UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
CAMPUS SAN JOAQUÍN

Arquitectura y Organización de Computadores

Índice

- ✓ Introducción
- ✓ Circuitos Aritméticos
- ✓ Sistemas Numéricos
- ✓ Bloques Secuenciales
- ✓ Arreglos de Memoria
- ✓ Arreglos Lógicos

Introducción

- ✓ Hasta este punto hemos estudiado el diseño de circuitos secuenciales y combinacionales utilizando ecuaciones booleanas, esquema y lenguaje HDL.

- ✓ El diseño y construcción del procesador nos hará necesitar una serie de componentes más estructurados con funciones específicas.

- ✓ Estos componentes incluyen: Circuitos aritméticos, multiplexores, decodificadores, contadores, registros, arreglos de memoria y lógicos.

Introducción

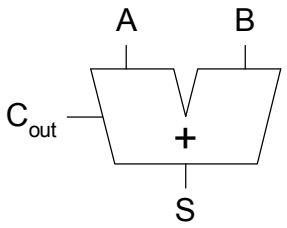
- ✓ Estos componentes siguen los principios de jerarquía, modularidad y regularidad.
- ✓ Jerarquía: Bloques construídos con elementos simples (compuertas)
- ✓ Modularidad: Cada bloque tiene una interfaz y funciones bien definidas.
- ✓ Regularidad: La estructura de cada bloque es fácil de extender a diferentes tamaños (en términos de bits)

Circuitos Aritméticos

- ✓ Estos circuitos corresponden a los componentes principales de un computador.
- ✓ Los computadores realizan una gran cantidad de funciones aritméticas: Suma, resta, comparación, desplazamientos, multiplicación y división.
- ✓ Suma: Una de las operaciones más importantes. Analizaremos el caso de 1 bit, luego generalizaremos.

Circuitos Aritméticos, Suma

Half Adder

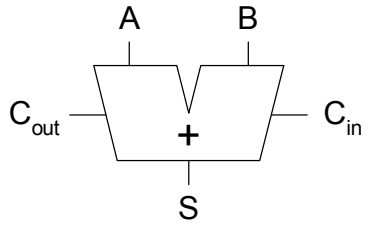


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	C_{out}	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

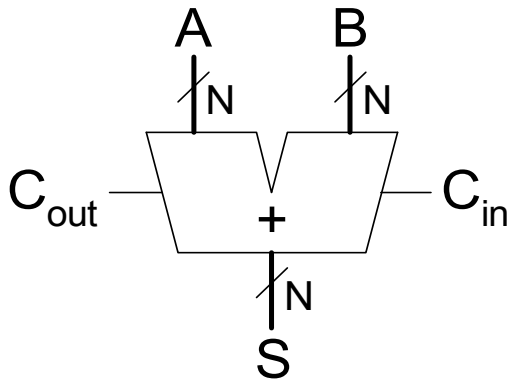
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Circuitos Aritméticos, Suma

- ✓ Un sumador de N-bits realiza la operación suma entre dos entradas de N-bits A y B y un carry de entrada. Produce un resultado de N-bits y un carry de salida.

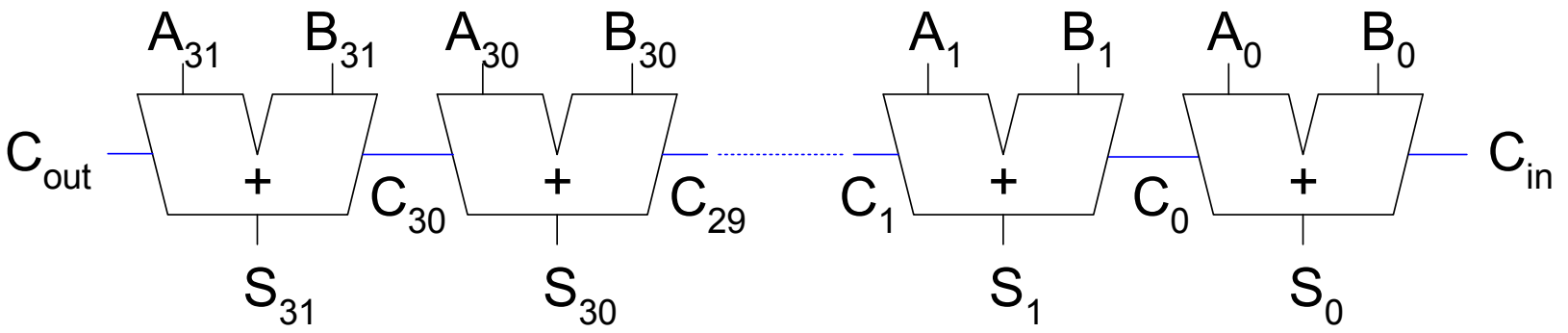
- ✓ Se conoce usualmente como Sumador propagador de carry (CPA) debido a que el carry de salida se propaga al siguiente bit.



Circuitos Aritméticos, Suma

- ✓ Existen 3 tipos:
  - ✓ Ripple Carry: La forma más sencilla de construir el elemento.
    - La idea es juntar N full adders.
    - Cout de una etapa actúa como Cin de la siguiente etapa.
    - Es un buen ejemplo de modularidad y regularidad: Se repite N veces para construir un sistema más complejo.
    - Es “lento” para N muy grande.
    - Se genera un delay t = N*tfa.
  - ✓ Carry Lookahead adder
  - ✓ Prefix adder

Circuitos Aritméticos, Suma

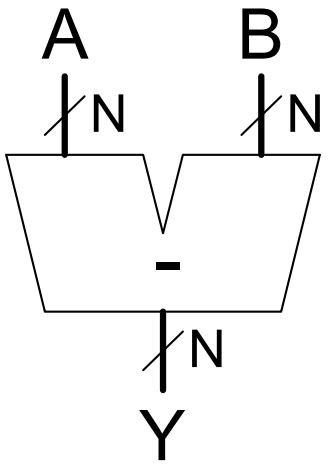


Circuitos Aritméticos, Resta

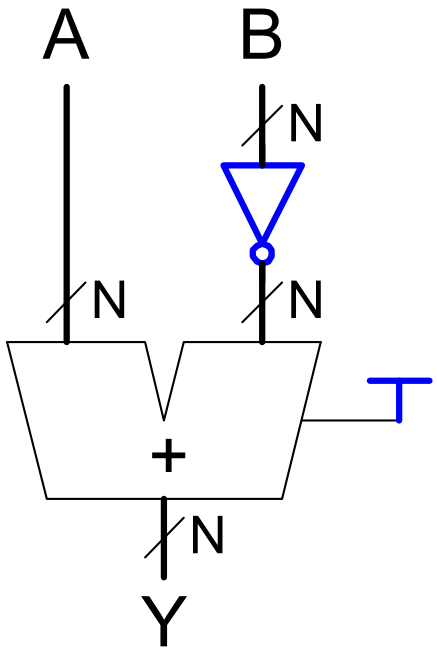
- ✓ El sumador puede sumar números positivos y negativos utilizando el estándar complemento dos.
- ✓ La resta es fácil. Cambiamos el signo del segundo número y luego sumamos. Cambiar el signo de un número complemento dos implica cambiar su valor de verdad y sumar 1.
- ✓ $Y = A - B = A + \overline{B} + 1$
- ✓ Podemos implementar esto utilizando un CPA con Cin = 1.

Circuitos Aritméticos, Resta

Symbol



Implementation



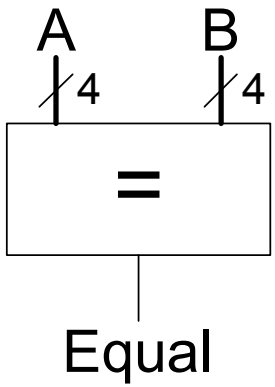
Circuitos Aritméticos, Comparador

- ✓ El comparador determina si dos números binarios de N bits son iguales o si uno es más grande que otro.

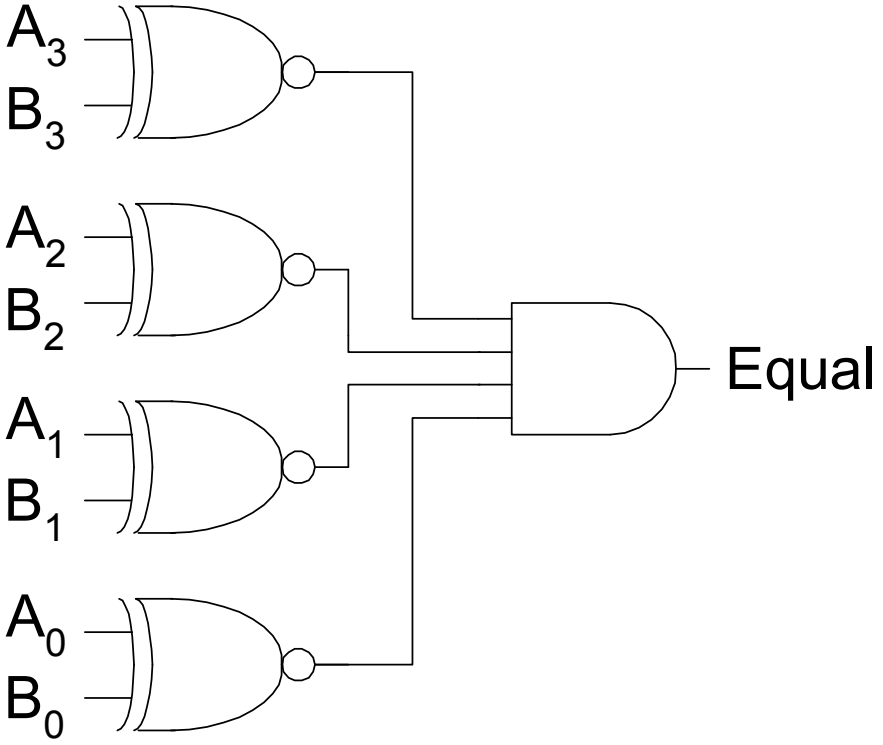
- ✓ Podemos identificar dos tipos:
  - ✓ Comparador de igualdad: Única salida. Si son iguales o no.
  - ✓ Comparador de Magnitud: Varias salidas. Magnitud de los números.

