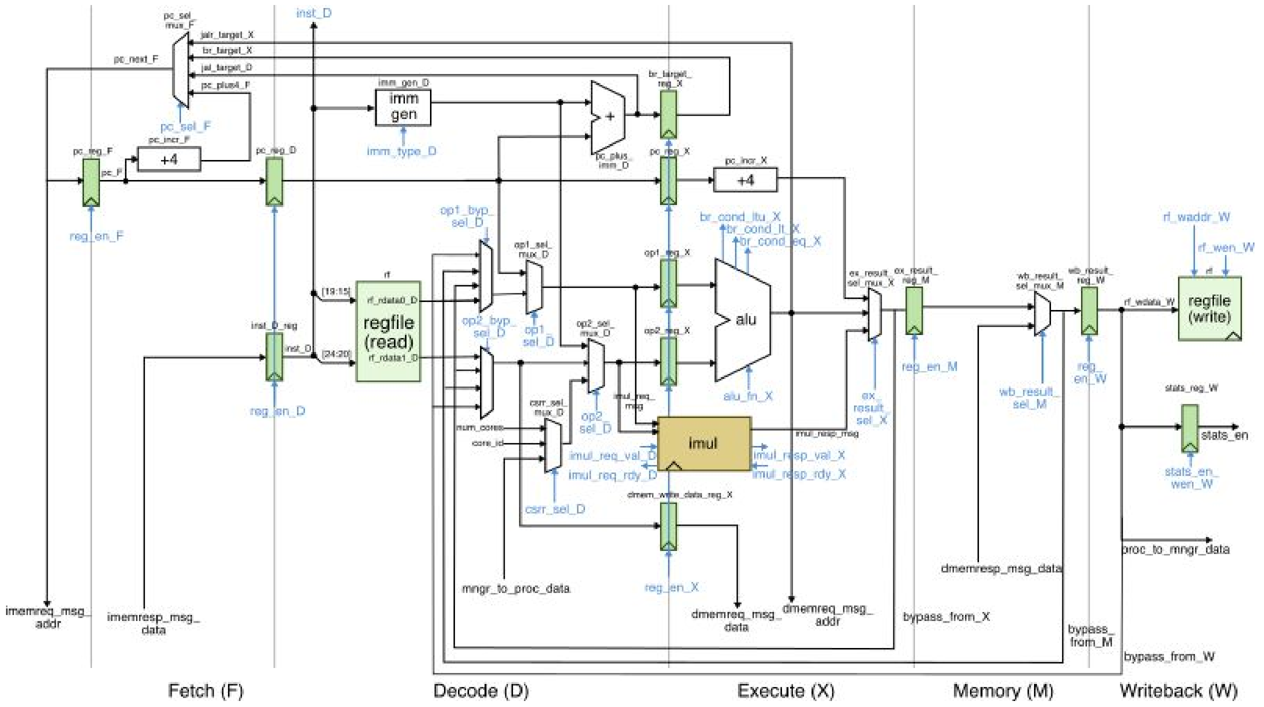
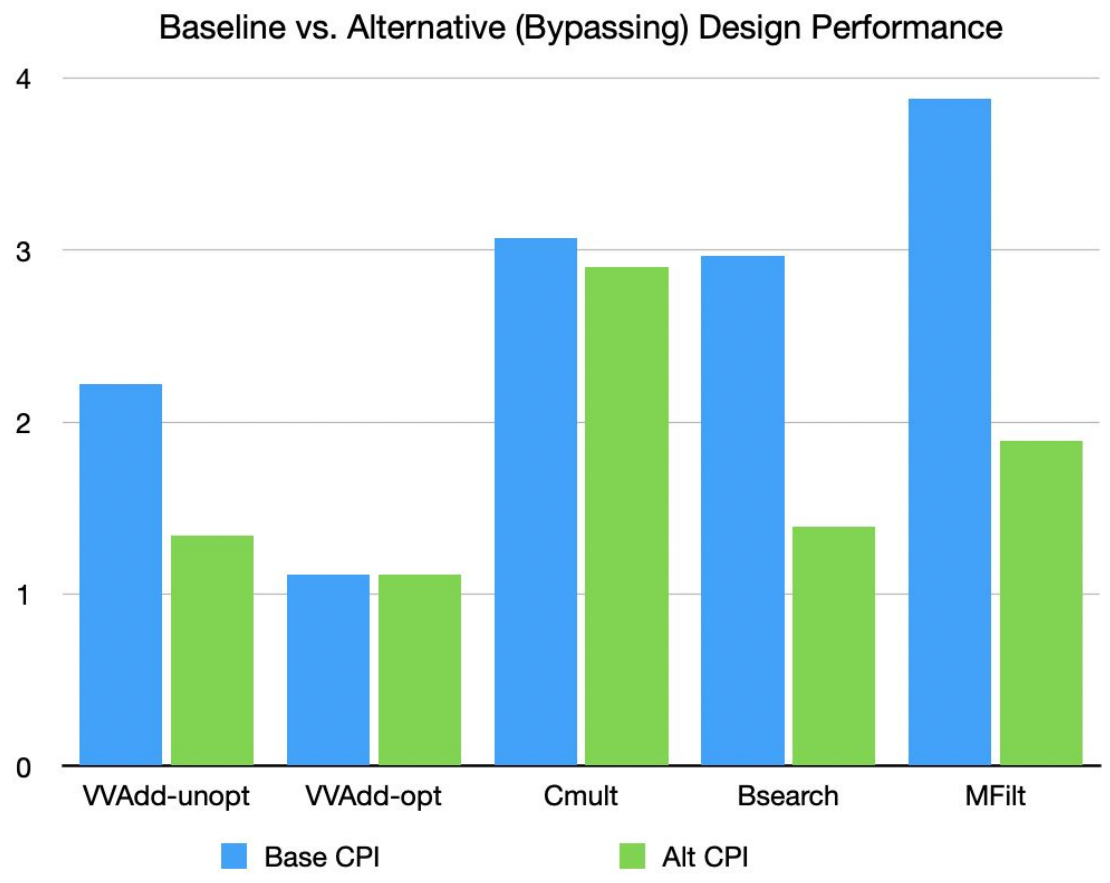


