

ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

First semester 2020/2021

Digital systems ENCS234

ALU Verilog Project

**Name : Ayham Maree ID:1191408**

**Doctor : Hanna Bullata Section : 5**

**ALU:**

An arithmetic logic unit (ALU) is a combinational digital

Electronic circuit that performs arithmetic and bitwise

Operations on integer binary numbers. This is in contrast

To a floating-point unit (FPU), which operates on floating

Point numbers. An ALU is a fundamental building block of

Many types of computing circuits, including the central

Processing unit (CPU) of computers, FPUs, and graphics

Processing units (GPUs). A single CPU, FPU or GPU may

Contain multiple ALUs.

Components:

We used many components in our project to do some functions; ***FA\_8BIT*** which do four functions, addition, subtraction, equality checking, and less-than checking; Reminder which calculates the ***Reminder***; ***BITWISE\_and*** which gives us the result of AND operation for each bit; ***BITWISE\_or*** that gives us the result of OR operation for each bit; ***Concatenate*** to manage the digits order; and other components like ***Mux8\_1*** multiplexer and other gates.

FA (Full Adder) :

**FA** circuit has three inputs, X, Y, and C\_in, it calculates

the summation of them and set it as output (***Sum***), then set the

reminder in C\_out output.

**CODE:**

module FA(Sum, C\_out, X, Y, C\_in);

input X, Y, C\_in;

output Sum, C\_out;

wire xoW, anW1, anW2; //xoW: XorWire //anW:AndWire

xor G0(xoW, X, Y); //G: gate

and G1(anW1, X, Y);

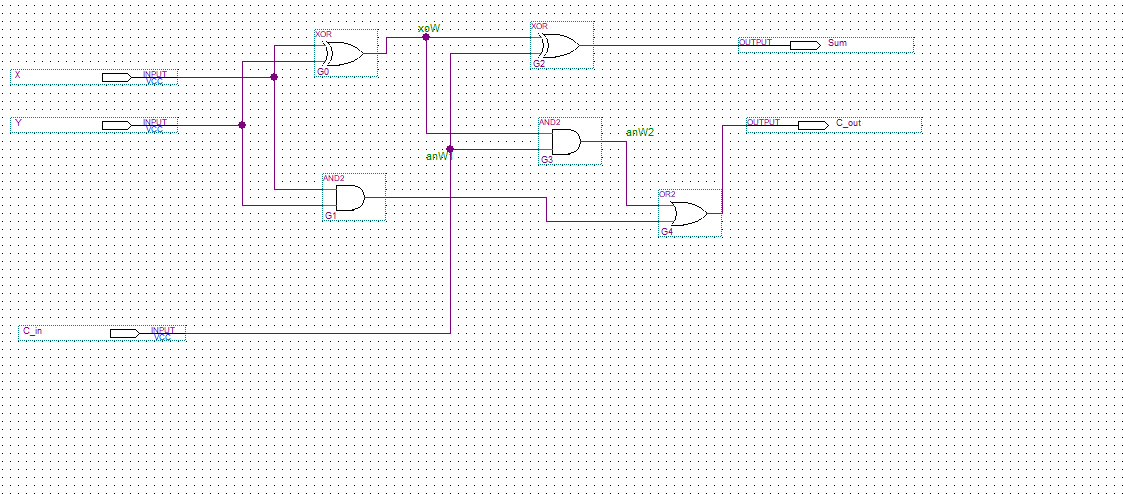
xor G2(Sum, xoW, C\_in);

and G3(anW2, xoW, C\_in);

or G4(C\_out, anW1, anW2);

endmodule

**Block Diagram :**

****

**Simulation:**

****

**FA\_8BIT (Full Adder 8 Bit);**

This circuit performs four functions: **addition**, **subtraction**,

**Equality** checking, and **less** -than checking.

Addition:

Adding two 8-bit binary numbers using full adders. This function outputs one 8-bit output (**sum**) and two 1-bit outputs (**C\_out**: Carry, **OV**: Overflow).

