**Politehnica University of Timișoara**

**Faculty of Automation and Computer Science**

**ALU 8-BIT SIMULATION**

**Coordinator teacher: Students:**

Bozdog AlexandruNemeș Lică

Remețan Mihnea-Florin

Tănase Elena-Alexandra

Tîrsîna Nicoleta

**2’nd Semester, 2024-2025**

**Table of Contents**

1. [Project Overview](#project-overview)
2. [Development Environment](#development-environment)
3. [Project Phases](#project-phases)
   * [Design Phase](#design-phase)
   * [Implementation Phase](#implementation-phase)
   * [Testing Phase](#testing-phase)
4. [Logisim Architecture Diagrams](#logisim-architecture-diagrams)
5. [Verilog Code Snippets](#verilog-code-snippets)
6. [Challenges & Resolutions](#challenges--resolutions)
7. Conclusion

* **Project Overview**

An **Arithmetic Logic Unit** (ALU) is a core component of any CPU or digital processing system. It performs arithmetic and logic operations on binary data. In a structural implementation, the ALU is not written using behavioral code like if or case statements, but instead is built using interconnected submodules, such as adders, logic gates, and multiplexers—mimicking actual hardware design.

The purpose is to understand digital arithmetic operations at the gate/module level, implement reusable components, and simulate behavior using both **Logisim Evolution (for architecture)** and **Verilog (for actual simulation)**.

This project aims to simulate a structurally designed 8-bit ALU in Verilog, integrated with four core arithmetic operations:

* **Addition (Ripple Carry Adder)**
* **Subtraction (2’s Complement via EXOR)**
* **Multiplication (Booth’s Radix-2 Algorithm)**
* **Division (Non-Restoring Algorithm)**

For **addition**, we chose the **Ripple Carry Adder (RCA)** because it is a simple and efficient design for small bit-width operations like 8-bit arithmetic. It uses chained full adders, where the carry-out of each stage is passed to the next, making it easy to implement structurally in Verilog, verifying the overflow case by using an EXOR gate between c7 and c8 carries.

For **subtraction**, we decided to implement it using the **EXOR gate** combined with an RCA to perform **two’s complement subtraction**. This approach is efficient because it reuses the existing adder module, and the EXOR gate allows us to invert the bits of the second operand.

For **multiplication** and **division**, we selected **Booth's Radix-2 Algorithm** and the **Non-Restoring Division Algorithm**, respectively. These algorithms are relatively simple to implement in Verilog and have a **clear, step-based architecture** that is easy to understand and follow. Booth’s algorithm effectively handles signed numbers, reducing the number of required additions or subtractions, while Non-Restoring Division simplifies control logic by avoiding multiple restore steps.

* **Development Environment**
* **Logisim Evolution** – for architectural design and visual debugging
* **Visual Studio Code** – for writing and managing Verilog files
* **Simulator**: GTKWave
* **Git** – version control

Folder structure:

8-bit-ALU-Simulation-Project/

├── ALU

│ ├── ALU.v

│ ├── ALU\_tb.v

├── Control\_Unit

│ ├── Control\_Unit.v

│ ├── Control\_Unit\_Structural.v

│ ├── Control\_Unit\_tb.v

├── Diagram

│ └── Diagram.jpg

├── RCA

│ ├── RCA.v

│ └── fac.v

└── Registers

├── A.v

├── M.v

├── Q.v

├── combinational.v

├── counter.v

├── ffd.v

└── testbenches.v

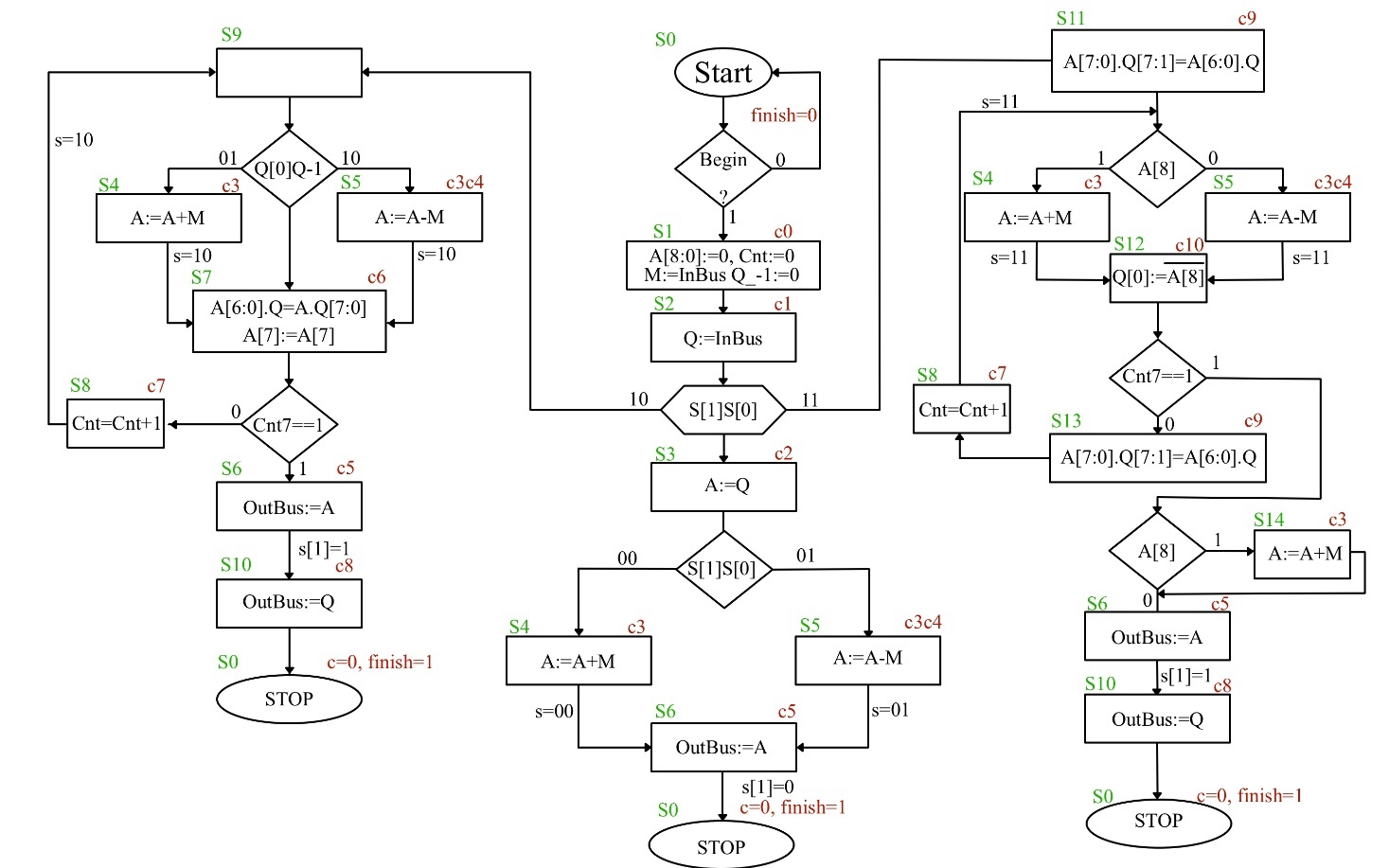
* **Project Phases**
* **Design Phase**

**➤ Logisim Architecture Design**

Using **Logisim Evolution**, the 8-bit ALU design was mapped out with separate modules for RCA, A-REGISTER, Q-REGISTER, M-REGISTER, COUNTER, SHIFT REGISTERS, CONTROL UNIT and ALU. These operations are selected via a control unit using a **2-bit variable.**

**➤ Operations Mapping (via Control Unit)**

|  |  |  |
| --- | --- | --- |
| **SEL** | **OPERATION** | **MODULES USED** |
| 00 | Addition | FAC -> RCA |
| 01 | Subtraction | RCA + EXOR |
| 10 | Multiplication | RCA + A,Q,M, Count |
| 11 | Division | RCA + A,Q,M, Count |

* **Implementation Phase**

Truth table:A table of numbers and digits

AI-generated content may be incorrect.

