Out of Order Execution RISC-IV Superscalar Pipelined Processor in SystemVerilog

Abbas Bakhshandeh

January 8, 2024

Abstract

Computer processors play a crucial role in the coordination of various components in computer systems, execution of instructions, and efficient completion of computing tasks. In an effort to improve processor performance, a number of microarchitectural optimizations such as OoO (Out of Order) execution, pipelining, and superscalar execution can be implemented. The goal for this capstone project is to create an OoO RISC-IV superscalar (2 issue) pipelined processor in SystemVerilog. The RISC-IV ISA was chosen due to its reduced complexity, open-source nature, and overall efficiency. In order to successfully enable OoO, architectural structures such as a reservation station, ROB (Reorder Buffer), free pool, and a physical register file were incorporated. Doing so improved processor throughput and efficiency by allowing execution of instructions out-of-order while preventing data hazards from occurring. Pipelining further increased throughput by allowing different modules to work on different instructions concurrently. Furthermore, the superscalar aspect of the processor allowed for two instructions to be processed in parallel throughout the pipeline, which increased throughput and efficiency.

1 Introduction

Computer processor design is a complex task that requires the combination of various microarchitectures in a manner that meets power, throughput, and resource utilization metrics. A simple processor can execute instructions one at a time, with separate modules each performing one aspect of the process. The general processing workflow involves fetching the instruction, decoding it, executing it, and reflecting the results in the architectural state or memory. In an effort to increase throughput, most processors use the concept of pipeling. Pipelining allows each module to process one part of a separate instruction, so that no modules - in most cases - are idle while they wait for other modules to complete. Another method used to increase throughput is superscalar execution. By making the processor superscalar, hardware is added in parallel to allow for the processing of multiple instructions at one stage in the pipeline. While these methods greatly improve throughput, by themselves they only allow processors to process instructions in the order that they are received. This can create issues where future instructions are waiting long periods of time for another instruction to finish before they can be processed, which results in a decrease in throughput.

In order to help alleviate this issue, out of order execution has been introduced to processors. Using out of order execution, the processor is able to have certain stages of the pipeline be completed out of the order that the instructions were received in. However, precautions must be taken to prevent data hazards that could occur as a result of this out of order processing of instructions. If a future instruction has a source register which depends on the result of a previous instruction, then it would be incorrect to allow it to complete before the results of that previous instruction are known and reflected in the architectural register file. The most common out of order processors complete the fetching of instructions, decoding of them, and retiring of them - where the results are reflected in architectural states and memory - in order. The stage that is allowed to complete out of order - with certain restrictions - is the execution stage.

In order to accomplish this out of order processing in the execution stage of the pipeline while preventing data hazards, various hardware blocks are created. The concept of physical registers is introduced, which are a larger set of registers that the processor places the architectural register values used in incoming instructions inside of. This larger set allows for many instructions that are referencing the same architectural register to have dedicated physical registers, allowing for out of order execution. A RAT (Register Alias Table), free pool, reservation station, ROB (Reorder buffer) are used to accomplish this. Table 2 provides a detailed description for each of these microarchitecture blocks. In their essence, they organize the incoming instructions - which have been decoded - in a manner that allows instructions to be executed out of order as long as their source registers don't depend on a previous instruction's result. They also keep track of which architectural registers have been mapped to which physical registers, so that they can be remapped in the correct order later in the pipeline. The result of the introduction of out of order execution to the processor architecture is an increase in throughput and overall efficiency.

2 Implementation

2.1 Module Descriptions

In order to implement this processor, the design was modularized and connected through a top module (see Appendix II) using the SystemVerilog HDL (Hardware Description Language). Interfaces were used to store information that could be accessed between various modules (see Appendix I). The modules that were created include Fetch, Decode, Rename, Dispatch, Fire, and Retire. Table 1 describes each module used in the design in detail.

2.2 Microarchitectures for Out of Order execution

Furthermore, within the modules there are various microarchitectural structures which were implemented in order to accomplish Out of Order (OoO) execution. Table 2 provides a more detailed description of each execution unit which was constructed.

3 Results

3.1 Functional Verification

Verification of digital processor designs is a complex task that must take into account a number of different possible cases to ensure correctness. Of the many test cases which were verified, two example test cases are shown. The first example involves instructions that do not require access to memory (ADDI, ANDI, ADD, SUB, XOR, SRA). The second example includes these instructions as well as LW and SW, both of which require access to memory in order to either load data or store data. In both cases, the input to the processor is a text file containing hexadecimal values, each corresponding to one byte of data (hence 4 creating a 32 bit instruction). Regarding the results, only the first 12 architectural registers are shown for brevity (rest are unused in these test cases).

Example 1. Instructions including ADDI, ANDI, ADD, SUB, XOR, SRA (Non memory):
These are the corresponding machine language instructions and calculations:

```
addi x2, x0, 6 -> x2 = 0 + 6 = 6 = 0110
addi x3, x0, 15 -> x2 = 0 + 15 = 15 = 1111
andi x5, x3, 1 -> x5 = 1111 & 0001 = 0001
addi x1, x0, 36 -> x1 = 0 + 36 = 36 = 100100
add x4, x5, x2 -> x4 = 0001 + 0110 = 0111
sub x0, x0, x0 -> x0 = 0 (by definition)
addi x9, x2, 23 -> x9 = x2 + 23 = 0110 + 10111 = 11101
xor x6, x2, x3 -> x6 = 0110 ^ 1111 = 1001
sub x2, x1, x6 -> x2 = 100100 - 1001 = 011011
sra x7, x3, x0 -> x7 = 1111 >> 0 = 1111
sra x3, x3, x5 -> x3 = 1111 >> 0001 = 0111
andi x9, x3, 23 -> x9 = 0111 & 10111 = 00110
```

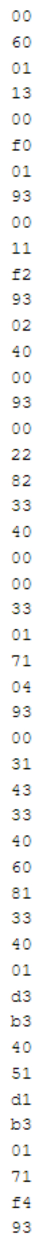
```
Final register file values:
x0: 00000000000000000000000000000000
```

Module	Description
Fetch	Every clock cycle, two input instructions are stored from a text file containing hexadecimal values. Each input instruction corresponds to 4 bytes (32 bits), so 8 bytes (64 bits) are fetched each clock cycle. (See Appendix III)
Decode	Extracts necessary information from the two instructions which are fetched every clock cycle, including the destination register, source registers, immediate values, function values, and control signals such as AluOp, aluSrc, memRead, memWrite, and memtoReg. These values are stored and passed to the next stages in the pipeline for further processing. The 8 possible instructions for this processor are ADDI, ANDI, ADD, SUB, XOR, SRA, LW, and SW. (See Appendix IV)
Rename	Register renaming is enabled in order to implement out-of-order execution. Architectural destination registers are assigned to available physical registers, and architectural source registers are mapped to their assigned physical register names. Doing this prevents data hazards that could arise when completing out-of-order execution. A RAT (Register Alias Table) in addition to a "free pool" structure is utilized to keep track of which architectural registers are mapped to which physical registers. A physical register file is also created to hold physical register 32 bit values. (See Appendix V)
Dispatch	The information collected in the previous stages - the decoded signals as well as the renamed physical registers - are placed inside a reservation station. The reservation station contains "ready" bits for each (physical) source register in each instruction, as well as a ROB number and a functional unit number. A ROB (Reorder buffer) is created that stores the new physical destination registers, old physical destination registers (before renaming), and a "complete" bit which indicates whether the instruction has been completed. When both source registers are ready (and a functional unit is available), the instruction can be issued to the functional unit and removed from the reservation station. Up to three instructions can be issued to the three functional units at one time, if they are available. Two of the functional units are for R-type and I-type instructions (ALU) and the third function unit is for memory instructions (LW and SW). (See Appendix VI)
Fire	Once the reservation station issues an instruction in the table to a functional unit, it is processed based on the instruction's control signals (which were calculated in the decode stage), the physical register names, and the values in the physical register file. The final results of these calculations are stored in the physical register file (at the correct physical destination register location). (See Appendix VII)
Retire	While the execution of instructions can be completed out-of-order, the retiring (which actually updates the architectural registers and memory) must be done in-order to prevent data hazards. To accomplish this, the ROB is checked to see which instructions have completed (after the fire stage). It will retire instructions which have had all instructions above it (indicated by a program counter) also retired. This will assign the values stored in the physical register file to their corresponding architectural registers or memory. Up to two instructions can be retired each clock cycle. After retiring, they are removed from the ROB and the corresponding physical registers are freed for further usage by future instructions. (See Appendix VIII)

Table 1: Processor Modules and Descriptions

Mircroarchitecture Block	Description
RAT (Register Alias Table)	The RAT is used to keep track of which architectural registers have been renamed to which physical registers. The RAT is created in the Rename module.
Free Pool	The free pool is an array containing 64 one bit values. Each one bit value indicates whether the corresponding physical register is occupied or not. For example, if (free pool)[7] == 1'b1 is true, then this means physical register 7 is occupied (and hence should not be assigned to any other architectural register). In this microarchitecture, p0 is therefore reserved and to not be occupied / maintain a value of 0. This is to aid further calculations within the pipeline as well as to maintain symmetry with the architectural registers in RISC-IV, of which x0 is reserved to always keep a value of 0. The free pool is created in the Rename module.
Reservation Station	The reservation station holds all the decoded information for incoming instructions as well as their corresponding renamed physical registers in a 2D table. This table also contains signal bits that indicate whether the (physical) source registers of an instruction are ready, which will allow - assuming a relevant functional unit is also ready - for the instruction to be executed. All relevant control signals and physical registers are stored in this table, which paired with the physical register file allows for the instructions to be executed. The reservation station is first created in the Dispatch module, and subsequently accessed in the Fire module.
ROB (Reorder Buffer)	The reorder buffer keeps track of what order the instructions are originally in, so that they can be retired in-order. While the processor efficiency benefits greatly from out-of-order execution, the ROB ensures that those instructions are still retired in order to prevent data hazards. The reorder buffer is created in the dispatch module and is subsequently accessed and modified in the retire module.

