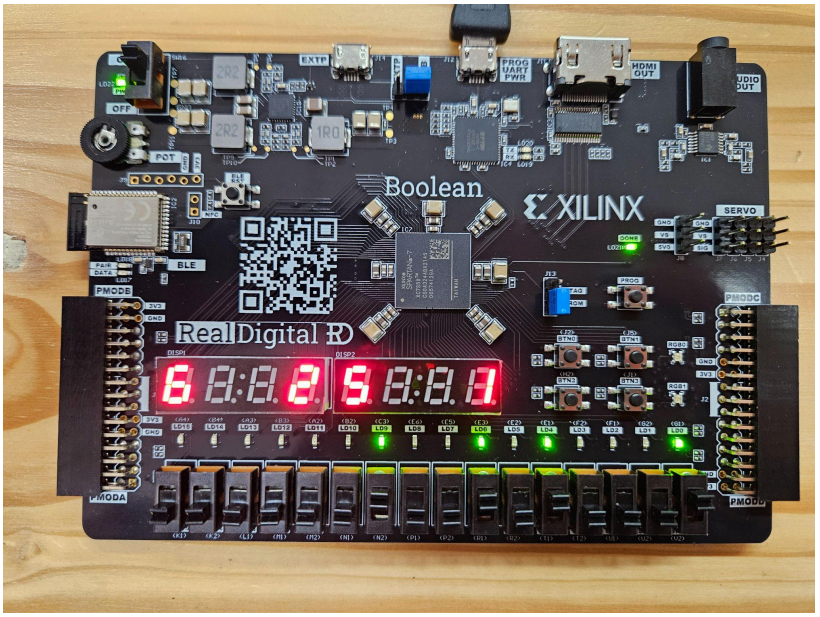
Laboratorio N°1 - Diseño e Implementación de una ALU de 4 bits

Objetivos

- Comprender el funcionamiento de una Unidad Aritmética-Lógica (ALU) elemental.
- Integrar diferentes módulos lógicos y aritméticos de forma estructural.
- Comprender el concepto de diseño jerárquico en sistemas digitales.
- Utilizar el entorno de desarrollo de AMD Vivado para la creación, simulación y síntesis de diseños digitales.
- Verificar la funcionalidad de los módulos implementados mediante simulación.
- Implementar el diseño en una plataforma basada en FPGA (AMD Boolean Board).

Recursos necesarios

- Software AMD Vivado instalado (REQUERIDO para el día de la experiencia).
- Proyecto Vivado preconfigurado con la estructura del *top level* de la ALU para ser utilizado como template (Provisto por la cátedra en repositorio GIT)
- Documento “Tutorial de Iniciación a SystemVerilog y Vivado” (disponible en AULA VIRTUAL)
- Placa AMD Boolean Board: El día de la experiencia presencial se proveerá una placa por grupo. De ser posible, cada grupo debe traer al menos un cable microUSB (NO USB-C) para conectarla a la computadora.



1. Introducción

La Unidad Aritmética-Lógica (ALU) es un componente fundamental de cualquier procesador. Es la encargada de realizar las operaciones aritméticas y lógicas sobre los datos referidos por cualquier instrucción de assembler. Una ALU típica recibe 2 operandos y una señal de control que especifica la operación a realizar, produciendo un resultado a su salida. Además, una ALU suele generar "flags" o indicadores que señalan ciertas propiedades del resultado, como sí hubo un acarreo, si el resultado es cero o negativo. En esencia, es un

circuito combinacional que permite a la CPU realizar los cálculos fundamentales. En el presente laboratorio se propone, haciendo uso de los conceptos de diseño de lógica combinacional y bloques de escala media de integración, el diseño de una ALU elemental de 4 bits y su implementación circuital mediante el uso de una plataforma de FPGA.

Una FPGA (Field-Programmable Gate Array) es un circuito integrado (o “chip”) que puedes configurar después de que se fabrica. Para entender su funcionamiento, imagina que tienes una gran cantidad de bloques elementales que se configuran para implementar funciones lógicas. A su vez, existe una gran estructura de interconexión entre estos bloques que puedes utilizar para combinarlos y generar funciones lógicas más grandes y complejas. A diferencia de un microprocesador que tiene una función fija (la de ejecutar un programa), una FPGA puede ser "reconfigurada" para convertirse en casi cualquier circuito digital que desees, desde un simple controlador hasta un complejo sistema de procesamiento. Esto las hace muy versátiles para prototipos, investigación y aplicaciones especializadas donde se requiere hardware personalizado y de alto rendimiento.

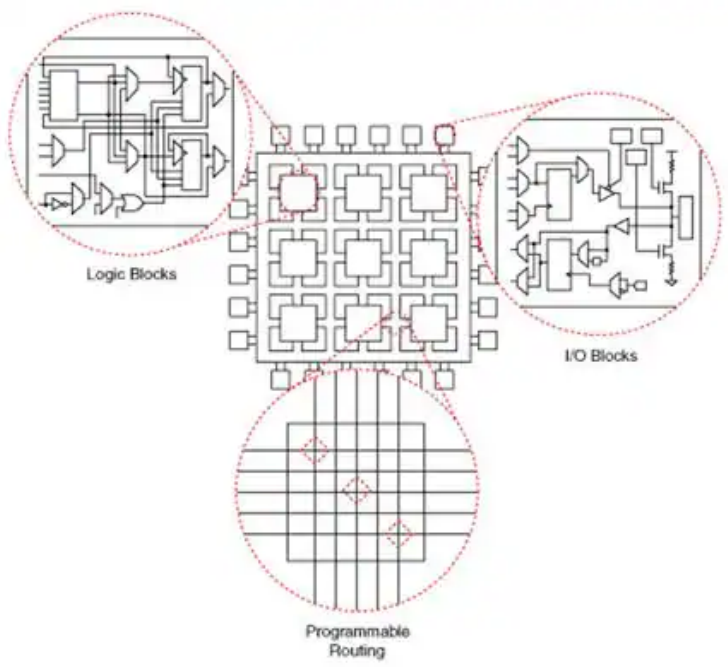


Fig. 1 - Arquitectura de una FPGA

A nivel técnico, una FPGA es un circuito integrado basado en una matriz de bloques lógicos configurables (CLBs, Configurable Logic Blocks) interconectados mediante una red de conexión programable (ver Fig. 1). Cada CLB típicamente contiene elementos lógicos combinacionales (compuertas) y elementos secuenciales (flip-flops D), permitiendo la implementación de funciones lógicas complejas. La configuración de estos bloques y sus interconexiones se define a través de un bitstream cargado en la memoria de configuración del dispositivo. Este bitstream se genera mediante el uso herramientas de síntesis y place-and-route (Vivado en nuestro caso) a partir de la descripción de un circuito. Para esta descripción es necesario acudir al uso de lenguajes de descripción de hardware (HDLs) como VHDL o Verilog. Esto no debe ser un problema para nosotros! Ya verán que la sintaxis es muy intuitiva y los ayudaremos con ejemplos y con el armado del proyecto.

2. Arquitectura de la ALU de 4 Bits

El esquema de la figura 2 ilustra el bloque de la ALU de 4 bits que desarrollaremos.

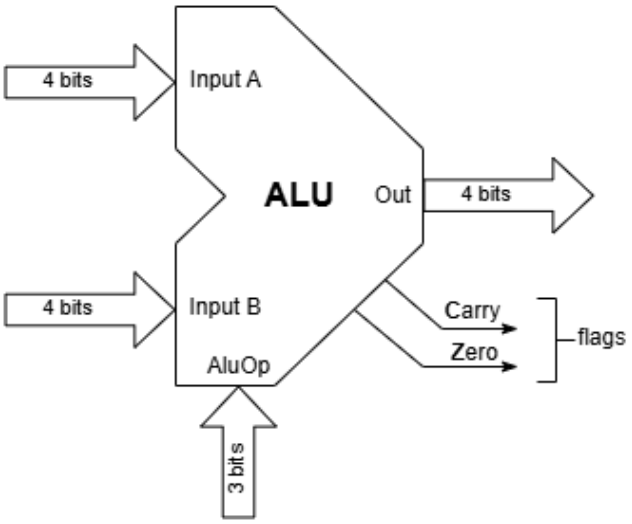


Fig. 2 - Diagrama de señales de la ALU de 4 bits

Analicemos cada una de sus señales:

- **Entradas de Operandos (Input A y B - 4 bits):** La ALU recibe dos operandos de entrada, **A** y **B**, ambos representados como números binarios de **4 bits**. Estos operandos son los datos sobre los cuales la ALU realiza las operaciones.
- **Salida de Resultado (Out - 4 bits):** La ALU produce un único resultado de **4 bits** en su salida. Este valor representa el resultado de la operación aritmética o lógica realizada sobre los operandos A y B.
- **Entrada de Control (AluOp - 3 bits):** La ALU no realiza una única operación, sino que es capaz de ejecutar diversas funciones dependiendo de una señal de control. En nuestra ALU, esta señal de control se denomina **AluOp** y está compuesta por **3 bits**. Estos 3 bits permiten seleccionar hasta 8 operaciones diferentes ($2^3 = 8$) que la ALU podrá realizar. En este laboratorio, desarrollaremos una ALU capaz de realizar las siguientes operaciones:
    - AND Lógico bit a bit entre dos números binarios de 4 bits.
    - OR Lógico bit a bit entre dos números binarios de 4 bits.
    - Suma aritmética de dos números binarios de 4 bits.
    - Resta aritmética de dos números binarios de 4 bits.
    - Pass input b del número binario de 4 bits (Out = Input B).

- **Banderas de Salida (flags):** Además del resultado principal, la ALU también genera una serie de señales de salida conocidas como **flags** o **banderas de estado**. Estas banderas proporcionan información adicional sobre el resultado de la operación. Nuestra ALU deberá generar las siguientes banderas:
    - **Cero (Z):** Esta bandera se activa (por alto a '1') si el resultado de la operación (la salida `Out`) es igual a cero (en binario: `0000`), caso contrario, la bandera vale '0'.
    - **Carry Out (C):** Esta bandera se activa cuando una operación aritmética (como la suma) genera un acarreo fuera del bit más significativo. Por ejemplo, al sumar dos números de 4 bits, si el resultado excede los 4 bits, se genera un carry out.

El desarrollo de esta ALU se realizará siguiendo un enfoque de **diseño jerárquico**. Esto significa que la ALU estará compuesta por módulos internos de menor jerarquía, cada uno encargado de realizar una función específica más básica que la anterior. Este enfoque facilitará su implementación mediante lenguajes de descripción de hardware. A continuación se muestra la arquitectura interna propuesta.

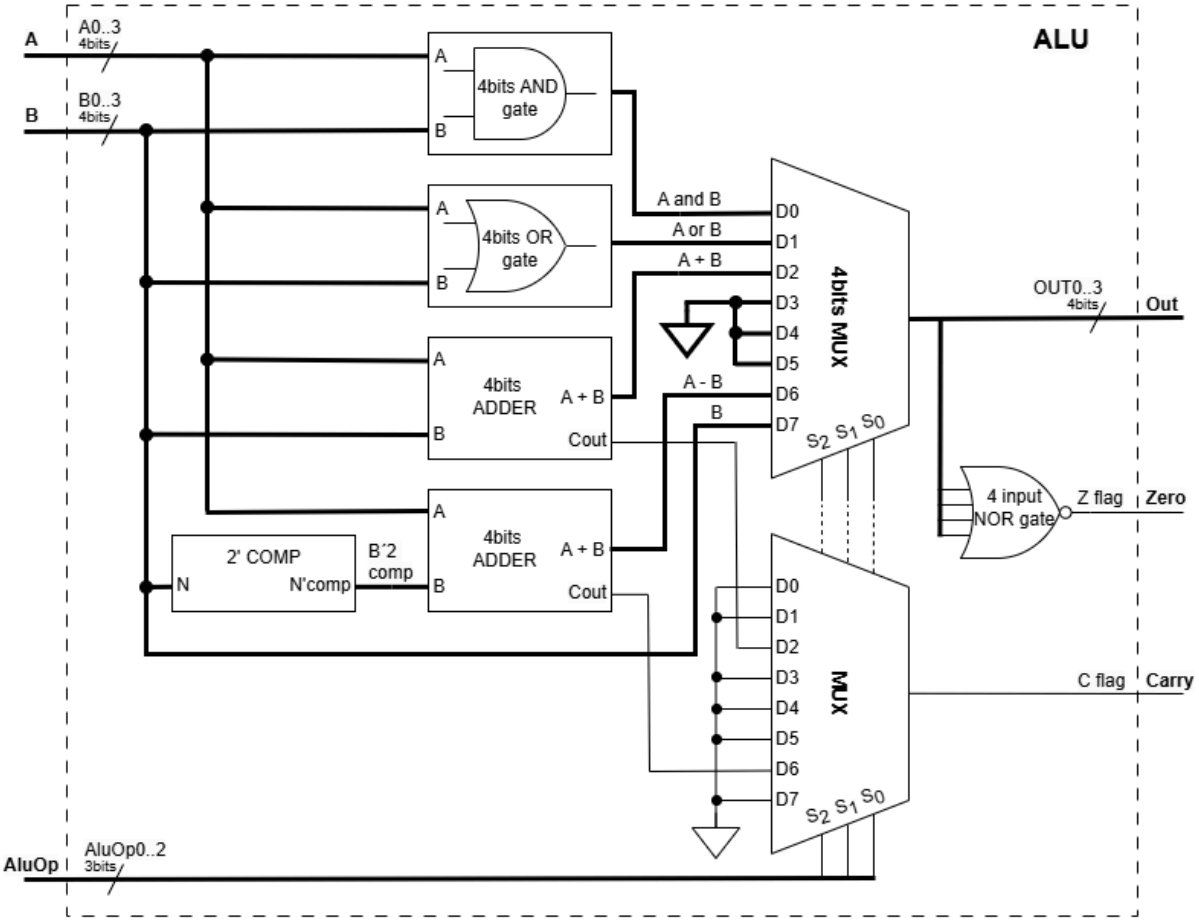


Fig. 3 - Arquitectura interna de la ALU de 4 bits

En el diagrama de la figura 3, se hace uso de un bloque sumador de 4 bits (*4bits ADDER*). Este puede implementarse a partir de 4 sumadores completos de un 1 bit en cascada, tal como se muestra en la figura 4. El bloque del complemento a 2 puede inferirse fácilmente a partir de sumadores y compuertas lógicas.

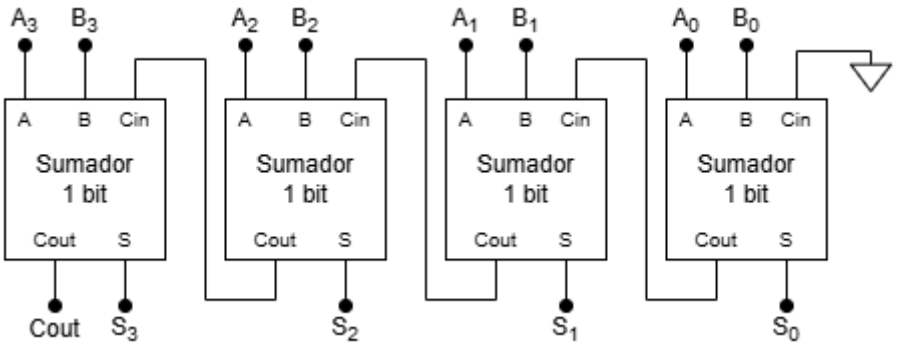


Fig. 4 - Implementación de un sumador de 4 bits a partir de sumadores completos de 1 bit

3. Estructura del Proyecto Vivado (Template)

Considerando que este es el primer contacto que Uds tienen con este tipo de herramientas de diseño (herramientas de síntesis de circuitos), hemos armado un proyecto base que utilizarán para la realización del laboratorio. Este proyecto contiene archivos de configuración conforme a la placa que usaremos y una estructura base de archivos de descripción de los circuitos a implementar, los cuales deberán modificar y completar a lo largo de las distintas actividades. Antes de abrir y comenzar a editar el proyecto, les recomendamos que lean el “Tutorial de Iniciación a SystemVerilog y Vivado” disponible en la sección del Lab 1 del aula virtual.

El proyecto cuenta con tres directorios principales:

- `constrs`: Que contiene los constrains de la placa, **este archivo NO debe ser modificado.**
- `source`: Contiene los archivos de descripción de hardware que modelan los circuitos que conforman la ALU en lenguaje Systemverilog (.sv). Algunos circuitos deben ser diseñados y escritos por ustedes en las distintas actividades.
- `sim`: Donde están los archivos de test, **estos NO se deben modificar** pero serán utilizados para verificar el correcto funcionamiento de la ALU por simulación.

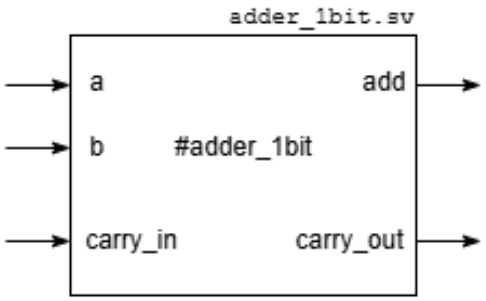
En el directorio de `source` hay cuatro archivos principales:

- `basic_gates.sv`: contiene la descripcion de compuertas lógicas básicas de dos entradas: AND, OR, XOR; de una entrada: NOT; y algunos circuitos de escala media de integración como multiplexores 2x1, 4x2 y 8x3. **Este archivo no debe modificarse,** sin embargo es interesante que lo revisen para familiarizarse con la sintaxis de SystemVerilog. Los módulos aquí contenidos representan los bloques elementales que cuentan para realizar el resto de las actividades.

- `adder_1bit.sv`: Donde se deberá describir el circuito de la actividad 1: un sumador de 1 bit utilizando las compuertas básicas contenidas en `basic_gates.sv`.
- `alu.sv`: Donde se deberá describir la estructura de la ALU utilizando las compuertas básicas y el sumador completo de 1 bit.
- `top.sv`: Es un archivo que instancia la ALU (`alu.sv`) y la interconecta con los recursos de la placa como leds, interruptores, display, etc. **Este archivo no debe modificarse.**
- `7seg_controller.sv`: Implementa el controlador del display de 7 segmentos de la placa. **Este archivo no debe modificarse.**

4. Tareas a realizar

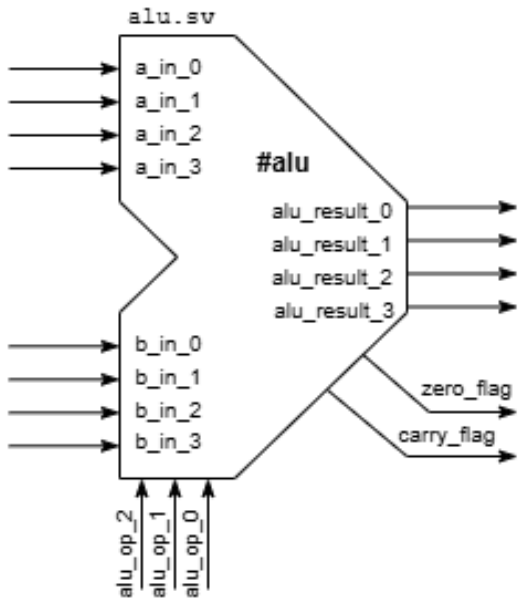
ACTIVIDAD #1: Implementación de un sumador completo de 1 bit




```
module adder_1bit(
      input a,
            input b,
      input carry_in,
      output add,
      output carry_out
);
```

A. Diseñar el circuito sumador completo de un bit a partir del uso EXCLUSIVO de compuertas básicas (OR, AND, XOR, etc.). Para esto utilizar los recursos de diseño aprendidos de circuitos combinacionales: (tabla de verdad, diagrama de Karnaugth, función simplificada y su implementación con compuertas básicas). También puede abordarse como la combinación de dos semisumadores y compuertas lógicas.
B. Editar el archivo `adder_1bit.sv` de la carpeta `src` para describir estructuralmente el circuito obtenido en el punto A. Para esto, instanciar la cantidad de módulos necesarios de compuertas lógicas contenidas en `basic_gates.sv` y las señales de conexión internas (Ejemplo: `logic nombre_cable;`)

ACTIVIDAD #2: Implementación de la ALU sin resta




```
module alu(
      input a_in_0,
      input a_in_1,
      input a_in_2,
      input a_in_3,
      input b_in_0,
      input b_in_1,
      input b_in_2,
      input b_in_3,
      input alu_op_0,
      input alu_op_1,
      input alu_op_2,
      output alu_result_0,
      output alu_result_1,
      output alu_result_2,
      output alu_result_3,
      output zero_flag,
      output carry_flag
);
```

A. Revisar el diseño de la ALU propuesto en la Fig 6, y completar el diseño estructural para la totalidad de las señales. Como se muestra en la figura, se debe utilizar solo los recursos disponibles: compuertas básicas de dos entradas (OR, AND, XOR, NOT etc.) y el bloque de sumador completo de un bit. Prestar especial atención a la señal de control **AluOp** que determinará qué operación realizará la ALU, como se describe en la siguiente tabla:

AluOp	Operación
000	AND
001	OR
010	suma
110	reservado resta
111	pass input b

Como en este caso la operación “resta” está excluida del diseño, las señales reservadas para esta operación deben ir conectadas a ‘0’.

B. Editar el archivo `alu.sv` de la carpeta `src` para describir estructuralmente el circuito obtenido en el punto A. Para esto, instanciar la cantidad de módulos necesarios de compuertas lógicas, multiplexores y sumadores. Recordar definir además todas las señales de conexión internas (Ejemplo: `logic nombre_cable;`)

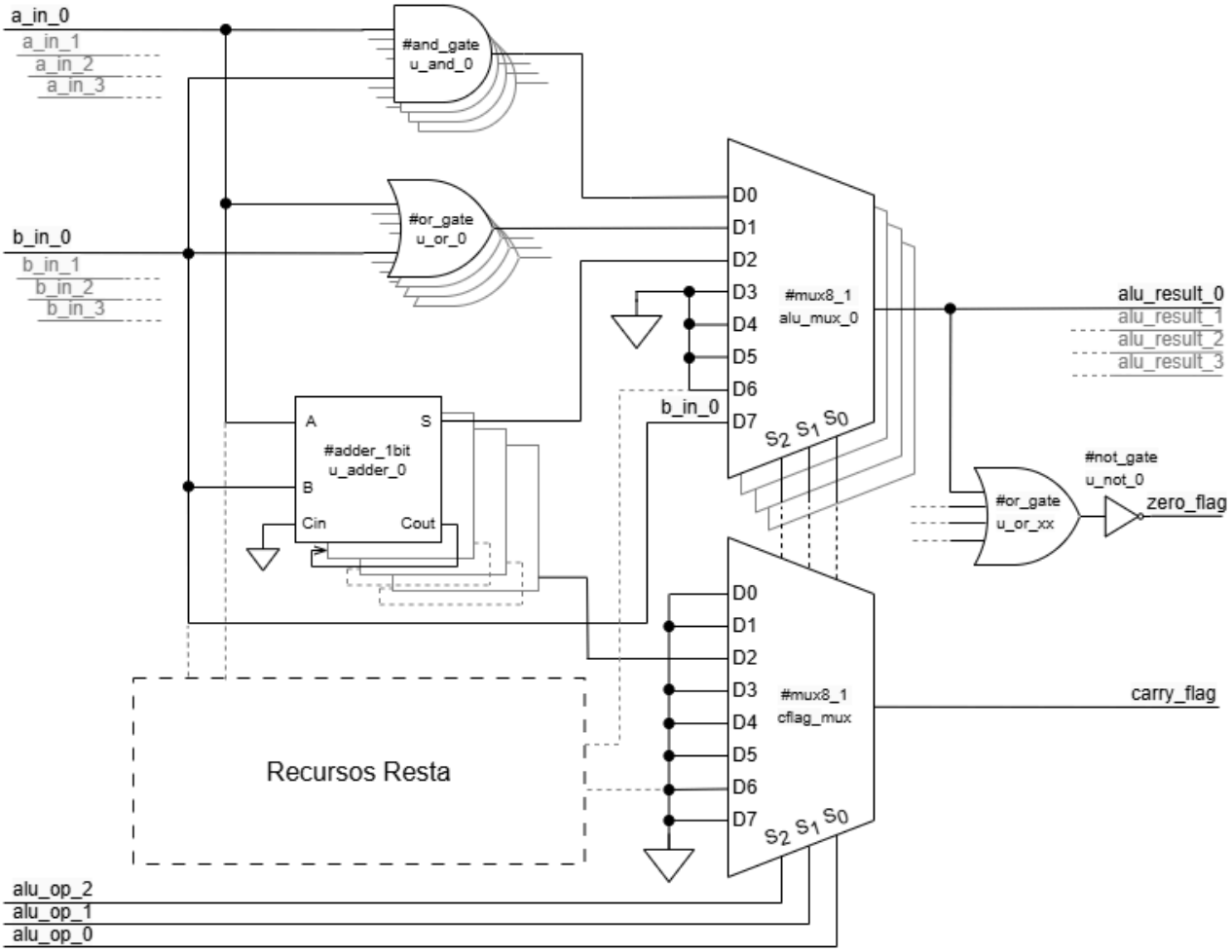


Fig. 5 - Diagrama estructural de la ALU de 4 bits SIN resta

ACTIVIDAD #3: Modificación de la ALU para implementar la operación de resta

A. Analizar la figura 3 de esta guía y proponer un diseño a incluir en la ALU de la actividad anterior, para permitir que la misma sea capaz de realizar la operación de resta (espacio reservado como “Recursos Resta” en el diagrama de la figura 5). Una vez diagramada la solución a implementar, realizar el diseño estructural para la totalidad de las señales y su incorporación al resto del circuito de la ALU. Se debe utilizar solo los recursos disponibles: compuertas básicas de dos entradas (OR, AND, XOR, NOT etc.) y el bloque de sumador completo de un bit. Prestar especial atención a las señales que se reservaron en el diseño de la actividad anterior para la resta.
B. Editar el archivo `alu.sv` de la carpeta `src` para describir estructuralmente el circuito obtenido en el punto A. Para esto, instanciar la cantidad de módulos necesarios de compuertas lógicas, multiplexores y sumadores. Recordar definir además todas las señales de conexión internas (Ejemplo: `logic nombre_cable;`)

NOTA: Al final de esta guía, en ANEXO 1, se encuentra una implementación de resta mediante el uso del complemento a 2. NO la consulten inmediatamente. La intención del laboratorio es que Uds. analicen y propongan una solución propia, pero por si acaso no les fuera posible, allí tienen una pequeña ayuda.

5. Simulación y Verificación

Dado que sintetizar, implementar y generar el bitstream es un proceso que requiere bastante tiempo, se recomienda utilizar la herramienta de simulación para verificar si el funcionamiento del circuito es el esperado. Para esto se utiliza el archivo `top_tb.sv` que se encuentra en la carpeta `sim`, el cual genera una serie de entradas pensadas para hacer una verificación inicial (no exhaustiva) de la ALU. Sobre el diseño de la ACTIVIDAD 1 y/o 2 se debe:

a. En la siguiente tabla se muestran los distintos casos de prueba que el testbench genera. Calcular cuál es la salida esperada para cada caso.

```
            Starting ALU Testbench...
            Time    A      B    (Op)  | Result Z C
-----------------------------------------------------
            20000: 0101 & 0011 (000)  |
            30000: 0101 | 0011 (001)  |
            40000: 0101 + 0011 (010)  |
            50000: 1001 + 1001 (010)  |
            60000: 1010 & 0101 (000)  |
            70000: 1010 - 0101 (110)  |
            80000: Pass B=1011 (111)  |
            90000: Undef Op=1111 (011)|
-----------------------------------------------------
```

b. Elegir el testbench como entidad top level (En el “Tutorial de Iniciación a SystemVerilog y Vivado” se muestra como hacerlo)
c. Correr la simulación y verificar en la barra de mensajes (“Tcl Console”) si la salida del circuito es la esperada.
d. Si encuentran errores, regresar a los módulos `adder_1bit` y `alu` para depurar el código SystemVerilog y luego volver a simular.

- **Síntesis, implementación y generación de bitstream:**
  a. Ejecutar el proceso de síntesis en Vivado.
  b. Analizar los recursos utilizados (LUTs, Flip-Flops, etc.) en el informe de síntesis.
  c. Ejecutar el proceso de implementación
  d. Analizar la forma en que la herramienta ubicó los distintos recursos necesarios para implementar la ALU.
  e. Generar el bitstream, conectar la placa y programarla.

- **Verificación en circuito**
    a. Configurar los interruptores según los casos dados en la tabla del punto 5.a y verificar si el resultado obtenido es el esperado. La imagen 6 muestra el mapeo de las señales de la ALU en la placa de la FPGA.
    b. Dado que este test no es exhaustivo, proponer y verificar otros casos que no hayan sido contemplados. En caso que los resultados obtenidos no se correspondan con lo esperado, analizar la causa, corregir el código y volver a probar.

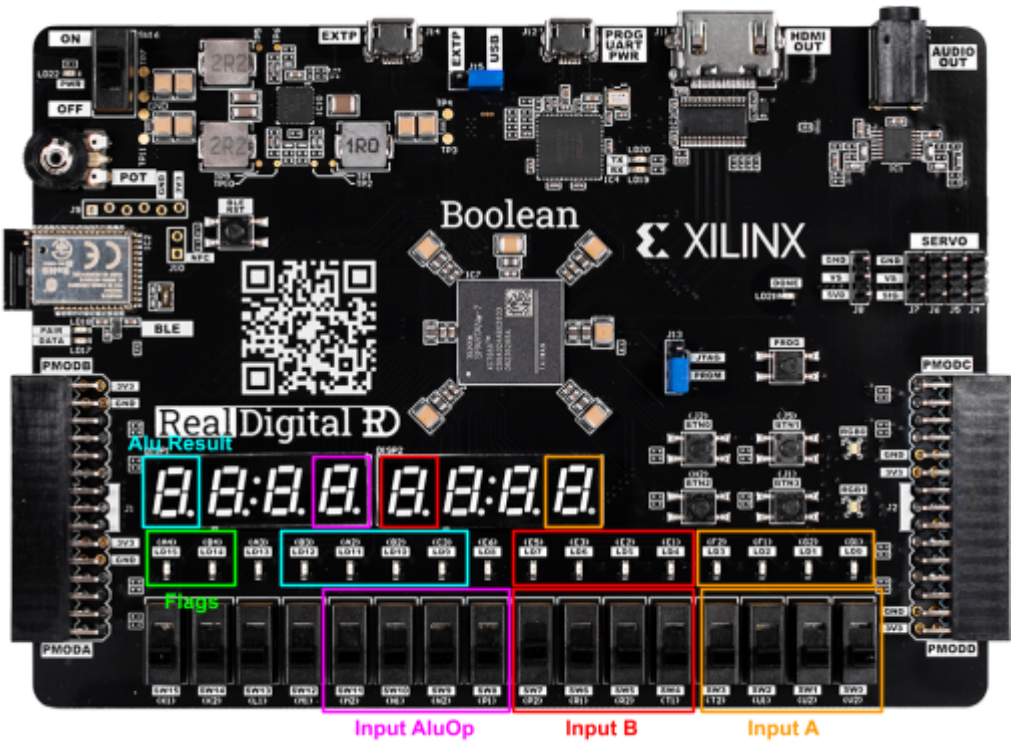


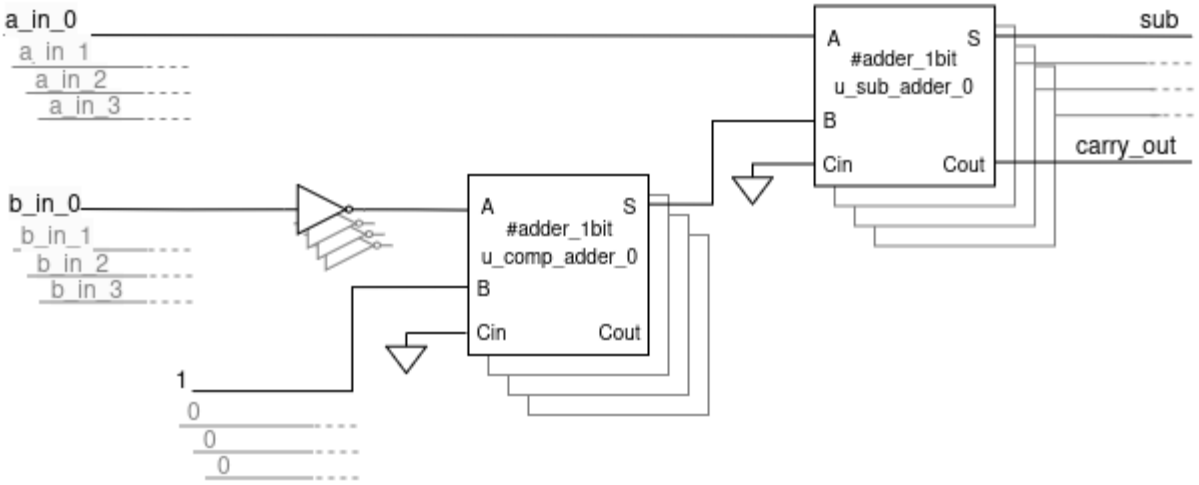
Fig. 6 - Distribución de las variables de entrada y salida en la placa de FPGA

6. Entregables

La entrega del laboratorio se hace mediante el mismo repositorio de github asignado por la cátedra. Se deben realizar commits y push del proyecto completo de forma en que todas las modificaciones queden registradas en el repositorio.

ANEXO 1: Implementación del restador de 4 bits

La actividad 2 propone implementar la resta de dos números de 4 bits utilizando el sumador de 1 bit implementado. Para esto es necesario complementar el operando B de forma en que la ALU haga la operación SUB = A + (-B). El complemento a 2 de un número binario se obtiene invirtiendo todos sus bits (complemento a 1) y luego sumándole 1, para esta última operación también se utilizarán sumadores de 1 bit. En la siguiente imagen se muestra una sugerencia de la implementación.



En el módulo `alu`, utilizar compuertas NOT para negar el operando B bit a bit, luego con sumadores de 1 bit sumarle “1” para así obtener el complemento a 2 del operando B. Finalmente, con más sumadores de 1 bit, sumarle el operando A. Notar que, a pesar que son instancias del sumador de 1 bit, en ningún momento se modifica la implementación de la operación suma ya implementada.