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.

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.

$$\mathbf{1}0000000\mathbf{0}_{(2)}$$

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 $m > 0$

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

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

Borrow

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

```
  0111
- 1000
  ====
  1011

  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)

$$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array} \qquad \begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array} \qquad \begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array} \qquad \begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array} \qquad \begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$$

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)

$$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array} \qquad \begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = Overflow \end{array} \qquad \begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = Overflow \end{array} \qquad \begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array} \qquad \begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$$

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)

¿Cuando hay overflow, carry o borrow?

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

```
  0101
+ 0110
  ====
  1011  - OV: si - C: no

  1000
+ 1100
  ====
  10100 - OV: si - Carry si

  1010
- 0100
  ====
  1000 - OV: si - Borrow no

  0100
- 1000
  ====
  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 $1111111111_{(2)}$?

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

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

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.

$$n = 3 \land m = 8$$

- Signo + Magnitud y exceso m: Se extiende con 0's luego del signo.
  - En 3 bits, -2 = 110
  - En 8 bits, -2 = **1**0000010
- Complemento 2 (C_2): Se extiende con el bit más significativo.
  - En 3 bits, -2 = 110
  - En 8 bits, -2 = **1**1111110

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:

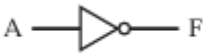
$$Sea\ F = X + \bar{Y}$$

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 ⊕ 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.

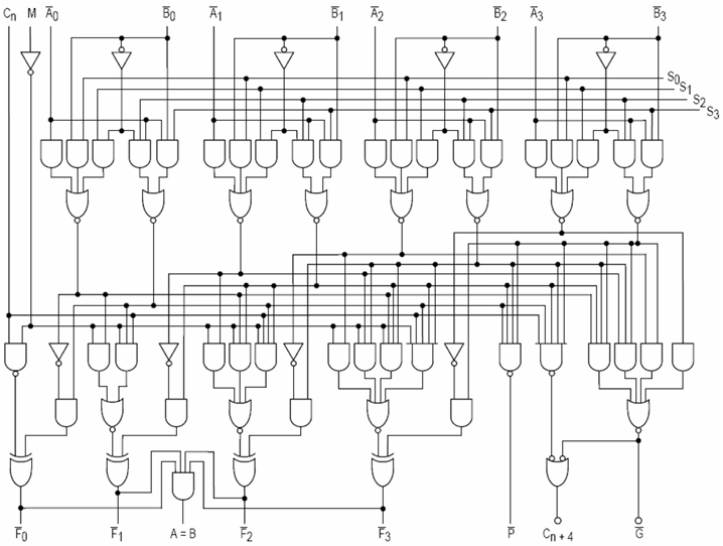
- NAND = $\overline{A \wedge B}$

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

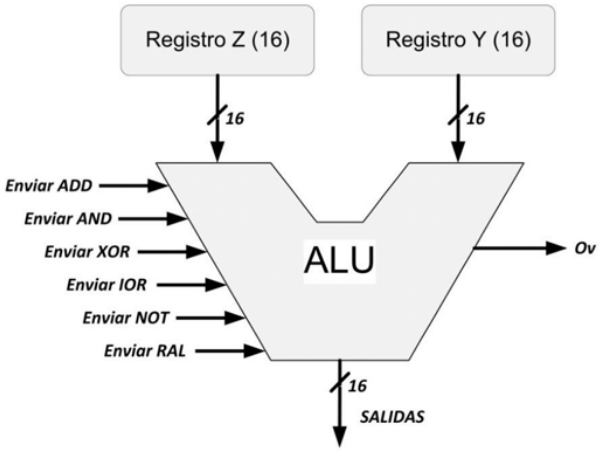
Compuertas en SystemVerilog

- 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

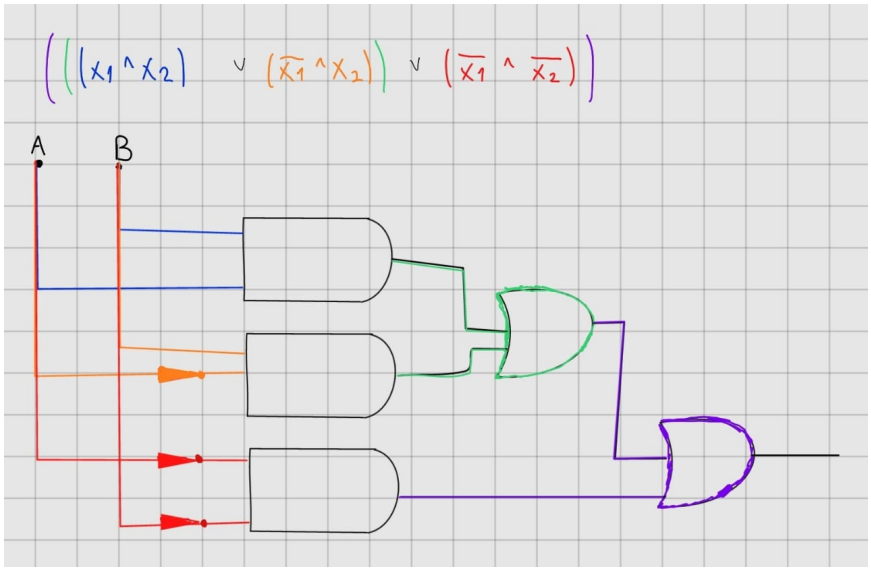
- 1. Solo consideramos de la función F, las filas verdaderas.
- 2. Cada fila verdadera tendrá su índice, y en ese índice estarán los valores de cada variable proposicional. Representamos a esa fila verdadera como t_i
- 3. Realizamos la conjunción de todas las variables de ese t_i
- 4. Realizamos la disyunción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

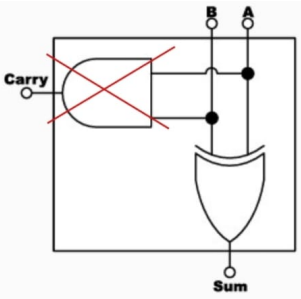
- F es solamente verdadera en la primera, segunda y tercer fila por lo tanto tenemos $t_1\ t_2\ y\ t_3$
- Por cada fila, hacemos la conjunción de los valores
  - $x_1 \wedge x_2$
  - $\bar{x_1} \wedge x_2$
  - $\bar{x_1} \wedge \bar{x_2}$
- Realizamos la disyunción de todos los t_i
  - $(x_1 \wedge x_2)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



Carry en circuitos

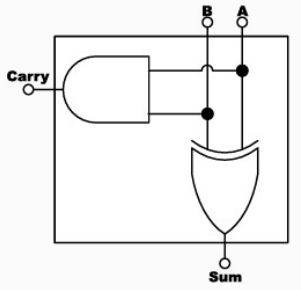
El carry debe colocarse en los circuitos en la suma porque en caso de no hacerlo, se nos pierden casos.

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

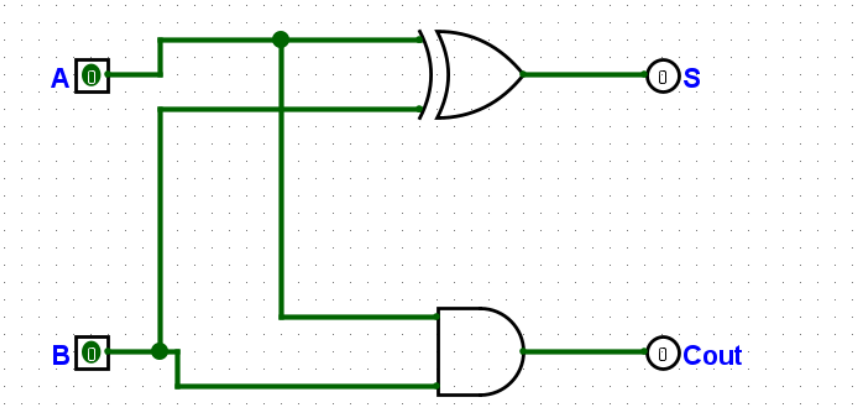


Nota: El carry es representado con un AND porque en la tabla de verdad solo da uno cuando A = 1 y B = 1. Luego, la función Sum es un XOR.

