Computer Architecture: CSEE 4824
Columbia University

Final Project: Final Report

May 2024

Aarya Agarwal (ASA2276)	asa2276@columbia.edu
Irfan Tamim (IT2304)	it2304@columbia.edu
Syed Muhammad Raza (SMR2263)	smr2263@columbia.edu
Ilgar Mammadov (IM2703)	im2703@columbia.edu
Lauren Chin (LMC2265)	lmc2265@columbia.edu

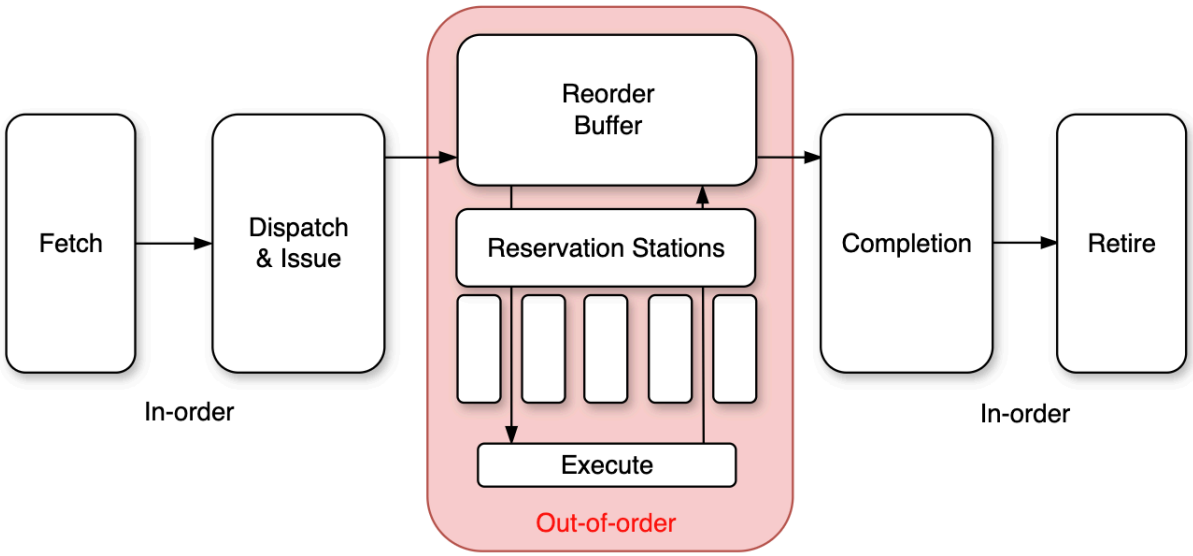
Contents

List of Figures and Tables

1 | Introduction

Our final report outlines the completion of our processor, inspired by Intel's P6 microarchitecture. Over the past several weeks, we designed and wrote a processor using SystemVerilog that captures the spirit of the P6 while emphasizing scalability and high performance, without sacrificing efficiency or throughput. Our team was able to successfully complete part of this project through incremental milestones, each time targeting different points within the pipeline.

We implemented an out-of-order pipelined processor architecture with five distinct stages:



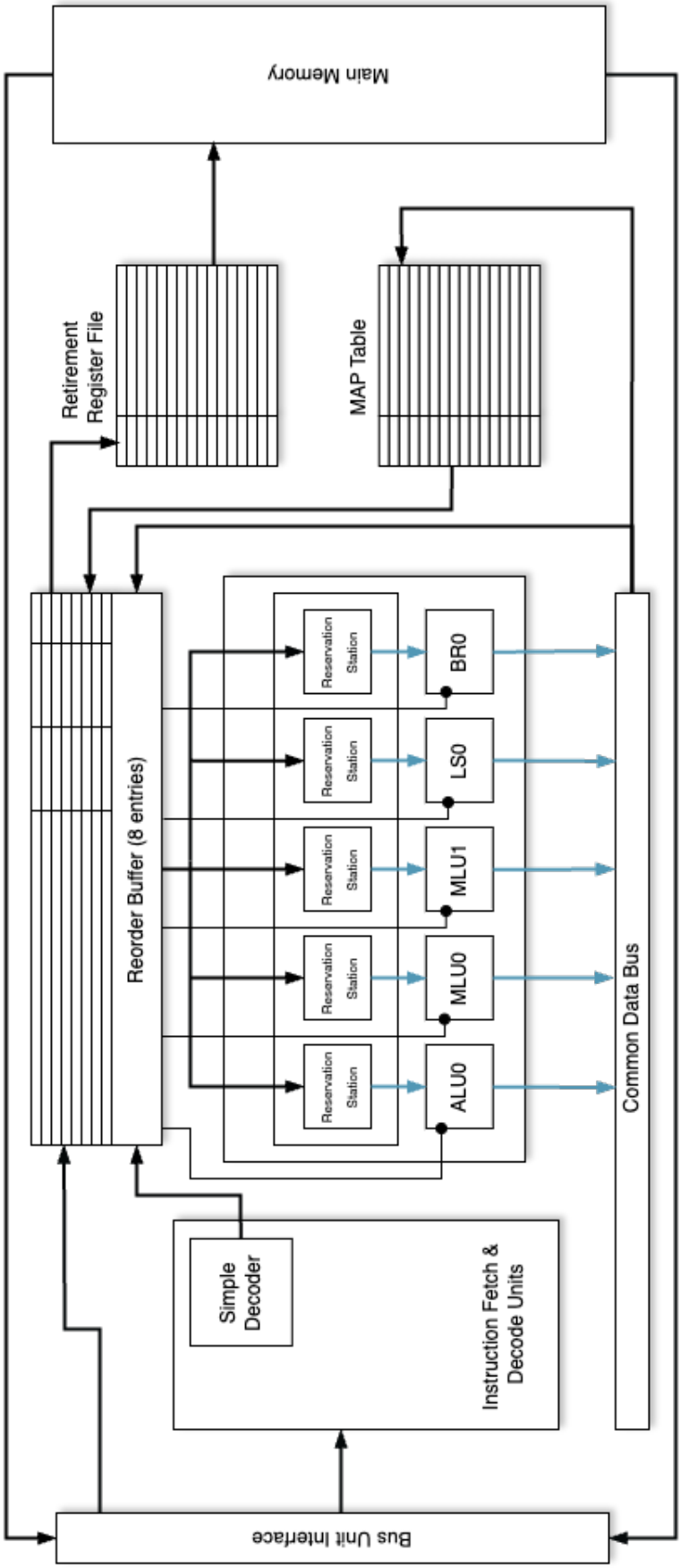
The IF stage fetches instructions from the unified memory bus based on the current program counter (PC) value. It sends the fetched instruction and the next PC to the IF/ID pipeline register. The ID stage then decodes the fetched instruction, and outputs a 3-bit `opcode` that is used to index into its appropriate functional unit, in preparation for EX stage.

The EX stage consists of a reservation station (RS) and a reorder buffer (ROB) to handle out-of-order execution and dependency resolution and five functional units, including an arithmetic logic unit (ALU), two multipliers, a load/store unit, and a branch unit. These functional units execute the instructions based on the information received from the ID stage, or wait in the

ROB / RS until its operands become available: either from the common data bus (CDB), the architectural register file, or the ROB. The results from the functional units are broadcast on the CDB to update the RS and the ROB.

The Complete stage handles memory operations, such as loads and stores. It also performs memory address calculation and handles memory access requests both to and from main memory, interfacing with the unified memory interface. The Retire stage then clears the instruction from the ROB, moving the head at the same time.

Below is a highlight of the progress with a block diagram showing the implemented P6 structure. Additionally, this report outlines our progress throughout this project and breaks down the choices we made.


Main Memory
Retirement Register File
MAP Table
Reorder Buffer (8 entries)
Reservation Station
Reservation Station
Reservation Station
Reservation Station
Reservation Station
BR0
LS0
MLU1
MLU0
ALU0
Common Data Bus
Simple Decoder
Instruction Fetch & Decode Units
Bus Unit Interface

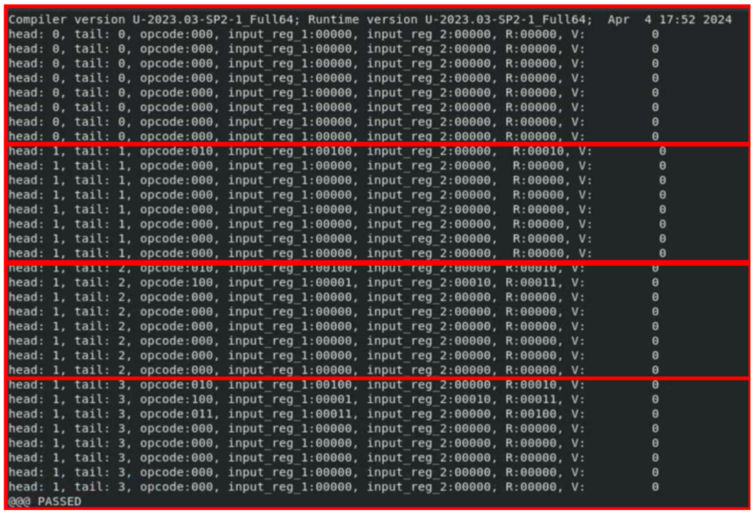
2 | Architectural Design & Features

2.1 Out-of-Order Execution

2.1.1 Reorder Buffer (ROB)

When an instruction is dispatched, it is written into the reorder buffer (ROB). The buffer has a `head` and `tail`: two pointers that mark the first and last instruction in the ROB, with the first instruction representing the oldest, and the `tail` representing the youngest instruction (its age represents instruction age in program order). Initially, both the head and tail pointers are 0 and then move to 1 together with the first instructions. Starting from the next cycle, tail is increased by one until the buffer is full in each instruction dispatch. For that purpose, we have a ”valid” signal in our ROB module.

