Lab Report 1

Yuhuang Chen (804449266), Zeyuan Xu (004255573)

1 Part 1: 1 Bit ALU

1.1 Introduction

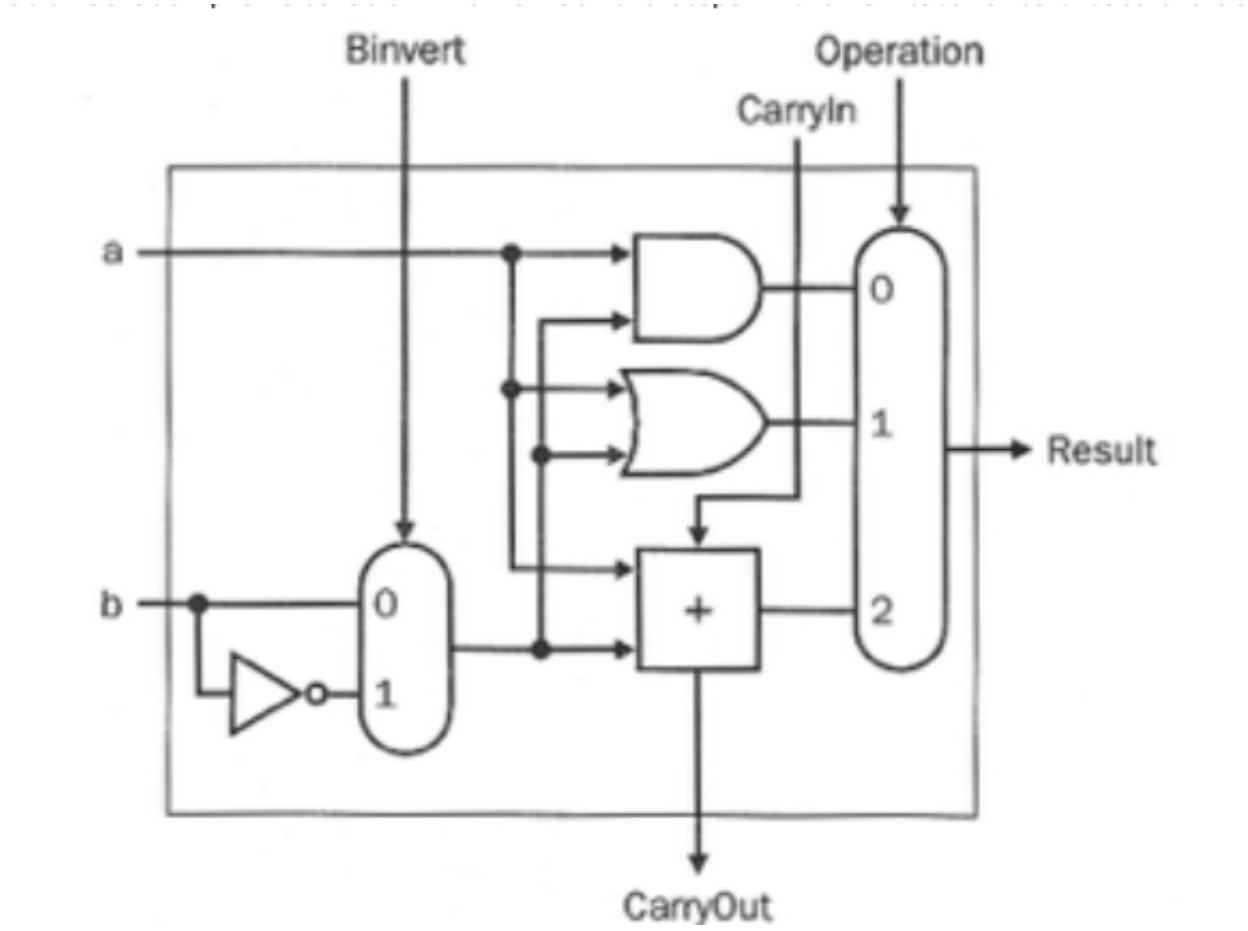


Figure 1: 1-Bit-ALU

In the first part, we simply need to follow the diagram (figure 1) to implement an 1-bit ALU, which has components as: OR gate, AND gate, NOT gate, 1-bit full adder, 2-1 mux, and 3-1 mux. The 2-1 mux is easily implemented via the logical gate specified in figure 2. The 4-1 mux is simply hooked up in the logic specified in figure 3. The schematic diagram is:

1.2 Simulation Result

The simulation result for the 1-bit ALU is shown in figure 4, in which all operations are tested and verified.

