Designing a processor with RISC-V

ECE 6993 Independent Study

Yimin Gao(yg9bq@virginia.edu)

April 2021

The whole project code was pushed to https://github.com/YiminGao0113/RISC-V-Processor

1 General Purpose

I took the class Computer Architecture and Design this semester. The class required students to get all the assigned labs (10 labs) checked off by TAs, while no lab reports were required except for the last lab (A short report demonstrating if the machine can run the given program).

However, I became interested in the region of computer architecture and VHDL coding. Thus I decided to compose all my detailed design ideas, additional thoughts of the topic, VHDL code, testbench results, synthesis results and some additional studies on related region as an independent study report beyond my CAD class requirements. Also, it will be convenient to have a report including all my detailed thoughts of the project when I need to look back into my processor design in the future.

2 Introduction

A RISC-V can be implemented using different microarchitecture. There are multiple ways of designing each elements, resolving different hazards. Even the whole pipeline design could be different but with same function (maybe with different efficiency). Also, different from traditional programming language, HDL code could run in either parallel or sequential order, or even both with regards to different type of design. So with the same hardware architecture, there are still a lot of ways to interpret in VHDL coding as well. This report was based on my own design method which was supposed to be clear and well-organized with high efficiency in resolving hazards.

The report includes all of my design ideas to build a RISC-V processor, from a simple component to the whole pipeline design. The processor was designed based on RISC-V, which is an instruction set architecture that was originally designed for research and education. It is an open resource that is freely available. A processor is a RTL(Register-Transfer Level) hardware that interprets instructions fetched from memory system. It performs basic arithmetic, logic, controlling, and input/output operations specified by the instructions. The whole design includes basic elements design (Register, multiplexer, incrementer, etc.),and then deploying basic elements to form an entire processor pipeline following the design descipline of RISC-V. One saliency of VHDL is that every design entity we built could be reused as a unit component in future design, which represents VHDL's unique feature of multiple levels of abstraction. The general procedure of a simple RTL design is to firstly programming in hardware description language (VHDL in this project), and then verifying the design with Multisim simulation using a testbench, and finally synthesizing the schematic of the design.

3 Design Principle

3.1 Multiple levels of abstraction

One general idea of computer architecture design is multiple levels of abstraction. The whole architecture is complicated and hard to interpret. Abstraction is a powerful tool for complexity management. Abstraction allows use of a construct while avoiding implementation details. For instance, building a fundamental device like a register or a multiplexer is relatively easy and straightforward. Such devices would perform as a unit for future design. By applying such units into design, we could get secondary design units such like incrementer, counter, etc. Moreover, when I designed a register file which is comparably complicated, however, I could simply implement the register file as an entity with configurable generic and port mapping in another design. Thus if we keep increasing the levels of abstraction, we could accomplish the design we want in an explicit and well-organized way.

3.2 Pipeline methods

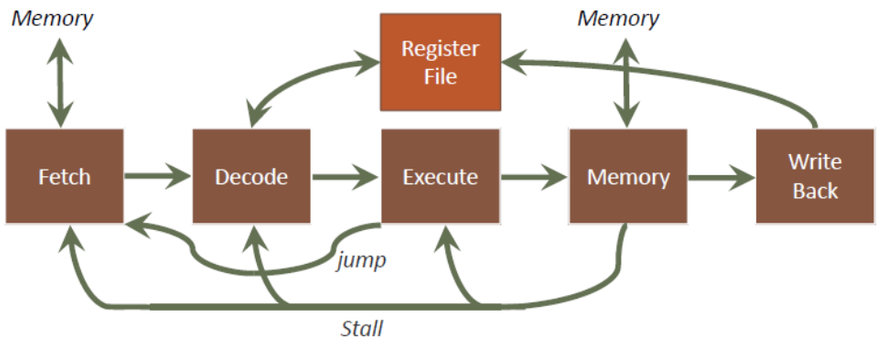


Figure 1: Pipeline flow

Leveraging on multiple levels of abstraction, a design idea named pipeline is also applied for hardware speed improvement(As shown in Figure 1). Each stage includes one set of registers for the input data from last stage. Thus, each stage takes one time cycle to perform. The pipeline method maximums the usage of each stage to enhance the speed of the machine. However, to make sure there exists no multiple occupation of instructions in each stage, there are several cases we need to consider. Firstly, the pipeline length should be the same for all types of instructions. For instance, the LUI instruction which operates as Load Upper Immediate only needs three stages instead of five(LUI only needs operations at fetch stage, decode stage and write back stage). Even so, the data flow must follow the entire five-stage pipeline path rigorously to make sure all instructions move to next stage continuously with no interruptions. Technically, the only instructions in RISC-V that are supposed to operate at each pipeline stage is the Load operations. But still, the pipeline design makes the whole process concept well-organized and makes it easier to solve hazard problems. I will talk about methods I used to resolve expected hazards later in each of the stage design. (How to solve control hazards like branch and jump operations in execute stage, how to solve structure hazards by adding an arbiter, how to solve RAW hazard by adding a register tracker)

4 Basic Design

4.1 Counter with synchronous reset

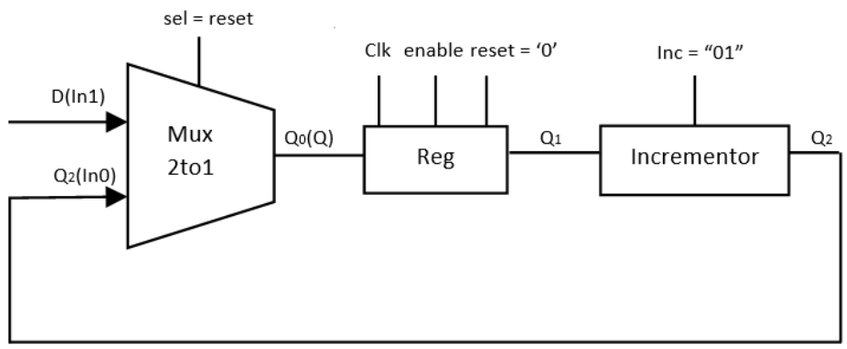


Figure 2: Counter Data flow

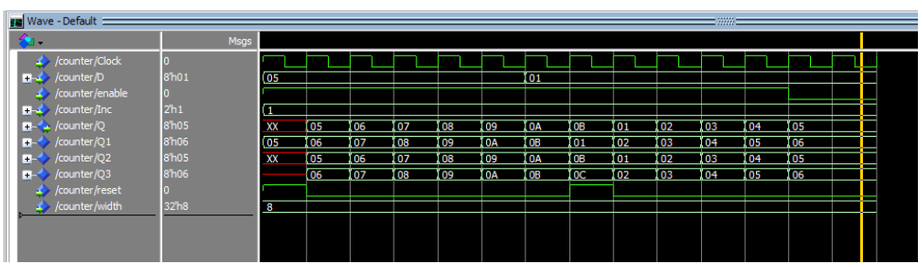


Figure 3: Counter Simulation

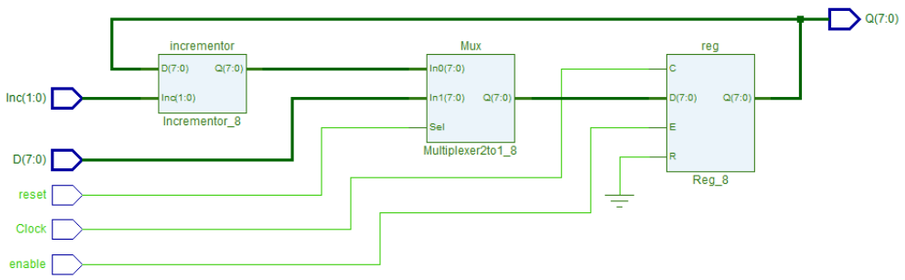


Figure 4: Counter Schematic

The counter should be triggered on the rising edge of clock and will keep counting until either enable equals to 0 or reset equals to 1 on the rising edge of the clock. There will be a input D so that if reset equals to 1 the counter will reset its value to D. To design a counter, we need to design a register with enable and reset, 2 to 1 multiplexer and an incrementer with selective adder number. After I finished designing these three components, they would serve as basic elements to build up a counter. The general design is shown in Figure 2. When the reset equaled to 0, the counter will be counting triggered by the rising edge of clock. As reset changed to 1, the register took D in as input on next clock rising edge, which performed as a synchronous reset.As shown in Figure 3, the output Q showed exactly what we want. The counter kept counting triggered by clock rising edge and reset to the input D when reset equaled to 1 at the rising edge. Also, the counter would stop counting when the enable of the counter which connected to the register enable was 0. As shown in Figure 4, the schematic was the same as our data flow diagram.

