**CECS 341 – Lab 2A – Simulation of the ALU**

Now that we have familiarized with Verilog and Xilinx, let us go ahead and “build” the Processor discussed in Chapter 4 of the text. It is an ARM Processor (actually toned down LEGv8 Processor). Our first step is to build the Arithmetic and Logic Unit (ALU). The first few pages of Section 4.4 in the Text discuss the ALU.

The ALU receives normally two values A and B, in this case, each 64 bits wide and it has 4 bits of control called ‘ALU operation’ (Figure 4.7b in p.265). It will output the result as ‘ALU result’ (64 bits) and also a 1 bit Zero output if the result is zero. See the Table below. This is similar to the Table on p.271 in the text. We are simulating the LEGv8 ALU for a small **subset** (say only 6) of the original ARMv8 ALU instructions. The actual ARM processor has more than 100 instructions. If there are 6 operations, the ALU operation bits need to be only 3 (to give 8 options) but the book uses 4 bits to have more options. (We will not use all the 4 bit combinations in our simple LEGv8 processor).

64

A

Zero

ALU result

64

B

64

4

ALU operation

|  |  |  |
| --- | --- | --- |
| ALU operation | Function | ALU result |
| 0000 | AND | A & B |
| 0001 | OR | A B |
| 0010 | ADD | A + B |
| 0110 | SUBTRACT | A - B |
| 0111 | PASS INPUT B | B |
| 1100 | NOR | ~ (A B) |

Modify the code given in p. A 36, Figure A.5.15 for the ALU module. For example, the first line should read

“module LEGv8 (ALU operation, A, B, ALU result, Zero);” Change the signal names – ALUctl to ALU operation, ALUOut to ALU result. **When the case is 7**, the ALU result is simply B.

Develop a Verilog Test Fixture for testing it (New Source associated with lab2 project – see lab 1 handout). You may initialize A and B as 64’h5555555555555555 and 64’haaaaaaaaaaaaaaaa. Remember these are antithetical. Results will be easy to check. Apply stimuli at different time slots to “exercise” the ALU’s different functions. Give the 6 possible values for ALU operation and check whether ALU result tallies with theory. You can right click on the signal in the column left to the waveform in the simulation to change the radix from binary to hex. Check the Zero output also. Can you justify the result for the Subtract operation? Work out the subtraction on a piece of paper and explain. (Hint: All computers perform subtraction by 2's complement addition (from first course in digital logic?)

**CECS 341 – Lab 2B – Simulation of the ALU Control Logic**

Objective: We want to simulate the combinational logic that develops the 4 bit output ALU operation with the following inputs - 2 bits of ALUOp and most significant 11 bits from the Instruction (I (31..21) called Opcode. See figure 4.14, p.274 top row. The different combinations of ALUOp and Opcode that should generate the 4 bit ALU operation are given in the table below. Do **not** use the Table on p.273 - Figure 4.13 in the text.

11

Opcode field

4

ALU operation

ALUOp

2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ALUOp\* | | Opcode field | | | | | | | | | | | ALU operation | |
| ALUOp1 | ALUOp2 | I[31] | I[30] | I[29] | I[28] | I[27] | I[26] | I[25] | I[24] | I[23] | I[22] | I[21] |  |  |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0000 | AND |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0001 | OR |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0010 | ADD |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0110 | SUB |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0111 | Pass B |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1100 | NOR |

(The top 4 rows are same as in Table of Figure 4.12 in p.272, though not in the same order)

You may be tempted to use the code given in the appendix (p.A-37, Figure A.5.16) but it has some errors. We need to correct them. The ‘always’ statement is not complete. It should be “always @ (ALUOp, Opcode field)” and then the case statement should follow. The case statements below should have values of the catenation of ALUOp and Opcode field (not just ALUOp). The switch values for the case statements given on the left extreme in decimal such as 32 are wrong. The values are 13 bits strong. After the switch values, the left side of assignment statements such as ALUOp <= 2; should be ALU operation <= 2; (ALU operation is an output, not ALUOp which is an input - remember we do not assign values to inputs but only to outputs).

Example for OR entry:

13’b10\_10101010000: ALU operation <= 1; // OR, The underscore is harmless but improves readability.

Make all the corrections needed. Then develop a Verilog Test Fixture and check whether ALU operation comes out as expected.

-------------------------------------------------------------------------------------------------------------------------------------

\* ALUOp distinguishes the major categories of instructions. As we all know, there are three major categories of instructions - Data Transfer (moving data in and around); Data Manipulation (data processing) and Program Flow Change (jumping around in the code). ALUOp takes the following values:

Data Transfer - 00; Data Manipulation - 10; Program Flow Change - 01; See Figure 4.12 in p.272. Data Manipulation group is also called R-type because they manipulate register operands only in RISC machines. They are also sometimes referred to as ALU Instructions.

Each major category of instructions defined by ALUOp can have many different instructions and they are delineated by the Opcode field. For example, Data Transfer has Load Register from Memory and Store Register to Memory. R type instructions can be And, Or etc. again differentiated by different Opcode fields. Program Flow Change can have jump unconditional, branch on condition etc. again with different Opcode fields.

**CECS 341 – Lab 2C – Simulation of the ALU with its Control Logic.**

Objective: Now we want to **combine** ALU and the Control Logic. We will apply the inputs to the control logic and the ALU and see whether the combined module works right. ALU operation will become wires (it was input in 2A and output in 2B).

64

Zero

64

64

B

A

ALU result

11

2

ALUOp

Opcode field (I31..21)

4

ALU operation

For the combined module, after the module statement with its port list and after declaring the inputs and outputs, declare ALU operation as “wire”. We must “instantiate” the two modules with arbitrary names but with proper connectivity. See below. Lab2b and Lab2a are the arbitrary ‘instance’ names. They could have been anything, say second and first or something else.

module ALUwithControl (ALUOp, Opcode field, A, B, ALU result, Zero);

//Declare inputs and outputs and intermediate ‘wires’

input [1:0] ALUOp;

input [10:0] Opcode field;

input [63:0] A;

input [63:0] B;

output [63:0] ALU result;

output Zero;

***wire [3:0] ALU operation;***

//**Instantiate** the two units

ALUControl Lab2b (ALUOp, Opcode field, ALU operation);

ALU Lab2a (ALU operation, A, B, ALU result, Zero);

endmodule

Develop a Verilog test fixture and check whether ALU result comes out right for different ALUOp and Opcode field.

We may like to see ALU operation also but the simulator by default shows only the inputs and outputs of the combined module. We can add any *intermediate* signal to the waveform by going to the Simulation Hierarchy window (Instances and Processes tab at the bottom) and expanding the + sign next to module name (say, uut). Right click on the signal of interest and choose Add to waveform. The signal will not have any changes yet because we have to rerun the simulation. Click on the Restart Simulation Icon. In the console, type ‘run 1000 ns’ and press enter. Check whether ALU operation became 0, 1 etc.

mc,s17