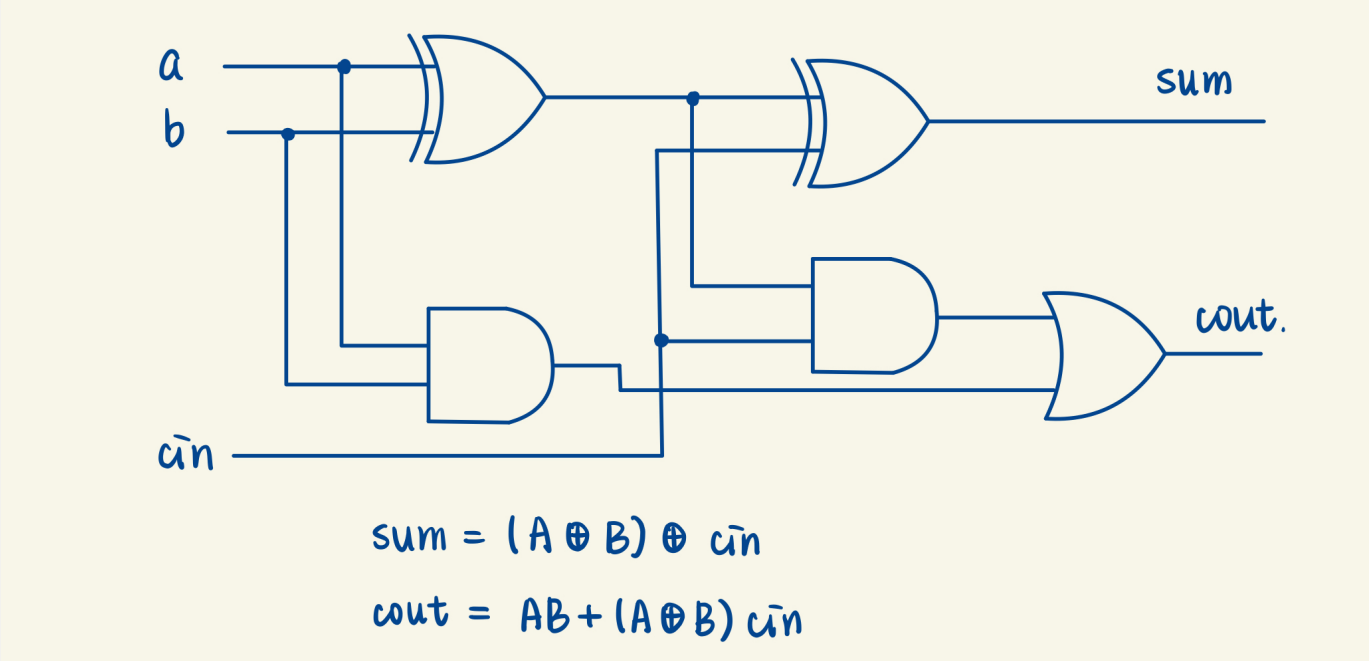
Computer Organization Lab 02 Report

109550121 温柏萱

1 - Bit ALU

Full Adder's logic diagram, Boolen Algebra expression and Verilog code.

```
module Full_adder (
  input A, input B, input cin, output reg sum, output cout
);

  assign sum = (A ^ B) ^ cin;
  assign cout = (A & B) | (cin & (A ^ B));
endmodule
```