4.2 Pipeline Fetch stage

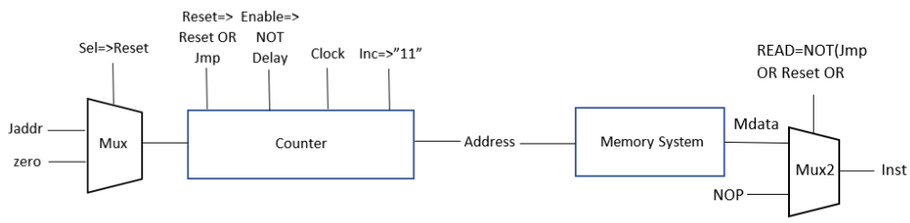


Figure 5: Fetch Stage Data flow

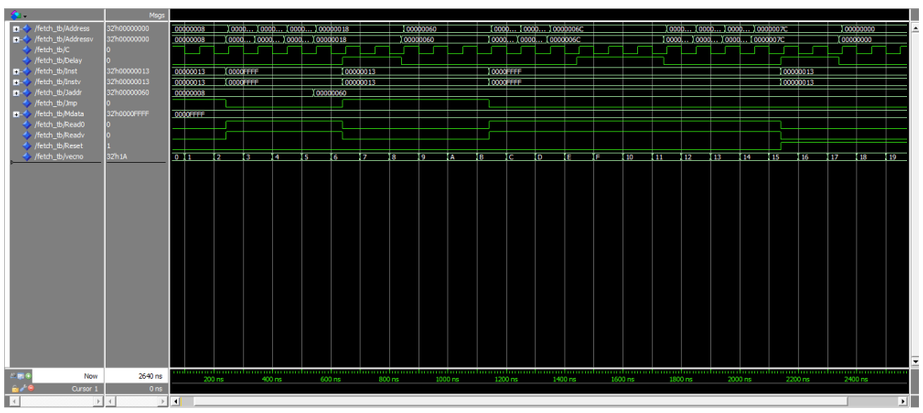


Figure 6: Fetch Simulation

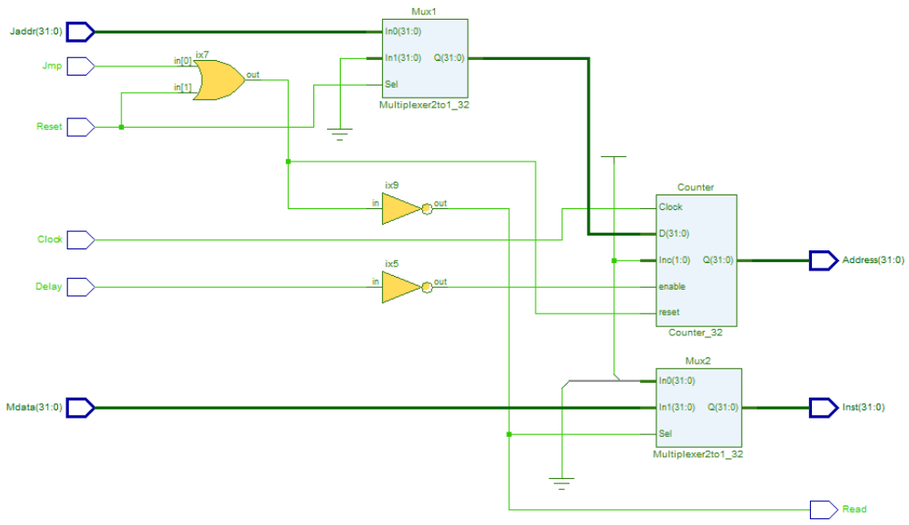


Figure 7: Fetch Stage Schematic

Likewise, the fetch stage applies multiplexers, a counter as basic elements for higher level of abstraction. The general data flow design in shown in Figure 5. The stage has four inputs: Jump, reset, delay and Jump address. The processor will go to the address in the memory system and fetch the instruction out as Mdata. However, we are not representing the memory system at this stage now, Mdata would be an input for test simulation and synthesis. Based on the general design, If there is a jump and reset equals to 0, the counter will stop counting and take Jump address as input on the next rising edge of clock. Also, if both jump and reset are asserted at the same clock rising edge, the system will ignore the jump request and reset the counter to zeros. The fetch stage delay is applied with a not gate and then connected to the counter enable port. The second multiplexer is used to detect if Mdata is a valid instruction at this time(Jump = Reset = 0). As shown in Figure 6, I used a testbench to test the system considering every possible circumstances. The schematic is shown in Figure 7 which is basically the same as I designed the data flow.

4.3 Pipeline Decode stage

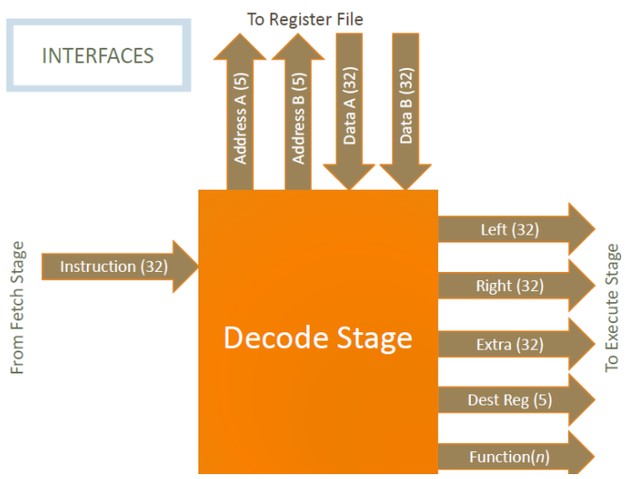


Figure 8: Decode Stage Data Flow

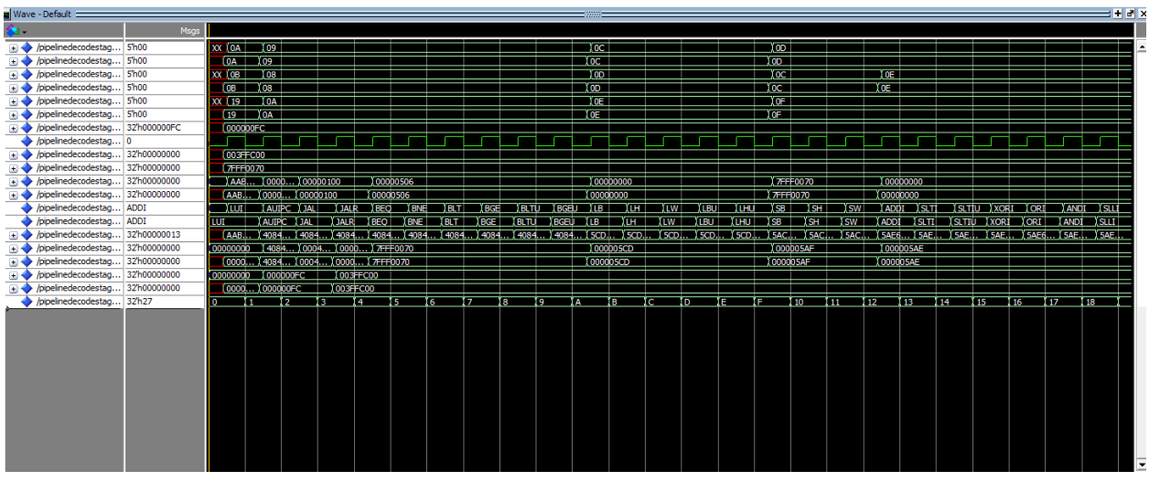


Figure 9: Decode Stage Simulation

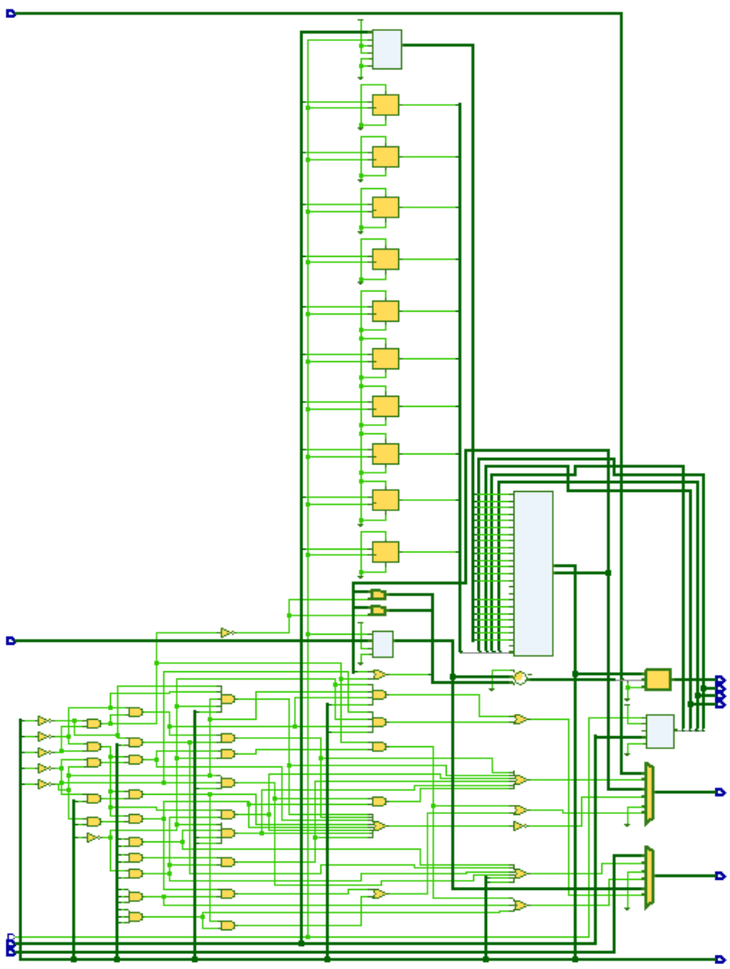


Figure 10: Decode Stage Schematic

The pipeline decode stage consists of a register, a decoder and multiplexers. In fact, the decoder is also made of multiplexers. But I designed a decoder first and then use it here as a basic element. Thus the stage only have a memory set for storing instructions. All the other components(multiplexers) are combinational devices. The register is for holding the instruction from fetch stage and passing it to decoder on every rising edge of clock. The decoder, of course, decodes the 32-bit instructions into function type, rs1(5 bits register address), rs2, rd, rs1v(1 or 0 to represent if the data of rs1 is valid), rs2v, rdv. The address of registers will be sent to the register file. The register file will return 2 32-bit data. After that the multiplexers decide how to assign these values from register file(RAM) and the decoder to the output of pipeline decode stage based on the function type. For example, if the instruction is a LB, then the multiplexers assign Left as Immediate value from decoder and assign right as the DataA from the register file.(Base Register Data) The decode stage send these values to execute stage. The left and right signal will link to

the ALU in the execute stage. The ALU will add them up to calculate the address to load from in the memory system and send the address the memory stage. The memory stage then fetch the data from the memory stage and send the data to the write back stage. The write back stage will then write back to the register file using the destination register address and the data to write sent from previous stage. So the certain output here in decode stage controlled by multiplexers are left, right and extra. The destination register address and the function type is directly connected to the next stage from the decoder and will keep transferring until the last stage.

As shown in Figure 9 and Figure 10, I wrote a testbench and created a testvector txt file to test all the functions in RISC-V. The generated schematic of the decode stage as shown in Figure 11 was really complicated as well. The schematic was complicated, but the idea was clear that we decoded the data out of the register set, and then sent them to execute stage and register file through several multiplexers. The multiplexers selected the correct path for each data based on the instruction type. It showed in the design area that the whole stage consists of 32 flip-flops, meaning the only sequential device here is the register to store the 32-bit instruction from the fetch stage. Thus both the decoder and multiplexers were combinational devices which meets my expectation.

