ECE 411 Computer Org & Design Report
Fall 2024
December 11th, 2024

Names:

Derek Chaw

Qingyuan Liu

NetIDs:

drchaw2

ql21

1. Introduction

The machine project Out-of-Order processor is the final one in ECE 411 and it includes everything that we have learned in the class, including pipeline, cache, Tomasulo out-of-order architecture, and so on. Below is the overall structure of Tomasulo we follow to implement our design and we will add more features to it as well as parameterize all the depth and lengths shown in the picture. Throughout the total of 6 weeks, we have implemented the out-of-order processor step by step from scratch. The fetch stage was finished in the first checkpoint, an overall out-of-order system without branch and memory operation was finished in the second checkpoint, and in the last checkpoint, an out-of-order processor that can pass all RV32IM instructions, verified by the provided test codes was implemented. We further explore some improvement possibilities by implementing some of the advanced features.

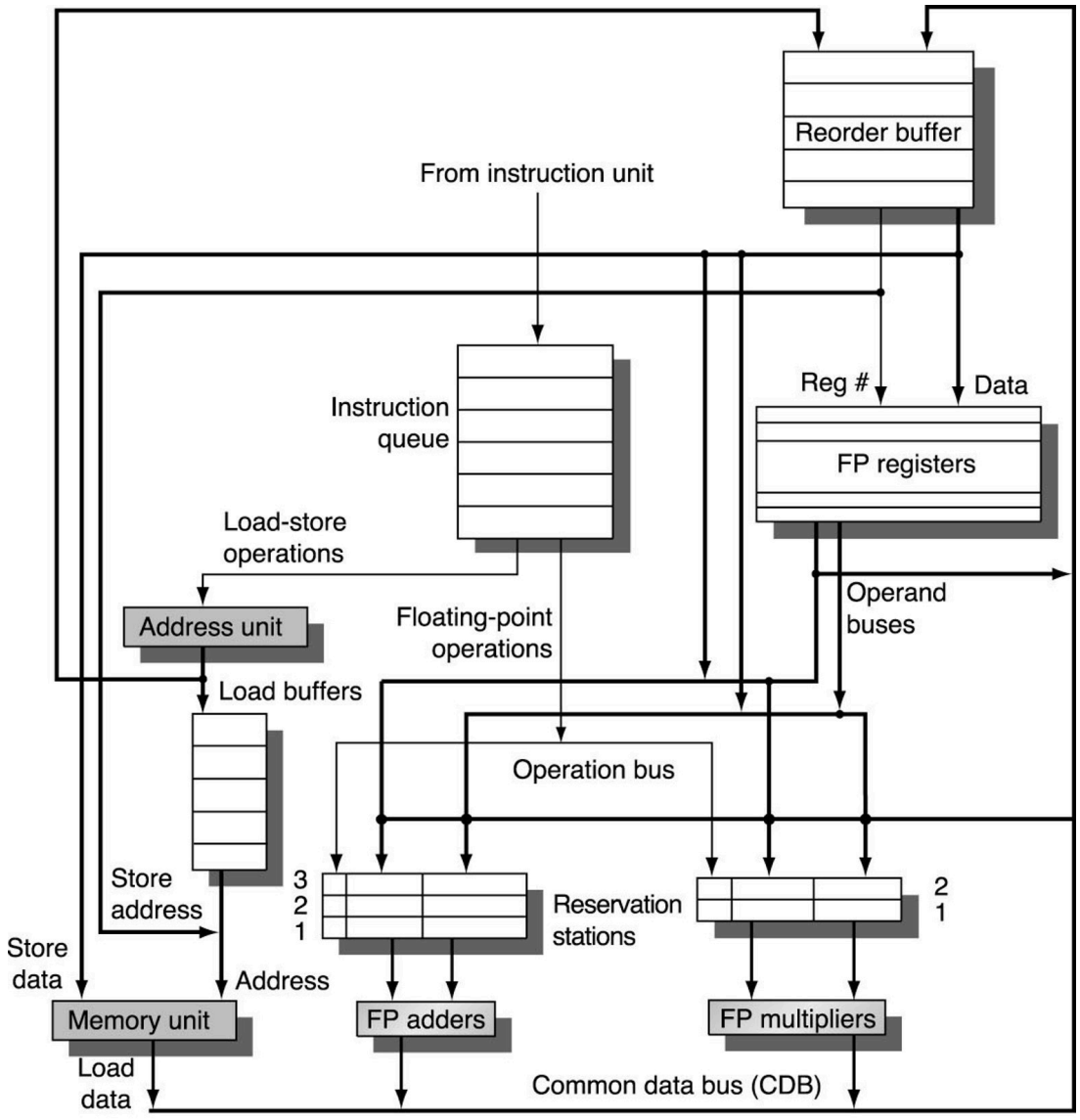

Fig 1: Tomasula overview

2. Project Overview

Our project goal is to implement an Out-of-Order processor that can pass all the RV32IM instructions. It will have an instruction queue to fire the instructions in order and then reservation stations can receive the instructions to do out-of-order processing. The block diagram of the overall structure is shown in Figure 2(initial version) and Figure 3(finalized version).

