ALU Sparc v8

TRABAJO FINAL CIRCUITOS LÓGICOS PROGRAMABLES (CLP)

Autor:
Ing. Iriarte Fernandez, Nicolás Ezequiel (NicolasIriarte95@gmail.com)

Docente:
Nicolás Alvarez.

Este documento fue realizado en el curso Circuitos Logicos Programables el 11 de Abril de 2024, cuarto bimestre.

Índice

Registros de cambios

Revisión	Detalles de los cambios realizados	Fecha
0	Creación del documento.	11 de Abril de 2024
1	Se aplican cambios sugeridos por Salamandri Santiago.	20 de noviembre de 2023

Documentos anexos

Ref.	**Nombre**	**Descripción**
AD.01	SPARCV8AM	Especificación de requerimientos de software.

Cuadro 1. Documentos anexos.

Glosario

Acronimo	**Definición**
ALU	Unidad Aritnética Lógica.
CLP	Circuitos Lógicos Programables.
ICC	Integer Condition Code.
SPARC	Scalable Processor Architecture.

Cuadro 2. Glosario.

Introducción

En el presente documento se detallarán los aspectos relacionados con el desarrollo e implementación del trabajo final de la matería "Circuitos Lógicos Programables (CLP)". El cual consiste en una ALU de la arquitectura Sparc V8 tal como se describe en **AD.01**.

Desarrollo, alcance y limitaciones

El desarrollo del trabajo final se realizaró en el lenguaje de descripción de hardware VHDL. Y se implementaó una ALU de 32 bits que soportará las operaciones aritméticas y lógicas básicas. Durante el planeamiento y desarrollo se intentó representar la arquitectura Sparc V8 de forma fiel, sin embargo, para limitar el alcance del trabajo final se decidieron hacer las siguientes simplificaciones:

- El componente no tiene clock asociado, por lo que la actualización de cualquier de sus entradas genera una salida.
- Se implementaron unicamente las siguientes instrucciones, respetando el formato de la trama tal como en el manual de arquitectura (**AD.01 Pag. 43**). Las instrucciones implementadas son:
  - **ADDcc (Opcode: 010000)**: Adición con actualización de ICC.
  - **SUBcc (Opcode: 010100)**: Resta con actualización de ICC.
  - **UMULcc (Opcode: 011010)**: Multiplicación sin signo con actualización de ICC.
  - **SMULcc (Opcode: 011011)**: Multiplicación con signo con actualización de ICC.
- Para simplificación, el mecanismo de ventanas descrito en el manual de arquitectura (**AD.01 Pag. 27**) no fue implementado. Y para recuperar los valores de algunos registros requeridos, se pasaron directamente como **señales** al componente.

A partir de estas simplificaciones, el componente "ALU" que se implementó tiene la siguiente estructura interna:

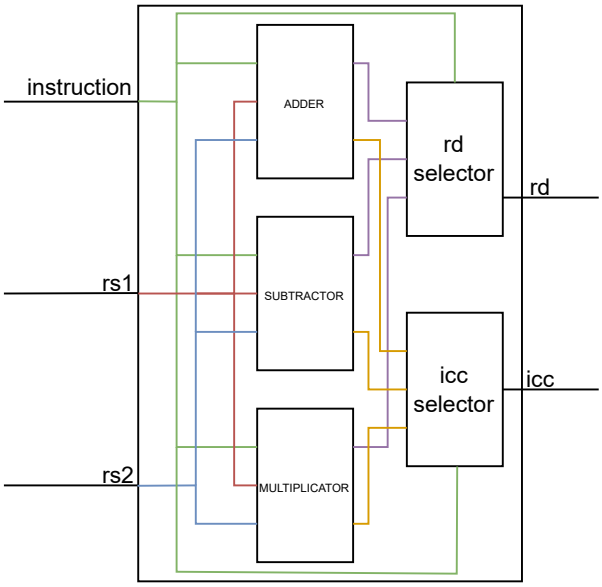

Figura 1. Diagrama de componentes.

Tal como se observa en la Figura 1, el componente “ALU” tiene tres entradas:

1. **instruction**: Señal de 32 bits que contiene la instrucción a ejecutar. La instrucción se decodifica y se extraen los campos necesarios para la ejecución de la operación. Para mayor información en el formato de la instrucción ver **AD.01**.
2. **rs1**: Señal de 32 bits que contiene el valor del registro fuente **RS1**.
3. **rs2**: Señal de 32 bits que contiene el valor del registro fuente **RS2**.

Y dos salidas:

1. **rd**: Señal de 32 bits que contiene resultado de la operación a ejecutar.
2. **icc**: Señal de 4 bits que contiene el valor del Integer Condition Code (ICC) actualizado. Para mayor información en el formato del ICC ver **AD.01 Pag. 28**.

Como filosofía de diseño, se optó por tenér un sub-componente para cada una de las instrucciones soportadas por la ALU, permitiendo de esta manera generar código VHDL más segmentado y limpio.

Dicho diseño tiene la desventaja de que todas las entradas y salidas de cada sub-componente están conectadas entre sí, y a demás, se debe agregar un selector por cada uno de las salidas esperadas.

El selector se encargará de interpretár la instrucción recibida, extraerá su Opcode y dependiendo de este último, seleccionará que sub-componente redireccionar como salida del mismo.

Simulaciones

Durante el desarrollo del presente trabajo, se realizaron multiples simulaciones de distintas entradas e instrucciones para verificar el correcto funcionamiento de los componentes implementados. Dichas entradas buscaron verificar distintos aspectos, tales como correcto funcionamiento de los ICC, busqueda de overflows, correcto calculo e interpretación de valores negativos. A continuación se deja un grafico generado a partir de la simulación del test-bench generado:

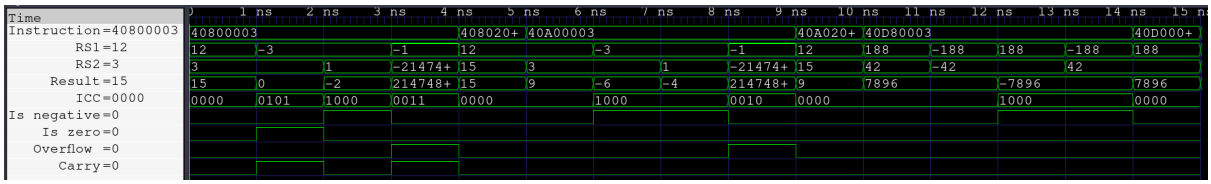

Figura 2. Simulación en Gtk-Wave.

Resumen del proyecto

Tras su desarrollo e implementación, se pudo sintetizar las operaciones deseadas. Generando de esta manera el siguiente resumen por parte de Vivado:

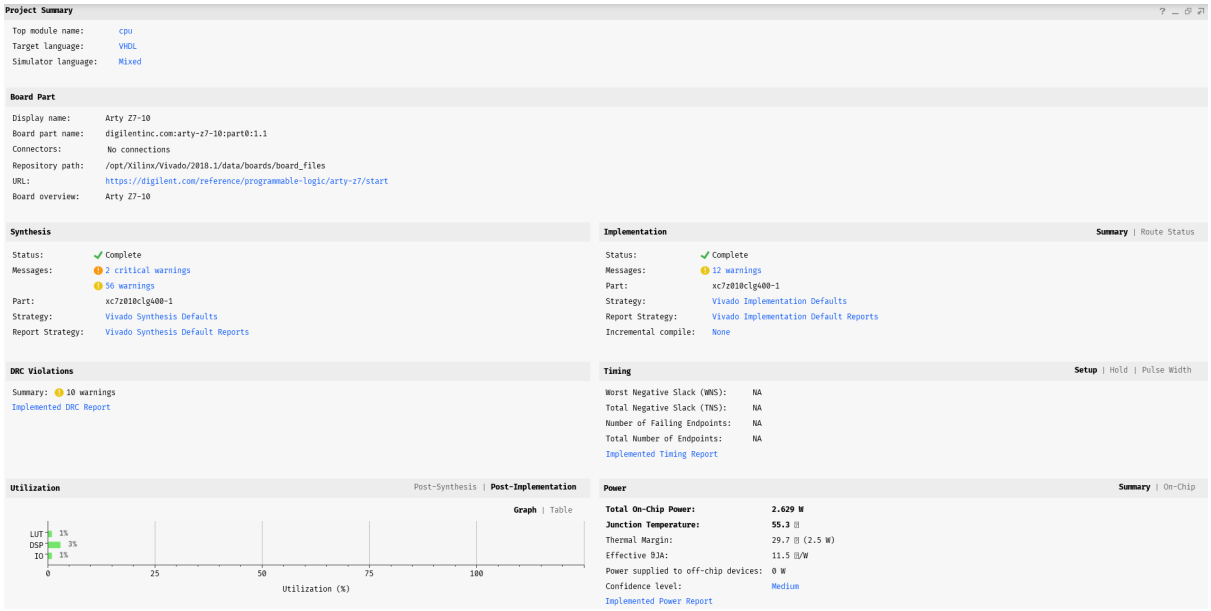

Figura 3. Resumen del proyecto por parte de la herramienta Vivado.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	18	17600	0.10
DSP	2	80	2.50
IO	1	100	1.00

Figura 4. Tabla de utilización de recursos.

Anexos

5.1. Ejemplo instrucción ADDcc

Dicha trama se explica en mayor detalle en el documento **AD.01** en la página 106 y 170. Tenér en cuenta que se hicieron algunas simplificaciones, tal como solo implementar las variaciones en donde el ICC es actualizado.

opcode	*op3*	*operation*
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

Format (3):

10	rd	op3	rs1	i=0	*unused(zero)*	rs2
31	*29*	*24*	*18*	*13*	*12*	*4* *0*

10	rd	op3	rs1	i=1	simm13
31	*29*	*24*	*18*	*13*	*12* *0*

Figura 5. Formato de trama en instrucción “ADDcc”.

Add Instructions

```
operand2 := if (i = 0) then r[rs2] else sign_extend(simm13);

if (ADD or ADDcc) then
      result ← r[rs1] + operand2;
else if (ADDX or ADDXcc) then
      result ← r[rs1] + operand2 + C;
next;
if (rd ≠ 0) then
      r[rd] ← result;
if (ADDcc or ADDXcc) then (
      N ← result<31>;
      Z ← if (result = 0) then 1 else 0;
      V ← (r[rs1]<31> and operand2<31> and (not result<31>)) or
            ((not r[rs1]<31>) and (not operand2<31>) and result<31>);
      C ← (r[rs1]<31> and operand2<31>) or
            ((not result<31>) and (r[rs1]<31> or operand2<31>))
);
```

Figura 6. Comportamiento de la instrucción “ADDcc”.