CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

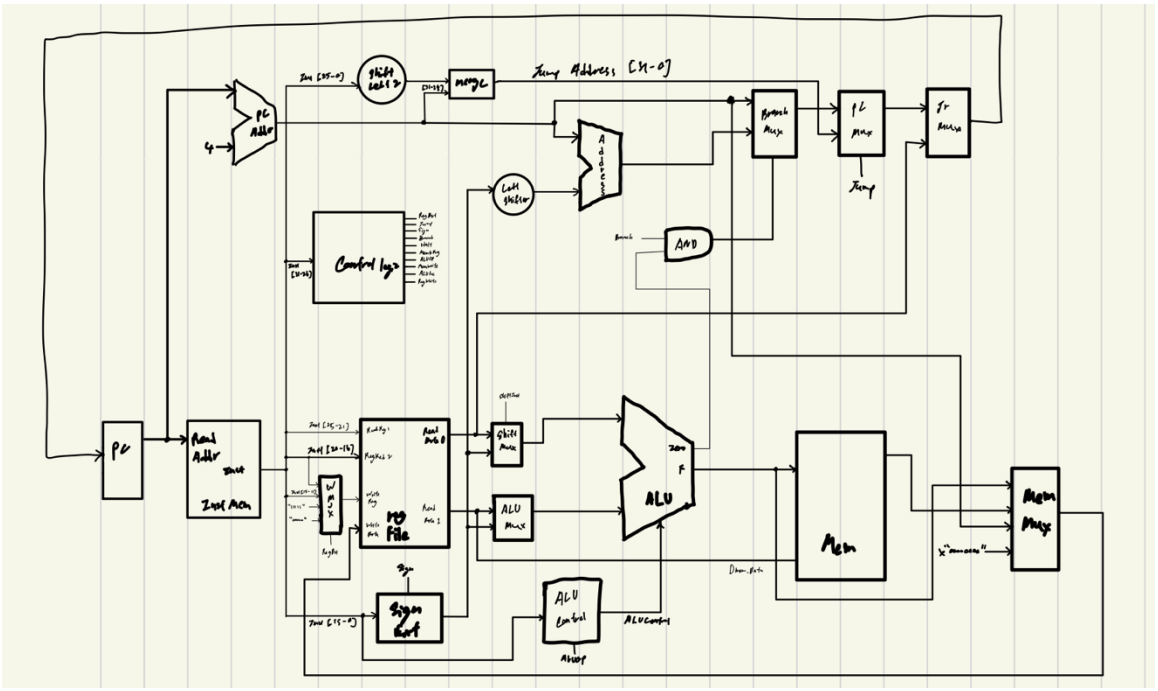
Team Members: Kai Heng Gan

Brian McCreary

Project Teams Group #: TermProj_2_04

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N*M* table where each row corresponds to the output of the control logic module for a given instruction.

Instruction	Opcode (Binary)	Funct (Binary)	Control Signals										
			RegDst	Jump	Sign	Branch	Halt	MemtoReg	ALUControl	s_DMemWr	ALUSrc	RegWrite	Function
									R-type				
add	000000	100000	01	0	0	0	0	00	0000	0	0	1	ADD
addu	000000	100001	01	0	0	0	0	00	0000	0	0	1	ADD
and	000000	100100	01	0	0	0	0	00	0000	0	0	1	AND
nor	000000	100111	01	0	0	0	0	00	0000	0	0	1	NOR
xor	000000	100110	01	0	0	0	0	00	0000	0	0	1	XOR
or	000000	100101	01	0	0	0	0	00	0000	0	0	1	OR
slt	000000	101010	01	0	0	0	0	00	0000	0	0	1	SET LESS THAN
sll	000000	000000	01	0	0	0	0	00	0000	0	0	1	SHIFT LEFT
srl	000000	000010	01	0	0	0	0	00	0000	0	0	1	SHIFT RIGHT
sra	000000	000011	01	0	0	0	0	00	0000	0	0	1	SHIFT RIGHT
sub	000000	100010	01	0	0	0	0	00	0000	0	0	1	SUB
subu	000000	100011	01	0	0	0	0	00	0000	0	0	1	SUB
jr	000000	001000	01	0	0	0	0	00	0000	0	0	1	JUMP
									I-type				
addi	"001000"	"------"	00	0	1	0	0	00	0001	0	1	1	ADD
addiu	001001	"------"	00	0	1	0	0	00	0001	0	1	1	ADD
lw	100011	"------"	00	0	1	0	0	01	0001	0	1	1	LOAD
sw	101011	"------"	00	0	1	0	0	00	0001	1	1	0	STORE
andi	001100	"------"	00	0	0	0	0	00	0001	0	1	1	AND
ori	001101	"------"	00	0	0	0	0	00	0011	0	1	1	OR
xori	001110	"------"	00	0	0	0	0	00	0100	0	1	1	XOR
slti	001010	"------"	00	0	1	0	0	00	1011	0	0	1	SET LESS THAN
lui	001111	"------"	00	0	0	0	0	00	1100	0	1	1	LOAD
									Branch				
bgez	011100	"------"	00	0	1	1	0	00	0101	0	0	0	BRANCH GREATER THAN EQUAL ZERO
bgezal	011101	"------"	10	0	1	1	0	10	0101	0	1	1	BRANCH GREATER THAN EQUAL ZERO
bgtz	000111	"------"	00	0	1	1	0	00	0110	0	0	0	BRANCH GREATER THAN ZERO
blez	000110	"------"	00	0	1	1	0	00	0111	0	0	0	BRANCH LESS THAN EQUAL ZERO
beq	000100	"------"	00	0	1	1	0	00	1000	0	0	0	SUB
bne	000101	"------"	00	0	1	1	0	00	1001	0	0	0	SUB
bltz	011110	"------"	00	0	1	1	0	00	1010	0	0	0	BRANCH LESS THAN ZERO
bltzal	011111	"------"	10	0	1	1	0	10	1010	0	1	1	BRANCH LESS THAN ZERO
									Jump				
j	000010	"------"	00	1	0	0	0	00	1111	0	0	0	JUMP
jal	000011	"------"	10	1	0	0	0	10	1111	0	0	1	JUMP

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).

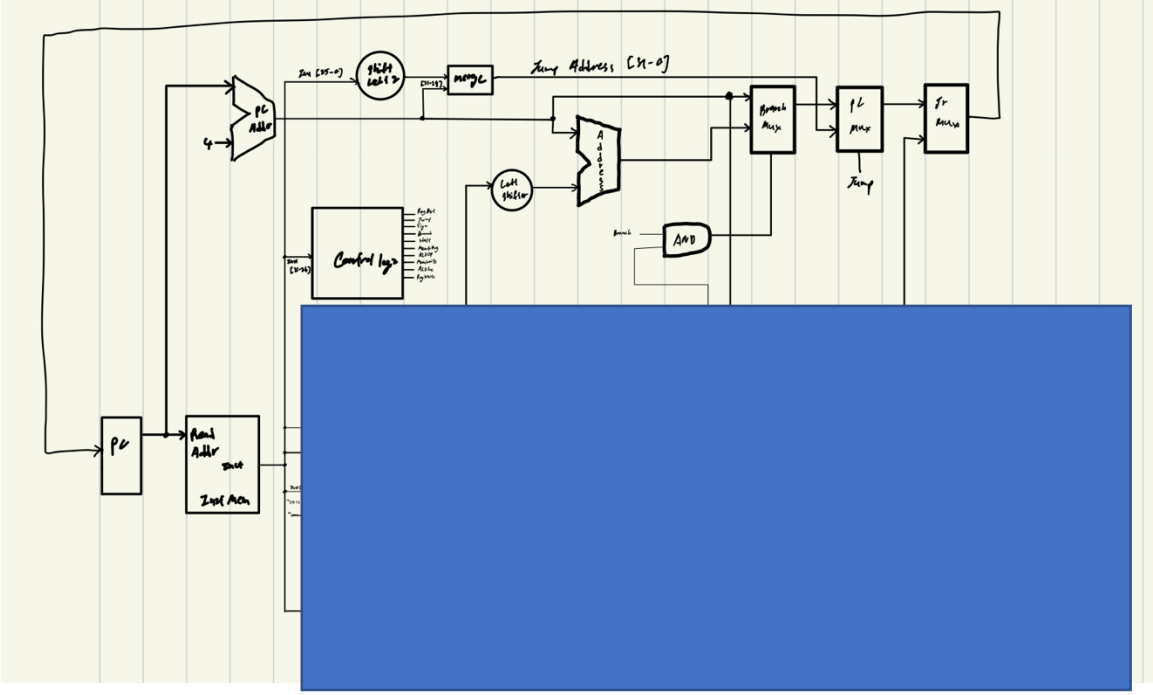
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

The control flow possibilities that my instruction fetch logic must support is addi, add, addiu, addu, and, andi, lui, lw, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sw, sub, subu, beq, bne, j, jal, jr, bgez, bgezal, bgtz, blez, bltz, bltzal and halt.

If the opcode is 000000, which means it is a R-type instruction. The Control component will execute the assigned ALUOp value to ALU Control component depends on the funct bit field value. Besides that, the Control component will also execute the assigned RegDst, Jump, Branch, Halt, MemtoReg, MemWrite, ALUSrc, and RegWrite signals to specified components.

Every J-type and I-type instruction has their unique opcode. The Control component will execute the assigned ALUOp value, RegDst, Jump, Branch, Halt, MemtoReg, MemWrite, ALUSrc, and RegWrite signals according to the fetched instruction.

[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



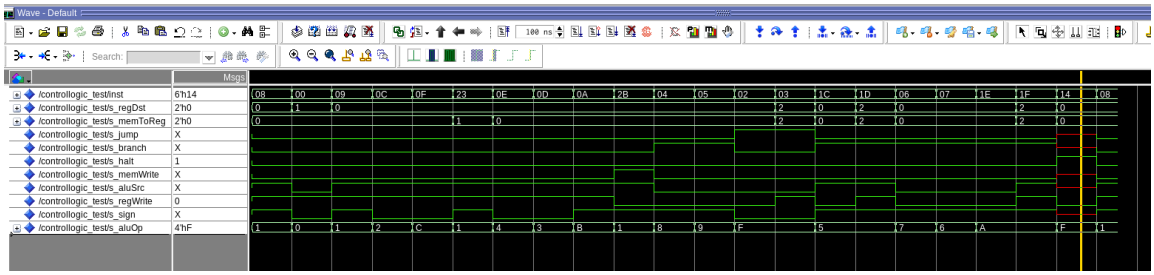
s_jumpReg - This signal is used to choose the PCMuxOutput or RsData for JrMux. It is used as a selector in JrMux.

Jump - This signal is used to choose between BranchMuxOutput or JumpAddress. It is used as a selector in PCMux.

BranchIsTrue - This signal is used to choose between AddressAdderOutput or PCAdderOutput. It is used as a selector in BranchMux. The BranchIsTrue is set to 1 when the branch condition was satisfied, Zero is 1, and the Branch signal is set to 1 due to the features of AND gate.

I added a merge component to compute the jump address. It takes the output of ShiftLeft2 and the PCAdder[31-28]. The merge component will first convert 4-bit PCAdderOutput[31-28] and 28-bits shiftLeft2Output to 32 bits. Then, it will use OR to merge them together and become jump address. The PCAdderOutput[31-28] will be the most four significant bits in the jump address.

[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.



Opcode: 001000, 001001 (addi, addiu),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 1
aluOp - 0001

Opcode: 000000 (R-type),
regDst - 01
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 1
sign - 0
aluOp - 0000

Opcode: 001001 (addiu),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 1
aluOp - 0001

Opcode: 001100 (andi),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 0
aluOp - 0010

Opcode: 001111 (lui),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 0
aluOp - 1100

Opcode: 100011 (lw),
regDst - 00
memToReg - 01
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 1
aluOp - 0001

Opcode: 001110 (xori),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1

sign - 0
aluOp - 0100

Opcode: 001101 (ori),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 0
aluOp - 0011

Opcode: 001010 (slti),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 1
aluOp - 1011

Opcode: 101011 (sw),
regDst - 00
memToReg - 00
jump - 0
branch - 0
halt - 0
memWrite - 1
aluSrc - 1
regWrite - 0
sign - 1
aluOp - 0001

Opcode: 000100 (beq),
regDst - 00
memToReg - 00
jump - 0
branch - 1
halt - 0
memWrite - 0

aluSrc - 0
regWrite - 0
sign - 1
aluOp - 1000

Opcode: 000101 (bne),
regDst - 00
memToReg - 00
jump - 0
branch - 1
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 0
sign - 1
aluOp - 1001

Opcode: 000010 (j),
regDst - 00
memToReg - 00
jump - 1
branch - 0
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 0
sign - 0
aluOp - 1111

Opcode: 000011 (jal),
regDst - 10
memToReg - 10
jump - 1
branch - 0
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 1
sign - 0
aluOp - 1111

Opcode: 011100 (bltz),
regDst - 00
memToReg - 00
jump - 0
branch - 1

halt - 0
memWrite - 0
aluSrc - 1
regWrite - 0
sign - 1
aluOp - 0101

Opcode: 011101 (bgezal),
regDst - 10
memToReg - 10
jump - 0
branch - 1
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 1
aluOp - 0101

Opcode: 000110 (blez),
regDst - 00
memToReg - 00
jump - 0
branch - 1
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 0
sign - 1
aluOp - 0111

Opcode: 000111 (bgtz),
regDst - 00
memToReg - 00
jump - 0
branch - 1
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 0
sign - 1
aluOp - 0110

Opcode: 011110 (bltz),
regDst - 00
memToReg - 00

jump - 0
branch - 1
halt - 0
memWrite - 0
aluSrc - 0
regWrite - 0
sign - 1
aluOp - 1010

Opcode: 011111 (bltzal),
regDst - 10
memToReg - 10
jump - 0
branch - 1
halt - 0
memWrite - 0
aluSrc - 1
regWrite - 1
sign - 1
aluOp - 1010

Opcode: 010100 (Halt),
regDst - 00
memToReg - 00
jump - X
branch - X
halt - 1
memWrite - X
aluSrc - X
regWrite - 0
sign - X
aluOp - 1111

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

`srl - This instruction will shift the value right logically. It will ignore the most significant bit of the data by adding 0 to the most significant bit.`

`sra - This instruction will shift the value right arithmetic. It will keep the most significant bit value and it is shifting the value. For example, "1110" shift right 1 time, the result will be "1111".`

The `sla` instruction does not exist in MIPS because we determine a signed value through the most significant bit.

[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

I declared two variables, DIR & MODE, which are used to determines the shifting direction and the shifting mode. DIR = 0 represents direction right; DIR = 1 represents direction left. MODE = 0 represents shift logically; MODE = 1 represents shift arithmetic. Then, I use the if conditional statement to implement the shifting operations according to the shifting direction and mode. If the DIR is 0 and MODE is 0, the program will first assign the value of DATA(k + shamt) into regNew(k), a 32-bits signal, through for-loop statement. If (k+shamt > 31), the program will exit the loop. There's another loop to assign the 0s into the leftmost bits that haven't assigned yet.

If the DIR is 0 and MODE is 1, the program will first assign the value of DATA(k + shamt) into regNew(k), a 32-bits signal, through for-loop statement. If (k+shamt > 31), the program will exit the loop. There's another loop to assign the most significant bit value of the DATA into the leftmost bits that haven't assigned yet.

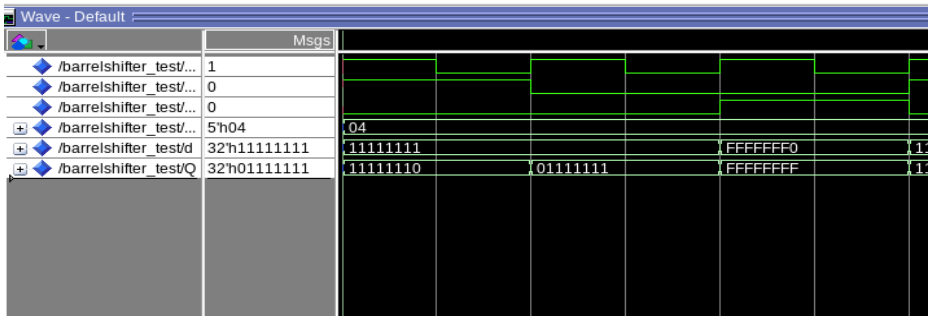
If the DIR is 1, the program will first assign the value of DATA(k-shamt) into regNew(k), a 32-bits signal, through for-loop statement (k starts at 31 downto 0). There's another loop assign the 0s into the rightmost bits that haven't assigned yet.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

Use two variables, DIR and MODE, to determine the shifting direction and mode.

If the DIR is 1, the program will first assign the value of DATA(k-shamt) into regNew(k), a 32-bits signal, through for-loop statement (k starts at 31 downto 0). There's another loop assign the 0s into the rightmost bits that haven't assigned yet.

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



At the beginning, the test case was doing sll. The program shift left 4 bits of 0x11111111 logically. As a result, the output was 0x11111110.

At the second clock cycle, the test case was doing srl. The program shift right 4 bits of 0x11111111 logically. As a result, the output was 0x01111111.

At the third clock cycle, the test case was doing sra. The program shift right 4 bits of 0xFFFFFFF0 arithmetically. As a result, the output was 0xFFFFFFFF.

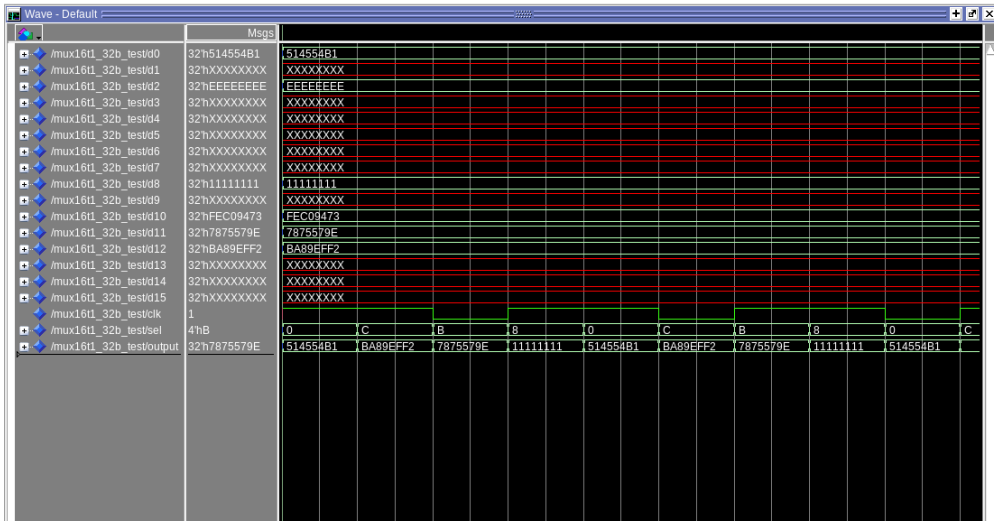
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

In the ALU component, I declared 3 input signals (A, B & ALUOp) and 4 (F, Cout, Overflow & Zero) output signals. Besides that, the ALU component is using seven components to compute the output, addSub_ALU, andg2_N, org2_N, xorg2_N, norg2_N, barrelShifter, and mux16t1_32b.