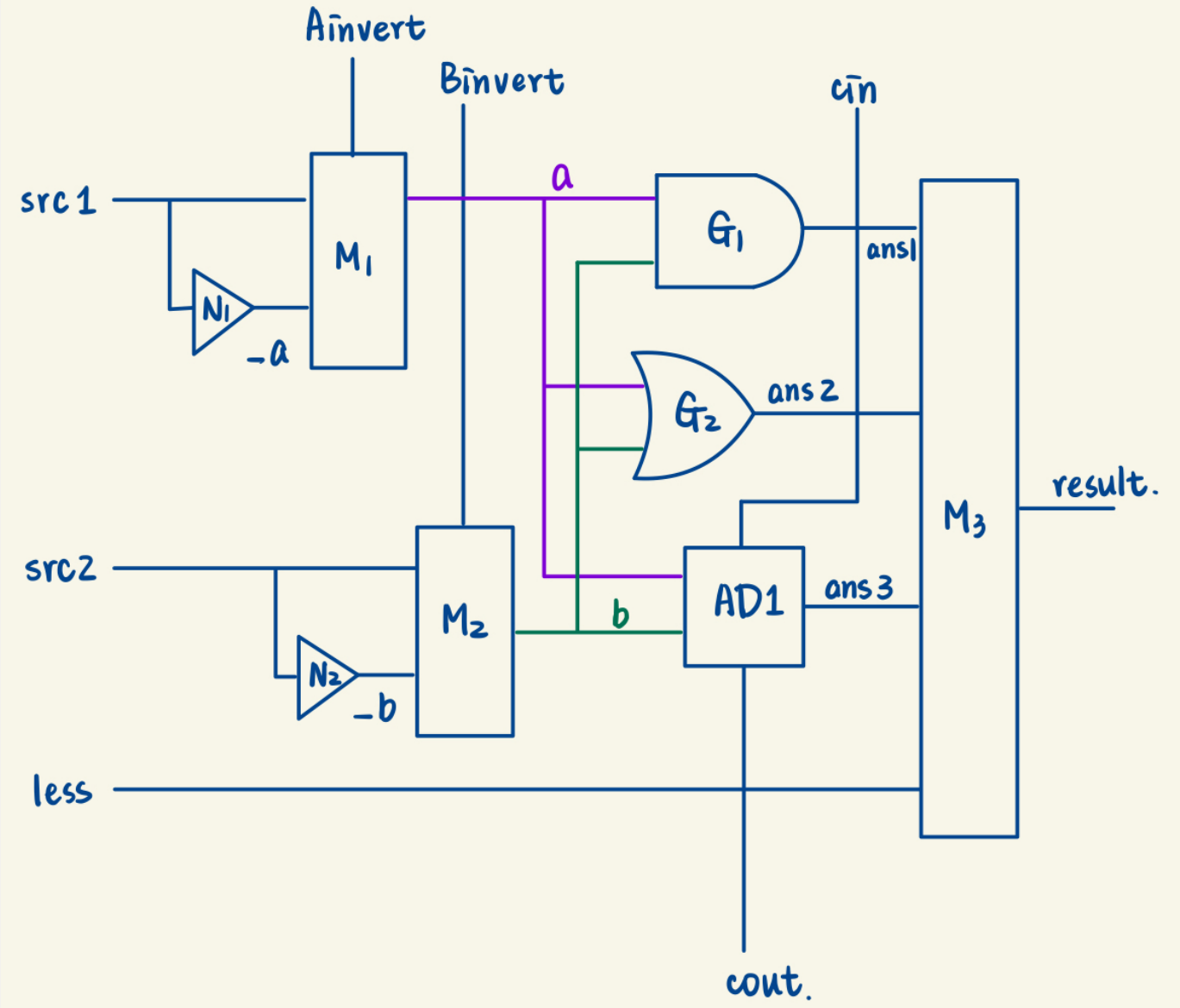


1 - bit ALU

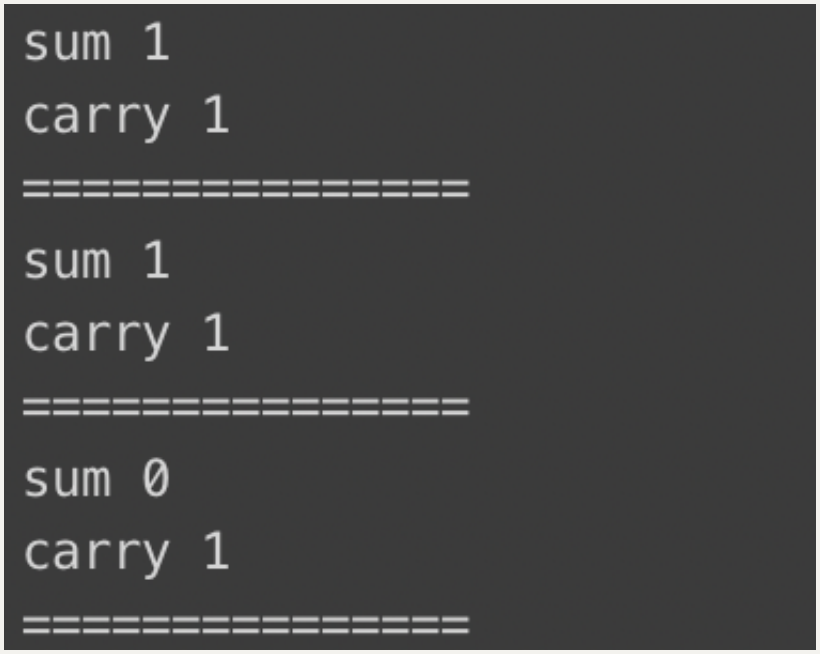
```
module alu_1bit(
```

```
    input       src1,       //1 bit source 1  (input)
    input       src2,       //1 bit source 2  (input)
    input       less,       //1 bit less      (input)
    input         Ainvert,    //1 bit A_invert  (input)
    input       Binvert,    //1 bit B_invert  (input)
    input         cin,        //1 bit carry in  (input)
    input       [2-1:0] operation,  //2 bit operation (input)
    output              result,     //1 bit result    (output)
    output              cout        //1 bit carry out (output)
    );
    wire  a, b, ans1, ans2, ans3;
    MUX2to1 M1(src1, ~src1, Ainvert, a);
    MUX2to1 M2(src2, ~src2, Binvert, b);

    and G1(ans1, a, b);
    or G2(ans2, a, b);
    Full_adder FA(a, b, cin, ans3, cout);
    MUX4to1 M3(ans1, ans2, ans3, less, operation, result);

endmodule
```

Since it is quite difficult for me to directly derive the Boolean Algebra expression, so I used gate-level scheme to write the 1-bit ALU. Thankfully there aren't too many gates or else just to name the gates would be an utter disaster. The following picture is a copy from the given slides and I added the name of my code on it.


Ainvert
Binvert
cin
src 1
a
G1
ans1
M1
N1
-a
G2
ans 2
result.
M3
src2
M2
AD1
ans 3
b
N2
-b
less
cout.

Result



32 - bit ALU

```
`timescale 1ns/1ps

module alu(
  input                  rst_n,         // negative reset
(input)
  input      [32-1:0] src1,          // 32 bits source 1         (input)
  input      [32-1:0] src2,          // 32 bits source 2         (input)
  input        [ 4-1:0]   ALU_control,   // 4 bits ALU control input
(input)
  output reg    [32-1:0]  result,        // 32 bits result
