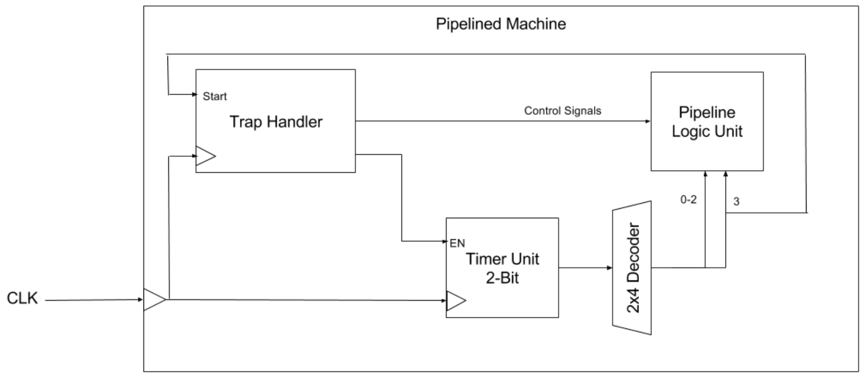
Project #4
Pipelined Machine

Mitch Belcher
Tyler Parcell

4/5/2016

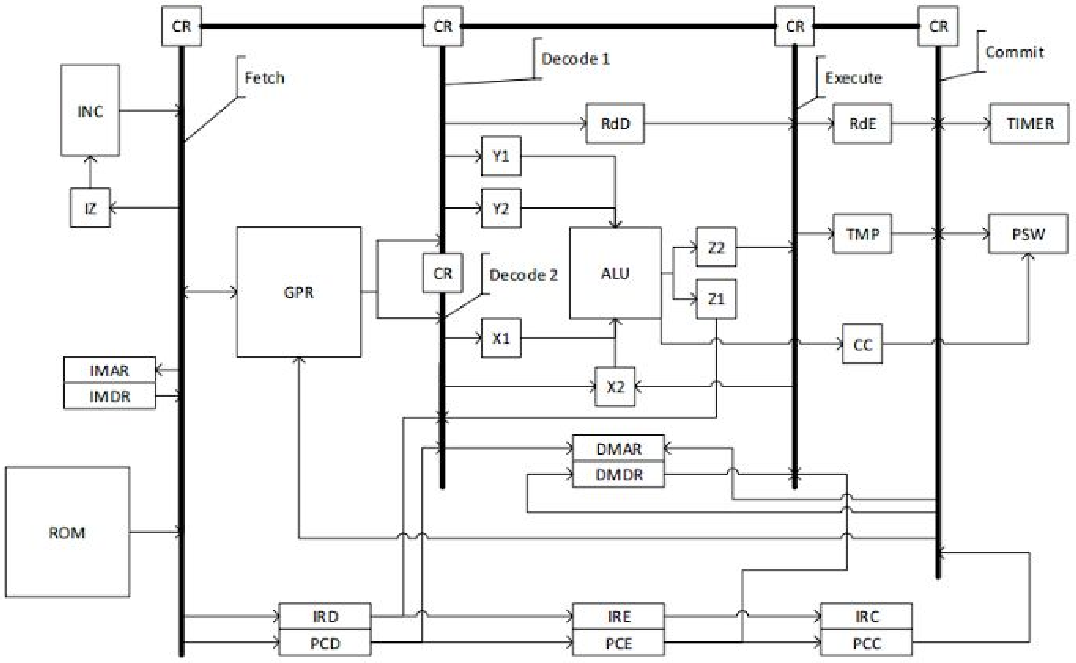
1. High-Level Design and Layout (Complete Machine)