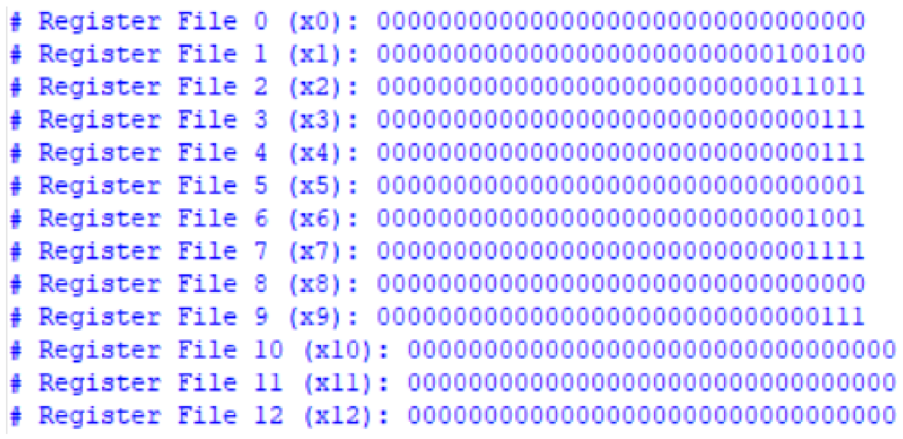
Table 2: Microarchitecture Blocks Necessary for Out of Order Execution



```
00
60
01
13
00
f0
01
93
00
11
f2
93
02
40
00
93
00
22
82
33
40
00
00
33
01
71
04
93
00
31
43
33
40
60
81
33
40
01
d3
b3
40
51
d1
b3
01
71
f4
93
```

Figure 1: Test Case 1 Non-memory Instructions (hexadecimal values)

```
x1: 00000000000000000000000000100100
x2: 00000000000000000000000000011011
x3: 00000000000000000000000000000111
x4: 00000000000000000000000000000111
x5: 00000000000000000000000000000001
x6: 00000000000000000000000000001001
x7: 00000000000000000000000000001111
x8: 00000000000000000000000000000000
x9: 00000000000000000000000000000111
x10: 00000000000000000000000000000000
x11: 00000000000000000000000000000000
x12: 00000000000000000000000000000000
```



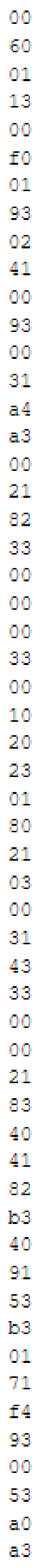
```
# Register File 0 (x0): 00000000000000000000000000000000
# Register File 1 (x1): 00000000000000000000000000100100
# Register File 2 (x2): 00000000000000000000000000011011
# Register File 3 (x3): 00000000000000000000000000000111
# Register File 4 (x4): 00000000000000000000000000000111
# Register File 5 (x5): 00000000000000000000000000000001
# Register File 6 (x6): 00000000000000000000000000001001
# Register File 7 (x7): 00000000000000000000000000001111
# Register File 8 (x8): 00000000000000000000000000000000
# Register File 9 (x9): 00000000000000000000000000000111
# Register File 10 (x10): 00000000000000000000000000000000
# Register File 11 (x11): 00000000000000000000000000000000
# Register File 12 (x12): 00000000000000000000000000000000
```

Figure 2: Final ModelSim architectural registers displaying correct values (Example 1)

Example 2. Instructions including LW, SW, ADDI, ANDI, ADD, SUB, XOR, SRA (Includes memory):

```
addi x2, x0, 6  -> x2 = 0 + 0110 = 0110
addi x3, x0, 15   -> x3 = 0 + 1111 = 1111
addi x1, x2, 36  -> x1 = 0110 + 36 = 101010
sw x3, x3, 9  -> 1111 + 1001 = 11000 (store 1111 in memory[11000])
add x4, x3, x2  -> x4 = 1111 + 0110 = 010101
add x0, x0, x0  -> x0 = 0 (by definition)
sw x1, x0, 0  -> 0 + 0 = 0 -> (store 101010 in memory[0])
lw x2, 24, x0  -> x2 = memory[0 + 11000] = memory[11000] = 1111
xor x6, x2, x3  -> x6 = 1111 ^ 1111 = 0
lw x3, xo, 0  -> x3 = memory[0 + 0] = 101010
sub x5, x3, x4  -> 101010 - 10101 = 010101
sra x7, x2, x9  -> x7 = 1111 >> 0 = 1111
andi x9, x3, 23  -> x9 = 101010 & 10111 = 000010
sw x5, x7, 1  -> 1111 + 1 = 10000 (store 10101 in memory[10000])

Final register file values:
x0: 00000000000000000000000000000000
x1: 00000000000000000000000000101010
x2: 00000000000000000000000000001111
x3: 00000000000000000000000000101010
x4: 00000000000000000000000000010101
x5: 00000000000000000000000000010101
x6: 00000000000000000000000000000000
x7: 00000000000000000000000000001111
x8: 00000000000000000000000000000000
x9: 00000000000000000000000000000010
```

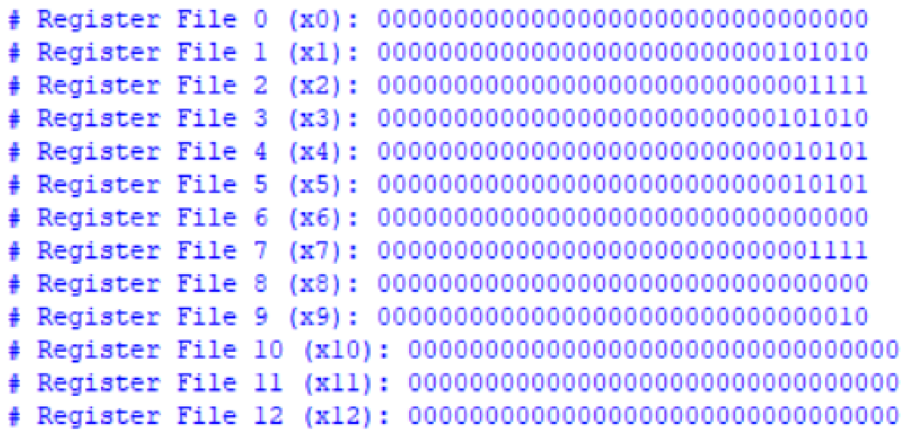


```
00
60
01
13
00
f0
01
93
02
41
00
93
00
31
a4
a3
00
21
82
33
00
00
00
33
00
10
20
23
01
80
21
03
00
31
43
33
00
00
21
83
40
41
82
b3
40
91
53
b3
01
71
f4
93
00
53
a0
a3
```

Figure 3: Test Case 2, All Instructions, including LW and SW (hexadecimal values)

```
x10: 00000000000000000000000000000000
x11: 00000000000000000000000000000000
x12: 00000000000000000000000000000000

memory[0]: 101010
memory[16]: 10101
memory[24]: 1111
```



```
# Register File 0 (x0): 00000000000000000000000000000000
# Register File 1 (x1): 00000000000000000000000000101010
# Register File 2 (x2): 00000000000000000000000000001111
# Register File 3 (x3): 00000000000000000000000000101010
# Register File 4 (x4): 00000000000000000000000000010101
# Register File 5 (x5): 00000000000000000000000000010101
# Register File 6 (x6): 00000000000000000000000000000000
# Register File 7 (x7): 00000000000000000000000000001111
# Register File 8 (x8): 00000000000000000000000000000000
# Register File 9 (x9): 00000000000000000000000000000010
# Register File 10 (x10): 00000000000000000000000000000000
# Register File 11 (x11): 00000000000000000000000000000000
# Register File 12 (x12): 00000000000000000000000000000000
```

Figure 4: Final ModelSim architectural registers displaying correct values (Example 2)

3.2 Area / Resource and Power Utilization

In addition to the verification of functionality, the area and power usage of the design must be taken into account. Since FPGA resources are fixed, the area usage is measured in terms of FPGA resource utilization. In Table 3, the area / resource utilization is shown. Table 4 displays the power usage for each module.

4 Discussion

From a functional standpoint, the different modules each work to ensure their part of the pipeline is completed correctly before the instruction reaches the next module. The presence of the ROB, reservation station, free pool, and RAT ensure that out of order execution is completed properly without data hazards. The OoO execution aspect of the processor greatly helps in efficiently executing instructions without having to wait unnecessarily for previous instructions to finish. While the instructions must still be retired in-order, having the execution finish based on which instructions are available greatly helps in improving throughput. Pairing this with pipelining and being 2-issue superscalar further increases the processor throughput and overall performance. Combining the different optimizations has a strong effect; for example, the superscalar aspect of the processor causes more instructions to be stored in the reservation station each clock cycle. This in turn presents more options to the OoO processor to complete instructions in whatever order is most efficient, rather than simply waiting for instructions to complete.

