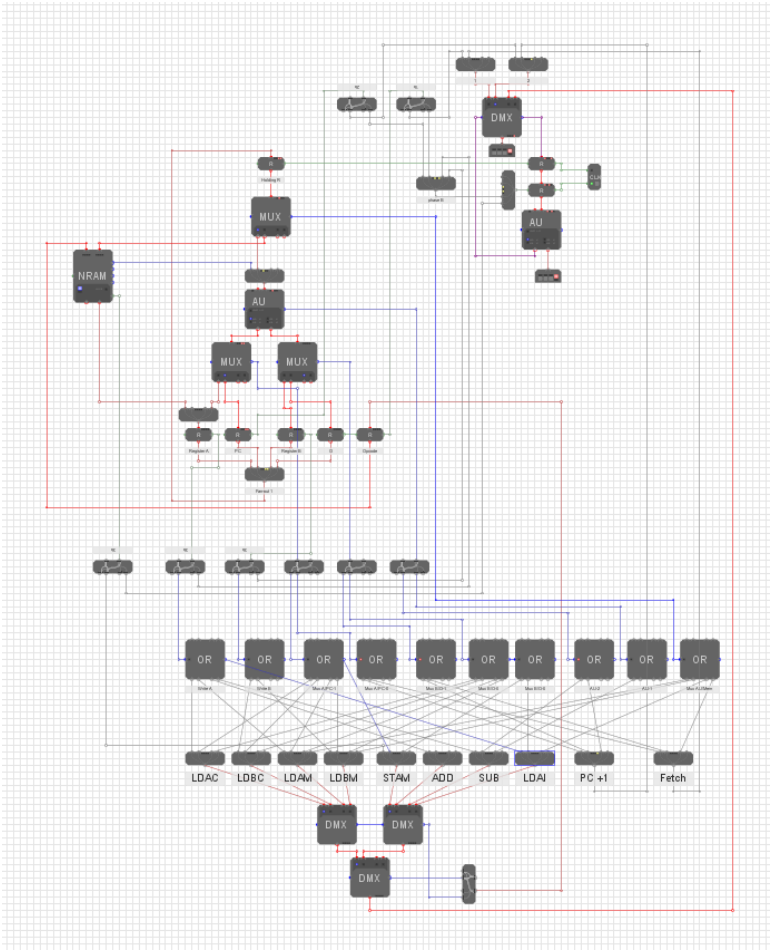
4-bit CPU design by ModuleSim, Logisim and Verilog

by
Jhih-Jia, Yin

Introduction

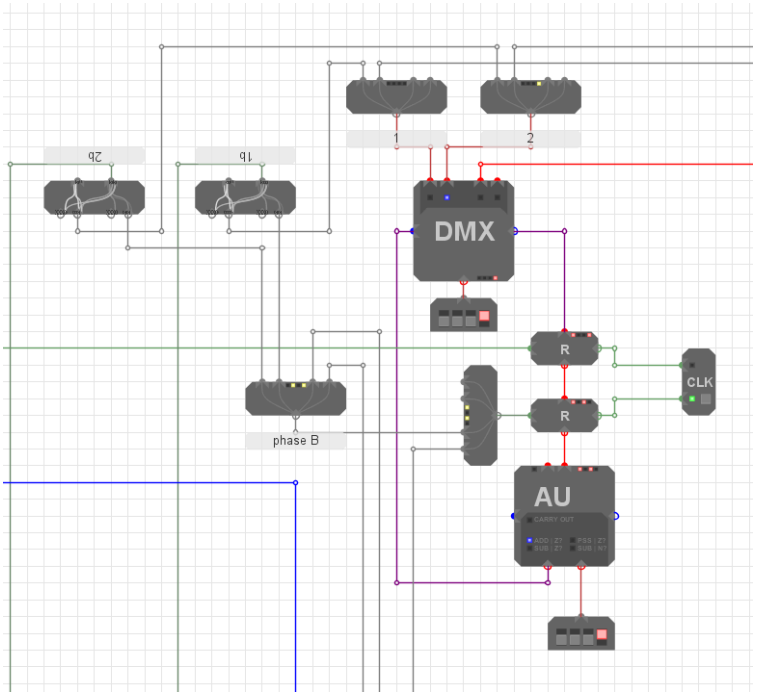
In this project, I followed the 4-bit CPU material from computer architecture and used Logisim and Verilog to complete the entire design. Picture 1 shows the complete blueprint for the 4-bit CPU. This CPU can be divided into three parts: clock, operand, and opcode.



Picture 1. Blueprint of 4-bit CPU in the ModuleSim

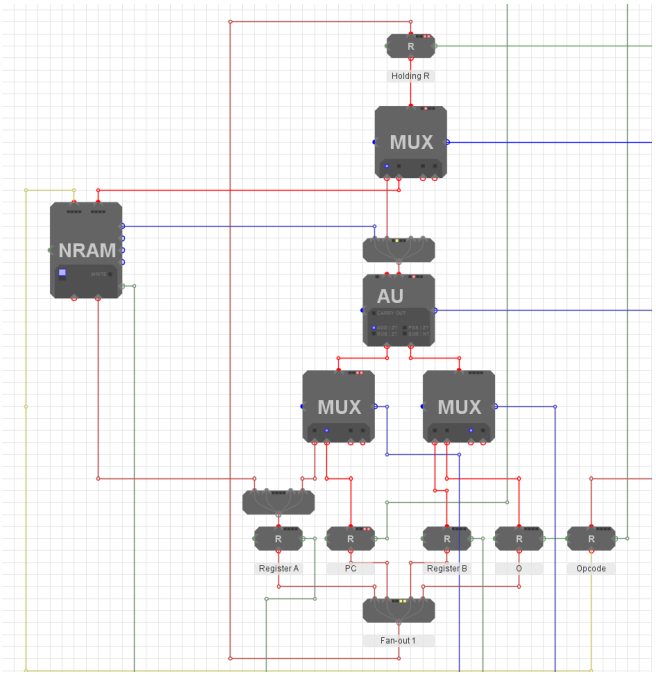
ModuleSim is a simple circuit design tool used in the computer science module "An Overview of Computer Architecture" at the University of Bristol. This tool can be found on GitHub at https://github.com/TeachingTechnologistBeth/ModuleSim/releases.

In the clock section, we obtain two signals. The first, called the A signal, enables the 1A, 2A, and 3A pipeline stages. The second, called the B signal, enables the 1B, 2B, and 3B pipeline stages. We also include an adder to advance the stages. After one cycle of the system clock, the adder increments a register by 1. The value in this register is then used to control a multiplexer, which outputs the enable signal to the specific channel corresponding to the appropriate stage.



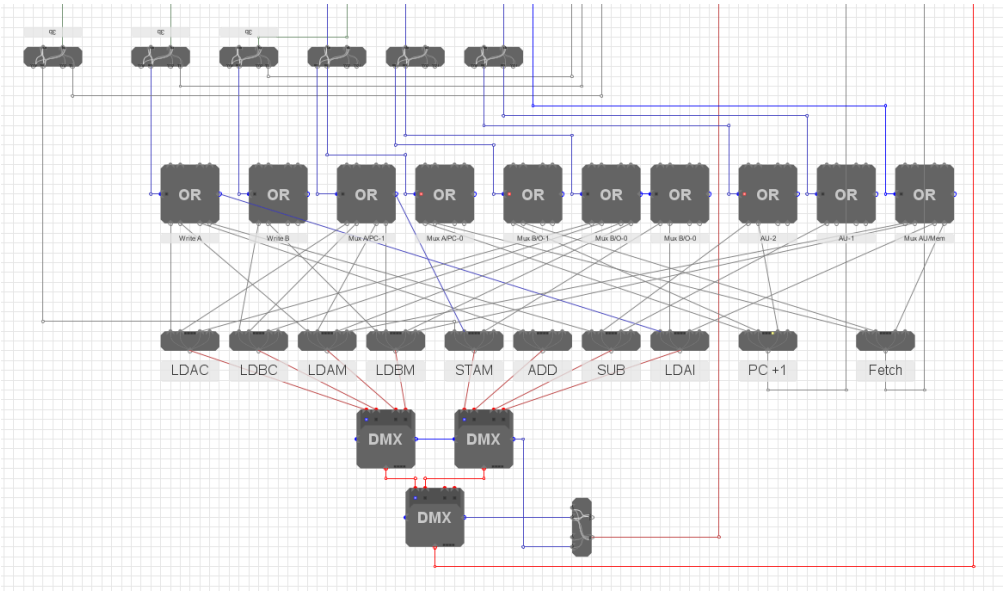
Picture 2 Clock section of 4-bit CPU in the ModuleSim.