Circuitos Aritméticos, Comparador Igualdad

Symbol

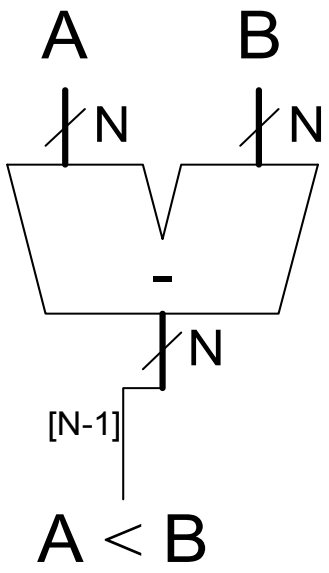


Implementation



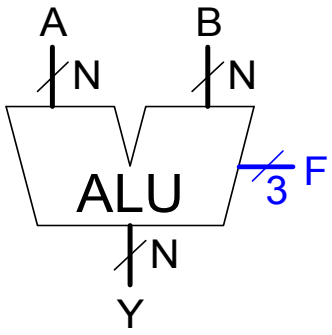
Circuitos Aritméticos, Comparador Magnitud

- ✓ Esta comparación se realiza con la operación A-B. Analizamos el signo del resultado. Si el resultado es negativo, A es más pequeño que B.

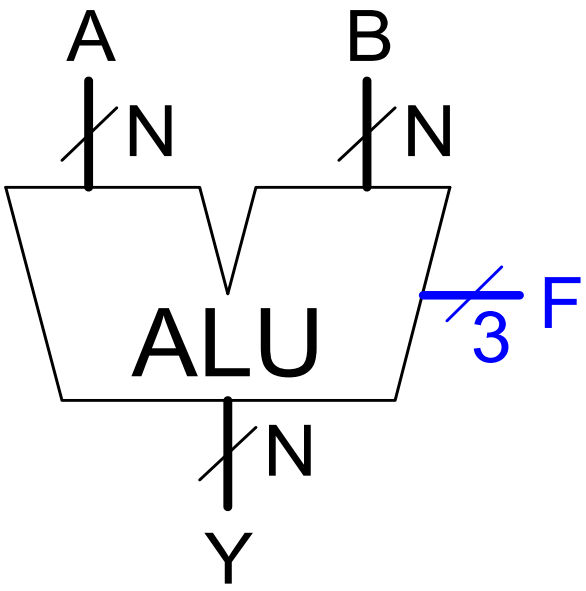


Circuitos Aritméticos, ALU

- ✓ Combina un variopinto de funciones matemáticas.
- ✓ Una ALU clásica debe realizar las operaciones de suma, resta, comparación de magnitud, AND y OR.
- ✓ La ALU es el corazón de los sistemas computacionales.

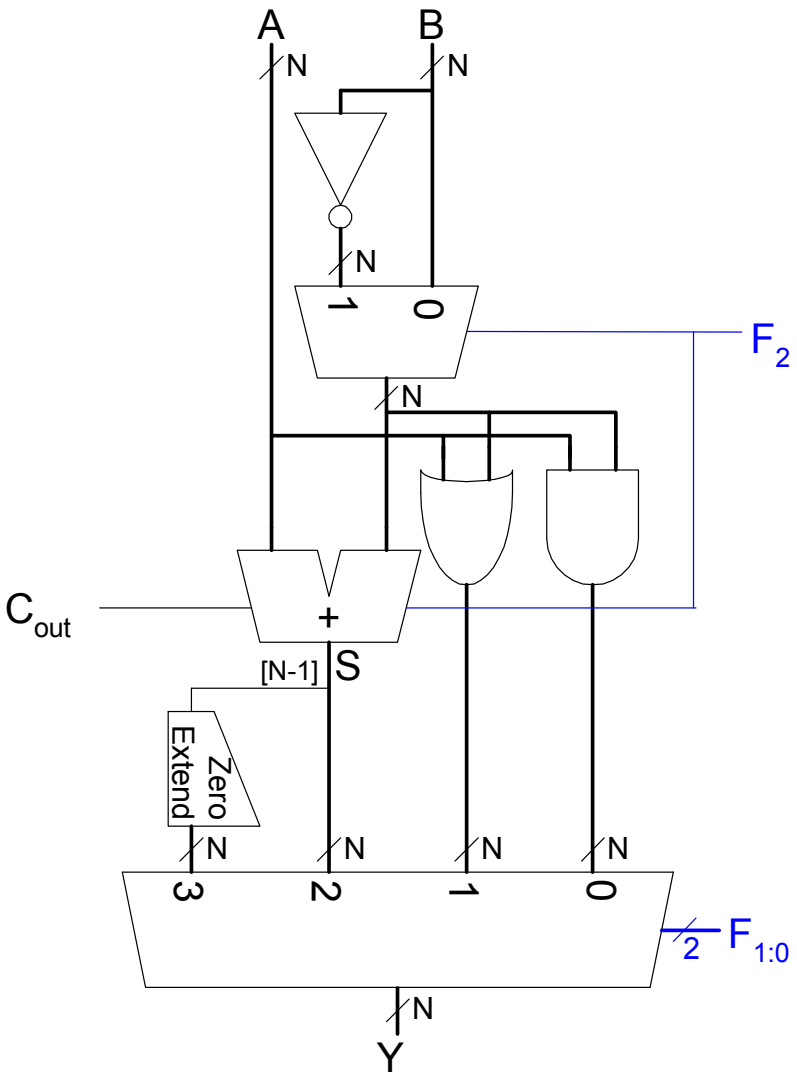


Circuitos Aritméticos, ALU



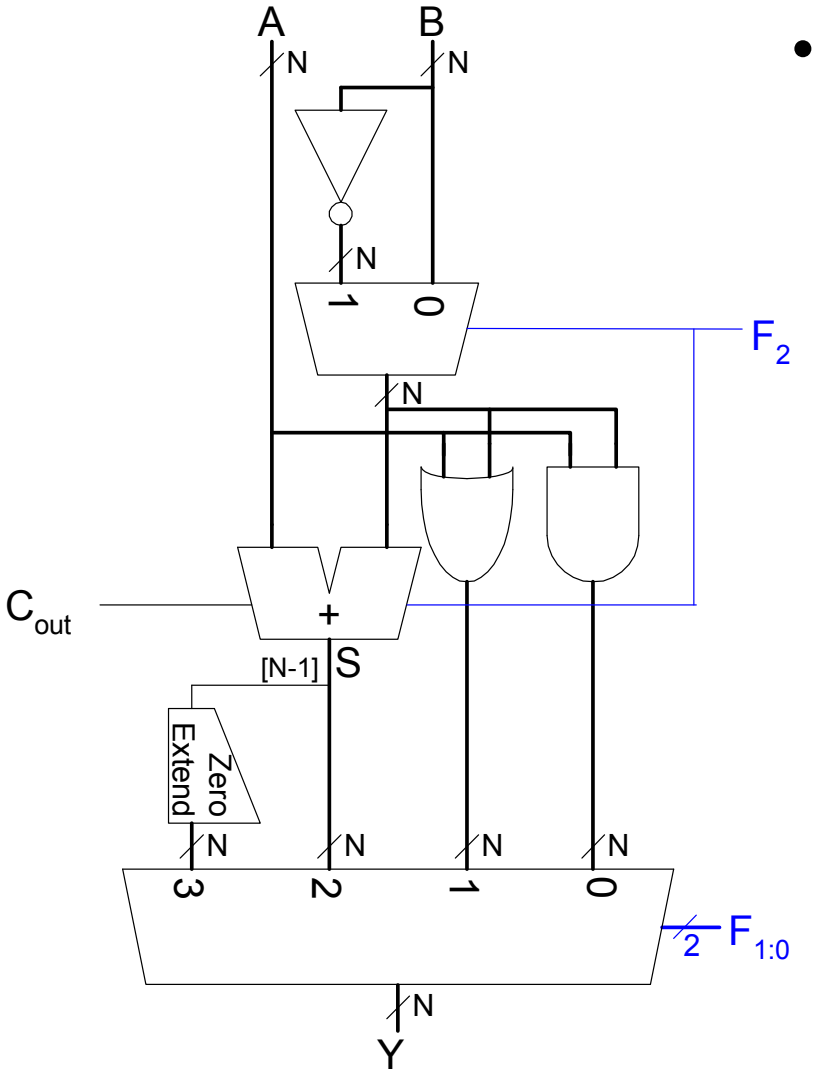
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Circuitos Aritméticos, ALU



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Circuitos Aritméticos, ALU



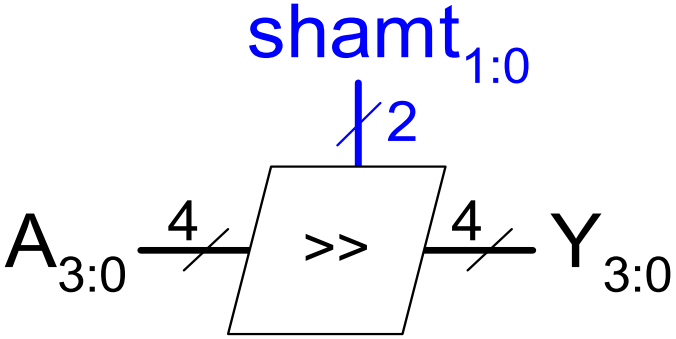
- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$
  - $A < B$, so Y should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0} = 111$
    - $F_2 = 1$ (adder acts as subtracter), so 25 - 32 = -7
    - -7 has 1 in the most significant bit ($S_{31} = 1$)
    - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

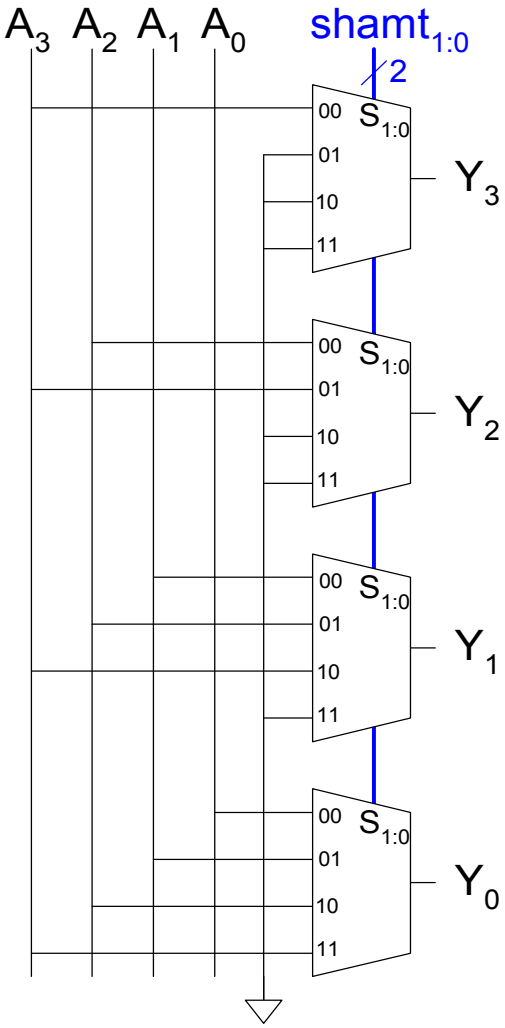
Circuitos Aritméticos, Desplazamientos

✓ La idea es mover bits a la izquierda o derecha según corresponda. Podemos implementar la multiplicación y/o división en potencias de dos.

- **Logical shifter:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
  - Ex: 11001 >>> 2 = **11**110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:**
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Circuitos Aritméticos, Desplazamientos





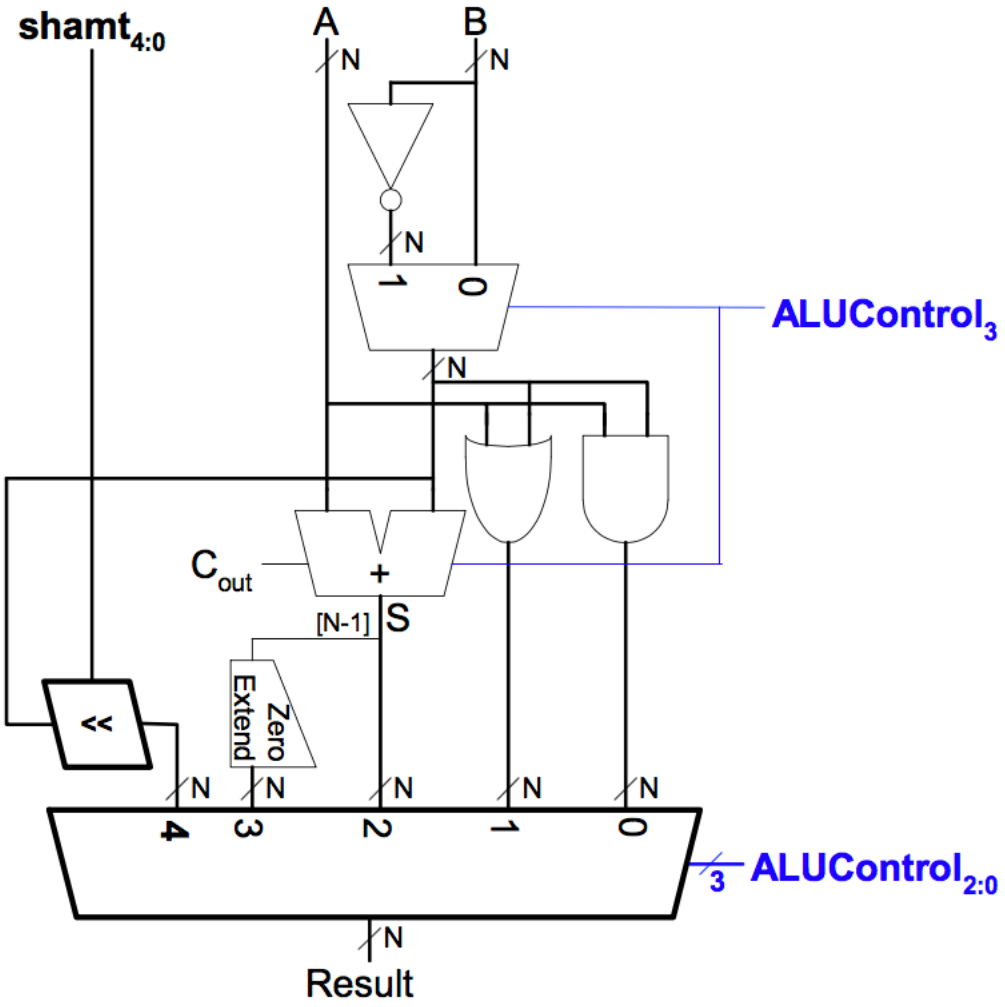
Circuitos Aritméticos, Desplazamientos

- $A << N = A \times 2^N$
  - **Example:** 00001 << 2 = 00100 $(1 \times 2^2 = 4)$
  - **Example:** 11101 << 2 = 10100 $(-3 \times 2^2 = -12)$
- $A >>> N = A \div 2^N$
  - **Example:** 01000 >>> 2 = 00010 $(8 \div 2^2 = 2)$
  - **Example:** 10000 >>> 2 = 11100 $(-16 \div 2^2 = -4)$

Circuitos Aritméticos, ALU

- ✓ Agregar la instrucción SLL a una ALU.

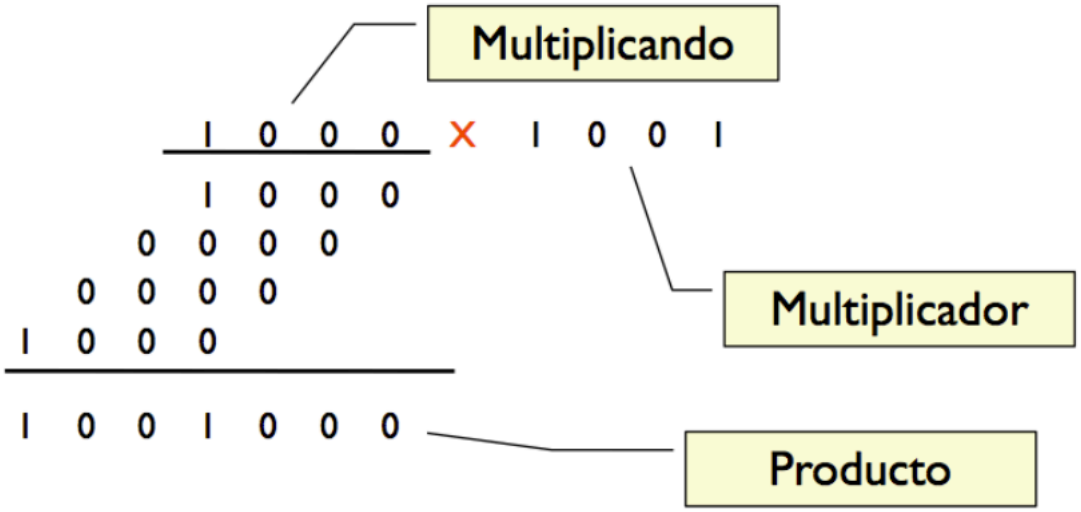
Circuitos Aritméticos, ALU



Circuitos Aritméticos, ALU

$ALUControl_{3:0}$	Function
0000	A AND B
0001	A OR B
0010	A + B
0011	not used
1000	A AND $\overline{B}$
1001	A OR $\overline{B}$
1010	A - B
1011	SLT
0100	SLL

Circuitos Aritméticos, Multiplicación



Circuitos Aritméticos, Multiplicación 1era V.

