Practica 2. La Unidad Aritmetico-Logico

Paul Vazquez
Ingenieria en Sistemas Digitales y Robotica
Tecnologico de Monterrey
Laboratorio de Arquitectura de computadoras A00819877

Rodolfo Cuan
Ingenieria en Sistemas Digitales y Robotica
Tecnologico de Monterrey
Laboratorio de Arquitectura de computadoras A01233155

1. Introducción

Las Unidades Aritmetica-Logicas (ALUs) son una pieza fundamental para el correcto funcionamiento de los microprocesadores modernos. Fueron introducidas por primera vez por John von Neumann en 1945[1]. y hoy en día todos los procesadores comerciales cuentan con múltiples ALUs, las cuales se utilizan para ejecutar operaciones básicas aritméticas (sumas y restas) y lógicas (AND, NOT, XOR, OR, XNOR) en unidades enteras representadas en binario. Para realizar estas operaciones se introducen dos operandos a la unidad y una señal que indica que operación se desea realizar. Como salida se tiene el resultado y diversos bits extras dependiendo de la arquitectura en la que se implementa la ALU. Algunas posibles banderas de salida pueden ser:

- Carry: Bit extra al realizar una suma.
- Negativo: Para indicar que el resultado de la operación es negativo.
- Overflow: Para indicar que se excedió el limite al realizar la operación.
- Paridad: Para indicar si es par o impar el resultado.

El objetivo de esta practica es desarrollar e implementar un ALU con 8 operaciones. Estas operaciones son especificadas por la practica y descritas en la Sección 2.

2. Desarrollo

El ALU fue desarrollado en VHDL en la plataforma ISE design, integrada en la computadora virtual proporcionada. El objetivo fue trabajado siguiendo las especificaciones en el manual de practicas [2].

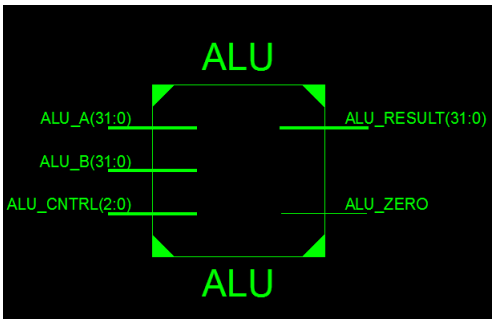

Figure 1: Schematico RTL del ALU.

El esquema del ALU desarrollado es presentado en Fig. 1. Como se puede observar cuenta con 3 entradas y 2 salidas. Las operaciones son de 32 bits por lo que las entradas A y B, y la salida Result, tienen esta longitud. La tercera entrada de 3 bits, CTRL, tiene como finalidad controlar cual de las 8 operación se va a concretar. Las operaciones son las mostradas en la Tabla 1. Las operaciones And, Or, Add y Subtract fueron implementadas utilizando los operadores correspondientes. Para la operación de Mov, simplemente se igualo la señal A al resultado. La salida de ALU_ZERO se activa cuando el resultado calculado es igual a 0.

En VHDL se opto por utilizar una solución concurrente, con 3 sentencias WITH-SELECT. La principal (Listing 1) es la que controla todas, la cual asigna el resultado a una señal auxiliar. El resultado depende de la señal ALU_CNTRL.

```
with ALU_CNTRL select
        aux_result <=
        ALU_A and ALU_B when "000",
        ALU_A or ALU_B when "001",
        ALU_A + ALU_B when "010",
        ALU_A when "011",
        ALU_B(15 downto 0)& x"0000" when "100",
        ALU_A - ALU_B when "110",
        set_less_than_aux when others;
```

Listing 1: Código VHDL WITH ALU_cntrl SELECT

La segunda sentencia WITH-SELECT (Listing 2), es utilizada para asignar el resultado de la ultima operación, Set Less Than, con un 0 o 1. Esto se realizo independientemente de la sentencia principal, puesto que el resultado del operador lógico <, regresa un valor booleano y se necesita un valor en bits. Este resultado es guardado en una señal llamada *set_less_than_aux*, e utilizada en la sentencia principal. La ultima sentencia (Listing 3) es para activar la salida de ALU_ZERO.

```
with aux_result select
        ALU_ZERO <=
        '1' when x"00000000",
        '0' when others;
```

Listing 2: Código VHDL WITH aux_result SELECT

```
with ALU_A < ALU_B select
        set_less_than_aux <=
        x"00000001" when True,
         x"00000000" when False;
```

Listing 3: Código VHDL WITH ALU_A <ALU_B SELECT

ALU_CTRL	Operación	Descripción
000	and	Result = A and B
001	or	Result = A or B
010	add	Result = A + B
011	mov	Result = A
100	B upper	Result = B[15:0]&"0000"
110	Substract	Result = A - B
111	Set Less Than	*Result = A <B

TABLE 1: Descripción de Operaciones

3. Resultados y Discusión

Para comprobar el funcionamiento adecuado de la ALU se realizo un TestBench que cubriera todas las operaciones. Para la primera operación de AND se incluyeron dos casos 0x00 AND 0x00 y 0xAA AND 0x55. Cabe recalcar que en ambos testcases el resultado de la operacion es 0x00, y el bit de ALUZero se encuentra prendido también.

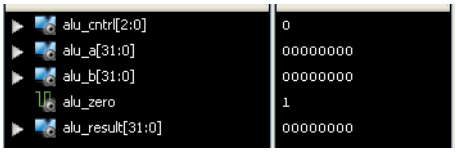

Figure 2: Primer caso AND ALU.

Segundo set de operandos:

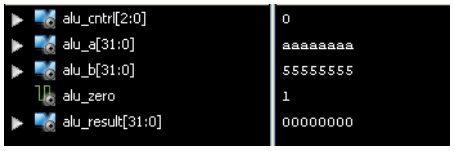

Figure 3: Segundo caso AND ALU.

Para el caso de OR se reutilizaron los operandos 0xAA, 0x55 debido a que tienen los bits intercalados por lo que es fácil calcular el resultado deseado: 0xFF.

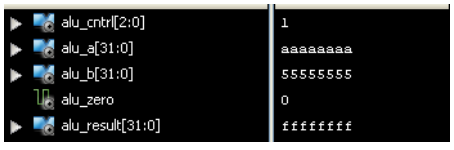

Figure 4: OR ALU.

Para la función de ADD se probo un caso sencillo de 0x01 + 0x01 dando como resultado 0x02:

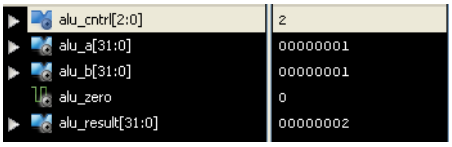

Figure 5: ADD ALU.

Posteriormente se probo MOV A con un valor diferente al de B y al del resultado previo, para observar que si se reflejara A en la salida correctamente.

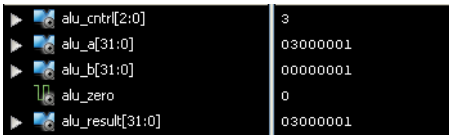

Figure 6: MOV ALU.

La siguiente operacion coloca los bits menos significativos del operando "B" y los coloca en los bits mas significativos del resultado.

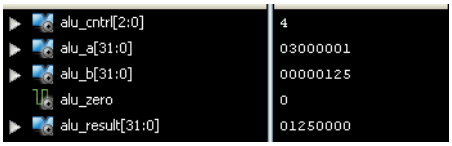

Figure 7: BUPPER ALU.

Para comprobar la operación de resta se introdujo 0x03 - 0x01 y el resultado de 0x02 fue el esperado:

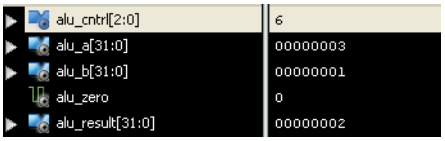

Figure 8: SUBSTRACT ALU.

Finalmente se probaron dos escenarios para Set Less than, cuando A si es menor a B. Cuando B es igual a A, por lo tanto el resultado es 0 y el bit de ALUZERO también se prende:

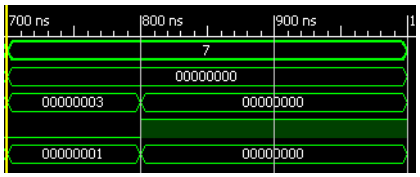

Figure 9: SET LESS THAN A ¡ B ALU.

4. Conclusión

4.1. Paul Vazquez

Con este ejercicio implementamos una pieza clave en la arquitectura general del MIPS. Logramos realizar el componente utilizando solamente circuitos concurrentes, lo que

resulto en mayor facilidad para probar que todo funcionara bien e introdujo menos errores en el código. Comparado con los componentes realizados en la practica anterior, fue mucho mas sencillo debido a la experiencia que obtuvimos y el hecho de que solamente es un solo modulo.

4.2. Rodolfo Cuan

En esta practica se introdujo uno de los componentes mas importantes en los procesadores, la ALU. Como se pudo ver en la implementación en VHDL, el componente, no es complejo. Utilizando los test benches planteados se pudo comprobar el correcto funcionamiento de cada una de las operaciones. Gracias a la practica anterior, el desarrollar el proyecto y los test benches fue mucho mas sencillo y rápido.

References

[1] Philip Levis. *"Jonathan von Neumann and EDVAC"*. cs.berkeley.edu, 2004.

[2] Diego Fernando Valencia Martínez. *Manual del laboratorio*. Escuela de Ingeniería y Ciencias Departamento de Ciencias Computacionales, 2020.