Sumadores

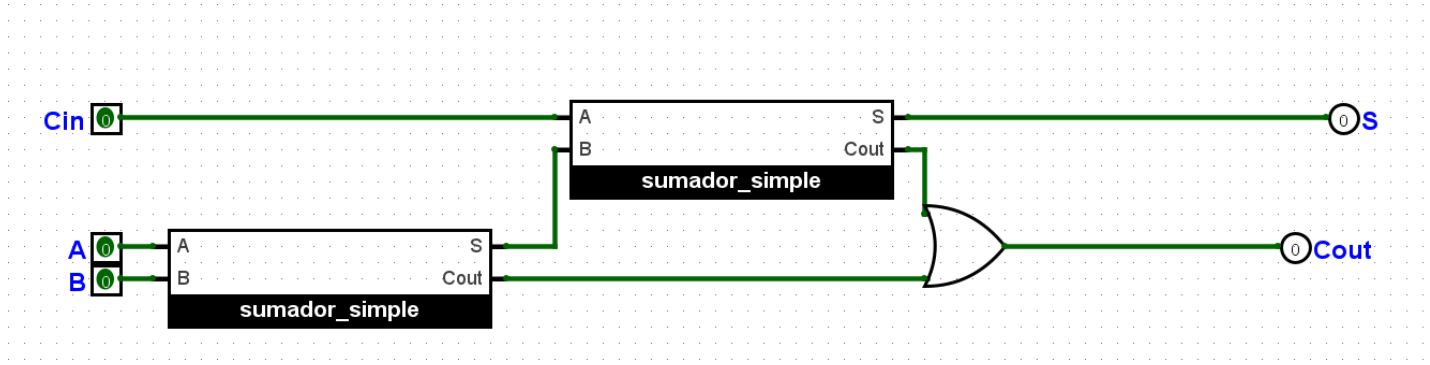
Los sumadores nos sirven para poder realizar operaciones entre bits. Es importante recalcar que llamamos half-adder a un sumador de 1 bit, donde solamente tiene una entrada A de 1 bit y una entrada B de 1 bit. Un sumador de 1 bit requiere:

- 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,

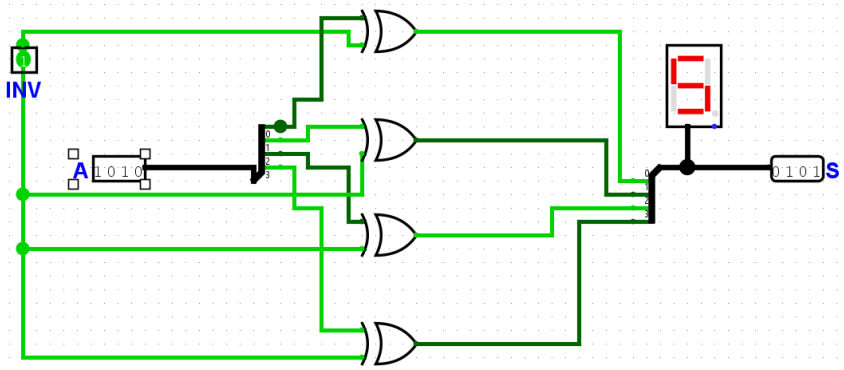
para 3 entradas necesito 2. Porque es (A+B) y luego res+C



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. Es una especie de switch/case en programación donde la condición del switch acá se llama entrada de control.

- Entradas de control: se indican de la manera c_n
- Entradas de datos: se indican de la manera e_n

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.

Timing

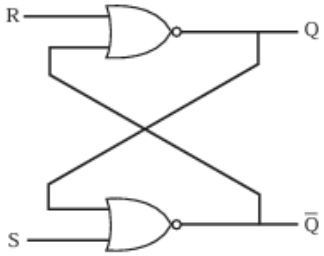
En un circuito combinatorio el tiempo que tarda la salida en estabilizarse depende de la cantidad de capas de compuertas (latencia). Para enfrentar el problema usamos secuenciales.

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:

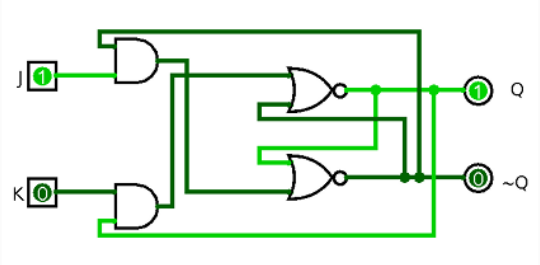


Tabla de verdad:

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)

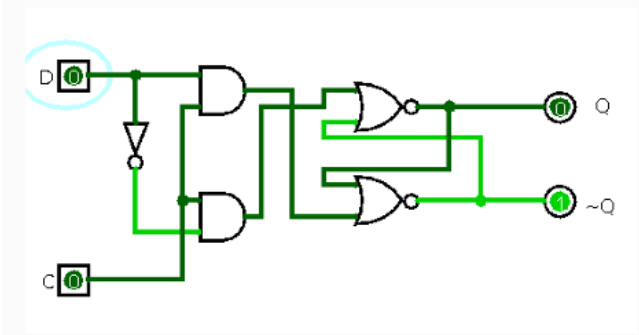
Latch D

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.

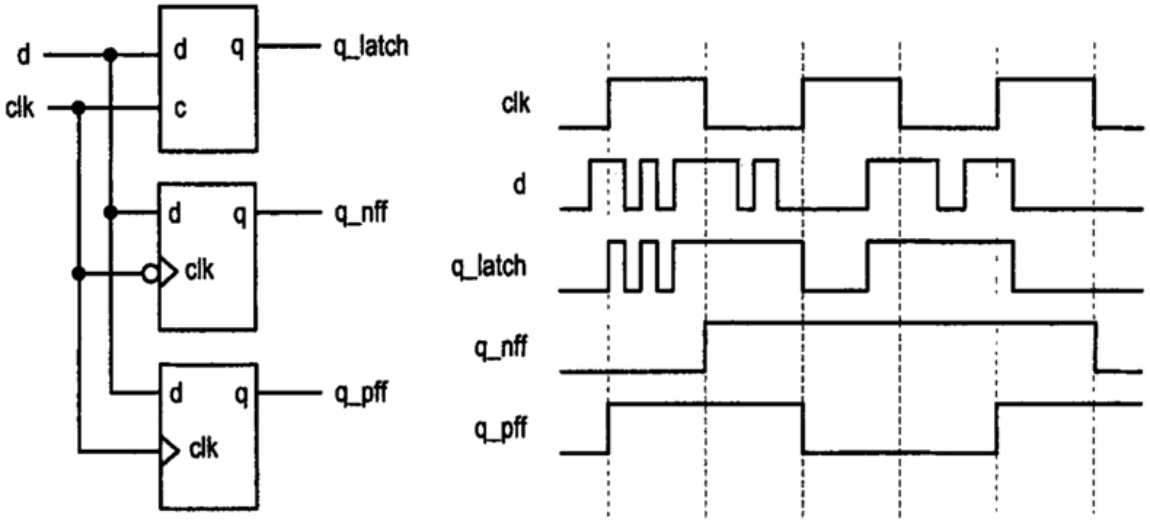
Este circuito es estable en todos los estados pero los tiempos no se pueden predecir porque dependen de D y puede causar carreras si existe un lazo en el circuito externo

- 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.