The addSub_ALU is used to compute the output of the instruction that related to add, sub, slt, and branch instruction.
The andg2_N, org2_N, xorg2_N, and norg2_N are used to compute the output of the instruction related to and, or, xor, and nor, respectively.
The barrelShifter is used to compute the output of sra, srl, and sll.
The mux16t1_32b is used to select which output is going to use as the ALU output, F, according to the ALUOp.
Besides that, there are if-conditional statements that will compute the output signals used as Cout, Overflow and Zero.

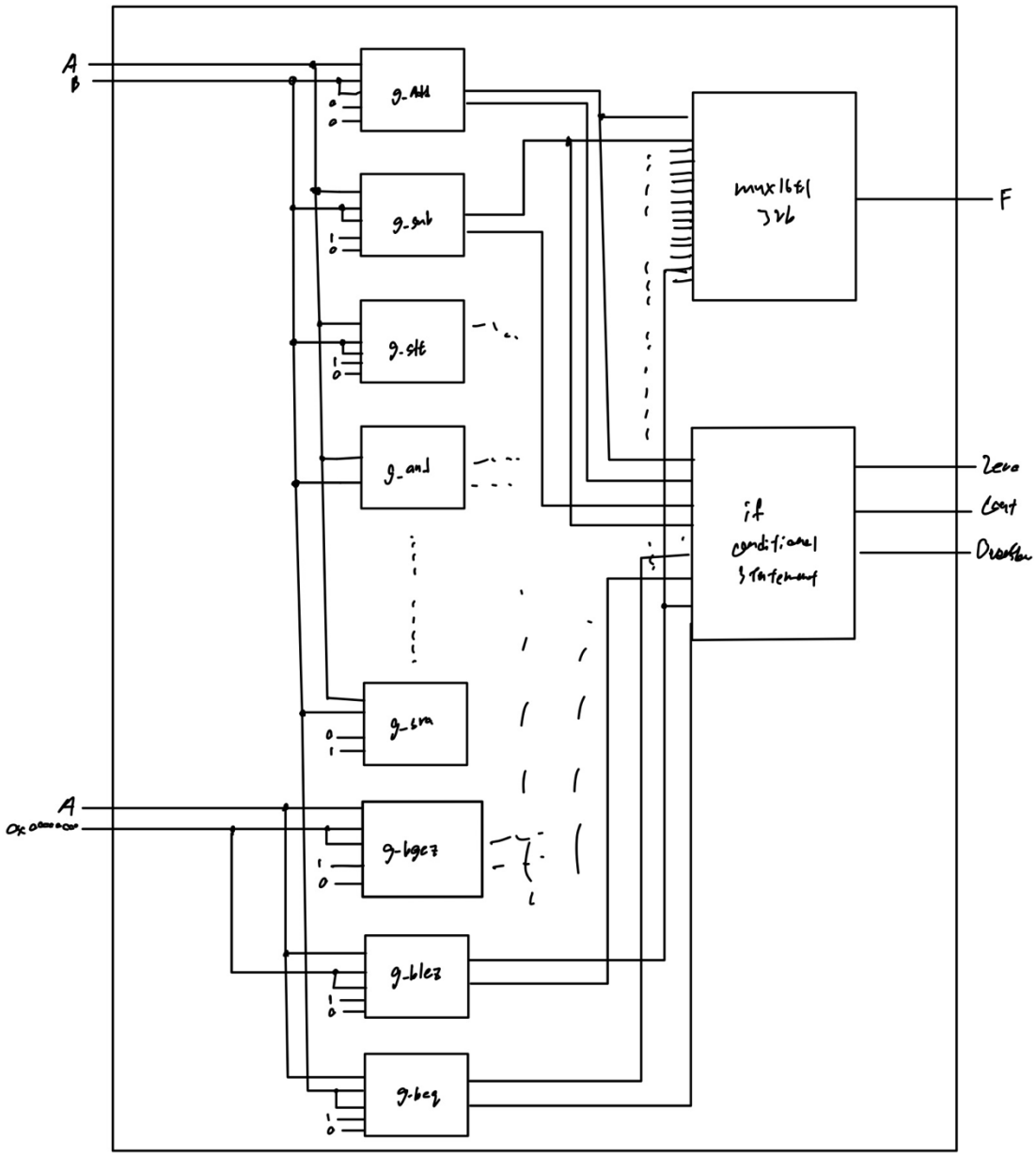
One of my additional components that I created is mux16t1_32b. It is used to select an output as the ALU component output between 16 32-bits inputs.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.



When the sel signal is 0, it selects the value of d0 as the output.
When the sel signal is C, it selects the value of d12 as the output.
When the sel signal is B, it selects the value of d11 as the output.
When the sel signal is 8, it selects the value of d8 as the output.
Therefore, the outputs showed in the waveform are correct.

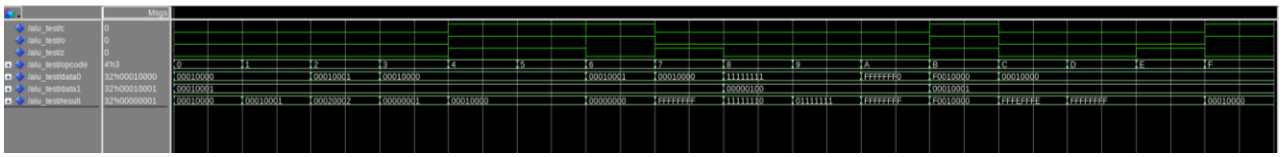
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



The Overflow is the last carry out from the addSub_ALU component. I use if-conditional statement to calculate out the Zero by setting the expression. For example, bne instruction, I use subtract the two values by using addSub_ALU. If the output of addSub_ALU is

0x00000000, which means the two values are equal and I will assign Zero to 1 in the conditional statement.
For the slt instruction, I also used the addSub_ALU to subtract the two values. If the output of the addSub_ALU is negative value, which means the first value is smaller than the second value. I use if-conditional statement to check the most significant bit of the output of the addSub_ALU. If the most significant bit is 1, which means the output is a negative value.

[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.



For ALUOP (0000), the test case was doing AND instruction, the output of 0x00010000 & 0x00010001 is 0x00010000. The Cout, Overflow, and Zero are 0,0,0 respectively. The outputs are correct.

For ALUOP (0001), the test case was doing OR instruction, the output of 0x00010000 & 0x00010001 is 0x00010001. The Cout, Overflow, and Zero are 0,0,0 respectively. The outputs are correct.

For ALUOP (0010), the test case was doing ADD instruction, the output of 0x00010001 & 0x00010001 is 0x00020002. The Cout, Overflow, and Zero are 0,0,0 respectively. The outputs are correct.

For ALUOP (0011), the test case was doing XOR instruction, the output of 0x00010000 & 0x00010001 is 0x00000001. The Cout, Overflow, and Zero are 0,0,0 respectively. The outputs are correct.

For ALUOP (0100), the test case was doing BGEZ instruction, the output of 0x00010000 & 0x00010001 is 0x00010000. The Cout, Overflow, and Zero are 1, 1, 1 respectively. The outputs are correct.

For ALUOP (0101), the test case was doing BGTZ instruction, the output of 0x00010000 & 0x00010001 is 0x00010000. The Cout, Overflow, and Zero are 1, 1, 1 respectively. The outputs are correct.

For ALUOP (0110), the test case was doing SUB instruction, the output of 0x00010001 & 0x00010001 is 0x00000000. The Cout, Overflow, and Zero are 1, 1, 1 respectively. The outputs are correct.

For ALUOP (0111), the test case was doing SLT instruction, the output of 0x00010000 & 0x00010001 is 0xFFFFFFFF. The Cout, Overflow, and Zero are 0, 0, 1 respectively. The outputs are correct.

For ALUOP (1000), the test case was doing SLL instruction, the output of 0x11111111 & 0x00000100 is 0x11111110. The Cout, Overflow, and Zero are 0, 0, 0 respectively. The outputs are correct.

For ALUOP (1001), the test case was doing SRL instruction, the output of 0x11111111 & 0x00000100 is 0x01111111. The Cout, Overflow, and Zero are 0, 0, 0 respectively. The outputs are correct.

For ALUOP (1010), the test case was doing SRA instruction, the output of 0xFFFFFFF0 & 0x00010001 is 0xFFFFFFFF. The Cout, Overflow, and Zero are 0, 0, 0 respectively. The outputs are correct.

For ALUOP (1011), the test case was doing BLEZ instruction, the output of 0x00010000 & 0x00010001 is 0x00010000. The Cout, Overflow, and Zero are 1, 1, 1 respectively. The outputs are correct.

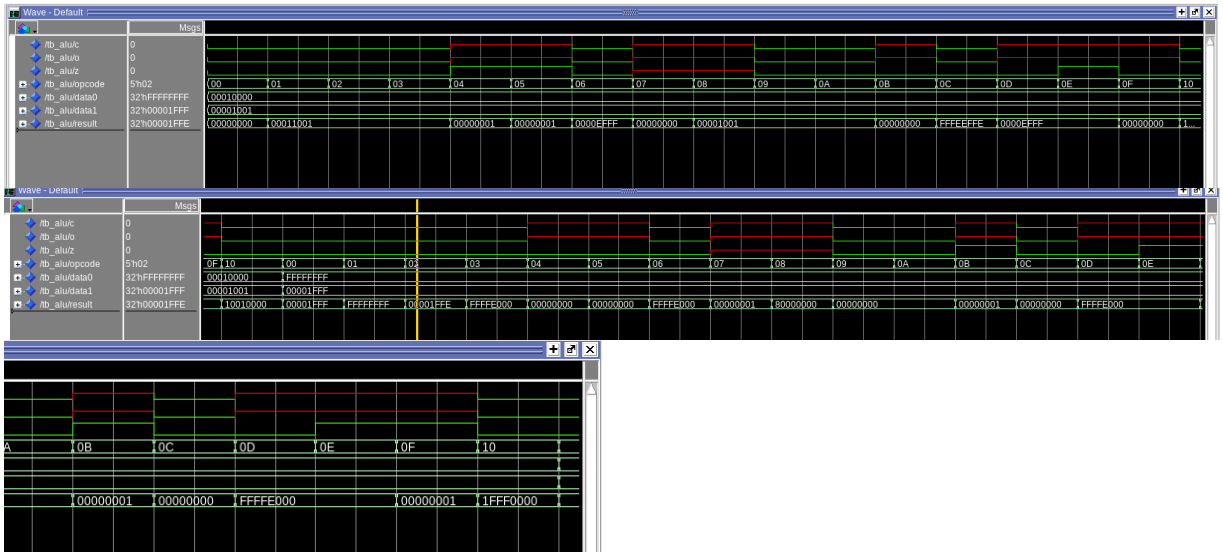
For ALUOP (1100), the test case was doing NOR instruction, the output of 0x00010000 & 0x00010001 is 0xFFFEFFFE. The Cout, Overflow, and Zero are 0, 0, 0 respectively. The outputs are correct.

For ALUOP (1101), the test case was doing BEQ instruction, the output of 0x00010000 & 0x00010001 is 0xFFFFFFFF. The Cout, Overflow, and Zero are 0, 0, 0 respectively. The outputs are correct.

For ALUOP (1110), the test case was doing BNE instruction, the output of 0x00010000 & 0x00010001 is 0xFFFFFFFF. The Cout, Overflow, and Zero are 0, 0, 1 respectively. The outputs are correct.

For ALUOP (1111), the test case was doing BLTZ instruction, the output of 0x00010000 & 0x00010001 is 0x00010000. The Cout, Overflow, and Zero are 1, 1, 0 respectively. The outputs are correct.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

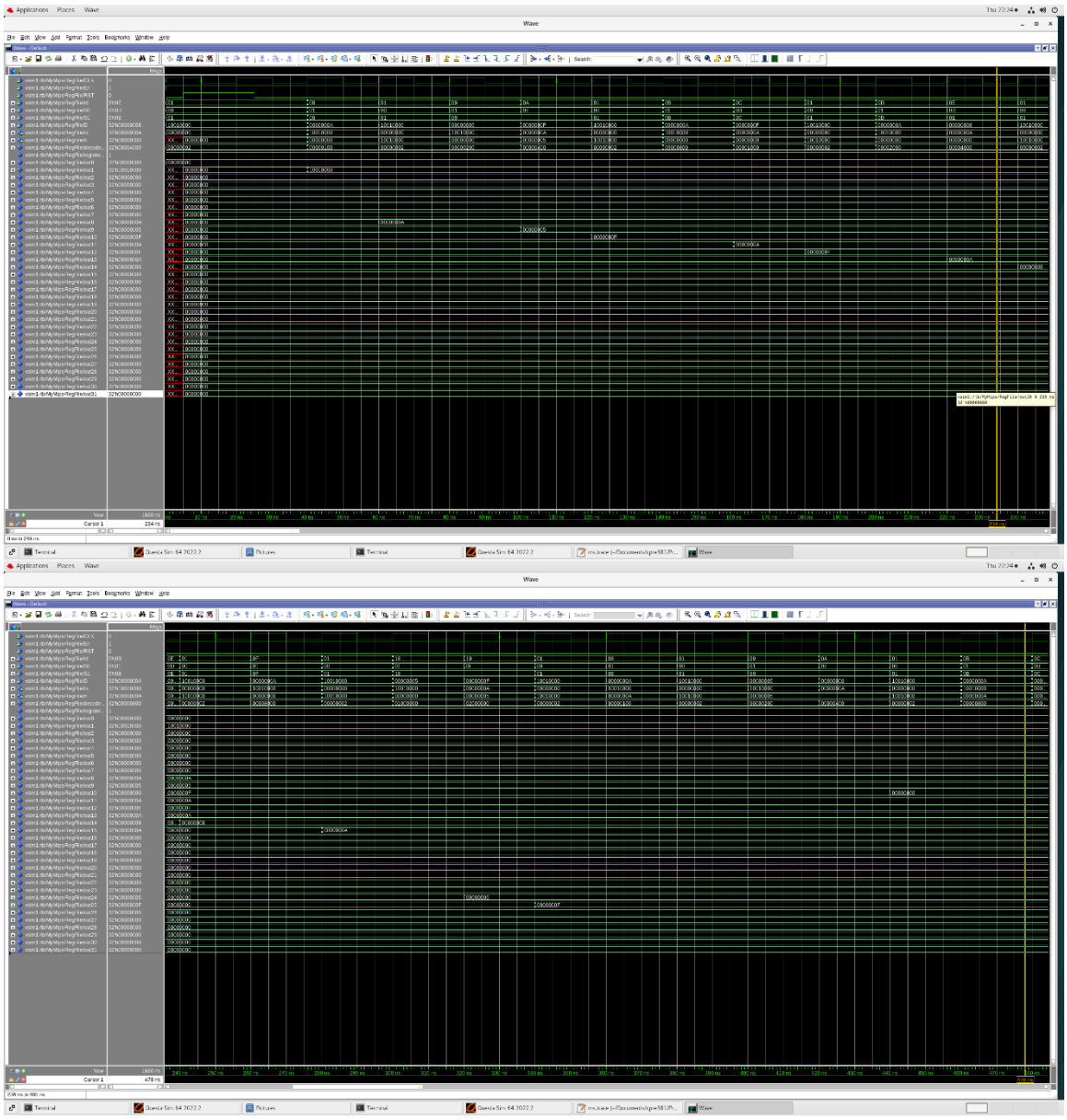


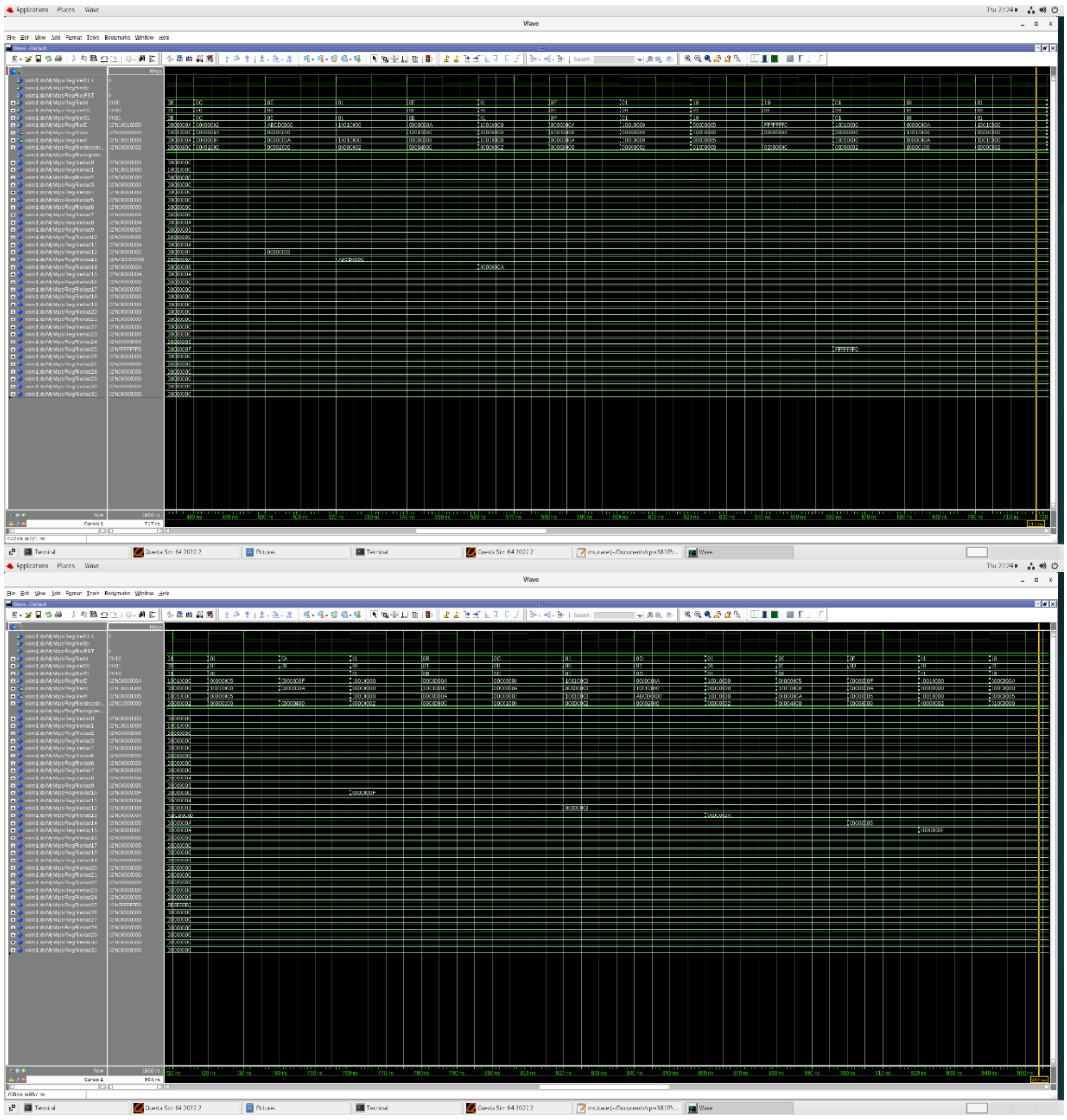
In this test bench, I was using two sets of different data. The first set is 0x00010000 and 0x00001001. The first set is containing two positive values. The second set is 0xFFFFFFFF and 0x00001FFF. The second set is containing a negative value and a positive value. Most of the instruction were using the addSub to compute the result. Therefore, I was changing positive-negative term for the first value of each set. ThePro test bench was testing whether the ALU could work expectedly when the value is positive or negative. As a result, the ALU component I built works as I expected.

