**ALU-8-BIT PROJECT**

**Documentation**

**Name: Georgescu Catalin**

**University: Politehnica Timisoara**

Obiectiv: Design and simulate an 8-bit Arithmetic Logic Unit (ALU) using a Hardware Description Language

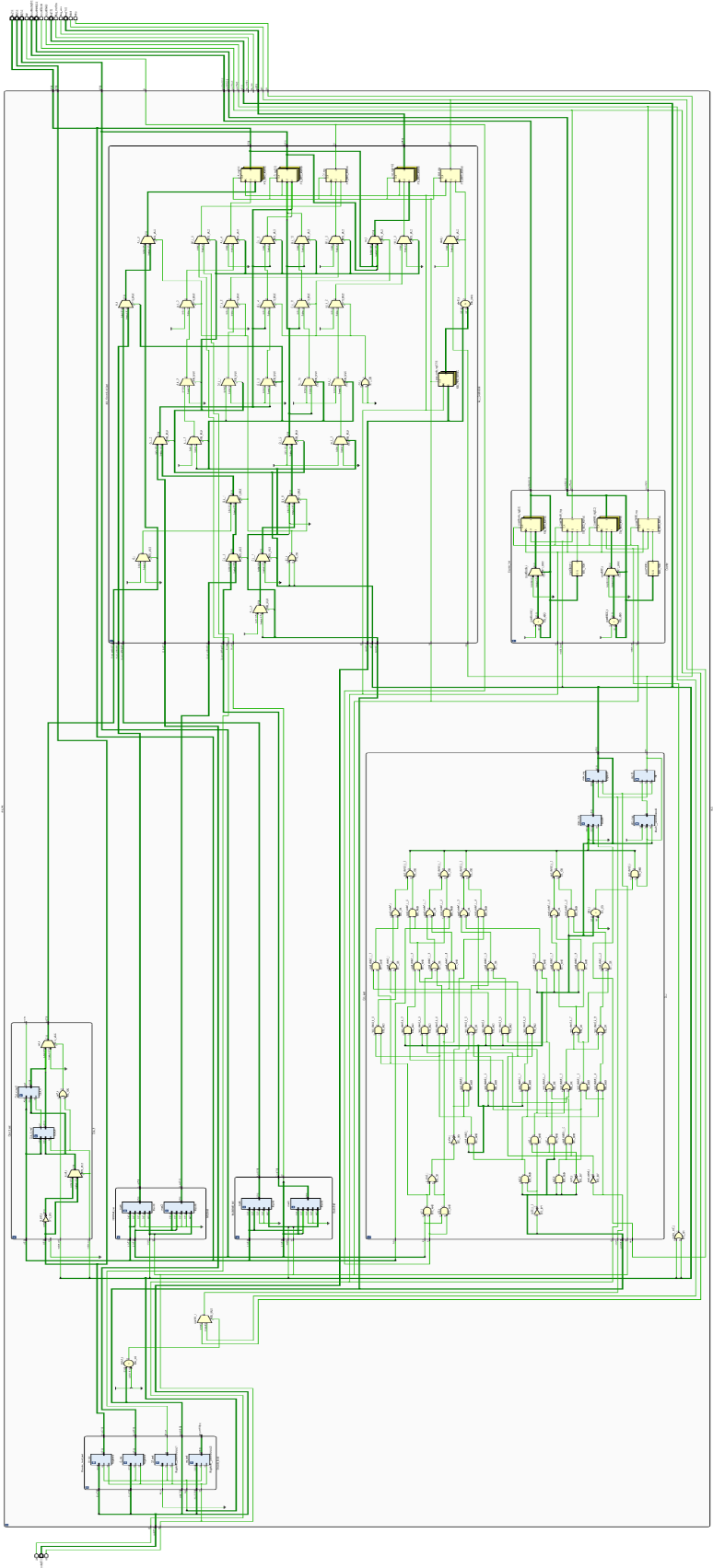
ALU must support the following arithmetic operations:

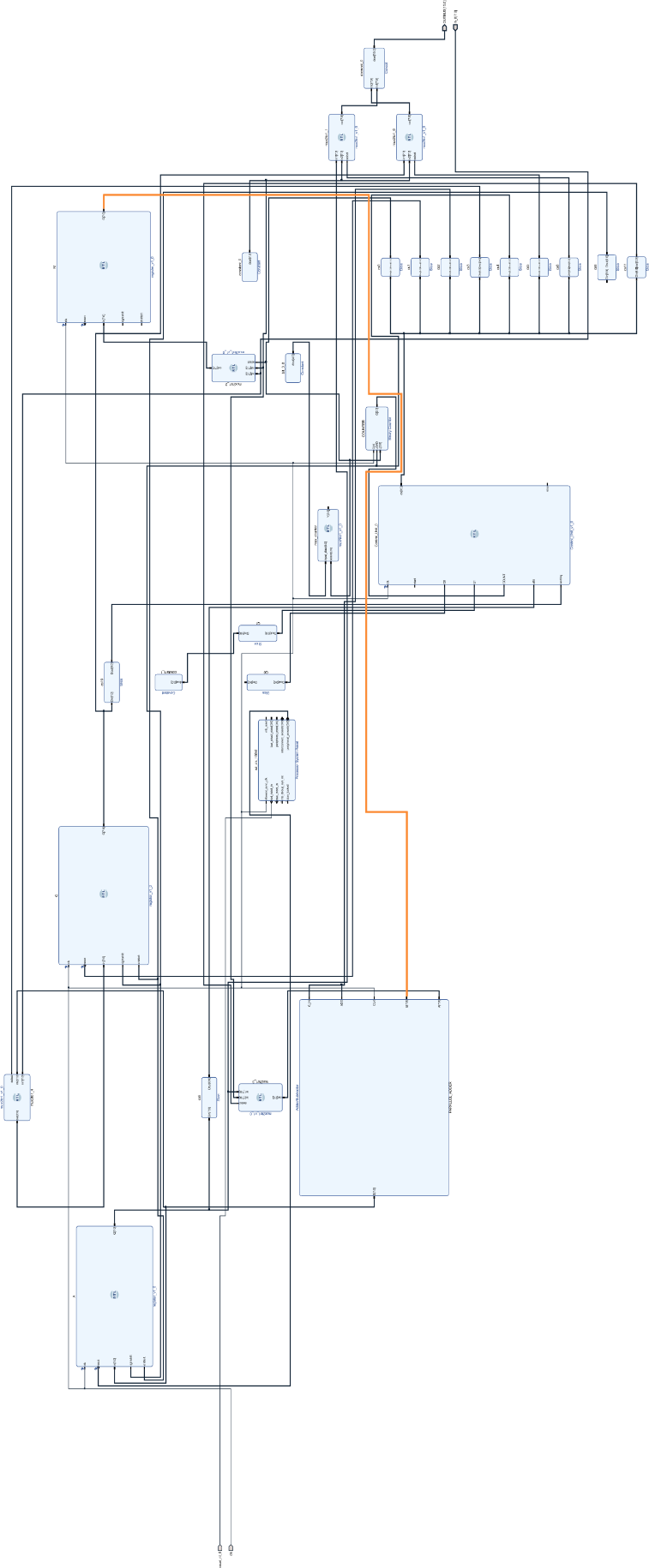
1. Addition
2. Subtraction
3. Multiplication
4. Division

Algorithms used:

* Booth Radix-2
* Non-Restoring Divison
* Carry LookAhead

Software:Xilinx Vivado

Initial Design Arhitecture:



Final Design Arhitecture:

**Objective**

The task of this project is designing and simulating an 8-BIT ALU(Arithmetic Logic Unit) in Verilog. To implement this I decided to use Vivado Xilinx as my primary software and debugging workspace. The main primary focus was emphasizing the hardware components rather than behaivoral descriptions

The main task involves:

* Addition
* Substraction
* Multiplication
* Divison

Using one of the following algorithms:

* Booth-Radix-2
* Modified Booth Algorithm
* Booth Radix-4
* Booth Radix-8
* Restoring Division
* Non-Restoring Divison
* SRT Radix-2
* SRT Radix-4

Only one of them, one algorithm for multiplication, one for division, for the addition and substraction its up to us. For my ALU implementation, I have only my parralel adder/substracter as a main operation module, I don t use dedicated modules for division/multiplication, it s all in the driven by states and the control signals, like in a real operating ALU.