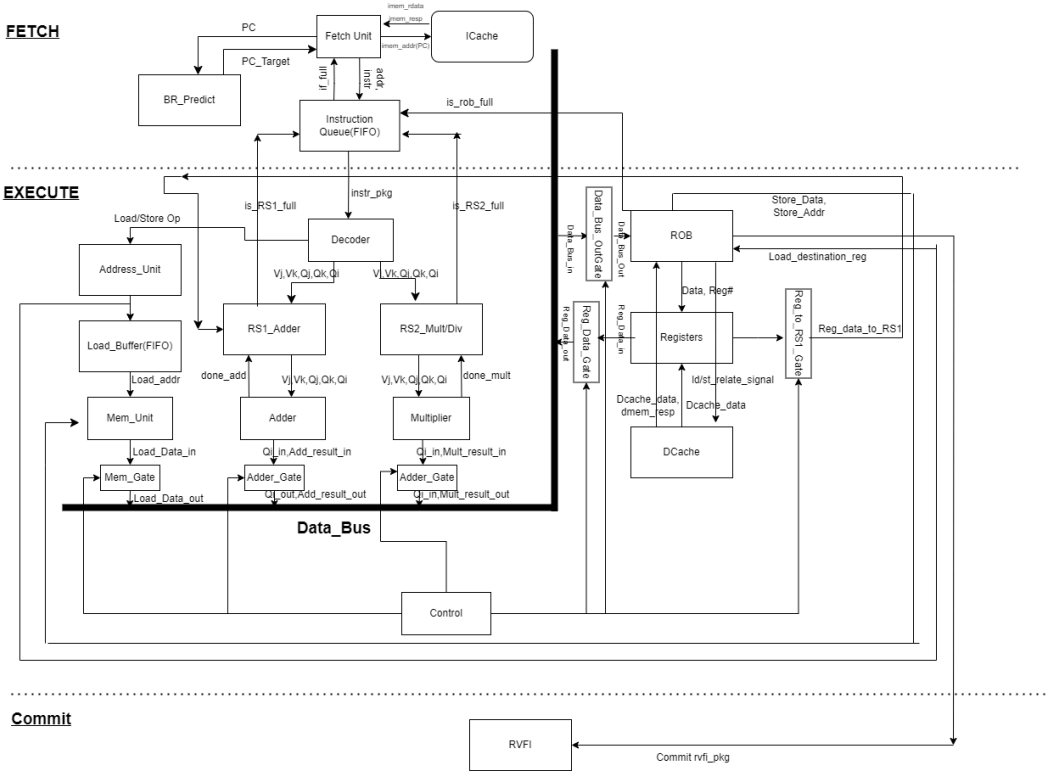

Fig 2: Original Block Diagram

	RS	CDB	ROB	IQ	LSQ
Number	4	4	1	1	2
Depth	4	N/A	8	16	4

Table 1: Size and number of each block

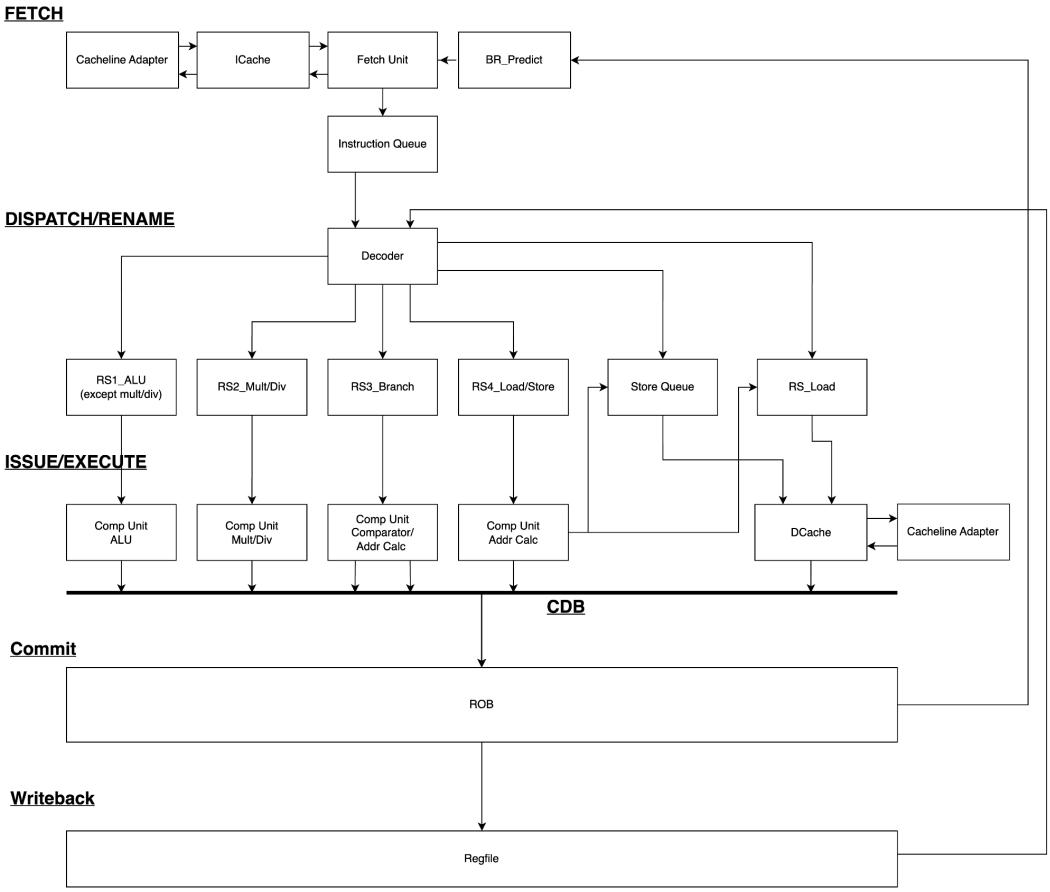

Fig 3: Updated Block Diagram

3. Design Description

a. Check Point 1:

The goal of the first checkpoint is to finish the fetch stage, including the PC_Fetch unit, Parameterized Instruction Queue with FIFO(First In First Out) structure, and cache line adapter to deserialize the burst out of the DRAM model.

Fetch Stage: Similar to mp_pipelilne, we simply reset the pc to x1ECEB000 and increment the pc with 4 each time.

Fig 4: Fetch result

Parameterized Instruction Queue: To first get the parameterized modules, we will use the structures shown below in Figure 5 – and we will use similar parameterizing structures in the later designs, such as Reservation Stations and ROB. To make the instruction queue FIFO, we will define two pointers: head and tail. The tail will be used as the entry point for the instructions and the head will be the leaving point for them. When a new instruction is sent from the fetch stage, the tail will increment by one and when the

instruction that is pointed by the head is about to pop, the head will increment by one. When the head pointer is at the same location as the tail pointer, we will say it is at full stage. In addition, this FIFO structure will be used again later in our ROB and store queue design. A simple demonstration of the FIFO structure is shown below in Figure 7.

```
module instruction_queue #(
    parameter IQ_DEPTH = 4,
    parameter IQ_WIDTH = 32
) (
```

Fig 5: Parameterized Instruction Queue

```
logic [(IQ_WIDTH - 1):0] instr_arr[IQ_DEPTH];
logic [(IQ_WIDTH - 1):0] pc_arr[IQ_DEPTH];
logic [(IQ_WIDTH - 1):0] pc_next_arr[IQ_DEPTH]
logic                    valid_arr[IQ_DEPTH];
```

Fig 6: Sample of using parameters

Fig 7: FIFO structure example

Cacheline Adapter: For the cache line adapter design, we want to achieve the following waveform for the read process shown in Figure 8. The main part here is to have a counter that starts with 0 and counts until 3 to make sure all 256 bits are obtained for the specific address and then read 32 bits instructions for eight cycles with respect to the specific PC values.