4.4 Pipeline Execute Stage

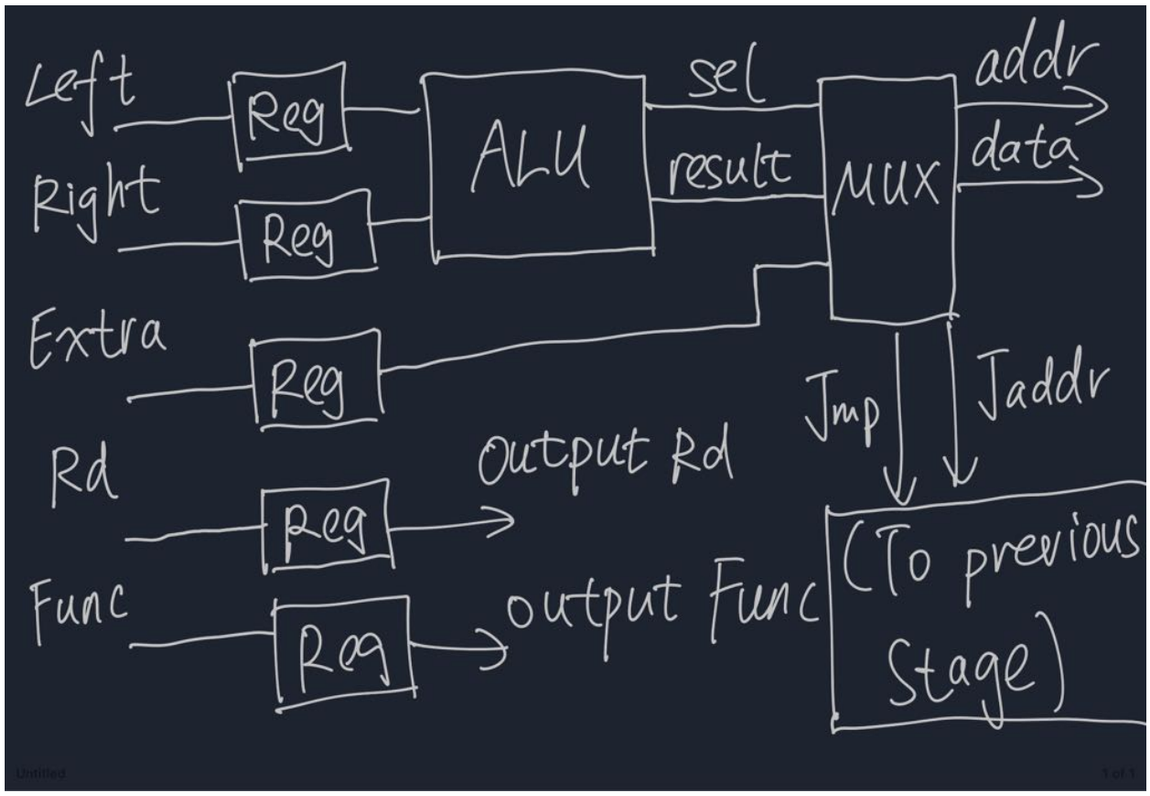


Figure 11: Execute Stage Design

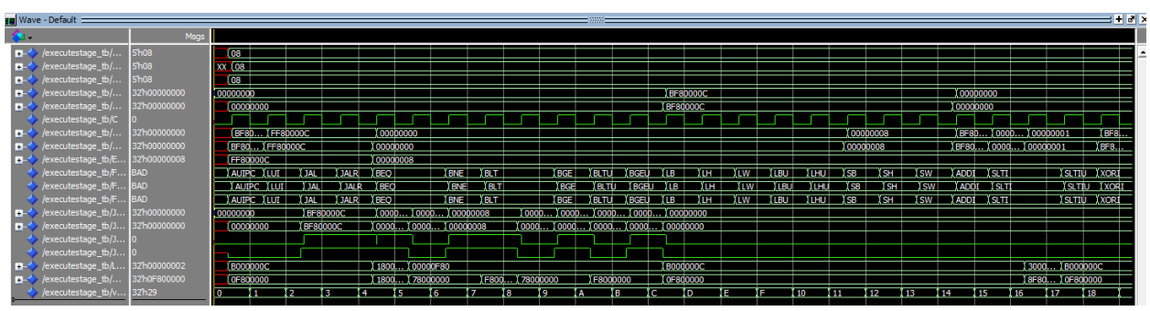


Figure 12: Execute Stage Simulation

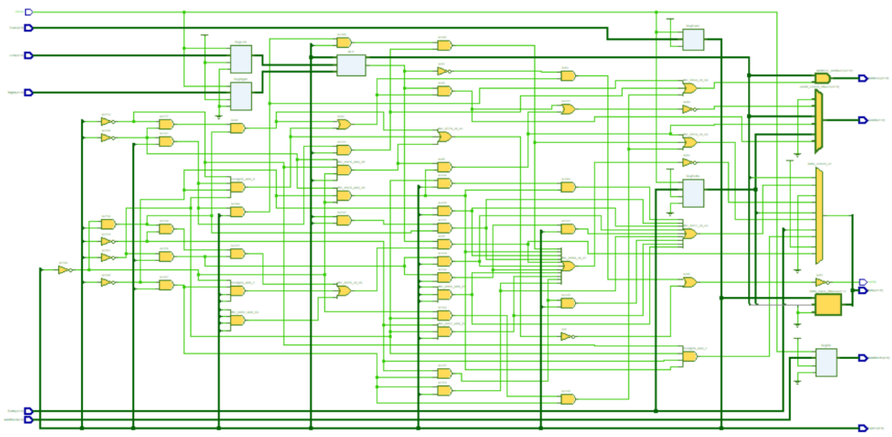


Figure 13: Execute Stage Schematic

The execute stage is similar to previous stage that it consists of five registers for storing the output from decode stage, an ALU and multiplexers. The ALU I designed takes Left and Right as inputs and has result, sel as output. The ALU is able to perform add, subtract, logic shift and branch operations. The sel output of ALU is set as whether the branch will be taken or not when the instruction is a branch(sel¡=1 if the branch is taken, otherwise 0). All the outputs are controlled by a set of combinational devices(multiplexers). If there is the instruction is a jump such as JAL or a branch that will be taken(determined in ALU), the execute stage will send a jump signal and a 32-bit jump address to previous stages. At the same time the jump is executed, all the previous stages should be sending a NOP to its next stage and the fetch stage should fetch the jump address on the next clock cycle. To achieve that, I connected 1-bit control jump to the synchronous reset port in fetch stage(Jmp in Figure 5), and to a multiplexer control to the decode stage which will push forward a NOP operation instead of the decoded operation. Likewise, as shown in Figure 14, I also created a large test bench testing all the function types in RISC-V. All the components except the registers holding the output of decoder stage are combinational devices. So as I synthesized the design (as shown in Figure 15)[2] the schematic includes 107 flip-flops (Left 32 + Right 32 + Extra 32 + Rd 5 + Func 6 = 107), which meets the requirement. During the design of execute stage, we developed the method to jump and pause all previous stages using the jump control. Thus we need to go back to decode stage and add up a multiplexer to send NOP when the jump is asserted.

4.5 Pipeline Memory Stage, Arbiter and Write Back Stage

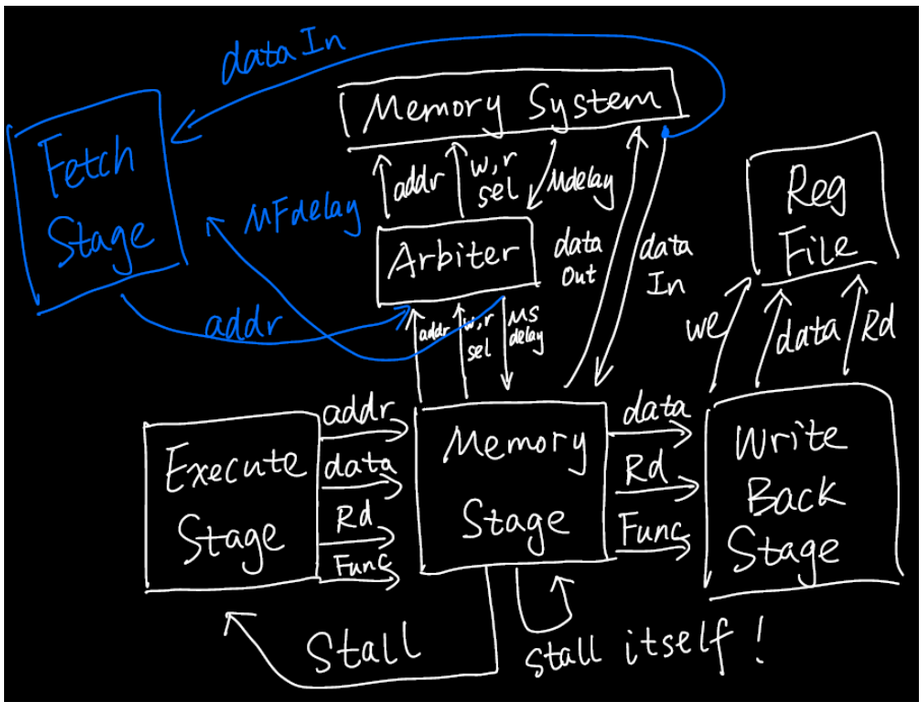


Figure 14: General Design Flow

4.5.1 General Design of Memory Stage and Arbiter

As the general design shown in Figure 14, the basic step to build up memory stage and write back stage is the same as previous stage. There should be one set of registers storing data from previous stage(data that has decoded instruction information) and multiplexers for providing the correct path for each data based on the function. However, there is one unique situation we need to consider. When the memory stage is trying to access (read or write) data from the memory system, the fetch stage is also accessing the memory system. That situation with multiple access to memory system at one time is called structure hazards. To avoid that, we need to add an arbiter to the system. The arbiter should be totally combinational devices(which are also multiplexers) to switch the correct data path for input address, access data length(sel=00:byte, sel=01:half word, sel=10:word) and output delay. However, since the times memory stage accessing the memory system(load or store type instructions) is a lot less than fetch stage accessing the memory system, we set that the fetch stage should always yield to memory stage when memory stage needs to access from memory system to easily avoid such hazards. I designed the sel signal out from memory stage which should be the data length (sel=00:byte, sel=01:half word, sel=10:word) to be the flag to determine if the memory stage needs to access data from memory system. If it does, the sel would be either 00, 01 or 10. If it doesn't, I set the sel to be 11 which means the memory stage won't access memory system for this function(instruction). Thus anytime the input sel of arbiter is "11", the delay output to fetch stage would be 1 to stall the fetch stage. The arbiter then transfer data from memory stage to memory system to perform the memory operation. Otherwise, if sel equals to "11", then the arbiter will transfer data from fetch stage to memory stage.

As the memory stage is performing a read or write, the output Mdelay should be 1 until the operation is finished. The delay should be sent to either fetch stage or memory stage based on the sel signal out of the memory stage(which represent which stage is using the memory system). However, for the 32-bit data we need to write or read from memory system, we can connect them directly to fetch stage and memory stage. Because it is easy to use the control signal to tell if the current stage will use the data or not. For instance, if the fetch stage is using the memory system to fetch an instruction, the instruction will be sent to both fetch stage and memory stage. However, the 32-bit data output from memory stage to write back stage is controlled by multiplxers based on the current function(6-bit data decoded in decode stage and then transfered forward to each stage). In other word, the combinational devices(multiplexers) will select the output data from execute stage instead of memory system this time. Thus the general idea to simplify the circuit is use control signal to control the dataflow instead of changing the data each time.

4.5.2 Write Back Stage

The write back stage takes 32-bit data, 5-bit destination register address and 6-bit enumerated type function from memory stage and provides 1-bit write enable, 32-bit data and 5-bit destination register address to the register file. Basically it includes the same register set to store the input like all other stages. The write back stage then transfer the data and destination register address to the register file and generates write enable based on the function. Like what I said, the control bit write enable is essential to minimum circuit design. So if the write enable is 0, the register file will not write the data even though the data and the Rd signal are being provided.

4.5.3 Testbench simulation and synthesis

Here are the simulation and synthesis for Memory Stage, Arbiter and Write Back Stage as shown in the figure. All the results met my expectation that I mentioned above. All my test bench codes and test vectors file were also uploaded into github.

