ENGR xD52: MP b001

Due October 9th 11PM EST

This lab assignment creates the first component of our processors: The ALU. Additionally, it will help you understand the timing constraints of your designs.

Work in groups of 3 or 4.

Summary

This lab creates the first component of our processors: The ALU. Each group will construct ALUs with identical behaviors, but the internal structures will vary based on the design decisions you each make.

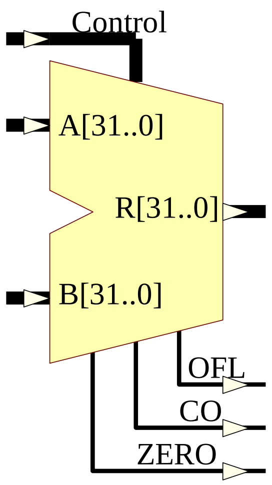
Our ALUs are a subset of the standard MIPS ALU. The number or operations supported has been reduced, but otherwise we are emulating that standard.

Module Definition

```
module  ALU(result, carryout, zero, overflow, operandA, operandB, command);
output[31:0]  result;
output        carryout;
output        zero;
output        overflow;
input[31:0]   operandA;
input[31:0]   operandB;
input[2:0]    command;

// code
endmodule
```

Operation	Result	ALU Command
ADD	R=A+B	b000
SUB	R=A-B	b001
XOR	R=A^B	b010
SLT	R=(A<B)?1:0	b011
AND	R=A&B	b100
NAND	R=~(A&B)	b101
NOR	R=~(A|B)	b110
OR	R=A|B	b111



Work Plan

This subdeliverable is due within 36 hours as an email to comparch14@gmail.com with the subject line formatted as [MP1 Work Plan] Name1 Name2 Name3". Any questions regarding this section should be directed to personal email addresses. Questions sent to the assignment submission address **will not** be read in a timely manner.

Create a work plan for this lab. Break down the lab in to small portions, and for each portion predict how long it will take (in hours) and when it will be done by (date). You will be comparing your predictions to reality later.

Other Deliverables

One representative from each team must submit all deliverables electronically to comparch14@gmail.com as a single compressed archive. Format the email's Header as "[CA] [MP2] Name1 Name2 Name3"

The Writeup

Create a formal write-up of your ALU design. Assume that the primary intended audience is the project manager in charge of the larger CPU into which your design will be incorporated. Convince her that your design is ready to be included and that you have done your due diligence. This PDF document needs to be self supporting and include all text deliverables and associated figures.

The Tests

Construct a (series of) test bench(es) for your ALU. It is highly recommended that you create the tests before you create the ALU.

Be intelligent in your selection of your test cases: Making use of the hierarchy of your design will allow you to avoid exhaustive testing.

For each operation of your ALU, include the following:

1) A written description of what tests you chose, and why you chose them. This should be roughly a paragraph or two per operation.
2) Specific instances where your test bench caught a flaw in your design.
3) As your ALU design evolves, you may find that new test cases should be added to your test bench. This is a good thing. When this happens, record specifically why these tests were added.

The ALU

Construct the ALU described above. It is recommended that use a Bitslice + Control LUT approach to designing your ALU.

Use Structural Verilog for all of your ALU, with the exception of the Control LUT. You may use behavioral Verilog to describe this Look Up Table. See the Notes section below on how to do this.

In order to be slightly more accurate, we will model gate delay as proportional to the number of inputs in the gate. Standardize on 10 units per input. Therefore, an inverter has delay 10, a 32-input NAND gate has delay 320. The basic gates are NAND, NOR, NOT. AND, OR, etc all have 'hidden' inverters that must be accounted for in your propagation delays.

This timing formula is still a horrible horrible lie that Brad will have to fix when you take VLSI. Sorry Brad!

Timing Analysis

Provide the worst case propagation delay for each of the operations of the ALU. This can be calculated or simulated. Note that if you choose to simulate these delays that some operations' prop delay depends heavily on your choice of operands.

Work Plan Analysis

As part of your write-up, compare how long each work unit to how long you predicted it would take. This will help you better schedule future labs.

Hints / Notes

Design Reuse

You may freely reuse code created for previous labs and homeworks. **Be sure to redo the timing!**

Documentation

Code legibility and commenting can be worth up to a third of your grade in this lab. The key is to make sure that your code is "inheritable" – your replacement should be able to understand what you were up to without re-inventing it. Block comments at the top of your modules, occasional intramodule comments describing interesting details.

Overflow

Only ADD and SUB emit overflow signals per the MIPS specification. The basic Boolean operations do not emit an overflow signal for obvious reasons, and during these operations that wire is always false.

The SLT operation is a bit weird here. As discussed, we need to figure out whether or not there was an overflow in the appropriate calculation of SLT. However, in SLT this signal is not propagated outside of the ALU. This is because the processor interprets the overflow signal as an "emergency" situation and begins an exception handling process. Overflows generated during SLT are expected and are not emergencies.

Alternatives to True and False

Sometimes Verilog will show a value for a wire or register as something other than True or False. These extra states may assist you in your debugging and/or analysis.

‘z’ means that the signal does not have a driving source. When you see this, it probably means you didn’t hook up an output correctly. Names are case sensitive.

‘x’ means that Verilog was unable to determine what state the signal is. At the start of your simulations, many signals will be ‘x’ until the propagation delays allow the information to reach that node in the circuit. You can artificially set a signal to ‘x’ as a debugging tool. Be wary of using this to calculate propagation delays: x&0 = 0, not x.

Define

Use the define syntax to make your code cleaner. For example, consider labeling your ALU commands with defines:

```
`define ADD 3'd0
`define SUB 3'd1
`define XOR 3'd2
`define SLT 3'd3
`define AND 3'd4
`define NAND 3'd5
`define NOR 3'd6
`define OR  3'd7
```

LUT Syntax

There are many ways to instantiate a Look Up Table in Verilog. Here is one of them:

```
module ALUcontrolLUT(muxindex, invertB, othercontrolsignal,
ALUcommand);
output reg[2:0]   muxindex;
output reg  invertB;
output reg  othercontrolsignal;
input[2:0]  ALUcommand;

always @(ALUcommand) begin
    case (ALUcommand)
    `ADD:  begin muxindex = 0; invertB=0; othercontrolsignal = ?; end
    `SUB:  begin muxindex = 0; invertB=1; othercontrolsignal = ?; end
    `XOR:  begin muxindex = 1; invertB=0; othercontrolsignal = ?; end
    `SLT:  begin muxindex = 2; invertB=?; othercontrolsignal = ?; end
    `AND:  begin muxindex = 3; invertB=?; othercontrolsignal = ?; end
    `NAND: begin muxindex = 3; invertB=?; othercontrolsignal = ?; end
    `NOR:  begin muxindex = ?; invertB=?; othercontrolsignal = ?; end
    `OR:   begin muxindex = ?; invertB=?; othercontrolsignal = ?; end
    endcase
end
endmodule
```

I like this syntax because it explicitly states what each control signal does for each command.

To add additional control signals modify the module definition, add the corresponding “output reg” lines, and add the state of the lines in each case. Be careful to be sure to set each control signal in each case. If you accidentally omit one, it will generate latches that you don't want to implement the behavior that you don't want.

Generate

The "generate" syntax in Verilog will reduce the repetitive aspect of instantiating large structures. The basic flow looks like this:

```
generate
genvar index;
    for (index = 0; index<32; index = index + 1) begin
        sometypeofgate gatename(output[index], input[index]);
    end
endgenerate
endmodule
```

If you choose to use this syntax, the carry chains may pose a bit of difficulty in getting the syntax correct, because the 0th unit and the N-1 th unit need to be handled slightly different than the units between. Hints:

1) Declare the carry chain as 1 bit wider than the other signals.
2) Buffers are "buf"
3) "index+1" is a valid way to address the next carry bit. We'll talk about why this doesn't silently generate adder structures in the background (Preprocessing)