

# Sistemas Digitales

Tomás Agustín Hernández



# 1. Introducción a los sistemas de representación

## Magnitud

Llamamos magnitud al tamaño de algo, dicho en una medida específica. Es representada a través de un sistema que cumple 3 conceptos fundamentales:

- Finito: Debe haber una cantidad finita de elementos.
- Composicional: El conjunto de elementos atómicos deben ser fáciles de implementar y componer.
- Posicional: La posición de cada dígito determina en qué proporción modifica su valor a la magnitud total del número.

Algunos de los sistemas de representación más utilizados son: binario, octal, decimal y hexadecimal.

## Bases

Una base nos indica la cantidad de símbolos que podemos utilizar para poder representar determinada magnitud.

Base	Símbolos disponibles
2 (binario)	0, 1
8 (octal)	0, 1, 2, 3, 4, 5, 6, 7
10 (decimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16 (hexadecimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Tabla 1: Bases más utilizadas

La tabla anterior representa los símbolos disponibles para las bases 2, 8, 10 y 16.

Consideremos por un momento que estamos en binario; ¿sería correcto que  $1 + 1 = 2$ ? ¡No! Porque 2 no es un símbolo válido en base 2.

Para indicar la base en la que está escrito un número, se coloca la base entre paréntesis en la esquina inferior derecha.

$1024_{(10)}$ : 1024 representado en base 10 (decimal)

**Importante:** 1 dígito hexadecimal son 4 bits en binario.

**Importante 2:** Cuando vemos  $0x123456$  en RISC-V solo se considera 123456 si necesitamos hacer cualquier operación, el  $0x$  indica solamente que está en hexadecimal.

## Dígitos/Bits

Sea  $n \in \mathbb{Z}$ , cuando decimos que tenemos  $n$  bits es lo mismo que decir que tenemos  $n$  dígitos.

- 0001: Representa el número 1 en binario, en 4 bits/dígitos.
- 0010: Representa el número 2 en binario, en 4 bits/dígitos.

## Teorema de división

Es una manera de poder realizar un cambio de base de un número decimal a otra base. La representación en la otra base es el resto visto desde abajo hacia arriba.

$$a = k * d + r \text{ con } 0 \leq r < |d|$$

donde:

- $k$  = cociente
- $d$  = divisor.
- $r$  = resto de la división de  $a$  por  $d$ .

Pasaje del número  $128_{(10)}$  a  $128_{(2)}$  en 8 bits

$$\begin{aligned}128 &= 64 * 2 + 0 \\64 &= 32 * 2 + 0 \\32 &= 16 * 2 + 0 \\16 &= 8 * 2 + 0 \\8 &= 4 * 2 + 0 \\4 &= 2 * 2 + 0 \\2 &= 1 * 2 + 0 \\1 &= 0 * 2 + 1\end{aligned}$$

Luego,  $128_{(2)} = 1000\ 0000$

### Bit más significativo / menos significativo

El bit más significativo en un número es el que se encuentra a la izquierda, mientras que el menos significativo es el que se encuentra a la derecha.

$$\textcolor{blue}{1}000000\textcolor{blue}{0}_{(2)}$$

### Tipos numéricos

Representemos números naturales y enteros a partir de la representación en base 2 (binario)

**Sin signo:** Representa únicamente números positivos. No se pueden utilizar los símbolos de resta (-) ni tampoco coma (,)

$$\begin{aligned}1_{(10)} &= 01_{(2)} \\128_{(10)} &= 10000000_{(2)}\end{aligned}$$

**Signo + Magnitud:** Nos permite representar números negativos en binario. El bit más significativo indica el signo

- 0: número positivo
- 1: número negativo.

$$\begin{aligned}18_{(10)} &= \textcolor{blue}{0}0010010_{(2)} \\-18_{(10)} &= \textcolor{blue}{1}0010010_{(2)}\end{aligned}$$

Representar números en S+M suele traer problemas porque el 0 puede representarse de dos maneras

$$\begin{aligned}+0_{(10)} &= \textcolor{blue}{0}0000000_{(2)} \\-0_{(10)} &= \textcolor{blue}{1}0000000_{(2)}\end{aligned}$$

Para solucionar este problema, las CPU utilizan la notación Complemento a 2 ( $C_2$ )

**Exceso m:** Sea  $m \in \mathbb{Z}$ , decimos que un número  $n$  está con exceso  $m$  unidades cuando  $m > 0$

$$\begin{aligned}n_0 &= n + m \\n = 1 \wedge m = 10 &\longrightarrow n_0 = -9\end{aligned}$$

Nota:  $n_0$  indica el valor original de  $n$  antes de ser excedido  $m$  unidades.

**Complemento a 2:** Los positivos se representan igual.

El bit más significativo indica el signo, facilitando saber si el número es positivo o negativo. Cosas a tener en cuenta

- **Rango:**  $-2^{n-1}$  hasta  $2^{n-1} - 1$
- **Cantidad** de representaciones del cero: Una sola
- **Negación:** Invierto el número en representación binaria positiva y le sumo uno.
  - $-2_{(2)} = \text{inv}(010) + 1$
  - $-2_{(2)} = 101 + 1$
  - $-2_{(2)} = 110$
- **Extender número a más bits:** Se rellena a la izquierda con el valor del bit del signo.
- **Regla de Desbordamiento:** Si se suman dos números con el mismo signo, solo se produce desbordamiento cuando el resultado tiene signo opuesto.

## Overflow / Desbordamiento

Hablamos de overflow/desbordamiento cuando

- El número a representar en una base dada, excede la cantidad de bits que tenemos disponibles.
- Si estamos en notación  $C_2$  al sumar dos números cambia el signo.

## Acarreo / Carry

Ocurre cuando realizamos una suma de números binarios y el resultado tiene más bits que los números originales que estamos sumando

## Borrow

Borrow  $\equiv$  Préstamo.

Se produce cuando el sustraendo (denominador) es mayor que el minuendo (numerador) por lo tanto se debe pedir.

```
1      0111
2      - 1000
3      ----
4      1011
5
6      El primer cero le pide al uno de su derecha.
```

## Suma entre números binarios

Se hace exactamente igual que una suma común y corriente.

Es importante prestar atención a la cantidad de dígitos que nos piden para representarlo, y en caso de estar en  $C_2$  que el signo no cambie.

Hagamos sumas en  $C_2$  (sin límite de bits)

$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$	$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline \textcolor{blue}{1}0011 = 3 \end{array}$	$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline \textcolor{blue}{1}1001 = -7 \end{array}$	$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline \textcolor{red}{1}110 = \text{Overflow} \end{array}$	$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline \textcolor{blue}{1}\textcolor{red}{0}110 = \text{Overflow} \end{array}$
---	---	---	--	---

Nota: El color azul indica el carry; El rojo indica qué es lo que produce overflow (cambio de signo).

Hagamos sumas en  $C_2$  (límite de bits: 4)

$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$	$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline \textcolor{blue}{1}0011 = \text{Overflow} \end{array}$	$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline \textcolor{blue}{1}1001 = \text{Overflow} \end{array}$	$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline \textcolor{red}{1}110 = \text{Overflow} \end{array}$	$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline \textcolor{blue}{1}\textcolor{red}{0}110 = \text{Overflow} \end{array}$
---	---	--	--	---

Nota: Al tener un límite de 4 bits, en las sumas que tenemos carry terminamos teniendo overflow.

## Resta entre numeros binarios

La resta entre números binarios debe realizarse en  $C_2$

La idea es que  $A - B \equiv A + (\text{INV}(B) + 1)$

## ¿Cuándo hay overflow, carry o borrow?

- POS + POS = NEG
- NEG + NEG = POS
- POS - NEG = NEG
- NEG - POS = POS

```

1      0101
2      + 0110
3      ====
4      1011 - OV: si - C: no
5
6      1000
7      + 1100
8      ====
9      10100 - OV: si - Carry si
10
11     1010
12     - 0100
13     ====
14     1000 - OV: si - Borrow no
15
16     0100
17     - 1000
18     ====
19     1000 - OV: si - Borrow si

```

## Rango de valores representables en n bits

Sean  $n, m \in \mathbb{Z}$  decimos que el rango de representación en base  $n$  y  $m$  bits acepta el rango de valores de:  $[-n^m, n^m - 1]$   
 ¿Es posible representar el 1024 en binario y 4 bits? No.

- $2^4 = 16 \implies [-16, 15]$
- Pero,  $1024 \notin [-16, 15]$
- Por lo tanto, 1024 no es representable en 4 bits.

## Pasar número binario a decimal

1. Si tenemos el mismo número todo el tiempo podemos usar la serie geométrica

¿Qué número decimal representa el número  $111111111_{(2)}$ ?

$$\sum_{i=0}^{j-1} 1 \cdot n^i = \frac{q^{n+1} - 1}{q - 1} \text{ Luego,}$$

$$\sum_{i=0}^9 1 \cdot 2^i = 2^{10} - 1 = 1023$$

2. Si no tenemos el mismo número todo el tiempo podemos multiplicar cada dígito por la base donde el exponente es la posición del bit.

$$10_{(2)} = 1 * 2^1 + 0 * 2^0 = 2_{(10)}$$

## Extender un número de n bits a m bits

Sea  $n, m \in \mathbb{Z}$  donde  $n$  es la cantidad de bits inicial y  $m$  es la cantidad a la que se quiere extender.

$$n = 3 \wedge m = 8$$

- Signo + Magnitud y exceso m: Se extiende con 0's luego del signo.
  - En 3 bits, -2 = 110
  - En 8 bits, -2 = **10000010**
- Complemento 2 ( $C_2$ ): Se extiende con el bit más significativo.
  - En 3 bits, -2 = 110
  - En 8 bits, -2 = **11111110**

## Cambios de base

Sea  $n, m \in \mathbb{Z}$  dos bases distintas, para pasar de base  $n$  a base  $m$  se debe realizar el siguiente proceso

- Pasar el número a base decimal.
- Aplicar el teorema de división utilizando la base deseada.

Encontremos en base 5, el número que corresponde a  $17_{(8)}$ :

- $17_{(8)} = 1 * 8^1 + 7 * 8^0 = 15_{(10)}$
- Usando ahora el teorema de división
  - $15 = 3 * 5 + 0$
  - $3 = 0 * 5 + 3$
  - Luego,  $30_{(5)}$
- Por lo tanto,  $17_{(8)} = 30_{(5)}$

## 2. Desplazamientos

Utilizamos los desplazamientos para poder mover los bits. Cada casillero representa los bits.

- Desplazamiento hacia la izquierda: Se desplazan los bits del dato tantas posiciones como se indiquen a la izquierda.  
 $variable \ll cantidad$

Posición	$v_3$	$v_2$	$v_1$	$v_0$
$a$	1	0	1	0
$c = a \ll 2$	1	0	0	0

- Desplazamiento lógico hacia la derecha: Se aplica desplazando los bits del dato tantas posiciones como se indiquen a la derecha.

$variable \gg_l cantidad$

Posición	$v_3$	$v_2$	$v_1$	$v_0$
$a$	1	0	1	0
$c = a \gg_l 2$	0	0	1	0

- Desplazamiento aritmético hacia la derecha: Se aplica desplazando los bits del dato tantas posiciones como se indiquen a la derecha, pero copiando el valor del bit más significativo.

$variable \gg_a cantidad$

Posición	$v_3$	$v_2$	$v_1$	$v_0$
$a$	1	0	1	0
$c = a \gg_a 2$	1	1	1	0

### 3. Operaciones lógicas

- OR (+):  $(1, 0), (0, 1), (1, 1) = 1$
- AND (\*):  $(1, 1) = 1$
- XOR ( $\oplus$ ):  $(1, 0), (0, 1) = 1$

### 4. Circuitos combinatorios

#### Negación

Sea  $p$  una variable proposicional, el opuesto de  $p$  lo escribimos como  $\bar{p}$ .

$$p = 1 \iff \bar{p} = 0$$

#### Propiedades para operaciones lógicas

Propiedad	AND	OR
Identidad	$1.A = A$	$0 + A = A$
Nulo	$0.A = 0$	$1 + A = 1$
Idempotencia	$A.A = A$	$A + A = A$
Inverso	$A.\bar{A} = 0$	$A + \bar{A} = 1$
Conmutatividad	$A.B = B.A$	$A + B = B + A$
Asociatividad	$(A.B).C = A.(B.C)$	$(A + B) + C = A + (B + C)$
Distributividad	$A + (B.C) = (A + B).(A + C)$	$A.(B + C) = A.B + A.C$
Absorción	$A.(A + B) = A$	$A + A.B = A$
De Morgan	$\overline{A.B} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}.\bar{B}$

#### Operaciones booleanas

Se resuelven utilizando las propiedades para operaciones lógicas

$$\text{Verifique si son equivalentes } (X + \bar{Y} = (\bar{X} * Y) * Z + X * \bar{Z} + (\bar{Y} + \bar{Z}))$$

- $\overline{\bar{X} * Y} * Z + X * \bar{Z} + (\bar{Y} * \bar{Z}) \implies \text{De Morgan}$
- $(X + \bar{Y}) * Z + \bar{X} * \bar{Z} + (\bar{Y} * \bar{Z}) \implies \text{De Morgan} \wedge \text{Distributiva}$
- $(X + \bar{Y}) * Z + \bar{Z} * (X + \bar{Y})$
- $(X + \bar{Y}) * (\bar{Z} + \bar{Z}) \implies \text{Inverso}$
- $(X + \bar{Y}) * 1 \implies \text{Identidad}$
- $(X + \bar{Y})$

Nota: También se pueden probar equivalencias utilizando tablas de verdad

#### Funciones booleanas

- AND =  $A * B$
- OR =  $A + B$
- NOT =  $\bar{A}$

#### Tablas de verdad

Nos permiten observar todas las salidas para todas las combinaciones de entradas dada una función. Veamos un ejemplo con una función F:



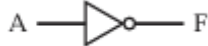



$$\text{Sea } F = X + \bar{Y}$$

Protip: El símbolo de + indica OR porque  $1 + 0 = 1$  mientras que el símbolo AND indica \* porque  $1 * 0 = 0$

X	Y	F
1	1	1
1	0	1
0	1	0
0	0	1

## Compuertas

Son modelos idealizados de dispositivos electrónicos que realizan operaciones booleanas.

Nombre	Símbolo gráfico	Función algebraica	Tabla verdad															
AND		$F = A \cdot B$ or $F = AB$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	0	1	0	0	1	1	1
A	B	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = A + B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	1
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$F = \overline{A}$ or $F = A'$	<table><tr><th>A</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	F	0	1	1	0									
A	F																	
0	1																	
1	0																	
NAND		$F = \overline{(AB)}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = \overline{(A + B)}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	0	1	0	0	1	1	0
A	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$F = A \oplus B$	<table><tr><th>A</th><th>B</th><th>A XOR B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	A XOR B	0	0	0	0	1	1	1	0	1	1	1	0
A	B	A XOR B																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

Nota:  $XOR = \oplus$  Nota: Estas compuertas devuelven una única salida. Imaginemos que tenemos solamente NAND ¿Como podemos conseguir una AND? Aplicando una NAND luego de la otra. Enlace a [Electronic HyperPhysics](#)

## Compuertas Universales

Nos permiten obtener otros operadores.

- $NAND = \overline{A \wedge B}$

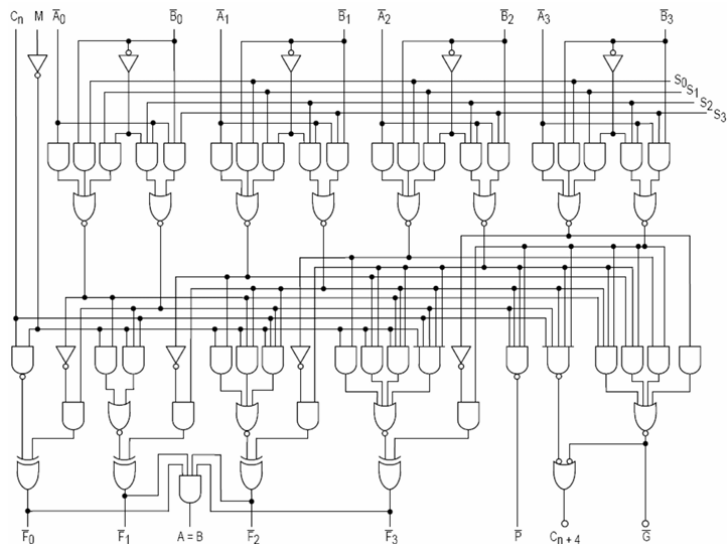


- $NOR = \overline{A \vee B}$
- $XNOR = \overline{A \oplus B} = \text{Si son iguales es V}$

## Compuertas en SystemVerilog

- $A \text{ AND } B \ (A * B) = (\text{assign } O = A \& B)$
- $A \text{ OR } B \ (A + B) = \text{assign } A | B$
- $A \text{ XOR } B = \text{assign } A \wedge B$
- $NOT \ A \ (\bar{a}) = (\sim A)$

## Caja Blanca / Caja Negra en Circuitos



(a) Caja Blanca



(b) Caja Negra. 16 indica los bits de entrada & salida

Nota: Ov indica Overflow

## Entradas / Salidas de un circuito

Se representan con flechas. En SystemVerilog se llaman input y output.

```
module ALU #(parameter DATA_WIDTH = 16)
  (input [DATA_WIDTH-1:0] operandoZ,
   input [DATA_WIDTH-1:0] operandoY,
   input [2:0] opcode,
   output [DATA_WIDTH-1:0] salidas,
   output overflow);
end module;
```

Nota: En las ALU no son funcionalmente iguales ni las entradas ni las salidas.

## Entradas y salidas: Datos vs Control

- Datos: Indican lo que tratamos de transformar.
  - Entrada: Registro Z y Registro Y.
  - Salida: El resultado de la operación
- Control: Indican como transformamos los datos.
  - Entrada: Enviar ADD, Enviar AND, Enviar XOR, ...
  - Salida: Ov

## Mecanismo de Traducción fórmula a circuito

Llamaremos  $\phi$  a una fórmula proposicional cualquiera

- 1. Solo consideramos de la función F, las filas verdaderas.
- 2. Cada fila verdadera tendrá su índice, y en ese índice estarán los valores de cada variable proposicional. Representamos a esa fila verdadera como  $t_i$
- 3. Realizamos la conjunción de todas las variables de ese  $t_i$
- 4. Realizamos la disyunción de todas las conjunciones de  $t_i$

Un ejemplo:

$$\text{Sea } F = X + \bar{Y}$$

X	Y	F
1	1	1
1	0	1
0	1	0
0	0	1

- F es solamente verdadera en la primera, segunda y tercer fila por lo tanto tenemos  $t_1$   $t_2$  y  $t_3$
- Por cada fila, hacemos la conjunción de los valores
  - $x_1 \wedge x_2$
  - $\bar{x}_1 \wedge x_2$
  - $\bar{x}_1 \wedge \bar{x}_2$
- Realizamos la disyunción de todos los  $t_i$ 
  - $(x_1 \wedge x_2) \vee (\bar{x}_1 \wedge x_2) \vee (\bar{x}_1 \wedge \bar{x}_2)$
- El resultado nos da  $\phi'$  que es una suma de productos y nos permite traducir fácilmente a un circuito combinatorio



## Carry en circuitos

El carry debe colocarse en los circuitos en la suma porque en caso de no hacerlo, se nos pierden casos.



Si eliminamos el carry se nos pierde el caso  $1 + 1$ , por lo tanto lo ideal sería que al hacer una suma nos quede así:



Nota: El carry es representado con un AND porque en la tabla de verdad solo da uno cuando  $A = 1$  y  $B = 1$ . Luego, la función Sum es un XOR.

## Sumadores

Los sumadores nos sirven para poder realizar operaciones entre bits. Es importante recalcar que llamamos half-adder a un sumador de 1 bit, donde solamente tiene una entrada A de 1 bit y una entrada B de 1 bit. Un sumador de 1 bit requiere:

- Dos entradas A y B de 1 bit
- Una compuerta XOR (para la suma): solo el resultado es 1 o 0
- Una compuerta AND (para el carry): si la suma del XOR es  $1+1$



Veamos un ejemplo con un sumador completo de 3 entradas: Si para dos entradas necesitabamos un sumador simple,

para 3 entradas necesito 2. Porque es  $(A+B)$  y luego  $res+C$



Nótese que para considerar si es carry al final de toda la suma o no usamos un or porque nos basta con que uno haya arrojado carry.

## Inversor

- Si me mandan  $INV=1$  entonces tengo que invertir los bits.
- La manera de hacer esto es utilizando un XOR.



## Multiplexor

Está conformado por varias entradas de control y entradas de datos. Existe una única salida.

- Entradas de control: se indican de la manera  $c_n$
- Entradas de datos: se indican de la manera  $e_n$
- Ojo: Antes de prender cualquier entrada de control, hay que asegurarse que entrada de datos esté puesta en el que valor que queremos

Te ayuda a ahorrar recursos, pero todas las acciones tardan mucho más tiempo porque solo se ejecuta una a la vez. Nos garantiza que la información solo se envía por un único canal.

Para poder calcular la cantidad de entradas de control  $c_n$  que necesito para una cantidad  $m$  de entradas de datos  $e_m$  hago el siguiente cálculo:  $m < 2^l$  hasta que me pase por primera vez.

- Entradas de datos: 30.
- Entradas de control: Necesito 5 entradas de control porque  $2^5$  es 32.

Cada entrada de control tiene un índice que podemos decirle individuo, por ejemplo, si tengo  $2^l$  entradas tengo 32 posibles combinaciones.

- Si tengo 00010 significa que la persona que está hablando la persona 2.

En los Multiplexores existen difurcaciones, que cuando llegamos a una de ellas se nos desvía el camino enviándonos a una compuerta. Si en algún momento se llega a un valor 0, entonces decimos que el camino ya finalizó.



Nótese que el multiplexor elige un camino u otro según OP; OP toma los valores de 00, 01, 10 y 11. En base a qué se envía como entrada de control el multiplexor decide qué entrada habilitar para dar la salida. En este caso, OP: 1 0 (marcado en rojo) corresponde a la opción de realizar A AND B bit a bit

## Timing

En un circuito combinatorio el tiempo que tarda la salida en estabilizarse depende de la cantidad de capas de compuertas (latencia). Para enfrentar el problema usamos secuenciales.

## Latches

Utilizan realimentación, es decir, la salida de una compuerta como entrada de otra.

### Latch RS

Tiene dos entradas: S (Set) y R (Reset), y dos salidas, Q y  $\bar{Q}$  y consiste en dos puertas NOR conectadas por realimentación. El circuito es consistente permanece estable  $\iff S = R = 0$ . Tabla de verdad del Latch



S	R	Q	$\bar{Q}$
1	0	1	0
0	1	0	1
0	0	$Q^*$	$\bar{Q}^*$ <sup>1</sup>
1	1	0	0

Funciona como un memorizador

- Cuando S está prendido entonces  $Q = 1$ .
- Cuando Q está prendido entonces  $\bar{Q} = 1$ .
- Si ninguno está prendido recuerda el estado anterior.

- Si ambos están prendidos,  $Q = 0$  y  $\bar{Q} = 0$ . Este caso no debería estar permitido porque la salida es inconsistente.

Para recordar el estado anterior usamos la notación de:  $Q^*$  y  $\bar{Q}^*$

**Importante:** El valor de las salidas depende de la implementación del latch. Por lo tanto, un Latch con NAND no sería lo mismo que un Latch con NOR.

## Latch JK

Acepta todas las combinaciones posibles de las entradas.

- Cuando J está prendido entonces  $Q = 1$ .
- Cuando K está prendido entonces  $\bar{Q} = 1$ .
- Si ninguno está prendido recuerda el estado anterior.
- Si ambos están prendidos, niega el estado anterior (¡necesita que haya un estado anterior!)

Latch JK:



Tabla de verdad:

$J$	$K$	$Q$	$\bar{Q}$
1	0	1	0
0	1	0	1
0	0	$Q^*$	$\bar{Q}^*$
1	1	$\bar{Q}^*$	$Q^*$

Cuando J y K son 1, la función realizada se denomina función de conmutación, la salida se invierte.  
El circuito oscila (estado inestable)

## Latch D

Es un almacén para un bit de datos. La salida del Latch D es siempre igual al valor más reciente aplicado a la entrada y por lo tanto la recuerda y la produce.

Tiene una entrada de datos y una de control.

Este circuito es estable en todos los estados pero los tiempos no se pueden predecir porque dependen de D y puede causar carreras si existe un lazo en el circuito externo

- Cuando D está apagado y C está prendido, se memoriza C.
- Si ambos están prendidos, se memoriza D.
- En cualquier otro caso, devuelve el valor memorizado.



$D$	$C$	$Q$	$\bar{Q}$
1	0	$Q^*$	$\bar{Q}^*$
0	1	0	1
0	0	$Q^*$	$\bar{Q}^*$
1	1	1	0

## Control de transición de estados: Clock



El clock que necesitamos utilizar es el 3ro. ¿Por qué? Porque nos interesa solamente memorizar o guardar los estados de los valores cuando el clock está en el pico.

No necesitamos estar constantemente escuchando cambios con el clock, sino que nos interesa solo en la subida.

Para solucionar este problema, podemos utilizar un detector de pulso.



(a) Detector de pulso implementado usando una compuerta AND.

Es importante notar, que el detector de pulso dará 1 (el pico) en algunos casos porque la compuerta NOT tiene un pequeño delay para poder negar la entrada. A continuación se muestra un ejemplo de esto.



La línea punteada indica el tiempo que tardó la entrada en ser negada. En ese momento es donde se nos ejecutan los picos que nosotros necesitamos.

Si mandamos input = 1, el primer momento queda 1 AND 1 y el AND es verdadero, pero luego de un momento queda 1 AND 0 y ahora la señal vuelve a estar baja. Si fuese 1 AND 1 todo el tiempo tendríamos un clock constante y no necesitamos eso.

Todas las compuertas tienen delay porque al estar compuestas de silicio, tardan un poco en entrar en calor.

Veamos un ejemplo en Logisim usando registros y una ALU con el tema este de la secuencialidad a la hora de escribir en el pico del clock.





- Al encender el clock en el primer flanco de subida la ALU recibió el valor en `ALU_A_WRITE` del `Registro_00` que expuso previamente en `Reg0_enableOut`.

En circuitos secuenciales se niega el clock para que cada operación pueda tomarse el tiempo que necesita, caso contrario pasa basura.

## Flip-Flop JK

Está armado en base a Latch JK + Clock.

El Latch JK funcionará igual pero solo memorizará el valor sii el clock está en el flanco de subida.

- Cuando J es 1 y el clock está prendido, se guarda el valor de J.
- Cuando K es 1 y el clock está prendido, se guarda el valor de K.
- Cuando el clock está apagado devuelve el valor de J/K guardado en el último pulso al clock.
- Cuando el clock está apagado y J, K = 1 se niega el resultado guardado en el último pulso al clock.
- En cualquier otro caso, sucede el ítem anterior.

## Flip-Flop D

Está armado en base a Latch D + Clock.

El Latch D funcionará igual pero solo memorizará el valor sii el clock está en el flanco de subida.

- Cuando el clock es 1, guarda el valor de D en ese instante.
- Cuando el clock está apagado, devuelve el valor de D guardado en el último pulso al clock. En criollo: Guarda el valor de D hasta que haya otro flanco de subida (ciclo) y guarde uno nuevo.

Lo podemos representar:



Tabla de verdad:

D	clk	$Q_{T+1}$	$\overline{Q_{T+1}}$
1	0	$Q_T$	$\overline{Q_T}$
0	1↑	0	1
0	0	$Q_T$	$\overline{Q_T}$
1	1↑	1	0

Siendo  $T = n \cdot T_{clock}$  y  $T + 1 = (n + 1) T_{clock}$ , donde:

- $T_{clock}$  es el período del clock (tiempo que dura un ciclo)
- $n$  es una cierta cantidad de pulsos de clock

Nota: 1 ↑ indica que solo es se evalúa cuando está en el flanco de subida.

## Registros

Para poder escribir registros necesitamos una entrada de control Enable que nos permitirá decirle al Flip-Flop cuando nosotros queremos permitir que nos cambie el valor / escriba la memoria.

Este flag de Enable deberá ir en un AND con el Clock, entonces si Enable = 1 y el Clock está en 1, entonces se le permite al Flip Flop guardar el valor de la operación realizada.



Recuerdo: El Flip Flop D solo almacena 1 bit. Si necesitáramos almacenar n bits (registro de n bits) necesitaríamos n Flip Flop D y UN solo Enable/Clock.

# Componentes de Tres Estados

Apagado, Encendido y Desconectado. Al estado Desconectado le decimos que es de Alta Impedancia y se simboliza **Hi-Z**

**Noción Eléctrica**



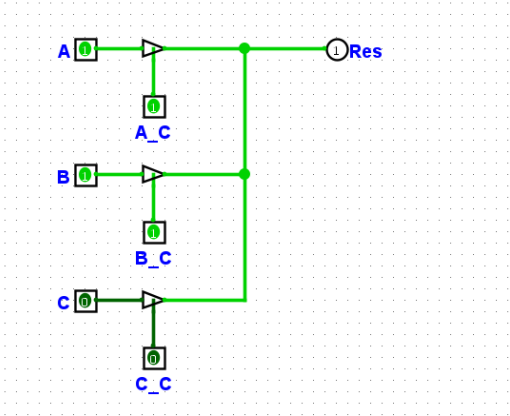
**Símbolo**



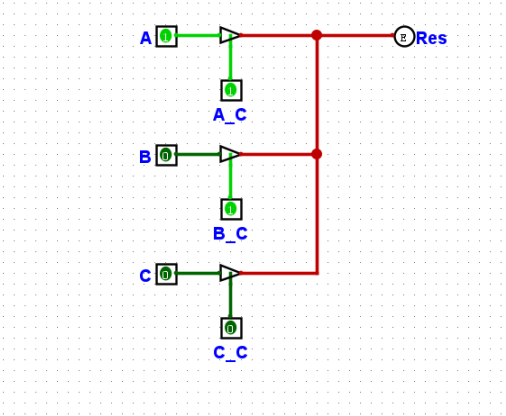
**Tabla de Verdad**

A	B	C
0	1	0
1	1	1
-	0	Hi-Z

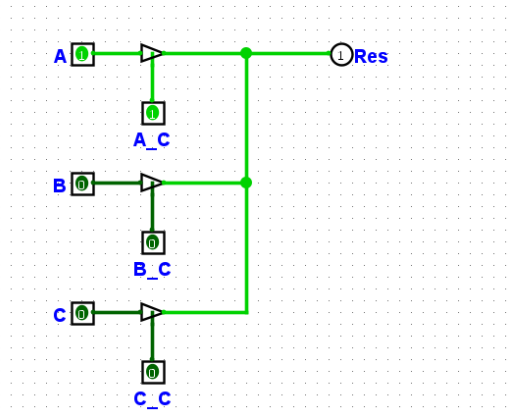
En la materia, una combinación basura de un componente de tres estados es que haya más de una entrada de control preñada con una misma entrada de dato. Esto es porque si bien en Logisim se acepta, no tendría sentido abrir dos conductos y mandar dos datos (iguales), la corriente siempre varía en algún momento y lo haría estallar.



(a) Caso basura (erróneo)



(b) Caso erróneo (explota)



(c) Caso ok

## Bus

Nos va a servir para poder conectar varios componentes. Es una vía de  $n$  bits que van a estar conectando todos los componentes de nuestra arquitectura.



Cada dispositivo sería cualquier operación, por ejemplo cada dispositivo podría ser un Flip-Flop D.

Ej.: Si  $Dispositivo_0$  tendría el número 2 escrito en 4 bits (0010) y el  $Dispositivo_1$  tendría el número 4 escrito en 4 bits (0100) entonces el bus tendría el valor de 6. Esto es un problema, porque básicamente se está haciendo una especie de conjunción de todas las cosas y no siempre vamos a querer que se haga de esa forma. Aquí aparece un concepto importante llamado Recurso Compartido.

Llamamos **recurso compartido** cuando tenemos más de un componente/dispositivo conectado en un mismo bus y necesitamos decidir quién usa cada componente.

Para prender uno de los dispositivos y no los demás, bastaría con poner en 1 el dispositivo que quiero mientras que los demás en 0.

## Reset

Coloca en 0 el componente. Comúnmente, el reset es asincrónico.

## Write Enable y Enable Out

Son dos instrucciones. No pueden pasar ambas a la vez. O escribimos, o leemos. Cada vez que hacemos el cambio de estado de Enable Out o Write Enable, en algún momento deberán volver al estado anterior.

## Esquema de interconexión de n registros

Ahora nos queda realizar el esquema



Utilizo 3 Flip-Flop D con la posibilidad de escribir cuando el clock está activo y un botón de reset  
Se añade un EnableOut para que se muestre el dato almacenado con lo armado en el paso 1

¿Como podríamos copiar el dato de R1 a R0?

- Utilizo Enable Out en R1 -  $\text{EnableOut-1} \leftarrow 1$
- Habilito WriteEnable en R0 -  $\text{WriteEnable-0} \leftarrow 1$
- Espero que el Clock esté funcionando en R0
- Deshabilito WriteEnable en R0 -  $\text{WriteEnable-0} \leftarrow 0$
- Deshabilito Enable Out en R1 -  $\text{EnableOut-1} \leftarrow 0$

De esta manera podemos pasar datos entre registros.

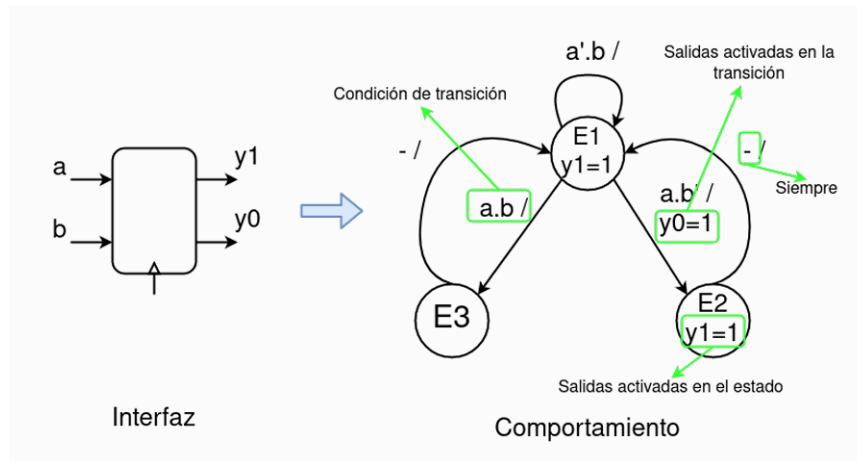
## 5. Máquinas de estado

Los circuitos secuenciales pueden ser pensados formalmente como una Máquina de Estados Finitos o FSM. Una máquina de estados queda definida por:

- Una lista de estados.
- Un estado inicial.
- Una lista de funciones que disparan las transiciones en función de las entradas.

## Diagramas de estado

Nos indican el comportamiento de los circuitos secuenciales en base al estado y como van avanzando



Nota:  $x'$  indica la variable negada.

## FSM - Moore

La salida depende solo del estado actual.

- La salida siempre cambia un clock después que se dispara la condición de transición.
- No produce glitches a la salida.
- La cantidad de estados para reproducir cierto comportamiento puede ser más grande que con otro tipo de FSM.

## FSM - Mealy

La salida depende del estado actual y las entradas.

- La salida pueda cambiar dentro del mismo clock en que se dispara la condición de transición.
- Produce glitches a la salida.
- La cantidad de estados para reproducir cierto comportamiento es más chica que en Moore.

## Lógica de próximo estado

Implementar una FSM en base a un registro de dos *bits* que siga los siguientes estados y que cada cambio se produzca al apretar un pulsador. Usando flip-flops D y compuertas básicas a elección.

Nos piden además que el componente a desarrollar cuente con una entrada de Reset.



$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	1	0	0
0	0	1	0
1	0	1	1
1	1	0	0

¿Qué valores deberían tener  $D_1$  y  $D_0$  para obtener los valores deseados en el tiempo  $t+1$ , es decir,  $Q_1(t+1)$  y  $Q_0(t+1)$ ? Usamos un flip-flop D, para que en vez de usar asignaciones dependa del estado anterior.

$$\begin{aligned}
 D_0 &= (Q_1 \cdot \bar{Q}_0) \\
 D_1 &= (\bar{Q}_1 \cdot \bar{Q}_0) + (Q_1 \cdot \bar{Q}_0) \\
 &= (\bar{Q}_1 + Q_1) \cdot \bar{Q}_0 \\
 &= 1 \cdot \bar{Q}_0 \\
 &= \bar{Q}_0
 \end{aligned}$$

## Lógica de salida

Consiste en hacer foco en como se deben vincular los estados porque muchas veces no nos es suficiente inferir comportamiento solo con la salida. Veamos un claro ejemplo donde tenemos que hacer uso de la lógica de salida porque debemos conocer el estado para poder conocer el siguiente valor. 00, 01, 11, y 10 son salidas.



Para poder decidir esto, usamos etiquetas de estado.

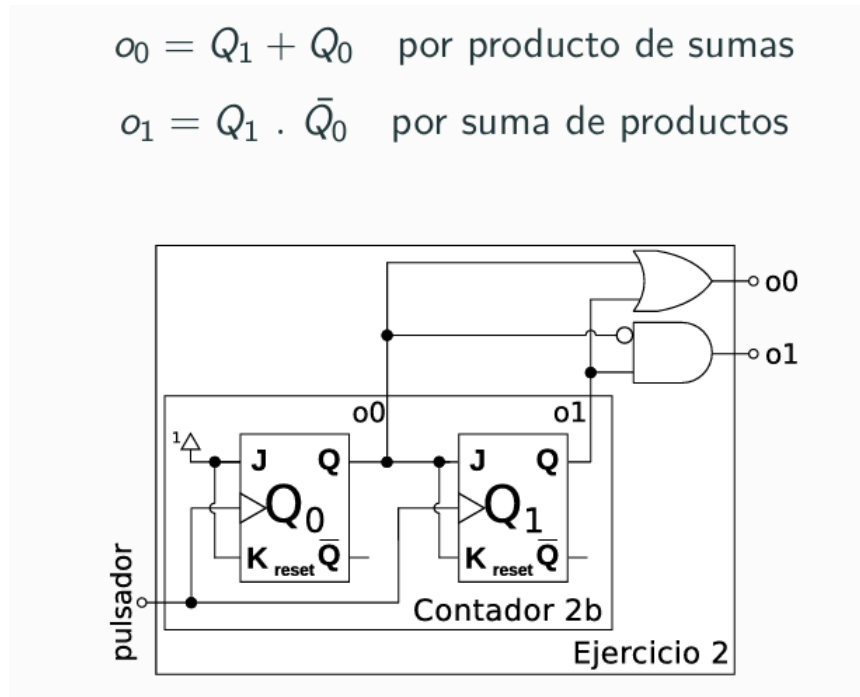
Renombremos a los estados con nombres únicos, por ejemplo:

$S_1 = 00$ ,  $S_2 = 01$ ,  $S_3 = 10$ ,  $S_4 = 11$

$Q_1$	$Q_0 \rightarrow o_1$	$o_0$
0	$0 \rightarrow 0$	0
0	$1 \rightarrow 0$	1
1	$0 \rightarrow 1$	1
1	$1 \rightarrow 0$	1

Nota:  $S_n$  n es el estado y el valor asignado es la codificación.

TODO: Preguntar como hizo para calcular la suma de productos en base a los estados.



## 6. Arquitectura

Observación: Nosotros vamos a manejarnos con 32 registros y data de 4 bytes.

¿Qué constituye una arquitectura?

- El conjunto de instrucciones
- El conjunto de registros
- La forma de acceder a la memoria

Observación: Utilizaremos la arquitectura Risc V como programa de Assembler en la materia.

Observación 2: El lenguaje ensamblador depende de cada arquitectura. Cuando un lenguaje es compilado, traduce a la arquitectura de tu equipo.

### Pasaje de lenguaje de alto nivel a bajo nivel

Para esto se necesitan programas de compilado, ensamblado y enlazado.

- El código de alto nivel es traducido por el compilador para pasar a código de bajo nivel.
- El código de bajo nivel es traducido por el ensamblador y se convierte en un código objeto (archivos **.o**).
- El código objeto es traducido por un enlazador y termina siendo binario ejecutable.

### Aclaraciones sobre RISC V

- Las operaciones que contiene son las justas y necesarias, son pocas y la idea es que los problemas se resuelvan formando expresiones atómicas.
- Las operaciones se llevan a cabo en el procesador.
- Todos los datos que se usan son de 32 bits.
- Todas las operaciones que implican accesos a memoria, los valores inmediatos se indican en bytes. Ej: lw, sw.
- Todas las operaciones que implican operaciones aritméticas o lógicas, los valores inmediatos se indican en bits. Ej: srli
- Se acostumbra a decir 32 bits = 4 bytes = 1 palabra

## Instrucciones atómicas y compuestas

Es exactamente lo mismo que en lógica. Las operaciones se separan y deben indicarse claramente como se realizan. Las instrucciones atómicas son aquellas que nos devuelven un valor irreducible, mientras que las instrucciones compuestas nos devuelven algo reducible.

## Operaciones en RISC V

En RISC-V todas las instrucciones compuestas, se reducen a instrucciones atómicas antes de devolver el resultado. Reciben el nombre de mnemónico e indica el tipo de operación que queremos realizar. Las operaciones reciben: **operandos de fuente** y un **operando destino**

$$a = b + c \equiv \text{add } a, b, c$$

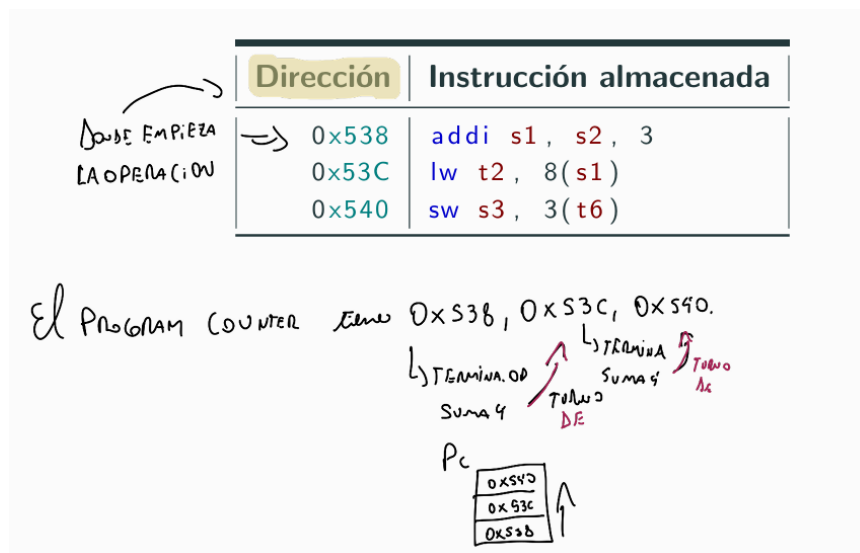
El operando destino a sería el primer parámetro del mnemónico add, mientras que los operandos fuente serían b y c.

## Comentarios en Risc V

Usamos # para comenzar una línea de comentarios.

## Program Counter

Recordatorio: Cada dirección se incrementa en múltiplos de 4 porque las instrucciones ocupan 4 bytes (1 palabra). El procesador ejecuta el programa almacenando la posición de memoria de la instrucción que se está ejecutando en un registro de 32 bits conocido como el Program Counter.



El Program Counter va cambiando la instrucción que está ejecutando gracias al aviso del procesador.

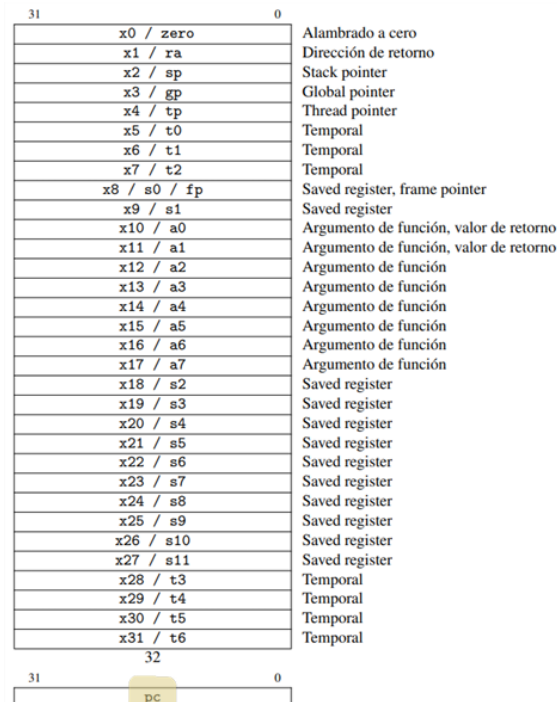
## Registros en Risc V - Register File

Viven en la CPU.

Cuenta con 32 registros que son implementados como un arreglo de memoria estática de 32 bits con varios puertos. Los registros pueden nombrarse por su índice, desde x0 a x31 o según su uso habitual.

- El registro zero (x0) almacena siempre el valor 0, y no puede ser escrito.
- Los registros s0 a s11 y los t0 a t6 se utilizan para almacenar variables: los registros pueden almacenar valores numéricos, direcciones de memoria a la RAM (punteros), punteros a funciones, etc.
- ra y de a0 a a7 tienen usos relacionados a llamadas de función.





El registro PC es el Program Counter y lleva el registro de lo que se está ejecutando en un momento dado. Cuando un programa ya termina, el PC sigue apuntando a la posición de memoria de lo último que ejecutó.  
 Nota: los valores que toman las operaciones no pueden ser de 32 bits porque la operación en sí ya ocupa 5 bits.

## Valores inmediatos

Son valores constantes que se utilizan como operandos. Se encuentran disponibles en la misma instrucción y no hace falta recuperar su valor a partir de un registro o desde la memoria.  
 El valor puede escribirse en decimal, hexadecimal (prefijo 0x) o binario (prefijo 0b).  
**Los valores inmediatos son de 12 bits y se extiende con el bit de signo a 32 bits antes de operar.**

- $\# s0 = a, s1 = b$
- $addi\ s0, s0, 4 \# a = a + 4$
- $addi\ s1, s0, -12 \# b = a - 12$

En este caso, el 4 y el -12 son extendidos a 32 bits para poder operar.

**Ojo: cuando el número inmediato a representar no está en el rango de  $[-2048, 2047]$  ( $2^{11}$  o 12 bits) las operaciones deben realizar un  $addi$  y un  $lui$  (load upper immediate).**

Véase ejercicio en [anexo](#)

## Asignación

Se hace  $zero +$  lo que queremos asignar. Consideremos que  $s0 = i$

- $addi\ s0, zero, 4 \# i = 4$

## Load Upper Immediate

Mejor conocida como  $lui$  en Risc-V. Toma los primeros 20 bits.

Es útil para cuando queremos agregar un dato que excede el rango de 12 bits, para luego combinarlo con un  $addi$  y cargar los últimos 11 bits y poder representar el número deseado.

## Valores inmediatos de 32 bits

Como en Risc-V podemos cargar valores inmediatos de 12 bits, cuando ya agregamos un bit adicional tenemos que hacer dos operaciones: un  $lui$  y un  $addi$ .

Acá se va a mostrar un ejemplo de como cargar un valor inmediato de 32 bits (excede los 12 bits).

1. Agarra los primeros 20 bits en binario. Hace un  $lui$ , es decir, de esos 20 extiende a la cantidad de bits total que es el número. En este caso, toma 20 extiende hasta 32 bits (agrega 12 bits).

2. Luego, hace un addi con la variable anterior, pero agregandole los 12 bits menos significativos.

Nótese que agarramos la parte alta y la extendemos, para despues pisar esos ceros través del addi poniendole los últimos 12 bits.

- 0xABCDE123 ¿excede los 12 bits para ser un inmediato? Pasemos a binario
  - 0001 0010 0011 0100 0101 0110 0111 1000
  - Sí, excede los 12 bits; Por lo tanto tenemos que usar un lui y un addi para cargarlo.
- De ese binario gigante, hago un lui de los 20 más significativos & addi de los 12 menos significativos.
- Entonces, si seguimos en binario nos queda algo así:
  - lui s2, 0001 0010 0011 0100 0101 como acá tengo 20 bits pero estoy en 32 bits, extendiendo con 0's.  
Rta: 0001 0010 0011 0100 0101 0000 0000 0000
  - addi s2, s2, 0110 0111 1000.  
Rta: 0001 0010 0011 0100 0101 0110 0111 1000
- En Hexa:
  - lui s2, 0xABCDE (nótese que cada letra son 4 bits,  $4 * 5 = 20$  bits).  
Rta: 0xABCDE000 (nótese que se agregaron tres ceros, porque son lo que falta para llegar a 32 bits).
  - Luego que nuestro s2 ya tiene los ceros para pisar, cargo la parte baja con addi: addi s2, s2, 0x123.  
Rta: 0xABCDE123

C	RISC V
<code>int a = 0xFEEDA987;</code>	<code>lui s2, 0xFEEDB #s2=0xFEEDB000</code> <code>addi s2, s2, -1657 #s2=0xFEEDA987</code>

Si la parte baja se expresa como un número negativo (bit más alto en 1), al extender el signo va a cargar con unos la parte alta. Por eso tenemos que tener esto en cuenta. La parte alta con todos unos equivale a un menos uno en complemento a dos, por lo cual, para compensar el efecto de la extensión del signo en la suma, se incrementa en uno la parte alta que vamos a cargar. En el ejemplo hacemos lui s2, 0xFEEDB en lugar de lui s2, 0xFEEDA.

Cuando dice que la parte baja se expresa como un número negativo ¿está tratando de decir que el valor inmediato de la suma es -1657?

## Memoria

Se estructura y se accede como si fuera un arreglo de elementos de 32 bits (4 bytes). El acceso a memoria es significativamente más lento que el acceso a registros pero nos permite guardar más data.

## Lectura en Memoria

Instrucción lw (load word)

lw carga en destino, el valor que tiene un registro que tiene un puntero hacia la ram.

*lw destino, desplazamiento(registroALeer)*

Ejercicio TypeScript:

```
1 const numbersArr: number[] = [1, 2, 3, 4, 5];  
2 const five: number = numbersArr[4];
```

Equivale en Risc-V a:

```

1 # s1: numbersArr -> cargado a traves de .data, s7: five
2 lw s7, 20(s1)

```

Donde el 20 sería el offset para movernos desde s1, es decir, 20 bytes (5ta palabra) Mapa mental:  $a \leftarrow b$

## Escritura en Memoria

Instrucción sw (store word)  
sw guarda un contenido dado, en un registro que tiene un puntero hacia la ram. Es decir, pisa el valor.

*sw registroAGuardar, desplazamiento(destino)*

Ejercicio TypeScript:

```

1 const numbersArr: number[] = [1, 2, 3, 4, 5];
2 numbersArr[2] = 8;

```

Equivale en Risc-V a:

```

1 # s1: numbersArr -> cargado a traves de .data
2 addi t1, 8, zero
3 sw t1, 12(s1)

```

Mapa mental:  $a \rightarrow b$

**Importante:** destino, en los casos de lw y sw debe ser un puntero a la RAM y debe haber sido inicializado. Véase [anexo](#) para ver ejemplos más claros.

## Máscaras

Una máscara es un valor binario que se utiliza para seleccionar, filtrar o modificar bits específicos dentro de un registro durante operaciones de manipulación de bits. Las máscaras se aplican en conjunto con operaciones lógicas.

Un ejemplo, utilizando la máscara  $\#0x00 \rightarrow \#0x001010$  AND  $\#0x00 = \#0x000000$

Si bien este es un ejemplo que deja mucho que desear, véase la sección de [colores RGB para obtener un color en particular](#).

## Instrucciones lógicas en RISC V

Sean dos registros cualesquiera, las operaciones lógicas se hacen bit a bit. Su uso viene combinado con máscaras. Ejemplo de XOR entre dos registros

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000
xor s5, s1, s2	1011 1001	0101 1110	1111 0001	1011 0111

Véase [colores RGB para obtener un color en particular](#). donde se usa una máscara y la operación bit a bit and.

## Instrucciones de desplazamiento

Como son operaciones aritmético/lógicas, la unidad de desplazamiento es en bits.

- sll (shift left logical): desplaza a la izquierda tantas unidades como se diga, como es lógico se añaden 0's a la derecha.
- srl (shift right logical): desplaza a la derecha tantas unidades como se diga, como es lógico se añaden 0's a la izquierda.
- sra (shift right arithmetic): desplaza a la derecha tantas unidades como se diga, como es aritmético se añade el bit más significativo a la izquierda.

Nota: También existen las versiones de estas operaciones con el immediate (i) para usar una constante.

s5	1111 1111	0001 1100	0001 0000	1110 0111
slli t0, s5, 7	1000 1110	0000 1000	0111 0011	<b>1000 0000</b>
srlt0, s5, 17	<b>0000 0000</b>	<b>0000 0000</b>	<b>0111 1111</b>	1000 1110
srait0, s5, 3	<b>1111 1111</b>	1110 0011	1000 0010	0001 1100

## Byte en particular con desplazamiento de bits

Como hablamos de desplazamiento y byte en particular, sabemos que tendremos que hacer un desplazamiento de bits, y además de eso utilizar una máscara para tomar el bit que nos interesa.

Esto es súper útil en el ámbito de procesamiento de gráficos donde se debe cambiar la intensidad de un color, aplicar filtros o analizar la composición de colores en una imagen.

Un ejemplo para ponernos en contexto sería la forma de expresar los RGB colors con hexadecimal es decir, #RRGGBB. ¿Cómo es que hacemos este proceso de "extraer un byte"? Consideremos que tenemos el color RGB #0xFF0044 donde como anteriormente mencionamos, solo nos interesa #FF0044 y en este caso en particular: 00.

Datos: var = #FF0044

- Desplazamos la cantidad de bits necesarias para que el valor que queramos esté en los menos significativos.
  - En este caso particular, tenemos que hacer un desplazamiento desde el 00 unos 8 bits (4 bits cada dígito en hexa)  
→ srl t0, var, 8
  - El resultado es t0 = 0x00FF00
- Aplicamos la máscara de 0xFF y guardamos el resultado
  - En este caso particular, tenemos andi s2, t0, 0xFF.
  - Luego, s2 tiene el valor de 00 (la intensidad del verde)


Véase [anexo](#) para más ejemplos de desplazamiento con bits.

## Control del flujo de ejecución

Se va pisando el PC (Program Counter).

## Etiquetas en RISC V

Nos permiten ponerle nombre a subproblemas; No ocupan memoria.

**Ejemplo y etiquetas**

```
1 | addi s0, zero, 4
2 | addi s1, zero, 1
3 | slli s1, s1, 2
4 | beq s0, s1, target
5 | addi s1, s1, 1
6 | sub s1, s1, s0
7 | target:
8 | add s1, s1, s0
```

*Handwritten notes:*

- Line 3: *si el desplazamiento es 6 igualar*
- Line 4: *salto a 7.*
- Line 7: *P: S = ELD*
- Below: *IF (S0 = S1) THEN JUMP*
- Below: *ELSE { LINEAS 1 LINEA 6 ^ LINEA 8 }*

## Orden que toman las operaciones en memoria

Las posiciones de memoria que ocupa cada instrucción se leen de arriba hacia abajo sin importar si hay un salto o no. Ver ejercicio en [anexo](#)

## Salto condicionales

Instrucciones que pisan el PC:

- beq(branch if equal): reemplaza el PC si los dos operandos son iguales.
- bne(branch if not equal): reemplaza el PC si los dos operandos son distintos.
- blt(branch if less than): reemplaza el PC si el primer operando es menor que el segundo.
- bge(branch if greater than or equal): reemplaza el PC si el primer operando es mayor o igual al segundo.

## Saltos incondicionales

Instrucciones que pisan el PC:

- j (jump): actualiza el valor del PC con el operando provisto (inmediato de 20 bits extendidos en signo a 32).
- jal (jump and link): que almacena el valor actual del PC en el registro indicado en el primer operando y actualiza el valor del PC con el segundo operando (este se usa para llamar a una función, terminar y volver a ese link anterior).

Véase [anexo](#)

## Funciones

En RISC V la función llamadora puede utilizar los registros del a0 hasta a7 para enviar argumentos. La función llamada usa a0 para devolver el resultado.


La función llamada no debe interferir con el estado de la función llamadora, es decir, debe respetar los valores de los registros (s0 a s11) y el registro ra (dirección de retorno).

El stack de la función llamadora debe mantenerse invariante al ingresar a la función llamada.

## Llamando funciones & enviando parámetros

Los inicializamos en la función llamadora, y luego en la función llamada simplemente accedemos a ellos.

**Ejemplo con argumentos**



C	RISC V
<pre>int main(){   int y;   ...   y = dif_sumas(2,3,4,5);   ... }</pre>	<pre>main: #s7=y   addi a0, zero, 2   addi a1, zero, 3   addi a2, zero, 4   addi a3, zero, 5   jal dif_sumas   add s7, a0, zero dif_sums: #s3=result   add t0, a0, a1   add t1, a2, a3   sub s3, t0, t1   add a0, s3, zero   jrr ra</pre>

*CARGA PARAMS* (pointing to the arguments in the C code and the RISC V code)

*GUARDA RA SALTA A dif\_sums* (pointing to the jal instruction in the RISC V code)

*PASAR PARAMS ≡ CARGARLOS EN MEMORIA ANTES DE ENTRAR A FUNCIÓN*

68

Nótese que cuando hacemos jal dif sumas, le estamos mandando la posición donde quedó antes de llamar a la función, entonces luego en la llamada podemos usar ra y volver a ese punto.

## Pila (stack)

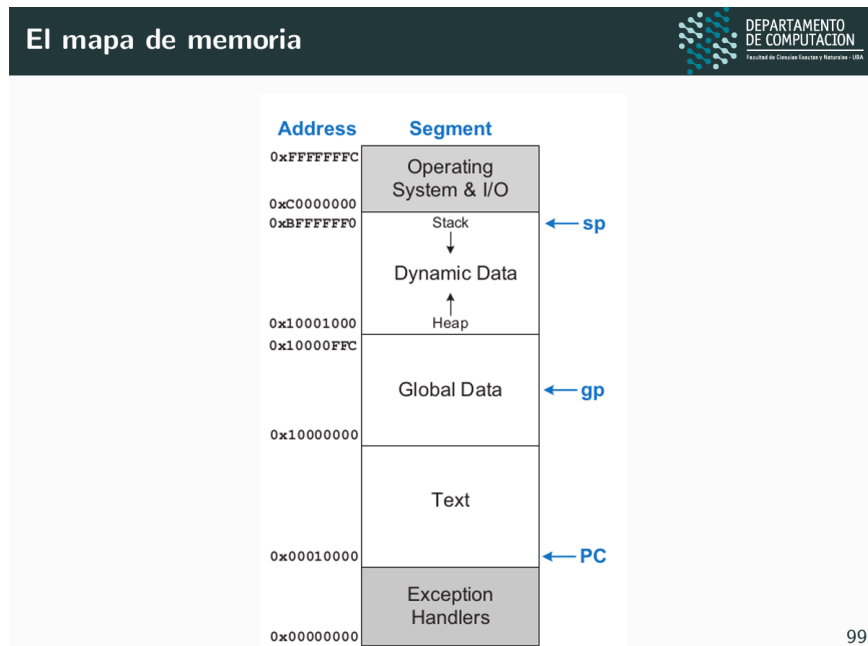
Es una parte de la memoria que se utiliza para almacenar información temporaria. La pila suele comenzar en las direcciones altas de la memoria y va tomando las direcciones inmediatamente más bajas. RISC V recomienda el registro de su Stack Pointer (sp)

## Reglas de llamada

- Regla para la llamadora: Antes de llamar debe guardar los valores de los registros temporarios que necesite utilizar al retornar (t0-t6, a0-a7)
- Regla para la llamada: Si va a utilizar los registros permanentes (s0-s11, ra) debe guardarlos al comenzar y restaurarlos antes de retornar

# Compilación, ensamblado y ejecución

## El mapa de memoria



## Directivas de ensamblado

Assembler Directive	Description
.text	Text section
.data	Global data section
.bss	Global data initialized to 0
.section .foo	Section named .foo
.align N	Align next data/instruction on 2 <sup>N</sup> -byte boundary
.balign N	Align next data/instruction on N-byte boundary
.globl sym	Label sym is global
.string "str"	Store string "str" in memory
.word w1,w2,...,wN	Store N 32-bit values in successive memory words
.byte b1,b2,...,bN	Store N 8-bit values in successive memory bytes
.space N	Reserve N bytes to store variable
.equ name, constant	Define symbol name with value constant
.end	End of assembly code

## Inicialización de datos

```
.section .data
# A partir de este punto comienzan los datos
largo: .word 0x4
caracter: .byte 10
arreglo: .word 0xc, 0x34d, 0x1, 0x0
.section .text
# A partir de este punto comienzan las
  instrucciones
```

Inicialización de datos en la sección .data que va a ubicar la información en lo que el mapa se muestra como Global Data.  
Programa final con inicialización de los datos en RISC V

```
1  .section .text
2  .global sumar_arreglo
3  sumar_arreglo:
4  # a0 = int a[], a1 = int largo, t0 = acumulador, t1
    = i
5  li    t0, 0          # acumulador = 0
6  li    t1, 0          # i = 0
7  ciclo: # Comienzo de ciclo
8  bge   t1, a1, fin    # Si i >= largo, sale del ciclo
9  slli  t2, t1, 2      # Multiplica i por 4 (1 << 2 = 4)
10 add   t2, a0, t2     # Actualiza la dir. de memoria
11 lw    t2, 0(t2)      # De-referencia la dir,
12 add   t0, t0, t2     # Agrega el valor al acumulador
13 addi  t1, t1, 1      # Incrementa el iterador
14 j     ciclo          # Vuelve a comenzar el ciclo
15 fin:
16 mv    a0, t0         # Mueve t0 (acumulador) a a0
17 ret                    # Devuelve valor por a0
```

# Cosas a utilizar en Risc V (machete)

Página 21 Guia Práctica de Risc V

31	0
x0 / zero	Alambrado a cero
x1 / ra	Dirección de retorno
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporal
x6 / t1	Temporal
x7 / t2	Temporal
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Argumento de función, valor de retorno
x11 / a1	Argumento de función, valor de retorno
x12 / a2	Argumento de función
x13 / a3	Argumento de función
x14 / a4	Argumento de función
x15 / a5	Argumento de función
x16 / a6	Argumento de función
x17 / a7	Argumento de función
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporal
x29 / t4	Temporal
x30 / t5	Temporal
x31 / t6	Temporal
32	
31	0
pc	
32	



31	25	24	20	19	15	14	12	11	7	6	0	
imm[31:12]								rd	0110111			U lui
imm[31:12]								rd	0010111			U auipc
imm[20:10:1 11:19:12]								rd	1101111			J jal
imm[11:0]				rs1	000			rd	1100111			I jalr
imm[12:10:5]		rs2	rs1	000			imm[4:1 11]	1100011			B beq	
imm[12:10:5]		rs2	rs1	001			imm[4:1 11]	1100011			B bne	
imm[12:10:5]		rs2	rs1	100			imm[4:1 11]	1100011			B blt	
imm[12:10:5]		rs2	rs1	101			imm[4:1 11]	1100011			B bge	
imm[12:10:5]		rs2	rs1	110			imm[4:1 11]	1100011			B bltu	
imm[12:10:5]		rs2	rs1	111			imm[4:1 11]	1100011			B bgeu	
imm[11:0]				rs1	000			rd	0000011			I lb
imm[11:0]				rs1	001			rd	0000011			I lh
imm[11:0]				rs1	010			rd	0000011			I lw
imm[11:0]				rs1	100			rd	0000011			I lbu
imm[11:0]				rs1	101			rd	0000011			I lhu
imm[11:5]		rs2	rs1	000			imm[4:0]	0100011			S sb	
imm[11:5]		rs2	rs1	001			imm[4:0]	0100011			S sh	
imm[11:5]		rs2	rs1	010			imm[4:0]	0100011			S sw	
imm[11:0]				rs1	000			rd	0010011			I addi
imm[11:0]				rs1	010			rd	0010011			I slti
imm[11:0]				rs1	011			rd	0010011			I sltiu
imm[11:0]				rs1	100			rd	0010011			I xori
imm[11:0]				rs1	110			rd	0010011			I ori
imm[11:0]				rs1	111			rd	0010011			I andi
0000000		shamt	rs1	001			rd	0010011			I slli	
0000000		shamt	rs1	101			rd	0010011			I srli	
0100000		shamt	rs1	101			rd	0010011			I srai	
0000000		rs2	rs1	000			rd	0110011			R add	
0100000		rs2	rs1	000			rd	0110011			R sub	
0000000		rs2	rs1	001			rd	0110011			R sll	
0000000		rs2	rs1	010			rd	0110011			R slt	
0000000		rs2	rs1	011			rd	0110011			R sltu	
0000000		rs2	rs1	100			rd	0110011			R xor	
0000000		rs2	rs1	101			rd	0110011			R srl	
0100000		rs2	rs1	101			rd	0110011			R sra	
0000000		rs2	rs1	110			rd	0110011			R or	
0000000		rs2	rs1	111			rd	0110011			R and	
0000	pred	succ	00000	000	00000	0001111	I fence					
0000	0000	0000	00000	001	00000	0001111	I fence.i					
000000000000			00000	000	00000	1110011	I ecall					
000000000001			00000	000	00000	1110011	I ebreak					
csr			rs1	001	rd	1110011	I csrrw					
csr			rs1	010	rd	1110011	I csrrs					
csr			rs1	011	rd	1110011	I csrrc					
csr			zimm	101	rd	1110011	I csrrwi					
csr			zimm	110	rd	1110011	I csrrsi					
csr			zimm	111	rd	1110011	I csrrci					

Figura 2.3: El mapa de opcodes de RV32I tiene la estructura de la instrucción, opcodes, tipo de formato y nombres (La Tabla 19.2 de [Waterman and Asanović 2017] es la base de esta figura).

Risc V cheatsheet

### 32-bit instruction format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
R	func							rs2				rs1				func				rd				opcode															
I	immediate												rs1				func				rd				opcode														
SB	immediate							rs2				rs1				func				immediate				opcode															
UJ	immediate																												rd				opcode						

# HDL

Los lenguajes de descripción de hardware (HDL) son herramientas que nos permiten describir la estructura y comportamiento de nuestros diseños de forma sencilla, sintética y escalable a la hora de diseñar e implementar hardware industrial a gran escala. Existen dos lenguajes dominantes actualmente: VHDL y System Verilog.

## Módulos

Son bloques de hardware que tienen entradas y salidas además de un nombre. Se los puede describir de dos maneras:

- Comportamental: Como se modifican las señales salida en base a la entrada y en qué estado queda el componente.
- Estructural: Como se vinculan sus entradas y salidas entre sí y describiendo la composición de diversos componentes (una cajita usa otra cajita).
- Ej: AND, OR, multiplexor o sumador son ejemplos de módulos.

## ¿Para qué sirve el código HDL?

Es la fuente de dos procesos.

- Simulación: Simula, a través de ondas como se comporta el programa dada una serie de entradas y verifica que la salida sea la esperada.
- Síntesis: Traduce el HDL a un conjunto de compuertas lógicas.

Véase [anexo](#) para un ejemplo.

Nota: en System Verilog, cuando vemos algo de tipo `logic[7:0]` significa que es un vector de 8 bits donde el 7 es el MSB y el 0 el LSB.

## Modelado comportamental

### Tipos de asignaciones

- `=`: asignación bloqueante (aka: síncrona). En microarquitectura es un circuito combinatorio.
- `always`: asignación no bloqueante (aka: asíncrona). En microarquitectura es un circuito secuencial que funciona con bloques `always`.

Véase [bloques always](#) para más información

### Operadores de reducción

Colapsan los bits de una señal de tamaño variable aplicando una operación lógica.

`assign y = &a`

Ejecuta una especie de and entre todos los valores que tiene a sin necesidad de hacer uno por uno (ej: `a[7] & a[6]...`)

### Asignación condicional

Infiere un multiplexor de dos entradas, es idéntico al ternario de C

$$assign y = s ? d1 : d0$$

donde:

- s: entrada de control
- y: salida
- d0: op 00
- d1: op 01

### Asignación condicional compuesto

Es un ternario anidado. Infiere un multiplexor de cuatro entradas, por lo tanto son dos ternarios hijos y uno padre.

`assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0)`

## Variables internas

Se realizan con logic dentro del módulo.

*logic* *p, g*; ← crea dos variables *p* y *g*.

Véase [anexo](#) para un ejemplo con un full adder

## Precedencia de operaciones

También conocido como "prioridad" de operaciones.

Op	Meaning
~	NOT
*, /, %	MUL, DIV, MOD
+, -	PLUS, MINUS
<<, >>	Logical Left/Right Shift
<<<, >>>	Arithmetic Left/Right Shift
<, <=, >, >=	Relative Comparison
==, !=	Equality Comparison
&, ~&	AND, NAND
^, ~^	XOR, XNOR
, ~	OR, NOR
?:	Conditional

## Valores numéricos

$$bits' + base + valor$$

ej: 8b10 → 8 bits, binario número 2 (0000 0010)

La base se indica con una sola letra: d (decimal), b (binario), o (octal), h (hexadecimal).

Nota: siempre se termina almacenando en binario.

## Alta Impedancia (Resistencia)

En System Verilog se representa con el valor de z.

```
1 module buffer3(input logic[3:0] a, input logic enable, output tri[3:0] y);  
2     assign y = en ? a : 4'bz;  
3 endmodule
```

enable es una señal de control que dice si deja pasar la energía o no. Si está apagada entonces estaría en Hi-Z caso contrario, deja pasar el valor que tenga a.

Nota: Cuando enable es false, se le asigna 4'bz porque y es de 4 bits según la entrada al módulo ([3:0])

## Valores desconocidos

En SystemVerilog, cuando los valores de una señal no puede definirse y se considera como desconocida o como de valor x se puede valer a dos razones

- Valor no inicializado: La señal no se ha inicializado
- Contención: Cuando dos buffers de tres estados están conectados al mismo bus pero envían señales distintas.

Si sucede esto, indica un error en el diseño.

## Manipulación de bits

- {}: concatena bits. El número de afuera indica la cantidad de repeticiones. Ej: {3{d[0]}} ≡ d[0]d[0]d[0]
- [] accede a una lista específica de bits. Tienen que ser números descendentes. Es decir a > b. Ej: c[a:b]

```
1 assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

# Modelado Estructural

Para poder componer módulos, tenemos que hacer instancias de ellos.

*componente nombreInstancia(parametros)*

Véase [anexo](#) para un ejemplo.

Nota: Utilizando el enfoque estructural, la implementación de un mismo comportamiento se puede conseguir como composición de distinto tipo de componentes.

## Circuitos Secuenciales

Se describen con componentes de memoria y se emplean bloques always.

### Bloques Always

Utilizados para indicar frente a qué evento se debe actualizar el estado de cada componente. Cada bloque always viene con una lista de eventos (sensitivity list) que les hace efecto.

Nota: Si el bloque always tiene más de una asignación se las debe agrupar en un par begin end

*bloque @(evento var)*

Véase [anexo](#) para un ejemplo.

### Reset asincrónico - sincrónico

Reset asincrónico: Cobra efecto en cuanto cambia de valor la señal.

Reset sincrónico: Cobra efecto frente al evento del flanco ascendente del reloj.

## Always Comb - Circuitos Combinatorios

No entendí

## Case - Circuitos Combinatorios

Se usa igual que en los lenguajes de programación pero acá es simplemente case.

```
1  case(data):  
2    0: segments = algo;  
3    1: segments = algo;  
4    ...  
5    default: segments = algo;  
6  endcase
```

Nota: segments es una variable de salida de 7 bits.

# Microarquitectura

Es la parte más cercana a la ingeniería. Nos sirve para poder entender como se manejan los flujos de información entre assembler (algo más abstracto) a circuitos combinatorios y secuenciales (algo más visual).

La microarquitectura se va a encargar de actualizar los registros y el program counter.

## Datapath

Un camino de datos o datapath es un conjunto de componentes funcionales, que realizan operaciones de procesamiento de datos, registros y buses. Junto con la unidad de control forman una CPU.

**Pero, ¿quién es el encargado de controlar el orden de las operaciones en el DataPath?** la unidad de control.

## Componentes del DataPath y responsabilidades

Glosario antes de empezar:

- A: Address
- RD: Read Destination
- WD: Write Destination
- WE: Write Enable

El DataPath está formado por

- Program Counter (CPU): Contiene la instrucción actual. PC Next: Si hay un salto condicional, entonces contiene la posición de memoria donde se encuentra el salto; en otro caso, se incrementan 4 bytes (una palabra).
- Instruction Memory (CPU): Recibe el address de la instrucción para poder hacerle el fetch. Es decir, obtener la representación binaria de la operación.
- Control Unit (CPU): Es el director de orquesta. Es quién le dice a cada componente qué es lo que tiene que hacer, y qué es lo que sigue luego de haber terminado su responsabilidad.
- Register File (CPU): Contiene los 32 registros x0-x31. Posee dos puertos de lectura (RD1 y RD2) que vuelcan el valor de las entradas de escritura (A1 y A2) respectivamente. WD3 también es un puerto de entrada de escritura que escribe el dato en A3 durante el flanco ascendente del reoj si la señal de control WE3 se encuentra alta.
- Multiplexores (CPU)
- ALU (CPU)
- Buses de Datos (CPU)
- Data Memory (RAM)

Véase anexo para un ejemplo de un datapath

## Colores en un Data Path

- Líneas gruesas: datos de 32 bits.
- Líneas delgadas: datos de 1 bit.
- Líneas intermedias: datos de otro tamaño.
- Líneas azules: señales de control

## Anexo - Ejercicios

### Hexadecimal en Risc-V

Recuerde que en Risc-V operamos siempre con 32 bits.

¿Qué realiza la operación `andi t0, 0xABCDE, 0xFFF`?

Sé que cada dígito hexadecimal son 4 bits, por lo tanto, pasemos todo a binario para que tengamos un mejor panorama, pero lo que va a terminar sucediendo es que nos quedará `0x00CDE`, osea `0xCDE`.

- $0xABCDE \equiv 0000\ 0000\ 0000\ 1010\ 1011\ 1100\ 1101\ 1110$
- $0xFFF \equiv 0000\ 0000\ 0000\ 0000\ 0000\ 1111\ 1111\ 1111$

Luego,  
`andi` realiza la operación de AND y a ojo podemos ver, que en binario nos queda: `t0 = 1100 1101 1110` que equivale a `0xCDE`

## Registros en lw y sw

```
1      addi x1, x2, 3 -> 5
2      sw x3, 0(x1) (se desplaza 0 bytes desde x1) -> se coloca el valor de x3 en x1
```

¿Es correcta esta implementación? Considere que x2 tiene es un registro que tiene el valor de 5.

No, la implementación no es correcta, porque se intenta escribir en x3, un supuesto puntero a la RAM x1, pero x1 es un valor. Esto sería un error grave.

```
1      lw x1, 8(x2) -> guarda en x1, la posicion de memoria de x2 corrida 2 palabras (desplaza 8 bytes).
2      sw x3, 0(x1) -> almacena el valor de x3 en la posicion de memoria de x1 (desplaza 0 bytes)
```

¿Es correcta esta implementación? Considere que x2 tiene es un registro que tiene un puntero a la RAM, tal que su valor decodificado es 5.

Sí, la implementación es correcta. Porque toma el registro x2 que tiene un puntero a la memoria, lo corre dos palabras y lo almacena en x1. Luego, x1 (que tiene una nueva dirección de memoria a la RAM (puntero)) lo guarda en x3.

## Desplazamiento de Bits para obtener un Byte particular

Almacene, en un registro cualesquiera el segundo byte del siguiente hexadecimal: #0x89ABCDEF sabiendo que el valor está en la memoria RAM; indique qué datos son relevantes, y qué mascara utilizará.

Datos: #89ABCDEF

Sé que cada dígito hexadecimal son 4 bits, por lo tanto, si tengo que sacar el segundo byte:  $4 + 4 = 8$  bits (1 byte) por lo tanto tengo que sacar CD pues  $(4 + 4 + 4 + 4 = 16 \equiv 2 \text{ bytes})$

Sé que como CD debo llevarlo a los más significativos, aplico un srl 8 bits a la derecha. Sé también que si el valor es una posición de memoria en la RAM, voy a tener que usar li pues necesito cargarlo desde la memoria principal (RAM).

Tengo que utilizar la máscara 0xFF (Rdo que al ser 32 bits, los últimos 8 son 1) con (#11111111) con un and pues necesito aislar los bytes que me interesan.

¿Por qué una máscara? porque una máscara se utiliza para seleccionar ciertos bits, descartando los que no necesitamos. Las máscaras las aplicamos haciendo operaciones lógicas bit a bit como AND, OR, XOR, NOT, entre otras. Código en Risc-V

```
1      li t0, 0x89ABCDEF
2      srli t1, t0, 8 (desplazamiento logico de 8 bits a la derecha) <- 0x0089ABCD
3      andi t1, t1, 0xFF <- 0x0000CD <- 0xCD
```

## Ejercicios con Saltos Condicionales / Incondicionales

Ejercicio llamado a procedimiento sin retorno, TypeScript: Salto sin retorno, y modificación de variable global.

```
1      let destinatarios = 2;
2
3      while(destinatarios>0){
4          destinatarios -= 1;
5      }
```

Equivale en Risc-V a:

```
1      #s2 = destinatarios
2      lw a0, s2
3      j loop
4      loop:
5          addi a0, a0, -1
6          beq a0, zero, fin
7          j loop
8      fin:
```

Ejercicio con recursión, TypeScript: Salto con retorno, y modificación de variable global.

```
1      let destinatarios = 2;
2
3      const hacerCero = (destinatarios: number): number => {
4          if(destinatarios == 0) return 0;
5          return hacerCero(destinatarios-1);
6      }
7
8      const respuesta: number = hacerCero(destinatarios);
```

Equivale en Risc-V a:

```

1  #s2 = destinatarios
2  lw a0, s2                      #0x0000 0000
3  jal ra, hacerCero              #0x0000 0004
4  sw v0, x3                      #0x0000 0008
5  ecall                          #0x0000 000C
6  hacerCero:
7      addi sp, sp, -4             #0x0000 0010
8      sw ra, 0(sp)               #0x0000 0014
9      beq a0, zero, finHacerCero #0x0000 0018
10     addi a0, a0, -1             #0x0000 001C
11     jal ra, hacerCero           #0x0000 0020
12 finHacerCero:
13     li v0, 0                   #0x0000 0024
14     lw ra, 0(sp)               #0x0000 0028
15     sw sp, 4(sp)               #0x0000 002C
16     jr ra                      #0x0000 0030

```

Nótese: que al usar recursión, reservamos siempre 4 bytes para guardar el ra por si llegamos a saltar a otra función en el beq.

## Hexa - Binario - Risc V

### Decodifique a instrucciones de Risc-V

1. 0x00700293

Pasamos cada numero hexadecimal a binario, cada hexa son 4 bits de binario

0000 0000 0111 0000 0000 0010 1001 0011

Ahora sabemos que el opcode son los últimos 7 bits, reagrupamos

0000 0000 0111 0000 0000 0010 1 0010011

Vemos que familia de operación corresponde a 0010011 en la tabla de la Figura 2.3. La familia es  $\Rightarrow$  I

Ahora vemos que lo que le sigue es el rd, por lo tanto reagrupo

0000 0000 0111 0000 0000 00101 0010011

Lo que sigue ahora es la func, que son 3 bits, por lo tanto reagrupo

0000 0000 0111 0000 0 000 00101 0010011

Lo que sigue ahora es el rs1 que toma 5 bits, por lo tanto reagrupo

0000 0000 0111 00000 000 00101 0010011

Lo que sigue ahora es el immediate (20 a 31) porque estamos en la familia del I, por lo tanto reagrupo

000000000111 00000 000 00101 0010011

Por último, traducimos a operaciones de Risc-V

0010011 000: ADDI

00101 rd: pasado a decimal es 5, por lo tanto en la tabla de registros x5 es t0.

00000 rs1: pasado a decimal es 0, por lo tanto en la tabla de registros x0 es zero.

0x007: es donde se guarda en memoria, eso en decimal es 7 por lo tanto en la tabla de registros x7 es t2.

Luego, juntamos todos los operandos en Risc-V

ADDI t0, zero, t2

## Traducción de programa de Risc V a castellano y observando las posiciones de memoria de instrucciones

```

1  li a0, 4228 <- 0x0000 a 0x0008 (0 a 8) sin incluir
2  li a1, 2114 <- 0x0008 a 0x0010 (8 a 16) sin incluir
3  jal ra, resta <- 0x0010
4  fin:
5      beq zero, zero, fin <- 0x0014
6  resta:
7  prologo:
8      addi sp, sp, -4 <- 0x0018
9      sw ra, 0(sp) <- 0x001c
10     sub a0, a0, a1 <- 0x0020
11     beq a0, zero, epilogo <- 0x0024
12 sigo:
13     jal ra, resta <- 0x0028

```

```

14 |     epilogo:
15 |         lw ra, 0(sp) <- 0x002c
16 |         addi sp, sp, 4 <- 0x0030
17 |         ret <- 0x0034

```

**Nota:** Recordar que las etiquetas no son funciones, son simplemente instrucciones que se leen. Es decir, por ejemplo, cuando va por prologo y llega a beq si es falso ejecuta sigo, pero no es necesario llamarla sino que lo de jal ra, resta es parte del prologo solo que se le dio un subnombre. Todo se lee de arriba hacia abajo, por lo tanto a epilogo tambien estaría dentro de prologo por así decirlo pero nunca llega porque el beq si es falso hace llamada recursiva. Solo llega a epilogo y cambia el PC si el beq es true.

Nota: fin ¿no termina nunca cierto? es re contra recursivo ¿Qué hace este código? línea por línea

- li a0, 4228: carga el valor de 4228 en el registro a0. Internamente son dos operaciones, lui y addi por lo tanto ocupan desde 0x0000 a 0x0008 (sin incluir).
- li a1, 2114: carga el valor de 2114 en el registro a1. Internamente son dos operaciones, lui y addi por lo tanto ocupan desde 0x0000 a 0x0010 (sin incluir)
- jal ra, resta: salta a la etiqueta de resta y guarda la dirección de retorno en ra (para que el return de resta vuelva acá)
- addi sp, sp, -4: reserva 4 bytes en el stack pointer bajándolo. Esto es un caso específico, podría bajar 8 bytes si se quisiera pero como solo quiere escribir un solo valor, basta con 4 bytes. Importante que luego de hacer todo el stack debe volver a su posición original.
- sw: guarda ra en la primera posición del stack pointer. esto es porque luego cuando salta a epilogo tiene que saber bien a donde volver.
- sub: guarda en a0 la resta entre a0 y a1.
- beq: verifica si a0 es igual a zero, si es igual a cero salta a epilogo, caso contrario hace una llamada recursiva nuevamente a resta.
- sigo: si beq es falso llega acá, hace la llamada recursivamente como en la inicial mandando un nuevo estado.
- lw: se guarda en ra el valor que estaba en 0(sp) cuando lo guardamos antes.
- addi: vuelve a subir el stack a su posición inicial porque ya no necesitamos hacer nada (nótese que es importante que subimos dejando todo como estaba pero nuestro ra que habíamos guardado antes en la primera pos lo recuperamos con el lw)
- ret: vuelve a jal ra, resta y ejecuta fin.
- el programa termina si zero es igual a zero pero luego ejecuta fin recursivamente. (no termina nunca)

Preguntas:

- a) Indicar en qué posiciones de memoria se encuentra cada etiqueta
  - fin: 0x0014
  - resta: 0x0010
  - sigo: 0x0028
  - epilogo: 0x002c
- b) Hecho arriba
- c) TODO
- d) TODO
- e) ¿Cual es el valor final de a1? El valor final de a1 sería 0, pues dejaría de hacer el paso recursivo y llegaría a 0.
- f) ¿Cuál es el valor final del PC? El último valor del PC es la última instrucción que ejecutó, sería 0x0014 (fin).
- g) Listar la secuencia descrita por el PC. 0x0000 0x0004 0x0008 0x0010 0x0018 0x001c 0x0020 0x0024 0x0028 (vuelve a entrar a prologo) 0x0018 0x001c 0x0020 0x0024 (evalua guarda, true va directo a epilogo) 0x002c 0x0030 0x0034 (el return nos manda de vuelta al jal) 0x0010 0x0014 (fin)
- h) Indique qué valores toman los registros ra y sp: al inicio, durante y al finalizar la ejecución
  - ra: 0x0010 antes de saltar a resta, dentro del prologo, ra se guarda en la primera posición del sp ocupando los primeros 4 bytes, finalmente en epilogo, a ra se le coloca la posición del elemento 0 del stack pointer. De igual manera, ra siempre tiene el mismo valor, nunca cambia.



- sp: Se desconoce que tiene inicialmente pero tiene cosas de memoria dinámica, variables globales, otras instrucciones, etc. Pero viendo solo el programa, el sp se mueve hacia abajo 4 bytes. Luego, en el espacio que reservamos se guarda el valor de ra. Por ultimo, los primeros 4 bytes del stack pointer se almacenan en ra y se liberan los 4 bytes que habíamos reservado.

- i) Reemplazar la segunda instrucción li a1, 1023 de modo que a1 sea a0 dividido 2 con una única instrucción (acá hay algo raro porque la segunda instrucción no es a1, 1023 es a1 2114)

Pasemos esto a TypeScript: A modo ilustrativo, si la función recursiva termina retornará true, caso contrario se quedaría colgada. En estos casos siempre termina xq los números dan.

```

1  const a0:number = 4228;
2  const a1:number = 2114;
3
4  const resta = (a0: number, a1: number): number | boolean => {
5      if(a0 == 0){
6          return true; //epilogo
7      }
8      else{
9          return resta(a0-a1, a1); //sigo
10     }
11 }
12
13 const res = resta(a0, a1);
14 console.log(res); //true

```

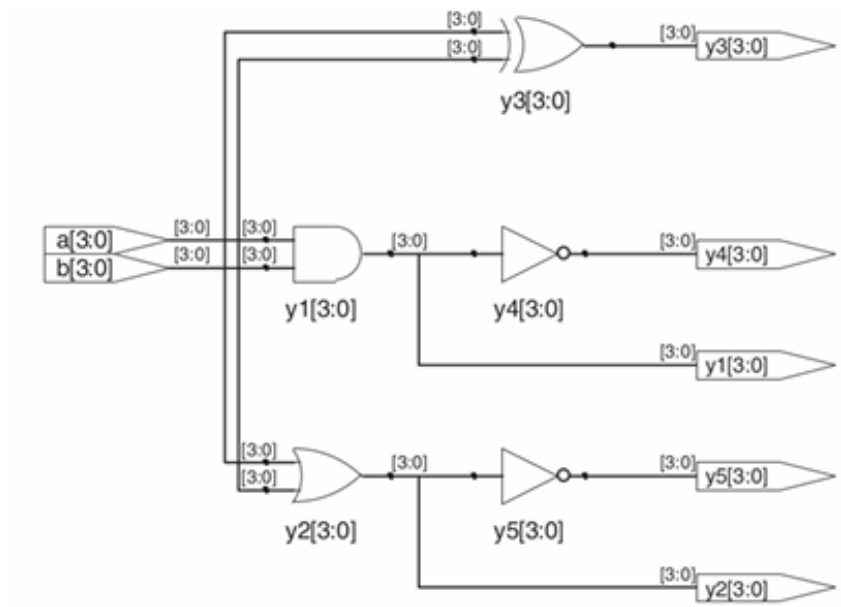
## System Verilog - Síntesis - Simulación

```

1  module compuertas(
2      input logic [3:0] a, b,
3      output logic [3:0] y1, y2, y3, y4, y5
4  );
5
6      assign y1 = a & b; //and
7      assign y2 = a | b; //or
8      assign y3 = a ^ b; //xor
9      assign y4 = ~(a & b) (alt gr y simbolo de ~ para escribirlo) //nand
10     assign y5 = ~(a | b); //nor
11 endmodule

```

Nota: [3:0] son 4 bits donde 3 es el bit más significativo (MSB) y 0 el menos significativo (LSB) Circuito sintetizado:



## System Verilog - Síntesis (Full Adder)

```

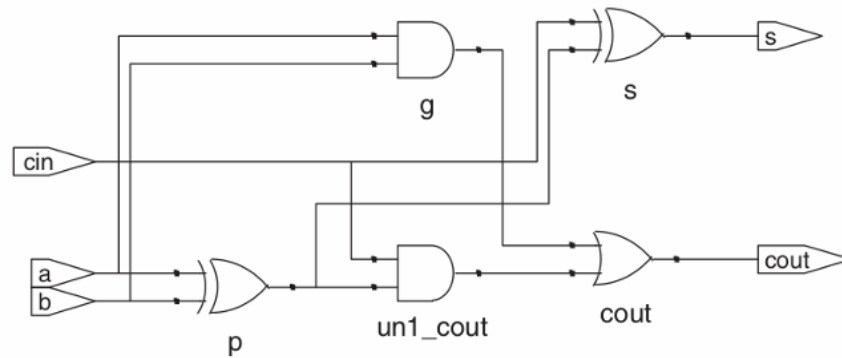
1  module fulladder(input logic a, b, cin, output logic s, cout);
2      logic p, g;
3      assign p = a ^ b;

```

```

4      assign g = a & b;
5      assign s = p ^ cin;
6      assign cout = p | (p & cin);
7  endmodule

```



## Composición de multiplexores - Comportamiento Estructural

```

1  module mux4(input logic [3:0] d0, d1, d2, d3, input logic[1, 0] s, output logic [3:0] y);
2  logic [3:0] low, high;
3  mux2 lowmux(d0, d1, s[0], low);
4  mux2 highmux(d2, d3, s[0], high);
5  mux2 finalmux(low, high, s[1], y);

```

Nótese que lowmux, highmux y finalmux son instancias de mux2.

## System Verilog - Asignación no bloqueante

```

1  module flop(input logic clk, input logic [3:0] d, output logic[3:0] q);
2      always_ff @(posedge clk)
3          q <= d;
4  endmodule

```

La asignación no bloqueante de d en q se hace cuando ocurre el evento posedge clk, lo que significa que se actualiza el estado en el flanco ascendente de la señal del clock.

## DataPath, ejemplo con instrucción Risc V

¿Cómo se conecta Risc V con el DataPath?

```

1  ADD x1, x2, x3 <- suma x2 y x3 y lo guarda en x1

```

El paso a paso:

- El PC se coloca sobre la instrucción
- Fetch: La instruction memory recibe el address de donde se encuentra la operación (ADD x1, x2, x3), y la respuesta es la operación en binario.
- Decode: La unidad de control, recibe la operación en binario realizada por la instruction memory, y empieza a analizar el opcode, la función que debe realizar, los operandos, etc.
- Execute: La unidad de control, una vez que decidió que debe realizarse (proceso decode), ejecuta las operaciones correspondientes en el orden.
  - Toma las posiciones de memoria de x2 y x3; las solicita en el registro para obtener el valor; se envían los valores de x2 y x3 en binario, y una flag que le indica qué operación realizar a la ALU (ordenado por la unidad de control)
  - El clock cambia de estado.
  - La ALU devuelve el resultado directamente al Register File (ordenado por la unidad de control), enviando el resultado a WD3 (Write Destination).
  - WE3 se enciende (por aviso de la unidad de control); el clock cambia de estado, escribiendo en WD3 el resultado de la ALU.
- El clock cambia de estado.
- Al no haber un salto condicional, PCNext está vacío e incrementa el PC actual 4 bytes (1 palabra) (ordenado por la unidad de control) para leer la instrucción. En este caso, termina el programa.