GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRONICS ENGINEERING

ELEC 451

Introduction to Digital Microelectronic Circuits

Final Project

Due: 14.1.2025

Prepared by: Masters of Magic

Umut Mehmet ERDEM	200102002025
Arda DERİCİ	200102002051
Mehmet Salih TURHAN	200102002053
Mehmet Fırat AYDIN	210102002025
Ahmet Ali TİLKİCİOĞLU	210102002163

Contents

1. Introduction

The design and implementation of an Arithmetic Logic Unit (ALU) serve as a foundational aspect of processor architecture, enabling efficient arithmetic and logical operations. This project aims to design, simulate, and layout an 8-bit ALU using the Skywater 130nm technology. The ALU integrates various arithmetic, logical, and shifting functionalities, providing the backbone for computational tasks in processors. The project emphasizes collaborative efforts, with the circuit divided into sub-blocks to ensure an optimized and functional design.

2. Design Approach

2.1. Circuit Design

In this project, the RTL (Register Transfer Level) schematic of the Arithmetic Logic Unit (ALU) design, written in Verilog, was generated using the Vivado tool. The RTL schematic visually represents the logical structure and data flow of the design. It is a crucial tool for understanding and verifying the operation of the design. The RTL schematic produced by Vivado clearly shows how the data paths connect various components and how each component functions.

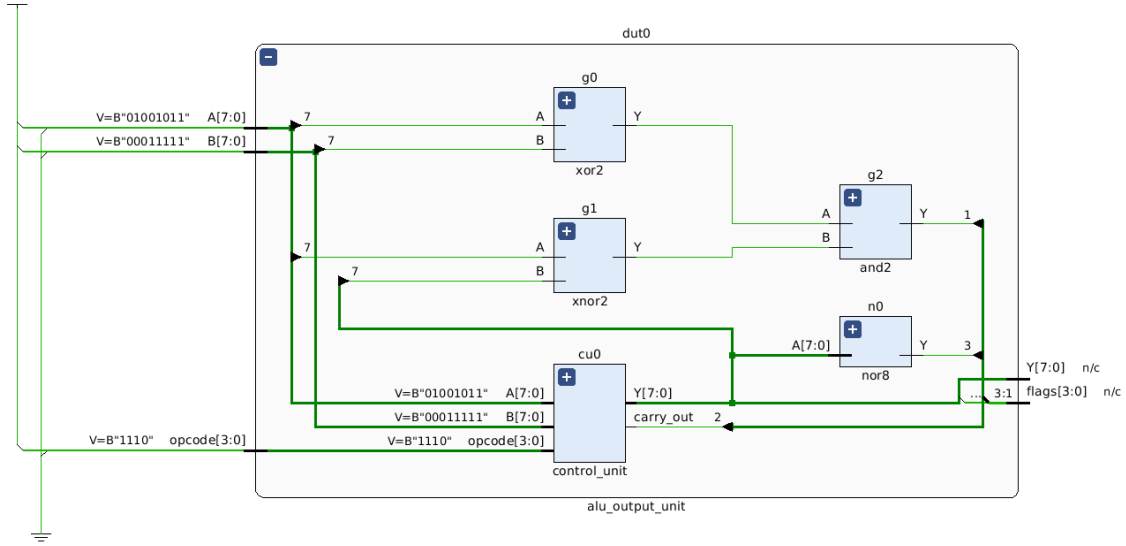

Figure 1: RTL schematic of the ALU design.

2.1.1. Arithmetic Unit

The Arithmetic Unit consists of three main components: an Adder, a Subtractor, and a Multiplier, each designed to perform basic arithmetic operations on 8-bit operands. Figure 2 shows the RTL schematic of the Arithmetic Unit.

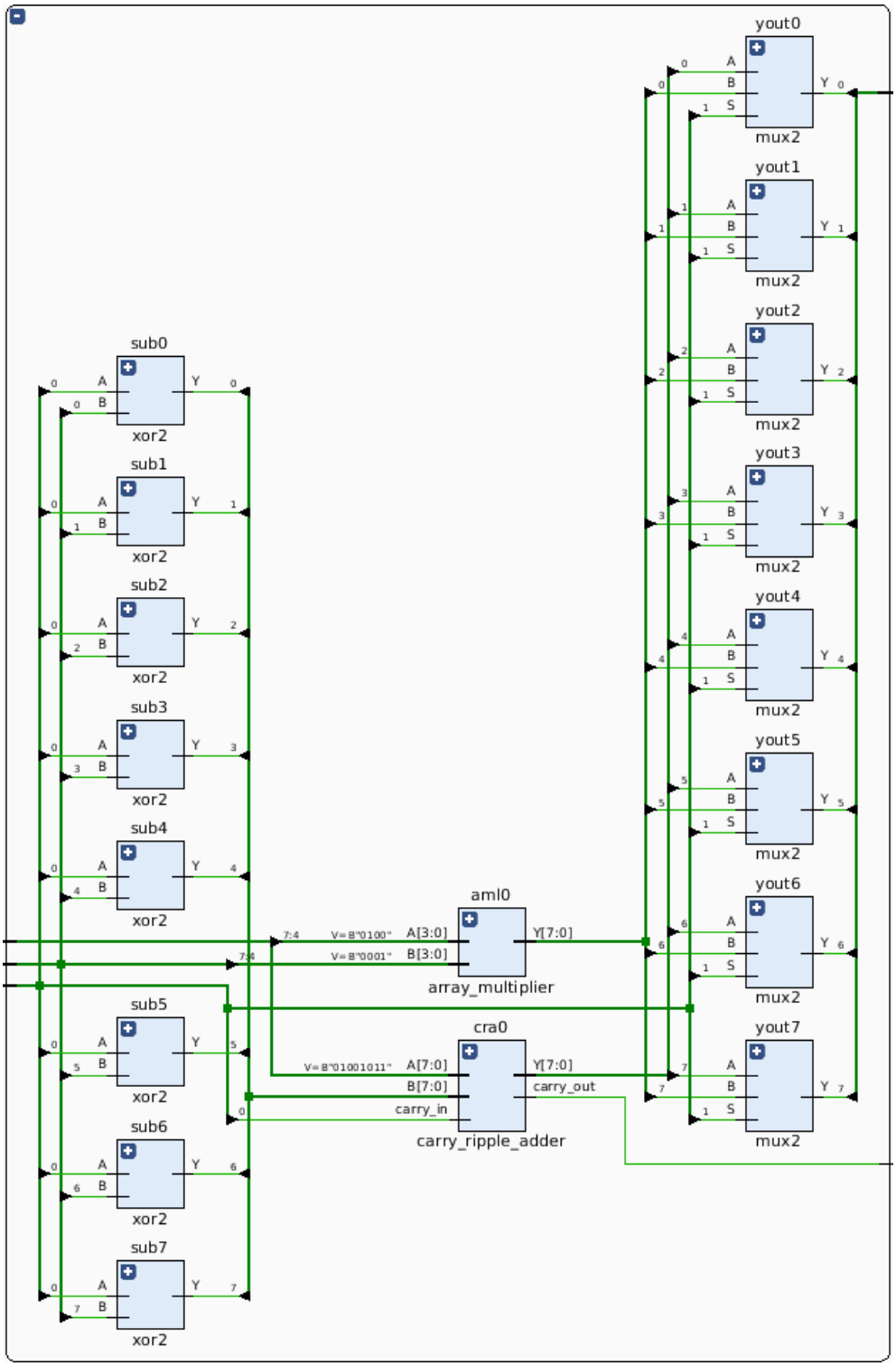

Figure 2: RTL schematic of the arithmetic unit.

2.1.2. Array Multiplier

The multiplier implements an array multiplier for 8-bit operands. It computes partial products, which are then added hierarchically to produce the result. Figure 3 illustrates the schematic of the Multiplier.

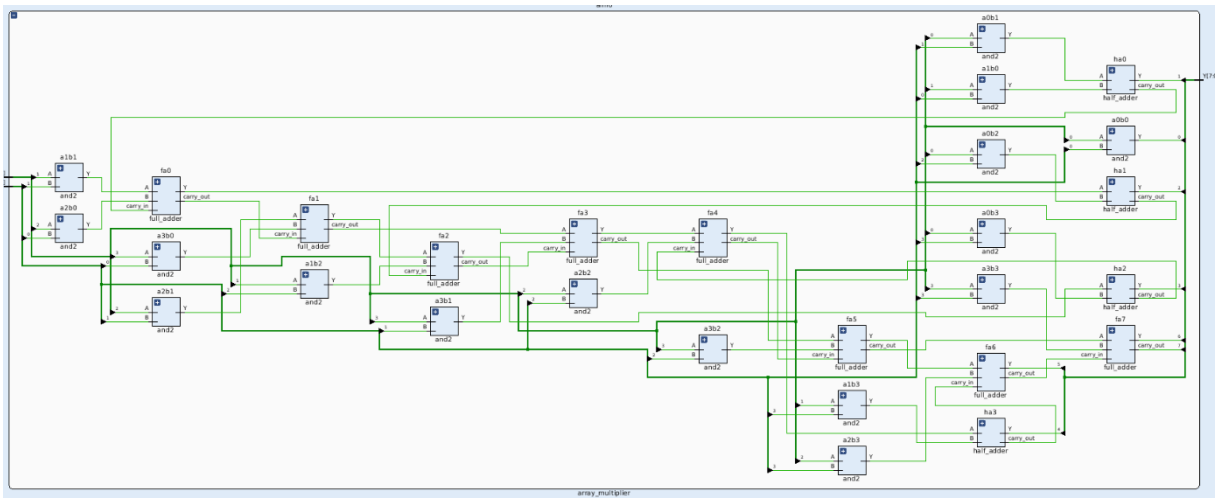

Figure 3: RTL schematic of the Multiplier.

2.1.3. Ripple Carry Adder

The Adder is an 8-bit Ripple Carry Adder (RCA) used for simplicity. It computes the sum of two 8-bit operands. Figure 4 illustrates the schematic of RCA.

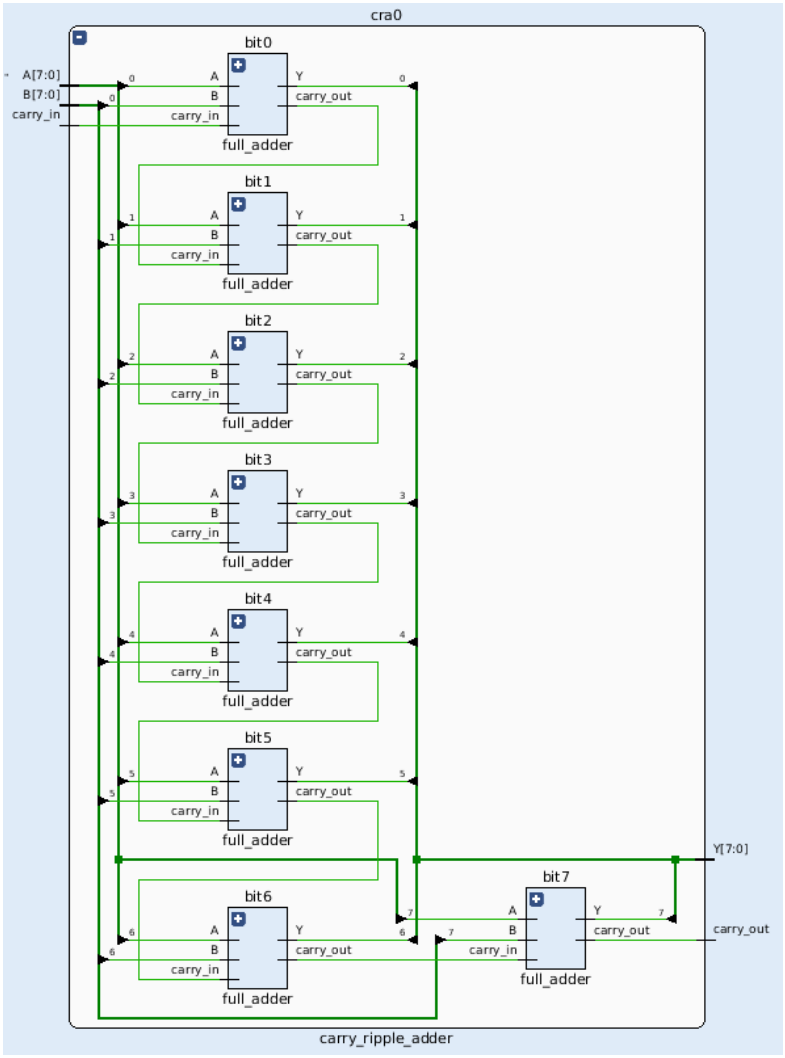

Figure 4: RTL schematic of the RCA

2.1.4. Shifter Unit

The Shifter Unit performs left and right shift operations on 8-bit input data. The left shift operation moves the bits of A [7:0] to the left by one or more positions, while the right shift can be implemented as either arithmetic (sign-extended) or logical (zero-filled). Figure 5 illustrates the RTL schematic of the Shifter Unit.

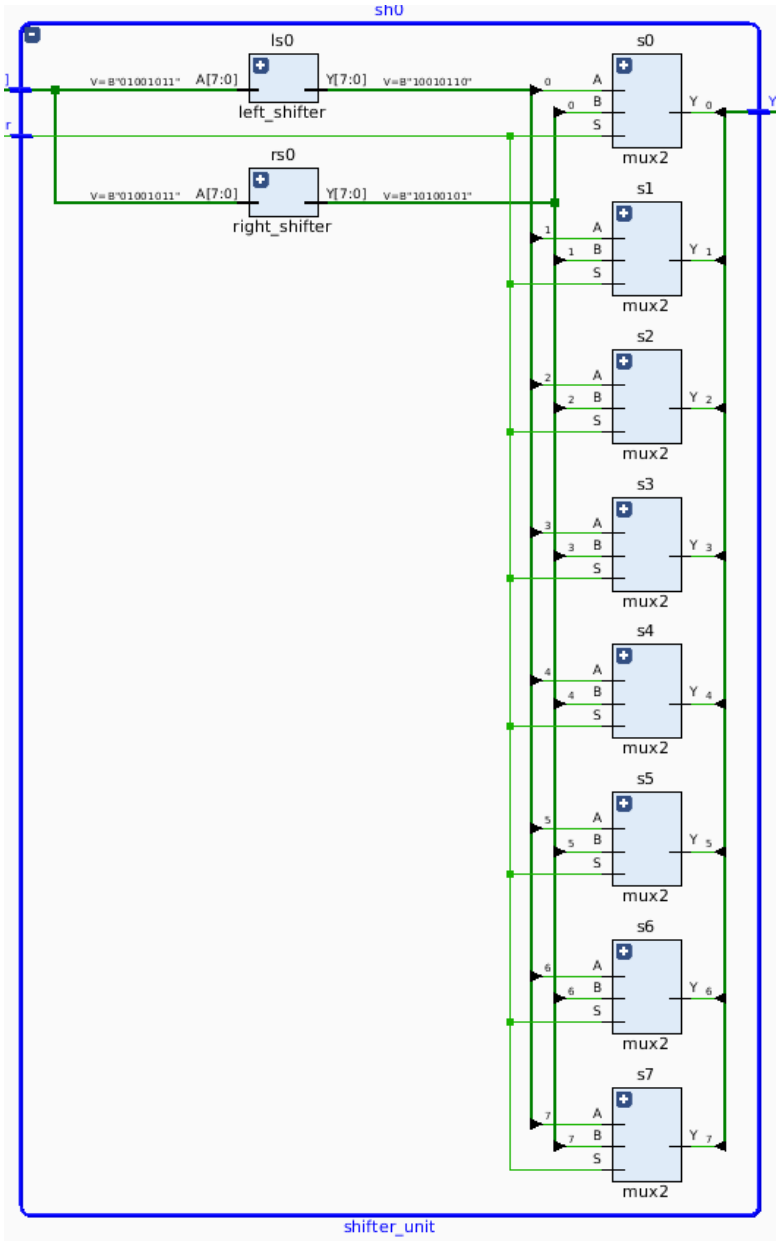

Figure 5: RTL schematic of the Shifter Unit.

2.1.5. Logical Unit

The Logical Unit performs basic logical operations such as AND, OR, and XOR using gate-level combinational circuits. It takes two 8-bit operands as input and produces the result based on control logic. Additionally, a NOT operation is performed to achieve bitwise negation of a single operand. Figure 6 illustrates the RTL schematic of the Logical Unit.

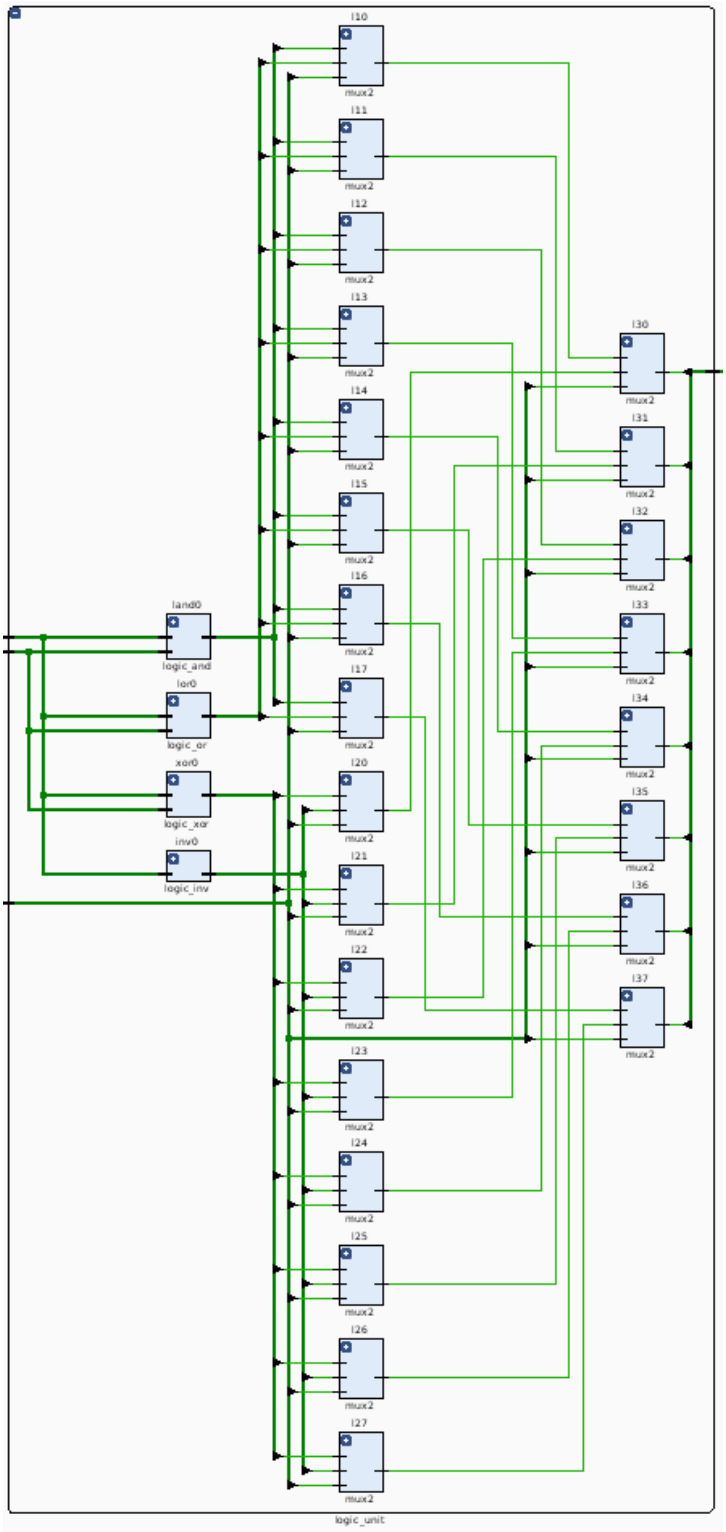

Figure 6: RTL schematic of the Logical Unit.