Each module was implemented in Verilog using **modular and structural design** principles.

**✅ Code Example: Ripple Carry Adder (RCA.v)**

module fac(

input x,y,ci,

output z,co

);

assign z=x^y^ci;

assign co=(x&y)|(x&ci)|(y&ci);

endmodule

module RCA(

input [8:0]x,y,

input ci,

output [8:0]z,

output co

);

wire w\_co[8:0];

genvar i;

generate

for(i=0;i<9;i=i+1)

begin

if(i==0)

fac f(

.x(x[0]),

.y(y[0]),

.ci(ci),

.z(z[0]),

.co(w\_co[0])

);

else if(i==8)

fac f(

.x(x[8]),

.y(y[8]),

.ci(w\_co[7]),

.z(z[8]),

.co(co)

);

else

fac f(

.x(x[i]),

.y(y[i]),

.ci(w\_co[i-1]),

.z(z[i]),

.co(w\_co[i])

);

end

endgenerate

endmodule

**✅ Code Snippet: ffd.v, A.v, Q.v, M.v, counter.v -> registers**

**These registers are implemented using *data flip-flops and multiplexers.***

**module** **ffd** (

input clk,rst\_b,

input en,d,

output reg q

);

always@(posedge clk or negedge rst\_b)

begin

if(!rst\_b)

q<=1'b0;

else if(en)

q<=d;

end

endmodule

[…] –some more code

**module** **mux\_2s** #(

parameter w=4

)(

input [w-1:0]d0,d1,d2,d3,

input [1:0] sel,

output [w-1:0]o

);

assign o=(sel==2'b00) ? d0:

(sel==2'b01) ? d1:

(sel==2'b10) ? d2:

(sel==2'b11) ? d3:{w{1'bz}};

endmodule

[…] – some more code

**module A**(

input clk,rst\_b,

input q7,

input [1:0]sel,

input [8:0]in,

output[8:0]out

);

wire[8:0]mux\_wire;

mux\_2s#(.w(1)) m0(.d0(out[0]),.d1(out[1]),.d2(q7),.d3(in[0]),.sel(sel),.o(mux\_wire[0]));

mux\_2s#(.w(1)) m1(.d0(out[1]),.d1(out[2]),.d2(out[0]),.d3(in[1]),.sel(sel),.o(mux\_wire[1]));

mux\_2s#(.w(1)) m2(.d0(out[2]),.d1(out[3]),.d2(out[1]),.d3(in[2]),.sel(sel),.o(mux\_wire[2]));

mux\_2s#(.w(1)) m3(.d0(out[3]),.d1(out[4]),.d2(out[2]),.d3(in[3]),.sel(sel),.o(mux\_wire[3]));

mux\_2s#(.w(1)) m4(.d0(out[4]),.d1(out[5]),.d2(out[3]),.d3(in[4]),.sel(sel),.o(mux\_wire[4]));

mux\_2s#(.w(1)) m5(.d0(out[5]),.d1(out[6]),.d2(out[4]),.d3(in[5]),.sel(sel),.o(mux\_wire[5]));

mux\_2s#(.w(1)) m6(.d0(out[6]),.d1(out[7]),.d2(out[5]),.d3(in[6]),.sel(sel),.o(mux\_wire[6]));

mux\_2s#(.w(1)) m7(.d0(out[7]),.d1(out[7]),.d2(out[6]),.d3(in[7]),.sel(sel),.o(mux\_wire[7]));

mux\_2s#(.w(1)) m8(.d0(out[8]),.d1(out[8]),.d2(out[8]),.d3(in[8]),.sel(sel),.o(mux\_wire[8]));

ffd f0(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[0]),.q(out[0]));

ffd f1(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[1]),.q(out[1]));

ffd f2(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[2]),.q(out[2]));

ffd f3(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[3]),.q(out[3]));

ffd f4(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[4]),.q(out[4]));

ffd f5(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[5]),.q(out[5]));

ffd f6(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[6]),.q(out[6]));

ffd f7(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[7]),.q(out[7]));

ffd f8(.clk(clk),.rst\_b(rst\_b),.en(1'b1),.d(mux\_wire[8]),.q(out[8]));

endmodule

**Note: Q, M Register are implemented similarly!**

**module counter** (

input clk,rst\_b,c\_up,

output [2:0]out

);

ffd f0(.clk(clk),.rst\_b(rst\_b),.en(c\_up),.d(~out[0]),.q(out[0]));

ffd f1(.clk(clk),.rst\_b(rst\_b),.en(c\_up),.d(out[0]^out[1]),.q(out[1]));

ffd f2(.clk(clk),.rst\_b(rst\_b),.en(c\_up),.d(out[2]&(~out[1]) | out[2]&(~out[0]) | (~out[2])&out[1]&out[0]),.q(out[2]));

endmodule

**✅ Control Unit (Control\_Unit.v)**

module Control\_Unit(

input clk,rst\_b,

input [1:0]s,

input start,q0,q\_1,a\_8,

input [2:0]cnt,

output [10:0]c,

output finish

);

wire [3:0] qout;

ffd f3(.clk(clk),.rst\_b(rst\_b),.en(1'b1),

.d(

~qout[3]&~qout[2]&qout[1]&~qout[0]&s[1]&~s[0]

| ~qout[3]&~qout[2]&qout[1]&~qout[0]&s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&s[0]

| ~qout[3]&qout[2]&qout[1]&~qout[0]&s[1]

| ~qout[3]&qout[2]&qout[1]&qout[0]&~(cnt[2]&cnt[1]&cnt[0])

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&~s[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&a\_8&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&~(cnt[2]&cnt[1]&cnt[0])

| qout[3]&qout[2]&~qout[1]&qout[0]

),

.q(qout[3]));

ffd f2(.clk(clk),.rst\_b(rst\_b),.en(1'b1),

.d(

~qout[3]&~qout[2]&qout[1]&qout[0]&~s[1]&~s[0]

| ~qout[3]&~qout[2]&qout[1]&qout[0]&~s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&~s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&~s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&s[0]

| ~qout[3]&qout[2]&qout[1]&qout[0]&cnt[2]&cnt[1]&cnt[0]

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&s[0]&a\_8

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&s[0]&~a\_8

| qout[3]&~qout[2]&~qout[1]&qout[0]&~q0&q\_1

| qout[3]&~qout[2]&~qout[1]&qout[0]&q0&~q\_1

| qout[3]&~qout[2]&qout[1]&qout[0]&a\_8

| qout[3]&~qout[2]&qout[1]&qout[0]&~a\_8

| qout[3]&qout[2]&~qout[1]&~qout[0]&a\_8&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&~a\_8&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&~(cnt[2]&cnt[1]&cnt[0])

| qout[3]&~qout[2]&~qout[1]&qout[0]&(q0~^q\_1)

| qout[3]&qout[2]&qout[1]&~qout[0]