In the operand section, we set up four registers: A, B, PC, and O, to store data from memory. Two multiplexers, controlled by operands, are used to select the specific registers. The first multiplexer handles the selection between the A register and the PC register, while the second one handles the selection between the B register and the O register. The arithmetic unit provides four functions: NOT, ADD, SUB, and comparison. The details of these functions will be described in the AU (Arithmetic Unit) design section. The NRAM unit consists of memory and control parts. We store data such as values and opcodes in the memory.



Picture 3, Operand section of 4-bit CPU in the ModuleSim.

In the opcode section, to handle more than eight types of instructions, we use three multiplexers. We divide the 4-bit opcode into the first 2 bits and the last 2 bits. The first 2 bits control the initial multiplexer, while the last 2 bits control the final multiplexer. Each multiplexer has four outputs. Therefore, this dual-multiplexer design can control 4 * 4 outputs.



Picture 4. Opcode execution section of 4-bit CPU in the ModuleSim.

ISA and Pipeline design

Instruction design

The opcode and operand designs are designed to match the requests of class materials.

Table 1. ISA design and description.

Machine code	Mnemonic	Description	Example	
0000	LDAC	Load register A with the operand value	00000101	Load A with 5
0001	LDBC	Load register B with the operand value	00010110	Load B with 6
0010	LDAM	Load register A with the value stored in the memory location that is addressed by the	00100101	Load A with the content of the 5th byte

		operand		
0011	LDBM	Load register B with the value stored in the memory location that is addressed by the operand	00110110	Load A with the content of the 6th byte
0100	STAM	Stored the value of A in the memory location addressed by the operand and reset the 4 MSBs of this location	01001010	Store A in the 10th byte of the memory and reset the 4 MSBs of this location
0101	ADD	A <= A+B	0101****	
0110	SUB	A <= A+B	0110****	
0111	LDAI	Load register A with the content of the memory location at an address calculated by adding the A register and the operand	01110011	Load A with the content of the memory [A+3]

Pipeline design

The pipeline has 3 stages, fetch instruction, increment the program counter and execution the instruction.

The detailed pipeline data,

1. Fetch the next instruction
    a. Fetch the next instruction from the memory and store its operand it in the holding register – phase “1a”.
    b. Store the instruction in the Opcode and O registers - phase “1b”.
2. Increment the program counter
    a. Add one to the value of the PC and store it in the holding register – phase “2a”.
    b. Store the new PC value in the PC register – phase “2b”.
3. Execute the instruction
    a. Store the result of the execution which can come either from the AU or from the memory for LDAM and LDBM in the holding register – phase “3a”.
    b. Store the result of the execution which come from the holding register in its final destination which can be a register or a memory location for STAM– phase “3b”.

Unit design

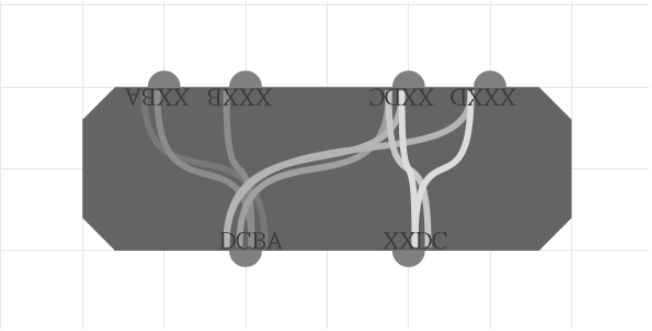
In this section, we will introduce the control values of the unit in ModuleSim, draw the circuit in Logisim, and write the Verilog code.

Split

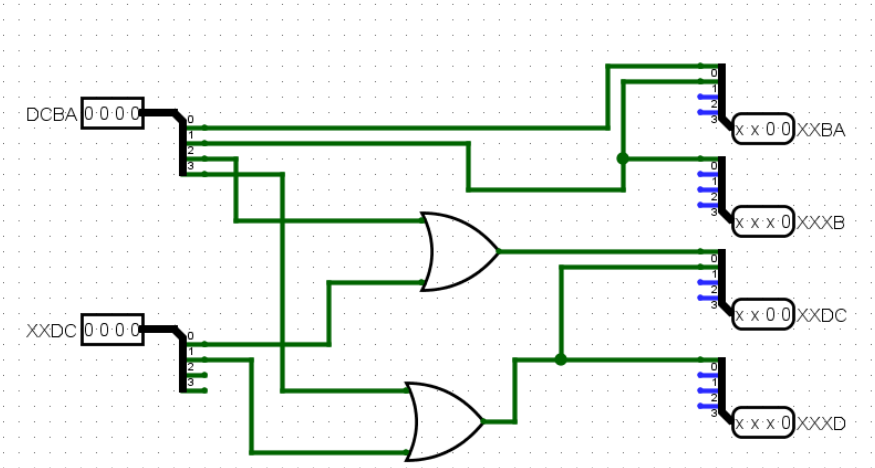
This unit splits the signal into a specific format, making it easier for designers to obtain the desired values.

ModuleSim description:

Input a	XXBA
Input b	XXXB
Input c	XXDC
Input d	XXXD
Output a	DCBA
Output b	XXDC



Logisim design:



Verilog design:

```
module main;
    reg [3:0] a;
    reg [3:0] b;
    wire [3:0] c;
    wire [3:0] d;
```

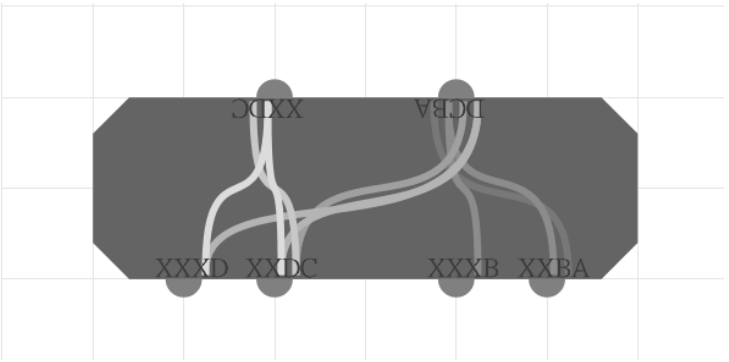
```
    wire [3:0] e;
    wire [3:0] f;

splite splite(
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .e(e),
    .f(f)
);

  initial
    begin
      $monitor("a:%b b:%b c:%b d:%b e:%b f:%b",a,b,c,d,e,f);
    end

    initial begin
        #1; a=4'b0000;  b=4'b0000;
        #1; a=4'b0001;  b=4'b0000;
        #1; a=4'b0011;  b=4'b0001;
        #1; a=4'b0000;  b=4'b0001;
        #1; a=4'b0001;  b=4'b0000;
    end

endmodule

module splite(
    input [3:0] a,
    input [3:0] b,
    output reg [3:0] c,
    output reg [3:0] d,
    output reg [3:0] e,
    output reg [3:0] f
);

    always @(*) begin
        c = {3'b000,d[0]};
        d = {2'b00,d[0],c[0]};
        e = {3'b000,b[0]};
        f= {2'b0,b[0],a[0]};
    end

endmodule
```

Merge

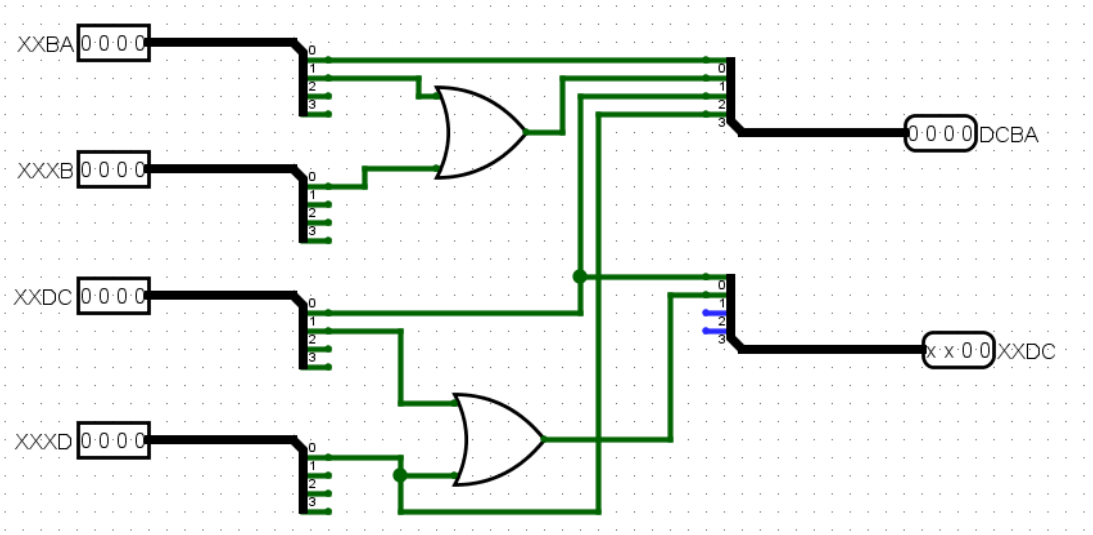
This unit merges the signal into a specific format, making it easier for designers to obtain the desired values.