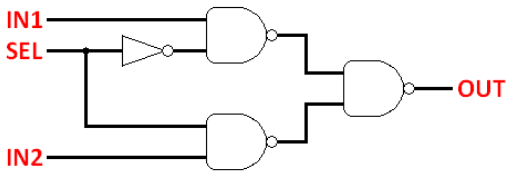


Figure 2: 2-1 mux

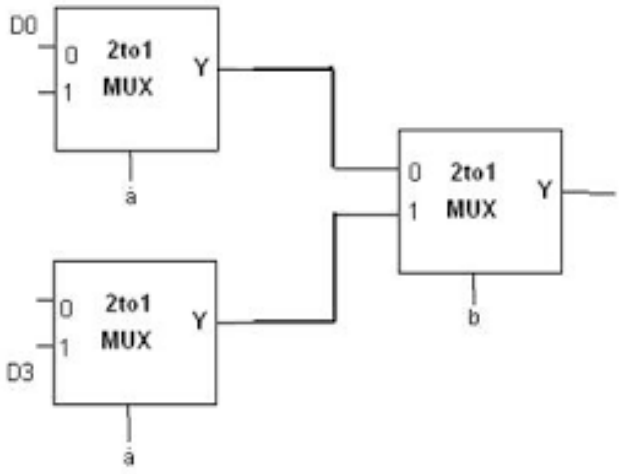


Figure 3: 4-1 mux

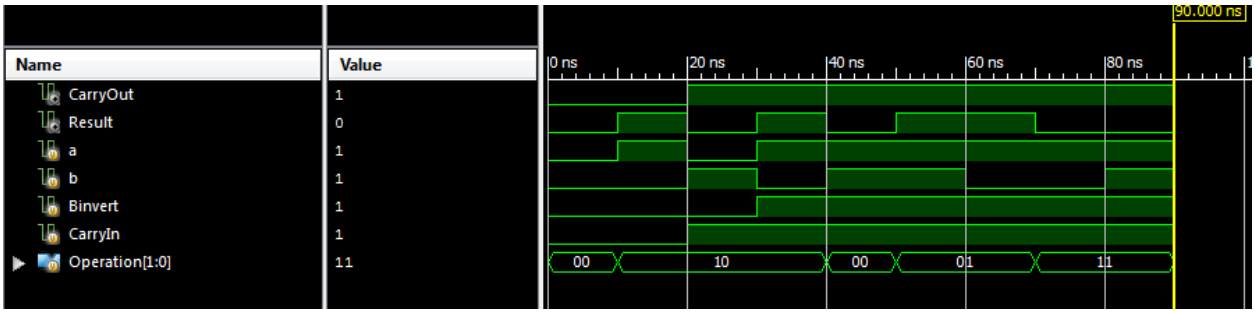


Figure 4: 1-bit ALU demo

1.3 Discussion

Nothing particular is notieable about the 1-bit ALU. It sets the logic for further implementation of 16 bit ALU, whose structural builds upon it.

2 Part 2: 16 Bit ALU

2.1 Introduction

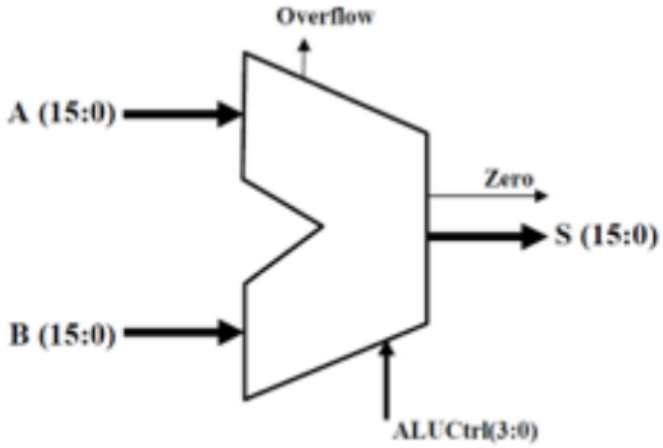
The 16-bit ALU consists consists of operations specified in figure 5. The most complicated part of the ALU design is to design the 16 bit full adder, which should also contain overflow detection. The 16-1 mux is built upon by 4-1 mux, and the logic is depicted in figure 6. The 16-1 mux is shown in figure 7. The adder is implemented as a ripple adder. When buiding the 16-bit adder, we first build 4-bit adder from 1-bit adders, then using the same method to construct the 16-bit adder from the 4-bit adders. The logic can be depicted in figure 8.