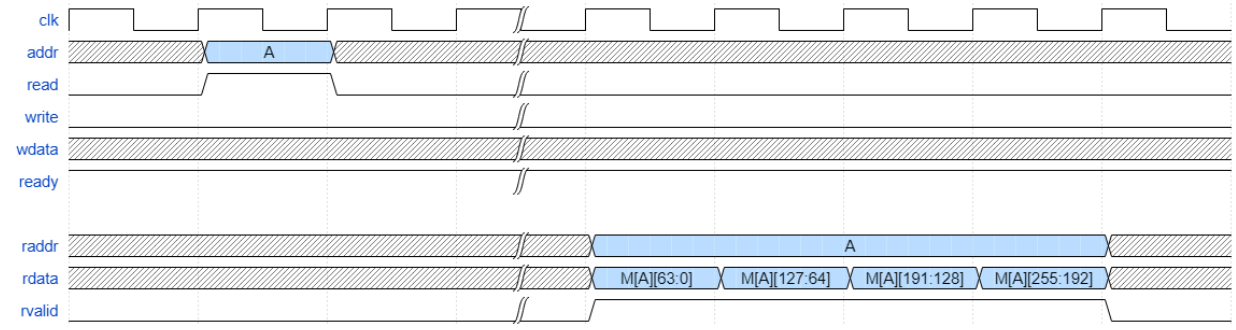

Fig 8: Read from DRAM

Fig 9: Result from cache line adapter

b. Check Point 2:

For checkpoint 2, the goal is to implement all instructions except the load/store and branch instructions and make sure they can run out of order but commit in order. In addition to that, we would integrate our design with the Synopsys IPs, which are used in the multiplication and division instructions.

Synopsys IPs:

We only use sequential multiplication and division IPs for this mp. The overall waveform and behaviors are shown below in Figure 10 and Figure 11. The reasons why we picked those two IPs are that both of them have a rst_mode, input_mode, and tc_mode for us to adjust the conditions. The rst_mode can be set to trigger by reset only once, so it will just behave like other always_ff blocks. The input_mode can be set to either have one cycle of input A and B or have multiple cycles until the computations are finished – we set it to one cycle since it will be easier for us to deal with the flush signals later. Finally, the tc_mode can be adjusted to fulfill the requirements, which are sign and unsign computations for multiplication and division. Moreover, the sequential IPs also offer us the choice to adjust the total clock cycles of computations, so we can adjust it quickly to balance the combinational logic with the delay.

