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)

Digítos/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.

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

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

$$128 = 64 * 2 + 0$$

$$64 = 32 * 2 + 0$$

$$32 = 16 * 2 + 0$$

$$16 = 8 * 2 + 0$$

$$8 = 4 * 2 + 0$$
$$4 = 2 * 2 + 0$$
$$2 = 1 * 2 + 0$$
$$1 = 0 * 2 + 1$$

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.

$$\mathbf{1}000000\mathbf{0}_{(2)}$$

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 (**,**)

$$1_{(10)} = 01_{(2)}$$
$$128_{(10)} = 10000000_{(2)}$$

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

- 0: número positivo
- 1: número negativo.

$$18_{(10)} = \mathbf{0}0010010_{(2)}$$
$$-18_{(10)} = \mathbf{1}0010010_{(2)}$$

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

$$+0_{(10)} = \mathbf{0}0000000_{(2)}$$
$$-0_{(10)} = \mathbf{1}0000000_{(2)}$$

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 $m > 0$

$$n_0 = n + m$$
$$n = 1 \wedge m = 10 \longrightarrow n_0 = -9$$

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.
  - $-2_{(2)} = inv(010) + 1$
  - $-2_{(2)} = 101 + 1$
  - $-2_{(2)} = 110$
- **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

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)

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

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)

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

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 ≡ A + (INV(B) + 1)

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 $1111111111_{(2)}$?

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

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

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 = **1**00000 10
- Complemento 2 (C_2): Se extiende con el bit más significativo.
  - En 3 bits, -2 = 110
  - En 8 bits, -2 = **1**11111 10

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 (⊕): (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	$A.\overline{A} = 0$	$A + \overline{A} = 1$
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	$\overline{A.B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}.\overline{B}$

Operaciones booleanas

Se resuelven utilizando las propiedades para operaciones lógicas

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

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

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:

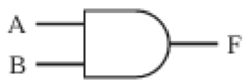
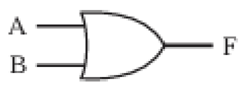
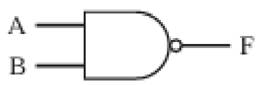
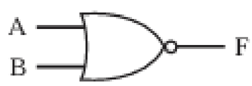
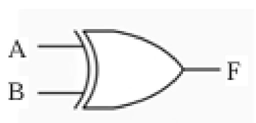
$$Sea\ F = X + \bar{Y}$$

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 • B or F = AB	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		F = A + B	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or F = A'	A | F 0 | 1 1 | 0
NAND		$F = (\overline{AB})$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (\overline{A + B})$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	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.

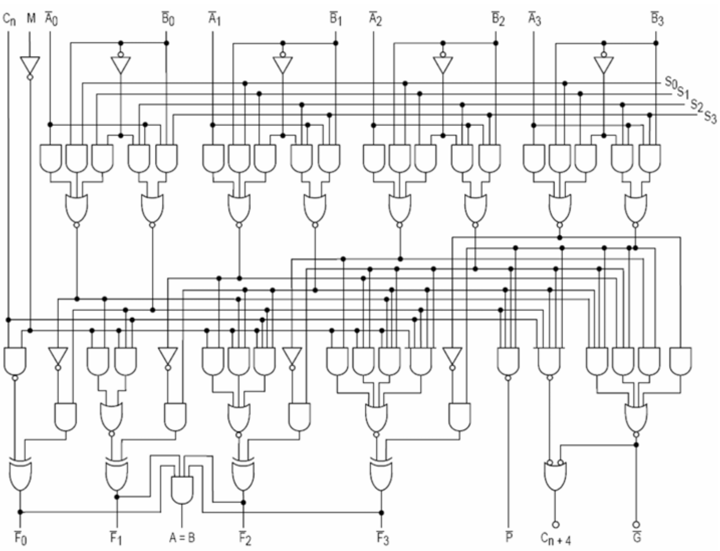
- NAND = $\overline{A \wedge B}$

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

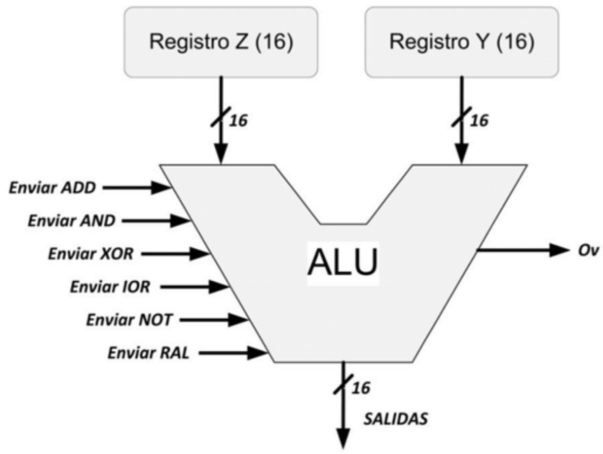
Compuertas en SystemVerilog

- A AND B ($A * B$) = (assign O = A & B)
- A OR B ($A + B$)= $assign\ A \mid B$
- A XOR B = $assign\ A\ \wedge\ B$
- 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 ϕ a una fórmula proposicional cualquiera

