TEC | Tecnológico de Costa Rica

Ambiente de Verificación para ALU de 8088.

Escuela de Ingeniería Electrónica

Verificación Funcional de circuitos integrados

Integrantes:

Kenneth González Gutiérrez

Gabriel Rodríguez Palacios

Kendall Salazar Carazo

Profesor

Kervin Sánchez Herrera

Marzo 2023

Tabla de contenido

Resumen ejecutivo

El proyecto de verificación funcional de la ALU del microprocesador 8088 tiene como objetivo principal garantizar que la unidad aritmético-lógica del procesador 8088 funcione correctamente y cumpla con los requisitos de diseño establecidos. La ALU es una parte crítica del procesador, responsable de realizar operaciones aritméticas y lógicas esenciales para el funcionamiento del microprocesador y también de todos los entornos programados.

El proyecto implica llevar a cabo pruebas funcionales básicas para verificar que la ALU del 8088 se comporte según lo esperado en diferentes escenarios y condiciones. Esto implica el diseño de casos de prueba que cubran una amplia gama de operaciones y situaciones posibles, incluyendo sumas, restas, multiplicaciones, divisiones y operaciones lógicas.

La importancia de este proyecto radica en garantizar la confiabilidad y precisión de la ALU del microprocesador 8088 que como la historia demuestra es un factor clave para la implementación de estos en sistemas más complejos como los aviones. Al realizar una verificación funcional de esta manera, se asegura que el procesador pueda ejecutar correctamente las instrucciones y realizar los cálculos necesarios en una amplia variedad de aplicaciones y escenarios, e incluso en algunos casos creándolos en entornos muy específicos, de esta forma se evita tener que invertir en pruebas empíricas de manera exhaustiva y se permite tener un mayor control sobre las variables lógicas- funcionales de la ALU. Un resultado exitoso en la verificación funcional de la ALU del 8088 proporciona la confianza necesaria en el procesador, asegurando un rendimiento óptimo y libre de errores en las operaciones aritméticas y lógicas.

Introducción

La rápida evolución de la tecnología ha llevado a una creciente demanda de microprocesadores capaces de realizar diversas tareas computacionales de manera eficiente y en tiempos aceptables. En el centro de estos circuitos multipropósito se encuentra la Unidad Aritmética Lógica (ALU), que desempeña un papel fundamental al realizar todas las operaciones necesarias para brindar un propósito general a cualquier microprocesador. [1]

El presente documento tiene como objetivo analizar la importancia y funcionalidad de la ALU en los circuitos multipropósito y microprocesadores. Para ello, se examinarán las características clave de la ALU y su relevancia en el contexto de los conjuntos de instrucciones, así como su capacidad para ejecutar operaciones programadas en Sistem Verilog y en base al microprocesador 8088.

Se hará especial hincapié en el principio básico de completitud, donde se espera que un set de instrucciones sea capaz de resolver cualquier tarea computacional en un tiempo aceptable. La ALU toma relevancia al ser la encargada de garantizar que estas instrucciones se ejecuten de manera eficiente, rápida y con precisión, evitando la generación de valores no deseados.

Además, se analizará la importancia de verificar todas las posibles opciones y asegurar el correcto funcionamiento de la ALU desde el punto de vista de la verificación. Esto se vuelve crucial para garantizar que las instrucciones retornen los valores deseados y evitar posibles errores en el proceso de cálculo.

A lo largo del informe, se examinarán ejemplos de operaciones complejas, que demuestran la capacidad de la ALU para abordar una amplia gama de tareas computacionales.

Marco Teórico

Microprocesador 8088:

El microprocesador 8088, desarrollado por Intel en 1979, fue uno de los primeros procesadores ampliamente utilizados en las computadoras personales. Tenía una arquitectura de 16 bits y funcionaba a una velocidad de reloj de hasta 5 MHz. El 8088 presentaba una arquitectura basada en registros, donde los registros internos jugaban un papel fundamental en la ejecución de instrucciones y el manejo de datos. Los registros principales incluían el acumulador (AX), el registro de datos (DX), el registro de puntero (CX) y el registro de base (BX). Además, existían registros de segmento para el manejo de memoria y registros de índice para operaciones de direccionamiento. [2]

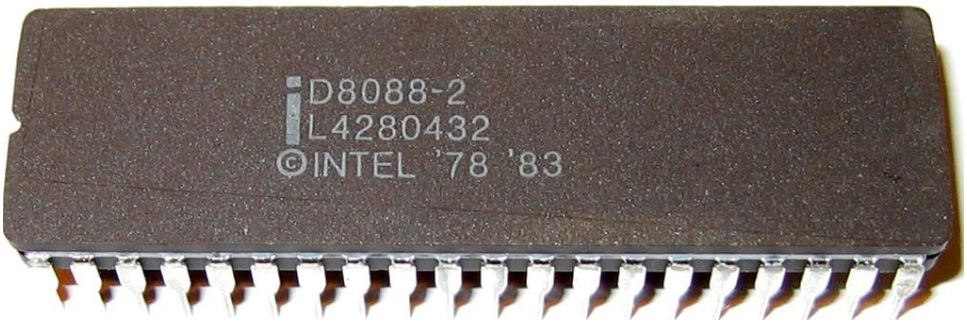


Figura 1: Microprocesador 8088.

