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

1 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.

2 Design Requirements

2.1 ALU Requirements

The ALU shall perform a set of logical and arithmetic operations given the sixteen bit inputs `a` and `b` and the four bit op-code `op`. The ALU must write the output of the encoded operation to a sixteen bit output `out` and set two flags (`ovf`, `zero`) if the operation results in overflow or a zero output respectively.

The operations to be encoded are:

OP	**Description**	**Symbolically**
0000	Subtraction	$a - b$
0001	Addition	$a + b$
0010	Bitwise OR	$a \mid b$
0011	Bitwise AND	$a \ \& \ b$
0100	Decrement	$a - 1$
0101	Increment	$a + 1$
0110	Inverse	$\sim a$
1000	Logical left shift	$a << b$
1001	Set on less than or equal	$a \leq b$
1010	Logical right shift	$a >> b$
1100	Arithmetic left shift	$a <<< b$
1110	Arithmetic right shift	$a >>> b$

2.2 Register File Requirements

The register file shall contain 32 sixteen bit registers and must include two sixteen bit read ports and a write port (`bus_a, bus_b, write_bus`) with their corresponding five bit register select ports (`ra, rb, rw`). It must also accept a write-enable signal (`wren`), a clock signal (`clk`) and a reset signal (`rst`) which sets the contents of each register to zero. Additionally, if a register is being read and written at the same time, the read should reflect the updated value immediately.

3 Implementation

3.1 ALU Implementation

Our ALU implementation essentially partitions each instruction into a single Verilog module. In the top level ALU module, the inputs `a` and `b` fan out into each of these instruction modules which each independently generate an `ovf` and `result` signal concatenated into a single `out` signal. These signals are then fed into a multiplexer which uses the op-code `op` to determine the ALU output. The `zero` flag is generated by OR-ing each bit of the output and inverting the result.

3.1.1 Output Multiplexer

Rather than designing a single multiplexer that would take 12 seventeen bit inputs and generate one seventeen bit output, we instead created a simple two-input multiplexer module. In our top level ALU module, we then instantiated arrays of these muxes of length 17 to handle the wide instruction outputs. To generalize the multiplexer to 12 inputs, we created a cascade of 2-input muxes where each stage was determined by a different bit of the op-code `op`. Figure 1 shows how each bit of the op-code selects a level of the hierarchy of muxes.