),

.q(qout[2]));

ffd f1(.clk(clk),.rst\_b(rst\_b),.en(1'b1),

.d(

~qout[3]&~qout[2]&~qout[1]&qout[0]

| ~qout[3]&~qout[2]&qout[1]&~qout[0]&~s[1]

| ~qout[3]&~qout[2]&qout[1]&~qout[0]&s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&~s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&~s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&~s[0]

| ~qout[3]&qout[2]&qout[1]&~qout[0]&s[1]

| ~qout[3]&qout[2]&qout[1]&qout[0]&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&a\_8&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&~a\_8&cnt[2]&cnt[1]&cnt[0]

| qout[3]&~qout[2]&~qout[1]&qout[0]&(q0~^q\_1)

| qout[3]&qout[2]&qout[1]&~qout[0]

),

.q(qout[1]));

ffd f0(.clk(clk),.rst\_b(rst\_b),.en(1'b1),

.d(

~qout[3]&~qout[2]&~qout[1]&~qout[0]&start

| ~qout[3]&~qout[2]&qout[1]&~qout[0]&~s[1]

| ~qout[3]&~qout[2]&qout[1]&~qout[0]&s[1]&~s[0]

| ~qout[3]&~qout[2]&qout[1]&~qout[0]&s[1]&s[0]

| ~qout[3]&~qout[2]&qout[1]&qout[0]&~s[1]&s[0]

| ~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&~s[0]

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&~s[0]

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&s[0]&~a\_8

| qout[3]&~qout[2]&~qout[1]&qout[0]&q0&~q\_1

| qout[3]&~qout[2]&qout[1]&qout[0]&~a\_8

| qout[3]&~qout[2]&~qout[1]&qout[0]&(q0~^q\_1)

| qout[3]&qout[2]&~qout[1]&~qout[0]&~(cnt[2]&cnt[1]&cnt[0])

),

.q(qout[0]));

assign c[10]=~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&s[0] | ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&s[0];

assign c[9]=~qout[3]&~qout[2]&qout[1]&~qout[0]&s[1]&s[0] | qout[3]&qout[2]&~qout[1]&~qout[0]&~(cnt[2]&cnt[1]&cnt[0]);

assign c[8]=~qout[3]&qout[2]&qout[1]&~qout[0]&s[1];

assign c[7]=~qout[3]&qout[2]&qout[1]&qout[0]&~(cnt[2]&cnt[1]&cnt[0]) | qout[3]&qout[2]&~qout[1]&qout[0];

assign c[6]=~qout[3]&qout[2]&~qout[1]&~qout[0]&s[1]&~s[0] | ~qout[3]&qout[2]&~qout[1]&qout[0]&s[1]&~s[0] | qout[3]&~qout[2]&~qout[1]&qout[0]&(q0^~q\_1);

assign c[5]=~qout[3]&qout[2]&~qout[1]&~qout[0]&~s[1]&~s[0]

| ~qout[3]&qout[2]&~qout[1]&qout[0]&~s[1]&s[0]

| ~qout[3]&qout[2]&qout[1]&qout[0]&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&~qout[1]&~qout[0]&~a\_8&cnt[2]&cnt[1]&cnt[0]

| qout[3]&qout[2]&qout[1]&~qout[0];

assign c[4]=~qout[3]&~qout[2]&qout[1]&qout[0]&~s[1]&s[0] | qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&s[0]&~a\_8 | qout[3]&~qout[2]&~qout[1]&qout[0]&q0&~q\_1 | qout[3]&~qout[2]&qout[1]&qout[0]&~a\_8;

assign c[3]=~qout[3]&~qout[2]&qout[1]&qout[0]&~s[1]&~s[0]

| ~qout[3]&~qout[2]&qout[1]&qout[0]&~s[1]&s[0]

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&s[0]&a\_8

| qout[3]&~qout[2]&~qout[1]&~qout[0]&s[1]&s[0]&~a\_8

| qout[3]&~qout[2]&~qout[1]&qout[0]&~q0&q\_1

| qout[3]&~qout[2]&~qout[1]&qout[0]&q0&~q\_1

| qout[3]&~qout[2]&qout[1]&qout[0]&a\_8

| qout[3]&~qout[2]&qout[1]&qout[0]&~a\_8

| qout[3]&qout[2]&~qout[1]&~qout[0]&a\_8&cnt[2]&cnt[1]&cnt[0];

assign c[2]=~qout[3]&~qout[2]&qout[1]&~qout[0]&~s[1];

assign c[1]=~qout[3]&~qout[2]&~qout[1]&qout[0];

assign c[0]=~qout[3]&~qout[2]&~qout[1]&~qout[0]&start;

assign finish=~qout[3]&qout[2]&qout[1]&~qout[0]&~s[1] | qout[3]&~qout[2]&qout[1]&~qout[0];

endmodule

**✅ Top Module (ALU.v)**

`include"../RCA/RCA.v"

`include"../Control\_Unit/Control\_Unit\_Structural.v"

`include"../Registers/A.v"

`include"../Registers/Q.v"

`include"../Registers/M.v"

`include"../Registers/combinational.v"

`include"../Registers/ffd.v"

`include"../Registers/counter.v"

module ALU (

input clk,rst\_b,start,

input[1:0]s,

input [7:0]inbus,

output [7:0]outbus,

output overflow,

output finish

);

wire [7:0]q\_out,m\_out;

wire [8:0]a\_out;

wire [2:0]cnt\_out;

wire q\_1out;

wire [10:0]c;

wire [8:0]z;

wire overfl;

A regA(.clk(clk),.rst\_b(rst\_b),.q7(q\_out[7]),.sel( {c[0]|c[2]|c[3]|c[9], c[0]|c[2]|c[3]|c[6]}),.in({9{c[0]}}&9'd0 | {9{c[2]}}&q\_out | {9{c[3]}}&z),.out(a\_out));

Q regQ(.clk(clk),.rst\_b(rst\_b),.a0(a\_out[0]),.sel( {c[1]|c[9]|c[10] , c[1]|c[6]|c[10]} ),.in({8{c[1]}}&inbus | {q\_out[7:1],(~a\_out[8])}&{8{c[10]}} ),.out(q\_out),.q\_1(q\_1out));

M regM(.clk(clk),.rst\_b(rst\_b),.en(c[0]),.in(inbus),.out(m\_out));

RCA adder(.x({1'b0,m\_out} ^ {9{c[4]}}),.y(a\_out),.ci(c[4]),.z(z),.co(),.overflow(overfl));

counter COUNT(.clk(clk),.rst\_b(rst\_b),.c\_up(c[7]),.out(cnt\_out));

Control\_Unit CU(.clk(clk),.rst\_b(rst\_b),.s(s),.start(start),.q0(q\_out[0]),.q\_1(q\_1out),.a\_8(a\_out[8]),.cnt(cnt\_out),.c(c),.finish(finish));

assign outbus={1'b0,{8{c[5]}}}&a\_out | {{8{c[8]}}}&q\_out;

assign overflow=overfl & ~s[1] & c[3];

endmodule

Handles integration of all modules and signal routing.

