PROJECT REPORT ENSC 403 Spring 2020

Paige Tuttösí, 301135559

RISC-V instruction sets contain memory-reference instructions, arithmetic-logic instructions, and conditional branching instructions. In this project a RISC-V processor was designed and implemented in order to take a subset of these instructions through datapaths of varying limitations and complexities. Several processor complications are examined including data hazards, pipelining and branching prediction, and experiments are conducted in order to asses the benefits and pitfalls of a number of solutions to these problems.

I. INTRODUCTION

THROUGHOUT the spring 2020 semester I have worked on constructing several components culminating in a fully pipelined RISC-V CPU. This report commences with a course overview presenting each of the labs, their components goals. The CPU was built from the ground up beginning with the instruction fetch unit and ending with branching prediction; the CPU has evolved through several stages of complexity. Each of these stages will be briefly outlined within this report and and several experiments are conducted in order to compare performance for various methods of implementation at the final stage of the CPU construction. These experiments include comparisons of fully and simplified designs, comparisons of branch prediction methods and their respective accuracy and an analysis of data hazard specifically as they relate to the use of a forwarding unit. Finally, suggestions are given as to other possible CPU designs and how their implementation and performance compares to the CPU constructed for this project.

II. COURSE OVERVIEW

Each lab consisted of one or more components of the CPU. Beginning in lab 4 these components were pipelined together to form the CPU with added complexity through to the final project. [1]

A. Lab 1

In lab 1 the Instruction Fetch (IF) unit was built (Fig. 2.). The IF unit consists of a Program Counter and a Instruction Memory unit. This lab also introduced the creation of a Memory Initialization file *.mif* file allowing specification of data values in instruction memory (for building .mif files see Appendix A, pg.9).

Program Counter

The Program Counter (PC) has one 8 bit register which, on each clock cycle, will either output an 8 bit input address or increment the previous output from the PC depending on the increment (inc) signal (Fig.1.). The load signal (ld) must also be active high in order to load the next PC (this will be used later in stall implementation pg.3)

[1] All components contain an asynchronous clear

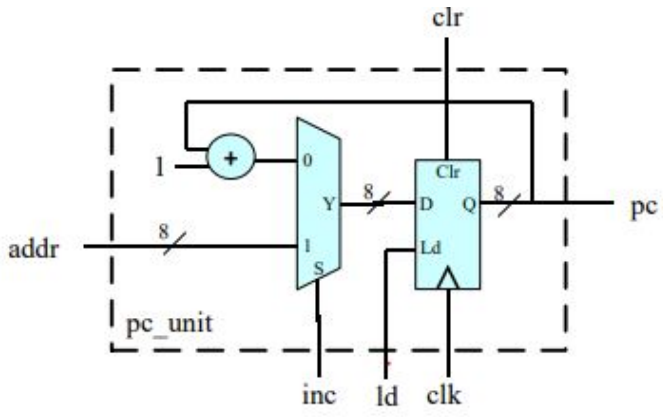

Fig. 1. Program Counter Diagram.

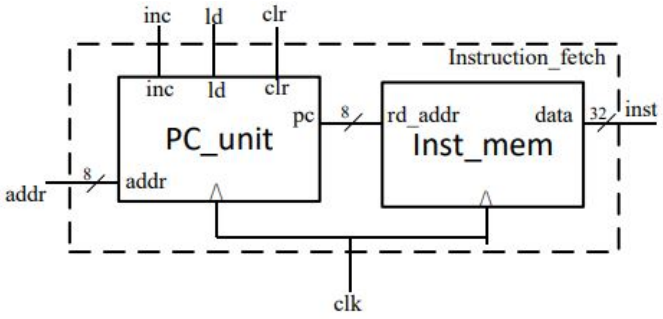

Fig. 2. Instruction Fetch Diagram.

Instruction Memory Unit

The synchronous Instruction Memory Unit (inst_mem) reads the 8 bit output address and retrieves the data from that location in the given *.mif* file. The output is a 32 bit instruction to be processed by the CPU. This component was generated through Quartus' plugin manager wizard.

B. Lab 2

Lab 2 consisted of the Arithmetic logic unit (ALU) components of the CPU. A 64 bit asynchronous ALU consisting of an adder/subtractor, logical left shift, logical right shift, arithmetic right shift, AND, OR and XOR was built and was tested on an FPGA board (Fig. 3., Table I). The ALU returns a result, carry and zero signal.

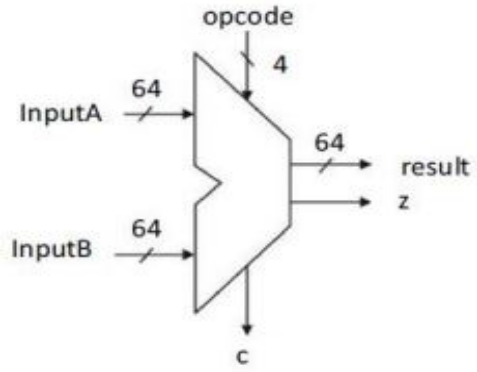

Fig. 3. Logic Unit diagram.

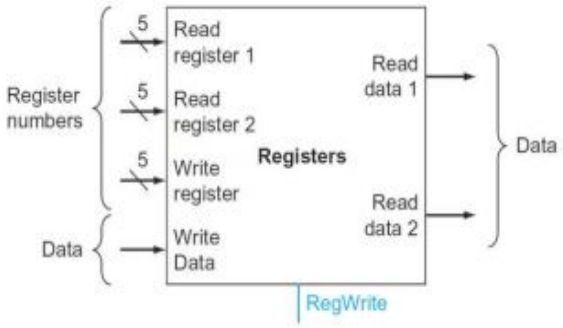

Fig. 4. Register File diagram.

