1. SystemVerilog Combinacional para la realizacion de Operaciones Logicas y Aritmeticas sobre Binarios de 8bits

Andreu Lechuga, 202073595-6, andreu.lechuga@usm.cl

1. Resumen

Se desarrollo un programa en SystemVerilog de naturaleza Combinacional capaz de manipular un conjunto de entradas en binario y realizar una de ocho operaciones. La entrada del programa estuvo compuesta de 2 números binarios de 8bits y uno de 3 bits, los dos números a operar y el selector de la operación respectivamente, además de un reloj y una salida de 8bits. Dentro de las operaciones tenemos suma y resta binaria, junto con las compuertas OR, AND y XOR, con tanto su versión bitwise como Bytewise.

Cada una de las operaciones nombradas quedo plenamente funcional y a lo largo de este informe se detallará el proceso de concepción y desarrollo del código entero, así como la lógica detrás de cada operación.

2. Introducción

En este experimento se planteó, conceptualizo y desarrollo un circuito complejo con la capacidad de recibir dos números binarios de 8 bits, realizando una de ocho operaciones sobre ellos. La operación a realizar es decidida a partir de un tercer input, numero binario de 3 bits, que identifica cada operación del 000 hasta el 111. Las operaciones fueron suma y resta binaria, junto con comparaciones bitwise (bit a bit) y Bytewise (cascada) de las compuertas AND, OR y XOR. En la tabla 1 del Anexo se muestra la asignación de código de cada una de dichas operaciones. Finalizada la operación, el programa retorna u numero de 8bits con el resultado, por el canal de salida

La entrada del programa está compuesta por
- 2 entradas de 8bit
- 1 entrada selectora de 3bit
- 1 reloj (1bit)
- 1 salida de 8bit

Por otro lado, programa contempla los siguientes módulos principales
- main
- suma
- resta
- and_bitwise
- or_bitwise
- xor_bitwise
- and_bytewise
- or_bytewise
- xor_bytewise

Ademas de los modulos auxiliares
- sumador
- C2

Explicado de manera práctica, una operación **bitwise** entre dos números binarios de N-bits realiza la operación dada (AND, por ejemplo) entre cada bit individual basado en su posición dentro del número binario.
Por otro lado, una operación **Bytewise** en las mismas condiciones realiza un efecto cascada con el resultado **Bitwise** entre los mismos números. Una vez obtenida, operando la misma operación (AND) pero ahora entre los bits del **Bitwise** por separado, llegando a un resultado de solamente 1bit.

3. Desarrollo

No hubo un orden lógico en cuanto a la realización de las partes del programa, por lo que procederé a desarrollar el procedimiento de manera casi cronológica (y resumida en lo demás).

Lo primero en hacerse fue el módulo *suma*. Fue pensando, al igual que la *resta*, en función del circuito hecho en Logisim hace 2 semanas atrás, puesto que la lógica era la misma.

Para la *suma* se creo el submódulo *sumador* a partir de la tabla de verdad característica de un sumador de bit a bit + carry.

De esta manera, *suma* coordina a suma de los números de 8bits usando *sumador*. Administrando el bit resultante y el carry, *suma* retorna una salida de 8bits con el resultado.

Luego se trabajó el módulo *resta*, dado que utiliza *suma*. Para la *resta* se creó el submódulo *C2*, cuya función consiste en recibir un binario de 8bits y retornar su Complemento 2. EN detalle, realiza la conversión del numero a su Complemento 1 y luego le suma 1 (00000001) al C1, utilizando *suma.*

De esta forma, *resta* recibe 2 números binarios de 8bits y complementa uno (el segundo) usando *C2*. Luego coordina la suma del primero numero con el complemento del segundo usando el módulo *suma* y retorna una salida de 8bits con el resultado.

Lo siguiente fueron las puertas AND, OR y XOR.

Primero se trabajo con las versiones **bitwise**, cuya lógica es encapsulale en una sola explicación, puesto que es la misma para las tres compuertas. Para los módulos **bitwise** se recibieron los números, se compararon los bits en función de su posición y de la operación correspondiente (AND, OR, XOR) y luego se retornó el valor final.

Luego se procedió a trabajar en la lógica de las operaciones **Bytewise** que, al igual que las bitwise, comparten una lógica común. Para los módulos **Bytewise** fue necesario utilizar su modulo bitwise correspondiente. Con este resultado, el siguiente paso fue realizar la operación correspondiente y conjunta de cada uno de los bits que lo conformaban, dando como resultado un valor de 1bit, que se retorna.
Es posible ver ejemplos de ambas implementaciones (bitwise y Bytewise) en la sección de Resultados.

Finalmente, se trabajo en el módulo main, cuya función era integrar la estructura que el programa requería y administrar su organiza modular como el módulo principal. Debido a esto, sus entradas coinciden con las mencionadas en la sección de Introducción. En su estructura interna encontramos la inicialización de variables (presente en todos los módulos anteriores también y parte de la sintaxis de SystemVerilog) y un apartado de tipo Demultiplexor.

El apartado mencionado (soy consciente del mal uso del punto aparte) funciona en base al selector de 3bits y redirecciona los dos inputs binarios hacia su operación correspondiente, evidentemente dada por el selector.

Hasta aquí ya hemos descrito el circuito completo con todo el detalle necesario, quizás con la única excepción de aquellos detalles propios de la sintaxis de SystemVerilog. En otras palabras, se podría decir que la lógica subyacente al código está cubierta. En la siguiente sección veremos un análisis en base a pruebas de todos los módulos recién mencionados.

4. Resultados

En este apartado se presentarán las distintas pruebas de cada uno de los módulos creados para el funcionamiento del programa. Para esto se utilizarán los ejemplos corroborados que nos otorga el departamento, como parte del enunciado.
Notar que esto excluye a los submódulos *sumador* y *C2* puesto que no estaban explícitamente contemplados, además de que su funcionalidad se ve demostrada indirectamente con el correcto funcionamiento de sus módulos padres correspondientes.

Ejemplo 1 - Suma

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 10111001

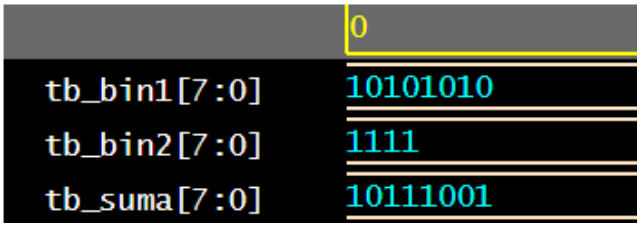

Figura 3: Consola EPWave. Suma

Ejemplo 2 – Resta

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 10011011

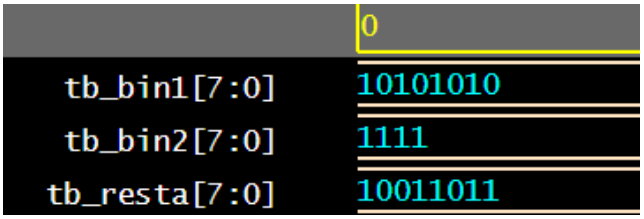

Figura 4: Consola EPWave. Resta

Ejemplo 3 – OR bitwise

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 10101111

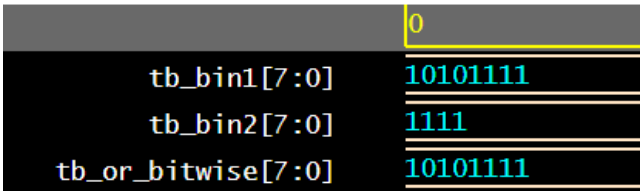

Figura 5: Consola EPWave. OR bitwise

Ejemplo 4 – AND bitwise

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 00001010

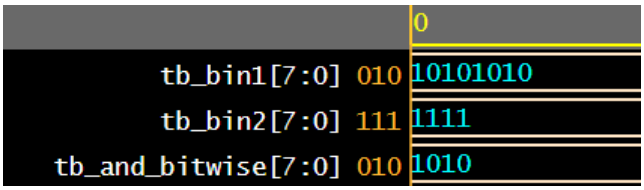

Figura 6: Consola EPWave. AND bitwise

Ejemplo 5 – XOR bitwise

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 10100101

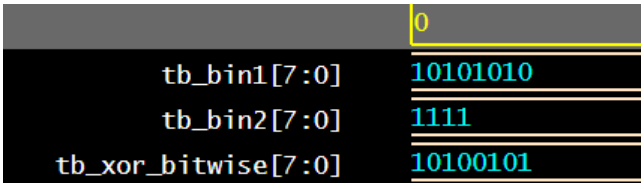

Figura 7: Consola EPWave. XOR bitwise

Ejemplo 6 – OR Bytewise

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 1 (11111111)

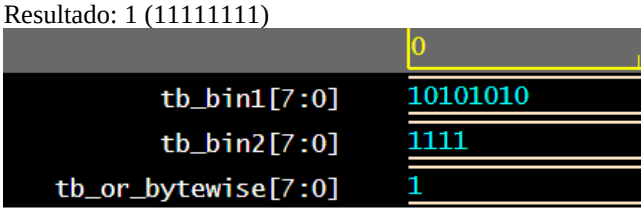

Figura 8: Consola EPWave. OR Bytewise

Ejemplo 7 – AND Bytewise

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 0 (00000000)

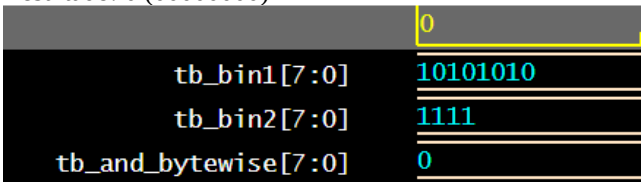

Figura 9: Consola EPWave. AND Bytewise

Ejemplo 8 – XOR Bytewise

Valores
Bin1: 10101010
Bin2: 00001111
Resultado: 0 (00000000)

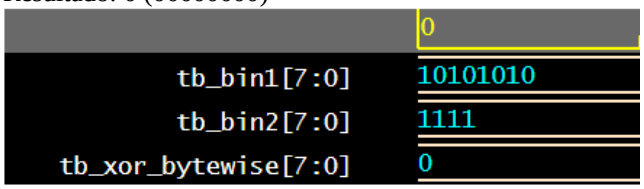

Figura 10: Consola EPWave. XOR Bytewise

De esta manera dejamos demostrado el funcionamiento de los módulos. De manera complementaria se pueden consultar en el anexo los *testbench* usados para las demostraciones. Pasamos a Análisis

5. Análisis

Honestamente el proceso de realización de proyecto entero, desde los primeros acercamientos hasta el final, fue bastante caótico. La compresión de que hacer y como hacerlo estuvo la mayor parte del tiempo fuera lo obvio a simple vista. Como lenguaje de muy bajo nivel es difícil cambiar el paradigma que levamos hasta ahora tan fácilmente. Aun así, luego de suficiente trabajo de mucho error y de difícil documentación, fue posible, como todo, lograr un código estable y funcional.

6. Conclusión

Con el proyecto realizado se profundizo en el conocimiento sobre Hard Description Lenguages, comprendiendo el funcionamiento de la maquina en su propio plano. Además, este conocimiento se vio a través de una lente Combinacional, puesto que por circunstancias de todo tipo fue la única parte que se llegó a desarrollar.

Se comprendió cuales son los recursos de la maquina y como los utiliza, que es la memoria, como funcionan los registros, entre otras cosas.

Mediante la superación, teórica y práctica, de bastantes imprevistos y dificultades, se completó con éxito el programa que se buscaba realizar.

7. Anexos

```
// Code your testbench here
// or browse Examples

module testbench_suma();
  logic [7:0] tb_bin1, tb_bin2, tb_suma;

  suma dut(.bin1(tb_bin1),
           .bin2(tb_bin2),
           .suma_bin(tb_suma));

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111; #10;

  end
endmodule

```

Testbench – Suma

```
// Code your testbench here
// or browse Examples


module testbench_resta();
  logic [7:0] tb_bin1, tb_bin2, tb_resta;

  resta dut(.bin1(tb_bin1),
            .bin2(tb_bin2),
            .resta_bin(tb_resta));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111; #10;

  end
endmodule


```

Testbench - Resta

```
// Code your testbench here
// or browse Examples

module testbench_bitwise_or();
  logic [7:0] tb_bin1, tb_bin2, tb_or_bitwise;

  or_bitwise dut(.bin1(tb_bin1),
                 .bin2(tb_bin2),
                 .or_output_b(tb_or_bitwise));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);

    tb_bin1 = 8'b1010_1111;
    tb_bin2 = 8'b0000_1111;
    #10;

  end
endmodule


```

Testbench – OR bitwise

```
// Code your testbench here
// or browse Examples


module testbench_bitwise_and();
  logic [7:0] tb_bin1, tb_bin2, tb_and_bitwise;

  and_bitwise dut(.bin1(tb_bin1),
                  .bin2(tb_bin2),
                  .and_output_b(tb_and_bitwise));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);

    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111; #10;

  end
endmodule


```

Testbench – AND bitwise

```
// Code your testbench here
// or browse Examples


module testbench_bitwise_xor();
  logic [7:0] tb_bin1, tb_bin2, tb_xor_bitwise;

  xor_bitwise dut(.bin1(tb_bin1),
                  .bin2(tb_bin2),
                  .xor_output_b(tb_xor_bitwise));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);

    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111;
    #10;

  end
endmodule


```

Testbench – XOR bitwise

```
// Code your testbench here
// or browse Examples


module testbench_bytewise_or();
  logic [7:0] tb_bin1, tb_bin2, tb_or_bytewise;

  or_bytewise dut(.bin1(tb_bin1),
                  .bin2(tb_bin2),
                  .or_output_B(tb_or_bytewise));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);

    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111;
    #10;

  end
endmodule


```

Testbench – OR Bytewise

```
// Code your testbench here
// or browse Examples


module testbench_bytewise_and();
  logic [7:0] tb_bin1, tb_bin2, tb_and_bytewise;

  and_bytewise dut(.bin1(tb_bin1),
                   .bin2(tb_bin2),
                    .and_output_B(tb_and_bytewise));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);

    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111;
    #10;

  end
endmodule

```

Testbench – AND Bytewise

```
// Code your testbench here
// or browse Examples


module testbench_bytewise_xor();
  logic [7:0] tb_bin1, tb_bin2, tb_xor_bytewise;

  xor_bytewise dut(.bin1(tb_bin1),
                   .bin2(tb_bin2),
                   .xor_output_B(tb_xor_bytewise));
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);

    tb_bin1 = 8'b1010_1010;
    tb_bin2 = 8'b0000_1111;
    #10;

  end
endmodule

```

Testbench – XOR Bytewise

8. Bibliografía

No hay bilbiografia