2.1.6. Control Unit

The Control Logic decodes a 4-bit opcode to select the operation performed by the ALU. Based on the opcode, various arithmetic and logical operations are executed. Figure 7 illustrates the schematic of the Control Logic.

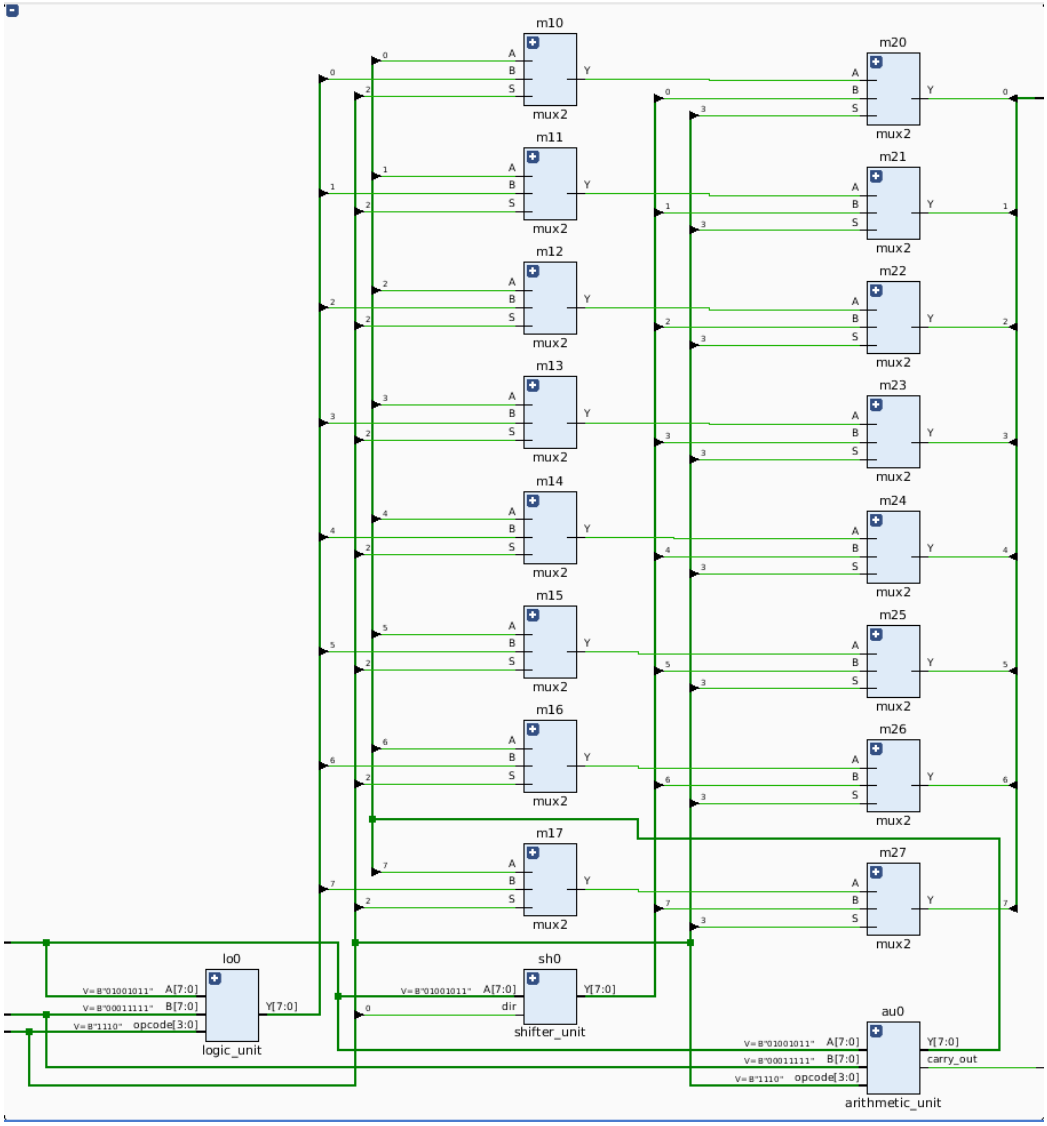

Figure 7: RTL schematic of the Control Logic.

2.2. Schematic

In this phase of the project, we take the RTL schematic generated from the previous steps and proceed to create a detailed schematic using CMOS technology. The design will be drawn using the xschem application, which allows for efficient creation of circuit schematics.

2.2.1. Arithmetic Unit

The Arithmetic Unit in this design is composed of two critical components: the Array Multiplier and the Ripple Carry Adder (RCA). These components work together to perform the essential arithmetic operations required for digital computation.

2.2.1.1. Array Multiplayer

The array multiplier is known for its regular structure and ease of implementation but can be slower compared to other more advanced multiplier designs. The schematic for the Array Multiplier is shown in Figure 8. The symbolic representation of the Array Multiplier can be found in Figure 9.

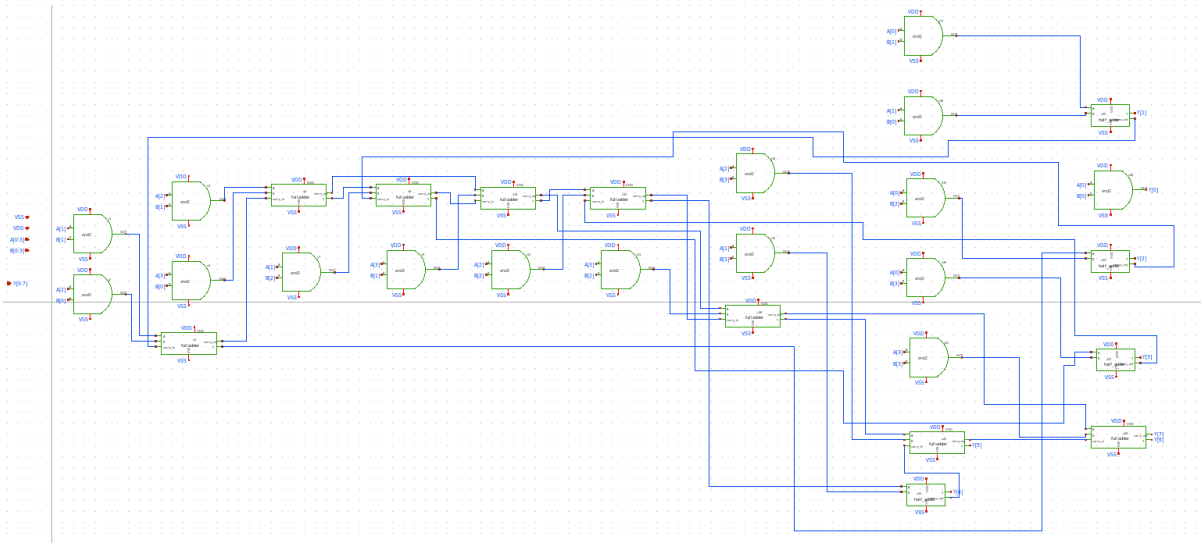

Figure 8: The Xschem schematic for the Array Multiplier.

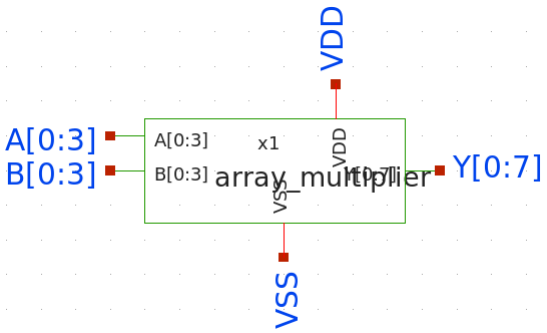

Figure 9: The Xschem symbol for the Array Multiplier

To verify the functionality of the multiplier, a simulation was conducted, and the results are presented in Figure 10.

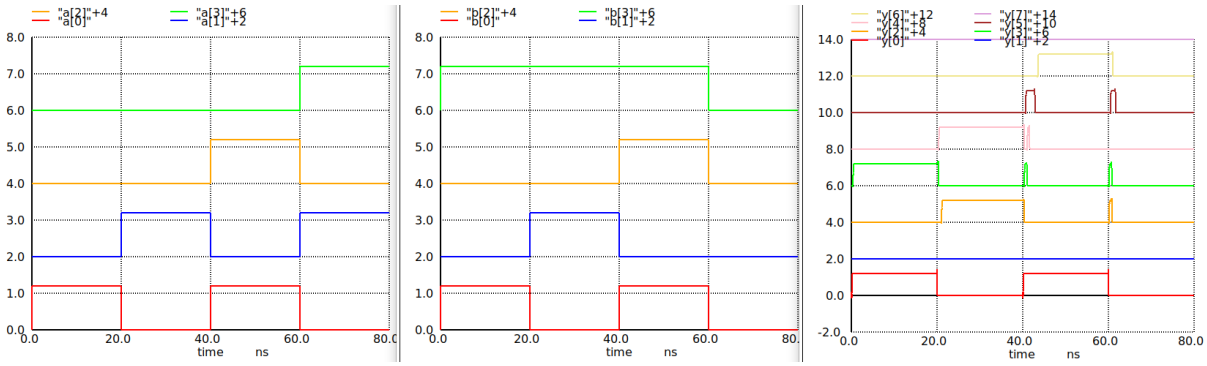

Figure 10: The simulation result for the Array Multiplier.

2.2.1.2. Ripple Carry Adder

The Xschem schematic for the Ripple Carry Adder can be found in Figure 11. This schematic illustrates the structure of the adder, showing how the full adders are connected in a chain to compute the sum and carry-out. The symbolic representation of the RCA can be found in Figure 12.

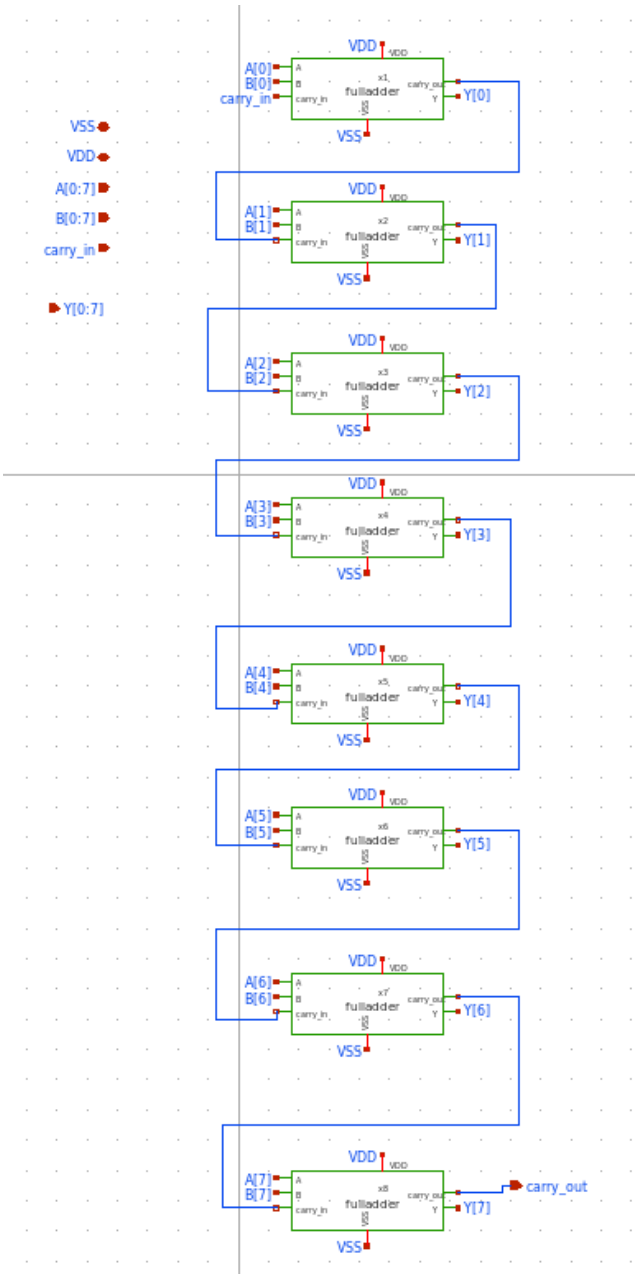

Figure 11: The Xschem schematic for the RCA.

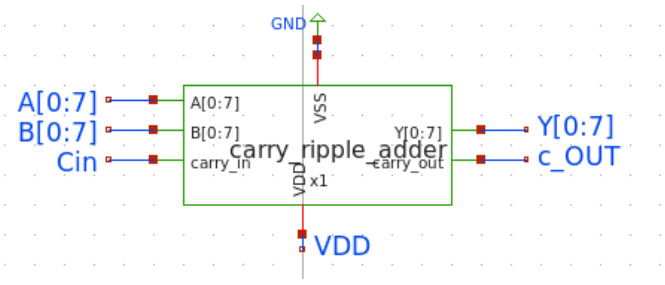

Figure 12: The Xschem symbol for the RCA

To verify the functionality of the Carry Ripple Adder (RCA), a simulation was conducted, and the results are presented in Figure 13.

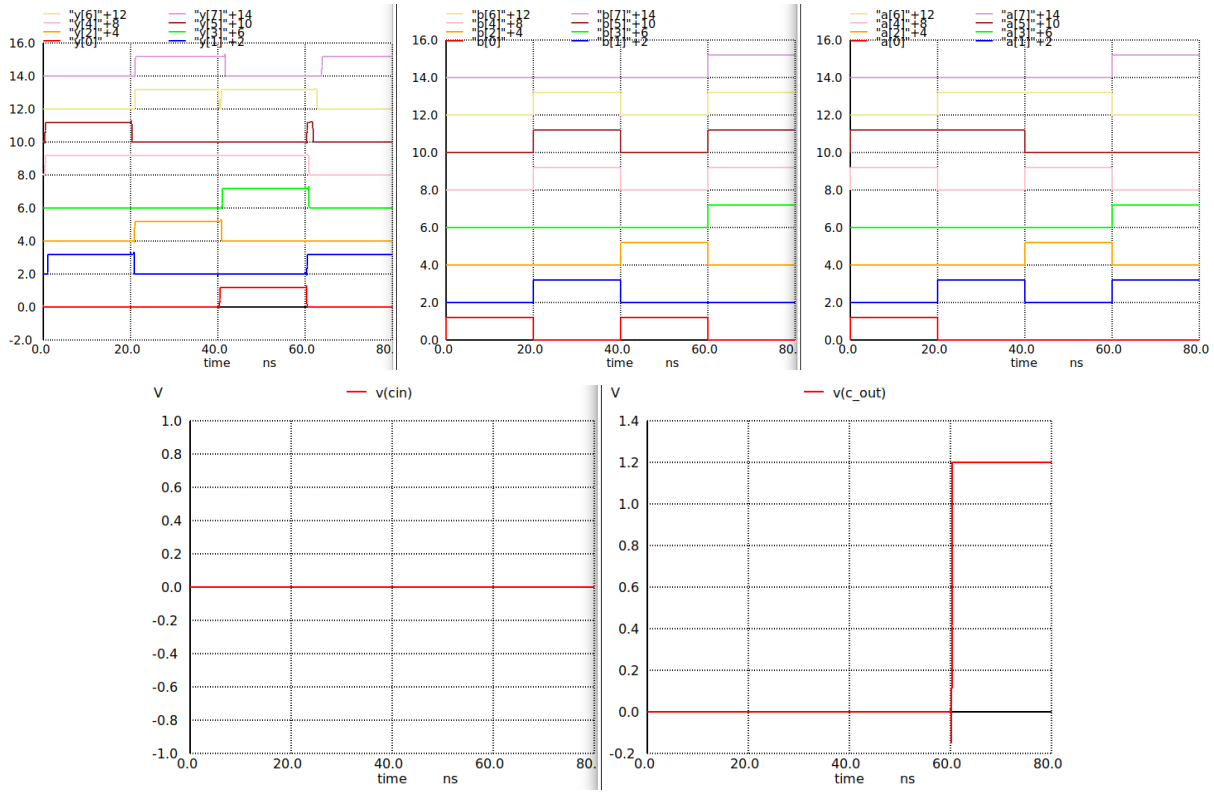

Figure 13: The simulation result for the RCA.

2.2.1.3. Arithmetic Unit Integration and Simulation

To verify the functionality of the Arithmetic Unit, a comprehensive simulation was conducted, and the results are presented in the following figures. Figure 14 shows the Xschem schematic of the Arithmetic Unit. This schematic illustrates how the various components, including the Adder, Multiplier, Subtractor, and Shifter Unit, are integrated to perform the arithmetic operations.

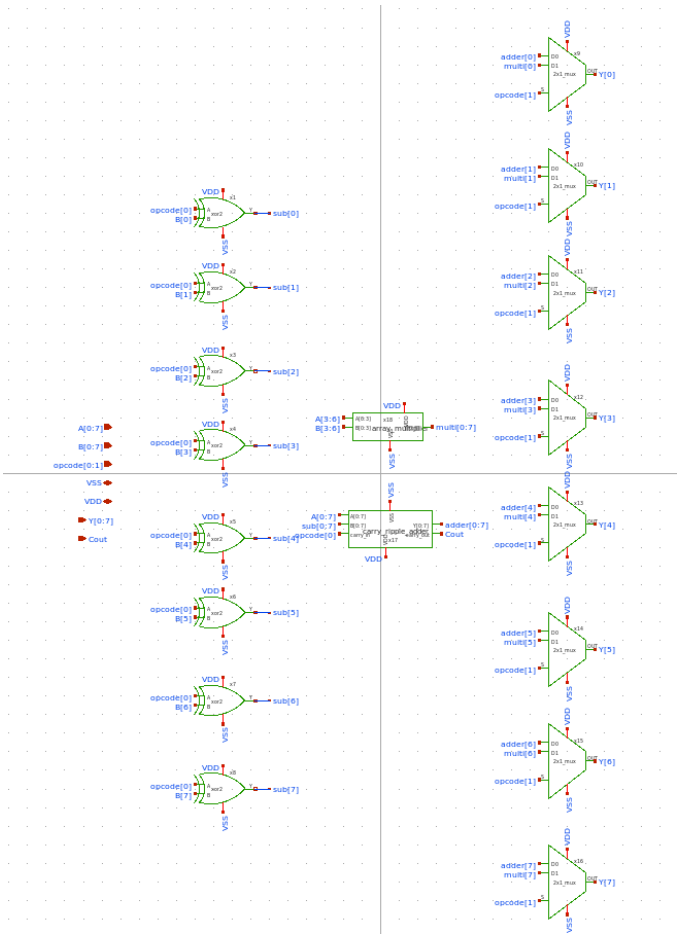

Figure 14: The Xschem schematic for the Arithmetic Unit

Table 1 provides the Opcode Mapping for the Arithmetic Unit, which is used by the control logic to determine the specific operation to be performed based on the 4-bit opcode.

