JoSDC'24

المسابقة الوطنية لتصميم الشرائح الإلكترونية

Jordan National Semiconductors Design Competition

Qualifying Phase Report - JoSDC'24

Team Name		
minimum electric design		
Team Members		
#	Name	Email
1	Feras Mohammad	fra0208687@ju.edu.jo
2	Omar Sweiss	amr0215904@ju.edu.jo
3	Amr Abuzayyad	amr0214597@ju.edu.jo
4	Abdallah Shbilat	abd0205000@ju.edu.jo
5	Marlo Rizkallah	mar2211499@ju.edu.jo

- ## Summary

Total number of bugs found	10
Total number of bugs fixed	10

- ## Corrected errors.

Bug Title:	ALUOp parameters in the ALU.v file.		
Bug ID:	1	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The parameters for the _AND and _ADD instructions were switched.		
Steps to reproduce	assign the _AND parameter to 3'b000 and the _ADD parameter to 'b010 in the ALU.v File to reproduce the bug.		
Expected Behavior	ALU should perform addition when receiving opSel with value = 000 and should perform bitwise AND when receiving opSel value = 010. The simulation showed contradictory behavior.		
Actual Behavior	ALU performs bitwise AND when receiving opSel with value = 000, and performs addition when receiving opSel with value = 010		
Solution implemented	Assigned the _AND parameter to 3'b010 and the _ADD parameter to 3'b000 in the ALU.v file.		

Bug Title:	Missing default case in the ALU.v File		
Bug ID:	2	Bug Type	Performance
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	In the opSel case statement in the ALU.v File, the synthesizer creates a latch due to the missing default statement.		
Steps to reproduce	Remove the default case statement and assign an opSel that is not defined within the parameters.		
Expected Behavior	Module synthesis should not produce any latches.		
Actual Behavior	Module works as intended but produces latches when synthesizing.		
Solution implemented	Added a default case statement.		

Bug Title:	Mux size in the mux2x1.v File		
Bug ID:	3	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The size of the multiplexer is larger than the parameter size by 1 due to incorrect assignment.		
Steps to reproduce	Assign the input and output of the mux as [size:0]. Using this mux in the simulation will show a single high impedance (Z) signal on the most significant bit.		
Expected Behavior	Multiplexer should not show a high impedance signal in the simulation if the inputs are correct.		
Actual Behavior	Multiplexer shows a high impedance signal in the output due to it being an incorrect size (bigger by 1)		
Solution implemented	Change the size of the input and output to [size-1:0].		

Bug Title:	WBMux inputs inverted in processor.v File.		
Bug ID:	4	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The inputs of the WBMux (memoryReadData, ALUResult) were inverted.		
Steps to reproduce	Assign in1 to memoryReadData and in2 to ALUResult. During simulation, instructions which require the WBMux to select the ALU result will select the Data memory instead and vice versa.		
Expected Behavior	Instructions which require the WBMux to select the ALU result should select the ALU result and instructions which require the data memory to be selected should select it.		
Actual Behavior	Instructions which require the WBMux to select the ALU result, selected the Data Memory instead, and vice versa.		
Solution implemented	Assign in1 for ALU Result, and in2 for Data Memory.		

Bug Title:	Incorrect bit position assignment for rt, rs, rd in processor.v File.		
Bug ID:	5	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	rd was assigned to bits [25:21] instead of rs. rs was assigned to bits [20:16] instead of rt. And rt was assigned to bits [15:11] instead of rd in the processor.v File.		

Steps to reproduce	Use these assignments for rd,rs,rt:
	
	assign rd = instruction[25:21];
	
	assign rs = instruction[20:16];
	
	assign rt = instruction[15:11];		

Expected Behavior	rs should always be the source register, while rd should be the destination register and in R-Type instructions, rt acts a source register, and in I-Type instructions it acts as a destination register.		
Actual Behavior	rs was behaving as the rt register, while rd was behaving as the rs register and rt was behaving as the rd register.		

Solution implemented	Changed rd, rs, rt to these following assignments:
	
	assign rs = instruction[25:21];
	
	assign rt = instruction[20:16];
	
	assign rd = instruction[15:11];		

Bug Title:	incorrect declaration for writeRegister in the processor.v File		
Bug ID:	6	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The output of RFMux was named WriteRegister. but the module was named writeRegister (Capitalization).		
Steps to reproduce	In the processor.v file, name the output of the RFMux as WriteRegister instead of writeRegister.		
Expected Behavior	In the simulation it should show the signal that the mux selects, not a high impedance signal.		
Actual Behavior	A high impedance signal shows in the simulation.		
Solution implemented	Change the name from WriteRegister to writeRegister.		