Instructions are completed when it reaches the `head` of the ROB, and must have already successfully completed execution and have a valid result. When an instruction reaches the `head` of the ROB, if that instruction is marked ready for completion (`commit`), its entry will be cleared from the ROB and the `head` will be incremented such that it points to the next entry in the ROB on the next rising edge. When a commit signal is asserted, the instruction marked with ”head” is considered as complete and ”head” is increased by 1. With wraparound functionality, our limited size ROB is able to handle however many instruction are necessary for a given program. In our current implementation, the ROB has 7 spots for instruction but this number could easily be increased in future implementations.



```
Compiler version U-2023.03-SP2-1_Full64; Runtime version U-2023.03-SP2-1_Full64;  Apr  4 17:52 2024
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 0, tail: 0, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 1, opcode:010, input_reg_1:00100, input_reg_2:00000,  R:00010, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 1, opcode:000, input_reg_1:00000, input_reg_2:00000,  R:00000, V:         0
head: 1, tail: 2, opcode:010, input_reg_1:00100, input_reg_2:00000, R:00010, V:         0
head: 1, tail: 2, opcode:100, input_reg_1:00001, input_reg_2:00010, R:00011, V:         0
head: 1, tail: 2, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 2, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 2, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 2, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 2, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 2, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 3, opcode:010, input_reg_1:00100, input_reg_2:00000, R:00010, V:         0
head: 1, tail: 3, opcode:100, input_reg_1:00001, input_reg_2:00010, R:00011, V:         0
head: 1, tail: 3, opcode:011, input_reg_1:00011, input_reg_2:00000, R:00100, V:         0
head: 1, tail: 3, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 3, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 3, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 3, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
head: 1, tail: 3, opcode:000, input_reg_1:00000, input_reg_2:00000, R:00000, V:         0
@@@ PASSED
```

Figure 2.1: ROB-test-results

Initial testing of our `ROB` module was conducted with three separate instructions, as seen above. Inputs were properly received; each column in the table above was set appropriately, per the test instructions.

2.1.2 Reservation Station (RS)

We designed and implemented our Reservation Station (RS) to accommodate up to five parallel instructions simultaneously. Fetched and decoded instructions are sent one by one to the Reorder Buffer (ROB), where each instruction is assigned a numerical tag that is used to maintain original program order, as the instruction enters the out-of-order part of the pipeline.

This ROB number plays a critical role in preserving the original instruction sequence, ensuring that even as instructions are executed out-of-order to maximize processing efficiency, their results are completed correctly, in original program order.

2.1.3 The MAP Table

This map table implementation is fairly simple, focused on reliably tracking which registers are in use by which instructions.

2.2 Functional Units

Our processor architecture has five unique functional units: one ALU, one `load/store`, one branch unit, and two pipelined `mult`ipliers that can calculate two parallel 32 by 32-bit products in four clock cycles.

Pipelined 4-Stage Multiplier Unit

Our `mult` module is pipelined into four separate stages: pipelining was achieved through the use of multiple `mult_stage` modules connected in sequence. Each `mult_stage` represents a pipeline stage that performs one quarter of the 64-bit multiplication operation.

The pipelining works as follows:

1. In the first clock cycle, `stage0` sets the initial 32-bit `multiplier` value from operand A and the `multiplicand` from operand B of the original RISC-V instruction, and then starts the multiplication process. It calculates the partial product and forwards the updated `multiplier`, `multiplicand`, and `partial_prod` to `stage1`.
2. In the second clock cycle, `stage1` receives the outputs from `stage0` and continues the multiplication process. It calculates the partial product based on the updated `multiplier` and `multiplicand` values and passes the results to `stage2`. Simultaneously, `stage0` can start processing a new set of inputs.
3. This process continues through `stage2` and `stage3`, with each stage performing its portion of the multiplication and passing the results to the next stage. At any given clock cycle, multiple multiplication operations can be in progress at each stage of the pipelined multiplier.
4. After the last stage (`stage3`), the final product is obtained, and the `done` signal is asserted to indicate the completion of the multiplication operation.

