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	$a >>> b$

Table 1: Specifications for each operation.

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 twelve 17-bit inputs and generate one 17-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.

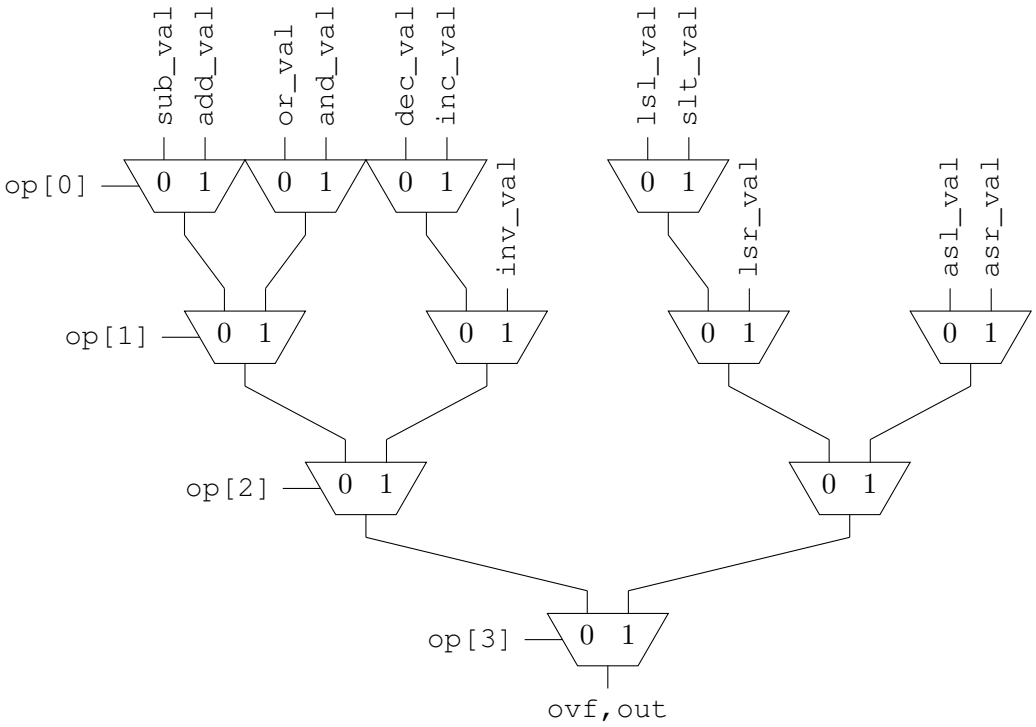

Figure 1: Tree of 2-input muxes to mimic the operation of a 12-input mux.

3.1.2 Addition Implementation

The ADD, SUB, INC, DEC, and SLT instructions are implemented using the adder module. The adder performs 16-bit two's-complement addition and concatenates the `ovf` bit to the front of the 16-bit `sum`. Our design is a simple implementation of a ripple-carry adder, with the 1-bit adder implemented in a separate module.

The 1-bit adder logic uses propagate (`p`) and generate (`g`) intermediate signals to determine the outputs `co` and `sum`.

```
input a, b, ci;
output sum, co;

wire p = a | b;
wire g = a & b;

assign co = g | (ci & p);
assign sum = ci ^ a ^ b;
```

This allows us to connect the carry signals with a single line of Verilog code in the 16-bit adder module.

```
input [15:0] a, b;
input ci;
output [16:0] out;

wire[15:0] sum;
wire[16:0] carry;
assign carry[0] = ci;

addbit adders[15:0](a,b,carry[15:0],sum,carry[16:1]);
```

Finally our overflow logic checks the two cases where the input signs are the same, and the sign of the sum differs. This bit is then concatenated to the front of the output.

```
assign overflow = (~sum[15] & a[15] & b[15])
  || (sum[15] & ~a[15] & ~b[15]);
assign out = {overflow,sum};
```

3.1.3 Other Instruction Modules

Each of the other instructions (logical shifts, arithmetic shifts, and bitwise operations) can be implemented using the permitted Verilog built-in operations. (<<, >>, <<<, >>>, &, |, ~).

3.2 Register File Implementation

The register file contains 32 16-bit registers, each initialized to zero. Upon receiving a positive clock edge, the reset bit is first checked and honored. If the reset signal is low, we perform two actions:

1. If the `wren` bit is high, we read from the `write_bus` and write to the register designated by `rw`.
2. We read from the registers designated by `ra` and `rb` unless a simultaneous write is taking place.