2.2 Simulation Result

There are 12 operations, so we have in total 12 simulation results, each for one operation.

1. Subtraction

Subtraction's simulation result is shown in the figure 9: the first test case and the fourth test cases do not have overflow, whereas the second and the third test cases, subtracting the largest positive number from the



ALU Ctrl	Description
0000	Subtraction
0001	Addition
0010	Bitwise OR
0011	Bitwise AND
0100	Decrement
0101	Increment
0110	Invert
1100	Arithmetic Shift Left
1110	Arithmetic Shift Right
1000	Logical Shift Left
1010	Logical Shift Right
1001	Set on Less than or Equal

Figure 5: 16-bit ALU

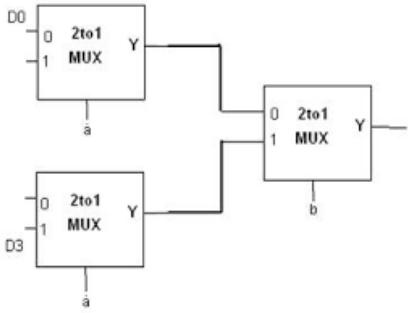


Figure 6: 4-1 mux

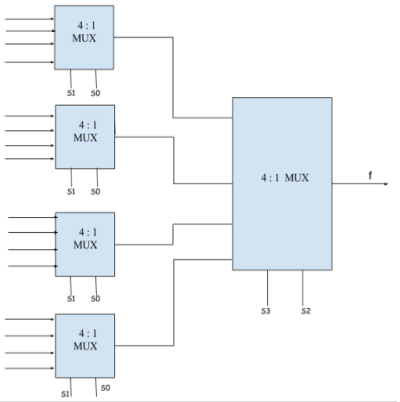


Figure 7: 16-1 mux

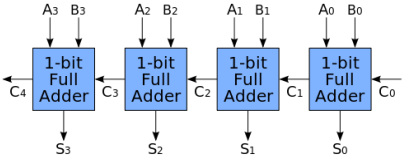


Figure 8: 4 bit adder

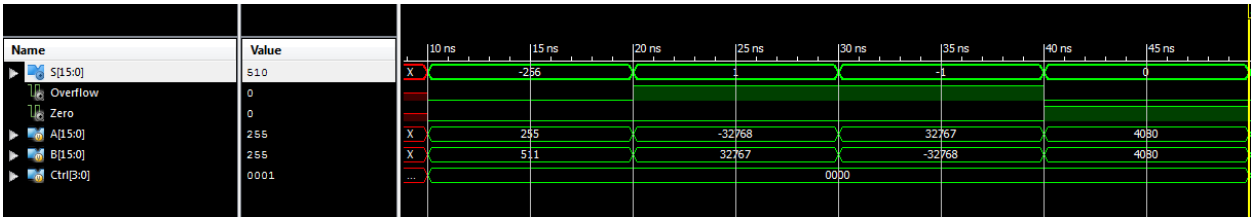


Figure 9: subtraction case

smallest negative number and subtracting the smallest negative number from the largest positive number, all trigger overflow, and the overflow bit is set to high in our simulation.

2. Addition

Addition's simulation result is shown in the figure 10: the first test case and the fourth test case do not have overflow, whereas the second and the third test cases, adding the largest positive number with itself and adding the smallest negative number with itself, all trigger overflow, and the overflow bit is set to high in our simulation.

3. Bitwise OR

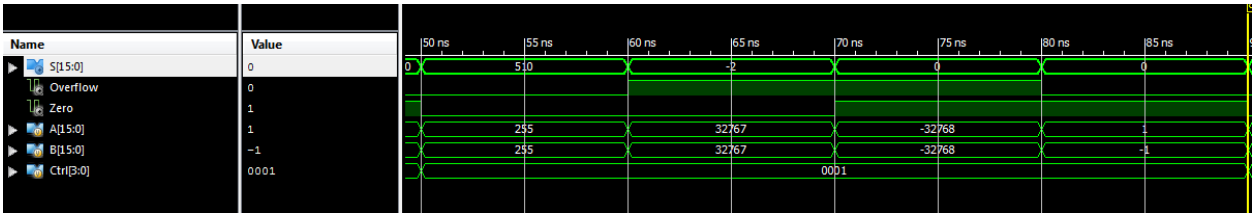


Figure 10: addition case

The bitwise OR's result is in figure 11. In this case, no overflow occurs.