Shown above in the figure is the extremely high-level diagram for the entire pipelined system. Essentially it consists of a basic state machine (shown as the combination of the “Timer Unit” and the “Decoder”), the pipelined data path itself (labelled as “Pipeline Logic Unit”), and the trap handling mechanism (labelled “Trap Handler”). When looking at the system from this high of a level, it is almost trivial how the system will work and move data around. The main things to note, however, is that the Timer Unit and the Trap Handler both control each other, since they are both state machines in themselves. This results in only one running at a time. For instance, upon system startup, the Trap Handler is always outputting a TRUE for the EN line of the Timer Unit. This means that every clock cycle that comes into the system is simply incrementing the Timer Unit. As the Timer Unit increments, the Decoder takes the numeric value of the Timer Unit and splits it into four separate control signals. The first three control signals (for Timer value 00 through 10) is used for the Pipeline Logic Unit’s instruction steps for each section (see section 2 for more details), while the fourth step (for Timer value 11) is used for register transfers within the Pipeline Logic Unit. Additionally, this fourth step also triggers the Trap Handler to start running, and causes it to output a FALSE for the EN on the Timer Unit. Essentially this means that the Timer Unit stops functioning on successive clock cycles, meaning the Pipeline Logic Unit also stops executing instructions; the machine is “frozen” in this state. Thus for successive clock cycles, the Trap Handler executes its steps, which involves checking for traps (see section 3 for more details). The Trap Handler can then issue direct control signals to the Pipeline Logic Unit to “fix” the system state in the event of a trap condition. Upon the completion of the trap handling, the Trap Handler will then output a TRUE for the EN of the Timer unit, causing BOTH state machines to reset to their zero-state on the next clock cycle, which then disables the Trap Handler, and the process continues.

2. Pipeline Unit Layout and Structure

2.1. Pipeline Sections



Shown in the image above is the overall pipeline structure of this machine. It consists of 5 BUSes, numerous registers, and a single ALU, as well as all other interfacing hardware, such as split instruction and data memories, GPR, ROM, PSW, and TIMER. A very conscious effort was put into reduction of hardware as an optimization. For instance, a basic pipeline design would employ input and output registers for each stage, then simply transfer data between the sages during successive clock cycles. Here, everything is simply stored in place, and registers can be accessed by all stages that need them. Additionally, it should be noted that we only used a single ALU. The ramifications to this will be discussed in a latter section of this report, and the control signals that had to go into making this possible, but overall it was with the mindset of again hardware reduction optimizations. In short, data in this system flows from left to right. The instruction is read and the system state updated (program counter) along the Fetch BUS, while all operands of other related addresses are calculated/gathered along the Decode BUSes. Meanwhile the Execute BUS then performs all calculations (ALU operations basically) and the Commi BUS is where all changes reach the system permanently. This independent commit stage was done to help with hazard considerations, particularly surrounding trap conditions, and is highlighted at length in section 3 of this report.

2.2. BUS Logistics and Hardware Access

Since the system's pipeline structure is very "smashed together" if you will (high cohesion), the BUS organization and access to hardware is very important. To achieve such high cohesion, particularly when sharing the data memory access and the single ALU, rules were implemented in the consideration of our control signals to ensure that no control hazards happened. These rules are as follows:

- Fetch
  - Step 1: Setup system to change state
  - Step 2: Read the Instruction Main Memory
  - Step 3: Grab data from Instruction Main Memory
- Decode
  - Step 1: Perform all ALU calculations for addresses
  - Step 2: NULL - No Operation
  - Step 3: Perform memory operations on the Data Main Memory
- Execute
  - Step 1: Grab any data from the Data Main Memory
  - Step 2: Perform all ALU operations
  - Step 3: NULL - No Operation
- Commit
  - Step 1: Write data and set condition codes.
  - Step 2: NULL or perform a Data Main Memory Write
  - Step 3: NULL - No Operation

