

Trabajo práctico: Sistemas combinacionales

Organización del Computador

Primer Cuatrimestre 2025

1. Enunciado

El trabajo práctico consistirá en desarrollar (casi) todos los circuitos combinatorios necesarios para realizar una ALU de 4 bits. Nuestra primera unidad de cómputo! Yeay!

Para la realización de este taller deberá utilizar el archivo `combinacionales.circ` provisto junto al presente enunciado.

2. Logisim Evolution *Cheatsheet*

2.1. El simulador

El simulador¹ opera en dos modos. Edición y Simulación:

- En el modo edición podremos definir el funcionamiento del circuito con todas las entradas y salidas, las compuertas lógicas o los componentes que lo componen, más el aspecto físico que tendrá el componente a la hora de ser utilizado por otros circuitos.
- El modo simulación nos permitirá testear el funcionamiento del componente, asignando valores a las entradas y comprobando el valor de las salidas.

2.2. Detalles adicionales

- a) Las entradas (*inputs*) se simbolizan con un cuadrado y tienen el comportamiento de una llave. (Prendido = 1; Apagado = 0). Las mismas cambian de estado de forma manual. Se pueden definir de más de un bit.
- b) Las salidas (*outputs*) se simbolizan con un círculo y serán “prendidas” cuando por su entrada haya un 1, o “apagadas” cuando por su entrada haya un 0. Se pueden definir de más de un bit.
- c) En caso de ser necesario que el circuito tenga una entrada fija en algún valor, podremos utilizar el componente “Wiring/Constant”.
- d) Un componente súper útil es el “Wiring/Pin”. Con este podemos generar entradas y salidas. Cuando quieran crear un componente con entradas y salidas definidas, deberán utilizar este componente. Notar el detalle que el orden gráfico de las entradas y salidas es el orden en el que se deben conectar en el circuito que utiliza dicho componente.

¹<https://github.com/logisim-evolution/logisim-evolution/>

- e) El simulador permite usar cables de más de un bit. Para ello, pueden utilizar el componente “Wiring/Splitter”, que permite unir o separar cables. Además, pueden usar el Multiplexor que se encuentra en “Plexers”.

3. Ejercicios

Completar el esqueleto provisto de los componentes que se enumeran a continuación. **Sólo podrán utilizarse compuertas lógicas básicas (AND, OR, XOR, NOT), *splitters* y multiplexores. Podrán utilizar también los sub-circuitos que les proveemos resueltos en el archivo combinacionales.circ**

Deberán diseñar cada uno de los mismos de manera modular, incorporando una funcionalidad por componente de forma de poder reutilizarlos.

3.1. Sumador de 4 bits

- a) **Sumador simple de 1 bit.** El mismo tendrá dos entradas **a** y **b** y dos salidas: **S** que representa la suma de **a** y **b** y **Cout** que representará el acarreo de dicha suma.
- b) **Sumador completo de 1 bit.** El mismo tendrá tres entradas **a**, **b** y **Cin**. Este último representa el carry de entrada. Y dos salidas: **S** que representa la suma de **a** y **b** considerando el acarreo, y **Cout** que representará el acarreo de dicha suma.
- c) **Sumador de 4 bits.** El mismo tendrá dos entradas de cuatro bits **A** y **B**, que representarán los numerales a sumar y otra **Cin**, con la misma interpretación anterior. Por otra parte, el mismo tendrá dos salidas **S** (de cuatro bits), donde se deberá ver reflejado el resultado y **Cout** que representará el acarreo final de la suma.

3.2. Sumador de 4 bits con “Flags”

- a) **Comparador con Cero, de 4 bits.** El mismo tendrá una entrada **A** de cuatro bits y una salida **Z** que deberá encenderse cuando la entrada es 0000.
- b) **Sumador de 4 bits con Flags Z,C,V,N.** Extender el sumador de 4 bits creando un componente similar pero que tenga salidas que reflejen lo sucedido durante la suma binaria (bit a bit):
- el resultado es cero \Leftrightarrow **Z** vale 1
 - la suma binaria produjo acarreo \Leftrightarrow **C** vale 1
 - la suma overflow \Leftrightarrow **V** vale 1²
 - el resultado es negativo \Leftrightarrow **N** vale 1³

²Utilizar el circuito provisto, e integrarlo

³Utilizar el circuito provisto, e integrarlo

Primer acercamiento a “Números negativos”

Por el momento no vamos a trabajar con representaciones numéricas, por lo que nuestro acercamiento a los números “negativos” será muy básico.

Para la implementación de esta unidad de cómputo inicial nos interesa empezar a conocer algunas cosas básicas de representaciones. La primera es que (según la representación que nosotros vamos a estar usando mayormente en la materia, y usan los procesadores, llamada *complemento a 2*) los números negativos empiezan con un bit en ‘1’. Es decir, el bit más a la “izquierda” o bit más significativo (**msb**, según sus siglas en inglés) nos sirve para verificar el signo del número: “1” negativo, “0” positivo.

En cuanto al **Overflow**, el circuito que les proveemos lo detecta. Hablaremos más sobre esto cuando veamos **representación de la información**.

3.3. ALU de 4 bits

- a) **ALU de 4 bits**. La misma tendrá tres entradas **A**, **B** y **OP**, ésta última de dos bits. Además, tendrá cinco salidas: **S** más **Z**, **C**, **V** y **N**.

La salida deberá expresar el resultado de (con sus respectivos “Flags”):

- $A + B \Leftrightarrow \mathbf{OP}$ vale 00

- $A - B \Leftrightarrow \mathbf{OP}$ vale 01.

Aquí **C** deberá informar si la resta binaria produjo *borrow* (dame uno), para ello les proveemos un circuito que detecta algunas condiciones especiales.

Además, recordar que $A - B = A + (-B)$, siendo $-B$ el inverso aditivo de B . Encontrarán resuelto un circuito que calcula $-B$ a partir de B llamado **inversor_4**.

Responder: ¿Qué operaciones hace ese circuito sobre el número B? (el porqué de esas operaciones lo veremos también en la unidad de *representación de la información*).

- $A \text{ AND } B$ (bit a bit) $\Leftrightarrow \mathbf{OP}$ vale 10

- $A \text{ OR } B$ (bit a bit) $\Leftrightarrow \mathbf{OP}$ vale 11

En las últimas dos operaciones, los flags **C** y **V** deberán tomar el valor 0. En todos los casos los *flags* reflejan el resultado de operación entre operandos en complemento a 2.

3.4. Introducción al testing

Junto con los archivos para resolver el trabajo práctico, se encuentra el resultado esperado de las operaciones llamado `salida_catedra.txt`. La idea es que puedan escribir un pequeño **Makefile** que verifique que la solución de ustedes logra los mismos resultados. Para ello utilizaremos las “capacidades” de **Logisim-Evolution** por línea de comandos. La siguiente línea⁴ ejecuta **logisim-evolution** y *loguea* la salida en una tabla, veamos:

```
$ logisim-evolution --toplevel-circuit "verificador" -tty\  
table combinacionales.circ > out.txt
```

La línea mostrada corre **Logisim-Evolution** diciéndole:

⁴notar que dado que la línea es muy larga fue cortada con el caracter “\”

- El diseño principal que queremos correr es el **verificador**
- Tiene que correrlo por consola (no GUI): `--tty` y que queremos le formato de tabla: `table`
- La tabla de resultados del archivo `combinacionales.circ` se escribe en el archivo `out.txt` (mediante una redirección del `standard output`)

Para entener el formato, conviene mirar el componente llamado verificador que les proveemos en el archivo `combinacionales.circ`. El mismo se encarga de generar una tabla exhaustiva con los resultados de las operaciones de suma, resta, AND, OR, con la ALU de 4 bits que ustedes deben implementar. **No modificar el componente verificador.**

El formato de la tabla que genera (que es el mismo formato que tiene el archivo `salida_catedra.txt`) es el siguiente:

A	B	OP	N	Z	V	C	res
0000	0000	00	0	1	0	0	0000
0001	0000	00	0	0	0	0	0001
...
1111	1111	11	1	0	0	0	1111

Por otro lado en linux/unix existe el comando `diff` de consola que compara archivos (ejecutar `man diff` en la consola para más info).

Utilizando esta información escribir un pequeño `Makefile` que verifique automáticamente si la solución realizada arroja los resultados adecuados. Es decir, que compare la el archivo `out.txt` con el archivo `salida_catedra.txt` y muestre un mensaje si son iguales o diferentes.

Por otro lado, notar que existe un archivo llamado `.labfiles` que en este caso contiene la lista de archivos que *no deben modificar*. Si los modifican, el PR será rechazado automáticamente.