Table 1: Opcode Mapping for the Arithmetic Unit

Operation	**Code**
Adder	00
Subtracter	01
Multiplayer	10
Multiplayer	11

Figure 15 shows the symbolic representation of the Arithmetic Unit in Xschem.

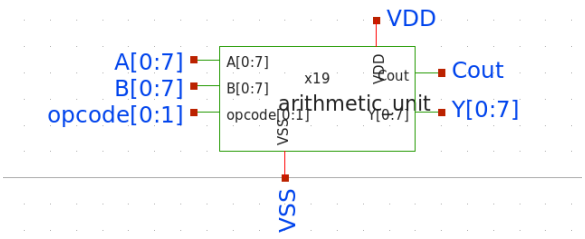

Figure 15: The Xschem symbol for the Arithmetic Unit.

Finally, Figure 16 presents the simulation results for the Arithmetic Unit. The results confirm that the unit correctly performs addition, subtraction, multiplication, and shifting operations, providing the expected outputs for various inputs.

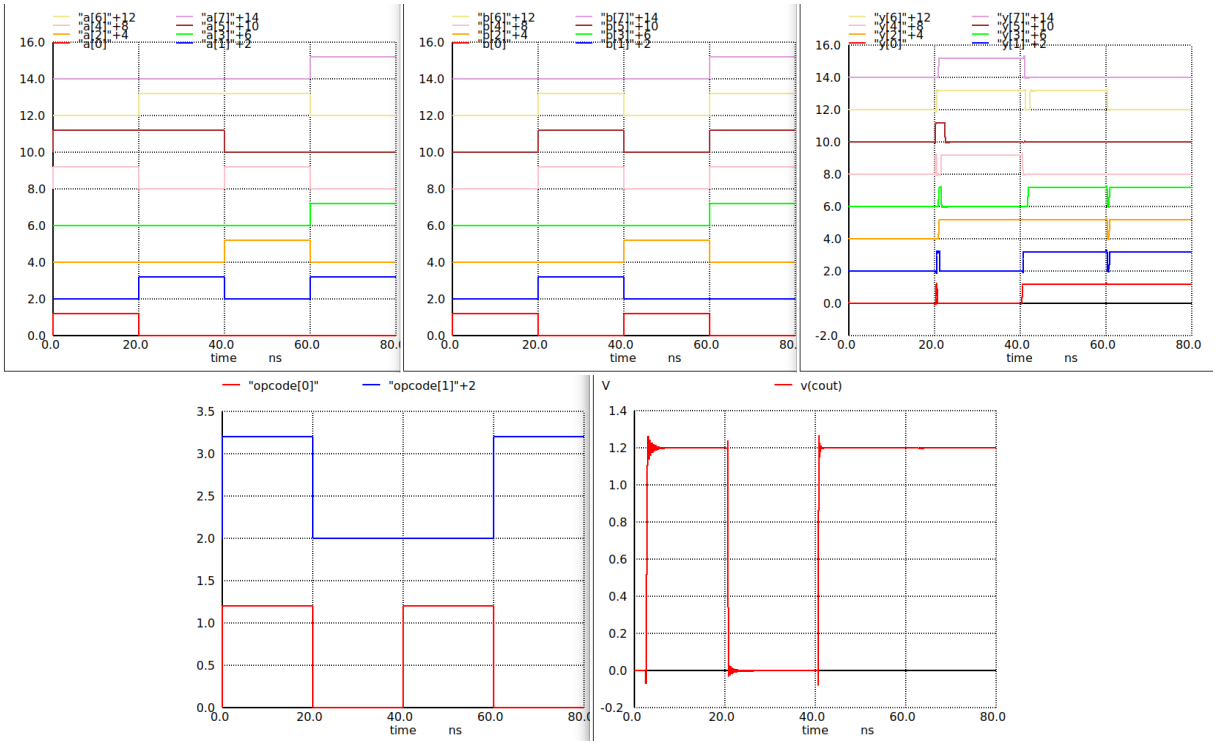

Figure 16: The simulation result for the Arithmetic Unit.

2.2.2. Logical Unit

Logical Unit is designed to perform basic logical operations, including AND, OR, XOR, and NOT. The unit operates on two 8-bit operands, A[7:0] and B[7:0], and provides the corresponding output based on the selected operation.

2.2.2.1. 8-bit AND Gate

To implement the AND operation in the Logical Unit, an AND gate is used. This gate computes the bitwise AND of the two input operands, A[7:0] and B[7:0], producing an 8-bit output where each bit is the logical AND of the corresponding bits from the operands. The xschem schematic for the AND gate is shown in Figure 17, illustrating the gate-level implementation of the AND operation. The inputs A[7:0] and B[7:0] are connected to the AND gate. Figure 18 shows the xschem symbol for the AND gate, simplifying the design and representing the AND operation as a single unit, which can be used in higher-level designs. The simulation results presented in Figure 19 confirm that the AND gate correctly performs the bitwise AND operation, producing the expected output for the given inputs.

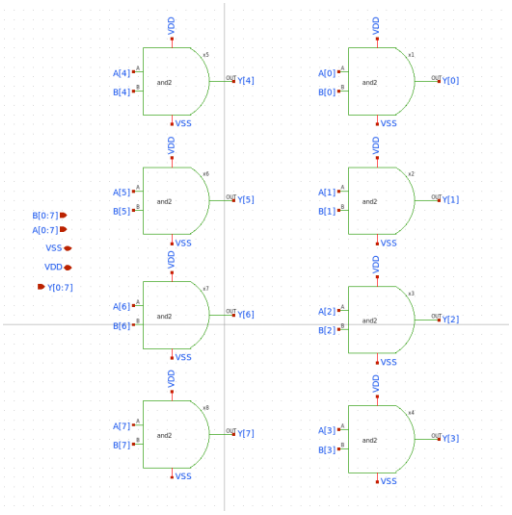

Figure 17: The Xschem schematic for the AND Gate

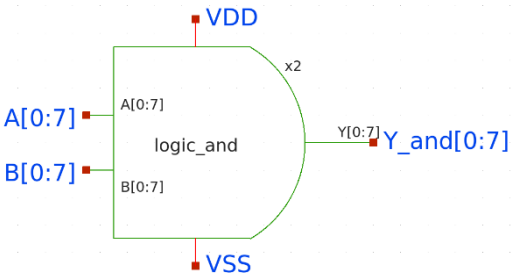

Figure 18: The Xschem symbol for the AND Gate

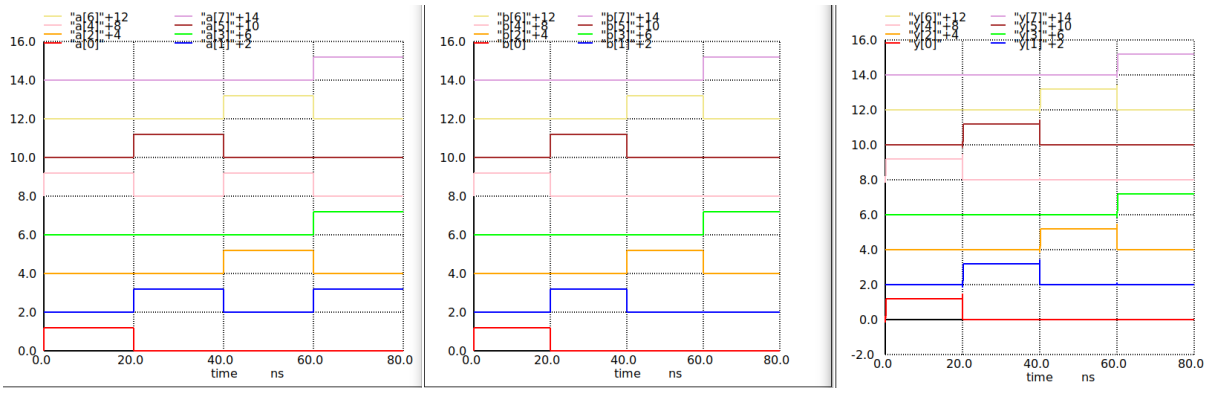

Figure 19: The simulation result for the AND Gate.

2.2.2.2. 8-bit OR Gate

To implement the OR operation in the Logical Unit, an OR gate is used. This gate computes the bitwise OR of the two input operands, A[7:0] and B[7:0], resulting in an 8-bit output where each bit is the logical OR of the corresponding bits from the operands. The xschem schematic for the OR gate is shown in Figure 20, representing the gate-level implementation of the OR operation. The inputs A[7:0] and B[7:0] are connected to the OR gate. Figure 21 shows the xschem symbol for the OR gate, which simplifies the design and represents the OR operation as a single unit. Figure 22 presents the simulation results for the

OR gate, confirming that it correctly performs the bitwise OR operation and produces the expected output.

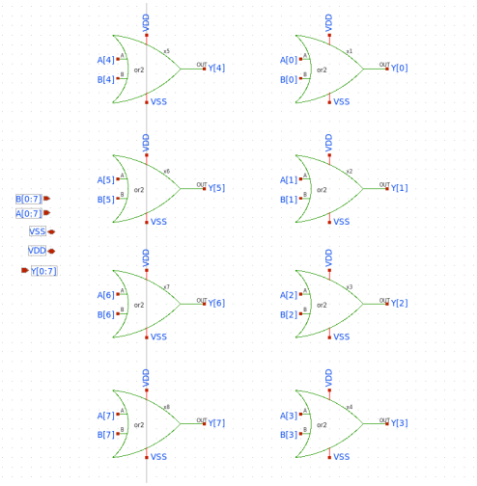

Figure 20: The Xschem schematic for the OR Gate

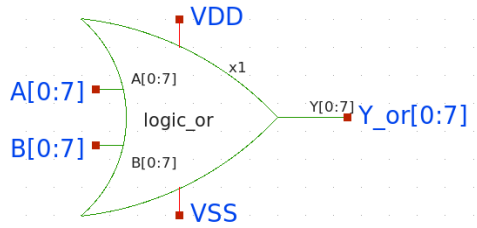

Figure 21: The Xschem symbol for the OR Gate

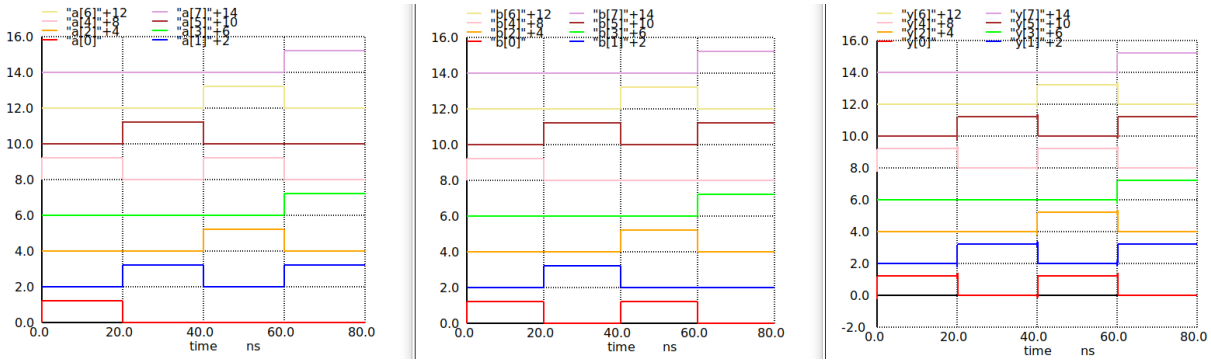

Figure 22: The simulation result for the OR Gate

2.2.2.3. 8-bit XOR Gate

To implement the XOR operation in the Logical Unit, an XOR gate is used. This gate computes the bitwise exclusive OR of the two input operands, A[7:0] and B[7:0], producing an 8-bit output where each bit is the logical XOR of the corresponding bits. The xschem schematic for the XOR gate is shown in Figure 23, illustrating the gate-level implementation of the XOR operation. The inputs A[7:0] and B[7:0] are connected to the XOR gate. Figure 24 shows the xschem symbol for the XOR gate, simplifying the design and representing the XOR operation as a single unit. Figure 25 presents the

simulation results for the XOR gate, confirming that the gate correctly performs the bitwise XOR operation and produces the expected outputs.

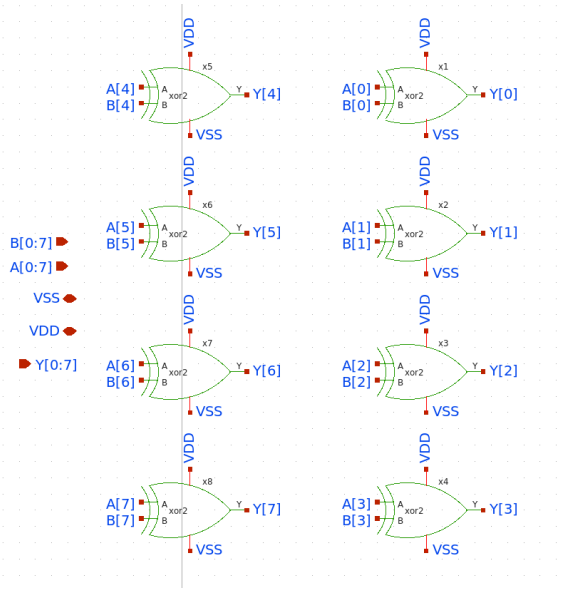

Figure 23: The Xschem schematic for the XOR Gate

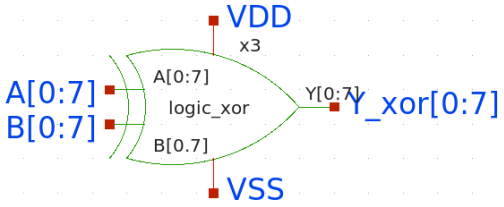

Figure 24: The Xschem symbol for the XOR Gate

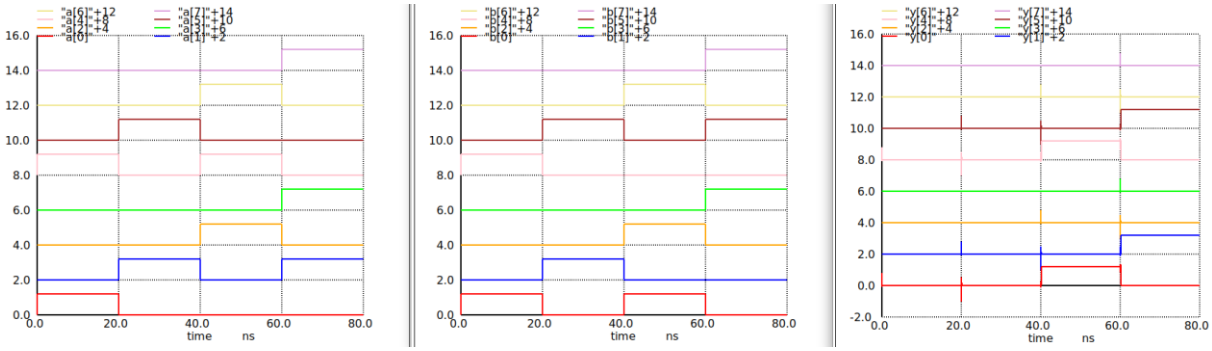

Figure 25: The simulation result for the XOR Gate

2.2.2.4. 8-bit NOT Gate

To implement the NOT operation in the Logical Unit, a NOT gate is used. This gate negates each bit of the operand A[7:0], producing the bitwise complement. The operand A[7:0] is the input, and the gate produces an 8-bit output NotA[7:0]. The Xschem schematic for the NOT gate is shown in Figure 26, representing the gate-level implementation of the NOT operation. The operand A[7:0] is connected to the NOT gate. Figure 27 shows the Xschem symbol for the NOT gate, simplifying the design and representing the NOT operation as a single unit. Figure 28 presents the simulation results for the NOT

gate, confirming that the gate correctly performs the bitwise negation and produces the expected bitwise complement of A[7:0].

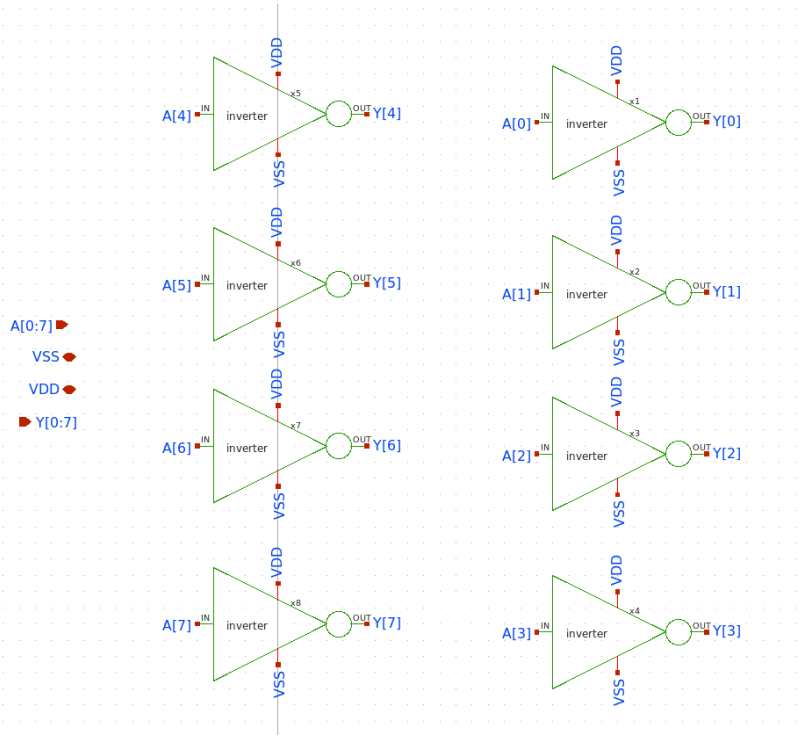

Figure 26: The Xschem schematic for the NOT Gate

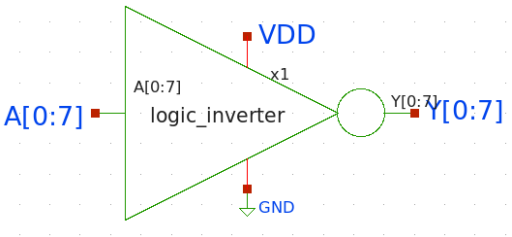

Figure 27: The Xschem symbol for the NOT Gate

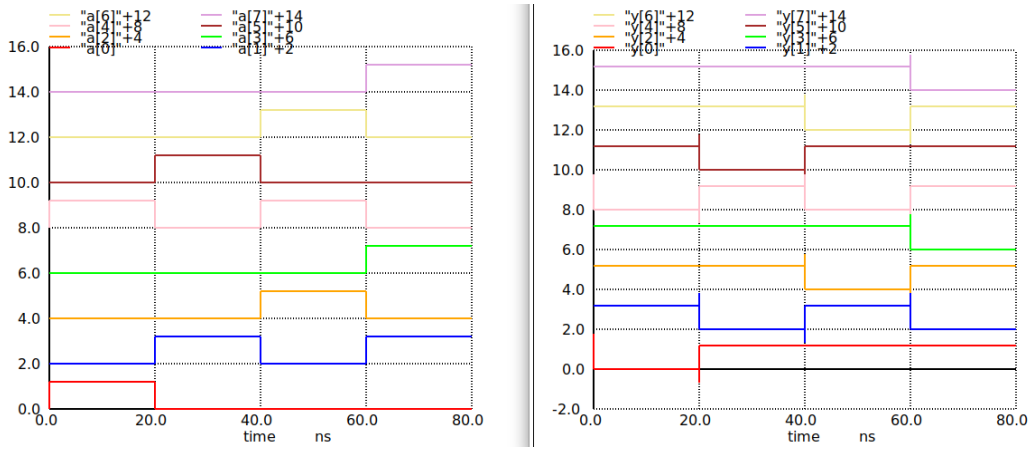

Figure 28: The simulation result for the NOT Gate

2.2.2.5. Logic Unit Integration and Simulation

The Logic Unit performs various logical operations such as AND, OR, XOR, and NOT on two 8-bit operands. These operations are implemented using standard logic gates. The xschem schematic for the Logic Unit is shown in Figure 29, which illustrates how the logical operations are performed in the unit. The inputs A[7:0] and B[7:0] are fed into the gates, and the appropriate output is generated based on the control logic. The opcode table in Table 2 maps the operation code to the corresponding logical operation (AND, OR, XOR, NOT). The xschem symbol for the Logic Unit is shown in Figure 30, which simplifies the design by representing the logic unit as a single unit in higher-level circuits. The simulation results are shown in Figure 31, demonstrating the correct functionality of the Logic Unit for each logical operation.