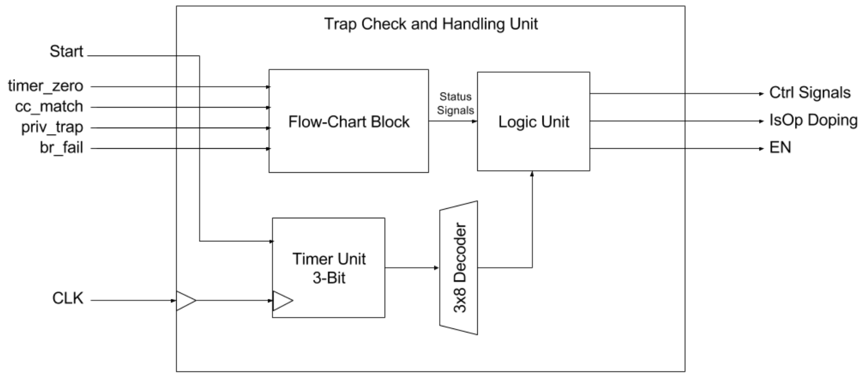
2.3. Control Signals

The control signals for this system varied on an instruction by instruction basis, though all similarities were grouped together to adhere to the generalized steps as mentioned in the previous sub-section of this report. By adhering to these generalized steps, the system's resources (hardware) will not undergo any structural hazards. In essence the establishing of these generalized steps prevented/solved these hazards. Specific control signals, however, are highlighted in the following table (next page) and is colored by logically by groups of similar functionality (unfortunately it is rotated to fit on a single page).

Instruction	Fetch			Decode			Execute			Commit		
	F1	F2	F3	D1	D2	D3	E1	E2	E3	C1	C2	C3
ADD	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	NULL	NULL	GPR[IRD.RS1]-D1out, Y2in, GPR[IRD.RS2]-D2out, X2-Din	NULL	ALU|ADD, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
ADDM	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, GPR[IRD.SO]-D2out, X1in, X1out, Y1out, ALU|ADD, Z1in	NULL	GPR[IRD.Rd]-D1out, Y2in, Z1-D2out, DMAR-D2in, Read_DMM	DMDRout, X2-Ein	ALU|ADD, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
SUB	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	NULL	NULL	GPR[IRD.RS1]-D1out, Y2in, GPR[IRD.RS2]-D2out, X2-Din	NULL	ALU|SUB, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
SUBM	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, GPR[IRD.SO]-D2out, X1in, X1out, Y1out, ALU|ADD, Z1in	NULL	GPR[IRD.Rd]-D1out, Y2in, Z1-D2out, DMAR-D2in, Read_DMM	DMDRout, X2-Ein	ALU|SUB, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
AND	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	NULL	NULL	GPR[IRD.RS1]-D1out, Y2in, GPR[IRD.RS2]-D2out, X2-Din	NULL	ALU|AND, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
SHL	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	NULL	NULL	GPR[IRD.RS1]-D1out, Y2in, GPR[IRD.RS2]-D2out, X2-Din	NULL	ALU|SHL, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
SHRA	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	NULL	NULL	GPR[IRD.RS1]-D1out, Y2in, GPR[IRD.RS2]-D2out, X2-Din	NULL	ALU|SHR, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
OR	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	NULL	NULL	GPR[IRD.RS1]-D1out, Y2in, GPR[IRD.RS2]-D2out, X2-Din	NULL	ALU|OR, Z2in, X2out, Y2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
NOT	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, GPR[IRD.SO]-D2out, X1in, X1out, Y1out, ALU|ADD, Z1in	NULL	Z1-D2out, DMAR-D2in, Read_DMM	DMDRout, X2-Ein	ALU|NOTX2, Z2in, X2out, CCin, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
LD	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, GPR[IRD.SO]-D2out, X1in, X1out, Y1out, ALU|ADD, Z1in	NULL	Z1-D2out, DMAR-D2in, Read_DMM	DMDRout, X2-Ein	ALU|PASSX2, X2out, Z2in, Z2out, TMPin	NULL	TMPout, GPR[IRC.Rd]-Cin, CCset	NULL	NULL
ST	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, GPR[IRD.SO]-D2out, X1in, X1out, Y1out, ALU|ADD, Z1in	NULL	Z1-D2out, X2-Din, GPR[IRD.Rd]-D1out, RdEin	NULL	ALU|PASSX2, X2out, Z2in	Z2out, TMPin	TMPout, DMAR-Cin, CCset	RdEout, DMDRin, Write_DMM	NULL
BR	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, IRD.LOout, X1in, ALU|ADD, Z1in	NULL	Z1-D2out, X2-Din	NULL	ALU|PASSX2, X2out, Z2in, Z2out, TMPin	NULL	NULL	NULL	NULL
JSR	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, IRD.SOout, X1in, ALU|ADD, Z1in	NULL	Z1-D2out, X2-Din	NULL	ALU|PASSX2, X2out, Z2in, Z2out, TMPin	NULL	GPR[IRC.Rd]-Cin, TMPout	NULL	NULL
RTS	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	GPR[IRD.RD]-D1out, Y1in, IRD.SOout, X1in, ALU|ADD, Z1in	NULL	Z1-D2out, X2-Din	NULL	ALU|PASSX2, X2out, Z2in, Z2out, TMPin	NULL	NULL	NULL	NULL
CLK	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, IRD.LOout, X1in, ALU|ADD, Z1in	NULL	Z1-D2out, DMAR-D2in, Read_DMM	DMDRout, X2-Ein	ALU|PASSX2, X2out, Z2in, Z2out, TMPin	NULL	TMPout, TIMERin, CCset	NULL	NULL
LPSW	GPR[7]-Fout, IMARin, IZin, PCDin	GPR[7]-Fin, INCout, Read_IMM	IMDRout, IRDin	PCD-D1out, Y1in, GPR[IRD.SO]-D2out, X1in, X1out, Y1out, ALU|ADD, Z1in	NULL	Z1-D2out, DMAR-D2in, Read_DMM	DMDRout, X2-Ein	ALU|PASSX2, X2out, Z2in, Z2out, TMPin	NULL	TMPout, PSWin, CCset	NULL	NULL