- ✓ Notar que estas versiones se basan en el algoritmo aprendido en básica para multiplicar.
- ✓ La primera versión del algoritmo de multiplicación es conocida como “secuencial” supone que el multiplicador se encuentra en un registro de 32 bits.
- ✓ Por otro lado, el producto se encuentra en un registro de 64 bits y está inicializado con puros 0’s.
- ✓ Del ejemplo inicial de multiplicación, podemos notar que es necesario mover hacia la izquierda el multiplicando por cada paso de la iteración del algoritmo.

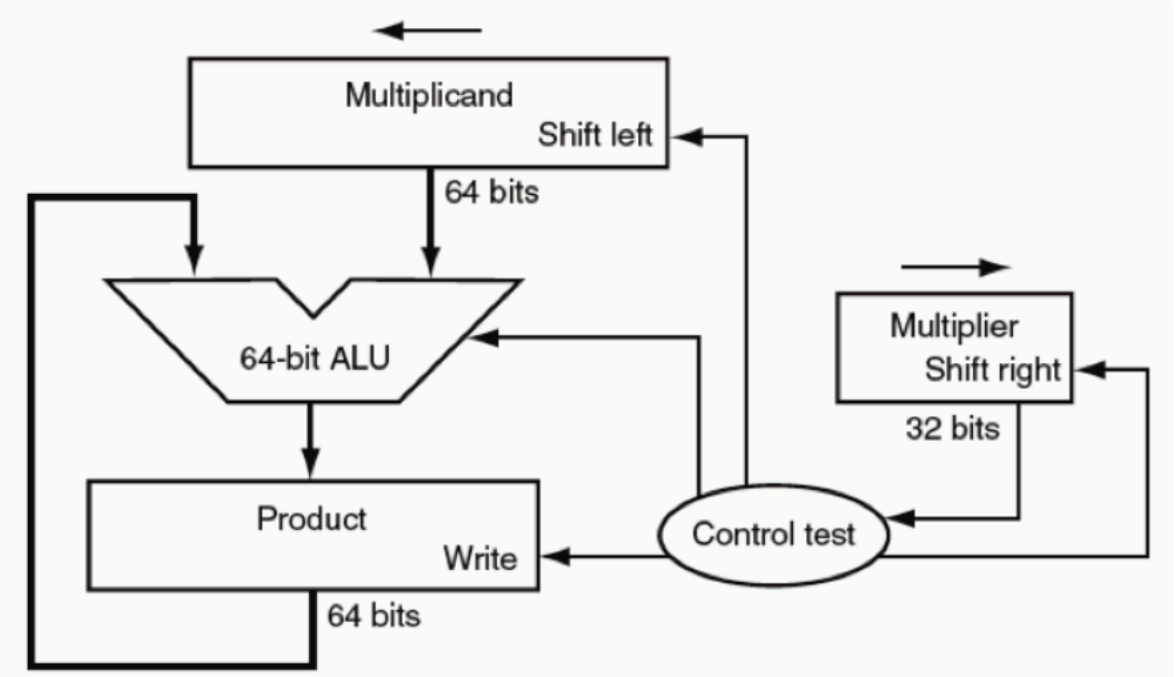
Circuitos Aritméticos, Multiplicación 1era V.

- ✓ Un registro de 32 bits sobre 32 iteraciones del algoritmo deberá mover esta misma cantidad de elementos hacia la izquierda, por lo que será necesario que el multiplicando sea un registro de 64 bits el cual deberá estar inicializado con los 32 bits del multiplicando en la parte de la “derecha” del registro, y el resto relleno de 0’s.

- ✓ Este registro será desplazado en 1 bit hacia la izquierda por cada iteración del algoritmo para alinear el multiplicando con la suma acumulada en el registro del producto de 64 bits.

- ✓ Finalmente, la primera versión corresponde al siguiente diagrama:

Circuitos Aritméticos, Multiplicación 1era V.

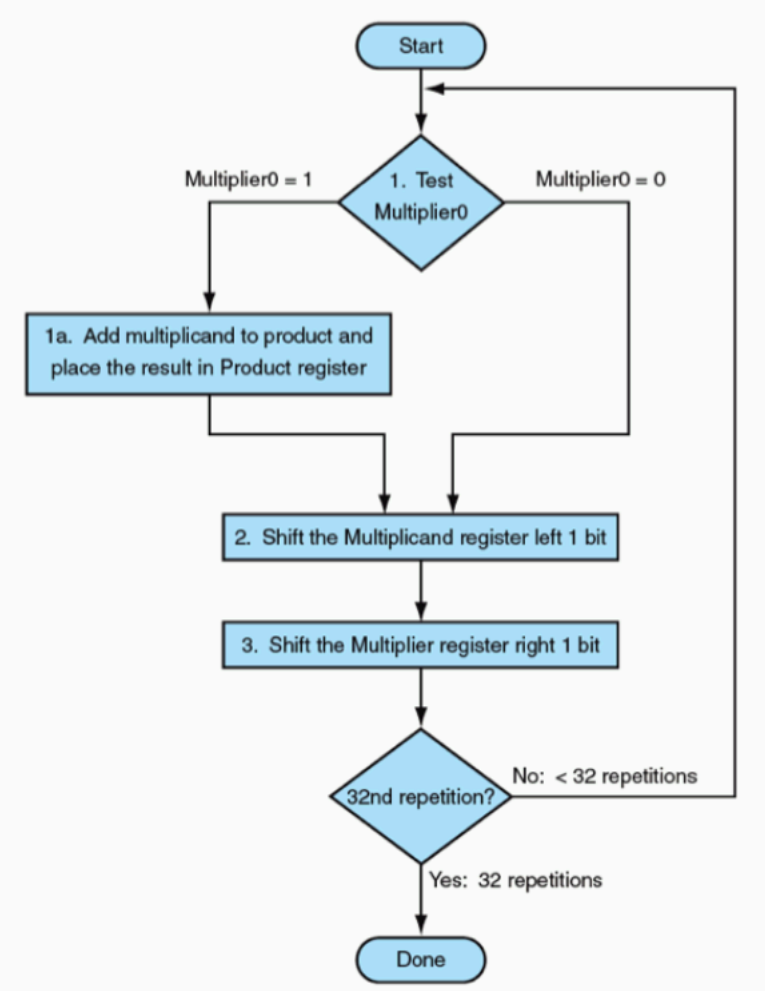


Circuitos Aritméticos, Multiplicación 1era V.

- ✓ La unidad de control que compone el diagrama es aquella que decide cuando desplazar el multiplicando y el multiplicador y cuando escribir nuevos valores en el registro del producto.

- ✓ El siguiente diagrama de flujo explica de mejor manera el proceso realizado:

Circuitos Aritméticos, Multiplicación 1era V.



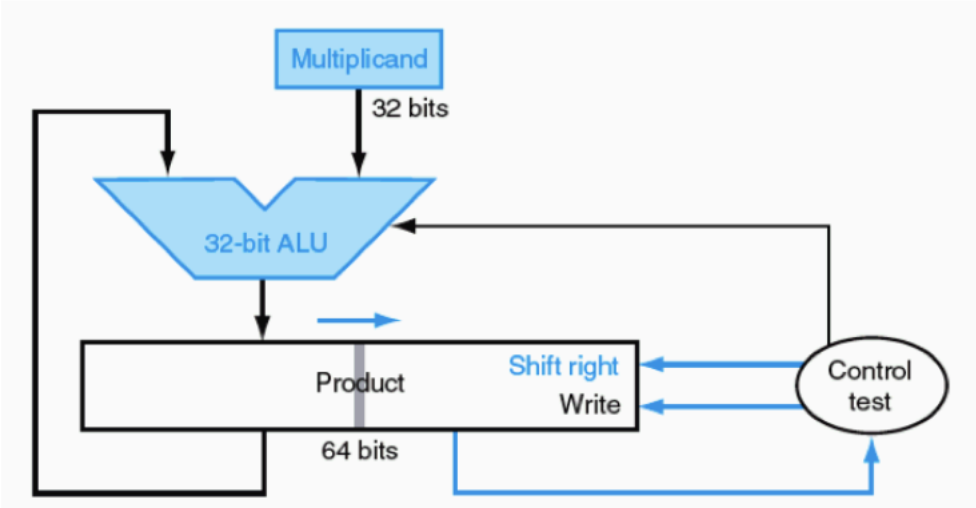
Circuitos Aritméticos, Multiplicación 1era V.

- ✓ El bit menos significativo del multiplicador determina cuando el multiplicando es agregado al registro del producto.
- ✓ El paso 2 tiene el efecto de mover los operandos intermedios a la izquierda.
- ✓ El paso 3 permite obtener el siguiente bit del multiplicador.
- ✓ Este paso se repite 32 veces, por lo que si cada etapa requiere 1 ciclo de reloj, el algoritmo requeriría al menos unos 100 ciclos de reloj ara multiplicar dos números de 32 bits.

