TRABAJO PRACTICO I

Arquitectura de computadoras

UNIDAD ARITMÉTICA LÓGICA - ALU

Integrantes:

- Sosa Ludueña Gabriel
- Cazajous Miguel

2016

Introducción

El presente trabajo tiene como objetivo implementar, validar y verificar el funcionamiento de una ALU de *n* bits sobre sus operandos. La implementación de la ALU se lleva a cabo a través del lenguaje Verilog para su posterior verificación de funcionamiento a través de una placa FPGA Spartan E3-100.
El programa utilizado para llevar a cabo la implementación de la ALU es ISE 14.6 de Xilinx.

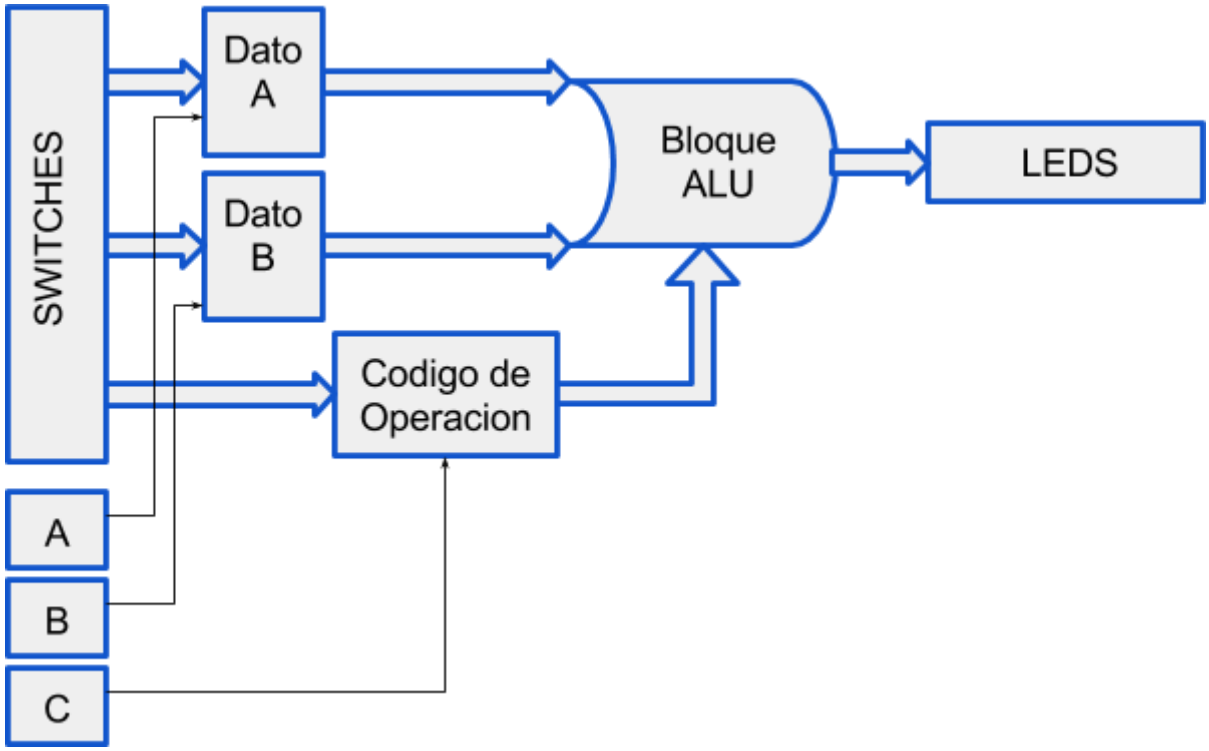
Consigna de trabajo

- Implementar en FPGA una ALU.
- Utilizar las placas de desarrollo Basys II.
- La ALU debe ser parametrizable (bus de datos) para poder ser utilizada posteriormente en el trabajo final.
- Validar el desarrollo por medio de Test Bench.
- La ALU deberá ser puramente combinacional y deberá ser instanciada desde otro módulo para poder manipularse.
- La bandera de salida overflow de la ALU en este caso no será necesaria ya que solo se verificará el funcionamiento de las operaciones en base a los bits indicados por parámetro para los operando como el resultado.