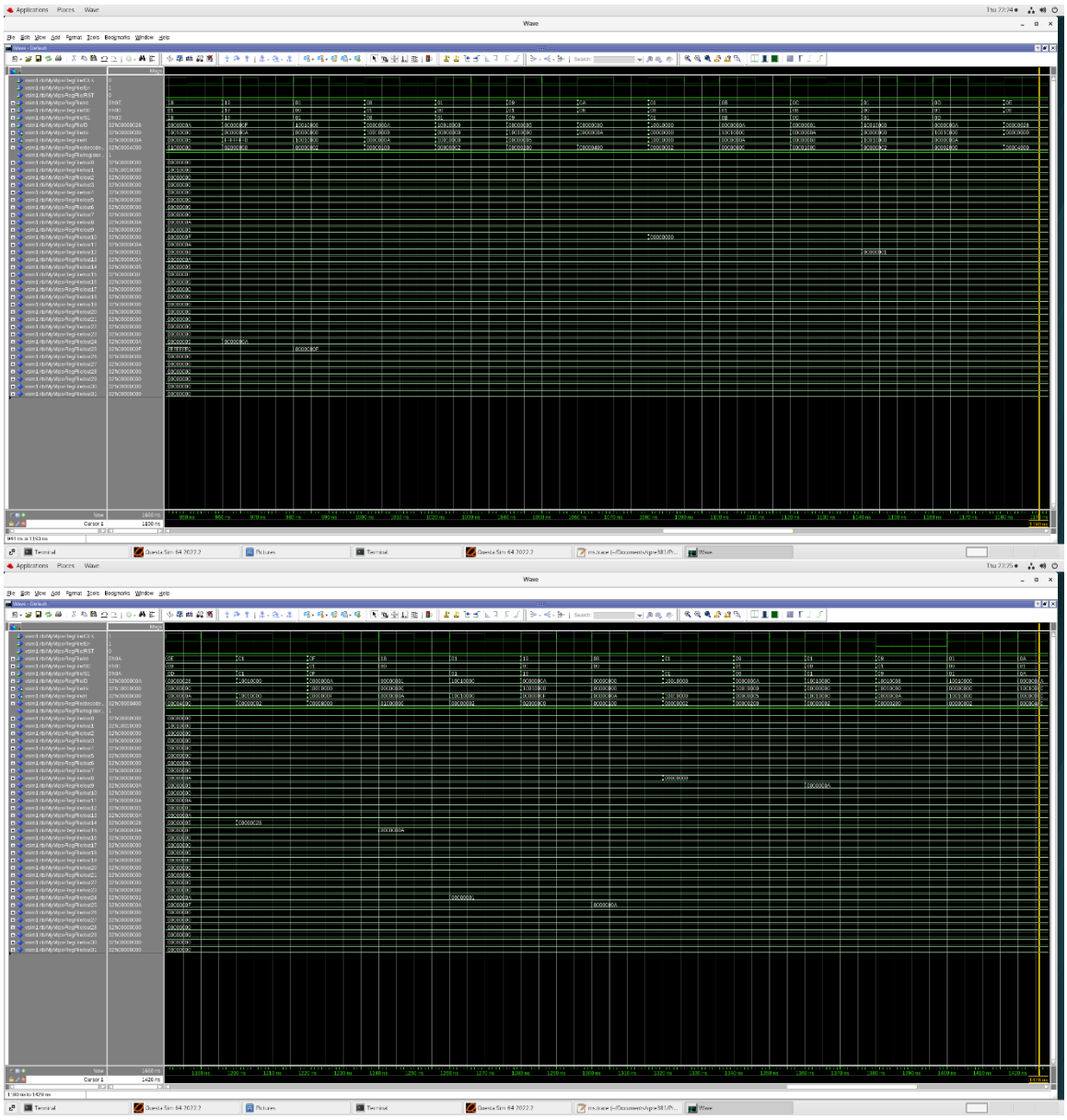
[Part 3] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

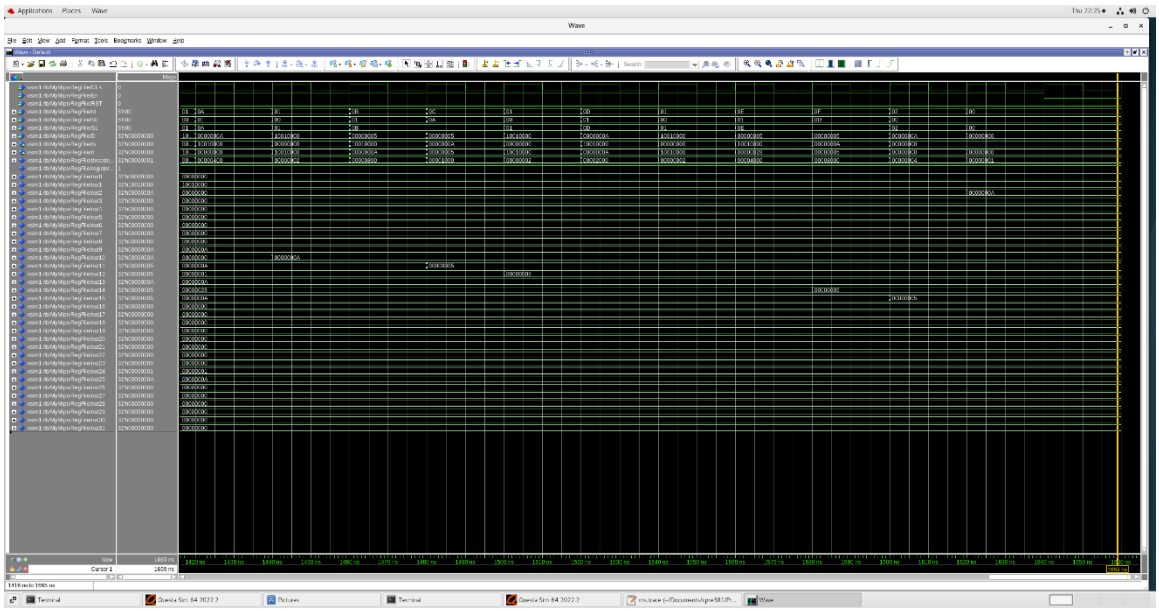
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

In this test application, it was doing a simple task. The test application was testing every required arithmetic or logical instruction once and make sure every instruction was performed correctly within the MIPS processor I created.

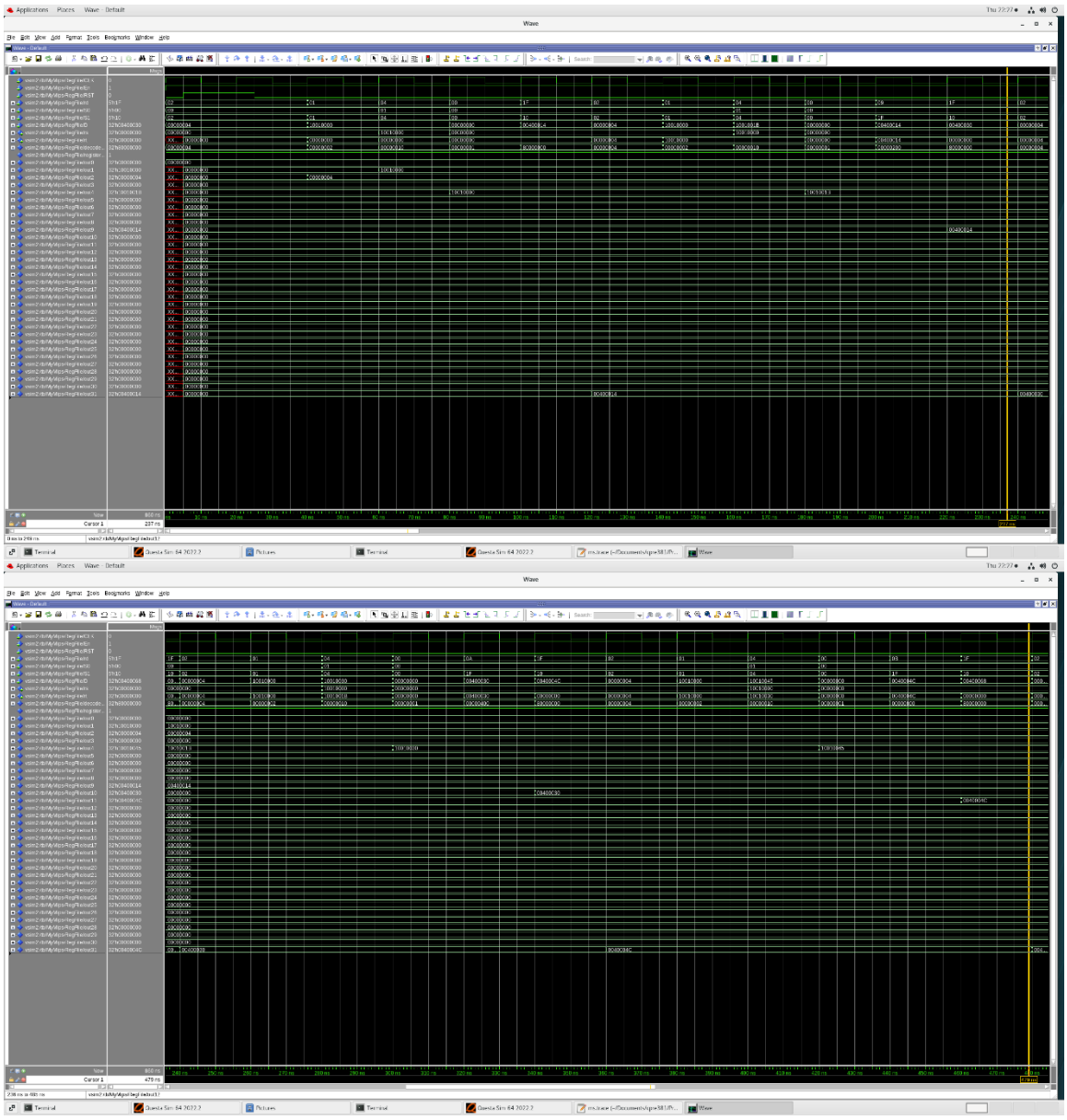


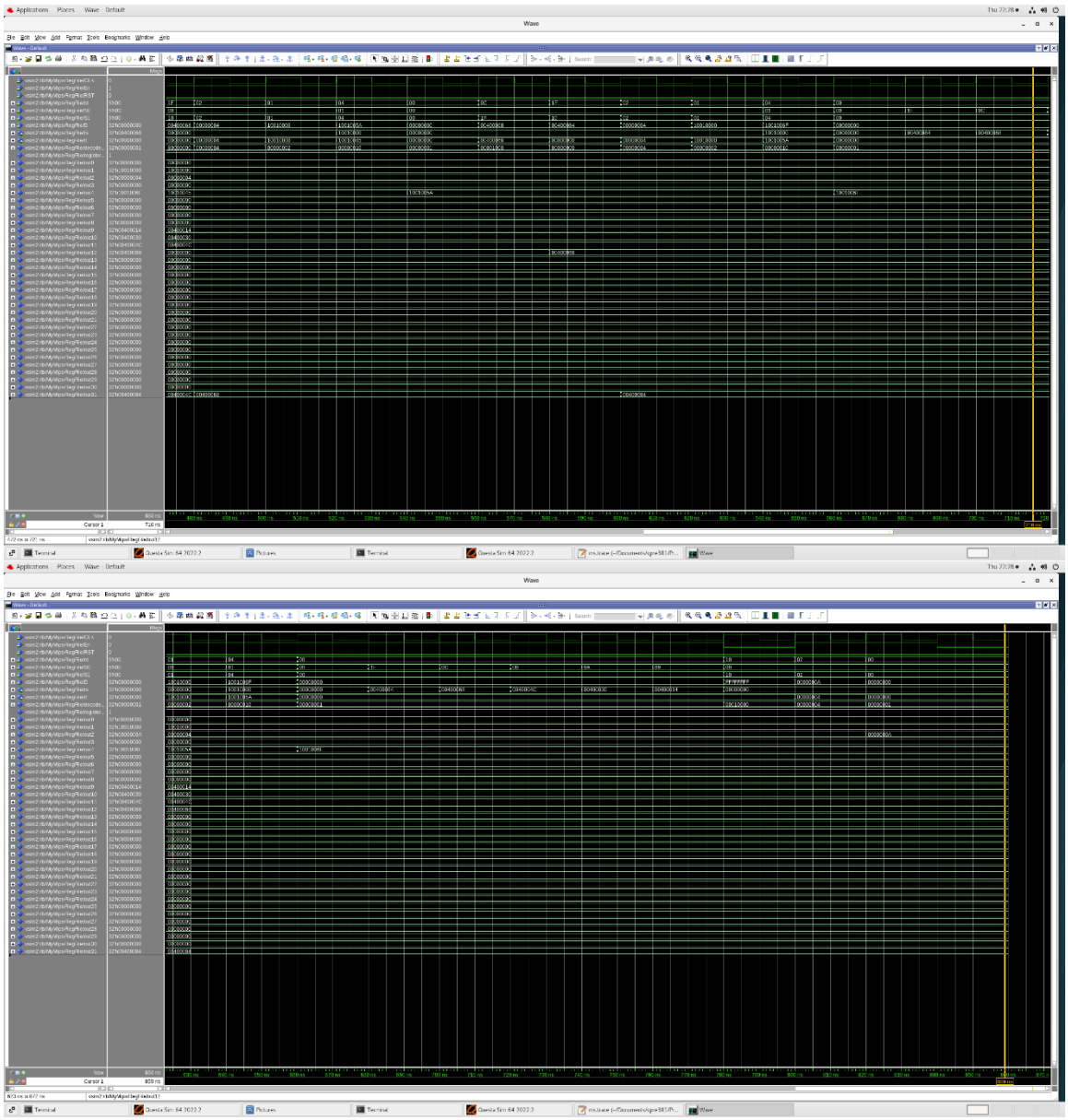






[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.





In this application, I was creating a recursive program. There are five functions. When the first function was called by the main program. It will print a message and call the second function within the first function. The second function will do the same thing and call another function. The third function and fourth function will also do the same thing until the fifth function. The fifth function will print a message and return back to the fourth function. And the fourth function will return back to the third function, and so on. Once it returned back to the main function. The main program will jump the “exit” and end the program.

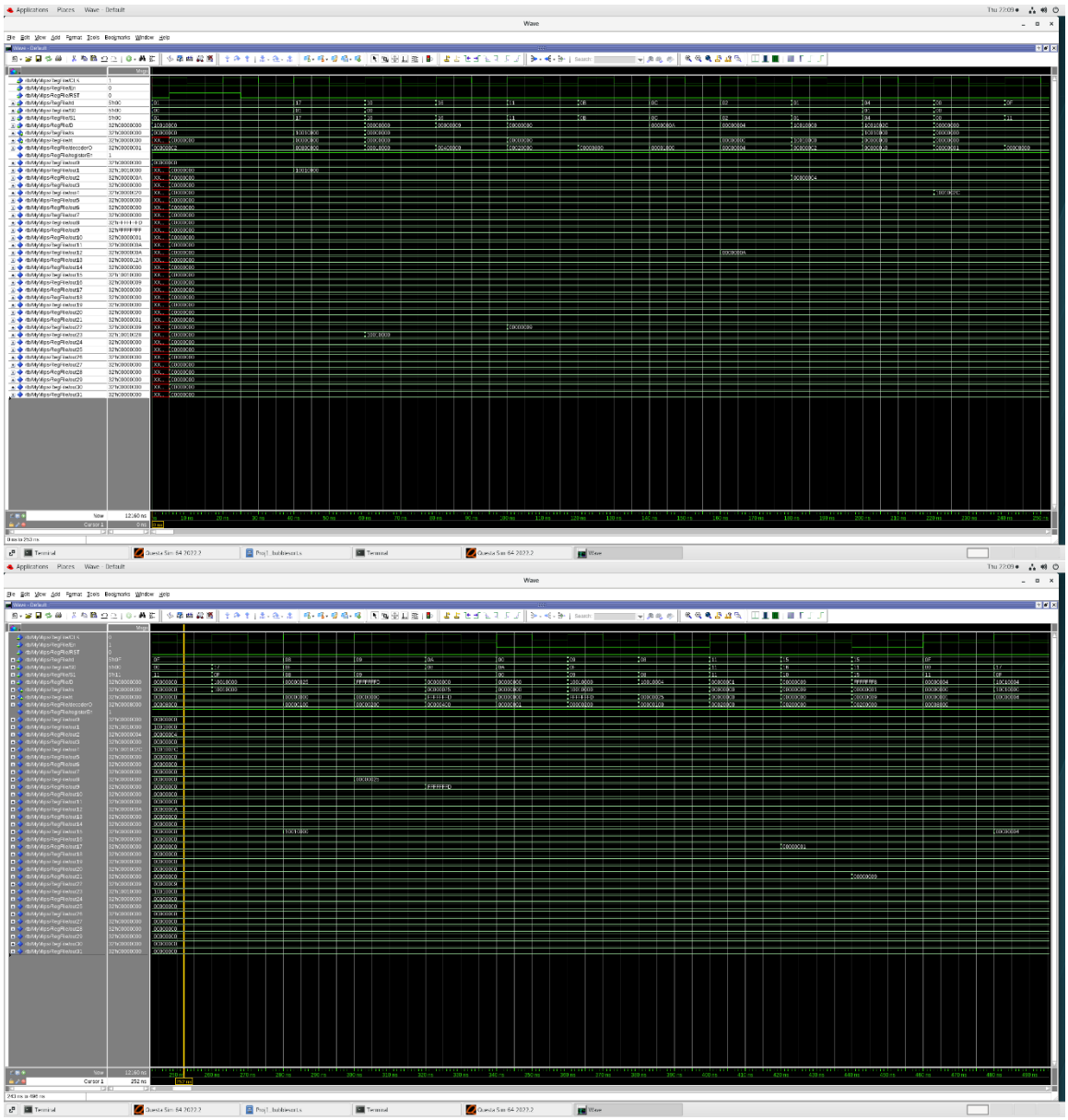
[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

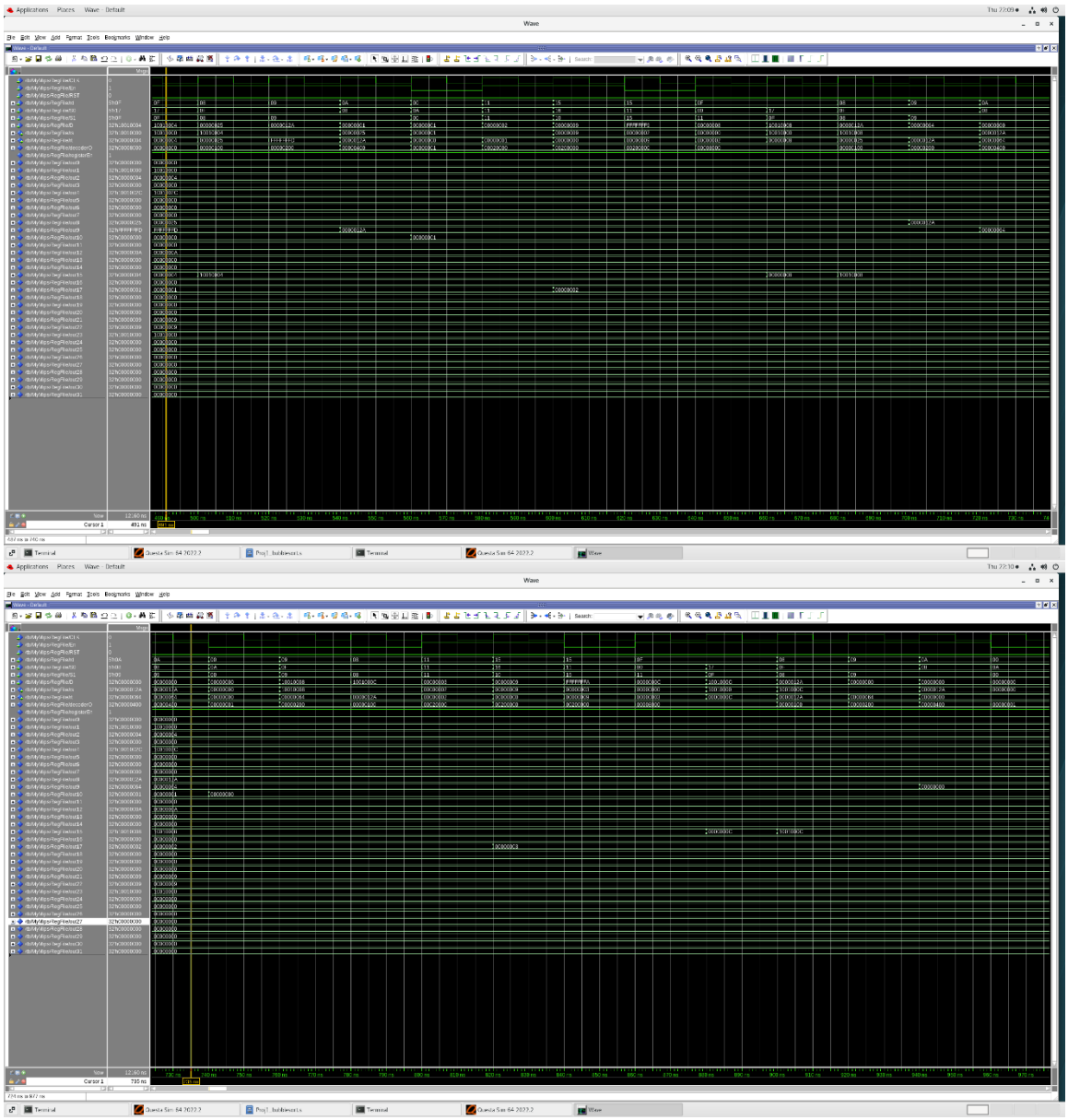
Since I couldn’t input the elements that need to be sorted. Therefore, I initialized the N elements in the MIPS program I created. The elements are 37, -3, 298, 100, 0, -1, 5, 28, 3, 5, 29. Once the program did the bubble sort, it will print “Sorted: -3 -1 0 3 5 5 28 37 100 298” as the result.