Circuitos Aritméticos, Multiplicación V. Final

- ✓ La solución anterior necesita que el registro multiplicando sea de 64 bits. Esto significa que la ALU también lo debe ser.
- ✓ En otras palabras, se usa una ALU de 64 bits para generar un resultado de 32 bits. La segunda versión propone utilizar una ALU de 32 bits, pero es necesario hacer algunas modificaciones.
- ✓ El objetivo de versión es realizar las operaciones de forma paralela.
- ✓ El multiplicador y el multiplicando son desplazados mientras el multiplicando es agregado al registro del producto. Todo esto ocurre si el bit del multiplicador es 1.

Circuitos Aritméticos, Multiplicación V. Final



Circuitos Aritméticos, Multiplicación V. Final

✓ Por ejemplo: Supongamos que queremos realizar la operación 2*3, cuyo resultado es 6. Notar que el bit encerrado en un círculo es aquél que determina la siguiente etapa en la iteración.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Circuitos Aritméticos, Multiplicación Rápida

- ✓ Otra forma de abordar e implementar la multiplicación por hardware es volviendo a los conceptos básicos de la multiplicación.
- ✓ La multiplicación de números binarios (sin signo) es muy parecida a la multiplicación de números decimales. En ambos casos el producto parcial está formado por la multiplicación de un dígito del multiplicador con todo el multiplicando.
- ✓ En general, una multiplicación N x N se realiza con dos números de N-bit's y generan un resultado de a lo más 2N-bit's.
- ✓ Podemos notar que las multiplicación binaria entre dos dígitos corresponde a la función AND entre ellas, por lo que esta compuerta podría servir para construir un multiplicador.

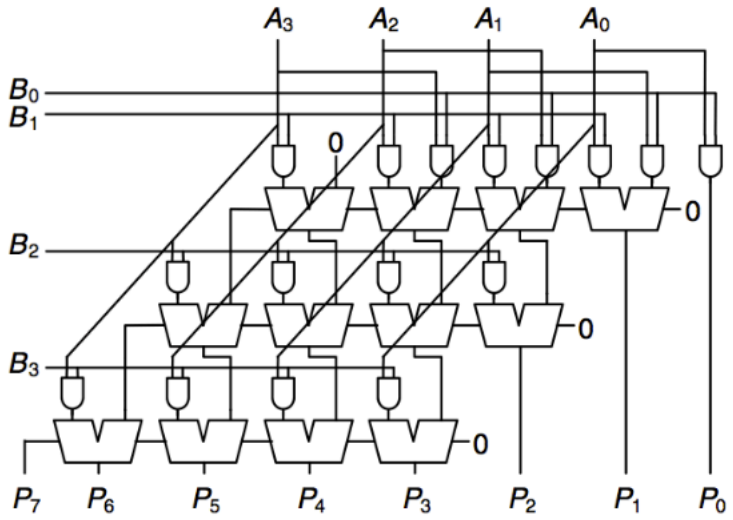
Circuitos Aritméticos, Multiplicación Rápida

- ✓ Por ejemplo, consideremos que vamos a multiplicar el número A (A3, A2, A1, A0) de 4 bits con B (B3, B2, B1, B0) de 4 bits. El resultado quedará expresado como P. Teóricamente, el resultado vendría dado por:

$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
 & & \times & B_3 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

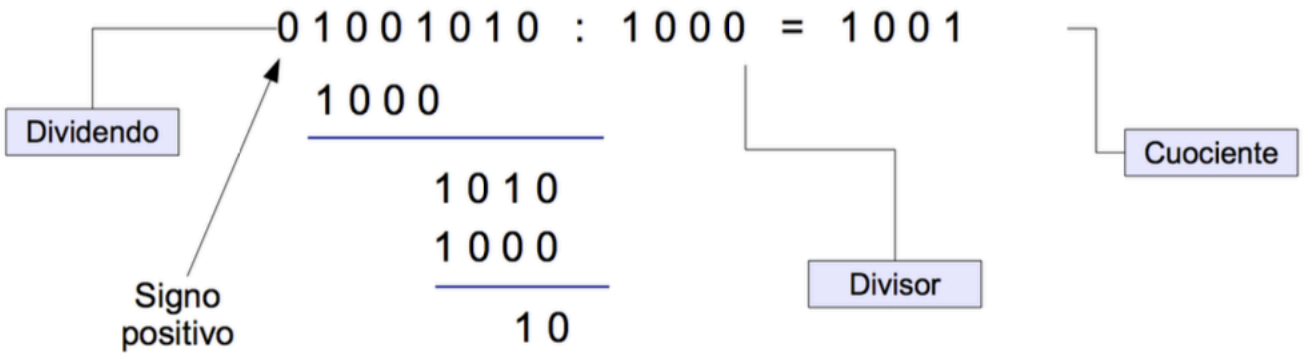
Circuitos Aritméticos, Multiplicación Rápida

- ✓ Según lo anterior, podemos más o menos “notar” una forma para generalizar el resultado de una multiplicación. Utilizando compuertas AND y sumadores, tendremos la siguiente implementación para una multiplicación de 4 x 4 bits.



Circuitos Aritméticos, División

- ✓ Al igual que en el caso de la multiplicación, la división es “algo” más complicada de implementar.
- ✓ Nuevamente, la forma más simple es recurrir a los algoritmos de la educación básica.



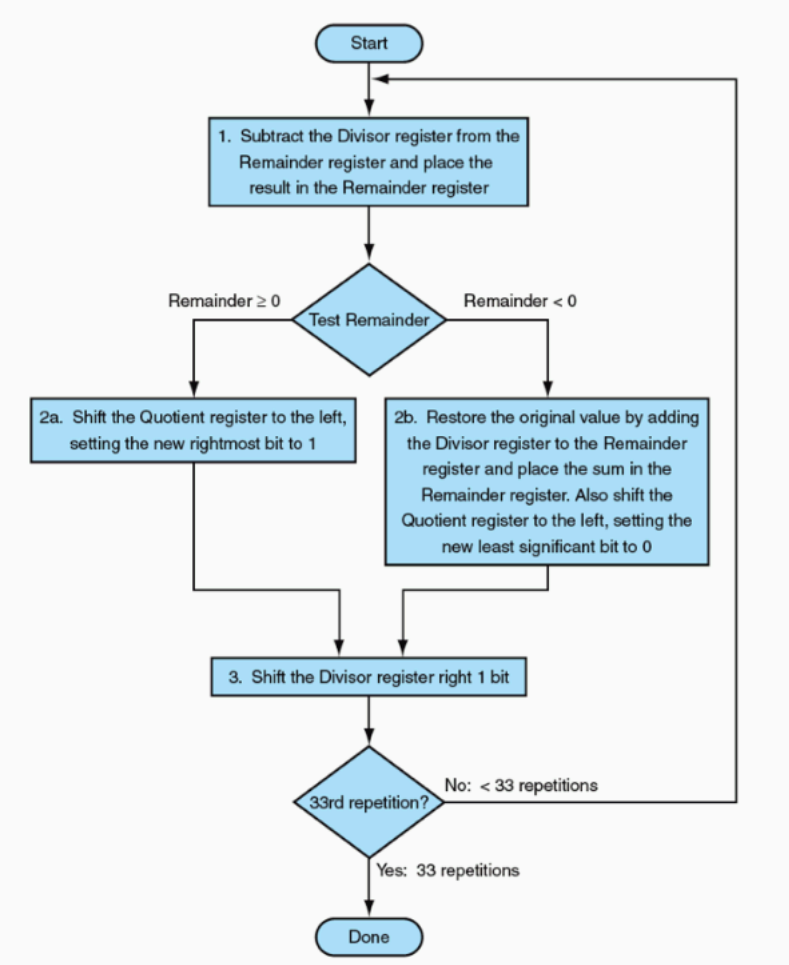
Circuitos Aritméticos, División

- ✓ **Dividendo**: Número a ser dividido.
- ✓ **Divisor**: Número que dividirá al dividendo.
- ✓ **Cuociente**: Resultado principal de la división. Si este número es multiplicado por el divisor y sumado al resto, se obtendrá el dividendo.
- ✓ **Resto**: Resultado secundario de la división.

