TECNOLOGÍA DE COMPUTADORES

Transparencias de la Asignatura

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

ÍNDICE GENERAL

TECNOLOGÍA DE COMPUTADORES

Tema 0: Introducción a la Asignatura

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

TECNOLOGÍA DE COMPUTADORES

- Asignatura obligatoria
- Impartida en castellano
- Créditos: 6
- Profesores:
  Dr. Luis Alberto Aranda
  Dr. Iván Ramírez
  Dr. Francisco García
  D. Sergio Hernández
- Requisitos: ninguno

Universidad Rey Juan Carlos

CRITERIOS DE EVALUACIÓN

- Convocatoria ordinaria
  - Examen teórico parte 1 (≥ 5): 30%
  - Prácticas parte 1 (≥ 5): 20%
  - Examen teórico parte 2 (≥ 5): 30%
  - Prácticas parte 2 (≥ 5): 20%

  Aprobado ⟶ **Nota final ≥ 5**

- Convocatoria extraordinaria
  - Mismas condiciones que en ordinaria
  - Se guarda la nota de aquellas actividades aprobadas

Universidad Rey Juan Carlos

TEMARIO DE LA ASIGNATURA

1. Introducción a la asignatura
   - Fundamentos de los sistemas digitales
   - Sistemas de numeración: binario
   - Lenguajes de descripción hardware
2. Sistemas combinacionales
   - Álgebra de Boole. Mapas de Karnaugh
   - Análisis y síntesis de circuitos
3. Sistemas secuenciales
   - Elementos de memoria: biestables
   - Máquinas de estado y otros circuitos

BIBLIOGRAFÍA RECOMENDADA

- Hermida, R. (1998), Fundamentos de computadores.
- Floyd, T. L. (2006), Fundamentos de sistemas digitales.
- Cuesta, A. (2009), Problemas de fundamentos y estructura de computadores.
- Harris, D. M. y Harris, S. L. (2015), Digital design and computer architecture.
- Mealy, B. y Tappero, F. (2013), Free range VHDL

MOTIVACIÓN

- Los **sistemas digitales** están presentes en multitud de dispositivos actuales
- Gracias a los avances en electrónica digital se desarrollaron microprocesadores, memorias y otros componentes más básicos que estudiaremos durante el curso
- Se pueden construir circuitos más complejos partiendo de módulos básicos:
  - Circuitos que codifican y decodifican datos
  - Circuitos que transmiten y reciben información con buses de datos
  - Circuitos que procesan datos realizando operaciones lógicas y aritméticas
  - Circuitos que almacenan la información temporalmente
- Al **abstraernos** de las ecuaciones físicas, se reduce el esfuerzo de diseño

OBJETIVOS

Al finalizar la asignatura conoceréis:

- Los fundamentos de la electrónica digital
- Algunas herramientas CAD de diseño de sistemas digitales
- El lenguaje VHDL, utilizado para diseñar, simular e implementar circuitos digitales
- El funcionamiento de un computador, su arquitectura y sus unidades funcionales

TECNOLOGÍA DE COMPUTADORES

Tema 1: Introducción a los Computadores

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. Introducción a los computadores
2. Modelo Von Neumann de un computador
3. Terminología y parámetros característicos
4. Fundamentos de los sistemas digitales

CONTENIDOS

1. **Introducción a los computadores**
2. Modelo Von Neumann de un computador
3. Terminología y parámetros característicos
4. Fundamentos de los sistemas digitales

¿QUÉ ES UN COMPUTADOR?

- Una máquina que recibe una información de entrada, la procesa siguiendo unas instrucciones que almacena internamente, y produce una información de salida
  - Entrada y salida: el computador puede comunicarse con el exterior
  - Procesa información: el computador realiza cálculos para resolver un problema concreto
  - Instrucciones: permiten programar el computador para indicarle lo que debe hacer
  - Almacena: el computador almacena información en su interior, tanto las instrucciones que tiene que realizar como los datos que va procesando

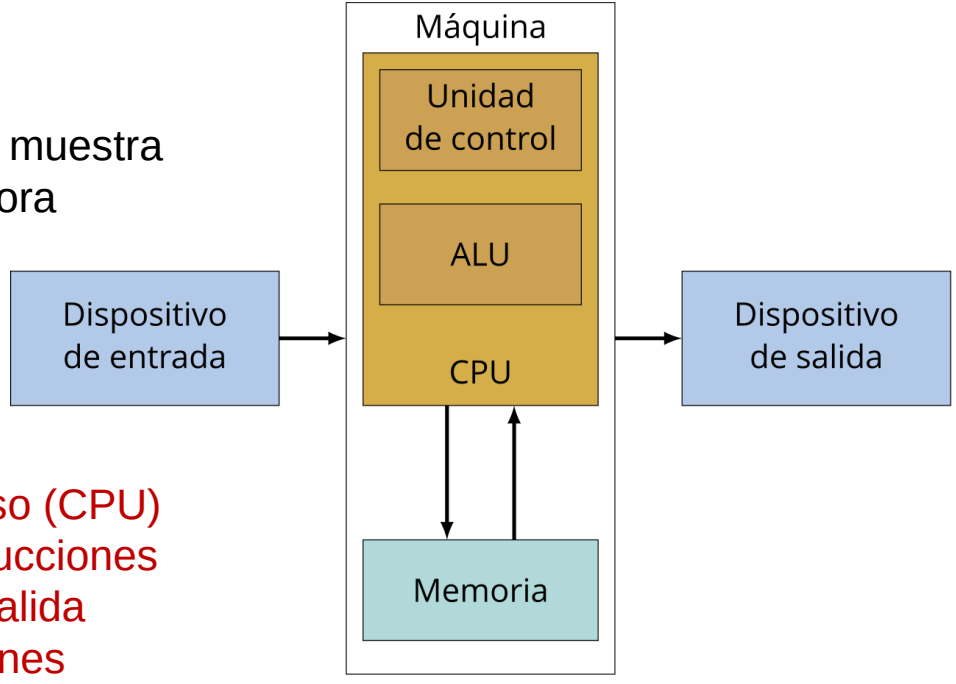
- Los computadores actuales se basan en la **Arquitectura Von Neumann**

CONTENIDOS

1. Introducción a los computadores
2. **Modelo Von Neumann de un computador**
3. Terminología y parámetros característicos
4. Fundamentos de los sistemas digitales

MODELO VON NEUMANN

- Es un modelo conceptual que muestra cómo funciona una computadora
- Explicado por John Von Neumann en 1945
- Formado por:
  - Unidad Central de Proceso (CPU)
  - Memoria de datos e instrucciones
  - Dispositivos de entrada/salida
  - Bus de datos e instrucciones



Fuente: https://commons.wikimedia.org/w/index.php?curid=62330690

MODELO VON NEUMANN

- La **CPU** contiene:
  - Una unidad de control encargada de buscar las instrucciones en la memoria, decodificarlas y ejecutarlas. Tiene un registro de instrucciones y un contador de programa
  - Una unidad aritmético lógica (**ALU**) para realizar las operaciones solicitadas

- **Desventaja**: dado que existe un único bus compartido entre datos e instrucciones, no puede realizarse una búsqueda de instrucciones y una operación de datos simultáneamente

- Esto se conoce como “cuello de botella Von Neumann” y se produce porque la velocidad de comunicación entre la CPU y la memoria es más baja que la velocidad a la que puede trabajar la CPU

CONTENIDOS

1. Introducción a los computadores
2. Modelo Von Neumann de un computador
3. **Terminología y parámetros característicos**
4. Fundamentos de los sistemas digitales

TERMINOLOGÍA

- **Bit**: unidad mínima de información. **Puede valer ‘0’ o ‘1’**
- **Byte**: conjunto de **8 bits**
- **Nibble**: conjunto de 4 bits
- **Palabra**: cadena de bits utilizada por algún elemento del computador. Suelen ser potencias de 2
- **Prefijos** más utilizados **en informática:**
  - Kilo: 2^{10}
  - Mega: 2^{20}
  - Giga: 2^{30}
  - Tera: 2^{40}

¡¡No confundir con los prefijos del sistema métrico decimal!!

PARÁMETROS CARACTERÍSTICOS

- **Capacidad** de almacenamiento **de una memoria**. Se mide en múltiplos de byte:
  - Memoria caché: desde KB a MB
  - Memoria principal (RAM): desde MB a GB
  - Memoria secundaria (pendrive o disco duro): desde GB a TB

- **Tiempo de acceso a memoria**: tiempo que tarda en realizarse una operación de memoria. Se mide en fracciones de segundo (ms, μs, ns, ps)

- **Frecuencia de trabajo del procesador**: ciclos por segundo del reloj del procesador. Se mide en múltiplos de Hz

PARÁMETROS CARACTERÍSTICOS

- **Tiempo de ejecución de un programa**: tiempo que tarda un programa desde su inicio hasta que finaliza su ejecución

- **Rendimiento de un computador**: es inverso al tiempo de ejecución. Se mide en tareas ejecutadas por unidad de tiempo

  Ej. MIPS: millones de instrucciones completadas por segundo

  Ej. MFLOPS: millones de instrucciones de coma flotante completadas por segundo

- **Benchmark**: programa de prueba utilizado para medir el rendimiento

  Ej. *Dhrystone* / *Whetstone* (para medir instrucciones enteras / flotantes), AnTuTu (para procesadores ARM), etc.

CONTENIDOS

1. Introducción a los computadores
2. Modelo Von Neumann de un computador
3. Terminología y parámetros característicos
4. **Fundamentos de los sistemas digitales**

ELECTRÓNICA ANALÓGICA Y DIGITAL

- **Electrónica analógica**: comprende aquellos circuitos con variables **continuas**, las cuales pueden tomar infinitos valores dentro de un rango.
- **Electrónica digital**: comprende aquellos circuitos con variables **discretas**. En este caso, las variables son binarias, pudiendo tomar únicamente valores de **‘0’ o ‘1’**
  - Circuito combinacional: el valor de las salidas del circuito únicamente depende del valor de las entradas del circuito
  - Circuito secuencial: el valor de las salidas del circuito depende del valor de las entradas actuales del circuito y de los valores que tuvieron previamente. Son circuitos que tienen memoria

FUNDAMENTOS DE SISTEMAS DIGITALES

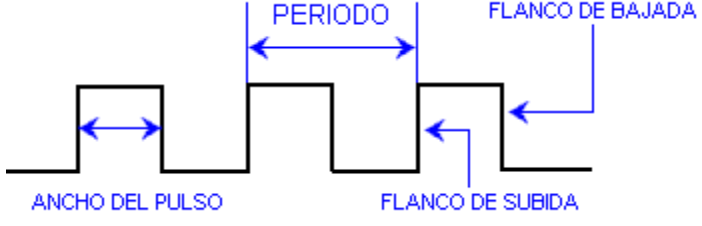
- Los computadores utilizan dos niveles de tensión que se traducen a ‘0’ o ‘1’

	TTL	CMOS	
$V_{H(máx.)}$	5 V	5 V	Nivel de tensión alto (High = ‘1’)
$V_{H(mín.)}$	2 V	3,5 V	
	Prohibido		Zona de incertidumbre
$V_{L(máx.)}$	0,8 V	1 V	Nivel de tensión bajo (Low = ‘0’)
$V_{L(mín.)}$	0 V	0 V	

FUNDAMENTOS DE SISTEMAS DIGITALES

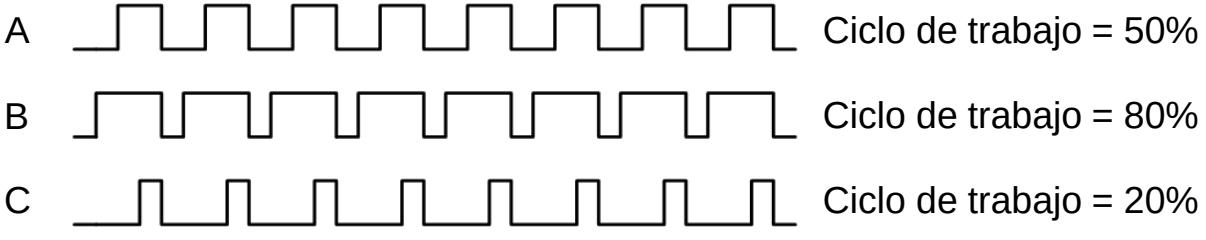
- Se puede representar el valor de una señal digital a lo largo del tiempo, esto se denomina **forma de onda de la señal digital**
- Las señales digitales pueden ser periódicas o no periódicas. Una señal de reloj varía de forma periódica y tiene duración infinita
- Si son periódicas se puede calcular su **ciclo de trabajo** como el cociente entre el ancho del pulso y el periodo

A



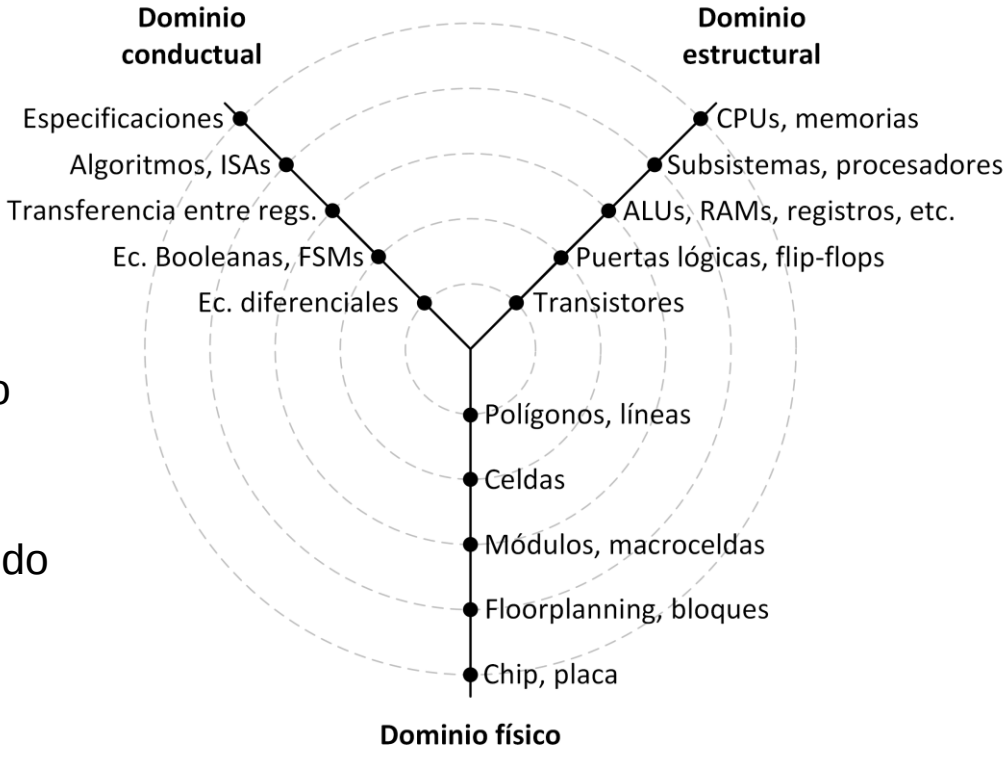
FUNDAMENTOS DE SISTEMAS DIGITALES

- Se puede representar el valor de una señal digital a lo largo del tiempo, esto se denomina **forma de onda de la señal digital**
- Las señales digitales pueden ser periódicas o no periódicas. Una señal de reloj varía de forma periódica y tiene duración infinita
- Si son periódicas se puede calcular su **ciclo de trabajo** como el cociente entre el ancho del pulso y el periodo



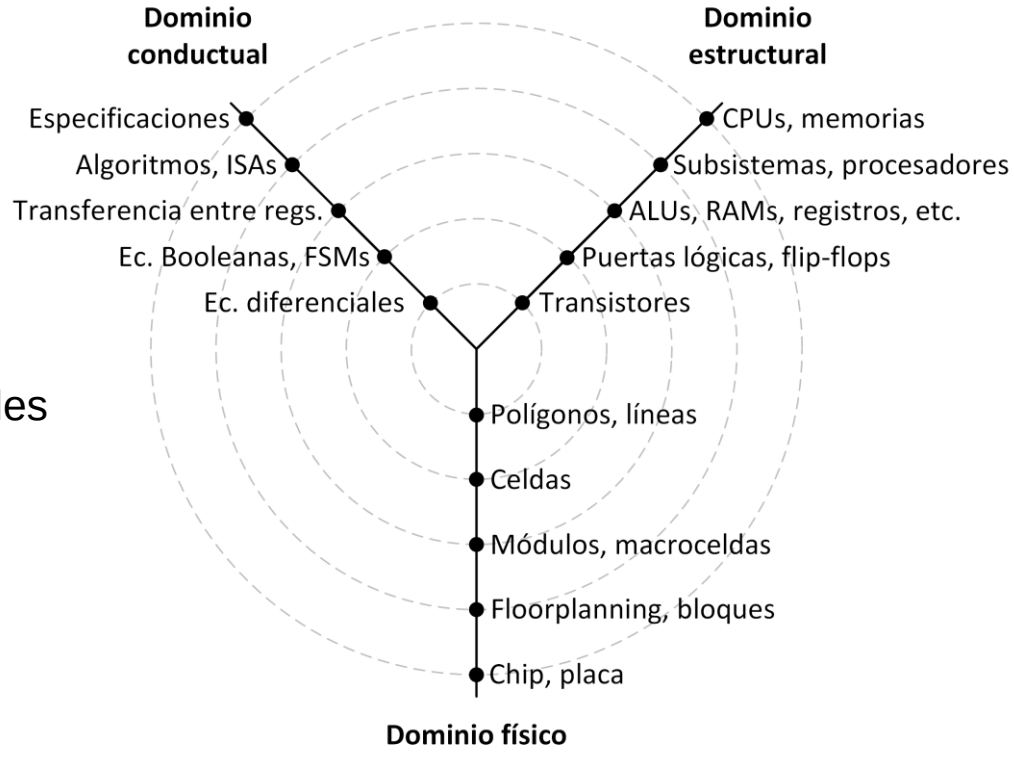
DOMINIOS Y NIVELES DE ABSTRACCIÓN

- Un sistema digital puede describirse desde diferentes dominios conceptuales:
  - Conductual: cómo se comporta
  - Estructural: qué bloques lo componen y cómo se interconectan
  - Físico: cómo está construido realmente



DOMINIOS Y NIVELES DE ABSTRACCIÓN

- Y con distintos niveles de abstracción:
  - Circuito: electrónica
  - Lógico: ‘0’ y ‘1’ lógicos
  - RT (transferencia entre registros): palabras, señales
  - Algorítmico: estructuras y dependencias
  - Sistema: protocolos de sincronización entre subsistemas



TECNOLOGÍA DE COMPUTADORES

Tema 2: Sistemas de Numeración

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. Sistemas de numeración. Conversión entre bases
2. Aritmética binaria sin signo
3. Aritmética binaria con signo
4. Otras codificaciones

CONTENIDOS

1. **Sistemas de numeración. Conversión entre bases**
2. Aritmética binaria sin signo
3. Aritmética binaria con signo
4. Otras codificaciones

SISTEMA DE NUMERACIÓN DECIMAL

- Los **dígitos del 0 al 9** representan una cantidad
- El dígito más a la derecha representa las unidades (peso 1)
- Si queremos una cantidad mayor, debemos añadir más dígitos a la izquierda (peso 10, 100, …, 10^n). Son potencia de 10 porque tenemos 10 dígitos (base 10)

Ej.

$$75438 = 7x10000 + 5x1000 + 4x100 + 3x10 + 8x1$$
$$= 7x10^4 + 5x10^3 + 4x10^2 + 3x10^1 + 8x10^0$$

$$0{,}3564 = 3 \times 0{,}1 + 5 \times 0{,}01 + 6 \times 0{,}001 + 4 \times 0{,}0001$$
$$= 3\,x10^{-1} + 5x10^{-2} + 6x10^{-3} + 4\,x10^{-4}$$

SISTEMA DE NUMERACIÓN BINARIO

- Los **dígitos del 0 al 1** representan una cantidad
- Los pesos en este caso son potencia de 2 porque tenemos 2 dígitos (base 2)

Ej.

$$1100_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 1 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1 = 12_{10}$$

Potencias de 2 positivas									Potencias de 2 negativas			
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
256	128	64	32	16	8	4	2	0	0,5	0,25	0,125	0,0625

SISTEMA DE NUMERACIÓN BINARIO

- En **decimal**, cuando llegamos al valor máximo del dígito (9) pasamos al 0 y sumamos 1 al siguiente dígito

  Ej. 19 → 20
  99 → 100

- En **binario**, cuando llegamos al valor máximo del dígito (1) pasamos al 0 y sumamos 1 al siguiente dígito

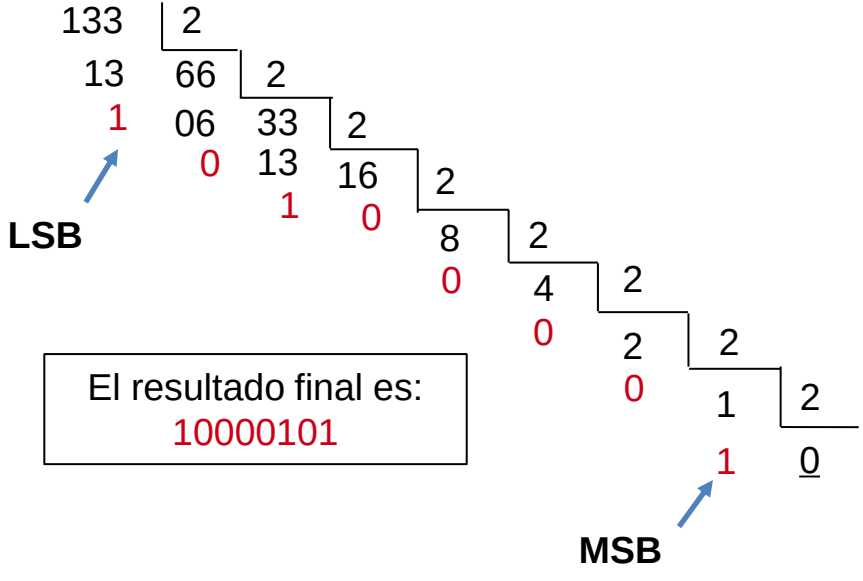
  Ej. 01 → 10
  011 → 100

Número decimal	Número binario
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

CONVERSIÓN DECIMAL A BINARIO

Para números enteros:

1. Se divide el número entre 2
2. Se repite el paso 1 con el cociente de la división anterior hasta que el cociente sea 0
3. Los restos de las divisiones conforman el número binario. El primer resto es el bit menos significativo (**LSB**) y el último el bit más significativo (**MSB**)



CONVERSIÓN DECIMAL A BINARIO

EJEMPLO

Convertir a binario los siguientes números enteros decimales:

- 12
- 45
- 58
- 82

CONVERSIÓN DECIMAL A BINARIO

Para números con parte fraccionaria:

1. Separamos el número en parte entera y parte fraccionaria
2. Se aplica el método visto anteriormente a la parte entera
3. La parte fraccionaria se multiplica por 2. La parte entera resultante conforma el número binario
4. Se repite el paso 3 hasta que la parte fraccionaria sea 0

Ej: 0,625

0,625 x 2 = 1,25 → 1 (**MSB**)
0,25 x 2 = 0,5 → 0
0,5 x 2 = 1,00 → 1 (**LSB**)

El resultado final es:
0,101

CONVERSIÓN DECIMAL A BINARIO

EJEMPLO

Convertir a binario los siguientes números fraccionarios decimales:

- 0,6
- 26,5
- 0,375

CONTENIDOS

1. Sistemas de numeración. Conversión entre bases
2. **Aritmética binaria sin signo**
3. Aritmética binaria con signo
4. Otras codificaciones

ARITMÉTICA BINARIA SIN SIGNO

- La **suma** binaria es similar a la suma decimal:

0 + 0 = 0 (acarreo 0)
0 + 1 = 1 (acarreo 0)
1 + 0 = 1 (acarreo 0)
1 + 1 = 0 (acarreo 1)

Decimal

acarreo → 1 1
74
+48
122

Binario

acarreo → 1 1
11
+01
100

3
+1
4

ARITMÉTICA BINARIA SIN SIGNO

EJEMPLO

Sumar los siguientes números binarios:

- 11 + 11
- 100 + 10
- 111 + 11
- 110 + 100
- 1111 + 1100

ARITMÉTICA BINARIA SIN SIGNO

- La **resta** binaria es similar a la resta decimal:

$$0 - 0 = 0$$
$$1 - 0 = 1$$
$$1 - 1 = 0$$
$$10 - 1 = 1$$

Decimal

acarreo → 1

$$\begin{array}{r} 74 \\ -\ 8 \\ \hline 66 \end{array}$$

Binario

acarreo → 1

$$\begin{array}{r} 10 \\ -\ 1 \\ \hline 01 \end{array} \qquad \begin{array}{r} 2 \\ -\ 1 \\ \hline 1 \end{array}$$

ARITMÉTICA BINARIA SIN SIGNO

EJEMPLO

Restar los siguientes números binarios:

- 11 – 01
- 11 – 10
- 111 – 100

ARITMÉTICA BINARIA SIN SIGNO

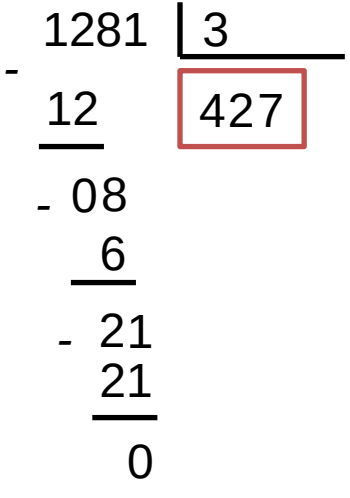
- La **multiplicación** binaria es similar a la decimal
- Primero se generan los productos parciales desplazando cada uno hacia la izquierda una posición y luego se suman verticalmente en binario

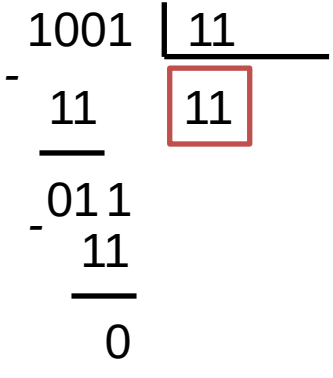
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

```
   0111    (7)
 x 1011    (11)
 ------
   0111
  0111
 0000
+ 0111
 -------
 1001101   (77)
```

ARITMÉTICA BINARIA SIN SIGNO

- La **división** binaria es también similar a la decimal
- Multiplico y resto de forma binaria





CONTENIDOS

1. Sistemas de numeración. Conversión entre bases
2. Aritmética binaria sin signo
3. **Aritmética binaria con signo**
4. Otras codificaciones

REPRESENTACIÓN SIGNO-MAGNITUD

- El número en binario está formado por un bit de signo (el MSB) y su magnitud
- Si el bit de signo = ‘0’ el número es positivo, si es ‘1’ entonces es negativo
- La magnitud está formada por el resto de bits del número

Ej.

011011 → +27

111011 → -27

Signo **Magnitud**

Ventajas:
- El rango de números es simétrico

Desventajas:
- Es más complejo operar aritméticamente
- El cero tiene doble representación (± 0)

REPRESENTACIÓN COMPLEMENTO A 1

- Los números positivos se representan como signo-magnitud
- Los número negativos se obtienen hallando el complemento a 1 del número positivo

  Ej.

  +27 → 011011

  ¿-27? → Invierto todos los bits del número → 100100

Ventajas:

- Las operaciones aritméticas son más sencillas de realizar
- El rango de números es simétrico
- **Ojo**: Los números positivos empiezan en ‘0’ y los negativos en ‘1’

Desventajas:

- El cero tiene doble representación (+0 = 0000 y -0 = 1111)
- Si sumo un número y su opuesto el resultado es -0

REPRESENTACIÓN COMPLEMENTO A 2

- Los números positivos se representan como signo-magnitud
- Los número negativos se obtienen hallando el complemento a 2 del número positivo

  Ej.

  +27 → 011011

  ¿-27? → Invierto todos los bits del número y sumo 1 → 100101

Ventajas:

- Las operaciones aritméticas son más sencillas de realizar
- **Ojo**: Los números positivos empiezan en '0' y los negativos en '1'
- El cero tiene una única representación
- Si sumo un número y su opuesto el resultado es 0

Desventajas:

- El rango de números no es simétrico $[-2^{N-1}, 2^{N-1} - 1]$

COMPLEMENTO A 2 A DECIMAL

Para convertir a decimal un número expresado en complemento a 2:

- Si el número es positivo se convierte normalmente como suma de pesos
- Si el número es negativo se convierte como suma de pesos salvo por el peso del bit de signo (MSB), que se resta

Ej.

$01010100 \rightarrow 0x2^7 + 1x2^6 + 0x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 0x2^0$

$= 0 + 64 + 0 + 16 + 0 + 4 + 0 + 0 = 84_{10}$

$10101100 \rightarrow 1x(-2^7) + 0x2^6 + 1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 0x2^0$

$= -128 + 0 + 32 + 0 + 8 + 4 + 0 + 0 = -84_{10}$

ARITMÉTICA BINARIA CON SIGNO

- La **suma** de números en complemento a 2 se hace igual que la sin signo
- Únicamente hay que tener cuidado con los **desbordamientos** (*overflow*)

Ej.

00000111 (+7)	00001111 (+15)	00010000 (+16)	11111011 (-5)
00000100 (+4)	11111010 (-6)	11101000 (-24)	11110111 (-9)
00001011 (+11)	1 00001001 (+9)	11111000 (-8)	1 11110010 (-14)

Overflow
(lo descartamos)

Overflow
(lo descartamos)

ARITMÉTICA BINARIA CON SIGNO

- ¿Y si hago 125 + 58?

```
  01111101  (+125)
  00111010  (+58)
  --------
  10110111  (+183)
```

CUIDADO: También se produce desbordamiento
(Entramos en el rango de números negativos
por lo que habría que añadir un 0 a la izquierda
o si no sería considerado como -73)

¿Cuál es el número positivo más grande que puedo representar con 8 bits?

ARITMÉTICA BINARIA CON SIGNO

- La **resta** se hace sumando al minuendo el complemento a 2 del sustraendo

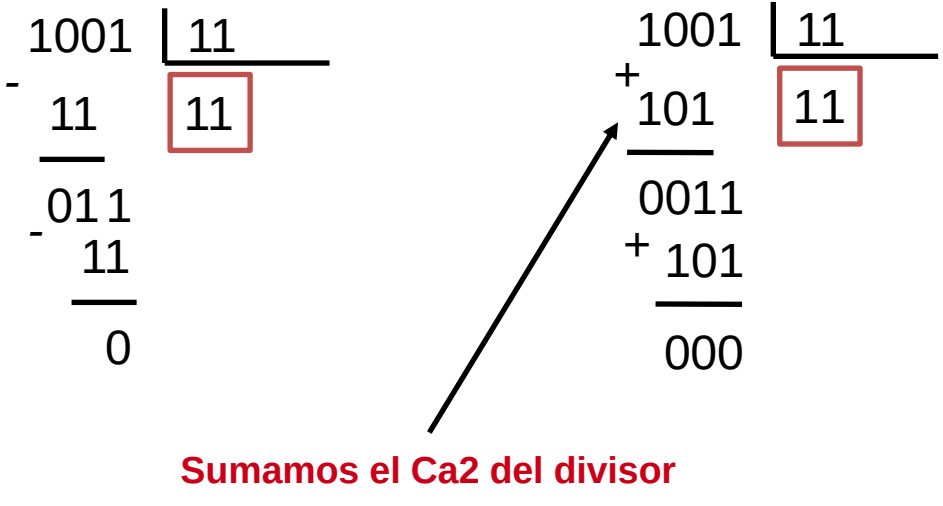
Ej. 16 – 24 = 16 + (-24) = 00010000 + 11101000 = 11111000 = -8

- La **multiplicación** contempla varios casos:
  - Si ambos números son positivos → se añade un ‘0’ en el MSB del resultado
  - Si ambos números son negativos → se hace el complemento a 2 de ambos números, se multiplican y se añade un ‘0’ en el MSB del resultado
  - Si uno es negativo y el otro positivo → se hace el complemento a 2 del negativo, se realiza la multiplicación y al resultado se le hace el complemento a 2 y se le añade un ‘1’ en el MSB

ARITMÉTICA BINARIA CON SIGNO

- La **división** se realiza sumando el complemento a 2 del divisor (en vez de restar)

Ej.



CONTENIDOS

1. Sistemas de numeración. Conversión entre bases
2. Aritmética binaria sin signo
3. Aritmética binaria con signo
4. **Otras codificaciones**

SISTEMA DE NUMERACIÓN HEXADECIMAL

- Tenemos 16 caracteres para representar una cantidad: números 0-9 y letras A-F

Decimal	0	1	2	3	4	5	6	7
Binario	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal	0	1	2	3	4	5	6	7

...

8	9	10	11	12	13	14	15
1000	1001	1010	1011	1100	1101	1110	1111
8	9	A	B	C	D	E	F

Cada carácter hexadecimal se corresponde con un número binario de 4 bits

SISTEMA DE NUMERACIÓN HEXADECIMAL

- Los pesos en este caso son potencia de 16 (base 16) por lo que:
  Ej.
  $F1A_{16} = F \times 16^2 + 1 \times 16^1 + A \times 16^0 = 15 \times 16^2 + 1 \times 16^1 + 10 \times 16^0 = 3866_{10}$

- Para pasar de hexadecimal a binario convertimos cada dígito a binario
  Ej.
  $10A4_{16} \rightarrow 1 \quad 0 \quad 10 \quad 4 \rightarrow 0001 \quad 0000 \quad 1010 \quad 0100$

- Para pasar de binario a hexadecimal tomamos los bits de 4 en 4 y convertimos
  Ej.
  $11111100011001_2 \rightarrow 11 \quad 1111 \quad 0001 \quad 1001 \rightarrow 3 \quad 15 \quad 1 \quad 9 \rightarrow 3F19_{16}$

SISTEMA DE NUMERACIÓN OCTAL

- Tenemos 8 dígitos (0-7) para representar una cantidad

Decimal	0	1	2	3	4	5	6	7
Binario	000	001	010	011	100	101	110	111
Octal	0	1	2	3	4	5	6	7

Cada carácter octal se corresponde con un número binario de 3 bits

- Los pesos en este caso son potencia de 8 (base 8) por lo que:

  Ej.

  $174_8 = 1 \times 8^2 + 7 \times 8^1 + 4 \times 8^0 = 124_{10}$

SISTEMA DE NUMERACIÓN OCTAL

- Para pasar de octal a binario convertimos cada dígito a binario
  Ej.
  $107_8 \rightarrow$ 001 000 111

- Para pasar de binario a octal tomamos los bits de 3 en 3 y convertimos
  Ej.
  $11111100011001_2 \rightarrow$ 11 111 100 011 001 $\rightarrow$ 3 7 4 3 1 $\rightarrow 37431_8$

TECNOLOGÍA DE COMPUTADORES

Tema 3: Introducción a los Lenguajes de Descripción de Hardware

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. Introducción a los dispositivos hardware. FPGA vs. ASIC
2. Flujo de diseño hardware. Herramientas para el diseño hardware
3. Introducción a los lenguajes de descripción hardware. VHDL

CONTENIDOS

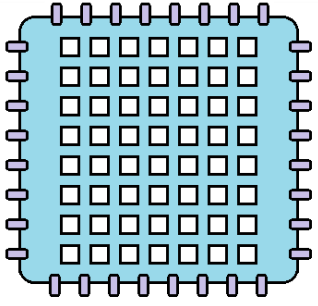
1. **Introducción a los dispositivos hardware. FPGA vs. ASIC**
2. Flujo de diseño hardware. Herramientas para el diseño hardware
3. Introducción a los lenguajes de descripción hardware. VHDL

INTRODUCCIÓN A LOS DISPOSITIVOS HW

- Un dispositivo hardware es un componente físico que forma parte del computador
- Comprende desde el teclado o el ratón, hasta la memoria RAM o la CPU
- En esta asignatura aprenderemos:
  - A diseñar circuitos que forman parte de estos dispositivos hardware
  - A usar las herramientas que permiten crearlos y verificarlos
- Para ello tendremos que aprender un lenguaje de descripción hardware junto con los fundamentos de la electrónica digital

FPGA vs. ASIC

- Una FPGA (*Field Programmable Gate Array*) es un **dispositivo semiconductor reprogramable tras su fabricación**
- Un ASIC (*Application Specific Integrated Circuit*) es un **dispositivo semiconductor fabricado para un propósito concreto**



FPGA

ASIC

FPGA vs. ASIC

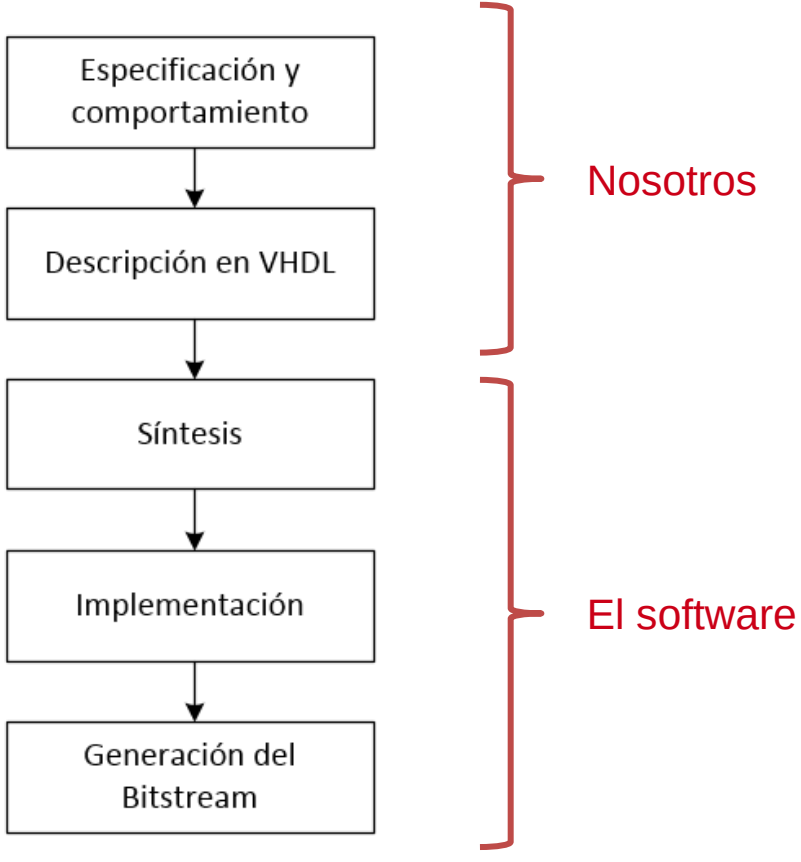
	FPGA	ASIC
Reprogramable	Sí	No
Fabricación	Rápida / Sencilla	Lenta / Compleja
Coste unitario	Alto	Bajo
Consumo de energía	Alto	Bajo
Rendimiento	Medio	Alto
Utilización	Prototipado	Aplicaciones concretas

CONTENIDOS

1. Introducción a los dispositivos hardware. FPGA vs. ASIC
2. **Flujo de diseño hardware. Herramientas para el diseño hardware**
3. Introducción a los lenguajes de descripción hardware. VHDL

FLUJO DE DISEÑO HARDWARE

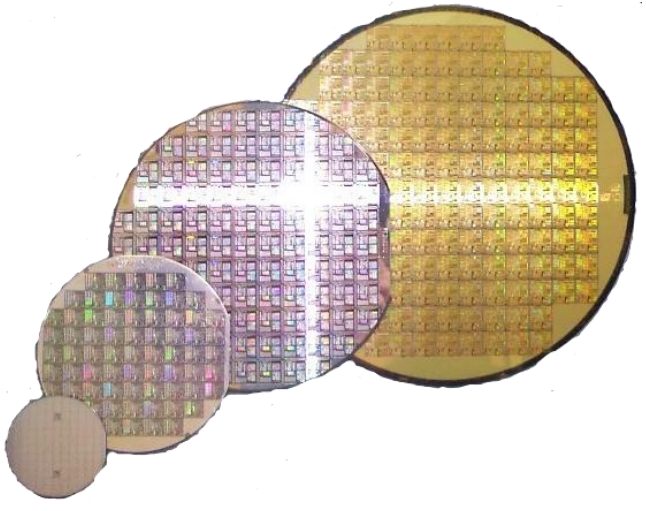
Una vez verificado el correcto funcionamiento del diseño, se procedería a su fabricación o implementación en FPGA



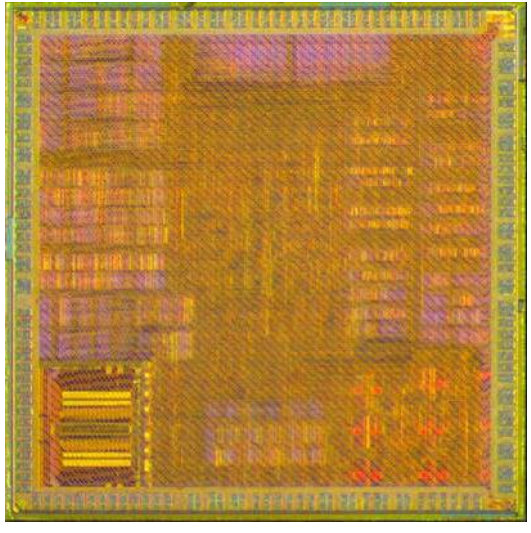
FLUJO DE DISEÑO HARDWARE

1. Especificación y comportamiento del circuito: descripción de su funcionalidad a alto nivel, los módulos que lo forman y sus interacciones
2. Descripción hardware: escritura del código VHDL/Verilog que describe el circuito
3. Síntesis: conversión del circuito a nivel de puertas lógicas
4. Implementación: conversión de esas puertas lógicas a recursos físicos hardware y sus interconexiones
5. Generación del bitstream: creación de un fichero que contiene una cadena de ceros y unos con la información de la implementación del circuito

FLUJO DE DISEÑO HARDWARE



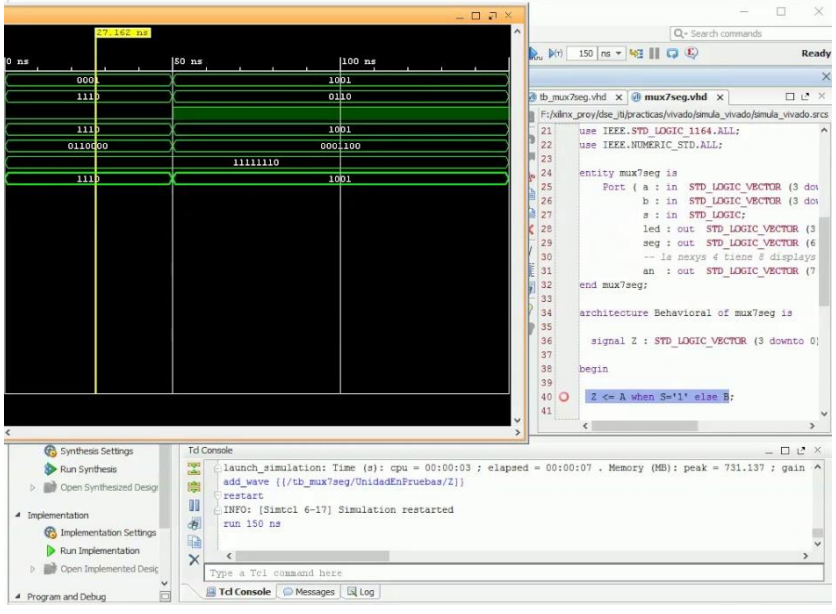
Oblea de silicio (fuente: wikipedia)
https://www.youtube.com/watch?v=UvluuAIiA50



Espectrómetro ASIC 65nm
(vista de microscopio)

HERRAMIENTAS DE DISEÑO HARDWARE

- **Software de diseño de circuitos**: facilitan la creación y verificación de diseños físicos mediante la automatización de algunas tareas



Vivado

HERRAMIENTAS DE DISEÑO HARDWARE

- **Esquemáticos**: diagramas de cajas y flechas que representan la **estructura** del sistema. Pueden incluir información sobre tiempos, señales, etc.
- **Grafos y diagramas de flujo**: permiten describir el sistema desde el punto de vista funcional o de **comportamiento**
- **Lenguajes de descripción hardware**: permiten describir un circuito digital desde diferentes niveles de abstracción. **Ej**: VHDL, Verilog

CONTENIDOS

1. Introducción a los dispositivos hardware. FPGA vs. ASIC
2. Flujo de diseño hardware. Herramientas para el diseño hardware
3. **Introducción a los lenguajes de descripción hardware. VHDL**

INTRODUCCIÓN A VHDL

- **Lenguaje de descripción hardware** definido por el **IEEE** (*Institute of Electrical and Electronics Engineers*) en los años 80
- **VHDL**: Contracción de las siglas **V**HSIC + **HDL**
  - VHSIC: *Very High-Speed Integrated Circuit*
  - HDL: *Hardware Description Language*

¿Por qué es necesario?

INTRODUCCIÓN A VHDL

- Describir las conexiones de circuitos digitales complejos a nivel de puerta lógica o de transistor puede ser inabordable o muy tedioso
- VHDL nos permite definir un nivel de abstracción superior:
  - Describimos las entradas/salidas del circuito y su comportamiento
  - Una herramienta software genera las puertas lógicas y conexiones por nosotros
  - Podemos reutilizar circuitos ya descritos, facilitando la modularidad
- **Cualquier circuito digital puede ser descrito en** pocas líneas de código utilizando **lenguaje VHDL**

OTROS LENGUAJES. SYSTEMVERILOG

- SystemVerilog es otro lenguaje ampliamente usado. ¿Por qué VHDL?

	VHDL	SystemVerilog
Desarrollado por	Un comité (IEEE)	Una empresa
Basado en	Lenguaje Ada	Lenguaje C
Enfoque	Top-down	Bottom-up
Tipado	Fuertemente	Medianamente
Librerías	Permitidas	No permitidas
Aprendizaje	Difícil	Fácil

Sabiendo un lenguaje se puede aprender el otro sin demasiado esfuerzo

REGLAS DE ORO DE VHDL

- **Regla 1: NO ES UN LENGUAJE DE PROGRAMACIÓN**
- **Regla 2**: Describe un hardware que se ejecuta **en paralelo**
- **Regla 3**: Es necesario conocer previamente el circuito a diseñar
- **Regla 4**: Todo hardware descrito en VHDL se puede implementar en una FPGA o se puede usar para fabricar un ASIC

CARACTERÍSTICAS DE VHDL

- Fin de línea: las instrucciones acaban con punto y coma **;**
- Sensibilidad: VHDL no es sensible ni a mayúsculas ni a espacios en blanco

```
S1 <= A and B;        =        s1        <=    a AnD      B;
```

- Comentarios: empiezan con dos guiones

```
-- Esto es un comentario en VHDL
```

- Paréntesis: se ponen por claridad, las operaciones tienen prioridad establecida

CARACTERÍSTICAS DE VHDL

- Palabras reservadas en VHDL

```
access    after     alias     all       attribute   block
body      buffer    bus       constant  exit        file
for       function  generic   group     in          is
label     loop      mod       new       next        null
of        on        open      out       range       rem
return    signal    shared    then      to          type
until     use       variable  wait      while       with
```

REPRESENTACIÓN DE CARACTERES

- Caracteres: se definen entre comillas simples ‘A’ o ‘a’, ‘0’ o ‘1’
- Cadenas de caracteres: se definen con comillas dobles “Una cadena”, “1010”
- Cadenas de bits: se escribe la base seguida de la cadena de caracteres
  - **Binario (B)**: B“1010”
  - **Hexadecimal (X)**: X“FA0”
  - **Octal (O)**: O“372”
  - **Decimal (D)**: D“23”

¡Ojo! Si se omite la base se considera que es una cadena en binario

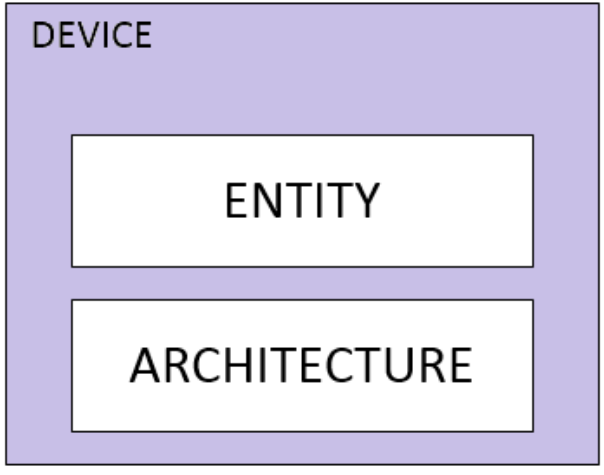
TIPOS STD_LOGIC Y STD_LOGIC_VECTOR

- Son un estándar industrial y los utilizaremos **SIEMPRE** para definir puertos
- STD_LOGIC: valor presente en un cable de 1 bit
- STD_LOGIC_VECTOR: para definir buses de datos (vector de bits)

Valores que pueden tomar:

- **'0'**: Nivel lógico bajo
- **'1'**: Nivel lógico alto
- **'U'**: Valor no definido, debido a que la señal no está inicializada
- **'X'**: Valor desconocido, debido a un **cortocircuito**
- **'Z'**: Alta impedancia, utilizado para "desconectar" señales

ESTRUCTURA DEL CÓDIGO VHDL



- Los diseños hardware en VHDL constan de:
  - Librerías: se añaden en el encabezado para incluir otros circuitos, variables, etc.
  - Entidad: describe nuestro circuito como una caja negra con sus puertos de entrada/salida
  - Arquitectura: describe el contenido de la caja negra y su comportamiento

TECNOLOGÍA DE COMPUTADORES

Tema 4: Álgebra de Boole

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. El álgebra de Boole
2. Propiedades y teoremas del álgebra de Boole
3. Puertas lógicas NOT, AND, OR y XOR
4. Implementación de puertas lógicas en VHDL

CONTENIDOS

1. **El álgebra de Boole**
2. Propiedades y teoremas del álgebra de Boole
3. Puertas lógicas NOT, AND, OR y XOR
4. Implementación de puertas lógicas en VHDL

EL ÁLGEBRA DE BOOLE

- Reglas algebraicas basadas en lógica matemática y teoría de conjuntos
- Denominado así en honor a **George Boole**, quien lo introdujo en 1854
- Claude E. Shannon lo aplicó a circuitos digitales en 1948
- Se puede utilizar como herramienta para el diseño (crear circuitos) y análisis de circuitos digitales (cómo funcionan)

EL ÁLGEBRA DE BOOLE

- Se usan variables simbólicas (A, B, etc.) para representar las señales digitales
- Estas variables únicamente pueden tomar dos valores ‘0’ (**LOW**) y ‘1’ (**HIGH**)
- Como sabemos, LOW y HIGH hacen referencia al voltaje de la señal
- Si, por ejemplo, se definen LOW = 0 V y HIGH = 5 V, entonces la señal conmuta entre esos dos valores de tensión y no puede tomar ningún otro

OPERACIONES DEL ÁLGEBRA DE BOOLE

- En el álgebra de Boole sólo están definidas tres operaciones lógicas:
  - Suma lógica (**+**): denominada también operación **OR**
  - Producto lógico (**·**): denominado también operación **AND**
  - Negación lógica (A' o Ā): denominada también operación **NOT**

A	B	A + B	A · B
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

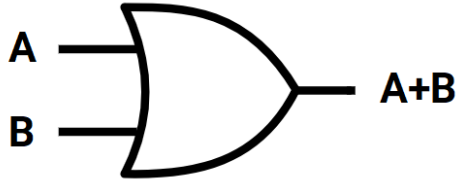
A	A'
0	1
1	0

- La **suma = 1** cuando alguna señal vale 1
- El **producto = 1** cuando ambas señales valen 1

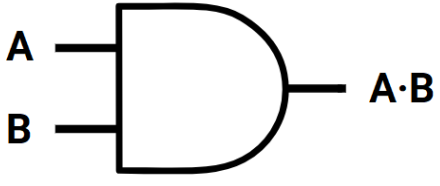
PUERTAS LÓGICAS BÁSICAS

- Estas tres operaciones se representan en circuitos digitales con puertas lógicas:

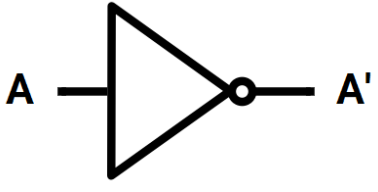
Puerta OR



Puerta AND



Puerta NOT



CONTENIDOS

1. El álgebra de Boole
2. **Propiedades y teoremas del álgebra de Boole**
3. Puertas lógicas NOT, AND, OR y XOR
4. Implementación de puertas lógicas en VHDL

PROPIEDADES DEL ÁLGEBRA DE BOOLE

- Las operaciones de suma y producto lógico cumplen las siguientes propiedades:
  - Propiedad conmutativa:

    $A + B = B + A$

    $A \cdot B = B \cdot A$
  - Propiedad distributiva:

    $A \cdot (B + C) = A \cdot B + A \cdot C$

    $A + B \cdot C = (A + B) \cdot (A + C)$
  - Elementos neutros:

    $A + 0 = A$

    $A \cdot 1 = A$

TEOREMAS DEL ÁLGEBRA DE BOOLE

- Teorema 1 (Abstracción digital):

  si A = 0 entonces A ≠ 1
  si A = 1 entonces A ≠ 0

- Teorema 2 (Identidad):

  $A + 1 = 1$
  $A \cdot 0 = 0$

- Teorema 3 (Idempotencia):

  $A + A = A$
  $A \cdot A = A$

- Teorema 4 (Involución):

  $(A')' = A$

- Teorema 5 (Complemento):

  $A + A' = 1$
  $A \cdot A' = 0$

- Teorema 6 (Absorción):

  $A + A \cdot B = A$
  $A \cdot (A+B) = A$

TEOREMAS DEL ÁLGEBRA DE BOOLE

- Teorema 7 (Asociatividad):

$$A + (B+C) = (A+B) + C$$
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

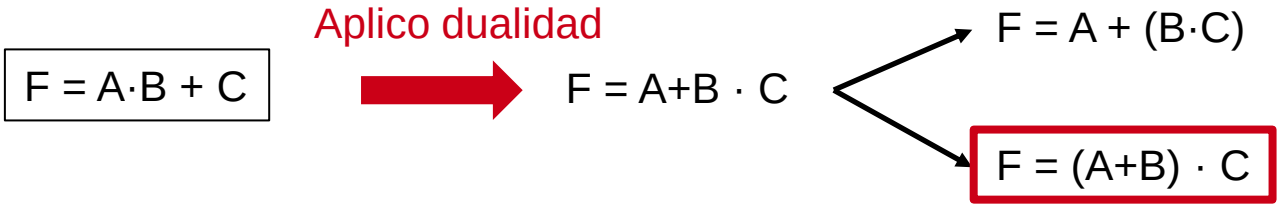
- Teorema 8 (**Leyes de De Morgan**):

$$(A+B)' = A' \cdot B'$$
$$(A \cdot B)' = A' + B'$$

- Principio de Dualidad: cualquier teorema o igualdad del álgebra de Boole sigue siendo cierto si se intercambian los ceros y los unos, y los operadores suma (+) y producto (·)

EJEMPLO: PRINCIPIO DE DUALIDAD

- Aplicar el principio de dualidad a la función F = A·B + C

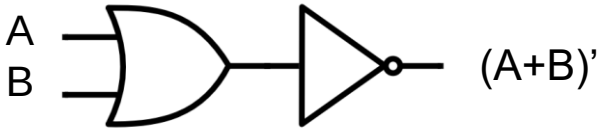


Prevalece el orden de las operaciones de la función inicial

EJEMPLO: LEYES DE DE MORGAN

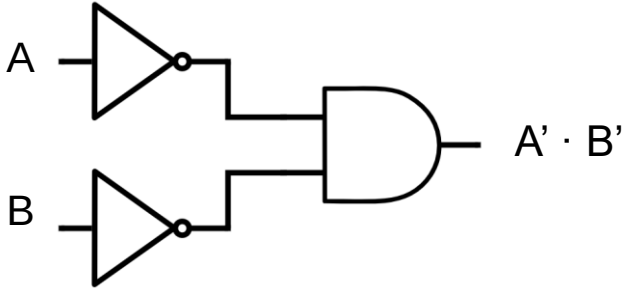
- Demostrar la primera ecuación de las Leyes de De Morgan: (A+B)' = A' · B'

Lado izquierdo de la igualdad:



A	B	A+B	(A+B)'	A'	B'	A'·B'
0	0	0	**1**	1	1	**1**
0	1	1	**0**	1	0	**0**
1	0	1	**0**	0	1	**0**
1	1	1	**0**	0	0	**0**

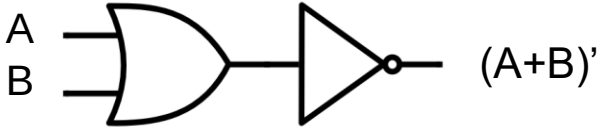
Lado derecho de la igualdad:

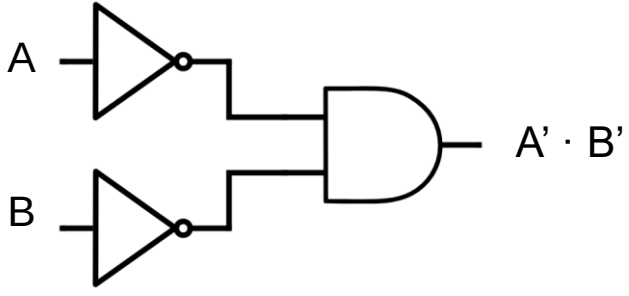


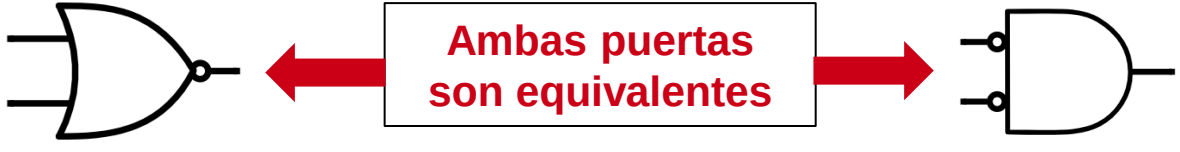
Ambas columnas coinciden

EJEMPLO: LEYES DE DE MORGAN

- Entonces, se puede concluir que:







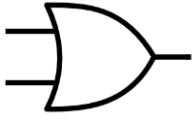
CONTENIDOS

1. El álgebra de Boole
2. Propiedades y teoremas del álgebra de Boole
3. **Puertas lógicas NOT, AND, OR y XOR**
4. Implementación de puertas lógicas en VHDL

PUERTAS LÓGICAS

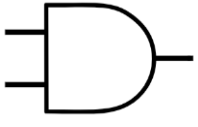
Puerta OR

A	B	A + B
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**



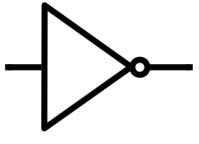
Puerta AND

A	B	A · B
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**



Puerta NOT

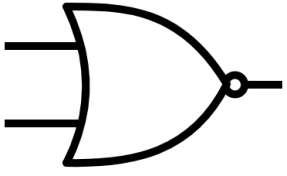
A	A’
0	**1**
1	**0**



PUERTAS LÓGICAS

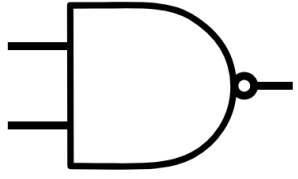
Puerta NOR

A	B	(A + B)’
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**



Puerta NAND

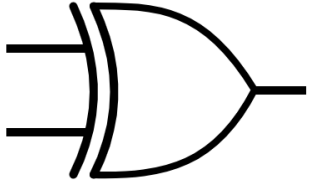
A	B	(A · B)’
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**



PUERTAS LÓGICAS

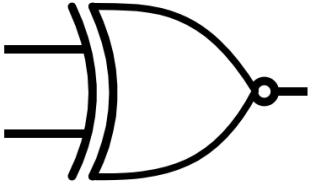
Puerta XOR

A	B	A ⊕ B
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**



Puerta XNOR

A	B	(A ⊕ B)’
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

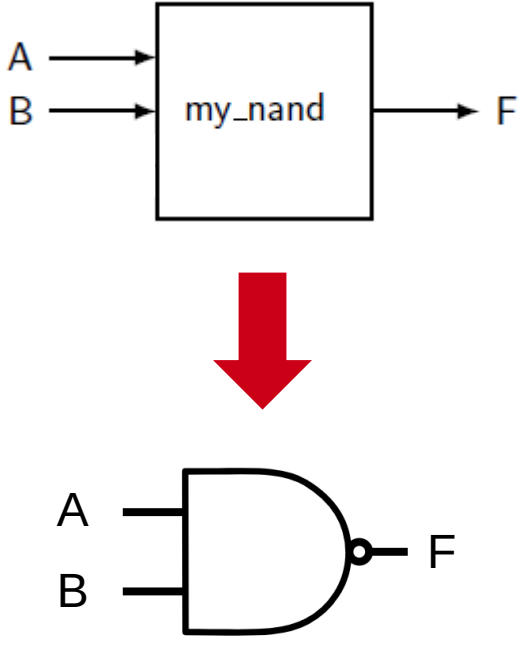


CONTENIDOS

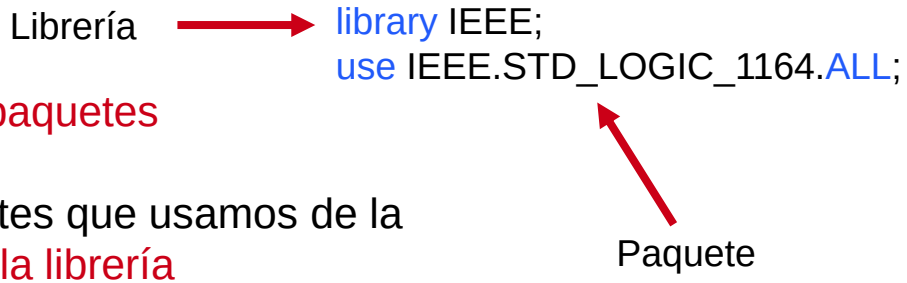
1. El álgebra de Boole
2. Propiedades y teoremas del álgebra de Boole
3. Puertas lógicas NOT, AND, OR y XOR
4. **Implementación de puertas lógicas en VHDL**

PUERTAS LÓGICAS EN VHDL

```
-- Library Declaration
library IEEE;
use IEEE.std_logic_1164.all;

-- Entity
entity my_nand is
        port ( A, B : in   std_logic;
                    F     : out std_logic
        );
end my_nand;

-- Architecture
architecture behavior of my_nand is
begin
        F <= not (A and B);
end behavior;
```



ESTRUCTURA DEL CÓDIGO VHDL: LIBRERÍAS



- Una librería es un conjunto de paquetes
- Debemos especificar los paquetes que usamos de la librería, no vale con incluir sólo la librería
- Los paquetes más comunes de la librería IEEE son:
  - STD_LOGIC_1164: Estándar para definir señales y operaciones lógicas
  - NUMERIC_STD: Para definir números *signed* y *unsigned* y operaciones aritméticas
  - STD_LOGIC_TEXTIO: Para trabajar con ficheros y cadenas de caracteres

Podemos crear nuestros propios paquetes y librerías

ESTRUCTURA DEL CÓDIGO VHDL: ENTITY

- La entidad describe cómo se conecta el componente a otros diseños
- Define el componente como “caja negra”
- Una entidad puede contener los siguientes campos:
  - **Ports**: son los puertos de entrada/salida que permiten que el componente se conecte con otros
  - **Generics**: permite definir parámetros del componente de modo que no sea necesario repetir su definición una y otra vez
  - **Constants**: permite definir constantes que se utilizarán en la arquitectura del componente

ENTITY: EJEMPLO

```
entity ejemplo is
        port ( in1  : in   std_logic;
               in2  : in   std_logic;
               out1: out std_logic
        );
        generic ( gen1 : integer := 4;
                  gen2 : time := 10 ns
        );
        constant : cnst1 : real := 1000.0;
end ejemplo;
```

2 puertos de entrada y uno de salida

Los *generic* son como los #define en C

Puedo definirla aquí o en la arquitectura

Una entidad puede contener los 3 campos… ¡o ninguno!

ENTITY: TIPOS DE PUERTOS

- **In:** define un puerto de entrada de datos. El valor de la señal:
  - Puede ser leído dentro de la entidad
  - No puede ser modificado dentro de la entidad

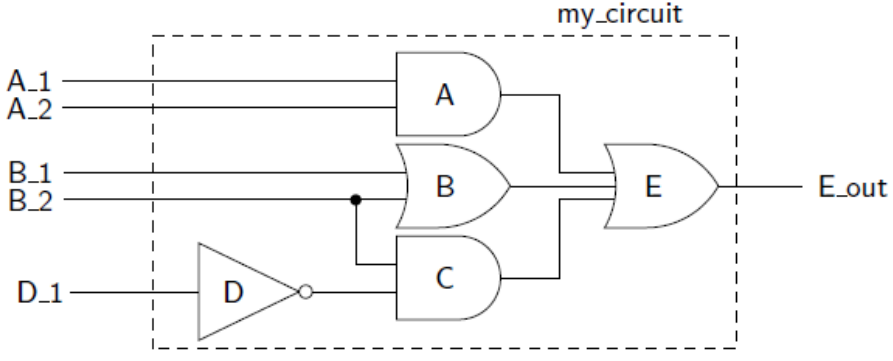
- **Out:** define un puerto de salida de datos. El valor de la señal:
  - No poder ser leído dentro de la entidad
  - Puede ser actualizado dentro de la entidad que lo define

- **Inout:** define un puerto que puede ser entrada y salida de datos. El valor de la señal:
  - Puede ser leído dentro de la entidad
  - Puede ser actualizado dentro de la entidad que lo define

- **Buffer:** define un puerto como el tipo inout pero:
  - Sólo se puede conectar a otra señal de tipo buffer
  - No puede tener más de un driver (más de una entidad que modifique su valor)

ESTRUCTURA DEL CÓDIGO VHDL: ARCHITECTURE

- La arquitectura describe el comportamiento del componente
- En la arquitectura se ejecutan las operaciones en paralelo



La entidad define la “caja negra” y la arquitectura el “contenido”

ARCHITECTURE: EJEMPLO

```
-- Architecture
architecture behavior of my_circuit is
        signal A_out, B_out, C_out : std_logic;
begin
        A_out <= A_1 and A_2;
        B_out <= B_1 or B_2;
        C_out <= (not D_1) and B_2;
        E_out <= A_out or B_out or C_out;
end behavior;
```

- Consta de dos partes:
  - Antes del *begin*: declaración de señales y variables
  - Después del *begin*: descripción del funcionamiento
- El operador <= nos permite conseguir el paralelismo/concurrencia de los circuitos digitales

¿Por qué no hemos declarado A_1, A_2, B_1, B_2, D_1 ni E_out?

TECNOLOGÍA DE COMPUTADORES

Tema 5: Especificación y Síntesis de Circuitos Combinacionales

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. Representación de funciones lógicas. Tablas de verdad y formas canónicas
2. Minimización de circuitos digitales. Mapas de Karnaugh
3. Conjuntos universales de puertas. Circuitos con NAND y NOR

CONTENIDOS

1. **Representación de funciones lógicas. Tablas de verdad y formas canónicas**
2. Minimización de circuitos digitales. Mapas de Karnaugh
3. Conjuntos universales de puertas. Circuitos con NAND y NOR

REPRESENTACIÓN DE FUNCIONES LÓGICAS

- Un circuito digital se puede expresar mediante:
  - Un esquemático
  - Un código escrito en un lenguaje de descripción de hardware
  - Una tabla de verdad
  - Una función lógica / booleana / de conmutación
  - Un mapa de Karnaugh
- Todas estas formas son equivalentes y podemos pasar de una a otra
- En este tema aprenderemos los conceptos para realizar estas conversiones y así hacer uso indistinto de todas las representaciones

TABLA DE VERDAD

- Se utiliza para representar el valor de una función o expresión booleana para cada combinación de entradas
- Proviene de la lógica Verdadero / Falso
- En cada celda de la tabla va un único valor de ‘1’ / ‘0’
- Las columnas de la izquierda (A, B y C) representan los valores de las señales de entrada del circuito
- La columna de la derecha (F) recoge los valores de la salida del circuito, que es función de las entradas

A	B	C	F(A,B,C)
0	0	0	F(0,0,0)
0	0	1	F(0,0,1)
0	1	0	F(0,1,0)
0	1	1	F(0,1,1)
1	0	0	F(1,0,0)
1	0	1	F(1,0,1)
1	1	0	F(1,1,0)
1	1	1	F(1,1,1)

DEFINICIONES

- Literal: una variable (A) o su complemento (A')
- Término producto: operación AND de literales
- Término suma: operación OR de literales
- Suma de productos: suma de términos producto
- Producto de sumas: un producto de términos suma
- Mintérmino: término producto en el que aparecen todas las variables de la función una vez
- Maxtérmino: termino suma en el que aparecen todas las variables de la función una vez
- **Suma canónica**: expresión booleana compuesta por la suma de mintérminos
- **Producto canónico**: expresión booleana compuesta por el producto de maxtérminos

MINTÉRMINOS Y MAXTÉRMINOS

- Un mintérmino se representa como **m_i** y vale ‘1’ para una única combinación
- Un maxtérmino se representa como **M_i** y vale ‘0’ para una única combinación
- El subíndice “i” indica el número en decimal de la combinación correspondiente

Ej.

Si tenemos 3 variables x_2, x_1, x_0

→ m_5 y M_5 por ej. se corresponden con $x_2 = 1; x_1 = 0; x_0 = 1$ ($101_2 = 5_{10}$)

→ Un mintérmino, por ej. m_5, se puede expresar como $m_5 = x_2 \cdot \bar{x}_1 \cdot x_0$

→ Un maxtérmino, por ej. M_5, se puede expresar como $M_5 = \bar{x}_2 + x_1 + \bar{x}_0$

FORMAS CANÓNICAS: EJEMPLO

- Determinar las dos formas canónicas de la siguiente tabla de verdad:

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- Suma canónica (suma de productos):

$$F(A,B,C) = m_0 + m_2 + m_3 + m_7 = \sum_{A,B,C}(0,2,3,7)$$

$$F(A,B,C) = \bar{A}\cdot\bar{B}\cdot\bar{C} + \bar{A}\cdot B\cdot\bar{C} + \bar{A}\cdot B\cdot C + A\cdot B\cdot C$$

FORMAS CANÓNICAS: EJEMPLO

- Determinar las dos formas canónicas de la siguiente tabla de verdad:

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- Producto canónico (producto de sumas):

$$F(A,B,C) = M_1 \cdot M_4 \cdot M_5 \cdot M_6 = \prod_{A,B,C}(1,4,5,6)$$

$$F(A,B,C) = (A + B + \bar{C}) \cdot (\bar{A} + B + C) \cdot (\bar{A} + B + \bar{C}) \cdot (\bar{A} + \bar{B} + C)$$

SÍNTESIS Y ANÁLISIS DE CIRCUITOS

- **Síntesis**: consiste en representar un circuito a partir de una descripción o texto

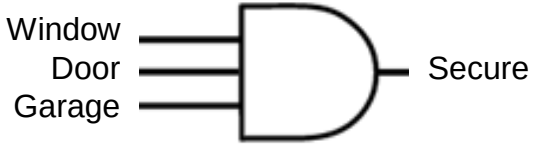
- **Análisis**: consiste en obtener la descripción formal de un circuito a partir de su esquemático o su función lógica

EJEMPLO: SÍNTESIS DE CIRCUITOS

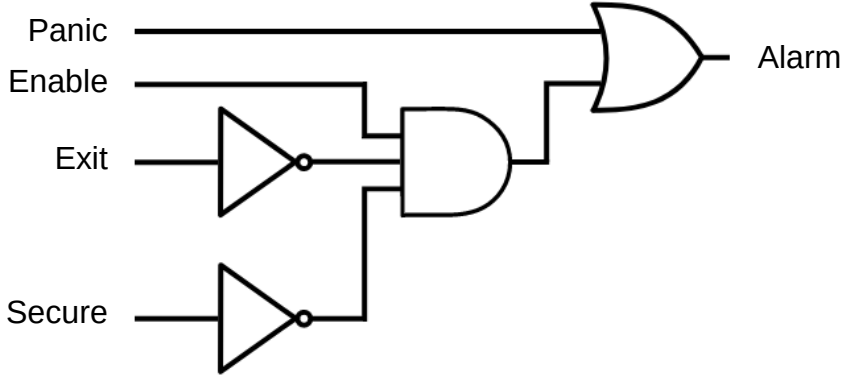
- Implementar un circuito de alarma con el siguiente comportamiento:
  - Alarm = 1 si Panic = 1 o si (Enable = 1, Exit = 0 y Secure = 0)
  - Secure = 1 si Window, Door y Garage son 1
- Por tanto tenemos las siguientes variables o literales:
  - Alarm, Panic, Enable, Exit, Secure, Window, Door y Garage
- El valor de Alarm depende de Panic, Enable, Exit y Secure
- El valor de Secure depende de Window, Door y Garage

EJEMPLO: SÍNTESIS DE CIRCUITOS

- Secure = 1 si Window, Door y Garage son 1:

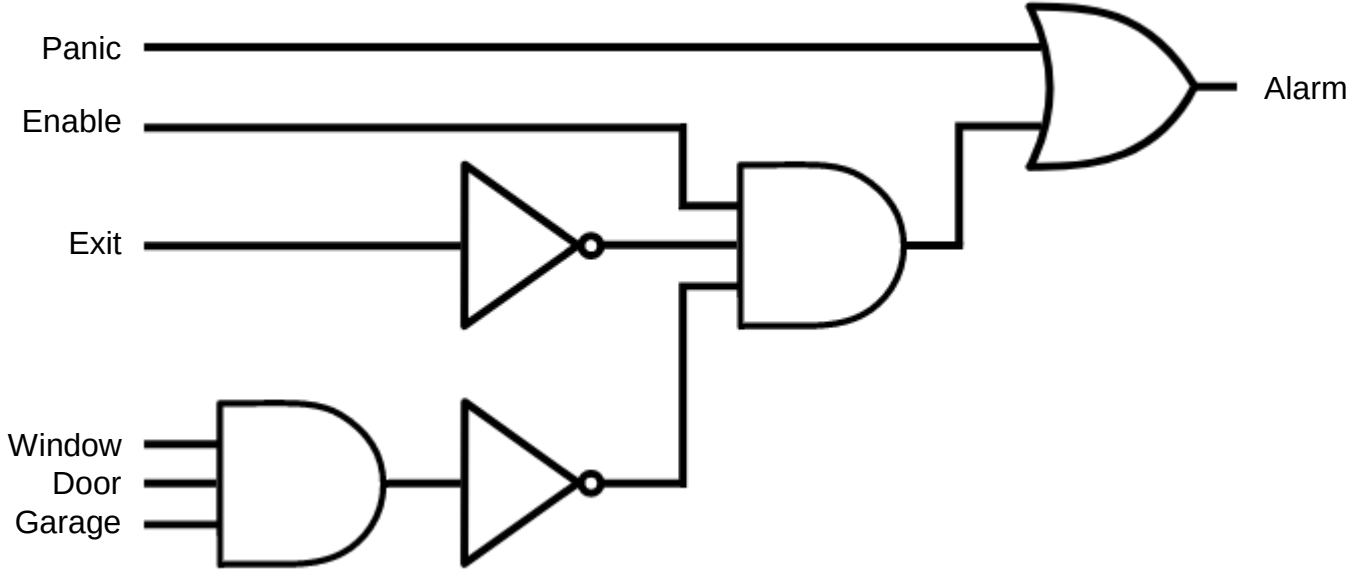


- Alarm = 1 si Panic = 1 o si (Enable = 1, Exit = 0 y Secure = 0):



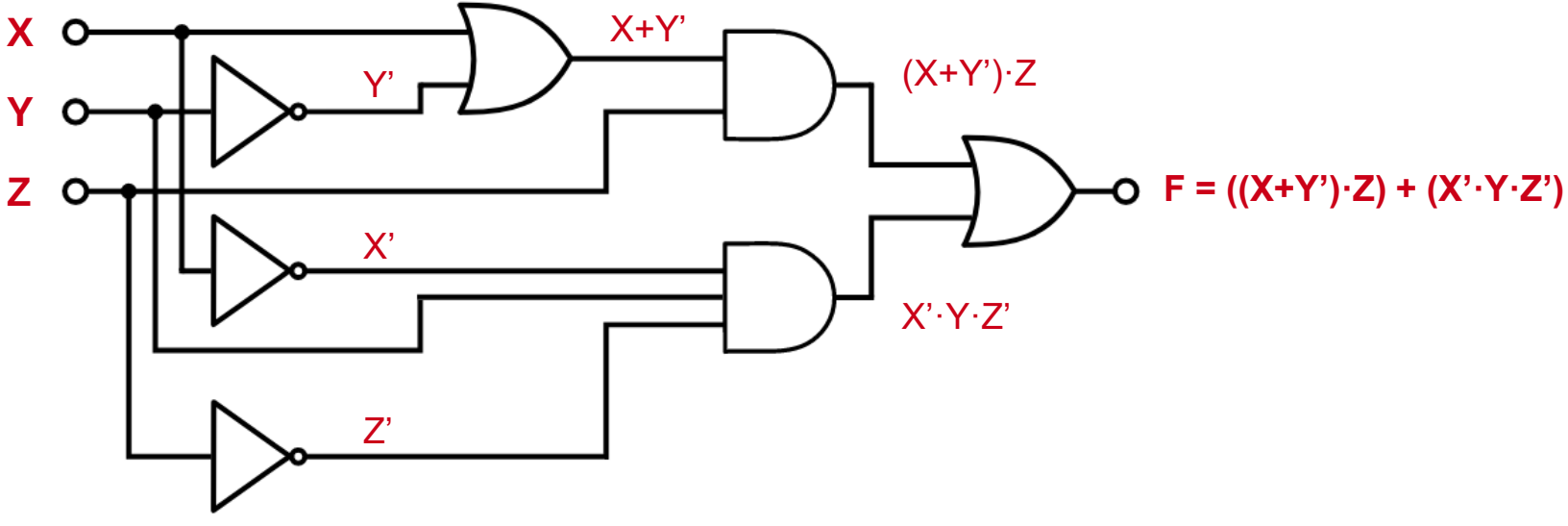
EJEMPLO: SÍNTESIS DE CIRCUITOS

- Juntando ambos circuitos obtenemos el circuito resultante:



EJEMPLO: ANÁLISIS DE CIRCUITOS

- Obtener la tabla de verdad del siguiente circuito combinacional:



EJEMPLO: ANÁLISIS DE CIRCUITOS

- Tenemos la siguiente función:

$$F(X,Y,Z) = ((X + Y') \cdot Z) + (X' \cdot Y \cdot Z')$$

- Damos valores:

X = 0; Y = 0; Z = 0

$$F(X,Y,Z) = \big((0 + 1) \cdot 0\big) + (1 \cdot 0 \cdot 1) = \ 0$$

Y así con el resto…

X	Y	Z	F(X,Y,Z)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

CONTENIDOS

1. Representación de funciones lógicas. Tablas de verdad y formas canónicas
2. **Minimización de circuitos digitales. Mapas de Karnaugh**
3. Conjuntos universales de puertas. Circuitos con NAND y NOR

MINIMIZACIÓN DE CIRCUITOS DIGITALES

- Existe una relación directa entre la expresión lógica y el circuito final
- A mayor número de variables y operaciones, mayor número de puertas lógicas
- Utilizando las propiedades del Álgebra de Boole, podemos convertir una expresión lógica en otra equivalente:

Ej.

$$F = (X \cdot Y \cdot Z) + (X \cdot Y' \cdot Z) = (X \cdot Z) \cdot (Y + Y') = X \cdot Z$$

Propiedad asociativa

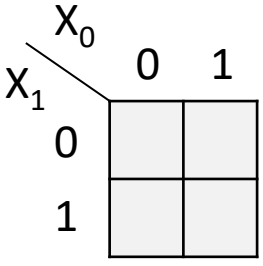
Teorema 5 (Complemento)

¿Hay alguna alternativa mejor?

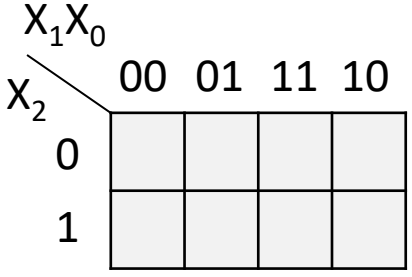
MAPAS DE KARNAUGH

- Representación gráfica de una función lógica usada para minimizar circuitos
- Se representa la tabla de verdad en forma de conjunto bidimensional de celdas, lo que permite realizar simplificaciones según la posición ocupada
- Muy útil para minimizar a mano funciones pequeñas de hasta **5 variables**
- Para más de 5 variables es recomendable utilizar software especializado

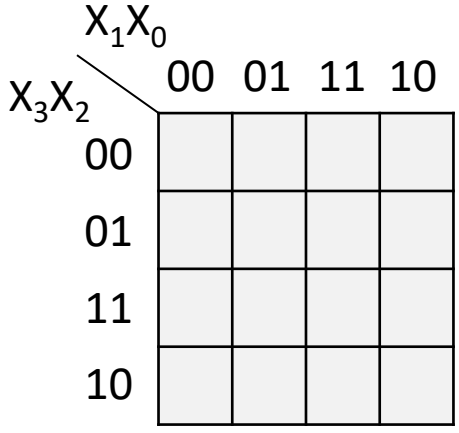
MAPAS DE KARNAUGH



2 variables

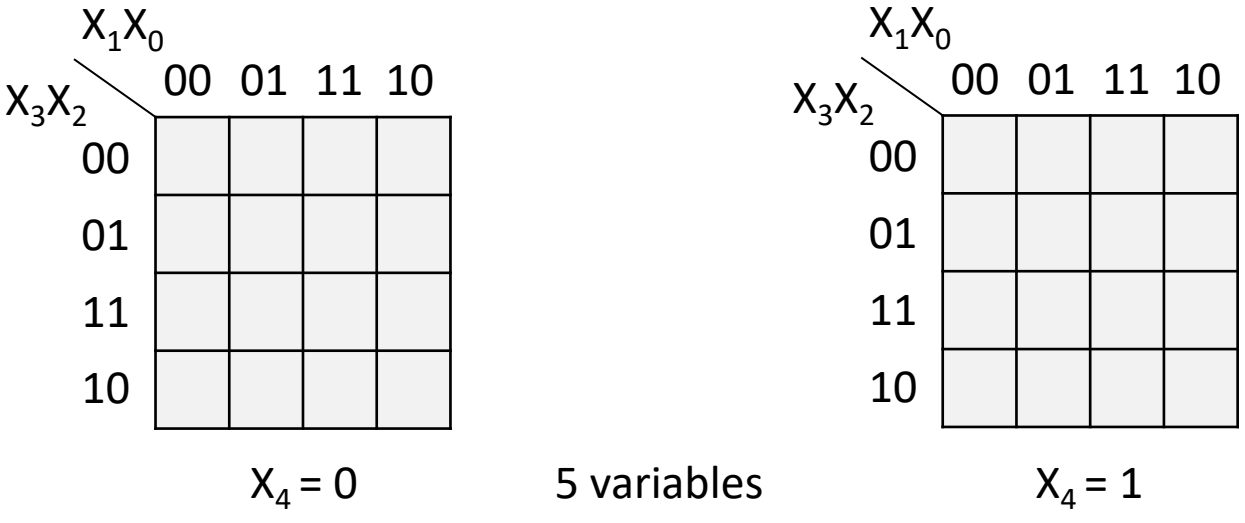


3 variables



4 variables

MAPAS DE KARNAUGH



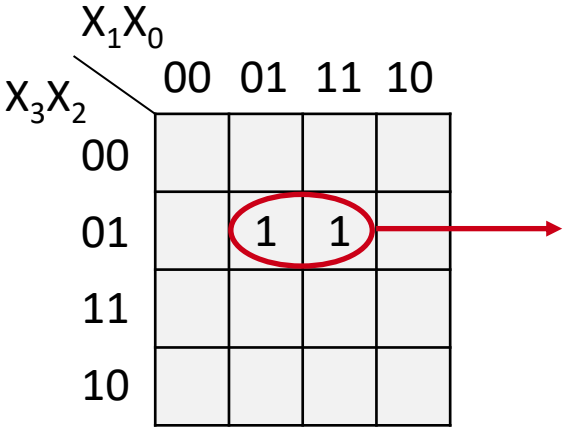
¡Tenemos un mapa tridimensional con dos niveles!

MAPAS DE KARNAUGH

- Se dice que dos celdas del mapa de Karnaugh son adyacentes cuando entre ellas sólo cambia el valor de una de las variables

- Para una función lógica cualquiera, rellenamos el mapa de Karnaugh con los valores de su tabla de verdad y después agrupamos los unos siguiendo las siguientes **reglas**:
  - Hay que crear grupos lo más grandes posible
  - Hay que crear el menor número de grupos posible
  - El número de unos en el interior de un grupo debe ser potencia de 2
  - Los unos pueden formar parte de varios grupos
  - Se pueden agrupar unos que estén en los extremos adyacentes del mapa

MAPAS DE KARNAUGH: EJEMPLO 1



$$m_5 + m_7 = \bar{x}_3 \cdot x_2 \cdot \bar{x}_1 \cdot x_0 + \bar{x}_3 \cdot x_2 \cdot x_1 \cdot x_0 =$$
$$= \bar{x}_3 \cdot x_2 \cdot x_0 \cdot (x_1 + \bar{x}_1) = \boldsymbol{\bar{x}_3 \cdot x_2 \cdot x_0}$$

Me ayuda a encontrar grupos simplificables pero…

¿puedo hacer esto más rápido sin tener que operar?

MAPAS DE KARNAUGH

- Según el número de unos agrupados se puede determinar el número de variables que se van a simplificar:
  - Si tenemos 2 celdas → 1 variable
  - Si tenemos 4 celdas → 2 variables
  - Si tenemos 8 celdas → 3 variables

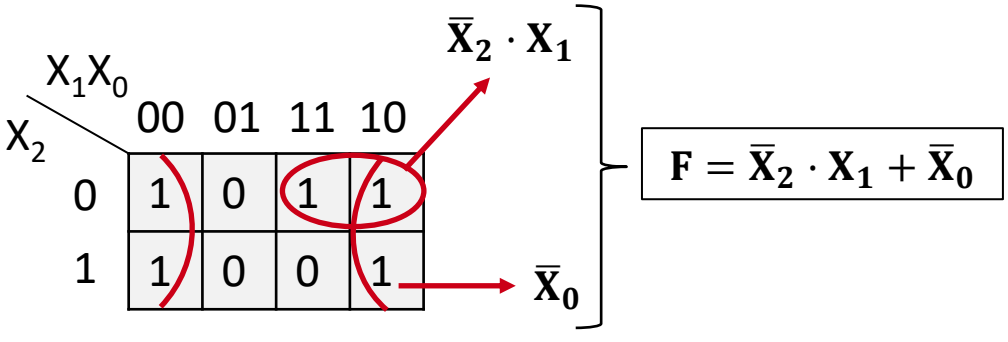
  …

- Se simplificarán aquellas variables que **NO** tengan el mismo valor en todas las celdas del grupo, las que tengan el mismo valor aparecerán en la expresión final

- En la expresión final, las variables aparecerán negadas cuando valgan '0' en todas las celdas del grupo y sin negar cuando valgan '1'

MAPAS DE KARNAUGH: EJEMPLO 2

- Minimizar la siguiente función expresada en forma de tabla de verdad:

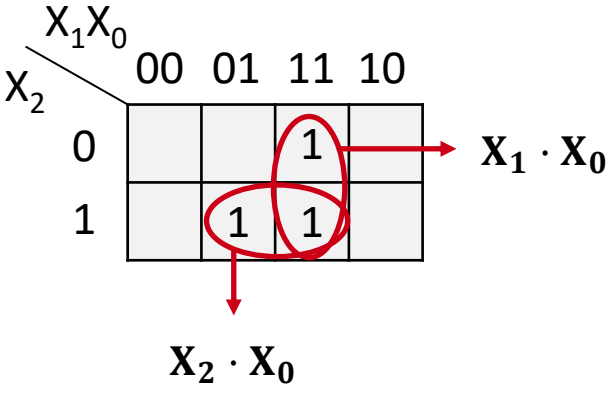


Agrupamos los unos

X_2	X_1	X_0	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

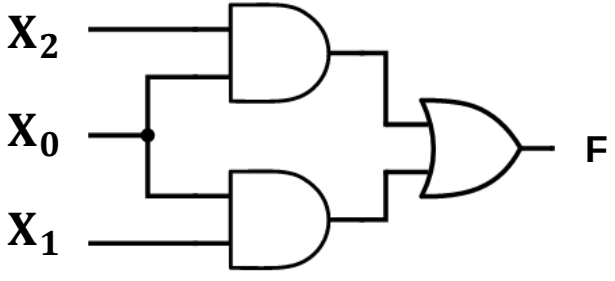
MAPAS DE KARNAUGH: EJEMPLO 3

- Minimizar el siguiente mapa de Karnaugh:



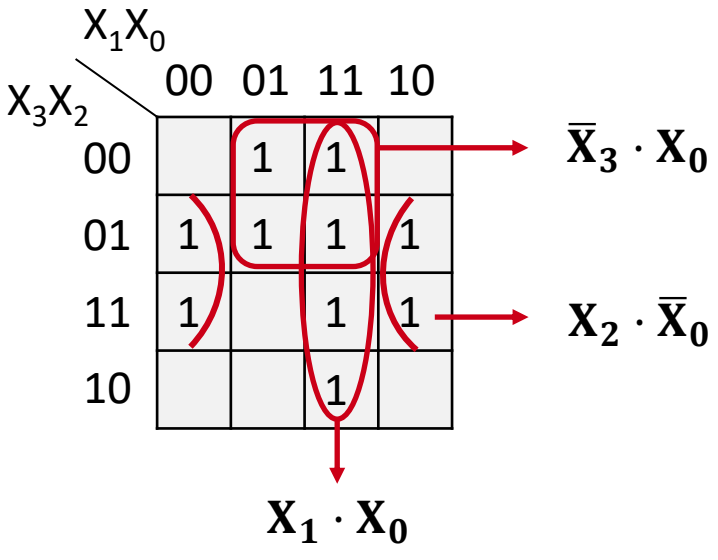
Función resultante

$$F = X_2 \cdot X_0 + X_1 \cdot X_0$$



MAPAS DE KARNAUGH: EJEMPLO 4

- Minimizar el siguiente mapa de Karnaugh:



Función resultante

$$\mathbf{F} = \overline{\mathbf{X}}_{\mathbf{3}} \cdot \mathbf{X}_{\mathbf{0}} + \mathbf{X}_{\mathbf{2}} \cdot \overline{\mathbf{X}}_{\mathbf{0}} + \mathbf{X}_{\mathbf{1}} \cdot \mathbf{X}_{\mathbf{0}}$$

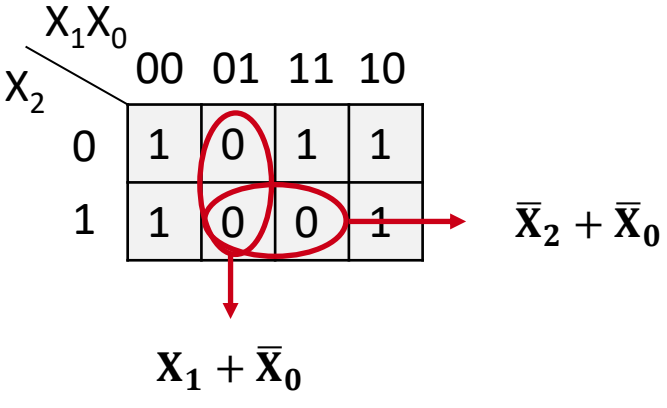
¿Cómo sería su esquemático?

MAPAS DE KARNAUGH: ¿Y LOS CEROS?

- Una función lógica también se puede minimizar agrupando los ceros del mapa de Karnaugh
- En este caso cada grupo será un maxtérmino o término suma
- La expresión final será un producto de sumas
- Se siguen las mismas reglas que con los unos salvo que, en la función lógica final, las variables aparecerán negadas cuando valgan '1' en todas las celdas del grupo y sin negar cuando valgan '0' (al revés que en el caso de los unos)

MAPAS DE KARNAUGH: EJEMPLO 5

- Minimizar el mapa de Karnaugh del ejemplo 2 pero ahora agrupando ceros:



Agrupando los unos teníamos:

$$F = \bar{X}_2 \cdot X_1 + \bar{X}_0$$

Agrupando los ceros:

$$F = (X_1 + \bar{X}_0)\cdot(\bar{X}_2 + \bar{X}_0)$$

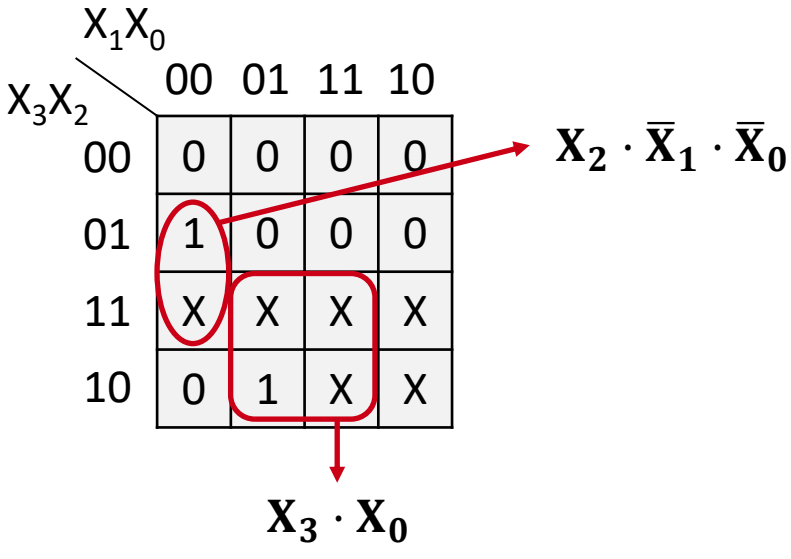
¡¡Genera un circuito con más puertas!!

MAPAS DE KARNAUGH: DON'T CARE

- Puede haber circuitos que no utilicen una determinada combinación de entradas
- En estos casos el valor de la salida da igual para esas combinaciones
- Para indicar esto en la tabla de verdad o en el mapa de Karnaugh se denota con una "**X**" o un "**–**"
- Estos términos se denominan "irrelevantes" o "don't care"
- A la hora de minimizar con Karnaugh, los términos don't care actúan como comodines: si me ayudan a crear mejores grupos los uso, pero si no no

MAPAS DE KARNAUGH: EJEMPLO 6

- Minimizar el siguiente mapa de Karnaugh:



Función resultante

$$\mathbf{F = X_3 \cdot X_0 + X_2 \cdot \overline{X}_1 \cdot \overline{X}_0}$$

CONTENIDOS

1. Representación de funciones lógicas. Tablas de verdad y formas canónicas
2. Minimización de circuitos digitales. Mapas de Karnaugh
3. **Conjuntos universales de puertas. Circuitos con NAND y NOR**

CONJUNTOS UNIVERSALES DE PUERTAS

- Un sistema combinacional cualquiera puede expresarse con una función lógica
- A su vez, una función lógica cualquiera puede representarse mediante variables y operadores AND, OR y NOT como hemos ido viendo a lo largo del tema
- Por tanto, cualquier sistema combinacional se puede construir utilizando únicamente puertas lógicas AND, OR y NOT
- A este grupo de 3 puertas lógicas se le conoce como **conjunto universal**
- Otros conjuntos universales de puertas, como los **NAND** y los **NOR**, son muy interesantes ya que permiten expresar circuitos utilizando un único tipo de puerta

TECNOLOGÍA DE COMPUTADORES

Tema 6: Módulos Combinacionales Básicos

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

COMPONENTES COMBINACIONALES

- Se pueden crear componentes más complejos a partir de puertas lógicas
- Un computador es un sistema complejo que:
  - Codifica y decodifica datos
  - Transmite y recibe información usando buses de datos
  - Procesa los datos realizando operaciones lógicas y aritméticas
  - Almacena temporalmente la información en memorias

En las siguientes secciones veremos los componentes que permiten realizar estas funciones

CONTENIDOS

1. Codificadores y decodificadores
2. Multiplexores y demultiplexores
3. Modularidad en VHDL
4. Circuitos aritméticos y de desplazamiento de bits
5. Memorias ROM

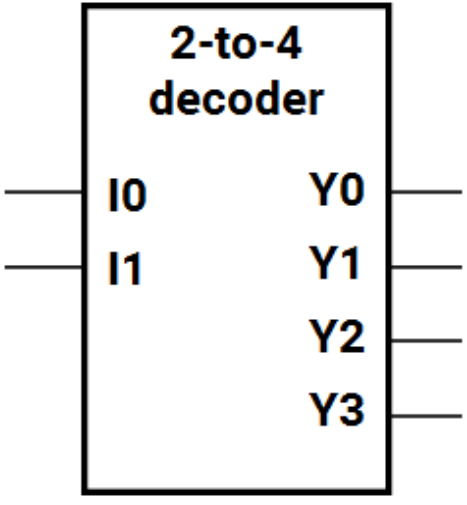
CONTENIDOS

1. **Codificadores y decodificadores**
2. Multiplexores y demultiplexores
3. Modularidad en VHDL
4. Circuitos aritméticos y de desplazamiento de bits
5. Memorias ROM

DECODIFICADORES

- Un decodificador tiene **n entradas** y **2^n salidas**
- Se nombran "*decodificador n-a-2^n*" (**ej**. 2-a-4, 3-a-8, 4-a-16, etc.)
- Se activa una única salida según la combinación de entrada
- Se pueden diseñar con lógica positiva o lógica negativa
- Pueden tener una señal de ***enable*** que "enciende o apaga" el componente

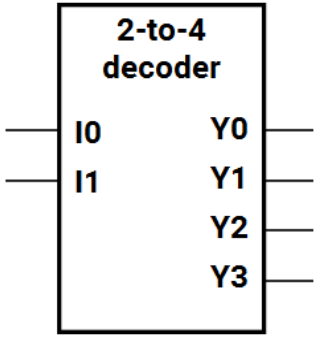
DECODIFICADOR 2-a-4: FUNCIONAMIENTO



Entradas		Salidas			
I1	I0	Y3	Y2	Y1	Y0
0	0	**0**	**0**	**0**	**1**
0	1	**0**	**0**	**1**	**0**
1	0	**0**	**1**	**0**	**0**
1	1	**1**	**0**	**0**	**0**

Se activa la salida decimal correspondiente al valor binario de entrada

DECODIFICADOR 2-a-4: VHDL




```
entity dec2_4 is
        port( I     : in   std_logic_vector(1 downto 0);
              Y     : out std_logic_vector(3 downto 0)
        );
end dec2_4;
```

```
architecture Behavior of dec2_4 is
begin
        Y <= "0001" when I = "00" else
             "0010" when I = "01" else
             "0100" when I = "10" else
             "1000";
end Behavior;
```

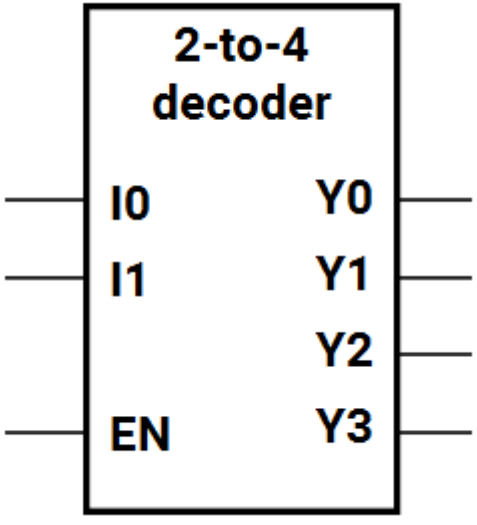
Esto se denomina **asignación condicional**

IMPORTANTE

Ambas arquitecturas son correctas y representan el mismo componente

```
architecture Behavior of dec2_4 is
begin
        Y(0) <= not (I(0)) and not (I(1));
        Y(1) <= not (I(0)) and I(1);
        Y(2) <= I(0) and not (I(1));
        Y(3) <= I(0) and I(1);
end Behavior;
```

DECODIFICADOR 2-a-4 CON ENABLE

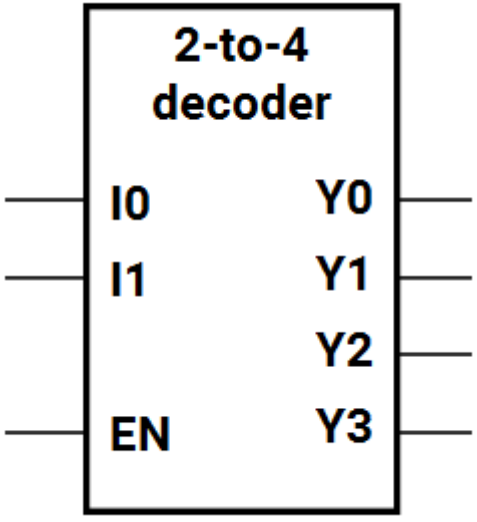


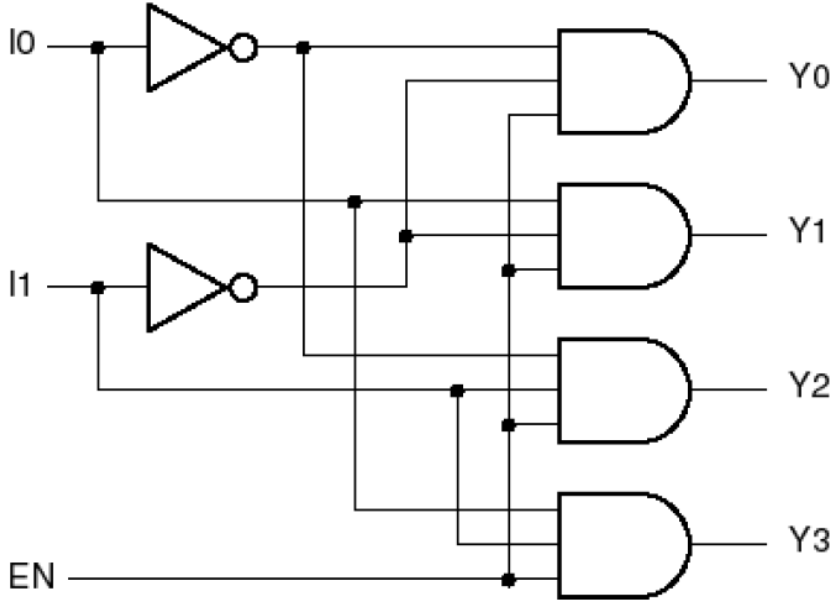
Don't care

Entradas			Salidas			
EN	I1	I0	Y3	Y2	Y1	Y0
0	X	X	**0**	**0**	**0**	**0**
1	0	0	**0**	**0**	**0**	**1**
1	0	1	**0**	**0**	**1**	**0**
1	1	0	**0**	**1**	**0**	**0**
1	1	1	**1**	**0**	**0**	**0**

La señal de Enable habilita / deshabilita el componente

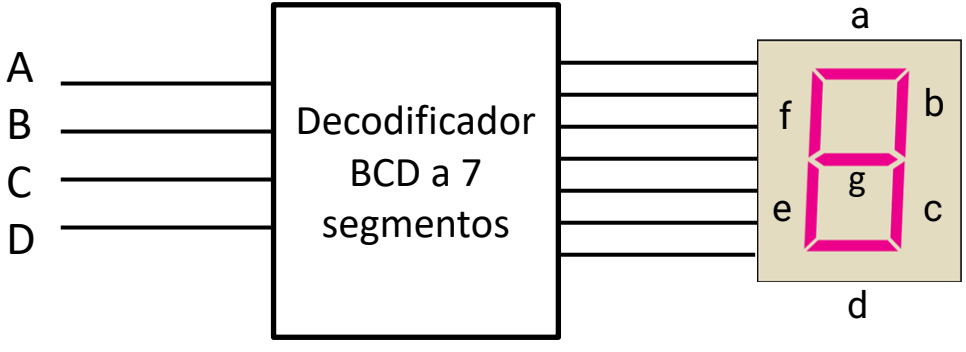
DECODIFICADOR 2-a-4 CON ENABLE



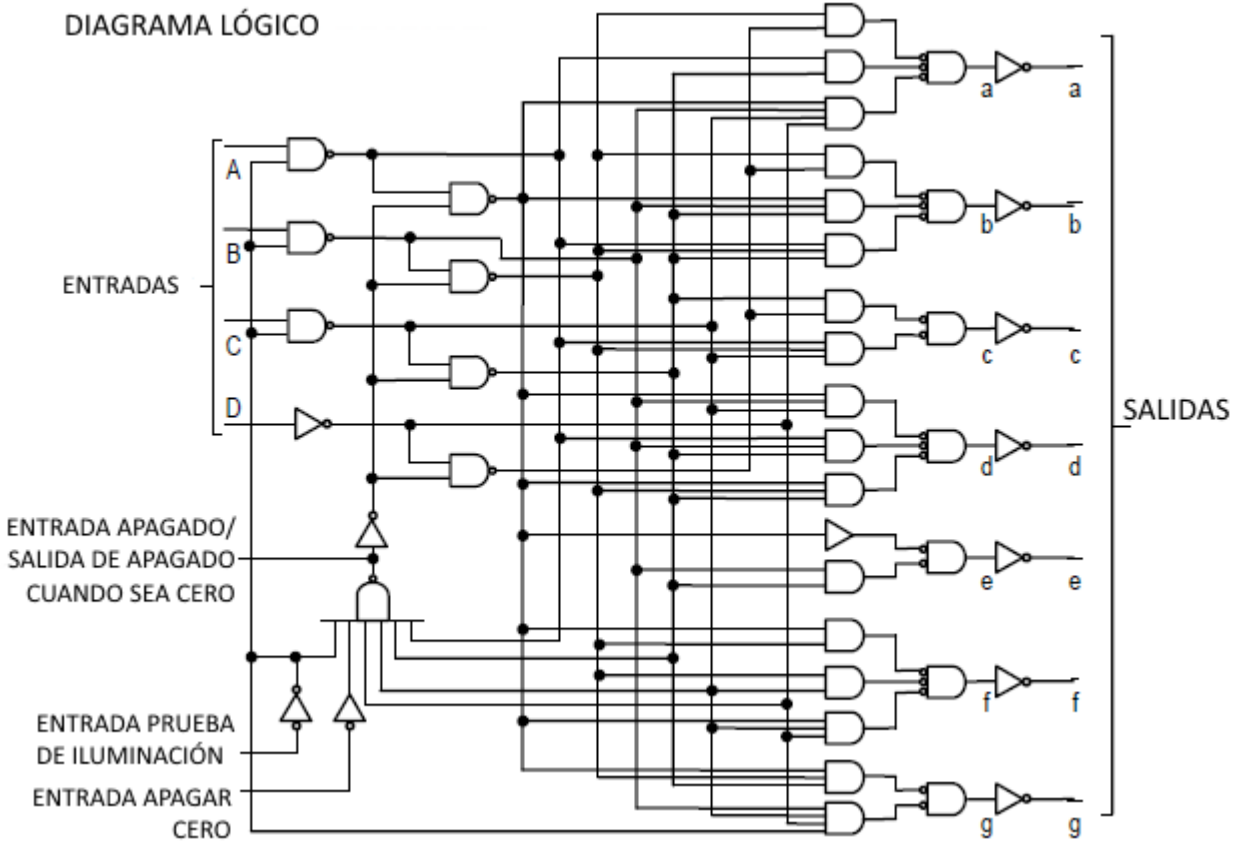


DECODIFICADOR BCD A 7 SEGMENTOS

- Hay decodificadores que activan más de una salida en cada combinación de entrada
- El decodificador de BCD (*binary coded decimal*) a 7 segmentos convierte un número binario a su representación en un display de 7 segmentos



DECODIFICADOR BCD A 7 SEGMENTOS



FUNCIONES LÓGICAS CON DECODIFICADORES

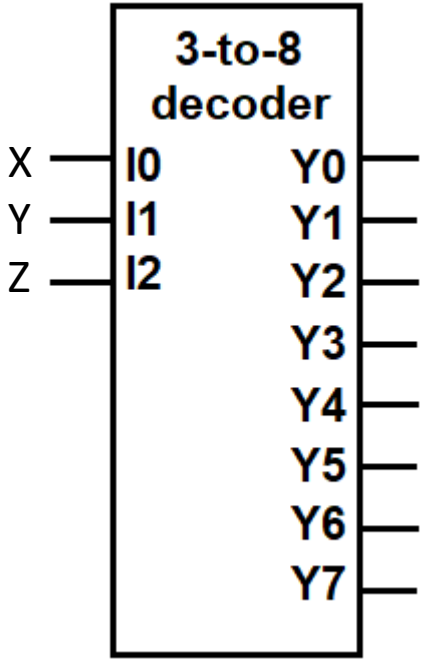
- Sabiendo que los decodificadores generales activan una única salida por cada combinación de entradas (quedando el resto de salidas desactivadas)

- Se puede construir un circuito combinacional con decodificadores y puertas lógicas que representen el comportamiento definido en una tabla de verdad

- Para ello:
  - Si el decodificador tiene las salidas activas a nivel alto utilizo puertas **OR**
  - Si el decodificador tiene las salidas activas a nivel bajo utilizo puertas **NAND**

EJEMPLO: FUNCIONES CON DECODIFICADORES

X	Y	Z	C	S
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**



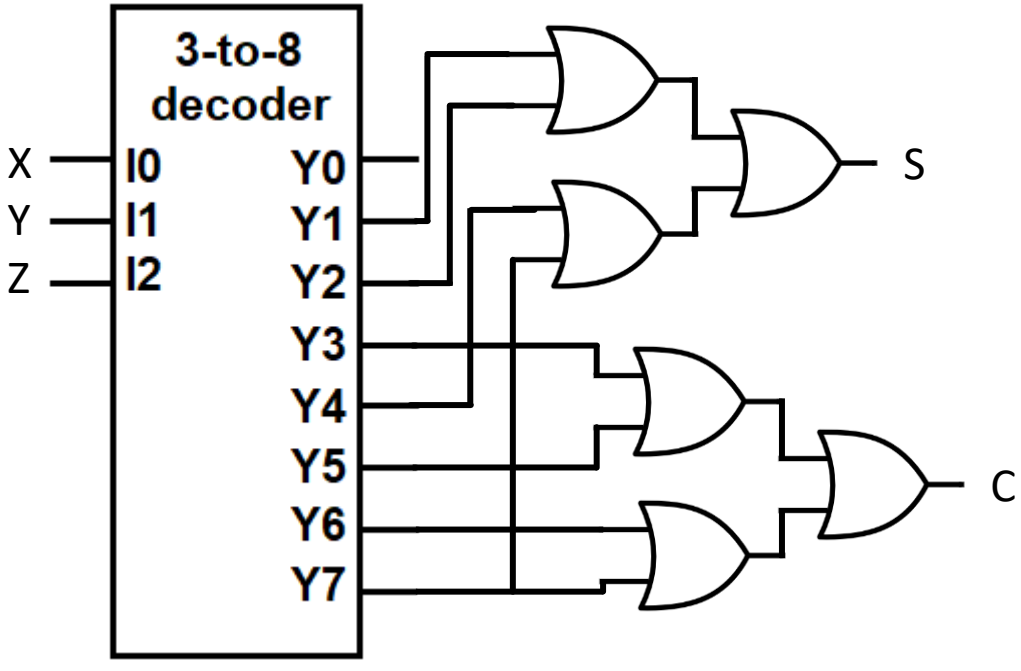
Conecto las salidas que valen uno a una puerta OR

En la salida C del ejemplo, conectaría los mintérminos m_3, m_5, m_6 y m_7

$$\sum_{X,Y,Z}(3,5,6,7)$$

EJEMPLO: FUNCIONES CON DECODIFICADORES

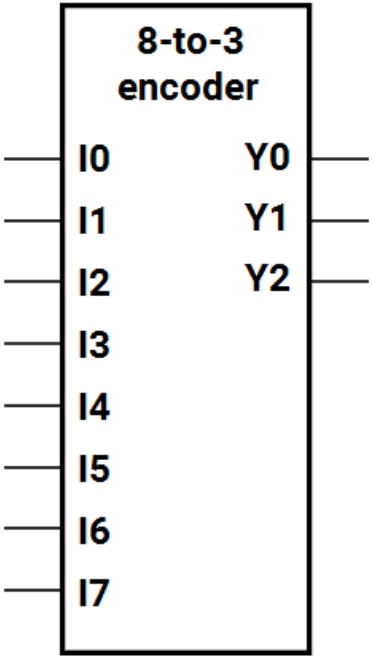
X	Y	Z	C	S
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**



CODIFICADORES

- Un codificador realiza la operación inversa al decodificador
- Tiene **2^n entradas** y **n salidas**
- Se nombran "*codificador 2^n-a-n*" (**ej**. 4-a-2, 8-a-3, 16-a-4, etc.)
- Se activa una combinación de salidas según la entrada activada
- Se pueden diseñar con lógica positiva o lógica negativa
- Pueden tener una señal de ***enable*** que "enciende o apaga" el componente

EJEMPLO: CODIFICADOR 8-a-3



Entradas								Salidas		
I7	I6	I5	I4	I3	I2	I1	I0	Y2	Y1	Y0
1	0	0	0	0	0	0	0	**1**	**1**	**1**
0	1	0	0	0	0	0	0	**1**	**1**	**0**
0	0	1	0	0	0	0	0	**1**	**0**	**1**
0	0	0	1	0	0	0	0	**1**	**0**	**0**
0	0	0	0	1	0	0	0	**0**	**1**	**1**
0	0	0	0	0	1	0	0	**0**	**1**	**0**
0	0	0	0	0	0	1	0	**0**	**0**	**1**
0	0	0	0	0	0	0	1	**0**	**0**	**0**

EJEMPLO: CODIFICADOR 8-a-3

Hay combinaciones de entrada que no se usan (ej. 11000000)

¿Qué ocurre si se activan varias a la vez?

Es necesario priorizar las entradas

Entradas								Salidas		
I7	I6	I5	I4	I3	I2	I1	I0	Y2	Y1	Y0
1	0	0	0	0	0	0	0	**1**	**1**	**1**
0	1	0	0	0	0	0	0	**1**	**1**	**0**
0	0	1	0	0	0	0	0	**1**	**0**	**1**
0	0	0	1	0	0	0	0	**1**	**0**	**0**
0	0	0	0	1	0	0	0	**0**	**1**	**1**
0	0	0	0	0	1	0	0	**0**	**1**	**0**
0	0	0	0	0	0	1	0	**0**	**0**	**1**
0	0	0	0	0	0	0	1	**0**	**0**	**0**

CODIFICADORES CON PRIORIDAD

- Permite que se activen más de una señal de entrada
- Se genera el valor de salida correspondiente a la entrada activa con mayor prioridad

Si la entrada n está activa, se ignora el valor de las entradas (n-1 ... 0)

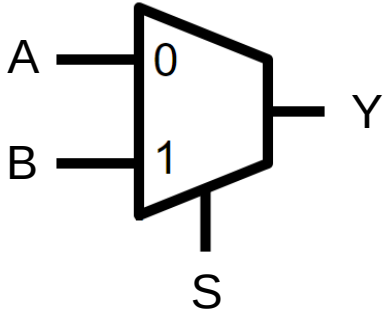
Entradas								Salidas		
I7	I6	I5	I4	I3	I2	I1	I0	Y2	Y1	Y0
1	X	X	X	X	X	X	X	**1**	**1**	**1**
0	1	X	X	X	X	X	X	**1**	**1**	**0**
0	0	1	X	X	X	X	X	**1**	**0**	**1**
0	0	0	1	X	X	X	X	**1**	**0**	**0**
0	0	0	0	1	X	X	X	**0**	**1**	**1**
0	0	0	0	0	1	X	X	**0**	**1**	**0**
0	0	0	0	0	0	1	X	**0**	**0**	**1**
0	0	0	0	0	0	0	X	**0**	**0**	**0**

CONTENIDOS

1. Codificadores y decodificadores
2. **Multiplexores y demultiplexores**
3. Modularidad en VHDL
4. Circuitos aritméticos y de desplazamiento de bits
5. Memorias ROM

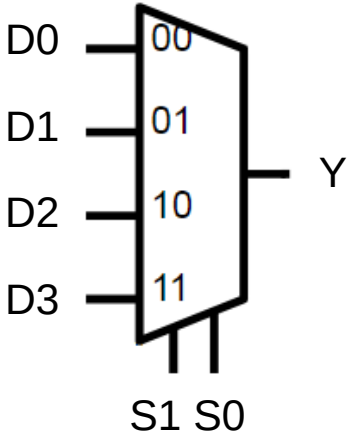
MULTIPLEXORES

- El multiplexor o MUX tiene varias entradas (*n*) y una única salida de datos
- Se utiliza para seleccionar y transmitir una de las entradas. De este modo, se comparte el mismo canal de transmisión entre todas las entradas
- Se nombran "*multiplexor n-a-1*" (**ej**. 2-a-1, 4-a-1, 8-a-1, etc.)

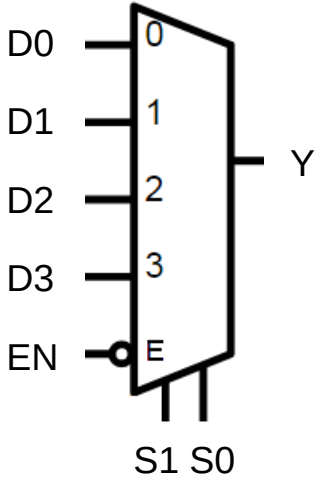


S	Y
0	**A**
1	**B**

MULTIPLEXOR 4-a-1



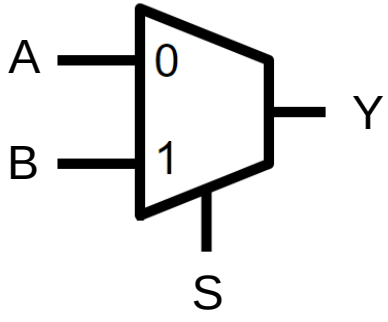
S1	S0	Y
0	0	**D0**
0	1	**D1**
1	0	**D2**
1	1	**D3**



EN	S1	S0	Y
1	X	X	**0**
0	0	0	**D0**
0	0	1	**D1**
0	1	0	**D2**
0	1	1	**D3**

La señal de Enable habilita / deshabilita el componente

MULTIPLEXOR 2-a-1: VHDL



S	Y
0	A
1	B

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;                    -- LIBRERÍA ESTÁNDAR

entity Mux2_1 is
        port( A : in   STD_LOGIC;
              B : in   STD_LOGIC;               -- ENTIDAD
              S : in   STD_LOGIC;
              Y : out STD_LOGIC
        );
end Mux2_1;

architecture Behavior of Mux2_1 is
begin
        Y <= A when (S = '0') else B;           -- COMPORTAMIENTO
end Behavior;
```

Se puede implementar con asignación condicional o con procesos (lo veremos más adelante)

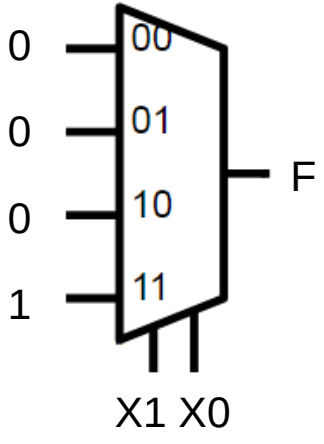
FUNCIONES LÓGICAS CON MULTIPLEXORES

- Al igual que ocurría con los decodificadores, con los multiplexores también podemos implementar funciones lógicas para reducir el área total del circuito

Ej.

X1	X0	F(X1, X0)
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

(Puerta AND)



Para una función de **N** entradas necesitaré un multiplexor de **2N:1**

FUNCIONES LÓGICAS CON MULTIPLEXORES

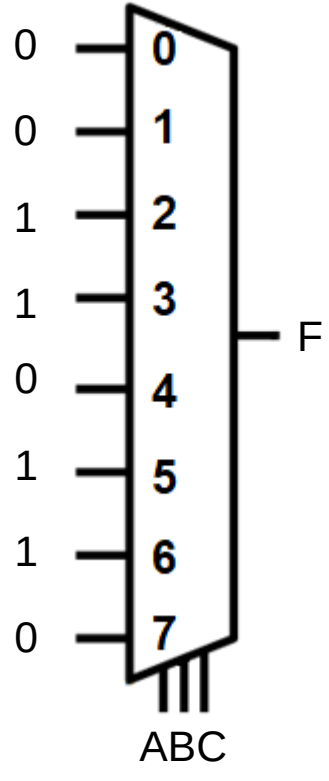
- **Ejercicio:** Implementar la siguiente función booleana utilizando:
  a) Un multiplexor de 8:1 y las puertas lógicas que sean necesarias
  b) Un multiplexor de 4:1 y las puertas lógicas que sean necesarias

$$F(A,B,C) = \sum_{A,B,C} m(2,3,5,6)$$

FUNCIONES LÓGICAS CON MULTIPLEXORES

a) Un multiplexor de 8:1

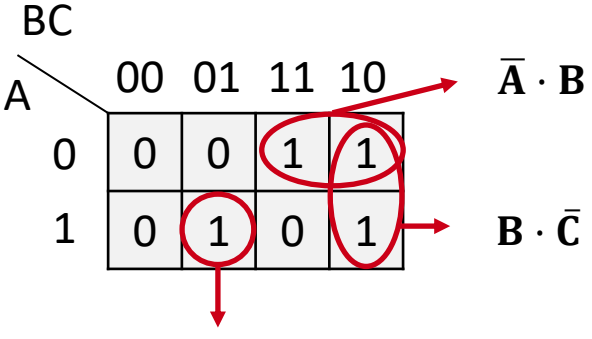
$$F(A,B,C) = \sum_{A,B,C} m(2,3,5,6)$$



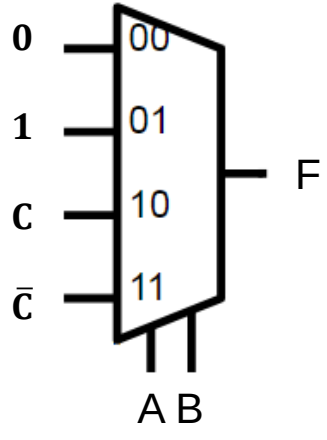
FUNCIONES LÓGICAS CON MULTIPLEXORES

a) Un multiplexor de 4:1

$$F(A,B,C) = \sum_{A,B,C} m(2,3,5,6)$$



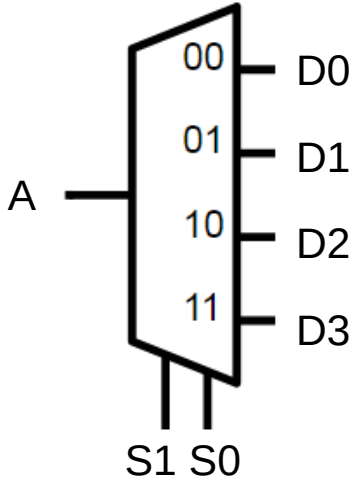
Doy valores



$$F = A \cdot \bar{B} \cdot C + B \cdot \bar{C} + \bar{A} \cdot B$$

DEMULTIPLEXORES

- El demultiplexor o DEMUX tiene una única entrada y varias salidas (*n*) de datos
- Se utiliza para repartir/direccionar la entrada por distintos canales de salida
- Se nombran "*demultiplexor 1-a-n*" (**ej**. 1-a-2, 1-a-4, 1-a-8, etc.)



S1	S0	D3	D2	D1	D0
0	0	**0**	**0**	**0**	**A**
0	1	**0**	**0**	**A**	**0**
1	0	**0**	**A**	**0**	**0**
1	1	**A**	**0**	**0**	**0**

CONTENIDOS

1. Codificadores y decodificadores
2. Multiplexores y demultiplexores
3. **Modularidad en VHDL**
4. Circuitos aritméticos y de desplazamiento de bits
5. Memorias ROM

FORMAS DE DISEÑAR HARDWARE

- **Data-flow modeling (**bajo nivel**):** se utilizan las ecuaciones lógicas
- **Behavioral modeling:** se describe el comportamiento del circuito
- **Structural modeling (**alto nivel**):** se utilizan componentes ya definidos con alguno de los modos anteriores y se conectan entre sí para conseguir el funcionamiento deseado. No es necesario conocer el funcionamiento interno de los componentes, sólo sus conexiones

Los circuitos más complejos utilizan una mezcla de todos

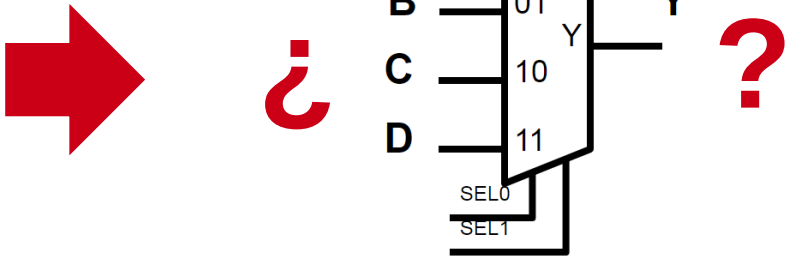
DISEÑO MODULAR O ESTRUCTURADO

- ***Structural modeling*** permite crear diseños modulares en VHDL usando:
  - Los códigos VHDL de los componentes necesarios
  - Un diagrama que nos muestre cómo se conectan entre sí
- Con la estructura ***component*** declaramos los componentes a usar y con la instrucción ***port map*** los ***instanciamos*** (conectamos) en nuestro diseño
- Los diseños modulares son más fáciles de entender que los diseños a bajo nivel
- Los módulos ya definidos se pueden reutilizar sin tener que reinventar la rueda

EJEMPLO: DISEÑO MODULAR

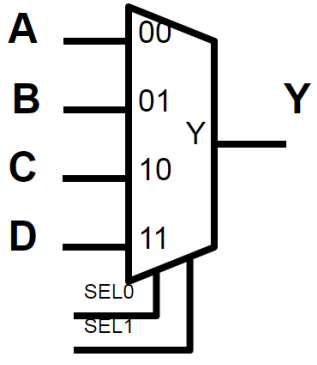
- Diseñar un multiplexor 4:1 utilizando el código VHDL de un multiplexor de 2:1

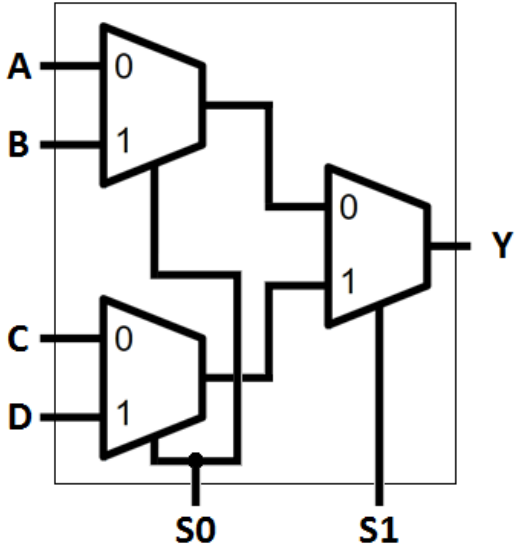
```
entity mux2_1 is
        port( A : in   STD_LOGIC;
              B : in   STD_LOGIC;
              S : in   STD_LOGIC;
              Y : out STD_LOGIC
        );
end mux2_1;

architecture Behavioral of mux2_1 is
begin
        Y <= A when (S = '0') else B;
end Behavioral;
```



EJEMPLO: DISEÑO MODULAR

S1	S0	Y
0	0	A
0	1	B
1	0	C
1	1	D

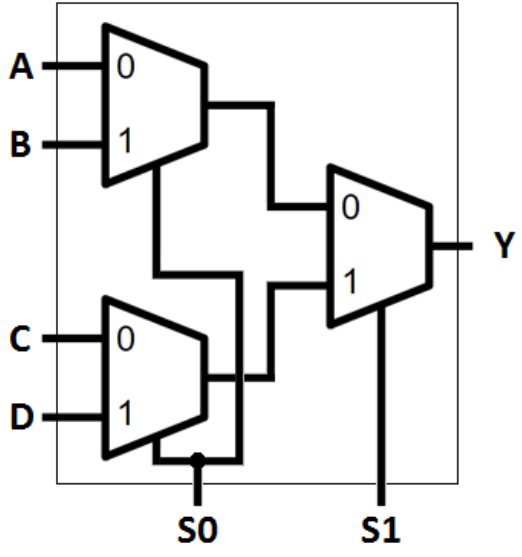




Con el diagrama del circuito final y el código del MUX 2:1 podemos construir el MUX 4:1

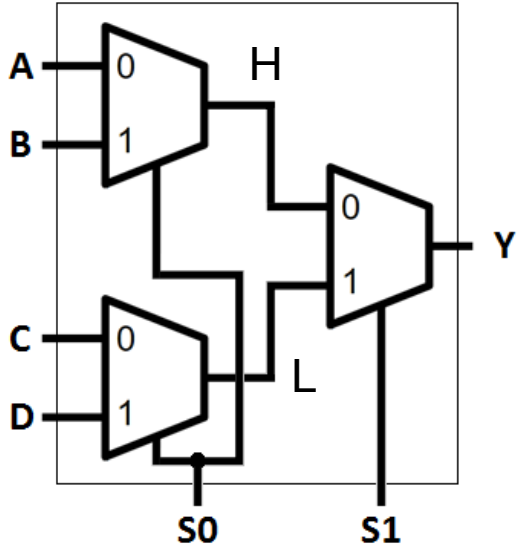
EJEMPLO: DISEÑO MODULAR

```
entity mux4_1 is
        port( A : in   std_logic;
              B : in   std_logic;
              C : in   std_logic;
              D : in   std_logic;
              S0, S1 : in std_logic;
              Y : out std_logic
        );
end mux4_1;

architecture Structural of mux4_1 is
        -- Definimos el componente a utilizar
begin
        -- Conectamos los componentes
end Structural;
```



EJEMPLO: DISEÑO MODULAR

```
architecture Structural of mux4_1 is
        -- Definimos el componente a utilizar
        component mux2_1 is
                port ( A : in std_logic;
                       B : in std_logic;
                       S : in std_logic;
                       Y : out std_logic
                );
        end component;
        -- Declaramos las señales internas
        signal H, L : std_logic;
begin
        -- Instanciamos los tres multiplexores 2:1
        highMux : mux2_1 port map (A, B, S0, H);
        lowMux  : mux2_1 port map (C, D, S0, L);
        finalMux : mux2_1 port map (H, L, S1, Y);
end Structural;
```

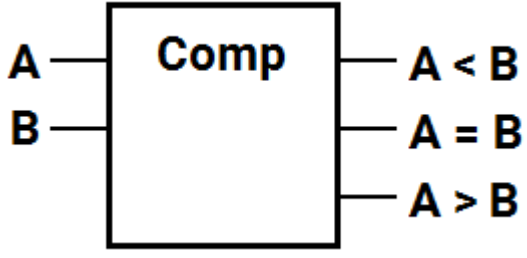


CONTENIDOS

1. Codificadores y decodificadores
2. Multiplexores y demultiplexores
3. Modularidad en VHDL
4. **Circuitos aritméticos y de desplazamiento de bits**
5. Memorias ROM

COMPARADORES

- Circuitos que comparan dos números y determinan cuál es mayor, menor o si ambos son iguales



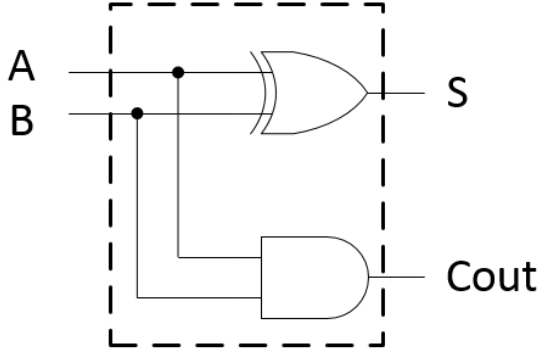
A	B	A < B	A = B	A > B
0	0	**0**	**1**	**0**
0	1	**1**	**0**	**0**
1	0	**0**	**0**	**1**
1	1	**0**	**1**	**0**

¡Es una puerta XNOR!

SEMISUMADOR

- Es el bloque sumador más sencillo
- Realiza la suma aritmética de dos números A y B
- A la salida se obtiene la suma (S) y el acarreo (C_{out})

PROBLEMA
No tiene bit de acarreo de entrada



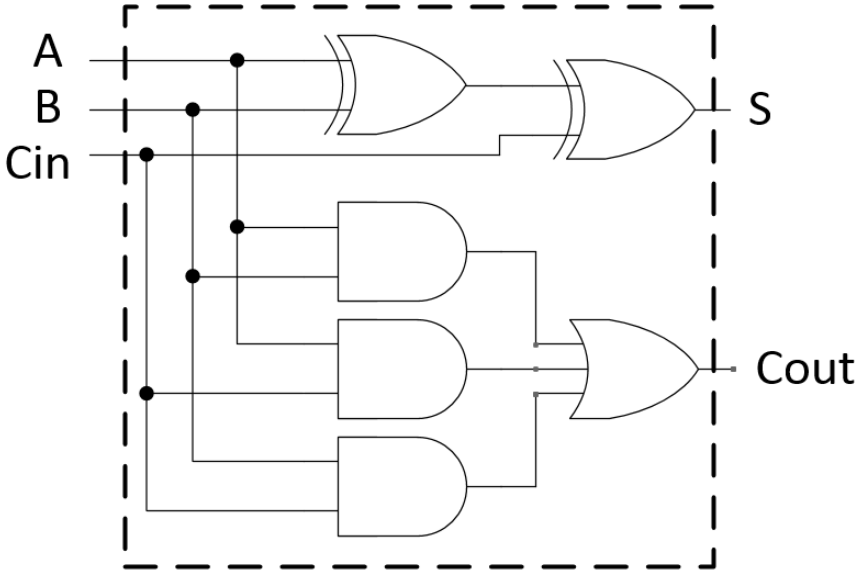
A	B	Cout	S
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

$S = A \oplus B$

$Cout = A \cdot B$

SUMADOR COMPLETO

- Añadir el acarreo de entrada (Cin) al semisumador

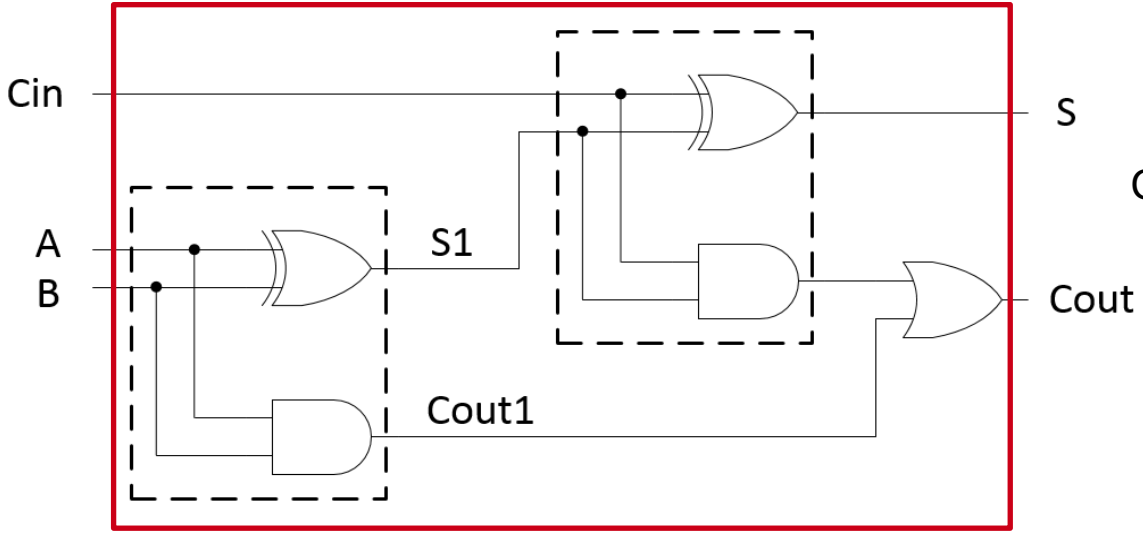


Cin	A	B	Cout	S
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

$$S = A \oplus B \oplus Cin$$
$$Cout = AB + ACin + BCin$$

SUMADOR COMPLETO

- También se puede diseñar a partir de dos semisumadores



$S = A \oplus B \oplus Cin$

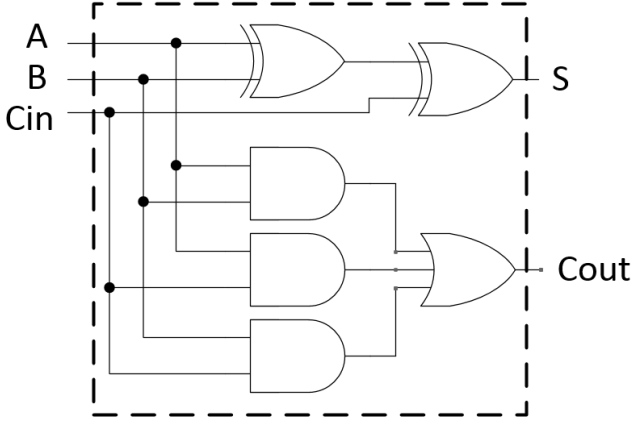
$Cout = AB + ACin + BCin$

$Cout = AB + Cin \cdot (A \oplus B)$

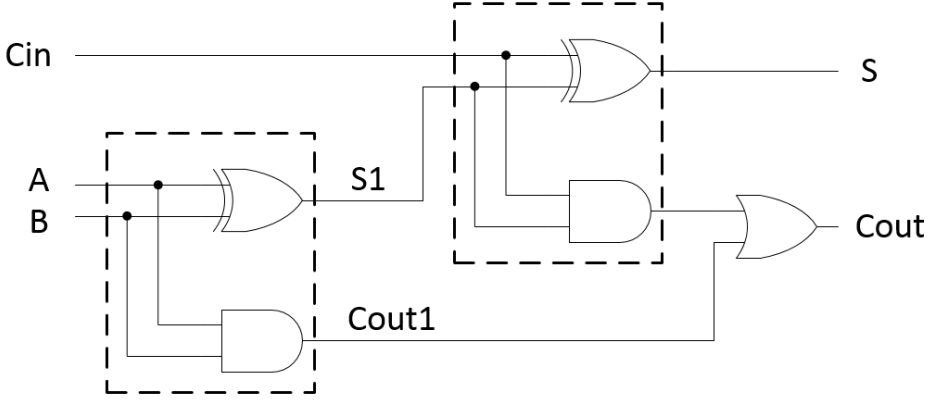
$Cout = AB + Cin \cdot S1$

SUMADOR COMPLETO

¿Cuál es mejor?



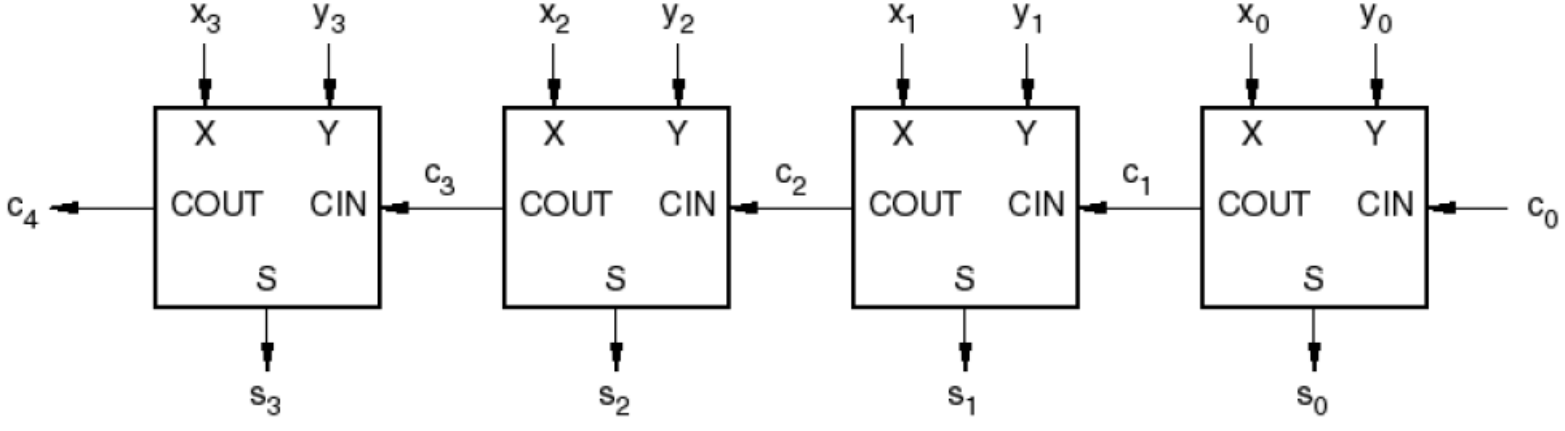
Mayor área
(Mayor número
de puertas)



Mayor retardo
(Cout tarda más
en calcularse)

SUMADOR CON ACARREO SERIE

- Para sumar números de *N* bits podemos encadenar *N* sumadores completos de 1 bit
- Son lentos cuando *N* es grande porque la velocidad de cálculo de la suma depende de la transmisión del acarreo



SUMADOR CON ACARREO ANTICIPADO

- Soluciona el problema de retardo del sumador con acarreo serie
- Calcula el acarreo de salida en cuanto se determina el acarreo de entrada
- Del sumador completo se tenía:

$$Cout = AB + ACin + BCin \Rightarrow Cout = \underbrace{AB}_{G} + \underbrace{Cin \cdot S1}_{P}$$

- Se descompone el acarreo en **acarreo generado** (G) y **acarreo propagado** (P)

SUMADOR CON ACARREO ANTICIPADO

$$P_i = A_i \oplus B_i$$
$$G_i = A_i \cdot B_i$$

- Podemos reescribir una fórmula para cada acarreo usando esta notación:

$Cout_1 = G_0 + P_0 \cdot Cout_0$ ← **Cin**

$Cout_2 = G_1 + P_1 \cdot Cout_1 = G_1 + P_1 \cdot (G_0 + P_0 \cdot Cout_0)$

$Cout_3 = G_2 + P_2 \cdot Cout_2 = G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot Cout_0))$

$Cout_4 = G_3 + P_3 \cdot Cout_3 = G_3 + P_3 \cdot (G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot Cout_0)))$

…

- Los acarreos de cada bit se pueden calcular de forma independiente
- Para muchos bits la lógica se complica y el área crece mucho
- Se suelen usar en bloques de 4 bits:

  (**Ej**: Si hay que sumar números de 32 bits usaremos 8 sumadores de 4 bits)

SUMADOR / RESTADOR

¿Cómo se hace una resta en binario?

- **Usando el Complemento a 2 (Ca2):** invertir los bits y sumar 1 al resultado
- **Ej.** Hacer la operación (8 – 6) en binario

· Representamos el número -6 usando el complemento a 2:

6 = 0110 → Invierto los bits → 1001 → Sumo 1 → 1010 = -6_{Ca2}

· Ahora sumo 8 + (-6):

8 + (-6) = 1000 + 1010 = 0010 = 2 El bit adicional lo ignoramos (overflow)

SUMADOR / RESTADOR

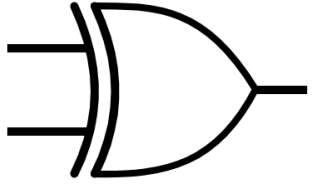
¿Cómo podríamos invertir los bits de un número?

¡Usando puertas lógicas XOR!

· Cuando X = 0 el resultado es igual a Y

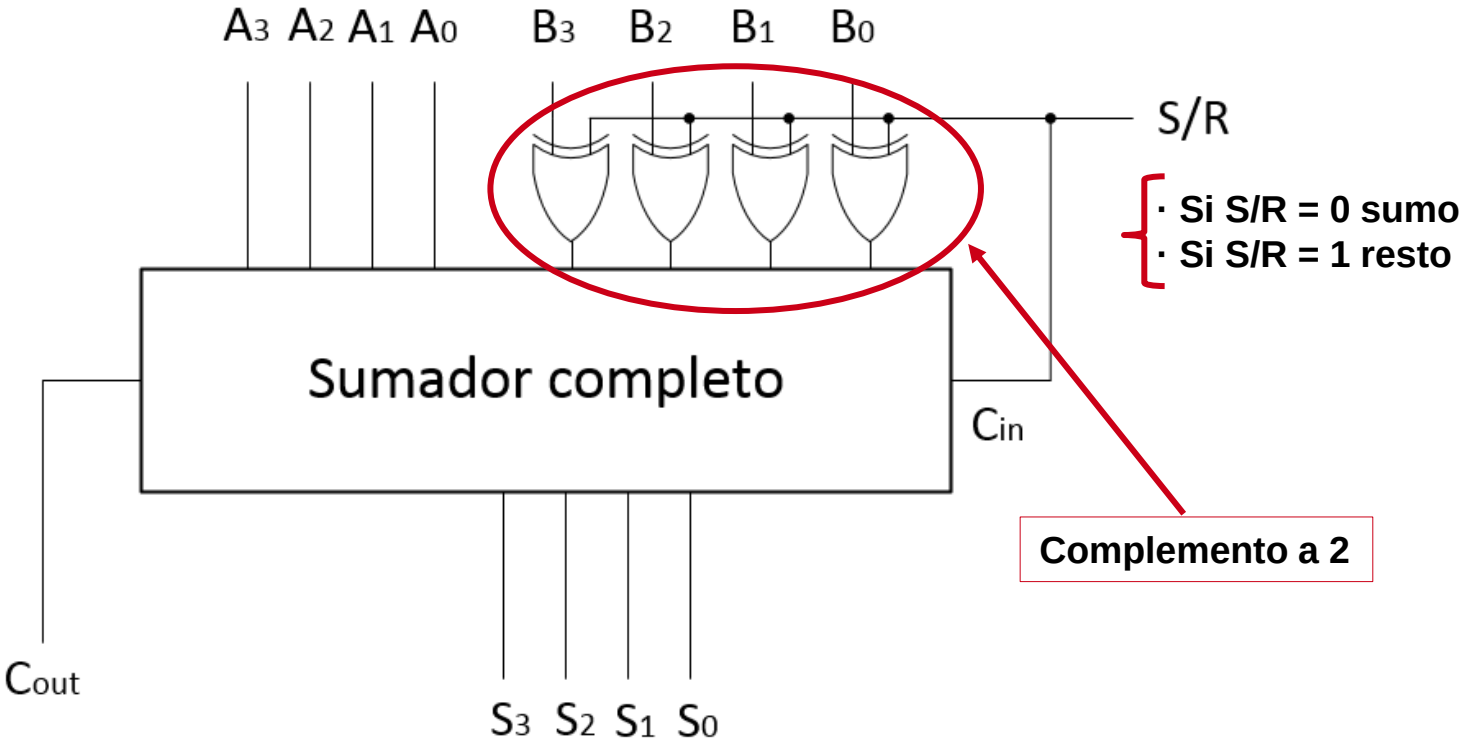
· Cuando X = 1 el resultado es el contrario de Y

X	Y	X⊕Y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

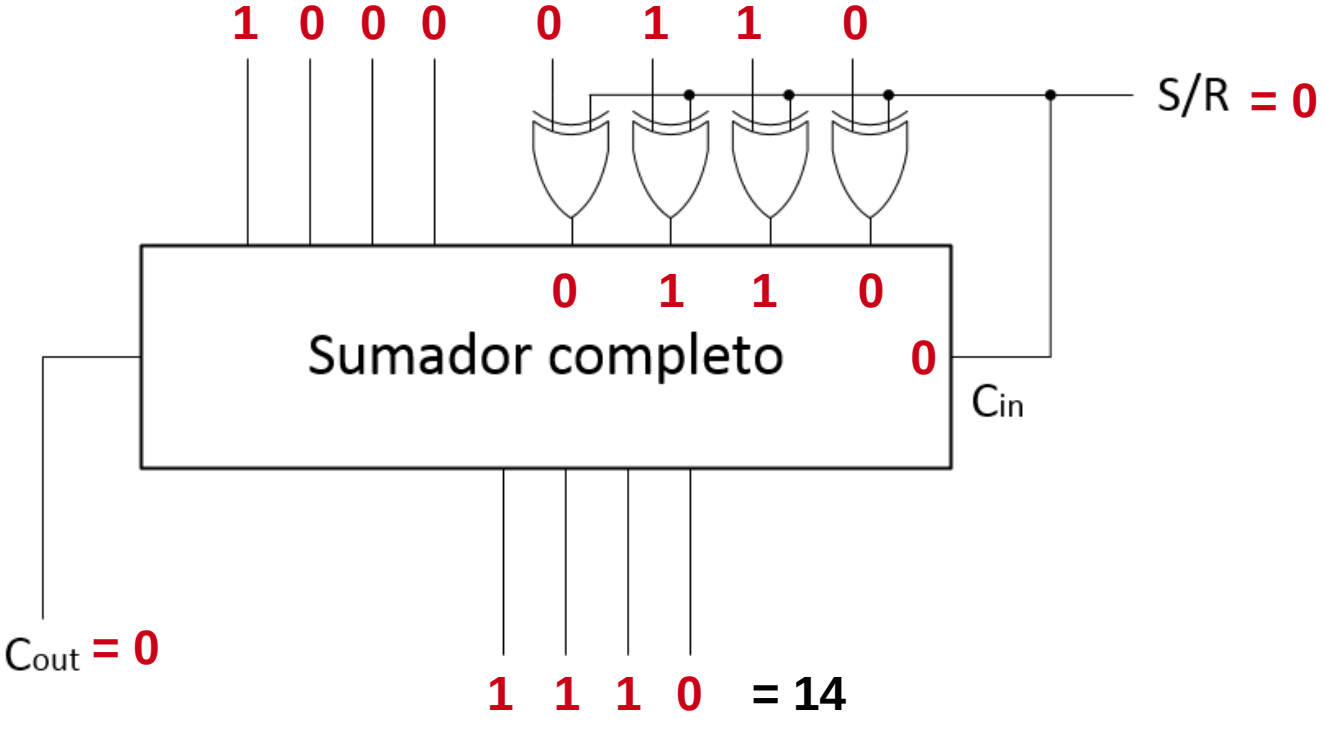


Se puede usar X como señal que cambia entre suma y resta

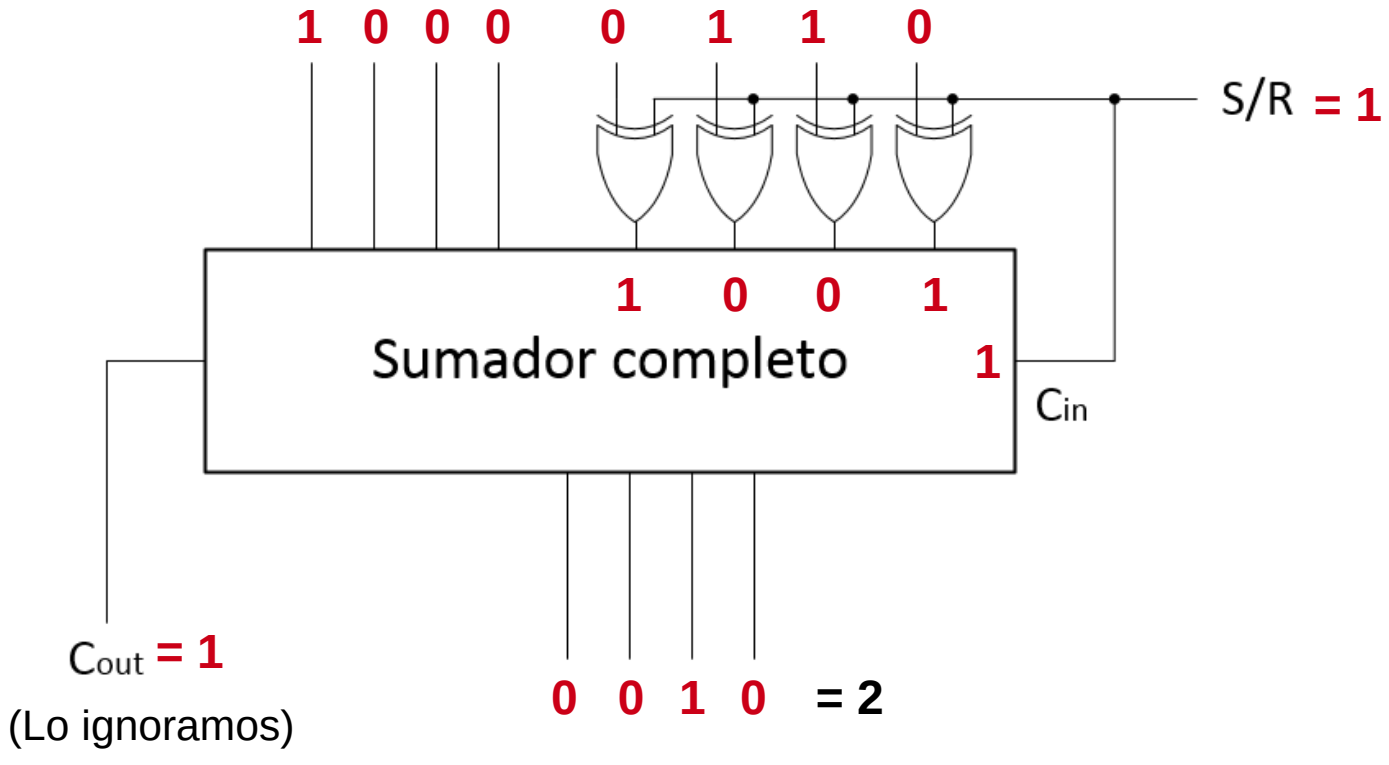
SUMADOR / RESTADOR



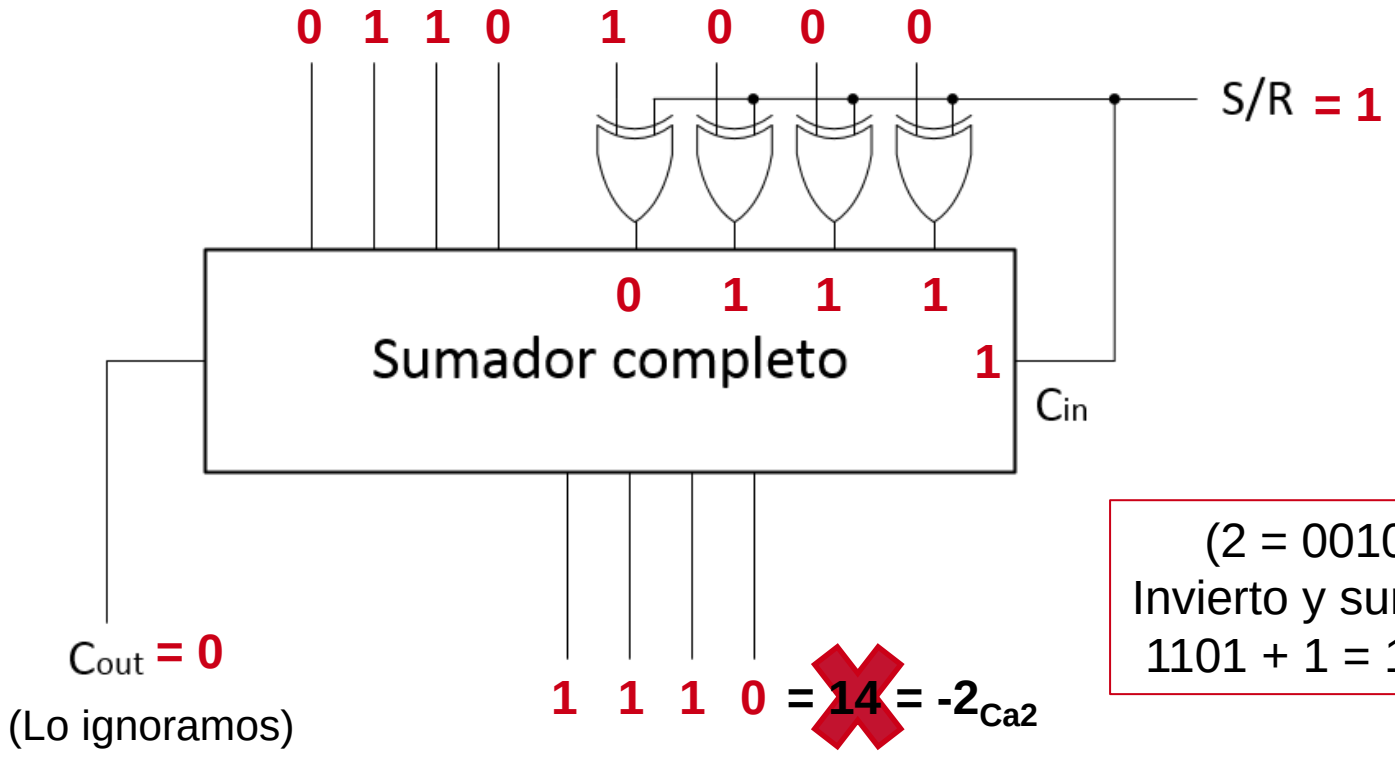
SUMADOR / RESTADOR: EJEMPLO (8 + 6)



SUMADOR / RESTADOR: EJEMPLO (8 – 6)



SUMADOR / RESTADOR: EJEMPLO (6 – 8)



(2 = 0010)
Invierto y sumo 1
1101 + 1 = 1110

SUMAS Y RESTAS EN VHDL

- Para realizar operaciones aritméticas en VHDL es necesario declarar el paquete **NUMERIC_STD** en la sección de librerías
- **Las señales std_logic_vector no se pueden sumar o restar**
- El paquete **NUMERIC_STD** define señales **signed** y **unsigned** con las que se puede operar aritméticamente
- Este paquete también define la conversión de std_logic_vector a unsigned y viceversa:

```
signal A : std_logic_vector(7 downto 0);
signal A_uns : unsigned(7 downto 0);

A_uns <= unsigned(A);
A <= std_logic_vector(A_uns);
```

LA UNIDAD ARITMÉTICO-LÓGICA (ALU)

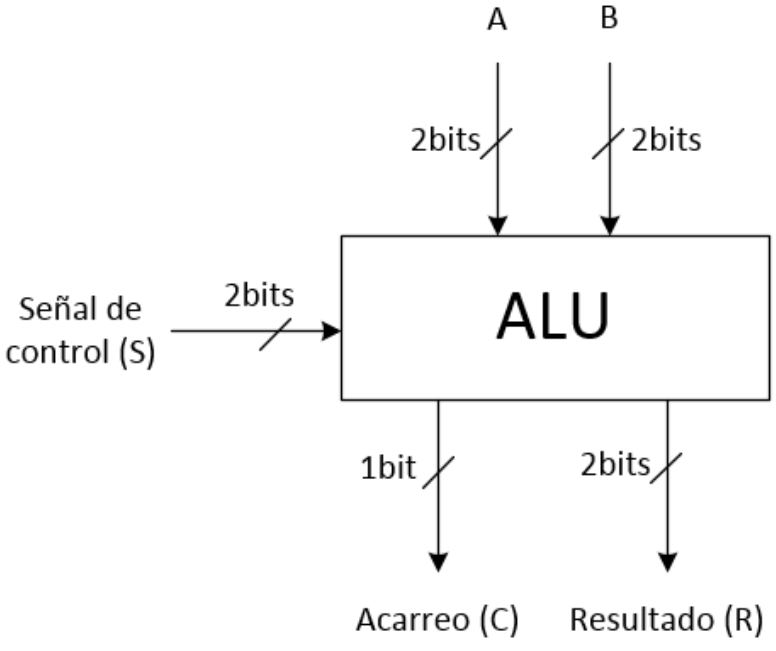
- Es la unidad encargada de procesar datos según una operación determinada por la instrucción en curso
- La unidad de control de la ALU es la que le manda los datos y la operación
- **La ALU realiza:**
  - **Operaciones aritméticas** (suma y resta)
  - **Operaciones lógicas** (and, or, not, xor, xnor y comparaciones)
  - **Operaciones de desplazamiento y rotación**
- Las operaciones de multiplicación y división se suelen realizar en otro módulo

LA UNIDAD ARITMÉTICO-LÓGICA (ALU)

- Ejemplo de una ALU sencilla:

Señal de control (S)	Operación
00	**Suma**
01	**Resta**
10	**AND**
11	**OR**

Código de operación (*Opcode*)

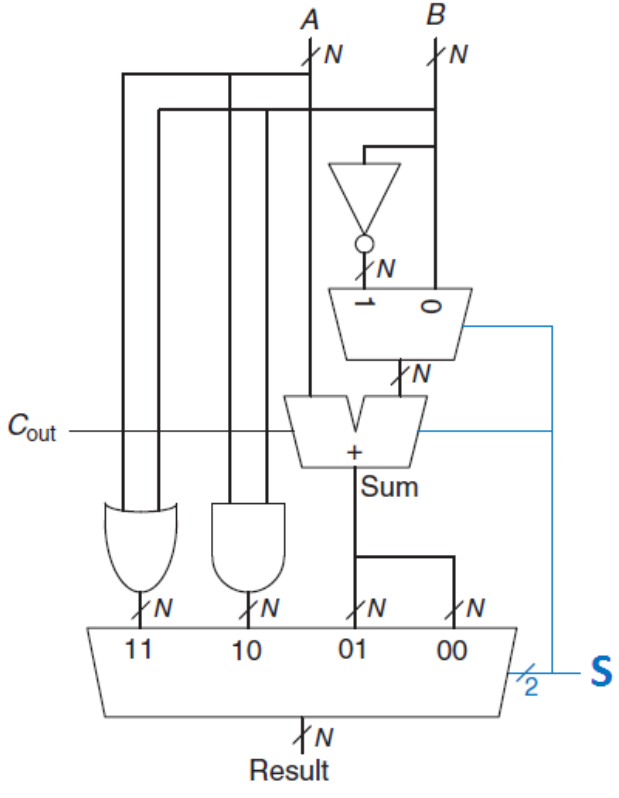


LA UNIDAD ARITMÉTICO-LÓGICA (ALU)

- Ejemplo de una ALU sencilla:

Señal de control (S)	Operación
00	**Suma**
01	**Resta**
10	**AND**
11	**OR**

Código de operación (*Opcode*)



DESPLAZAMIENTO Y ROTACIÓN

- Se utilizan en la ALU para multiplicar o dividir entre potencias de 2:
  - **Desplazamiento lógico (LS):** desplaza los bits N posiciones hacia la derecha (LSR) o hacia la izquierda (LSL) rellenando los huecos con ceros

  *Ej. **11001** ; LSR (2bits) = 00110 ; LSL (2bits) = 00100*
  - **Desplazamiento aritmético (AS):** igual que el anterior pero el LSR mantiene el bit más significativo. ASL es igual que el LSL

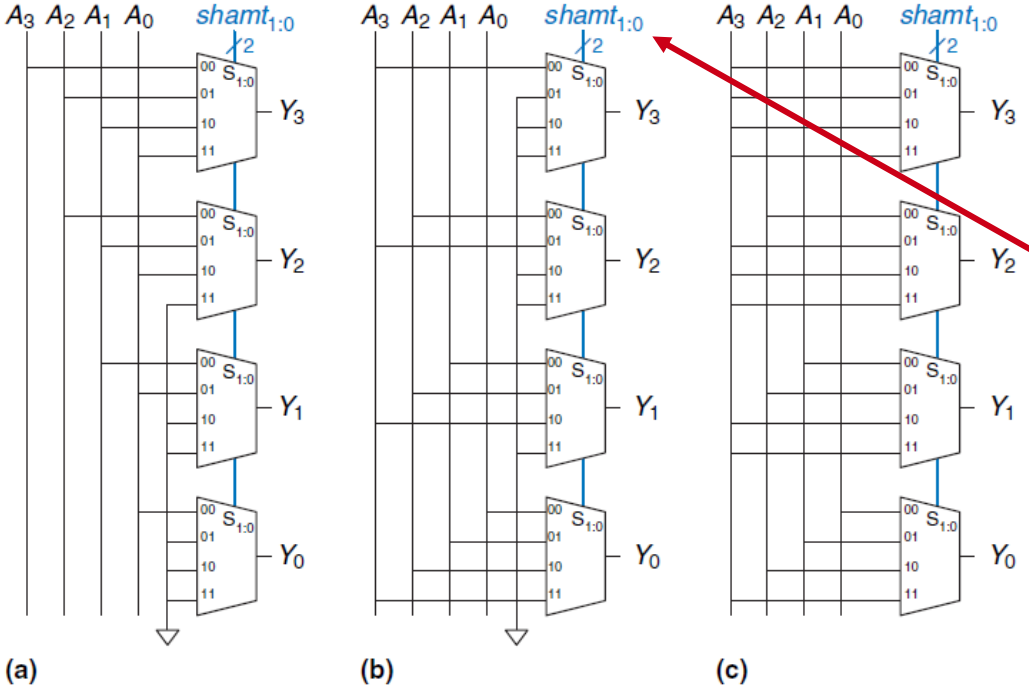
  *Ej. **11001** ; ASR (2bits) = 11110 ; ASL (2bits) = 00100*
  - **Rotación (RO):** desplaza los bits en círculo, de modo que los que salen por un lado entran por el otro extremo

  *Ej. **11001** ; ROR (2bits) = 01110 ; ROL (2bits) = 00111*

DESPLAZAMIENTO Y ROTACIÓN

¿Cómo se implementaría un desplazador con componentes hardware?

(a) LSL
(b) LSR
(c) ASR



shamt es la cantidad de bits a desplazar

(si *shamt* = 00 entonces Y = A)

CONTENIDOS

1. Codificadores y decodificadores
2. Multiplexores y demultiplexores
3. Modularidad en VHDL
4. Circuitos aritméticos y de desplazamiento de bits
5. **Memorias ROM**

MEMORIAS ROM

- Las memorias ROM (read-only memory) se utilizan para almacenar información
- **No se puede escribir** información en ellas, sólo se permite la lectura
- Se pueden usar para almacenar tablas de verdad, pudiendo realizar cualquier función lógica
- Las entradas son las direcciones de lectura de la ROM
- Las salidas son los datos almacenados en esas direcciones

FUNCIONES LÓGICAS CON MEMORIAS ROM

- Podemos implementar las siguientes funciones lógicas en una ROM:

F0 = A'B'C + AB'C' + AB'C

F1 = A'B'C + A'BC' + ABC

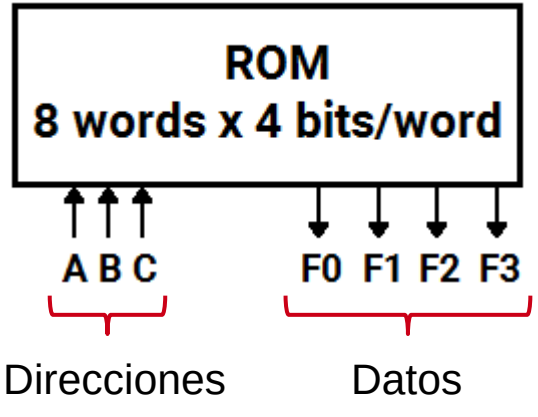
F2 = A'B'C' + A'B'C + AB'C'

F3 = A'BC + AB'C' + ABC'

Dirección			Datos			
A	B	C	F0	F1	F2	F3
0	0	0	**0**	**0**	**1**	**0**
0	0	1	**1**	**1**	**1**	**0**
0	1	0	**0**	**1**	**0**	**0**
0	1	1	**0**	**0**	**0**	**1**
1	0	0	**1**	**0**	**1**	**1**
1	0	1	**1**	**0**	**0**	**0**
1	1	0	**0**	**0**	**0**	**1**
1	1	1	**0**	**1**	**0**	**0**

FUNCIONES LÓGICAS CON MEMORIAS ROM

- La ROM actúa como una tabla de consulta que almacena las salidas pre-calculadas de las funciones



Dirección			Datos			
A	B	C	F0	F1	F2	F3
0	0	0	**0**	**0**	**1**	**0**
0	0	1	**1**	**1**	**1**	**0**
0	1	0	**0**	**1**	**0**	**0**
0	1	1	**0**	**0**	**0**	**1**
1	0	0	**1**	**0**	**1**	**1**
1	0	1	**1**	**0**	**0**	**0**
1	1	0	**0**	**0**	**0**	**1**
1	1	1	**0**	**1**	**0**	**0**

TECNOLOGÍA DE COMPUTADORES

Tema 7: Elementos de Memoria

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CIRCUITOS SECUENCIALES

- En un circuito secuencial, las salidas del mismo dependen de las entradas actuales y de la secuencia de entradas anteriores
- Debido a esto se dice que los circuitos secuenciales “tienen memoria”
- La memoria se consigue mediante la realimentación de señales, lo que da lugar a elementos secuenciales básicos como los **biestables**
- Las secuencias de entradas anteriores se denominan **estados**. Un estado contiene toda la información acerca del comportamiento pasado del circuito

CONTENIDOS

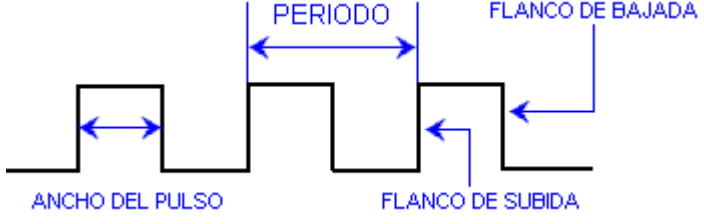
1. Biestables. Tipos y propiedades
2. Instrucciones secuenciales en VHDL: el *process*
3. Ejemplos en VHDL de diseños con bloques *process*
4. Simulación hardware con VHDL: el *testbench*

CONTENIDOS

1. **Biestables. Tipos y propiedades**
2. Instrucciones secuenciales en VHDL: el *process*
3. Ejemplos en VHDL de diseños con bloques *process*
4. Simulación hardware con VHDL: el *testbench*

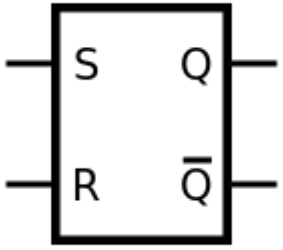
BIESTABLES

- Son componentes con dos estados usados para almacenar información binaria
- **Latch**: biestable asíncrono que muestrea las entradas continuamente y cambia las salidas en cualquier momento. No tiene señal de reloj
- **Flip-flop**: biestable síncrono que muestrea las entradas y cambia las salidas en ciertos instantes de tiempo marcados por una **señal de reloj**

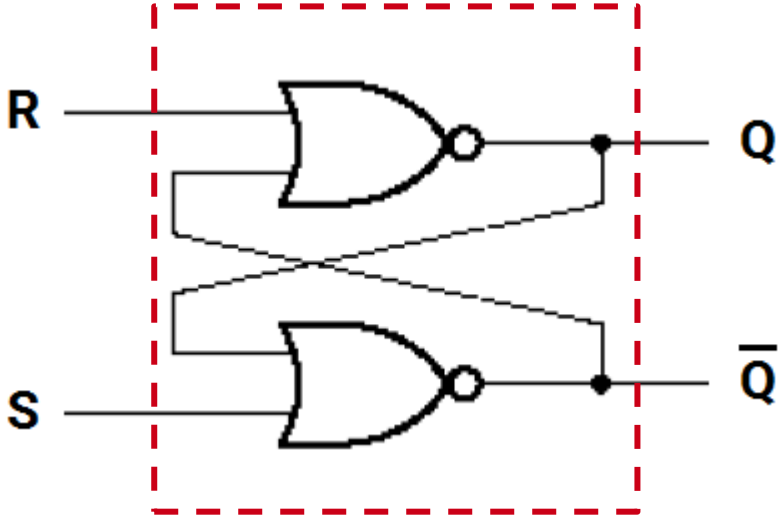


Utilizaremos bloques *process* para diseñarlos en VHDL

LATCH S-R (SET-RESET)

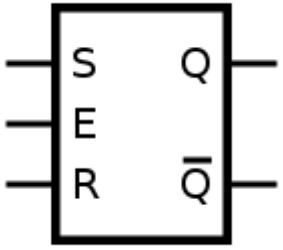


- La señal de set activa el estado a 1 y lo mantiene hasta que ocurre un reset

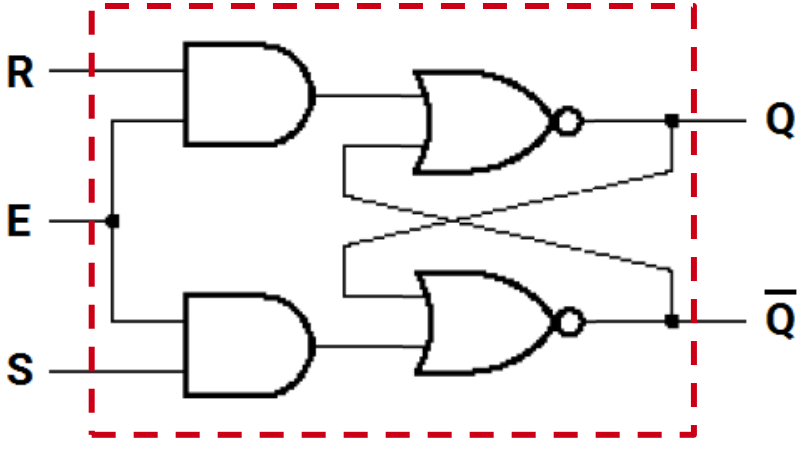


Set	Reset	Q	$\overline{Q}$
0	0	Q_{prev}	$\overline{Q}_{prev}$
0	1	**0**	**1**
1	0	**1**	**0**
1	1	**0**	**0**

LATCH S-R CON ENABLE



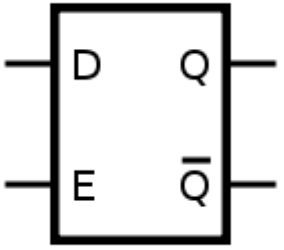
- La señal de *enable* activa o desactiva el componente



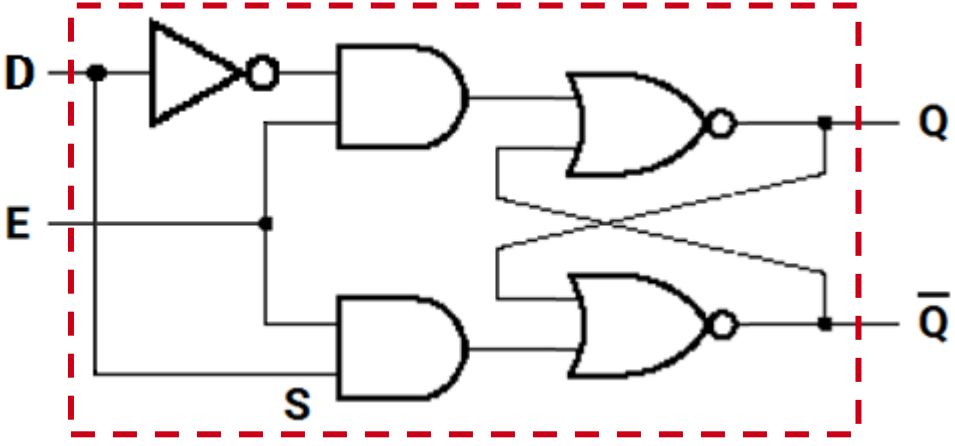
Enable	Set	Reset	Q	$\overline{Q}$
0	X	X	Q_{prev}	$\overline{Q}_{prev}$
1	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	**0**	**1**
1	1	0	**1**	**0**
1	1	1	**0**	**0**

El latch S-R tiene un problema de ambigüedad cuando S = R = 1

LATCH D



- Combina las entradas set y reset en una única señal de datos D

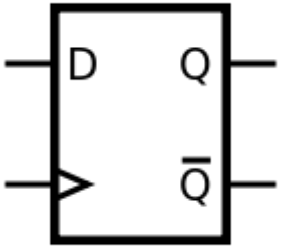


Enable	D	Q	$\overline{Q}$
0	X	Q_{prev}	$\overline{Q}_{prev}$
1	0	**0**	**1**
1	1	**1**	**0**

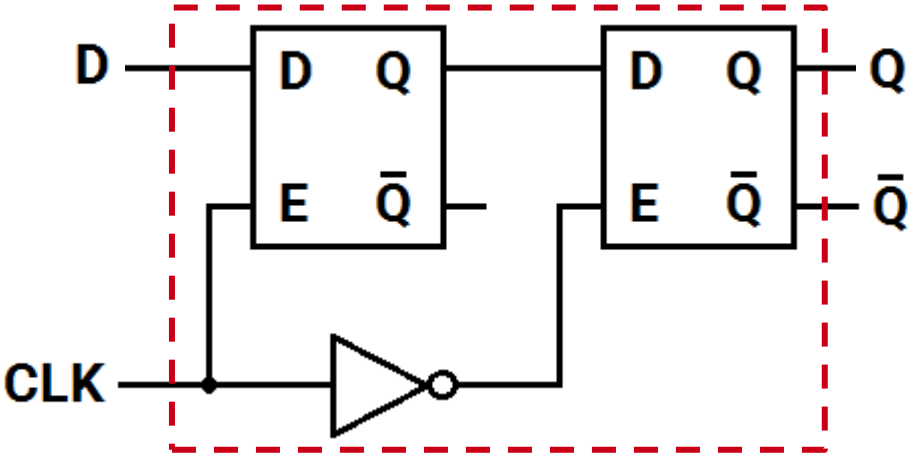
DISPARO POR FLANCO: FLIP-FLOPS

- La entrada de habilitación (*enable*) funciona por nivel. Si *enable* está a nivel alto se registran nuevas entradas en el latch, si está a nivel bajo se mantiene el estado
- ¿Y si sólo consideramos el momento de la transición de ‘0’ a ‘1’ o viceversa?
- En este caso la señal de habilitación será una señal de reloj (CLK)

FLIP-FLOP D

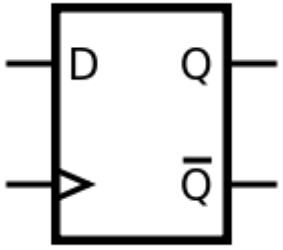


- Formado por dos latches D con señales de habilitación invertidas



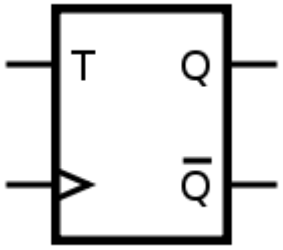
CLK	D	Q	$\overline{Q}$
↑	0	0	1
↑	1	1	0
0	X	Q_{prev}	$\overline{Q}_{prev}$
1	X	Q_{prev}	$\overline{Q}_{prev}$

FLIP-FLOP D



- Es el más usado en circuitos secuenciales para almacenar 1 bit de información
- Permite retrasar los datos de entrada 1 ciclo de reloj y sincronizar la información. Se pueden conectar varios en serie para retrasar más ciclos de reloj
- Si se conectan 8 en paralelo podemos obtener un registro de 8-bits
- También hay flip-flops D con señal de *enable*. En este caso, la señal de *enable* permite retener el dato de entrada hasta que se necesite y liberarlo en un determinado ciclo de reloj

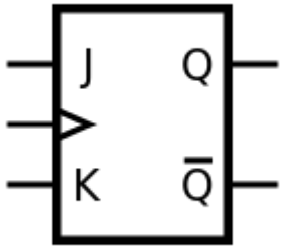
FLIP-FLOP T



- Cambia su estado mediante una entrada T (*toggle*)
- Si T = 1 cambia de estado, si T = 0 no cambia de estado
- También los hay con o sin señal de *enable*
- Muy utilizados como detectores de flanco, contadores o divisores de frecuencia

T	Q	$Q_{siguiente}$
0	0	0
0	1	1
1	0	1
1	1	0

FLIP-FLOP J-K



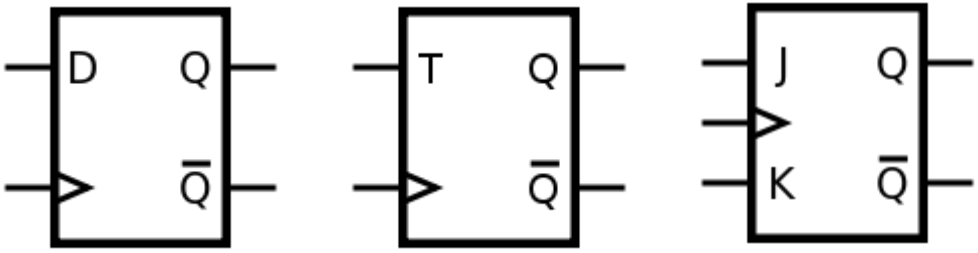
- Tiene dos entradas J y K que permiten cambiar el estado
- Si J = 0 y K = 1 se aplica un reset, si J = 1 y K = 0 un set
- Si J = 0 y K = 0 se mantiene el estado actual
- Si **J = 1 y K = 1 se invierte el estado actual**
- Si se mantiene la combinación J = K = 1, el flip-flop invertirá su contenido en cada ciclo de reloj. Esto se utiliza mucho en contadores binarios

J	K	Q
0	0	$\mathbf{Q_{prev}}$
0	1	0
1	0	1
1	1	$\mathbf{\overline{Q}_{prev}}$

ECUACIONES CARACTERÍSTICAS

- **Flip-flop D**: $Q_{siguiente} = D$
- **Flip-flop T**: $Q_{siguiente} = T \oplus Q$
- **Flip-flop J-K**: $Q_{siguiente} = J \cdot \overline{Q} + \overline{K} \cdot Q$

$Q_{siguiente}$ se representa como Q^*



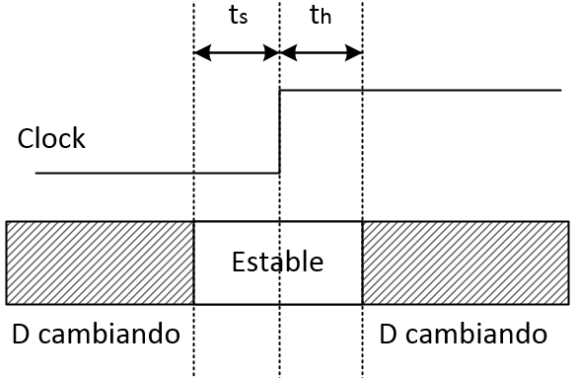
PROPIEDADES DE LOS FLIP-FLOPS

Para que un flip-flop tenga un comportamiento predecible la señal de entrada no debe violar los tiempos de establecimiento y de mantenimiento

- **Tiempo de establecimiento (t_s)**: la cantidad de tiempo que la señal de entrada debe estar estable antes de que llegue el flanco de subida del reloj
- **Tiempo de mantenimiento (t_h)**: la cantidad de tiempo que la señal de entrada debe estar estable después del flanco de subida del reloj

Si se viola alguno de estos tiempos, el biestable entra en un estado de **metaestabilidad**

¡¡¡LA SALIDA DEL BIESTABLE ES IMPREDECIBLE!!!



CONTENIDOS

1. Biestables. Tipos y propiedades
2. **Instrucciones secuenciales en VHDL: el *process***
3. Ejemplos en VHDL de diseños con bloques *process*
4. Simulación hardware con VHDL: el *testbench*

PROCESOS EN VHDL

- Construcción que nos permite ejecutar instrucciones de forma secuencial
- En el *process* las instrucciones se ejecutan una tras otra de arriba abajo
- Fuera del *process* las instrucciones se siguen ejecutando en paralelo
- Si hay varios *process*, se ejecutan en paralelo entre ellos

```
-- Ejemplo de process
process (lista_de_sensibilidad) is
        -- declaración de variables
begin
        -- instrucciones secuenciales
end process;
```

El process se ejecuta cuando alguno de los valores de la lista de sensibilidad cambia

OPERADORES EN VHDL

- **Operadores booleanos**: los operadores and, or, not, nand, nor, xor
- **Operadores aritméticos**: los operadores de suma (+), resta (-), multiplicación (*), división (/) módulo (mod), resto (rem), valor absoluto (abs) y exponencial (**)
- **Operadores de comparación**: mayor (>), menor (<), igual (=), distinto (/=), menor o igual que (<=) y mayor o igual que (>=)

OPERADORES EN VHDL

- **Desplazamiento**: VHDL tiene 6 funciones de desplazamiento predefinidas
  - Desplazamiento lógico hacia la izquierda (sll)
  - Desplazamiento lógico hacia la derecha (srl)
  - Desplazamiento aritmético hacia la izquierda (sla)
  - Desplazamiento aritmético hacia la derecha (sra)
  - Rotación a la izquierda (rol)
  - Rotación a la derecha (ror)

- **Concatenación**: existe una función de concatenación denotada por el símbolo &

```
A <= "1111"; B <= "000";
out1 <= A & B & '1';          -- out1 = 11110001
```

DECISIONES Y BUCLES EN VHDL

- **If-then-else:**

```
if (a = b) then
        out1 <= 1;
elsif (a > b) then
        out1 <= 2;
else
        out1 <= 0;
end if;
```

- **For:**

Vector de 8 bits 

```
signal a : std_logic_vector(7 downto 0);
for i in 0 to 7 loop
        a(i) <= '1';
end loop;
```

¡Los bucles infinitos no existen en VHDL porque no se pueden sintetizar!

DECISIONES Y BUCLES EN VHDL

- **Case-when:**

```
case a is
        when ‘0’ =>
            out1 <= 1;
        when ‘1’ =>
            out1 <= 2;
        when others =>
            out1 <= 0;
end case;
```

Cuando ninguna de las otras condiciones se cumple

- **Exit:**

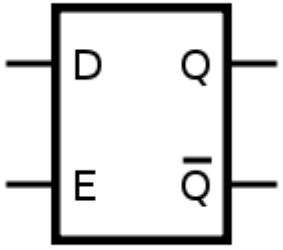
```
for i in 0 to 7 loop
        if (i = 4) then
            out1 <= ‘1’;
            exit;
        end if;
end loop;
```

Para salir de un bucle cuando no hace falta seguir en él

CONTENIDOS

1. Biestables. Tipos y propiedades
2. Instrucciones secuenciales en VHDL: el *process*
3. **Ejemplos en VHDL de diseños con bloques *process***
4. Simulación hardware con VHDL: el *testbench*

EJEMPLO 1: LATCH D

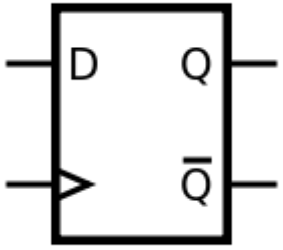


- **Funcionamiento**: el latch D mantiene el valor almacenado (estado) cuando la señal de *enable* = 0, y deja fluir el dato cuando *enable* = 1

```
entity latchD is
        port(  D : in std_logic;
               E : in std_logic;
               Q : out std_logic
        );
end latchD;

architecture Behavior of latchD is
begin

end Behavior;
```

```
process (D, E) begin
        if (E = '1') then
                Q <= D;
        end if;
end process;
```

EJEMPLO 2: FLIP-FLOP D

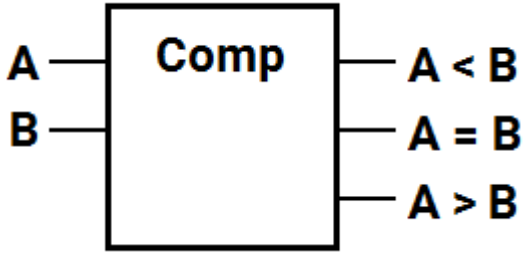


- **Funcionamiento**: el flip-flop D actualiza el valor almacenado (estado) cuando llega un flanco de subida del reloj

```
entity flipflopD is
        port(  CLK : in std_logic;
               D : in std_logic;
               Q : out std_logic
        );
end flipflopD;

architecture Behavior of flipflopD is
begin

end Behavior;
```

```
process (CLK) begin
        if (rising_edge(CLK)) then
                Q <= D;
        end if;
end process;
```

EJEMPLO 3: COMPARADOR

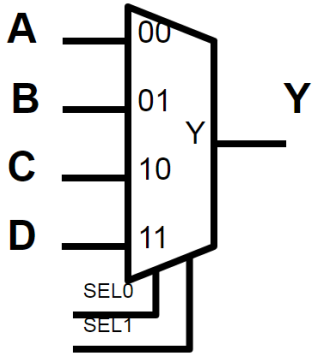



```
entity comparator is
        port( A, B : in   std_logic_vector(7 downto 0);
              ALB, AGB, AEB : out std_logic
        );
end comparator;


architecture Behavior of comparator is
begin
        process (A, B)
        begin

        end process;
end Behavior;
```

```
if (A < B) then
        ALB <= '1';
        AGB <= '0';
        AEB <= '0';
elsif (A > B) then
        ALB <= '0';
        AGB <= '1';
        AEB <= '0';
else
        ALB <= '0';
        AGB <= '0';
        AEB <= '1';
end if;
```

¡¡Los procesos también permiten describir circuitos combinacionales!!

EJEMPLO 4: MULTIPLEXOR 4:1




```
entity mux4_1 is
        port( A, B, C, D : in std_logic;
              SEL : in std_logic_vector(1 downto 0);
              Y : out std_logic
        );
end mux4_1;

architecture Behavior of mux4_1 is
begin
        process (A, B, C, D, SEL)
        begin

        end process;
end Behavior;
```

```
case SEL is
    when “00” =>
        Y <= A;
    when “01” =>
        Y <= B;
    when “10” =>
        Y <= C;
    when “11” =>
        Y <= D;
    when others =>
        Y <= ‘0’;
end case;
```

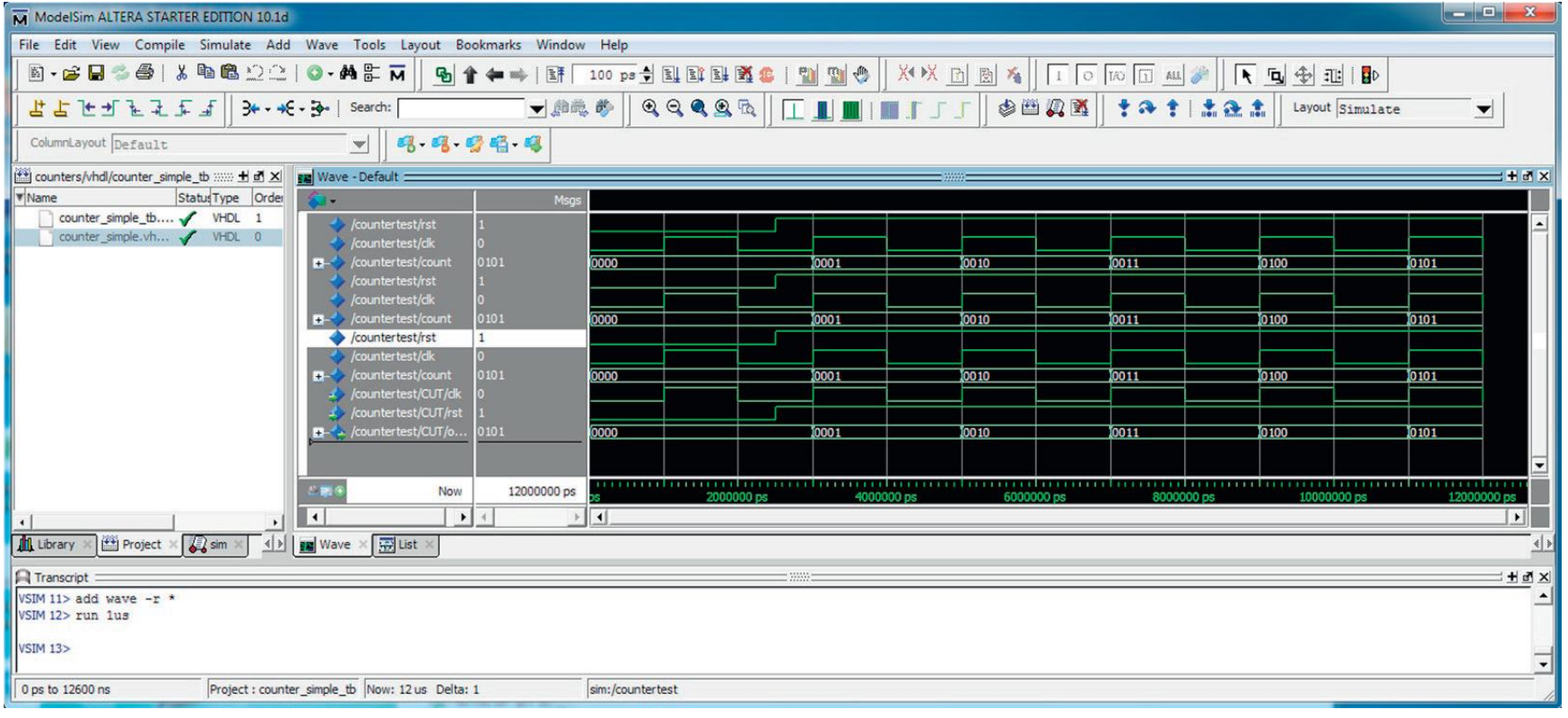
CONTENIDOS

1. Biestables. Tipos y propiedades
2. Instrucciones secuenciales en VHDL: el *process*
3. Ejemplos en VHDL de diseños con bloques *process*
4. **Simulación hardware con VHDL: el *testbench***

SIMULACIÓN DE DISEÑOS HARDWARE

- Antes de fabricar un ASIC o de implementar un circuito en una FPGA podemos asegurarnos de su correcto funcionamiento con una simulación
- Existen dos tipos de simulación:
  - **Simulación de comportamiento**: nos permite identificar errores de sintaxis o de conexión y verificar que el circuito se comporta según lo esperado
  - **Simulación post-síntesis**: nos permite comprobar cómo funcionaría nuestro diseño en un hardware específico. Permite verificar que los tiempos se cumplen
- La simulación se realiza con un entorno software como **ModelSim** o **Vivado**

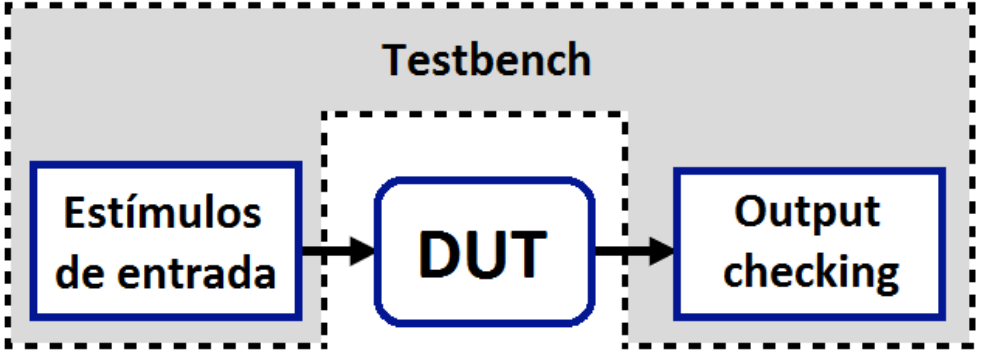
ENTORNO DE SIMULACIÓN: MODELSIM



Universidad Rey Juan Carlos

SIMULACIÓN HARDWARE: EL TESTBENCH

- Un ***testbench*** o “banco de pruebas” es un diseño en VHDL o Verilog que nos permite generar estímulos en nuestro diseño y observar su respuesta



DUT: *Design Under Test*, nuestro diseño a probar

¿Cómo es un testbench en VHDL?

EL TESTBENCH EN VHDL: EJEMPLO 1

- Vamos a simular el comportamiento de una **puerta lógica AND**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity andGate is
        port( A : in   std_logic;
              B : in   std_logic;
              Y : out std_logic
        );
end andGate;

architecture Dataflow of andGate is
begin
        Y <= A and B;
end Dataflow;
```

¿Qué estímulos debemos elegir?

Comportamiento esperado:

A	B	Y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

EL TESTBENCH EN VHDL: EJEMPLO 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testbench is
end testbench;

architecture Behavioral of testbench is
        -- Defino el componente
        component andGate is
                port ( A : in std_logic;
                       B : in std_logic;
                       Y : out std_logic
                );
        end component;
        -- Declaro las señales
        signal A, B, Y : std_logic;
begin
        -- Siguiente diapositiva
end Behavioral;
```

Definimos la librería

¡Un testbench no tiene entradas ni salidas!

Definimos el componente con la puerta AND

Declaramos todas las señales internas (no hay entradas ni de salidas así que también hay que definirlas)

EL TESTBENCH EN VHDL: EJEMPLO 1

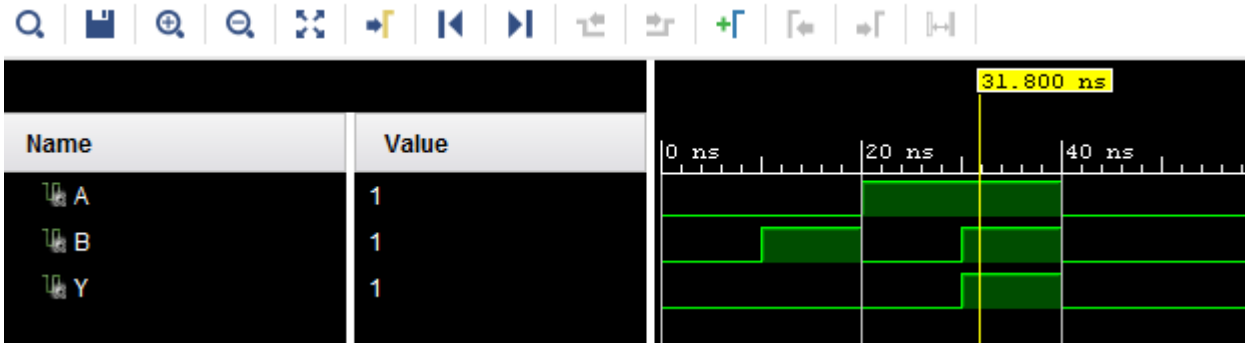
```
architecture Behavioral of testbench is
        -- Defino el componente
        component andGate is
                port ( A : in std_logic;
                       B : in std_logic;
                       Y : out std_logic
                );
        end component;
        -- Declaro las señales
        signal A, B, Y : std_logic;
begin
        -- Instancio el componente
        DUT : andGate port map(A, B, Y);

end Behavioral;
```

Instancio el *design under test* (DUT)

Un *process* sin lista de sensibilidad se dispara inmediatamente

```
-- Process con los estímulos
    process begin
        A <= '0'; B <= '0'; wait for 10ns;
        A <= '0'; B <= '1'; wait for 10ns;
        A <= '1'; B <= '0'; wait for 10ns;
        A <= '1'; B <= '1'; wait for 10ns;
        A <= '0'; B <= '0'; wait;
    end process;
```

EL TESTBENCH EN VHDL: EJEMPLO 1



- La instrucción ***wait for …*** nos permite parar la ejecución durante un tiempo determinado. Si no especificamos tiempo (***wait;***) el *process* se quedará parado indefinidamente en ese punto, evitando que continúe o se repita

- **Wait no es sintetizable**, sólo se usa en *testbenches* para realizar simulaciones

EL TESTBENCH EN VHDL: EJEMPLO 2

- Ahora vamos a simular un **contador** que cada 8 ciclos de reloj activa una salida

```
entity contador is
        port( CLK : in   std_logic;
              RST : in   std_logic;
              Y   : out std_logic
        );
end contador;

architecture Behavioral of contador is
        signal cuenta : integer range 0 to 7;
begin

end Behavioral;
```

```
process (CLK) begin
    if (rising_edge(CLK) then
        if (RST = '1') then
            cuenta <= 0;
            Y <= '0';
        else
            if (cuenta < 7) then
               cuenta <= cuenta + 1;
               Y <= '0';
            else
               cuenta <= 0;
               Y <= '1';
            end if;
        end if;
    end if;
end process;
```

EL TESTBENCH EN VHDL: EJEMPLO 2

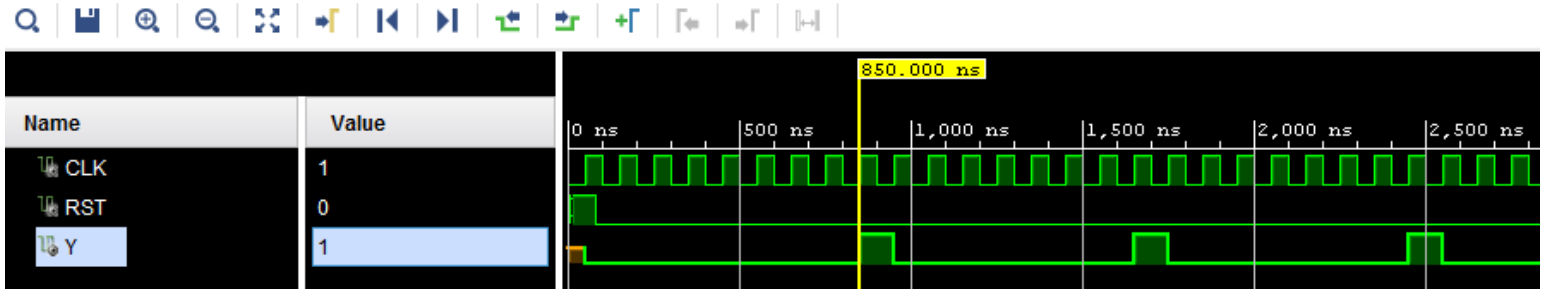
- Creamos el testbench:

```
entity tb_contador is
end tb_contador;

architecture Behavioral of tb_contador is
        component contador is
                port ( CLK : in std_logic;
                       RST : in std_logic;
                       Y   : out std_logic
                );
        end component;
        signal CLK, RST, Y : std_logic;
begin

        [ ]

end Behavioral;
```

```
DUT : contador port map(CLK, RST, Y);

-- Process para generar el reloj
process begin
        CLK <= '0'; wait for 50 ns;
        CLK <= '1'; wait for 50 ns;
end process;

-- Process para generar un pulso de RST
process begin
        RST <= '0'; wait for 10 ns;
        RST <= '1'; wait for 70 ns;
        RST <= '0'; wait;
end process;
```

EL TESTBENCH EN VHDL: EJEMPLO 2

- Resultado de la simulación:



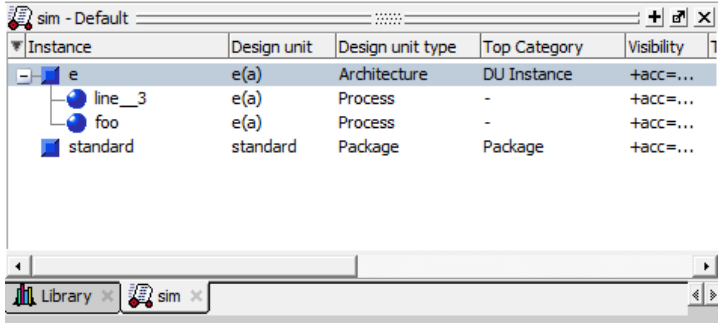
- Como vemos, la señal de salida (Y) se activa cada 8 ciclos de reloj

- También se puede observar que antes del reset la señal tiene un color naranja. Esto indica un valor indeterminado ya que hasta que no se produce el reset no se define el valor de la salida

SIMULACIÓN HARDWARE: ETIQUETAS

```
entity e is
end e;

architecture Behavioral of e is
…
begin
        -- Process sin etiqueta
        process begin
                A <= ‘0’; wait for 10 ns;
                A <= ‘1’; wait for 20 ns;
        end process;

        -- Process con etiqueta
        foo : process begin
                B <= ‘0’; wait for 10 ns;
                B <= ‘1’; wait for 20 ns;
        end process;
end Behavioral;
```

Etiqueta



- La etiqueta nos permite identificar el *process* en la simulación
- Se pueden utilizar para identificar otros bloques de código como if-else, for, etc.

Universidad Rey Juan Carlos

TECNOLOGÍA DE COMPUTADORES

Tema 8: Máquinas de Estados Finitos

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. Máquinas de estados finitos
2. Síntesis y análisis de máquinas de estados
3. Diseño de máquinas de estados con VHDL

CONTENIDOS

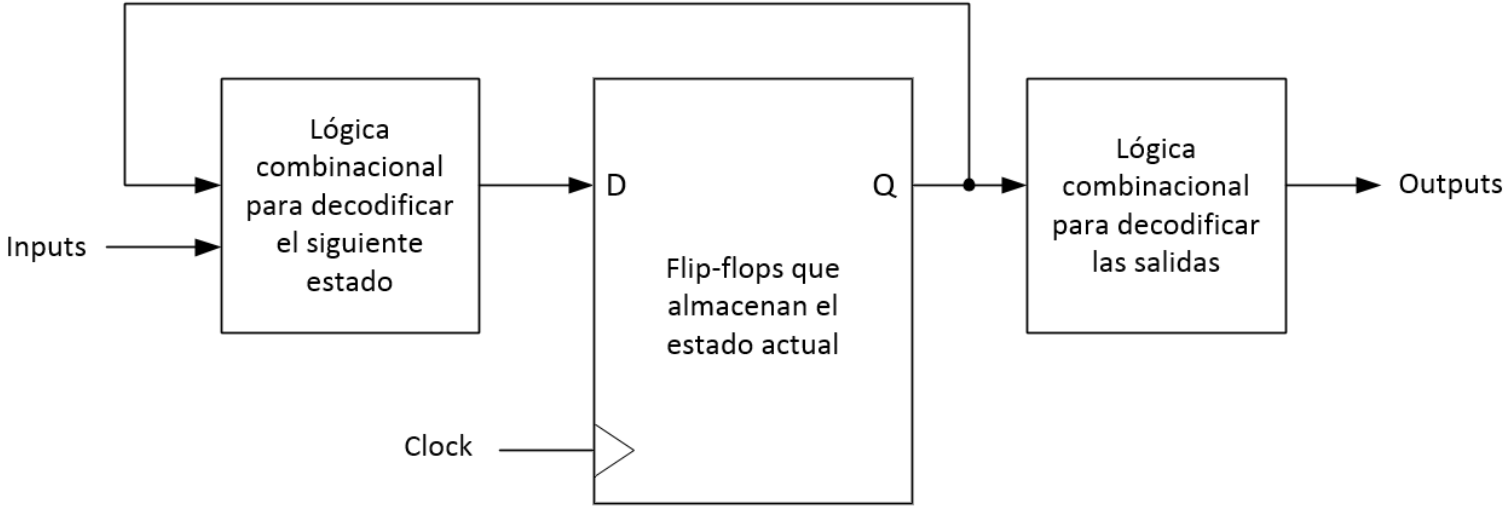
1. **Máquinas de estados finitos**
2. Síntesis y análisis de máquinas de estados
3. Diseño de máquinas de estados con VHDL

MÁQUINAS DE ESTADOS FINITOS

- Una **máquina de estados finitos** (FSM) es un circuito secuencial utilizado en gran variedad de situaciones:
  - Control de sistemas electrónicos automatizados
  - Gestión de protocolos de comunicaciones
  - Generación de patrones y detección de secuencias
- Habitualmente son circuitos síncronos controlados por una señal de reloj
- Los cambios de estado ocurren en los flancos de reloj
- Existen máquinas de estado de **Moore** y de **Mealy**

MÁQUINA DE MOORE

- Las **salidas sólo dependen del estado actual** de la máquina



MÁQUINA DE MEALY

- Las **salidas dependen del estado actual y del valor de las entradas**

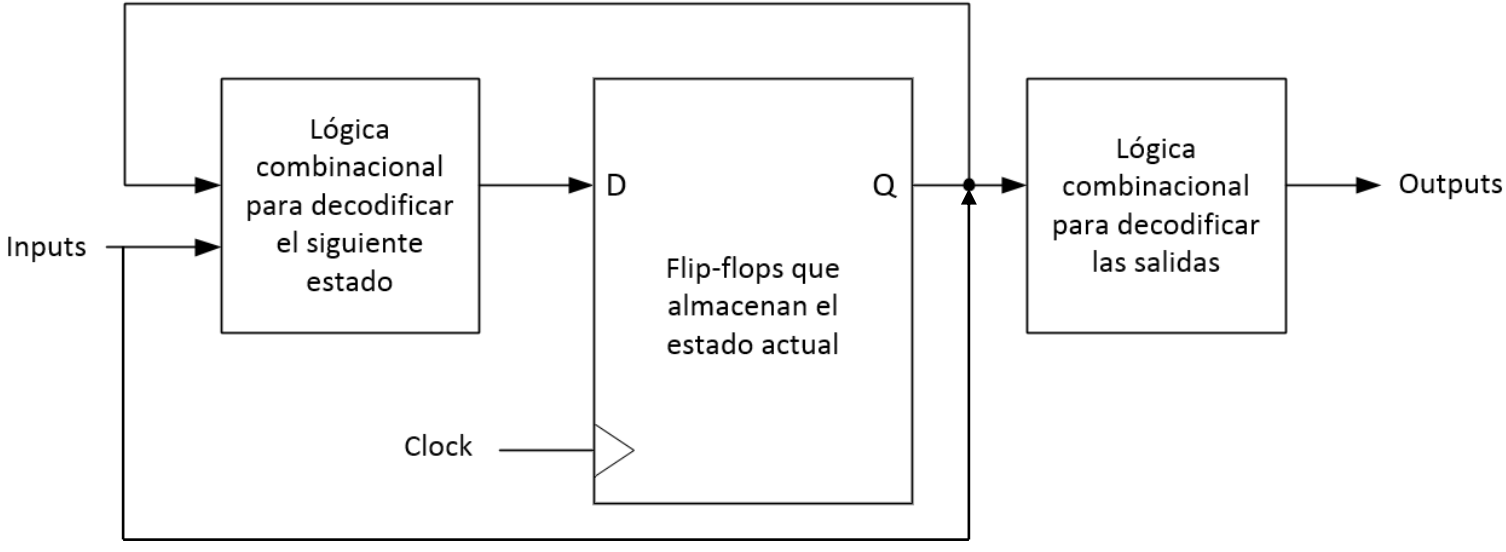


DIAGRAMA DE ESTADO

- Es la representación gráfica de una máquina de estados
- Consiste en un **diagrama de burbujas y flechas**
  - Las burbujas indican los estados
  - Las fechas indican las transiciones entre estados
- Las máquinas de Moore tienen diagramas distintos a las de Mealy

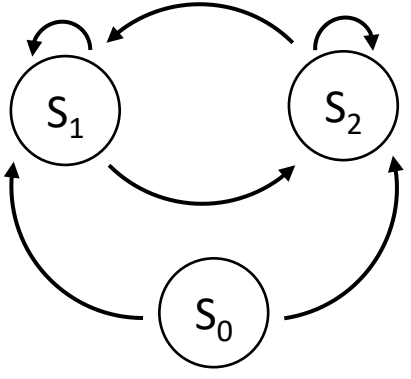


DIAGRAMA DE ESTADO DE MOORE

- Dentro de cada burbuja de estado se indica el valor de la salida
- En cada flecha de transición se indica la condición de entrada que hace transitar

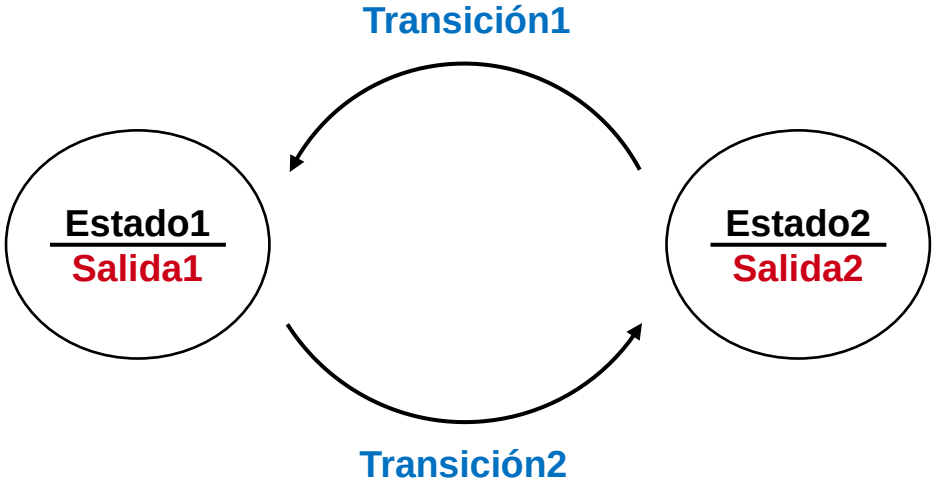
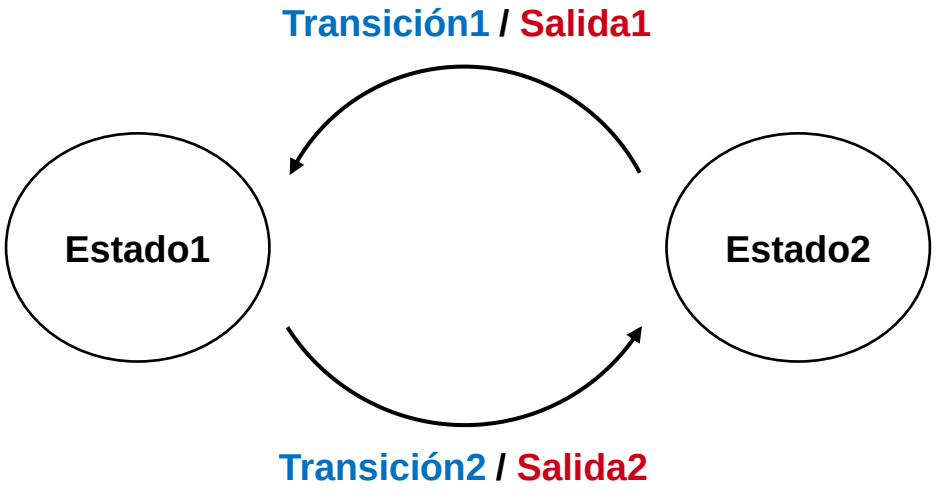


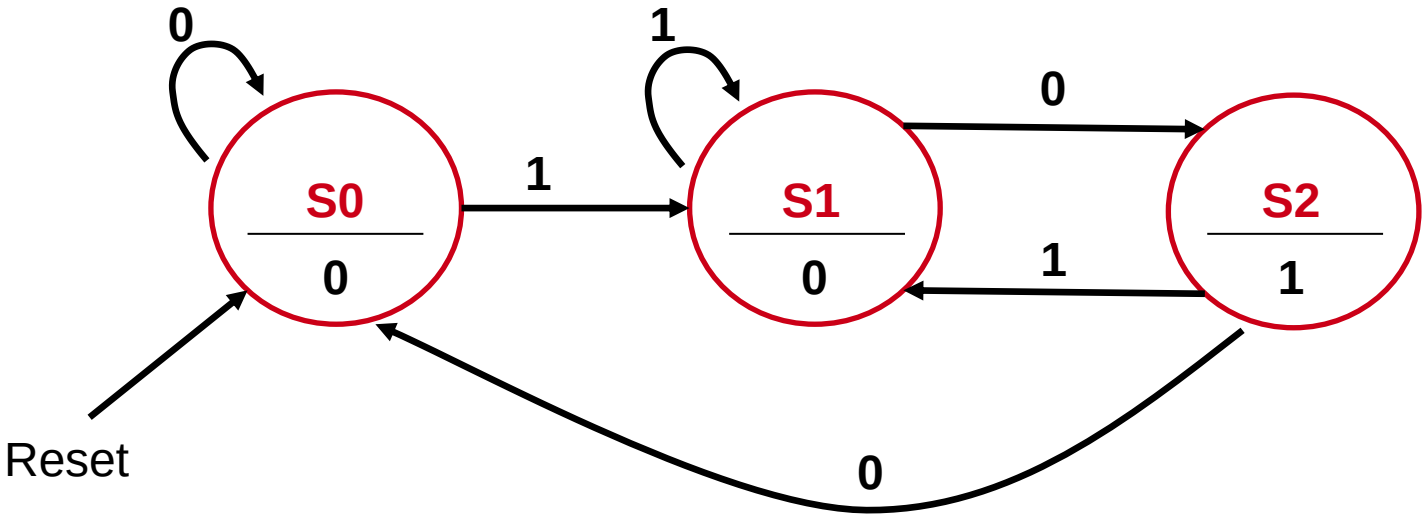
DIAGRAMA DE ESTADO DE MEALY

- En las burbujas únicamente se indica el nombre del estado
- En cada flecha de transición se indica la condición de entrada y el valor de salida



EJEMPLO: MÁQUINAS DE MOORE Y MEALY

- Realizar el diagrama de una máquina Moore que detecta la secuencia “10”:

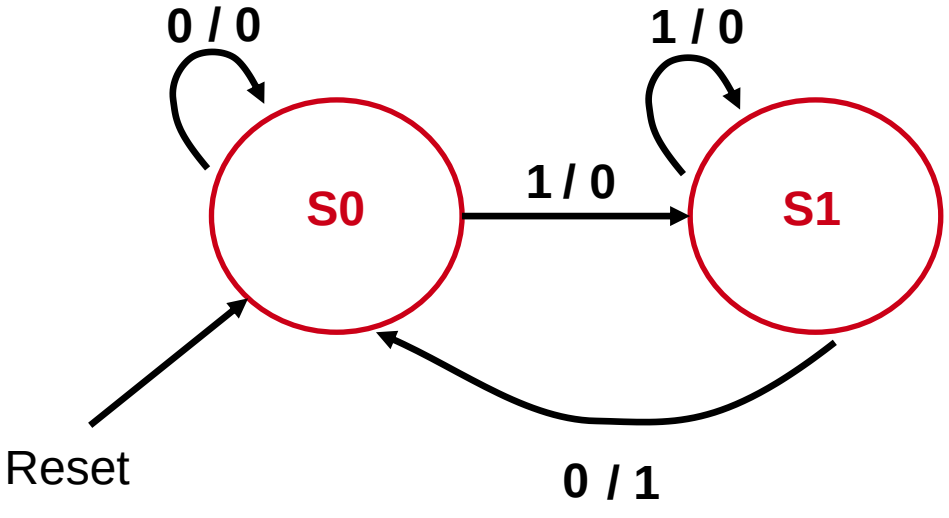


EJEMPLO: MÁQUINAS DE MOORE Y MEALY

- Realizar el diagrama de una máquina Moore que detecta la secuencia “10”:
- Tenemos los siguientes estados
  - **S0**: Estado inicial. Únicamente nos llegan ceros
  - **S1**: Siguiente estado. Aparece un 1
  - **S2**: Estado de detección. Aparece un 0 después de detectar un 1.
- La salida (secuencia detectada) es de 1 bit, y valdrá 1 cuando alcancemos el estado S2, y 0 en el resto de estados

EJEMPLO: MÁQUINAS DE MOORE Y MEALY

- Detectar ahora la secuencia “10” con una máquina Mealy:



EJEMPLO: MÁQUINAS DE MOORE Y MEALY

- Detectar ahora la secuencia “10” con una máquina Mealy:
- Tenemos los siguientes estados
  - **S0**: Estado inicial. Sólo nos llegan ceros
  - **S1**: Estado de detección. Se detecta la secuencia “10”
- La salida (secuencia detectada) es de 1 bit, y valdrá 1 cuando alcancemos el estado S1 y la entrada valga 0

COMPARATIVA MOORE VS. MEALY

- **Ambas máquinas de estado son funcionalmente equivalentes**
- La máquina de Moore es conceptualmente más sencilla de diseñar
- La máquina de Mealy normalmente requiere menos estados, por lo que el circuito resultante tendrá menos componentes y consumirá menos energía
- La máquina de Mealy calcula las salidas en cuanto se produce un cambio en las entradas, por lo que responde un ciclo de reloj antes. Sin embargo, la máquina de Moore genera señales de salida síncronas
- La máquina de Moore es más robusta frente a errores de tipo “glitch” producidos por rápidas variaciones en las señales de entrada

CONTENIDOS

1. Máquinas de estados finitos
2. **Síntesis y análisis de máquinas de estados**
3. Diseño de máquinas de estados con VHDL

SÍNTESIS DE MÁQUINAS DE ESTADOS

1. Identificar las señales de entrada y salida presentes en el sistema
2. Escoger un tipo de máquina de estados y dibujar el diagrama de estados*****
3. Construir la tabla de estados y escoger una combinación de estados
4. Crear la tabla de transición
5. Escoger un tipo de flip-flop y construir la tabla de excitación
6. Obtener las ecuaciones de excitación y de salida usando Karnaugh*****
7. Dibujar el circuito lógico*****

****** *El VHDL se puede escribir en cualquiera de estos puntos*

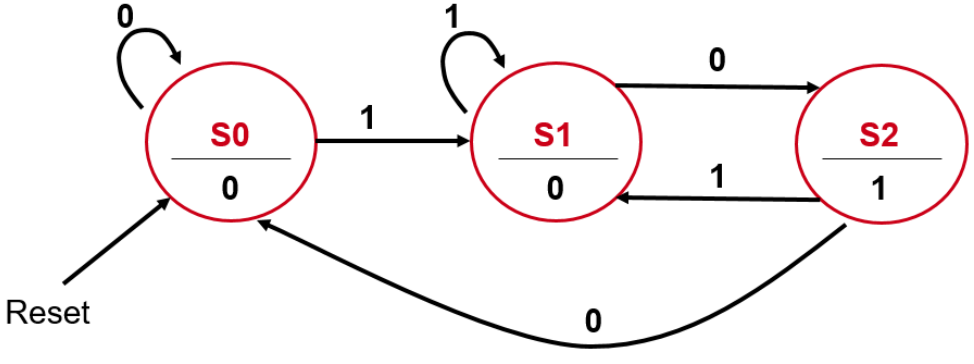
EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

- Diseñar una máquina de estados que detecte la secuencia “10”:

1. Identificar las señales de entrada y salida presentes en el sistema

   Entrada: X
   Salida: Z

2. Escoger una máquina de estados y dibujar el diagrama de estados

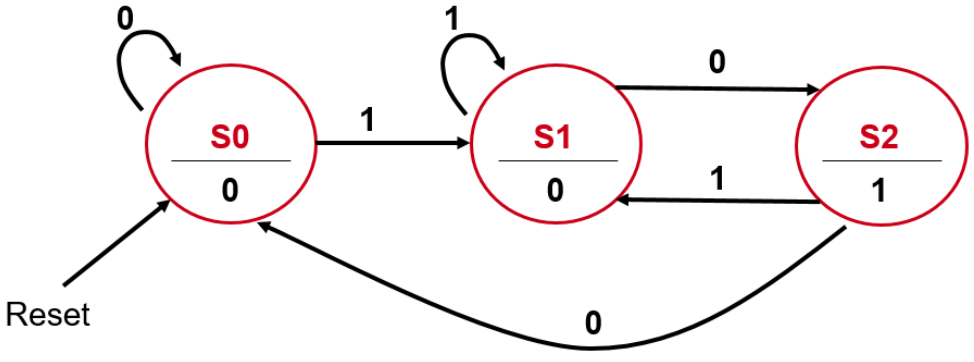


Elegimos Moore

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

3. Construir la tabla de estados y escoger una combinación de estados

ESTADO	X=0	X=1
S0	**S0**	**S1**
S1	**S2**	**S1**
S2 / 1	**S0**	**S1**



EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

3. Construir la tabla de estados y escoger una combinación de estados

ESTADO	X=0	X=1
S0	**S0**	**S1**
S1	**S2**	**S1**
S2 / 1	**S0**	**S1**

Combinación de estados

S0: 00
S1: 01
S2: 10

$(Q_1 Q_0)$

Nos indica el número de flip-flops necesarios para diseñar la máquina

¡La combinación “11” no existe!

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

4. Crear la tabla de transición

ESTADO	X=0	X=1
S0	**S0**	**S1**
S1	**S2**	**S1**
S2 / 1	**S0**	**S1**

S0: 00
S1: 01
S2: 10

Entrada	Estado actual		Estado siguiente		Salida
X	Q_1	Q_0	Q_1^*	Q_0^*	Z
0	0	0	**0**	**0**	**0**
0	0	1	**1**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**X**	**X**	**X**
1	0	0	**0**	**1**	**0**
1	0	1	**0**	**1**	**0**
1	1	0	**0**	**1**	**1**
1	1	1	**X**	**X**	**X**

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

5. Escoger un tipo de flip-flop y construir la tabla de excitación

Escogemos flip-flop D:

$D_0 = Q_0^*$
$D_1 = Q_1^*$

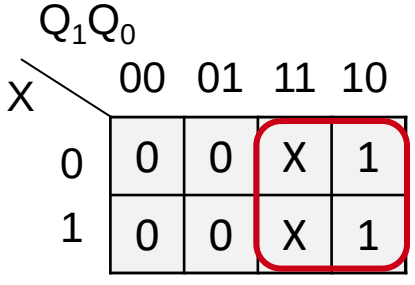
¡Recordad su ecuación característica!

Entrada	Estado actual		Excitación FFs		Salida
X	Q_1	Q_0	D_1	D_0	Z
0	0	0	**0**	**0**	**0**
0	0	1	**1**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**X**	**X**	**X**
1	0	0	**0**	**1**	**0**
1	0	1	**0**	**1**	**0**
1	1	0	**0**	**1**	**1**
1	1	1	**X**	**X**	**X**

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

6. Obtener las ecuaciones de excitación y de salida usando Karnaugh

Z



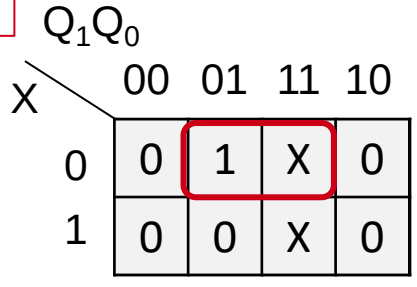
$$Z = Q_1$$

Entrada	Estado actual		Excitación FFs		Salida
X	Q_1	Q_0	D_1	D_0	Z
0	0	0	**0**	**0**	**0**
0	0	1	**1**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**X**	**X**	**X**
1	0	0	**0**	**1**	**0**
1	0	1	**0**	**1**	**0**
1	1	0	**0**	**1**	**1**
1	1	1	**X**	**X**	**X**

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

6. Obtener las ecuaciones de excitación y de salida usando Karnaugh

D_1

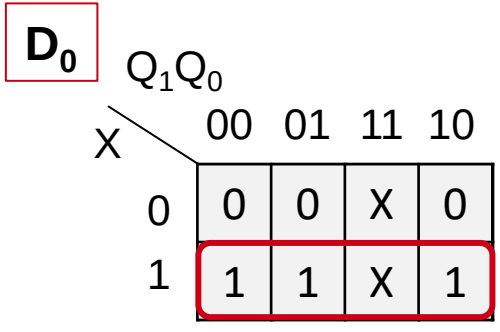


$$D_1 = X' \cdot Q_0$$

Entrada	Estado actual		Excitación FFs		Salida
X	Q_1	Q_0	D_1	D_0	Z
0	0	0	**0**	**0**	**0**
0	0	1	**1**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**X**	**X**	**X**
1	0	0	**0**	**1**	**0**
1	0	1	**0**	**1**	**0**
1	1	0	**0**	**1**	**1**
1	1	1	**X**	**X**	**X**

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

6. Obtener las ecuaciones de excitación y de salida usando Karnaugh



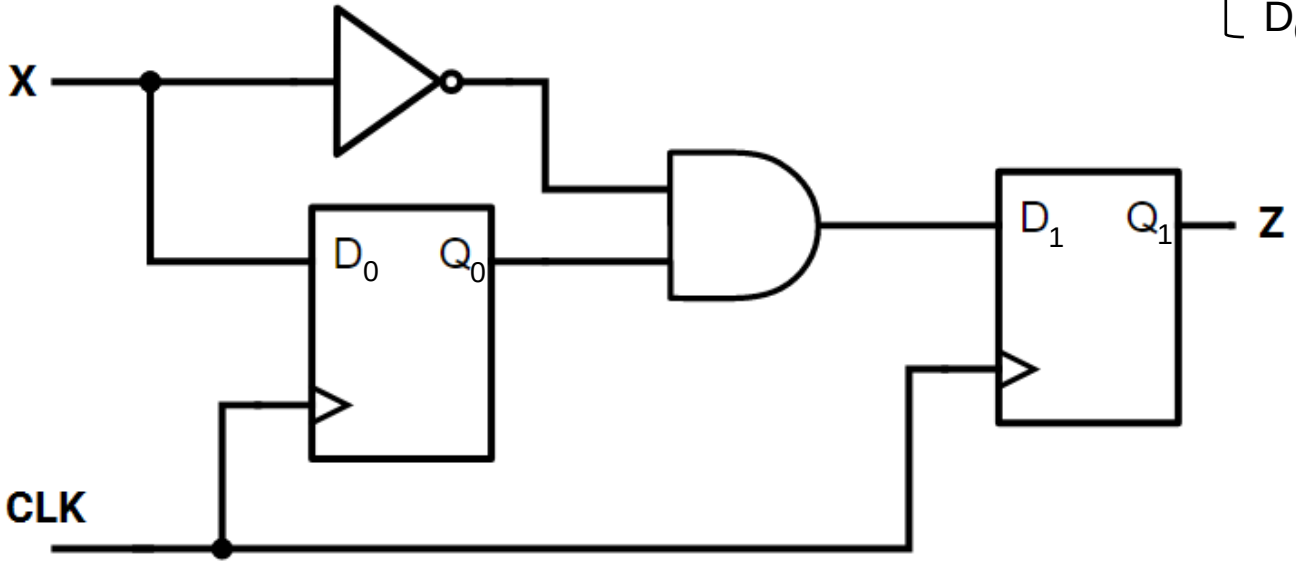
$D_0 = X$

Entrada	Estado actual		Excitación FFs		Salida
X	Q_1	Q_0	D_1	D_0	Z
0	0	0	**0**	**0**	**0**
0	0	1	**1**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**X**	**X**	**X**
1	0	0	**0**	**1**	**0**
1	0	1	**0**	**1**	**0**
1	1	0	**0**	**1**	**1**
1	1	1	**X**	**X**	**X**

EJEMPLO: SÍNTESIS CON MÁQUINA MOORE

7. Dibujar el circuito lógico

$$\begin{cases} Z = Q_1 \\ D_1 = X' \cdot Q_0 \\ D_0 = X \end{cases}$$



ANÁLISIS DE MÁQUINAS DE ESTADOS

- Consiste en obtener el diagrama de estados de un circuito realizando los pasos en sentido inverso

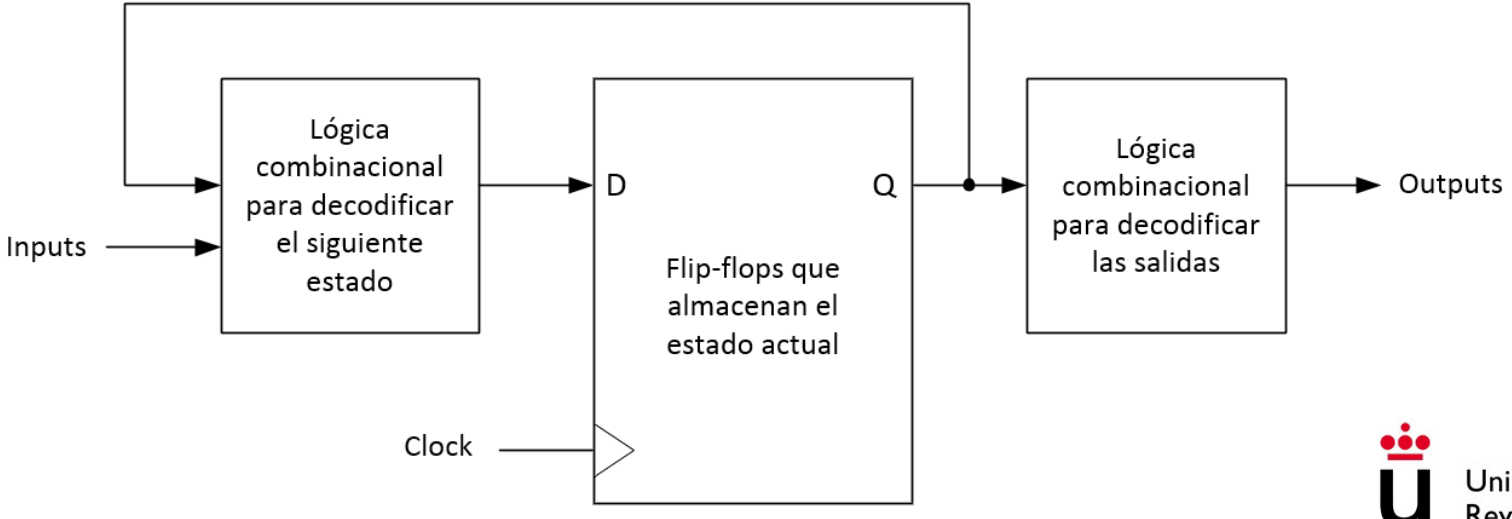
6. Dibujar el diagrama de estados
5. Escoger una combinación de estados y construir la tabla de estados
4. Crear la tabla de transición
3. Construir la tabla de excitación sabiendo el tipo de flip-flop usado
2. Obtener las ecuaciones de excitación y de salida directamente del circuito
1. Identificar entradas y salidas así como la máquina de estados dibujada

CONTENIDOS

1. Máquinas de estados finitos
2. Síntesis y análisis de máquinas de estados
3. **Diseño de máquinas de estados con VHDL**

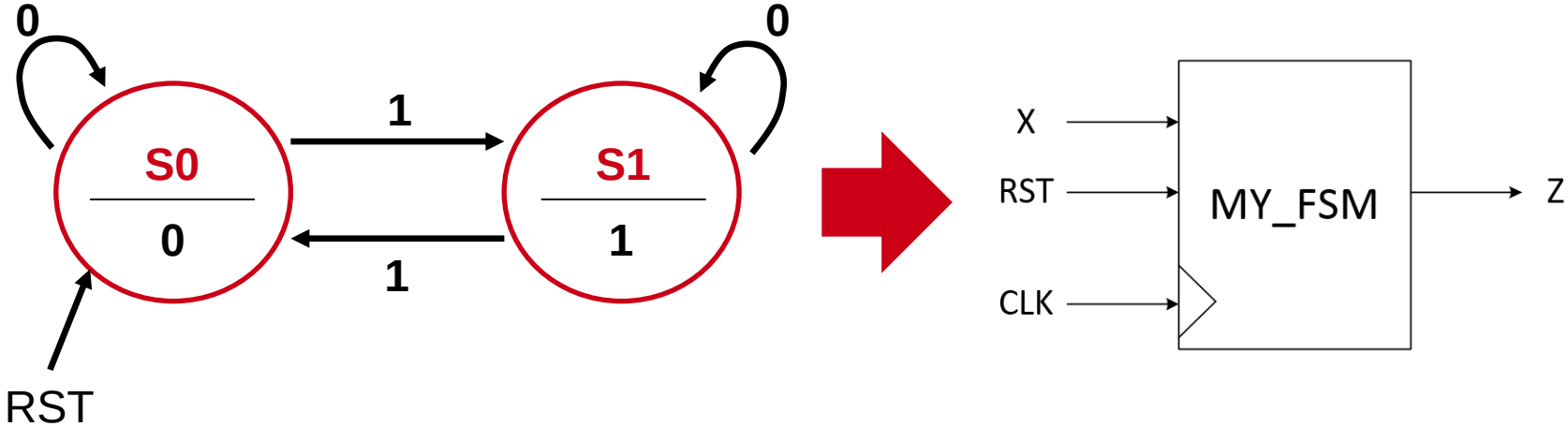
PARTES DE UNA MÁQUINA DE ESTADOS

- **Decodificador del estado siguiente**: utiliza el estado actual y el valor de las entradas para cambiar el estado de la máquina (ecuaciones de excitación)
- **Registro de estados**: conjunto de flip-flops que almacenan el estado actual
- **Decodificador de salida**: utilizado para generar la salida deseada



EJEMPLO: MÁQUINA DE MOORE EN VHDL

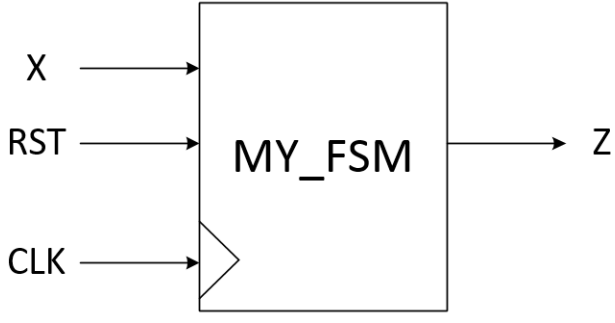
- Dado el siguiente diagrama de estados de una máquina de Moore con una señal de entrada de un bit (X) y una señal de salida de un bit (Z), obtener su código VHDL



EJEMPLO: MÁQUINA DE MOORE EN VHDL

- Empezamos definiendo la librería y la entidad:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MY_FSM is
        port( CLK : in   std_logic;
              RST : in   std_logic;
              X   : in   std_logic;
              Z   : out std_logic
        );
end MY_FSM;
```



EJEMPLO: MÁQUINA DE MOORE EN VHDL

```
architecture Behavior of MY_FSM is
        -- Declaración de variables de estado
begin
        -- Proceso secuencial síncrono para
        controlar los flip-flop

        -- Proceso combinacional case-when para
        controlar los cambios de estado y la salida
end Behavior;
```

Utilizo **type** para definir los estados en VHDL

```
type estados is (S0, S1);
signal current_state, next_state : estados;
```

EJEMPLO: MÁQUINA DE MOORE EN VHDL

```
architecture Behavior of MY_FSM is
        type estados is (S0, S1);
        signal current_state, next_state : estados;
begin
        -- Proceso secuencial síncrono para
        controlar los flip-flop

        -- Proceso combinacional case-when para
        controlar los cambios de estado y la salida
end Behavior;
```

Utilizo **flip-flops D** para el cambio de estado

```
process(CLK) begin
        if (rising_edge(CLK)) then
                if (RST = '1') then
                        current_state <= S0;
                else
                        current_state <= next_state;
                end if;
        end if;
end process;
```

EJEMPLO: MÁQUINA DE MOORE EN VHDL

```
architecture Behavior of MY_FSM is
        type estados is (S0, S1);
        signal current_state, next_state : estados;
begin
        -- Proceso secuencial
        process(CLK) begin
                if (rising_edge(CLK)) then
                        if (RST = '1') then
                                current_state <= S0;
                        else
                                current_state <= next_state;
                        end if;
                end if;
        end process;

        -- Proceso combinacional case-when para
        controlar los cambios de estado y la salida
end Behavior;
```

```
process(current_state, X) begin
        case current_state is
            when S0 =>
                Z <= '0';      -- Salida de Moore
                if (X = '1') then next_state <= S1;
                else next_state <= S0;
                end if;
            when S1 =>
                Z <= '1';      -- Salida de Moore
                if (X = '1') then next_state <= S0;
                else next_state <= S1;
                end if;
            -- Condición de control
            when others =>
                Z <= '0';
                next_state <= S0;
        end case;
end process;
```

TECNOLOGÍA DE COMPUTADORES

Tema 9: Módulos Secuenciales Básicos

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

CONTENIDOS

1. Registros
2. Contadores

CONTENIDOS

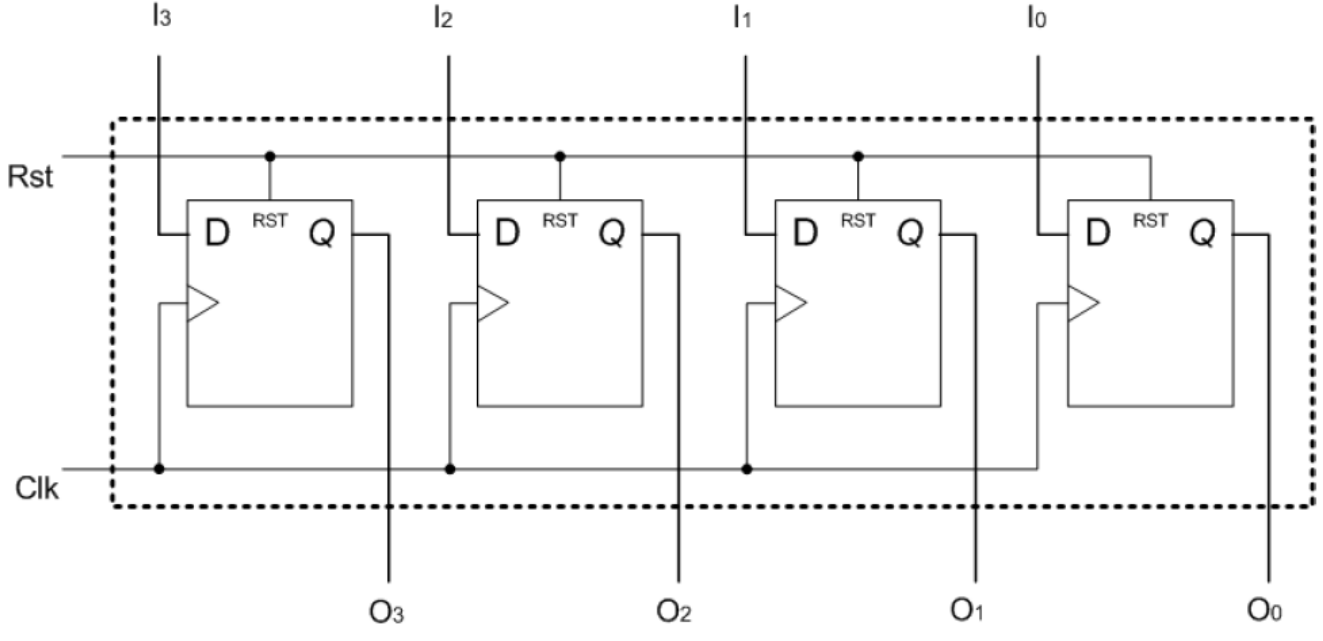
1. **Registros**
2. Contadores

REGISTROS

- Se denomina registro al **conjunto de flip-flops** que funcionan al unísono, compartiendo las señales de control
- Las entradas y salidas del registro pueden ser en paralelo o en serie
- Señales de control típicas:
  - Señal de reloj
  - Señal de clear o reset: entrada asíncrona que carga un ‘0’ en todos los flip-flops
  - Señal de preset o set: entrada asíncrona que carga un ‘1’ en todos los flip-flops
  - Señal de habilitación de entrada / reloj: permite el registro de nuevos datos de entrada
  - Señal de habilitación de salida: conecta/desconecta la salida del registro

REGISTRO PARALELO / PARALELO

- Muy comunes. Utilizados para almacenar datos



REGISTRO PARALELO / PARALELO EN VHDL

```
entity registro8bits is
        port(  CLK, RST : in std_logic;
               D : in std_logic_vector(7 downto 0);
               Q : out std_logic_vector(7 downto 0)
        );
end registro8bits;

architecture Behavior of registro8bits is
begin

end Behavior;
```

```
process (CLK) begin
        if (rising_edge(CLK)) then
            if (RST = '1') then
                Q <= (others => '0');
            else
                Q <= D;
        end if;
end process;
```

Es equivalente a
Q <= “00000000”;

REGISTRO PARALELO / PARALELO EN VHDL

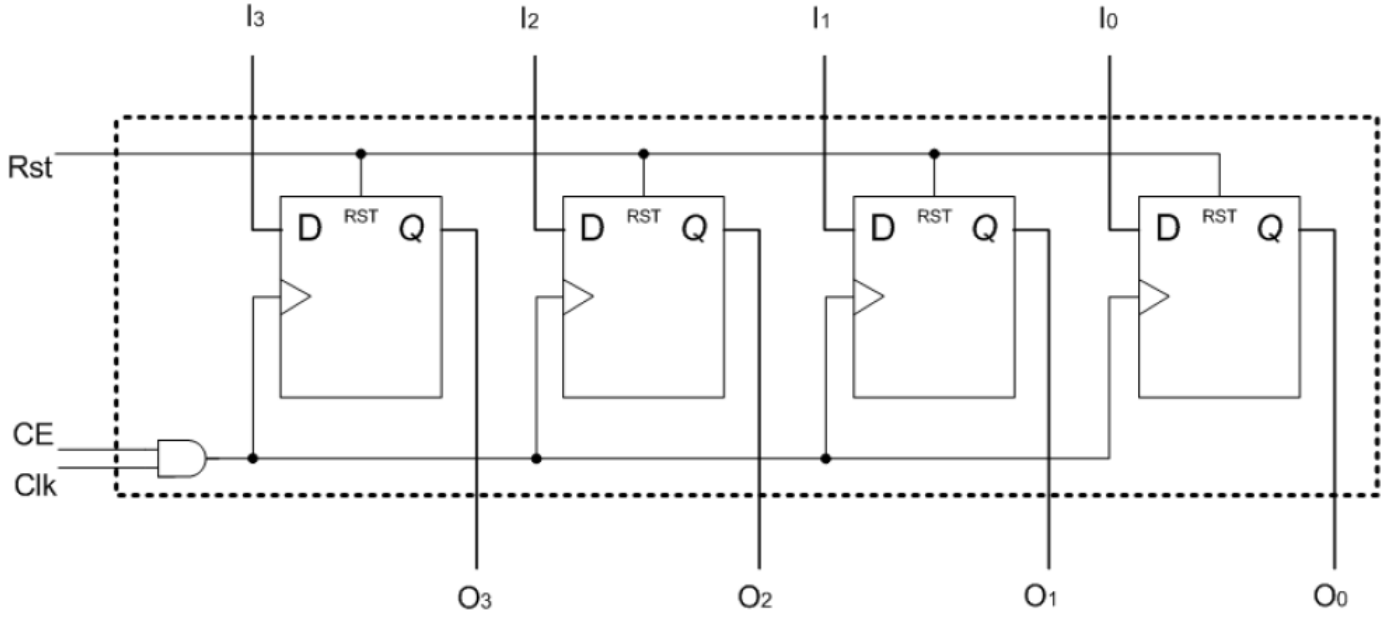
```
entity registro8bits is
        port(  CLK, RST : in std_logic;
               D : in std_logic_vector(7 downto 0);
               Q : out std_logic_vector(7 downto 0)
        );
end registro8bits;

architecture Behavior of registro8bits is
begin

end Behavior;
```

El reset puede ser asíncrono

```
process (CLK, RST) begin
        if (RST = '1') then
                Q <= (others => '0');
        elsif (rising_edge(CLK)) then
                Q <= D;
        end if;
end process;
```

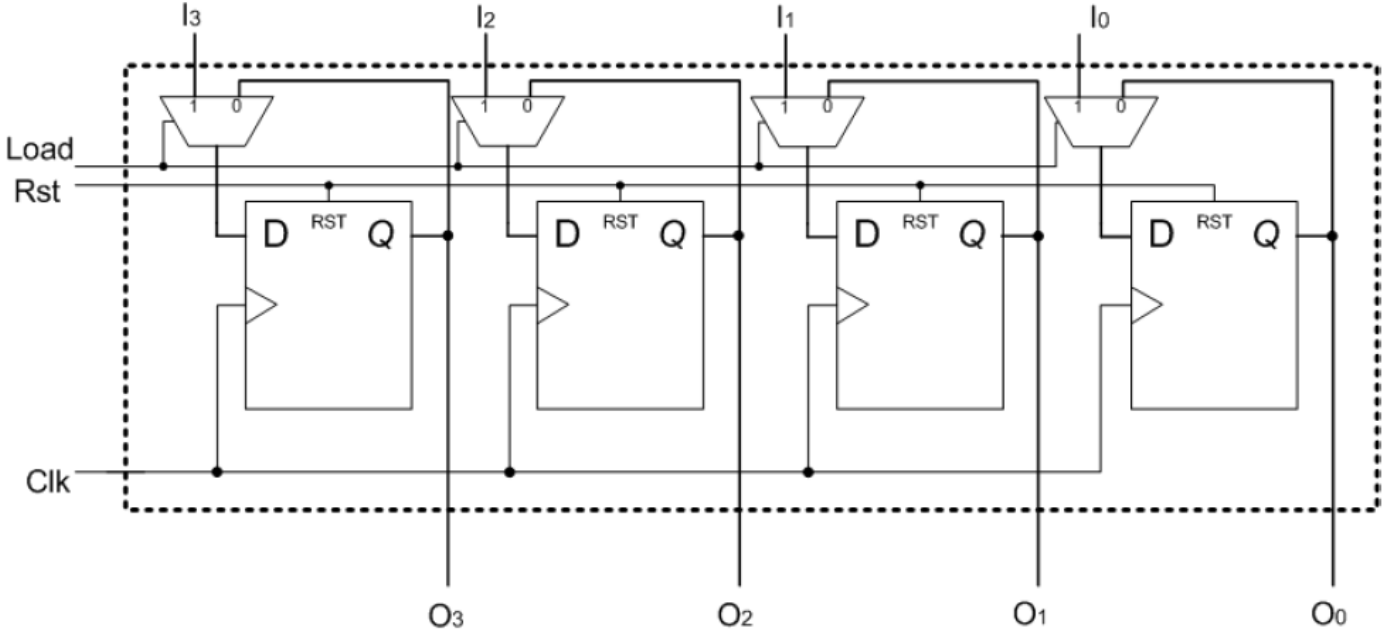
REGISTRO PARALELO / PARALELO

- Se pueden diseñar con habilitación de reloj (**CE**)



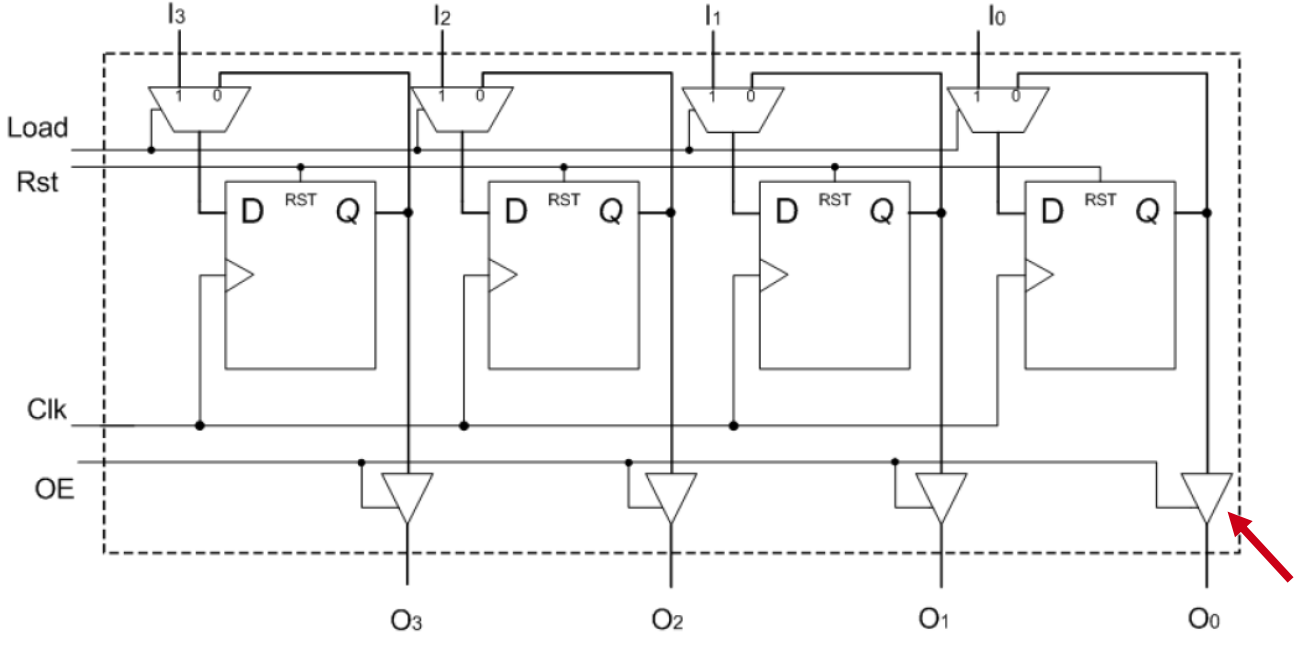
REGISTRO PARALELO / PARALELO

- Con habilitación de entrada (**Load**)



REGISTRO PARALELO / PARALELO

- Y con habilitación de salida (**OE**)



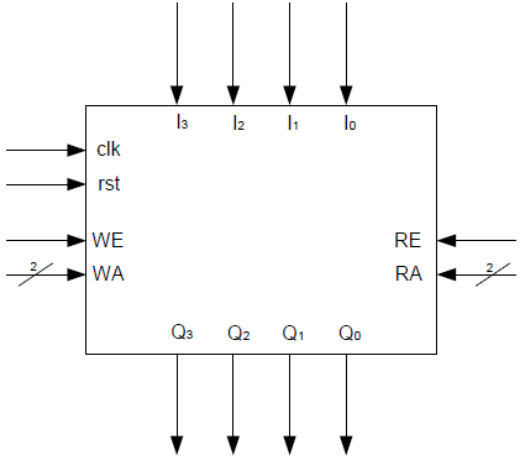
REGISTRO PARALELO / PARALELO EN VHDL

```
entity registro8bits is
        port(  CLK, RST, LOAD, OE : in std_logic;
               D : in std_logic_vector(7 downto 0);
               Q : out std_logic_vector(7 downto 0)
        );
end registro8bits;

architecture Behavior of registro8bits is
        signal Q_aux : std_logic_vector(7 downto 0);
begin

end Behavior;
```

```
-- Escritura
process (CLK, RST) begin
        if (RST = '1') then
                Q_aux <= (others => '0');
        elsif (rising_edge(CLK) and LOAD = '1') then
                Q_aux <= D;
        end if;
end process;

-- Lectura
process (Q_aux, OE) begin
        if (OE = '1') then Q <= Q_aux;
        else   Q <= (others => 'Z');
        end if;
end process;
```

BANCO DE REGISTROS

- **Conjunto de registros** que comparten las líneas de entrada y las líneas de salida
- Se selecciona el registro que se lee o en el que se escribe mediante unos **decodificadores** de lectura y escritura
- Señales típicas:
  - WE (write enable): habilita la escritura
  - WA (write address): dirección de escritura
  - RE (read enable): habilita la lectura
  - RA (read address): dirección de lectura



REGISTROS DE DESPLAZAMIENTO

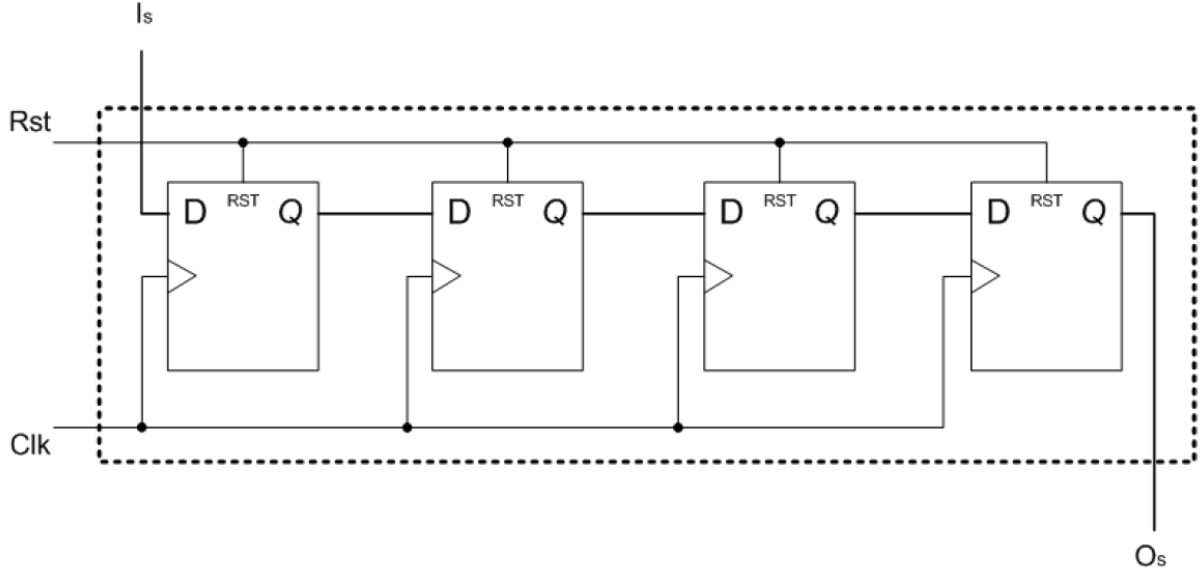
- Conjunto de flip-flops utilizados para almacenar y transferir datos
- Los bits de datos se desplazan internamente entre flip-flops
- Existen tres configuraciones:
  - Serie-serie
  - Serie-paralelo
  - Paralelo-serie

La configuración paralelo-paralelo (vista anteriormente) no desplaza los bits

- Utilizados para añadir retardo entre la entrada y la salida y en interfaces que convierten datos entre serie y paralelo (ej. UART)

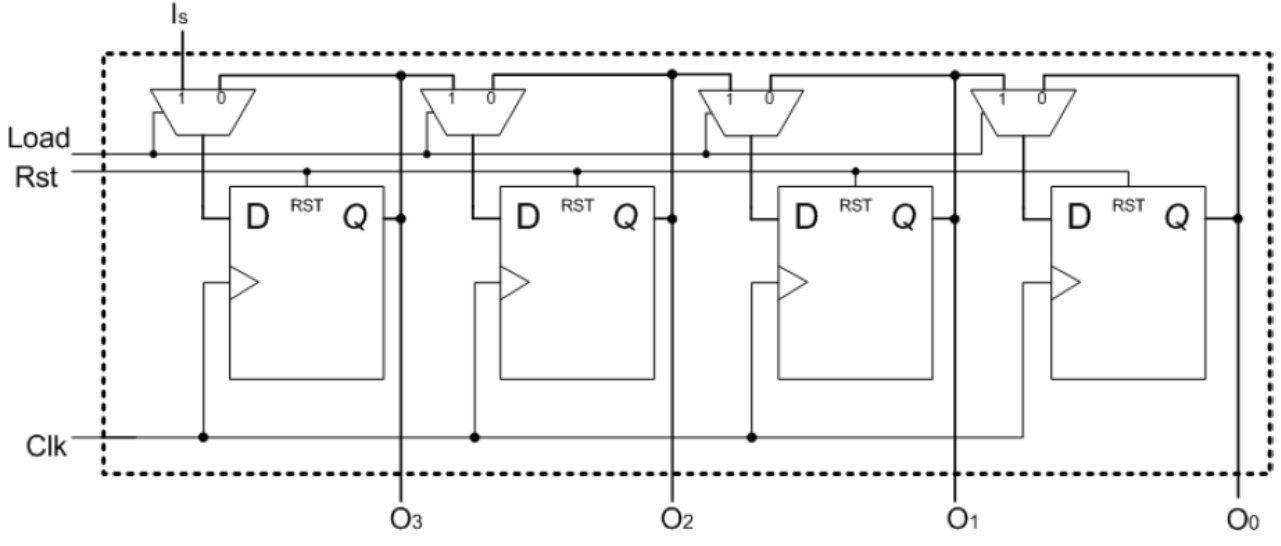
DESPLAZAMIENTO SERIE-SERIE

- La entrada y salida son de 1 bit. El dato de entrada se va propagando de un biestable a otro en cada flanco de reloj hasta llegar a la salida
- La salida es igual a la entrada retrasada *N* ciclos de reloj (*N = nº biestables*)



DESPLAZAMIENTO SERIE-PARALELO

- La entrada es de 1 bit y la salida de N bits. Se producen tantos bits a la salida como flip-flops internos haya



CONTENIDOS

1. Registros
2. **Contadores**

CONTADORES

- Circuito secuencial que genera una secuencia numérica concreta de salida
- Pueden tener señales de precarga (load), de habilitación y de control de cuenta ascendente o descendente (up/down)
- Pueden ser síncronos o asíncronos
- Suelen tener retardo de propagación

DISEÑO DE CONTADORES SÍNCRONOS

- Se pueden diseñar como una máquina de estados en la que cada estado representa un valor de la cuenta
- Se transita de un estado a otro en cada ciclo de reloj o según una señal de cuenta ascendente / descendente

DISEÑO DE CONTADORES EN VHDL

- Además de como una máquina de estados, se pueden diseñar describiendo el comportamiento del contador mediante procesos

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity contador8bits is
        port(  CLK, RST, LD, UP : in std_logic;
               DIN : in std_logic_vector(7 downto 0);
               COUNT : out std_logic_vector(7 downto 0)
        );
end contador8bits;
```

Añadimos la librería *numeric_std* para poder incrementar la cuenta

DISEÑO DE CONTADORES EN VHDL

```
architecture Behavior of contador8bits is
        signal CNT_aux : unsigned(7 downto 0);          -- Señal auxiliar para la cuenta interna
begin
        process (CLK, RST) begin
                if (RST = '1') then CNT_aux <= (others => '0');
                elsif (rising_edge(CLK)) then
                        if (LD = '1') then CNT_aux <= unsigned(DIN);
                        else
                                if (UP = '1') then CNT_aux <= CNT_aux + 1;
                                else CNT_aux <= CNT_aux – 1;
                                end if;
                        end if;
                end if;
        end process;
        COUNT <= std_logic_vector(CNT_aux);          -- Convierto a std_logic_vector
end Behavior;
```

TECNOLOGÍA DE COMPUTADORES

Tema 10: Estructura de un Computador

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos

DISEÑO DE MICROARQUITECTURAS

- En temas anteriores se han sentado las bases de la electrónica digital y se han visto circuitos combinacionales y secuenciales
- En este último tema se dan unas nociones básicas de estructura y arquitectura de computadores mediante el ejemplo de diseño de un procesador sencillo
- Se verá la importancia del juego de instrucciones de un procesador
- Se analizará paso a paso la creación de un **procesador MIPS monociclo**

CONTENIDOS

1. El procesador MIPS. Juego de instrucciones de un procesador
2. Implementación monociclo de un procesador MIPS: Ruta de datos
3. Implementación monociclo de un procesador MIPS: Unidad de control

CONTENIDOS

1. **El procesador MIPS. Juego de instrucciones de un procesador**
2. Implementación monociclo de un procesador MIPS: Ruta de datos
3. Implementación monociclo de un procesador MIPS: Unidad de control

EL PROCESADOR MIPS

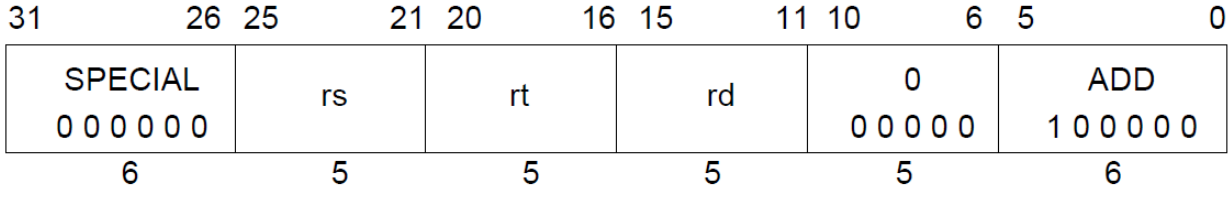
- **MIPS** (*Microprocessor without Interlocked Pipeline Stages*) es una familia de microprocesadores de arquitectura **RISC** desarrollada por MIPS Technologies
- Existen múltiples implementaciones del MIPS tanto en 32 como en 64 bits
- Utilizados en sistemas empotrados (**ej**. rúters), ordenadores personales, servidores e incluso videoconsolas (**ej**. N64, PS1, PS2 y PSP)
- Debido a su claro y reducido juego de instrucciones son muy utilizados en el entorno universitario para explicar arquitectura de computadores

JUEGO DE INSTRUCCIONES (ISA)

- Una **instrucción** es una sentencia que le indica al computador la operación a realizar y los operandos que necesita
- Un **juego de instrucciones (ISA)** es un conjunto de instrucciones seleccionado de tal forma que permite ejecutar gran variedad de programas
- Estas instrucciones indican al hardware lo que debe hacer, por lo que deben estar codificadas en binario, ya que es el idioma del hardware (**lenguaje máquina**)
- La arquitectura de un procesador está definida por su juego de instrucciones
- Un mismo juego de instrucciones puede dar lugar a distintas **microarquitecturas**

EJEMPLO DE INSTRUCCIONES DEL MIPS

ADD **Add Word**



Format: ADD rd, rs, rt MIPS I

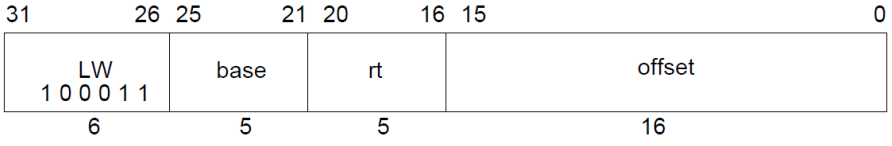
Purpose: To add 32-bit integers.

Description: rd ← rs + rt

[1]

EJEMPLO DE INSTRUCCIONES DEL MIPS

LW — **Load Word**

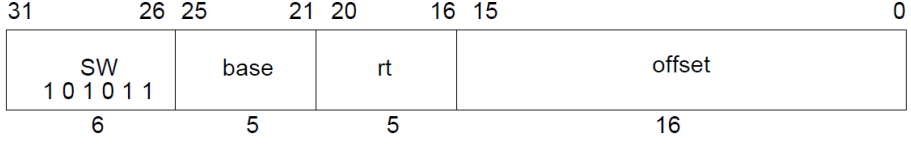


Format: LW rt, offset(base) MIPS I

Purpose: To load a word from memory as a signed value.

Description: rt ← memory[base+offset]

SW — **Store Word**



Format: SW rt, offset(base) MIPS I

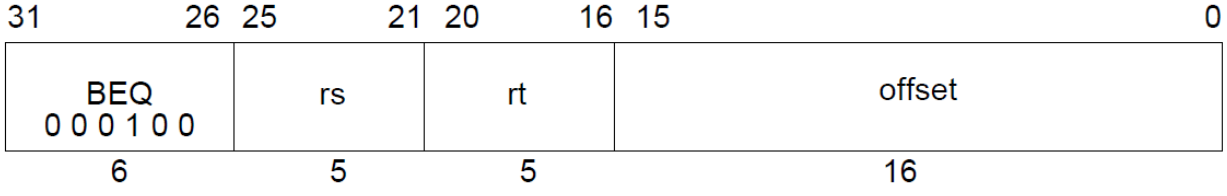
Purpose: To store a word to memory.

Description: memory[base+offset] ← rt

[1]

EJEMPLO DE INSTRUCCIONES DEL MIPS

BEQ **Branch on Equal**



Format: BEQ rs, rt, offset MIPS I

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

[1]

MICROARQUITECTURA

- La **microarquitectura** detalla las conexiones y organización interna de los distintos elementos necesarios para crear una arquitectura (registros, máquinas de estados, ALUs, memorias y otros bloques lógicos)

- Un microprocesador puede tener distintas microarquitecturas, cada una de ellas con sus cualidades de rendimiento, coste y complejidad, pero todas podrán ejecutar el mismo programa

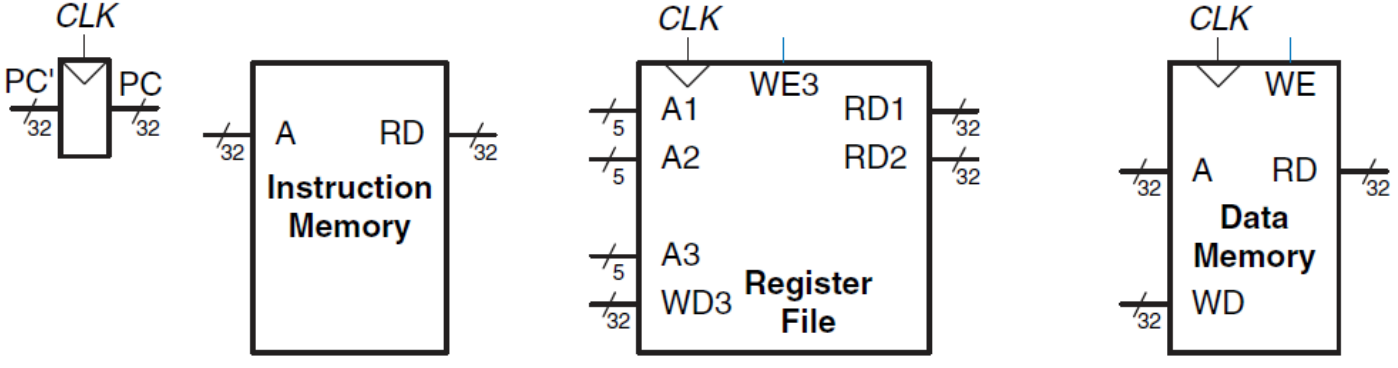
- Veremos el diseño de una microestructura monociclo del procesador MIPS que utiliza únicamente las siguientes instrucciones:
  - Instrucciones lógicas/aritméticas: `add, sub, and, or, slt`
  - Instrucciones de memoria: `lw, sw`
  - Instrucciones de salto: `beq`

CONTENIDOS

1. El procesador MIPS. Juego de instrucciones de un procesador
2. **Implementación monociclo de un procesador MIPS: Ruta de datos**
3. Implementación monociclo de un procesador MIPS: Unidad de control

DISEÑO DE LA RUTA DE DATOS

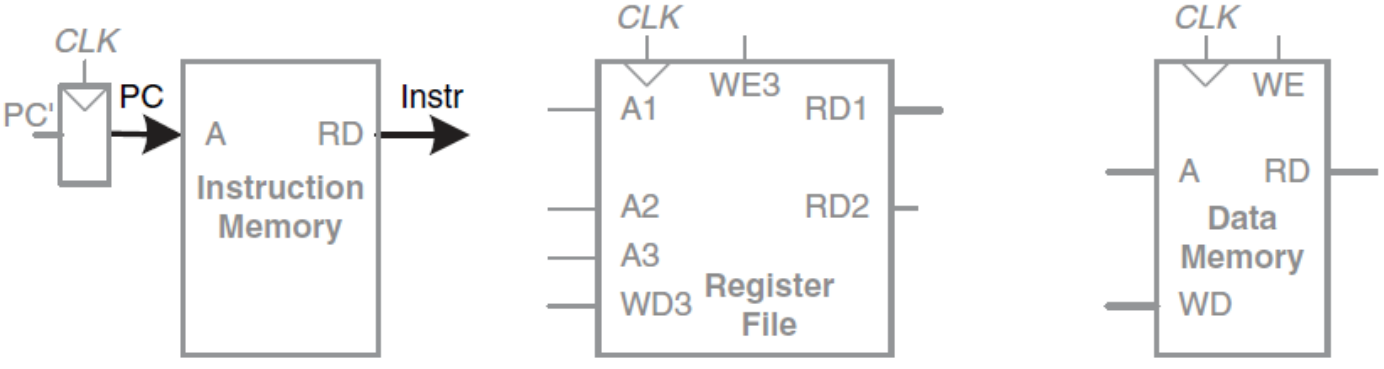
- Partimos del contador de programa o **PC** (registro de 32 bits), el **banco de registros** (32 registros de 32 bits) y las **memorias de instrucciones y datos**



- El procesador a diseñar será **monociclo**, por lo que las instrucciones se ejecutarán en un único ciclo [2]

DISEÑO DE LA RUTA DE DATOS

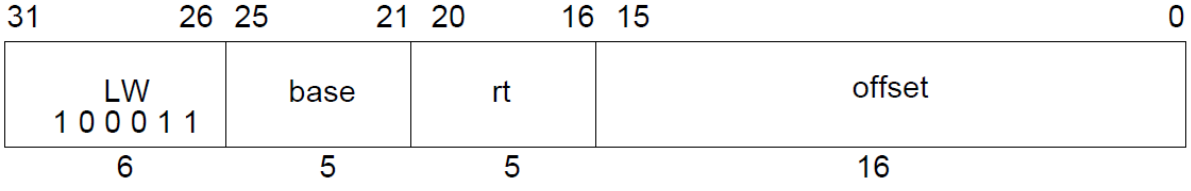
- Leemos la instrucción a ejecutar (*Instr*) de la memoria de instrucciones, la cual contiene todas las instrucciones del programa en curso (etapa de *fetch*)



- Según la instrucción obtenida, el procesador hará una cosa u otra, comenzaremos por la instrucción `lw` [2]

RECORDEMOS: INSTRUCCIÓN LW

LW **Load Word**



Format: LW rt, offset(base) MIPS I

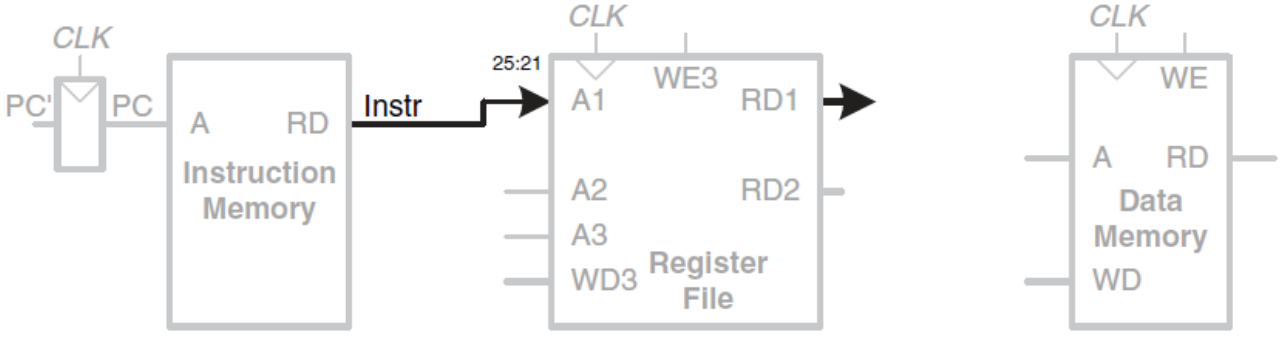
Purpose: To load a word from memory as a signed value.

Description: rt ← memory[base+offset]

[1]

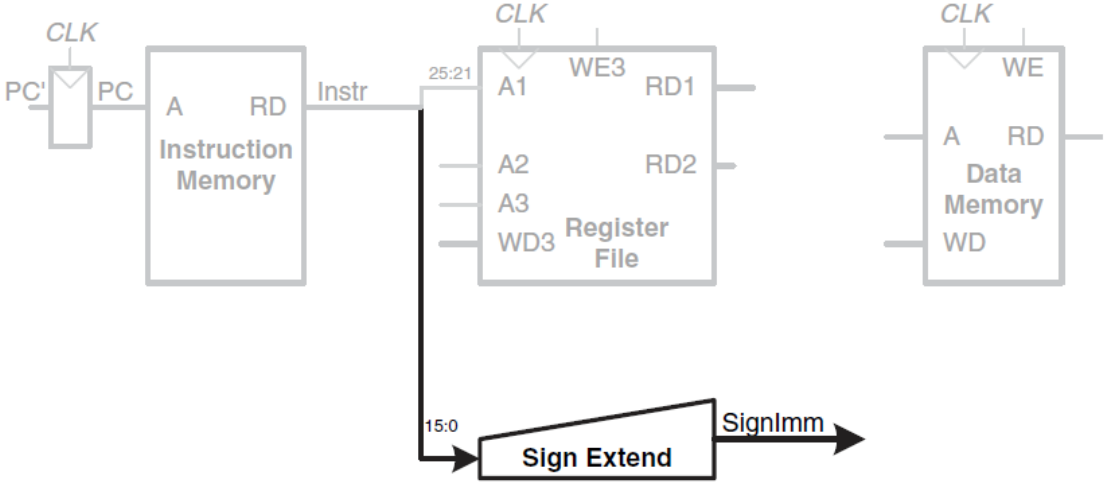
DISEÑO DE LA RUTA DE DATOS: LW

- Obtenemos el campo *base address* leyendo del registro correspondiente



DISEÑO DE LA RUTA DE DATOS: LW

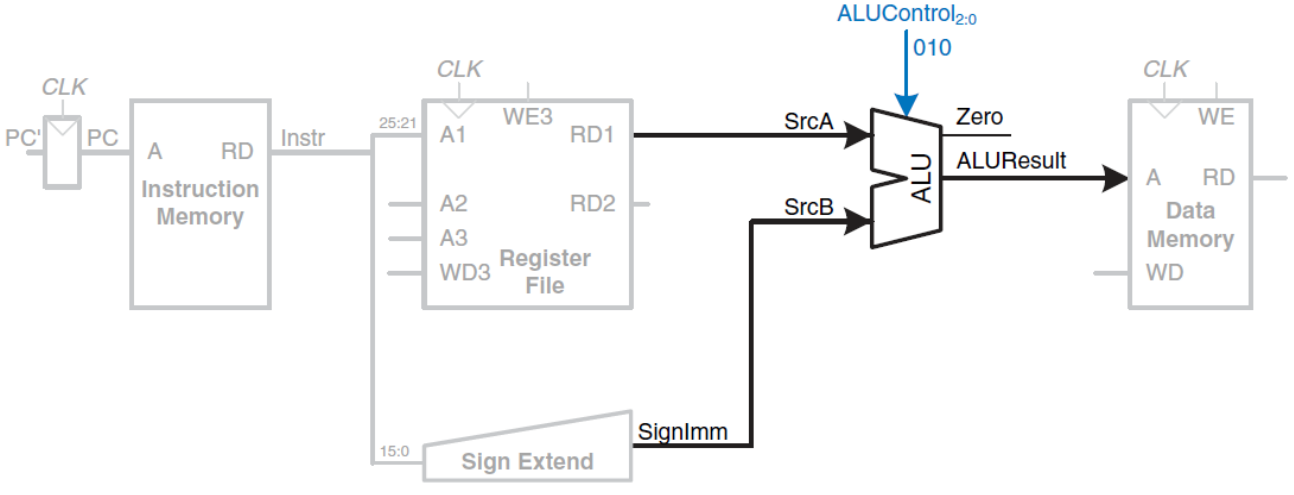
- El *offset* está almacenado en los 16 LSBs de la instrucción, pero este valor puede ser positivo o negativo, por lo que hay que realizar una extensión de signo a 32bits



[2]

DISEÑO DE LA RUTA DE DATOS: LW

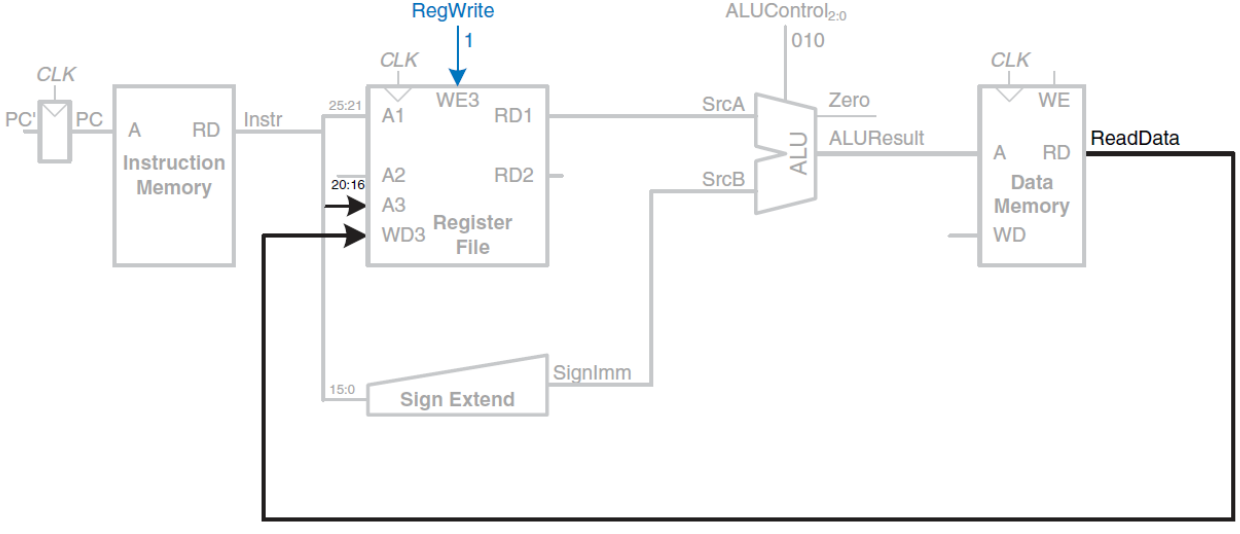
- Para así poder sumar *base* + *offset* y obtener la dirección de memoria de la que hay que leer la palabra. Usaremos una ALU para la suma y guardaremos el resultado en la memoria de datos



[2]

DISEÑO DE LA RUTA DE DATOS: LW

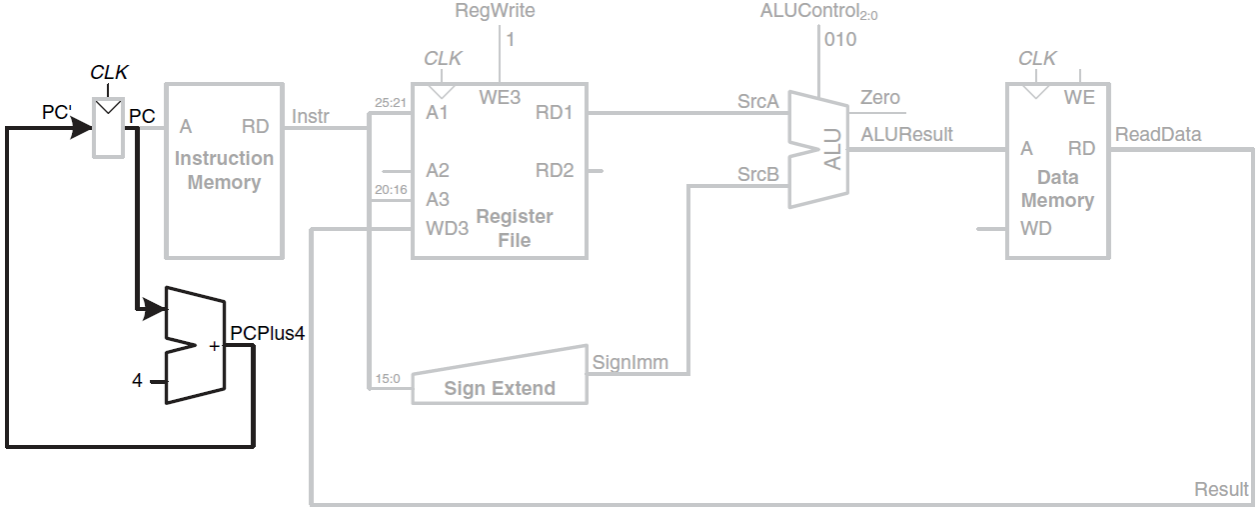
- Ahora leeremos el dato de memoria y lo guardaremos en el registro de destino *rt* especificado por los bits 20:16 de la instrucción `lw`



[2]

DISEÑO DE LA RUTA DE DATOS: LW

- Mientras se ejecuta la instrucción `lw`, el procesador debe computar la dirección de la siguiente instrucción a ejecutar, como son 32 bits (PC = PC + 4 bytes)

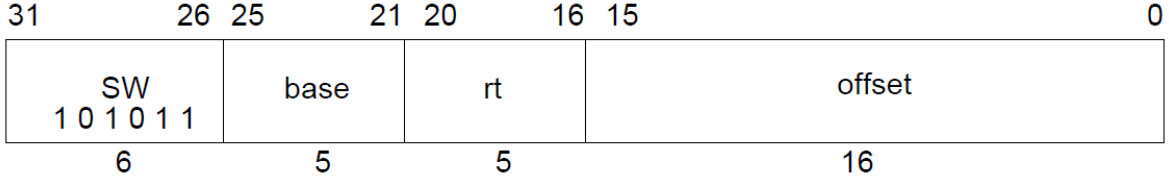


[2]

RECORDEMOS: INSTRUCCIÓN SW

- Completado el datapath de `lw`, ahora vamos a ampliarlo para `sw`

SW **Store Word**



Format: SW rt, offset(base) MIPS I

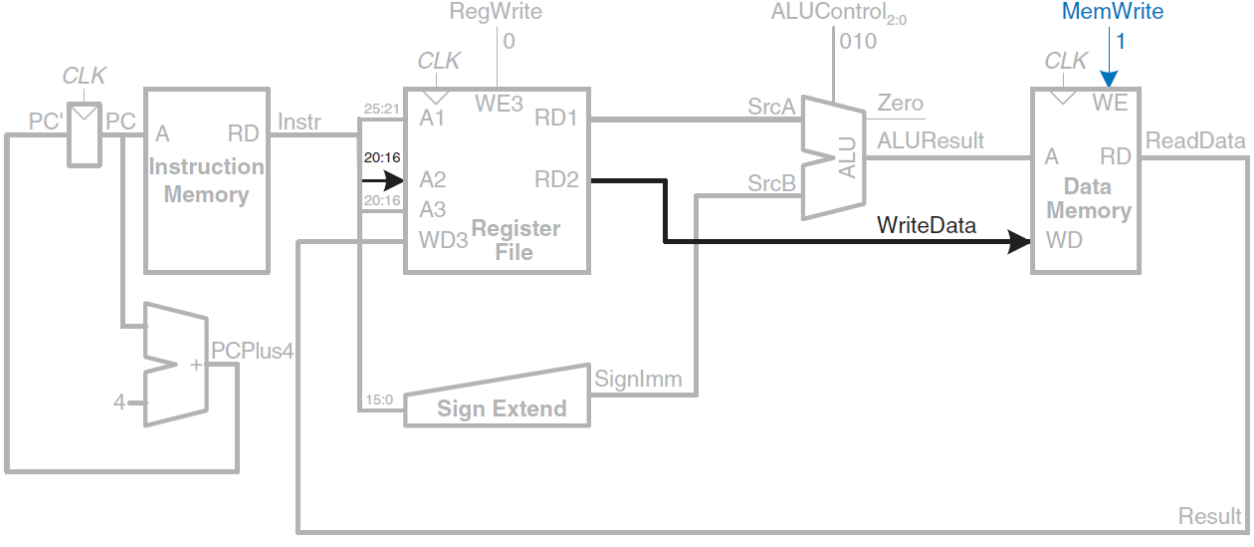
Purpose: To store a word to memory.

Description: memory[base+offset] ← rt

[1]

DISEÑO DE LA RUTA DE DATOS: SW

- Hay que leer el registro *rt* y… ¡el resto de funcionalidades ya están implementadas!

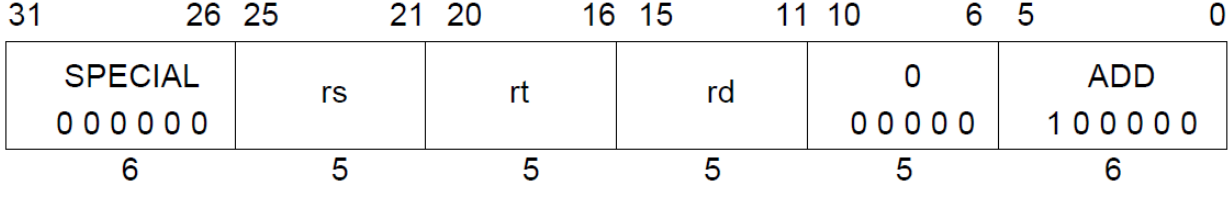


[2]

RECORDEMOS: INSTRUCCIONES ARITMÉTICAS

- Requieren leer dos registros y realizar algún tipo de operación con la ALU. El resultado se guarda en un tercer registro. La principal diferencia es la operación

ADD — **Add Word**



Format: ADD rd, rs, rt — MIPS I

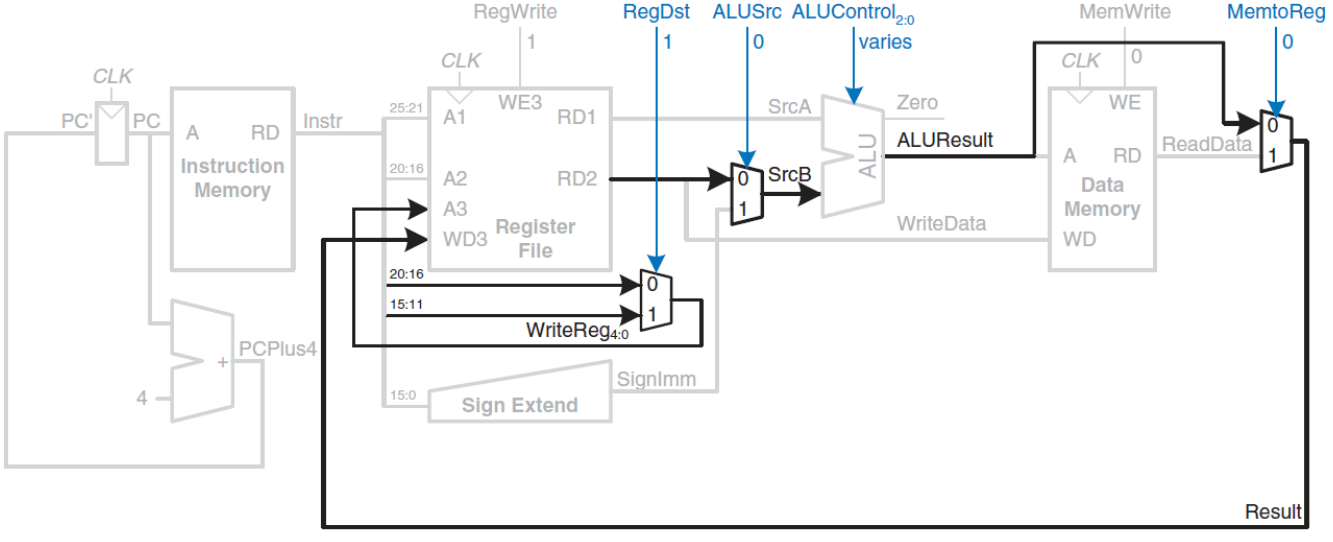
Purpose: To add 32-bit integers.

Description: rd ← rs + rt

[1]

DISEÑO DE LA RUTA DE DATOS: ADD

- Añadimos tres multiplexores para elegir entre *SignImm* o el registro 2, el dato leído o el generado por la ALU, y entre la dirección *Instr20:16* o la del registro *rd*

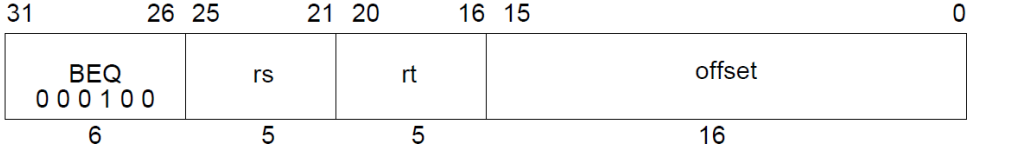


[2]

RECORDEMOS: INSTRUCCIÓN BEQ

- Compara dos registros, si son iguales salta un determinado número de instrucciones indicado por el offset (que puede ser positivo o negativo)
- Por tanto, será necesario implementar: **PC' = PC + 4 + SignImm x 4**

BEQ **Branch on Equal**



Format: BEQ rs, rt, offset MIPS I

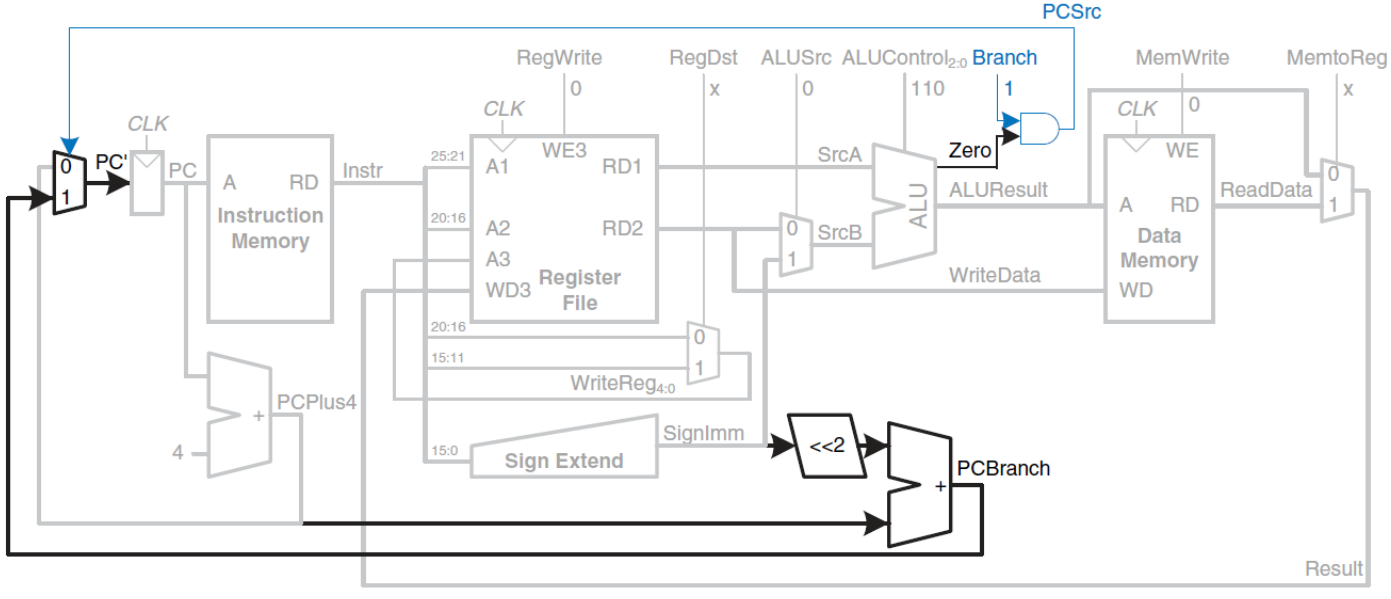
Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

[1]

DISEÑO DE LA RUTA DE DATOS: BEQ

- La multiplicación x 4 la hacemos un desplazamiento

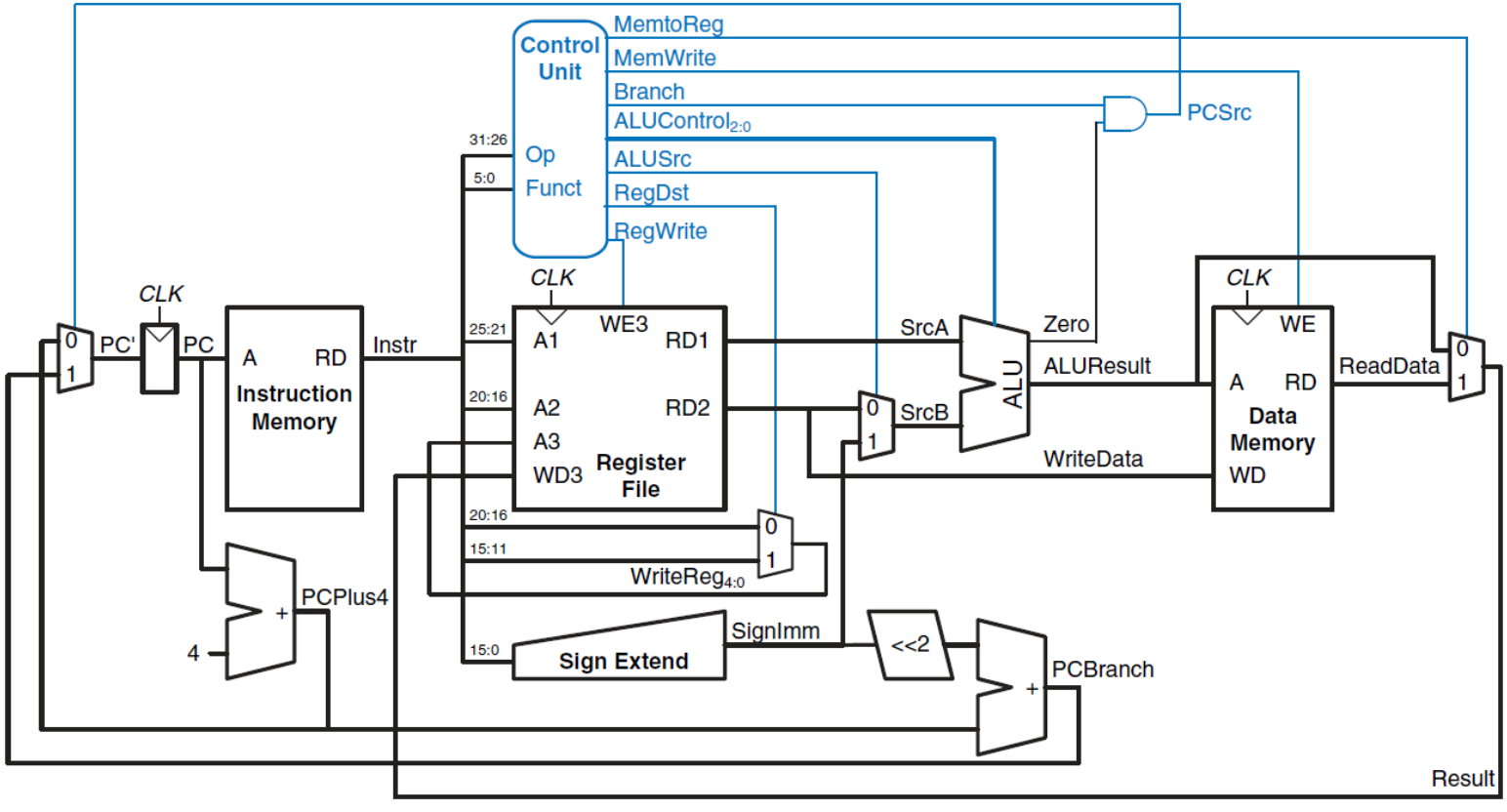


[2]

CONTENIDOS

1. El procesador MIPS. Juego de instrucciones de un procesador
2. Implementación monociclo de un procesador MIPS: Ruta de datos
3. **Implementación monociclo de un procesador MIPS: Unidad de control**

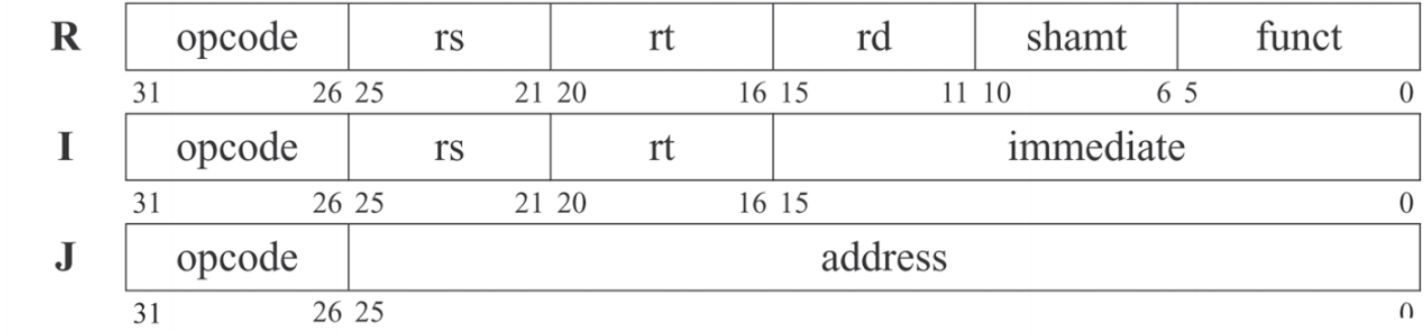
DISEÑO DE LA UNIDAD DE CONTROL



[2]

DISEÑO DE LA UNIDAD DE CONTROL

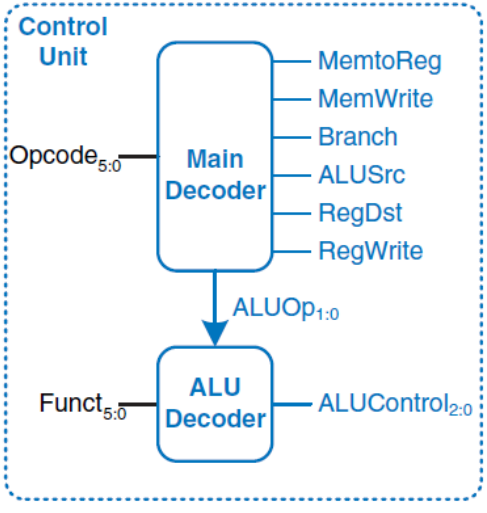
- La unidad de control computa las señales de control (en azul) basándose en los campos *opcode* y *funct* de cada instrucción



- La unidad de control tendrá, por tanto, dos decodificadores

DISEÑO DE LA UNIDAD DE CONTROL

- Podemos construir una tabla de verdad para la señal ALUOp

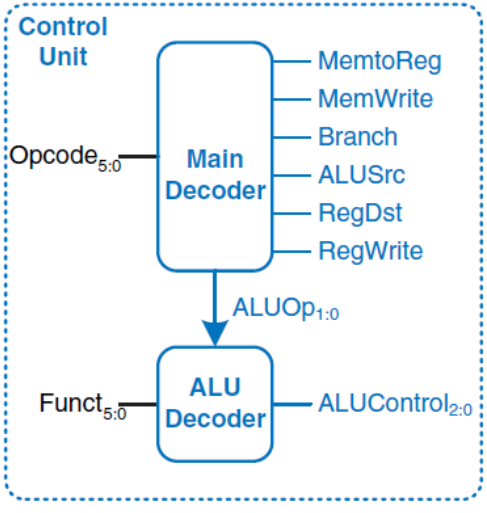


ALUOp	Significado
00	add
01	subtract
10	Mirar el campo funct
11	Nada

[2]

DISEÑO DE LA UNIDAD DE CONTROL

- Incluyendo el campo *funct* quedaría:



ALUOp	Funct	ALUControl
00	X	010 (add)
01	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

[2]

DISEÑO DE LA UNIDAD DE CONTROL

- Según el opcode, las señales a generar serían:

Instrucción	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
and,or,etc.	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

- Ya sólo quedaría diseñar la unidad de control utilizando tu método favorito ☺

REFERENCIAS TEMA 10

Las imágenes del juego de instrucciones han sido obtenidas de:

[1] MIPS IV Instruction Set.

Las imágenes del procesador MIPS monociclo han sido obtenidas de:

[2] Harris D. and Harris S., "Digital Design and Computer Architecture", 2012.

TECNOLOGÍA DE COMPUTADORES

Prof. Dr. Luis Alberto Aranda
Prof. Dr. Iván Ramírez

Universidad Rey Juan Carlos