Informe Tarea 3

Lenguajes de Descripción de Hardware

Grupo 38

Diego Acevedo Pizarro 202023528-7

Diego Paz Letelier 202004502-k

19 de noviembre de 2022

Resumen

Según lo solicitado para esta tarea, debimos diseñar virtualmente una ALU que fuese capaz de procesar multiples operaciones con A y B números binarios de 8 bits cada uno. Dichas operaciones son identificadas por medio de una secuencia de 3 bits, entregando como salida el resultado de la operación, añadiendo a este el reporte de una serie de flags en caso de que alguna de las condiciones que describen a las mismas se produzca.

Para comprender mejor el input, incluimos un ejemplo:

OP-Code	A	B
000	23	56

Tabla 1: Ejemplo de input

Por otro lado, el formato de output para el ejemplo anterior sería:

Salida	Flags
79	02

Tabla 2: Ejemplo de output

Índice

1. Desarrollo de la tarea

Los comandos solicitados para el desarrollo de la tarea, se resumen en la siguiente lista:

OP-Code	**Comandos Solicitados**
000	Suma simple
001	Resta simple
010	Suma positiva
011	Resta positiva
100	Rotación izquierda
101	Rotación derecha
110	Duplicación
111	División binaria

Además, se nos solicita reportar una serie de flags dependiendo de si se cumplen sus condiciones. Estas son:

Flags Solicitadas
N Negative
Z Zero
C Carry
V Overflow
G Greater
Q Equal
O Odd
P Pairs

Entonces, para ir creando el código para los comandos posibles de la ALU, empezamos con las sumas y restas, en estas la diferencia es en como se interpreta los números que nos entregan, puesto que en las sumas y restas simples, se interpretan todos los números en formato *2-Complemento o signed*, mientras que en las sumas y restas positivas, se interpretan como *Magnitud o unsigned*, y debido a esto las flags generadas por estas operaciones difieren.

A continuacion el código que se usa en estas operaciones:

```
    case(opcode)
3'b000: //Suma 2-complemento
    begin
        temp_carry = 0;
        A_comp = ~A_in + 8'h01;
        B_comp = ~B_in + 8'h01;
        {temp_carry,resultado} = ~(A_comp + B_comp) + 8'h01;
    end
3'b001: //Resta 2-complemento
    begin
        temp_carry = 0;
        A_comp = ~A_in + 8'h01;
        B_comp = ~B_in + 8'h01;
        {temp_carry,resultado} = ~(A_comp - B_comp) + 8'h01;
    end
3'b010: //Suma magnitud
    begin
        temp_carry = 0;
        {temp_carry,resultado} = A_in + B_in;
    end
3'b011: //Resta magnitud
    begin
        temp_carry = 1;
        {temp_carry,resultado} = A_in - B_in;
    end
```

Figura 1: Extracto de código

En el código se puede observar que se hace uso de una variable *temp_ carry*, esta está para indicar el carry que producen algunas operaciones cuando son números grandes o negativos dependiendo del tipo de suma o resta. Para las operaciones suma y resta simple/2-complemento, realizamos un cambio a 2-complemento tanto a los inputs como al output resultado, sin embargo al ser cíclico el cambio a 2-complemento, el resultado se ve inalterado. Para las operaciones suma y resta positiva/magnitud la operación se realiza sin mucho cambio a una operación normal pero para la resta se establece un carry positivo, puesto que la resta en lugar de entregar un carry, pide un borrow, que seria lo contrario, entoces el 'carry' positivo está para inicar si se realizó un borrow en el MSB.

Las siguientes operaciones son las rotaciones izquierda y derecha, y para estas se usó concatenación de bits, tal como está mostrado en el siguiente snippet de código:

```
    3'b100: //Rotación izquierda
    begin
        temp_carry = 0;
        temporal = A_in;
        repeat(B_in) begin
            temporal = {temporal[6:0],temporal[7]};
        end
        resultado = temporal;
    end
3'b101: //Rotación derecha
    begin
        temp_carry = 0;
        temporal = A_in;
        repeat(B_in) begin
            temporal = {temporal[0],temporal[7:1]};
        end
        resultado = temporal;
    end
```

Figura 2: Extracto de código

Se hizo uso del comando *repeat* de SystemVerilog para no hacer uso del ciclo for.