(output)
  output reg             zero,          // 1 bit when the output is 0, zero
must be set (output)
  output reg             cout,          // 1 bit carry out
(output)
  output reg             overflow       // 1 bit overflow
(output)
  );

  wire [32:0] carryOut;
```

```
  wire [31:0] buffer;
  wire set, dummy, tmp_zero, tmp_cout;
  genvar i;

  alu_1bit ALU[31:1](
      .src1(src1[31:1]),       //1 bit source 1 (input)
      .src2(src2[31:1]),       //1 bit source 2 (input)
      .less(1'b0),             //1 bit less     (input)
      .Ainvert(ALU_control[3]),     //1 bit Ainvert (input)
      .Binvert(ALU_control[2]),     //1 bit Binvert (input)
      .cin(carryOut[31:1]),       //1 bit cin (input)
      .operation(ALU_control[1:0]),  //operation      (input)
      .result(buffer[31:1]),   //1 bit result   (output)
      .cout(carryOut[32:2])       //1 bit cout(output)
  );

  Full_adder FA1(src1[31], ~src2[31], carryOut[31], set, tmp_cout);

  alu_1bit ALU1(src1[0], src2[0], set, ALU_control[3], ALU_control[2],
ALU_control[2], ALU_control[1:0], buffer[0], carryOut[1]);

  assign dummy = (ALU_control[1] && (!ALU_control[0])) ? carryOut[32] ^
carryOut[31] : 1'b0;

  assign tmp_zero =
buffer[0]|buffer[1]|buffer[2]|buffer[3]|buffer[4]|buffer[5]|buffer[6]|buffer
[7]|

buffer[8]|buffer[9]|buffer[10]|buffer[11]|buffer[12]|buffer[13]|buffer[14]|b
uffer[15]|

buffer[16]|buffer[17]|buffer[18]|buffer[19]|buffer[20]|buffer[21]|buffer[22]
|buffer[23]|

buffer[24]|buffer[25]|buffer[26]|buffer[27]|buffer[28]|buffer[29]|buffer[30]
|buffer[31];

  always @(*) begin
    overflow = dummy;
```

```
    zero = ~tmp_zero;
    if (ALU_control[1] && !ALU_control[0])
      cout = carryOut[32];
    else cout = 1'b0;
    result = buffer;
  end


endmodule
```

Implementation details

First I used the part select method to generate the 1 ~ 31 bit ALU, then generate set wire with a full adder performing subtraction. At last put the set as the less input in the ALU calculating the 0^{th} bit. Finally, copy to the results to the outputs.

Results

```
**************************************************
*              PATTERN RESULT TABLE              *
**************************************************
* PATTERN *              Result            * ZCV *
**************************************************
*      Congratulation! All data are correct!     *
**************************************************
Correct Count: 30
```

Encountered Problems

At first, I tried to use generate for to generate 31 $1-$bit ALUs, but for some mysterious reason, it didn't work for me. So I typed literally 32 $1-$bit ALUs. But there were 3 problems.

1. The overflow condition can only happen on addition and subtraction instructions. But I output the result of carryOut[31] $\oplus$ carryOut[32] directly. A conditional expression could solve it.

2. There can only be carry outs when performing addition and subtraction, thus putting a 0 to `cout` when we are not performing the above operations.
3. The full adder is used to generate the set value, thus we are using it to perform subtraction, thus the input of B should be inverted.

Before they were fixed, the result looked like this.

```
**************************************************
* No. 3 error!                                   *
* Correct result: 9bf5fea6     Correct ZCV: 000  *
* Your result: 9bf5fea6        Your ZCV: 01x     *
**************************************************
* No. 4 error!                                   *
* Correct result: 00000000     Correct ZCV: 110  *
* Your result: 00000000        Your ZCV: 11x     *
**************************************************
* No. 5 error!                                   *
* Correct result: fffffffe     Correct ZCV: 001  *
* Your result: fffffffe        Your ZCV: 00x     *
**************************************************
* No. 6 error!                                   *
* Correct result: 70459a76     Correct ZCV: 011  *
* Your result: 70459a76        Your ZCV: 01x     *
**************************************************
* No. 8 error!                                   *
* Correct result: 00000001     Correct ZCV: 000  *
* Your result: 00000000        Your ZCV: 100     *
**************************************************
* No. 9 error!                                   *
* Correct result: ffffffff     Correct ZCV: 000  *
* Your result: ffffffff        Your ZCV: 010     *
**************************************************
* No.12 error!                                   *
```