* **Testing Phase**

Each module was verified with a **dedicated testbench**:

**Arithmetic Logic Unit’s Testbench:**

`timescale 1ns/1ps

`include "ALU.v"

**module ALU\_tb**;

reg clk,rst\_b,start;

reg[1:0]s;

reg [7:0]inbus;

wire[7:0]outbus;

wire finish;

ALU alu(

.clk(clk),

.rst\_b(rst\_b),

.start(start),

.s(s),

.inbus(inbus),

.outbus(outbus),

.finish(finish)

);

initial begin

$dumpfile("dump.vcd");

$dumpvars;

clk=0;

rst\_b=1;

start=0;

inbus=0;

end

initial begin

#10; rst\_b=0;

#20; rst\_b=1;

end

integer i;

initial begin

for(i=0;i<100;i=i+1)

begin

#50; clk=~clk;

end

#50;

end

initial begin

#180; start=1;

#80; start=0;

end

initial begin

s=2'b01;

#100;

#230; inbus=8'b00111001; // inbus=8'b00111001;//M 57

#90; inbus=8'b01010110; // inbus=8'b01010110; //Q 86

//#230; inbus=8'b10100001;//M -119 inmultire

//#90; inbus=8'b10001001;//Q -95 inmultire

//#250; inbus=8'b01011001;//M 89 inmultire

//#100; inbus=8'b10011111;//Q -97 inmultire

//#250; inbus=8'b10100011;//M -93 inmultire

//#100; inbus=8'b10001101;//Q -115 inmultire

//#250; inbus=8'b10100011;//M 13 impartire

//#100; inbus=8'b10001101;//Q 217 impartire

end

endmodule

**Control Unit’s Testbench:**

`timescale 1ns/1ps

`include "Control\_Unit.v"

module **Control\_Unit\_tb**;

reg clk,rst\_b;

reg [1:0]s;

reg start,q0,q\_1,a\_8; //q\_neg1 e q[-1],begin e start

reg [2:0]cnt;

wire [11:0]c;

wire finish;

Control\_Unit con(

.clk(clk),

.rst\_b(rst\_b),

.s(s),

.start(start),

.q0(q0),

.q\_1(q\_1),

.a\_8(a\_8),

.cnt(cnt),

.c(c),

.finish(finish)

);

initial begin

$dumpfile("dump.vcd");

$dumpvars;

clk=0;

rst\_b=0;

s=11;

start=0;

q0=0;

q\_1=1;

a\_8=0;

cnt=7;

#50;

end

initial begin

#25; rst\_b=~rst\_b;

end

initial begin

#180; start=1;

#80; start=0;

end

integer j;

initial begin

for(j=1;j<=100;j=j+1)

begin

#50; clk=~clk;

end

#50;

end

initial begin

$display("TIME\ts\tstart\tq0\tq\_1\ta\_8\tcnt\tc\t\tcurrent\_state\tnext\_state\tfinish");

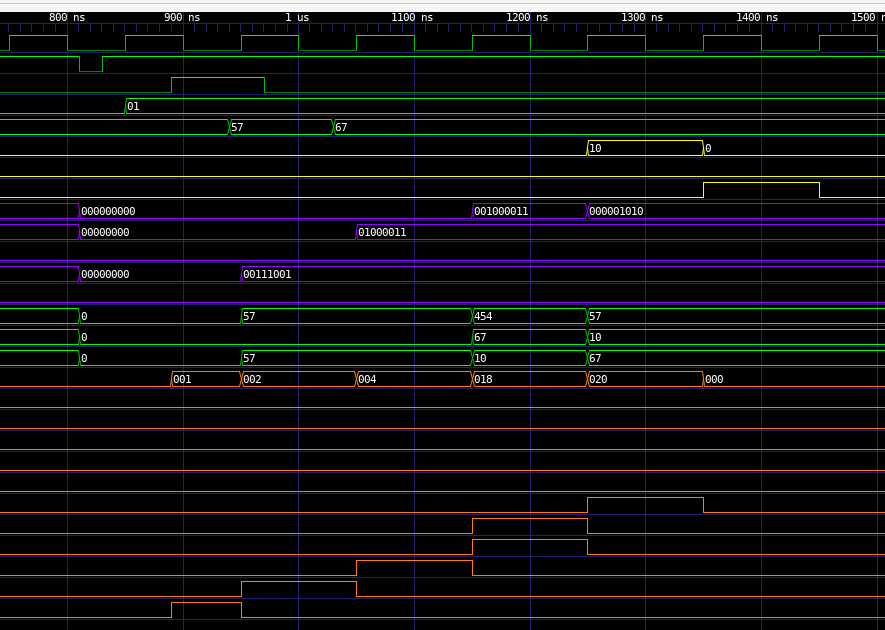
$monitor("%0t\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%d\t\t%d\t\t%b",$time,s,start,q0,q\_1,a\_8,cnt,c,con.st,con.st\_next,finish);

end

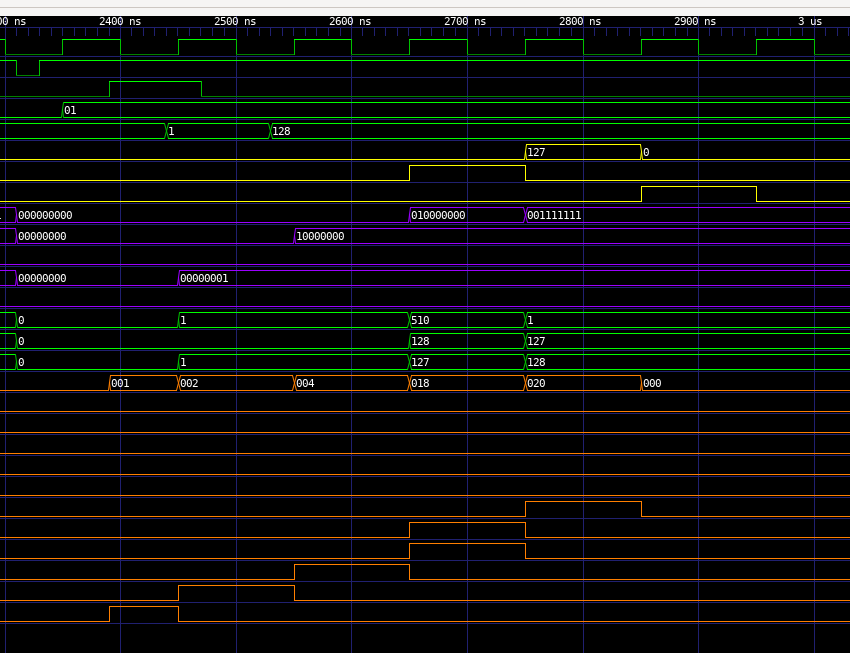
endmodule

* **Simulation Output**

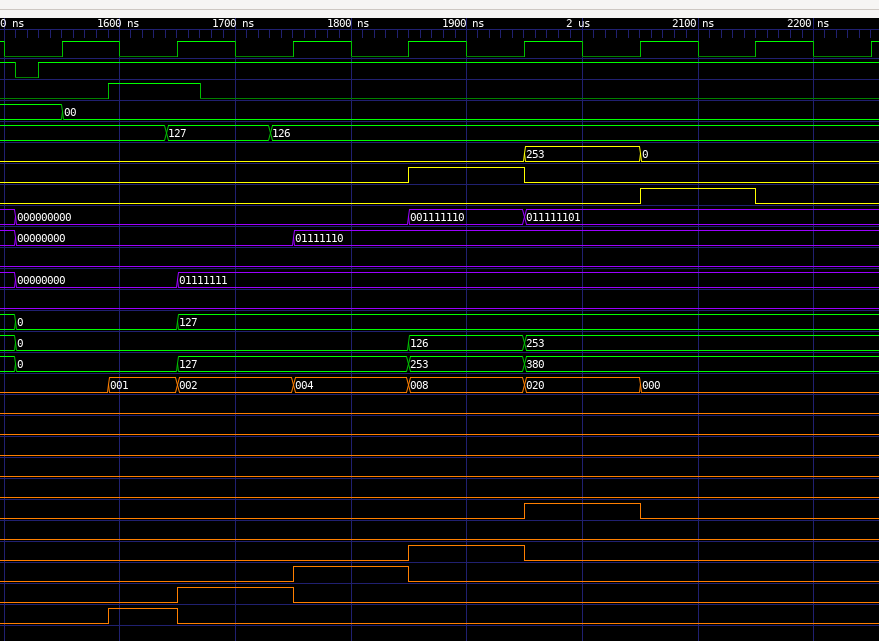
**Waveform for Subtraction** (Without Overflow):