Las siguientes operaciones son la duplicación y division binaria, para estas operaciones se utilizaron los *Logical Left Shift y Logical Right Shift*, como se muestra en el código:

```
    3'b110: //Duplicación
    begin
        temp_carry = 0;
        {temp_carry,resultado} = (A_in << B_in)|(A_in >> (8 - B_in));
    end
3'b111: //División binaria
    begin
        temp_carry = 0;
        resultado = A_in >> B_in;
    end
default: resultado = 8'b0;
endcase
```

Figura 3: Extracto de código

Y así terminan las operaciones, entonces lo siguiente son las flags que emiten los resultados de estas, entonces empezando tenemos la flag N, que indica si el número del resultado es negativo, el código de esta flag es sencillo y indica que si el MSB es 1, el resultado es negativo, y hace las excepciones en las operaciones que trabajan con números unsigned:

```
    case (opcode)
    3'b010:
        flags[7] = 0;
    3'b011:
        flags[7] = 0;
    3'b110:
        flags[7] = 0;
    3'b111:
        flags[7] = 0;
    default: begin
        if (resultado[7]) begin  //N, el valor de salida es negativo
            flags[7] = 1;
        end else begin
            flags[7] = 0;
        end
    end
endcase
```

Figura 4: Extracto de código

Las siguientes flags son Z, el resultado es cero y C, la operación produce un carry, estas funcionan tal que en el caso de la primera checkea si el resultado es cero y en el caso de la segunda solo evalua el valor de temp_bit:

```
    if (resultado == 0) begin //Z, el valor de salida es cero
    flags[6] = 1;
end else begin
    flags[6] = 0;
end

if (temp_carry) begin //C, la operación produce un carry de salida
    flags[5] = 1;
end else if ((opcode == 3'b011) & (temp_carry == 0)) begin
    flags[5] = 1;
end else begin
    flags[5] = 0;
end
```

Figura 5: Extracto de código

La siguiente flag, G tiene particularidades ya que debe evaluar si el número está en 2-complemento o no, por lo que no puede simplemente comparar los números, entonces se debió hacer un case que fuera por cada operacion estableciendo las condiciones, el código de esta flag es extenso por lo que se pondrá al final.

Las siguientes flags son la flag Q que significa que A y B son iguales, la flag O que significa que el resultado es impar y la flag P que significa que en el resultado hay la misma cantidad de 0s y 1s. La primera flag se implemento con una comparación simple, la segunda flag se implementó checkeando el LSB, y en el caso que este fuese 1, el resultado era impar, y la ultima flag se implementó tal que sumaba todos los bits del resultado y si estos eran igual a 4, se activaba la flag. A continuación el código de estas flags:

```
    if (A_in == B_in) begin //Q, el valor de A es igual al de B
    flags[2] = 1;
end else begin
    flags[2] = 0;
end

if (resultado[0]) begin //O, el valor de salida es impar
    flags[1] = 1;
end else begin
    flags[1] = 0;
end

sum = resultado[7] + resultado[6] + resultado[5] +  //
resultado[4] + resultado[3] + resultado[2] + resultado[1] + resultado[0];
if (sum == 4) begin //P, el valor de salida tiene la misma cantidad de 0's y 1's
    flags[0] = 1;
end else begin
    flags[0] = 0;
end
```

Figura 6: Extracto de código

Y un ejemplo de la flag G, que es muy grande para mostrarla es el siguiente código que tiene los casos para cuando la operacion maneja numeros en 2-complemento y cuando solo maneja numeros en magnitud:

```
3'b001:
    begin
        if (A_in[7] & (B_in[7] == 0)) begin
            tempbit1 = A_in;
            tempbit2 = B_in;
            if (tempbit1 > tempbit2) begin
                flags[3] = 1;
            end
        end
        if ((A_in[7] == 0) & B_in[7]) begin
            tempbit1 = A_in;
            tempbit2 = B_in;
            if (tempbit1 > tempbit2) begin
                flags[3] = 1;
            end
        end
        if (A_in[7] & B_in[7]) begin
            tempbit1 = A_in;
            tempbit2 = B_in;
            if (tempbit1 > tempbit2) begin
                flags[3] = 1;
            end
        end
        if ((A_in[7] == 0) & (B_in[7] == 0)) begin
            tempbit1 = A_in;
            tempbit2 = B_in;
            if (tempbit1 > tempbit2) begin
                flags[3] = 1;
            end
        end
    end
3'b010:
    if (A_in > B_in) begin
        flags[3] = 1;
    end
```