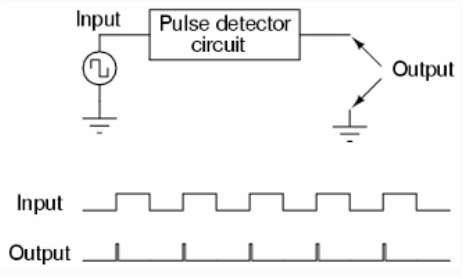


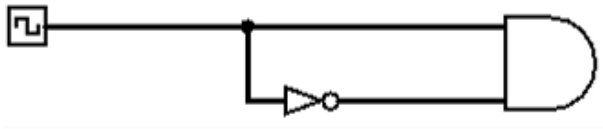
D	C	Q	$\overline{Q}$
1	0	$Q*$	$\overline{Q}*$
0	1	0	1
0	0	$Q*$	$\overline{Q}*$
1	1	1	0

Control de transición de estados: Clock



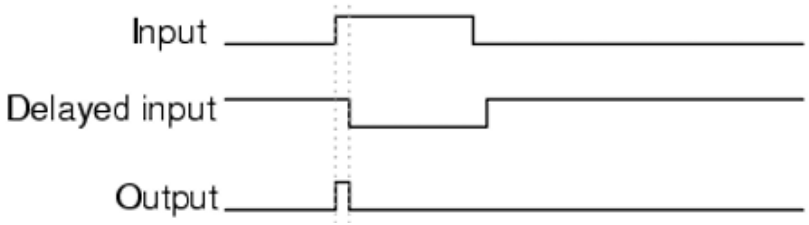
El clock que necesitamos utilizar es el 3ro. ¿Por qué? Porque nos interesa solamente memorizar o guardar los estados de los valores cuando el clock está en el pico.
No necesitamos estar constantemente escuchando cambios con el clock, sino que nos interesa solo en la subida.
Para solucionar este problema,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





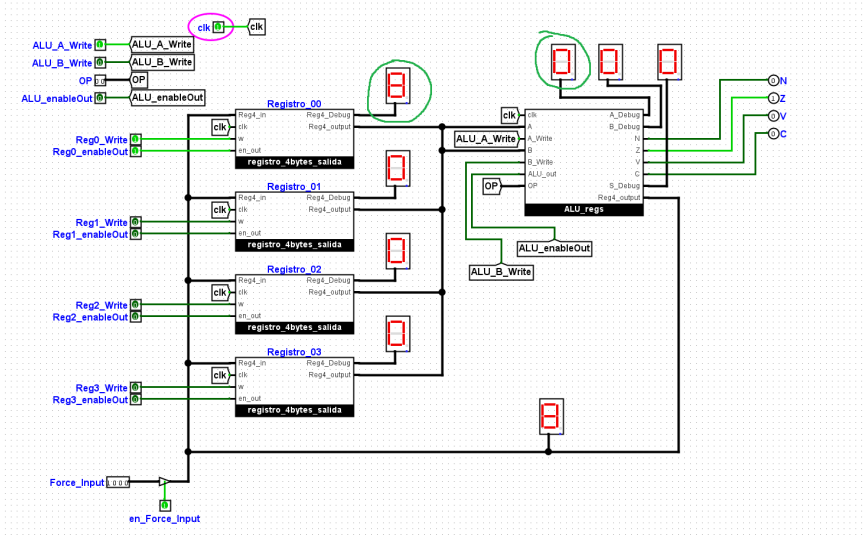
(a) Detector de pulso implementado usando una compuerta AND.

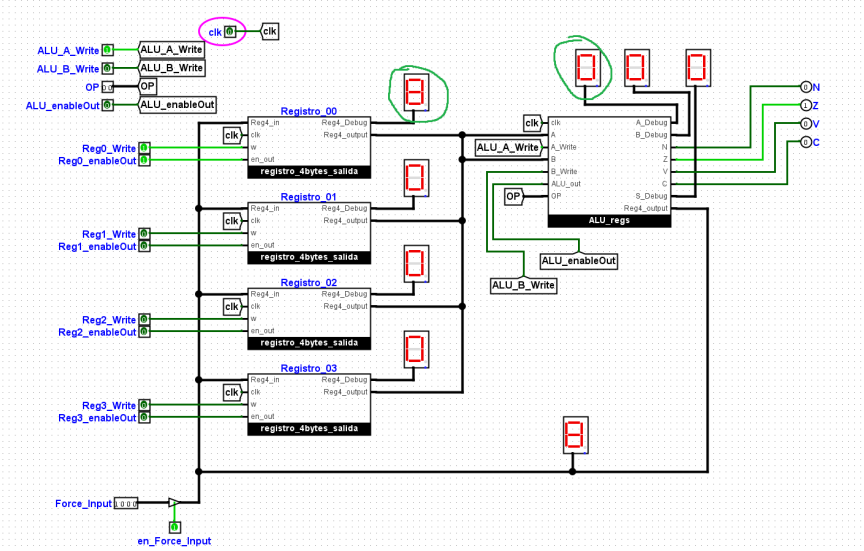
Es importante notar, que el detector de pulso dará 1 (el pico) en algunos casos porque la compuerta NOT tiene un pequeño delay para poder negar la entrada. A continuación se muestra un ejemplo de esto.

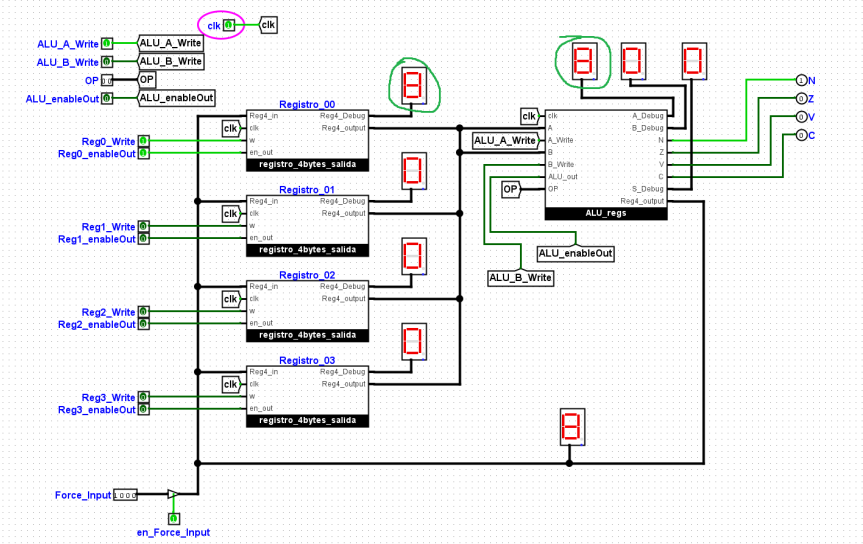


La línea punteada indica el tiempo que tardó la entrada en ser negada. En ese momento es donde se nos ejecutan los picos que nosotros necesitamos.
Si mandamos input = 1, el primer momento queda 1 AND 1 y el AND es verdadero, pero luego de un momento queda 1 AND 0 y ahora la señal vuelve a estar baja. 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.

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.







- Se activan las instrucciones de `Reg0_Write`, `Reg0_enableOut` (que permite exponer su valor a los demás) y se activa `ALU_A_WRITE` para que la ALU reciba el valor que expone `Reg0_enableOut`.
- La primera operación que toma el clock en el flanco de subida es la escritura del número en `Registro_00`.
- Al apagar el clock, todo sigue igual.
- 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:

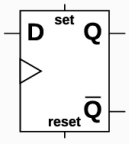


Tabla de verdad:

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

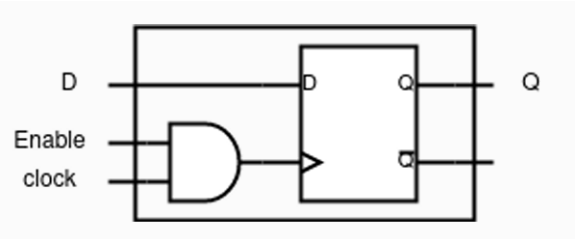
Siendo $T = n.T_{clock}$ y $T + 1 = (n + 1)T_{clock}$, donde:

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

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

Registros

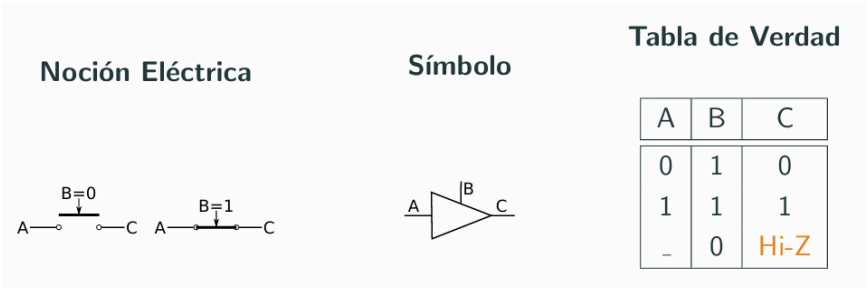
Para poder escribir registros necesitamos una entrada de control Enable que nos permitirá decirle al Flip-Flop cuando nosotros queremos permitir que nos cambie el valor / escriba la memoria.
Este flag de Enable deberá ir en un AND con el Clock, entonces si Enable = 1 y el Clock está en 1, entonces se 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**

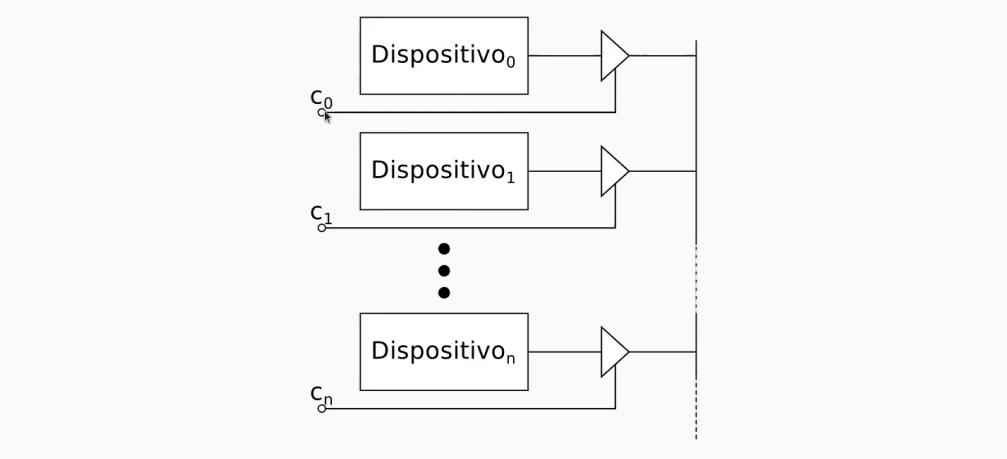


A	B	C
0	1	0
1	1	1
_	0	Hi-Z

En la materia, una combinación basura de un componente de tres estados es que haya más de una entrada de control prendida 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.
Nota: Antes de prender cualquier entrada de control, hay que asegurarse que entrada de dato tenga el valor que deseamos cargar.

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

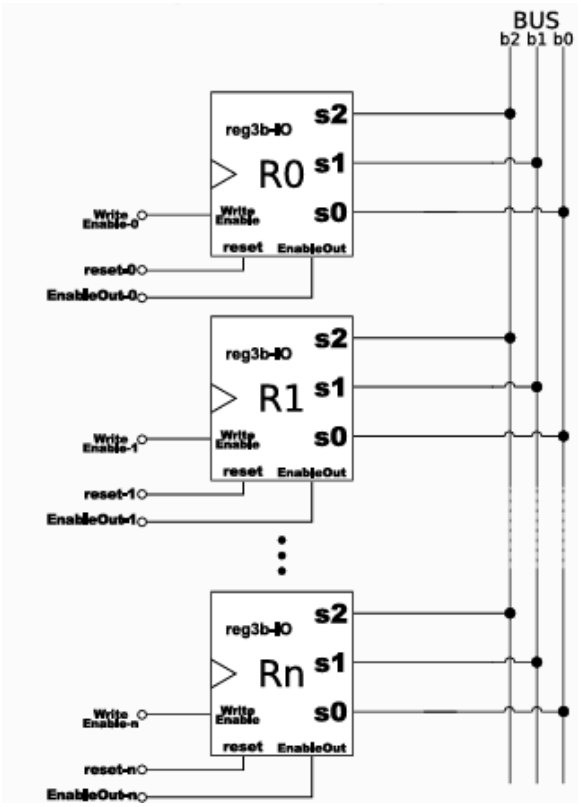
Coloca en 0 el componente. Comúnmente, el reset es asincrónico.
Para entender el concepto de asincrónico, véase reset asincrónico - sincrónico

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 ← 0
- Deshabilito Enable Out en R1 - EnableOut-1 ← 0

De esta manera podemos pasar datos entre registros.

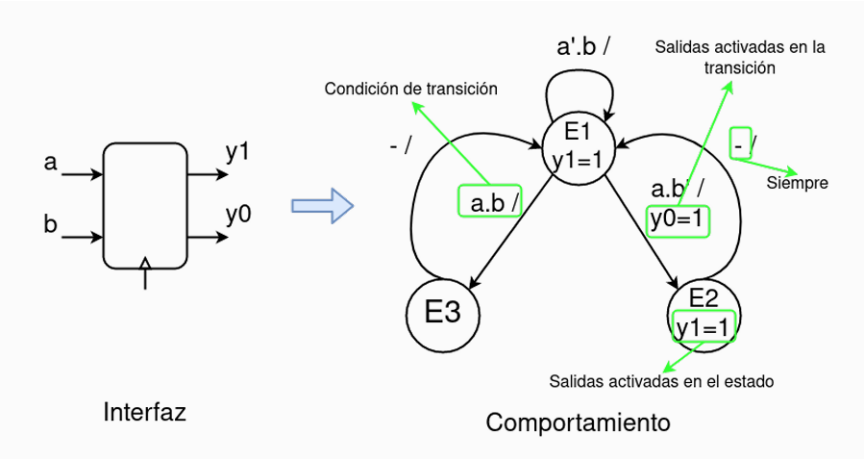
5. Máquinas de estado

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

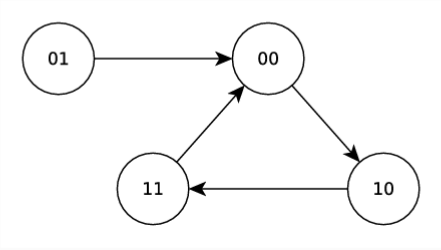
La salida depende del estado actual y las entradas.

- 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`.



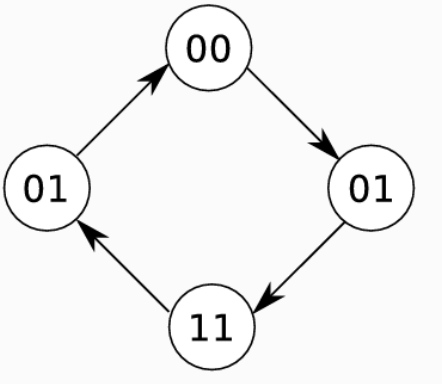
$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	1	0	0
0	0	1	0
1	0	1	1
1	1	0	0

¿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.

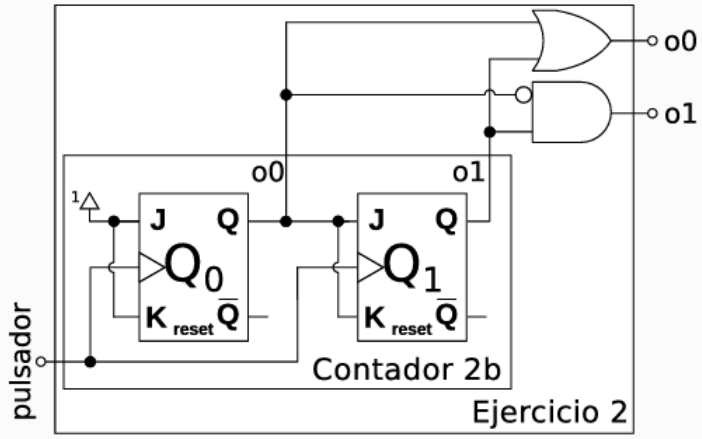
Renombremos a los estados con nombres únicos, por ejemplo:
$S_1 = 00, \quad S_2 = 01, \quad S_3 = 10, \quad S_4 = 11$

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

$o_0 = Q_1 + Q_0$ por producto de sumas

$o_1 = Q_1 \; . \; \bar{Q_0}$ 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.