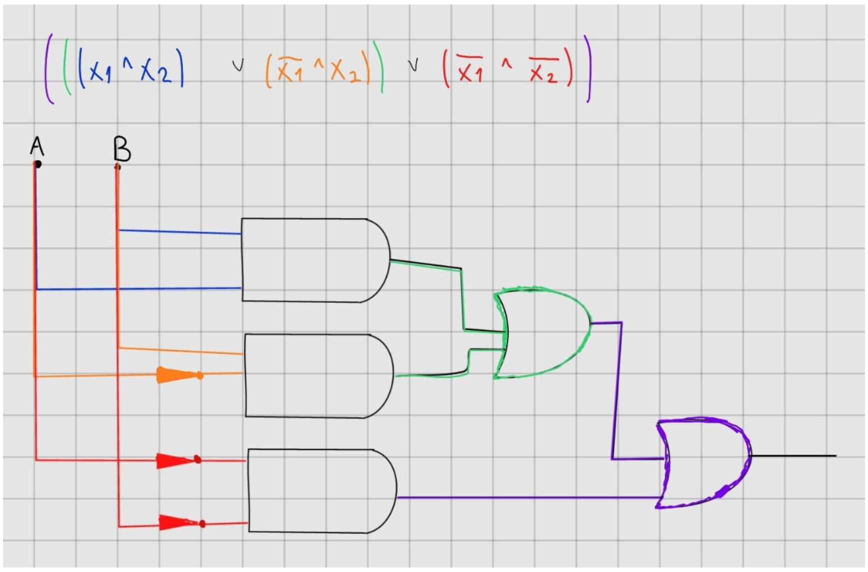
- 1. Solo consideramos de la función F, las filas verdaderas.
- 2. Cada fila verdadera tendrá su índice, y en ese índice estarán los valores de cada variable proposicional. Representamos a esa fila verdadera como t_i
- 3. Realizamos la conjunción de todas las variables de ese t_i
- 4. Realizamos la disyunción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

Un ejemplo:

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

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

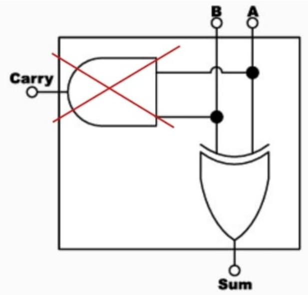
- F es solamente verdadera en la primera, segunda y tercer fila por lo tanto tenemos t_1 t_2 y t_3
- Por cada fila, hacemos la conjunción de los valores
  - $x_1 \wedge x_2$
  - $\bar{x_1} \wedge x_2$
  - $\bar{x_1} \wedge \bar{x_2}$
- Realizamos la disyunción de todos los t_i
  - $(x_1 \wedge x_2) \vee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



Carry en circuitos

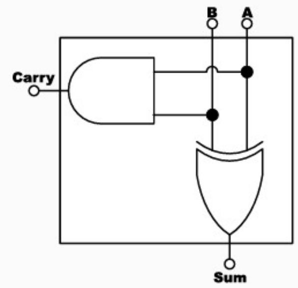
El carry debe colocarse en los circuitos en la suma porque en caso de no hacerlo, se nos pierden casos.

A	B	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
~~1~~	~~1~~	~~0~~	~~1~~



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:

A	B	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

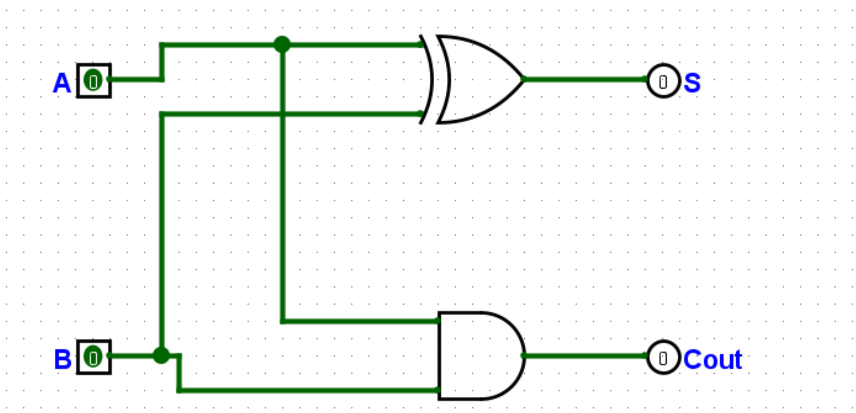


Nota: El carry es representado con un AND porque en la tabla de verdad solo da uno cuando A = 1 y B = 1. Luego, la función Sum es un XOR.