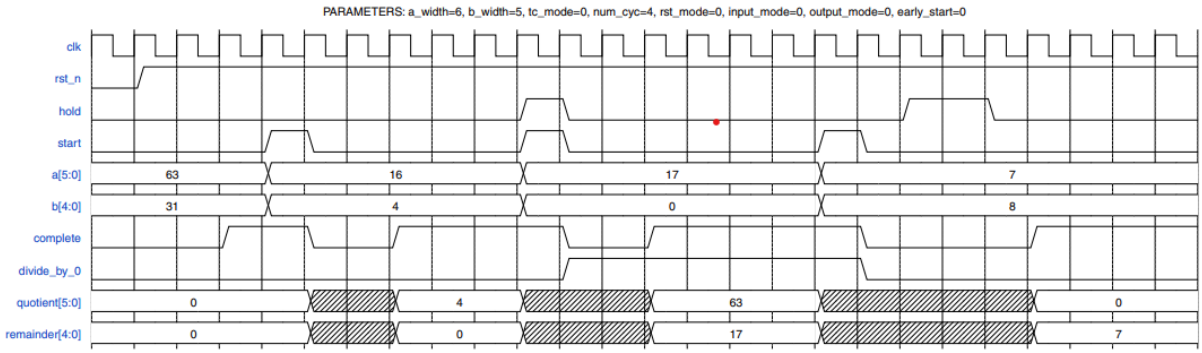

Figure 10: Sequential division waveform

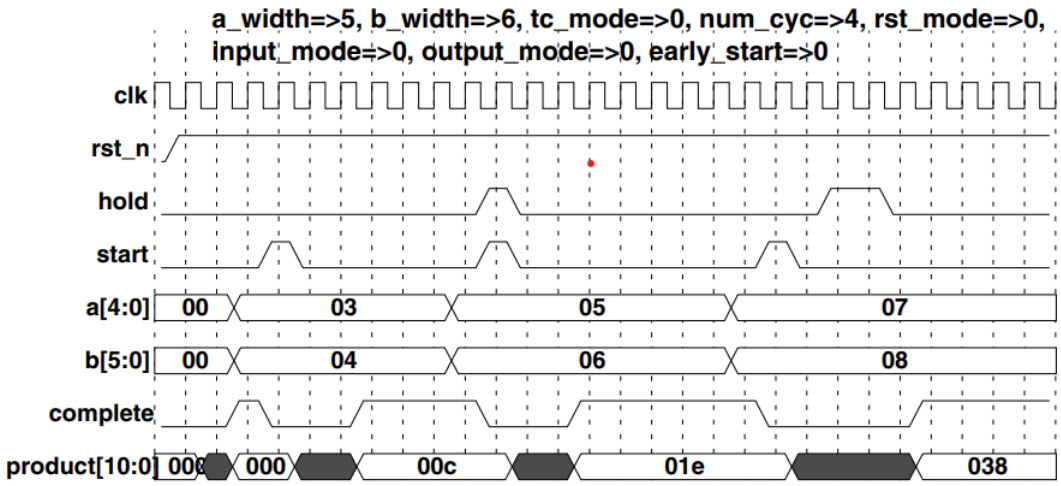

Figure 11: Sequential multiplication waveform

RV32IM Instructions implement and Out-of-Order behavior:

Since we have finished Checkpoint 3, it is not necessary to show the achievement of the instructions implementations excluding branch and load/store here. We will elaborate how to achieve the out-of-order execution while maintaining the in-order commit sequence. In our design, the only place that can do out-of-order computation is in RS. After the decode stage, ROB and RS will both receive the instructions and RS will receive the corresponding tag from ROB tail position to mark that as the destination. In addition to that, two tags and valid bits will be added to RS to show when should RS send the instruction to its connected computations blocks, such as ALU, Mult/Div, Load/Store, and Branch/Comparator. The Figure below shows how the instructions get ready in the ROB. Since it is a FIFO structure, the instructions are sent in order, while they finish out of order – by looking at the read_commit bit, we can see when will the instructions get ready.

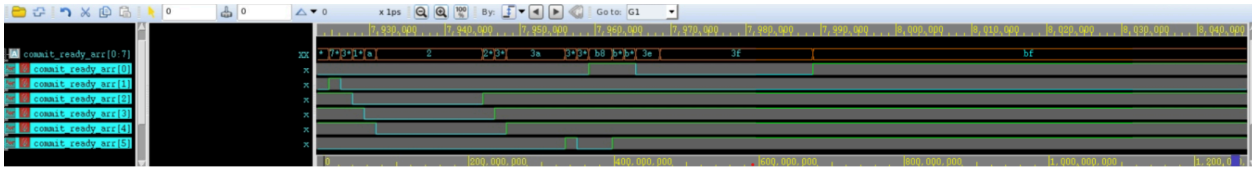

Figure 12: Out-of-order execution behavior

c. Check Point 3:

For the last checkpoint, we need to implement every instruction in RV32IM and pass coremark test case, which is the industry standard one. To be specific, we would implement store/load and branch functions for our design. For the branch part, our first idea is that whenever the branch is taken, we will flush everything including RS, ROB, and Instruction Queue, which can have a lot of penalties for unnecessary flushes. For the load/store part, since we have not fixed our mp_cache, we first sticks with the cache

provided by the course staff. One trick about the provided cache is that it does not have any way associativity and it is multi-cycles, which means that we should not only provide the address and masks until the read and write finish but also take the larger miss rates. Due to the fact that our original idea is to simply pass the checkpoint instead of aiming for the rank on the leaderboard, we finish the implementations without any optimizations and advanced features. The result of the first finished Out-of-Order processor is shown below on the left. Even though it passed all test cases locally, it failed some of the test cases on the AG due to the low IPC. After scrutinizing our design, we noticed it was due to the cache misses problem since the provided one misses every eight instruction fetches, not to mention the data load store operation. It turns out that the design is really bad with only approximately 0.1 IPC since there are a lot of improvements that could be done to that, including branch recovery, branch prediction, out-of-order load, and so on.

Tests	Result	IPC	Delay (µs)	Power (mW)	$PD^3A^{1/2}$
SRAM	☐				
compile	✅				
lint	✅				
synth	✅				
Area (μm^2)	`67500`				
f_{max} (MHz)	`500.00`				
coremark	❌				
aes_sha	❌				
cnn	✅	`0.1047`	`26384.51`	`13.906`	`66360.66`
compression	✅	`0.1360`	`6177.83`	`13.912`	`852.20`
fft	✅	`0.1066`	`9655.96`	`13.446`	`3145.01`
mergesort	❌				
raytracing	✅	`0.1560`	`9262.22`	`14.675`	`3029.42`
rsa	❌				

Fig 13: Initial design of Out-of-Order and pass the coremark testcase

d. Advanced Features:
Branch Predictor and BTB: We chose the Gshare branch predictor with the combination of BTB to make sure the implementation is simple and also decrease the aliasing of the same PC values by doing the XOR computation in the predictor. The result is shown in the below figure and the improvements are really obvious – the flush numbers decrease from 33000 to about 11000.

Fig 14: Total number of flushes vs branch taken

Out-of-Order load: In our original design, we only had one LSQ for both load and store instructions, which will lead to some unnecessary stalls for load that can be executed if they do not have any dependency on the store. Hence, we split the LSQ into two parts – a store queue same as before, and a load RS that only performs out-of-order load. When we add a load instruction into the load RS, it will first check if there are any previous stores for it to wait. If there are no stores, load can be performed whenever it is ready. The only edge case here is if the load enters while the store just finished and was about to commit. In this case, the load needs to wait until the commit finishes and then enter the RS. The result waveform is shown in the figure below. In addition to that, we used pipelined cache from mp_cache. After the modification, our final IPC for coremark improved to 0.3807, which significantly improved compared to the initial one.

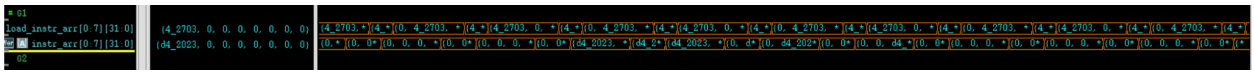

Fig 15: Out-of-Order Load waveform

Tests	Result	IPC	Delay (µs)	Power (mW)	$PD^3A^{½}$
SRAM	✅				
compile	✅				
lint	✅				
synth	✅				
Area (μm^2)	182619				
f_{max} (MHz)	636.94				
coremark	✅	0.3807	1202.33	45.339	33.68
aes_sha	✅	0.3130	3275.30	43.172	648.23
cnn	✅	0.2680	8090.67	43.955	9947.97
compression	✅	0.4997	1319.64	45.860	45.04
fft	✅	0.3684	2192.74	42.893	193.25
mergesort	✅	0.4716	1554.06	47.651	76.43
raytracing	✅	0.1834	6184.74	42.757	4322.62
rsa	✅	0.1977	19532.03	43.415	138246.42

Figure 16: Final results

4. Additional Observations

Some major improvements in our design will be reducing the load and store cycles under hit conditions to 1 and doing the branch recovery. As shown in the diagram below, our memory has three states for both load and store: IDLE, LOAD/STORE, and DONE. However, it is very inefficient since in the most optimistic case, we can achieve the 1-cycle load and store if we hit every tag in the cache. However, the trade-off for doing that is a very hard implementation for both the ROB and LSQ design. In addition to that, we need to also improve the write process in the LSQ to make it achieve 1 cycle per write – it is 0.5 write per cycle in mp_cache. Hence, if we can achieve that, we can ideally achieve approximately 1 IPC even with tons of loads/stores without a miss. The second major improvement for our design will be branch recovery. We can see that even with the branch prediction, there are still around ten thousand of the flushes

happening in the instructions. For now, we chose our ROB to be small – depth = 4 – to decrease the flush penalties.

Fig 3: LSQ FSM

5. Conclusion

To wrap up, from the Out-of-Order machine project, we have applied almost all the skills we built from the previous mp and the knowledge in the classes to achieve that. There are lots of details that could be added to our design, such as superscalar and branch recovery, and we hope to build better CPU in the future