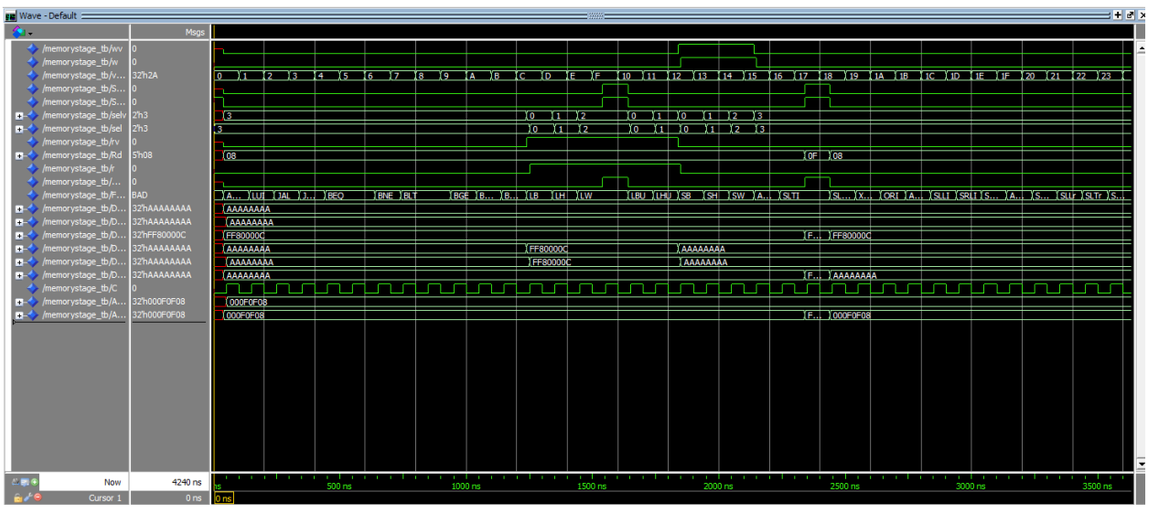


Figure 15: Memory Stage Simulation

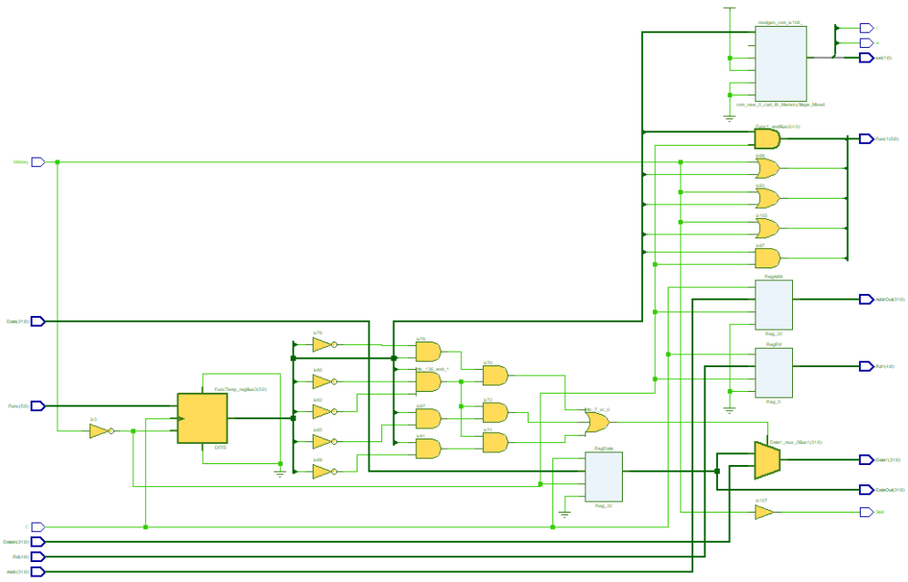


Figure 16: Memory Stage Schematic

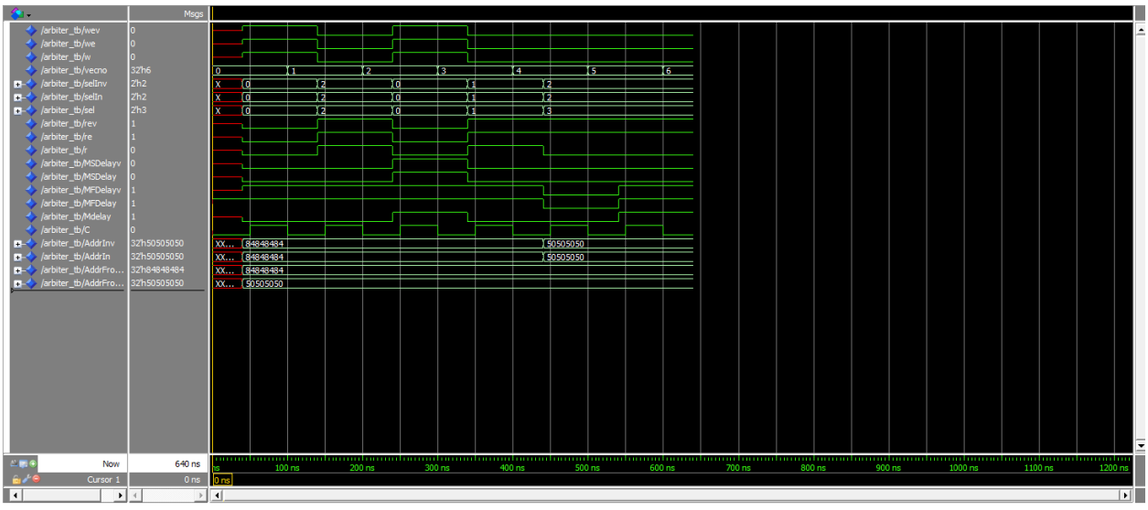


Figure 17: Arbiter Simulation

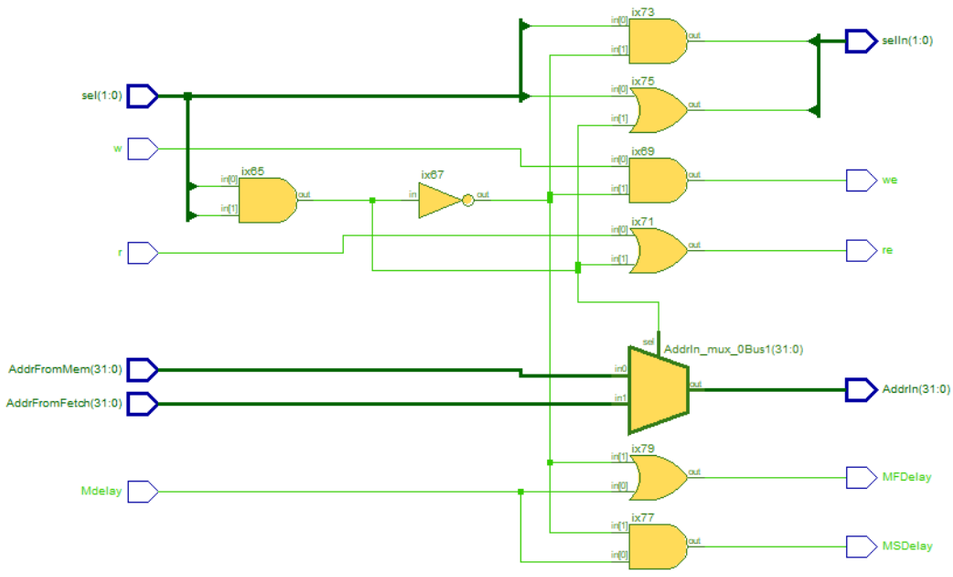


Figure 18: Arbiter Schematic

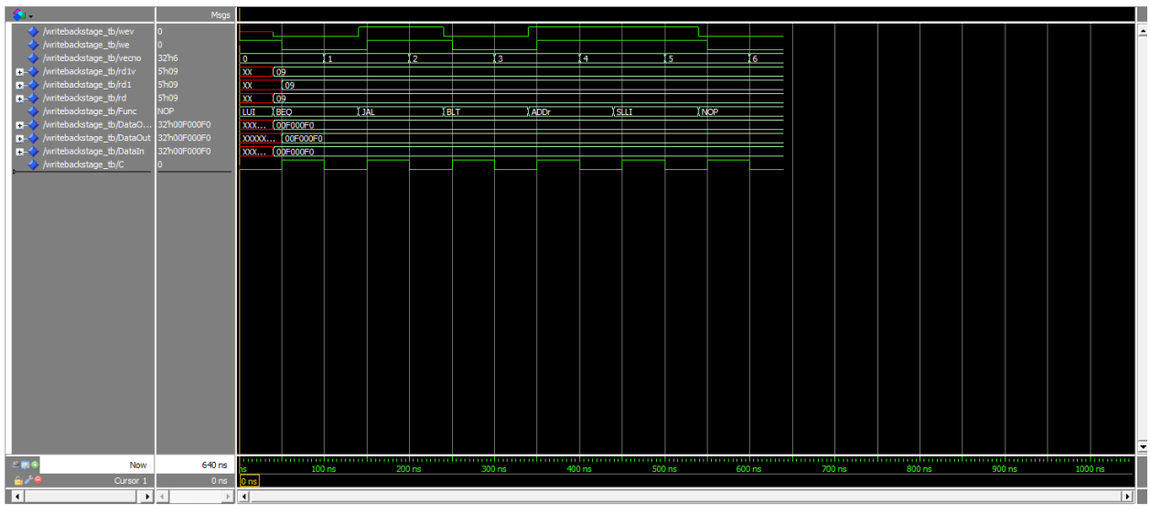


Figure 19: Write Back Stage Simulation

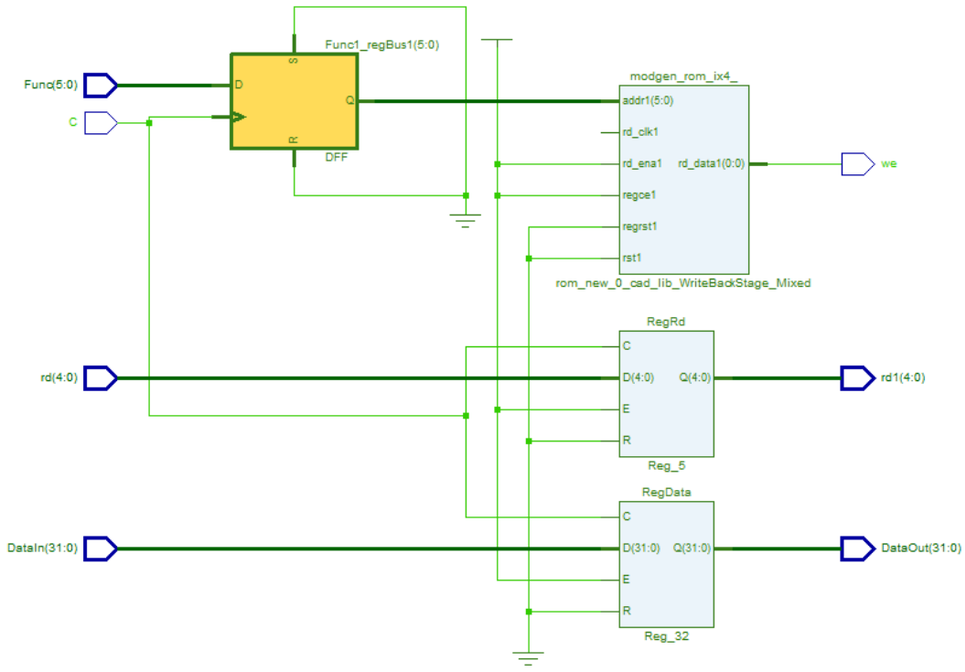


Figure 20: Write Back Stage Schematic

4.6 Register File and Register Tracker

To finish the entire design, we need a register file which has 32 32-bit registers as RAM. The register file should read two register addresses from the decode stage and write to one register address from write back stage when the write enable signal from write back stage is one. Since the read operation will not make change the register file and it is combinational without clock affected, I designed the register file to read the two registers no matter the system needs or not. The following execute stage will evaluate if the two data is valid or not by the function data we keep pushing to each stage.

