Rene Sanchez A11866286
Shiyao Liu A10626758
CSE141L Lab 2

Q1:
Make sure that you have read and understood the code before answering these questions.

- **The opcode space in the Vanilla instruction format is fixed. How many more instructions beyond the basic instruction set can be supported?**
- There are currently 22 instructions in use. The documentation states that we will use a
- 5-bit opcode. 2^5 = 32 possible opcode values.
- So we have 32 – 22 = 10 new instructions we can add.
- **Add a left bitwise rotate instruction (ROL) to alu.sv and add the mapping to definitions.sv (HINT: refer to the manual and understand why the bitwise shift instructions are defined the way they are.**
- In alu.sv: kROL:
- result_o = ((rd_i) << rs_i[4:0]) | ((rd_i) >> (32 - rs_i);
-
- Logic behind this:
- ((rd_i) << rs_i[4:0]) : is the normal shift left
- ((rd_i) >> (32 - rs_i) : moves the MSB bits of rd_i to the LSB
- OR'ing these two values gives allows us to move the original MSB bits of rd_i to their LSB.

Q2:
Make sure that you have read and understood the code before answering these questions.

- **The Vanilla core has a separate memory for data and instructions. What are the advantages of this type of architecture?**
- It helps us avoid problems where memory from one section shifting into other memory sections. For example, the overflow of an add might be placed into an area of memory that is meant for other data.
-
- It is also more efficient to have them separated since we can read and write to different parts of memory at the same time. So it's better for pipelining.
-
- **What is the maximum size of an assembly program in terms of # of instructions?**
- As shown in instr_mem.sv, by the line:
- instruction_s [0:(2**addr_width_p)-1] mem;

- the max size is 2^(address width).

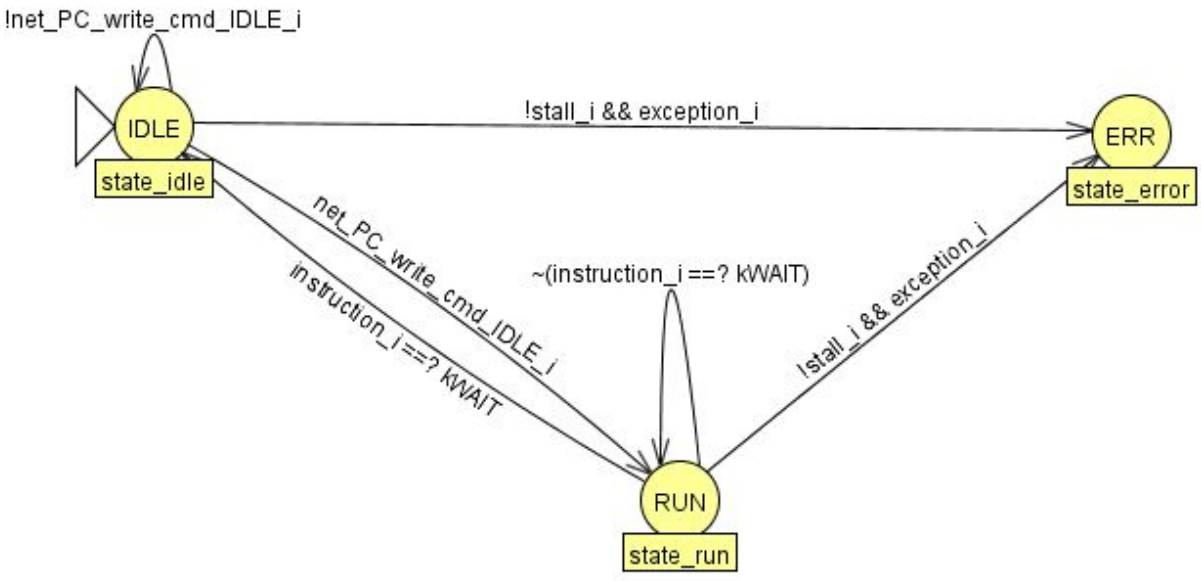
- **Is reading from the instruction memory synchronous?**
- Yes, it waits for the clock edge.

Q3:
How many cycles will an operation that accesses data memory take in the best case?
As noted at the very bottom of data_mem.sv, there is no delay for this memory. As soon as it gets the valid signal from the core memory, the data is yumi'd and thus ready to be stored.