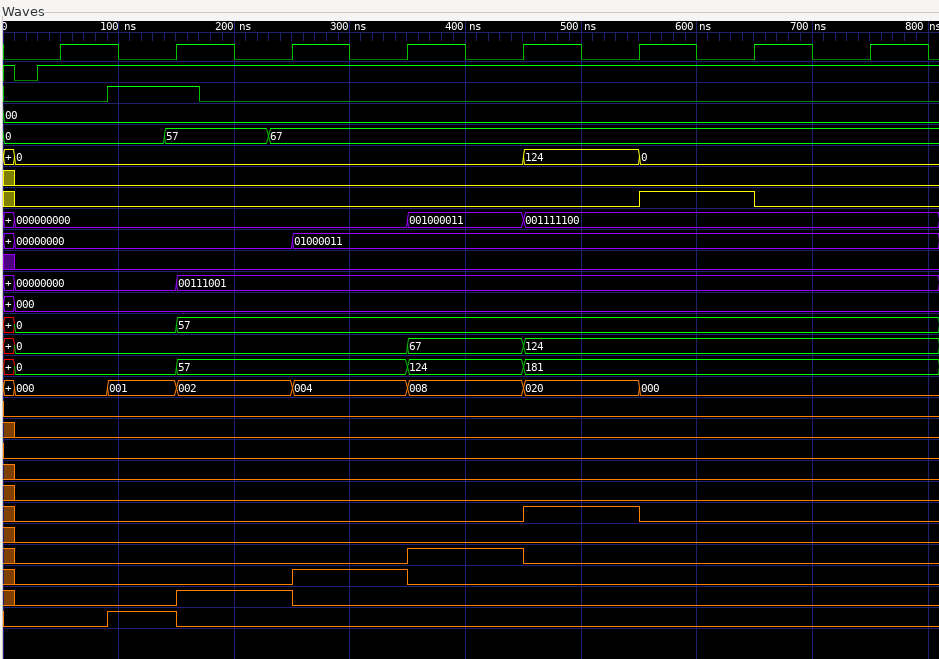
**Waveform for Subtraction** (Including Overflow):



**Waveform for Addition** (including Overflow):



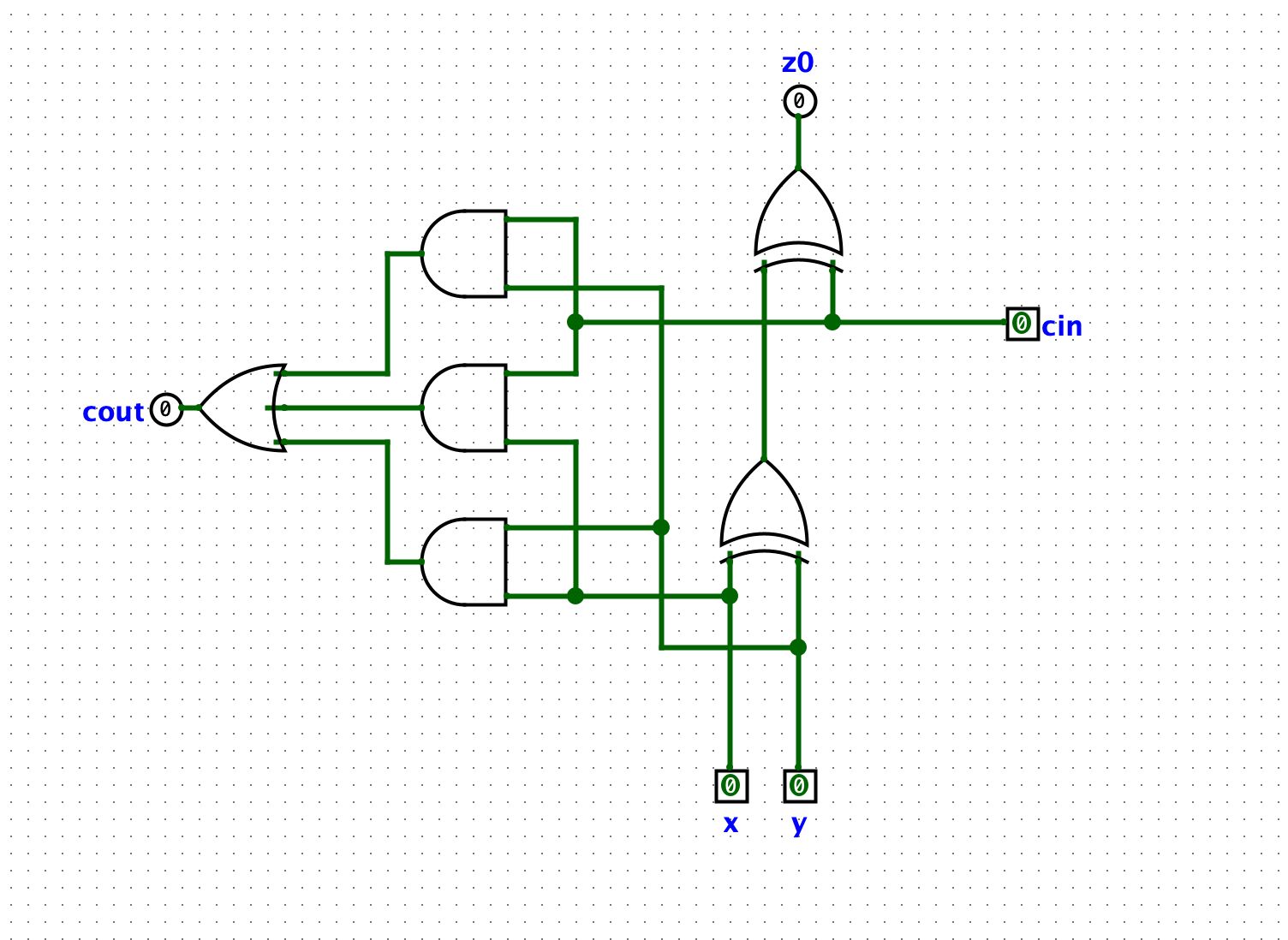
**Waveform for Addition** (without Overflow):