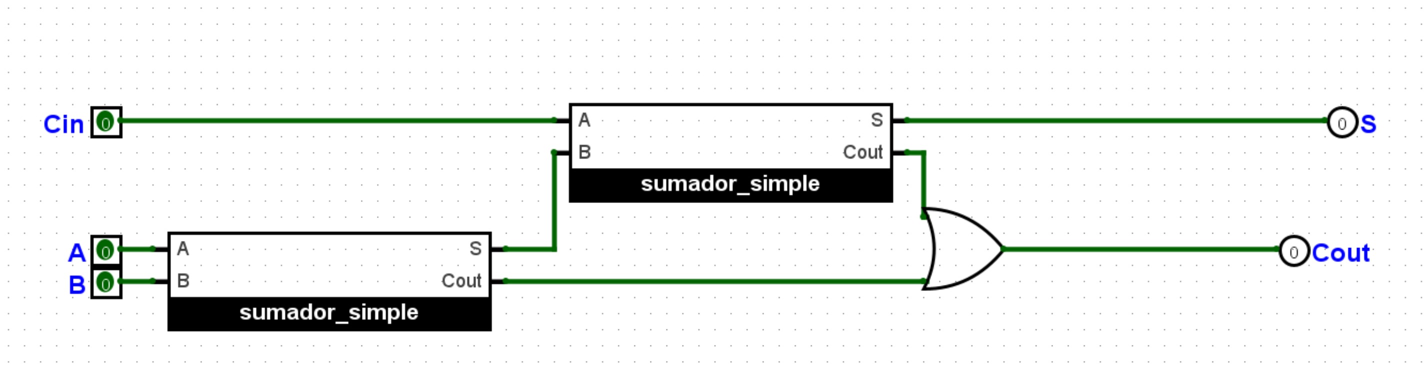
Sumadores

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



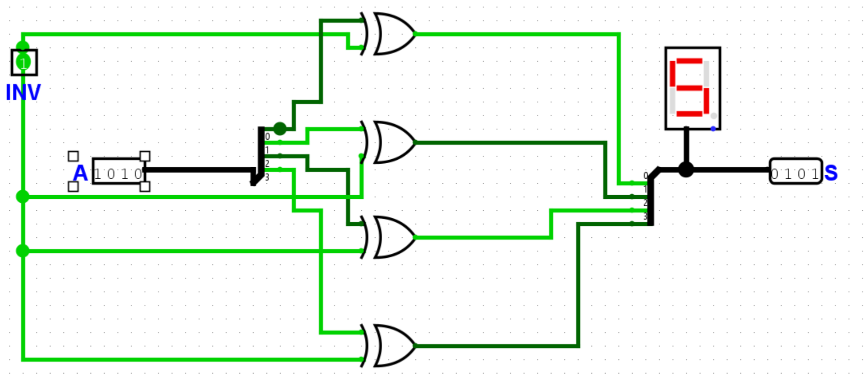
Veamos un ejemplo con un sumador completo de 3 entradas: Si para dos entradas necesitabamos un sumador simple, para 3 entradas necesito 2. Porque es (A+B) y luego res+C



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

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.

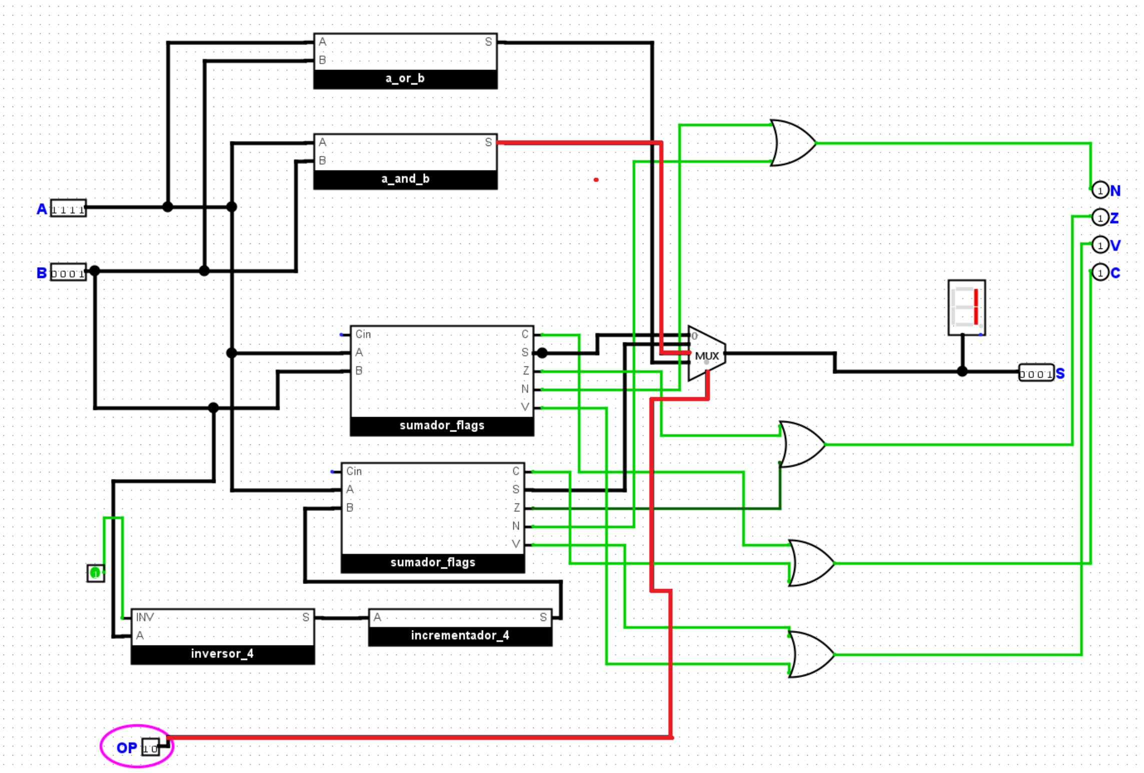
Para poder calcular la cantidad de entradas de control c_n que necesito para una cantidad m de entradas de datos e_m hago el siguiente cálculo: m $< 2^l$ hasta que me pase por primera vez.

- 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

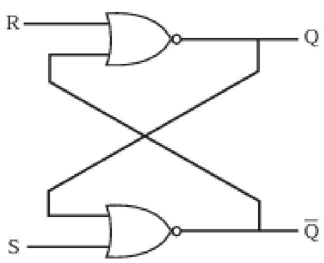
En un circuito combinatorio el tiempo que tarda la salida en estabilizarse depende de la cantidad de capas de compuertas (latencia). Para enfrentar el problema usamos secuenciales.

Latchs

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	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$ [1]
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:

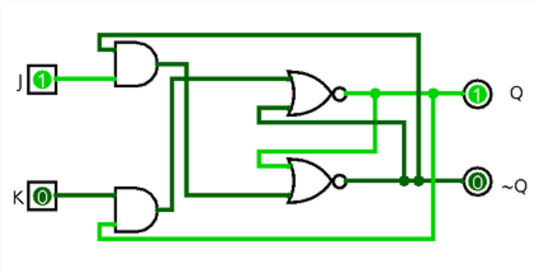


Tabla de verdad:

J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	$\overline{Q}*$	$Q*$

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

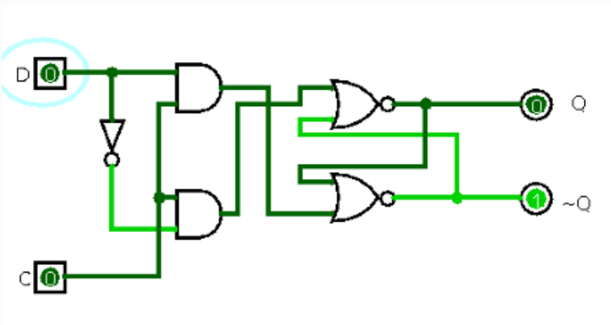
Latch D

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.