Therefore, a single request + reply can occur within a **single cycle.**

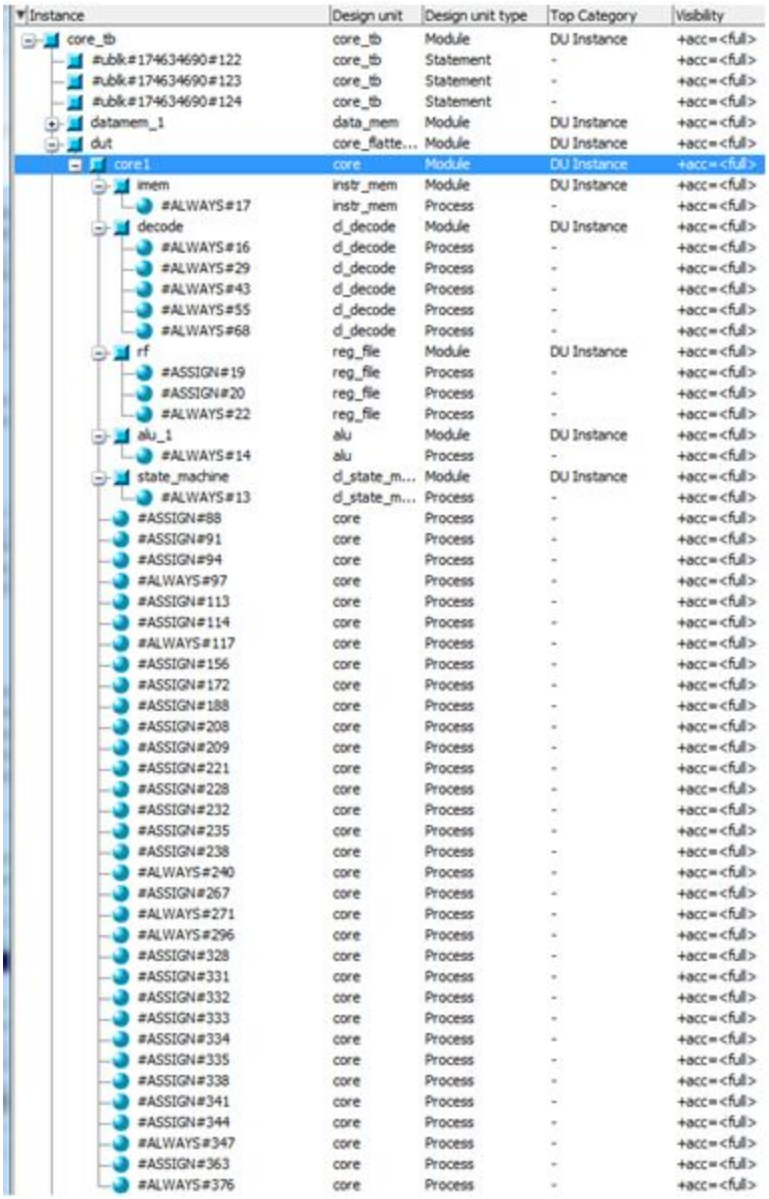
Q4:
Make sure that you have read and understood the code before answering these questions (a few of the details of how the network interfaces to the core may be unclear at this point).

- **Draw a state machine for the Vanilla core, including the signals that cause a state transition.**



- **When does the execution of the core stall?**
- As noted in core.sv:
- assign stall = stall_non_mem || (mem_stage_n != DMEM_IDLE) || (state_r != RUN);
-

- So it stalls whenever either of those 3 statements is true.
-
-
- **Draw a module hierarchy for the Vanilla core.**
- (Taken from ModelSim, upon doing a run –all)