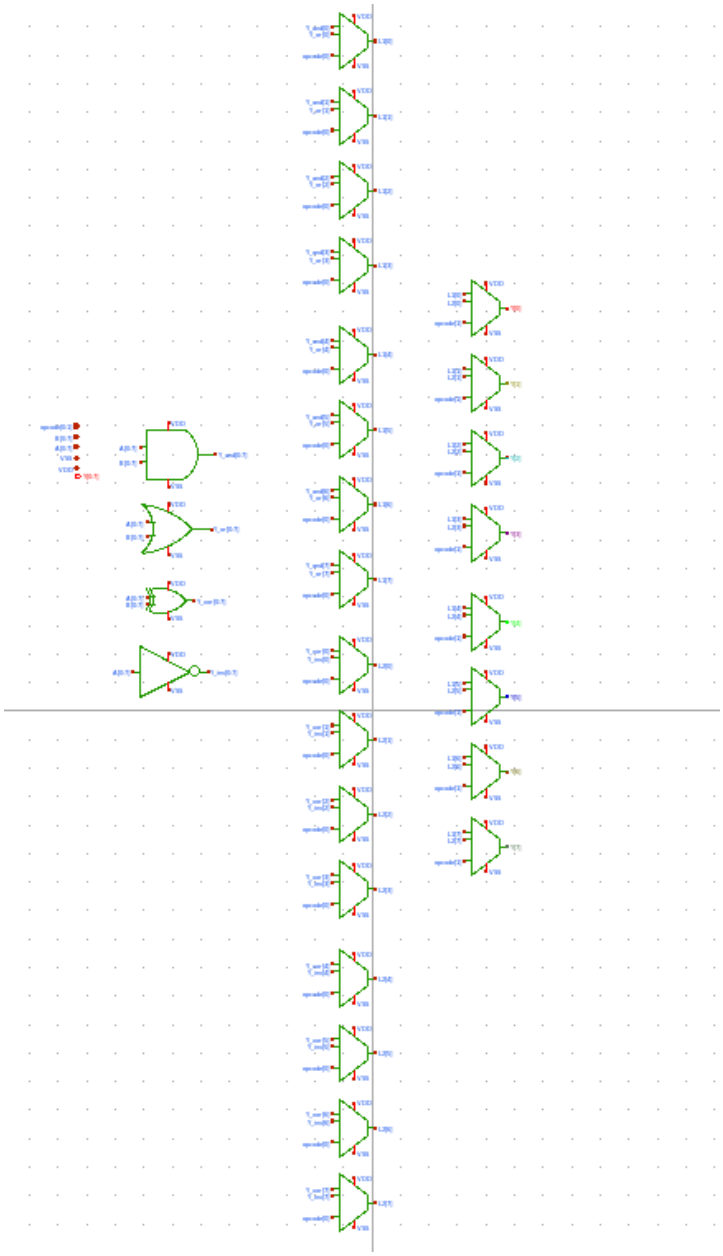

Figure 29: The Xschem schematic for the Logic Unit

Table 2: Opcode Mapping for the Logic Unit

Operation	Code
AND	00
OR	01
XOR	10
Inverter	11

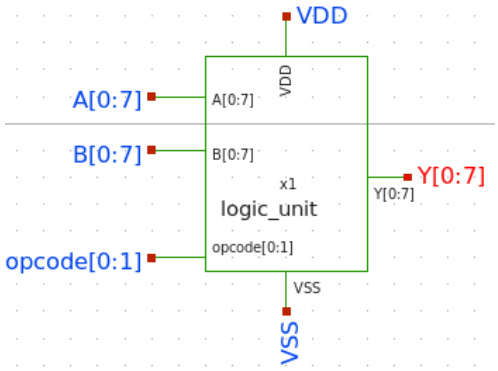

Figure 30: The Xschem symbol for the LogicUnit

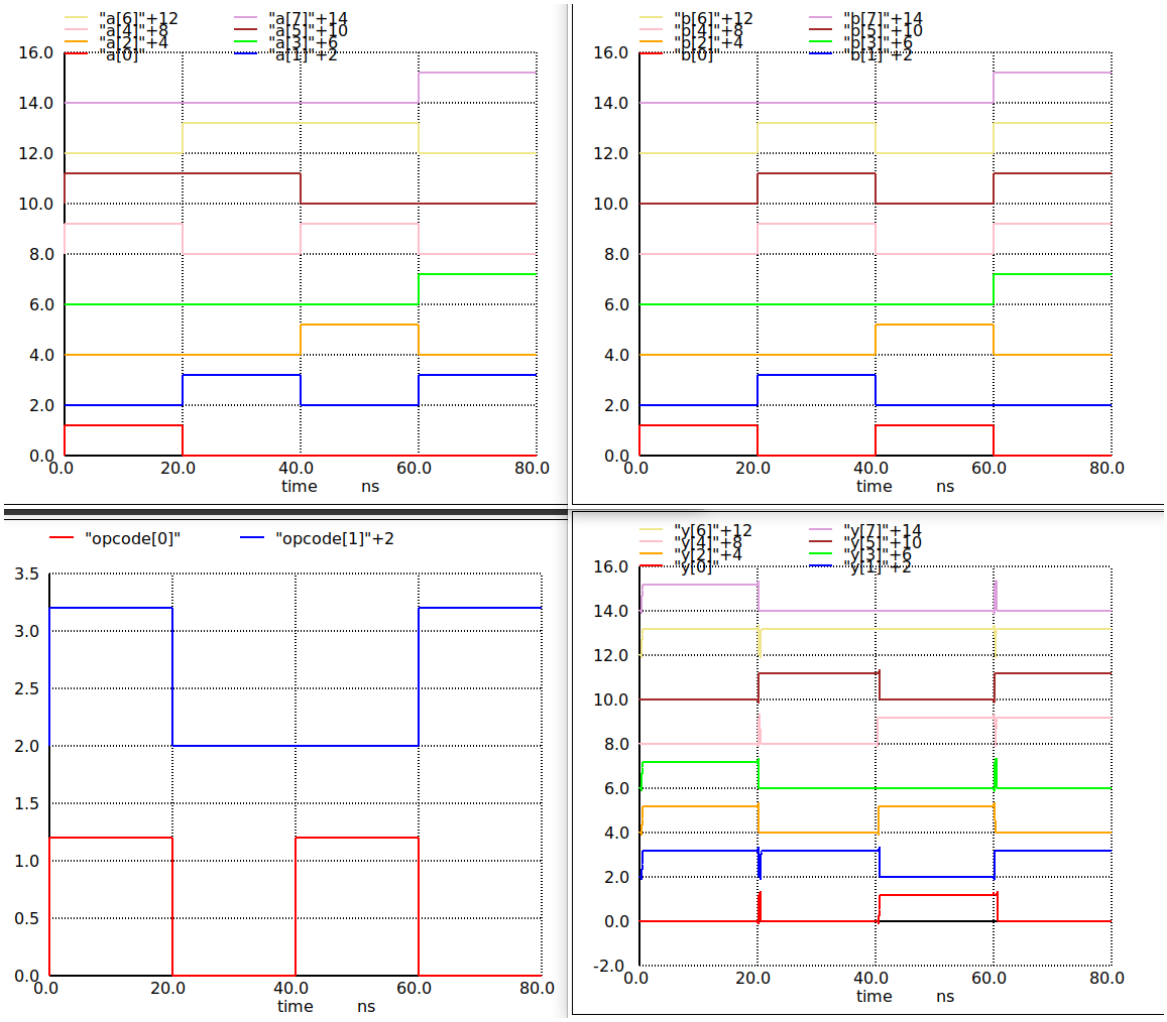

Figure 31: The simulation result for the Logic Unit.

2.2.3. Shifter Unit

The shifter unit is the unit that shifts the 8-bit data coming to its input to the right or left. The direction of the unit is manipulated according to the 'dir' input of this unit. When the 'dir' bit is 0, the input is shifted to the left, and when it is 1, the input is shifted to the right. The schematic drawing of the shifter unit is shown in Figure 32. The schematic drawn was then given as a symbol as in Figure 33. All situations are simulated as in Figure 34.

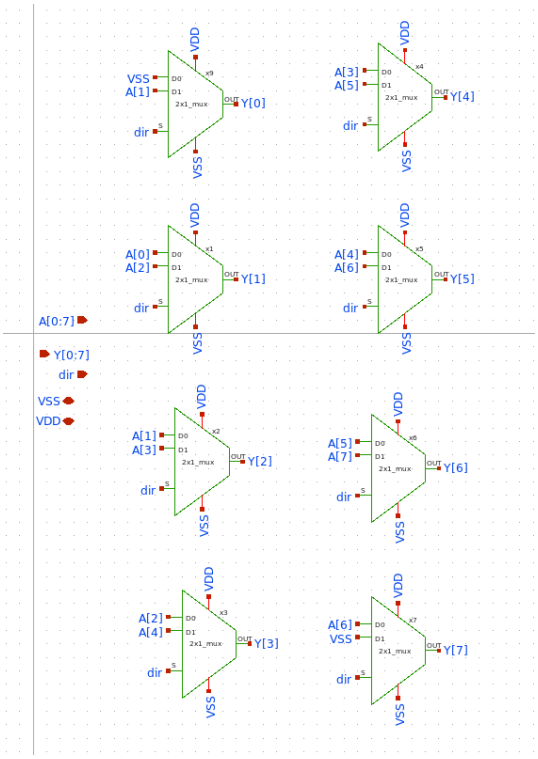

Figure 32: The Xschem schematic for the Shifter Unit

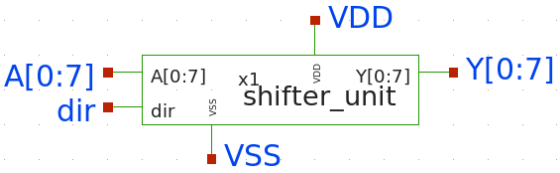

Figure 33: The symbol schematic for the Shifter Unit.

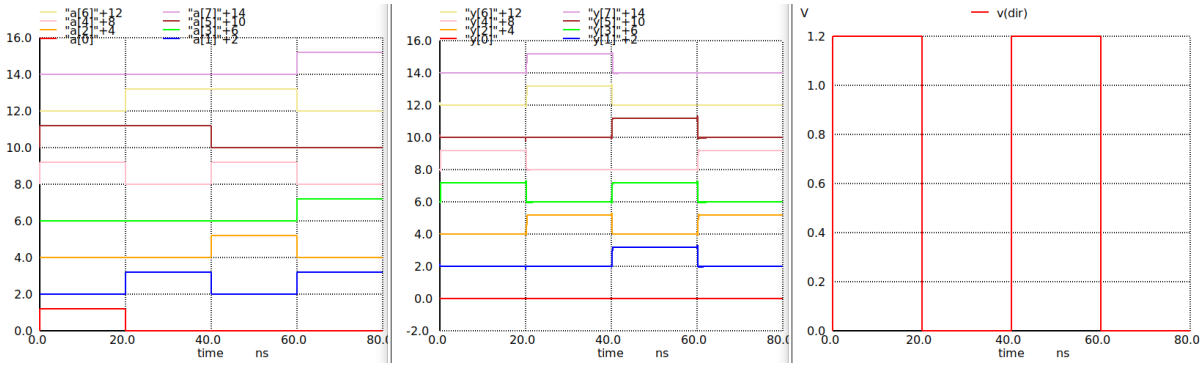

Figure 34: The simulation result for the Shifter Unit.

2.2.4. Control Unit

Control Unit is the unit that contains shifter, logic unit and arithmetic units. The units in the control unit manipulate the 8-bit inputs coming to the input with the 4-bit opcode bit. All cases are given as seen in Table 3, the functions of the opcode codes. In Figure 35, there is a combined state of the control unit and other designed units. The designed control unit is symbolized as in Figure 36. In Figure 37, output is obtained at certain opcode values. When Figure 37 is examined, the simulation works correctly.

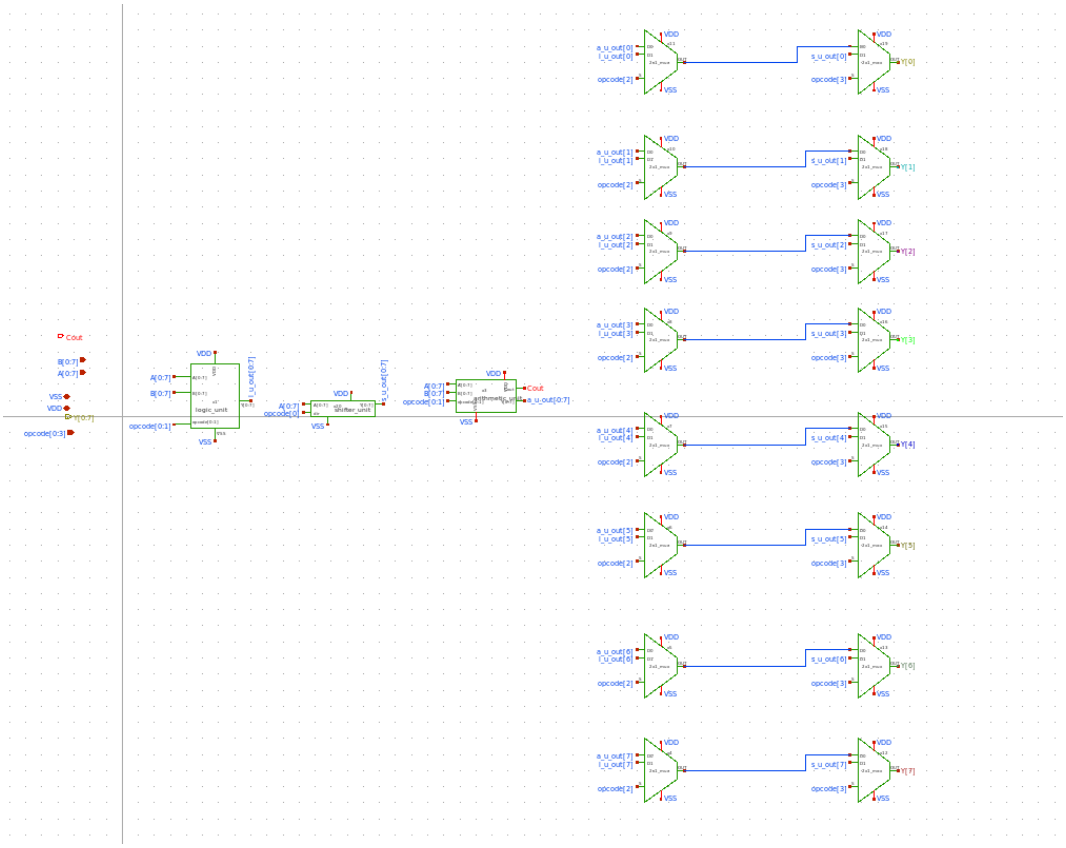

Figure 35: The Xschem Schematic for the Control Unit

Table 3: Opcode Mapping for the Control Unit

Operation	**Code**
Adder	0000
Subtracter	0001
Multiplayer	0010
Multiplayer	0011
AND	0100
OR	0101
XOR	0110
Inverter	0111
Right Shift	1xx1
Left Shift	1xx0

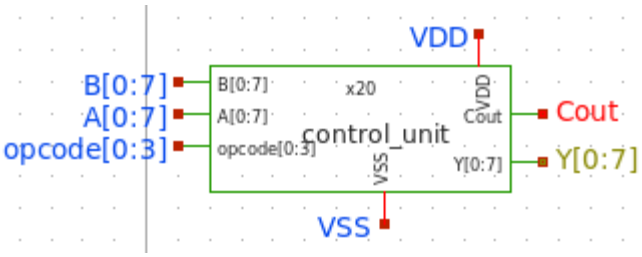

Figure 36: The Symbol Schematic for the Control Unit

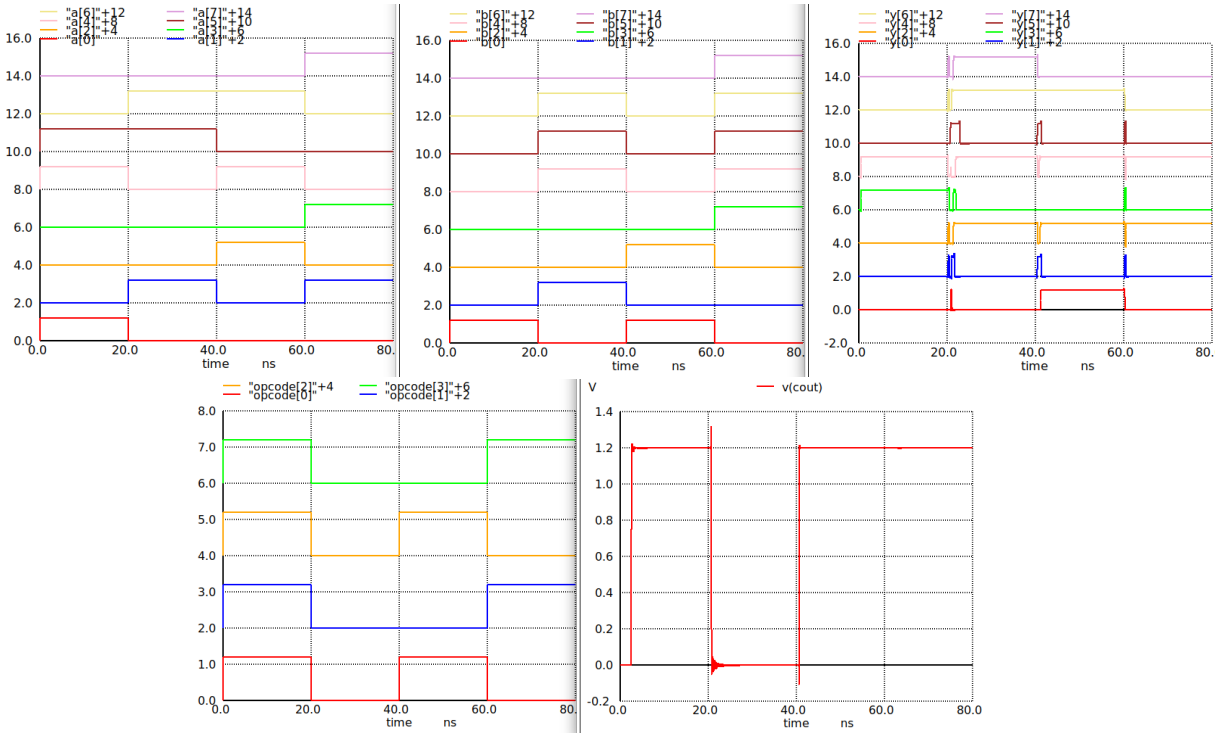

Figure 37: The Simulation Result for the Shifter Unit

2.2.5. Arithmetic Logic Unit

There are control unit and flag circuits designed in the ALU design. An 8-bit NOR gate is used for the Zero Flag. The 1 bit coming from here tells us that all bits of the ALU output are zero. The carry output of the control unit is processed in the circuit as the Carry Flag. The last bit of the output is taken as the Sign Flag after passing through the inverter. XOR and SNOR gates are used for the Overflow Flag and are given as in Figure 38. In this way, the ALU gives a 4-bit flag output. The control unit is manipulated with the opcode and the codes given in Table 3. While the symbol of the ALU is given in Figure 39, the simulation is performed with certain opcode values in Figure 40.