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.
- Los valores se extienden por el bit más significativo
- Las operaciones se llevan a cabo en el procesador.
- Todos los datos que se usan son de 32 bits.
- Los valores inmediatos toman como máximo, valores de 12 bits.
- Todas las operaciones que implican accesos a memoria, los valores inmediatos (se extienden a 12 bits) se indican en bytes. Ej: lw, sw.
  lw s7, 8(s3): el 8 RISC-V lo lee como 0000 0000 0100
- Todas las operaciones que implican operaciones aritméticas o lógicas, los valores inmediatos (se extienden a 12 bits) 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.

Importante: Las instrucciones se guardan en la memoria RAM durante la ejecución del programa. Es decir, al ejecutar el programa carga todas las instrucciones de una en la RAM, luego, por cada línea que va pasando, agarra ese puntero a la RAM y hace el fetch & decode & execute.

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.

Comentarios en Risc V

Usamos # para comenzar una línea de comentarios.

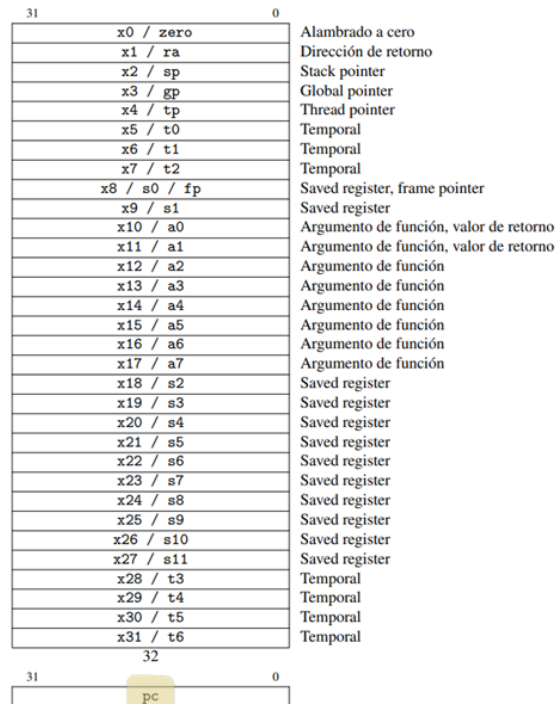
Program Counter

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.

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 (si llegase a ser menor de 12 bits, se extiende con 0's del lado del bit más significativo) 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).**

- addi t0, t0, 0xFF → como FF son 8 bits, se extiende a 12 0x0FF. Ahora como ya es de 12 bits, extiende a 32 por signo → 0x000000FF
- addi t0, t0, 0xFF → como FFF son 12 bits, se extiende a 32 por signo → 0xFFFFFFFF

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.

Control de valores inmediatos mayores a 12 bits (positivos)

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.
Explicación genérica de como cargar un valor inmediato de 32 bits:

- Agarra los primeros 20 bits en binario.
- Aplica un lui: carga los primeros 20 bits, pero antes extiende el número con 0's desde la izquierda. Luego, guarda el resultado del lui.
- Hace un addi utilizando el resultado del proceso anterior y los 12 bits menos significativos del número inicial (habiendo sido esta parte extendida a 32 bits).

1. Véase anexo para un ejemplo de esta aplicación de lui y addi con valores inmediatos mayores a 12 bits.

Control de valores inmediatos mayores a 12 bits (negativos)

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`.

Instrucciones Sin Signo

Figuran al final de la instrucción con la letra *u* que indica **unsigned**.

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, en caso de que no haya un puntero a la RAM pero si un valor, se pone ese valor como si fuese una posición de memoria en la RAM. Solo carga UNA palabra.

$$lw\ destino,\ desplazamiento(registroALeer)$$

Ejercicio TypeScript:

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

Equivale en RISC-V a:

```
# s1: numbersArr -> cargado a traves de .data, s7: five
lw s7, 20(s1)   -> carga el valor que esta en la palabra del byte 20 al 23
```

Donde el 20 sería el offset para movernos desde s1, es decir, 20 bytes (5ta palabra) Mapa mental: a ← b
Instrucción la (load address) Carga en el registro indicado, un puntero a un registro. NO pueden hacerse desplazamientos.

$$la\ registro,\ punteroARegistro$$

```
.data
    arr: .word 1, 2, 3, 4, 5 #inicializo array. esto se ensambla, y arr se guarda en una posicion de memoria de la ram.
    n: .word 5 #inicializo la longitud del array. para saber hasta donde poder iterar. esto se ensambla y n se guarda en una
         posicion de memoria de la ram.
.text
    la a0, arr #cargo en a0 la posicion de memoria donde esta el array (es decir, el puntero)
    lw a1, n #cargo el valor de n, osea a1 = 5
```

Lectura en Memoria en Half-Words y Bytes

Además de poder cargar words (palabras) podemos cargar half-words y bytes. ¿Para qué necesitamos esto? Bueno, en muchas ocasiones la información viene de esa manera. Recordemos el por qué cargabamos words:

```
.data
    arr: .word 0x11111111 0x22222222 0x33333333
.text
    la t0, arr
    lw t1, 0(t0)
```

En este caso, cargamos la primera palabra de t0 que, en este caso es 0x11111111.
Cargando Half-Words Pero, ¿qué pasa en este caso?

```
.data
    arr: .half 0x81BC 0x00F0 0xA200 0x1000
.text
    la t0, arr
    lh t1, 0(t0)
```

Acá es realmente importante notar, que como una half-word no son 32 bits, RISC-V debe extender el valor para operar antes. Por lo que, cargar 0x81BC como half-word guardaría en el registro t1 el valor de 0xFFFF81BC. Nótese que completa con F pues como el MSB de 8 (1000), una cadena de 4 bits de 1 forma F.
Sabiendo esto, ¿qué pasará a la hora de querer cargar bytes?
Cargando Bytes

```
.data
    arr: .byte 0x2C 0xF0 0x00 0x00
.text
    la t0, arr
    lb t1, 0(t0)
```

Como claramente cargar un byte no son 32 bits, sino que solamente tenemos 8, RISC-V extenderá el valor antes de usarlo para operar. Luego, cargará en t1 el valor de 0x0000002C. Nótese que completa con 0s pues el MSB de 2 (0010) es 0.

Escritura en Memoria

Instrucción sw (store word)
sw guarda un contenido dado, en un registro en una posición que esté en ram.

$$sw\ registroAGuardar,\ desplazamiento(destino)$$

Ejercicio TypeScript:

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

Equivale en RISC-V a:

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

Mapa mental: a → 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.

Position Independent Code

El código independiente de su posición significa que todos los branches (saltos) a instrucciones y referencias a datos en un archivo son correctos independientemente de donde sea puesto el código.
Básicamente, cuando no hablamos de posiciones de memoria hardcodeadas a la hora de hacer saltos condicionales, incondicionales o a la hora de escribir en memoria o leer de memoria.

Tener código que NO es position independent code es más propenso a errores porque dependemos de qué esperemos exactamente que lo que estamos cargando tenga lo que nosotros creemos por si dependemos de eso para hacer otra operación.

- beq a0, a1, #0x12 //position dependent code porque no sé que tiene #0x12 pero quiza después agrego otra línea y #0x12 no tiene lo que yo espero para saltar.
- beq a0, a1, fin //position independent code porque el ensamblador se encarga de buscar en memoria donde está el fin y sin importar donde coloque fin en el código, siempre lo encontrará.

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.

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. Un uso común es dividir o multiplicar por 2 sin mucho esfuerzo.