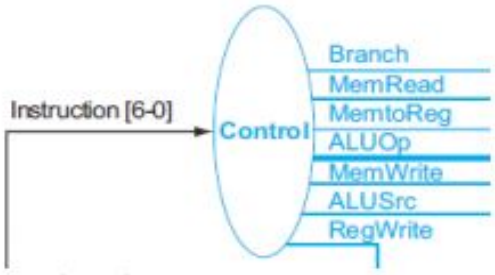

Fig. 5. Control Unit diagram.

TABLE I
ALU OPCODE AND OPERATIONS

Instruction	Opcode	Description
add / addi	0000	result = InputA + InputB
sub	0010	result = InputA – InputB
sll / slli	0001	result = InputA << InputB
xor / xori	0100	result = InputA xor InputB
srl / srli	0101	result = InputA >> InputB (logical)
sra / srai	0011	result = InputA >> InputB (arithmetic)
or / ori	0110	result = InputA OR InputB
and / andi	0111	result = InputA AND InputB

Adder/Subtractor

The adder/subtractor (addsub) contains a full a 1 bit full adder and returns a result and carry signal. To compute subtraction the 2's complement is taken of the second input and added using the one bit full adder.

C. Lab 3

For lab 3 a 32 bit decoder and a register file were built.

Decoder

The decoder is asynchronous and takes a 32 bit instruction from the IF and breaks it down into several signals depending on the instruction type. The instruction type is determined by the opcode, the last 7 bits in the instruction. The output of the decoded bits for each instruction type can be seen in Table II.

Register File

The register file is comprised of 32 registers accessed by a 5 bit register address (Fig. 4.). On each clock cycle the register file reads two register addresses and outputs two sets of data, 64 bits each. Given that the write enable is active and the 5 bit register address is provided, one register can be written to per clock cycle.

TABLE II
DECODED INSTRUCTION BITS

Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

D. Lab 4

In Lab 4 the previous components the initial datapath was constructed only allowing sequential instructions. The datapath diagram can be seen in Appendix B, pg 10. The inputs to the datapath are the same as those for the IF unit and address may be provided. In order to facilitate the data path, a Control Unit, ALU Control and Immediate Generator were designed. Instructions must not include subsequent write reads to the same registers as at this point data hazards are not handled by the datapath (see pg.3 for more on data hazards).

Control Unit

The Control Unit input is the 7 bit opcode that determines the output enable signals for the datapath (Fig. 5.).

ALU Control

The ALU Control determines the 4 bit ALU opcode which instructs the ALU which operation to perform. The inputs are the ALUOp from the Control Unit and the Funct7/3 from the decoder. Table III gives signal values and resulting opcodes.

Immediate Generator

The Immediate value is determined by the decoder as per Table I. In order to perform logic operations in the ALU the Immediate Generator (imm_gen) appends 32 bits of zeros to a positive 2's complement 32 bit Immediate and ones to a negative 2's complement Immediate.

E. Lab 5

Lab 5 introduces control flow through branching to the datapath (Appendix B. pg. 10). This datapath is partially

TABLE III
ALU CONTROL UNIT SPECIFICATION

ALUOp	Operation	Funct7	Funct3	ALU Action	ALU Opcode Input
00 (ld)	Load word	XXXXXX	XXX	Add	0000
00 (sd)	Store word	XXXXXX	XXX	Add	0000
01 (beq)	Branch if equal	XXXXXX	XXX	Sub	0010
10/11 (R/I-type)	Add	0000000	000	Add	0000
10 (R-type)	Sub	0100000	000	Subtract	0010
10 /11(R/I-type)	And	0000000	111	AND	0111
10 /11(R/I-type)	Or	0000000/na	110	OR	0110
10/11 (R/I-type)	Xor	0110011/na	100	Xor	0100
10/11 (R/I-type)	Sll	0000000	001	Sll	0001
10 /11(R/I-type)	Sra	0010011 / 0100000	101	Sra	0011
10 /11(R/I-type)	Srl	0000000	101	Srl	0101

pipelined. The datapath is now self contained with no inputs and was tested by converting a loop of C code into assembly that was in turn written in instructions into the *.mif* (see mif generator pg. 9). Data hazards are still not handled at this point so read and write registers should be appropriately dispersed.

F. Final Datapath Versions

The final lab consisted of creating several versions of a fully pipelined datapath (Appendix B. pg 10). Each datapath was built and comparisons were made between each of their performances. More in depth descriptions of the datapaths and their differences can be found in the experiment section.

Datapath 1

Datapath 1 is fully pipelined and includes a stall to during branching which clears the datapath with NOOPS (no operations) for the four preceding clock cycles (PC, inst_mem, IF and those currently coming into the register, immgen and control unit) while the branch address is resolved which happens in the execution cycle. Data hazards still must be avoided.

Datapath 2

Datapath two finally handles data hazards by implementing a Forwarding Unit. This component at the execution stages compares the read registers to the write registers exiting the execution stage and the data memory stage, if they are the same their associated data is read as the ALU signal rather than what is read from the register. This way we no longer have to wait two clock cycles to read from a register and the two subsequent instructions can be read directly from the datapath.

Datapath 3

Datapath 3 is the first of the datapaths implementing branch prediction. Static Branch prediction: never branch, is used. See experiment (pg. 5) for further description of prediction methods.

Datapath 4

Datapath 4 is the first of the datapaths implementing branch prediction. Static Branch prediction: always branch, is used. See experiment for further description of prediction methods.

Datapath 5

Datapath 3 is the first of the datapaths implementing branch prediction. Dynamic Branch prediction:2 bit dynamic prediction, is used. See experiment for further description of prediction methods.