Also, we need a register tracker to avoid RAW(Read after write hazard). The hazard is caused by reading data in decode stage before the data is written correctly. For instance, if there are two sequential instructions, a load and an add. The second instruction add use the register the first one loaded as an input. A load will write the value back to register file at the last stage of pipeline, while the following add execute the ALU operation at the third stage(execute stage). Thus incorrect input values leads to a incorrect result. To avoid that, we could add another smaller register file which has 32 2-bit registers. At the decode stage reads out rd(destination register) and rdv(destination register valid or not), if rd is valid (rdv = 1), the tracker will add one to the specific register in register tracker. So if there are multiple operations using the same destination register in a short time period, the flag of the register will go up (3 at maximum since there are only three stages between decode stage and write back stage). The write back stage will subtract one to the rd when it is valid. By importing such register tracker, the decode stage will check the flag of r1 (when r1v = 1) and the flag of r2 (when r2v = 1).

Test and simulation Here are the simulation and synthesis for register file and register tracker. I tested all the possible conditions that will occur in my processor pipeline. The register file can operate two reads and one write in a single time cycle. The register tracker can assert stalls to decode stage and fetch stage when there was a RAW hazard. All results met my expectation.

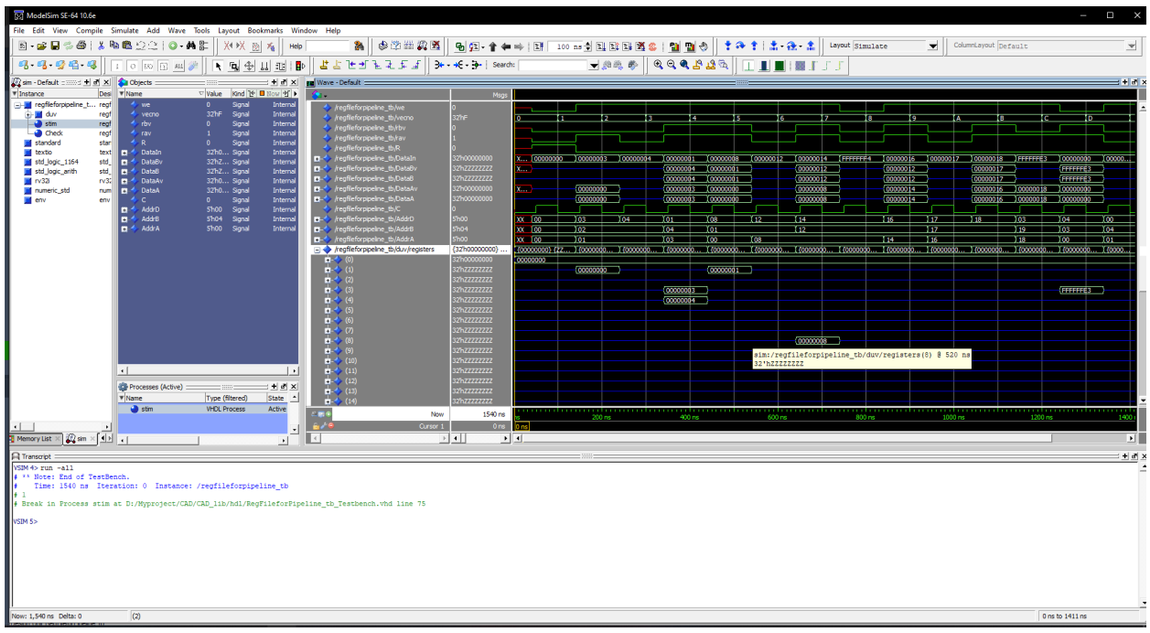


Figure 21: Register File Simulation

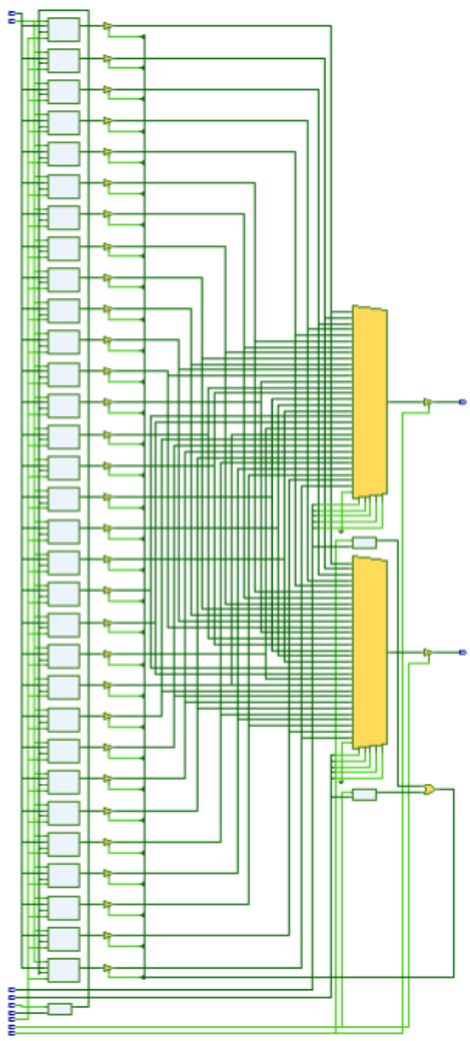


Figure 22: Register File Schematic

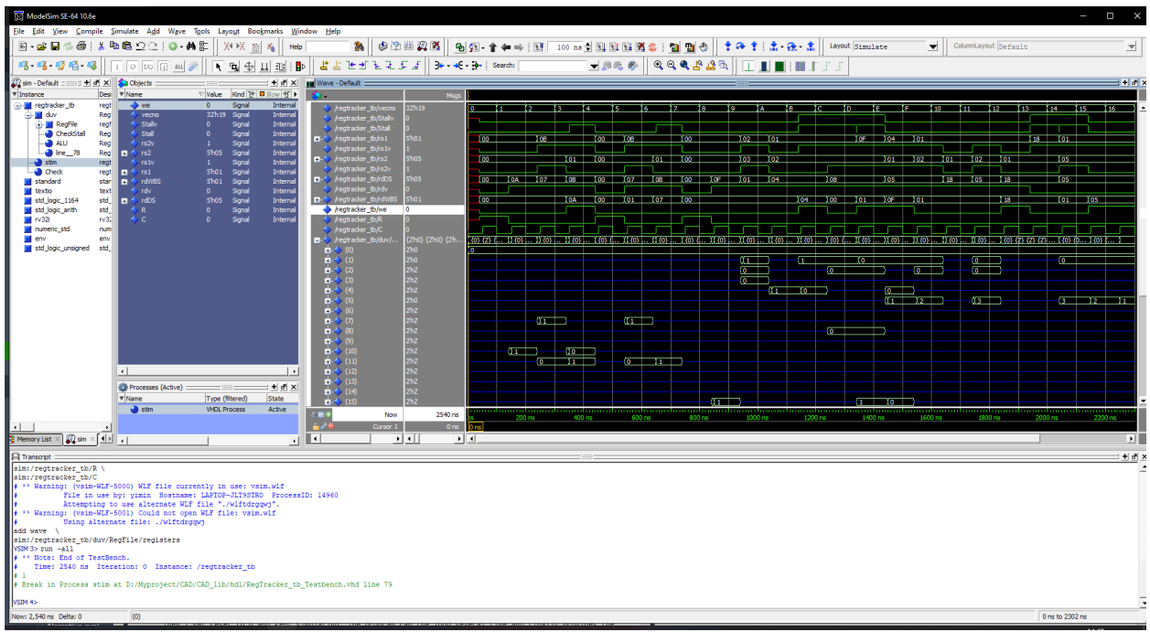


Figure 23: Register Tracker Simulation

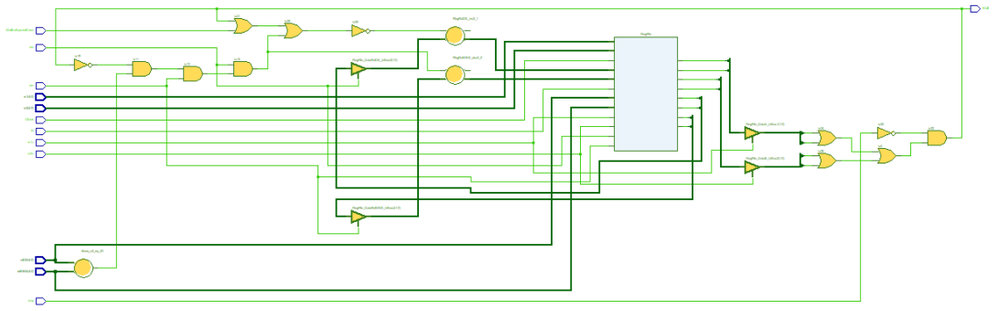


Figure 24: Register Tracker Schematic

5 Memory System test

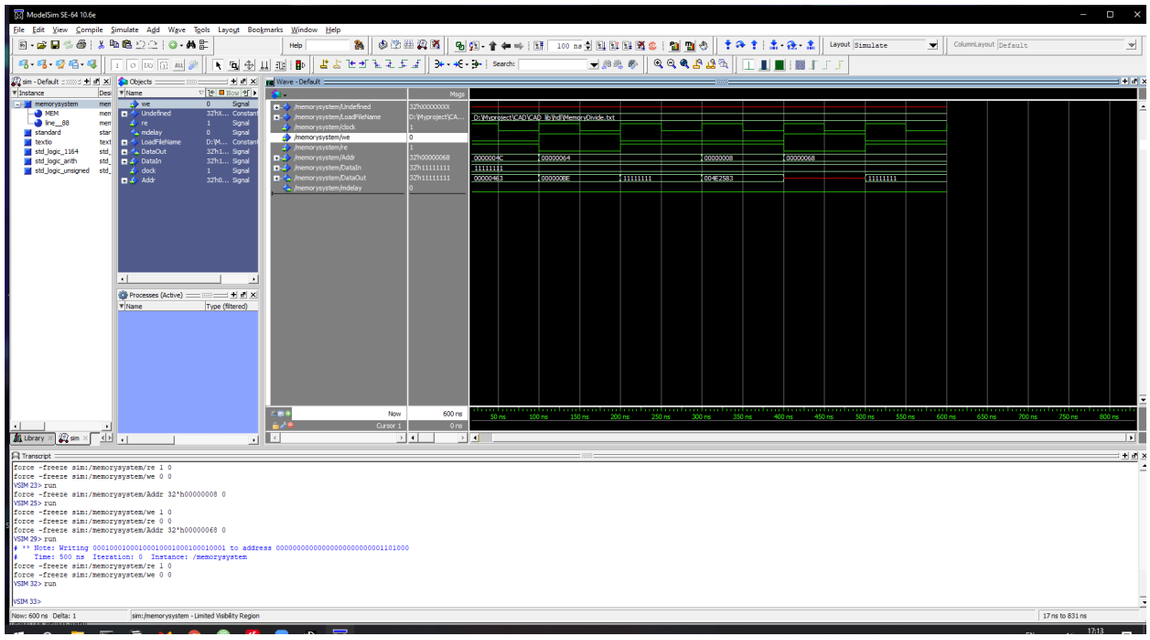


Figure 25: Memory Test

5.1 Narrative Description of Memory System and Tests

Here is a memory system design unit which was given by professor to test the processor. Basically it read out all the 32-bit binary instructions from the txt file MemoryDivide and and stored the data using an array. The instruction set was supposed to operate a division with multiple branches. The given VHDL code performed like a simulation of real memory system which can operate either one read or one write at each time cycle. Also, to avoid structure hazard, there will be a stall asserted (on either memory stage or fetch stage determined by the arbiter) from the memory system beginning to read or write until the operation is completed. Thus, to ensure the memory system is able to interact with my processor, here I tested the basic read and write operations of the memory system. I set the input data to be always 11111111(Hex). I read out the data of the address 0000004C(Hex) on the first time cycle. The memory system provided the correct output 00000463(Hex). Then I wrote the input data to the address 00000064(Hex), the modelsim then reported the log in transcript window as I excepted : *Note: Writing 00010001000100010001000100010001 to address 00000000000000000000000001101000.* On the next time cycle, I read the data in the address 00000064(Hex), the memory system provided the correct data (which was written by me on the last clock cycle) as 11111111(Hex). Then I sequentially read the address 00000008, wrote to the address 00000068, read the address 00000068. The results all met my expectation. The memory system was fully functional and was supposed to port to my processor successfully.