- sll (shift left logical): desplaza a la izquierda tantas unidades como se diga, como es logico se añaden 0's a la derecha.
  - Ej: slli en una unidad multiplica un número por dos. 010 (2) y desplazo una unidad a la izquierda el resultado es 100 (4).
- srl (shift right logical): desplaza a la derecha tantas unidades como se diga, como es lógico se añaden 0's a la izquierda.
  - Ej: srli en una unidad divide un número por dos (no conserva signo) 100 (4) y desplazo una unidad a la derecha el resultado es 010 (2)
- 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.
  - Ej: srai en una unidad divide un número por dos (conservando signo)

Nota: También existen las versiones de estas operaciones con el immediate (i) para usar una constante. **Importante**: Considere utilizar desplazamientos aritméticos si debe conservar el signo cuando divide/multiplica por dos.

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

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 #0xFF3344 donde como

anteriormente mencionamos, solo nos interesa #FF3344 y en este caso en particular: 33.
Importantísimo: Escriban las máscaras con todos los bytes, no dejen a RISC-V en decidir si extenderlo o no porque a veces lo extiende con el MSB y no nos sirve.

- 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 33 unos 8 bits (4 bits cada dígito en hexa) → srli t0, var, 8
  - El resultado es t0 = 0x00FF33
- Aplicamos la máscara de 0x33 (0x00000033) y guardamos el resultado
  - En este caso particular, tenemos andi s2, t0, 0x33.
  - Luego, s2 tiene el valor de 0x33 (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.

Véase ejercicio en anexo

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

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).

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.
Véase ejercicio en anexo

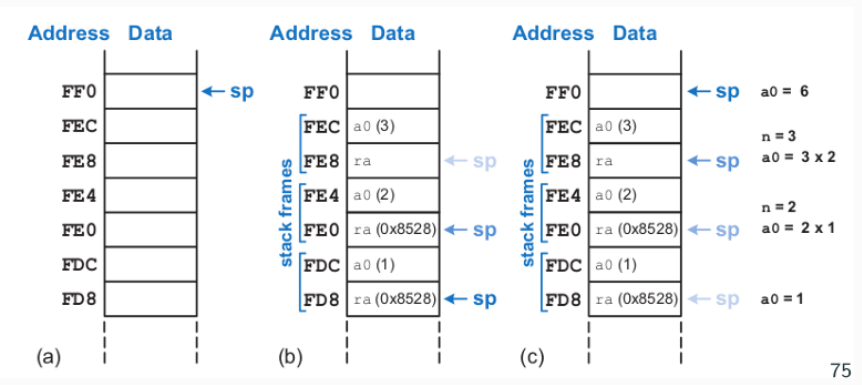
Pila (stack)

Es una parte de la memoria RAM que se utiliza para almacenar información temporaria.

RISC-V tiene su propio registro de stack pointer (sp), donde sp contiene una lista de elementos, donde cada elemento es un puntero a la RAM.

Stack frame

El Stack Frame es el espacio reservado en cada llamada recursiva para poder mantener el estado de la misma.



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) (porque la función llamada puede manipularlos, y si queremos recuperar el valor no podemos)
- Regla para la llamada: Si va a utilizar los registros permanentes (s0-s11, ra) debe guardarlos al comenzar y restaurarlos antes de retornar (si llega a tocar t0-t6 o a0-a7 no es necesario restablecerlos porque no es su obligación. lo que si debe volver a la normalidad es el stack, ra y s0-s11).

Esto es importantísimo. Si vamos a usar librerías o cosas así, si no respetamos las convenciones nos va a fallar siempre.

Preserved (*callee*-saved)	Nonpreserved (*caller*-saved)
Saved registers: `s0-s11`	Temporary registers: `t0-t6`
Return address: `ra`	Argument registers: `a0-a7`
Stack pointer: `sp`	
Stack above the stack pointer	Stack below the stack pointer

Callee Saved: obligaciones de la llamada
Caller Saved: obligaciones de la llamadora

Pseudoinstrucciones

Como su nombre lo dice, son una etiqueta que realidad es una composición de operaciones

`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`
`li s8, 0x7EF`	`addi s8, zero, 0x7EF`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	`jal L1`
`call L5`	`auipc ra, imm31:12` `jalr ra, ra, imm11:0`
`ret`	`jalr zero, ra, 0`

Las más importantes

- j: salta a un label sin guardar retorno.
- jal: salta a una direccion y guarda el retorno.
- ret: vuelve al lugar de donde fue llamado con un jal.

Fetch & Decode & Execute

Es el núcleo del ciclo de instrucciones de un CPU, asegurando que las instrucciones se recuperen de memoria, se interpreten y se ejecuten.

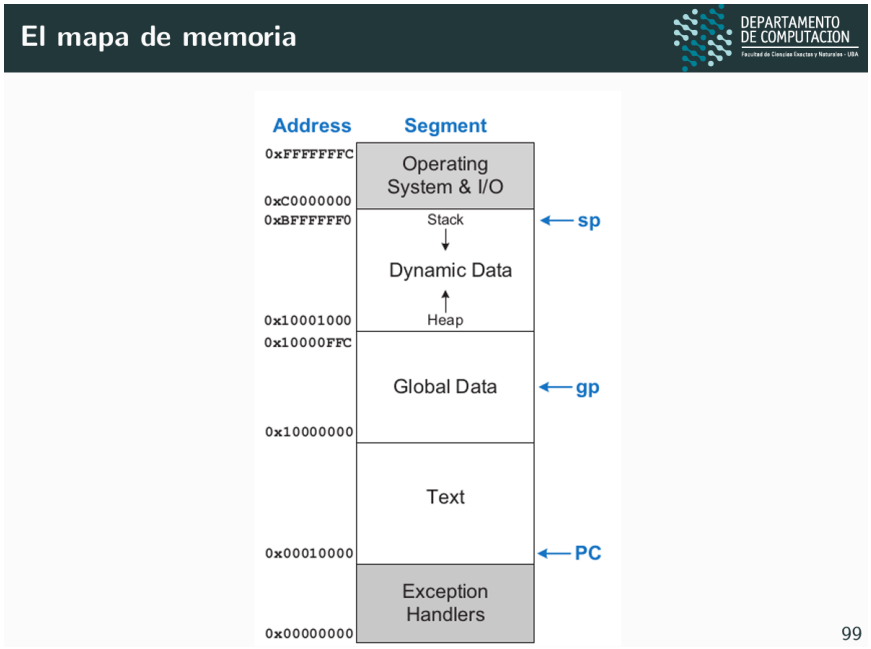
- Fetch: El CPU recupera la siguiente instrucción (sabe cuál es gracias al Program Counter) a ejecutar desde la memoria.
- Decode: El CPU interpreta la instrucción recuperada, detectando el opcode, operandos, operación a realizar, etc. Además, se generan las señales de control correspondientes para que la ejecución tenga la configuración necesaria.
- Execute: El CPU ejecuta la instrucción. Luego, se actualiza el PC.

Nota: Las instrucciones que se deben ejecutar a medida que el Program Counter avanza están almacenadas en la memoria RAM desde que el programa se ejecuta.
Referencias:

- ¿Cómo se interpretan las instrucciones de RISC-V?: TODO
- Proceso de Fetch, Decode y Execute desde un punto de vista de microarquitectura

Compilación, ensamblado y ejecución

El mapa de memoria



En la materia solo nos importa: Stack, Data y Text.

Directivas de ensamblado

En la materia vamos a usar, .data, .text, .word.

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^N-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

- .text: es el código de risc-v compilado cargado en la memoria RAM. Cada etiqueta, al ser compilado pasa a tener una posición en memoria de la instrucción donde comienza ese bloque.
- .data: es la data que vamos usar en nuestro código de risc-v, vive en la memoria RAM. Es posible utilizar etiquetas para esos valores, pero a la hora de usarlos en .text, si colocamos el nombre de la etiqueta en realidad hacemos referencia a la posición de memoria en la RAM donde está ese valor.

```
.data
    arr: .word 1, 2, 3, 4, 5 #inicializo array. esto se ensambla, y arr se guarda en una posicion de memoria de la ram.
    n: .word 5 #inicializo la longitud del array. para saber hasta donde poder iterar. esto se ensambla y n se guarda en una
        posicion de memoria de la ram.
.text
    la a0, arr #cargo en a0 la posicion de memoria donde esta el array (es decir, el puntero)
    lw a1, n #cargo el valor de n, osea a1 = 5
```

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
.setion .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.