III. METHODOLOGY

Beginning in Lab 4 we combine the previously constructed components into CPUs of increasing complexity. The specifications, benefits, and drawbacks of each design are outlined here. Each datapath is built off the previous datapath, so it can be assumed once one datapath is fully pipelined all subsequent datapaths will be similarly pipelined. This applies to all aspects of the datapaths with the exception of branch prediction units, where each datapath implements a different version of branch prediction.

Timing

We determine our period for our clock cycles based on our maximum delay. This occurs during subtraction, as it has a bit flip and an addition. Our longest case delay in subtraction occurs when when we subtract 0 from 1 with a carry on each level. Because we are using 2's complement this means we start with 11111111 – 11111111, so when the bits are flipped for subtraction we get 11111111+ 00000001. The '1' in the LSB starts the carry. This delay was is approximately 53 ns, so to be sure that we can handle this delay we set our period to 55 ns.

Simple Datapath

The simple datapath (Lab 4 and Lab 5) is partially pipelined and has branching capability, but is unable to handle data hazards and requires long stall times when waiting for the branch address to resolve. Both these simple data paths and the fully pipelined datapaths up until the branch prediction use stalls to hold the pipeline until a branching instruction is resolved. In these cases every time a branch instruction is encountered at the execution stage a signal is set through the pipeline to hold the instructions until the branch is resolved and the next PC address is available. The stall signal resets all of the stages upstream to zero and gates all the stages downstream so that no incorrect data is written into memory.

Fully pipelined Datapath

The next datapath (Datapath 1) is fully pipelined, but is unable to handle data hazards and although it still requires a stall when waiting for the branch address to resolve, this becomes easier to implement by flushing registers at pipeline stages. A fully pipelined datapath divides its instruction over component stages controlled by registers on rising edge clock cycles. Rather than waiting for an entire instruction to complete a pass though the datapath, several instructions are running at different stages along the datapath simultaneously. Figure 6 shows a laundry analogy of a non pipelined vs pipelined laundry process where if we are not using a pipeline we wait for the entire laundry process to be complete before starting another load, whereas if we use a pipeline we have a load being worked on in each of the stages of the laundry process. The preliminary overlapping pipelined processes for our datapath can be seen in Figure 7. Although pipelining may slow down the execution of individual instructions the benefit of a fully pipelined datapath is increased *throughput*. The reduction in speed is a result of each stage requiring a delay to match that of the slowest stage in the pipeline. Using the laundry analogy is washing takes 5 minutes, drying takes 10 minutes and folding takes 15 minutes, whereas, if we pipeline we

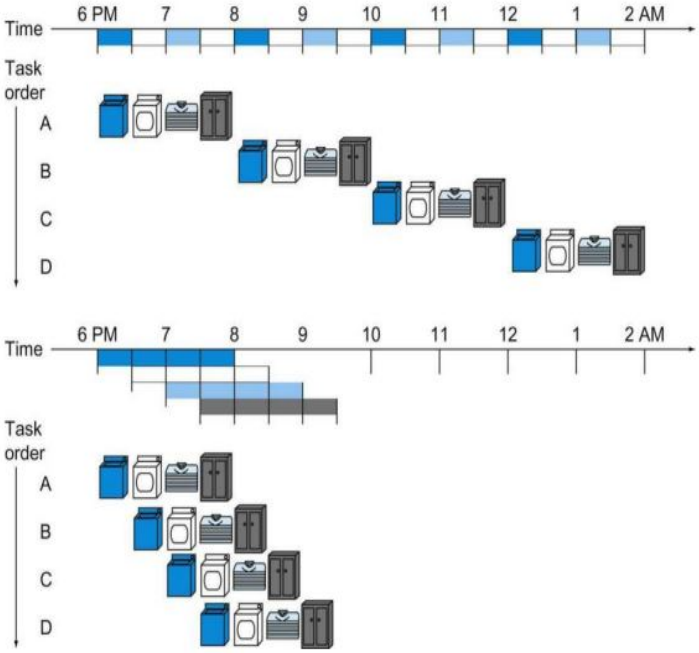

Fig. 6. Laundry Pipeline Analogy.

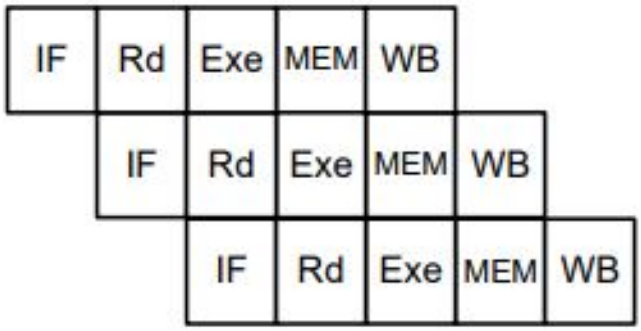

Fig. 7. Datapath Pipeline example.

require 15 minutes for each stage as we must wait for the slowest stage to finish before we can move each of the loads through the pipeline. In this case a single load of laundry would require half an hour to complete with no pipeline and would take 45 minutes to complete with a pipeline, however, with no pipeline it would take 150 minutes to complete 5 loads of laundry compared to only 105 minutes with the pipelined version.

Data Forwarding Datapath

It is at this stage in our datapaths that we solve the data hazards.