6 Final processor

Address (Hex)	Value (Hex)	Label	Instruction	Comment
0000 0000	0600 6E13		ORI x28,x0,060h	Register 28 = 0000 0060
0000 0004	000E 2503		LW x10,x28,0	Register 10 holds Divisor
0000 0008	004E 2583		LW x11,x28,4	Register 11 contains Dividend
0000 000C	0405 0263		BEQ x10,x0,BAD	Branch to BAD if Divisor = 0
0000 0010	04A5 C063		BLT x11,x10,BAD	Branch to BAD if Dividend < Divisor
0000 0014	0105 1513		SLLI x10,x10,16	Shift Divisor into high half-word
0000 0018	0100 6313		ORI x6,x0,16	Initialize counter to 16
0000 001C	0005 83B3	DLOOP	ADDr x7,x11,x0	Copy Dividend into x7
0000 0020	0015 9593		SLLI x11,x11,1	Shift Dividend left
0000 0024	01F3 D393		SRLI x7,x7,31	Shift copy of Dividend right 31 places
0000 0028	0003 9463		BNE x7,x0,SKIP	Branch if Dividend MSB = 1
0000 002C	00A5 C663		BLT x11,x10,NOSUB	Branch if Dividend less than Divisor
0000 0030	40A5 85B3	SKIP	SUBr x11,x11,x10	Dividend = Dividend - Divisor
0000 0034	0015 8593		ADDI x11,x11,1	Dividend = Dividend + 1
0000 0038	FFF3 0313	NOSUB	ADDI x6,x6,-1	Decrement x6
0000 003C	FE03 10E3		BNE x6,x0,DLOOP	Loop if not zero
0000 0040	FFFF 03B7		LUI x7,FFFF0h	
0000 0044	0103 D393		SRLI x7,X7,16	Register 7 holds 0000 FFFF
0000 0048	0075 F5B3		ANDr x11,x11,x7	Register 11 holds quotient
0000 004C	0000 0463		BEQ x0,x0,SAVE	Branch to save
0000 0050	00B5 C5B3	BAD	XORr x11,x11,x11	Set register 11 to zero
0000 0054	00BE 2423	SAVE	SW x11,x28,8	Save quotient
0000 0058	0000 0063	ENDLP	BEQ x0,x0,0	Wait Spin
0000 0060	0000 0013		-- DATA --	Divisor
0000 0064	0000 00BE		-- DATA --	Dividend
0000 0068	1010 1010		-- DATA --	Quotient

Figure 26: Divide operation program : 190/19

6.1 General Design Idea

Program to run As I mentioned above, the program to test represents a division operation of 190 / 19. So the simulation should be able to provide 10 as the correct answer.

Building up the entire processor is relatively easy now. Since I have all the individual components working well, all I need to do to apply all the entities and port them together in a single entity unit. Also, considering the memory system is not synthesizable, I ported all the components(fetch stage, decode stage, execute stage, memory stage, write back stage, arbiter, register file and register tracker) together with input clock and reset and other ports connected to memory system. After checking the synthesis of the design is correct, I implemented the processor in a new design entity and also imported the memory system entity to the design. Then I simulated the design in modelsim. I set up the clock and I set the reset to 1 on the first time cycle to reset the register in fetch stage and all my regtracker registers flag to 00. Then I set reset to 0 and kept simulating. The program ran the loop several time and finally provides the message Note: Writing 00000000000000000000000000001010 to address 00000000000000000000000001101000, which is the correct output we want for 190/19.

6.2 Processor test, simulation and synthesis

6.2.1 Statement that machine ran program

As shown in Figure 27, the machine can keep reading instructions from memory system. All instructions are performed correctly and the operation writes back to memory system (SW) is showed in the transcript window as we expected. (Write the result 10 to address 00000068)

6.2.2 Screenshots of machine running program

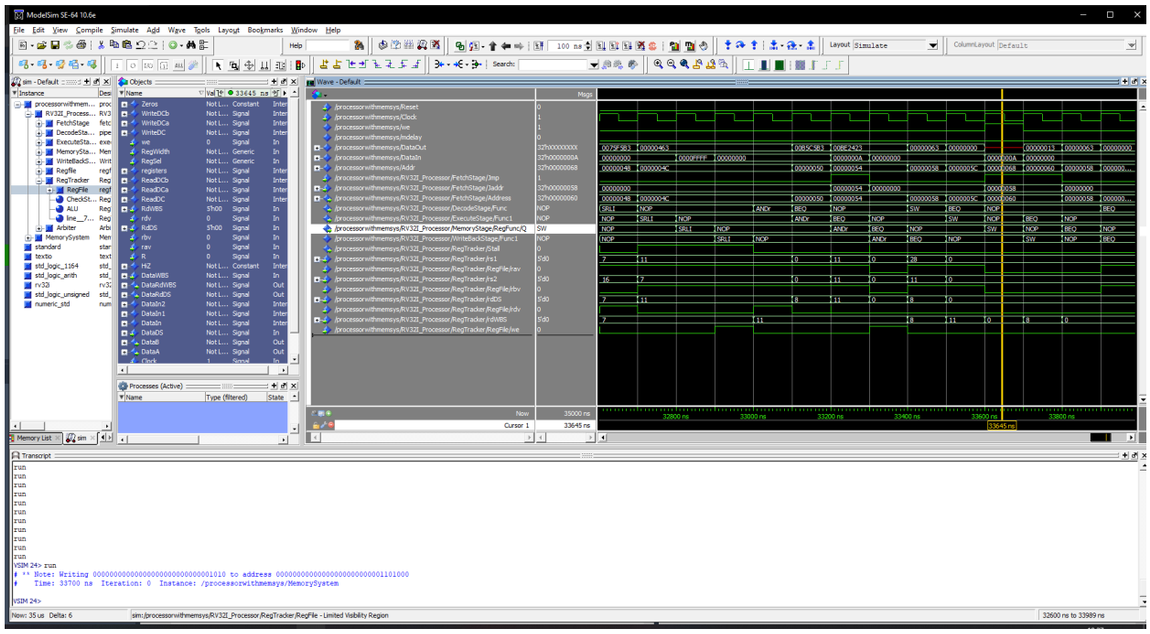


Figure 27: The final SW operation

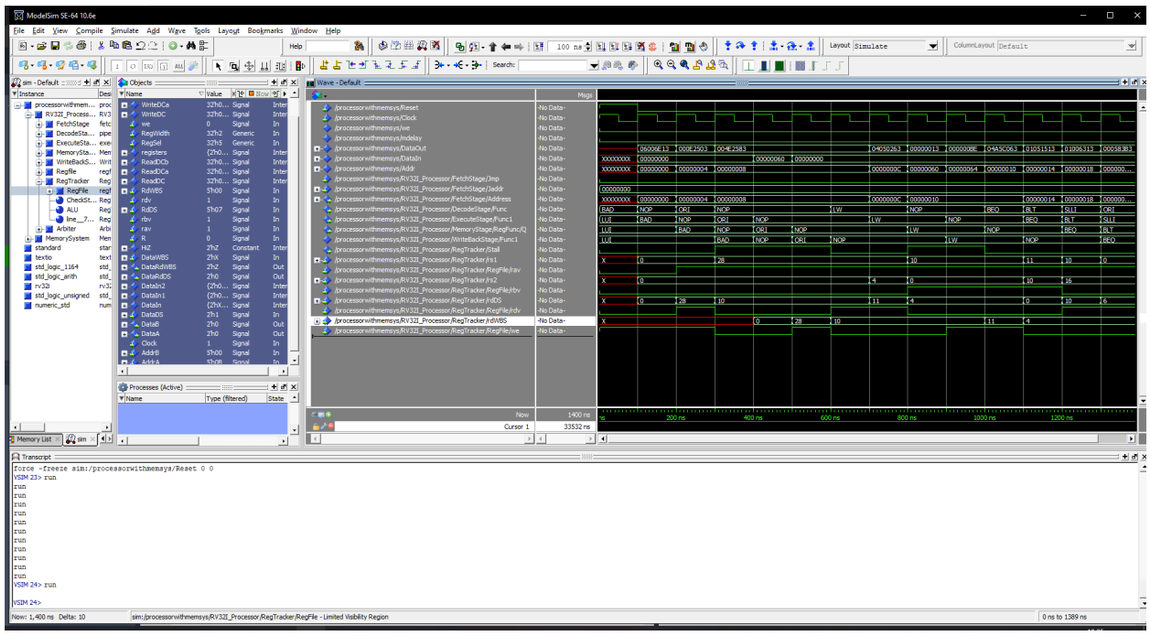


Figure 28: At beginning

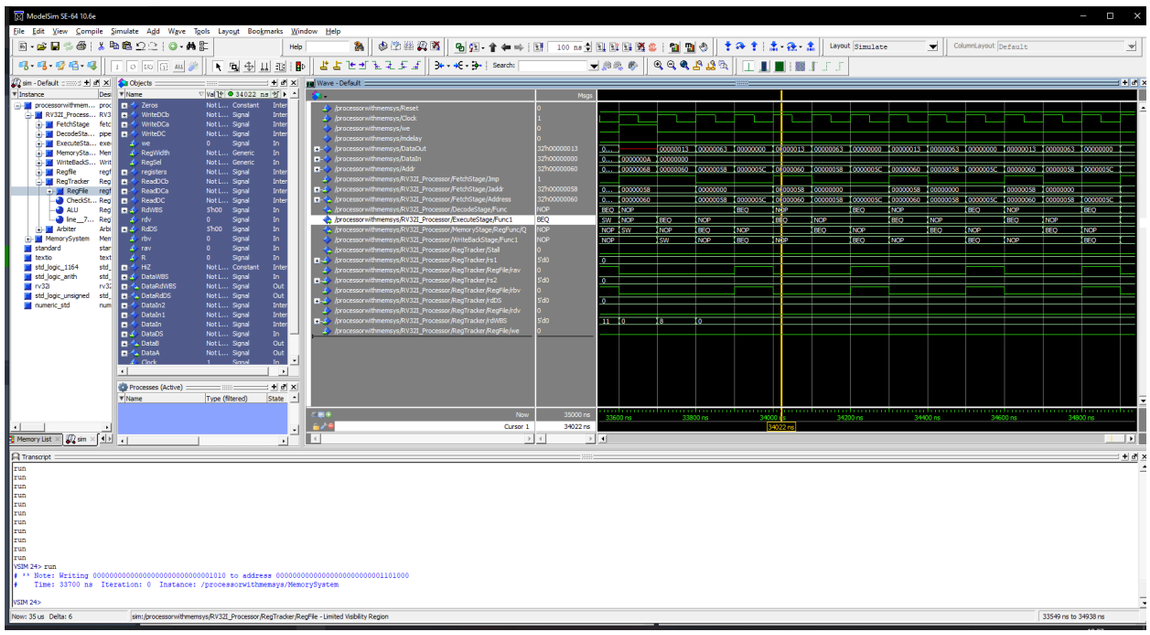


Figure 29: Wait spin after SW

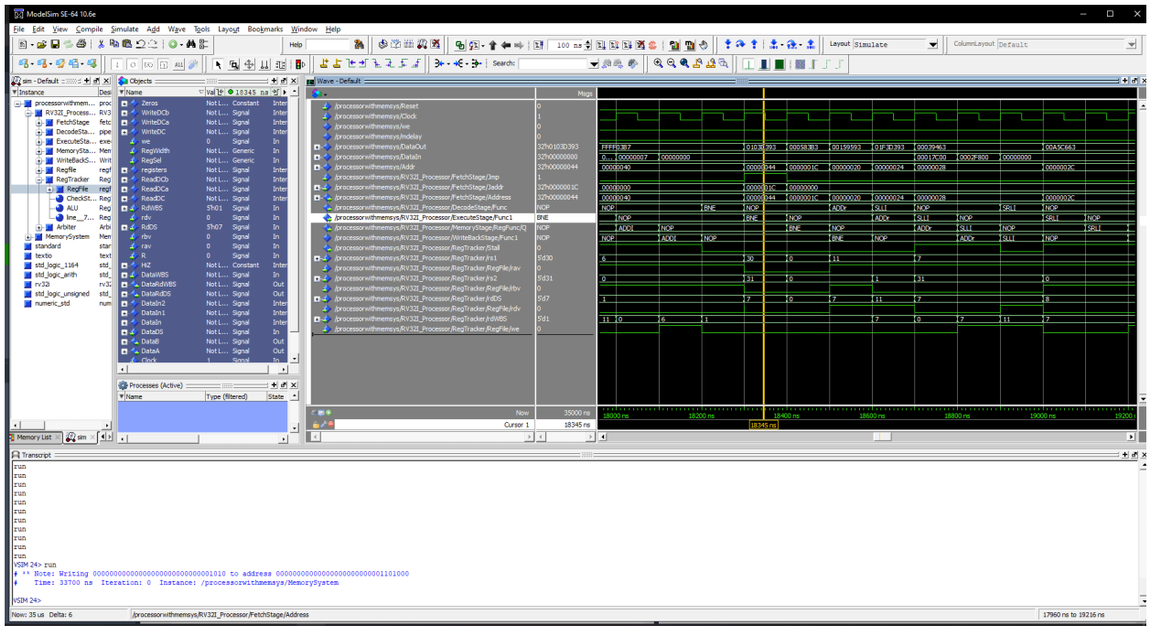


Figure 30: Taken branch

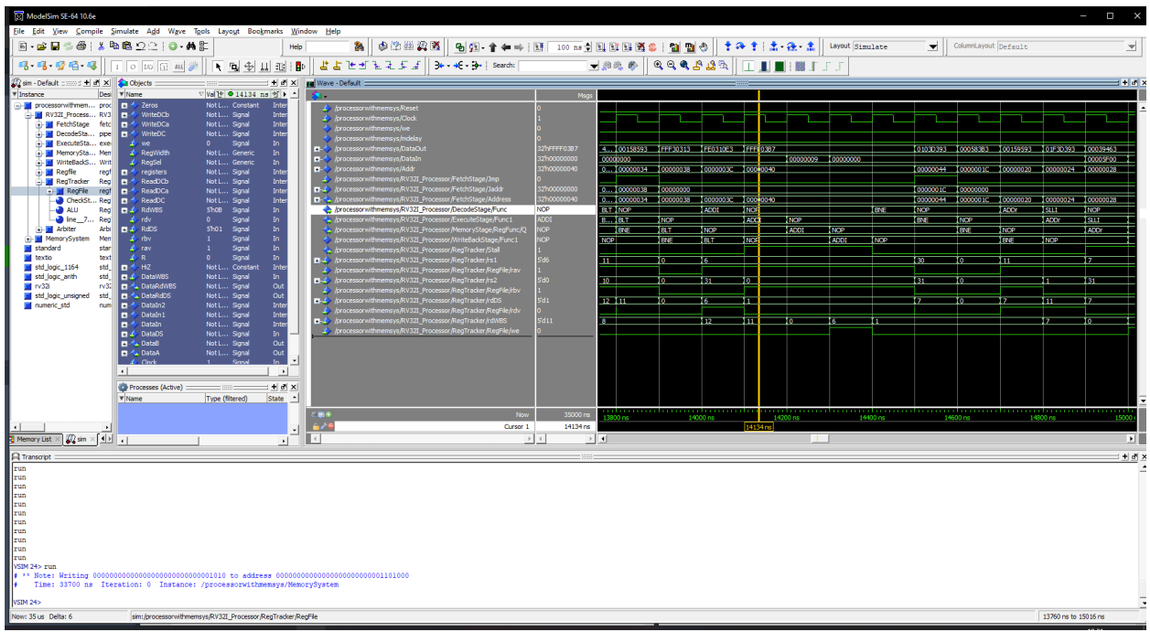


Figure 31: Resolving RAW hazard

6.2.3 Narrative explaining screenshots of machine running program

The whole program took 33700 ns (including initialization) to operate until the processor wrote the quotient back to memory system. Thus I just took five screenshots out of the whole simulation to exhibit some performance of my processor. The processor ported with memory system has two inputs(clock and reset) and can automatically run the given program after initialization.

Initialization As shown in Figure 28, I reset the system on the first clock cycle(reset the register that held the address in fetch stage, reset all the registers that held the 2-bit register reservation count in the register tracker to 00), and then set the reset signal to zero at the second clock cycle. The machine began to read instructions from the first address 00000000(Hex) and then added four to the address each time cycle to read the next address when there was no jump or stall asserted.

Resolving control hazards As shown in Figure 29, after the final SW operation, the next program instruction was BEQ x0,x0,0. The instruction jumped to its own address to perform a wait spin. Each time the branch BEQ x0,x0,0 was taken in execute stage, the 1-bit jump signal was set to 1, the instructions in the fetch stage and decode stage were replaced by NOP and the fetch stage jumped to the jump address 00000058(Hex) on the next time cycle. Another example of a taken branch (BNE x6,x0,DLOOP) in Figure 30 also exhibits the jump ability of the design.

Resolving RAW hazards As shown in Figure 31, there was an instruction ADDI x6,x6,-1 followed by BNE x6,x0, Dloop. Register x6 was reserved when ADDI was in decode stage. So there was a stall when BNE tried to read register x6 in decode stage on the next time cycle. The processor stalled the decode stage and fetch stage at this point. The fetch stage stopped reading the next instruction and the decode stage kept sending NOP to execute stage until the stall was resolved.

Final SW operation As shown in Figure 27, there was eventually a SW instruction in the pipeline which stored the final result of the division to memory system. In the transcript window, the data to be stored 00000000000000000000000000001010 is 10 in binary, which is the correct result of the division 190 / 19.