The design was simulated and verified using ModelSim. In Quartus, the design was synthesized to find area and power results. In processor design, PPA (Power, Performance, and Area) values are of interest to use as an evaluation metric. Regarding the area results, the number of logic elements is lowest in the Fetch module, which intuitively makes sense due to the simple nature of the task (simply assigning values from a text file to memory). The Fire module uses the most amount of logic elements, which is to be expected as it contains the logic where multiple instructions are taken from the reservation station and executed in each of the functional units. Most of the logic elements are allocated for the more complex modules later in the pipeline, such as fire, retire, and dispatch. There are a number of loops in each of these modules which require many logic element resources to

Fetch	Total Logic Elements: 1 Total Registers: 0 Total Modular Pins: 65 Total Combinational Functions: 1 Logic element usage by number of LUT inputs 4 input functions: 0 3 input functions: 0 ≤2 input functions: 1
Decode	Total Logic Elements: 183 Total Registers: 138 Total Modular Pins: 189 Total Combinational Functions: 101 Logic element usage by number of LUT inputs 4 input functions: 46 3 input functions: 29 ≤2 input functions: 26
Rename	Total Logic Elements: 423 Total Registers: 85 Total Modular Pins: 185 Total Combinational Functions: 412 Logic element usage by number of LUT inputs 4 input functions: 240 3 input functions: 150 ≤2 input functions: 23
Dispatch	Total Logic Elements: 1576 Total Registers: 1400 Total Modular Pins: 1922 Total Combinational Functions: 204 Logic element usage by number of LUT inputs 4 input functions: 179 3 input functions: 2 ≤2 input functions: 13
Fire	Total Logic Elements: 257472 Total Registers: 3212 Total Modular Pins: 4909 Total Combinational Functions: 257472 Logic element usage by number of LUT inputs: 4 input functions: 245651 3 input functions: 9029 ≤2 input functions: 2792
Retire	Total Logic Elements: 33098 Total Registers: 1060 Total Modular Pins: 36293 Total Combinational Functions: 33098 Logic element usage by number of LUT inputs: 4 input functions: 32789 3 input functions: 150 ≤2 input functions: 159

Table 3: Area / Resource Utilization of Modules on FPGA

Fetch	45.76 mW
Decode	46.20 mW
Rename	46.19 mW
Dispatch	46.12 mW
Fire	46.05 mW
Retire	45.96 mW

Table 4: Power Usage of Modules on FPGA

implement. The input pins to the Retire module seem high (36293), but the majority of them are unused memory bits. If desired, the memory available to the module can be lessened to reduce these modular pins. The default toggle rate used for input I/O signals for power calculations was set to 12.5%, which is the default in Quartus. The default toggle rate for the remaining signals - non I/O - was also set to 12.5%. In regards to the power usage of the design, the modules are similar in their estimated usage. The fetch module has the lowest estimated power usage which is to be expected due to its simple logic.

5 Conclusion

The design of processors can be complex with many tradeoffs needing to be taken into account. The processor built for this project successfully accomplishes OoO (out of order) execution, superscalar execution (2 issue), and pipelining. The processor achieves this by utilizing a reservation station, ROB (Re-order buffer), free pool, register renaming, forwarding, and a physical register file. In addition to functionality, processors should be designed in a manner that optimizes for throughput, efficiency, power, and area / resource utilization. Each module should be designed with minimal redundancy in logic and hardware in order to maximize the efficiency of the processor.

6 Acknowledgements

This project was primarily created with the knowledge given in Professor Nader Sehatbakhsh's UCLA ECE M116C (Computer Architecture) course. Here are links to the lecture videos from Fall 2023 which are most relevant to this project:

1. L10 - Superscalar and Intro to OoO
2. L11 - Out of Order Execution
3. L12 - Cache Design

References

[1] Nader Sehatbakhsh, *Superscalar and Intro to OoO*, https://www.youtube.com/watch?v=nrJxRNfmgbQ&ab_channel=NaderSehatbakhsh

[2] Nader Sehatbakhsh, *Out of Order Execution*, https://www.youtube.com/watch?v=db6UHxfw0C4&ab_channel=NaderSehatbakhsh

[3] Nader Sehatbakhsh, *Cache Design*, https://www.youtube.com/watch?v=RwdiDAaK6U8&ab_channel=NaderSehatbakhsh

Appendix I - interface.sv file

```

interface InstructionInterface;
  logic  [31:0]  firstinstruction;
  logic  [31:0]  secondinstruction;
endinterface


interface DecodeInterface;
  // Signals related to the first
  instruction logic [31:0] firstinstruction;
  logic [6:0] opcode1;
  logic [4:0] rs1_1;
  logic [4:0] rs2_1;
  logic [4:0] rd_1;
  logic regWrite1;
  logic [2:0] aluOp1;
  logic aluSrc1;
  logic memWrite1;
  logic memRead1;
  logic memtoReg1;
  logic signed [31:0] imm1;

  // Signals related to the second instruction
  logic [31:0] secondinstruction;
  logic [6:0] opcode2;
  logic [4:0] rs1_2;
  logic [4:0] rs2_2;
  logic [4:0] rd_2;
  logic regWrite2;
  logic [2:0] aluOp2;
  logic aluSrc2;
  logic memWrite2;
  logic memRead2;
  logic memtoReg2;
  logic signed [31:0] imm2;
endinterface


interface RegisterRenamingInterface;
  logic [4:0] src1_instr;
  logic [4:0] src2_instr;
  logic [4:0] dest_instr;
  logic [4:0] src1_instr2;
  logic [4:0] src2_instr2;
  logic [4:0] dest_instr2;

  logic [5:0] src1_phys;
  logic [5:0] src2_phys;
  logic [5:0] dest_phys;
  logic [5:0] src1_phys2;
  logic [5:0] src2_phys2;
  logic [5:0] dest_phys2;

  // Pass in the extra items from decode that
  // will be eventually needed by dispatch
  // (don't need the architectural registers since creating
  // the physical registers in this stage)

  // Signals related to the first instruction
  logic [6:0] renameopcode1;
  logic regWrite1;
  logic [2:0] aluOp1;
  logic aluSrc1;
  logic memWrite1;
  logic memRead1;
  logic memtoReg1;
  logic signed [31:0] imm1;

  // Signals related to the second instruction
  logic [6:0] renameopcode2;
  logic regWrite2;
  logic [2:0] aluOp2;
  logic aluSrc2;
  logic memWrite2;
  logic memRead2;
  logic memtoReg2;
  logic signed [31:0] imm2;

  logic ready_src1_instr;
  logic ready_src2_instr;
  logic ready_src1_instr2;
  logic ready_src2_instr2;
  logic [63:0] readyregs;

  logic [5:0] overwrittendest_phys1;
  logic [5:0] overwrittendest_phys2;

  logic [4:0] destihold;
  logic [4:0] destihold2;

  logic [5:0] free_regs [0:1];


endinterface

interface DispatchingInterface;
  logic [5:0] src1_phys;
  logic [5:0] src2_phys;
  logic [5:0] dest_phys;
  logic [5:0] src1_phys2;
  logic [5:0] src2_phys2;
  logic [5:0] dest_phys2;

  // Info that is coming in from decode->rename->dispatch

  // Signals related to the first instruction
  logic [6:0] dispatchopcode1;
  logic regWrite1;
  logic [2:0] aluOp1;
```

```systemverilog
  logic aluSrc1;
  logic memWrite1;
  logic memRead1;
  logic memtoReg1;
  logic signed [31:0] imm1;

  // Signals related to the second instruction
  logic [6:0] opcode2;
  logic regWrite2;
  logic [2:0] aluOp2;
  logic aluSrc2;
  logic memWrite2;
  logic memRead2;
  logic memtoReg2;
  logic signed [31:0] imm2;
  logic [84:0] rs_array[0:15];
  logic [18:0] rob[0:15];

  logic ready_src1_instr;
  logic ready_src2_instr;
  logic ready_src1_instr2;
  logic ready_src2_instr2;
  logic [63:0] readyregs;

  logic [5:0] overwrittendest_phys1;
  logic [5:0] overwrittendest_phys2;

  logic [4:0] destihold;
  logic [4:0] destihold2;

  logic [15:0] complete_array;
  logic [3:0] retire_rob [0:1];
  logic [3:0] clear_rs [0:2];

endinterface

interface FireInterface;
  logic [84:0] rs_array[0:15];
  logic [18:0] rob[0:15];
  logic [31:0] physregisters [63:0];
  logic [15:0] complete_array;
  logic [31:0] store_address;
  logic [31:0] store_value;
  logic [31:0] load_address;
  logic [15:0] store_complete_array;
  logic [31:0] memory [0:1023];
  logic [3:0] clear_rs [0:2];
endinterface


interface RetireInterface;
  logic [18:0] rob[0:15];
  logic [31:0] physregisters [63:0];
  logic [31:0] regfile [0:31];
  logic [15:0] complete_array;
  logic [31:0] store_address;
  logic [31:0] store_value;
  logic [31:0] load_address;
  logic [15:0] store_complete_array;
  logic [31:0] memory [0:1023];
  logic [3:0] retire_rob [0:1];
  logic [5:0] free_regs [0:1];
endinterface


```

Appendix II – TopModule.sv file

```

module TopModule;

  reg clk;

  reg reset;

  InstructionInterface myInterface();

  DecodeInterface decodeInterface();

  RegisterRenamingInterface renamingInterface();

  DispatchingInterface dispatchinterface();

  FireInterface fireinterface();

  RetireInterface retireinterface();

  // Module instantiation of Fetch
  module Fetch2 myModule (
    .clk(clk),
    .firstinstruction(myInterface.firstinstruction),
    .secondinstruction(myInterface.secondinstruction)
  );

// Module instantiation of Decode module
 Decode2 decodeModule (
  .clk(clk),
  .firstinstruction(decodeInterface.firstinstruction),
  .secondinstruction(decodeInterface.secondinstruction),
  .opcode1(decodeInterface.opcode1),
  .rs1_1(decodeInterface.rs1_1),
  .rs2_1(decodeInterface.rs2_1),
  .rd_1(decodeInterface.rd_1),
  .regWrite1(decodeInterface.regWrite1),
  .aluOp1(decodeInterface.aluOp1),
  .aluSrc1(decodeInterface.aluSrc1),
  .memWrite1(decodeInterface.memWrite1),
  .memRead1(decodeInterface.memRead1),
  .memtoReg1(decodeInterface.memtoReg1),
  .imm1(decodeInterface.imm1),
  .opcode2(decodeInterface.opcode2),
  .rs1_2(decodeInterface.rs1_2),
  .rs2_2(decodeInterface.rs2_2),
  .rd_2(decodeInterface.rd_2),
  .regWrite2(decodeInterface.regWrite2),
  .aluOp2(decodeInterface.aluOp2),
  .aluSrc2(decodeInterface.aluSrc2),
  .memWrite2(decodeInterface.memWrite2),
  .memRead2(decodeInterface.memRead2),
  .memtoReg2(decodeInterface.memtoReg2),
  .imm2(decodeInterface.imm2)
```

```
);


// Module instantiation of Rename module
  RegisterRenaming renamingModule (
    .clk(clk),
    .src1_instr(renamingInterface.src1_instr),
    .src2_instr(renamingInterface.src2_instr),
    .dest_instr(renamingInterface.dest_instr),
    .src1_instr2(renamingInterface.src1_instr2),
    .src2_instr2(renamingInterface.src2_instr2),
    .dest_instr2(renamingInterface.dest_instr2),
    .rename_enable(1'b1),
    .src1_phys(renamingInterface.src1_phys),
    .src2_phys(renamingInterface.src2_phys),
    .dest_phys(renamingInterface.dest_phys),
    .src1_phys2(renamingInterface.src1_phys2),
    .src2_phys2(renamingInterface.src2_phys2),
    .dest_phys2(renamingInterface.dest_phys2),
    .reset(reset),

  //  .outrenameopcode1(predispatchinterface.outrenameopcode1),
  //  .outrenameopcode2(predispatchinterface.outrenameopcode2),

    .renameopcode1(renamingInterface.renameopcode1),
    .renameopcode2(renamingInterface.renameopcode2),

    .ready_src1_instr(renamingInterface.ready_src1_instr),
    .ready_src2_instr(renamingInterface.ready_src2_instr),
    .ready_src1_instr2(renamingInterface.ready_src1_instr2),
    .ready_src2_instr2(renamingInterface.ready_src2_instr2),
    .readyregs(renamingInterface.readyregs),

    .overwrittendest_phys1(renamingInterface.overwrittendest_phys1),
    .overwrittendest_phys2(renamingInterface.overwrittendest_phys2),

    .destihold(renamingInterface.destihold),
    .destihold2(renamingInterface.destihold2),
    .free_regs(renamingInterface.free_regs)
  );

// Module instantiation of Dispatch module
  Dispatch2 dispatchModule (
    .clk(clk),
    .src1_phys(dispatchinterface.src1_phys),
    .src2_phys(dispatchinterface.src2_phys),
    .dest_phys(dispatchinterface.dest_phys),
    .src1_phys2(dispatchinterface.src1_phys2),
    .src2_phys2(dispatchinterface.src2_phys2),
    .dest_phys2(dispatchinterface.dest_phys2),

   // Signals that came from decode->rename->dispatch
  .dispatchopcode1(dispatchinterface.dispatchopcode1),
  .regWrite1(dispatchinterface.regWrite1),
  .aluOp1(dispatchinterface.aluOp1),
  .aluSrc1(dispatchinterface.aluSrc1),
  .memWrite1(dispatchinterface.memWrite1),
  .memRead1(dispatchinterface.memRead1),
  .memtoReg1(dispatchinterface.memtoReg1),
  .imm1(dispatchinterface.imm1),
  .decode_enable(1'b1),
  .opcode2(dispatchinterface.opcode2),
  .regWrite2(dispatchinterface.regWrite2),
  .aluOp2(dispatchinterface.aluOp2),
  .aluSrc2(dispatchinterface.aluSrc2),
  .memWrite2(dispatchinterface.memWrite2),
  .memRead2(dispatchinterface.memRead2),
  .memtoReg2(dispatchinterface.memtoReg2),
  .imm2(dispatchinterface.imm2),
  .rs_array(dispatchinterface.rs_array),
  .rob(dispatchinterface.rob),

  .ready_src1_instr(dispatchinterface.ready_src1_instr),
  .ready_src2_instr(dispatchinterface.ready_src2_instr),
  .ready_src1_instr2(dispatchinterface.ready_src1_instr2),
  .ready_src2_instr2(dispatchinterface.ready_src2_instr2),
  .readyregs(dispatchinterface.readyregs),

  .overwrittendest_phys1(dispatchinterface.overwrittendest_phys1),
  .overwrittendest_phys2(dispatchinterface.overwrittendest_phys2),

  .destihold(dispatchinterface.destihold),
  .destihold2(dispatchinterface.destihold2),
  .complete_array(dispatchinterface.complete_array),
  .retire_rob(dispatchinterface.retire_rob),

  .clear_rs(dispatchinterface.clear_rs)
  );

// Module instantiation of Fire module
Fire2 fireModule (
  .clk(clk),
  .rs_array(fireinterface.rs_array),
  .rob(fireinterface.rob),
  .physregisters(fireinterface.physregisters),
  .complete_array(fireinterface.complete_array),
  .store_address(fireinterface.store_address),
  .store_value(fireinterface.store_value),
  .load_address(fireinterface.load_address),
  .store_complete_array(fireinterface.store_complete_array),
  .memory(fireinterface.memory),
  .clear_rs(fireinterface.clear_rs)
  );

// Module instantiation of Retire module
Retire2 retireModule (
  .clk(clk),
  .rob(retireinterface.rob),
  .physregisters(retireinterface.physregisters),
  .complete_array(retireinterface.complete_array),
  .store_address(retireinterface.store_address),
  .store_value(retireinterface.store_value),
  .load_address(retireinterface.load_address),
  .store_complete_array(retireinterface.store_complete_array),
  .memory(retireinterface.memory),
  .retire_rob(retireinterface.retire_rob),
  .free_regs(retireinterface.free_regs)
  );


  // Connect the fetch output to the decode input
 always_comb begin
    decodeInterface.firstinstruction  = myInterface.firstinstruction;
    decodeInterface.secondinstruction = myInterface.secondinstruction;
  end;


// Connect the renaming output to the decode input
always_comb begin
  renamingInterface.src1_instr = decodeInterface.rs1_1;
  renamingInterface.src2_instr = decodeInterface.rs2_1;
  renamingInterface.dest_instr = decodeInterface.rd_1;

  renamingInterface.src1_instr2 = decodeInterface.rs1_2;
  renamingInterface.src2_instr2 = decodeInterface.rs2_2;
  renamingInterface.dest_instr2 = decodeInterface.rd_2;

  // Pass through the items from Decode that
  // will eventually be used by dispatch
  renamingInterface.renameopcode1 = decodeInterface.opcode1;
  renamingInterface.regWrite1 = decodeInterface.regWrite1;
  renamingInterface.aluOp1 = decodeInterface.aluOp1;
  renamingInterface.aluSrc1 = decodeInterface.aluSrc1;
  renamingInterface.memWrite1 = decodeInterface.memWrite1;
  renamingInterface.memRead1 = decodeInterface.memRead1;
  renamingInterface.memtoReg1 = decodeInterface.memtoReg1;
  renamingInterface.imm1 = decodeInterface.imm1;

  renamingInterface.renameopcode2 = decodeInterface.opcode2;
  renamingInterface.regWrite2 = decodeInterface.regWrite2;
  renamingInterface.aluOp2 = decodeInterface.aluOp2;
  renamingInterface.aluSrc2 = decodeInterface.aluSrc2;
  renamingInterface.memWrite2 = decodeInterface.memWrite2;
  renamingInterface.memRead2 = decodeInterface.memRead2;
  renamingInterface.memtoReg2 = decodeInterface.memtoReg2;
  renamingInterface.imm2 = decodeInterface.imm2;

  renamingInterface.free_regs = retireinterface.free_regs;
end;


always_comb begin
  dispatchinterface.src1_phys  = renamingInterface.src1_phys;
  dispatchinterface.src2_phys = renamingInterface.src2_phys;
  dispatchinterface.dest_phys  = renamingInterface.dest_phys;
  dispatchinterface.src1_phys2 = renamingInterface.src1_phys2;
  dispatchinterface.src2_phys2  = renamingInterface.src2_phys2;
  dispatchinterface.dest_phys2 = renamingInterface.dest_phys2;

  dispatchinterface.dispatchopcode1 = renamingInterface.renameopcode1;
  dispatchinterface.regWrite1 = renamingInterface.regWrite1;
  dispatchinterface.aluOp1 = renamingInterface.aluOp1;
  dispatchinterface.aluSrc1 = renamingInterface.aluSrc1;
  dispatchinterface.memWrite1 = renamingInterface.memWrite1;
  dispatchinterface.memRead1 = renamingInterface.memRead1;
  dispatchinterface.memtoReg1 = renamingInterface.memtoReg1;
  dispatchinterface.imm1 = renamingInterface.imm1;

  dispatchinterface.opcode2 = renamingInterface.renameopcode2;
  dispatchinterface.regWrite2 = renamingInterface.regWrite2;
  dispatchinterface.aluOp2 = renamingInterface.aluOp2;
  dispatchinterface.aluSrc2 = renamingInterface.aluSrc2;
  dispatchinterface.memWrite2 = renamingInterface.memWrite2;
  dispatchinterface.memRead2 = renamingInterface.memRead2;
  dispatchinterface.memtoReg2 = renamingInterface.memtoReg2;
  dispatchinterface.imm2 = renamingInterface.imm2;

  dispatchinterface.ready_src1_instr = renamingInterface.ready_src1_instr;
  dispatchinterface.ready_src2_instr = renamingInterface.ready_src2_instr;
  dispatchinterface.ready_src1_instr2 = renamingInterface.ready_src1_instr2;
  dispatchinterface.ready_src2_instr2 = renamingInterface.ready_src2_instr2;
  dispatchinterface.readyregs = renamingInterface.readyregs;


  dispatchinterface.overwrittendest_phys1 = renamingInterface.overwrittendest_phys1;
  dispatchinterface.overwrittendest_phys2 = renamingInterface.overwrittendest_phys2;

  dispatchinterface.destihold = renamingInterface.destihold;
  dispatchinterface.destihold2 = renamingInterface.destihold2;

  dispatchinterface.retire_rob = retireinterface.retire_rob;

  dispatchinterface.clear_rs = fireinterface.clear_rs;
end;


always_comb begin
  fireinterface.rs_array = dispatchinterface.rs_array;
  fireinterface.rob = dispatchinterface.rob;
  fireinterface.complete_array = dispatchinterface.complete_array;
end;


always_comb begin
  retireinterface.rob = fireinterface.rob;
  retireinterface.physregisters = fireinterface.physregisters;
  retireinterface.complete_array = fireinterface.complete_array;
  retireinterface.store_address = fireinterface.store_address;
  retireinterface.store_value = fireinterface.store_value;
  retireinterface.load_address = fireinterface.load_address;
  retireinterface.store_complete_array = fireinterface.store_complete_array;
  fireinterface.memory = retireinterface.memory;
end;

```

```
// Clock signal
  always
  begin
    #5 clk = 1;
    #5 clk = 0;
  end

// Display for clock signal
/*
initial begin
  $display("Time %0t: clk = %b", $time, clk);
end
*/


  // Reset logic
  initial begin
    // Initialize reset at the beginning of simulation
    reset = 1'b1;
    #10 reset = 1'b0; // Unset reset after 10 time units
  end


// Display for positive clock edge
/*
 always_ff @(posedge clk) begin
    // Rotate the data_from_file array to get the next 4 values on each clock cycle
$display("Clock edge detected at time %0t", $time);

end
*/

endmodule
```

Appendix III – Fetch2.sv file

```

module Fetch2 (
  input logic clk,
  output reg [31:0] firstinstruction,
  output reg [31:0] secondinstruction
);

  reg [63:0] data_from_file = 32'h0;
  reg [7:0] mem [0:255];
  reg clk1;

  // Logic to read values from text file (can change filepath if needed)
  initial begin
    $readmemh("C:/Users/abbas/OneDrive/Documents/quartusfiles/try1.txt", mem);

    // Display the contents of the memory after reading from the file
    $display("Memory contents after reading from file:");
    for (int i = 0; i <= 30; i = i + 1) begin
      $display("mem[%0d] = %h", i, mem[i]);
    end

  end

  int i = -8;

  always @(posedge clk) begin
    // Rotate the data_from_file array to get the next 4 values on each clock
    cycle $display("Clock edge detected at time %0t", $time);

    data_from_file <= {mem[i+7], mem[i+6],mem[i+5],mem[i+4],mem[i+3], mem[i+2],mem[i+1],mem[i]
};

    i <= i + 8;

    firstinstruction <= {mem[i], mem[i+1],mem[i+2],mem[i+3]};
    secondinstruction <= {mem[i+4], mem[i+5],mem[i+6],mem[i+7]};
   ////$display("data_from_file = %b",  data_from_file);
   ////$display("memory =%h,%h,%h,%h,%h,%h,%h,%h,", mem[i+7], mem[i+6],mem[i+5],mem[i+4],
mem[i+3], mem[i+2],mem[i+1],mem[i]);
   ////$display("firstinstruction = %b, secondinstruction = %b", firstinstruction,
secondinstruction);
   ////$display("i = %d",  i);
   $display("firstinstruction = %b, secondinstruction = %b", firstinstruction,
secondinstruction);
  end

endmodule


```

Appendix IV - Decode2.sv file

```

module Decode2 (
  input logic clk,
  input logic [31:0] firstinstruction,
  input logic [31:0] secondinstruction,
  output logic [6:0] opcode1,
  output logic [6:0] opcode2,
  output logic [4:0] rs1_1,
  output logic [4:0] rs1_2,
  output logic [4:0] rs2_1,
  output logic [4:0] rs2_2,
  output logic [4:0] rd_1,
  output logic [4:0] rd_2,
  output logic regWrite1,
  output logic regWrite2,
  output logic [2:0] aluOp1,
  output logic [2:0] aluOp2,
  output logic aluSrc1,
  output logic aluSrc2,
  output logic memWrite1,
  output logic memWrite2,
  output logic memRead1,
  output logic memRead2,
  output logic memtoReg1,
  output logic memtoReg2,
  output logic signed [31:0] imm1,
  output logic signed [31:0] imm2
);

  static logic [2:0] funct3_1, funct3_2;
  static logic [6:0] funct7_1, funct7_2;

logic [6:0] firstopcode1;
logic [6:0] firstopcode2;
logic [2:0] firstfunct3_1;
logic [2:0] firstfunct3_2;
logic [6:0] firstfunct7_1;
logic [6:0] firstfunct7_2;

always_ff @(posedge clk) begin
    firstopcode1 <= firstinstruction[6:0];
    firstopcode2 <= secondinstruction[6:0];
    firstfunct3_1 <= firstinstruction[14:12];
    firstfunct3_2 <= secondinstruction[14:12];
    firstfunct7_1 <= firstinstruction[31:25];
    firstfunct7_2 <= secondinstruction[31:25];
end

always_ff @(posedge clk) begin
    //$display("firstinstruction = %b, secondinstruction = %b",
firstinstruction, secondinstruction);
    //$display("opcode1 = %b, opcode2 = %b",opcode1, opcode2);

    opcode1 = firstopcode1;
    opcode2 = firstopcode2;
    funct3_1 = firstfunct3_1;
    funct3_2 = firstfunct3_2;
    funct7_1 =firstfunct7_1;
    funct7_2 = firstfunct7_2;

    // Decode logic for the first instruction
    case (firstinstruction[6:0])
      7'b0010011: begin // ADDI and ANDI
        rs1_1 <= firstinstruction[19:15];
    rs2_1 <= 0;
        rd_1 <= firstinstruction[11:7];
        imm1 <= {{20{firstinstruction[31]}}, firstinstruction[31:20]};
        regWrite1 <= 1;
    //$display("funct3_1 = %b",funct3_1);

if (firstinstruction[14:12] == 3'b000) begin
aluOp1 <= 3'b000; // ADDI
aluSrc1 <= 0;
end

if (firstinstruction[14:12] == 3'b111) begin
aluOp1 <= 3'b111; // ANDI
aluSrc1 <= 0;
end

        memWrite1 <= 0;
        memRead1 <= 0;
        memtoReg1 <= 0;
      end

      7'b0110011: begin // ADD, SUB, AND, XOR, SRA
        rs1_1 <= firstinstruction[19:15];
        rs2_1 <= firstinstruction[24:20];
        rd_1 <= firstinstruction[11:7];
        regWrite1 <= 1;
        imm1 <= 0;

if (firstinstruction[14:12] == 3'b000) begin
if (firstinstruction[31:25] == 7'b0000000) begin
aluOp1 <= 3'b000; // ADD
aluSrc1 <= 0;
end
if (firstinstruction[31:25] == 7'b0100000) begin
aluOp1 <= 3'b001; // SUB
aluSrc1 <= 0;
end
end

if (firstinstruction[14:12] == 3'b100) begin
  aluOp1 <= 3'b100; // XOR
  aluSrc1 <= 0;
end

if (firstinstruction[14:12] == 3'b101) begin
```

```
  aluOp1 <= 3'b101; // SRA
            aluSrc1 <= 0;
end
        memWrite1 <= 0;
        memRead1 <= 0;
        memtoReg1 <= 0;
      end

      7'b0100011: begin // SW
        rs1_1 <= firstinstruction[19:15];
        rs2_1 <= firstinstruction[24:20];
    rd_1 <= 0;
        imm2 <= {secondinstruction[31:25], secondinstruction[11:7]};
        regWrite1 <= 0;
        aluOp1 <= 3'b000;
        aluSrc1 <= 1;
        memWrite1 <= 1;
        memRead1 <= 0;
        memtoReg1 <= 0;
      end

    7'b0000011: begin // LW
        rs1_1 <= firstinstruction[19:15];
    rs2_1 <= 0;
        rd_1 <= firstinstruction[11:7];
        imm1 <= {{20{firstinstruction[31]}}, firstinstruction[31:20]};
        regWrite1 <= 1;
        aluOp1 <= 3'b000;
        aluSrc1 <= 1;
        memWrite1 <= 0;
        memRead1 <= 1;
        memtoReg1 <= 1;
      end

      default: begin
        rs1_1 <= 5'bxxxxx;
        rs2_1 <= 5'bxxxxx;
        rd_1 <= 5'bxxxxx;
        regWrite1 <= 1'bx;
        aluOp1 <= 3'bxxx;
        aluSrc1 <= 1'bx;
        memWrite1 <= 1'bx;
        memRead1 <= 1'bx;
        memtoReg1 <= 1'bx;
    imm1 <= 1'bx;
        // $display("Debug: Unknown opcode detected for instruction 1");
      end
    endcase


 // Decode logic for the second instruction
    case (secondinstruction[6:0])
      7'b0010011: begin // ADDI and ANDI
        rs1_2 <= secondinstruction[19:15];
    rs2_2 <= 0;
        rd_2 <= secondinstruction[11:7];
```

```
            imm2 <= {{20{secondinstruction[31]}}, secondinstruction[31:20]};
            regWrite2 <= 1;

        if (secondinstruction[14:12] == 3'b000) begin
        aluOp2 <= 3'b000; // ADDI
        aluSrc2 <= 0;
        end

        if (secondinstruction[14:12] == 3'b111) begin
        aluOp2 <= 3'b111; // ANDI
        aluSrc2 <= 0;
        end

            memWrite2 <= 0;
            memRead2 <= 0;
            memtoReg2 <= 0;
          end

          7'b0110011: begin // ADD, SUB, XOR, SRA
            rs1_2 <= secondinstruction[19:15];
            rs2_2 <= secondinstruction[24:20];
            rd_2 <= secondinstruction[11:7];
            regWrite2 <= 1;
            imm2 <= 0;

    if (secondinstruction[14:12] == 3'b000) begin
    if (secondinstruction[31:25] == 7'b0000000) begin
    aluOp2 <= 3'b000; // ADD
    aluSrc2 <= 0;
    end
    if (secondinstruction[31:25] == 7'b0100000) begin
    aluOp2 <= 3'b001; // SUB
    aluSrc2 <= 0;
    end
    end

    if (secondinstruction[14:12] == 3'b100) begin
      aluOp2 <= 3'b100; // XOR
      aluSrc1 <= 0;
    end

    if (secondinstruction[14:12] == 3'b101) begin
      aluOp2 <= 3'b101; // SRA
      aluSrc2 <= 0;
    end

            memWrite2 <= 0;
            memRead2 <= 0;
            memtoReg2 <= 0;
          end

          7'b0100011: begin // SW
            rs1_2 <= secondinstruction[19:15];
            rs2_2 <= secondinstruction[24:20];
        rd_2 <= 0;
            imm2 <= {secondinstruction[31:25], secondinstruction[11:7]};
            regWrite2 <= 0;
            aluOp2 <= 3'b000;
            aluSrc2 <= 1;
            memWrite2 <= 1;
            memRead2 <= 0;
            memtoReg2 <= 0;
          end


        7'b0000011: begin // LW
            rs1_2 <= secondinstruction[19:15];
        rs2_2 <= 0;
            rd_2 <= secondinstruction[11:7];
            imm2 <= {{20{secondinstruction[31]}}, secondinstruction[31:20]};
            regWrite2 <= 1;
            aluOp2 <= 3'b000;
            aluSrc2 <= 1;
            memWrite2 <= 0;
            memRead2 <= 1;
            memtoReg2 <= 1;
          end

          default: begin
            rs1_2 <= 5'bxxxxx;
            rs2_2 <= 5'bxxxxx;
            rd_2 <= 5'bxxxxx;
            regWrite2 <= 1'bx;
            aluOp2 <= 3'bxxx;
            aluSrc2 <= 1'bx;
            memWrite2 <= 1'bx;
            memRead2 <= 1'bx;
            memtoReg2 <= 1'bx;
        imm2 <= 1'bx;
            //$display("Debug: Unknown opcode detected for instruction 2");
          end
        endcase

        // Additional debug information if needed

        $display("Debug: rd_1=%0d, rs1_1=%0d, rs2_1=%0d, imm_1=%d", rd_1, rs1_1, rs2_1, imm1);
        $display("Debug: regWrite_1=%0b, aluOp_1=%0b, aluSrc_1=%0b", regWrite1, aluOp1, aluSrc1);
        $display("Debug: memWrite_1=%0b, memRead_1=%0b, memtoReg_1=%0b", memWrite1,
    memRead1, memtoReg1);

        $display("Debug: rd_2=%0d, rs1_2=%0d, rs2_2=%0d, imm_2=%d", rd_2, rs1_2, rs2_2, imm2);
        $display("Debug: regWrite_2=%0b, aluOp_2=%0b, aluSrc_2=%0b", regWrite2, aluOp2, aluSrc2);
        $display("Debug: memWrite_2=%0b, memRead_2=%0b, memtoReg_2=%0b", memWrite2,
    memRead2, memtoReg2);

    end
    endmodule
```

Appendix V – Rename2.sv file

```

module RegisterRenaming (
  input wire [4:0] src1_instr,    // Source register 1 from instruction 1
  input wire [4:0] src2_instr,    // Source register 2 from instruction 1
  input wire [4:0] dest_instr,    // Destination register from instruction 1
  input wire [4:0] src1_instr2,   // Source register 1 from instruction 2
  input wire [4:0] src2_instr2,   // Source register 2 from instruction 2
  input wire [4:0] dest_instr2,   // Destination register from instruction 2
  input wire rename_enable,       // Enable register renaming
input wire clk,                   // Clock signal
input wire reset,                 // Reset signal
output logic [5:0] src1_phys,     // Physical register for source 1 in instruction 1
output logic [5:0] src2_phys,     // Physical register for source 2 in instruction 1
output logic [5:0] dest_phys,     // Physical register for destination in instruction 1
output logic [5:0] src1_phys2,    // Physical register for source 1 in instruction 2
output logic [5:0] src2_phys2,    // Physical register for source 2 in instruction 2
output logic [5:0] dest_phys2,     // Physical register for destination in instruction 2

  output logic [4:0] destihold,
  output logic [4:0] destihold2,
  input wire [6:0] renameopcode1,
  input wire [6:0] renameopcode2,
  input logic [5:0] free_regs [0:1],
  output logic ready_src1_instr,
  output logic ready_src2_instr,
  output logic ready_src1_instr2,
  output logic ready_src2_instr2,
  output logic [63:0] readyregs = '1,
  output logic [5:0] overwrittendest_phys1,
  output logic [5:0] overwrittendest_phys2

);

    logic [6:0] intopcode1;
    logic [6:0] intopcode2;

    parameter NUM_PHYSICAL_REGISTERS = 64;

    logic [5:0] rat [NUM_PHYSICAL_REGISTERS-1:0]; // Register Alias Table
    logic [NUM_PHYSICAL_REGISTERS-1:0] free_pool; // Free pool (for physical registers)

    // Function that finds the position of the first '0' in a free pool,
    // which signifies the first free physical register (excluding p0, which is reserved
as the value 0)
    function int find_first_zero(logic [NUM_PHYSICAL_REGISTERS-1:0] value);
        int result;

    for (int i = 1; i < NUM_PHYSICAL_REGISTERS; i++) begin
            if (value[i] == 0) begin
                result = i;
        ////$display("result: ", result);
                break;
            end
    end
    return result;
  endfunction

 always_ff @(posedge clk) begin
for (int x = 0; x < 2; x++) begin
free_pool[free_regs[x]] = 1'b0;
end
end

  // Register renaming
  always_ff @(posedge clk or posedge reset) begin
    //$display("reset: ", reset);
    overwrittendest_phys1 = 5'b00000;
    overwrittendest_phys2 = 5'b00000;


      // Source register renaming logic for instruction 1

      src1_phys <= (src1_instr != 5'b00000) ? rat[src1_instr] : 5'b0;
      src2_phys <= (src2_instr != 5'b00000) ? rat[src2_instr] : 5'b0;
      // Display messages for source register renaming in instruction 1
     //// $display("Clock %0t: Source Register1 %0d renamed to Physical Register %0d for Instruction 1", $time, src1_instr, src1_phys);
     //// $display("Clock %0t: Source Register2 %0d renamed to Physical Register %0d for Instruction 1", $time, src2_instr, src2_phys);

    // Source register renaming logic for instruction 2
      src1_phys2 <= (src1_instr2 != 5'b00000) ? rat[src1_instr2] : 5'b0;
      src2_phys2 <= (src2_instr2 != 5'b00000) ? rat[src2_instr2] : 5'b0;

  // Check to see if the source registers for instruction 1 are ready or not
   ready_src1_instr <= readyregs[src1_phys];
   ready_src2_instr <= readyregs[src2_phys];
     ready_src1_instr2 <= readyregs[src1_phys2];
     ready_src2_instr2 <= readyregs[src2_phys2];

    if (reset) begin
      //$display("entered1: ", reset);
      // Reset logic (initialize RAT and free pool)
      for (int i = 0; i < NUM_PHYSICAL_REGISTERS; i++) begin
        rat[i] <= 5'b0;
      end
      free_pool <= 5'b0; // Initialize free pool to all 0's
      dest_phys <= 5'b0; // Assign a default value to dest_phys
    end else if (rename_enable) begin
      //$display("entered2: ", reset);
      // Register renaming logic
      if (dest_instr != 5'b0000) begin
        // Destination register is not x0
        //$display("entered3: ", reset);

        dest_phys = find_first_zero(free_pool);

    ////$display("find_first_zero: ", find_first_zero(free_pool));

    //$display("VALUE AT rat[dest_instr]: %b", rat[dest_instr]);
```

```
        overwrittendest_phys1 <= rat[dest_instr];

        destihold <= dest_instr;

                rat[dest_instr] = dest_phys;
        ////$display("dest_instr: ", dest_instr);
                //free_pool = free_pool | (64'b1 << dest_phys); // Mark the physical register as in use
        free_pool[dest_phys] = 1;

        readyregs[dest_phys] = 0;
                ////$display("Clock %0t: Register %0d renamed to Physical Register %0d for Instruction 1", $time, dest_instr, dest_phys);
        end else begin
                dest_phys <= 5'b00000;
        destihold <= dest_instr;

            end
         if (dest_instr2 != 5'b0000) begin

        dest_phys2 = find_first_zero(free_pool);
        ////$display("find_first_zero: ", find_first_zero(free_pool));

        overwrittendest_phys2 <= rat[dest_instr2];

        destihold2 <= dest_instr2;

                rat[dest_instr2] = dest_phys2;
        ////$display("dest_instr: ", dest_instr2);
                //free_pool = free_pool | (64'b1  << dest_phys);
        free_pool[dest_phys2] = 1;

        readyregs[dest_phys2] = 0;
                ////$display("Clock %0t: Register %0d renamed to Physical Register %0d for Instruction 2", $time, dest_instr2, dest_phys2);

    end else begin
                dest_phys2 <= 5'b00000;
                destihold2 <= dest_instr2;
            end

    end


    //$display("VALUE AT overwrittendest_phys1: %b", overwrittendest_phys1);
    //$display("VALUE AT overwrittendest_phys2: %b", overwrittendest_phys2);

        // Display the entire RAT
        $display("Clock %0t: Register Alias Table (RAT): %p", $time, rat);

        // Display the free pool
        $display("Clock %0t: Free Pool: %b", $time, free_pool);

    /*
    $display("Clock %0t: readyregs: %b", $time, readyregs);
    $display("Clock %0t: ready_src1_instr: %b", $time, ready_src1_instr);
    $display("Clock %0t: ready_src2_instr: %b", $time, ready_src2_instr);
    $display("Clock %0t: ready_src1_instr2: %b", $time, ready_src1_instr2);
    $display("Clock %0t: ready_src2_instr2: %b", $time, ready_src2_instr2);
    */
    end

endmodule


```

Appendix VI – Dispatch2.sv file

```

module Dispatch2 (
  input logic [5:0] src1_phys,
  input logic [5:0] src2_phys,
  input logic [5:0] dest_phys,
  input logic [5:0] src1_phys2,
  input logic [5:0] src2_phys2,
  input logic [5:0] dest_phys2,
  input wire clk,
  // signals from decode->rename->dispatch
  input logic [6:0] dispatchopcode1, input
  logic [6:0] opcode2,
  input logic regWrite1,
  input logic regWrite2,
  input logic [2:0] aluOp1,
  input logic [2:0] aluOp2,
  input logic aluSrc1, input
  logic aluSrc2, input logic
  memWrite1, input logic
  memWrite2, input logic
  memRead1, input logic
  memRead2, input logic
  memtoReg1, input logic
  memtoReg2,
  input logic signed [31:0] imm1,
  input logic signed [31:0] imm2,
  input logic decode_enable,

  input logic ready_src1_instr,
  input logic ready_src2_instr,
  input logic ready_src1_instr2,
  input logic ready_src2_instr2,
  input logic [63:0] readyregs,

  output logic [84:0] rs_array[0:15],
  output logic [18:0] rob[0:15],
  output logic [15:0] complete_array = '{default:1'b0},
  input logic [5:0] overwrittendest_phys1,
  input logic [5:0] overwrittendest_phys2,

  input logic [4:0] destihold,
  input logic [4:0] destihold2,
  input logic [3:0] retire_rob [0:1],
  input logic [3:0] clear_rs [0:2]

);


logic [6:0] finopcode1;
logic [6:0] finopcode2;
logic finregWrite1;
logic finregWrite2;
logic [2:0] finaluOp1;
logic [2:0] finaluOp2;
logic finaluSrc1;
logic finaluSrc2;
logic finmemWrite1;
logic finmemWrite2;
logic finmemRead1;
logic finmemRead2;
logic finmemtoReg1;
logic finmemtoReg2;
logic signed [31:0] finimm1;
logic signed [31:0] finimm2;

// Buffer all the docode signals once so that they align with the correct renamed physical registers
  always_ff @(posedge clk) begin
  finregWrite1 <= regWrite1;
  finregWrite2 <= regWrite2;
  finaluOp1 <= aluOp1;
  finaluOp2 <= aluOp2;
  finaluSrc1 <= aluSrc1;
  finaluSrc2 <= aluSrc2;
  finmemWrite1 <= memWrite1;
  finmemWrite2 <= memWrite2;
  finmemRead1 <= memRead1;
  finmemRead2 <= memRead2;
  finmemtoReg1 <= memtoReg1;
  finmemtoReg2 <= memtoReg2;
  finimm1 <= imm1;
  finimm2 <= imm2;
end

  logic [6:0] finfinopcode1;
  logic [6:0] finfinopcode2;

always_ff @(posedge clk) begin
  finfinopcode1 <= finopcode1;
  finfinopcode2 <= finopcode2;
end


  logic [15:0] row_index = 0;
  logic [15:0] new_rs = 0;

  logic [9:0] clk_counter = 0;

  logic [1:0] funcunit1 = 2'b0;
  logic [1:0] funcunit2 = 2'b0;
  logic [5:0] ROBnumber1 = 0;
  logic [5:0] ROBnumber2 = 0; // 5'b1;



  logic newready_src1_instr;
  logic newready_src2_instr;
  logic newready_src1_instr2;
  logic newready_src2_instr2;
```

```

always_ff @(posedge clk) begin

    // Display the rs_array values (reservation station)
$display("Value at rs_array: %p", rs_array);
$display("Value at rs_array0: %b", rs_array[0]);
$display("Value at rs_array1: %b", rs_array[1]);
$display("Value at rs_array2: %b", rs_array[2]);
$display("Value at rs_array3: %b", rs_array[3]);
$display("Value at rs_array4: %b", rs_array[4]);
$display("Value at rs_array5: %b", rs_array[5]);
$display("Value at rs_array6: %b", rs_array[6]);
$display("Value at rs_array7: %b", rs_array[7]);
$display("Value at rs_array8: %b", rs_array[8]);
$display("Value at rs_array9: %b", rs_array[9]);
$display("Value at rs_array10: %b", rs_array[10]);
$display("Value at rs_array11: %b", rs_array[11]);
$display("Value at rs_array12: %b", rs_array[12]);
$display("Value at rs_array13: %b", rs_array[13]);
$display("Value at rs_array14: %b", rs_array[14]);
$display("Value at rs_array15: %b", rs_array[15]);
$display("Value at rs_array16: %b", rs_array[16]);

// Display the ROB (re-order buffer) values
$display("Value at ROB0: %b", rob[0]);
$display("Value at ROB1: %b", rob[1]);
$display("Value at ROB2: %b", rob[2]);
$display("Value at ROB3: %b", rob[3]);
$display("Value at ROB4: %b", rob[4]);
$display("Value at ROB5: %b", rob[5]);
$display("Value at ROB6: %b", rob[6]);
$display("Value at ROB7: %b", rob[7]);
$display("Value at ROB8: %b", rob[8]);
$display("Value at ROB9: %b", rob[9]);
$display("Value at ROB10: %b", rob[10]);
$display("Value at ROB11: %b", rob[11]);
$display("Value at ROB12: %b", rob[12]);

for (int x = 0; x < 2; x++) begin
rob[retire_rob[x]] = 19'b0000000000000000000;
end

end

  always_ff @(posedge clk) begin
    for (int x = 0; x < 3; x++) begin
     rs_array[clear_rs[x]] = 85'b0000000000000000000000000000000000000000
0000000000000000000000000000000000000000000;
    end
  end

  always_ff @(posedge clk) begin
   clk_counter = clk_counter + 1'b1;
   newready_src1_instr = readyregs[src1_phys];
   newready_src2_instr = readyregs[src2_phys];
   newready_src1_instr2 = readyregs[src1_phys2];
```

```
    newready_src2_instr2 = readyregs[src2_phys2];

 if (finaluOp1 !== 3'bxxx) begin
 //$display("dispatchopcode1: %b", dispatchopcode1);
 //$display("opcode2: %b", opcode2);
 //$display("dest_phys: %b", dest_phys);
 //$display("dest_phys2: %b", dest_phys2);

 /*
 $display("dispatch %0t: ready_src1_instr: %b", $time, newready_src1_instr);
 $display("dispatch %0t: ready_src2_instr: %b", $time, newready_src2_instr);
 $display("dispatch %0t: ready_src1_instr2: %b", $time, newready_src1_instr2);
 $display("dispatch %0t: ready_src2_instr2: %b", $time, newready_src2_instr2);

 $display("dispatch %0t: src1_phys: %b", $time, src1_phys);
 $display("dispatch %0t: src2_phys: %b", $time, src2_phys);
 $display("dispatch %0t: src1_phys2: %b", $time, src1_phys2);
 $display("dispatch %0t: src2_phys2: %b", $time, src2_phys2);
 $display("dispatch %0t: readyregs: %b", $time, readyregs);
 */


  //Code to reset row index if you get to the end of the reservation station
 if (row_index == 4'b10000) begin
 row_index = 4'b0000;
 end

 // Useful for debugging, places x between each separate element of reservation station
 //rs_array[row_index] = {ROBnumber1,1'bx,funcunit1,1'bx,finimm1,1'bx,finmemtoReg1,1'bx,
 finmemRead1,1'bx,finmemWrite1,1'bx,finaluSrc1,1'bx,finaluOp1,1'bx,finregWrite1,1'bx,
 newready_src2_instr, 1'bx, src2_phys,1'bx,newready_src1_instr,1'bx,src1_phys,1'bx,dest_phys,
 1'bx,dispatchopcode1,1'b1};

 //rs_array[row_index+1] = {ROBnumber2,1'bx,funcunit2,1'bx,finimm2,1'bx,finmemtoReg2,1'bx,
 finmemRead2,1'bx,finmemWrite2,1'bx,finaluSrc2,1'bx,finaluOp2,1'bx,finregWrite2,1'bx,
 newready_src2_instr2, 1'bx, src2_phys2,1'bx,newready_src1_instr2,1'bx,src1_phys2,1'bx,
 dest_phys2,1'bx,opcode2,1'b1};


 rs_array[row_index] = {ROBnumber1,funcunit1,finimm1,finmemtoReg1,finmemRead1,finmemWrite1,
 finaluSrc1,finaluOp1,finregWrite1, newready_src2_instr, src2_phys,newready_src1_instr,
 src1_phys,dest_phys,dispatchopcode1,1'b1};

 rs_array[row_index+1] = {ROBnumber2,funcunit2,finimm2,finmemtoReg2,finmemRead2,finmemWrite2,
 finaluSrc2,finaluOp2,finregWrite2, newready_src2_instr2, src2_phys2,newready_src1_instr2,
 src1_phys2,dest_phys2,opcode2,1'b1};

 //rs_array[new_rs] = {ROBnumber1,1'bx,funcunit1,1'bx,finimm1,1'bx,finmemtoReg1,1'bx,
 finmemRead1,1'bx,finmemWrite1,1'bx,finaluSrc1,1'bx,finaluOp1,1'bx,finregWrite1,1'bx,
 newready_src2_instr, 1'bx, src2_phys,1'bx,newready_src1_instr,1'bx,src1_phys,1'bx,dest_phys,
 1'bx,dispatchopcode1,1'b1};

 //rs_array[new_rs+ 1] = {ROBnumber2,1'bx,funcunit2,1'bx,finimm2,1'bx,finmemtoReg2,1'bx,
 finmemRead2,1'bx,finmemWrite2,1'bx,finaluSrc2,1'bx,finaluOp2,1'bx,finregWrite2,1'bx,
 newready_src2_instr2, 1'bx, src2_phys2,1'bx,newready_src1_instr2,1'bx,src1_phys2,1'bx,
 dest_phys2,1'bx,opcode2,1'b1};

 rob[row_index] = {complete_array[row_index],destihold,overwrittendest_phys1,dest_phys,1'b1};

```

```
rob[row_index+1] = {complete_array[row_index+1'b1],destihold2,overwrittendest_phys2,
dest_phys2,1'b1};

ROBnumber1 = ROBnumber1 + 1'b1;

ROBnumber2 = ROBnumber2 + 1'b1;

row_index = row_index + 2'b10;

end

end

endmodule


```

Appendix VII – Fire2.sv file

```

module Fire2 (
  input logic clk,
  input logic [84:0] rs_array [0:15],
  input logic [18:0] rob [0:15],
  output logic [31:0] physregisters [63:0]= '{default:32'h0},
  output logic [15:0] complete_array, // = '{default:1'b0},
  output logic [15:0] store_complete_array = '{default:1'b0},

  output logic [31:0] store_address,
  output logic [31:0] store_value,
  output logic [31:0] load_address,

  input logic [31:0] memory [0:1023],
  output logic [3:0] clear_rs [0:2]
);

reg [7:0] mem_actual [0:255];      // Temporary memory to store data

parameter NUM_PHYSICAL_REGISTERS = 64; // Adjust based on your architecture

logic src1_ready;
logic src2_ready;

logic [0:3] clearval;
logic fu3_src1;
logic fu3_src2;
logic fu3_dest;
logic fu3_imm;

logic all_units_busy;
logic fu1_busy;
logic fu2_busy;
logic fu3_busy;


logic [2:0] countfinal;

logic [64-1:0] regfreedfire = '0;
logic [4:0] clearcount = 5'b0;

always_ff @(posedge clk) begin
complete_array = {default:1'b0};

// Display statemetns for physical register file
$display("Value at physregisters0: %b", physregisters[0]);
$display("Value at physregisters1: %b", physregisters[1]);
$display("Value at physregisters2: %b", physregisters[2]);
$display("Value at physregisters3: %b", physregisters[3]);
$display("Value at physregisters4: %b", physregisters[4]);
$display("Value at physregisters5: %b", physregisters[5]);
$display("Value at physregisters6: %b", physregisters[6]);
$display("Value at physregisters7: %b", physregisters[7]);
```

```
$display("Value at physregisters8: %b", physregisters[8]);
$display("Value at physregisters9: %b", physregisters[9]);
$display("Value at physregisters10: %b", physregisters[10]);
$display("Value at physregisters11: %b", physregisters[11]);
$display("Value at physregisters12: %b", physregisters[12]);
$display("Value at physregisters13: %b", physregisters[13]);
$display("Value at physregisters14: %b", physregisters[14]);

/*
// Display statemetns for important memory values
$display("memory[0]: %b", memory[0]);
$display("memory[16]: %b", memory[16]);
$display("memory[24]: %b", memory[24]);
*/

fu1_busy = 0;
fu2_busy = 0;
fu3_busy = 0;
countfinal = 0;
  all_units_busy = (fu1_busy && fu2_busy && fu3_busy);

  for (int i = 0; i < 16; i++) begin

 all_units_busy = (fu1_busy && fu2_busy && fu3_busy);

if ((!all_units_busy)) begin

      src1_ready = rs_array[i][20];
      src2_ready = rs_array[i][27];

      if ((src1_ready && src2_ready) || ((regfreedfire[rs_array[i][19:14]] == 1'b1) && (regfreedfire[rs_array[i][26:21]] == 1'b1))) begin

    if (countfinal < 2'b10) begin

        if ((rs_array[i][7:1] == 7'b0010011) || (rs_array[i][7:1] == 7'b0110011)) begin

            // ADDI (I) type instructions
            if (rs_array[i][7:1] == 7'b0010011) begin
                //ADDI

                if(rs_array[i][31:29] == 3'b000) begin
                    physregisters[rs_array[i][13:8]] <= physregisters[rs_array[i][19:14]] + rs_array[i][67:36];

                end
                //ANDI
                if(rs_array[i][31:29] == 3'b111) begin
                    physregisters[rs_array[i][13:8]] <= physregisters[rs_array[i][19:14]] & rs_array[i][67:36];
                end
            end

            // ADD (R) type instructions
            if (rs_array[i][7:1] == 7'b0110011) begin
            //ADD
            if(rs_array[i][31:29] == 3'b000) begin
```

```
                    physregisters[rs_array[i][13:8]] <= physregisters[rs_array[i][19:14]] +
physregisters[rs_array[i][26:21]];
                end
            //SUB
            if(rs_array[i][31:29] == 3'b001) begin
                    physregisters[rs_array[i][13:8]] <= physregisters[rs_array[i][19:14]] -
physregisters[rs_array[i][26:21]];
                    end

            //XOR
            if(rs_array[i][31:29] == 3'b100) begin
                    physregisters[rs_array[i][13:8]] <= physregisters[rs_array[i][19:14]] ^
physregisters[rs_array[i][26:21]];
                end
            //SRA
            if(rs_array[i][31:29] == 3'b101) begin
                    physregisters[rs_array[i][13:8]] <= physregisters[rs_array[i][19:14]] >>>
physregisters[rs_array[i][26:21]];
                end

            end
if(countfinal == 0 ) begin
clear_rs[0] = i;
end
if (countfinal == 1) begin
clear_rs[1] = i;
end

countfinal = countfinal + 1'b1;
regfreedfire[rs_array[i][13:8]] <= 1'b1;

complete_array[i] <= 1'b1;
        end

    fu1_busy = 1'b1;
    end


    // FU 3 for sw and lw instructions
    if (!fu3_busy) begin
        if (rs_array[i][7:1] == 7'b0100011 || rs_array[i][7:1] == 7'b0000011) begin
            // SW
            if(rs_array[i][7:1] == 7'b0100011) begin
            memory[physregisters[rs_array[i][19:14]] + rs_array[i][67:36]] <=
physregisters[rs_array[i][26:21]];
            end
            //LW
            if(rs_array[i][7:1] == 7'b0000011) begin
                 load_address <= physregisters[rs_array[i][19:14]] + rs_array[i][67:36];
             physregisters[rs_array[i][13:8]] <= memory[physregisters[rs_array[i][19:14]] +
rs_array[i][67:36]];

            end
  clear_rs[2] = i;
  fu3_busy = 1'b1;

  complete_array[i] <= 1'b1;
  store_complete_array[i] <= 1'b1;
  regfreedfire[rs_array[i][13:8]] <= 1'b1;
        end

    end

      end
    end
  end
end

endmodule
```

Appendix VIII – Retire2.sv file

```

module Retire2 (
  input logic clk,
  input logic [18:0] rob [0:15],
  input logic [31:0] physregisters [63:0],
  output logic [31:0] regfile [0:31] = '{default:1'b0},
  input logic [15:0] complete_array,
  input logic [31:0] store_address,
  input logic [31:0] store_value,
  input logic [31:0] load_address,
  input logic [15:0] store_complete_array = '{default:1'b0},
  output logic [31:0] memory [0:1023],
  output logic [3:0] retire_rob [0:1],
  output logic [5:0] free_regs [0:1]
);

logic [3:0] val = 3'b0;
logic [15:0] checkcompleted = '0;
logic [15:0] storeloadcheckcompleted = '0;
logic [2:0] countretire = 2'b0;

  always_ff @(posedge clk) begin

val = 3'b000;

countretire = 2'b00;

for (int i = 0; i < 16; i++) begin

if (complete_array[i] == 1) begin

if ((checkcompleted[i] !== 1'b1) && (countretire < 2'b10) ) begin
 regfile[rob[i][17:13]] <= physregisters[rob[i][6:1]];

  retire_rob[val] <= i;

  free_regs[val] =  rob[i][12:7];

  val = val + 3'b001;
  checkcompleted[i] = 1'b1;
  countretire = countretire+2'b01;

end
end
end


for (int i = 0; i < 16; i++) begin

  if (store_complete_array[i] == 1) begin

    if (storeloadcheckcompleted[i] !== 1'b1) begin

```

```verilog
        storeloadcheckcompleted[i] <= 1'b1;
end
end
end

// Display statements for architectural register file
$display("Register File 0 (x0): %b", regfile[0]);
$display("Register File 1 (x1): %b", regfile[1]);
$display("Register File 2 (x2): %b", regfile[2]);
$display("Register File 3 (x3): %b", regfile[3]);
$display("Register File 4 (x4): %b", regfile[4]);
$display("Register File 5 (x5): %b", regfile[5]);
$display("Register File 6 (x6): %b", regfile[6]);
$display("Register File 7 (x7): %b", regfile[7]);
$display("Register File 8 (x8): %b", regfile[8]);
$display("Register File 9 (x9): %b", regfile[9]);
$display("Register File 10 (x10): %b", regfile[10]);
$display("Register File 11 (x11): %b", regfile[11]);
$display("Register File 12 (x12): %b", regfile[12]);


//$display("memory[0]: %b", memory[0]);
//$display("memory[16]: %b", memory[16]);
//$display("memory[24]: %b", memory[24]);

end

endmodule
```