Data hazards, in the case of these pipelines, occur when an instruction attempts to read from a register that is the write register of a recent instruction (the number of clock cycles that a data hazard will persist depends on the implementation of the pipeline). The desired calculation in the ALU would use the data from the previous instruction, however, until this instruction reaches the end of the pipeline we are not able to read this data. In previous datapaths we made sure to use several registers waiting a sufficient amount of clock cycles between reading and writing to the same register. For this datapath a forwarding unit is used in order to access write data which has yet to be written to a register and is currently residing within either the execution or memory stage of the datapath. The forwarding unit takes the addresses of the read registers of the current instruction as inputs and compares these with the addresses of the write registers at the execution and memory stage. If the read address is equal to the execution write address the accompanying write data is used as an input to the ALU in place of the respective register. If the write address is the same at the memory stage, but not at the execution stage then this write data will be used as the input to the ALU. In the case where the second input is the *immediate* signal, this will always be used regardless of the read register, and if the read register is reg0 we will always use reg0, even if one of our write addresses is reg0. Reg0 is never to be written to and instead always holds all zeros, in the case where our write register is set to reg0 this means we are executing an instruction that does not write back to a register (eg. store or branch) and the write data will likely contain garbage that we do not want contaminating our ALU. Because reg0 will always hold the same value we can safely ignore the forwarding when it is used as a read register.

Once the forwarding unit is implemented the datapath can now read and write to the same register in subsequent instructions and there is no need to be cautious of register use, aside from choosing appropriate temporary registers.

Branch Predictor Datapath

In the previous datapaths, when a branch is predicted, the pipeline is stalled in order to resolve the branching address. This can cause significant delays in CPU (for the previous datapaths the delay is 3 clock cycles for every branch instruction). Branch prediction attempts to reduce the delay caused by branch resolution through predetermination of a branch address. There are two methods of branch prediction, static and dynamic prediction. Static prediction will give the same prediction independent of any previous branching and predictions. Dynamic prediction considers previous branches (taken or not taken) and will update the prediction depending on these values. Always branch, and never branch were the two static methods chosen for this analysis, and two bit dynamic branching is our dynamic choice (see experiment pg 5. for further explanations of the branch prediction techniques). Assuming an efficient prediction method is chosen, branch prediction will speed up the CPU, stalls will only occur when a hazard is detected (the incorrect address was taken), and all other instructions whether they be branching or not will run sequentially saving 3 clock cycles per correct prediction.

Analytical Methods

By constructing the CPU one component at a time one is able to gain a more thorough understanding of each of the individual components, and how they contribute to the final pipeline. Furthermore, the designer has a higher level of confidence in each of the individually tested components and will likely require less time to construct and test the final datapath. Once the five versions of the final datapath are constructed experiments are run to assess each of their performance. Throughout the experiments each of the datapaths are

assessed and compared for complexity and performance. Ease of implementation/complexity will be taken into consideration to scale against the increased performance. Quartus Prime 18.1 Lite edition was used to generate *.vwf* timing wave forms; these wave forms are the basis of the analysis.

IV. Experimental Setup

This experiment assesses the performance of the the final four datapaths, we choose the final four as these are the only datapaths where we are not concerned with data hazards (however we use the datapath from lab 5 as a baseline to test pipeline performance). These datapaths are: fully pipelined with data forwarding, fully pipelined with data forwarding and static branch prediction *never taken*, fully pipelined with data forwarding and static branch prediction *always taken*, fully pipelined with data forwarding and dynamic branch prediction *two bit dynamic prediction*. Each of the datapaths evaluated based on complexity, speed, throughput and, for the branch prediction, accuracy. Each of the datapaths will be run on a set of three *.mif* files in order to test two extreme and one moderate set of instructions, these will be the benchmarks for analysis.

Complexity

First each datapath is assessed for a qualitative level of complexity. These levels are derived relatively based on the other datapaths starting with least complex, the fully pipelined datapath with forwarding, to most complex, the datapath using dynamic branching prediction.

Speed

Speed refers to the average number of clock cycles for one instruction to move through the datapath from PC to Write from the data memory. A minimum, maximum and median is given for each of the datapaths.

Throughput

The datapath will be run for 20ns, the number instructions that have been completed at that time will be counted for the throughput of that datapath.

Prediction Accuracy

For each datapath, the total number of branch instructions will be counted for the instruction set of interest [2], this will be compared to the number of incorrect branch instructions in order to give a proportion. The number of incorrect instructions is collected internally by the PC the *count* signal is incremented each time a hazard is flagged (see section on branch prediction and hazards pg.4).

Benchmarks

The three *.mif* files were generated by the *mif-file-generator*.py* (see section on mif files.. pg). Each contains a set of instructions to be stored in the Instruction Memory and each tests a different aspect of branch prediction. The first is a moderate set of instruction which has been used throughout the semester to test the datapath. These instructions thoroughly test each type of instruction and all possible branching combinations, as well, at the end of the instructions of interest there is a test for the C code below:

[2]each .mif file is filled with 256 instructions, however, we only are testing or care about a known subset of these instructions, the rest are padded to fill the file)

```
//test branching
int a = 2; int b = 3;
for(int i = 0; i < 3; i++){
  if(a == b)
    a--;
  else
    b--;
}
```

The next instruction set fills registers with 75% equal values and 25% unique values. Thirty random combinations of these registers are compared through Branch if Equal (BEQ) and most will branch. We expect the static algorithm *always taken* to perform well in this case. The last instruction set fills registers with 25% equal values and 75% unique values. Thirty random combinations of these registers are compared through Branch if Equal (BEQ) and most will not branch. We expect the static algorithm *never taken* to perform well this case.

Prediction Methods

Branch prediction speeds up the pipeline by potentially avoiding stalls encountered on branch instructions. Without branch prediction a stall is needed to flush the pipeline following a branch prediction, this is done in case a branch is taken to ensure the instruction that follows comes from the correct address. Branch prediction attempts to avoid this stall by predicting the direction that a branch will take when a branch instruction is encountered. If the prediction is correct no cycles are lost waiting for a new address to be resolved. It is only in the case of a misprediction that the pipeline must be flushed in order to take the correct branch.