This behavior is implemented with the following Verilog code:

```
// Honor the reset signal
if (rst == 1) begin
  for (idx=0; idx < 32; idx=idx+1) begin
    registers[idx] <= 16'b0;
  end
```

```
end else begin
// Write to register rw
if (wren == 1) begin
  registers[rw] <= write_bus;
end
// Read from register ra
if (wren == 1 && ra == rw) begin
  bus_a <= write_bus;
end else begin
  bus_a <= registers[ra];
end
// Read from register rb
if (wren == 1 && rb == rw) begin
  bus_b <= write_bus;
end else begin
  bus_b <= registers[rb];
end
```

4 Testbench

4.1 ALU Test Cases

We tested the ALU by hard coding a series of test cases for each instruction and checking the output. If the output does not match the expected output, the failure will be printed to the console. Some instructions required us to dedicate more test cases, by the nature of their edge cases. Below is a comprehensive table of the test cases for each instruction:

Operation	**Input** A	**Input** B	**Output**	**Overflow**
Subtraction	91	47	44	0
	-91	-47	-44	0
	91	100	-9	0
	-50	-132	82	0
	30000	-3456	X	1
	-32000	10000	X	1
	-15	-15	0	0
Addition	15	19	34	0
	-20	-144	-164	0
	15	-19	-4	0
	-15	19	4	0
	31748	1020	X	1
	-30851	-1918	X	1
Bitwise OR	0x55AA	0xAA55	0xFFFF	0
	0x00FF	0xAA55	0xAAFF	0
	0x00FF	0x0FF0	0x0FFF	0
Bitwise AND	0x55AA	0xAA55	0x0000	0
	0x55AA	0x00FF	0x00AA	0
	0x55AA	0x55AA	0x55AA	0

Operation	Input A	Input B	Output	Overflow
Decrement	21930	X	21929	0
	0	X	-1	0
	-32768	X	X	1
Increment	21930	X	21931	0
	-1	X	0	0
	32767	X	X	1
Bitwise INV	0x0000	X	0xFFFF	0
	0x55AA	X	0xAA55	0
	0xFFFF	X	0x0000	0
Logical Shift Left	0x55AA	1	0xAB54	0
	0x55AA	16	0x0000	0
	0x55AA	0	0x55AA	0
Set on Less Than Or Equal	36	45	1	0
	36	15	0	0
	32767	-2	0	0
	-10	10	1	0
	10	-10	0	0
	-32768	1	1	0
	10	10	1	0
Logical Right Shift	0x55AA	1	0x2AD5	0
	0x55AA	16	0x0000	0
	0x55AA	0	0x55AA	0
Arithmetic Left Shift	0x55AA	1	0xAB54	1
	0xF000	16	0x0000	1
	0xAA55	0	0xAA55	0
Arithmetic Right Shift	0xAA55	1	0xD52A	0
	0xAA55	16	0xFFFF	0
	0xAA55	0	0xAA55	0

Table 2: Test cases.

4.2 ALU Waveforms

Included in this section are a subset of the waveforms obtained from our ALU testbench simulation.

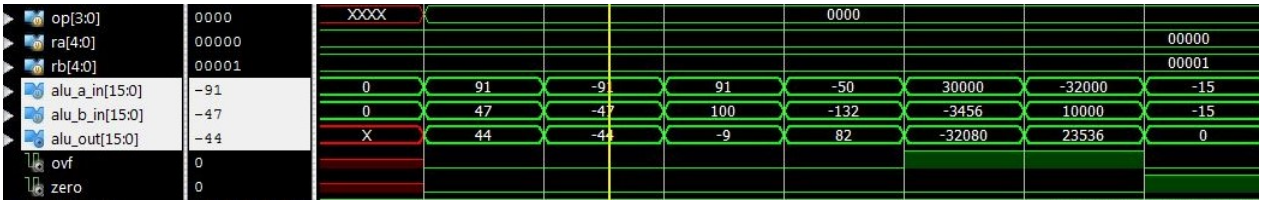

Figure 2: Subtraction.

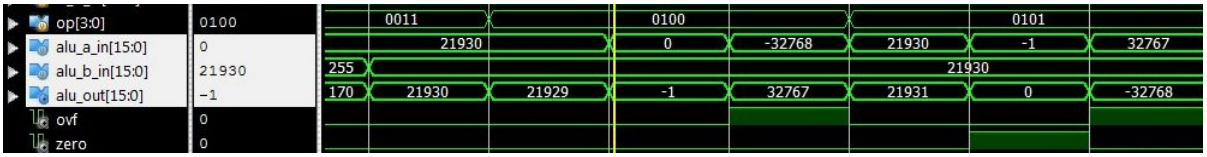

Figure 3: Increment and decrement.

Figure 4: Bitwise AND and OR operations.

