CDA 4205L Lab #7: ALU & Control Unit Design

University of South Florida

Lab Date: Feb. 26

Report due: One week after the lab section, see Canvas

***This is a group lab**. One submission per group is sufficient.*

This lab does not use Vivado!

Welcome to CDA 4205L Lab #7! The goal of this lab is to help you understand the inner workings of an Arithmetic Logic Unit (ALU) and Control Unit (CU) modules with a basic implementation in Verilog.

Prelab

Arithmetic Logic Unit (ALU)

As the name implies, the ALU is responsible for the majority of the arithmetic and logic operations in a processor. This can include basic binary operations, such as: addition, subtraction, shifting, AND/OR/XOR, comparators, etc.

In RISC-V, the ALU normally operates on 32-bits or 64-bits data. For example: for 64-bit ALU, it takes in 2x 64-bit inputs and produce a 64-bit output. Depending on the microarchitecture, other arithmetic operations (such as multiply and divide, which can take multiple cycles) can also be part of the ALU, or can be in a separate functional unit. This lab will use a similar version of RV32IM standards (RISC-V 32-bits integer ISA with multiply and divide).

For the basic ALU, you can think of it as a `case` statement (similar to a switch in C/C++), or a large `if`/`else`. The output of the ALU will depend on the `operation` control signals. Recall that the `operation` control signals are all in some way derived from the instruction itself. In RISC-V this comes from the `opcode`, `funct3`, and `funct7` fields.

Control Unit (CU)

The CU is responsible for decoding the `opcode` field in set to 0 or 1 the respective CPU control lines. These control lines, as the name implies, controls the behavior of the CPU. For this lab, there is a total of 10-bits of control lines:

- *ALUSrc* (1-bit): it indicates where the 2nd ALU operand comes from: (0) Reg2 (rs2) or (1) the sign-extended immediate.
- *MemToReg* (1-bit): it indicates which data to use to write (if enabled) the destination register (rd) with: (0) thenALU result or (1) the data memory
- *RegWrite* (1-bit): it enables the write operation for the destination (rd)
- *MemRead* (1-bit): it enables the read operation for the data memory for the respective address (= ALU Result)

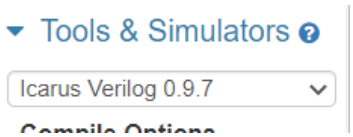
- *MemWrite* (1-bit): it enables the write operation for the data memory for the respective address (= ALU Result) with the value from Reg2 (rs2).
- *Branch* (1-bit): if the branch condition is met, then set PC=PC+Imm
- *Jump* (1-bit): if this instruction is "jal" or "jalr", then write PC+4 to destination register
- *Jalr* (1-bit): if this instruction is "jalr", thus sets PC=Reg1+Imm
- *ALUOp* (2-bit): used by the ALU Control module to assist in determining the correct ALU `operation` control signal:
    - "00" is reserved to indicate load, store, jalr, and jal instructions.
    - "01" is reserved to indicate branches and comparison instructions.
    - "10" is reserved to indicate branches arithmetic, shift, and bitwise boolean logic instructions.
    - "11" is unused for "RV32IM" ISA standard.

Lab

Setup

1. Go to https://edaplayground.com/. Since you're just doing ALU and CU designs, and not performing synthesis/implementation (to gather power-performance-area or *PPA* results), we will *not* be using Vivado for this lab.

2. From the left panel, select "Icarus Verilog 0.9.7" as the simulator from the "Tools & Simulators" menu.

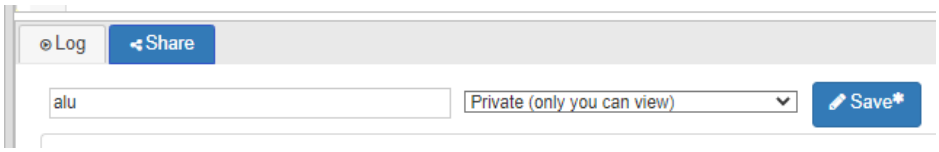


3. On the right-hand side, you will create the Verilog module representing the ALU (part 1) or CU (part 2) design. On the left, you will create the *testbench* for each part.

   Note: you will have to create 2 different tabs: one for your Part 1 design and another for Part 2. Do NOT combine the design and testbenches. For each tab you will have to follow these same setup steps (1-3).