Static Branching

Static branching is determined at compilation time and does not rely on past execution. These methods predict the same branch outcome throughout the duration of the runtime. *Never taken* will predict to never take the branch and will perform well on instruction sets where relatively few branches are taken. In contrast if there is a set of instructions where many branches are expected to be taken (e.g. a long loop) *always taken* may be a better choice as it always predicts to take the branch.

Dynamic Branching

Dynamic branching considers the outcomes of past branch instructions and as such is determined at run time. There are several different methods of dynamic branch prediction, the one chosen for this project is *two bit dynamic branch prediction*. One bit prediction will begin either with a branch or no branch prediction and each time there is a misprediciton it will change its prediction accordingly. These penalties can be too harsh and may not reflect well in one off branches within the instruction. Two bit prediction places a lesser penalty on a prediction and instead requires two mispredictions in a row to change the outcome of the predictor. This means that one off predictions are skipped and the longer sets of branch predictions are the focus of our change in predicted outcome. The Finite State Machine (FSM) used to facilitate the two bit branch prediction can be found in Figure 8.

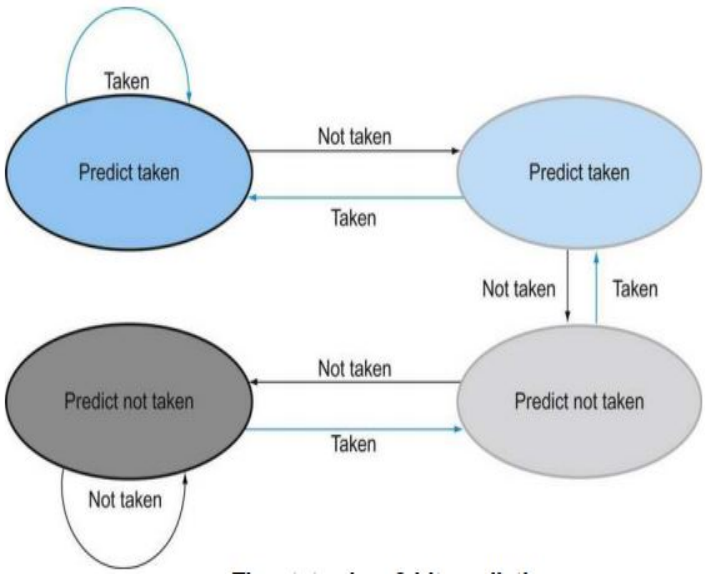

Fig. 8. Two Bit Dynamic Prediction FSM.

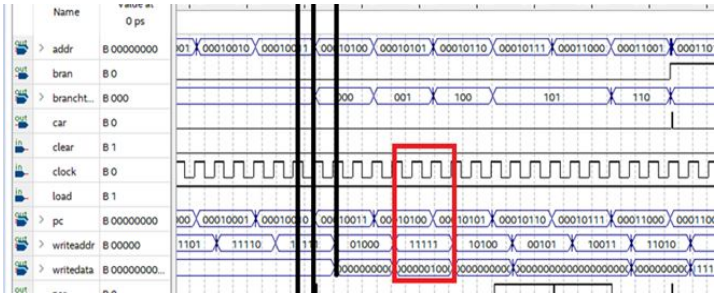

Fig. 9. Waveform for partially pipelined datapath. Red box shows 4 clock cycles required to process one instruction. Black lines show instruction moving through the datapath on each clock cycle (for a full picture see Appendix B, pg.10).

V. Results

Fully vs. Partially Pipelined

First the partially pipelined datapath from lab 5 was compared with the the fully pipelined datapath with data forwarding. The two were compared for the number of clock cycles necessary to complete one instruction and the throughput of the instructions of interest (to the end of the C code).

Speed

The partially pipelined datapath (Figure 9.) requires 4 clock cycles to complete an instruction. The fully pipelined datapath requires 6 clock cycles (Figure 10.) in the best case and 9 cycles (Figure 11.) in the worst case, i.e. during a stall, to complete an instruction.

Throughput

The partially pipelined datapath requires 176 clock cycles to complete the instruction set whereas the fully pipelined datapath requires 116. For waveforms see Appendix B, pg 10.

From these experiments we support the earlier claim that although a full pipeline will slow the speed of execution for an individual instruction, we will see increased throughput,

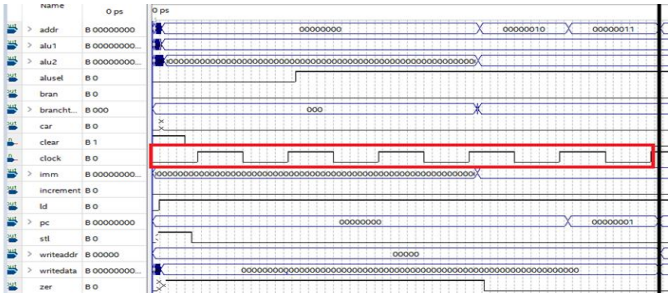

Fig. 10. Waveform for best case fully pipelined datapath. Red box shows 6 clock cycles required to process one instruction.

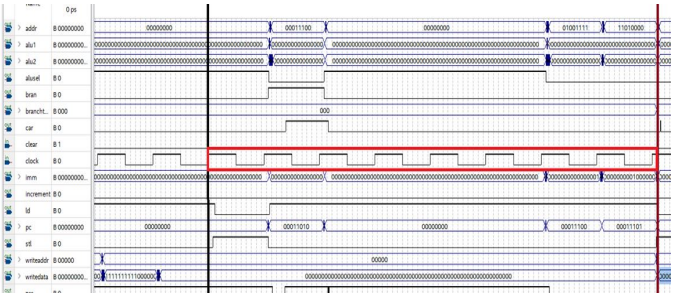

Fig. 11. Waveform for worst case fully pipelined datapath. Red box shows clock cycles required to process one instruction.

which only becomes more drastic with increased instruction counts.