**All the project files can be found here**:

[**https://github.com/Grin1234/ALU-8-bit**](https://github.com/Grin1234/ALU-8-bit)

***How it works***

You send a 18 bit code, the code if formatted like this:

* **First 8 bits: Q**
* **Next 8 bits: M**
* **Last 2 bits: opcode**

The opcode selects whitch operation needs to be done on the 2 values:

* **00 – Addition**
* **01 – Substraction**
* **10 – Multiplication**
* **11 - Division**

For example you send the code: ***000010100000010110***

The ALU Decodes the code and gets: **Q = 00001010**

**M = 00000101**

**op = 10**

After decoding, it loads the values into registers, and starts doing the operation given with the help of the Control Unit that sends signals, for my CLU I use 8 signals

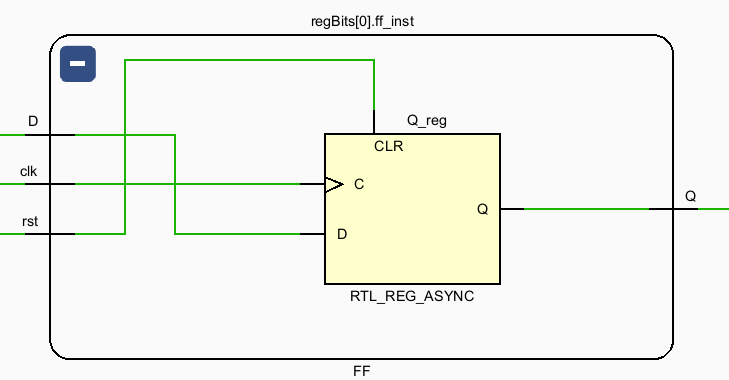
* **INIT = b00000000;**
* **LOAD = 00000010;**
* **ADD = 00000100;**
* **SUBSTRACT = 00001000;**
* **SHIFT = 00010000;**
* **OUTPUT = 00100000;**
* **WAIT\_AFTER\_LOAD= 01000000;**
* **WAIT\_AFTER\_OP = 10000000;**

**Implementation**

After finishing my initial design, I started implementing the modules, first of all, I needed that basic modules and the main adder.

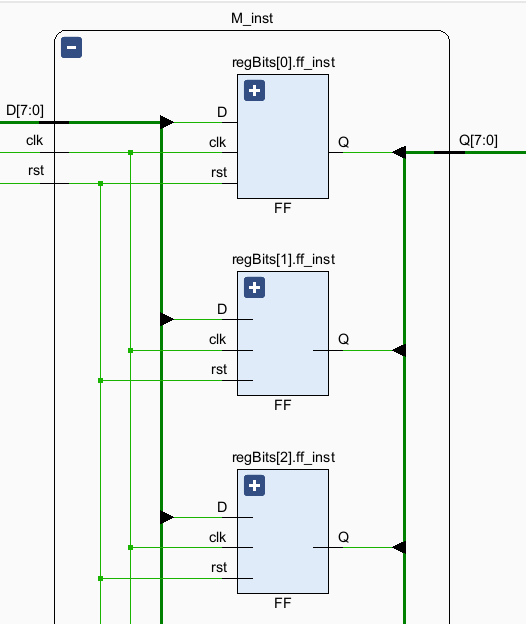
**FLIP FLOP**

The flip flop, D Latch, is the first thing I needed, for real time control using signals, I needed a register that can be easily manipulated, accesed and debugged easily. Having acces easily for each bit. This is the very little behaivoral code used in this project, I think it s quite essential.

****This is an instance of a FF in my design, D latched.

**RegisterN(Using FF)**

Using the FF above, I created a variable length register, of WIDTH length, each bit in the register is an instance of the flip flop module.



This is an register called M with the WIDTH = 8, the photo is not full but, you can see 1 bit is equal to one flip flop, in total being 8 flip flops.

**This is possible because this code sequence in my register module:**

 generate

    for(i = 0; i < WIDTH; i = i +1 ) begin: regBits

        FF ff\_inst(

            .clk(clk),

            .rst(rst),

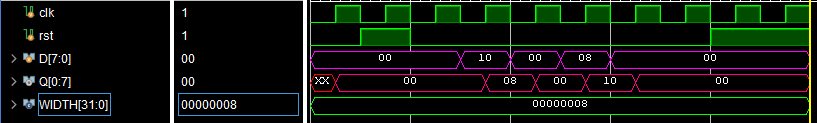
            .D(D[i]),

            .Q(Q[i])

        );

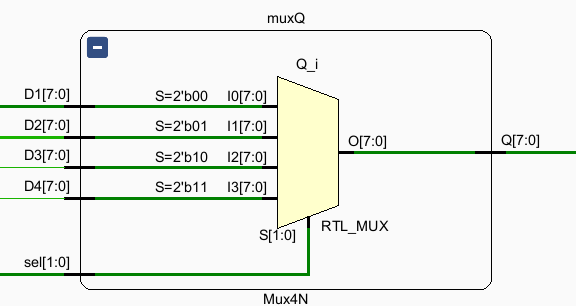
        end

 endgenerate

As you can see, even if it s 8 bits, N bits, all the values are loaded instantly.