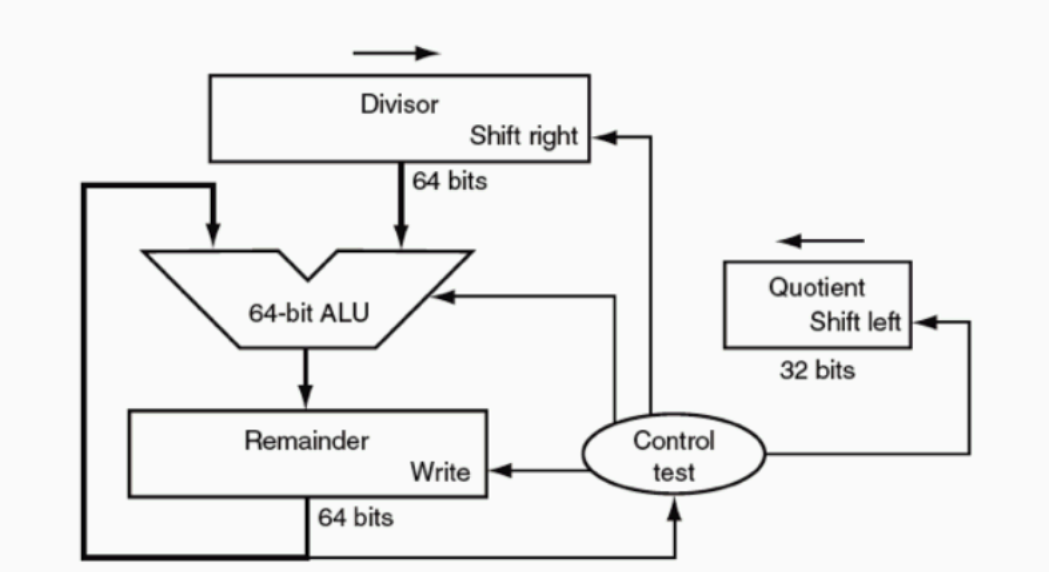
Circuitos Aritméticos, División

- ✓ En general, la división sigue la regla: Dividendo = Cuociente x Divisor + Resto
- ✓ La implementación de la división por hardware imita la técnica aprendida en la básica.
- ✓ La implementación comienza con un registro para el Cuociente de 32 bits inicializado en 0.
- ✓ Cada iteración del algoritmo necesita mover el divisor a la derecha un dígito, es por esto que el divisor es ubicado en la mitad izquierda de un registro de 64 bits.
- ✓ Generalizando, el algoritmo de división se comporta de la siguiente manera:

Circuitos Aritméticos, División



Circuitos Aritméticos, División



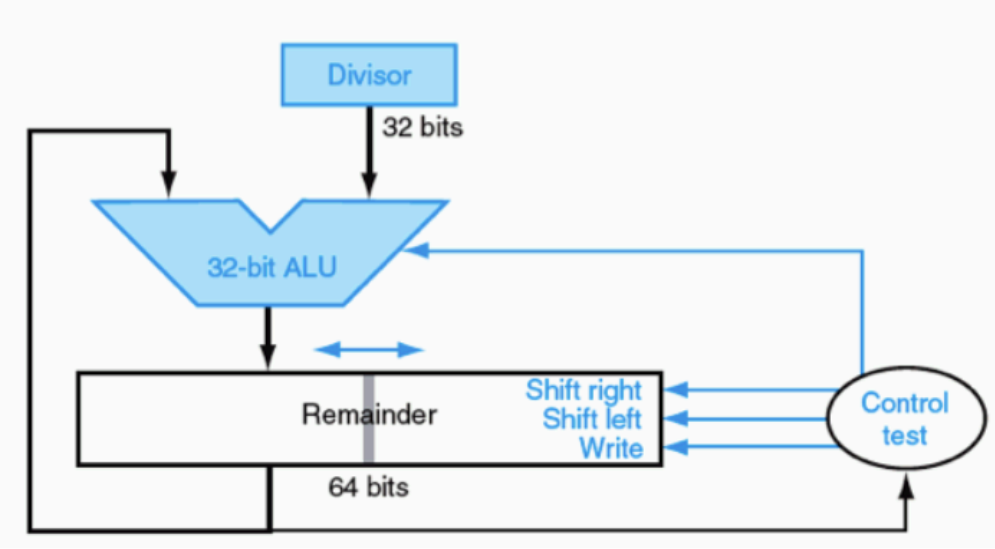
Circuitos Aritméticos, División

- ✓ El algoritmo y el hardware requerido pueden ser mejorados. La mejora viene realizando las operaciones de desplazamiento a los operandos y al cuociente al mismo tiempo que la resta.

- ✓ Se utiliza un registro para el Divisor y para la ALU 32 bits.

- ✓ El registro del resto tiene 64 bits, esto permite (al igual que en la multiplicación) almacenar el cuociente en la mitad derecha de este registro.

Circuitos Aritméticos, División



Circuitos Aritméticos, División

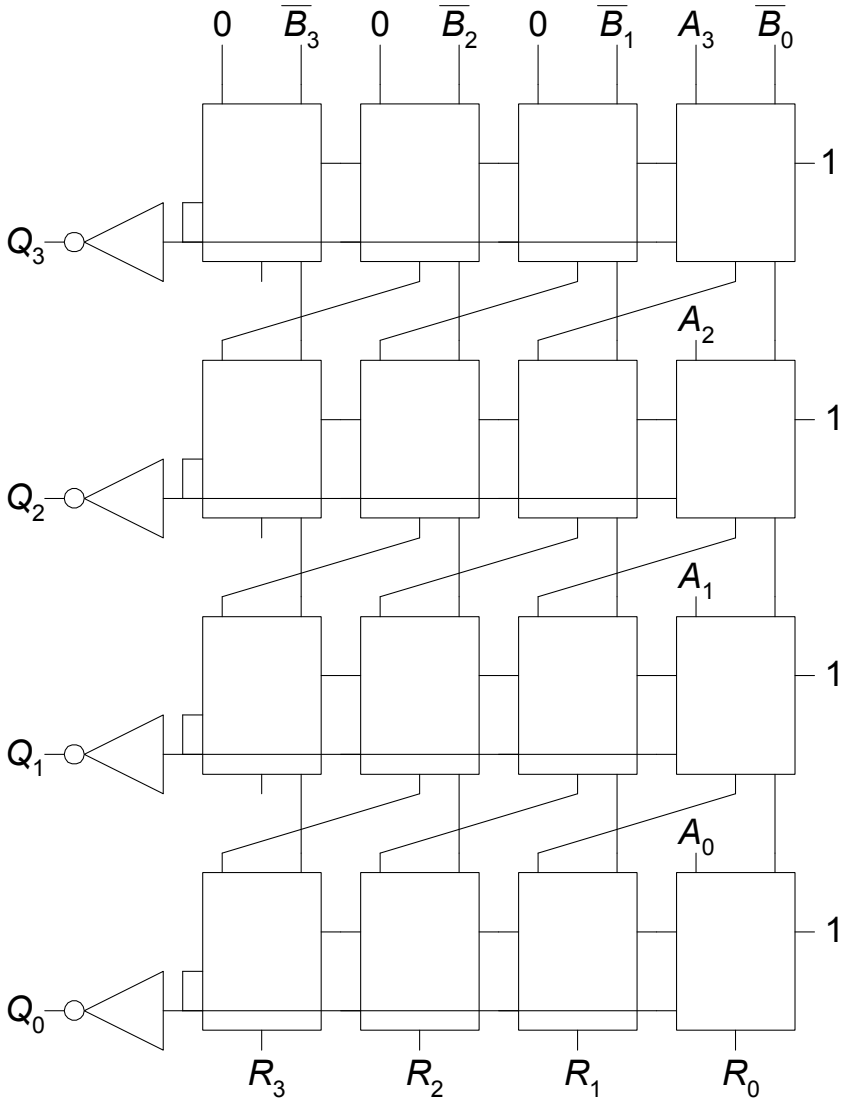
- ✓ Ejemplo: Revisemos el comportamiento del algoritmo para realizar la división 7/2.
- ✓ Notar que el resultado corresponde a que el cuociente vale 3 y el resto 1.
- ✓ El bit examinado para determinar la siguiente etapa está encerrado en un círculo.

Circuitos Aritméticos, División

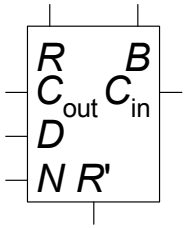
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

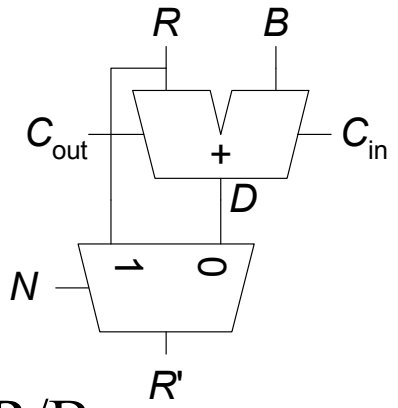
✓ Finalizando, la división es una operación lenta y costosa en términos de hardware, por lo que debería ser utilizada lo menos posible.

Circuitos Aritméticos, División



Legend





A/B = Q + R/B

Algorithm:

R' = 0

for i = N-1 to 0

R = {R' << 1. A_i}

D = R - B

if D < 0, Q_i=0, R'=R

else Q_i=1, R'=D

R'=R

Sistemas Numéricos

- ✓ La representación de los números y como son almacenados en un registro ya ha sido estudiada:
  - ✓ Números Positivos
    - Números binarios sin signo.
  - ✓ Números Negativos
    - Complemento dos.
    - Signo y Magnitud.

- ✓ ¿Qué ocurre con las fracciones?

Sistemas Numéricos

- Dos posible notaciones:
  - Fixed point.
  - Float point.

- Fixed point: 6.75 utilizando 4 bits para entero y fracción:
  - 6.75: 01101100 -> 0110.110

- Otro ejemplo: 7.5 utilizando 4bits para ambas partes.
  - 7.5: 0111.1000

Sistemas Numéricos

- ✓ Representaciones para Negativos:
  - ✓ Signo y Magnitud.
  - ✓ Complemento dos.