Branch Prediction

The next experiments compare the three branch prediction methods (never taken, always taken, dynamic two bit). Each is compared for speed, throughput and prediction accuracy.

Speed

Each of these prediction methods are based off of the fully pipelined datapath with data forwarding, therefore, they share a speed with the previously assessed datapath. Six clock cycles in the best case, and 9 clock cycles in the worst case to complete an instruction.

Throughput

These subset of instructions used to calculate these datapaths ends as the C code starts. This requires 116 clock cycles for the partially pipelined, and 64 for the fully pipelined datapath.

Never taken requires 50 clock cycles to complete the instruction set for the instruction with moderate branching, 123 cycles for many branches and 57 for few branches.

Always taken requires 52 clock cycles to complete the instruction set for the instruction with moderate branching, 57 cycles for many branches and 126 for few branches.

Dynamic two bit requires 54 clock cycles to complete the instruction set for the instruction with moderate branching, 61 cycles for many branches and for 58 few branches.

Prediction Accuracy

Never taken has 40% prediction accuracy with moderate branching, 29% for many branches and 93% for few branches.

Always taken has 60% prediction accuracy with moderate branching, 71% for many branches and 7% for few branches.

TABLE IV
PREDICTION METHOD DATA

datapath	cylesmod	cylcesmany	cyclesfew	accuracymod	accuracymany	accuracyfew	complexity
never	50	123	57	40	29	93	complex
always	54	57	126	60	71	7	complex
dynamic	54	61	58	20	71	93	most complex

TABLE V
PERFORMANCE COMPARISON DATA

datapath	speed	throughput	complexity
partial	4.0	116	simple
fully	7.5	64	moderate
never	7.7	50	complex
always	7.7	54	complex
dynamic	7.7	54	most complex

Dynamic two bit has 20% prediction accuracy with moderate branching, 71% for many branches and 93% for few branches.

The results are listed in Table IV and table V.

Figure 14. displays the drastic increase in throughput for pipelined datapaths. Considering the scales in the plot one can see that the relative loss in speed is minuscule in comparison with the overall gains from pipelined execution.

Both the accuracy and throughput of the datapaths using branching prediction relies heavily on the type of instructions which it is fed. From the extreme tests (many and few branching instructions) we see that both static branching methods perform either very well or very poorly. These predictive measures are much too rigid as they make no change to their behaviour, even if they continue to mispredict for the entirety

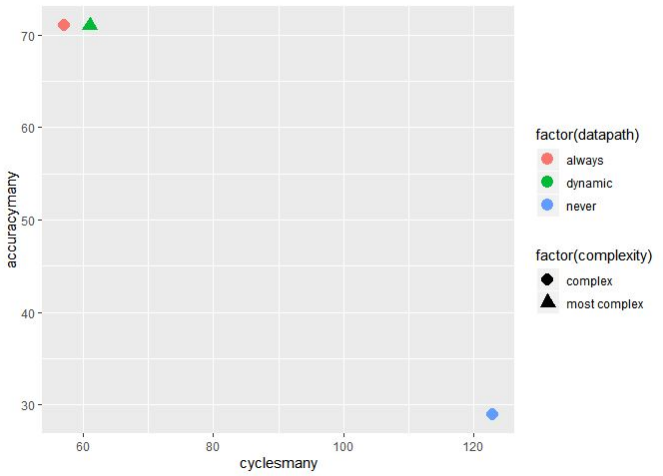

Fig. 12. Accuracy vs. Clock Cycles for many branching instructions.

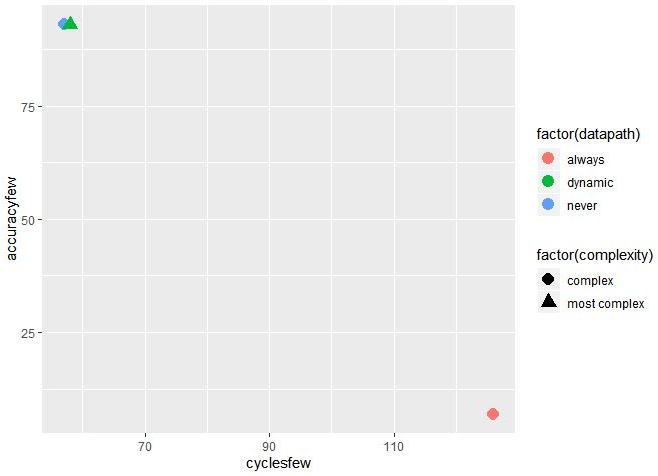

Fig. 13. Accuracy vs. Clock Cycles for few branching instructions.

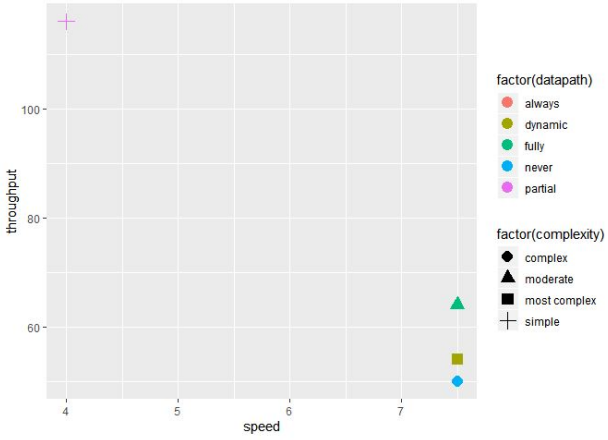

Fig. 14. Speed of instruction completion vs. Throughput.