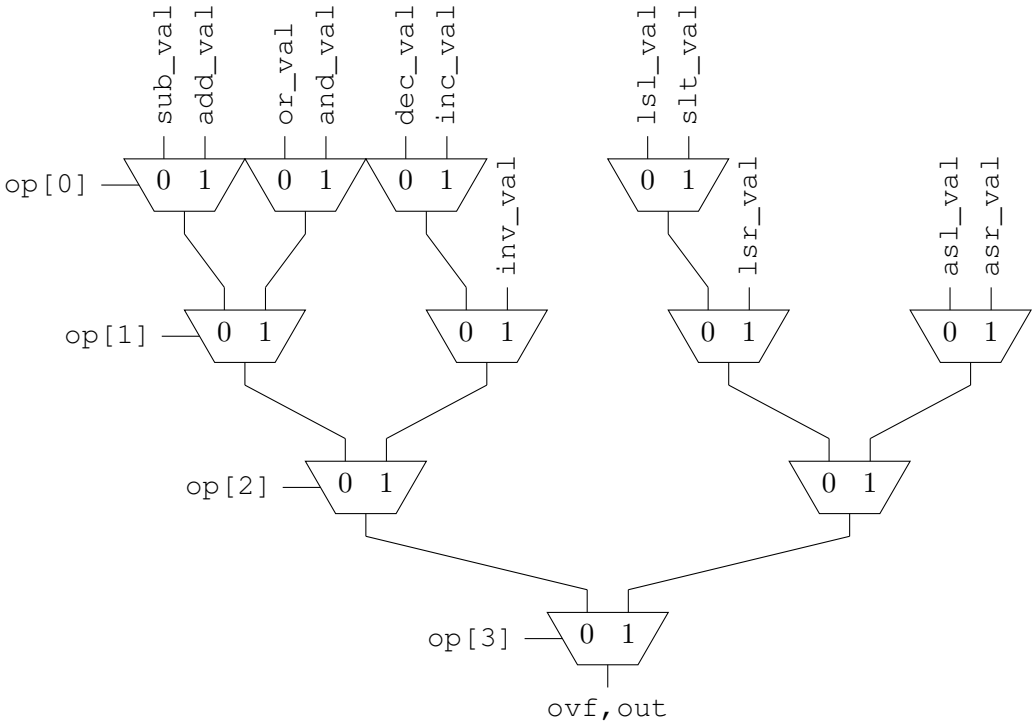


Figure 1: Tree of 2-input muxes to mimic the operation of a 16-input mux.

The implementation of the bitwise and the shift instructions was relatively straightforward, since we were permitted to use the Verilog built-in operations (`&`, `|`, `^`, `«`, `»`, `«<`, `»>`). The ADD, SUB, INC, DEC, and SLT instructions are implemented using the adder module. The adder performs 16-bit two's-complement and concatenates the `ovf` bit to the front of the `sum`.

4 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).

Implementation of a multiplexer using structural Verilog:

```
input select_bit, input_a, input_b;
output out;

//Structural multiplexer (only uses logic gates):
assign out = (input_a & !select_bit) |
             (input_b & select_bit);
```

Implementation of a multiplexer using behavioral Verilog:

```
input select_bit, input_a, input_b;
output reg out;

//Behavioral multiplexer (uses abstracted logic):
always @ (select_bit or input_a or input_b) begin
  if (select_bit == 1'b0) begin
    out <= input_a;
  end else begin
    out <= 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:

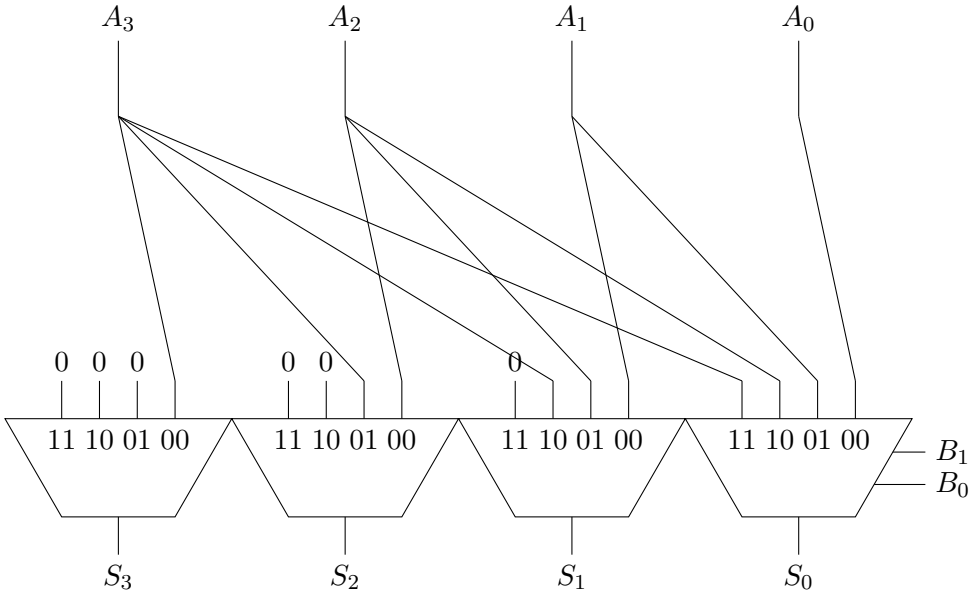
```
input select_bit, input_a, input_b;
output out;

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

3. The difference between the implementations of the shifters are deter-

mined 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.

Shifter



Arithmetic Right Shifter