Instance	Design unit	Design unit type	Top Category	Visibility
core_tb	core_tb	Module	DU Instance	+acc=<full>
#ublk#174634690#122	core_tb	Statement	-	+acc=<full>
#ublk#174634690#123	core_tb	Statement	-	+acc=<full>
#ublk#174634690#124	core_tb	Statement	-	+acc=<full>
datamem_1	data_mem	Module	DU Instance	+acc=<full>
dut	core_flatte...	Module	DU Instance	+acc=<full>
core1	core	Module	DU Instance	+acc=<full>
imem	instr_mem	Module	DU Instance	+acc=<full>
#ALWAYS#17	instr_mem	Process	-	+acc=<full>
decode	d_decode	Module	DU Instance	+acc=<full>
#ALWAYS#16	d_decode	Process	-	+acc=<full>
#ALWAYS#29	d_decode	Process	-	+acc=<full>
#ALWAYS#43	d_decode	Process	-	+acc=<full>
#ALWAYS#55	d_decode	Process	-	+acc=<full>
#ALWAYS#68	d_decode	Process	-	+acc=<full>
rf	reg_file	Module	DU Instance	+acc=<full>
#ASSIGN#19	reg_file	Process	-	+acc=<full>
#ASSIGN#20	reg_file	Process	-	+acc=<full>
#ALWAYS#22	reg_file	Process	-	+acc=<full>
alu_1	alu	Module	DU Instance	+acc=<full>
#ALWAYS#14	alu	Process	-	+acc=<full>
state_machine	d_state_m...	Module	DU Instance	+acc=<full>
#ALWAYS#13	d_state_m...	Process	-	+acc=<full>
#ASSIGN#88	core	Process	-	+acc=<full>
#ASSIGN#91	core	Process	-	+acc=<full>
#ASSIGN#94	core	Process	-	+acc=<full>
#ALWAYS#97	core	Process	-	+acc=<full>
#ASSIGN#113	core	Process	-	+acc=<full>
#ASSIGN#114	core	Process	-	+acc=<full>
#ALWAYS#117	core	Process	-	+acc=<full>
#ASSIGN#156	core	Process	-	+acc=<full>
#ASSIGN#172	core	Process	-	+acc=<full>
#ASSIGN#188	core	Process	-	+acc=<full>
#ASSIGN#208	core	Process	-	+acc=<full>
#ASSIGN#209	core	Process	-	+acc=<full>
#ASSIGN#221	core	Process	-	+acc=<full>
#ASSIGN#228	core	Process	-	+acc=<full>
#ASSIGN#232	core	Process	-	+acc=<full>
#ASSIGN#235	core	Process	-	+acc=<full>
#ASSIGN#238	core	Process	-	+acc=<full>
#ALWAYS#240	core	Process	-	+acc=<full>
#ASSIGN#267	core	Process	-	+acc=<full>
#ALWAYS#271	core	Process	-	+acc=<full>
#ALWAYS#296	core	Process	-	+acc=<full>
#ASSIGN#328	core	Process	-	+acc=<full>
#ASSIGN#331	core	Process	-	+acc=<full>
#ASSIGN#332	core	Process	-	+acc=<full>
#ASSIGN#333	core	Process	-	+acc=<full>
#ASSIGN#334	core	Process	-	+acc=<full>
#ASSIGN#335	core	Process	-	+acc=<full>
#ASSIGN#338	core	Process	-	+acc=<full>
#ASSIGN#341	core	Process	-	+acc=<full>
#ASSIGN#344	core	Process	-	+acc=<full>
#ALWAYS#347	core	Process	-	+acc=<full>
#ASSIGN#363	core	Process	-	+acc=<full>
#ALWAYS#376	core	Process	-	+acc=<full>

- **In MIPS the program counter advances by four i.e. PC + 4, if a branch does not occur. What is the step size for the Vanilla core?**
- The pc_plus1 logic variable implies that there is a single step.
- So just like in normal MIPS, the Vanilla core *should* have a step size of 1.
- Note: 1 step = 16 bits.

Q5:

- **Add the left bitwise rotate instruction (ROL) to the assembler.**
- Added 'ROL' to the opcodes list.
- Gave it an opcode values:
- opcodeTable.put("ROL" , "11101");
- Same as in the definitions.sv
-
- And also added it to the case statement switch.
-
-
- **Write a unit test for the new instruction in the style of the other tests in tester.asm, add it to the tester.asm file, and assemble it.**
- // Test for ROL

```
.const %ROL_ANS     ,  0x1ffffffe
.const %ROL_ORIGINAL ,  0x0fffffff
.const %ROL_SHIFT_BY ,  0x00000001
ADDU  $R5, %ONE
MOV   $R7, %ROL_ORIGINAL
MOV   $R2, %ROL_SHIFT_BY
ROL   $R7, $R2
MOV   $R6, $ROL_ANS
JALR  $R1, %CHECK
```

- **Run the tester.asm program on the simulator and verify that functional (aka behavioural) simulation is successful**
- **List the reported Fmax of your design before and after adding ROL.**

Before: Fmax: 55.53 MHz
After: Fmax = 53.6MHz

- **List the cycle time of your design before and after adding ROL. Show your work.**

Before: Cycle time: 1 / 55.53MHz = 18.008 ns

After: Cycle time = 1 / 53.6MHz = 18.657 ns

- **List the number of registers used, the number of combinational functions used, and the number of memory bits used before and after adding ROL.**

Before:
Registers: 2,069
Comb. Functions: 3,900
Memory bits: 16,384

After:
Registers: 2,069
Comb. Functions: 4,048
Memory bits: 16,384

Part 2

Q6:
Up to this point we have given you the scripts to compile and simulate your designs in ModelSim. Use these as a reference to fill in the compile.tcl and sim.tcl scripts to compile and start the simulations for simple_core_tb. Note that this must work for our auto-grader to run your code!