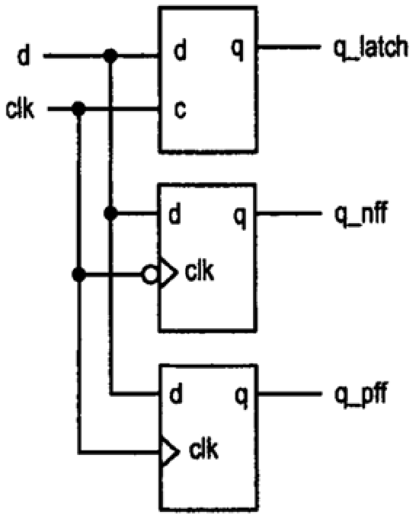
Este circuito es estable en todos los estados pero los tiempos no se pueden predecir porque dependen de D y puede causar carreras si existe un lazo en el circuito externo

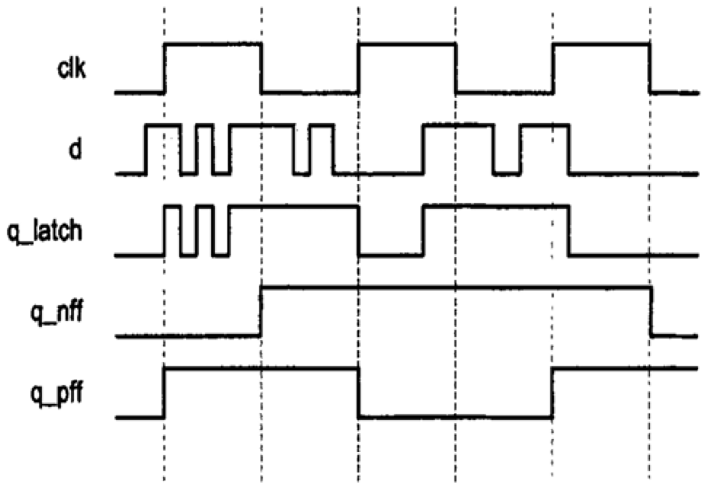
- Cuando D está apagado y C está prendido, se memoriza C.
- Si ambos están prendidos, se memoriza D.
- En cualquier otro caso, devuelve el valor memorizado.



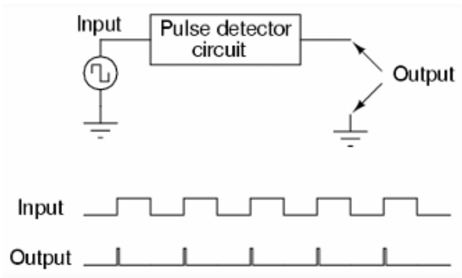
D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	1	0

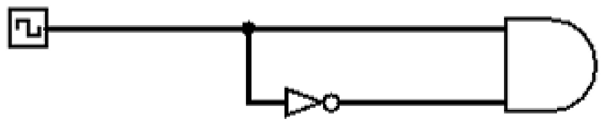
Control de transición de estados: Clock





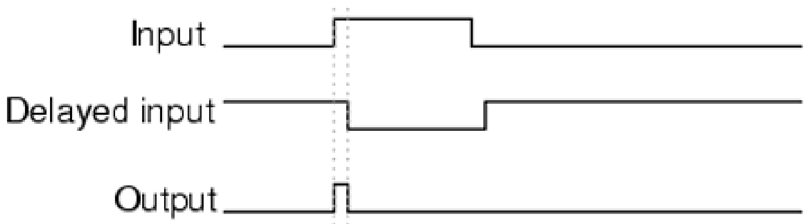
El clock que necesitamos utilizar es el 3ro. ¿Por qué? Porque nos interesa solamente memorizar o guardar los estados de los valores cuando el clock está en el pico.
No necesitamos estar constantemente escuchando cambios con el clock, sino que nos interesa solo en la subida.
Para solucionar este problema, 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.

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:

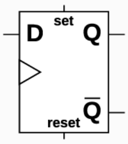


Tabla de verdad:

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

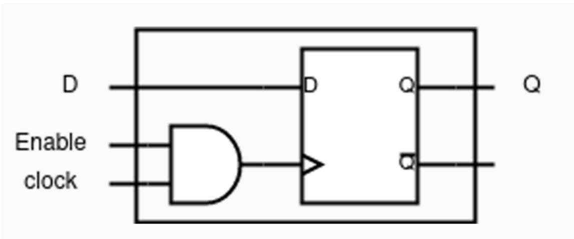
Siendo $T = n.T_{clock}$ y $T + 1 = (n+1)T_{clock}$, donde:

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

Nota: $1\uparrow$ 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.

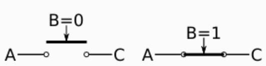


Recuerdo: El Flip Flop D solo almacena 1 bit. Si necesitaramos almacenar n bits (registro de n bits) necesitaríamos n Flip Flop D y UN solo Enable/Clock.