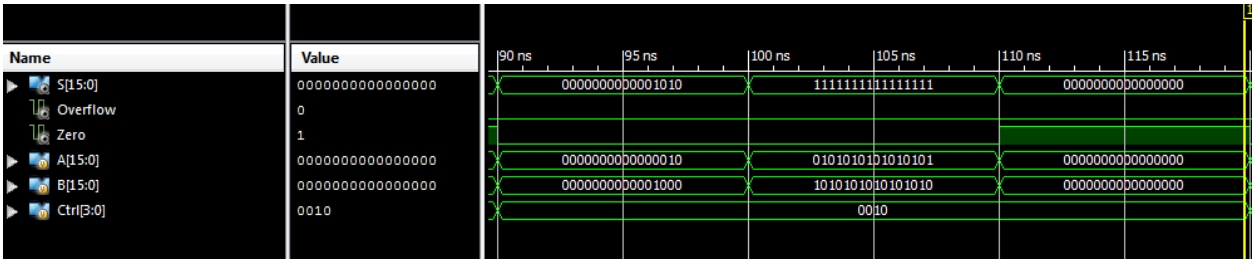


Figure 11: bitwise OR case

4. Bitwise AND

The bitwise AND's result is in figure 12. In this case, no overflow occurs.

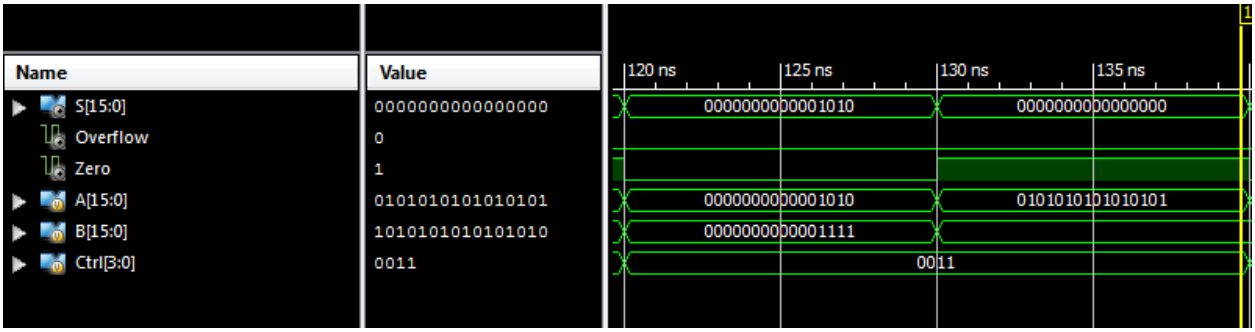


Figure 12: bitwise AND case

5. Decrement

The Decrement result is in figure 13. The first and the last cases do not overflow, but the second case,

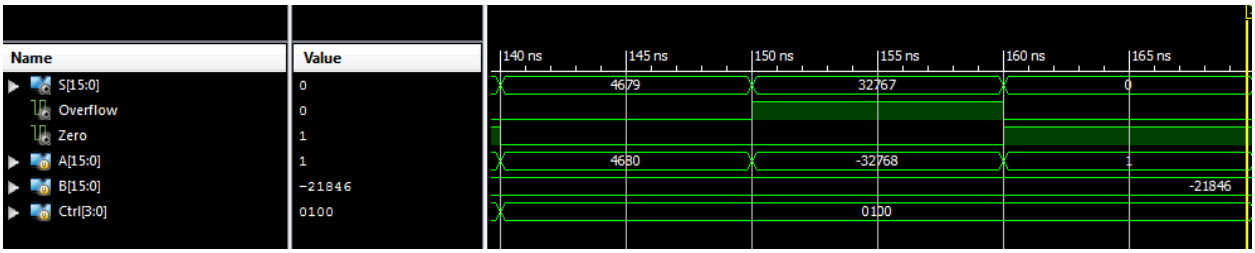


Figure 13: Decrement case

which increments the largest positive number, triggers an overflow and was captured in the simulation.