Compile.tcl

```
# Compile .sv files.
vlog -work work "../simple_definitions.sv"
vlog -work work "../simple_alu.sv"
vlog -work work "../simple_core.sv"
vlog -work work "../simple_imem.sv"
vlog -work work "../simple_reg_file.sv"
vlog -work work "simple_core_tb.sv"
```

sim.tcl

```
vsim work.simple_core_tb

#                  Group Name                Radix             Signal(s)
add wave    -noupdate   -group {simple_core}        -radix hexadecimal  /dut/*
```

Q7:
Add the both pipecuts to your design as described above and once your code is in a working state (passes simple_core_tb), answer the following questions:

- **What signals must you pass through across the first pipecut? The second?**
  First pipecut:
  The instruction_s instruction (specifically, instruction.rs and instruction.rd).

  Second pipecut:
  rs_val
  rd_val
  instruction, but delayed by 2 cycles. We do the following:

```
always_ff@(posedge clk) begin
        inst_regOne <= instruction;
        inst_regTwo <= inst_regOne;
end
```

and we pass inst_regTwo to the ALU.

- **List the number of registers used, the number of combinational functions used, and the number of memory bits used before and after adding your pipecuts.**
  Before:
  Registers: 37
  Comb. functions: 27
  Memory bits: 0

  After:
  Registers: 53
  Comb. functions: 29
  Memory bits: 0
- **List the reported Fmax of your design before and after adding your pipecuts.**
  Before: 259.88 MHz

  After: 288.85 MHz
- **List the cycle time of your design before and after adding your pipecuts. Show your work.**
  Before: 1/259.88 MHz = 3.87 ns

  After: 1 / 288.85 Mhz = 3.46 ns

Original time: 361,892 ns
Old cycles: 361,890
Old instruction count: 350,961

Time to beat: 180,946 ns

Q8:

- **Describe all the new instructions that your team has implemented.**
  1. BXOR: Bitwise XOR.
  2. ROR: Bitwise ROR.
  3. ANOT: Bitwise AND with the first register being notted (~A & B)
  We also optimized CH, MAJ, SmallSig and BigSig assembly functions by removing MOV
  statements that were unnecessary after implementing the above instructions.

Q9:

- **List the reported Fmax of your design before and after optimization.**
  Before: 53.6 MHz

  After: 48.98 MHz
- **List the cycle time of your design before and after optimization. Show your work.**
  Before: 1/53.6 MHz = 18.65 ns

  After: 1/ 48.98 MHz = 20.41 ns
- **List the number of registers used, the number of combinational functions used, and the number of memory bits used before and after optimization.**
  Before:
  Registers: 2069
  Comb functions: 4048
  Memory bits: 16384

  After:
  Registers: 2069
  Comb functions: 4455
  Memory bits: 16384

Q10:

- **How many cycles did it take to find a bitcoin before and after optimization?**
  Before: 361,890

  After: 179,874

- **What was your hash rate in hashes per cycle before and after optimization?**
  Before: 361,890/9 = 40210

  After: 179,874/9 = 19986

Q11:

- **How many instructions did it take to find a bitcoin before and after optimization?**
  Before: 350,961

  After: 168,945
- **What was your hash rate in hashes per instruction before and after optimization?**

Before: 350,961/9 = 38995.66

After: 168,945/9 = 18771.66

Q12:

- **What is the execution time (# of cycles x cycle time) to find a bitcoin before and after optimization?**
  Before: 18.65*361890 = 6749248.5

  After: 179,874*20.41 = 3671228.34
- **What was your hash rate in hashes per second before and after optimization?**
  Before: 361,892 ns/9 = 0.000361892 seconds /9 = 0.00004

  After: 179876ns /9 = 0.0001798 /9 = 0.000019
- **What is your speedup (baseline execution time/optimized execution time)?**
  361,892 ns / 179876ns = 2.011