Part 1: ALU Design

4. For part 1 you will be designing an ALU. First, create a tab following the setup steps (1-3). Make sure to write the name of your project as "alu", set to "Private", and then **save**. If it asks you to register or sign-in, use your USF email but with a different password.



5. Refer to the ALU operations list below (obtained from your assembler labs) for part 1:

Category	Operation	Name	Meaning
Arithmetic	0010	add	A + B
	0110	sub	A – B
	0111	mul	A * B
	1111	div	A / B
Shift	0100	sll	A << B
	1100	srl	A >> B
	1110	sra	A >>> B
Bitwise Boolean Logic	0000	and	A & B
	0001	or	A | B
	0011	xor	A ^ B
Comparison	1000	eq	A == B
	1001	neq	A != B
	1010	lt	A < B
	1011	geq	A >= B

6. Your module definition should take a 4-bits `operation` input, two 32-bits operands inputs (`A` and `B`), and one 32-bit `ALUResult` output register (`reg`). Set all these values as `signed`.

   Note: Your module should also have a `` `timescale `` directive to allow for proper simulation. Add `` `timescale 1ns/1ns `` to the top of your module. (This means that the simulator runs in 1ns increments, and the smallest step is 1ns).

```
// Code your design here
`timescale 1ns / 1ns

module alu(
  // Inputs
  input signed [3:0]  operation,
  input signed [31:0] A,
  input signed [31:0] B,

  // Output
  output reg signed [31:0] ALUResult
);


  /* circuit design goes here */


endmodule

```

7. You should use the Verilog `case` statement to structure your ALU logic. The `case` statement should be inside an `always` block.

```
15  always @(*) begin
16    case(operation)
17
18      /* case statements goes here */
19
20    endcase
21  end
```

8. Fill in the `case` statement with all possible ALU operations.

   For example, the syntax for the first operation (addition) would be:
   4'b0010: <operation when operation = 0010>
   Also, the default case should set ALUResult = 0. See example below for “add” and “default” cases:

```
15  always @(*) begin
16    case(operation)
17
18      4'b0010: ALUResult = A + B;
19
20      // insert the other operation cases here
21
22      default: ALUResult = 0;
23
24    endcase
25  end
```

Once you have designed the ALU, you will need to create a testbench. In EDA Playground, the testbench is written on the left-hand window.

9. Your testbench should be called `module alu_testbench();` (a testbench does not have any inputs or outputs). For every `input` to your ALU, declare a corresponding `reg`. For every `output` in your ALU, declare a corresponding `wire`. See example below:

```
// Code your testbench here
`timescale 1ns / 1ns

module alu_testbench();

  // ALU Inputs
  reg signed [3:0] operation;
  reg signed [31:0] A;
  reg signed [31:0] B;

  // ALU Output
  wire signed [31:0] ALUResult;



endmodule
```

10. Instantiate the circuit you are testing (unit under test). The syntax is:

    <name of your module> <instance name> (<port list>);

    In this case,

```
  alu u0 (operation, A, B, ALUResult);
```

    Note: the "instance name" can be anything but not the same as the module's name. Also, the port list order matter, must match as in the respective module ports.

11. Exercise the inputs to your circuit using an `initial` block. The block should include a `$monitor` directive at the start – this will automatically display the specified values whenever they change.

```
initial begin
  $monitor("%b\t%d\t%d\t%d", operation, A, B, ALUResult);

  /* tests goes here */

end
```

12. Next, set your initial values for `A` and `B` – this can be anything you like. Simply assign a value to `A` and `B`. Below is an example initialization (select any other random number)

```
initial begin
  $monitor("%b\t%d\t%d\t%d", operation, A, B, ALUResult);

  /* Example Initial Values */
  A = 197;
  B = 237;

end
```

13. Change the operation by setting `operation` equal to different codes. Be sure to include a `#delay` at the start of the statement. For example,

```
18  initial begin
19    $monitor("%b\t%d\t%d\t%d", operation, A, B, ALUResult);
20
21    /* Example Initial Values */
22    A = 197;
23    B = 237;
24
25    /* Test #01: add */
26    #5 operation = 4'b0010;
27
28    /* insert a test for each operation here */
29
30  end
```

will delay by `5` **time units** (as defined by the `` `timescale `` directive) and then set `operation` equal to `4'b0010`.

14. Repeat step 13 to exercise all possible ALU operations (see reference table in step 5).
15. Save and press **Run**.

> T1: Take a screenshot of your output when `A` and `B` are both positive values. Confirm that the results are correct.