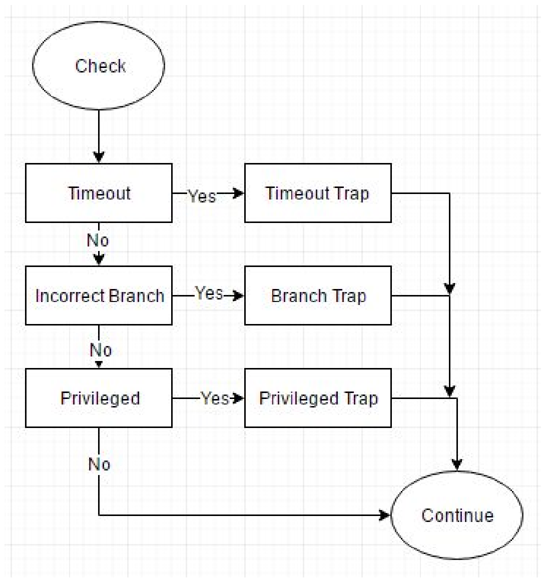
3. Trap Handling

3.1. Trap Checking System



We designed a small section of circuitry at a high level, that would be used to handle any traps that occur and check for them in the first place. This can be seen in the above image. To start with, the whole thing is locked into the cycle that the 3-bit timer is on, and we then take in four inputs to the flowchart (shown in section 3.2 below), if the timer has reached zero, if the condition codes match, if we are executing a privileged instruction in user mode, and if we failed a branch prediction. These inputs are going to determine the progression made through the flowchart. Based on them, we will most certainly be able to tell what kind of trap, if any, we will be dealing with. We then will handle each trap individually, for modularity, as well as reducing circuit complexity coupled with the desire to design this pipelined system for the common-case (traps are not common occurrences for the most part); the way these are handled individually is laid out in the control signals shown in sections 3.3, 3.4 and 3.5 (these are timeout traps, privileged mode traps and incorrect branch prediction traps). Each of these unique sections designed to handle a particular trap process, will have control signal outputs that will be sent to the pipelined logic unit and used to appropriately resolve that trap issue, an enable line for the timer/clock system that controls the rest of the machine (so that we are halted in our cyclic process until the trap has been dealt with), as well as whatever "doping" is necessary for the shift register that handles the running list of op/no op bits (in handling a trap, it may be necessary to send no ops through in certain situations, like flushing a bus line and it's registers, therefore we need to make sure those are recorded into their designated register).

3.2. Trap Priority Logistics



There is actually a bit of priority on the different possible traps of the system, and we outlined that in the above figure. Due to the fact that a timeout trap at the commit stage means that the instruction executed just before, should never have been executed anyway, means it takes priority over the other two trap types. The other two trap types are basically at the same priority level, given that they can never both happen at the same time, but are both less important than the timeout trap.