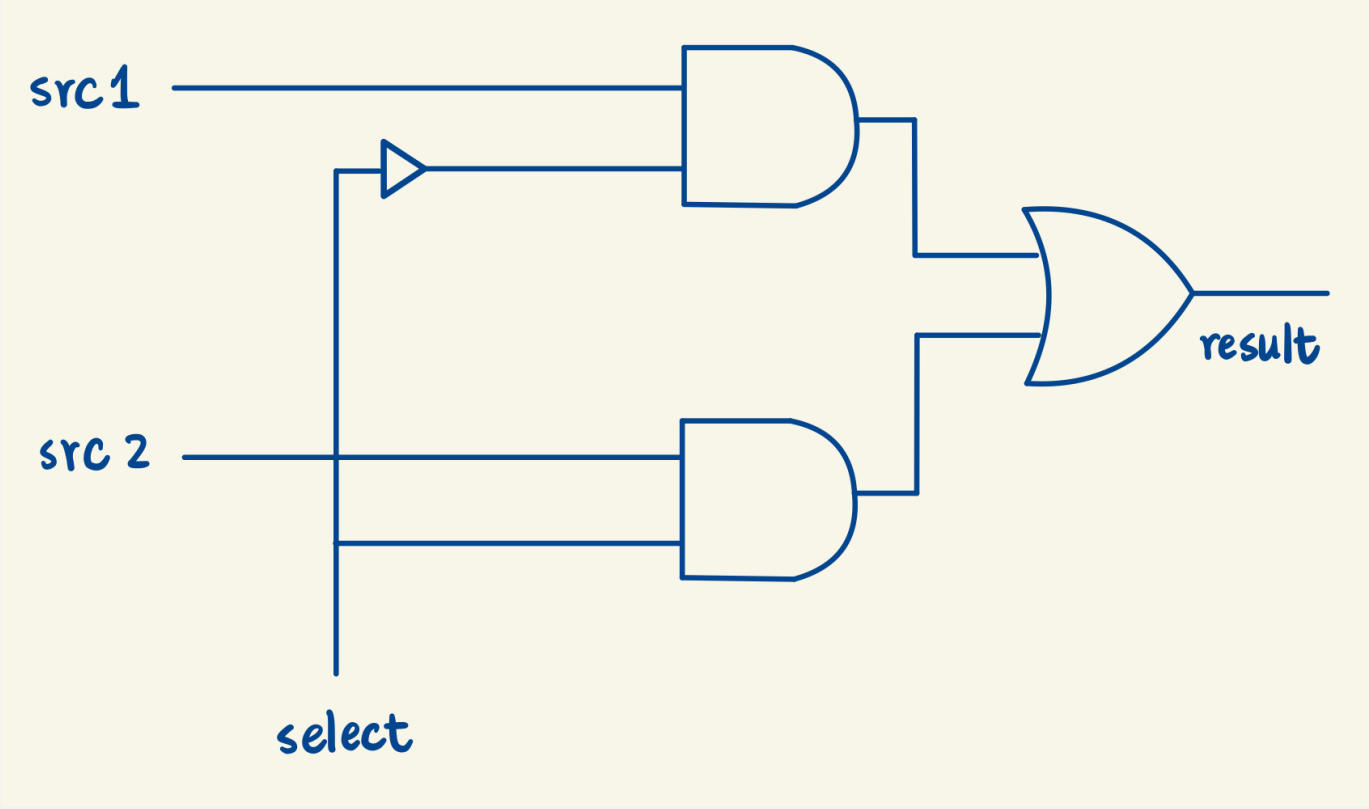
```
* Correct result: 00000000     Correct ZCV: 110   *
* Your result: 00000000        Your ZCV: 11x      *
**************************************************
* No.17 error!                                    *
* Correct result: 00000000     Correct ZCV: 100   *
* Your result: 00000001        Your ZCV: 010      *
**************************************************
* No.18 error!                                    *
* Correct result: 00000001     Correct ZCV: 000   *
* Your result: 00000000        Your ZCV: 100      *
**************************************************
```

and this.

```
*************************************************
*              PATTERN RESULT TABLE             *
*************************************************
* PATTERN *              Result              * ZCV *
*************************************************
* No. 8 error!                                   *
* Correct result: 00000001     Correct ZCV: 000  *
* Your result: 00000000        Your ZCV: 100     *
*************************************************
* No.17 error!                                   *
* Correct result: 00000000     Correct ZCV: 100  *
* Your result: 00000001        Your ZCV: 000     *
*************************************************
* No.18 error!                                   *
* Correct result: 00000001     Correct ZCV: 000  *
* Your result: 00000000        Your ZCV: 100     *
*************************************************
* No.19 error!                                   *
* Correct result: 00000000     Correct ZCV: 100  *
* Your result: 00000001        Your ZCV: 000     *
*************************************************
* No.20 error!                                   *
* Correct result: 00000000     Correct ZCV: 100  *
* Your result: 00000001        Your ZCV: 000     *
*************************************************
* No.21 error!                                   *
* Correct result: 00000001     Correct ZCV: 000  *
* Your result: 00000000        Your ZCV: 100     *
*************************************************
```

2 - 1 MUX

```
module MUX2to1(
  input      src1,
  input      src2,
  input      select,
  output     result
  );

assign result = (src1 && (!select)) || (src2 && select);

endmodule
```