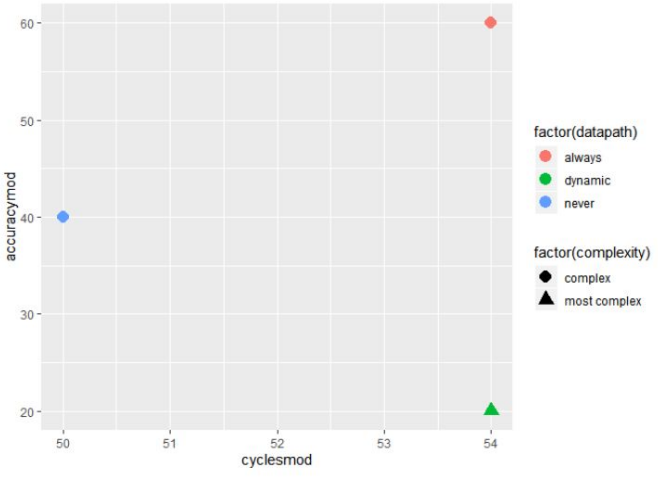

Fig. 15. Accuracy vs. Clock Cycles for moderate branching instructions.

Fig. 16. Accuracy vs. Clock Cycles for moderate branching instructions.

of an instruction set. These types of prediction methods may work well in cases of specific and known inputs. They are less complex than the dynamic prediction and therefore cause a lower overhead if being designed for use with a certain set of instruction in mind.

The dynamic predictor seems to function much better in regards to these extreme cases. It is flexible enough that it reaches a similar accuracy and low number of clock cycles to that of optimal the static prediction method for their respective instruction sets (Figure 12, 13). From Figure 15, however, we do see that dynamic branching has it's limitations. The moderate branching contained a relatively small amount of branches, however, in this case increased repetition of these specific branches would likely not improve the performance of the dynamic branching. Dynamic branching operates using a FSM (Figure 8.). In the top two states (blue) the component will predict to branch; in the bottom two (grey) it will predict to increment. It takes three mispredictions to move between the two left states, and two to change the prediction output if the system is currently in one of these states. It does, however, only take only one misprediction to move between the two right states. If you have a case of instructions that evenly swap, back and forth, between branch and don't branch this FSM will predict with 50% accuracy as it will never change state. There is, nonetheless, a worse case, which occurs in the moderate testing instructions. Fist there must be two mispredictions, this will cause the system to change prediction outputs. Next the system continues to flip flop between the states. Because we only made it to the unstable state on the right side of the table every mispredition will cause a change in the prediction outcome. On account of this the dynamic prediction unit will always predict one branch behind, and always cause a misprediction. We saw this at the end of our moderate instructions (Figure 16., blue circles show stalls due to mispredicitons following a branch in red and orange). In general processors are not flipping back and forth between branch and no branch for extended periods of time. More often we see groups of continuously incremented instructions, followed by a series of branches (loops). The two bit dynamic predictor is designed to perform well in these, more common situations. Keeping this in mind and considering the dynamic prediction unit's ability to adapt to both extreme instruction cases, this is the recommended prediction technique amongst the three tested in this experiment.

VI. Future Directions

Several complications have been resolved in the datapaths studied for this project. First pipelining was used to have multiple instructions executing at once in the datapath, speeding up throughput. Then data forwarding was introduced in order to avoid data hazards. Finally predictive branching was used as another technique to speed up our datapath by avoiding unnecessary stalls for branching instructions. There are, however, several more advanced techniques which will provide further performance enhancements for the datapath, three of which include superscalar processors, dynamic pipeline scheduling and Out-of-Order processors.

Superscalar Processors

Superscalar, refers to an advanced pipelining technique that allows the processor to execute multiple instructions during one clock cycle. During execution the processor will decide the number of instructions to execute, zero, one, or multiple. This means not only do we have an instruction being processed in each stage of the pipeline, as we achieved with simple pipelining, we can have multiple instructions being processed at any given stage in the pipeline.

Dynamic Pipeline Scheduling

Superscalar processors often include a secondary technique called dynamic pipeline scheduling. Dynamic pipeline scheduling attempts to avoid stalls by reordering instructions. This technique is quite complicated and requires the addition of reservation stations and a commit unit containing a reorder buffer. The reordering buffer functions similarly to the forwarding unit in the datapaths for this project. This unit and buffer control the release of operation results determining when it is safe to do so in order to avoid a stall. The reservation stations hold both the operand and operations.

Out-of-Order Processors

Dynamic pipeline scheduling is also employed within Out-of-Order processors. Out-of-Order processors allow the instructions to be executed in a different order than they were fetched. Any instruction that cannot be completed will not cause a stall in the following instructions. First instructions are copied into reservation stations, followed by any available operands from the register file or reorder buffer. If there are no available operands the functional unit that is to produce the operand is tracked and once available it is copied into the waiting reservation station. This bypasses the registers, again working in a similar matter to our forwarding unit (Figure 17.). Because operations are saved into the reservation stations the register copies are no longer needed and the pipeline does not need to stall to avoid overwriting these values. Using the buffers, reservation stations and commit unit of the dynamic pipeline schedule Out-of-Order processors track and analyze the the data flow structure of the program and increase decrease the amount of wasted clock cycles that would otherwise be required when waiting for resolution of operands.

VII. Conclusion

With this step by step, ground up construction of a RISC-V processor I was able to gain insight into the components, construction and design of a CPU. Each component

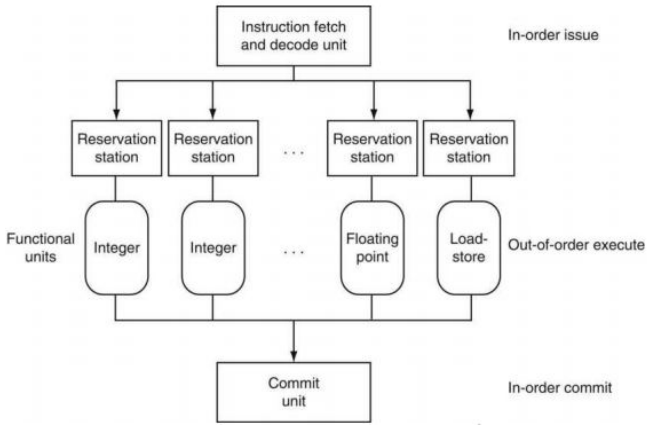

Fig. 17. Example of dynamic pipeline scheduling.

was constructed throughout several labs over the semester. This provided a strong foundation for the understanding the function of individual pieces of the CPU and how they are integrated to work together. The components were assembled into a datapath that underwent several upgrades in order to increase performance and avoid hazards. First pipelining was introduced to increase throughput by allowing instructions to be simultaneously executed in different stages of the pipeline. It was found that when compared to a partially pipelined datapath the fully pipelined datapath, although requiring on average 2 clock cycles more to process an individual instruction, completed the set of instructions in 35% fewer clock cycles. Next data forwarding was implemented to avoid data hazards that may occur during pipelining. These hazards occur when an output remains within the pipeline but is needed as an operand for a subsequent instruction. The forwarding unit accesses different stages of the datapath in order to extract the necessary data for a given operation. Finally, experiments were preformed to evaluate the accuracy and performance of various branch prediction techniques. Most instructions are read sequentially from the instruction memory, however, there are cases in which the processor may need to move forward or backward though the instructions (e.g. loops). In these cases a stall is applied to the datapath as the branching address is not resolved until the execution stage. In order to avoid these stalls three branch prediction techniques are applied: *never taken*, *always taken* and *dynamic two bit prediction*. These techniques attempt to predict whether or not a branch will be taken, and provide the associated address. When a correct prediction is made no stall is required and the pipeline is sped up, only in the case of a misprediction will a stall occur. Each technique was assessed with three benchmarks, one with moderate amounts of branch instructions resulting in a branch, one with many and finally one with very few branch instructions resulting in the branch being taken. After applying each of the techniques it was found that, as was expected, static techniques perform very well with their complementary instructions, i.e.*never taken* performs well with few branches (93% accuracy in 57 clock cycles), and *always taken* performs well with many branches(71% accuracy in 57 clock cycles). Our specific moderate example lead to relatively poor results for *dynamic two bit prediction*, however, it performed well and showed its adaptability when applied to both of the extreme cases, matching the clock cycles and accuracy of the respective optimal static branching technique. Further advancements can be made to the CPU, such as allowing multiple instruction to be implemented on the same clock cycle within a single stage (superscalars), or the ability to hold and swap the ordering of instructions to avoid unnecessary staling (out-of-order). Based on the results of the experiments, and only considering techniques applied in this course, it is recommended that the RISC-V CPU be designed using pipelining, data forwarding and dynamic branching prediction to achieve optimal performance results. It is also recommended that anyone interested in learning the inner workings of a CPU follow a similar ground up approach in order to gain a more fundamental understanding of of the way all the components function in unison.

VIII. Appendix A

A set of python scripts were written to created several different *.mif* files for use throughout the semester. These include:

- 1:1 Generator stores the data correlation to the address in which it is stored. This was used for testing in early labs and was used to create the data_mem for the datapaths.
- Instruction Generator used for most of the testing. This produces a set of instructions which test every possible instruction to the datapath and each type of branching with their respective outcomes. It also includes the C instructions described on pg....
- Many Branches Generator fills registers with values that are 75% equal and 25% unique values. Thirty random combinations of these registers are compared through Branch if Equal (BEQ) and most will branch.
- Few Branches Generator fills registers with values that are 25% equal and 75% unique values. Thirty random combinations of these registers are compared through Branch if Equal (BEQ) and most will not branch.

Source Code

IX. Appendix B

VHDL Source code, Waveforms, Word documents with expected instructions and Excel files with instructions. Located at the end of this file.

X. Appendix C

Data visualization for experiments.

XI. Acknowledgments

I would like to thank Dr. Anita Tino for all of her help and support over the semester. I have learned so much throughout this course and hope she is able to share it with more students in the future. I also want to thank the bunny Simon, cats Lily and Suzy, and dogs Lacie, Penny and Abbie and the

gecko Gex who graciously allowed me to use their names for signals thorough the semester and whose photos helped me get through some of the tougher parts of this course.

REFERENCES

[1] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design, 3rd ed.*. New York, N.Y.: The McGraw-Hill Companies, Inc., 2009.

[2] O. Mutlu. (2017). Design of Digital Circuits Lecture 17: Pipelining Issues [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik$_1$7/$lecture/onur - DigitalDesign - 2017 - lecture17 - pipelining - issues - afterlecture.pdf$

[3] O. Mutlu. (2017). Design of Digital Circuits Lecture 16: Dependence Handling [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk /inst-infsec/system-security-group-dam/education/Digitaltechnik$_1$7/$lecture/onur - DigitalDesign - 2017 - lecture16 - dependence - handling - afterlecture.pdf$

[4] D. Patterson and H. Hennessey, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Cambridge, M.A.: Elsevier Inc, 2018.