3.3. Branch Prediction Failure Trap

3.3.1. Didn't Branch, and Should Have

1. TMPout, GPR[7]-Cin
2. NULL
3. FLUSH

3.3.2. Branched, and Shouldn't Have

1. PCCout, CROSS, IZin
2. IZout, GPR[7]-Fin
3. FLUSH

3.4. Timeout Trap

1. FLUSH F bus, FLUSH D bus, FLUSH E bus
2. CROSS, ROM[4]out, DMAR-D2in
3. CROSS, PSWout, DMDRin, Write_DMM

4. CROSS, ROM[5]out, DMAR-D2in
5. CROSS, PCCout, DMDRin, Write_DMM
6. CROSS, ROM[6]out, DMAR-D2in, Read_DMM
7. CROSS, DMDRout, PSWin
8. CROSS, ROM[7]out, DMAR-D2in, Read_MM
9. CROSS, DMDRout, GPR[7]-Fin

3.5. Privileged Trap

1. FLUSH F bus, FLUSH D bus
2. CROSS, ROM[0]out, DMAR-D2in
3. CROSS, PSWout, DMDRin, Write_DMM
4. CROSS, ROM[1]out, DMAR-D2in
5. CROSS, PCEout, DMDRin, Write_DMM
6. CROSS, ROM[2]out, DMAR-D2in, Read_DMM
7. CROSS, DMDRout, PSWin
8. CROSS, ROM[3]out, DMAR-D2in, Read_MM
9. CROSS, DMDRout, GPR[7]-Fin

4. Branch Prediction

4.1. Overview of Optimization

The main purpose of the branch predictor is to improve the performance of the Pipelined machine by making an educated guess at the branching behaviors of the system. Since the machine is pipelined, the instructions are executed in several stages (in this casee four separate stages). That means that when an instruction completes, three other instructions are partially completed already (the next three). If this leading instruction happens to be a branch instruction, then it cannot possibly be known whether the system was supposed to take the branch or not until later in the pipeline. As a result, the system can send "NoOps" (Null Operations) through the pipeline to wait and see where to go for the next instruction. Since, however, branching is very common (occurring about 25% of the time), there would be a huge loss in the performance gain from pipelining to begin with. As a result, we really don't want to have to send NoOps through the pipeline. To remedy this, the system employs a Branch Predictor, which predicts whether the system is likely going to branch or not, then continues executing instructions based on this prediction. If the prediction was correct, then the system continues execution. If the prediction was incorrect, then the system flushes the work it has done and goes to the correct instruction to begin execution again. Even if this predictor was completely static (resulting in correct guesses only 50% of the time), there would still be a dramatic performance boost, since it would only be flushing half the time. Of course, the higher accuracy in predictions the better. As a result, we implemented our own multi-method branch predictor (one was pattern based and one is static) for handling branch prediction. These two methods are highlighted in the next two subsections).

4.2. Simple Static Predictor

The simple branch predictor simply consists of statically predicting a FALSE, or branch not taken every time it is polled. Obviously this method of prediction is very simplistic, and doesn't get much optimization to it. This however, is necessary for when the latter branch predictor isn't sure what to predict. Technically yes, the pattern-based branch predictor could still be used, and it wouldn't be much different in terms of optimization, but the way it works makes this a difficult implementation, since the prediction is pulled from an unpopulated register, if it isn't "ready" to predict.