Figure 5: Evaluation Results

Test	Base # Cycles	Base CPI	Alt # Cycles	Alt CPI
VVAdd-unopt	2014	2.22	1214	1.34
VVAdd-opt	589	1.11	589	1.11
Cmult	5248	3.07	4948	2.90
Bsearch	4537	2.97	2116	1.39
MFilt	5241	3.88	2546	1.89



Role and Task Table

Carly Swetz	**Kyle Betts**	**Raghave Kumar**
RTL Design Engineer (architect)	RTL Design Engineer	RTL Verification Engineer
● Implemented the following instructions: **slt, sltu, sra, srl, sll, slti , sltiu, srai, srli, slli, sw, jal, jalr** ● Debugged the above instructions if they were not passing our tests ● Assisted in merging the necessary files to combine all instructions into our baseline design ● Brainstormed and drew out the updated datapath for our alternative design ● Helped implement the alternative design by adding the necessary wires, muxes, and control logic ● Debugged the alternative design when bypassing was not working as expected ● Wrote the tests inside ***ProcDpathCompenents RTL_test.py*** to confirm our additional ALU functions were working correctly ● Went through line tracing in detail to debug an issue with the stalling logic for our multiplier unit (along with Kyle) ● Collected the statistics for the evaluation section ● Completed the introduction and baseline design sections of the report	● Created the svg images for the lab report ● Implemented the following instructions: **add, sub, and, or, xor, addi, ori, andi, xori, bne, beq, blt, bltu, bge, bgeu** ● Debugged the above instructions if they were not passing our tests ● Assisted in merging the necessary files to combine all instructions into our baseline design ● Brainstormed and drew out the updated datapath and stall logic for the alternative design ● Helped implement the alternative design by adding the necessary logic, muxes, and control logic ● Debugged the alternative design when bypassing was not working as expected ● Merged the test files written by Raghave to the branch with all datapath and control to perform the testing ● Went through the line tracing in detail to debug the stalling logic for our multiplier unit (along with Carly) ● Wrote the Alternate Design section and assisted with the Eval section for lab report	● Implemented all tests for register-register, register-immediate, branch, jump and memory instructions. ● Integrated the multiplier into the baseline processor. ● Brianstormed edge cases and came up with the 'random delay loop' test which adds random delays to all tests that the processor is tested on. ● Wrote new templates for the store word (sw), jump and link (jal) and jump and link register (jalr) instruction. ● Designed a framework (wrote a formula) to add multiple tests using the jump instruction template. Parametrized the number of nops one could insert into the jump test cases, allowing for a variety of jump test cases. ● Analysed the performance of the baseline and the alternative design on the benchmarks, doing a deep dive into the assembly code for each benchmark. ● Wrote the test section and laid down the skeleton for the eval section for the report.