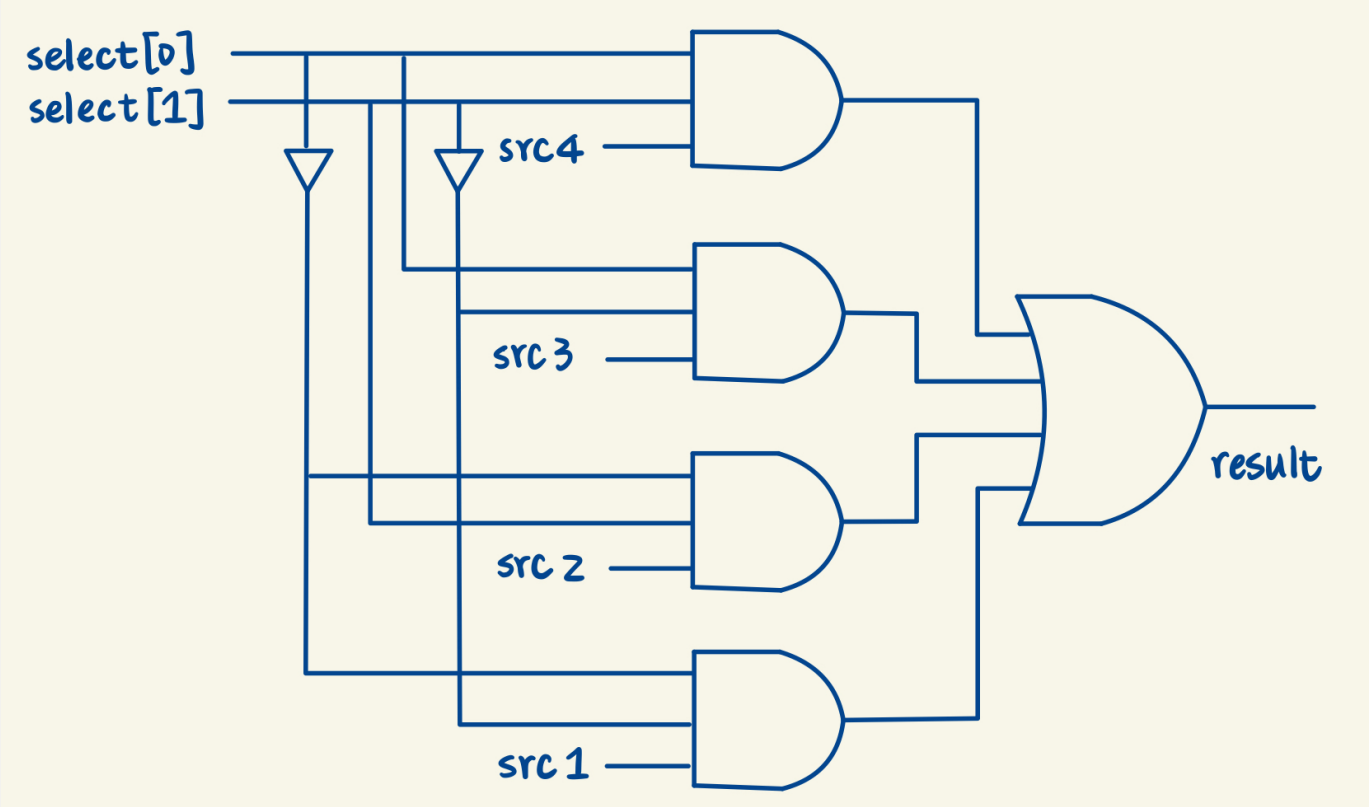


The result $= src1\bar{S} + src2S$

For simplicity, I used dataflow method to implement it.

4 - 1 MUX

```
module MUX4to1(
  input     src1,
  input     src2,
  input     src3,
  input     src4,
  input   [2-1:0] select, // input [1:0] select
  output      result
  );
assign result = ((!select[0]) && select[1] && src1) || (select[0] &&
(!select[1]) && src2) || ((!select[0]) && select[1] && src3) || (select[0]
&& select[1] && src4);

endmodule
```



The result can be written as $\bar{S}_0\bar{S}_1src1 + S_0\bar{S}_1src2 + \bar{S}_0S_1src3 + S_0S_1src4$

Thus I used dataflow method to implement the $4-1$ multiplexer.