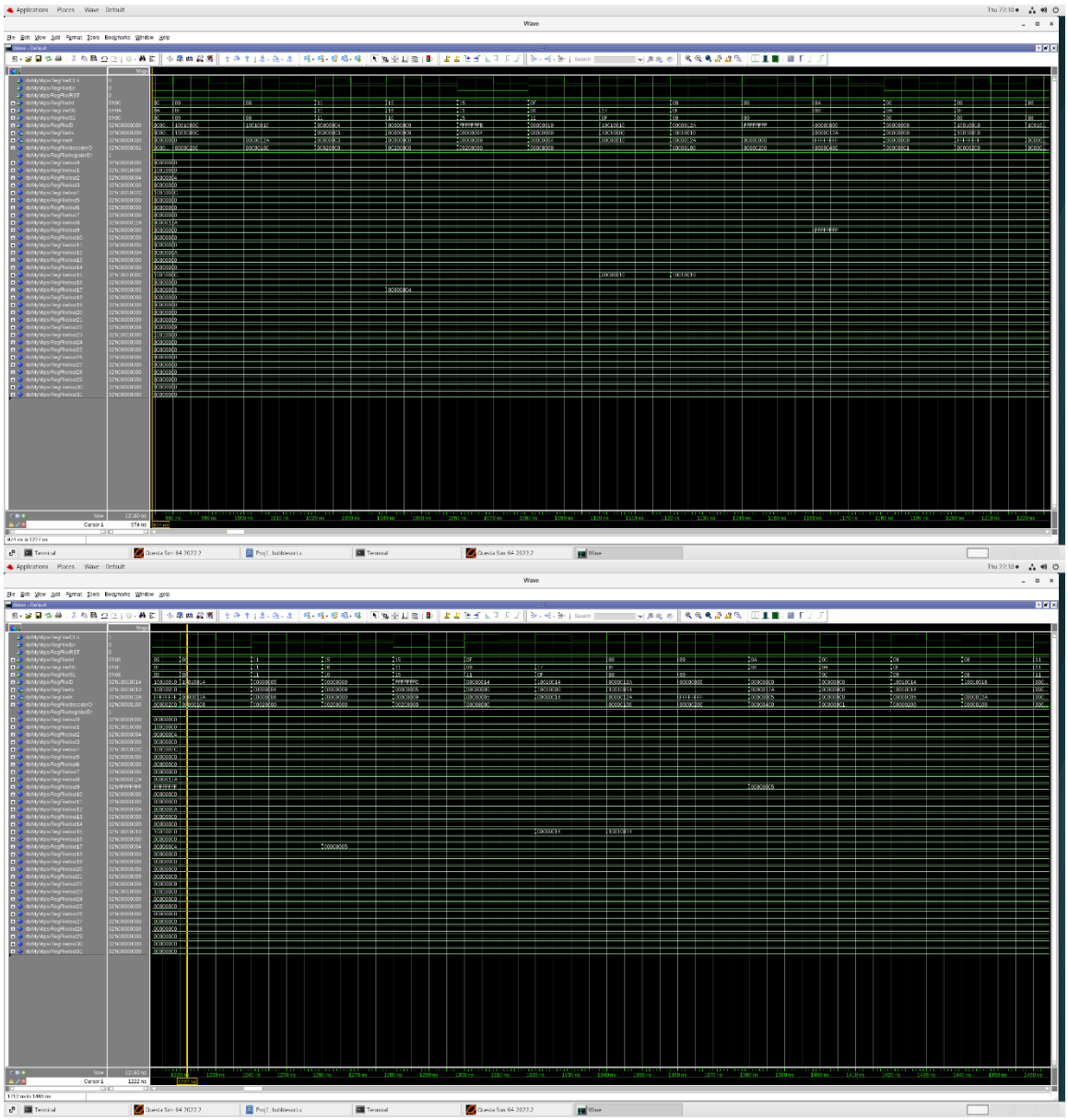
In the MIPS program, the array of integers is defined in the .data section of the program using the "numbers" label. The program initializes the counter $s0 to 0 for the first loop and the counter $s1 to 0 for the second loop. The counter $s6 is initialized to 9 because the program will perform 9 passes through the array. The program then loads the address of the "numbers" array into the $s7 register.

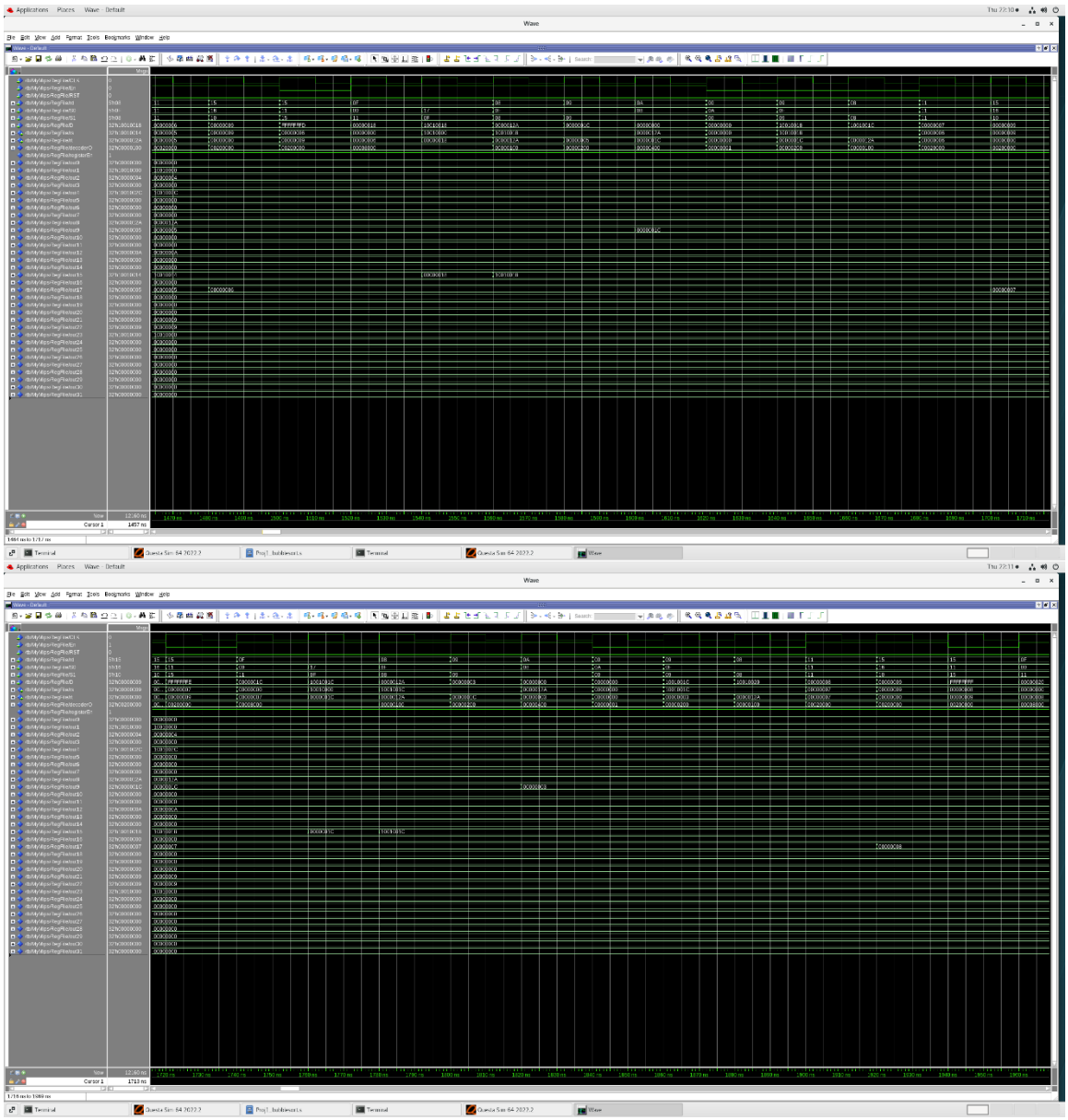
The program enters the bubbleSort loop where it loads two adjacent elements of the array into $t0 and $t1, compares them using the slt (set less than) instruction, and swaps them if the second element is smaller than the first element. The program then increments the $s1 counter and checks if the counter has reached the value of $s6 - $s0. If the counter has not reached this value, the program loops back to the bubbleSort label and continues sorting the array. If the counter has reached this value, the program increments the $s0 counter and resets the $s1 counter to 0. The program then checks if the $s0 counter has reached the value of $s6, which is 9. If it has not, the program loops back to the bubbleSort label and continues sorting the array.

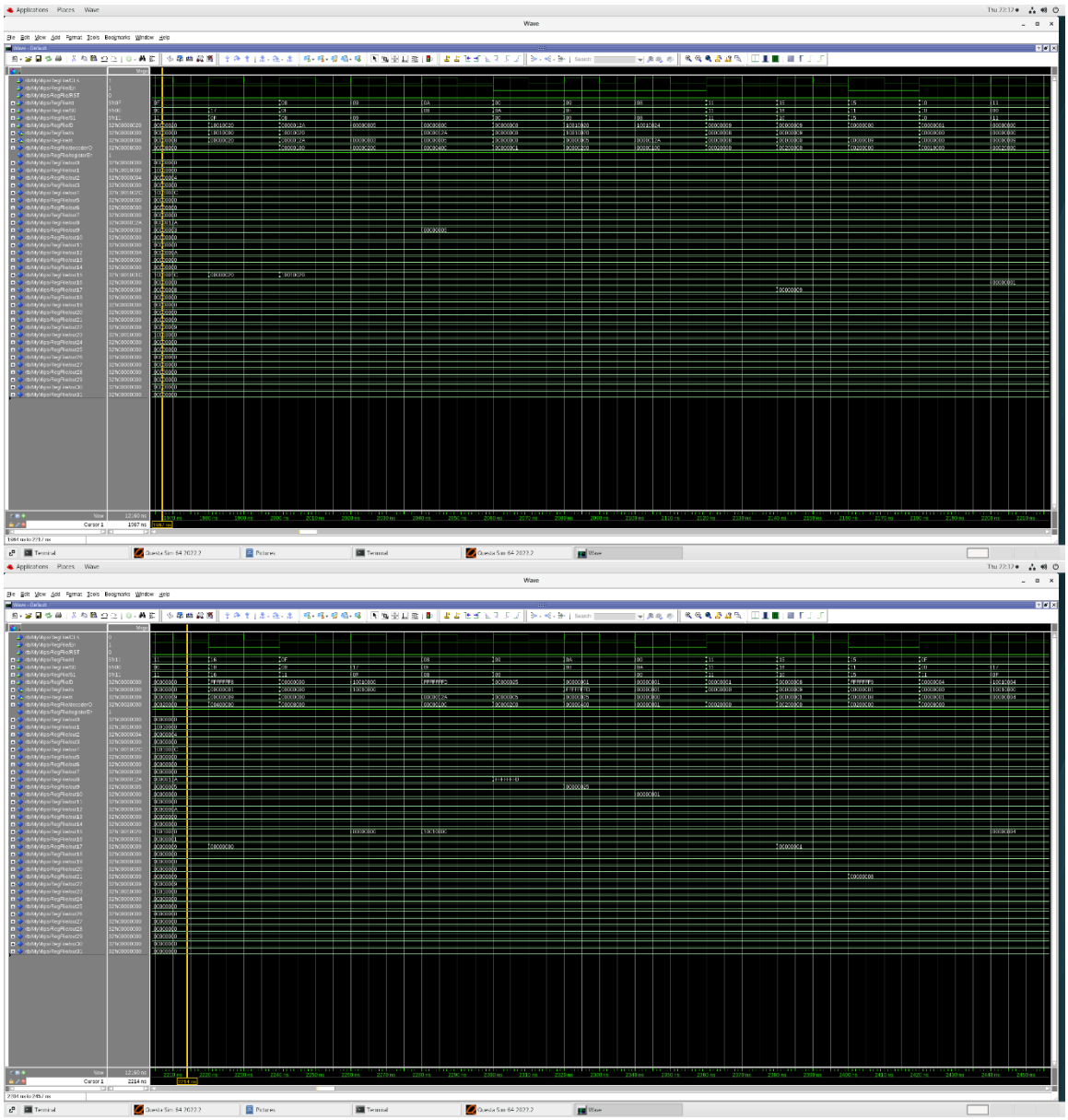
After the array has been sorted, the program enters the print loop where it prints each element of the sorted array separated by a space. The program loads the first element of the array into $t5, prints it using the syscall instruction with the code 1, and increments the $s7 register to load the next element of the array. The program checks if the $t3 counter has reached the value of 10, which is the length of the array, and if it has, the program jumps to the exit label to end the program. If the counter has not reached 10, the program prints a space using the syscall instruction with the code 11 and jumps back to the print label to continue printing the sorted array. Finally, the program ends by using the syscall instruction with the code 10 to terminate the program.