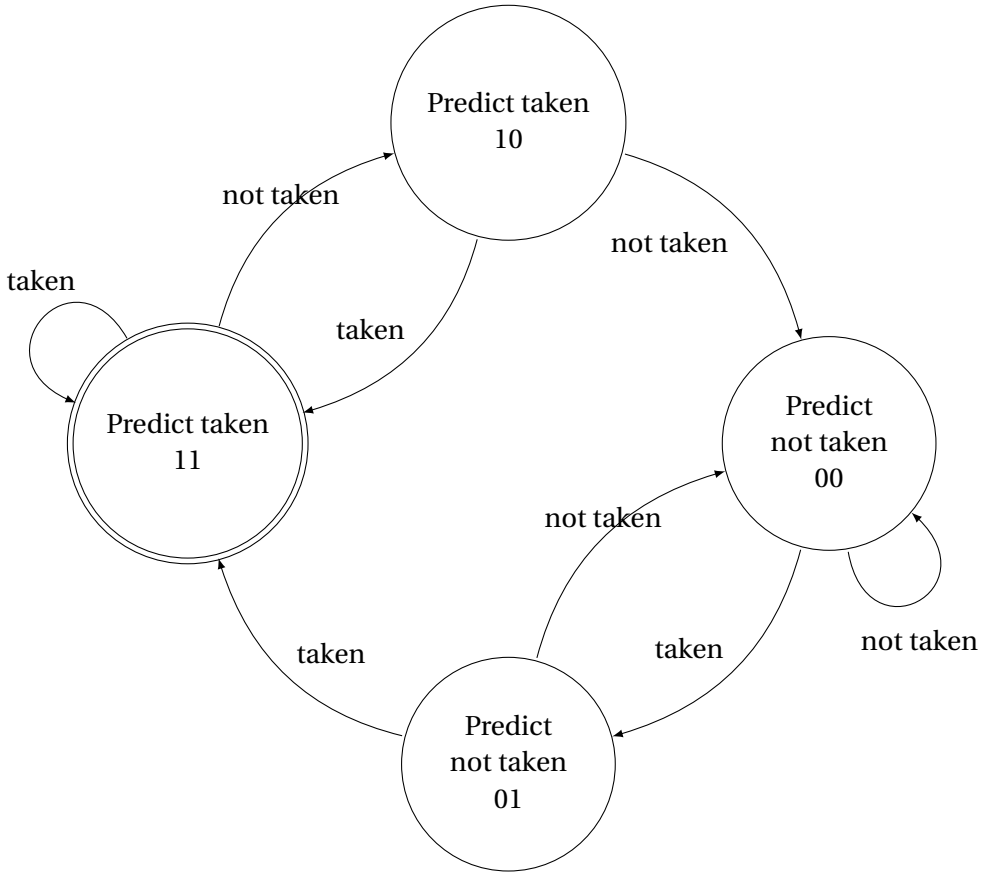
Our four-cycle multiplier successfully computes `MUL`, `MULH`, and `MULHU`. Unfortunately, we were unable to figure out how to compute `MULHSU` correctly, as the carry bit seemingly does not properly carry over.

We also developed a standalone multiplication testbench for our multiplier, in order to test edge cases for each of the RISC-V multiplication extension arithmetic functions (`MUL`, `MULH`, `MULHU`, and `MULHSU`).

Our multiplier will terminate early if the product can be determined at an earlier stage, i.e., an instruction to multiply `32'h00` by another number will not require four cycles to complete.

2.3 Branch Prediction

As required by the instructions of the project, we included a branch predictor that went beyond just predicting whether a branch was found or not found. The finite state machine below illustrates the branch predictor used:



3 | Implementation

3.1 Hazard Detection & Resolution

3.1.1 Read After Write Hazards

In our single-issue, out of order pipeline, RAW hazards are handled in the Reservation station by tagging an instruction with the dependency based on information found in the Map Table.

The Reservation Station will stall the dependent instruction until it sees that the necessary information is available on the CDB at which point the Read will occur ensuring that the write can happen simultaneously in a different part of the pipeline.

3.1.2 Write After Read Hazards

Write after Read hazards are handled by the Reservation station in our implementation. Whenever an instruction needs to read a register, it just stores the value into the reservation station, ensuring that any WAR hazards that could potentially change the value before the first register finishes executing will have no impact.

3.1.3 Write After Write Hazards

In our single-issue, out-of-order execution pipeline, *write-after-write (WAW) hazards* are handled dynamically by the use of register renaming and a reorder buffer (ROB), similar to how P6 resolves WAW hazards.

Example: Two multiplication instructions with the same destination register

Consider two `MUL` instructions, executing in parallel:

```
MUL  R1,  R2,  R3
MUL  R1,  R4,  R5
```

IF & ID

- **Cycle 1:** the first `MUL R1, R2, R3` instruction is fetched and issued.
- **Cycle 2:** the second `MUL R1, R4, R5` instruction is fetched and issued.

Out-of-Order Execution

Suppose both multiplication functional units (FUs) are available upon dispatch of the older instruction to the reservation station (RS). Upon entering the RS, the older instruction cannot begin execution due to unresolved data dependencies on its operands, necessitating a two cycle stall. Conversely, the younger instruction is able to begin execution immediately upon arrival at the RS.