ModuleSim description:

Input a	DCBA
Input b	XXDC
Output a	XXBA
Output b	XXXB
Output c	XXDC
Output d	XXXD



Logisim design:



Verilog design:

```
module main;
    reg [3:0] a;
    reg [3:0] b;
    reg [3:0] c;
    reg [3:0] d;
    wire [3:0] e;
    wire [3:0] f;

merge mer(
    .a(a),
```

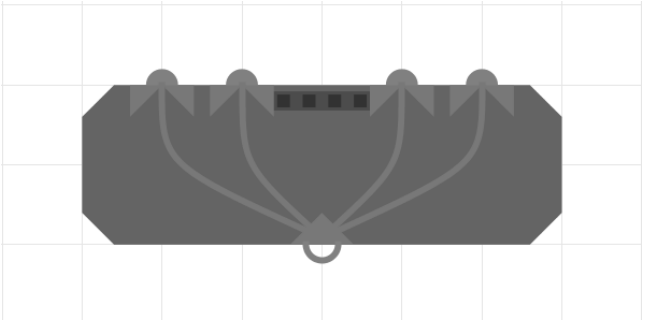
```
    .b(b),
    .c(c),
    .d(d),
    .e(e),
    .f(f)
);

  initial
    begin
      $monitor("a:%b b:%b c:%b d:%b e:%b f:%b",a,b,c,d,e,f);
    end

    initial begin
        #1; a=4'b0000;  b=4'b0000;  c=4'b0000;  d=4'b0000;
        #1; a=4'b0001;  b=4'b0000;  c=4'b0001;  d=4'b0000;
        #1; a=4'b0011;  b=4'b0001;  c=4'b0001;  d=4'b0011;
        #1; a=4'b0000;  b=4'b0001;  c=4'b0000;  d=4'b0001;
        #1; a=4'b0001;  b=4'b0000;  c=4'b0001;  d=4'b0001;
    end

endmodule

module merge(
    input [3:0] a,
    input [3:0] b,
    input [3:0] c,
    input [3:0] d,
    output reg [3:0] e,
    output reg [3:0] f
);

    always @(*) begin
        e = {a[0],b[0],c[0],d[0]};
        f= {2'b0,c[0],d[0]};
    end

endmodule
```

Fanout

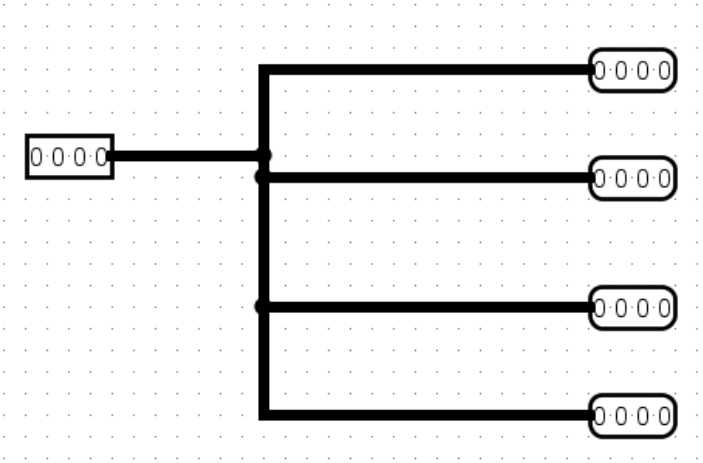
This unit transforms a single signal channel into multiple channels. Compared to Logisim, which allows for direct circuit drawing, using a one-to-one line approach provides a clearer way to handle signals in the ModuleSim.

ModuleSim description:

Input a	DCBA
Output a	DCBA
Output b	DCBA
Output c	DCBA
Output d	DCBA



Logisim design



Verilog design:

```
module main;
    reg [3:0] a;
    wire [3:0] b;
    wire [3:0] c;
    wire [3:0] d;
    wire [3:0] e;

fanout fanout(
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .e(e)
```

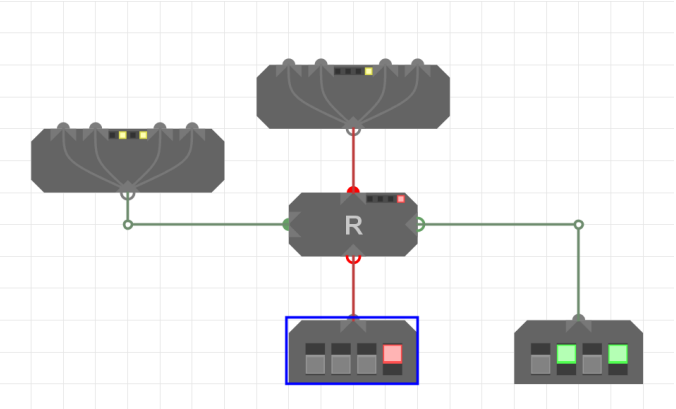
```
);

  initial
    begin
      $monitor("a:%b b:%b c:%b d:%b e:%b",a,b,c,d,e);
    end

    initial begin
        #1; a=4'b0000;
        #1; a=4'b0001;
        #1; a=4'b1111;
        #1; a=4'b0111;
        #1; a=4'b1001;
    end

endmodule

module fanout(
    input [3:0] a,
    output reg [3:0] b,
    output reg [3:0] c,
    output reg [3:0] d,
    output reg [3:0] e

);

    always @(*) begin
        b = a;
        c = a;
        d = a;
        e = a;
    end

endmodule
```

Register

The 4-bit register consists of four memory units, each based on a D latch. In addition to the register's memory unit, we combine the reset and enable functions. The 0 and 2 bits of the control input signal are the enable signals, and the 1 bit is the reset signal. Therefore, when the control input signal is X101, the register will be enabled to store data. Conversely, when the control input signal is XX1X, the register will reset its value.

ModuleSim description:

Input a	source
Input Control	clk + operand
Output a	result
Output Control	Equal Input Control

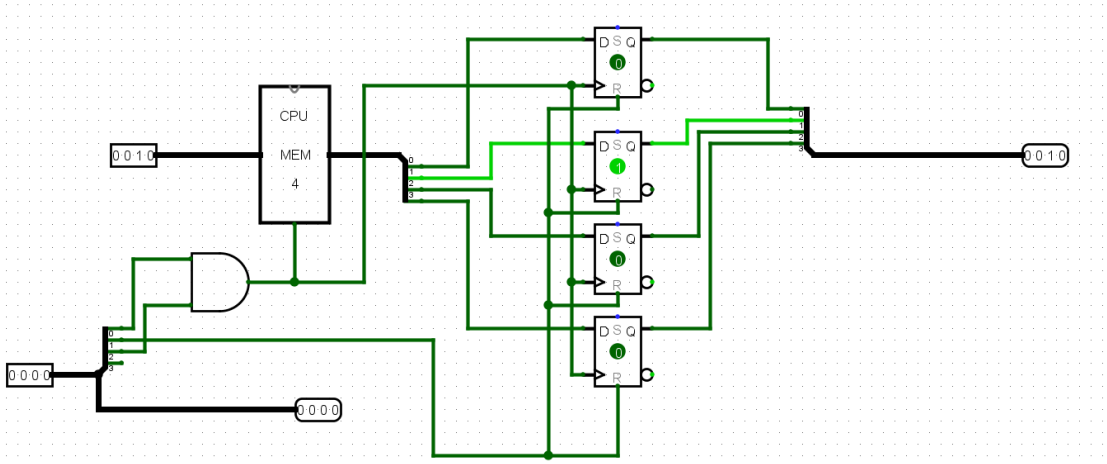


Input Control Signal function Table:

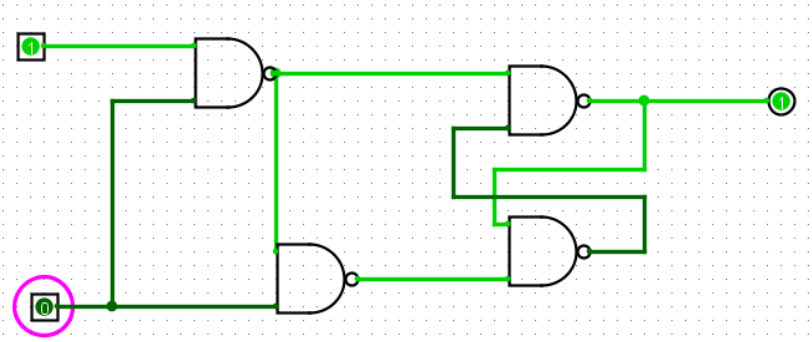
Bit	Function	Values and outputs
0	Clock	0 - no change, 1 - read input if enable=1
1	Reset	Reset the register to 0
2	Enable	0 - no change, 1 - read input if clock=1

Logisim design:

(1) 4-bit register



(2) memory units



Verilog design:

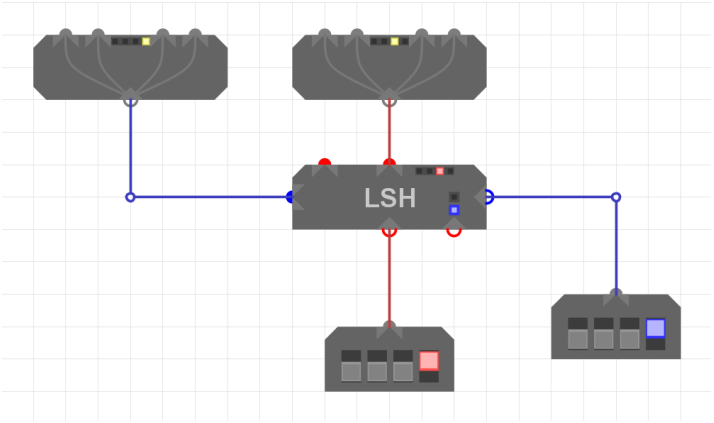
```
module main;
    reg [3:0] a;    reg [3:0] b;
    wire [3:0] c;    wire [3:0] d;

register register(
    .a(a), .b(b), .c(c), .d(d)
);

  initial begin
      $monitor("a:%b b:%b c:%b d:%b",a,b,c,d);
    end

    initial begin
        #1; a=4'b0001; b=4'b0000;
        #1; a=4'b0001; b=4'b0001;
        #1; a=4'b1111; b=4'b0101;
        #1; a=4'b0101; b=4'b0010;
        #1; a=4'b0111; b=4'b0111;
        #1; a=4'b1001; b=4'b0101;
    end
endmodule

module register(
    input [3:0] a,
    input [3:0] b,
    output reg [3:0] c,
    output reg [3:0] d

);
    initial begin
        assign d = b;
    end

    always @(b) begin
        if (b[1] == 1) begin
            c = 4'b0000 ;
        end else if (b == 4'b0101) begin
            c = a ;
        end
    end
endmodule
```

LSH - Left Shift

The LSH unit consists of a demultiplexer and several splitters. The demultiplexer selects a specific channel based on the input control signal. If the input control signal is 1'b1, the demultiplexer will select the first channel, thereby shifting the connection by 1 bit and so on.

ModuleSim description:

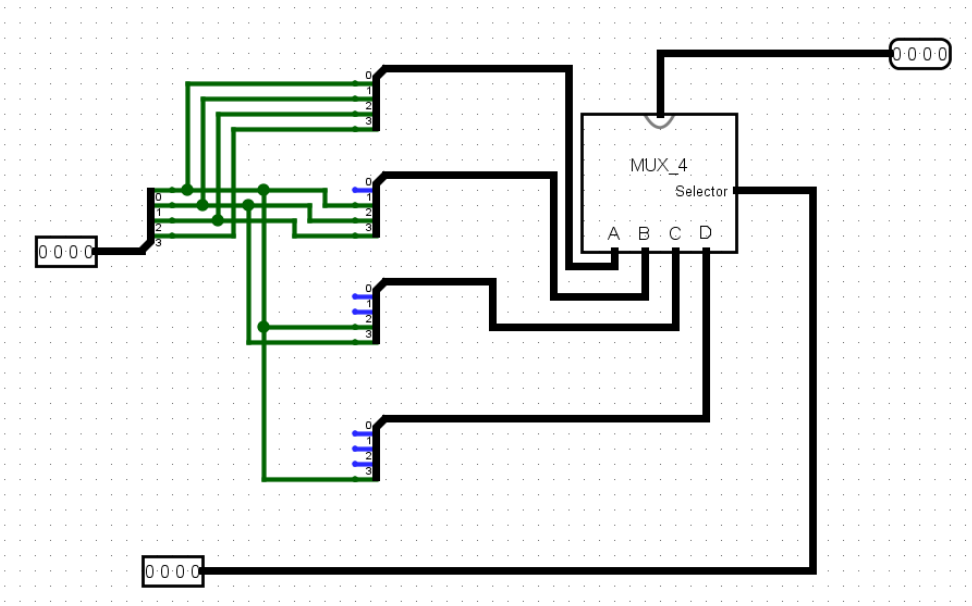
Input A	DCBA
Input Control	
Output A	CBAX BAXX AXXX XXXX
Output Control	Equal Input Control



Input Control

	Values and outputs
0001	Input A left shift 1-bit
0010	Input A left shift 2-bit
0011	Input A left shift 3-bit

Logisim design:



Verilog design:

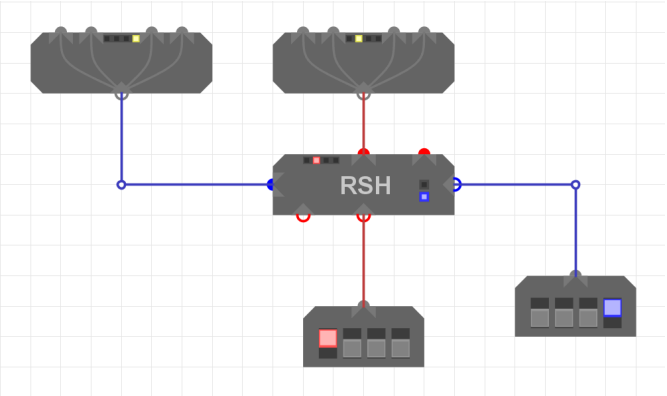
```
`timescale 1ns/1ns

module main;

    reg [3:0] a;
    reg [3:0] b;
    wire [3:0] c;

    lsh lsh(
        .a(a),
        .b(b),
        .c(c)
    );

 initial
    begin
      $monitor("a:%b b:%b c:%b ",a,b,c);
    end

    initial begin
        #1; a=4'b0001; b=4'b0000;
        #1; a=4'b0001; b=4'b0001;
        #1; a=4'b0001; b=4'b0010;
        #1; a=4'b0001; b=4'b0011;
        #1; a=4'b0001; b=4'b0100;
        #1; a=4'b0001; b=4'b0101;
    end

endmodule

module lsh(
    input  [3:0] a,
    input  [3:0] b,
    output reg [3:0] c
);

    always @(b) begin
       case (b)
            4'b0001: c = (a << 1);
            4'b0010: c = (a << 2);
            4'b0011: c = (a << 3);
            4'b0100: c = (a << 4);
            default: c = a; // Default assignment
        endcase
    end

endmodule
```

RSH - Right Shift

The RSH unit consists of a demultiplexer and several splitters. The demultiplexer selects a specific channel based on the input control signal. If the input control signal is 1'b1, the demultiplexer will select the first channel, thereby shifting the connection by 1 bit and so on.

ModuleSim description:

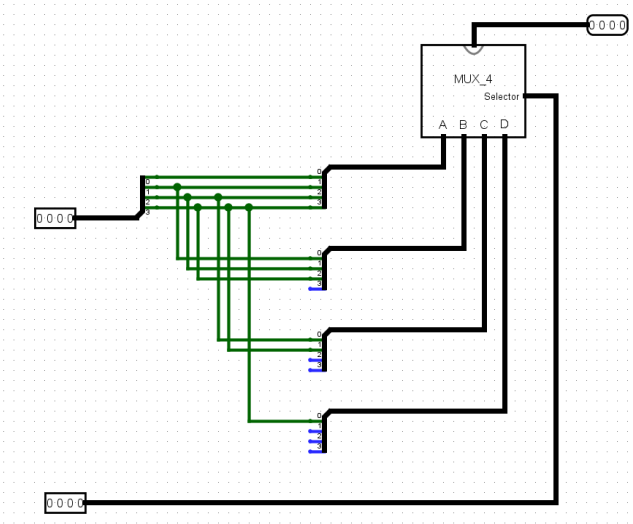
Input a	DCBA
Input Control	4’b0000
Output a	XDCB XXDC XXXD XXXX
Output Control	Equal Input Control



Input Control table:

	Values and outputs
0001	Input A right shift 1-bit
0010	Input A right shift 2-bit
0011	Input A right shift 3-bit

Logisim design:



Verilog design:

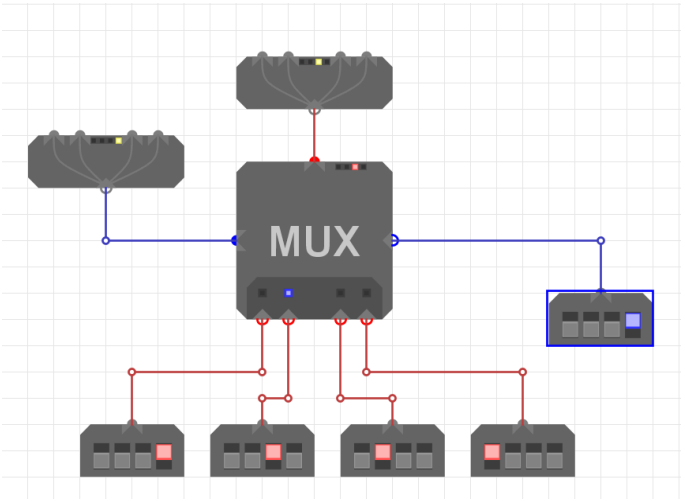
```
`timescale 1ns/1ns

module main;

    reg [3:0] a;
    reg [3:0] b;
    wire [3:0] c;

    lsh lsh(
        .a(a),
        .b(b),
        .c(c)
    );


 initial
    begin
      $monitor("a:%b b:%b c:%b ",a,b,c);
    end

    initial begin
        #1; a=4'b1000; b=4'b0000;
        #1; a=4'b1000; b=4'b0001;
        #1; a=4'b1000; b=4'b0010;
        #1; a=4'b1000; b=4'b0011;
        #1; a=4'b1000; b=4'b0100;
        #1; a=4'b1000; b=4'b0101;
    end


endmodule


module lsh(
    input  [3:0] a,
    input  [3:0] b,
    output reg [3:0] c
);

    always @(b) begin
       case (b)
            4'b0001: c = (a >> 1);
            4'b0010: c = (a >> 2);
            4'b0011: c = (a >> 3);
            4'b0100: c = (a >> 4);
            default: c = a; // Default assignment
        endcase
    end


endmodule
```

MUX - Multiplexer

The 4-bit multiplexer consists of four 1-bit multiplexers. In this project, we use the 2 least significant bits (LSBs) to handle the input signals. According to binary principles, only 2 bits are needed to control 2*2 channels. If we apply an 8-channel structure in the future, we will use 3 bits to control the multiplexer.

ModuleSim description:

	Format
Input a	4’b0000
Input b	4’b0000
Input c	4’b0000
Input d	4’b0000
Input Control	4’b0000
Output a	4’b0000
Output Control	Equal Input Control

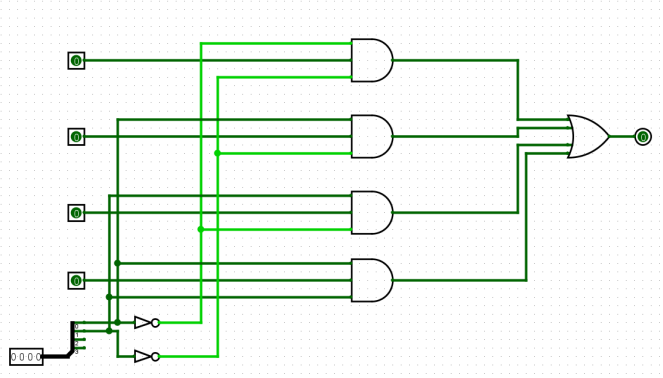


Input control signal table:

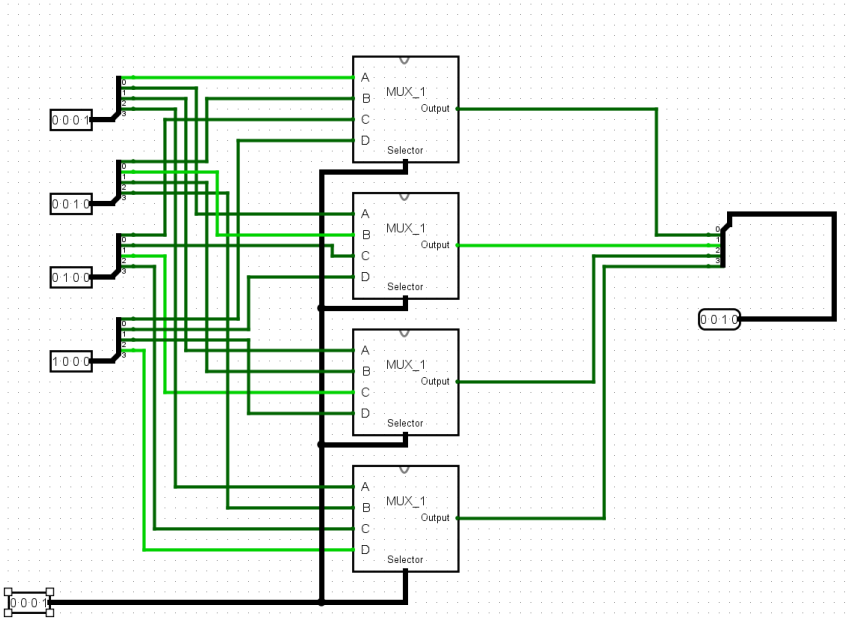
Input Control	Input selected
XX00	A
XX01	B
XX10	C
XX11	D

Logisim design:

(1) 1-bit multiplexers



(2) 4-bit multiplexer



Verilog design:

```
module main;

    reg [3:0] a;
    reg [3:0] b;
    reg [3:0] c;
    reg [3:0] d;
    reg [3:0] opcode;
    wire [3:0] opcode1;
    wire [3:0] result;

    mux mux(
        .opcode(opcode),
        .a(a),
        .b(b),
        .c(c),
        .d(d),
        .opcode1(opcode1),
        .result(result)
    );

    initial begin
        $display("Hello, World");
        $monitor("a:%b b:%b c:%b d:%b opcode:%b _opcode:%b result:%b", a, b, c, d,
opcode, opcode1, result);
    end

    initial begin
        #1; a = 4'b0001; b = 4'b0001; c = 4'b0001; d = 4'b0001; opcode = 4'b0001;
        #1; a = 4'b0010; b = 4'b0010; c = 4'b0010; d = 4'b0010; opcode = 4'b0010;
        #1; a = 4'b0100; b = 4'b0100; c = 4'b0100; d = 4'b0100; opcode = 4'b0011;
        #1; a = 4'b1000; b = 4'b1000; c = 4'b1000; d = 4'b1000; opcode = 4'b0000;
```

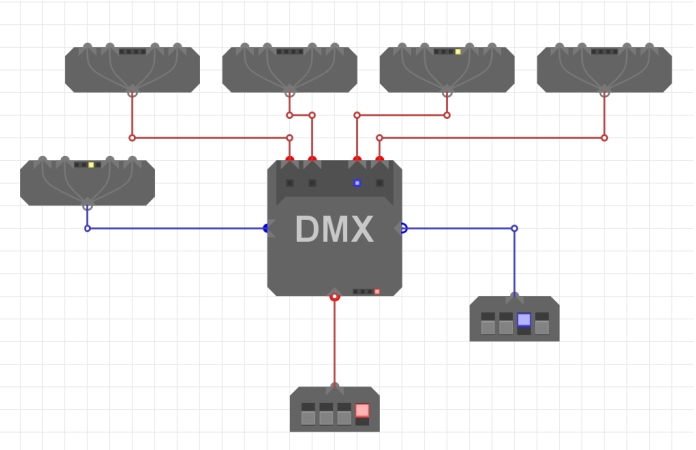
```
    end

endmodule

module mux(
    input [3:0] opcode,
    input [3:0] a,
    input [3:0] b,
    input [3:0] c,
    input [3:0] d,
    output reg [3:0] opcode1,
    output reg [3:0] result
);

    always @(*) begin
        case(opcode)
            4'b0000: result = a;
            4'b0001: result = b;
            4'b0010: result = c;
            4'b0011: result = d;
            default: result = 4'b0000; // Default case to avoid latches
        endcase
    end

    always @(*) begin
            opcode1 = opcode;
    end


endmodule
```

DMX - Demultiplexer

Compared to a multiplexer, which selects one of four inputs and exports it to a single channel, the demultiplexer selects the output channel from four output channels. The 4-bit demultiplexer consists of four 1-bit demultiplexers, and it outputs the value to only one specific channel at a time.