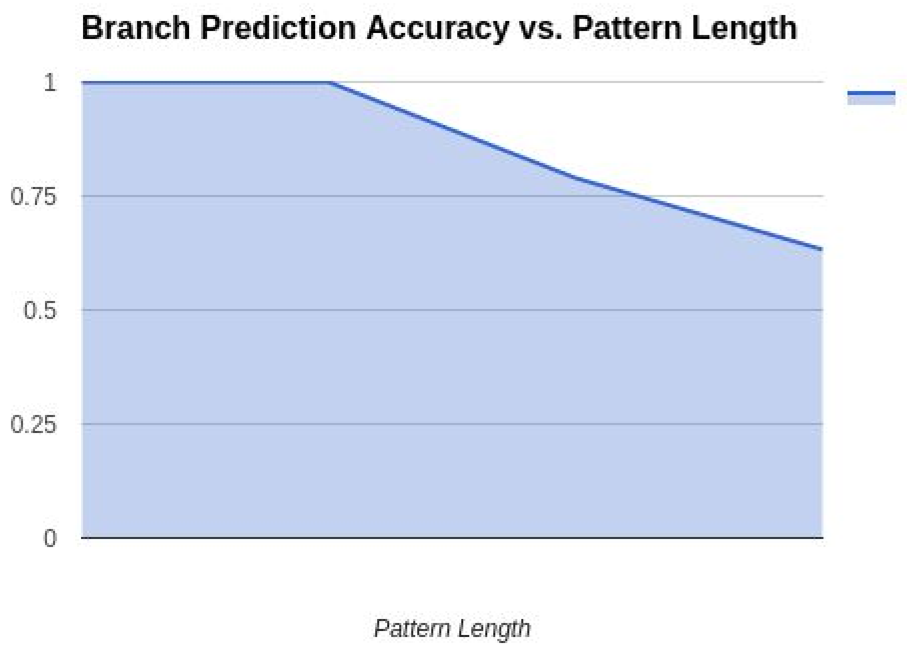
4.3. Pattern-Predictor Overview

This main goal of this branch predictor is to analyze the system's branching habits/behavior for patterns. This is very likely to occur because of the organization of computer programs being so focused on looping around data with internal conditionals. As a result, the branch predictor that we came up with and implemented in this design takes in data into a shift register to keep track of the last several branches the system has made. The moment it detects a pattern, it will attempt to make branch predictions based on this pattern and evaluate whether or not it is predicting correctly. Obviously this is a very high-level description of how it works, but if you take a look at the attached Python code (in the appendix) the methodologies behind its operation become much more clear.

Essentially, the predictor consists of two registers, the "branch history" and the "branch guess" register. Initially, the branch history register and the guess register are both empty - full of zeros. When a 1 gets clocked into the branch history (indicating that a branch was taken), then the system starts to work to find a pattern. To find a pattern, the system always looks for another 1 to be clocked into the branch history register. When this occurs, it will assume that a pattern has been found, and that this 1 represents the start of the pattern again. It is at this instant that it will copy this register into the guess register. Upon populating the guess register, the guess register will start issuing predictions, instead of the static 0 guess mentioned before. This will continue, and the history register wont accept any more history until the guess register predicts incorrectly, in which case the guess register is cleared, and more history begins being taken, looking for yet another 1 to be fed into the system for the process to repeat. It should be noted that repeated failures will result in longer patterns being detected. Once the history register becomes saturated though (full), the system will "give up" on looking for longer patterns, and clear both registers on the next failure and begin the process all over again. It turns out that this method (when tested with Python code) is extremely effective at detecting short patterns and is also quick to lock onto patterns when they switch, which is exactly what was desired out of the system. Specific performance metrics are discussed later in the next sub-section of this report.