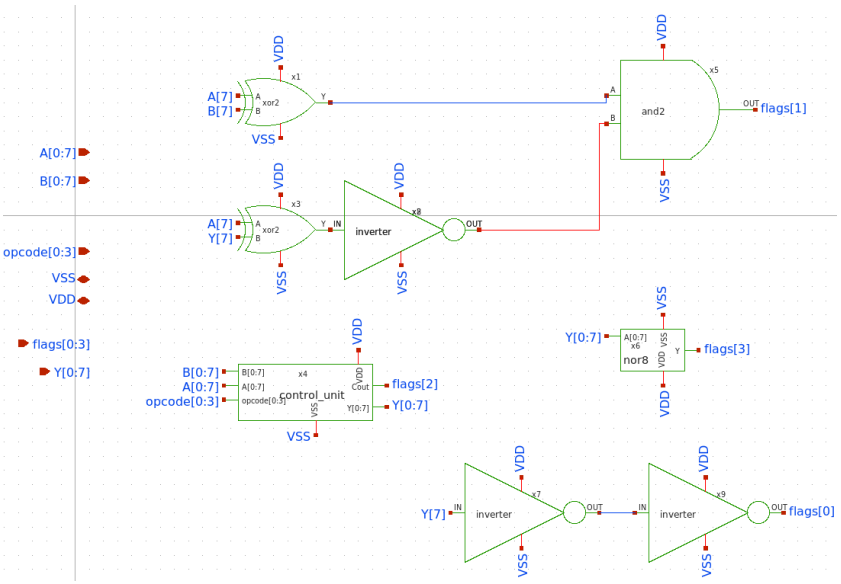

Figure 38: The Xschem Schematic for ALU.

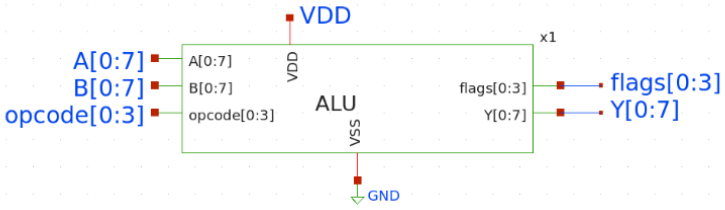

Figure 39: The Symbol Schematic for ALU.

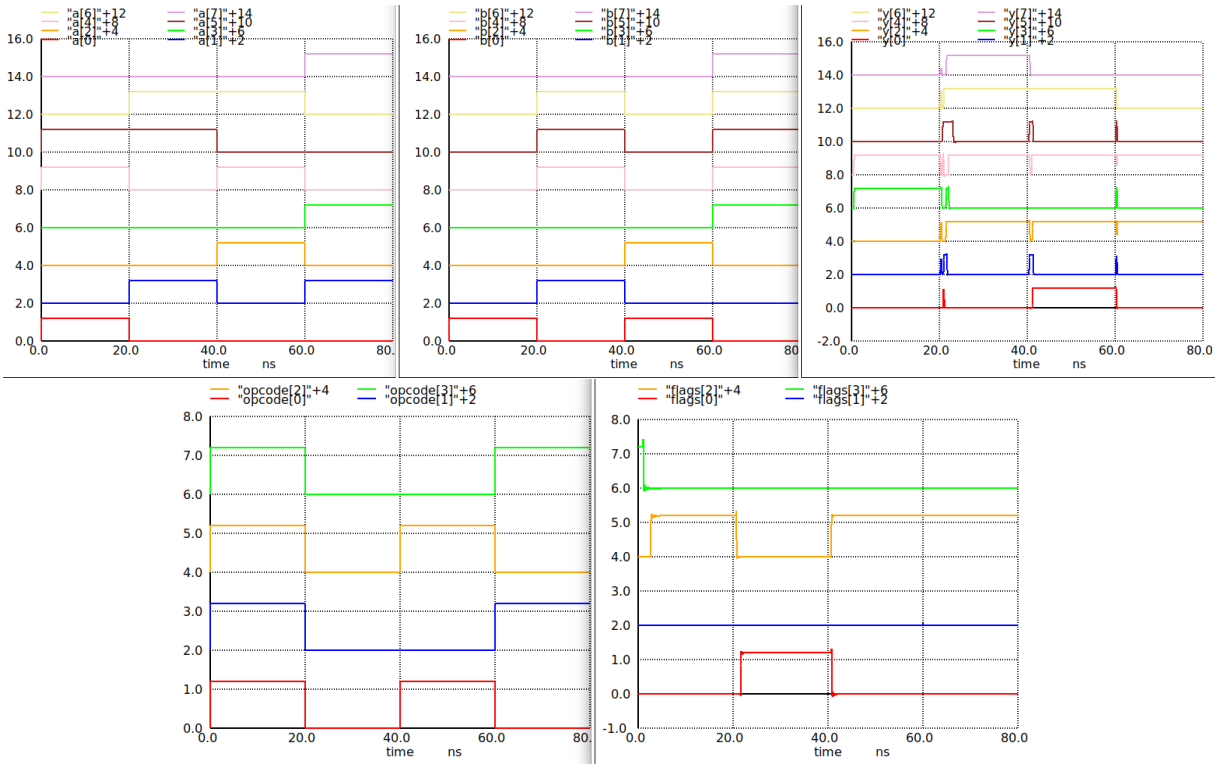

Figure 40: The Simulation Result for the ALU.

2.3. Layout

Physical layouts are generated that ensure adherence to Design Rules Check (DRC) and verifying with Layout and Schematic (LVS) checks.

2.3.1. Arithmetic Unit

One of the sub-blocks that form the 8-bit ALU is the arithmetic unit. The arithmetic unit consists of adder, subtractor and multiplier.

2.3.1.1. Adder

Figure 41 shows the layout of the ripple carry adder, which is chosen for its simplicity in design.

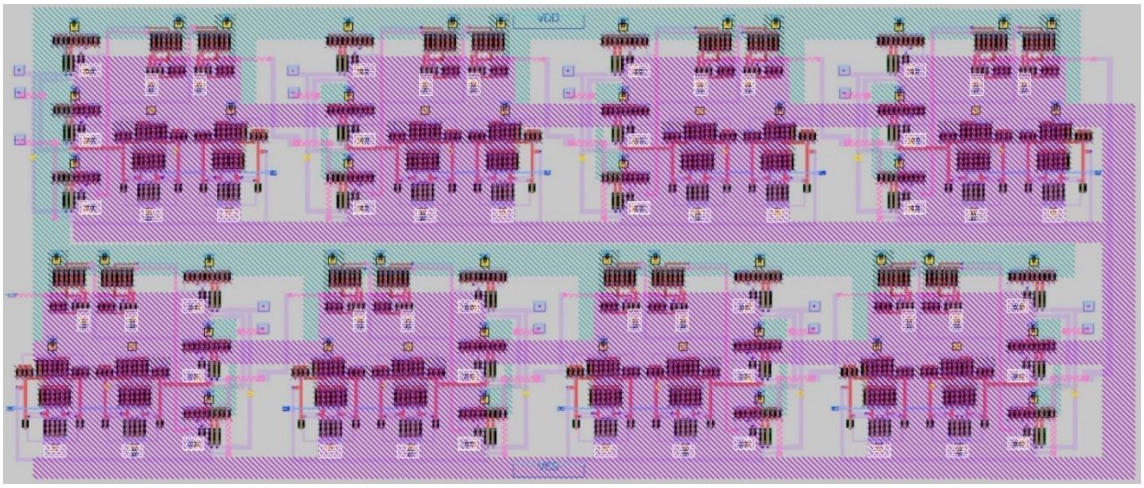

Figure 41: Layout of ripple carry adder (RCA).

The 8-bit RCA computes the sum of two 8-bit operands, with inputs labelled as A[7:0], B[7:0], and Cin, and outputs as Y[7:0] and Cout. The implementation consists of a chain of full adders connected serially to perform the addition operation.

Post-layout simulations are done to evaluate the functionality, showing that the adder operation is correctly performed across all 8 bits. The simulation results, demonstrating the correct operation, can be seen in Figure 42.

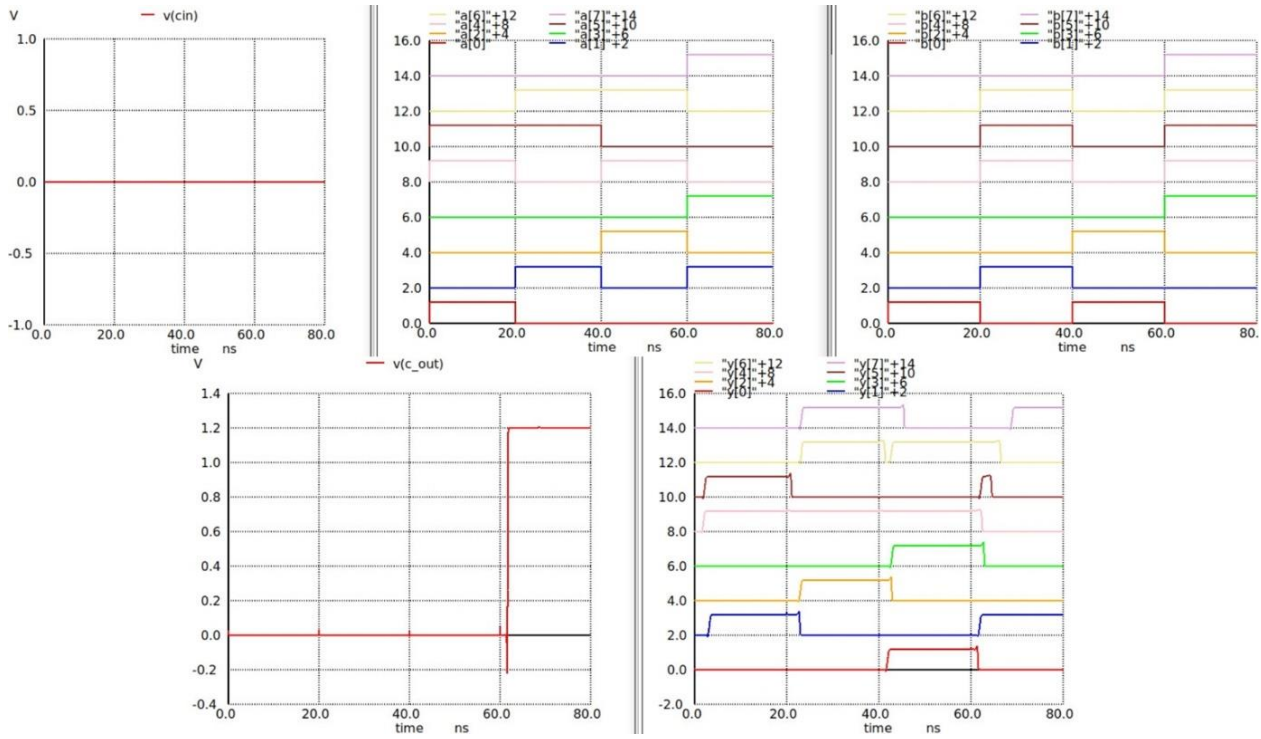

Figure 42: Post--layout simulation waveforms of ripple carry adder.

2.3.1.2. Subtractor

The subtractor layout is in the arithmetic unit integration layout. Post-layout simulations are done to evaluate the functionality, showing that the subtractor operation is correctly performed across all 8 bits. It can be seen in Figure 43.

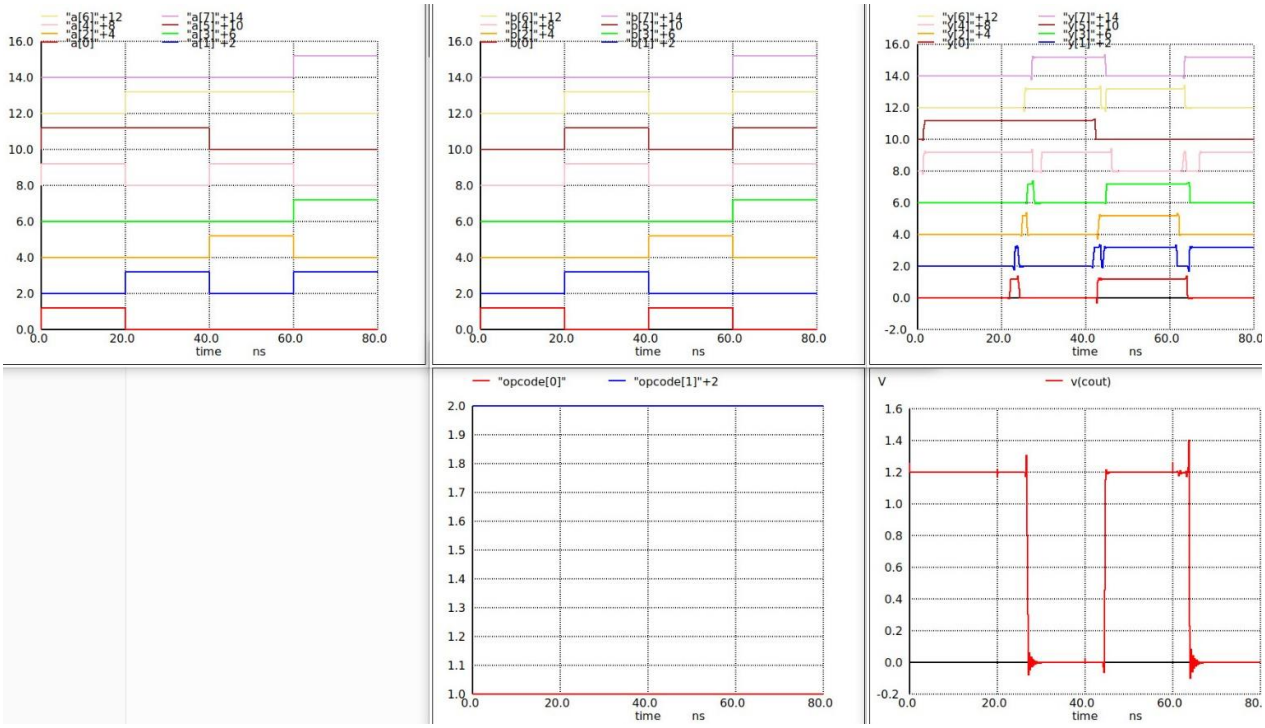

Figure 43: Post-layout simulation waveforms of subtractor.

2.3.1.3. Multiplier

Figure 44 shows the layout of the array multiplier.

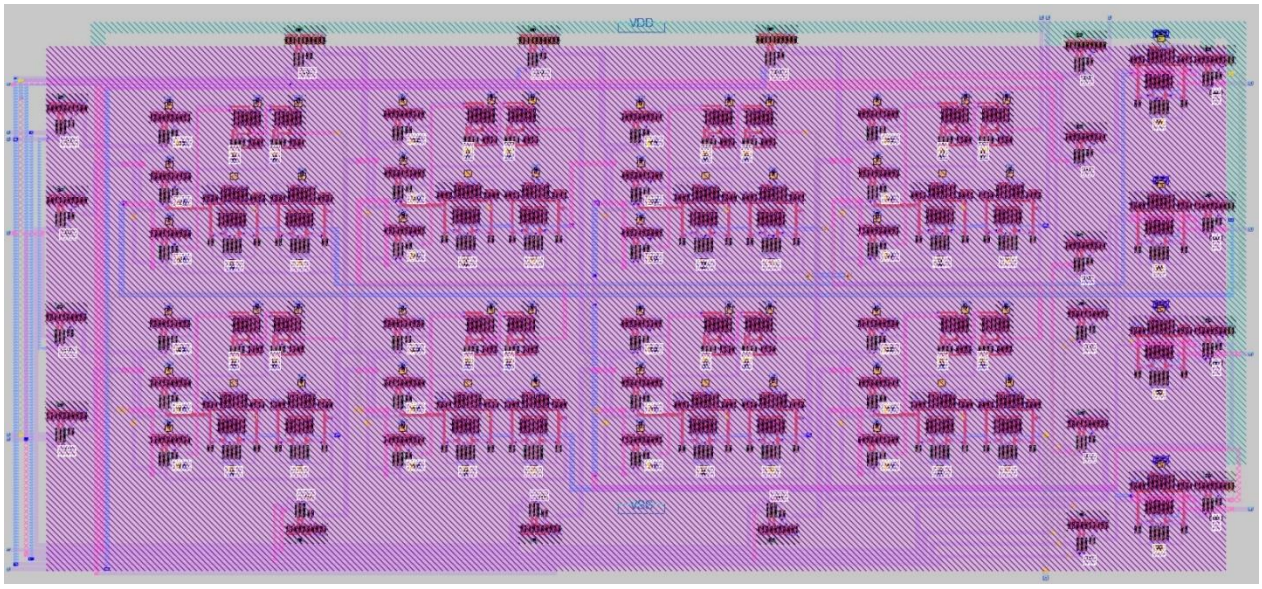

Figure 44: Layout of array multiplier.

Post-layout simulations are done to evaluate the functionality, showing that the multiplier operation is correctly performed across all 8 bits. It can be seen in Figure 45.

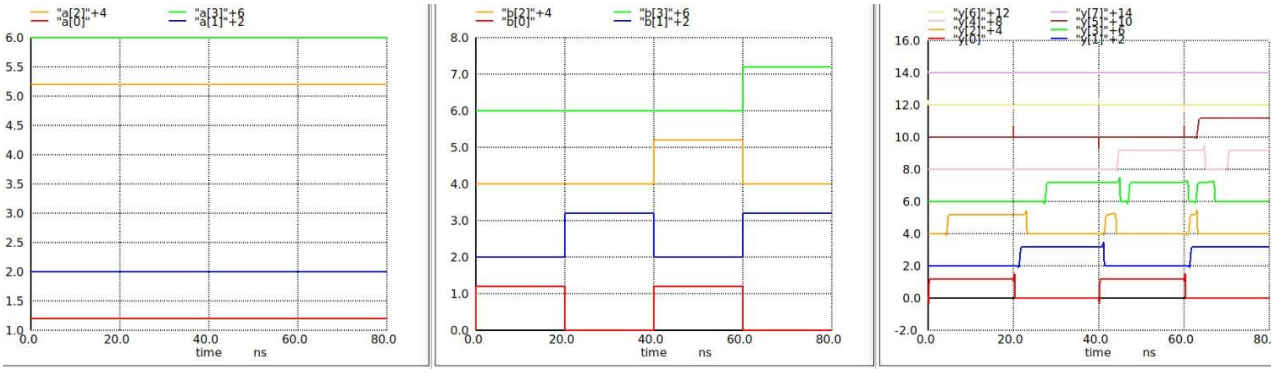

Figure 45: Post-layout simulation waveforms of array multiplier.

2.3.1.4. Arithmetic Unit Integration

The arithmetic unit is generated by integrating the adder, subtractor, and multiplier sub-blocks into a single layout. Layout is in Figure 46. The design is optimized for area and functionality, ensuring correct interaction between the sub-blocks.

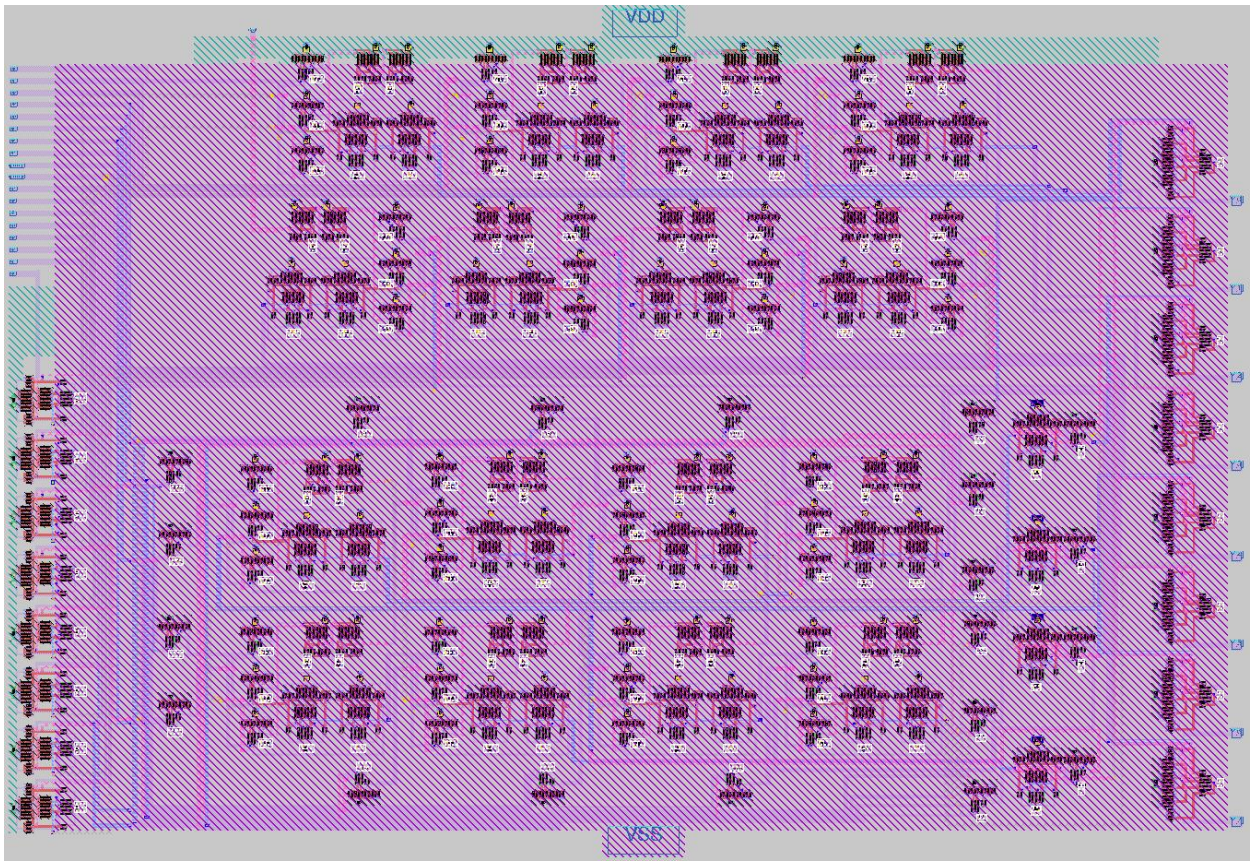

Figure 46: Layout of arithmetic unit.

Post-layout simulations are done to evaluate the functionality, showing that the arithmetic unit operation is correctly performed across all 8 bits. It can be seen in Figure 47.

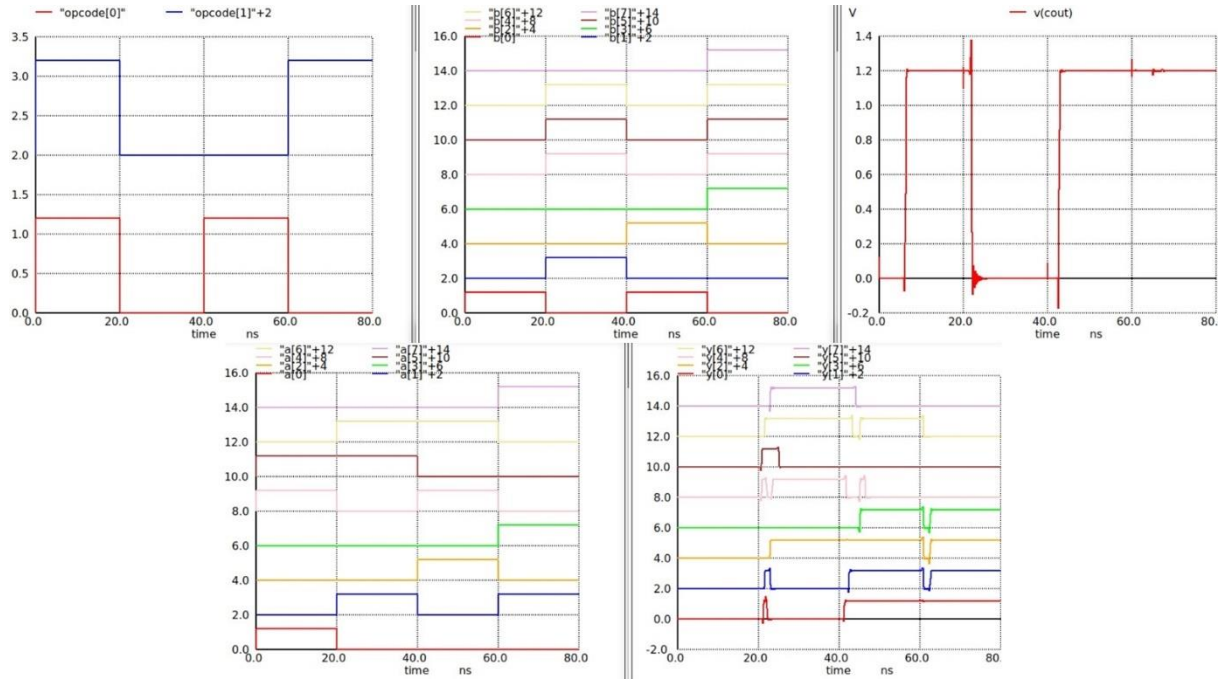

Figure 47: Post- layout simulation waveforms of arithmetic unit.

2.3.2. Logical Unit

One of the sub-blocks that form the 8-bit ALU is the logical unit. This logical unit performs operations such as 8-bit AND, OR, XOR, and NOT.

2.3.2.1. 8-bit AND Gate

Figure 48 shows the layout of the 8-bit AND gate is designed with optimal area usage while adhering to Design Rules Check (DRC).

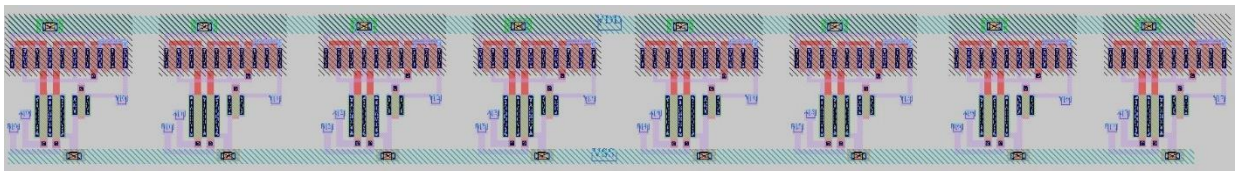

Figure 48: Layout of 8-bit AND gate.

Figure 49 shows the schematic is verified against the layout using Layout vs. Schematic (LVS) checks to confirm its correctness.

```
Circuit 1 contains 48 devices, Circuit 2 contains 48 devices.
Circuit 1 contains 42 nets,    Circuit 2 contains 42 nets.

Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

Figure 49: LVS for 8-bit AND gate.

Post-layout simulations are done to evaluate the functionality, showing that the AND operation is correctly performed across all 8 bits. It can be seen in Figure 50.

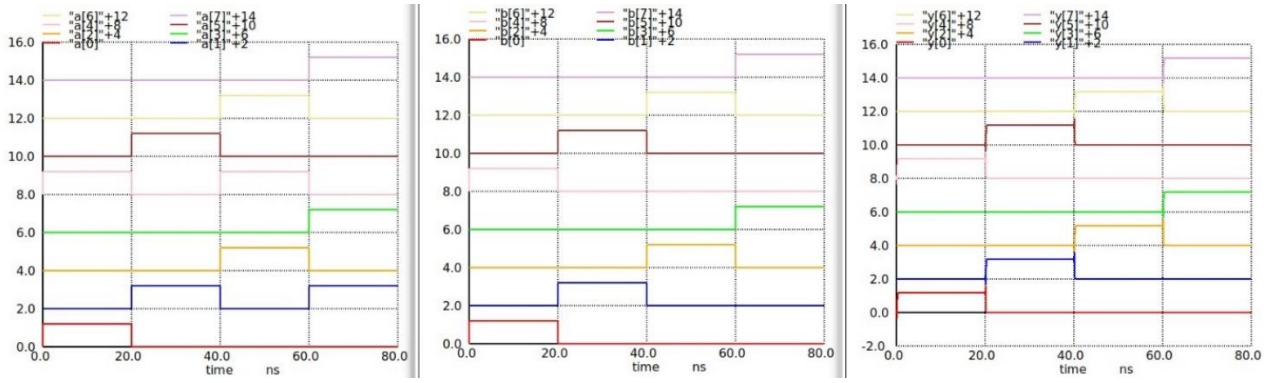

Figure 50: Post-layout simulation waveforms of 8-bit AND gate.

2.3.2.2. 8-bit OR Gate

The layout of the 8-bit OR gate is designed with optimal area usage while adhering to Design Rules Check (DRC). It can be seen in Figure 51.

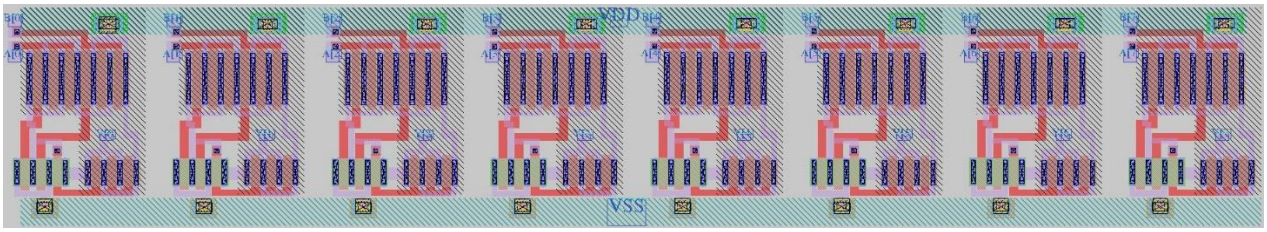

Figure 51: Layout of 8-bit OR gate.

Figure 52 shows the schematic is verified against the layout using Layout vs. Schematic (LVS) checks to confirm its correctness.

```
Circuit 1 contains 48 devices, Circuit 2 contains 48 devices.
Circuit 1 contains 42 nets,    Circuit 2 contains 42 nets.


Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

Figure 52: LVS for 8-bir OR gate.

Post-layout simulations are done to evaluate the functionality, showing that the OR operation is correctly performed across all 8 bits. It can be seen in Figure 53.

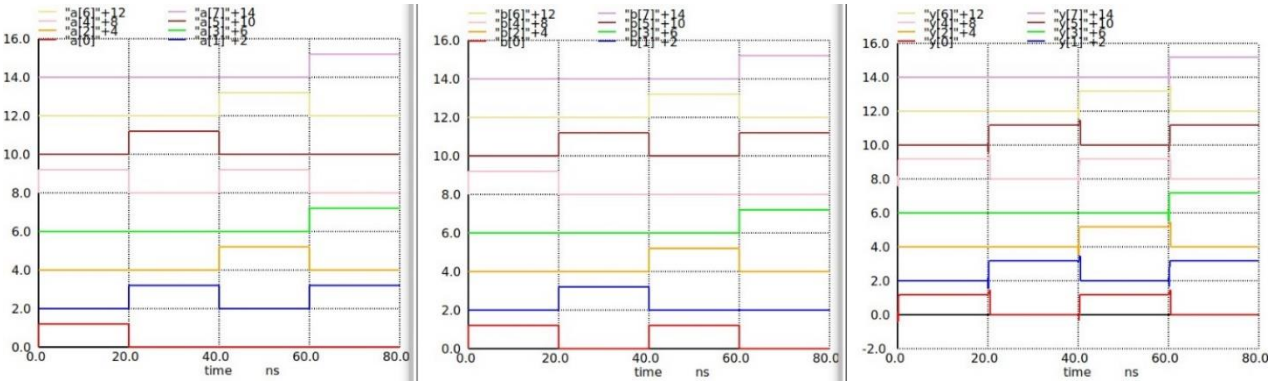

Figure 53: Post-layout simulation waveforms of 8-bit OR gate.

2.3.2.3. 8-bit XOR Gate

The 8-bit OR gate layout is designed to match the style of other logic gates in the logical unit. There is no problem in the DRC rules. It can be seen in Figure 54.

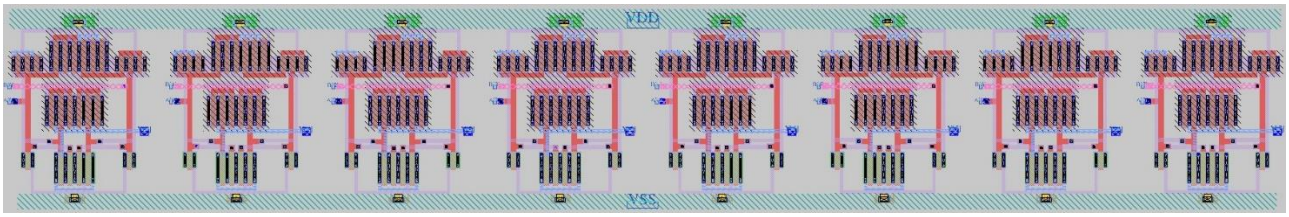

Figure 54: Layout of 8-bit XOR gate.

Figure 55 shows the schematic is verified against the layout using Layout vs. Schematic (LVS) checks to confirm its correctness.

```
Circuit 1 contains 96 devices, Circuit 2 contains 96 devices.
Circuit 1 contains 66 nets,    Circuit 2 contains 66 nets.


Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

Figure 55: LVS for 8-bir XOR gate.

Post-layout simulations (Figure 56) show that the XOR operation give the right outputs for all 8 bits, proving the design works well.

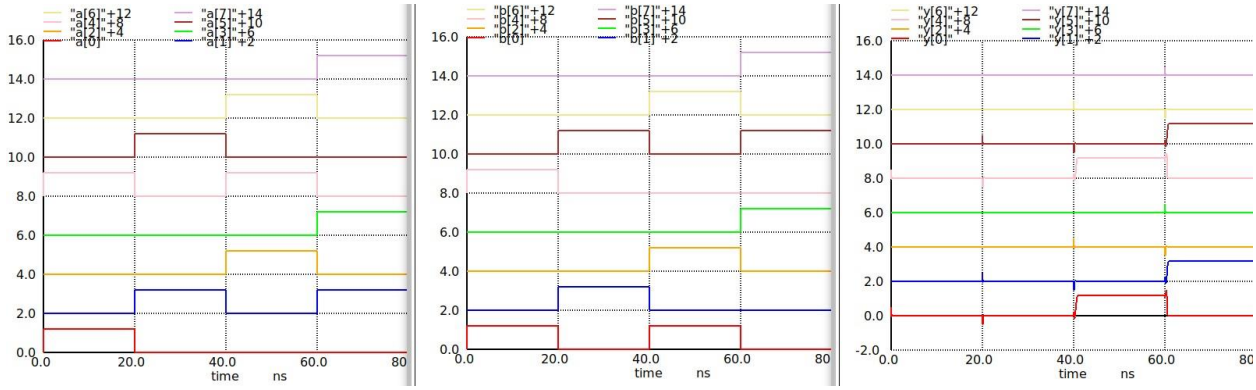

Figure 56: Post-layout simulation waveforms of 8-bit XOR gate.

2.3.2.4. 8-bit NOT Operation

Figure 57 shows the layout of the 8-bit NOT gate is designed with optimal area usage while adhering to Design Rules Check (DRC).

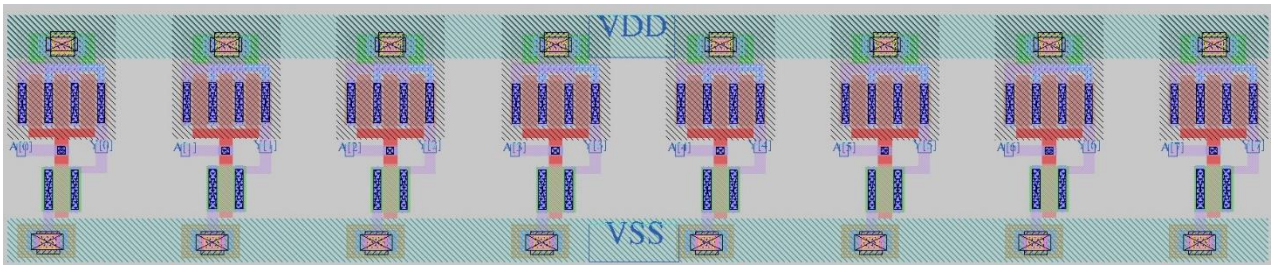

Figure 57: Layout of 8-bit NOT gate.

Figure 58 shows the schematic is verified against the layout using Layout vs. Schematic (LVS) checks to confirm its correctness.

```
Circuit 1 contains 16 devices, Circuit 2 contains 16 devices.
Circuit 1 contains 18 nets,    Circuit 2 contains 18 nets.


Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

Figure 58: LVS for 8-bit NOT gate.

Post-layout simulations are done to evaluate the functionality, showing that the NOT operation is correctly performed across all 8 bits. It can be seen in Figure 59.

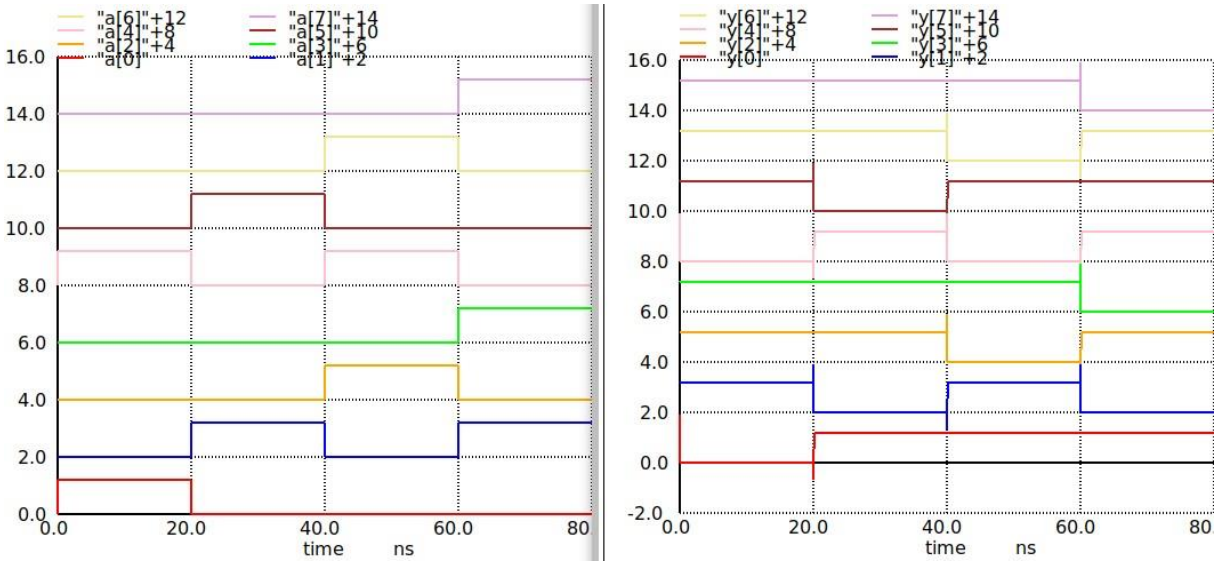

Figure 59: Post-layout simulation waveforms of 8-bit NOT gate.

2.3.2.5. Logical Unit Integration

The integration of the 4 gates (8-bit AND, OR, XOR, and NOT) into a single logical unit is carefully designed to ensure functionality and area efficiency. Figure 60 shows the layout of the integrated logical unit.

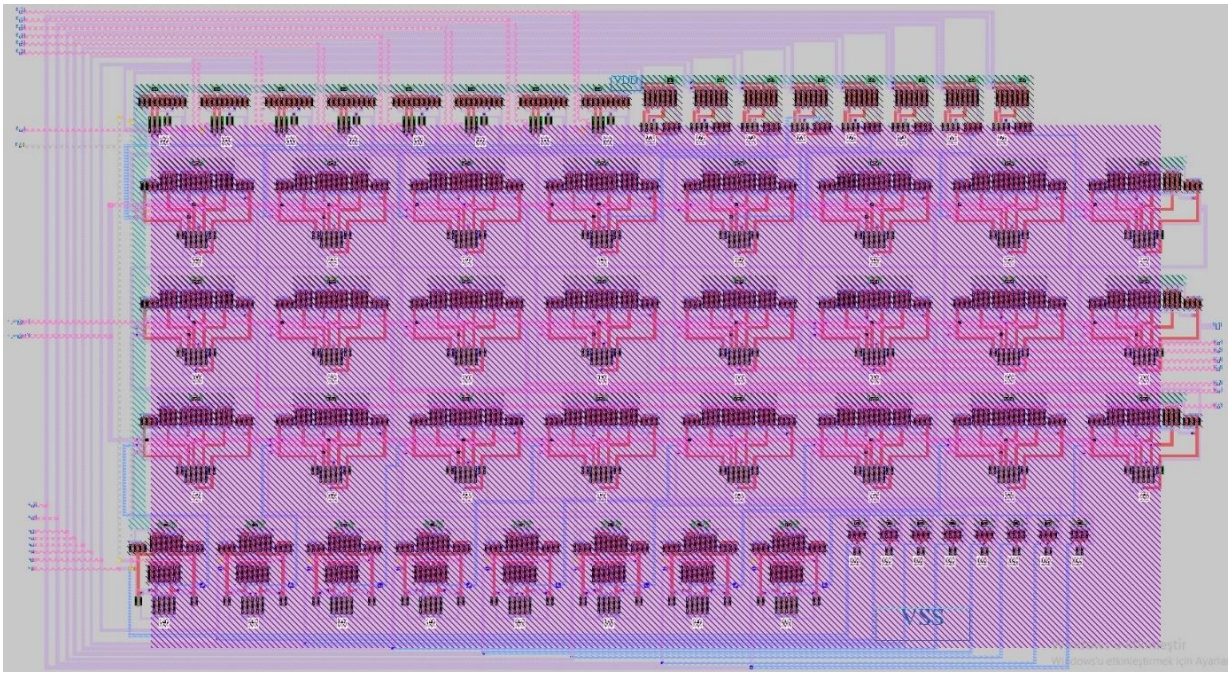

Figure 60: Layout of logic unit.

When drawing the layout, attention is paid to the inputs being on the left, outputs being on the right, VDD being on top and VSS being on the bottom. Also, VDD is on the M4 (metal 4) layer and VSS is on the M5 (metal 5) layer.

Figure 61 shows the schematic for the logical unit integration is verified against the layout using Layout vs. Schematic (LVS) checks to confirm its correctness.

```
Circuit 1 contains 496 devices, Circuit 2 contains 496 devices.
Circuit 1 contains 268 nets,    Circuit 2 contains 268 nets.


Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

Figure 61: LVS for logical unit.

Post-layout simulations are done to evaluate the functionality, showing that the logical unit operation is correctly performed across all 8 bits. It can be seen in Figure 62.

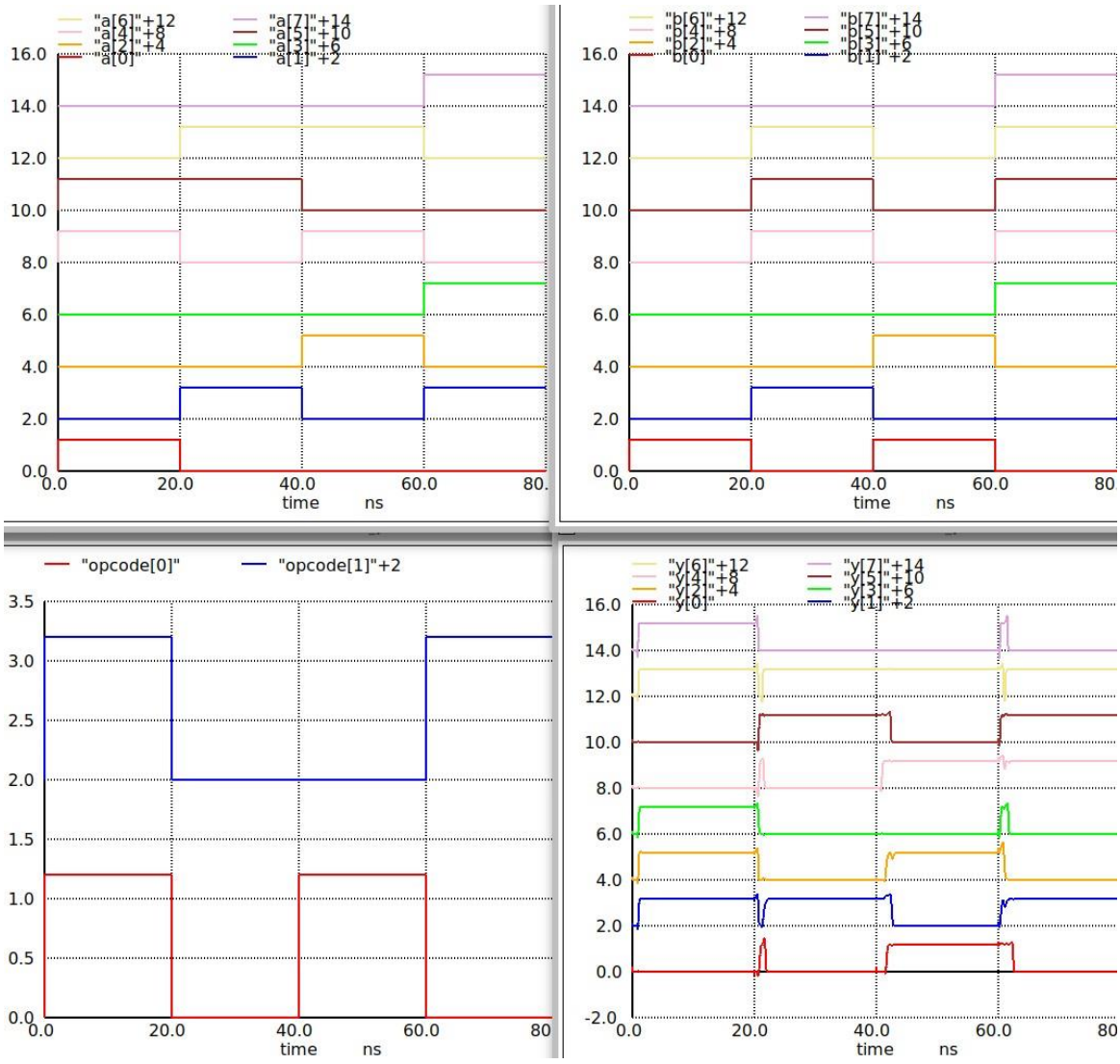

Figure 62: Post- layout simulation of 8-bit logical unit.

2.3.3. Shifter Unit

Figure 63 shows the layout of the 8-bit shifter unit is designed with optimal area usage while adhering to Design Rules Check (DRC). When drawing the layout, attention is paid to the inputs being on the left, outputs being on the right, VDD being on top and VSS being on the bottom.

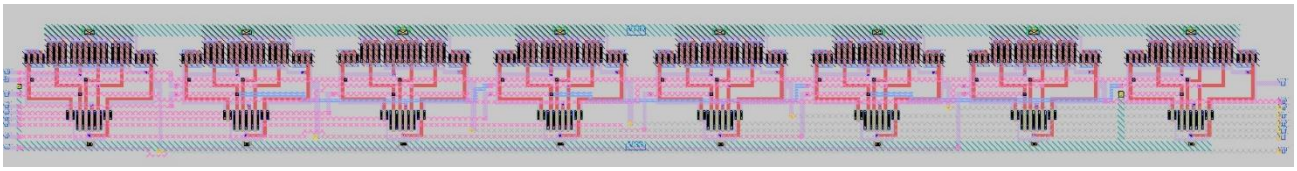

Figure 63: Layout of shifter unit.

Figure 64 shows the schematic for the 8-bit shifter unit integration is verified against the layout using Layout vs. Schematic (LVS) checks to confirm its correctness.

```
Circuit 1 contains 96 devices, Circuit 2 contains 96 devices.
Circuit 1 contains 59 nets,    Circuit 2 contains 59 nets.


Final result:
Circuits match uniquely.
.
Logging to file "comp.out" disabled
LVS Done.
```

Figure 64: LVS for 8-bit shifter unit.

Post-layout simulations are done to evaluate the functionality, showing that the shifter operation is correctly performed across all 8 bits. It can be seen in Figure 65.

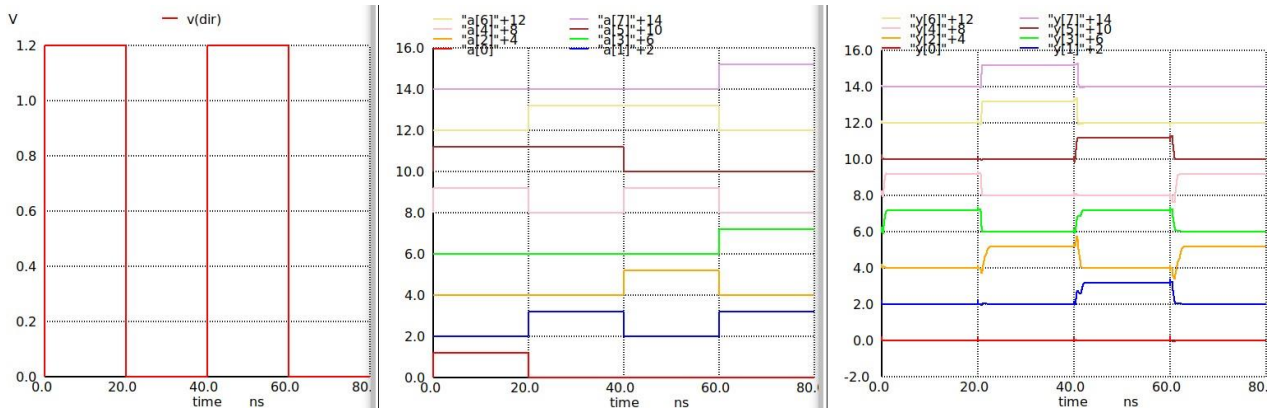

Figure 65: Post-layout simulation of 8-bit shifter unit.

2.3.4. Control Unit

The control unit consists of logic unit, shifter unit and arithmetic unit. In this section, their layout integration is drawn. It can be seen in Figure 66.

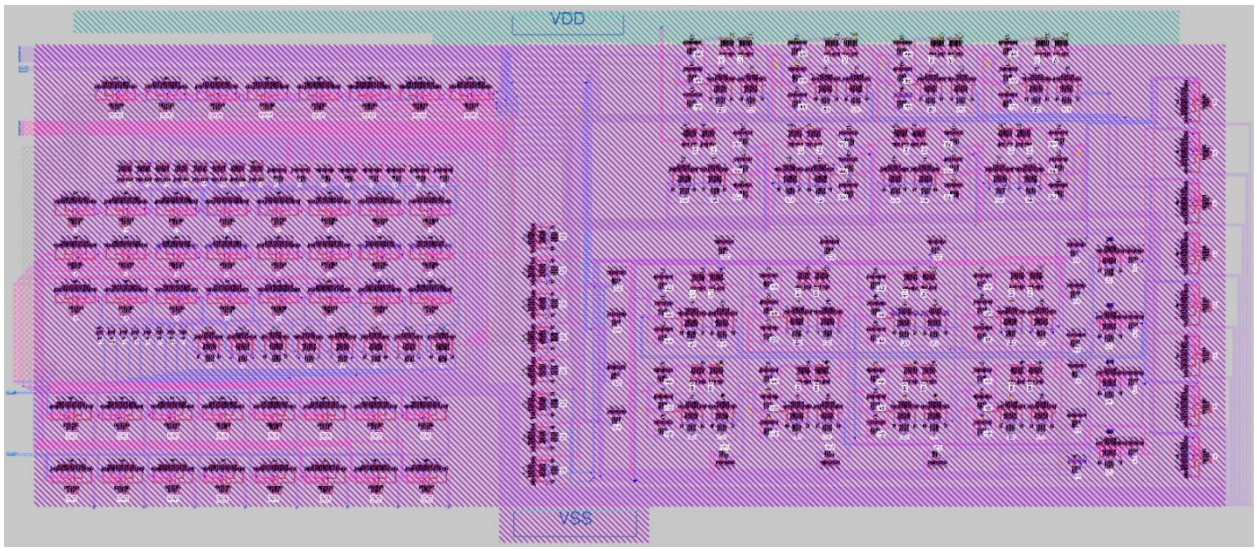

Figure 66: Layout of control unit.

Post-layout simulations are done to evaluate the functionality, showing that the control unit operation is correctly performed across all 8 bits. It can be seen in Figure 67.

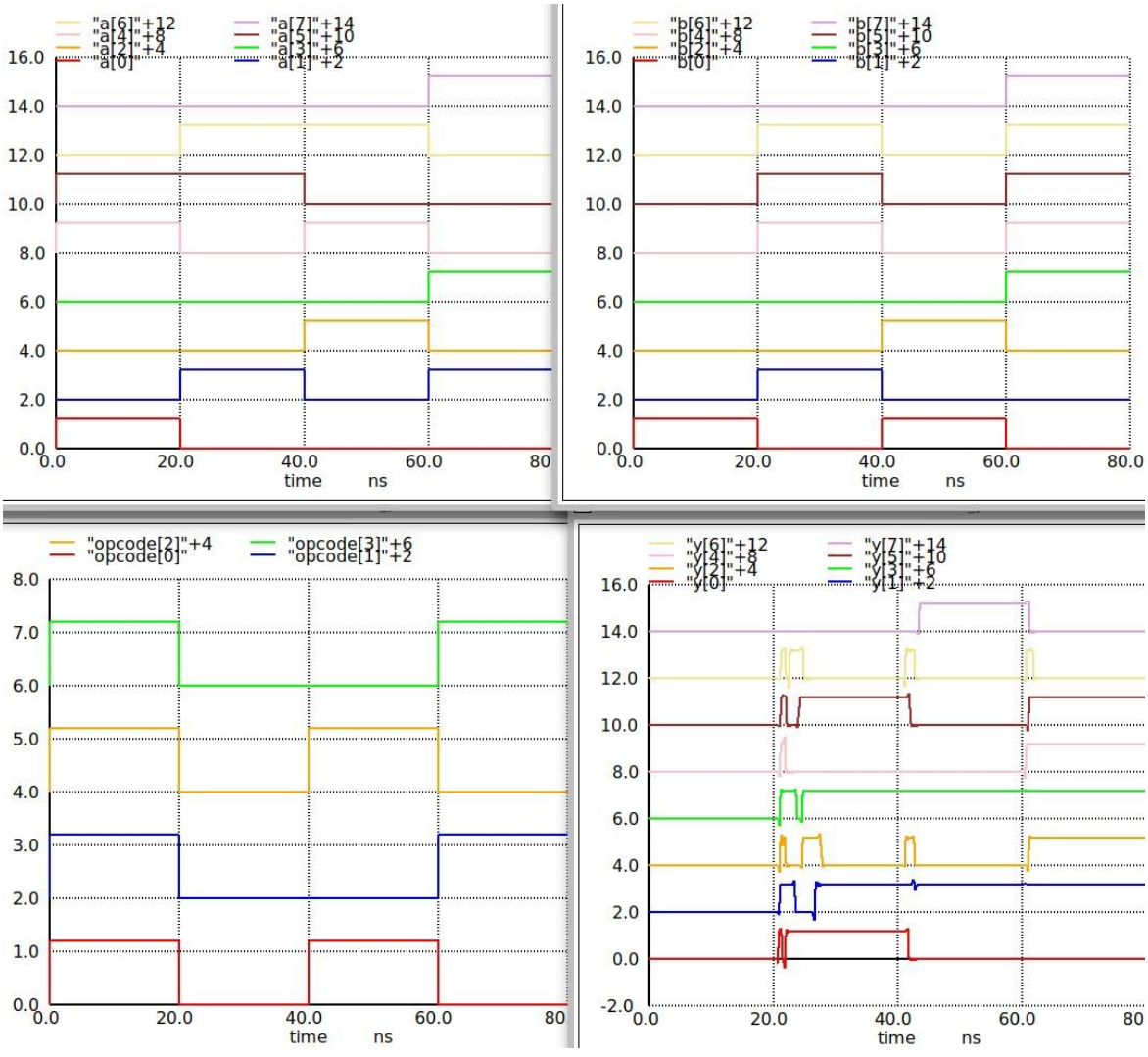

Figure 67: Post-layout simulation of 8-bit control unit.

2.3.5. ALU Integration

The combination of all sub-blocks in a hierarchical ALU design has not been achieved.

3. Conclusions

This project successfully demonstrates the design and implementation of a complex Arithmetic and Logic Unit (ALU) using CMOS technology. Through the integration of various components such as the Ripple Carry Adder (RCA), Array Multiplier, Logical Unit, and Shifter Unit, we were able to create a robust and efficient ALU capable of performing multiple arithmetic and logical operations.

The process involved extensive simulation and debugging to ensure each unit performed correctly, followed by the integration of these units into a cohesive system. The use of Xschem for schematic design and simulation provided a clear visualization of the circuits, facilitating the debugging and optimization phases.

By overcoming various challenges, including the complexity of design, timing synchronization, and component integration, the project provided valuable insights into the practical aspects of digital circuit design. The final simulation results confirmed the functionality of the ALU, showcasing its ability to handle different operations as intended.

In conclusion, this project not only achieved its objectives but also enhanced our understanding of digital design principles, circuit simulation, and the use of tools like Xschem. The successful implementation of the ALU serves as a solid foundation for future enhancements and more advanced digital designs.

4. References

- "Youtube.com/watch?v=RPppaGdjbj0&t=1948s", YouTube. [Online]. Available: https://www.youtube.com/watch?v=RPppaGdjbj0&t=1948s. [Accessed: 14-Jan-2025].
- "Skywater PDK Assumptions", Skywater-pdk.readthedocs.io. [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/rules/assumptions.html. [Accessed: 14-Jan-2025].
- "Youtu.be/GxOUWHxIfSQ", YouTube. [Online]. Available: https://youtu.be/GxOUWHxIfSQ. [Accessed: 14-Jan-2025].

5. Verilog Code