Bug Title:	funct case statement in the controlUnit.v File.		
Bug ID:	7	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	In the funct case statement, _or_ was assigned to 011 in the decimal representation, not binary.		
Steps to reproduce	Assign ALUOp = 3'd011, instead of ALUOp = 3'b011.		
Expected Behavior	Bitwise or instructions should work properly.		
Actual Behavior	Bitwise or instructions do work properly because the first 3 bits in the binary representation of 11 in the decimal representation is 011. which is as intended for the bitwise or operation.		
Solution implemented	Changed the _or_ case from ALUOp = 3'd011 to ALUOp = 3'b011		

Bug Title:	Control signals for the lw instruction in the controlUnit.v File.		
Bug ID:	8	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The control signals for the RegDst, MemReadEn, MemWriteEn were incorrect.		
Steps to reproduce	Assign RegDst= 1, MemReadEn = 0, MemWriteEn = 1.		
Expected Behavior	lw instructions should read from the memory and store in the rt register.		
Actual Behavior	lw instructions write on the memory and store in the rs register which does not exist in I-Type instructions.		
Solution implemented	Assign RegDst= 0, MemReadEn = 1, MemWriteEn = 0.		

Bug Title:	Control signals for the beq instruction in the controlUnit.v File		
Bug ID:	9	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The ALUSrc control signal was assigned to 1 instead of 0 for the beq instruction.		
Steps to reproduce	Assign ALUSrc = 1 for the beq instruction.		
Expected Behavior	ALU should perform subtraction on the two registers and produce the zero flag if they are equal.		
Actual Behavior	ALU performs subtraction between the first register and the sign extended immediate value.		
Solution implemented	Change ALUSrc from 1 to 0.		

Bug Title:	Slt operand in ALU.v File		
Bug ID:	10	Bug Type	Logical
Reported by:	minimum electric design	Open Date	29/9/2024
Assigned to:	minimum electric design	Close date	3/10/2024
Description	The result for the SLT function was assigned as operand2 < operand1 (incorrect order)		
Steps to reproduce	_SLT: result = (operand2 < operand1) ? 1: 0;		
Expected Behavior	SLT should give produce an output of 1 only if rs < rt and zero otherwise.		
Actual Behavior	SLT gives an output of 1 only if rt < rs and zero otherwise.		
Solution implemented	_SLT: result = (operand1 < operand2) ? 1 : 0; Switched operand 1 & 2.		

- **Full Analysis**

Table 2 Control Unit Analysis Table

Instruction	RegDst	Branch	MemReadEn	MemToReg
ADD	1	0	0	0
ADDI	0	0	0	0
SUB	1	0	0	0
AND	1	0	0	0
OR	1	0	0	0
SLT	1	0	0	0
LW	0	0	1	1
SW	0	0	0	0
BEQ	0	1	0	0

Instruction	ALUOp	MemWriteEn	ALUSrc	RegWriteEn
ADD	000	0	0	1
ADDI	000	0	1	1
SUB	001	0	0	1
AND	010	0	0	1
OR	011	0	0	1
SLT	100	0	0	1
LW	000	0	1	1
SW	000	1	1	0
BEQ	001	0	0	0