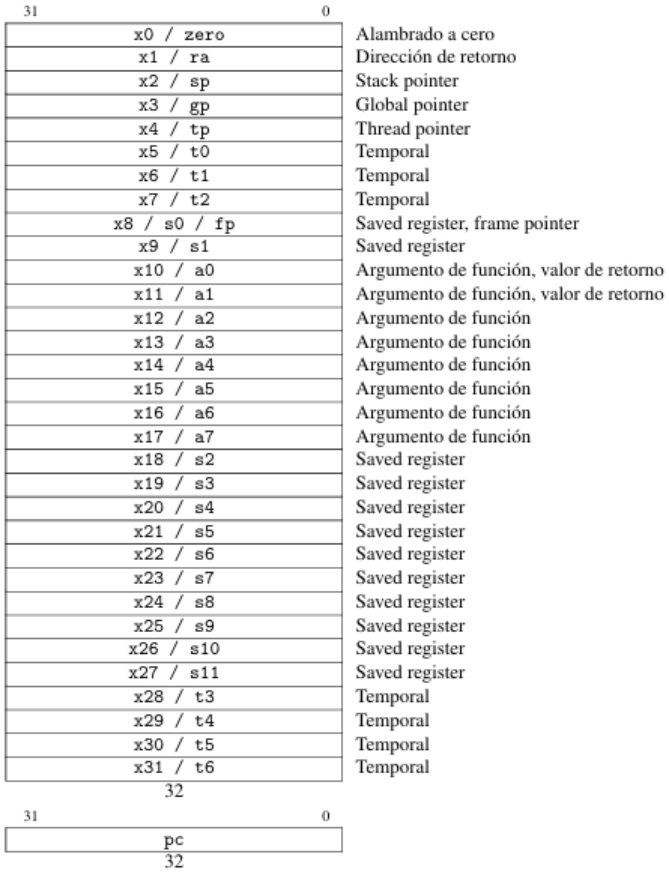
Programa final con inicialización de los datos en RISC V

```
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)

Registros RISC-V

Viven en el procesador.



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

Nota: utilizar la escritura del registro de la izquierda es lo mismo que los de la derecha. Pero queda más claro usar los de la derecha.
Para más información, véase Página 21 Guia Práctica de Risc V

Mapa de opcodes RISC-V

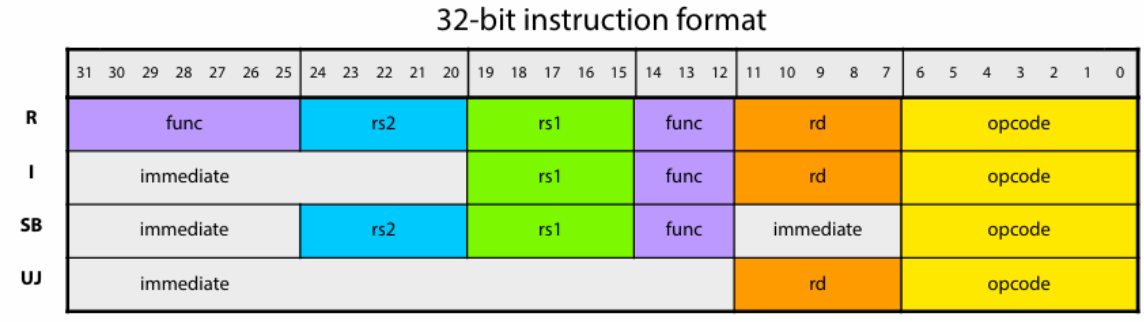
Página 18 de Guia Práctica de Risc V

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).

Formato de instrucciones en RISC-V

La siguiente imagen puede ser abrumadora sin ningún tipo de contexto, pero no es nada complicado.



Esta imagen es la que utiliza el procesador para hacer el proceso de **decode**, para poder entender qué es lo que realiza la instrucción almacenada en la RAM. Veamos como interpretar cada cosa. Iremos desde los campos que tienen en común.

- opcode: Es el código de operación. Nos dice qué familia (I, R, S, B, U, etc) de operaciones tiene la operación. Puede encontrar las distintas familias en la **la última columna** de la imagen de la sección anterior
    - R: Utilizan dos registros como operandos fuentes (rs1, rs2) y uno como operando destino rd. El campo op junto con funct7 y funct3 determinan el tipo de instrucción codificada.

- I: Utilizan registros como operando fuente (rs1), un inmediato de 12 bits (imm) y uno como operando destino rd. El campo op junto con funct3 determinan el tipo de instrucción codificada.
- S (instrucciones de carga): Codifica un inmediato de 12 bits en la instrucción. El desplazamiento (offset) siempre se desplaza una posición a izquierda antes de sumarlo al PC ya que se encuentra siempre en posiciones pares. Ej: sw.
- B (instrucciones de saltos condicionales): Codifica un inmediato de 13 bits en la instrucción. El desplazamiento (offset) siempre se desplaza una posición a izquierda antes de sumarlo al PC ya que se encuentra siempre en posiciones pares. Ej: beq
- U (instrucciones de inmediato superior): Codifica un inmediato de 20 bits en la instrucción. El desplazamiento (offset) siempre se desplaza una posición a izquierda antes de sumarlo al PC ya que se encuentra siempre en posiciones pares.
- J (instrucciones de saltos incondicionales): Codifica un inmediato de 21 bits en la instrucción. El desplazamiento (offset) siempre se desplaza una posición a izquierda antes de sumarlo al PC ya que se encuentra siempre en posiciones pares.

- rd (Read Destination): Es donde se debe almacenar el resultado, usualmente es un registro. Imaginemos que esos bits, en nuestra instrucción es 00101, pasamos el valor a decimal (5) y esto equivale al registro x5/t0. Puede más información en la sección de Registros
- func: Es el código de la función. Es importante que el código de la función también depende de la familia. Por lo tanto hay que ver familia + código de función. Puede encontrar las distintas funciones de familia en la **la 3ra columna** de la imagen de la sección anterior.
  Por fuera de la tabla dice Familia + Operación.
  Ej: 101 rd 0110011 es: I srai
- rs1 (Source Register 1): Es el primer operando que se utiliza en una operación dada. Imaginemos que esos bits, en nuestra instrucción es 01001, pasemos el valor a decimal (9) y esto equivale al registro x9/s1 (luego que sabe el registro, debe buscar el valor).
- rs2: Misma idea que rs1, es el segundo operando fuente.
- immediate: Se utiliza para operaciones que tienen inmediatos. Como por ejemplo: addi, srli, srai. Imaginemos que esos bits, en nuestra instrucción es 0000 0000 0111, pasemos el valor a decimal (7). Luego, el valor inmediato que se quiere sumar es 7.

Nota: no hay más que dos operandos fuente porque las operaciones de RISC-V se hacen entre operaciones atómicas, es decir, en las cuales ya con una única operación podemos deducir el valor de verdad.

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.
- <=: asignación no bloqueante (aka: asincrónica). 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.

```
module buffer3(input logic[3:0] a, input logic enable, output tri[3:0] y);
    assign y = en ? a : 4'bz;
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: $\{3\{d[0]\}\} \equiv d[0]d[0]d[0]$
- [] accede a una lista específica de bits. Tienen que ser números descendentes. Es decir a > b. Ej: c[a:b]

```
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.