6.2.4 Screenshot of RTL of the complete processor (excluding memory) from synthesis

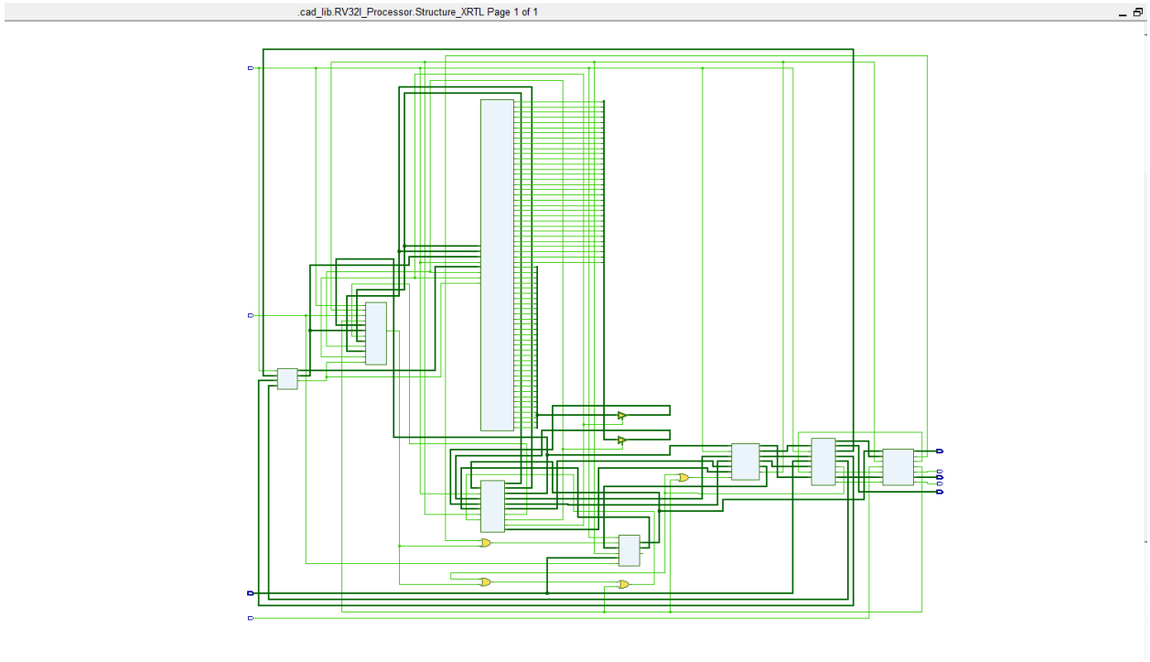


Figure 32: Processor Schematic

6.2.5 Screenshot of the area report using Spartan 6 target

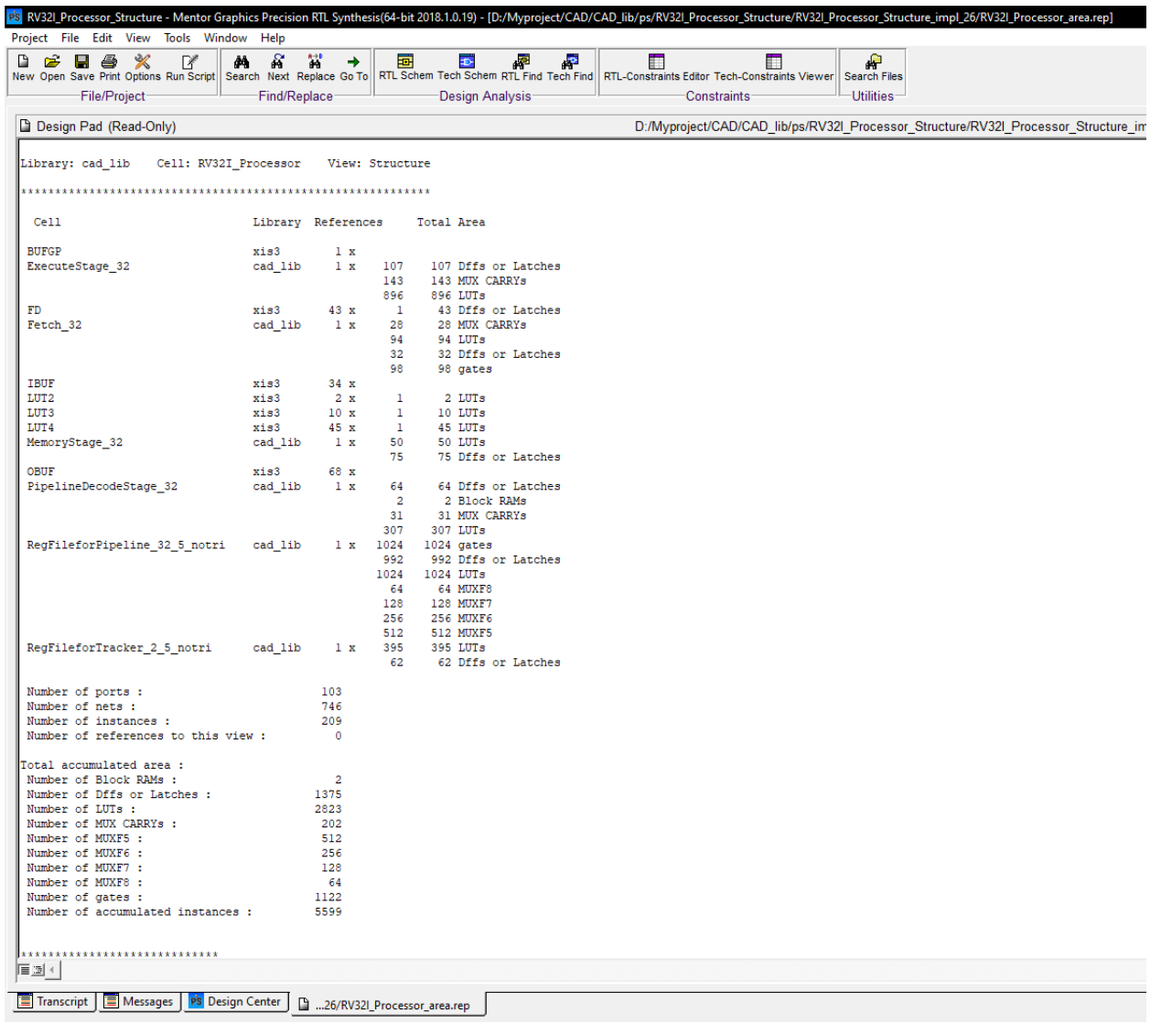


Figure 33: Area Report

As I mentioned above, I ported all the components(fetch stage, decode stage, execute stage, memory stage, write back stage, arbiter, register file and register tracker) together with input clock

and reset and other ports which will link to memory system later. The schematic looked well-organized and clean since I was just using the previous components I design without additional code. The report area showed as 1375 Dffs or Latches shown in Figure 33, which met our design expectation.

Register File : 32 x 31 = 992 (Register 0 always returns 0s)

Register Tracker : 2 x 31 = 62 (Register 0 cannot be reserved)

Fetch Stage : 32 (Address)

Decode Stage : 32 x 2 = 64 (Address, Instruction)

Execute Stage : 32 + 32 + 32 + 5 + 6 = 107 (Left, Right, Extra, Rd, Enumerated 6 bits signal representing Function)

Memory Stage : 32 + 32 + 5 + 6 = 75 (Address, Data, Rd, 6-bit Function)

Write Back Stage : 32 + 5 + 6 = 43 (Data, Rd, 6-bit Function)

Arbiter : 0 (Totally combinational)

Total : 992 + 62 + 32 + 64 + 107 + 75 + 43 = 1375

6.2.6 Creative elements of my design

I used a register tracker with 32 2-bit registers to hold the reservation count for each register in the register file, which can resolve RAW data hazards with less stalls comparing to a register tracker with 32 1-bit registers. The tracker set a 2-bit flag number for each register in the register file. When the instruction was decoded as a function that needs to write back to a register in the future, the register tracker read out the specific register flag, incremented the flag by one and wrote back to the register that held the reservation count. When the write back stage executed a write back, the register tracker read out the specific register flag, subtracted it by one and wrote back to the register that held the register flag. We also need to consider a special case that when decode stage and write back stage are manipulating the flag of the same register, then the register flag should remain unchanged. This method enables the pipeline's capability to have consecutive (or more than one in each three) instructions that write back to the same register, while a register tracker with 32 1-bit registers doesn't have such ability to count the register reservation times will lead to a stall in such cases.

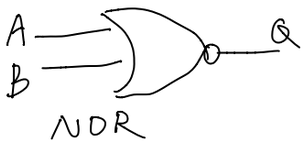
There was this condition that the memory stage was operating a store and the execute stage was operating a jump (taken branch) at the same time. I set the fetch stage to stall whenever the memory stage was using the memory system to resolve structure hazards. However, all things worked well and it turned out the jump address would be lost on the next time cycle. Since this is a rare situation(a store or load in memory stage and a jump in execute stage at the same time), it is not necessary to add another register to hold the jump address. Thus I set up an additional stall signal for this. I added an input jump and an output additional stall to memory arbiter. The arbiter would set the additional stall to one when the memory stage was using the memory system and the jump was asserted the same time. Then I ported the additional stall to decode stage and execute stage to make them stall as well as fetch stage for this situation. The result turned out to be good. All the stages before memory stage was stall when SW was in memory stage and the fetch stage jumped to the jump address 00000058(Hex) on the following time cycle. (As shown in Figure 27 and Figure 29)

6.2.7 Particularly challenging parts of the project

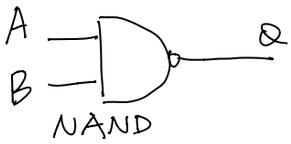
Although the register tracker was fully functional when I tested it using a testbench, there were some situations that I forgot to consider. So when I tested the final processor with the memory system, the program went into deadlock several times because of the register tracker. For example, I forgot to consider the current state of the decode stage when the register tracker incremented the reservation times of a register. The register kept incrementing the reservation number when the decode stage was already stalled. After the write back stage released the reservation flag of the register by one, the flag was still not zero which caused a deadlock. Also, I forgot to cancel the register reservation in decode stage when there was a jump asserted in execute stage, which also leads to a deadlock later. There were so many signals in the final simulation, so it took me really

long time to debug. It was much more difficult to find the cause of a problem in the final test than in the previous tests for each components.

7 Additional study on latches

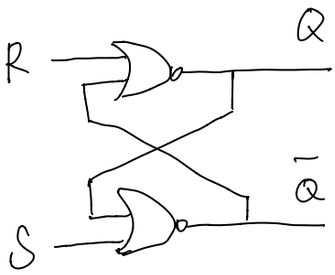


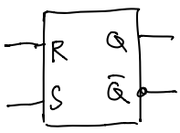
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0



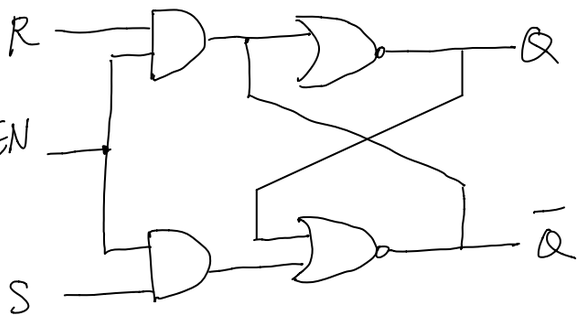
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

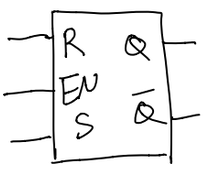
SR Latch



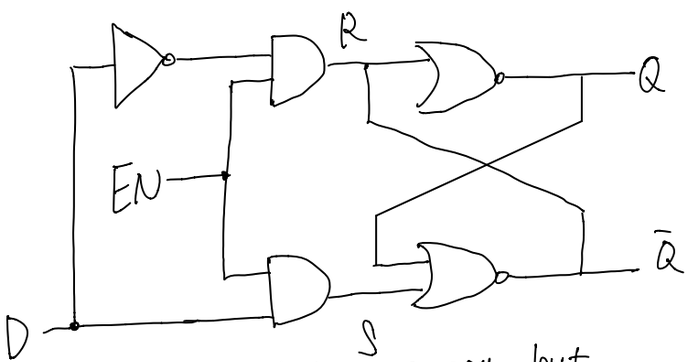


SR Latch with enable.





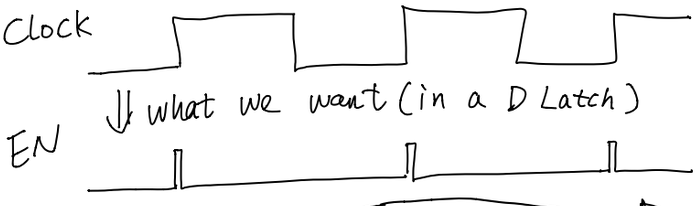
D Latch



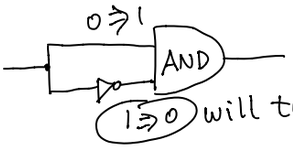
It is one bit memory but it is not latching!

D flip flop (to have a clock instead of enable)

How to change the value of latch only at rising edge of a clock?



We will need an **edge detector**!

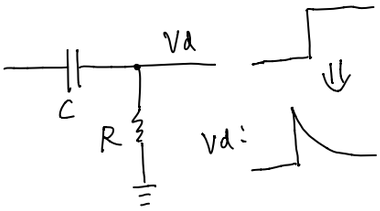
(A) 

will take more time!

So there is a moment when **1 AND 1 ⇒ EN=1**

In Comp Arch, the amount of time the inverter needs to perform is the aperture time of clock!

(B) Another way (simpler)



But we do need to use much more inverters to get enough aperture time to satisfy Comp Arch Discipline.

So by manipulating R and C, we can easily control the aperture time of clock!