ALU (Unidad aritmética lógica):

Esta unidad se encargaba de realizar distintas funciones aritméticas como serian sumas, restas, multiplicaciones y divisiones a su vez realizaba funciones lógicas como lo serían las AND, XOR, OR y demás, a su vez desempeñaba funciones de transferencia de datos. En estos casos la ALU opera con instrucciones en formato binario manipulando bits individuales que se agrupaban de manera tal que se podía codificar el tipo de instrucción, los registros objetivo y destino e incluso los modos de direccionamiento a los registros. [3]

Como tal la ALU del 8088 podía ejecutar un total de 34 operaciones diferentes según el opcode, específicamente las siguientes:

Tabla 1. operaciones de la ALU del 8088

1. ADD (Suma)	2. AAS (Ajuste después de la resta en decimal)
3. OR (Operación lógica OR)	4. DAA (Ajuste después de la suma en BCD)
5. ADC (Suma con acarreo)	6. DAS (Ajuste después de la resta en BCD)
7. SBB (Resta con acarreo)	8. MUL (Multiplicación de 8 bits)
9. AND (Operación lógica AND)	10. MUL (Multiplicación de 16 bits
11. SUB (Resta)	12. IMUL (Multiplicación con signo de 8 bits)
13. XOR (operación lógica XOR)	14. IMUL (Multiplicación con signo de 16 bits)
15. CMP (Comparación)	16. DIV (División de 8 bits)
17. ROL (Rotación a la izquierda)	18. DIV (División de 16 bits)
19. ROR (Rotación a la derecha)	20. IDIV (División con signo de 8 bits
21. RCL (Rotación a la izquierda con acarreo)	22. IDIV (División con signo de 16 bits)
23. RCR (Rotación a la derecha con acarreo)	24. NOT (Operación lógica NOT)
25. SHL (Desplazamiento a la izquierda)	26. NEG (Negación aritmética)
27. SHR (Desplazamiento a la derecha lógico)	28. CBW (Conversión de byte a palabra)
29. SAL (Desplazamiento aritmético a la izquierda)	30. CWD (Conversión de palabra a doble palabra
31. SAR (Desplazamiento aritmético a la derecha)	32. AAM (Ajuste después de la multiplicación en BCD) 33.
34. AAA (Ajuste después de la suma en decimal)	35. AAD (Ajuste antes de la división en BCD) 36.

Para llevar a cabo estas distintas instrucciones, el 8088 contaba con un ciclo de instrucción que tenía varias etapas, estas etapas incluían el ciclo de búsqueda de instrucción, donde se recuperaba la instrucción de memoria; el ciclo de decodificación, donde se identificaba el opcode y los operandos; el ciclo de ejecución, donde se realizaban las operaciones requeridas; y finalmente, el ciclo de escritura, donde los resultados se almacenaban en registros o memoria. [3]

Como tal esta ALU no tenía tecnologías tan avanzadas de procesamiento en paralelo como seria la ejecución fuera de orden o el multithreading no se podía hablar de una ejecución en paralelo por parte de la ALU, sin embargo, gracias a que la arquitectura del 8088 permitía acceder de manera paralela a las otras etapas de ejecución, se podía reducir los tiempos de espera de la ALU, lo que le permitía

ejecutar más operaciones y terminaba resultando en un aumento en el rendimiento y la eficiencia de las operaciones realizadas por el microprocesador.

Banderas (Flags):

El registro de banderas es de suma importancia para la ALU, ya que en este se guardan resultados que resultan de gran relevancia para el sistema, es un registro de 16 bits que lo que realiza es una concatenación entre los distintos tipos de banderas. Entre los cuales encontramos las siguientes: [3]

1. Desbordamiento (OF):
    - Indica si se ha producido un desbordamiento en operaciones aritméticas con signo.
    - Se establece en 1 si se produce un desbordamiento, lo que significa que el resultado es demasiado grande para ser representado correctamente en el formato de bits utilizado.
    - Ayuda a detectar errores en operaciones con números con signo.
2. Dirección (DF):
    - La Flag de dirección es utilizada en operaciones de movimiento de cadenas de datos en el microprocesador 8088.
    - Cuando la DF está en 0 (valor por defecto), los datos se mueven de manera ascendente, es decir, desde direcciones de memoria más bajas hacia direcciones más altas.
    - Si se establece la DF en 1, los datos se mueven de manera descendente, desde direcciones de memoria más altas hacia direcciones más bajas.
    - Esta flag permite controlar el sentido en el que se mueven los datos en operaciones de manipulación de cadenas.
3. Interrupción (IF):
    - La Flag de Interrupción es una bandera utilizada para habilitar o deshabilitar las interrupciones en el microprocesador 8088.

- Cuando la IF está en 1, las interrupciones externas están habilitadas y el microprocesador responderá a las solicitudes de interrupción que reciba.
- Si se establece la IF en 0, las interrupciones externas se deshabilitan y el microprocesador no responderá a ninguna solicitud de interrupción.
- Esta bandera permite controlar si el microprocesador debe atender las interrupciones externas o si debe ignorarlas temporalmente.

4. Trampa (TF):
    - La Flag de Trampa se utiliza principalmente para fines de depuración y control de ejecución paso a paso en el microprocesador 8088.
    - Cuando la TF está en 1, el procesador se encuentra en modo de trampa o modo de depuración, lo que significa que después de ejecutar cada instrucción, se genera una interrupción del tipo Trap (interrupción 1).
    - Esto permite a un depurador o monitor de ejecución controlar el flujo de ejecución del programa, deteniéndose en cada instrucción para su análisis y seguimiento.
    - La Flag de Trampa se utiliza principalmente en entornos de desarrollo de software y no tiene una función directa en el funcionamiento normal de una aplicación.
5. Signo (SF):
    - Indica si el resultado de una operación es negativo o positivo.
    - Si el resultado es negativo, el bit de signo más significativo se copia al flag de signo; si el resultado es positivo, se establece en 0.
    - Es utilizado para realizar comparaciones y saltos condicionales basados en el signo del resultado
6. Cero (ZF):
    - Indica si el resultado de una operación es igual a cero.
    - Si el resultado es cero, esta bandera se establece en 1; si el resultado no es cero, se establece en 0.

- Se utiliza para realizar comparaciones y saltos condicionales en el código de programa.

7. Auxiliar de acarreo (AF):
    - Se utiliza en operaciones de suma y resta para indicar un acarreo o desbordamiento en los 4 bits menos significativos (nibble bajo).
    - Normalmente no se utiliza en la programación del lenguaje de alto nivel y es más relevante para la programación en ensamblador.
8. Paridad (PF):
    - Se utiliza en operaciones de suma y resta para indicar un acarreo o desbordamiento en los 4 bits menos significativos (nibble bajo).
    - Normalmente no se utiliza en la programación del lenguaje de alto nivel y es más relevante para la programación en ensamblador.
9. Acarreo (CF):
    - Utilizado para indicar un acarreo o desbordamiento en operaciones de suma o resta.
    - Si el resultado de la operación genera un acarreo o desbordamiento, esta bandera se establece en 1; de lo contrario, se establece en 0.
    - Es útil para realizar cálculos de números grandes o realizar ajustesen casos de desbordamiento. [4]

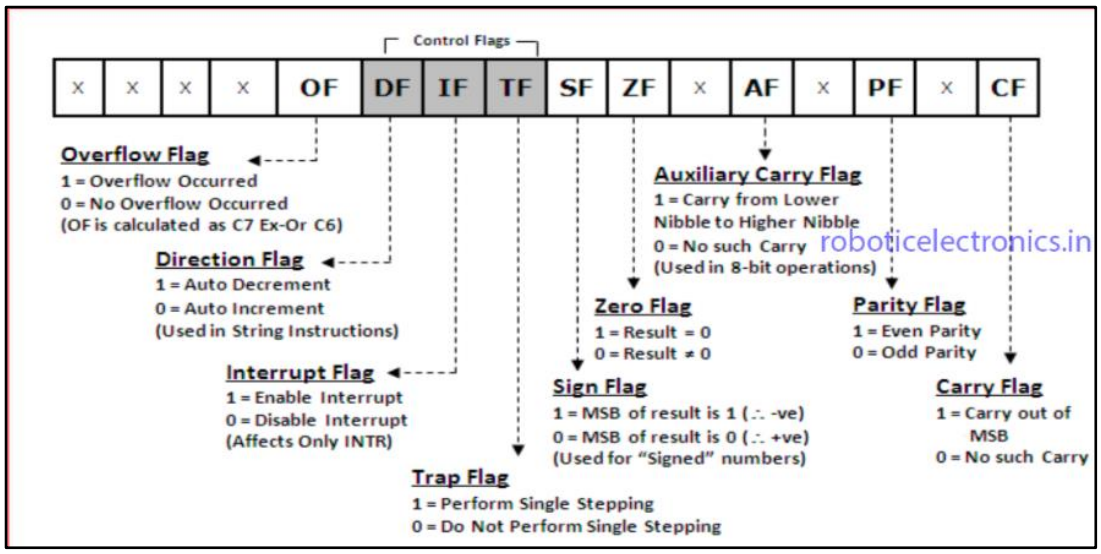


Figura 2. banderas del 8088.

Descripción de la solución

La solución propuesta para el desarrollo de la plataforma de verificación modularizada para el módulo ALU de la arquitectura x86 (8088) se basa en los siguientes pasos:

Adquisición de conocimientos en Cover Groups, Score y Tester: Se realizará un estudio de los fundamentos de los Cover Groups, Score y Tester. Esto implicará la investigación relacionada con la verificación funcional y los conceptos mencionados. Se buscará comprender cómo se aplican estos elementos en la verificación de hardware y cómo contribuyen a evaluar la cobertura de pruebas.

Utilización de clases, métodos estáticos y variables: Con los conocimientos adquiridos en clase, se utilizarán clases, métodos estáticos y variables en el lenguaje de programación SystemVerilog para crear y gestionar los Cover Groups, Scoreboard y Tester. Estos elementos se diseñarán de manera modular, permitiendo una fácil reutilización y configuración de las pruebas. Se implementarán métodos estáticos para obtener información de cobertura y evaluar la calidad de las pruebas realizadas.

Desarrollo de módulos de verificación: Se diseñarán y desarrollarán los módulos de verificación específicos para el módulo ALU de la arquitectura x86 (8088). Esto implicará la creación de casos de prueba que cubran las funcionalidades del módulo, la generación de estímulos para el módulo bajo prueba y la implementación de la lógica de verificación necesaria para asegurar que el comportamiento del módulo cumpla con las especificaciones. Se utilizarán los Cover Groups y Score para evaluar la cobertura de las pruebas y asegurar que se han probado adecuadamente todos los casos.

Análisis de resultados

A continuación, se va a realizar un análisis de testeo realizado a la ALU del 8088 proporcionado por el profesor, en la figura 1 podemos ver una parte del testeo que va desde la iteración 24 a la 37, donde la distinta información que se observa en la consola se encuentra de la siguiente forma:

Iteracion: se encuentra en decimal y lleva el conteo de las iteraciones realizadas.

Operacion: se encuentra en binario y es de 6 bits.

A: se encuentra en decimal y es de 16 bits, por lo que el máximo valor en decimal es 65 535.

B: se encuentra en decimal y es de 16 bits, por lo que el máximo valor en decimal es 65 535.

D: se encuentra en decimal y es de 16 bits, por lo que el máximo valor en decimal es 65 535.

R: se encuentra en decimal y es de 16 bits por lo que el máximo valor en decimal es 65 535 en todos los casos excepto en las multiplicaciones (MUL, IMUL), ya que se concatenan R1 y R2 por lo que es de 32 bits y su máximo valor en decimal es de 4 294 967 295.

residuo: se encuentra en decimal y es de 16 bits, por lo que el máximo valor en decimal es 65535.

Flags: se encuentra en binario y es de 6 bits, donde se muestra en consola la salida FL, que es la concatenación de ciertos bits de la salida FLAGS.

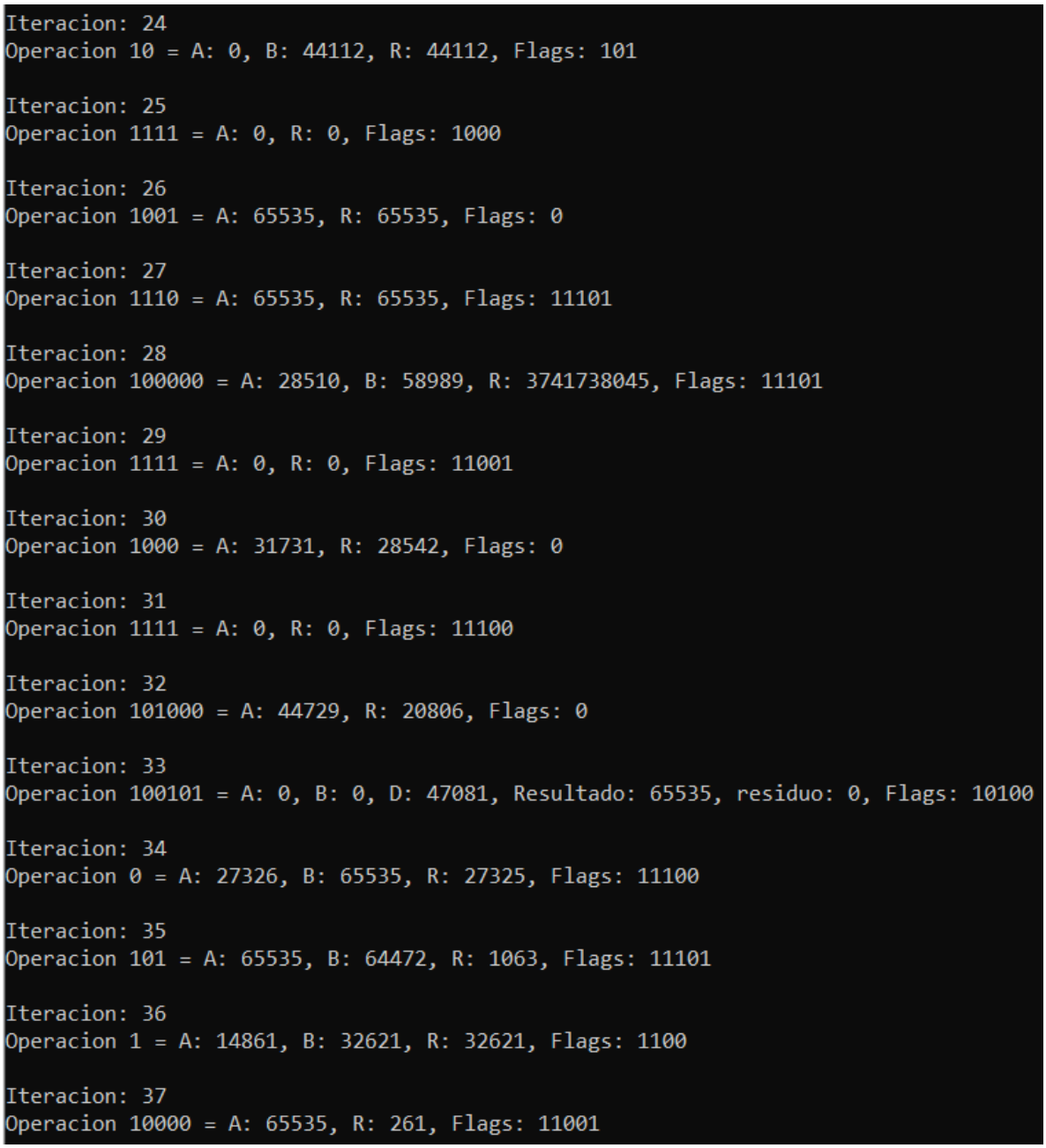


Figura 3: Testeo de las operaciones de la ALU con sus respectivos resultados.

En la iteración 24 se observa que se realizó la operación 10 que es lo mismo que 000010 solo que la consola no coloca los ceros más significativos, la operación es la ADC (Suma con acarreo) la cual podemos ver que se suma A y B, en este caso el valor de A es cero por lo que al realizar la suma se va obtener en R el mismo

valor que B como se observa en la Figura 1 da lo esperado y para Flags da 101 que indica un 1 en SF (Sign Flag) y un 1 en PF (Parity Flag).

Luego en la iteración 25 se observa que realiza la operación 1111 que es la SAR (Desplazamiento aritmético a la derecha) donde se observa que realiza a la 0 por lo que era de esperarse que el resultado sea también cero y en Flags muestra un valor de 1000 donde ese 1 lógico es de ZF (Zero Flag) lo que también era de esperarse ya que el resultado es cero.

En la iteración 26 se puede observar que realiza la operación 1001 que es una ROR (Rotación a la derecha) al numero 65535 por lo que era de esperar a la salida R el mismo valor.

En la iteración 28 se realiza la operación 100000 la cual es la MUL (Multiplicación de 8 bits) de A con valor de 28 510 y B con valor de 58 989 lo que da de resultado 3 741 738 045 el cual da un resultado incorrecto ya que realice la concatenación de R = {R1, R2}, si hubiera realizado la concatenación R = {R2, R1} me hubiera dado correctamente en consola y en Flags da 11101, como podemos observar a continuación el resultado en consola y el correcto:

R = 3 741 738 045 = 1101 1111 0000 0110 0110 0100 0011 1101

28 510 * 58 989 = 1 681 776 390 = 0110 0100 0011 1101 1101 1111 0000 0110

En la iteración 33 se realiza la operación 100101 la cual es la DIV (División de 16 bits) con A y B igual a cero y D igual a 47081 dando como resultado 65535, en la división se realiza la concatenación en el numerador de [D:A] y el denominador es B por lo que el resultado sería infinito al dividir por cero, entonces el resultado obtenido se puede decir que da bien al obtener su máximo valor, el valor de Flags es 10100 lo que nos da 1 lógicos en CF (Carry Flag) y SF (Sign Flag) lo que también era de esperar ya que seguiría escribiendo unos en la respuesta dando el signo en uno lógico y que hay acarreo.

Después en la iteración 34 se realiza la operación cero ADD(Suma) de los valores de A igual a 27326 y el valor de B es 65535, el resultado de R es 27325 a simple vista parece ser incorrecto debido a que no es posible que una suma me de

menor a uno de los valores que estoy sumando, pero lo que está ocurriendo es que hay un 1 lógico que se está perdiendo de información, en Flag nos da 11100 en donde el importante a observar el CF que nos dice que hay un carreo, las demás banderas que nos tira es ZF y SF, a continuación podemos ver los resultados en binario.

27 325 = 0110 1010 1011 1101

27 326 + 65 535 = 92 861 = 0001 0110 1010 1011 1101

Conclusiones

- Se concluye que la ALU proporcionada por el profesor funciona correctamente, después de haber realizado el análisis de las distintas operaciones del tester.
- Se concluye que el tester realizado funciona correctamente solo con un error en la concatenación de la respuesta de las multiplicaciones el cual se reparó en el apartado de anexos.

Anexos

Testbench para la ALU del 8088 proporcionada por el profesor:

```
`include "all.v"

`timescale 1ns / 1ps

typedef enum bit[5:0] { ADD =  6'b000000,  //// de aqui
                        OR =   6'b000001,
                        ADC =  6'b000010,
                        SBB =  6'b000011,
                        AND =  6'b000100,
                        SUB =  6'b000101,
                        XOR =  6'b000110,   /// hasta aqui ocupa 2 A
                        CMP =  6'b000111,
                        ROL =  6'b001000,
                        ROR =  6'b001001,   //// de aqui
                        RCL =  6'b001010,
                        RCR =  6'b001011,
                        SHL =  6'b001100,  //afectados por el V
                        SHR =  6'b001101,
                        SAL =  6'b001110,
                        SAR =  6'b001111,   ///hasta aqui
                        AAA =  6'b010000,
                        AAS =  6'b010101,
                        DAA =  6'b011000,
                        DAS =  6'b011101,
                        MUL =  6'b100000, // 8BITS     ///de aqui
                        MUL2 = 6'b100001, // 16BITS
                        IMUL = 6'b100010, // 8BITS
                        IMUL2 =6'b100011, // 16BITS   ///hasta aqui ocupa 2 A
                        DIV =  6'b100100, // 8BITS    //// de aqui
                        DIV2 = 6'b100101, // 16BITS
                        IDIV = 6'b100110, // 8BITS
                        IDIV2 =6'b100111, // 16BITS   //hasta aqui ocupa 3 A
```

```
                NOT =  6'b101000,
                NEG =  6'b101001,
                CBW =  6'b101100,
                CWD =  6'b101101,
                AAM =  6'b101110,
                AAD =  6'b101111}operacion;

//////////////////////////Clase que me genera todos los valores randoms//////////////////////
    class generador_de_randoms;
        function bit[5:0] get_op();
            bit [4:0] op_choice;
            bit op_choice2;
            op_choice = $random;
            op_choice2 = $random;
            case (op_choice)
                5'b00000 : begin
                    if(op_choice2 == 0)
                        return ADD;
                    else
                        return AAM;
                end
                5'b00001 : begin
                    if(op_choice2 == 0)
                        return OR;
                    else
                        return AAD;
                end
                5'b00010 : return ADC;
                5'b00011 : return SBB;
                5'b00100 : return AND;
                5'b00101 : return SUB;
                5'b00110 : return XOR;
```

```
                5'b00111 : return CMP;
                5'b01000 : return ROL;
                5'b01001 : return ROR;
                5'b01010 : return RCL;
                5'b01011 : return RCR;
                5'b01100 : return SHL;
                5'b01101 : return SHR;
                5'b01110 : return SAL;
                5'b01111 : return SAR;
                5'b10000 : return AAA;
                5'b10001 : return AAS;
                5'b10010 : return DAA;
                5'b10011 : return DAS;
                5'b10100 : return MUL;
                5'b10101 : return MUL2;
                5'b10110 : return IMUL;
                5'b10111 : return IMUL2;
                5'b11000 : return DIV;
                5'b11001 : return DIV2;
                5'b11010 : return IDIV;
                5'b11011 : return IDIV2;
                5'b11100 : return NOT;
                5'b11101 : return NEG;
                5'b11110 : return CBW;
                5'b11111 : return CWD;
            endcase // case (op_choice)
        endfunction : get_op

 //Generacion Aleatoria de DATA de 16 bits
function bit[15:0] get_data16();
    bit [1:0] zero_ones;
    zero_ones = $random;
```

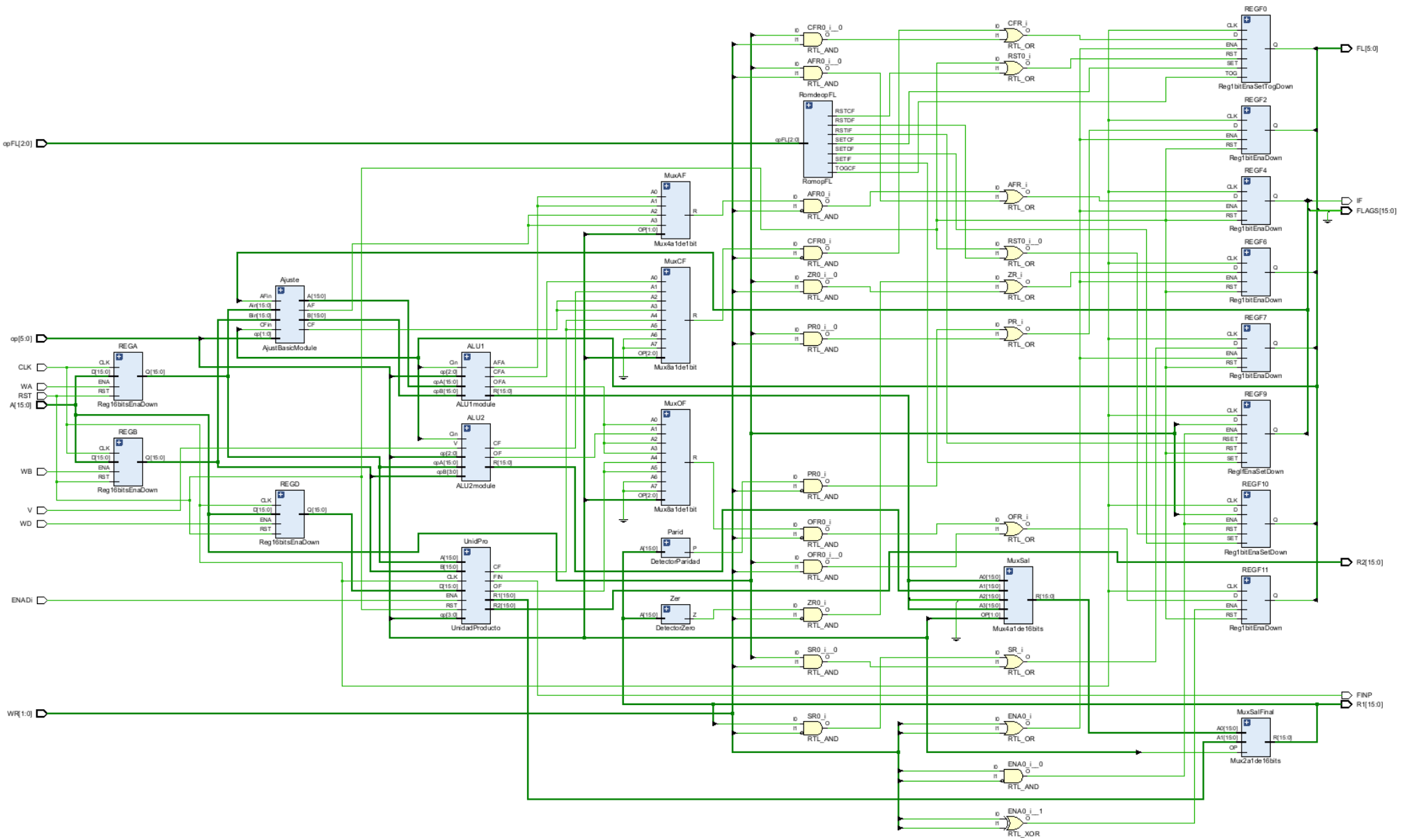
```
            if (zero_ones == 2'b00)
                        return 16'h0000;
            else if (zero_ones == 2'b11)
                        return 16'hFFFF;
            else
                        return $random ;
        endfunction : get_data16
          endclass : generador_de_randoms

module principal;
        bit CLK;
        bit RST;
        bit [15:0] A,a,b,d;
          bit V;
          bit [5:0] op;
          bit WA;
          bit WB;
          bit WD;
          bit ENADi;
          bit [1:0] WR;
          bit [2:0] opFL;
            bit [31:0]Rt;

          output wire [15:0] R1,R2,FLAGS;
          output wire [5:0] FL;
          output wire FINP,IF;

          bit [10:0] iteration;

          ALU DUT (.CLK,.RST,.A,.V,.op,.WA,.WB,.WD,.ENADi,.WR,.opFL,.R1,.R2,.FLAGS,.FL,.FINP,.IF);
```

```
////////////////////////////////Definiciones Iniciales -> CLK//////////////////////////////////////////////////
        initial begin
           CLK = 1'b0;
           forever begin
              #10;
              CLK = ~CLK;
           end
        end

        initial begin
           #1000000 $finish;
        end

////////////////////////////////////Instancias de las clases//////////////////////////////////////////////////
   generador_de_randoms gen;
/////////////////////////////////////////Tester del proyecto en general/////////////////////////////////////////
        initial begin : tester
        //Secuencia de reset para poner el sistema en un estado conocido
        assign RST = 1'b0;
        @(posedge CLK);
        assign RST = 1'b1;
        @(posedge CLK);
        assign RST = 1'b0;

        //Variable para ir imprimiendo luego el numero de iteracion
        iteration = 0;
        repeat (1000) begin
                iteration =  iteration + 1;
                $display("\nIteracion: %0d", iteration);
                gen = new();
                op = gen.get_op();
                //$display("\nOperacion %0d , %0b",op,op);
```

```
if (op <= 6'b000111)begin
        A = gen.get_data16();
        a = A;
        WA = 1;
        @(negedge CLK);
        WA = 0;
        gen = new();
        A = gen.get_data16();
        b = A;
        WB = 1;
        @(negedge CLK);
        WB = 0;
        WR = 3;
        @(negedge CLK);
        WR = 0;
        $display("Operacion %0b = A: %0d, B: %0d, R: %0d, Flags: %0b",op, a, b, R1, FL);
        end
else if(op >= 6'b100100 && op <= 6'b100111) begin  ///division
        A = gen.get_data16();
        a = A;
        WA = 1;
        @(negedge CLK);
        WA = 0;
        gen = new();
        A = gen.get_data16();
        b = A;
        WB = 1;
        @(negedge CLK);
        WB = 0;
        gen = new();
        A = gen.get_data16();
        d = A;
```

```
            WD = 1;
            @(negedge CLK);
            WD = 0;
            ENADi = 1;
            #100
            ENADi = 0;
            while (FINP == 0)begin
                  @(negedge CLK);
                  //$display("\nWhile");
            end
            WR = 3;
            @(negedge CLK);
            WR = 0;
            $display("Operacion %0b = A: %0d, B: %0d, D: %0d, Resultado: %0d, residuo: %0d,
            Flags: %0b",op, a, b, d, R1,R2,FL);
end
else if(op >= 6'b100000 && op <= 6'b100011) begin //multiplicaciones
            A = gen.get_data16();
            a = A;
            WA = 1;
            @(negedge CLK);
            WA = 0;
            gen = new();
            A = gen.get_data16();
            b = A;
            WB = 1;
            @(negedge CLK);
            WB = 0;
            @(negedge CLK);
            @(negedge CLK);
            Rt = {R2,R1};
            WR = 3;
            @(negedge CLK);
```

```
                    WR = 0;
                    $display("Operacion %0b = A: %0d, B: %0d, R: %0d, Flags: %0b",op, a, b, Rt,FL);
                end
                else if((op >= 6'b000111 && op <= 100000) || op >= 6'b101000)begin // || op == 6'b101001
|| op == 6'b101100 || op == 6'b101101 || op == 6'b101110 || op == 6'b101111)begin
                    A = gen.get_data16();
                    a = A;
                    WA = 1;
                    @(negedge CLK);
                    @(negedge CLK);
                    WA = 0;
                    WR = 3;
                    @(negedge CLK);
                    WR = 0;
                    $display("Operacion %0b = A: %0d, R: %0d, Flags: %0b",op, a, R1, FL);
                end
            end
        end : tester
endmodule : principal
```


opFL[2:0]
op[5:0]
CLK
WA
RST
A[15:0]
WB
V
WD
ENADi
WR[1:0]
REGA
REGB
REGD
Reg16bitsEnaDown
Ajuste
AjustBasicModule
ALU1
ALU1module
ALU2
ALU2module
UnidPro
UnidadProducto
MuxAF
Mux4a1de1bit
MuxCF
Mux8a1de1bit
MuxOF
Parid
DetectorParidad
Zer
DetectorZero
RomopFL
CFR0_i__0
AFR0_i__0
RST0_i
AFR0_i
CFR0_i
ZR0_i__0
PR0_i__0
PR0_i
OFR0_i
OFR0_i__0
ZR0_i
SR0_i__0
SR0_i
ENA0_i
ENA0_i__0
ENA0_i__1
RTL_AND
RTL_OR
RTL_XOR
CFR_i
AFR_i
RST0_i__0
ZR_i
PR_i
OFR_i
SR_i
MuxSal
Mux4a1de16bits
MuxSalFinal
Mux2a1de16bits
REGF0
REGF2
REGF4
REGF6
REGF7
REGF9
REGF10
REGF11
Reg1bitEnaSetTogDown
Reg1bitEnaDown
RegIfEnaSetDown
Reg1bitEnaSetDown
FL[5:0]
IF
FLAGS[15:0]
R2[15:0]
FINP
R1[15:0]

Bibliografía

[1] Bembibre, V. (2009). Defínción de ALU. DefinicionABC. https://www.definicionabc.com/tecnologia/alu.php

[2] J. Vasquez. Red tercer milenio. "Arquitectura de computadoras 1". Available: https://tesuva.edu.co/phocadownloadpap/Arquitectura_computadoras_I.pdf

[3] Roca, J. (s.f.). Así es cómo tu CPU y GPU realizan los cálculos matemáticos en tu PC. HardZone. https://hardzone.es/reportajes/que-es/alu/

[4] D. Alpern. "Los microprocesadores 8086 y 8088". Available: https://www.alpertron.com.ar/8088.HTM