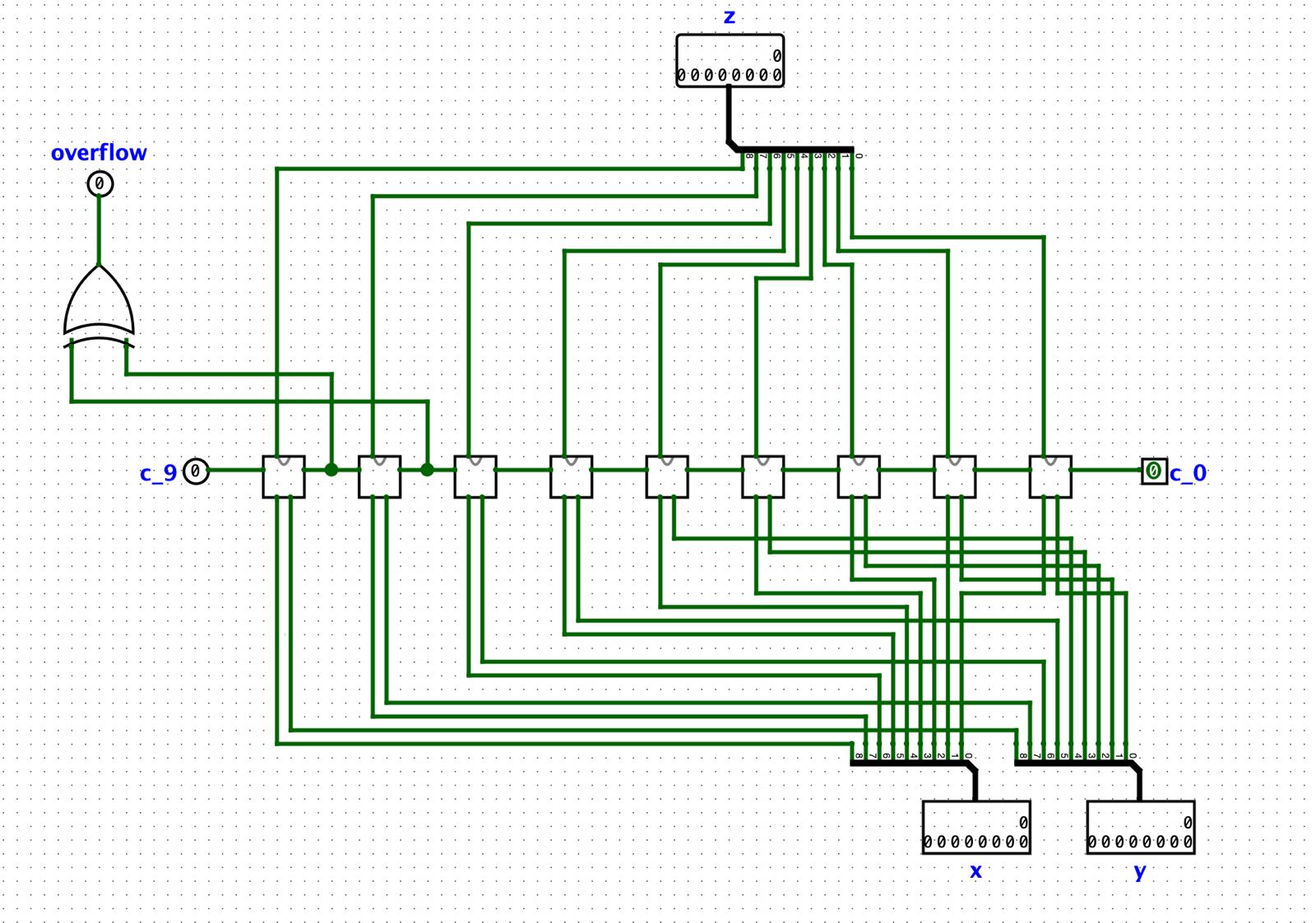
* **Logisim Architecture Diagrams**

*Include screenshots of the Logisim circuit with labeled buses and control flow.*