Figura 7: Extracto de código

2. Resultados y análisis

Al ejecutar nuestra ALU con el testbench obtuvimos los siguientes resultados, cabe destacar que hubo flags erroneas en el ejemplo y están en el siguiente log, se puede asumir que el flag que dió es el correcto en la mayoria de los casos

```
    # KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: Esto dio: 79 y debia ser 79 en Suma 2-complemento, y en binario: 01111001
# KERNEL: Con flags: 02 que debian ser 02, y en binario: 00000010
# KERNEL: Esto dio: cf y debia ser CF en Suma 2-complemento, y en binario: 11001111
# KERNEL: Con flags: 92 que debian ser 92, y en binario: 10010010
# KERNEL: Esto dio: f2 y debia ser F2 en Suma 2-complemento, y en binario: 11110010
# KERNEL: Con flags: 80 que debian ser 80, y en binario: 10000000
# KERNEL: Esto dio: 3c y debia ser 3C en Suma 2-complemento, y en binario: 00111100
# KERNEL: Con flags: 21 que debian ser 01, y en binario: 00100001
# KERNEL: Esto dio: f7 y debia ser F7 en Suma 2-complemento, y en binario: 11110111
# KERNEL: Con flags: a2 que debian ser AA, y en binario: 10100010
# KERNEL: Esto dio: 1d y debia ser 1D en Suma 2-complemento, y en binario: 00011101
# KERNEL: Con flags: 23 que debian ser 32, y en binario: 00100011
# KERNEL: Esto dio: 27 y debia ser 27 en Resta 2-complemento, y en binario: 00100111
# KERNEL: Con flags: 0b que debian ser 0B, y en binario: 00001011
# KERNEL: Esto dio: e7 y debia ser E7 en Resta 2-complemento, y en binario: 11100111
# KERNEL: Con flags: ba que debian ser 8A, y en binario: 10111010
# KERNEL: Esto dio: ff y debia ser FF en Resta 2-complemento, y en binario: 11111111
# KERNEL: Con flags: a2 que debian ser 82, y en binario: 10100010
# KERNEL: Esto dio: ff y debia ser FF en Suma magnitud, y en binario: 11111111
# KERNEL: Con flags: 0a que debian ser 0A, y en binario: 00001010
# KERNEL: Esto dio: 00 y debia ser 00 en Suma magnitud, y en binario: 00000000
# KERNEL: Con flags: 78 que debian ser 78, y en binario: 01111000
# KERNEL: Esto dio: f0 y debia ser F0 en Resta magnitud, y en binario: 11110000
# KERNEL: Con flags: 29 que debian ser 29, y en binario: 00101001
# KERNEL: Esto dio: 10 y debia ser 10 en Resta magnitud, y en binario: 00010000
# KERNEL: Con flags: 20 que debian ser 30, y en binario: 00100000
# KERNEL: Esto dio: f0 y debia ser F0 en RotaciÃ³n izquierda, y en binario: 11110000
# KERNEL: Con flags: 89 que debian ser 89, y en binario: 10001001
# KERNEL: Esto dio: 3c y debia ser 3C en RotaciÃ³n derecha, y en binario: 00111100
# KERNEL: Con flags: 09 que debian ser 09, y en binario: 00001001
# KERNEL: Esto dio: 20 y debia ser 20 en DuplicaciÃ³n, y en binario: 00100000
# KERNEL: Con flags: 00 que debian ser 00, y en binario: 00000000
# KERNEL: Esto dio: 00 y debia ser 00 en DuplicaciÃ³n, y en binario: 00000000
# KERNEL: Con flags: 40 que debian ser 70, y en binario: 01000000
# KERNEL: Esto dio: 3f y debia ser 3F en DivisiÃ³n binaria, y en binario: 00111111
# KERNEL: Con flags: 0a que debian ser 09, y en binario: 00001010
# KERNEL: Esto dio: 00 y debia ser 00 en DivisiÃ³n binaria, y en binario: 00000000
# KERNEL: Con flags: 44 que debian ser 44, y en binario: 01000100
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
```

Estos resultados nos convencen ya que en su mayoria estos eran correctos, lo que nos asegura un buen funcionamiento de la ALU y una buena implementación de las flags.

3. Conclusiones

Como era de esperarse, el resultado de nuestra ALU de SystemVerilog funciona correctamente en todas las operaciones y flags solicitados. Esto fue testeado con los casos de ejemplo provistos en la tarea, además de diversos casos de ejemplo particulares para evaluar de forma independiente tanto los operadores como la identificación de las flags respectivas.

Una vez realizado este proyecto, comprendimos que una vez diseñado este sistema de comandos, nuestra ALU será fácilmente modificable para la introducción de nuevas operaciones, he incluso el diseño de detección de errores en el formato, los cuales podrían ser añadidos en una versión futura de la ALU diseñada.