6. Increment

The increment case result is in figure 14. The first and the last cases do not overflow, but the second case,

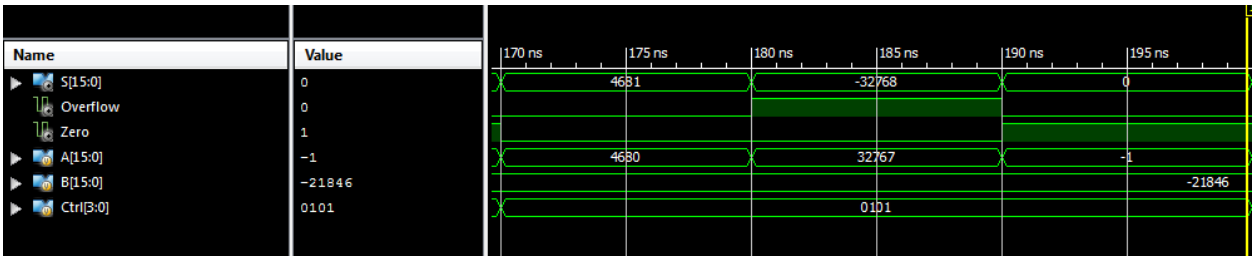


Figure 14: Increment case

which decrements the smallest negative number, triggers an overflow and was captured in the simulation.

7. Invert

The invert case result is in figure 15. The first and the second case do not overflow, but the last case, which

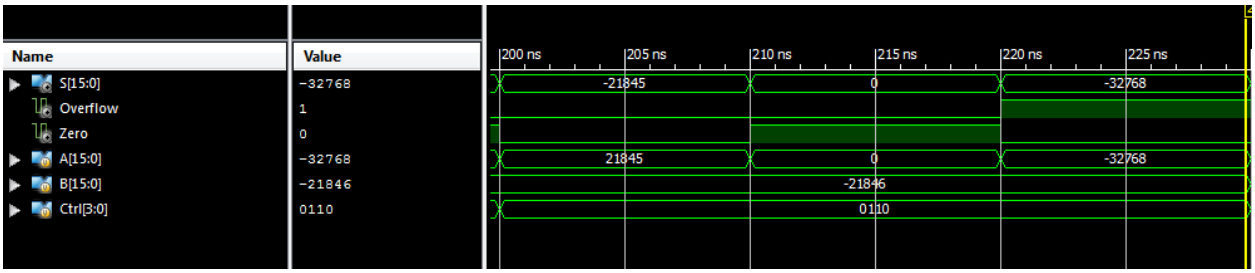


Figure 15: Invert case

inverts the number 16'b1000000000000000, results in an overflow, since the inversion gives the same result back.

8. Arithmetic Shift Left

The result for arithmetic shift left is in figure 16. The first case triggers an overflow. This is because we

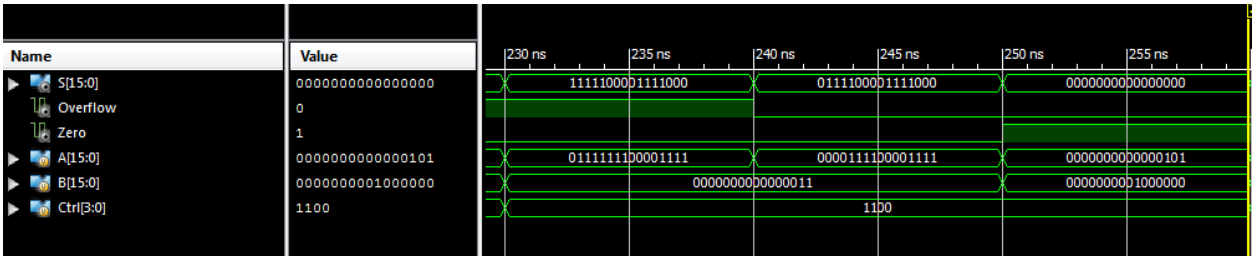


Figure 16: Arithmetic Shift Left

have an positive number, and left shift should makes another positive number, but a negative number is obtained.

9. Arithmetic Shift Right

The result for arithmetic shift right is in figure 17. No overflow occurs for this case.

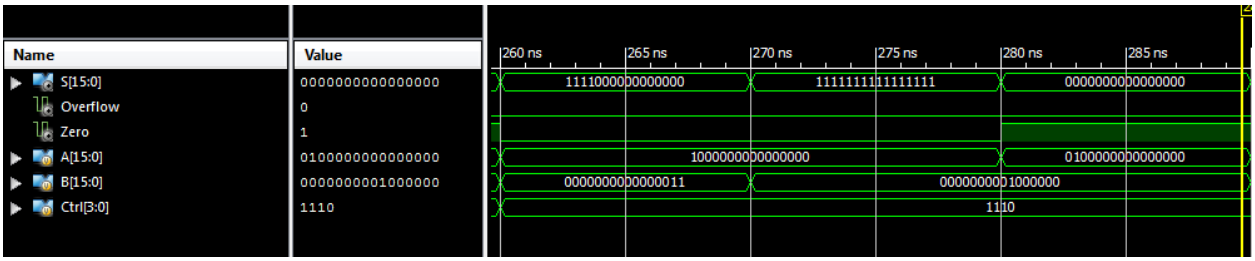


Figure 17: Arithmetic Shift Right

10. Logical Shift Left

The result for logical shift left is in figure 18. Since it is a logical shift, there is no overflow.