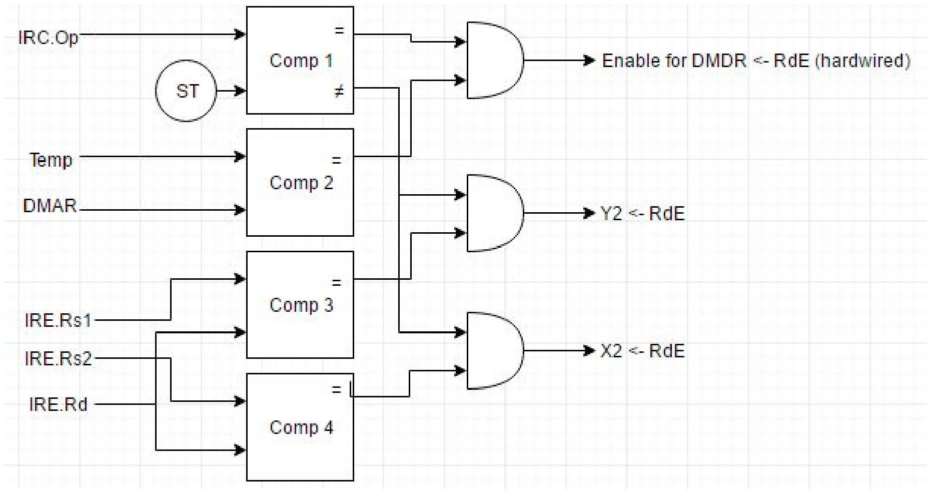
4.4. Performance Data

Shown below in the figure is the results of putting the branch predictor into a situation where it had to detect several patterns. Specifically, the tests that were conducted involved generating 100 random 2-bit, 3-bit, 4-bit, and 5-bit patterns, then running each for anywhere between 100 and 10,000 times through the branch predictor, then tracking how often it would guess correctly. As you can see, the branch predictor is phenomenal at predicting 2 and 3 bit patterns, yielding an almost perfect prediction rate (the data showed about 99.96% accuracy for both). WHen looking at the 4-bit pattern, the average for the tests was more like 78.94% accuracy and the 5-bit pattern was around 63.26% accuray. When the test was then conducted using equal probabilities of each showing up for all 100 generated patterns, the average ended up being 80.91% accuracy. So overall, he branch predictor seems to have an 80% accuracy in predicting the branches of the system, which is quite good overall. How does this affect the system? Since branches occur about 25% of the time in the system's execution of a program, and the predictor only guesses incorrectly about only 20% of the time, whenever a given instruction is executed, it will only result in a flushing of previous steps (due to the incorrect branch prediction) 5% of the time (20%*25%); the other 95% of the time, the system's pipeline is unaffected and operating smoothly.



5. Hazard Reduction

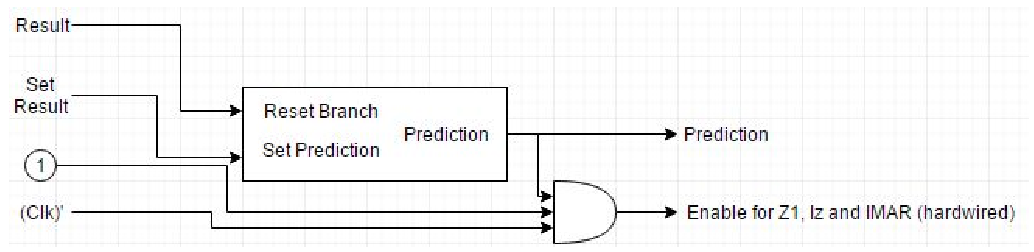
5.1. Data Forwarding (Execute and Decode, Commit and Execute)



Data Forwarding, in general, is a common solution to a data hazard (an issue where you are trying to use data in, say, an execute step relies on a piece of information that must be decoded in the decode state). It works by "forwarding" the operands to the step that needs them. Specifically for our case, we could have ran into an issue where the execute stage is using data from the main memory or registers that are going to be overwritten by the commit stage. To solve this problem, we send that destination information from commit back to execute to ensure it is using the correct values. We decided that this would most efficiently be executed on the falling edge of the clock signal, because we originally weren't taking advantage of that edge.

From the image above, you can see an abstraction of this implementation. We have three possible control signals to send to forward the data back to a previous stage in question. Firstly, we compare the value of the current opcode to the opcode corresponding to the "store" instruction. If we are indeed executing a store instruction, then we send a 1 from the top output. Similarly, in the event that the Data MAR address and the address in the temp register are the same, we send another one to the same AND gate. This will initiate a tri-state buffer that will now allow information to be passed through hardwired connections to send the current value of the RdE register into the Data MDR. This is because in the event we are executing a store instruction, we will potentially have an issue trying to read and write from the Data section of the main memory (this is the only instruction with this hazard), and therefore we need to alleviate the issue by sending the correct data back. Likewise, the other two outputs for hardwired control signals, are only ever active under certain circumstances where two stages are attempting to read and modify data at the same time, in which case, we backtrack the values that the previous stage needs so that it’s cycling correctly.