**MUX 2N, 4N**

After implementing the basic register, I implemented a basic N-WIDTH multiplexer, this module is essential in a structural coding style, I specifically use it in the CU(Control Unit) and in the Shifting Algorithms:



This is an example used in my Shifters, I will show the full shifter diagram later below.

module Mux2N #(

    parameter WIDTH = 8

)(

    input wire sel,

    input wire [WIDTH-1:0] D1,

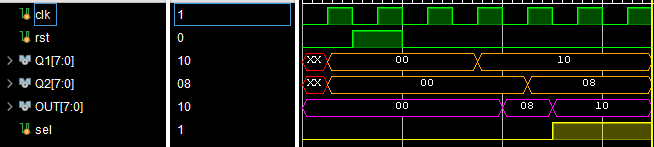
    input wire [WIDTH-1:0] D2,

    output wire [WIDTH-1:0] Q

    );

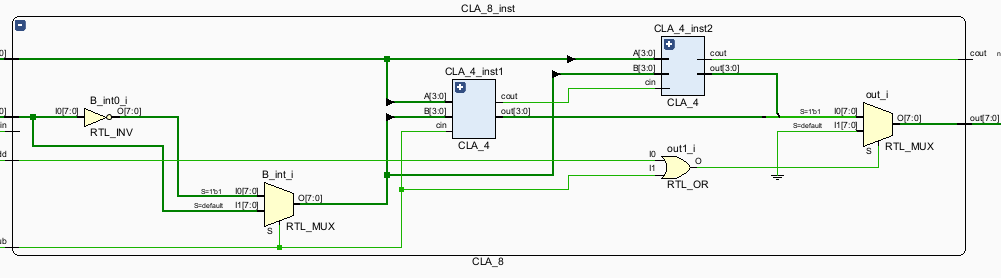
    assign Q = (sel) ? D1 : D2;

endmodule

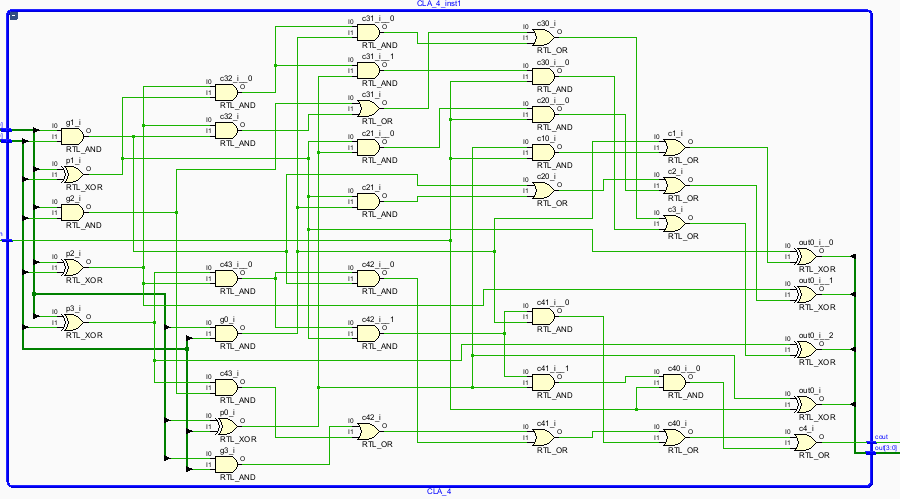


The MUX **DOESENT work on clock,** i used it for my registers, as you can see, at the rising edge of the sel = 1, it chooses Q1, if sel = 0 it choses Q2.

**Carry Lookahead Adder/Substacter(CLA)**

This is basically the main component of my ALU, for a 8 bit architecture it was the best option I could think, with more extra logic but faster in any way possible, it was perfect for my project expectations. Every operation in the ALU is using this adder/substracter(as I said I don t use specific modules for the divider/multiplier) I implemented it fully structurally using two 4 bit CLA modules.