```
module alu_output_unit(A,B,Y,opcode,flags);
  input wire [7:0]A;
  input wire [7:0]B;
  input wire [3:0]opcode;
  output wire [7:0]Y;
  output wire [3:0]flags; //ZCVS

  wire flag_zero;
  wire flag_overflow;
  wire in_signs_same;
  wire out_signs_not_same;

  xor2 g0(.A(A[7]), .B(B[7]), .Y(in_signs_same));
  xnor2 g1(.A(A[7]), .B(Y[7]), .Y(out_signs_not_same));
  and2 g2(.A(in_signs_same), .B(out_signs_not_same), .Y(flags[1]));


  control_unit cu0(.A(A), .B(B), .Y(Y), .opcode(opcode), .carry_out(flags[2]));
  nor8 n0(.A(Y), .Y(flag_zero));

  assign flags[0] = Y[7];
  assign flags[3] = flag_zero;

endmodule
```

```
module inv(A,Y);
  input wire A;
  output wire Y;

  assign Y = ~A ;

endmodule

module and2(A,B,Y);
  input wire A;
  input wire B;
  output wire Y;

  assign Y = A & B ;

endmodule

module or2(A,B,Y);
  input wire A;
  input wire B;
  output wire Y;

  assign Y = A | B ;

endmodule

module or3(A,B,C,Y);
  input wire A;
  input wire B;
  input wire C;
  output wire Y;

  assign Y = A | B | C;
```

```
module nand2(A,B,Y);
  input wire A;
  input wire B;
  output wire Y;

  assign Y = ~(A & B) ;

endmodule

module nor8(A,Y);
  input wire [7:0]A;
  output wire Y;

  assign Y =~(|A) ;

endmodule


module xnor2(A,B,Y);
  input wire A;
  input wire B;
  output wire Y;

  assign Y = ~(A ^ B);

endmodule

module xor2(A,B,Y);
  input wire A;
  input wire B;
  output wire Y;

  assign Y = A ^ B;

endmodule

module xor3(A,B,C,Y);
  input wire A;
  input wire B;
  input wire C;
  output wire Y;

  assign Y = A ^ B ^ C;

endmodule

module mux2(A,B,S,Y);
  input wire A;
  input wire B;
  input wire S;
  output wire Y;

  wire [1:0]nand_out;
  wire S_not;

  inv gate0(.A(S), .Y(S_not));

  nand2 gate1(.A(A), .B(S), .Y(nand_out[0]));
  nand2 gate2(.A(S_not), .B(B), .Y(nand_out[1]));
  nand2 gate3(.A(nand_out[0]), .B(nand_out[1]), .Y(Y));

endmodule
```

```
module control_unit(A,B,opcode,Y,carry_out);
  input wire [7:0]A;
  input wire [7:0]B;
  input wire [3:0]opcode;
  output wire [7:0]Y;
  output wire carry_out;

  wire [7:0]arithmetic_out;
  wire [7:0]logic_out;
  wire [7:0]arithmeticlogic_out;
  wire [7:0]shifter_out;

  arithmetic_unit au0(.A(A), .B(B), .opcode(opcode), .Y(arithmetic_out), .carry_out(carry_out));
  logic_unit lo0(.A(A), .B(B), .opcode(opcode), .Y(logic_out));
  shifter_unit sh0(.A(A), .Y(shifter_out), .dir(opcode[0]));

  mux2 m10(.A(arithmetic_out[0]), .B(logic_out[0]), .S(opcode[2]), .Y(arithmeticlogic_out[0]));
  mux2 m11(.A(arithmetic_out[1]), .B(logic_out[1]), .S(opcode[2]), .Y(arithmeticlogic_out[1]));
  mux2 m12(.A(arithmetic_out[2]), .B(logic_out[2]), .S(opcode[2]), .Y(arithmeticlogic_out[2]));
  mux2 m13(.A(arithmetic_out[3]), .B(logic_out[3]), .S(opcode[2]), .Y(arithmeticlogic_out[3]));
  mux2 m14(.A(arithmetic_out[4]), .B(logic_out[4]), .S(opcode[2]), .Y(arithmeticlogic_out[4]));
  mux2 m15(.A(arithmetic_out[5]), .B(logic_out[5]), .S(opcode[2]), .Y(arithmeticlogic_out[5]));
  mux2 m16(.A(arithmetic_out[6]), .B(logic_out[6]), .S(opcode[2]), .Y(arithmeticlogic_out[6]));
  mux2 m17(.A(arithmetic_out[7]), .B(logic_out[7]), .S(opcode[2]), .Y(arithmeticlogic_out[7]));

  mux2 m20(.A(arithmeticlogic_out[0]), .B(shifter_out[0]), .S(opcode[3]), .Y(Y[0]));
  mux2 m21(.A(arithmeticlogic_out[1]), .B(shifter_out[1]), .S(opcode[3]), .Y(Y[1]));
  mux2 m22(.A(arithmeticlogic_out[2]), .B(shifter_out[2]), .S(opcode[3]), .Y(Y[2]));
  mux2 m23(.A(arithmeticlogic_out[3]), .B(shifter_out[3]), .S(opcode[3]), .Y(Y[3]));
  mux2 m24(.A(arithmeticlogic_out[4]), .B(shifter_out[4]), .S(opcode[3]), .Y(Y[4]));
  mux2 m25(.A(arithmeticlogic_out[5]), .B(shifter_out[5]), .S(opcode[3]), .Y(Y[5]));
  mux2 m26(.A(arithmeticlogic_out[6]), .B(shifter_out[6]), .S(opcode[3]), .Y(Y[6]));
  mux2 m27(.A(arithmeticlogic_out[7]), .B(shifter_out[7]), .S(opcode[3]), .Y(Y[7]));

endmodule
```

```
module arithmetic_unit(A,B,opcode,Y,carry_out);
  input wire [7:0]A;
  input wire [7:0]B;
  input wire [3:0]opcode;
  output wire [7:0]Y;
  output wire carry_out;

  wire [7:0]sub_op2;
  wire [7:0]carry_ripple_adder_out, array_multiplier_out;

  xor2 sub0(.A(opcode[0]), .B(B[0]), .Y(sub_op2[0]));
  xor2 sub1(.A(opcode[0]), .B(B[1]), .Y(sub_op2[1]));
  xor2 sub2(.A(opcode[0]), .B(B[2]), .Y(sub_op2[2]));
  xor2 sub3(.A(opcode[0]), .B(B[3]), .Y(sub_op2[3]));
  xor2 sub4(.A(opcode[0]), .B(B[4]), .Y(sub_op2[4]));
  xor2 sub5(.A(opcode[0]), .B(B[5]), .Y(sub_op2[5]));
  xor2 sub6(.A(opcode[0]), .B(B[6]), .Y(sub_op2[6]));
  xor2 sub7(.A(opcode[0]), .B(B[7]), .Y(sub_op2[7]));

  carry_ripple_adder cra0(.A(A), .B(sub_op2), .Y(carry_ripple_adder_out), .carry_in(opcode[0]), .carry_out(carry_out));
  array_multiplier aml0(.A(A[7:4]), .B(B[7:4]), .Y(array_multiplier_out));

  mux2 yout0(.A(carry_ripple_adder_out[0]), .B(array_multiplier_out[0]), .S(opcode[1]), .Y(Y[0]));
  mux2 yout1(.A(carry_ripple_adder_out[1]), .B(array_multiplier_out[1]), .S(opcode[1]), .Y(Y[1]));
  mux2 yout2(.A(carry_ripple_adder_out[2]), .B(array_multiplier_out[2]), .S(opcode[1]), .Y(Y[2]));
  mux2 yout3(.A(carry_ripple_adder_out[3]), .B(array_multiplier_out[3]), .S(opcode[1]), .Y(Y[3]));
  mux2 yout4(.A(carry_ripple_adder_out[4]), .B(array_multiplier_out[4]), .S(opcode[1]), .Y(Y[4]));
  mux2 yout5(.A(carry_ripple_adder_out[5]), .B(array_multiplier_out[5]), .S(opcode[1]), .Y(Y[5]));
  mux2 yout6(.A(carry_ripple_adder_out[6]), .B(array_multiplier_out[6]), .S(opcode[1]), .Y(Y[6]));
  mux2 yout7(.A(carry_ripple_adder_out[7]), .B(array_multiplier_out[7]), .S(opcode[1]), .Y(Y[7]));


endmodule
```

```
module logic_unit(A,B,opcode,Y);
  input wire [7:0]A;
  input wire [7:0]B;
  input wire [3:0]opcode;
  output wire [7:0]Y;

  wire [7:0] and_out;
  wire [7:0] or_out;
  wire [7:0] xor_out;
  wire [7:0] inv_out;
  wire [7:0] andor_out;
  wire [7:0] xorinv_out;

  logic_and land0(.A(A), .B(B), .Y(and_out));
  logic_or lor0(.A(A), .B(B), .Y(or_out));
  logic_xor xor0(.A(A), .B(B), .Y(xor_out));
  logic_inv inv0(.A(A), .Y(inv_out));


  mux2 l10(.A(and_out[0]), .B(or_out[0]), .S(opcode[0]), .Y(andor_out[0]));
  mux2 l11(.A(and_out[1]), .B(or_out[1]), .S(opcode[0]), .Y(andor_out[1]));
  mux2 l12(.A(and_out[2]), .B(or_out[2]), .S(opcode[0]), .Y(andor_out[2]));
  mux2 l13(.A(and_out[3]), .B(or_out[3]), .S(opcode[0]), .Y(andor_out[3]));
  mux2 l14(.A(and_out[4]), .B(or_out[4]), .S(opcode[0]), .Y(andor_out[4]));
  mux2 l15(.A(and_out[5]), .B(or_out[5]), .S(opcode[0]), .Y(andor_out[5]));
  mux2 l16(.A(and_out[6]), .B(or_out[6]), .S(opcode[0]), .Y(andor_out[6]));
  mux2 l17(.A(and_out[7]), .B(or_out[7]), .S(opcode[0]), .Y(andor_out[7]));

  mux2 l20(.A(xor_out[0]), .B(inv_out[0]), .S(opcode[0]), .Y(xorinv_out[0]));
  mux2 l21(.A(xor_out[1]), .B(inv_out[1]), .S(opcode[0]), .Y(xorinv_out[1]));
  mux2 l22(.A(xor_out[2]), .B(inv_out[2]), .S(opcode[0]), .Y(xorinv_out[2]));
  mux2 l23(.A(xor_out[3]), .B(inv_out[3]), .S(opcode[0]), .Y(xorinv_out[3]));
  mux2 l24(.A(xor_out[4]), .B(inv_out[4]), .S(opcode[0]), .Y(xorinv_out[4]));
  mux2 l25(.A(xor_out[5]), .B(inv_out[5]), .S(opcode[0]), .Y(xorinv_out[5]));
  mux2 l26(.A(xor_out[6]), .B(inv_out[6]), .S(opcode[0]), .Y(xorinv_out[6]));
  mux2 l27(.A(xor_out[7]), .B(inv_out[7]), .S(opcode[0]), .Y(xorinv_out[7]));

  mux2 l30(.A(andor_out[0]), .B(xorinv_out[0]), .S(opcode[1]), .Y(Y[0]));
  mux2 l31(.A(andor_out[1]), .B(xorinv_out[1]), .S(opcode[1]), .Y(Y[1]));
  mux2 l32(.A(andor_out[2]), .B(xorinv_out[2]), .S(opcode[1]), .Y(Y[2]));
  mux2 l33(.A(andor_out[3]), .B(xorinv_out[3]), .S(opcode[1]), .Y(Y[3]));
  mux2 l34(.A(andor_out[4]), .B(xorinv_out[4]), .S(opcode[1]), .Y(Y[4]));
  mux2 l35(.A(andor_out[5]), .B(xorinv_out[5]), .S(opcode[1]), .Y(Y[5]));
  mux2 l36(.A(andor_out[6]), .B(xorinv_out[6]), .S(opcode[1]), .Y(Y[6]));
  mux2 l37(.A(andor_out[7]), .B(xorinv_out[7]), .S(opcode[1]), .Y(Y[7]));
endmodule

module logic_and(A,B,Y);
  input wire [7:0]A;
  input wire [7:0]B;
  output wire [7:0]Y;

  and2 a0(.A(A[0]), .B(B[0]), .Y(Y[0]));
  and2 a1(.A(A[1]), .B(B[1]), .Y(Y[1]));
  and2 a2(.A(A[2]), .B(B[2]), .Y(Y[2]));
  and2 a3(.A(A[3]), .B(B[3]), .Y(Y[3]));
  and2 a4(.A(A[4]), .B(B[4]), .Y(Y[4]));
  and2 a5(.A(A[5]), .B(B[5]), .Y(Y[5]));
  and2 a6(.A(A[6]), .B(B[6]), .Y(Y[6]));
  and2 a7(.A(A[7]), .B(B[7]), .Y(Y[7]));

endmodule
```

```
module logic_or(A,B,Y);
  input wire [7:0]A;
  input wire [7:0]B;
  output wire [7:0]Y;

  or2 o0(.A(A[0]), .B(B[0]), .Y(Y[0]));
  or2 o1(.A(A[1]), .B(B[1]), .Y(Y[1]));
  or2 o2(.A(A[2]), .B(B[2]), .Y(Y[2]));
  or2 o3(.A(A[3]), .B(B[3]), .Y(Y[3]));
  or2 o4(.A(A[4]), .B(B[4]), .Y(Y[4]));
  or2 o5(.A(A[5]), .B(B[5]), .Y(Y[5]));
  or2 o6(.A(A[6]), .B(B[6]), .Y(Y[6]));
  or2 o7(.A(A[7]), .B(B[7]), .Y(Y[7]));

endmodule

module logic_xor(A,B,Y);
  input wire [7:0]A;
  input wire [7:0]B;
  output wire [7:0]Y;

  xor2 xo0(.A(A[0]), .B(B[0]), .Y(Y[0]));
  xor2 xo1(.A(A[1]), .B(B[1]), .Y(Y[1]));
  xor2 xo2(.A(A[2]), .B(B[2]), .Y(Y[2]));
  xor2 xo3(.A(A[3]), .B(B[3]), .Y(Y[3]));
  xor2 xo4(.A(A[4]), .B(B[4]), .Y(Y[4]));
  xor2 xo5(.A(A[5]), .B(B[5]), .Y(Y[5]));
  xor2 xo6(.A(A[6]), .B(B[6]), .Y(Y[6]));
  xor2 xo7(.A(A[7]), .B(B[7]), .Y(Y[7]));

endmodule

module logic_inv(A,Y);
  input wire [7:0]A;
  output wire [7:0]Y;

  inv i0(.A(A[0]), .Y(Y[0]));
  inv i1(.A(A[1]), .Y(Y[1]));
  inv i2(.A(A[2]), .Y(Y[2]));
  inv i3(.A(A[3]), .Y(Y[3]));
  inv i4(.A(A[4]), .Y(Y[4]));
  inv i5(.A(A[5]), .Y(Y[5]));
  inv i6(.A(A[6]), .Y(Y[6]));
  inv i7(.A(A[7]), .Y(Y[7]));

endmodule
```

```
module shifter_unit(A, Y, dir);
  input wire [7:0]A;
  input wire dir;
  output wire [7:0]Y;

  wire [7:0]left_shifter_out, right_shifter_out;

  left_shifter ls0(.A(A), .Y(left_shifter_out));
  right_shifter rs0(.A(A), .Y(right_shifter_out));

  mux2 s0(.A(left_shifter_out[0]), .B(right_shifter_out[0]), .S(dir), .Y(Y[0]));
  mux2 s1(.A(left_shifter_out[1]), .B(right_shifter_out[1]), .S(dir), .Y(Y[1]));
  mux2 s2(.A(left_shifter_out[2]), .B(right_shifter_out[2]), .S(dir), .Y(Y[2]));
  mux2 s3(.A(left_shifter_out[3]), .B(right_shifter_out[3]), .S(dir), .Y(Y[3]));
  mux2 s4(.A(left_shifter_out[4]), .B(right_shifter_out[4]), .S(dir), .Y(Y[4]));
  mux2 s5(.A(left_shifter_out[5]), .B(right_shifter_out[5]), .S(dir), .Y(Y[5]));
  mux2 s6(.A(left_shifter_out[6]), .B(right_shifter_out[6]), .S(dir), .Y(Y[6]));
  mux2 s7(.A(left_shifter_out[7]), .B(right_shifter_out[7]), .S(dir), .Y(Y[7]));


endmodule

module left_shifter(A, Y);
  input wire [7:0]A;
  output wire [7:0]Y;

  assign Y = {A[6:0],A[7]};
endmodule


module right_shifter(A, Y);
  input wire [7:0]A;
  output wire [7:0]Y;

  assign Y = {A[0],A[7:1]};
endmodule
```