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

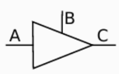
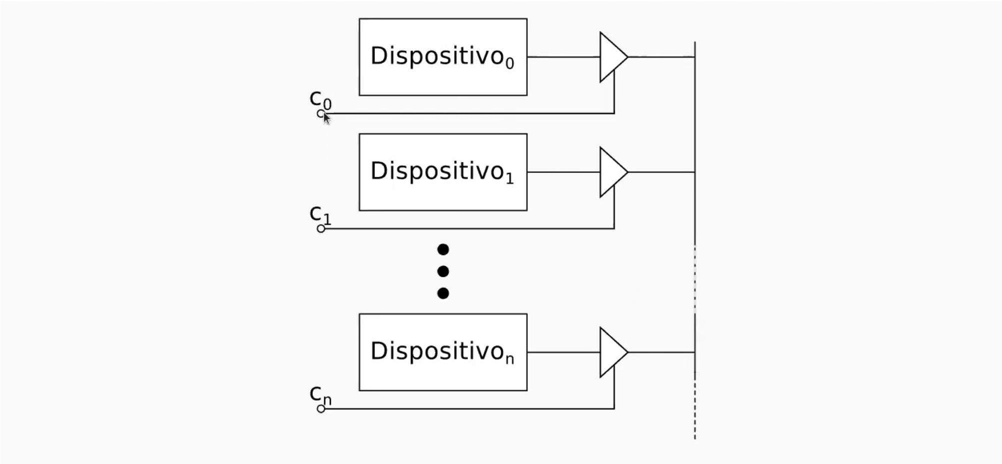


Tabla de Verdad

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

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

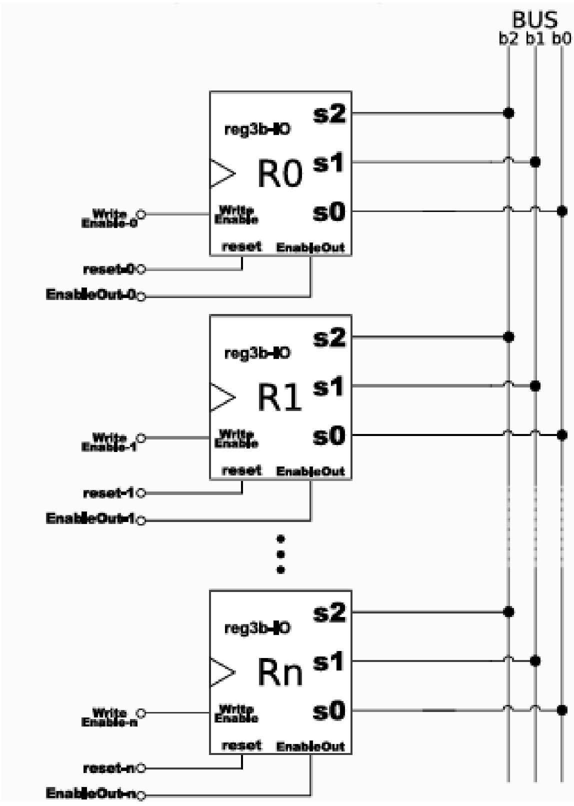
Coloca en 0 el componente. Comúnmente, el reset es asincrónico.

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

De esta manera podemos pasar datos entre registros.

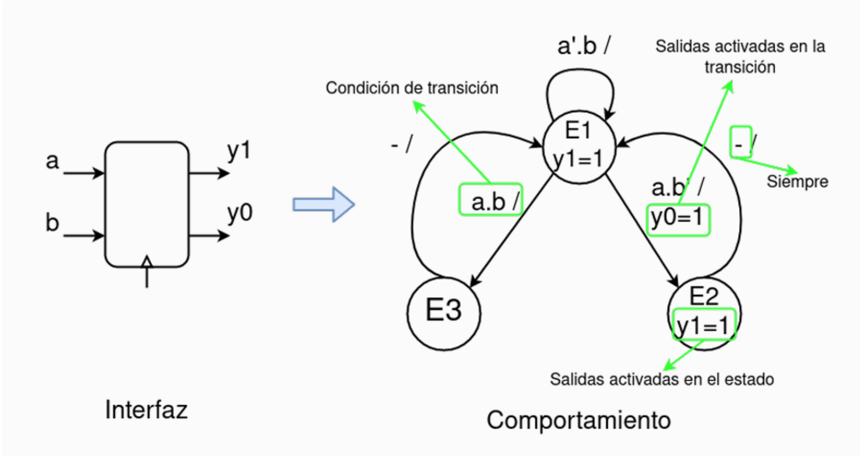
5. Máquinas de estado

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

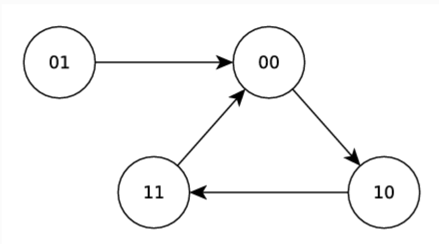
La salida depende del estado actual y las entradas.

- 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`.



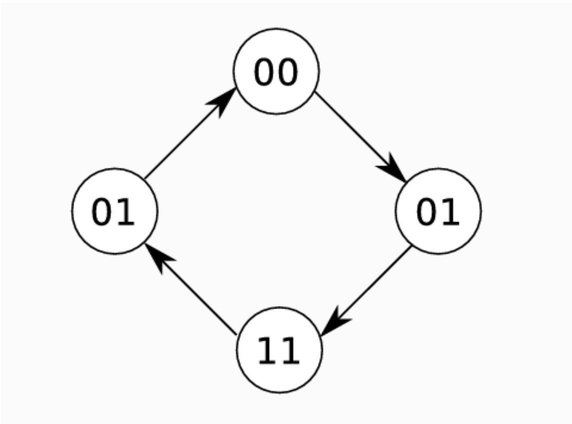
$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	1	0	0
0	0	1	0
1	0	1	1
1	1	0	0

¿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.

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

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:

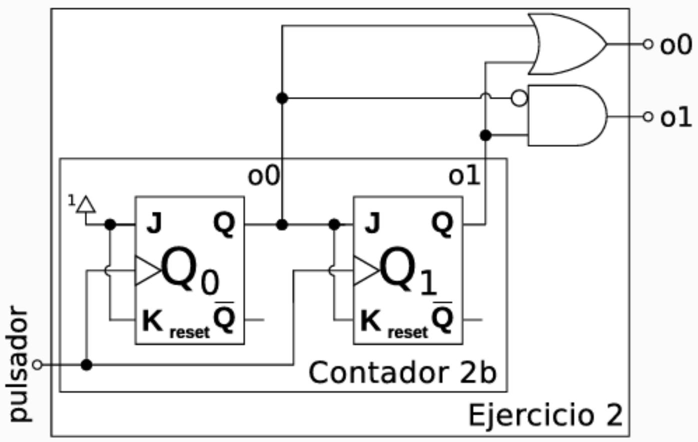
$S_1 = 00, \qquad S_2 = 01, \qquad S_3 = 10, \qquad S_4 = 11$

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

$$o_0 = Q_1 + Q_0 \quad \text{por producto de sumas}$$

$$o_1 = Q_1 \ . \ \bar{Q}_0 \quad \text{por suma de productos}$$



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.

Operaciones en Risc V

Reciben el nombre de mnemónico e indica el tipo de operación que queremos realizar.
Las operaciones reciben: **operandos de fuente** y un **operando destino**

$$a = b + c \equiv add\ a,\ b,\ c$$

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

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.
Calculemos: $a = b + c - d$

- $add\ t,\ b,\ c$ # t=b+c
- $sub\ a,\ t,\ d$ # a=t-d

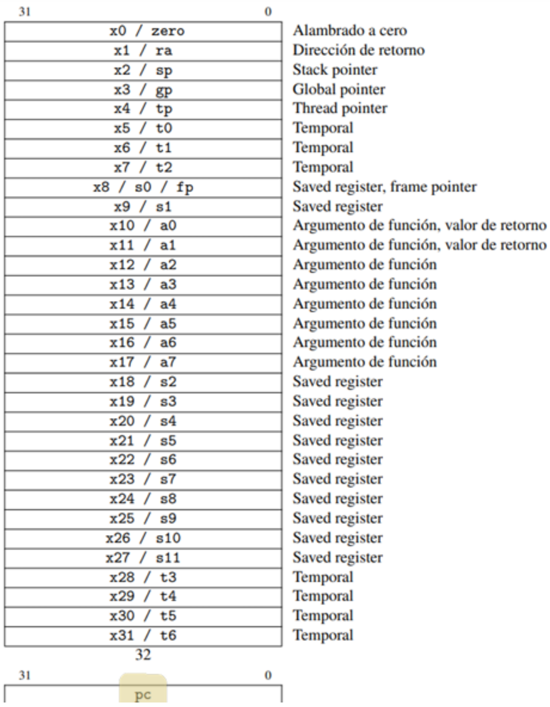
Comentarios en Risc V

Usamos # para comenzar una línea de comentarios.

Registros en Risc V - Register File

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.
- 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.
Hagamos el cálculo que habiamos hecho antes (a = b + c - d) pero con las variables en los registros.

- # s0 = a, s1 = b
- # s2 = c, s3 = d, t0 = t
- $add\ t0,\ s1,\ s2$ # t=b+c
- $sub\ s0,\ t0,\ s3$ # a=t-d

Nota: los valores que toman las operaciones no pueden ser de 32 bits porque la operación en sí ya ocupa 5 bits.
Preguntar: ¿Cómo es que s0 tiene efectivamente a **a**, s1 tiene a **b**?

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.

Asignación

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

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

Valores inmediatos de 32 bits

C	ejemplo muy específico RISC V
`int a = 0xABCDE123;`	`lui s2, 0xABCDE #s2=0xABCDE000` `addi s2, s2, 0x123 #s2=0xABCDE123`

Como los valores inmediatos son de 12 bits y se los extiende respetando el signo a 32 bits cuando realizamos una operación, cargar una constante de 32 bits requiere que hagamos dos operaciones. Primero cargamos los veinte bits más altos con la instrucción `lui`(load upper inmediate) y luego los 12 bits más bajos con un `addi` como veníamos haciendo.

¿La idea sería pasar de derecha a izquierda, o pasar lo de la izquierda a la derecha?
¿Por qué los bits más altos son considerados ABCDE?
¿Cual es la regla para decir cuales son altos o bajos? (en este contexto manual) Lo que no entiendo es, los bits mas altos están seleccionados a mano (ABCDE) y los más bajos también (123) pero no podrían ser los más altos ABCD y los más bajos E123?

C	RISC V
`int a = 0xFEEDA987;`	`lui s2, 0xFEEDB #s2=0xFEEDB000` `addi s2, s2, -1657 #s2=0xFEEDA987`

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.

RISC V permite acceder a la memoria con índices (direcciones) de 32 bits osea 4.294.967.296 índices posibles. El índice apunta a un byte en particular.

Escritura y Lectura en Memoria

Lectura: Instrucción lw (load word); Escritura: Instrucción sw (store word).

Las direcciones se definen como:

$$direccion = base + desplazamiento$$

base: valor de un registro
desplazamiento: constante con signo de 12 bits

Lectura en Memoria (asignación a variable)

C	RISC V
`int a = mem[2];`	`#s7 = a, s3 = mem` `lw, s7, 8(s3)`

lw, variable, desplazamiento(lugarEnMemoria)
elem = cantidad de bytes por dato ∗ indice al que quiero llegar.
s3 tiene la posición de memoria en el registro donde está el arreglo.

Escritura en Memoria

sw, variable, desplazamiento(lugarEnMemoria)
Programa: $mem[5] = 33$

- # s3 = mem
- $addi\ t3,\ zero,\ 33$
- $sw,\ t3,\ 20(s3)$

El programa anterior crea una variable en el registro temporal t3, le asigna el valor de 33 y luego en la memoria posición 5 del array le asigna t3. Preguntar: Como t3 es un registro temporal, cuando se borra luego de guardar el valor en el array?

Instrucciones lógicas en RISC V

Sean dos registros cualesquiera, las operaciones lógicas se hacen bit a bit.
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

Or: Combina dos registros que solo tienen asignada la parte alta y baja. Ej: $0xFEED0000 \vee 0x0000F0CA \equiv 0xFEEDF0CA$
and: Limpia las partes de un registro, si quisieramos preservar solamente la parte baja de $0xBABAC0C0$ podemos hacer $0xBABAC0C0 \wedge 0x0000FFFF \equiv 0x0000C0C0$ Preguntar xor: Negación del valor

Instrucciones de desplazamiento

- sll (shift left logical): desplaza a la izquierda tantas unidades como se diga, como es logico 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.

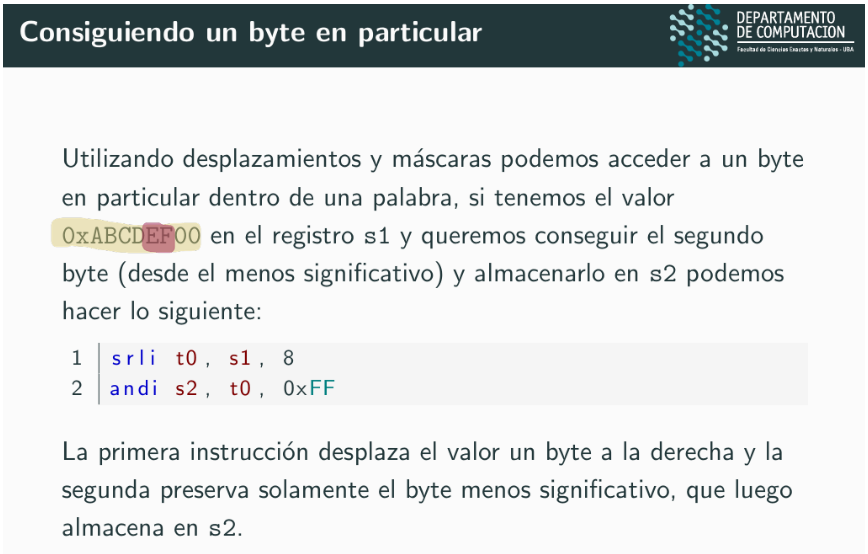
Nota: También existen las versiones de estas operaciones con el immediate (i) para usar una constante.

s5	1111 1111	0001 1100	0001 0000	1110 0111
slli t0, s5, 7	1000 1110	0000 1000	0111 0011	1**000 0000**
srli t0, s5, 17	**0000 0000**	**0000 0000**	**0**111 1111	1000 1110
srai t0, s5, 3	**111**1 1111	1110 0011	1000 0010	0001 1100

Lo que hago yo esto para hacerlo fácil, en el caso de 17 por ejemplo es: como me piden mover 17 a la derecha, borro los 17 últimos (xq no tengo espacio donde guardarlos), reordeno de a pares de 4 y despues donde faltan agrego de la izq.

- 1111 1111 & 0001 1100 & 0001 0000 & 1110 0111
- 1111 1111 & 0001 110 $\Longrightarrow$ acá tengo que reordenar xq de la der no me pueden quedar 3 bits solos
- 111 1111 & 1000 1110 $\Longrightarrow$ reordené
- 0000 0000 & 0000 0000 & 0111 1111 & 1000 1110 $\Longrightarrow$ agregué los 0's

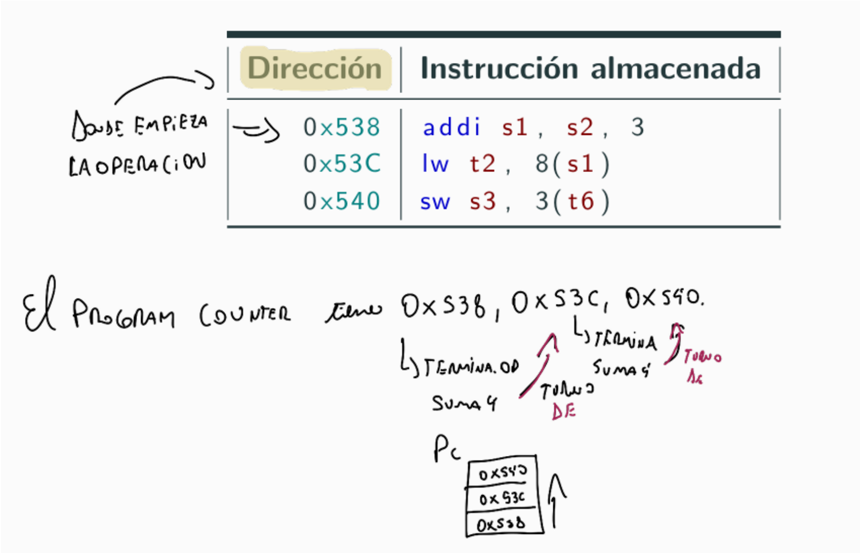
Byte en particular con desplazamiento



Preguntar, no entendi

Importancia del Program Counter

Recordatorio: Cada dirección se incrementa en múltiplos de 4 porque las instrucciones ocupan 4 bytes.
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.



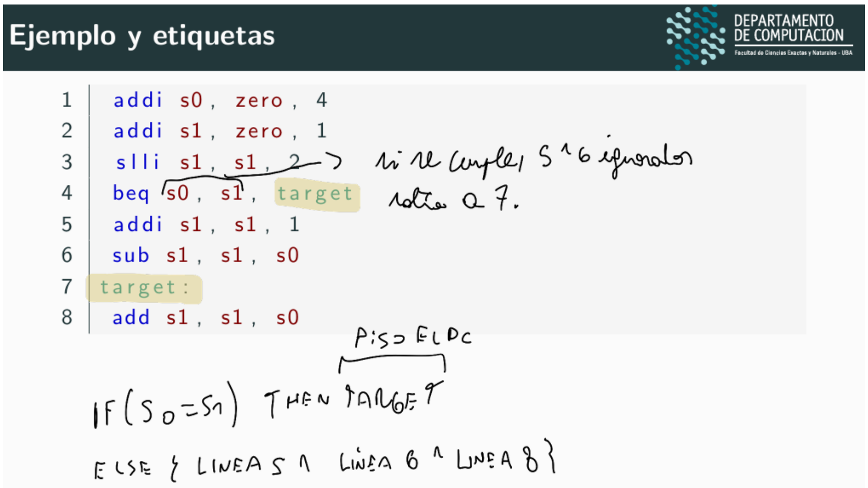
El Program Counter va cambiando la instrucción que está ejecutando gracias al aviso del procesador.

Control del flujo de ejecución

Se va pisando el PC (Program Counter).

Etiquetas en RISC V

Nos permiten ponerle nombre a subproblemas.



Saltos 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 operando son distintos.
- blt(branch if less than): reemplaza el PC si el primer operando es menor que el segundo.
- bge(branch if greather 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).

```
 j target
 srai s1, s1, 2
 addi s1, s1, 1
 sub s1, s1, s0
target:
 add s1, s1, s0
```

C

```c
// calcula el valor de x
// tal que 2 a la x es 128
int pow = 1:
int x = 0;

while(pow != 128){
    pow = pow * 2;
    x = x + 1;
}
```

RISC V

```
#s0=pow, s1=x
 addi s0, zero, 1
 add s1, zero, zero
#t0=128
 addi t0, zero, 128
while:
 beq s0, t0, fin
 slli s0, s0, 1 #pow=pow*2
 addi s1, s1, 1 #x+=1
 j while
fin:
```

C

```c
int i;
int scores[200];

for(i = 0; i < 200; i = i + 1){
    scores[i] = scores[i] + 10;
}
```

Dirección absoluta del dato

RISC V

```
#s0=dir. scores, s1=i
 addi s1, zero, 0
 addi t2, zero, 200
for:
 bge s1, t2, fin
 slli t0, s1, 2
 add t0, t0, s0
 lw t1, 0(t0)
 addi t1, t1, 10
 sw t1, 0(t0)
 addi s1, s1, 1
 j for
fin:
```

Si llega, sale

Desplazamiento

Base

- se inicializa una variable permanente con el valor zero.
- se inicial la variable temporal en 200.
- el for se realiza mientras s1 sea menor a t2.
- se inicializa una variable temporal t0 con el valor de s1 (¿que seria el 2, entiendo que un desplazamiento logico hacia la izquierda pero para que?).
- a la variable temporal t0 se le suma s0 (s0?)
- se inicializa una variable temporal t0 que tiene el valor de t0 en el indice 0.
- a la variable temporal t1 se le suma 10 (scores[i] + 10)

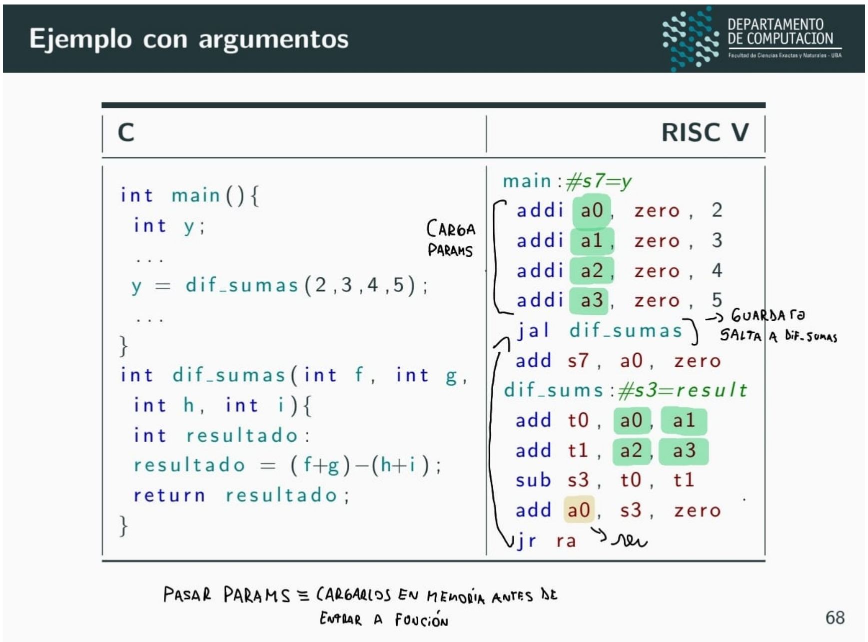
- se cambia el valor de t1 indicando que tiene el valor de t0 en el indice 0.
- se modifica la variable de s1 (incrementadora) sumandole 1 unidad.

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.



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.ar