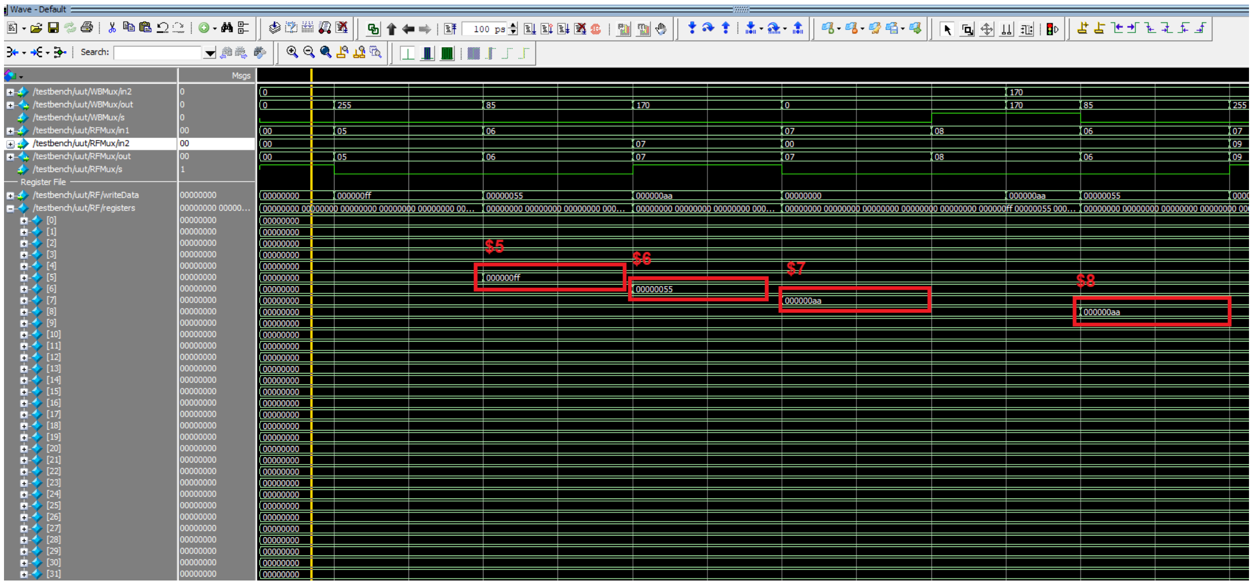
- **Functional Testing**

Table 3 Functional Testing Benchmark 1

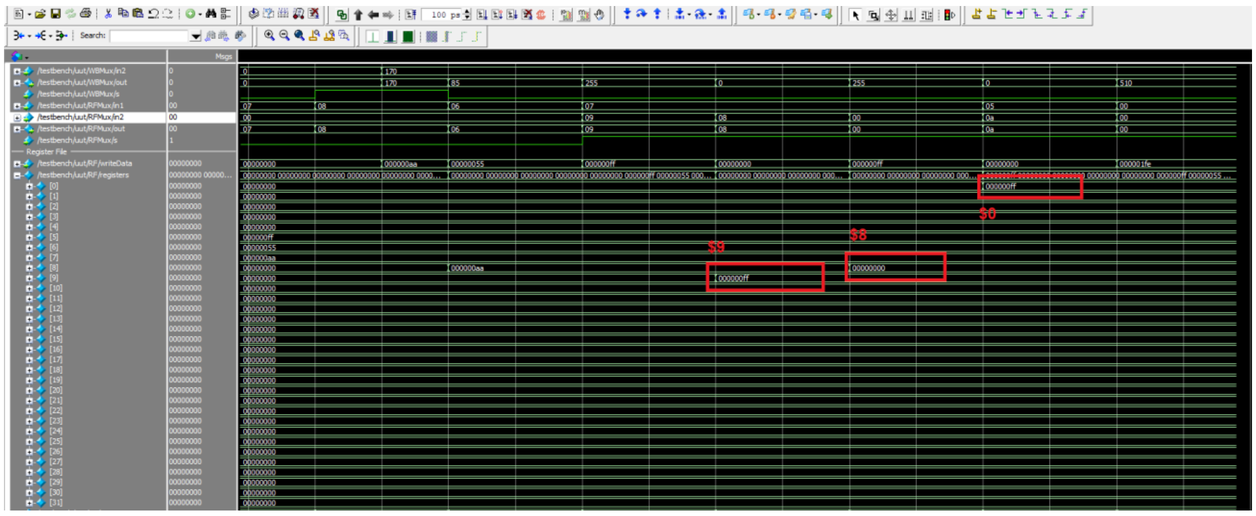
#	Instruction	Hexadecimal (Machine Code)	Result
1	addi $5, $0, 0xff	0x200500FF	$5 = 0x000000ff
2	addi $6, $0, 0x55	0x20060055	$6= 0x00000055
3	sub $7, $5, $6	0x00A63822	$7 = 0x000000AA
4	sw $7, 0x0($0)	0xAC070000	DataMemory[0] = 0x000000AA
5	lw $8, 0x0($20)	0x8E880000	$8 = 0x000000AA
6	beq $6, $7, fin	0x10E60003	branch not taken
7	or $9, $6, $7	0x00C74825	$9 = 0x000000FF
8	and $8, $6, $7	0x00C74024	$8 = 0x00000000
9	add $0, $6, $7	0x00C70020	$0 = 0x000000FF
10	fin : slt $10, $0, $5	0x0005502A	$10 = 0x00000000

To check that our debugging was successful we simulated our design with the given instructions, as can be seen, the behavior shown by the design conforms to the expected behavior that was written in the table above.

First, we can see the results of the first five instructions reflected on the registers $5, $6, $7 and $8.



Second, after the instruction on $8 register, we can observe that the beq instruction was indeed not taken; due to the $9's value changing to 0xFF as previously expected.