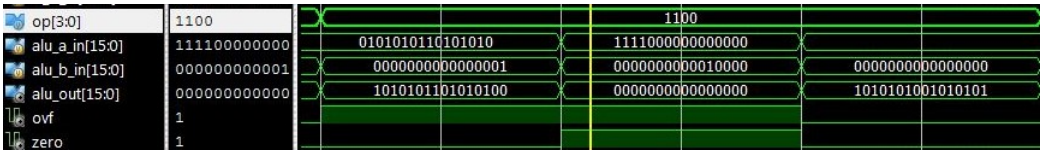

Figure 5: Arithmetic left shift.

4.3 Register File Testing

To test the register file, we first write to each register the value 0x55AA individually. We then loop through and read from each register (two at a time) to verify that the contents match the written value. In the other test, we write to R_0 at the same time that we are reading from R_0 and R_1, verifying that the read is updated instantly.

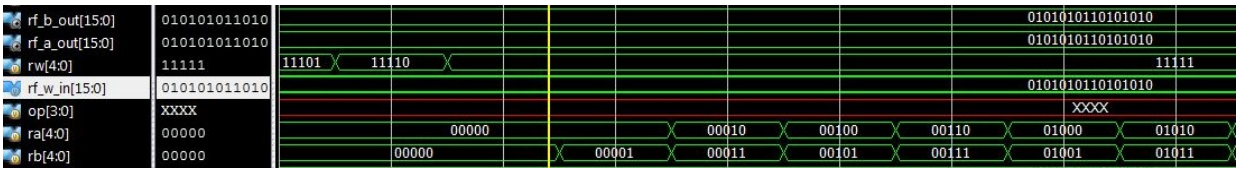

Figure 6: Reading each register from the register file. Even numbered registers are read from port A and odd numbered registers are read from port B.

5 Challenges & Solutions

We had 2 main issues when implementing the ALU. The first was trying to detect overflow. We initially started off by XOR'ing the 17th bit and the 16th bit of the output. While this would detect overflow, it also considerred other normal cases overflow as well. We realized this after checking the overflow bit for the case when there is a small positive number added with a large negative number resulting in a negative number which our ALU had incorrectly believed to be an overflow case. Our solution was to check if both inputs had the same sign and if the output had a different sign compared to both inputs The second issue we had was implementing a portion of the SLT operation. Since our SLT uses an adder, we realized that comparing a large positive number to a

large negative number can result in overflow. After hearing that our SLT had to implement logic to handle that case, we first did not know how to approach it. After thinking about it for a while, we realized that all we had to for this case was check the signs of the inputs. If the first input was positive and the second was negative, then SLT would automatically result in a 1. If the first input was negative and the second was positive, then SLT would automatically result in a 0. If both inputs had the same sign, then we use the implementation we already had in our SLT which is using the adder module to subtract the second number and checking to see whether or not the result was positive or negative.

6 Questions

1. **What is the difference between structural and behavioral Verilog? Please provide an example of a structural and behavioral implementation of a multiplexer.**

   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. **What is the difference between an asynchronous and synchronous Multiplexer? Please provide a brief explanation on how you could implement both using behavioral Verilog.**

   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. **What is the difference between an arithmetic and logical shifter?**

   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. **Assuming that you did NOT use Behavioral Verilog to implement an arithmetic shifter, how could you design one from scratch? Please include a simple diagram.**

   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.

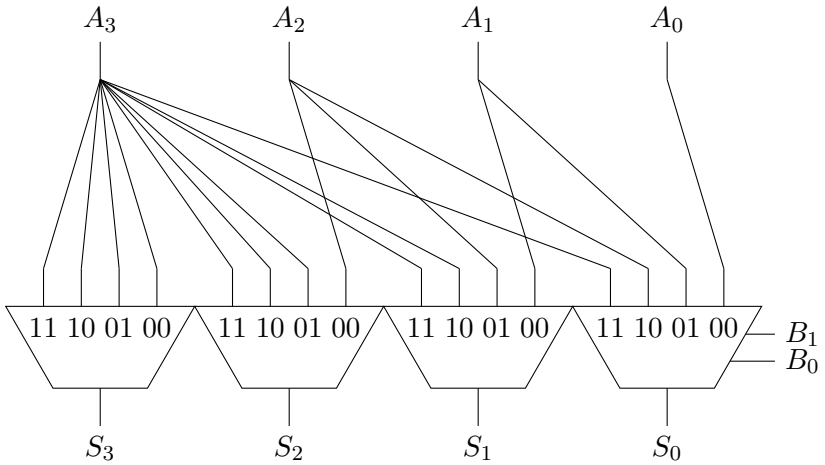

Figure 7: Arithmetic right shifter.