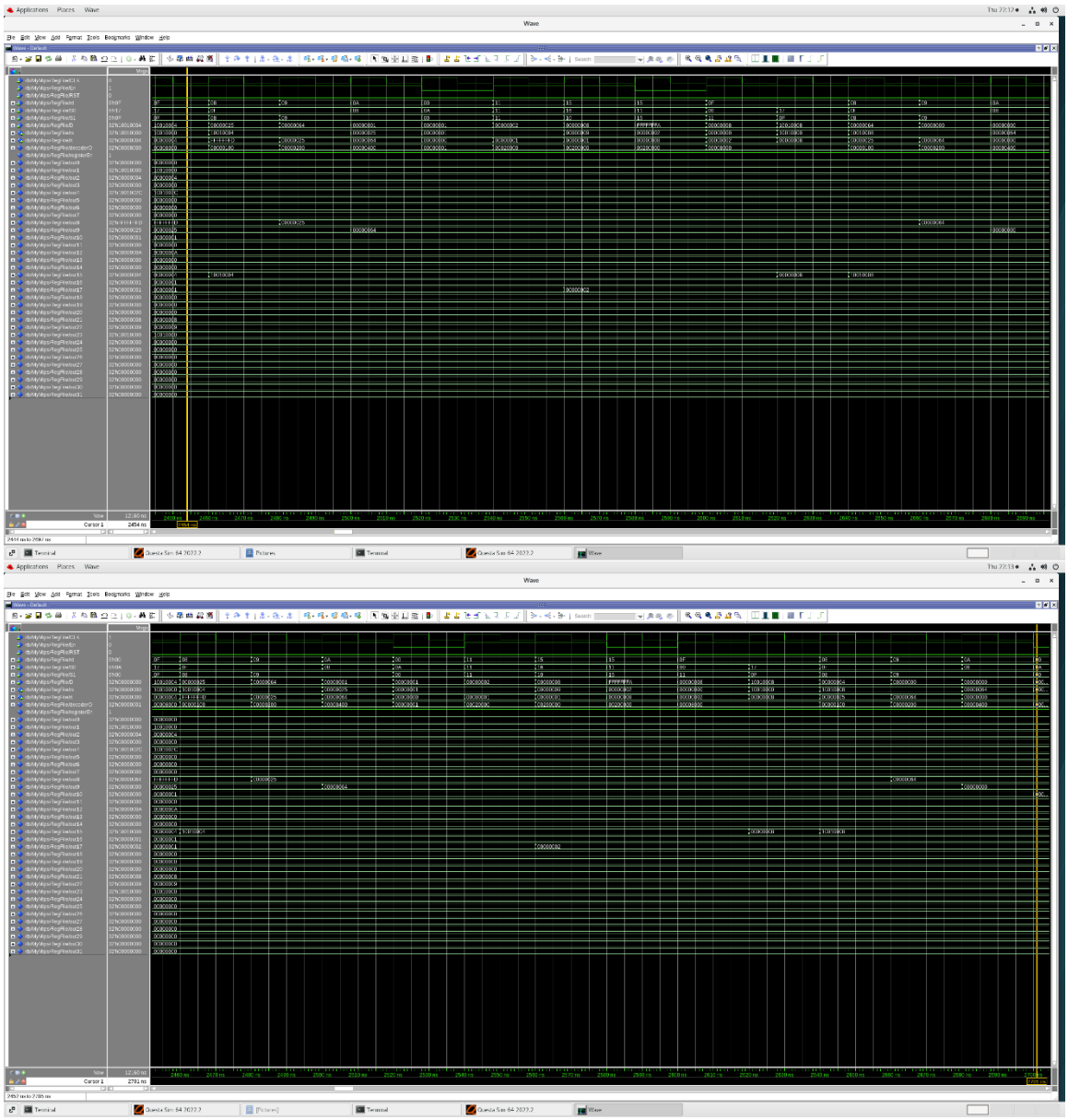


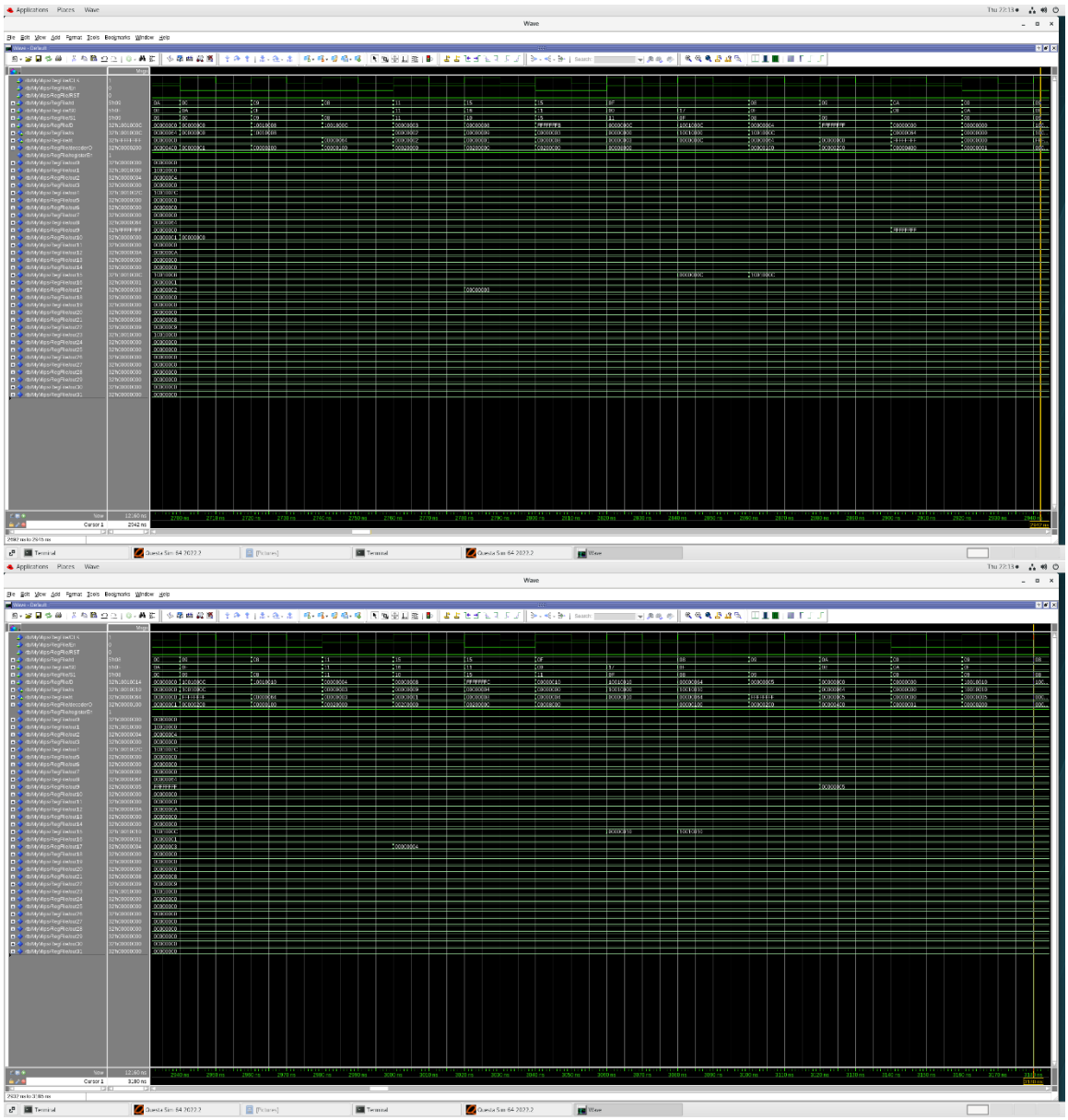


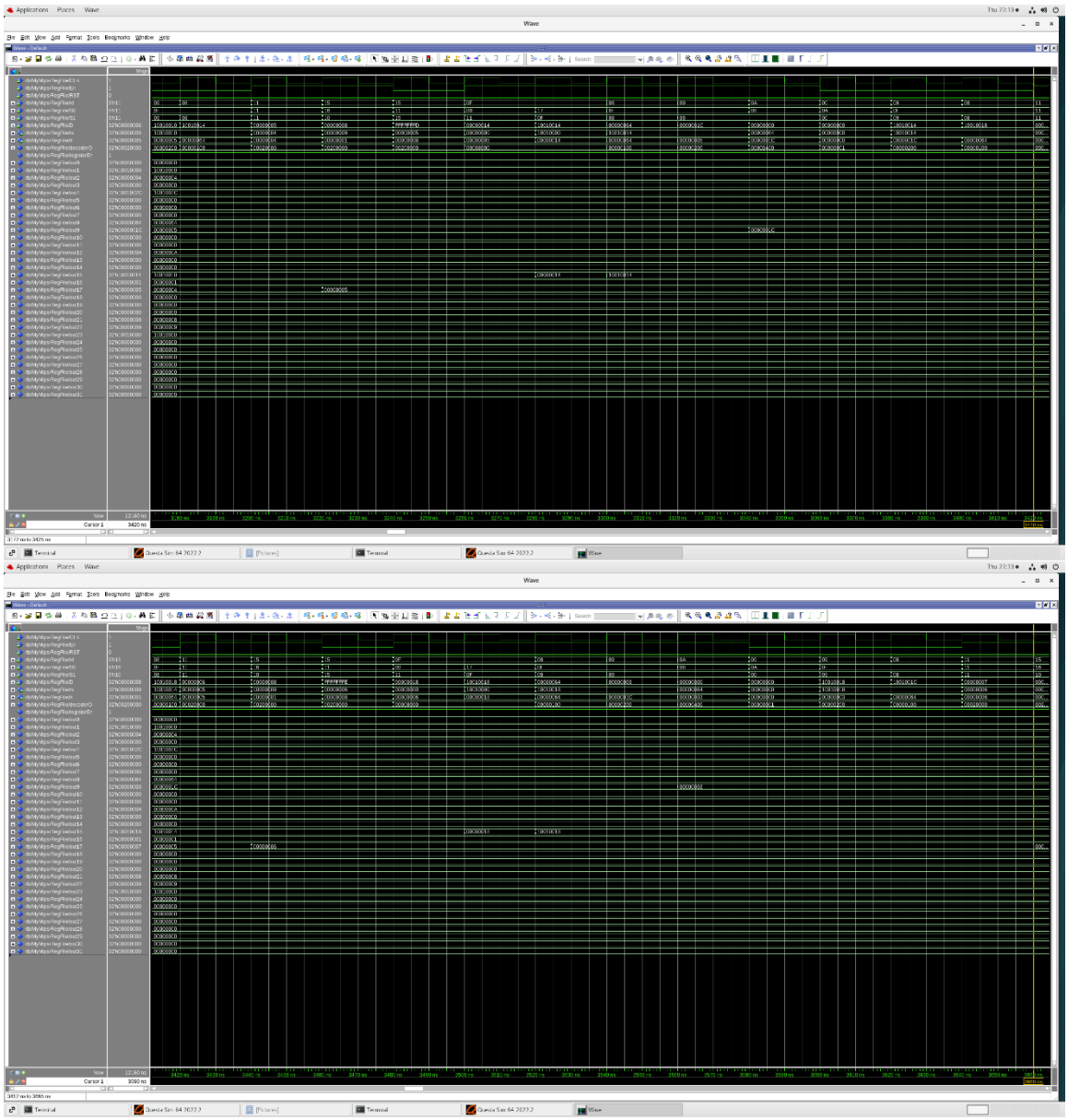


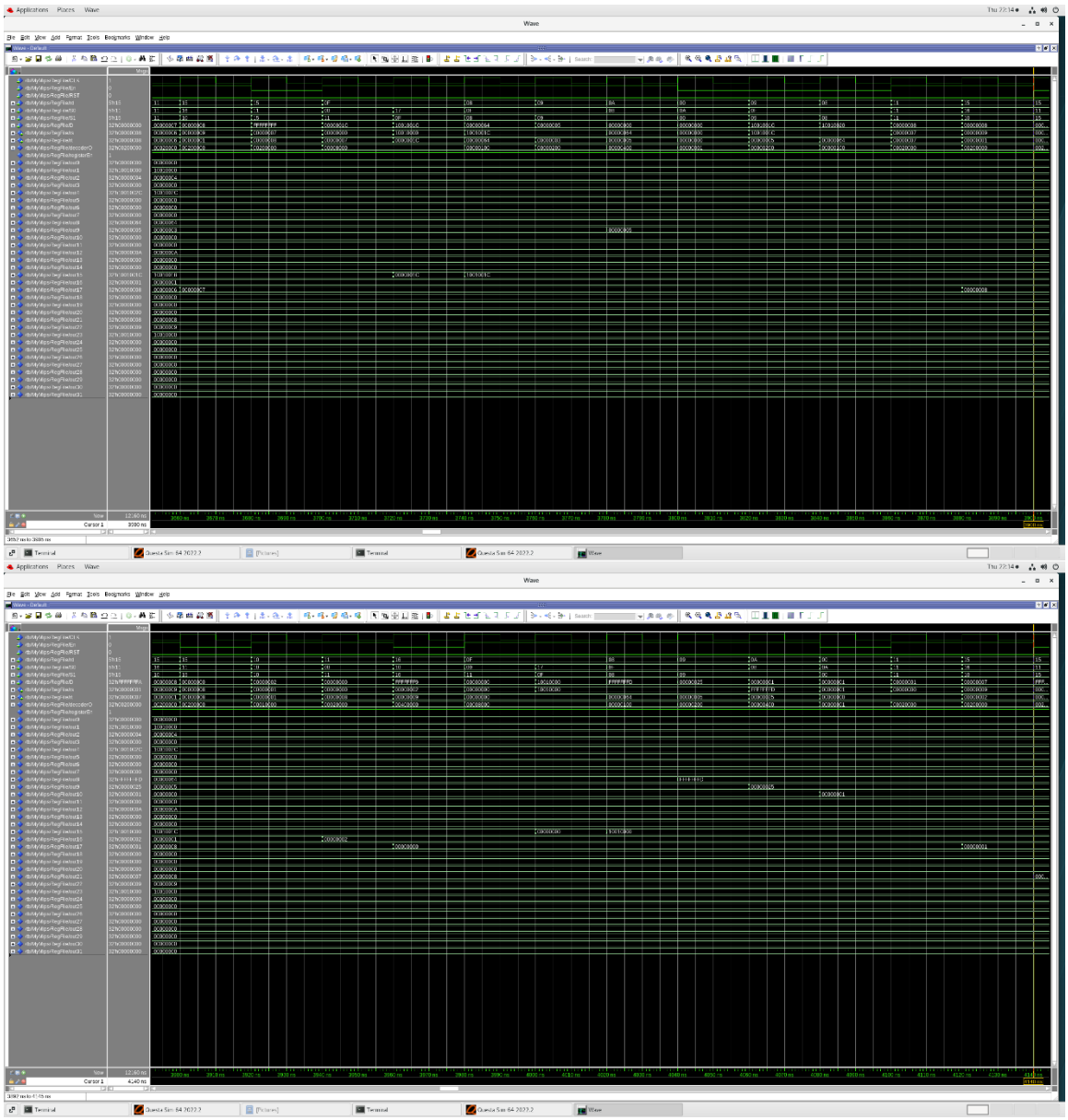


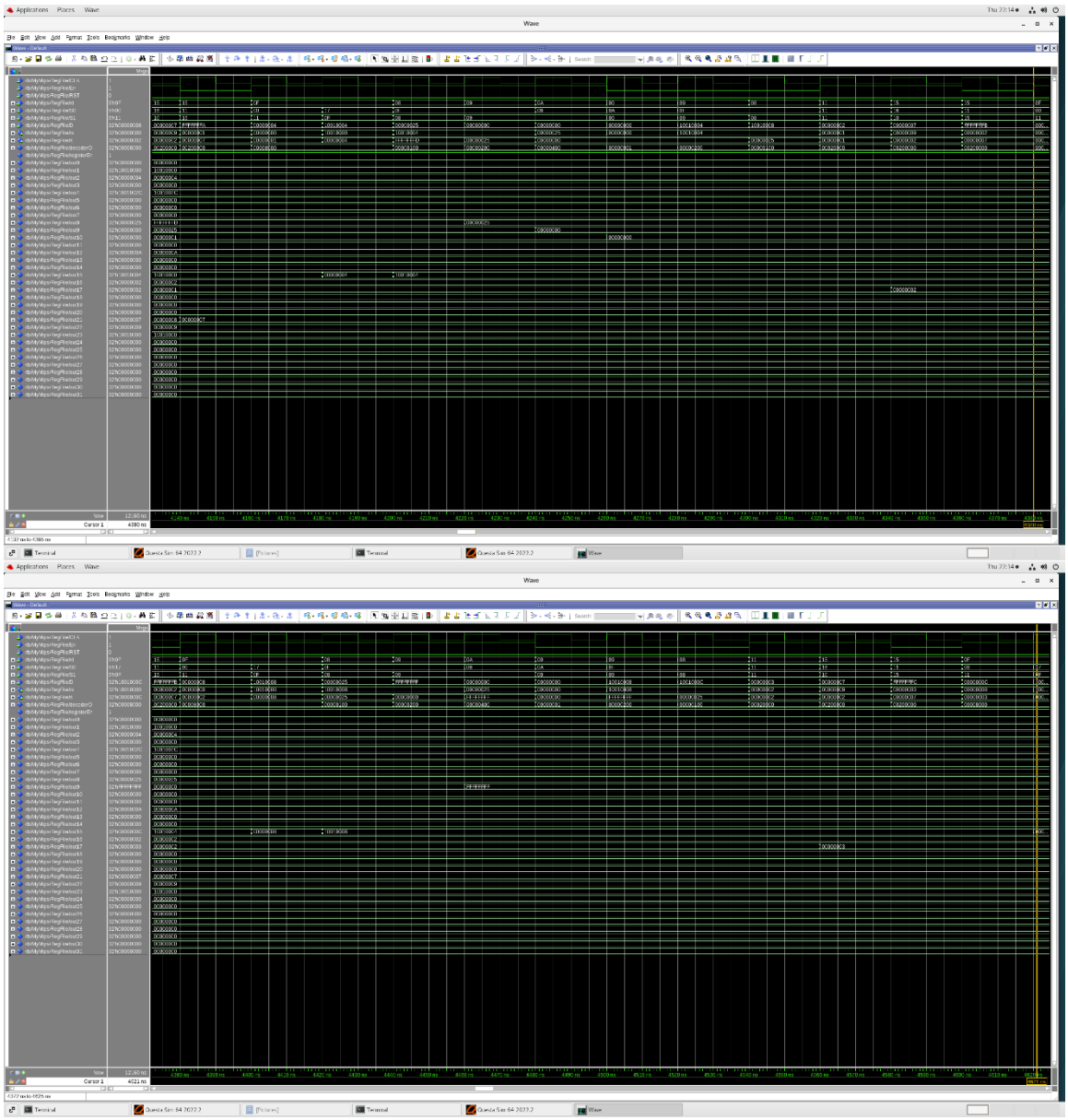


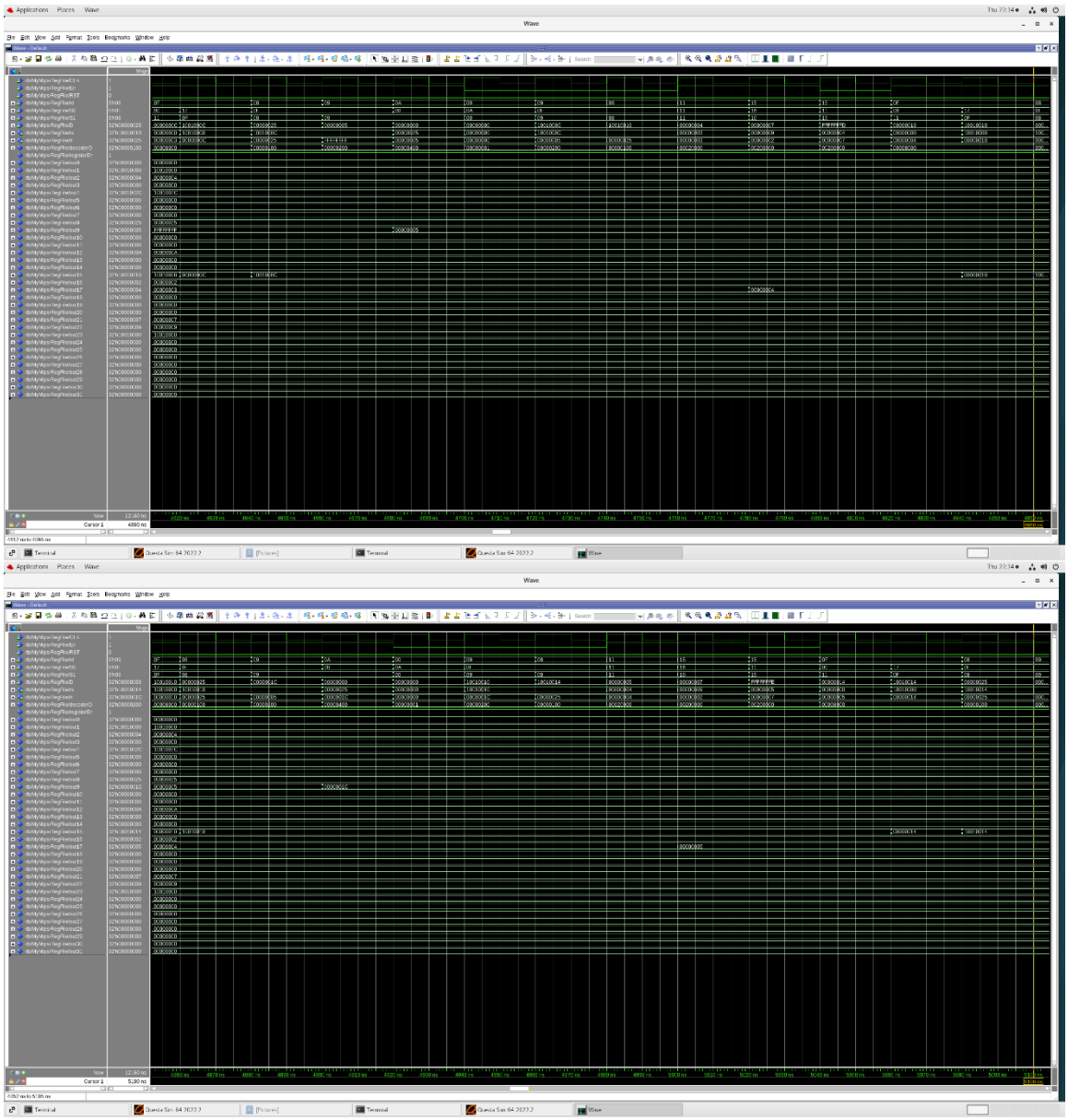


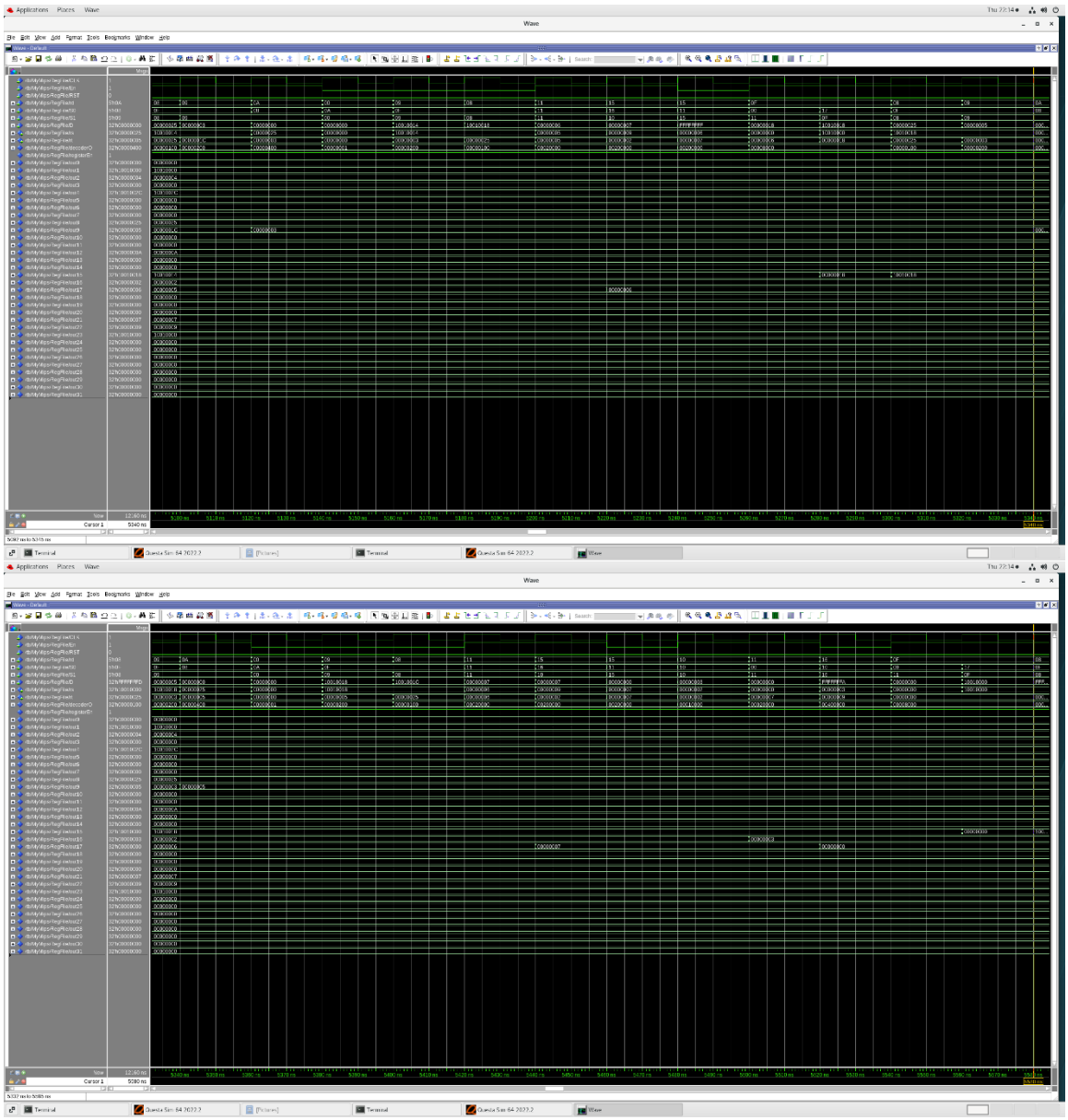


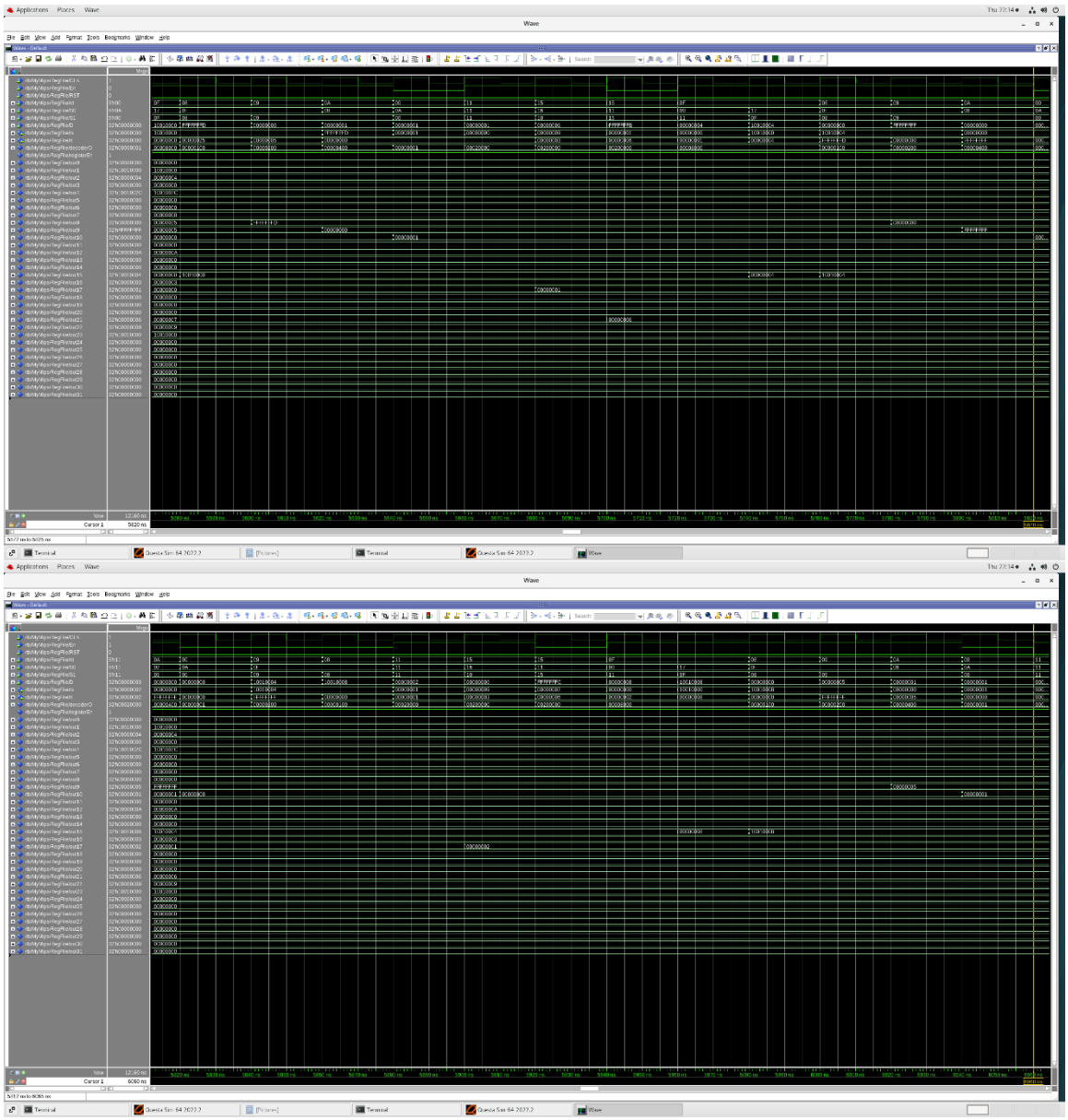