- **Performance Results (Optional)**

Table 4 Performance Data

		Metric	Value	Description
		Clock Frequency	34.48 MHz	clock frequency you configured in the Quartus tool.
		Design Size (LE)	1894	Size of design in terms of logic elements
model	Slow 85C	Fmax	35.1MHz	Fmax : The highest frequency at which the processor can operate reliably. Setup Time : The time required to set up signals before the clock edge. Hold Time : The minimum time signals must remain stable after the clock edge.
		Setup Slack	0.255	
		Hold Slack	0.783	
	Slow 0C	Fmax	38.18MHz	
		Setup Slack	1.403	
		Hold Slack	0.724	
	Fast 0C	Fmax	84.93MHz	
		Setup Slack	8.613	
		Hold Slack	0.307	

The commands in the SDC file:

```
// Time Information

set_time_format -unit ns -decimal_places 3


// Create Clock

create_clock -name {clk} -period 29.000 -waveform { 0.000 14.500 } [get_ports { clk }]




// Set Clock Uncertainty

set_clock_uncertainty -rise_from [get_clocks {clk}] -rise_to [get_clocks {clk}]    0.020

set_clock_uncertainty -rise_from [get_clocks {clk}] -fall_to [get_clocks {clk}]    0.020

set_clock_uncertainty -fall_from [get_clocks {clk}] -rise_to [get_clocks {clk}]    0.020

set_clock_uncertainty -fall_from [get_clocks {clk}] -fall_to [get_clocks {clk}]    0.020
```

- **Free Space**

1. Challenges Faced and Solutions:

One major challenge was identifying the logical bugs, as they required thorough understanding of the original design to enable us to be able to notice them. Most of the bugs were tricky to notice. For example, the _or_ assignment being in decimal and not in binary; the expected behavior doesn't change.

2. Debugging Strategies:

We used the testbench to simulate our design on modelsim, where we found several inconsistencies with the expected behavior of the design and marked them. Then we would do several more tests to accurately pinpoint the root cause. A decent example of this would be the ADD and AND instructions where we noticed that they were switched when we noticed the unexpected outputs.

Incremental testing was also utilized, where we tested the modules individually and discerned the incorrect outputs of the modules. For example, we found the slt bug through this method.

3. Testing Approach:

We used a combination of functional simulation and waveform analysis to verify that each component worked as expected using the modelsim tool.

4. Additional information:

We created a simple python script to turn the assembly instructions to machine code to save time and avoid the hassle of figuring out the machine code for all instructions.

The script:

```
def int_to_binary(num, num_bits):
    if num >= 0:

        binary = format(num, f'0{num_bits}b')
    else:

        binary = format((1 << num_bits) + num, f'0{num_bits}b')

    return binary

def hex_to_bin(hex_num, num_bits):

    decimal_num = int(hex_num[2:], 16)

    return int_to_binary(decimal_num, num_bits)

def assemble(inp):
  parts = inp.split()
  opcode = parts[0]
  rtype = ['add','sub','and','or','slt']
  itype = {'sw':'101011', 'lw':'100011', 'addi':'001000', 'beq':'000100'}
  jtype = {'jump': '001100', 'jr':'001110'}
  if opcode in rtype:
    _,dest,src1, src2,  = parts
    opcode_binary = format(0b000000,'06b')
    src1_binary = format(int(src1[1:]), '05b')
    src2_binary = format(int(src2[1:]), '05b')
    dest_binary = format(int(dest[1:]), '05b')
    shamt = format(0b00000, '05b')
    funct = format({'add': 0b100000, 'sub': 0b100010, 'and': 0b100100, 'or':
0b100101,'slt': 0b101010}[opcode], '06b')
    machine_code =
f"{opcode_binary}{src1_binary}{src2_binary}{dest_binary}{shamt}{funct}"
    return machine_code

  elif opcode in itype:
    _,dest,src1, imm = parts
    opcode_binary = itype[opcode]
    src1_binary = format(int(src1[1:]), '05b')
    dest_binary = format(int(dest[1:]), '05b')
    imm_binary = hex_to_bin(imm, 16)
    machine_code = f"{opcode_binary}{src1_binary}{dest_binary}{imm_binary}"
    return machine_code
  elif opcode in jtype:
    if opcode == 'jump':
      _,  imm = parts
      opcode_binary = jtype[opcode]
      imm_binary = hex_to_bin(imm, 26)
      machine_code = f"{opcode_binary}{imm_binary}"
      return machine_code
    else:
      _,  src1 = parts
      opcode_binary = jtype[opcode]
      src1_binary = format(int(src1[1:]), '05b')
      padding = format(0b0, '021b')
      machine_code = f"{opcode_binary}{src1_binary}{padding}"
      return machine_code
  else:
    return 'instruction not found'

instructions = """addi $5 $0 0xff
addi $6 $0 0x55
sub $7 $5 $6
sw $7 $0 0x00
lw $8 $20 0x00
beq $6 $7 0x03
or $9 $6 $7
and $8 $6 $7
add $0 $6 $7
slt $10 $0 $5
addi $11 $11 0x20 """
i = 0
  print(f"{i} : {assemble(instruction)};")
  i += 1
```