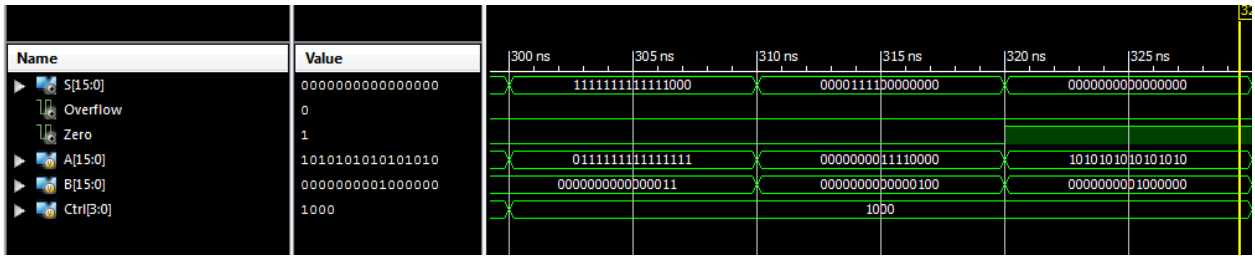


Figure 18: Logical Shift Left

11. Logical shift Right

The result for logical shift right is in figure 19. Since it is a logical shift there is no overflow.

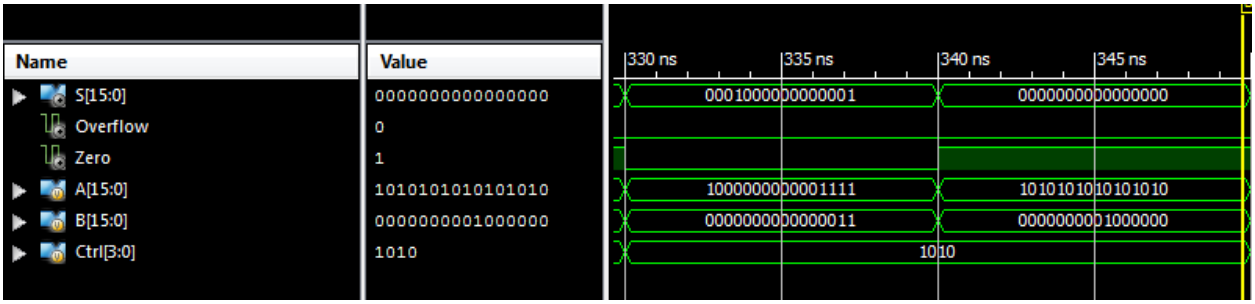


Figure 19: Logical Shift Right

12. Set on Less than or Equal

The result for the set less than or equal case is in figure 20. Even though the subtraction logic might trigger overflow, it should not and is not captured here.

2.3 Overflow Detection

Subtraction: A - B has overflow in two cases: A is positive, B is negative and the result is negative; and A is negative, B is positive and the result is positive. So overflow happens when A and B have different sign and result is the opposite sign as A. We implemented this logic by:

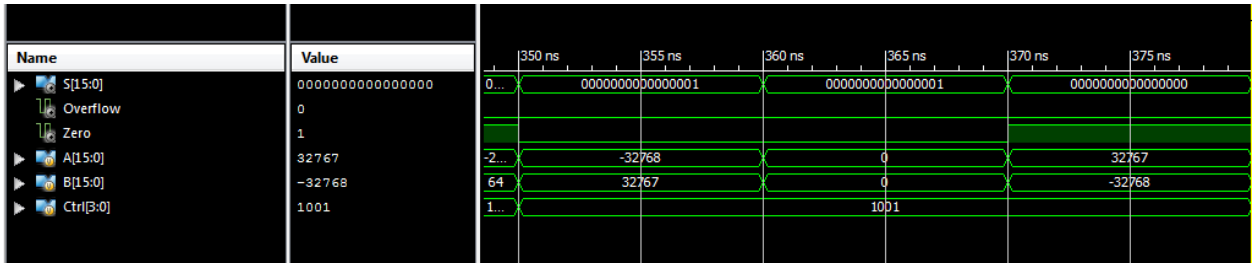


Figure 20: slt case

```
assign o0 = (A[15] ^ B[15]) & (A[15] ^ r0[15]);
```

Addition: A + B has overflow in two cases: A is positive, B is positive, and the result is negative; and A is negative, B is negative, and the result is positive. So overflow happens when A and b have the same sign and the result is the opposite sign. We implemented this logic by:

```
assign o1 = (A[15] ~^ B[15]) & (A[15] ^ r1[15]);
```

Left Shift: Both arithmetic and logic left shift can trigger overflow: when before and after the operation, the sign of the operand changes, we detect an overflow. We implemented this with the logic:

```
assign o12 = A[15] ^ r12[15];
assign o14 = A[15] ^ r14[15];
```

Set Less Than: Since the set less than only cares about one bit (either 0 or 1), the code is as follows:

```
assign r9[15:1] = 15'b000000000000000;
assign r9[0] = (~o9 & ((~r0[15]) | subtract_zero)) | (o9 & (~A[15])); //consider overflow
```

3 Part 3: Register File

3.1 Introduction

The register file is depicted in figure 21:
The register file has two 5 inputs and 2 outputs: bus signals are 16 bits and Ra, Rb, Rw are 5 bits. It also has reset and clock pins.
The register is implemented in behavior Verilog.

3.2 Simulation Result

The simulation result is shown in figure 22. The result behaves correctly.

3.3 Discussion

Not many problems are encountered in the register implementation.