Los códigos de operación que considera la ALU son los siguientes.

Operación	Código
ADD	100000
SUB	100010
AND	100100
OR	100101
XOR	100110
SRA	000011
SRL	000010
NOR	100111

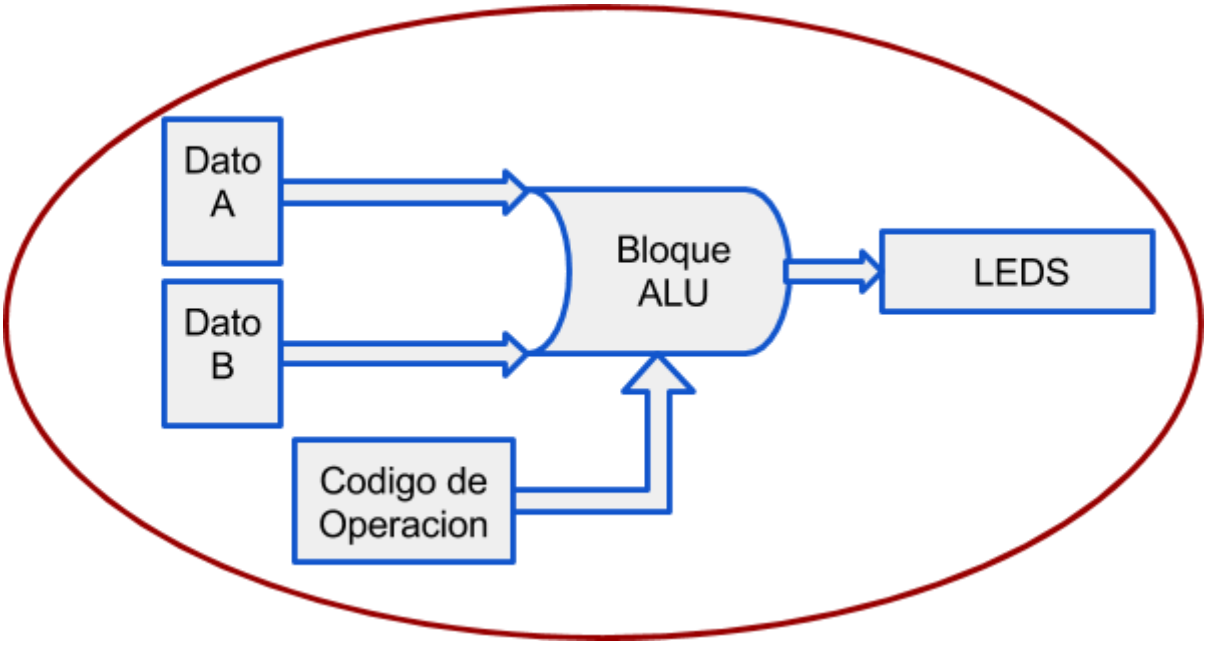
La implementación de la ALU se realizó siguiendo el diseño del siguiente esquema:



Desarrollo

Bloque ALU

Pensando el bloque ALU como una caja negra tenemos que empezar definiendo sus entradas y salidas. Como vimos en el esquema de diseño de la ALU el bloque ALU solo consiste de la entrada de los operandos A, B y del código de operación a llevar a cabo. Por otro lado la salida consiste en buffer de salida para representar los resultados en base a la operación seleccionada ha realizar sobre los operandos.



Como vimos en la *Consigna de trabajo* del presente trabajo, el bloque ALU debe ser puramente combinacional, por lo cual podemos suponer que sus entradas y salidas son buffers que no almacenan valores sino que en base a los valores que se presentan en estas entradas se obtendrá una salida correspondiente. Pero si suponemos además que tanto los datos de los operandos A y B como también el código de operación son registros externos que conectamos a estos buffers del bloque ALU entonces podemos decir que a medida que cambien los valores de los registros esto será detectado por los buffers del bloque ALU y se determinará una salida correspondiente a las entradas. De esta manera es como se planteó el desarrollo del bloque ALU para que sea puramente combinacional pero pensando en conectar los buffers de entrada a los registros correspondientes y así poder obtener y mantener los resultados deseados por el bloque ALU.

Por lo cual como vemos en el diseño del bloque ALU, para este caso las entradas y salidas que se tienen en cuenta son las siguientes.

Entradas bloque ALU

input wire buf_A: buffer de entrada que será conectado a un registro externo que mantendrá los valores cargados sobre el operando A.

input wire buf_B: buffer de entrada que será conectado a un registro externo que mantendrá los valores cargados sobre el operando B.

input wire buf_Op: buffer de entrada que será conectado a un registro externo que mantendrá los valores cargados como código de operación.

Salidas bloque ALU

output wire buf_R: buffer de salida que se encargará de mostrar el resultado de la operación llevada a cabo según el código de operación.

Código verilog de bloque ALU

En base a lo descrito sobre las variables de entrada y salida el código de verilog que implementa la ALU (o bloque ALU) se muestra como sigue.

```
1.  module bloque_ALU
2.  #(parameter nbits = 8) // Parámetros externos, de instanciación
3.  (buf_A, buf_B, buf_Op, buf_R);
4.
5.  // Parámetros internos
6.  parameter msb = nbits-1;// bit más significativo
7.
8.  // Entradas de ALU
9.  input wire signed[msb:0] buf_A, buf_B;
10. input wire [5:0] buf_Op;
11.
12. // Salida de ALU
13. output wire signed [msb:0] buf_R; // puede ser reg dato_R para forma 2.
14.
15. // Circuito puramente combinacional forma 1:
16.         assign buf_R = (buf_Op==`ADD) ? buf_A+buf_B:// ADD
17.                                (buf_Op==`SUB) ? buf_A-buf_B:// SUB
18.                                (buf_Op==`AND) ? buf_A&buf_B:// AND
19.                                (buf_Op==`OR) ? buf_A|buf_B://OR
20.                                (buf_Op==`XOR) ? buf_A^buf_B:// XOR
21.                                (buf_Op==`SRA & buf_A[msb]== 1) ? buf_A >>>
    (buf_B&8'b00011111): // SRA -> Se Se realiza ANDing para no superar 31 avances
22.                                (buf_Op==`SRA & buf_A[msb]== 0) ? buf_A >>
    (buf_B&8'b00011111):
23.                                (buf_Op==`SRL) ? buf_A >> buf_B:// SRL
24.                                (buf_Op==`NOR) ? ~(buf_A|buf_B): 0;// NOR
25. endmodule
```

Como se observa el módulo del bloque ALU desarrollado es parametrizable, esto quiere decir que el bloque ALU brinda la posibilidad de adaptarse a la cantidad de bits deseados sobre los operandos, es decir que, si se desea realizar la prueba de la ALU con 4 o 32 bits es decisión del usuario que haga uso del módulo, en caso de no ingresar el parámetro se inicializa por defecto a 8 bits sobre los operandos.

Dentro de las operaciones de desplazamiento aritmetico cuando un numero es negativo los lugares desocupados deberán llenarse con dicho signo, por lo cual se utiliza la entrada A cómo signed pero además debe tenerse en cuenta que como la arquitectura de la FPGA utilizada permite hasta 31 desplazamientos, el operando B no puede superar dicha cantidad y es por lo cual se utilizó la técnica de ANDing sobre el operando B, el cual nunca podrá superar los 31 desplazamientos en SRA.

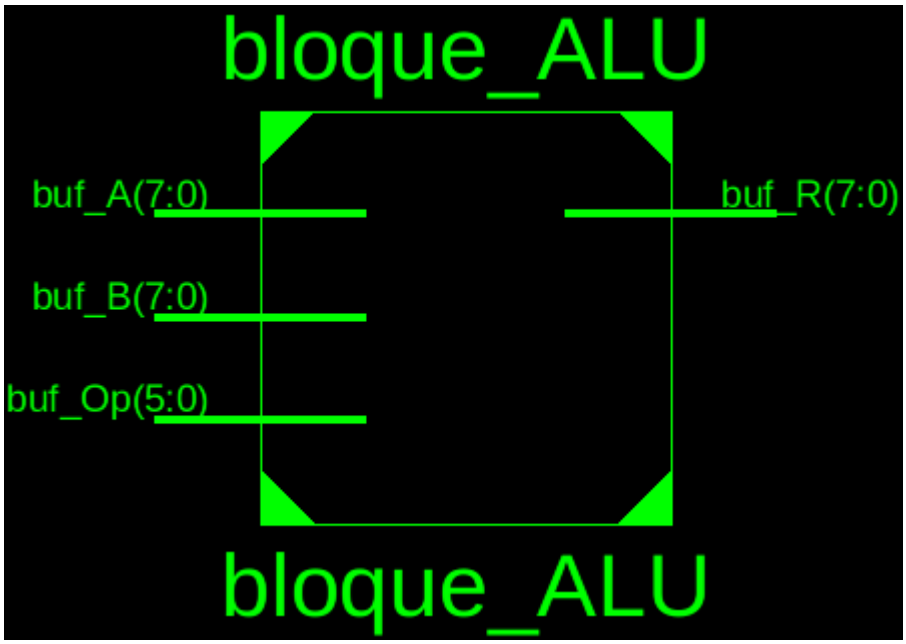
Por otro lado los valores constantes utilizados como por ejemplo `ADD son valores definidos previamente para cumplir con tal funcionalidad, es decir funcionar como constantes pero utilizadas como macros sobre el código de verilog. En nuestro caso las macros utilizadas son las siguientes.

```
1.  // Macros utilizadas sobre código verilog
2.  `define ADD  6'b100000
```

```
3.  `define SUB  6'b100010
4.  `define AND  6'b100100
5.  `define OR   6'b100101
6.  `define XOR  6'b100110
7.  `define SRA  6'b000011
8.  `define SRL  6'b000010
9.  `define NOR  6'b100111
```

Diagrama esquemático del bloque ALU

El diagrama esquemático generado por herramientas del programa ISE es el siguiente.



Bloque manejador de bloque ALU

Para realizar una instanciación del bloque ALU es necesario hacerlo desde otro módulo de verilog. El módulo manejador de ALU tiene por objetivo realizar la instanciación del bloque ALU para posteriormente poder ser probada a través de los testbench y por último llevarla a la practica sobre una placa FPGA.

El módulo manejador de la ALU es el encargado de proporcionar los registros hacia los buffers de entrada de la ALU ya que como dijimos esta es puramente combinacional. De esta manera el manejador de la ALU representa el esquema de diseño inicial.

Por lo cual para este caso imaginandolo también como una caja negra que usará entradas y salidas debemos pensarla desde el punto de vista del hardware que utilizaremos. Para nuestro caso hacemos uso de una placa Spartan E3 Basys2 que nos proporciona 8 switches y 4 pulsadores como entradas, y nos proporciona 8 leds y 4 displays 7 segmentos como salidas. Combinando estas características de hardware de las que disponemos y el software que proporciona la placa podemos llevar a cabo el diseño inicial. Por parte de hardware ya está todo dicho y por parte de software podemos decir que utilizando 3 registros para almacenar los operandos A,B y uno más para el código de operación ya es suficiente. Esto combinado con el único buffer que tenemos en la placa (que serían los switches de la misma) lo podemos combinar con los pulsadores para que cargado un dato deseado sobre el buffer se almacene en dato A si se presiona pulsador A, sino se almacena en dato B si presionamos

pulsador B y lo mismo para el codigo de operacion si presionamos el pulsador C. De esta manera podemos cargar los valores deseados y se mantienen sobre la placa ya que son registros.
Entonces teniendo en cuenta esto pasamos a definir las entradas y salidas que serán tomadas a través del módulo manejador de la ALU.

Entradas manejador ALU
input buf_in: buffer de entrada necesario para la captura de los valores a almacenar sobre los operandos o para almacenar la selección del código de operación.
input p_a: entrada de pulsador asociada a la captura de buf_in para almacenar el operando A.
input p_b: entrada de pulsador asociada a la captura de buf_in para almacenar el operando B.
input p_c: entrada de pulsador asociada a la captura de buf_in para almacenar el codigo de operacion.
input clk: entrada de clock o reloj para mantener el funcionamiento secuencial del circuito.

Variables internas sobre el manejador de ALU
Las variables internas para el módulo manejador son los registros de los operandos y el código de operación ya que no son ni salidas ni entradas pero si variables que maneja el módulo para mantener la información asociada a cada registro y así poder dar el funcionamiento deseado sobre el bloque ALU.
reg signed dato_A: variable interna utilizada para almacenar el operando A capturado desde el buf_in.
reg signed dato_B: variable interna utilizada para almacenar el operando B capturado desde el buf_in.
reg dato_Op: variable interna utilizada para almacenar el codigo de operacion capturado desde el buf_in.

Salidas manejador de ALU
output wire dato_R: salida conectada a los resultados obtenidos sobre los registros operando A y B.

Código de módulo manejador de ALU
El código verilog del módulo manejador se muestra como sigue.

```
1.  module manejador_ALU(clk, p_abc, buf_in, dato_R);
2.  //Parametros internos
3.  parameter nbits = 8; // numero de bits para buffers
4.  parameter msb = nbits-1; // bit mas significativo
5.
6.  // Entaradas manejador ALU
7.  input clk; // clock
8.  input [2:0]p_abc; // pulsadores a -> p_abc[0], b-> p_abc[1] y c -> p_abc[2]
9.  input [msb:0]buf_in; // buffer de entrada para capturar datos de operandos y
    codigos de op
10.
11. // Salidas manejador ALU
12. output wire [msb:0] dato_R; // registro de salida asociado a ALU
13.
14. // Variables internas
```

```
15. reg [msb:0] dato_A, dato_B; // registros que almacenaran los operandos
16. reg [5:0] dato_Op; // registro que almacena el codigo de operacion
```

Luego para detectar cuales son los pulsadores se generó un circuito de estados que según el pulsador guardamos el operando en base a ese pulsador, todo teniendo en cuenta la señal de clock.

```
17. // Captura de datos a través de buf_in
18. always @(posedge clk)
19. begin
20.     if(p_abc[0])
21.         dato_A = buf_in;
22.     else if (p_abc[1])
23.         dato_B = buf_in;
24.     else if(p_abc[2])
25.         dato_Op = buf_in[5:0];
26. end
```

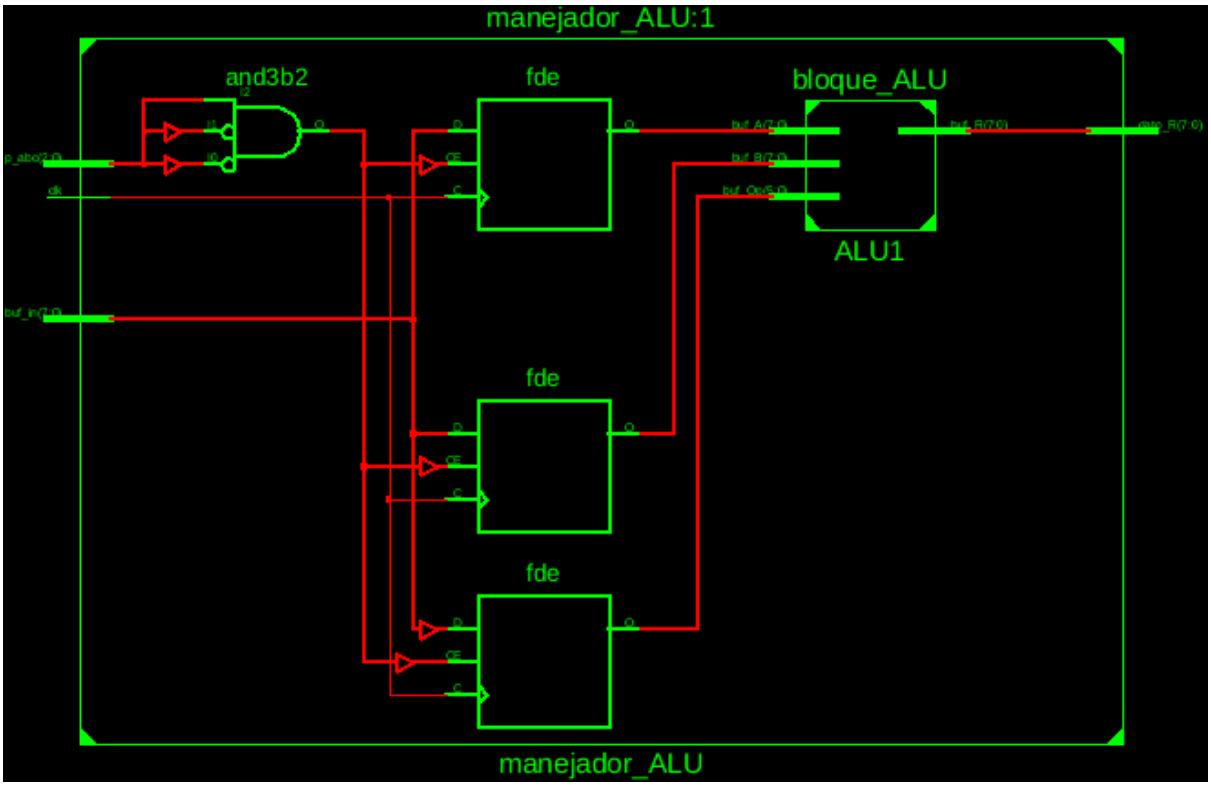
Por último se realiza la instanciación del bloque ALU para que a través del ingreso de valores a los registros se pueda verificar el funcionamiento de la misma.

```
27. // Instanciación de bloque ALU
28. bloque_ALU #(.nbits(nbits)) // paso de parametro, 8 bits en este caso
29. ALU1(
30.     .buf_A(dato_A),
31.     .buf_B(dato_B),
32.     .buf_Op(dato_Op),
33.     .buf_R(dato_R)
34. );
35.
36. endmodule
```

Concluyendo de esta manera el código del módulo manejador de ALU.

Diagrama esquemático de manejador de ALU

Como dijimos el diagrama esquemático consiste en llevar a cabo el esquema del diseño inicial y eso fue lo que se realizó con el módulo manejador de ALU, en donde podemos observar que la generación del esquemático que realiza el programa ISE es muy similar a lo deseado como se nota en la siguiente captura


manejador_ALU:1
and3b2
fde
bloque_ALU
ALU1
fde
fde
manejador_ALU

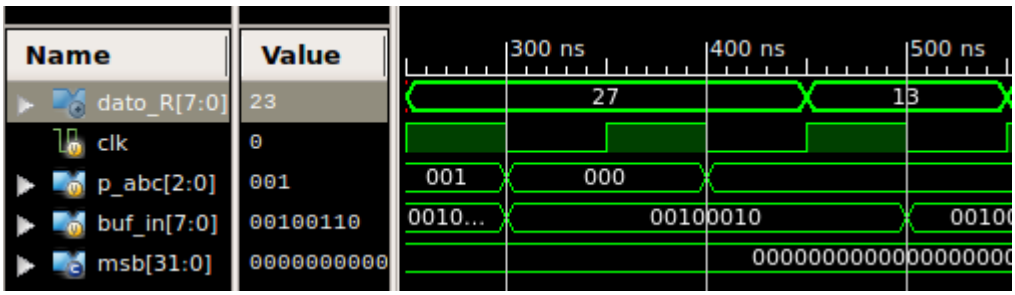
Testeo de funcionalidad

Para la verificación y validación de la funcionalidad del módulo ALU se generan testbench, herramienta proporcionada por el programa ISE para llevar a cabo las pruebas.

En el testbench se genero un código de validación, para verificar el funcionamiento general de las operaciones.

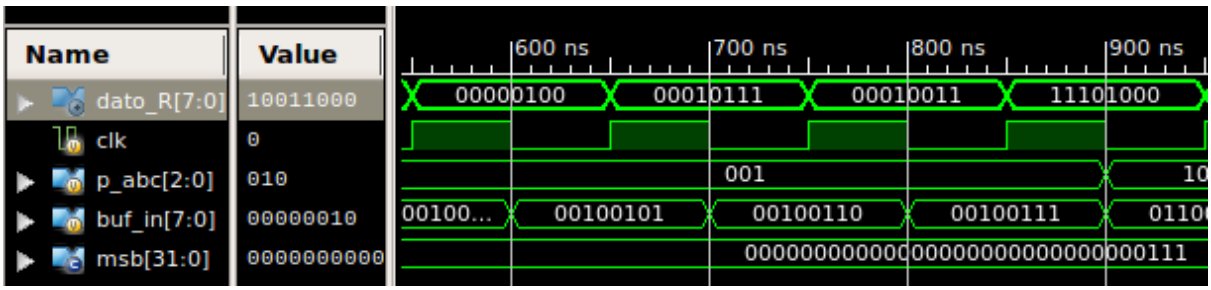
Verificación de operaciones aritméticas
Se carga el operando A = 20 = 00010000
Se carga el operando B = 7 = 00000111
Se selecciona operacion ADD = 100000
Se selecciona operacion SUB = 100010

El resultado de la suma se verifica correctamente como se observa en la captura



Al igual que la operación resta de los operandos.

Verificación de operaciones lógicas
Luego verificamos las operaciones de lógica como se observa en la siguiente captura

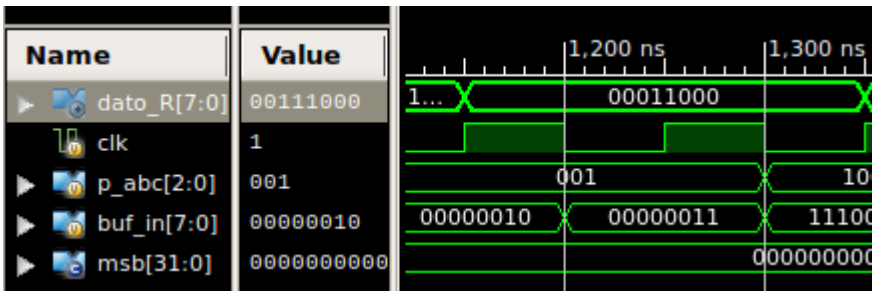


Donde en primer lugar se realiza la operación AND sobre los operandos cargados previamente que eran 20 sobre A y 7 sobre B. Por último con dichas cargas se verifica la funcionalidad de las operaciones OR, XOR y NOR.

Verificación de operaciones de desplazamiento
Por último verificamos las operaciones de desplazamiento, en donde se tienen en cuenta las siguientes cargas para los operandos.

Se carga el operando A = 01100000
Se carga el operando B = 2 = 00000111
Se selecciona operacion SRA = 000011

y luego se selecciona operacion SRL = 000010
Y se verifica la funcionalidad, ya que para este caso tanto SRL como SRA son iguales en funcionalidad debido a que se desplaza hacia la derecha los 2 lugares y se agregan ceros en los lugares desocupados.



A continuación realizamos las siguientes cargas
Se carga el operando A = 11100000
Operando sigue como B = 2 = 00000111
Se selecciona operacion SRA = 000011
y luego se selecciona operacion SRL = 000010

Donde para este caso son diferentes SRA y SRL ya que SRA tiene en cuenta el signo y es lo que se verifica en la siguiente captura.