This is the highest abstractization view of the Adder, 8 bit CLA adder, as I said it uses 2 instances of 4 bit Adders, for refrence this is how a 4 bit design looks:



This is all the gates needed to calculate the generate(g) and propagate(p) signals, in a 4 bit adder after synthesis.

assign p0 = A[0] ^ B[0],

       p1 = A[1] ^ B[1],

       p2 = A[2] ^ B[2],

       p3 = A[3] ^ B[3];

assign g0 = A[0] & B[0],

       g1 = A[1] & B[1],

       g2 = A[2] & B[2],

       g3 = A[3] & B[3];

assign c1 = g0 | (p0 & cin),

       c2 = g1 | (p1 & g0) | (p1 & p0 & cin),

       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin),

       c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & cin);

assign out[0] = p0 ^ cin,

       out[1] = p1 ^ c1,

       out[2] = p2 ^ c2,

       out[3] = p3 ^ c3;

assign cout = c4;

This is my implementation on the CLA, It’s the basic algorithm but written in Verilog.



In the testbench every result is given almost instantly, as it is asynchronous, structural design. I ve tried to make it work on a clock(to update the result at a posedge) but I had problems with timings and with control states, even if not optimal I decided to leave it as it is. In my implementation I used 2 signals enable\_add and enable\_sub that chooses between being an adder/substracter.

module CLA\_8(

    input wire [7:0] A,

    input wire [7:0] B,

    input wire cin,

    input wire enable\_add,

    input wire enable\_sub,

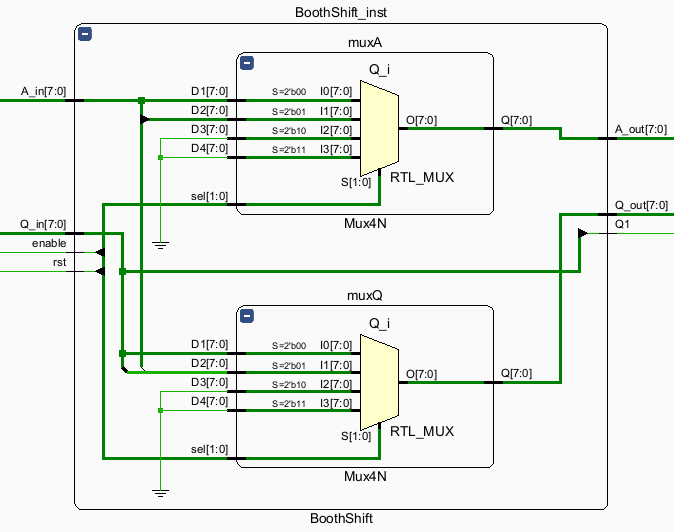
    output wire [7:0] out,

    output wire cout

);

**Shifter(Booth and NRD)**

For Booth-Radix 2 and Non-Restoring Division I needed modules for shifting, as both algorithms require it, I implemented these shifters using Multiplexer, 4N ones to be exact(4 inputs, N WIDTH).



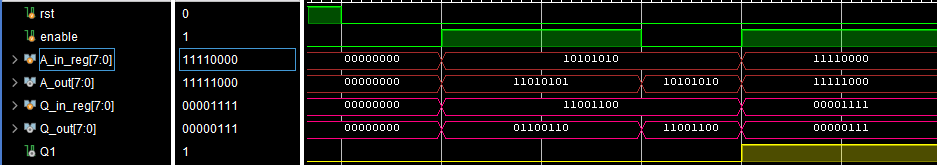
In takes 4 inputs Outputs:

* A\_in A\_out
* Q\_in Q\_out
* enable Q1(Q[-1])
* rst

In this implementation I used Multiplexer with 4 inputs, I used 2 because I have the A and Q signal, if I had 1 signal, one would be enough, for N variables, its possible to generalize to make N multiplexers(of course, if you want a structural approach)

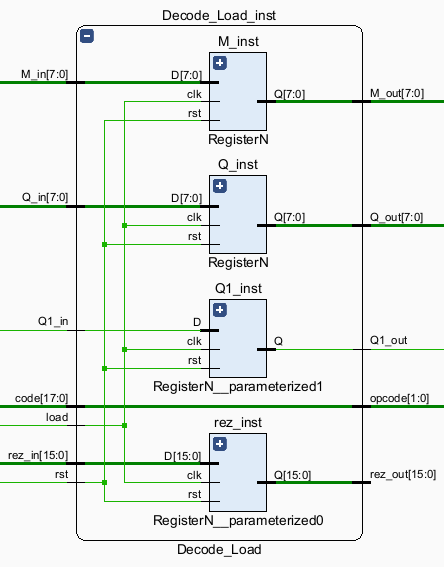
I also made a general module that only shifts 1 input, with custom serial in, custom right shift/left shift, but didn t find a use in this context, so I left it unused for now.