ModuleSim description:

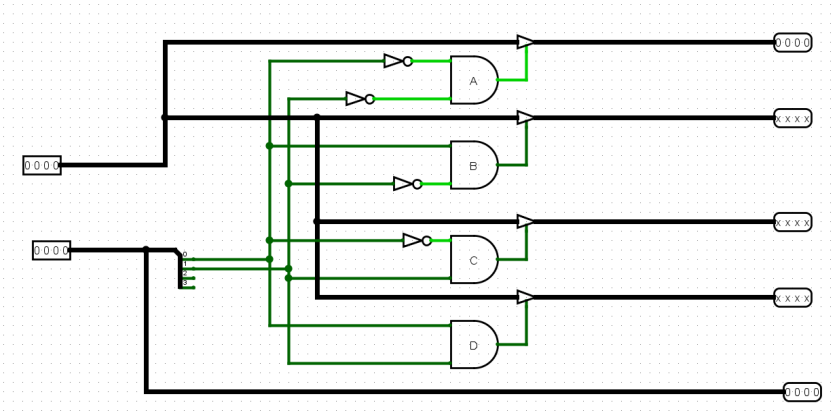
	Format
Input a	4’b0000
Input Control	4’b0000
Output a	4’b0000
Output b	4’b0000
Output c	4’b0000
Output d	4’b0000
Output Control	Equal Input Control



Input control signal table:

Control input	Input selected
XX00	A
XX01	B
XX10	C
XX11	D

Logisim design:



Verilog design:

```
module main;

    wire [3:0] a;
    wire [3:0] b;
    wire [3:0] c;
    wire [3:0] d;
    reg [3:0] opcode;
    wire [3:0] opcode1;
    reg [3:0] source;

    dmx dmx(
        .opcode(opcode),
        .a(a),
        .b(b),
        .c(c),
        .d(d),
        .opcode1(opcode1),
        .source(source)
    );

    initial begin
        $display("Hello, World");
        $monitor("a:%b b:%b c:%b d:%b opcode:%b opcode1:%b source:%b", a, b, c, d,
opcode, opcode1, source);
    end

    initial begin
        #1; opcode = 4'b0000; source = 4'b0001;
        #1; opcode = 4'b0001; source = 4'b0001;
        #1; opcode = 4'b0010; source = 4'b0001;
        #1; opcode = 4'b0011; source = 4'b0001;
    end

endmodule

module dmx(
    input [3:0] opcode,
    output reg [3:0] a,
    output reg [3:0] b,
    output reg [3:0] c,
    output reg [3:0] d,
    output reg [3:0] opcode1,
    input [3:0] source
);

    always @(*) begin
        a = 4'b0000;
        b = 4'b0000;
        c = 4'b0000;
        d = 4'b0000;

        case(opcode)
            4'b0000: a = source;
            4'b0001: b = source;
```

```
            4'b0010: c = source;
            4'b0011: d = source;
            default: begin
                a = 4'b0000;
                b = 4'b0000;
                c = 4'b0000;
                d = 4'b0000;
            end
        endcase
    end

    always @(*) begin
            opcode1 = opcode;
    end


endmodule
```

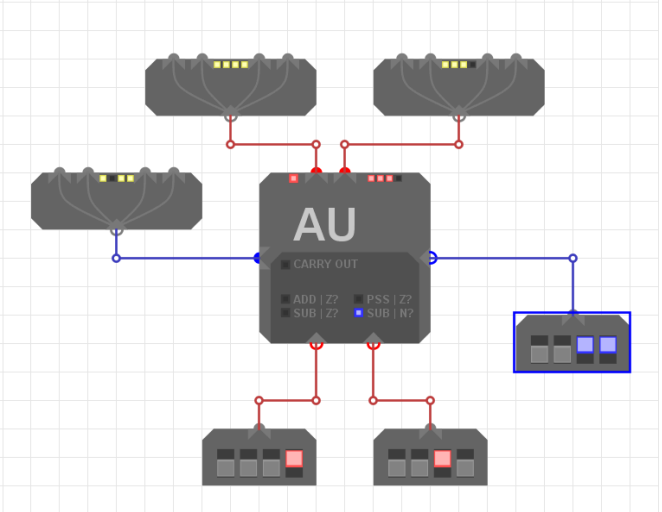
AU - Arithmetic Unit

This AU (Arithmetic Unit) consists of a full adder, a subtractor, and a comparator. The full adder is made up of four 1-bit adders, each supporting carry-in and carry-out functions. The subtractor uses the full adder to implement its function by using 2's complement to represent negative integers. Therefore, the subtractor's operation involves adding a positive integer to a 2's complement.

The comparator in the AU unit provides three comparison methods: greater, smaller, and equal. In this project, the AU unit compares two input values and outputs the result from the top-left output site. Note that the comparison function is only supported in the subtractor function.

ModuleSim description:

	Format
Input a	4'b0000
Input b	4'b0000
Input Control	4'b0000
Output a	4'b0000
Output b	4'b0000
Output Control	Equal Input Control

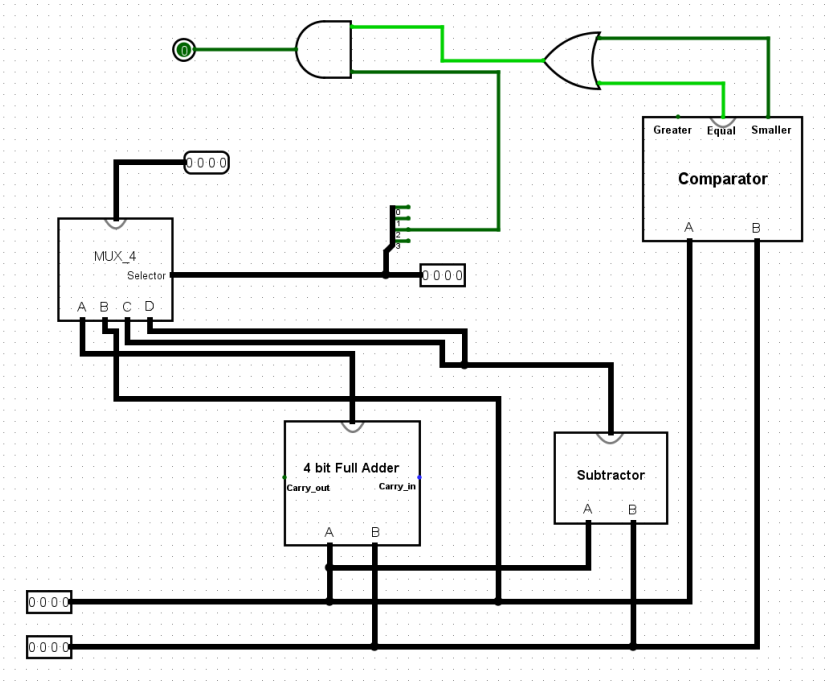


Input control signal table:

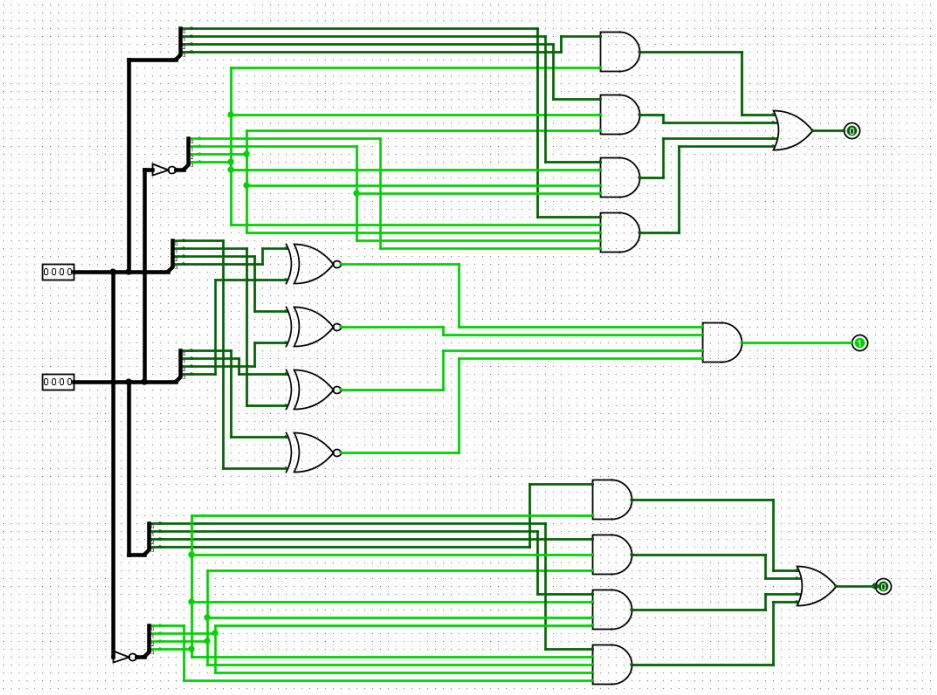
Control Inputs (right input)	Function selected	Comparison selected
x000	Addition	Output = 0
x001	Pass through A	Output = 0
x110	A - B	A = B
x111	A - B	A < B

Logisim design:

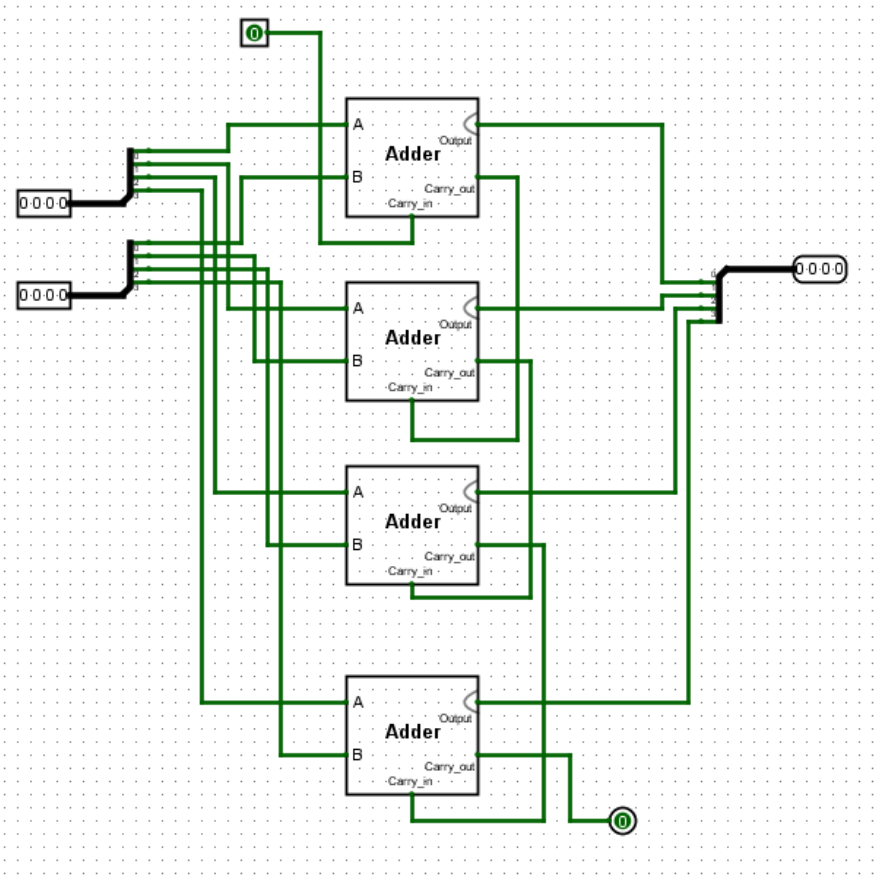
(1) AU (Arithmetic Unit):



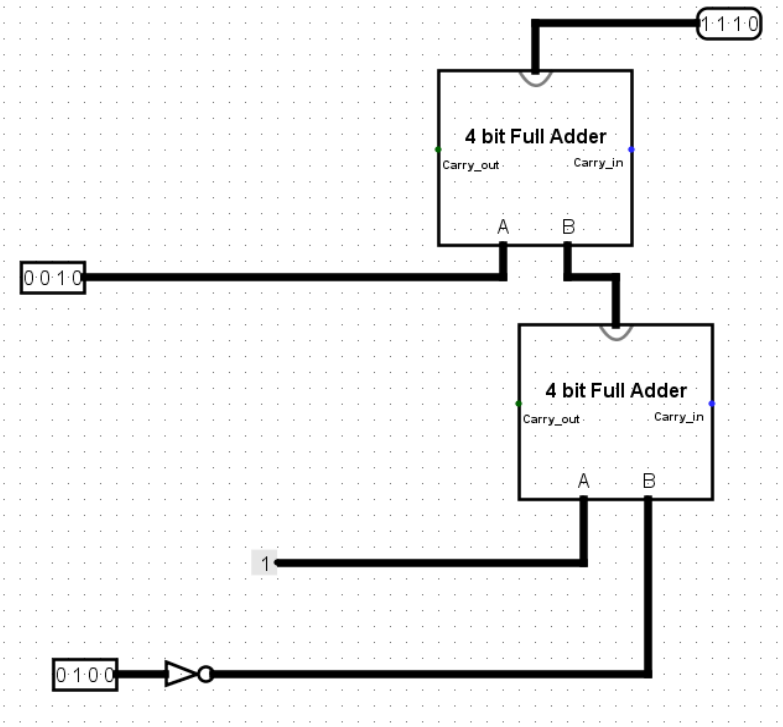
(2) Comparator:



(3) Full adder:



(4) Subtractor:



Verilog design:

```
`timescale 1ns/1ns

module main;

    reg [3:0] a;
    reg [3:0] b;
    reg [3:0] opcode;
    wire [3:0] result;
    wire [3:0] cmp;
    wire [3:0] overflow;

    au au(
        .a(a),
        .b(b),
        .opcode(opcode),
        .result(result),
        .cmp(cmp),
        .overflow(overflow)
    );

 initial
    begin
      $monitor("a:%b b:%b opcode:%b result:%b cmp:%b
overflow:%b",a,b,opcode,result,cmp,overflow);
    end

    initial begin
        #1; a=4'b0000; b=4'b0000; opcode= 4'b0000;
        #1; a=4'b0001; b=4'b0001; opcode= 4'b0000;
        #1; a=4'b1000; b=4'b1000; opcode= 4'b0000;
        #1; a=4'b0100; b=4'b0010; opcode= 4'b0001;
        #1; a=4'b0100; b=4'b0010; opcode= 4'b0111;
        #1; a=4'b0100; b=4'b0100; opcode= 4'b0110;
    end

endmodule

module au(
    input  [3:0] a,
    input  [3:0] b,
    input  [3:0] opcode,
    output reg [3:0] result,
    output reg [3:0] cmp,
    output reg [3:0] overflow
);

    reg [4:0] temp_result;

    always @(opcode or a or b) begin
        overflow = 0; // Initialize overflow to 0 by default

        case (opcode[2:0])
            3'b000: begin
                temp_result = a + b;
```