```
case(data):
 0: segments = algo;
 1: segments = algo;
 ...
 default: segments = algo;
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 (hacia la memoria RAM). 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 ≡ 0000 0000 0000 1010 1011 1100 1101 1110
- 0xFFF ≡ 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

```
addi x1, x2, 3 -> 5
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.
Sí, la implementación es correcta. Carga en x3, el valor que tenga la posición #0x00000005

```
lw x1, 8(x2) -> guarda en x1, la posicion de memoria de x2 corrida 2 palabras (desplaza 8 bytes).
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.

```
addi t0, t0, 4
addi t1, t1, 5
sw t1, 0(t0)
```

¿Es correcta esta implementación? Sí, guarda el valor de t1(5) en la posición de memoria #0x00000004 hasta #0x00000007 inclusive en la RAM

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 ≡ 2 bytes)
Sé que como CD debo llevarlo a los más significativos, aplico un srli 8 bits a la derecha. Sé tambien 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 0x000000FF (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

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

Control de valores inmediatos mayores a 12 bits $2^{11}(12\ bits)$

- 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, extiendo 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

Ejercicios con Saltos Condicionales / Incondicionales

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

```
let destinatarios = 2;

while(destinatarios>0){
   destinatarios -= 1;
}
```

Equivale en RISC-V a:

```
#s2 = destinatarios
lw a0, s2
j loop
loop:
    addi a0, a0, -1
    beq a0, zero, fin
    j loop
fin:
```

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

```
let destinatarios = 2;

const hacerCero = (destinatarios: number): number => {
    if(destinatarios == 0) return 0;
    return hacerCero(destinatarios-1);
}

const respuesta: number = hacerCero(destinatarios);
```

Equivale en RISC-V a:

```
#s2 = destinatarios
lw a0, s2                           #0x0000 0000
jal ra, hacerCero                   #0x0000 0004
sw v0, x3                           #0x0000 0008
hacerCero:
    addi sp, sp, -4                 #0x0000 000C
    sw ra, 0(sp)                    #0x0000 0010
    beq a0, zero, finHacerCero      #0x0000 0014
    addi a0, a0, -1                 #0x0000 0018
    j hacerCero                     #0x0000 001C
finHacerCero:
    li v0, 0                        #0x0000 0020
    lw ra, 0(sp)                    #0x0000 0024
    addi sp, sp, 4                  #0x0000 0028
    jr ra                           #0x0000 002C
```

Nótese: que al usar recursión, reservamos siempre 4 bytes para guardar el ra por si llegamos a saltar a otra función en el beq.
Nótese: a0 se puede utilizar en hacerCero pues, a0 previo a una llamada a una función sería un argumento.

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 $\implies$ 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

```
    li a0, 4228 <- 0x0000 a 0x0008 (0 a 8) sin incluir
    li a1, 2114 <- 0x0008 a 0x0010 (8 a 16) sin incluir
    jal ra, resta <- 0x0010
    fin:
        beq zero, zero, fin <- 0x0014
    resta:
    prologo:
        addi sp, sp, -4 <- 0x0018
        sw ra, 0(sp) <- 0x001c
        sub a0, a0, a1 <- 0x0020
        beq a0, zero, epilogo <- 0x0024
    sigo:
        jal ra, resta <- 0x0028
    epilogo:
        lw ra, 0(sp) <- 0x002c
        addi sp, sp, 4 <- 0x0030
        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 posicion inicial porque ya no necesitamos hacer nada (nótese que es importante que subimos dejando todo como estaba pero nuestro ra que habiamos 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 descripta 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 finalzar la ejecución
  - ra: 0x0010 antes de saltar a resta, dentro del prologo, ra se guarda en la primera posicion 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.

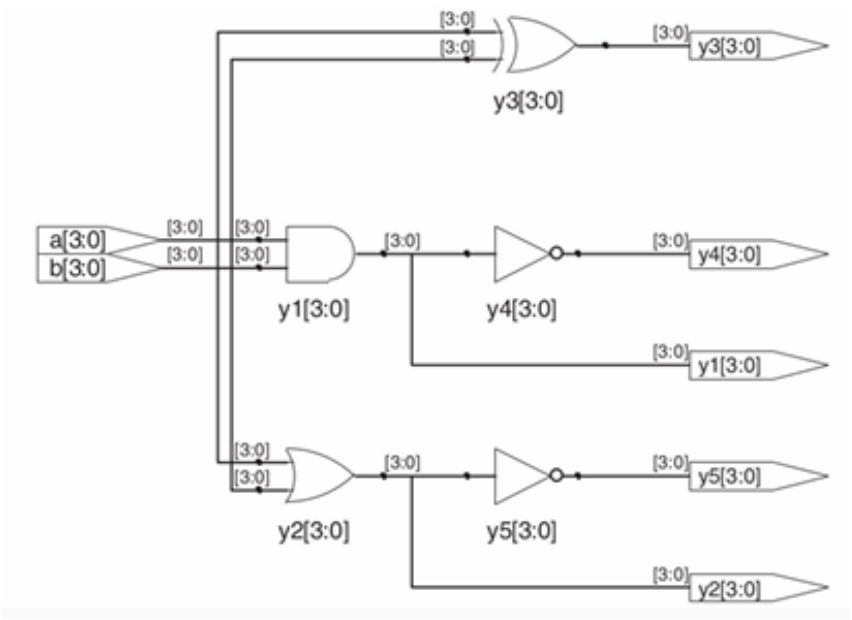
```
const a0:number = 4228;
const a1:number = 2114;

const resta = (a0: number, a1: number): number | boolean => {
    if(a0 == 0){
        return true; //epilogo
    }
    else{
        return resta(a0-a1, a1);  //sigo
```

```
        }
    }

    const res = resta(a0, a1);
    console.log(res); //true
```

System Verilog - Síntesis - Simulación

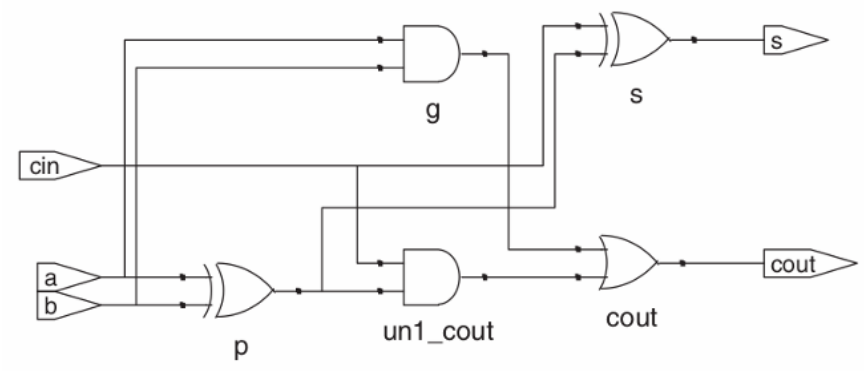
```
module compuertas(
    input logic [3:0] a, b,
    output logic [3:0] y1, y2, y3, y4, y5
    );

    assign y1 = a & b; //and
    assign y2 = a | b; //or
    assign y3 = a ^ b; //xor
    assign y4 = ~(a & b) (alt gr y simbolo de ~ para escribirlo) //nand
    assign y5 = ~(a | b); //nor
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)

```
module fulladder(input logic a, b, cin, output logic s, cout);
    logic p, g;
    assign p = a ^ b;
    assign g = a & b;
    assign s = p ^ cin;
    assign cout = p | (p & cin);
endmodule
```



Composición de multiplexores - Comportamiento Estructural

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

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

System Verilog - Asignación no bloqueante

```
module flop(input logic clk, input logic [3:0] d, output logic[3:0] q);
    always_ff @(posedge clk)
    q < = d;
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?

```
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: Gracias a la dirección proporcionada por el PC, busca la operación en la memoria RAM.
- Decode: Recibe la operación de la memoria RAM, y la interpreta en base al instruction format. La instruction memory, quien es encargado de este proceso, le pasa el control al siguiente.
- Execute: Una vez que se decidió que debe realizarse, ejecuta las operaciones correspondientes en el orden. Quien dicta el orden y prioridades es la control unit, y el CPU quien las ejecuta.
  - 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.

Anexo - Preguntas Teóricas para el parcial