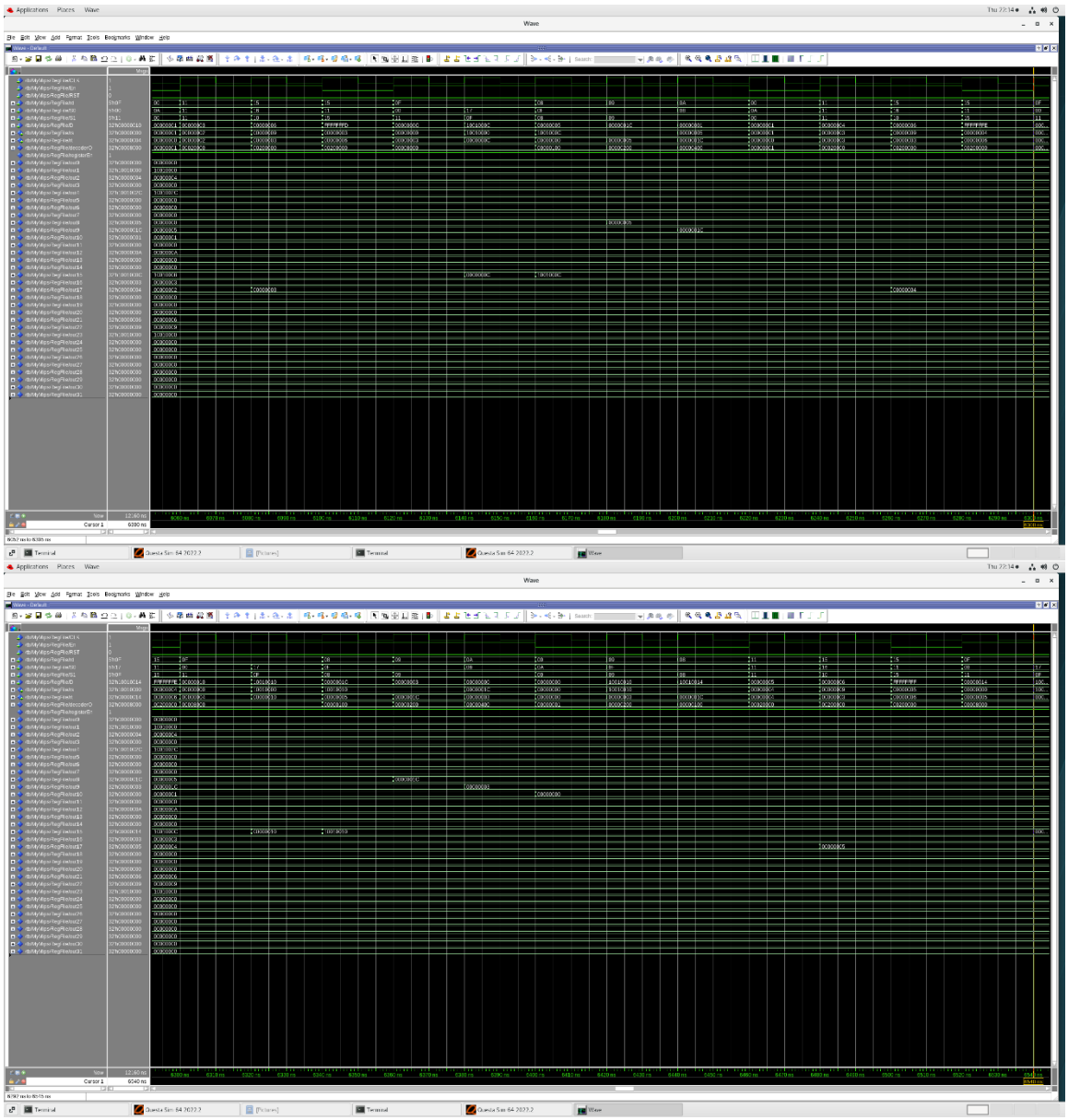


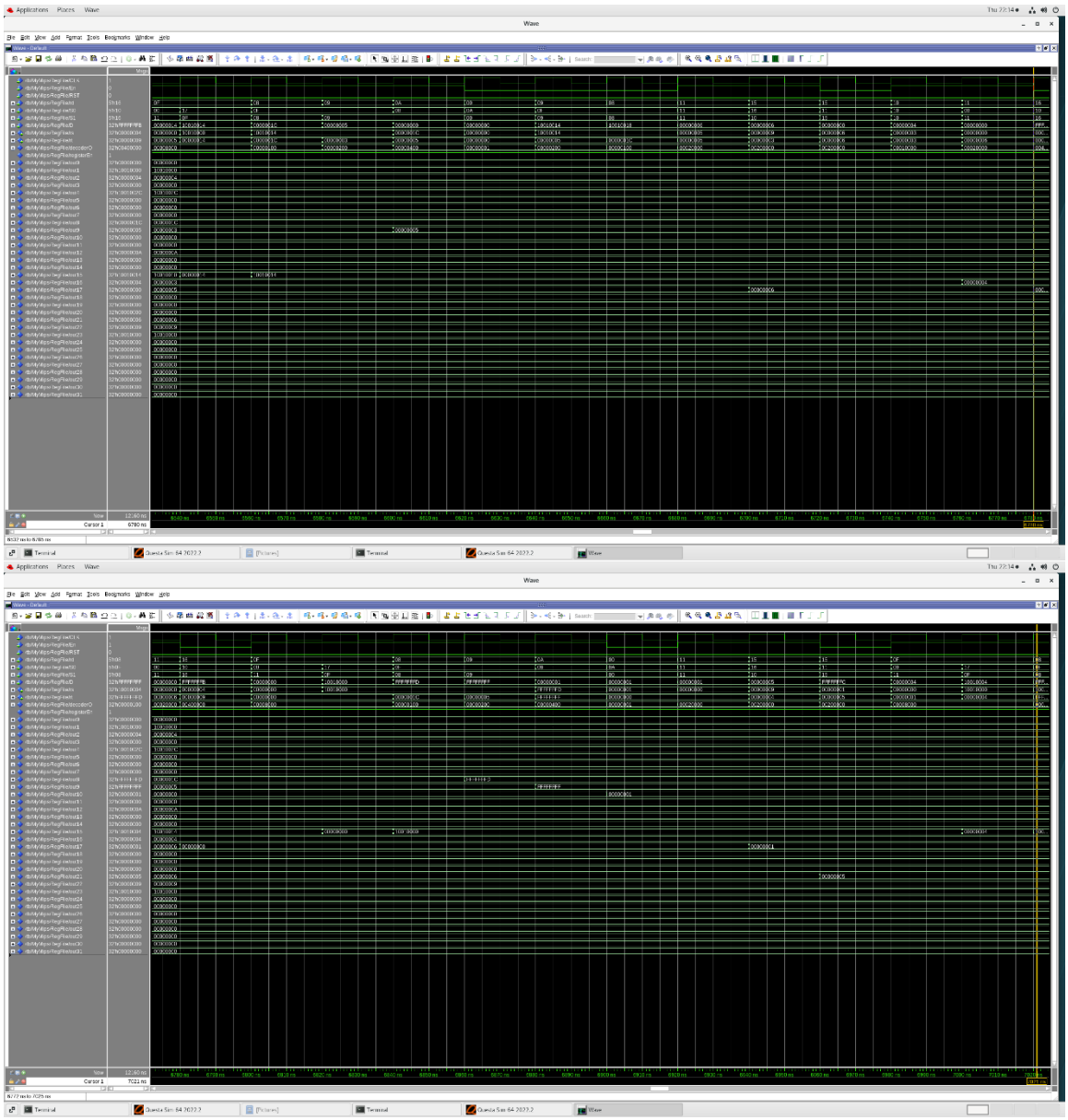


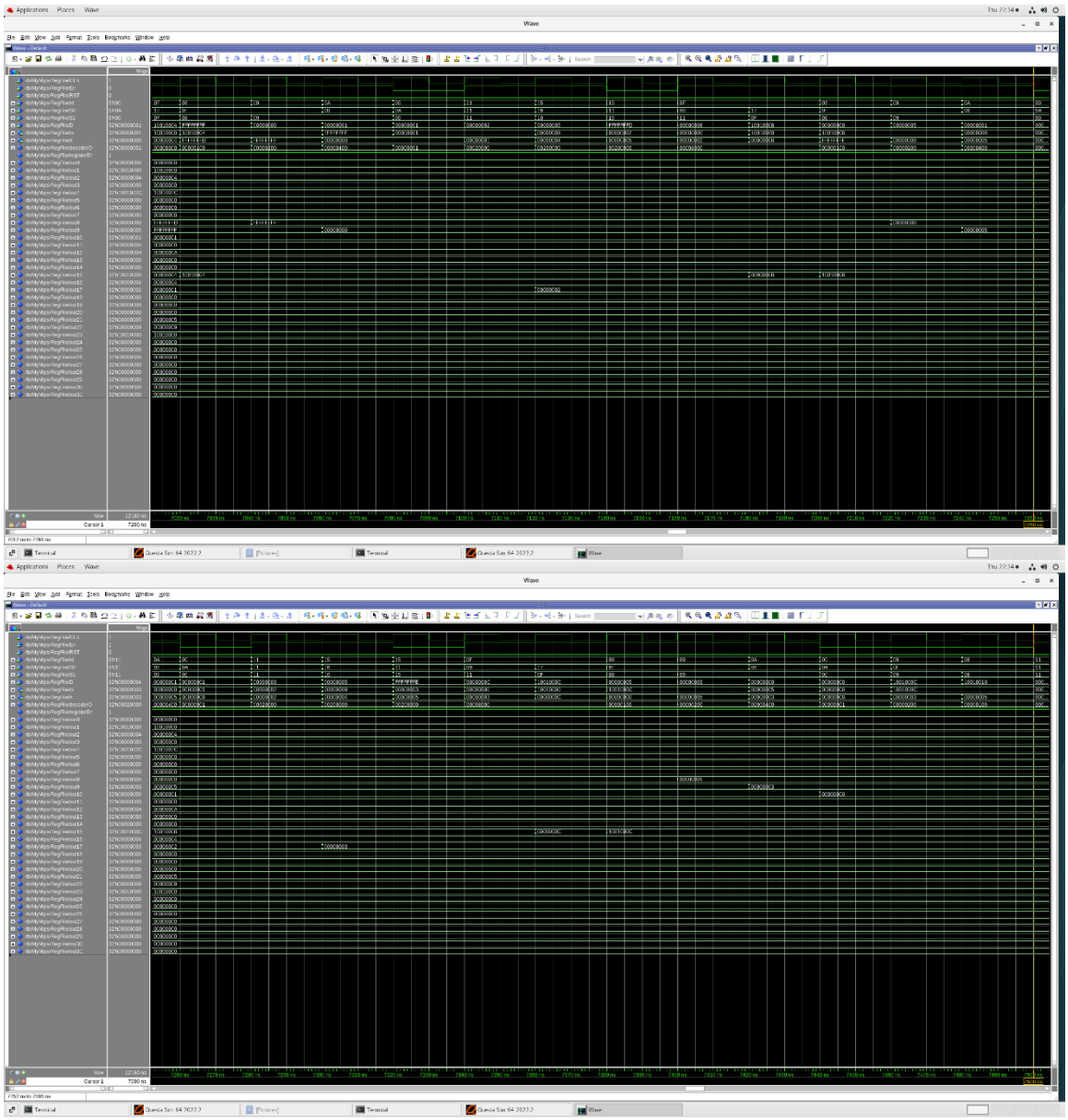


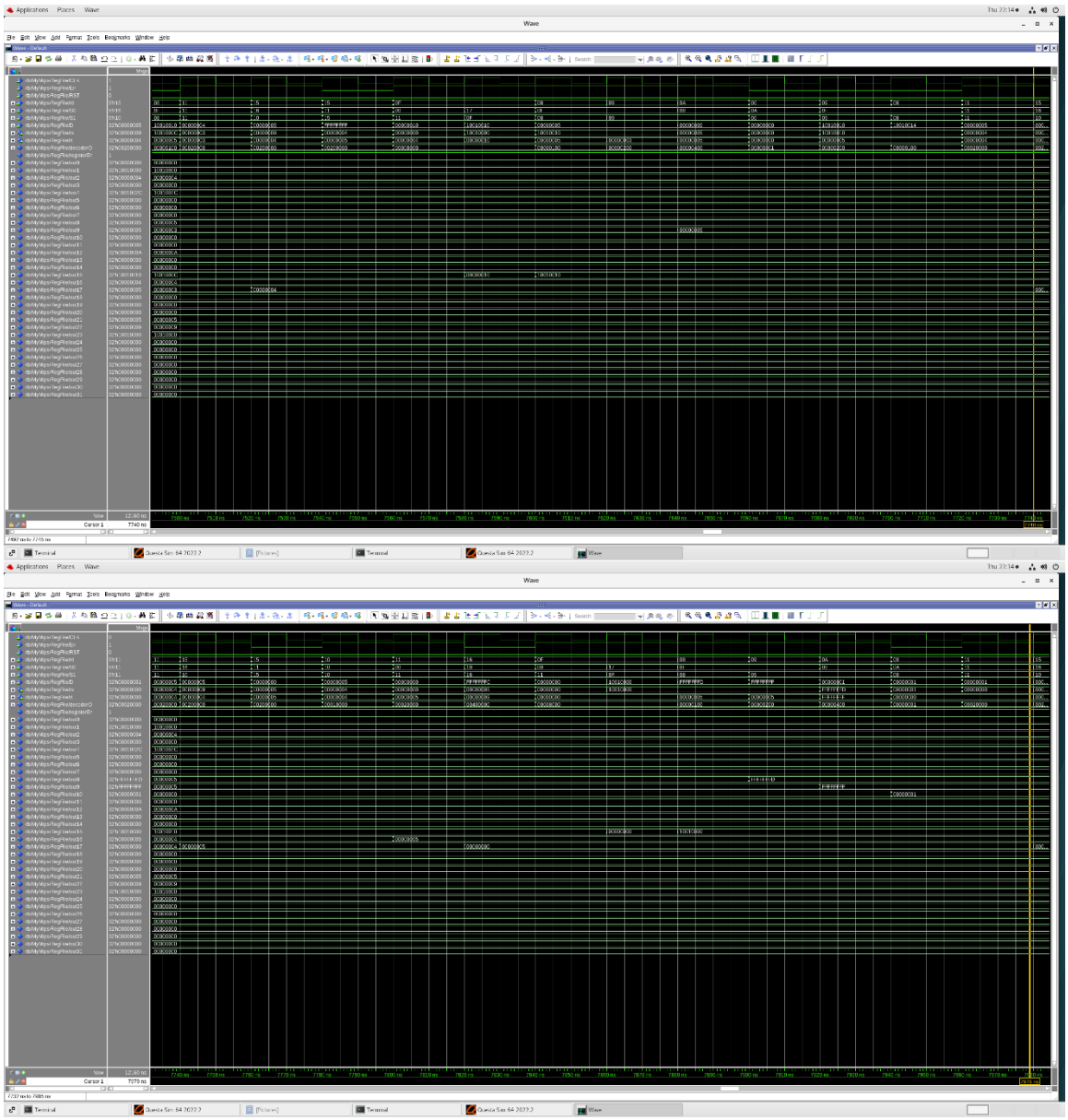


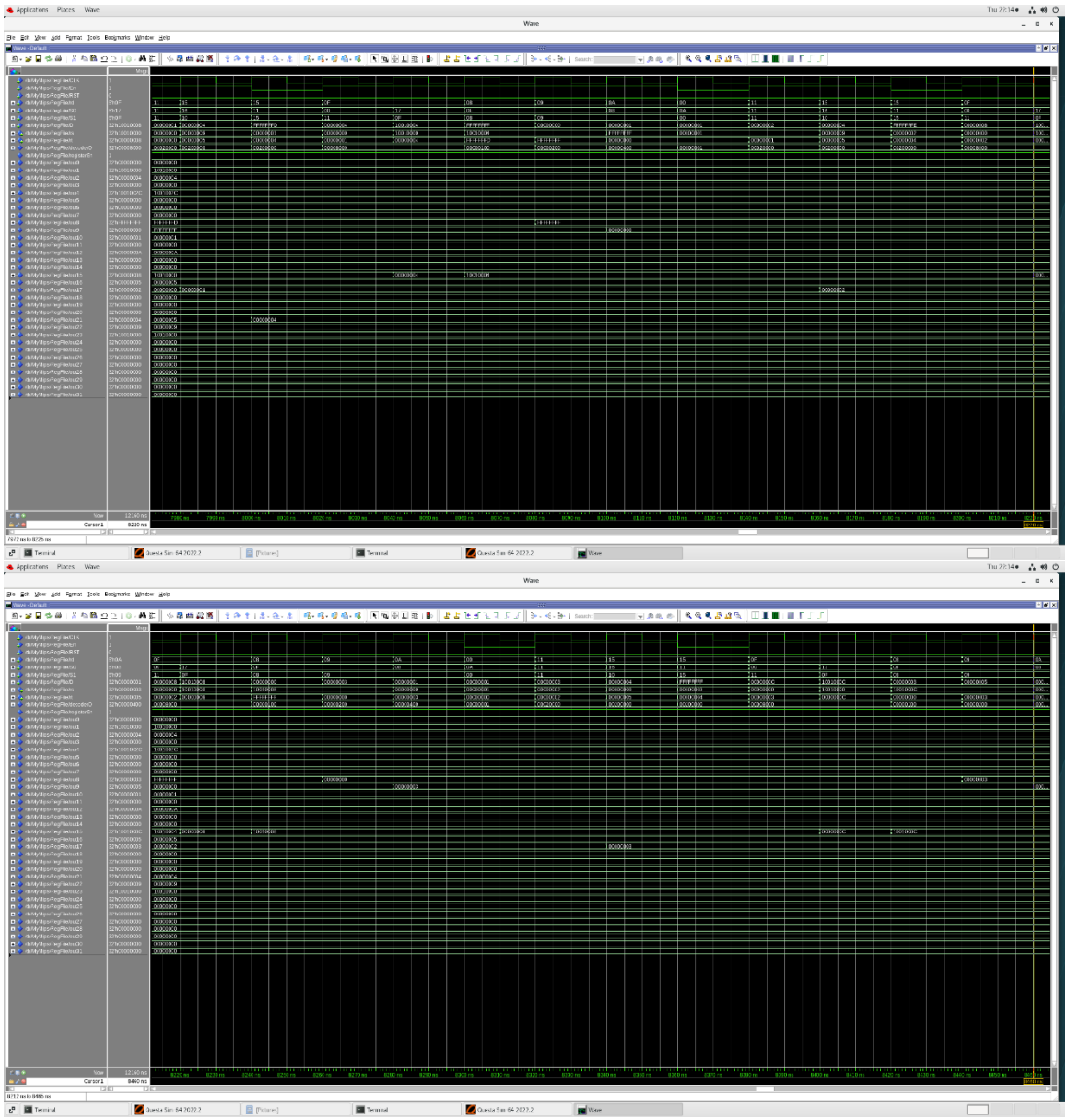


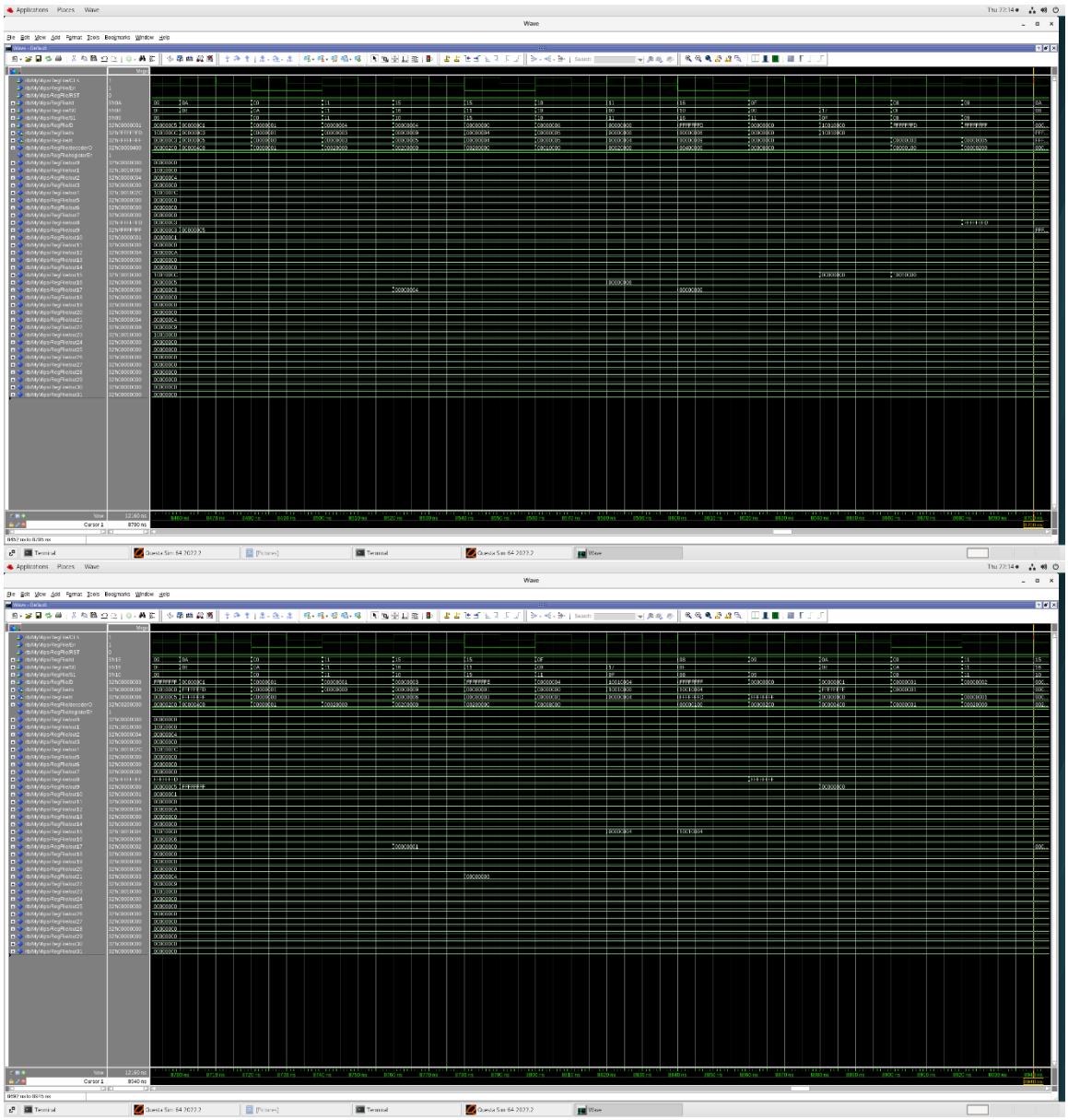


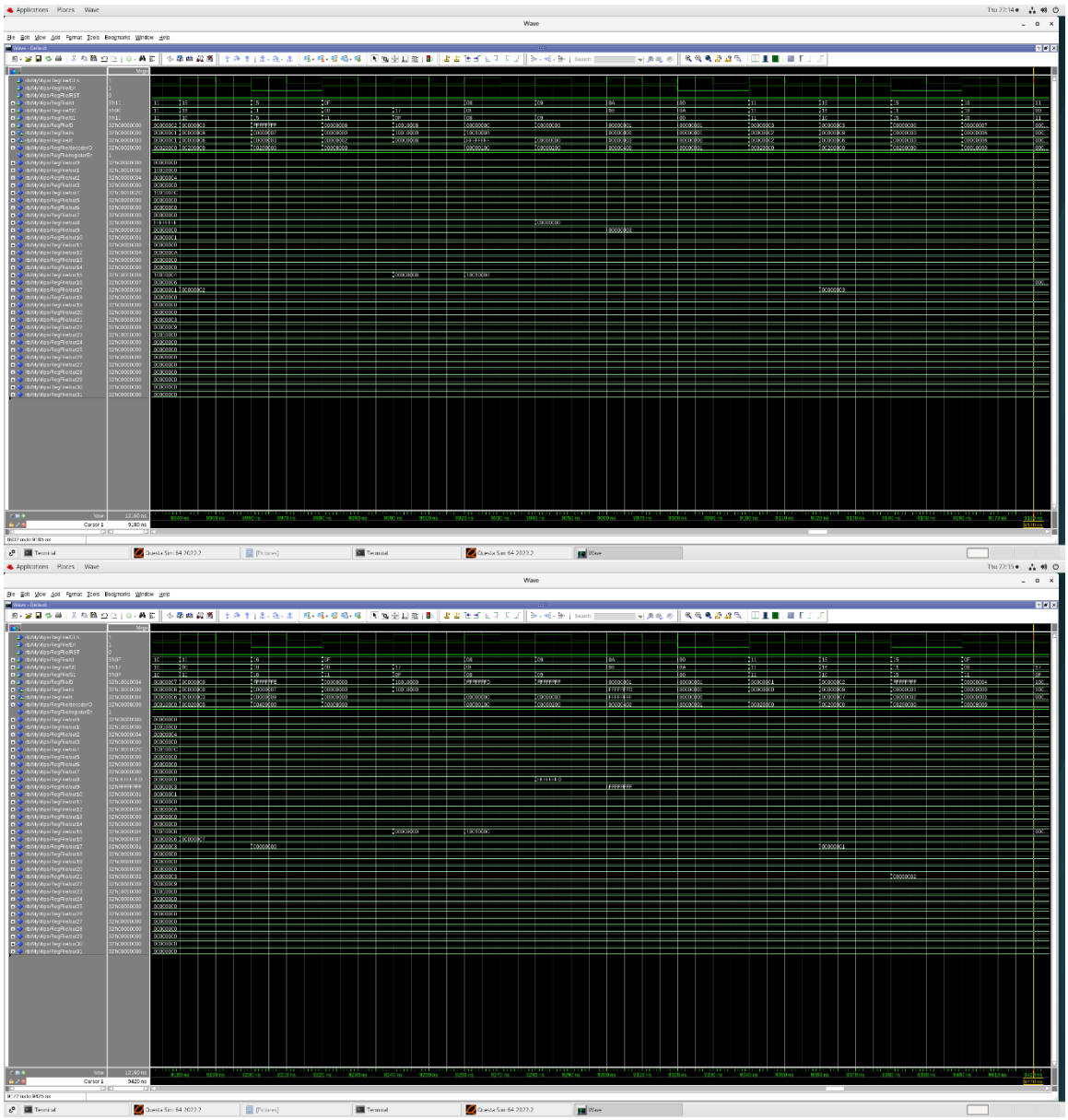


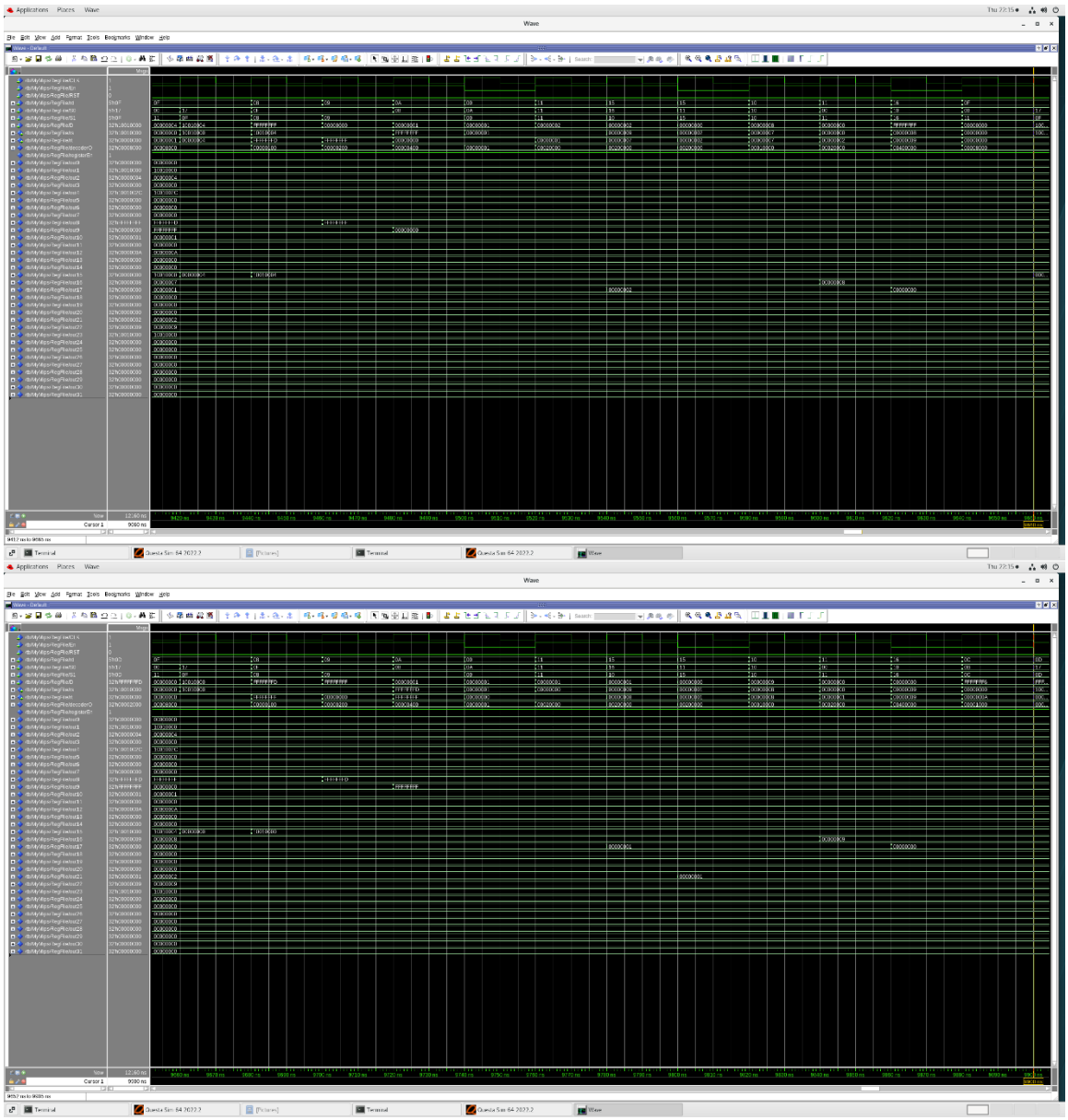


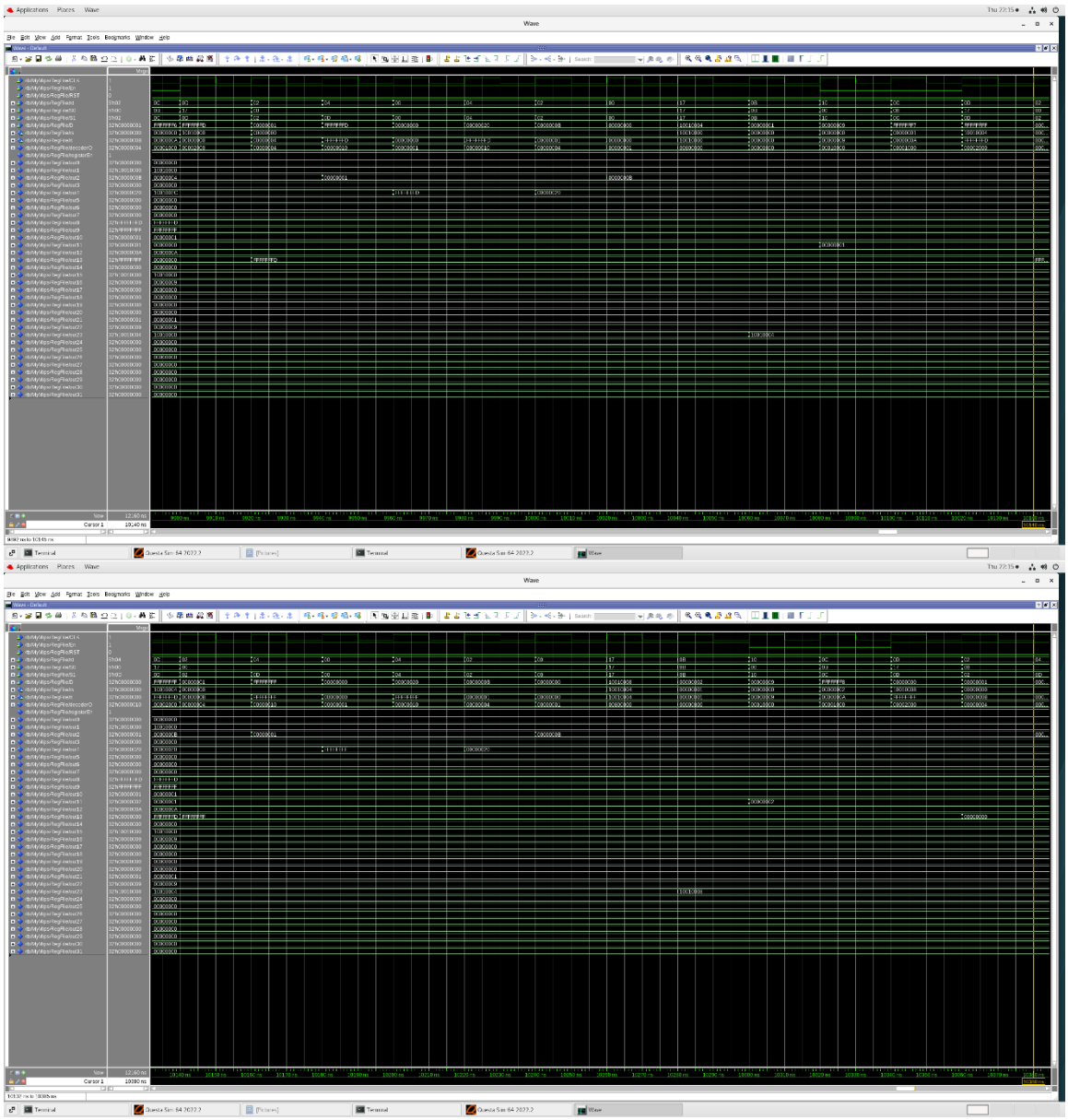