4 Part 4: Questions and Answers

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

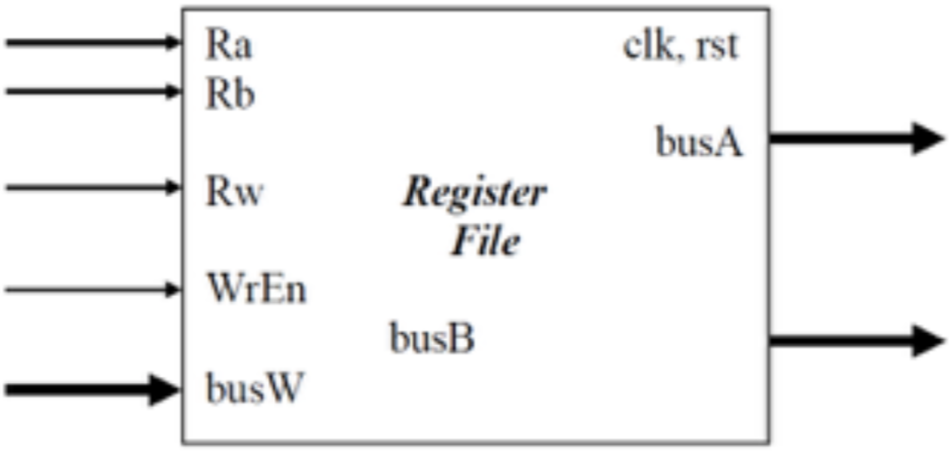


Figure 21: register

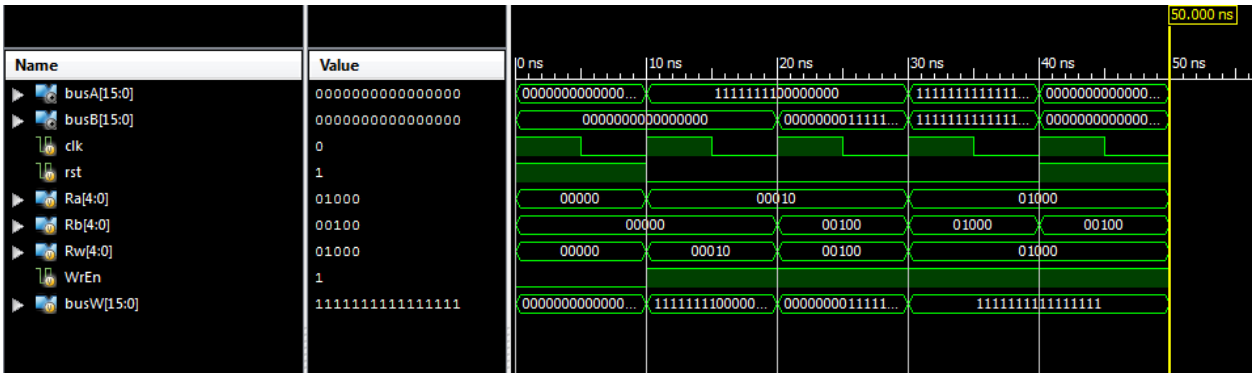


Figure 22: register result

A: behavior verilog is more high level and could use always block and assign operations; structural verilog focuses more on the individual building blocks of a module, such as building a complex module using gates.
Multiplexer structural implementation:

```
module mux7( select, d, q );

input[1:0] select;
input[3:0] d;
output     q;

wire       q, q1, q2, q3, q4, NOTselect0, NOTselect1;
wire[1:0]  select;
wire[3:0]  d;

not n1( NOTselect0, select[0] );
not n2( NOTselect1, select[1] );

and a1( q1, NOTselect0, NOTselect1, d[0]  );
and a2( q2,  select[0], NOTselect1, d[1]  );
and a3( q3, NOTselect0,  select[1], d[2]  );
```

```
and a4( q4,  select[0],  select[1], d[3]  );

or o1( q, q1, q2, q3, q4 );

endmodule
```

Multiplexer behavior implementation:

```
module mux1( select, d, q );

input[1:0] select;
input[3:0] d;
output     q;

wire      q;
wire[1:0] select;
wire[3:0] d;

assign q = d[select];

endmodule
```

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.

3. What is the difference between an arithmetic and logical shifter?
A: arithmetic shifter and logical shifter behaves the same when doing left shift, but arithmetic shift right replicates the sign bit, while logical shift only keeps padding zeros for the shifted bits.

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.
A: it can be implemented using gates, as shown in figure 23: this shifter shifts a 4-bit value, and similar logic can be applied to an n-bit shifter as well.

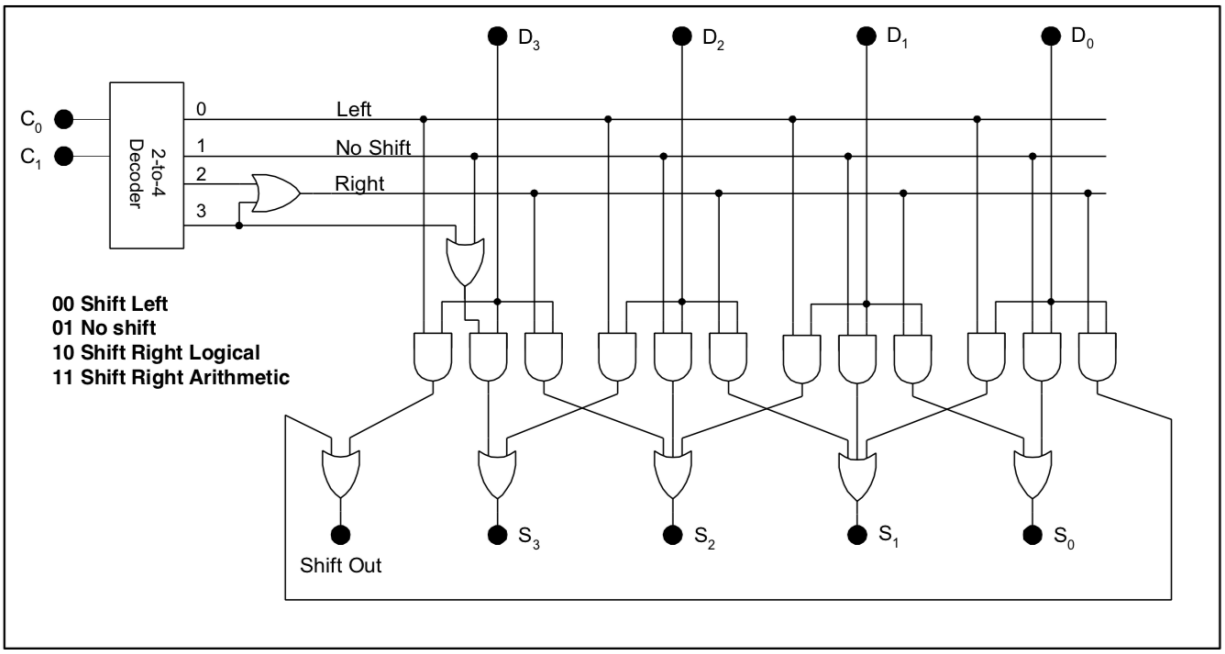


Figure 23: shifter