5.2. Branch Predictor Address Forwarding (Decode and Fetch)



Similarly to the previous section (5.1) regarding data forwarding, we also encounter a hazard in which occurs when we predict that we need to branch, but we have already started passing through the addresses and data from the next instruction in the list. To combat this, we designed a data forwarding system, that like the previous section works on the falling edge of the clock signal, that will pass the proper branched address back into the system to read. This works by using the Branch Predictor, already outlined in section 4, along with the inverted clock signal and the timer/state machine signal for the first step. What will happen in this scenario is, only when we are going to predict a branch, and we are in the first step of the cycle and the clock signal is low, we are going to enable hardwired connections on tri-state buffers to pass this new branched address, into the Z1 register so that we can send it to the other two registers: the Instruction MAR (to read the right address, before it was set to the next incremented instruction address, now it’s a branched address), as well as sending it into the Iz register on the Fetch bus to start our new incrementing of the instructions from our branched address.

5.3. Commit Stage Purpose (Trap Handling)

We decided to add a “commit” stage to our state machine operation and design for a few reasons. The main one being that, including a stage such as this allows the possible issue of incorrectly predicting a branch to be resolved much simpler. In a normal case without a stage that operates like this, we could have something like the following take place: We predict a branch path, and proceed to execute that new (branched) instruction. It’s possible that our branch prediction circuitry incorrectly predicted the branch (we branched, but we weren’t supposed to, or we didn’t branch and were supposed to). In this event, normally, without the commit stage, we would then need to go back to all of the register entries and main memory addresses and return them to their appropriate values for what was supposed to happen. This would require saving the information along the way, so that we could overwrite the information if something like this happened.

However, if you have a commit stage, this issue is resolved much simpler. A commit stage allows us to write values at the last possible moment, giving us time to determine if the branch was correctly predicted or not. In the event that it wasn't, this stage allows us to simply flush all of the buses, and reset our program counter to where it should be. We won't have to save any of the information along the way, because we intentionally don't overwrite anything until we're sure that we were correct to begin with.

To activate the commit stage tri-state buffers, we require multiple inputs to determine whether or not this information should be written. We use a small four-bit shift-register to contain the last four opcode states (1 for an opcode, 0 for no opcode). If the last of these signals in the register at the end of the shifting is a 1, we send a 1 to an AND gate, and if it was a 0, we do nothing. This signifies whether or not we want to commit the data and whether or not the current values sitting throughout the control unit/data path are garbage values or not. Now the other input to that AND gate is the output of some simple circuitry that checks the two condition code bits for the current instruction from the ALU against the current PSW bits to determine if that instruction was supposed to be executed or not. If this is a 1, then we are good to send the all clear to the commit stage tri-state buffers and proceed in writing the data to their locations. If not, then we shouldn't have executed that data, and therefore, we won't actually commit any of the data calculations/changes we just came up with.

This is a big benefit to our design because it means we are always assuming the best case and the most common case: that we are doing exactly what we are supposed to do. Instead of changing everything in the event we get something wrong or adding a massive amount of circuitry to withhold values as we go in case we overwrite things that shouldn't have been overwritten, we simply use some simple logic to determine whether or not we should let the writing of the data happen or not.

6. Appendix

6.1. Branch Predictor Raw Data

	2-Pattern	3-Pattern	4-Pattern	5-Pattern	Mixed
Test 1	0.9976588465	0.9997394626	0.8406845004	0.6368842897	0.8375348977
Test 2	0.9996657406	0.9997510591	0.8158370703	0.64761996	0.7729328173
Test 3	0.9997971497	0.9995822585	0.8081267761	0.6558355847	0.8526096767
Test 4	0.9997885098	0.9997184711	0.6995061189	0.6358231741	0.8270728274
Test 5	0.9991129032	0.9994990044	0.7665358268	0.572564131	0.8099138641
Test 6	0.9997616184	0.9995782137	0.7997979179	0.5939628441	0.8221717169
Test 7	0.9994366825	0.9992459158	0.7497900763	0.6070293454	0.8028114865
Test 8	0.995037594	0.9988871224	0.7578393598	0.6475792079	0.8213603675
Test 9	0.9995311628	0.9996883742	0.7913035775	0.6432703372	0.7967724544
Test 10	0.9997276853	0.999196804	0.8331138096	0.6419244392	0.7700135545
Test Averages	0.9996106126	0.9995933268	0.7893566891	0.6325625818	0.8091123011

6.2. Python Code

The following pages are the python code for the branch predictor, added in addition to the document to preserve the syntax highlighting.