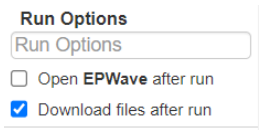
16. Next, change your `A` and `B` to test signed arithmetic. Once again, test these values for all possible operations.

> T2: Take a screenshot of your output when `A` is positive and `B` is negative. Confirm that the results are correct.

> T3: Remove the `signed` modifier from `A` in the ALU module and rerun using the same `A` and `B` values as in T2. Take a screenshot of your output and explain what happens.

> T4: Remove the `signed` modifier from `B` in the ALU module and rerun using the same `A` and `B` values as in T2. Take a screenshot of your output and explain what happens.

17. Once you confirm it is working, select “Download files after run” under “Tools & Simulation”. Then, rerun your code. It should automatically generate a zip file with both the ALU and testbench. Rename this zipfile to “alu.zip”.



Part 2: CU Design

18. For part 2 you will be designing a CU following similar steps as Part 1. First, create a new tab following the setup steps (1-3). Make sure to write the name of your project as "cu", set to "Private", and then **save**. If it asks you to register or sign-in, use your USF email but with a different password.

19. Refer to the CU control lines list below (obtained from your assembler labs) for part 1:

Format	Opcode	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	Jump	Jalr	ALUOp
R-type	0110011	0	0	1	0	0	0	0	0	10
Load	0000011	1	1	1	1	0	0	0	0	00
Jalr	1100111	1	0	1	0	0	0	1	1	00
I-type	0010011	1	0	1	0	0	0	0	0	10
S-type	0100011	1	0	0	0	1	0	0	0	00
B-type	1100011	0	0	0	0	0	1	0	0	01
J-type	1101111	0	0	1	0	0	0	1	0	00

20. Your "cu" `module` definition should take a 7-bits `opcode` input, eight 1-bit `output reg` (`ALUSrc`, `MemtoReg`, `RegWrite`, `MemRead`, `MemWrite`, `Branch`, `Jump`, `Jalr`), and one 2-bits `ALUOp` `output reg`. All of these are unsigned values.

    Note: There is no "unsigned" keyword. Just by not writing `signed`, then it will make them unsigned by default.

21. Take similar steps (6-8) as part 1 to make your CU design. Name this `module` "cu" (all lowercased) and set the correct dimensions/data types of your inputs/outputs, as before.

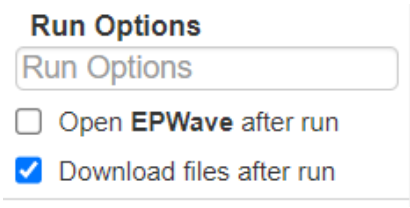
    Note: `begin` and `end` behaves similar to curly braces `{}` in C/C++. Since your `case` statements for the CU design have multiple lines, you will need to wrap each case in a `begin`/`end` block.

```
9   always begin
10    case(opcode)
11
12      7'b0110011: begin
13
14        /* enter code here */
15
16      end
17
18      // insert the other control cases here
19
20    endcase
21  end
```

22. Then, take similar steps (9-15) to create a testbench for your "`cu`" `module`. Name this testbench "`cu_testbench`". Make sure to test all possible "`opcode`" possibilities (see table in step 19).

T5: Take a screenshot of your output with all "opcodes". Confirm that the results are correct.

23. Once you confirm it is working, select "Download files after run" under "Tools & Simulation". Then, rerun your code. It should automatically generate a zip file with both the CU and testbench. Rename this zipfile to "cu.zip".



Questions

24. Answer the following questions:

T6: The `#delay` specified in step 13 was `5` "time units". What is the real value of these `5` "time units" in the simulation?

T7: What is the purpose of a testbench? Why is it needed for each module (ALU and CU)?

T8: Can we obtain power, performance, area (PPA) results with the testbench? Why?

In-class work - Complete **T1, T2, T3, and T4**, show it to the TA for review and submit it on Canvas before the lab session ends. Submit your in-class work as a Word document (.docx) format.

Final Report - Submit your report pdf with answers to all questions and screenshots. Also, submit the final compressed "alu.zip" file (with the ALU module and respective testbench files) and "cu.zip" file (with the CU module and respective testbench files). Thus, total of 3 files in your .zip file (one pdf and two zip files inside). One submission per group.