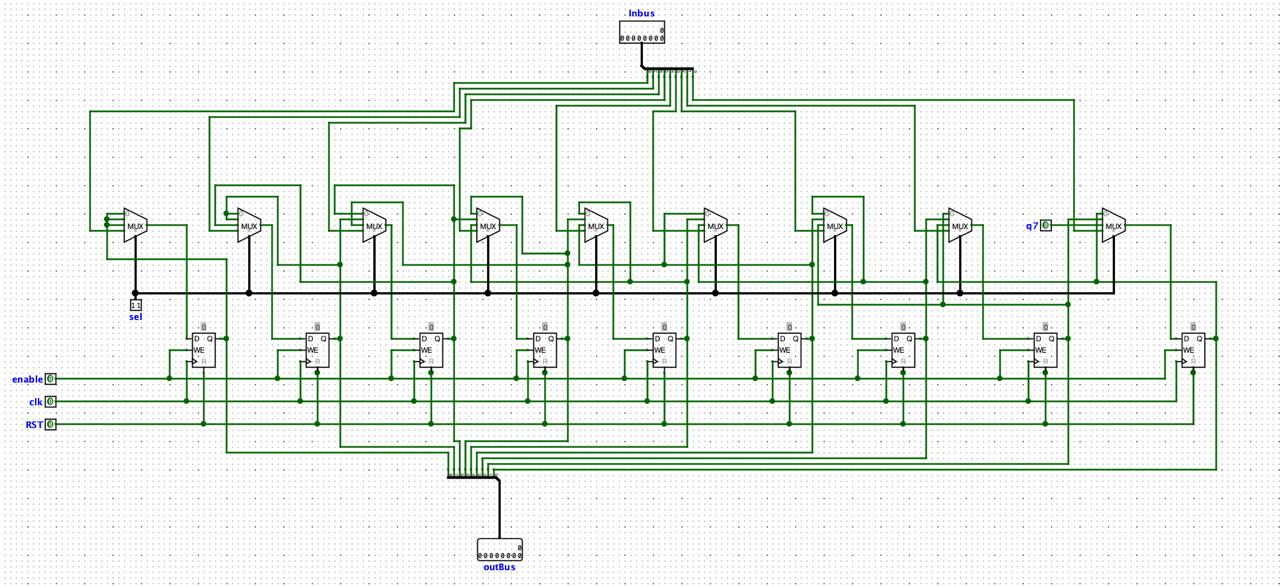
**Full Adder Cell:**



**Adder/Subtracter:**

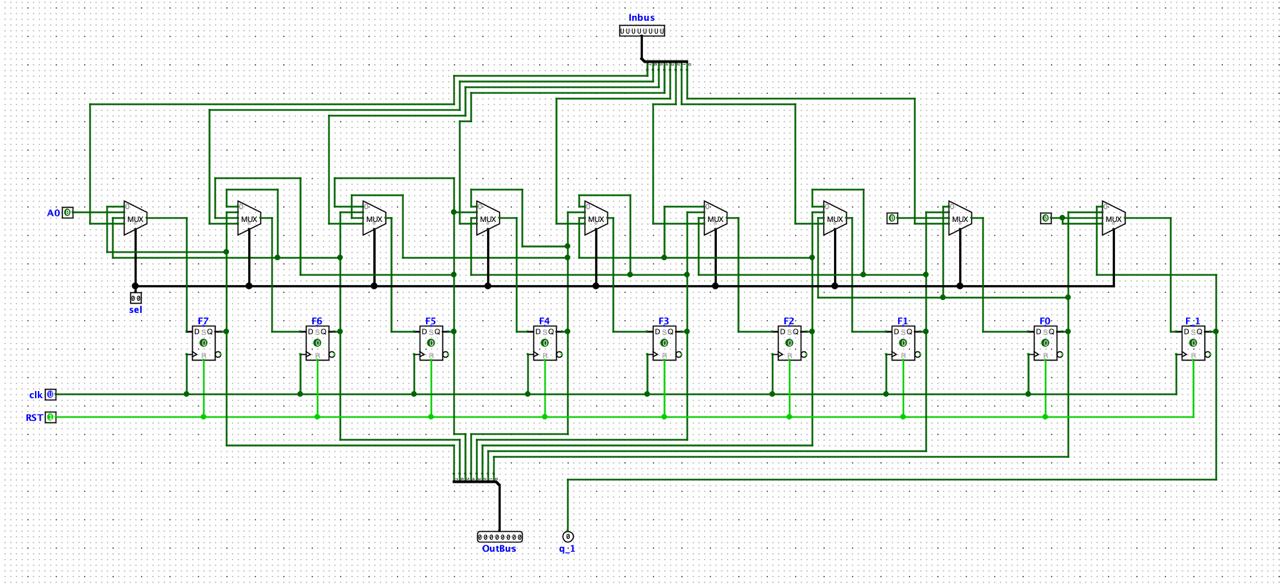


**A Register:**

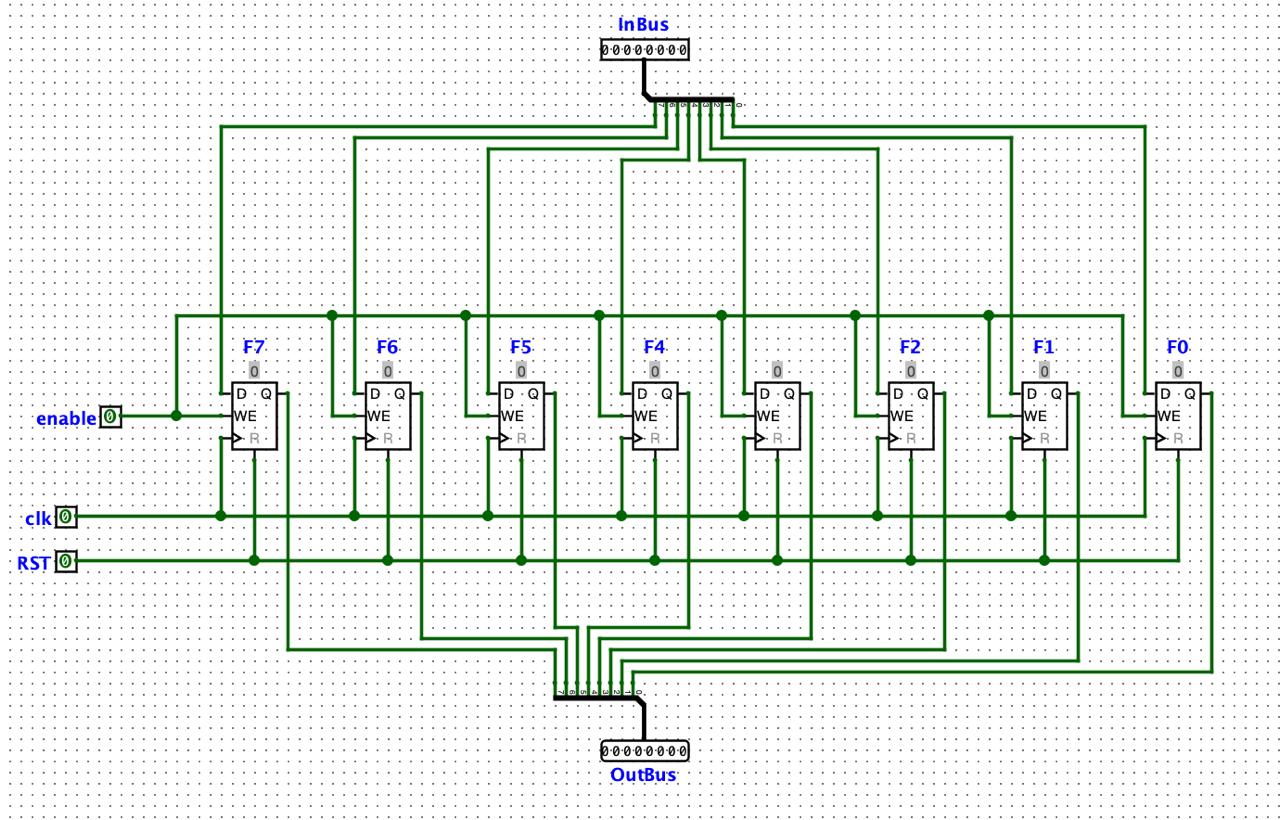


Q Register:

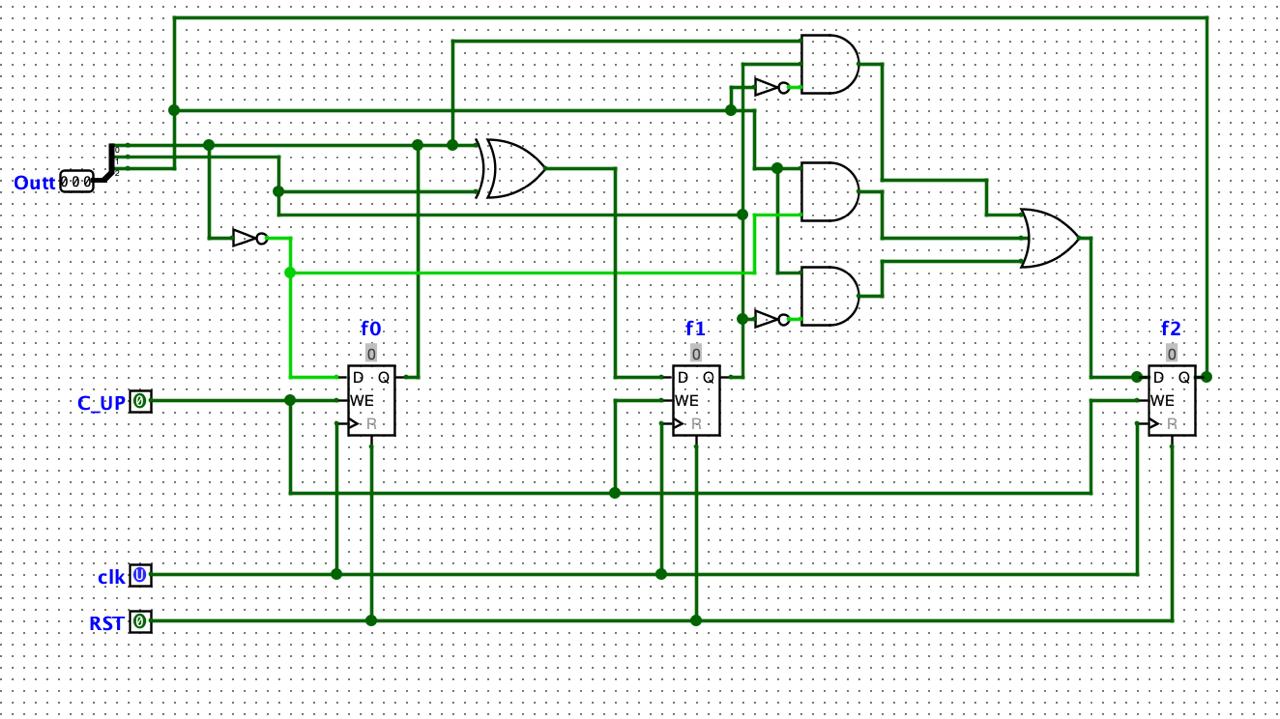
**Q Register:**



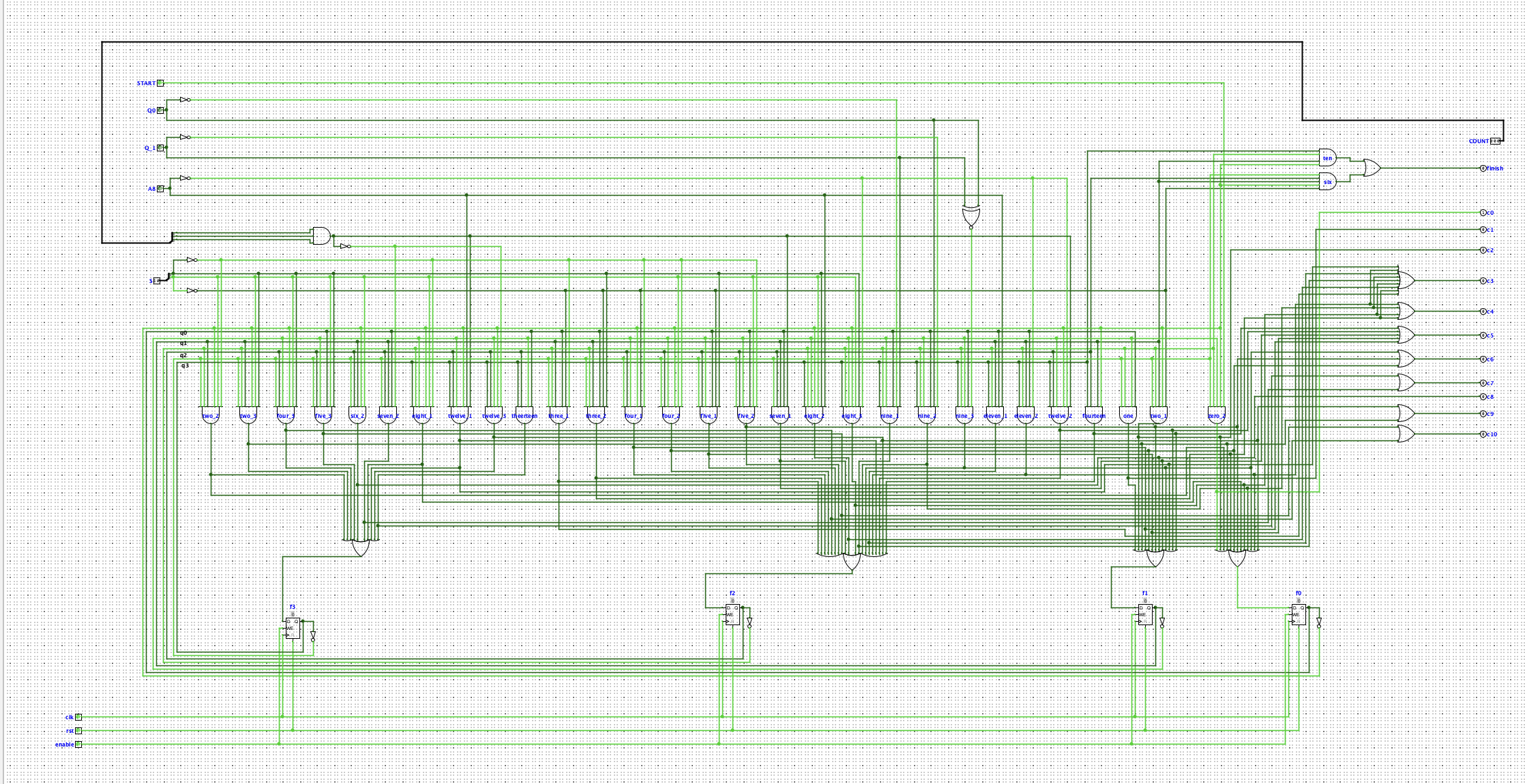
**M Register:**



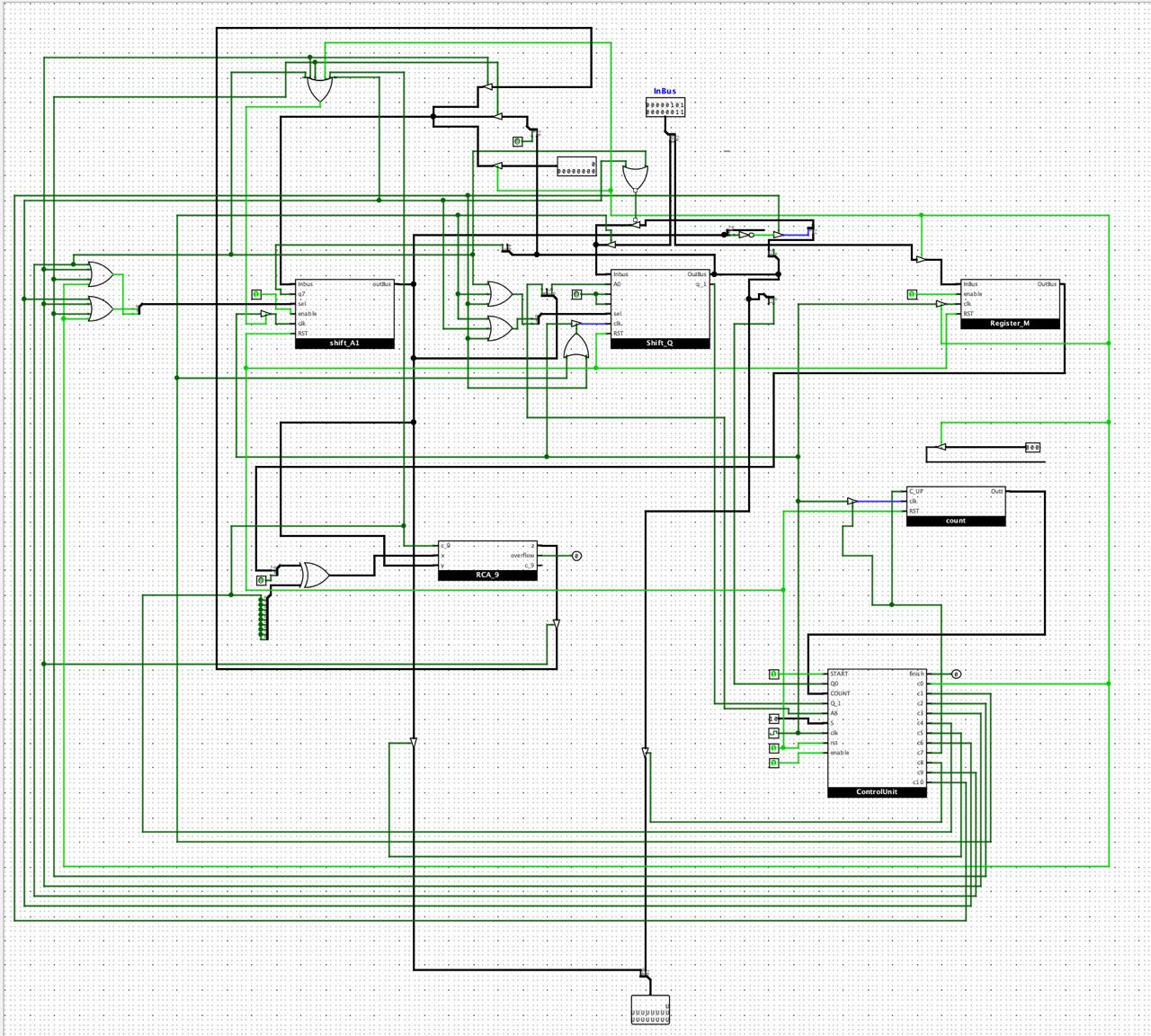
**Counter:**



**Control Unit:**



**Arithmetic Logic Unit:**



* **Challenges & Resolutions**

|  |  |
| --- | --- |
| **Challenge** | **Resolution** |
| Sign extension in Booth logic | Add arithmetic right shift and sign bit mask |
| Glitches in Logisim control (signal processing errors, oscillation apparent) | Synchronize clock with the respective signal for the specific operation |
| Verilog module order confusion | Modular top-level ALU.v for clean instancing |
| Write code without if-cases and build it structurally with logic gates | Implement the truth table and write feedback equations based on it |
| Insufficient time for writing and reading in a single clock cycle | Add one empty state for simulating clock cycle delay |

* **Conclusion**

This project successfully built and simulated a modular 8-bit ALU capable of performing key arithmetic operations. The use of structural Verilog and Logisim gave a dual-layered insight—both visual and code-based—into hardware design.