- ✓ Ejemplo: -7.5 utilizando 4bits para ambas partes.
  - ✓ S/M: 1111.1000
  - ✓ C2: 01111000 -> 10000111 + 1 = 1000.1000

Sistemas Numéricos

- ✓ Sumar 0.75 + -0.625

	0000.1010	Binary Magnitude
	1111.0101	One's Complement
+	1	Add 1
	1111.0110	Two's Complement

	0000.1100	0.75
+	1111.0110	+ (–0.625)
	10000.0010	0.125

Precisión Punto Flotante

- ✓ Nos centraremos en la precisión simple 32 bits.

1 bit	8 bits	23 bits
S	E	M

- ✓ S: Signo (0: positivo, 1: negativo)
- ✓ E: Exponente sesgado
- ✓ M: Mantisa (magnitud del número normalizado)

Precisión Punto Flotante

- ✓ El exponente sesgado corresponde al valor:

$$E = e + (B^{n-1} - 1)$$

- ✓ Donde e es el exponente real y n el número de bits para representar el exponente (8 para p.s 32).

- ✓ Para precisión simple tenemos: $E = e + 127$

Precisión Punto Flotante

- ✓ La normalización de un número corresponde a escribirlo de la forma:

$$\pm 1.b_1 b_2 b_3 \ldots . b_{23} \times 2^{\pm e}$$

- ✓ En los 23 bits de la mantisa, se almacenan los bits desde el b1 al b23 sin considerar el 1 que está a la izquierda de la coma, llamado bit oculto.

- ✓ La representación IEEE 754 en precisión simple corresponde a:

$$(-1)^{S} \times (1 + Mantisa) \times 2^{(E-127)}$$

Precisión Punto Flotante

- ✓ Por ejemplo: escribir -118,625 en precisión simple 32 bits.
- ✓ El primer paso, será convertir dicho número a base binaria.

118	2
59	0
29	1
14	1
7	0
3	1
2	1
1	1

$$118_{10} = 1110110_2$$

0.625	2
1.25	2
0.5	2
1.00	2

$$0.625_{10} = 0.101_2$$

$$118.625_{10} = 1110110.101_2$$

Precisión Punto Flotante

- ✓ Dicho resultado, lo podemos escribir como:

$$1.110110101 \times 2^{6}$$

- ✓ Es decir, hemos desplazado la coma 6 ubicaciones hacía la izquierda. Con esto podemos calcular el valor de E = 6 + 127 = 133

- ✓ Con esto ya podemos calcular todos los valores correspondientes:
  - ✓ S: 1 (número negativo).
  - ✓ E: 10000101 (133 en binario)
  - ✓ M: 11011010100000000000000

S (1 bit)	E (8 bits)	M (23 bits)
1	10000101	110110101000…000

Precisión Punto Flotante

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

Precisión Punto Flotante

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127

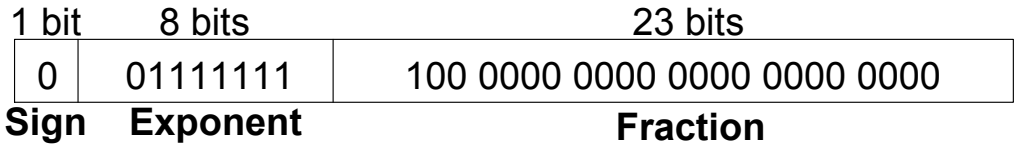
- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

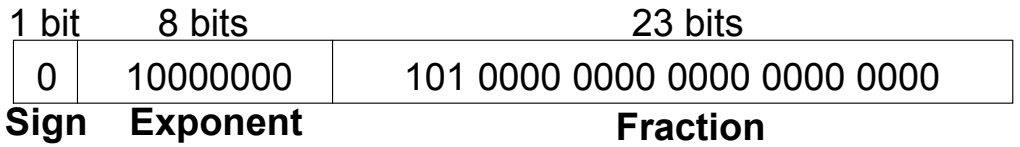
Precisión Punto Flotante

- ✓ Sumar dos números en p.s no resulta tan fácil como por ejemplo en complemento 2.
- ✓ Ejemplo: Sumar los números 0x3FC00000 y 0x4050000.

Precisión Punto Flotante

1. Extract exponent and fraction bits





For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

Precisión Punto Flotante

3. **Compare exponents**

   $127 - 128 = -1$, so shift N1 right by 1 bit

4. **Shift smaller mantissa if necessary**

   shift N1's mantissa: $1.1 >> 1 = 0.11$ $(\times 2^1)$

5. **Add mantissas**

$$\begin{array}{r} 0.11 \times 2^1 \\ +\ 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

Precisión Punto Flotante

6. **Normalize mantissa and adjust exponent if necessary**

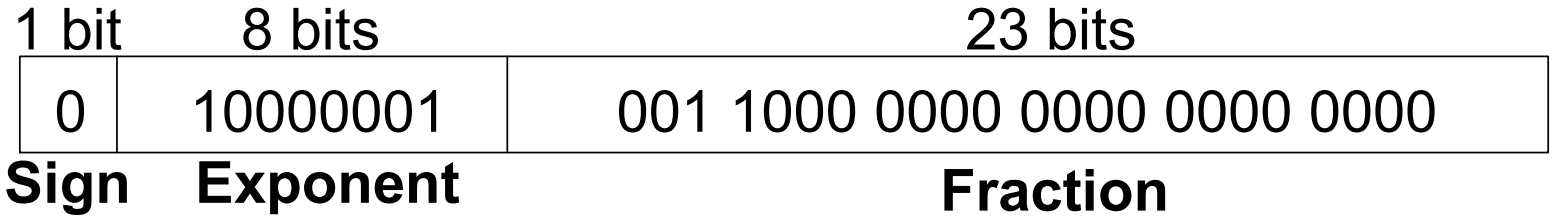
   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. **Round result**

   No need (fits in 23 bits)

8. **Assemble exponent and fraction back into floating-point format**

   $S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$



in hexadecimal: **0x40980000**

Repaso

- ✓ Convertir los siguientes números en expresión signo magnitud “fixed point” utilizando 8 bits para las partes enteras y fraccionarias.
  - ✓ -13,5625
  - ✓ 42,3125
  - ✓ -17,15625

- ✓ Convertirlos en C2.

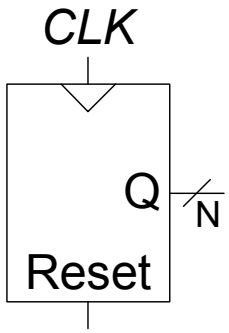
Repaso

- ✓ Convertir los siguientes números representados en C2 fixed point a decimal.
  - ✓ 01001.10000
  - ✓ 1111.1111
  - ✓ 1000.0000
- ✓ Sumar en p.s 32:
  - ✓ 0xC0D20004 + 40DC0004
  - ✓ 0xC0D20004 + 72407020

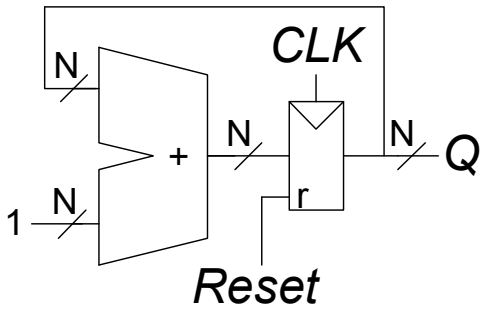
Bloques Secuenciales, Contadores

- ✓ Un contador en un bloque aritmético secuencial que utiliza de entradas el clock y variables de reset y una salida de N bits.
- ✓ Incrementa el valor en cada ciclo de reloj (subida).
- ✓ Ejemplo: Utilizado para generar todas las posibles combinaciones de entradas de variables. 000, 001, 010, 011…..,111, 000…
- ✓ Algunas implementaciones clásicas:
  - ✓ Display de relojes digitales.
  - ✓ Contadores de programa.

Symbol



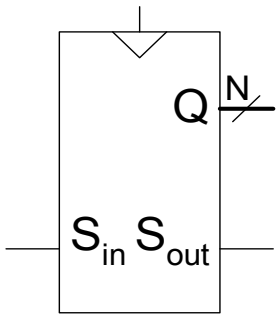
Implementation



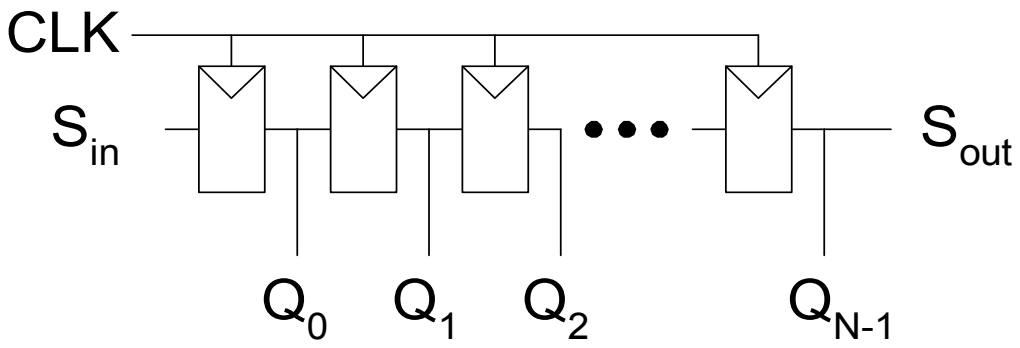
Bloques Secuenciales, Shift Registers

- ✓ Tienen un clock, una entrada serial S_in, una salida S_outs y N salidas paralelas.
- ✓ En cada ciclo (subida) realiza el desplazamiento.