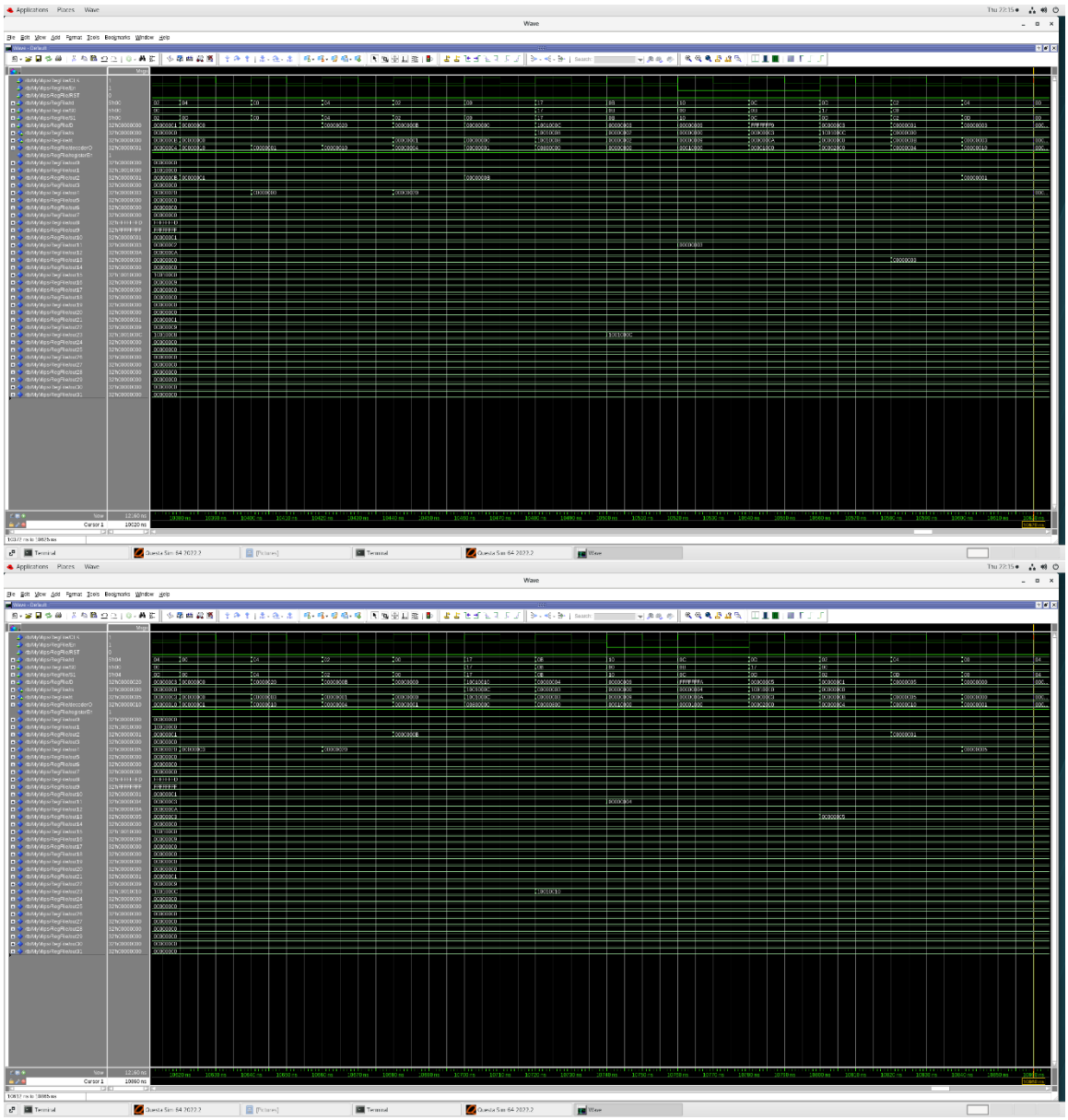


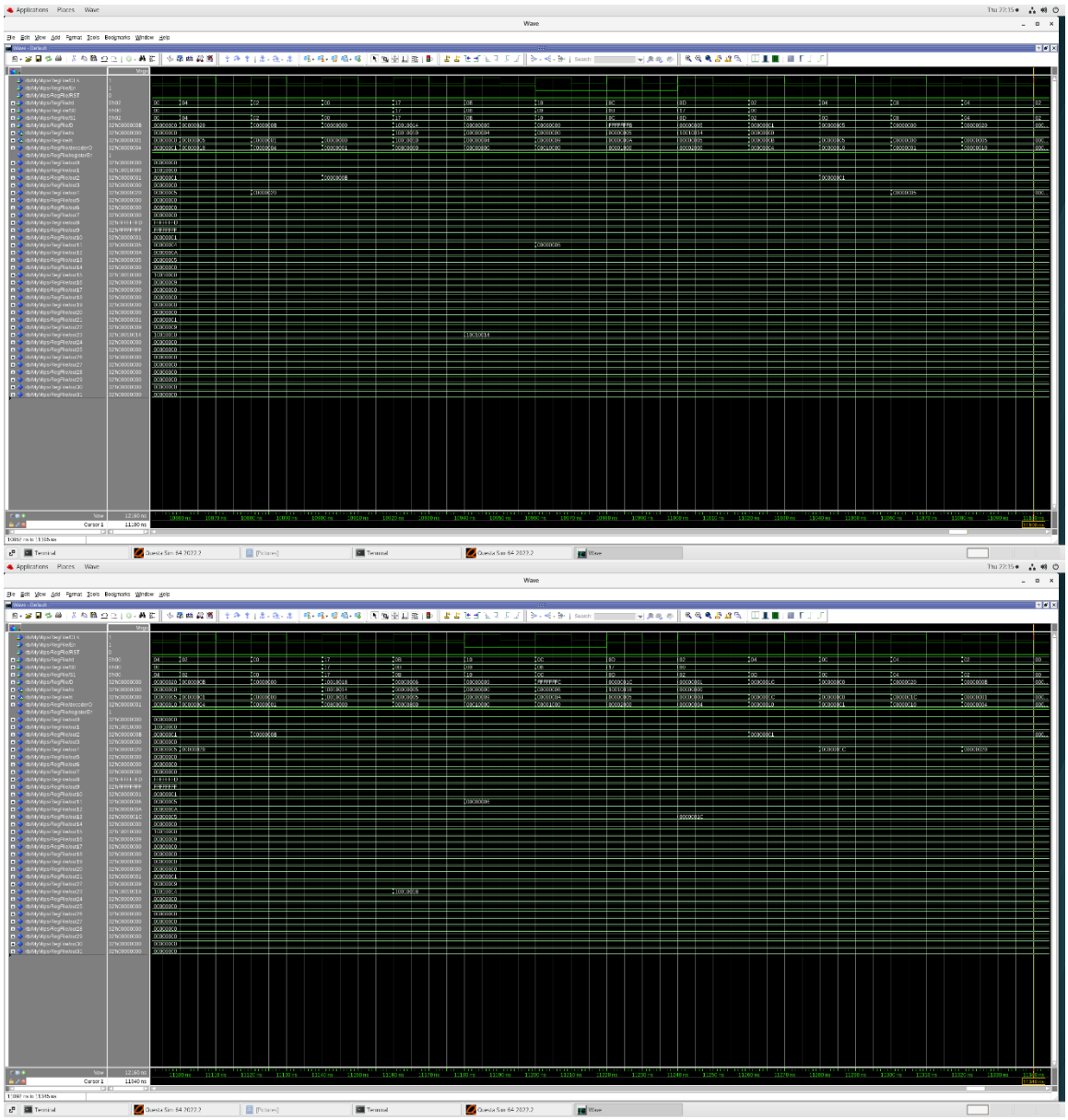


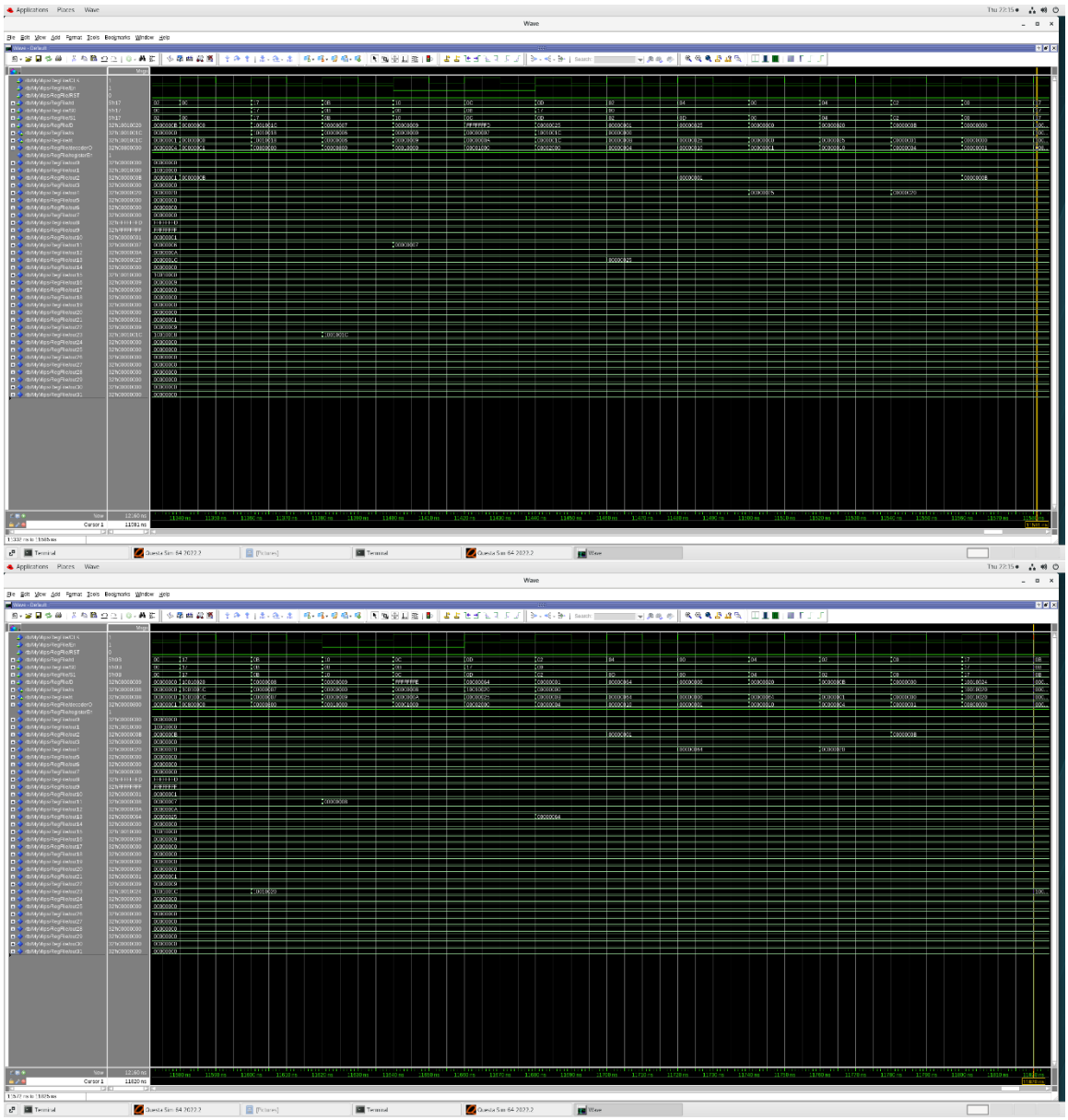


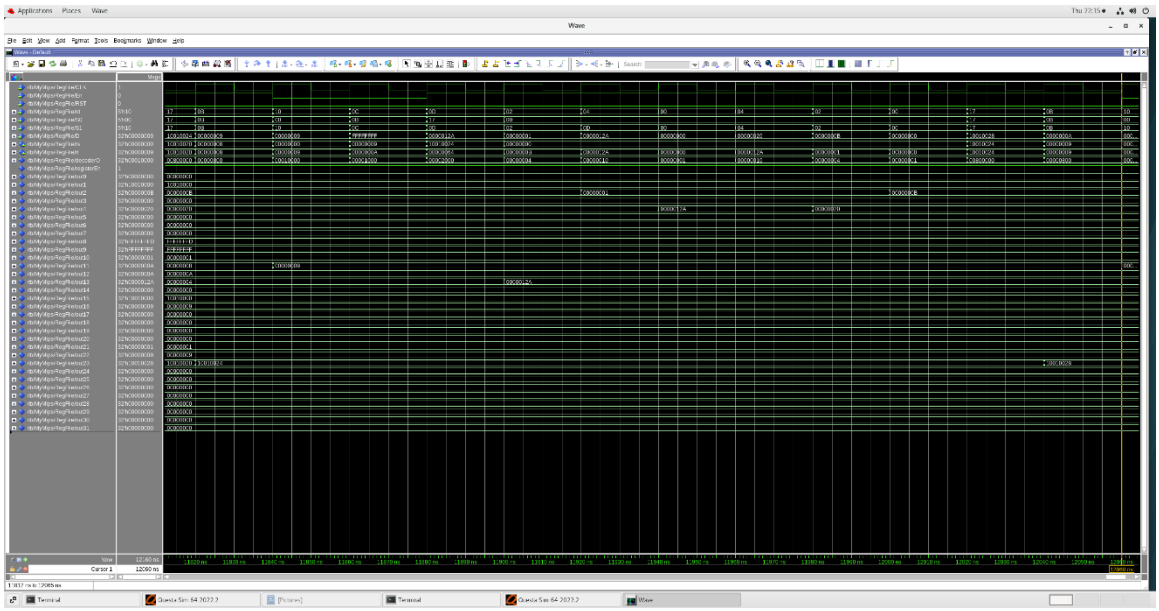








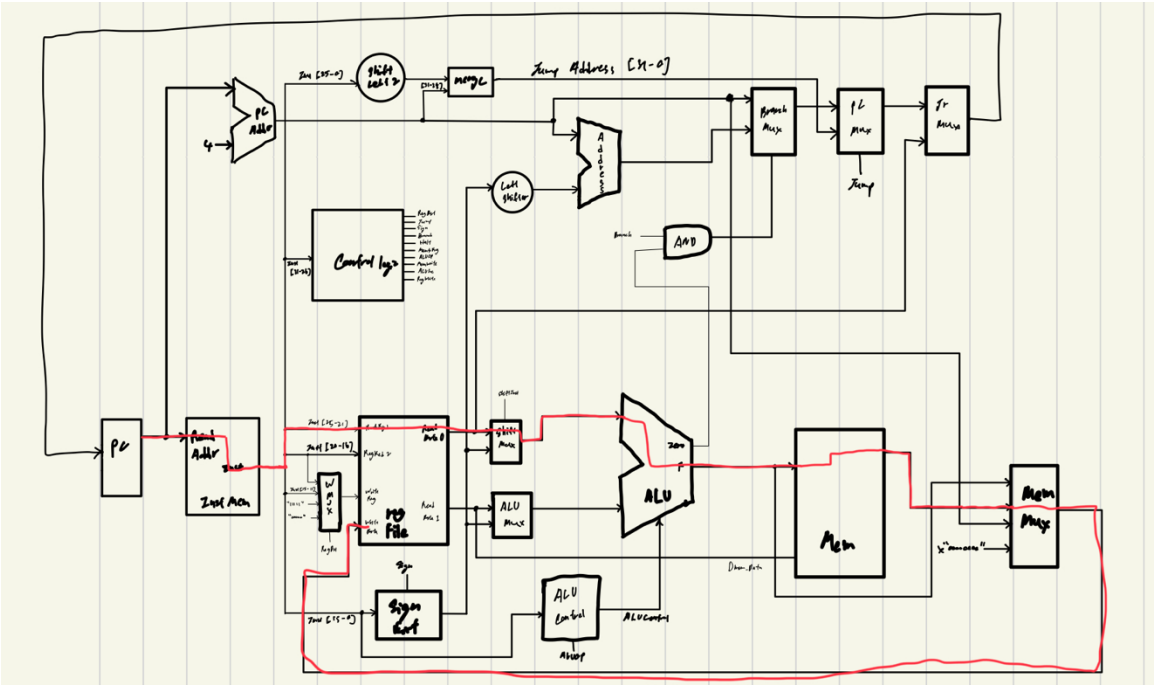




[Part 4] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic from part 1. What components would you focus on to improve the frequency?

The max frequency: 25.40 mhz

Critical Path:



I would focus on my ALU design to improve the frequency because my ALU will first compute all the result for every MIPS instruction with the two input data and select the

output according to the ALUControl. I could make the ALU only compute the MIPS instruction according to the ALUControl.