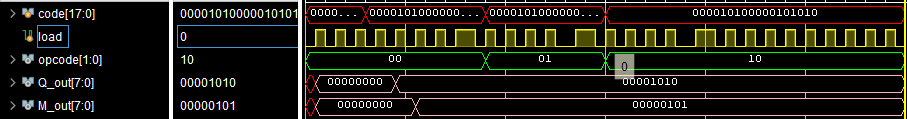
* assign Q1 = Q\_in[0];
* wire [WIDTH-1:0] shiftA = { A\_in[WIDTH-1], A\_in[WIDTH-1:1]};
* wire [WIDTH-1:0] shiftQ = { A\_in[0], Q\_in[WIDTH-1:1]};
* wire [WIDTH-1:0] zero   = {WIDTH{1'b0} };
* wire [1:0] sel = {rst, enable};
* Mux4N #
* (.WIDTH(WIDTH))
* muxA (
* .sel(sel),
* .D1(A\_in),
* .D2(shiftA),
* .D3(zero),
* .D4(zero),
* .Q(A\_out)
* );
* Mux4N #(.WIDTH(WIDTH))
* muxQ (
* .sel(sel),
* .D1(Q\_in),
* .D2(shiftQ),
* .D3(zero),
* .D4(zero),
* .Q(Q\_out)
* );

This module is asynchronous, when enable is 1 it right shifts(I used Booth-Radix 2 shift for this example, it also gives as output Q[-1] which is used in the next booth multiplication step

**Decoder\_Loader**

This is the module that decodes the code given to the ALU, check out page 4 for more information, its given an 18 bit code, its main task Is to separate the values and opcode(operation code), also to load the values into registers(made of FF).

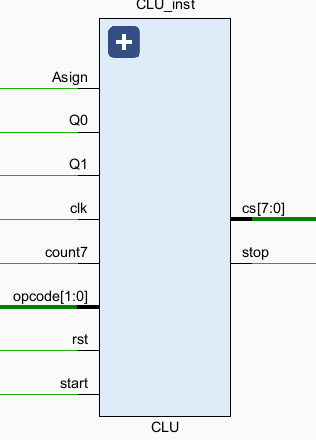


****

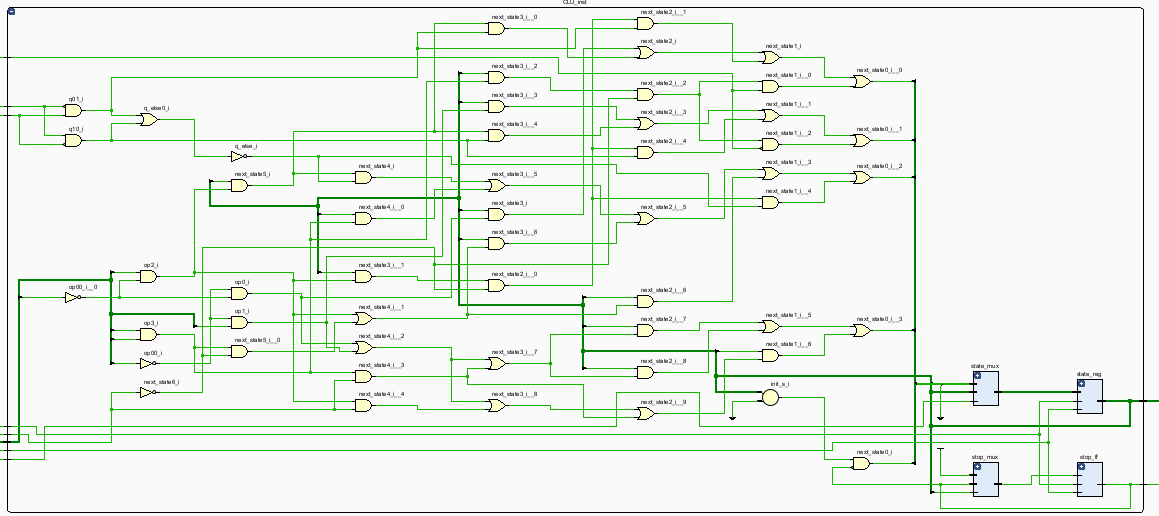
In the testbench we receive a 18 bit code, a load signal that is frequent(only for test purpose) the code changes its opcode 4 times, and the opcode updated, also the Q, M are staying the same because only the opcodes change, they are loaded at the beginning.

**Control Logic Unit(CLU)**

This is the brain of the ALU, this module is the most essential part to make the whole ALU work, its task is to provide control signals based on its current inputs, current state, etc. \*for more information on the control states check out page 4\*.



It takes as inputs the necessary signals to provide the correct control signals, Q0 Q1 to decide the next step in Booth Algorithm, count7 that signals when to stop, the opcode that gives the operation code, etc. As output it has a stop and an 8 bit signal that is coded in the on hot mode(only 1 bit active), the control signals(cs) are inputs in various modules(in the Adder/Substracter, in the Loader, Shifter, Counter etc), everything is done with a order.

For reference, everything is done structurally, so this is how the module looks at the lowest level of abstractization:

It uses the custom made register to update the state, and also using multiplexers for decision making.

  wire init\_s    = (state == INIT);

  wire load\_s    = state[1];

  wire add\_s     = state[2];

  wire sub\_s     = state[3];

  wire rsh\_s     = state[4];

  wire wait\_ld\_s = state[6];

  wire wait\_op\_s = state[7];

  wire op0 = ~opcode[1] & ~opcode[0];  // 00 - adunare

  wire op1 = ~opcode[1] &  opcode[0];  // 01 - scadere

  wire op2 =  opcode[1] & ~opcode[0];  // 10 -inmultire

  wire op3 =  opcode[1] &  opcode[0];  // 11 -impartire

  wire q01    = ~Q0 &  Q1;  // {Q0,Q1} - 01 - adunare

  wire q10    =  Q0 & ~Q1;  // {Q0,Q1} - 10 - scadere

  wire q\_else = ~(q01 | q10);

Every state has its condition when to be activated(this is just for my implementation that uses Booth Radix 2 and NRD), what is useful is that using just the ADDER, I can make multiplication and division using thsese states, both Booth and NRD use almost the same operations with the main difference being the way its shifted, but that s not a problem for the CLU(it just sends the shift signal, it doesen t care about these things)

  assign next\_state[0] = 1'b0;

  assign next\_state[1] = init\_s & ~stop;

  assign next\_state[6] = load\_s;

  assign next\_state[2] = (wait\_ld\_s & op0) | (wait\_ld\_s & op2 & q01) | (wait\_op\_s & op2 & ~count7 & q01) | (wait\_op\_s & op3 & ~count7 & Asign);

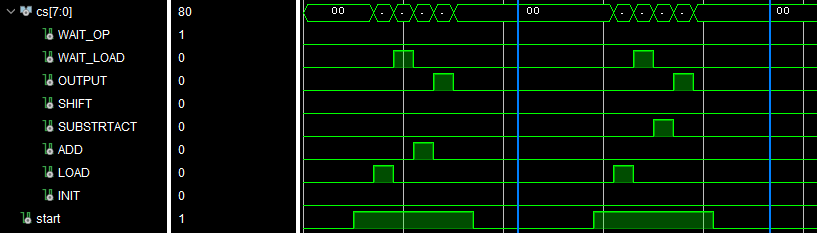
  assign next\_state[3] =(wait\_ld\_s & op1)| (wait\_ld\_s & op2 & q10)| (wait\_op\_s & op2 & ~count7 & q10)| (wait\_op\_s & op3 & ~count7 & ~Asign);

  assign next\_state[4] = (wait\_ld\_s & op2 & q\_else) | (wait\_ld\_s & op3) | (add\_s & (op2 | (op3 & ~count7)))| (sub\_s     & (op2 | (op3 & ~count7))) | (wait\_op\_s & op2 & ~count7 & q\_else);

  assign next\_state[7] = rsh\_s;

  assign next\_state[5] = (add\_s & (op0 | op1 | (op3 & count7)))| (sub\_s  & (op0 | op1 | (op3 & count7))) | (wait\_op\_s & (op0 | op1 | (op2 & count7) | (op3 & count7)));

Using this every operation is possible, also it can be extended to do many operations like to simulate gates, bitwise operations etc, the only thing that needs to be updated are the opcode size and some of the states.



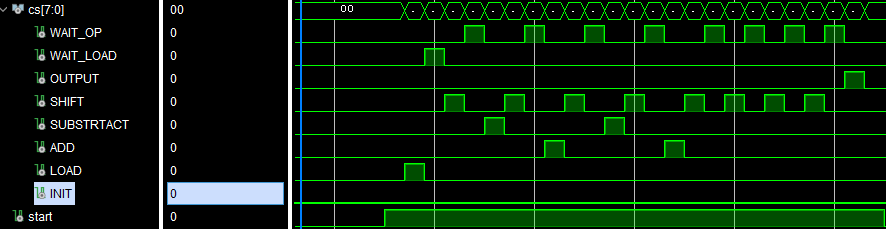
This is an example testbench, for instance, lets look at the graph till the first we reach the first marker, the control signals given are in this order.

* LOAD
* WAIT\_LOAD
* ADD
* OUTPUT;

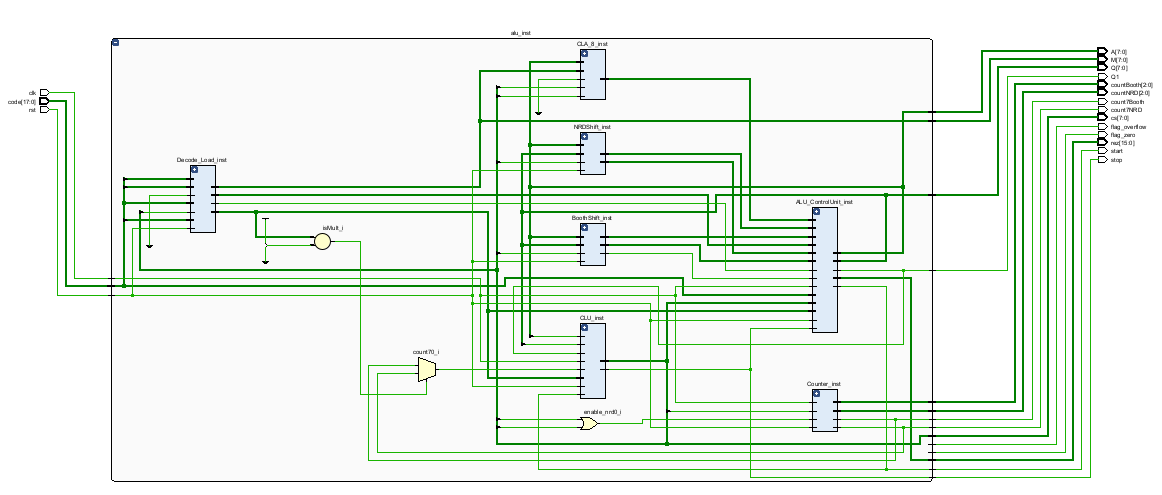
This is just the control signals given for basic addition. If we look from the first marker to the second marker we see almost the same thing but instead of addition we have substraction.

Every state is updated at the rising edge of the clock, for the basic addition or substraction or addition it uses about **5 clock periods**, and about **60 ns**

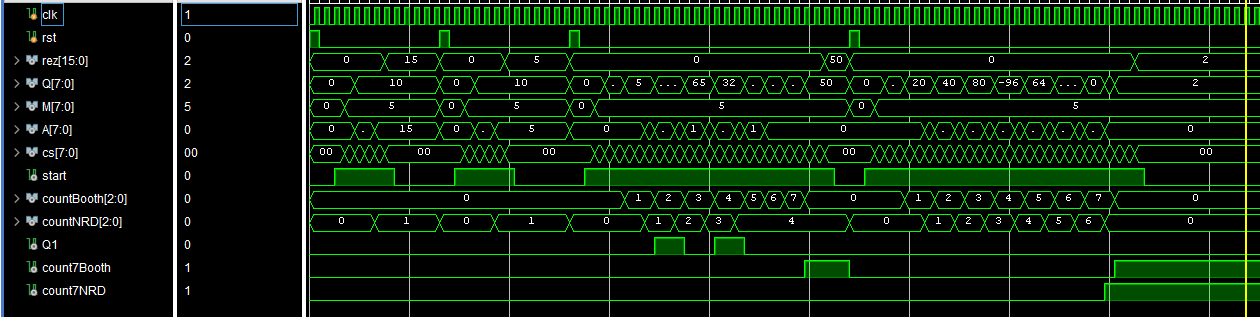
I know its quite long, I struggeled with the timing, this time includes the time it takes to load the variables and to output them too.

This is for multiplication, it has more states, including shifting, addition, substraction, etc, it takes 8 steps to finish(because we work with 8 bits variables), it takes about **300 ns** to finish calculating, of course considering I run a 50 mhz clock its decent, but not quite optimal.

**Arithmetic Logic Unit(ALU)**

This unit holds everything together, the CLU(Control Logic Unit), the Adder(CLA), the Shifters, Counters.

If you put everything together of what I presented till now, this is how it will look(at the highest level of abstractization)



In this testbench I test the addition/substraction/multiplicated/division. Together with the states it will look something like this:

