Rojan Karn ECSE 4780: Advanced Computer Architecture

Project 3: Reorder Buffer

This report will cover my implementation of the 2-wide superscalar out-of-order execution using a reorder buffer. This project is based on the previous project which used a Register Alias Table to implement Tomasulo's algorithm. However, unlike the previous project, this reorder buffer implementation aims to implement the "in-order commit" stage of the process which helps in the case of exception handling, such as branch mispredictions.

The architecture of this project is shown in the diagram below, and contains a lot of the same blocks that were used in Project 2.

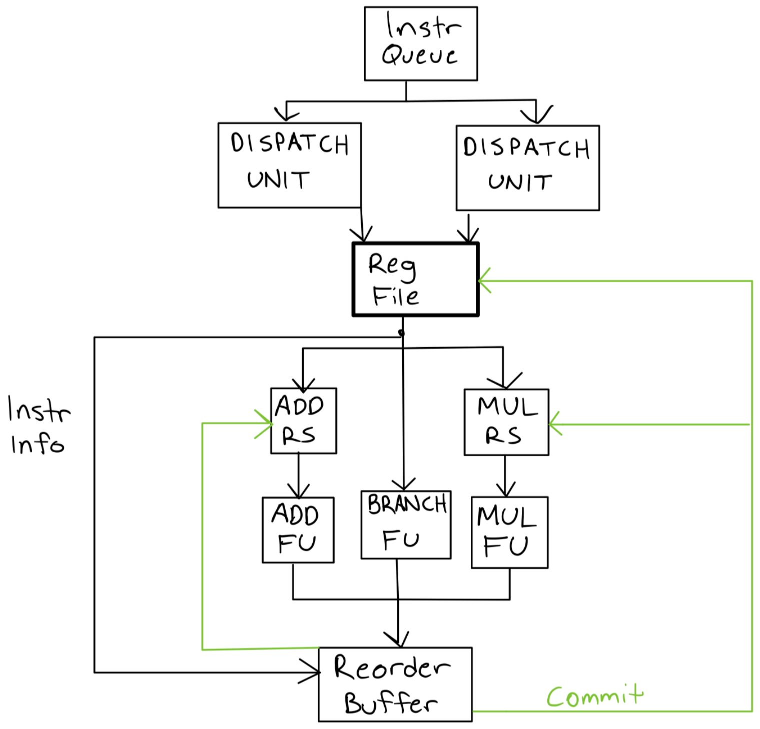


Figure 1: Block-level diagram with direction of signals between blocks

*Note: The "Reg File" block encapsulates the Architectural Register File and the mapping table

Overview of Blocks

Many of these block descriptions are the same as they were described in the Project 2 report. If notable changes were made to the module, they will be adequately described in the block's subsection.

Instruction Queue

The instruction queue is simply implemented as a synchronous FIFO with two output ports, one going to dispatch unit 1 and the other going to dispatch unit 2. The FIFO is able to hold 16 RISC V instructions which are each 32 bits wide. On the read_enable signal coming from the dispatch units, the FIFO sends the first two instructions in the queue to the dispatch units. For I/O, the instruction queue takes inputs from the top-level/testbench modules to load instructions into the FIFO. The outputs of the instruction queue are the two instructions that are sent to the DUs.

Dispatch Unit

The inputs of the dispatch unit are the instruction from the instruction queue, available spots list from both reservation stations, next available spot in the reorder buffer, and instruction information from the other dispatch unit. The outputs are the read_enable signal to the instruction queue, instruction information to the other dispatch unit, and the renaming information and instruction information to the register file module.

The dispatch unit extracts the rd, rs1, rs2, and instruction type (add, multiply, and "branch if equal") from the 32 bit instruction. The dispatch unit also has support for NOP instructions, which simply just stall the dispatch of the instruction. It uses the instruction type and instruction type coming from the other dispatch unit to check the availability of the corresponding reservation stations AND the availability in the reorder buffer. If there is space available, both of the instructions' information is sent to the register file to update it with the renaming of the destination register. However, if there is not enough space available in the reservation stations to dispatch both instructions at the same time, both dispatch units stall until there is enough space.

A notable distinction that is made between Project 2 and Project 3 is how the register renaming is done. In Project 2, destination registers were renamed based on the entry of the reservation station that they would take up. However, in this project, registers are instead renamed based on the entry of the reorder buffer that they will take up. The reorder buffer has a constant "next_available" signal to both of the dispatch units which lets the DU know what to rename their destination registers to. The "renamed_tag" is set to the next_available ROB tag (or next_available + 1 if it is the low priority DU) and is sent to the register file where the register translation is stored in the mapping table.

Register File

This register file (RF) module is mostly the same as the "Register Alias Table" from Project 2 with a couple of changes. The full functionality of this module is described in this section. The register file takes inputs from both DU1 and DU2, and outputs signals to both of the reservation stations and the branch functional unit (as there is no need for a branch reservation station - more on the branch FU in a later subsection). Also, the RF listens on the common data bus for updates from the reorder buffer to tag/value assignments in both the register file and mapping table. For each DU, the RF takes in the instruction information and the "renamed_tag". To each reservation station, the RAT has ports for two instructions to be outputted to each reservation station in the case that both instructions are of the same type.

The RF has 8 entries, each of which contain the register ID (R1, R2, …, R7) and the corresponding value. The mapping table also has 8 entries, each of which contain the register ID, the tag (which corresponds to the ROB entry that the register is mapped to), and the valid bit. The valid bit is high when the register-to-tag translation is active, and low when the register does not need to be translated.

When both dispatch units send the "ready" signal, the RF updates the destination register entry in the mapping table by setting the valid bit to 1 for each instruction and setting the new tag (which is the index of the ROB that this instruction will be stored in). Then, the outputs are set accordingly based on the type of both instructions, as they need to be sent to the correct reservation stations. The RF also checks whether an instruction is a branch instruction, and if so, it sends the tag and source register data values to the branch functional unit.

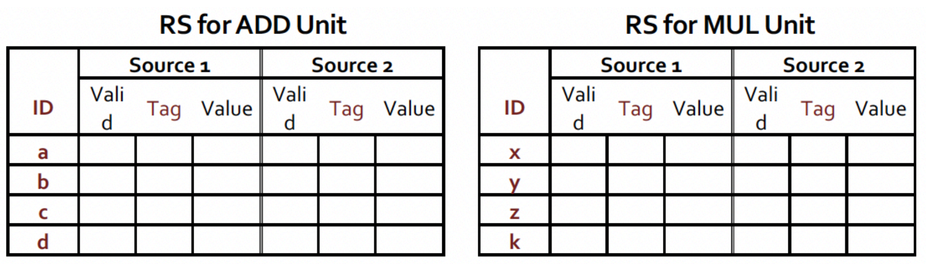
When sending instructions to the reservation stations/branch functional unit, the RF also sends instruction information to the reorder buffer. It sends the destination register for both instructions simultaneously and also sends the two sources and their valid bits for each of those two instructions. When sending the source registers, it checks whether the translation for those registers is valid first. If the translation is valid, it sends the ROB tag as the source register, but if the translation is not valid, it sends the register itself (as this register has not been renamed).

Also, when the bus indicates an update with the "ROB_bus_trigger" signal, the RF iterates through its table to check for if an entry is invalid AND its tag matches the broadcasted tag. If this condition is met, the value is updated to be the broadcasted value, the translation is invalidated (because data is present now), and the tag is nullified.

Reservation Stations

Both the adder and multiplier reservation stations (RS) are implemented in essentially the same way. The inputs to the RS are the instruction information to the one OR two instructions incoming from the RF. The Register File sends a signal that indicates whether the RS should load one or load two instructions, and based on that, the RS reads the input ports accordingly. The outputs go to the corresponding functional unit (adder or multiplier) and contain the two operands as well as the tag of the destination register. As mentioned, the RSs also output a list to the dispatch units of the available spots that it has. The reservation stations also listen on the ROB bus for potential updates to entries that it is holding.

Both reservation stations are implemented exactly as shown in the lecture slides:



RS for ADD Unit

ID	Source 1			Source 2		
	Valid	Tag	Value	Valid	Tag	Value
a						
b						
c						
d						

RS for MUL Unit

ID	Source 1			Source 2		
	Valid	Tag	Value	Valid	Tag	Value
x						
y						
z						
k						

Figure 2: Reservation Station structure

To assign the list of available spots that the RS outputs to the dispatch units, it checks whether both of the entries' sources are both invalid and have a null tag. This indicates that the entry is available to take a new instruction in.

Based on the "load_one" or "load_two" signals, the RS reads the input ports and fills in its entries accordingly. For the execution logic, on every positive edge of the clock, if the functional unit is ready for a new instruction, the RS iterates through its entries until it finds one where both sources are valid. Then, it assigns the outputs to the FU for it to execute.

The RS also listens for the "ROB_bus_trigger" signal and looks for an invalid entry source whose tag matches the broadcasted tag, and updates the value to the broadcasted value if the tags do match.

Functional Units (ADD and MULTIPLY)
The functional units (FU) for both addition and multiplication are implemented in essentially the same way as well. The adder FU has a delay of 4 clock cycles and the multiplier FU has a delay of 6 clock cycles. In this implementation, the delay is modeled using a mod-4 and mod-6 counter, respectively.

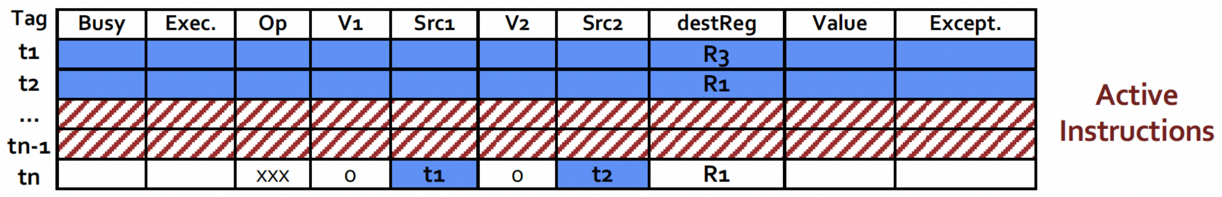
The I/O is simple, each functional unit takes in the two operands from its reservation station and the tag. It outputs a "ready" signal to the reservation station when it is available to take a new instruction. The FU result value and ROB tag are outputted to the reorder buffer where the corresponding tag is updated with the resulting value of the operation.

Branch Functional Unit
This module is new for Project 3 as the system now supports "branch if equal" (beq) instructions. This module is very simple, it just takes in the ROB tag of the instruction and the two operands to be compared. Its outputs are a trigger to the ROB to notify it that an update is necessary and also sends the tag of the entry where the update is needed.

In this implementation, we are always predicting "not taken" for beq instructions. This functional unit simply checks whether the two operands are equal, and if they are, then sends the "update" signal to the ROB. The update that is being referred to here is simply to let the ROB know that there is a branch misprediction in the system, so set the "exception" bit high on the corresponding instruction.

Reorder Buffer (ROB)

The reorder buffer is the main new module that was added to this project which implements the "in-order commit" stage of the process. The structure of the ROB is the same as is shown in the lecture slides:



Tag	Busy	Exec.	Op	V1	Src1	V2	Src2	destReg	Value	Except.
t1								R3		
t2								R1		
...	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
tn-1	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
tn			xxx	0	t1	0	t2	R1		

Active Instructions

Figure 3: Reorder Buffer structure

In the lecture slides, the tail pointer is the "next to commit" and the head pointer is the "next available". However, in this implementation, the head and tail pointers have the opposite functionality: head pointer is "next to commit" and tail pointer is "next available entry". As mentioned before, the tail pointer is made available to the dispatch units for ease of register renaming.

The inputs to the ROB include new instruction information from the register file as well as result information coming from the ADD, MUL, and BRANCH functional units. The outputs of the ROB are placed on the common data bus that goes to the reservation stations and the register file which broadcasts the tag and corresponding value that can be updated in those modules now that the instruction result is being committed (in order).

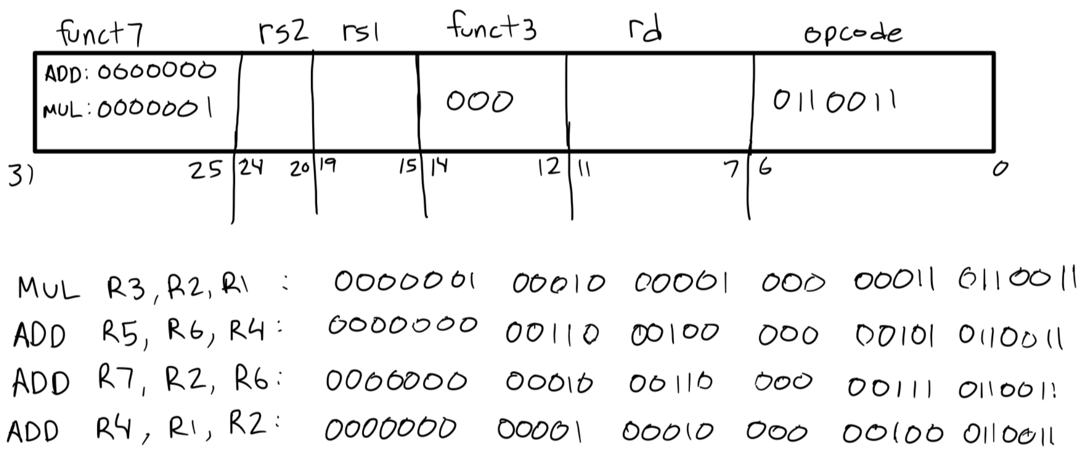
When the RF sets the "new_instr" signal, the ROB loads two instructions at a time at indexes "tail_ptr" and "tail_ptr + 1" and then increments the tail pointer by 2. If the "branch_update" signal is asserted, the exception bit of the entry at the branch_tag is set high. If the ADD FU sends a new result, the value column of the corresponding ROB tag coming from the ADD FU is set to the ADD FU result value. Then, the rest of the ROB is checked to see if there are any invalid sources whose tag matches the tag coming in from the FU, and if so, update the source value to the new value coming from the FU (and validate the source). The same exact functionality happens when the MUL FU sends a new result as well.

For commit logic, the reorder buffer checks the following 4 columns at the entry that the head_pointer is pointing at during every clock cycle: source 1 valid, source 2 valid, no exception, and value > 0. If the head_ptr entry meets these requirements, the head_ptr is broadcasted as the "ROB_bus_tag" and the data inside the value column is broadcasted as the "ROB_bus_value". Then, that entry of the ROB is cleared and the head_pointer is incremented by 1.

For exception handling on branch mispredictions, the ROB clears every entry in between the head pointer and the tail pointer. Also, the ROB tells the Register File to restore the Register-Tag mappings back to how they were before the branch instruction. The ROB tells the Reservation Stations to clear all entries that have a tag that is greater than the tag at which the branch misprediction was detected at.

Testing

The following figure shows the test cases that were used to prove the functionality of the reorder buffer system (without exceptions).



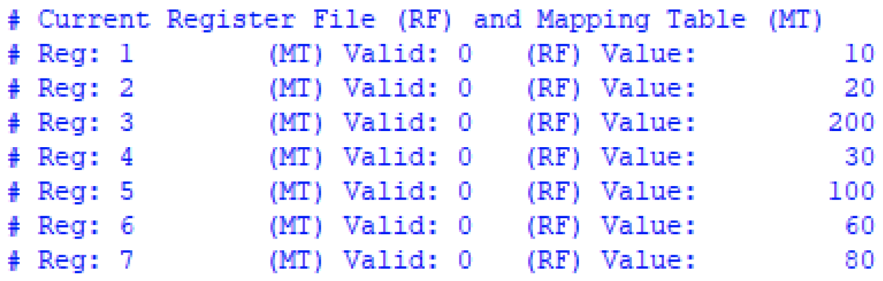
MUL R3, R2, R1 : 0000001 00010 00001 000 00011 0110011
ADD R5, R6, R4 : 0000000 00110 00100 000 00101 0110011
ADD R7, R2, R6 : 0000000 00010 00110 000 00111 0110011
ADD R4, R1, R2 : 0000000 00001 00010 000 00100 0110011

Figure 4: Initial test cases description

To preface, it should be known that the initial state of the register file contains values that are 10x the register number. For example, R1 has the value 10, R2 has 20, R3 has 30, …, and R7 has 70. Of course, at the end of the testing, the new values of each of the destination registers should be as follows:

R3 = 20 * 10 = 200, R5 = 60 + 40 = 100, R7 = 20 + 60 = 80, R4 = 10 + 20 = 30

The following figure shows the state of the Register File at the end of completing these instructions:



```
# Current Register File (RF) and Mapping Table (MT)
# Reg: 1         (MT) Valid: 0   (RF) Value:        10
# Reg: 2         (MT) Valid: 0   (RF) Value:        20
# Reg: 3         (MT) Valid: 0   (RF) Value:       200
# Reg: 4         (MT) Valid: 0   (RF) Value:        30
# Reg: 5         (MT) Valid: 0   (RF) Value:       100
# Reg: 6         (MT) Valid: 0   (RF) Value:        60
# Reg: 7         (MT) Valid: 0   (RF) Value:        80
```

Figure 5: Register File values at the end of running initial test cases (from Figure 4)

Clearly, the destination registers contain the correct values at the end of the tests. This proves that my system can take in 4 instructions and execute them 2-at-a-time and update the register file accordingly.

Now, let's take a look at the waveforms of the reorder buffer during these tests.

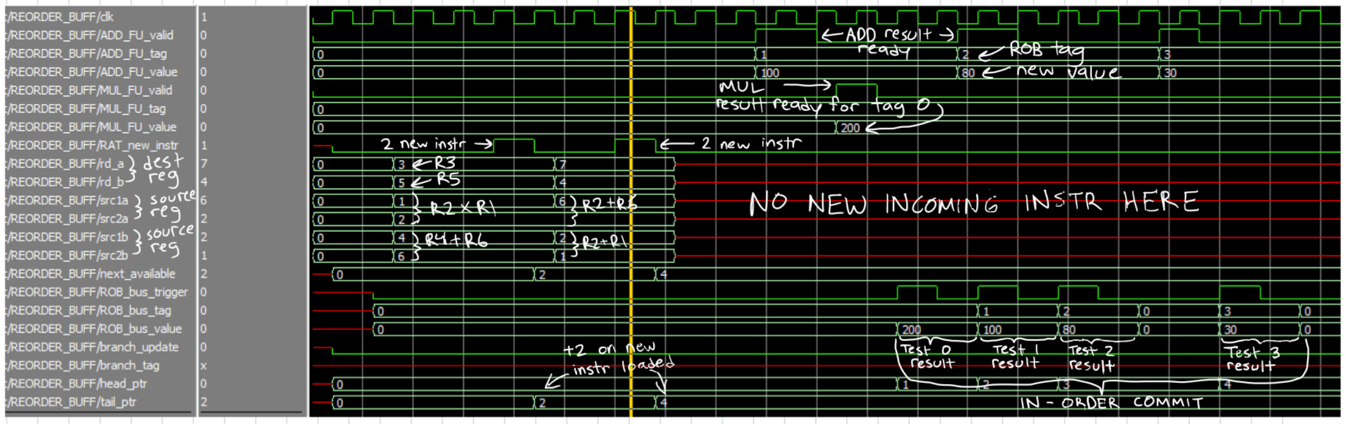
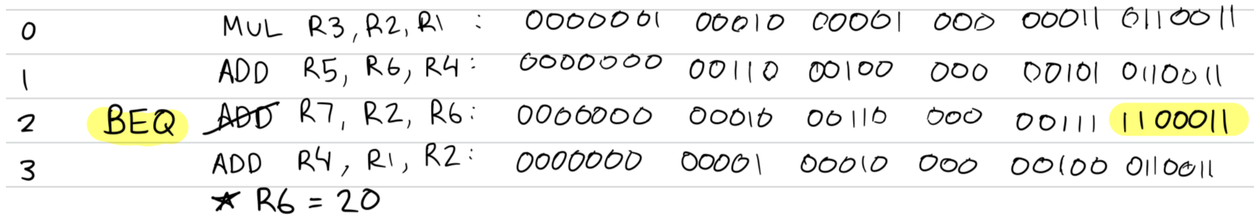


Figure 6: Annotated waveforms for reorder buffer (ROB) behavior during Figure 4 tests

Testing Branch Misprediction Exception Functionality

To test how the system handles exceptions when there is a branch misprediction, the following test cases were used. Note: the value assigned to R6 is now 20 in order to cause an exception at instruction 3.



0		MUL R3, R2, R1 :	0000001	00010	00001	000	00011	0110011
1		ADD R5, R6, R4 :	0000000	00110	00100	000	00101	0110011
2	BEQ	~~ADD~~ R7, R2, R6 :	0000000	00010	00110	000	00111	1100011
3		ADD R4, R1, R2 :	0000000	00001	00010	000	00100	0110011
		★ R6 = 20						

Figure 7: Test cases for exception handling testing

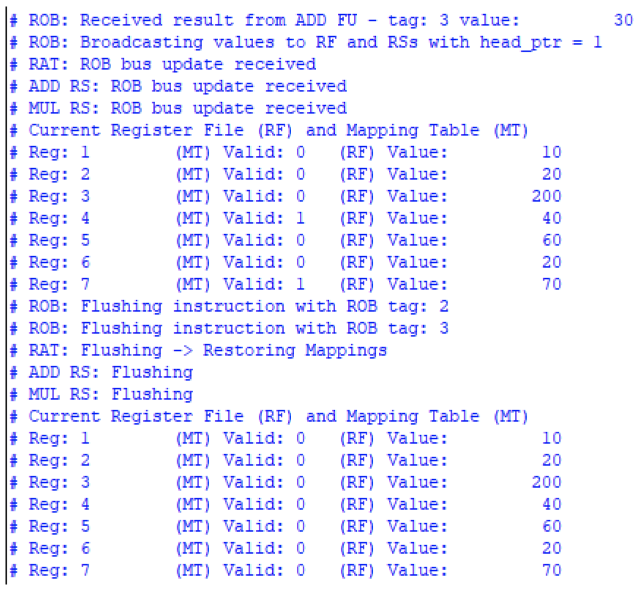
For ease of implementation, BEQ instructions are extracted the same as ADD and MUL instructions, except that the opcode is different so the system knows that this is a BEQ instruction. On a correctly predicted branch (not taken), the value 1 is put into the destination register of the BEQ instruction. However, on a branch misprediction, no new value is assigned to the destination register. Ideally, with no exceptions, the destination registers should have:

R3 = 20 * 10 = 200,   R5 = 20 + 40 = 60,   R7 = good prediction = 1,   R4 = 10 + 20 = 30

However, since I am forcing an exception at instruction 3, the instructions after it are flushed, so the destination registers should have:

R3 = 20 * 10 = 200,   R5 = 20 + 40 = 60,   R7 = misprediction = 70,   R4 = flushed = 40

The following screenshot of the ModelSim output shows that the branch misprediction handling works as expected (flush following instructions, restore mappings):



```
# ROB: Received result from ADD FU - tag: 3 value:         30
# ROB: Broadcasting values to RF and RSs with head_ptr = 1
# RAT: ROB bus update received
# ADD RS: ROB bus update received
# MUL RS: ROB bus update received
# Current Register File (RF) and Mapping Table (MT)
# Reg: 1        (MT) Valid: 0   (RF) Value:         10
# Reg: 2        (MT) Valid: 0   (RF) Value:         20
# Reg: 3        (MT) Valid: 0   (RF) Value:        200
# Reg: 4        (MT) Valid: 1   (RF) Value:         40
# Reg: 5        (MT) Valid: 0   (RF) Value:         60
# Reg: 6        (MT) Valid: 0   (RF) Value:         20
# Reg: 7        (MT) Valid: 1   (RF) Value:         70
# ROB: Flushing instruction with ROB tag: 2
# ROB: Flushing instruction with ROB tag: 3
# RAT: Flushing -> Restoring Mappings
# ADD RS: Flushing
# MUL RS: Flushing
# Current Register File (RF) and Mapping Table (MT)
# Reg: 1        (MT) Valid: 0   (RF) Value:         10
# Reg: 2        (MT) Valid: 0   (RF) Value:         20
# Reg: 3        (MT) Valid: 0   (RF) Value:        200
# Reg: 4        (MT) Valid: 0   (RF) Value:         40
# Reg: 5        (MT) Valid: 0   (RF) Value:         60
# Reg: 6        (MT) Valid: 0   (RF) Value:         20
# Reg: 7        (MT) Valid: 0   (RF) Value:         70
```

Figure 8: Testing results for test cases from Figure 7

The first line in Figure 8 shows that instruction 3 (ADD R4, R1, R2) was run and placed into the ROB. However, we see that in the final register file at the bottom of Figure 8, R4 still holds the value of 40 (not 30), which means that instruction 3 was not committed due to the exception that was detected and handled at instruction 2.

In between the 2 printings of the register file, we see that the ROB flushes the instructions in between the head pointer and the tail pointer. Also, the Register File (shown as "RAT") restores the mappings of those instructions that were flushed. We can see that this works by observing the "valid" bits in both of the register file printings. Finally, in the reservation stations, instructions that have an ROB tag greater than the tag at which the exception occurred are cleared.

These testing sections have shown the correct, working functionality of the 2-wide superscalar reorder buffer system with out-of-order execution and in-order commit. Various test cases were used to prove the required features of the project, and waveforms were shown to visualize how I/O signals behave during these tests.

As a final remark, I want to sincerely thank Dr. Liu Liu for all that I have learned from him in both Computer Hardware Systems and Advanced Computer Architecture as this material has set me up for success in my future hardware engineering endeavors.