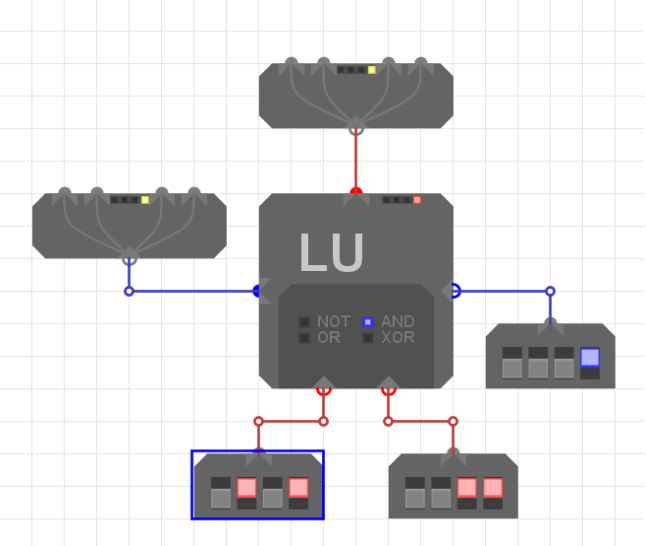
```
                overflow = (temp_result > 4'b1111) ? 1 : 0; // Check for overflow
                result = temp_result[3:0];
            end
            3'b001: begin
                result = a;
            end
            3'b110: begin
                result = a - b;
                if (a == b) begin
                    cmp = 4'b0001;
                end else begin
                    cmp = 4'b0000;
                end
            end
            3'b111: begin
                result = a - b;
                if (a < b) begin
                    cmp = 4'b0001;
                end else begin
                    cmp = 4'b0000;
                end
            end
            default: begin
                result = a; // Default assignment
            end
        endcase
    end
endmodule
```

LU - Logic Unit

The LU (Logic Unit) uses a multiplexer and several logic gates. In the sub-functions of the LU unit, we use NOT, AND, OR, and XOR logic gates to process the input values. An input control signal is used to enable specific functions. Interestingly, we don’t use an explicit enable function in this unit. For example, if the input control signal is 4'b0001, the AND function is enabled while the others are disabled. The main reason for this approach is cost reduction. If we needed to control each channel individually, we would have to add a register to each channel. Therefore, we refer to the ModuleSim blueprint of the 4-bit CPU, and we can see a register behind the LU unit controlled by the pipeline.

ModuleSim description:

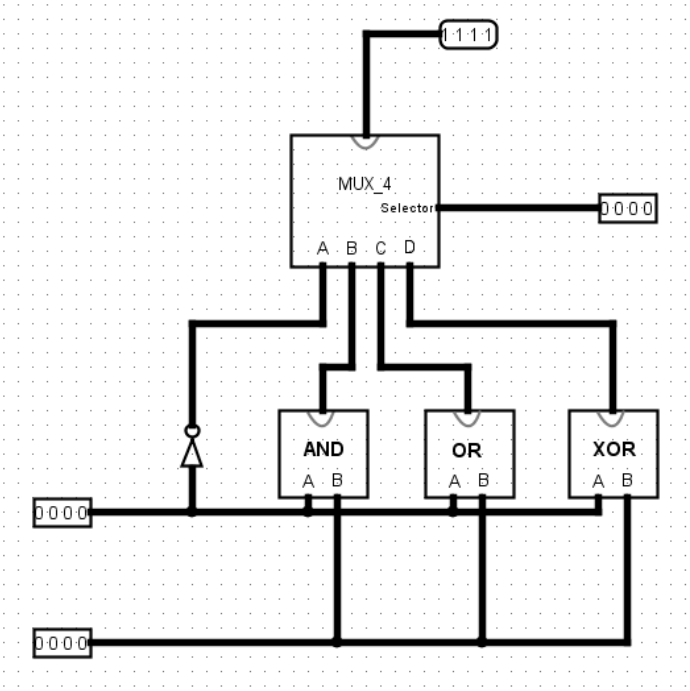
	Format
Input a	4’b0000
Input b	4’b0000
Input Control	4’b0000
Output a	4’b0000
Output Control	Equal Input Control



Input control signal table:

Control input	Function selected
XX00	NOT
XX01	AND
XX10	OR
XX11	XOR

Logisim design:



Verilog design:

```
`timescale 1ns/1ns

module main;

    reg [3:0] a;
    reg [3:0] b;
    reg [3:0] opcode;
    wire [3:0] result;
    wire [3:0] opcode1;

    lu lu(
        .a(a),
        .b(b),
        .opcode(opcode),
        .result(result),
        .opcode1(opcode1)
    );


 initial
    begin
      $display("Hello, World");
      $monitor("a:%b b:%b opcode:%b result:%b opcode1:%b",a,b,opcode,result,opcode);
    end

    initial begin
        #1; a=4'b0000; b=4'b0000; opcode= 4'b0000;
        #1; a=4'b0001; b=4'b0001; opcode= 4'b0000;
        #1; a=4'b1000; b=4'b1000; opcode= 4'b0000;
```

```
        #1; a=4'b0100; b=4'b0010; opcode= 4'b0001;
        #1; a=4'b0100; b=4'b0010; opcode= 4'b0111;
        #1; a=4'b0100; b=4'b0100; opcode= 4'b0110;
    end


endmodule


module lu(
    input  [3:0] a,
    input  [3:0] b,
    input  [3:0] opcode,
    output reg [3:0] result,
    output reg [3:0] opcode1
);


    always @(opcode or a or b) begin

        case (opcode[1:0])
            2'b00: begin
                result = ~a;
            end
            2'b01: begin
                result = a & b;
            end
            2'b10: begin
                result = a | b;
            end
            2'b11: begin
               result = a^b;
            end
            default: begin
                result = a; // Default assignment
            end
        endcase
    end


    always @(opcode) begin
        opcode1 = opcode;
    end
endmodule
```

RAM - Memory unit

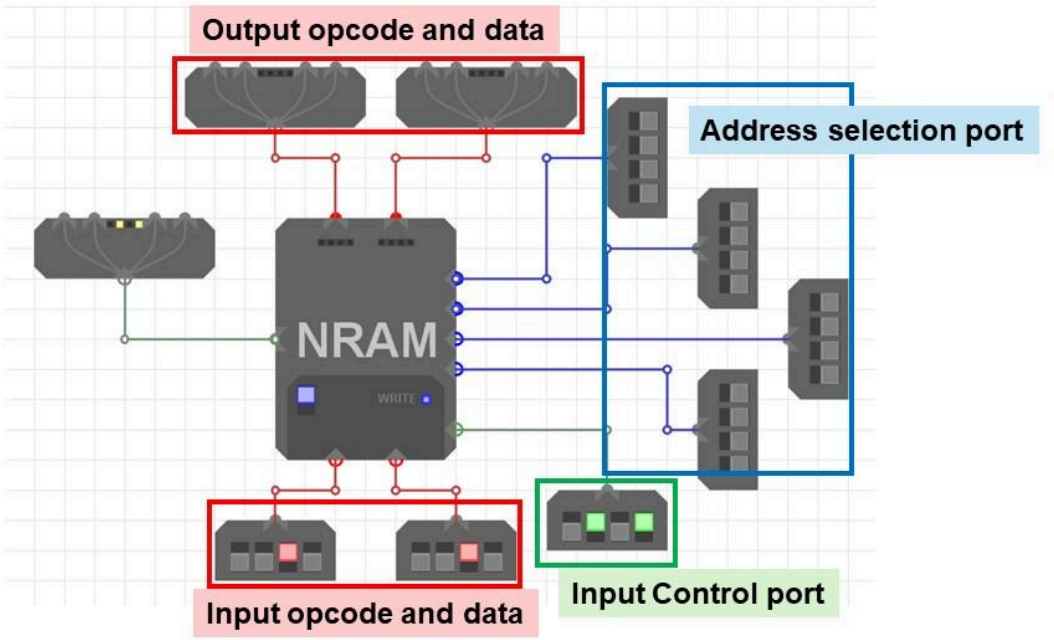
This unit supports 8-bit data input. In this project, we divide the 8 bits into two 4-bit segments: the first 4 bits are used for the opcode, and the last 4 bits are used for data. Consequently, you can see two input ports at the bottom of the diagram. On the right side, there are four blue lines representing the address input ports. Each input port supports a 4-bit input, allowing the RAM unit to address 2^16 locations.

The green line represents the control signal. When the input signal is 4'b0101, the RAM will store the value. At the top of the unit are the output ports, and the output result is determined by the selected address signals.

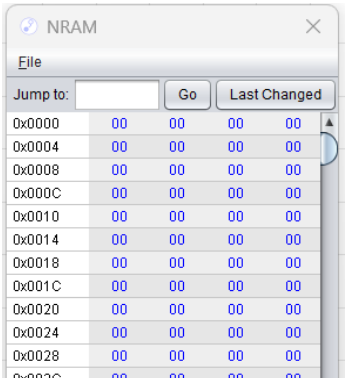
However, in Logisim, we modify the RAM structure. We use two 3-to-8 multiplexers to implement the RAM. Due to the limitation of 8 * 8 address selection inputs, this RAM supports only 64 bytes of data.

ModuleSim description:

(1) NRAM unit:



(2) NRAM data viewer:



Logisim design: