Com Sci 152B Digital Design
Lab 1: ALU & Register File Implementation

Michael Hale	004-620-459
Matthew Nuesca	904-440-067
Shilin Patel	904-569-866
Bingxin Zhu	704-845-969

Overview

In this lab we implemented two key components found in modern processors: the arithmetic logic unit (ALU) and the register file. The purpose of the ALU is to perform a variety of operations on arbitrary 16-bit integers and write the result to an output port. On the other hand, the register file performs no computation and simply reads and writes to a series of internal 16-bit registers.

Questions

1) Structural Verilog is an implementation of a system using only basic module building blocks, such as logic gates. No logic abstracted further than wires and gates is allowed. Behavioral Verilog can implement abstractions on top of logic gates such as conditionals and looping. Behavioral Verilog also may rely on always blocks to abstract clock edge based operations that in structural Verilog would require far more effort to manipulate. Using this type of syntax may limit knowledge of what is actually occurring on the hardware (as is natural with every abstraction).

To implement a multiplexer in both a structural and behavioral method requires:

```
input select_bit;
input input_a;
input input_b;

output output;

Structural multiplexer (only uses logic gates):

assign output = (input_a & !select_bit) | (input_b &
   select_bit);
```

```
Behavioral multiplexer (uses abstracted logic):

always @ (select_bit or input_a or input_b)
begin
if (select_bit == 1'b0) begin
output <= input_a;
end else begin
output <= input_b;
end
end
```

2) An asynchronous multiplexer is the one implemented in the previous question. This approach is always checking for either input or select bits to determine which option to choose. The actual assignment of one of the inputs to the output is non blocking, meaning it will happen independent of the order of other operations occurring in the system. A synchronous mux implements the same logic but uses blocking assignments and doesn't necessitate an always block. It will run in a specific order with other modules and will complete at a known time.

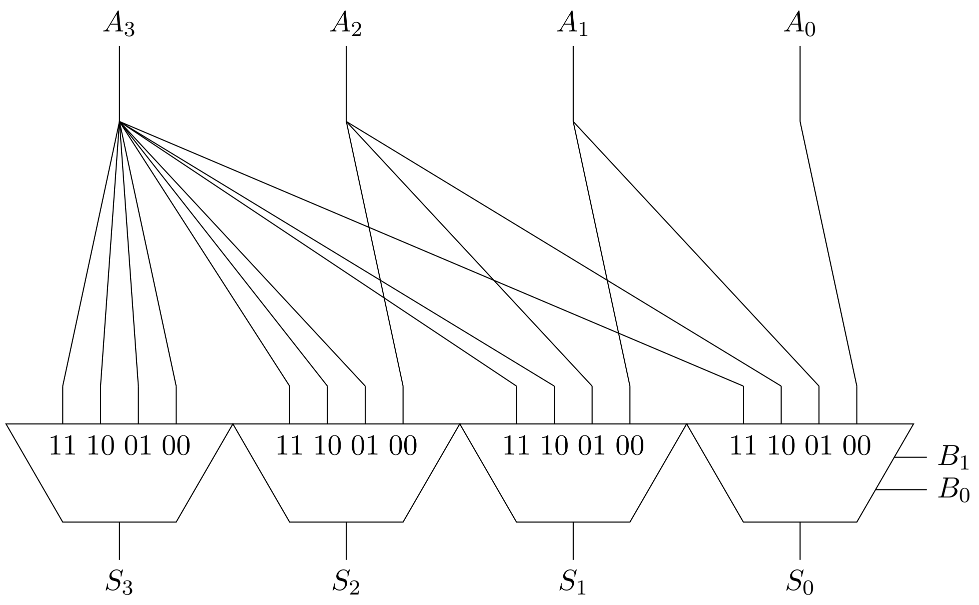
Behavioral synchronous multiplexer:

```
assign output = select_bit : input_a ? input_b;
```

3) The difference between the implementations of the shifters are determined by the characteristics of each shift. Arithmetic and logical left shifts are almost identical. When the bits are shifted left (towards the MSB), the now open position is filled with a 0. The difference is that the logical shift can not overflow, while the arithmetic shift will overflow if the new MSB is different from the previous MSB. Arithmetic right shifts shift towards the LSB, with the open position becoming a copy of the sign bit (if two's complement representation), or the MSB (in a general case). Logical right shift will be similar to the arithmetic right shift, except that the open bit will always be replaced by a zero, not the previous MSB. Shifts, arithmetic («<, »>) and logical («, »), are implemented using behavioral logic for this project, however the arithmetic overflow is determined by using an XOR on the MSB of the input and the MSB of the output.

4) Using only structural components, let's implement an arithmetic right shifter. This is achievable using only multiplexers. Below is a diagram for a four bit shifter. The 2 to 1 multiplexer logic is abstracted to a black box using the implementation described above. There is no overflow for the right shift as the sign bit will never change.

Arithmetic Right Shifter



Obstacles and Solutions

There were 2 main issues that we encountered. Our first issue was determining how to implement overflow. We first assumed that we can check for overflow just by XOR'ing the most significant bit in our output (16th bit) compared to the carry out bit in out output (17th bit). This insured that positive overflow would be caught when 2 positive numbers were added and negative overflow would be caught when 2 negative numbers were added. What we didn't realize was that the overflow bit would be set when we added a small positive number to a big negative number to result in a negative output. After looking through waveforms to figure out what was happening we found that to really track overflow, we had to check when our 2 inputs were positive numbers and our output was negative or when our 2 inputs were

negative numbers and our output resulted in being positive.
The second issue we ran into was figuring out the logic for our SLT. Our SLT uses subtraction to check SLT and so subtracting a negative number from a really big positive number can result in overflow. After thinking about it for a while, we realized that the SLT logic for this case is to check for a sign difference on the two inputs.