Subtraction:

Subtracting two 8-bit binary numbers using full adders and xor gates that change the number to its 2’s complement. This function outputs one 8-bit output (**sum**) and two one-Bit outputs (**C\_out: Carry, OV: Overflow**).

Equality:

This function checks if the two inputs (**X, Y**) equal or not by making subtracting operation on the inputs, if the result is zero, the numbers are equal. FA\_8BIT checking if the result zero using nor gate, this gate has 8 inputs (The eight bits of the result of subtracting).

Less:

This function checks if the first input (**X**) is less than the second input (**Y**) by making Subtracting operation, if the carry out (**C\_out**) is zero, then X is less than Y. The result of this function is simply the opposite of (**C\_out).**

**CODE:**

module FA\_8BIT (Equal, Less, Sum, C\_out, OV, A, B, C\_in);

input [7:0] A, B;

input C\_in;

output OV, C\_out, Equal, Less;

output reg [7:0] Sum;

wire [7:1] c;

wire [7:0] s, bb;

genvar i;

generate

for(i=0; i<8; i=i+1)

begin: AYHAM

xor (bb[i], C\_in, B[i]);

end

FA(s[0], c[1], A[0], bb[0], C\_in);

for(i=1; i<7; i=i+1)

begin: FA

FA(s[i], c[i+1], A[i], bb[i], c[i]);

end

FA(s[7], C\_out, A[7], bb[7], c[7]);

xor G1(OV, c[7], C\_out);

not G2(Less, C\_out);

nor G3(Equal, s[7], s[6], s[5], s[4], s[3], s[2], s[1], s[0]);

always @(s) Sum = s;

endgenerate

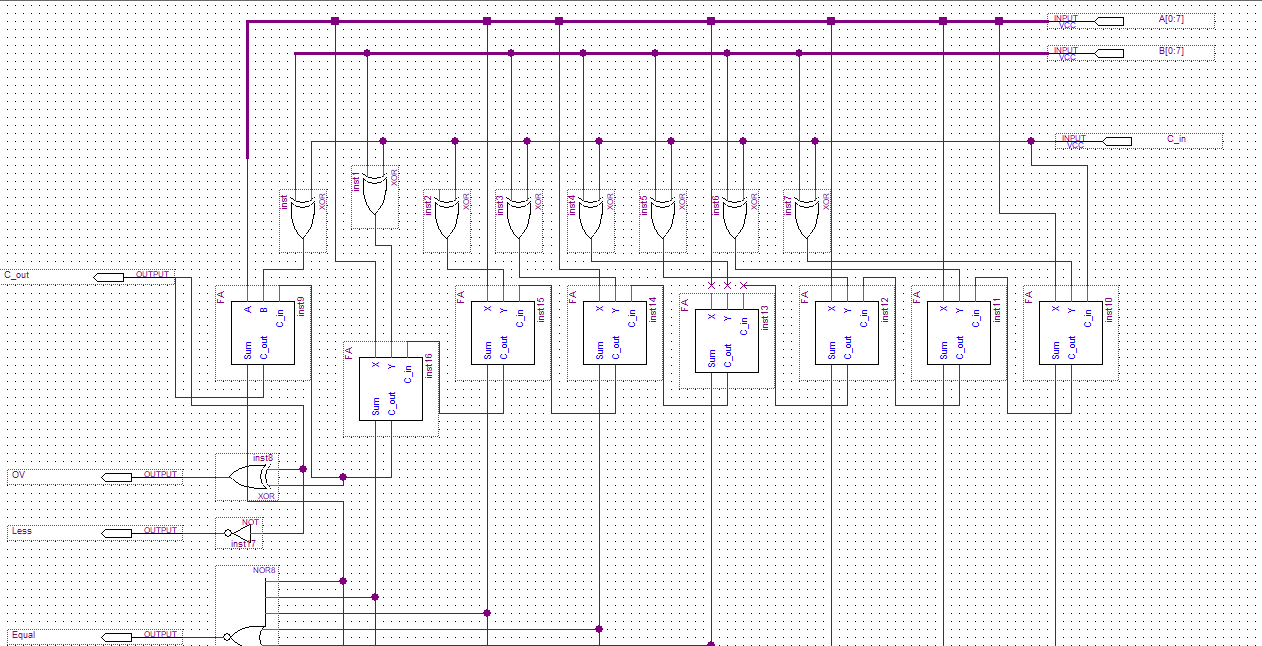
endmodule

Block Diagram:

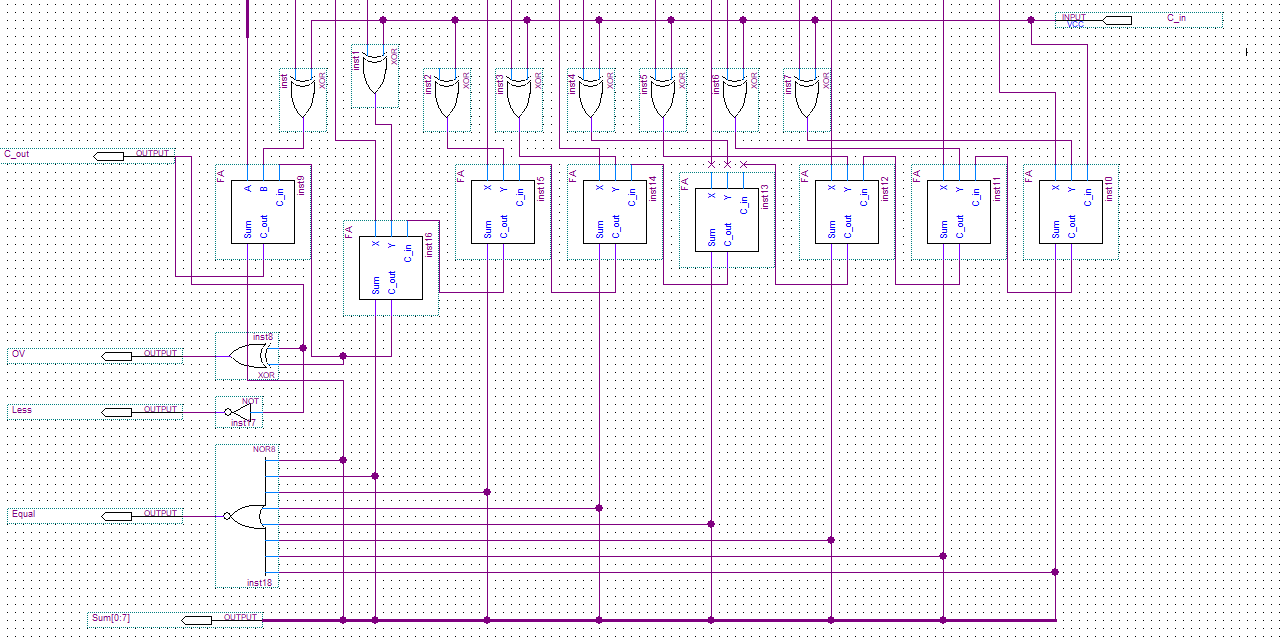
***ملاحظة :***

**(تم تقسيم الصورة الى جزئين بسبب عدم اتساع مساحة التصوير لكنها مكملة لبعضها البعض)**

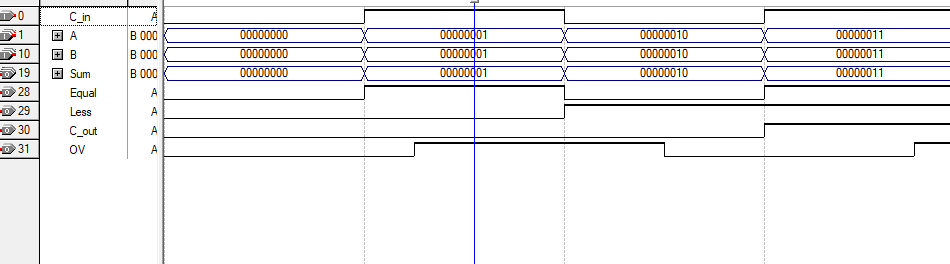
**PART ONE:**



**PART TWO:**



Simulation:



Reminder:

Reminder block has two inputs, X and Y, it calculates the "left

Over" number after dividing one binary number by another

And set it as output (W).

CODE:

module Reminder(W, X, Y);

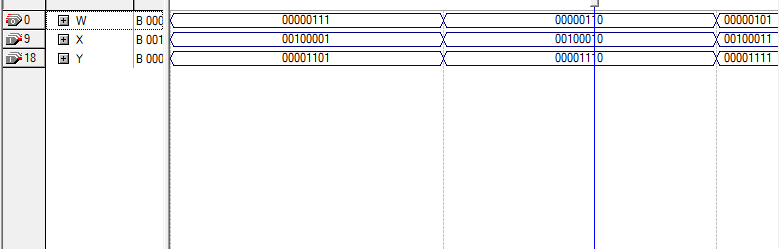
input [7:0] X, Y;

output reg [7:0] W;

always @(X, Y) W = X % Y;

endmodule

Simulation:



BITWISE\_and:

This block has two 8-bit inputs (X and Y), and one output (Z),

The ***BITWISE\_and*** circuit check X and Y bit by bit, if they are

Both one, the result of this bit in the Z output will be one,

Otherwise, it will be zero.

CODE:

module BITWISE\_and(Z, X, Y);

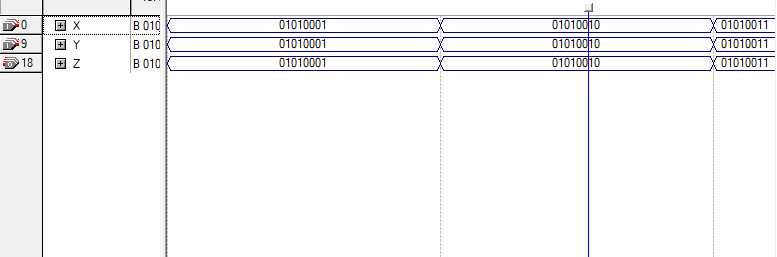
input [7:0] X, Y;

output [7:0] Z;

assign Z = X & Y;

endmodule

Simulation:



BITWISE\_or:

This block has two 8-bit inputs (X and Y), and one output (Z), the

***BITWISE\_or*** circuit check X and Y bit by bit, if one of them is one,

The result of this bit in the Z output will be one, otherwise, it will

Be zero.

CODE:

module BITWISE\_or(Z, X, Y);

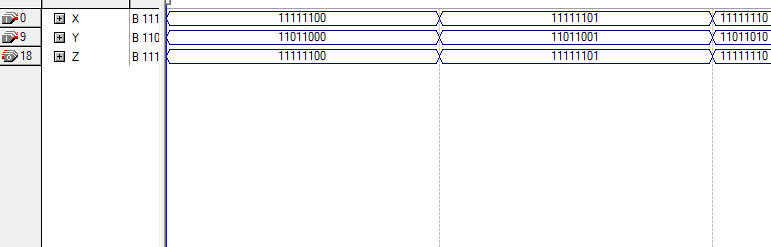
input [7:0] X, Y;

output [7:0] Z;

assign Z = X | Y;

endmodule

Simulation:



Concatenate:

This block has one 8-bit output (Z), depends on the less significant

Four bits from both X and Y. in other words, the less significant four

Bits of Z is the less significant four bits from Y, and the high significant

Four is from X.

CODE:

module Concatenate(Z, X, Y);

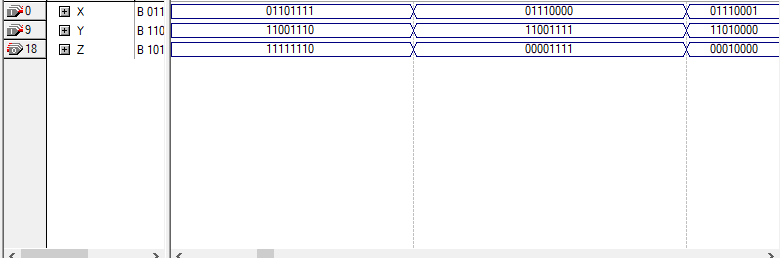
input [7:0] X, Y;

output reg [7:0] Z;

always @(X, Y) {Z} = {X[3:0], Y[3:0]};

endmodule

Simulation:



Mux8\_1 (Multiplexer 8\*1):

This multiplexer has six 8-bit inputs, one 3-bit input, and two

1-bit inputs, and has one 8-bit output. The output has value

Same as one of the 8-bit or 1-bit inputs depending on the

Selection input (the 3-bit input).

CODE:

module Mux8\_1(Z, C, Adder, Subt, Remi, And, Or, Concat, Equal, Less);

input [7:0] Adder, Subt, Remi, And, Or, Concat;

input Equal, Less;

input [2:0] C;

output reg [7:0] Z;

always @(Adder, Subt, Remi, And, Or, Concat, Equal, Less, C)

case(C)

0: Z=Adder;

1: Z=Subt;

2: Z=Remi;

3: Z=And;

4: Z=Or;

5: Z=Concat;

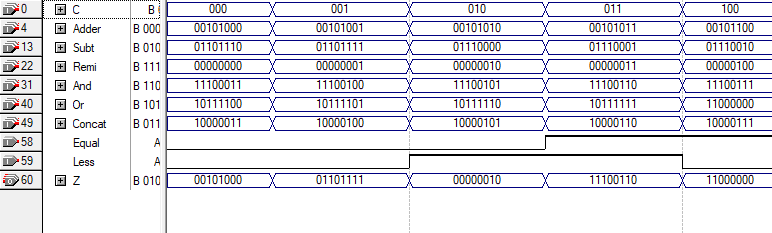
6: Z=Equal;

7: Z=Less;

endcase

endmodule

Simulation:



MYALU (ALU):

ALU is the block that collects all of our project components, it

Has two 8-bit inputs that the operations depend on, and one

3-bit input which determines the operation, Also it has one 8-

Bit output and two 1-bit outputs.

CODE:

module MYALU(Z, OV, C\_out, C, X, Y);

input [7:0] X, Y;

input [2:0] C;

output OV, C\_out;

output [7:0] Z;

wire C\_o, OV\_o, EnableControl, orC\_in;

wire [7:0] Sum, Equal, Less, Remi, And, Or, Concat;

FA\_8BIT (Equal, Less, Sum, C\_o, OV\_o, X, Y, orC\_in);

Reminder (Rem, X, Y);

BITWISE\_and (And, X, Y);

BITWISE\_or(Or, X, Y);

Concatenate (Concat, X, Y);

or G0(orC\_in, C[1], C[0]);

nor G1(EnableControl, C[2], C[1]);

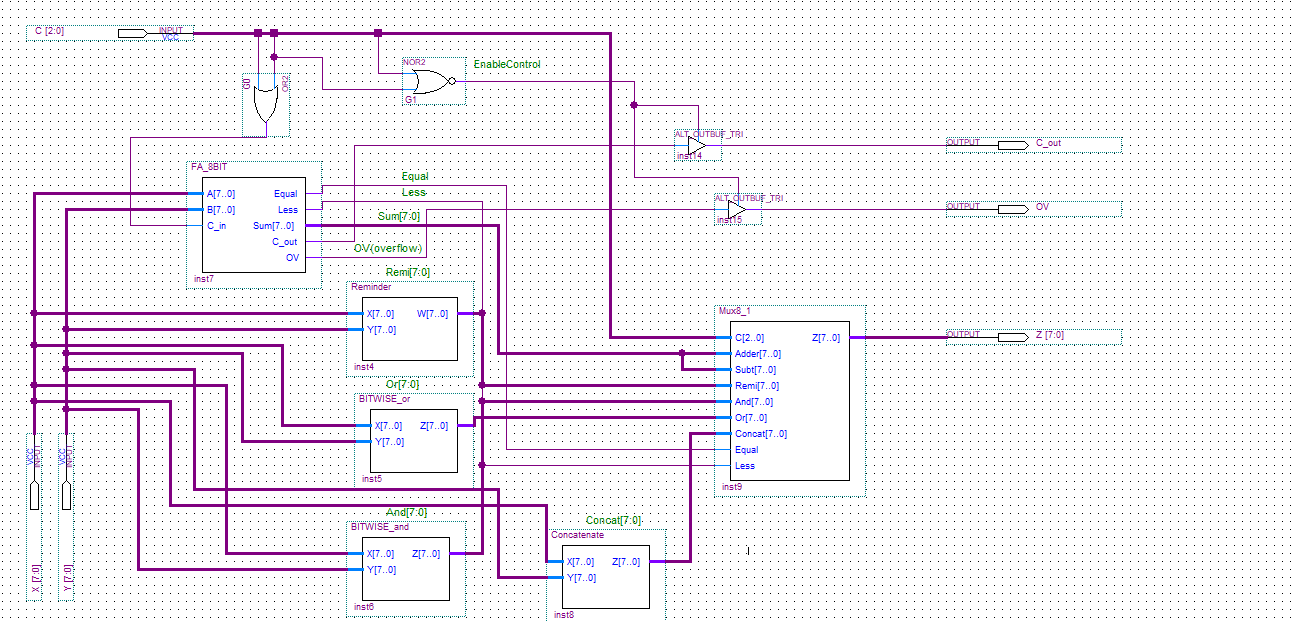
bufif1 Buf0(C\_out, C\_o, EnableControl);

bufif1 Buf1(OV, OV\_o, EnableControl);

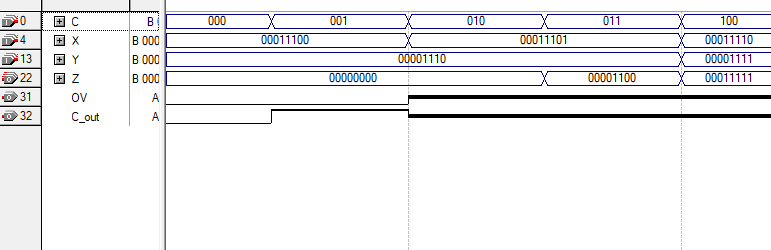
Mux8\_1 (Z, C, Adder, Subt, Remi, And, Or, Concat, Equal, Less);

endmodule

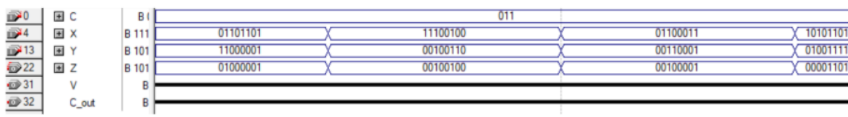
Block Diagram:



Simulations:

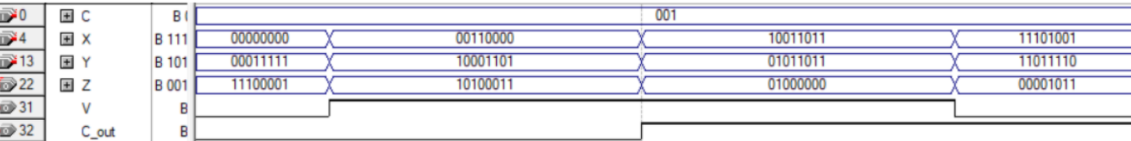


**ALU**

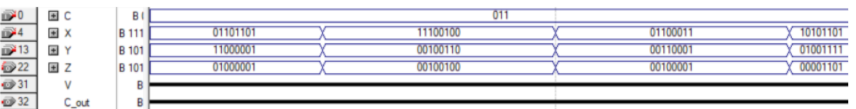


**ALU With Addition**

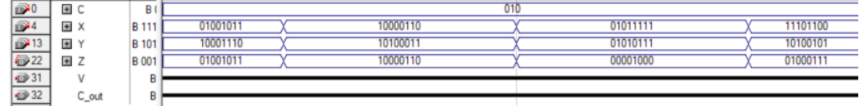
**ALU With Subtraction**



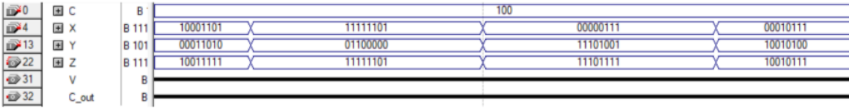
**ALU AND**



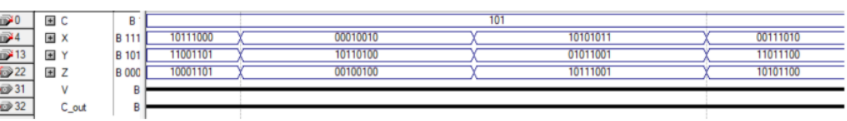
**ALU Reminder**

****

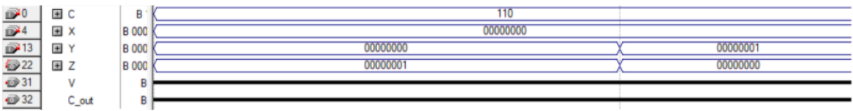
**ALU OR**

****

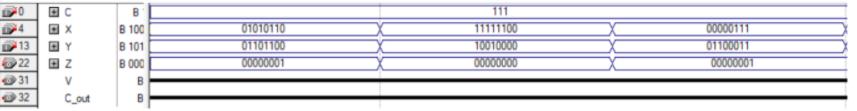
**ALU CONCATENATE**

****

**ALU EQUALITIY**

****

**ALU LESS**

****

**Simple Explanation about ALU:**

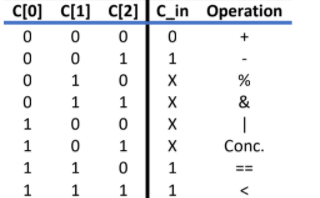
* It appears several other gates in the diagram, the **OR** gate which labeled by

“**G0**” is to make the value of **C\_in** equal one in all **FA\_8BIT** cases except in

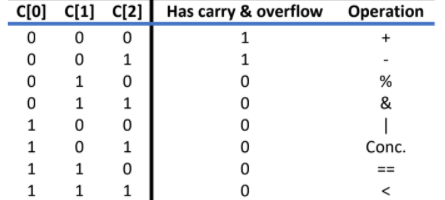
The addition operation (**C = 000**), because the value of **C\_in** should be 1 in

Subtraction, and the equality and less-than depend on subtraction operation,

So we should make the value of **C\_in** equal one in all cases expect in addition.



* There is no need to output carry (**C\_out**) and over flow (**OV**) in all operations except on addition and subtraction, so the NOR and BUFFER gates were put. For example, if we select to do equality operation (**C = 110**), we should have just output that tell us if the numbers are equal or not, without carry or overflow outputs.

****