Execution Completion

Recall that our multiplication FUs require four cycles to compute a 32-bit product. Since the younger instruction commenced execution first, it completes first. However, to avoid a WAW hazard and ensure that instructions write to their destination registers in program order, the ROB manages the completion sequence of the instructions, independent of when they finish executing.

Lastly, the potential issue of overlapping register names is effectively addressed through register renaming, facilitated by our MAP table (i.e., register alias table).

4 | Testing

Individual testbench programs were developed in parallel with the development of each individual component of our single-issue pipeline, to ensure correct and efficient functionality.

To achieve a functional P6 architecture, we needed to implement several stages:

- Instruction Fetch
- Instruction Dispatch and Issue
- Execution
- Complete
- Retire

Below we have described the results of all these stages to show that they are functioning properly: Instruction fetch was a simple module from the Project 3 with minimal changes required. We added the branch predictor detailed above in chapter 2. Outside of that the module is more or less the same, thus we start the documentation with dispatch.

4.1 Dispatch and issue

```
ROB Contents:
Head:   1,   Tail:   3, Buffer Full: 0
------------------------------------------------------------
Operation codes: 001 - ALU, 010 - MULT, 011 - LD/ST, 100 -BRANCH

+-------------+---------+---------+----------+-----------------+
| Inst opcode |   RS1   |   RS2   |   Dest   |   V             |
+-------------+---------+---------+----------+-----------------+
 xxx          |   0     |   0     |   0      |   0             |
 001          |   0     |   0     |   6      |   0             |
 011          |   0     |   0     |   2      |   0             |
 000          |   0     |   0     |   0      |   0             |
 000          |   0     |   0     |   0      |   0             |
 000          |   0     |   0     |   0      |   0             |
 000          |   0     |   0     |   0      |   0             |
 +-------------+---------+---------+----------+-----------------+

Reservation Stations
+----+------+-----+-------+----+----+--------------------------------+----+--------------------------------+
| n  | busy | run | FU    | n  | T1 | V1                             | T2 |                                |
+----+------+-----+-------+----+----+--------------------------------+----+--------------------------------+
| 00 | 0    | x   | --    | 00 | 00 | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
| 01 | 0    | 1   | ALU0  | 01 | 00 | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
| 02 | 1    | 1   | L/S0  | 02 | 00 | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
| 03 | 0    | 0   | BR0   | 00 | 00 | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
| 04 | 0    | 0   | MULT0 | 00 | 00 | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
| 05 | 0    | 0   | MULT1 | 00 | 00 | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
+----+------+-----+-------+----+----+--------------------------------+----+--------------------------------+

    Map Table (only non-zero registers are displayed):
      Register      2:  2, Ready_bit: 0
      Register      6:  1, Ready_bit: 0
```

It can be seen that 2 instructions have been dispatched and issued successfully to the reservation station and reorder buffer. The map table holds the destination registers and sets the completed instructions' destination registers as ready-in-rob.

4.2 Execution and complete

Similarly, in the figure, it can be seen that the 3rd instruction is being executed and the 2nd instruction has completed its execution, writing its result (4096) into the CDB and the "Value" fields of the Reorder Buffer, which is the "complete" stage.

```
-----------------------------------------------------
CDB tag= 2   |   CDB value=      4096   |    CDB valid=1   |
-----------------------------------------------------
  ROB  3 is currently executing, ID_EX_PACKET:
      Instruction: 00000000101000000000111110010011
      PC:    8                  NPC (PC+4):   12
      Dest Reg Index:   31      RS1 Value:    0      RS2 Value:          x
      ALU Function: Unknown Function
      Read Memory: 0          Write Memory: 0
      Conditional Branch: 0  Unconditional Branch: 0
      Halt: 0, Illegal Instruction: 0, CSR Operation: 0
      Valid: 1
   Stage_EX: ROB  3 executing now
```

Figure 4.1: Execution and Complete

```
-----------------------------------------------------
CDB tag= 2   |   CDB value=      4096   |    CDB valid=1   |
-----------------------------------------------------
ROB Contents:
Head:  1, Tail:  6, Buffer Full: 0
-------------------------------------------------------------------------------------
Operation codes: 001 - ALU, 010 - MULT, 011 - LD/ST, 100 - BRANCH
|  Instruction          | RS1          | RS2          | DEST          | V          |
  xxx                  |  0           |  0           |           0           |          0          |
  001                  |  0           |  0           |           6           |          0          |
  011                  |  0           |  0           |           2           |       4096          |
  001                  |  0           | 10           |          31           |          0          |
  010                  |  6           | 31           |           3           |          0          |
  011                  |  2           |  3           |           0           |          0          |
  000                  |  0           |  0           |           0           |          0          |
  000                  |  0           |  0           |           0           |          0          |
-------------------------------------------------------------------------------------
```

Figure 4.2: Execution and Complete 2

4.3 Retire

In the figure, it can be observed that while the third instruction has completed its execution and has written its result to the ROB, the first instruction has retired from the ROB and its content has been cleared, with the head moving to the next instruction.

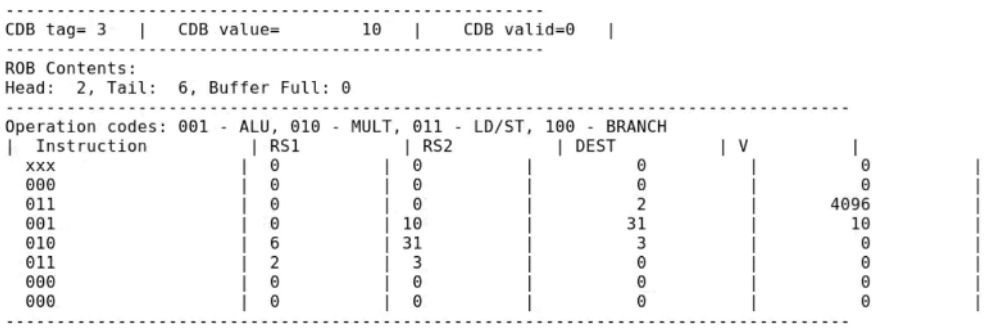


Figure 4.3: Retire

4.4 Writeback to regfile

The writeback results have been written to the ".wb" file. Although our system worked perfectly including that stage, we encountered difficulty with resolving branches, causing the program to stop at the end of the first iteration and not run a second iteration. Hence, we can only display the first iteration's writeback results.

```
PC=00000000, REG[ 6]=00000000
PC=00000004, REG[ 2]=00001000
PC=00000008, REG[31]=0000000a
PC=0000000c, REG[ 3]=00000000
PC=00000014, REG[ 4]=xxxxxxxx
PC=0000001c, REG[ 2]=00001008
PC=00000020, REG[ 6]=00000001
PC=00000024, REG[ 5]=00000001
PC=00000028, REG[ 5]=0000000c
```

Figure 4.4: Writeback

5 | Progress Relative to the Original Schedule & Direction

5.1 Milestone 1: Reservation Station

We successfully implemented the Reservation Station (RS) component of the P6 architecture to support dynamic scheduling of instructions and out-of-order execution.

In comparison to our original scope of our project, there are no significant changes. As stated in the group proposal, this phase has focused on establishing a foundational component for handling instruction dependencies and facilitating out-of-order execution. However, we do plan on implementing a priority system for instructions to ensure that critical instructions are executed first. We are also planning on completing dynamic scheduling, which allows us to vary workload characteristics. This is anticipated to be completed during the next phase of our project, leading up to Milestone #2.

As per our original proposal, every member had to put in 20-25 hours per week to the project. All members have been consistent in their weekly hours. This has been critical in not only keeping the project on track but also in keeping the working environment conducive. The dedication of the team members was important in keeping the project on schedule.

5.1.1 Test suite development

We created additional test cases that model common, real world scenarios, along with developing a series of complex instruction sequences in order to ensure the robustness, efficiency, and correctness of our RS.

Common use cases

- **Loop constructs**
- **Concurrent data accesses**
- **System calls and I/O operations**

Edge cases

- **Dependency chains:** sequences that test our RS's ability to handle data hazards and manage dependencies both efficiently and correctly.
- **Branch prediction challenges:** sequences that involve complex branching logic. For example, nested branches and branches that depend on the outcome of previous computations.
- **Memory pressure scenarios:** sequential, heavily interleaved `read/write` operations that require haphazard, spatially distant data accesses.

This approach is aimed at identifying and mitigating potential bottlenecks, optimizing resource allocation, and ensuring overall robustness and correctness.

5.1.2 Test results of Milestone 1 RS module

Once we completed the Reservation Station module, in order to check its functionality, we created a testbench and simulated the first 3 cycles of the P6 example given in the course presentation.

These cycles from presentation and test results are shown in the following figures:

Reorder Buffer							Map Table		CDB	
ht	#	Instruction	R	V	S	X	C	Register	T+	T	V
ht	1	ldf X(r1), f1	f1					f0			
	2	mulf f0, f1, f2						f1	ROB #1		
	3	stf f2, Z(r1)						f2			
	4	addi r1, 4, r1						f3			
	5	ldf X(r1), f1									

Figure 5.1: P6: Cycle 1

Reorder Buffer							Map Table		CDB	
ht	#	Instruction	R	V	S	X	C	Register	T+	T	V
ht	1	ldf X(r1), f1	f1		c2			f0			
t	2	mulf f0, f1, f2	f2					f1	ROB #1		
	3	stf f2, Z(r1)						f2	ROB #2		
	4	addi r1, 4, r1						f3			
	5	ldf X(r1), f1									

Figure 5.2: P6: Cycle 2

In the test results figure, we iterate through all cells of the RS table to check the accuracy. Each 5-line output corresponds to one cycle.

```
Reset:1  busy_signal:00000 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:1  busy_signal:00000 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:1  busy_signal:00000 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:1  busy_signal:00000 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:1  busy_signal:00000 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:00010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:00010 T:         1 T1:         0 T2:         0 V1:         4 V2:         6 out_opcode:010
Reset:0 busy_signal:00010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:00010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:00010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:01010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:01010 T:         1 T1:         0 T2:         0 V1:         4 V2:         6 out_opcode:010
Reset:0 busy_signal:01010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:01010 T:         2 T1:         0 T2:         1 V1:         8 V2:         0 out_opcode:100
Reset:0 busy_signal:01010 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:01100 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:01100 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
Reset:0 busy_signal:01100 T:         3 T1:         2 T2:         0 V1:         0 V2:        14 out_opcode:011
Reset:0 busy_signal:01100 T:         2 T1:         0 T2:         1 V1:         8 V2:         0 out_opcode:100
Reset:0 busy_signal:01100 T:         0 T1:         0 T2:         0 V1:         0 V2:         0 out_opcode:000
@@@ PASSED
```

Figure 5.3: Test-Results

As can be seen from the results, initially the reset signal is asserted and all cells of the RS table are 0. In the first cycle, the LD command is dispatched, so the busy signal illustrates that the corresponding row for the LD unit (row 2) is busy. Similarly, T has been set to 1 to show that the ROB1 is currently processed in the LD unit, and V1 and V2 store the corresponding values of the input registers. Another functionality, which is emptying the unit can be observed in the third cycle, when the LD is completed and done signal is sent to the RS module, so the content of the corresponding cells are removed.

We believe that our RS module work correctly and we can proceed to create other modules and integrate them into the whole project.

5.2 Milestone 2: Other Key Modules

5.2.1 Test suite development

We plan on creating additional test cases that model common, real world scenarios, along with developing a series of complex instruction sequences in order to ensure the robustness, efficiency, and correctness of our RS.

Common use cases

- **Loop constructs**
- **Concurrent data accesses**
- **System calls and I/O operations**
- **Floating-point calculations**

Edge cases

- **Dependency chains:** sequences that test our RS's ability to handle data hazards and manage dependencies both efficiently and correctly.
- **Branch prediction challenges:** sequences that involve complex branching logic. For example, nested branches and branches that depend on the outcome of previous computations.
- **Memory pressure scenarios:** sequential, heavily interleaved `read`/`write` operations that require haphazard, spatially distant data accesses.

This approach is aimed at identifying and mitigating potential bottlenecks, optimizing resource allocation, and ensuring overall robustness and correctness.

5.2.2 Test results of Milestone 2 Map Table module

As can be seen with the companion testbench for the map table, we tested our 2-wide map table against the same variety of situations that the reservation station was tested against, checking every cycle that the table updated properly. With the map table successfully passing the test suite, we believe that our implementation works properly and should work when integrated with the rest of our project.

5.3 Milestone 3: Pipeline Integration

5.3.1 Integration Process

This milestone was focused on integrating the modules from previous milestones and taking the first steps towards a comprehensive processor. We started by implementing the ROB and RS within the pipeline.sv file and ensuring that data was properly passing between the two of them. While doing this, we decided to scrap the map table module from milestone 2 and take a different approach. We integrated our map-table into the RS with the requisite logic just being handled by the RS. This simplified the implementation within the pipeline.sv file and should provide simplification going forward.

5.3.2 Execution Units

This milestone also saw us revising our approach to execution units and how they are going to be integrated into our process. We started by modifying the ALU module from project 3 to create 3 separate modules for ALU, LD, and ST operations. We also made 2 multiplication units for a total of 5. The reason we choose 3 multiplication units was the far higher CPI for these instructions, necessitating that they be computed in parallel.

5.3.3 Test Suite Development

We used a similar credence to previous milestones for this one as well, using the sample instructions provided in the notes, measuring output signals across our modules to ensure proper functionality. We have included the pipeline-test.sv file with the ability to simulate these instructions, comparing the outputs with the expected values. As you can see by executing this testbench, we have proper functionality. Our test suite is sure to address most of the following issues:

Common use cases

- **Loop constructs**
- **Concurrent data accesses**
- **System calls and I/O operations**
- **Floating-point calculations**

Edge cases

- **Dependency chains:** sequences that test our RS's ability to handle data hazards and manage dependencies both efficiently and correctly.

- **Branch prediction challenges:** sequences that involve complex branching logic. For example, nested branches and branches that depend on the outcome of previous computations.
- **Memory pressure scenarios:** sequential, heavily interleaved `read`/`write` operations that require haphazard, spatially distant data accesses.

6 | Changes in the Direction or Scope

6.1 Changes in the Direction or Scope of the Project Relative to the Original Proposal

In the original project we had planned to do the following features/advanced features

- 2-way Superscalar: Enables the processor to execute multiple instructions simultaneously, significantly increasing throughput.
- Pre-fetching Instruction Cache: Improves efficiency by fetching instructions before they are needed, reducing wait times.
- Load Store Queue: Facilitates the reordering of memory operations to optimize execution flow.
- Non-blocking L1 Data Cache: Allows the processor to continue execution even when some data accesses are unresolved, reducing stalls.
- Branch Target Buffer: Predicts the targets of branches to minimize the impact of control hazards.

With the time that it took to implement the base processor, we were unable to implement all these advanced features for the additional points associated with them.

7 | Appendix

```
====== Cycle 12: Dispatched ADDI inst: 00000000010000000000001000010011 (NPC: 00000014) ======
-----------------------------------------------------
CDB tag= 3   |   CDB value=         8   |    CDB valid=1   |
-----------------------------------------------------
GOT THE VAL TO REGFILE, IDX =  2 data=         2
  ROB  4 is currently executing, ID_EX_PACKET:
      Instruction: 00000000010000000000001000010011
      PC:   12                  NPC (PC+4):   16
      Dest Reg Index:    4      RS1 Value:    0      RS2 Value:          x
      ALU Function: Unknown Function
      Read Memory: 0         Write Memory: 0
      Conditional Branch: 0  Unconditional Branch: 0
      Halt: 0, Illegal Instruction: 0, CSR Operation: 0
      Valid: 1
   Stage_EX: ROB  4 executing now
ROB Contents:
Head:  2, Tail:  5, Buffer Full: 0
-------------------------------------------------------------------------------------
Operation codes: 001 - ALU, 010 - MULT, 011 - LD/ST, 100 - BRANCH
|  Instruction          | RS1          | RS2          | DEST          | V          |
  xxx                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  001                  |  0          |  2          |          2          |          2          |
  001                  |  0          |  8          |          3          |          8          |
  001                  |  0          |  4          |          4          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
-------------------------------------------------------------------------------------
```

Figure 7.1: 4th Instruction is being executed while 3rd instruction is completed and the result is added to ROB Value field

```
ROB Contents:
Head:  1, Tail:  3, Buffer Full: 0
-------------------------------------------------------------------------------------
Operation codes: 001 - ALU, 010 - MULT, 011 - LD/ST, 100 - BRANCH
|  Instruction          | RS1          | RS2          | DEST          | V          |
  xxx                  |  0          |  0          |          0          |          0          |
  001                  |  0          |  0          |          6          |          0          |
  011                  |  0          |  0          |          2          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
  000                  |  0          |  0          |          0          |          0          |
-------------------------------------------------------------------------------------
Reservation Stations
    +----+------+------+--------+------+-----+----------------------------------+-----+----------------------------------+
    | #  | busy | run  |  FU    | ROB# | T1  |                V1                | T2  |                V2                |
    +----+------+------+--------+------+-----+----------------------------------+-----+----------------------------------+
    | 00  |  0  |  x   |  ????  |  00  |  00  | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
    | 01  |  0  |  1   |  ALU0  |  01  |  00  | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
    | 02  |  1  |  1   |  L/S0  |  02  |  00  | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
    | 03  |  0  |  0   |  BR0   |  00  |  00  | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
    | 04  |  0  |  0   |  MUL0  |  00  |  00  | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
    | 05  |  0  |  0   |  MUL1  |  00  |  00  | 00000000000000000000000000000000 | 00 | 00000000000000000000000000000000 |
    +-----+-----+------+--------+------+-----+----------------------------------+-----+----------------------------------+

    Map Table (only non-zero registers are displayed):
     Register           2:  2, Ready_bit: 0
     Register           6:  1, Ready_bit: 0
```

Figure 7.2: Instruction Dispatch enters the new instruction to ROB and RS

Figure 7.3: Writeback to regfile in copy.out

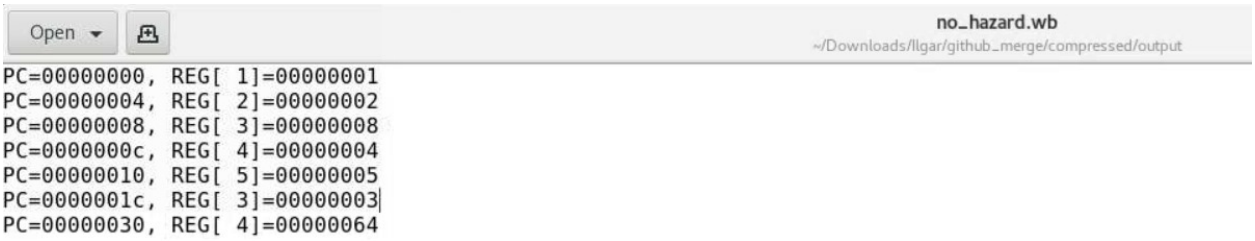


Figure 7.4: Writeback to Regfile in no$_h azard.out$