Symbol:

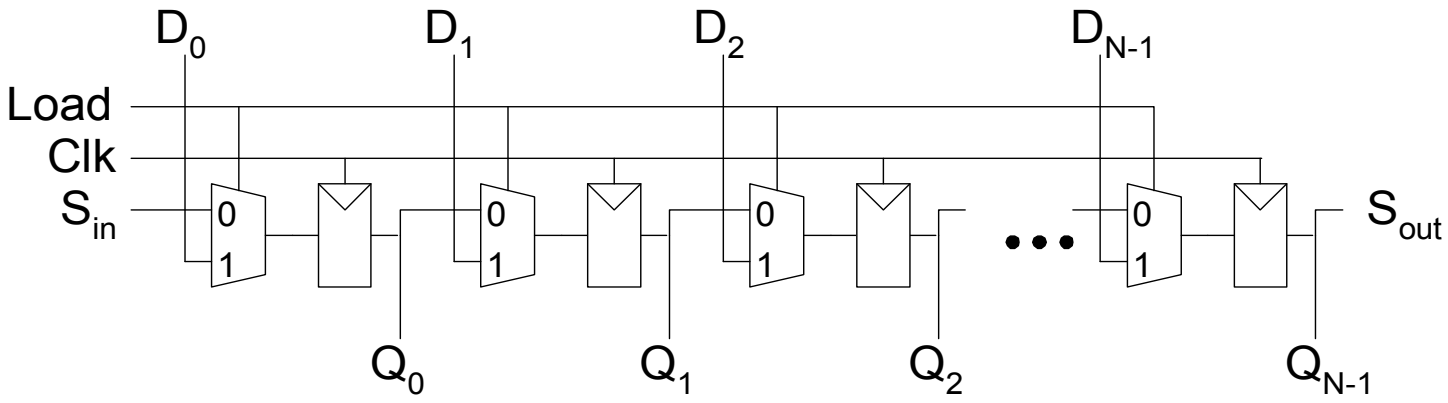


Implementation:

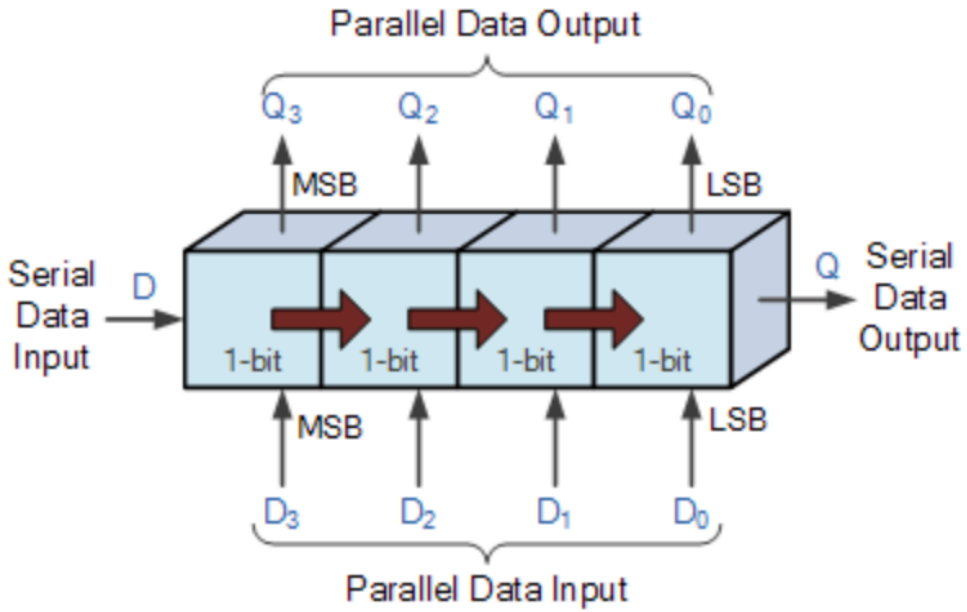


Bloques Secuenciales, Shift Registers Parallel

- ✓ Podemos modificar la implementación anterior para construir un registro “parallel-to-serial” que carga N bits en paralelo y los desplazada en la salida uno a la vez.
- ✓ Si Load = 1, actúa como un registro de N bits.
- ✓ Si Load = 0, actúa como un shift register.

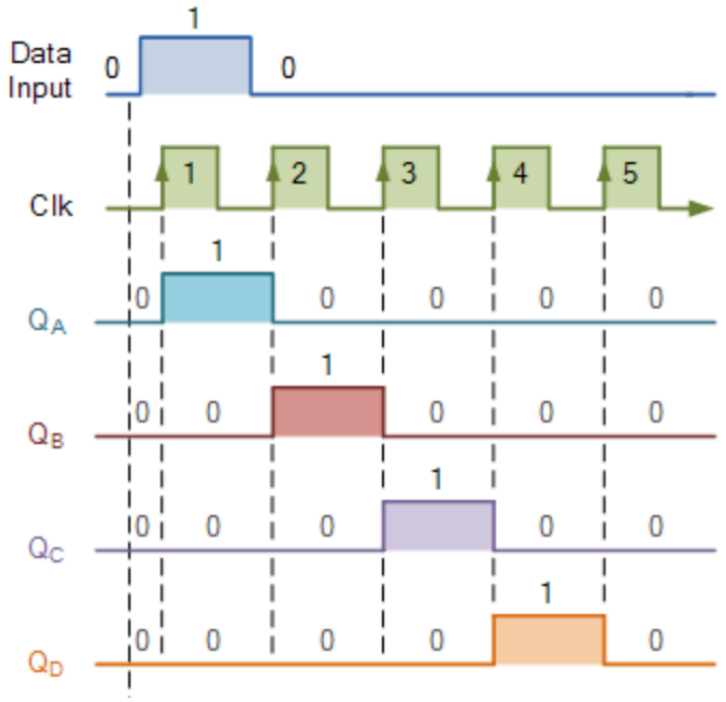


Bloques Secuenciales, Shift Registers Parallel



Bloques Secuenciales, Shift Registers Parallel

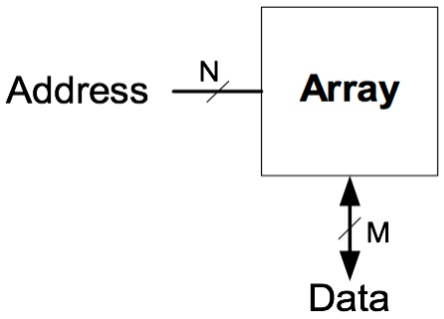
Clock Pulse No	QA	QB	QC	QD
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0



Memoria

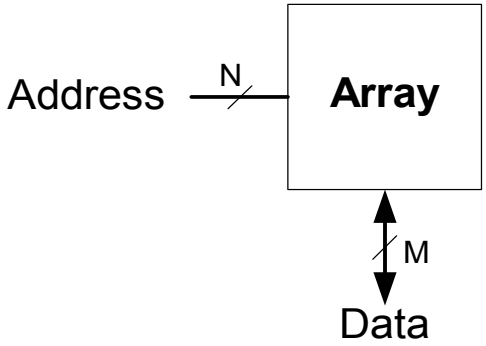
- ✓ El objetivo principal es poder almacenar una mayor cantidad de información.
- ✓ 3 tipos:
  - ✓ Dynamic random access memory (DRAM)
  - ✓ Static random access memory (SRAM)
  - ✓ Read only memory (ROM)

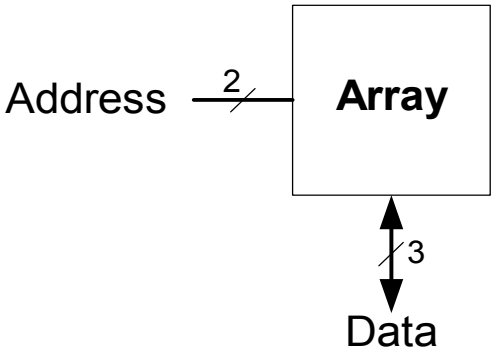
- ✓ M bits de datos / escritos y/o leídos de una dirección única.

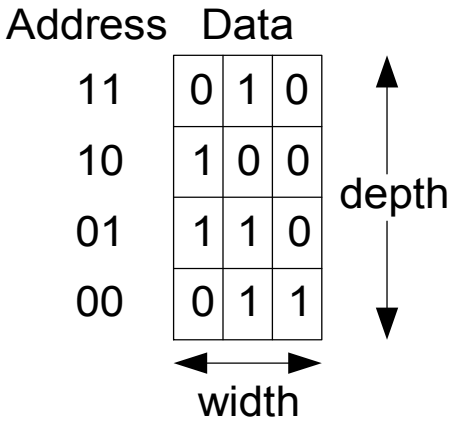


Memoria

- ✓ Podemos verla como un arreglo bi dimensional de celdas de bits.
- ✓ Cada bit es almacenado en una celda.
- ✓ N bits para la dirección, M para los datos.
  - ✓ 2^N filas y M columnas.
  - ✓ Tamaño = 2^N x M

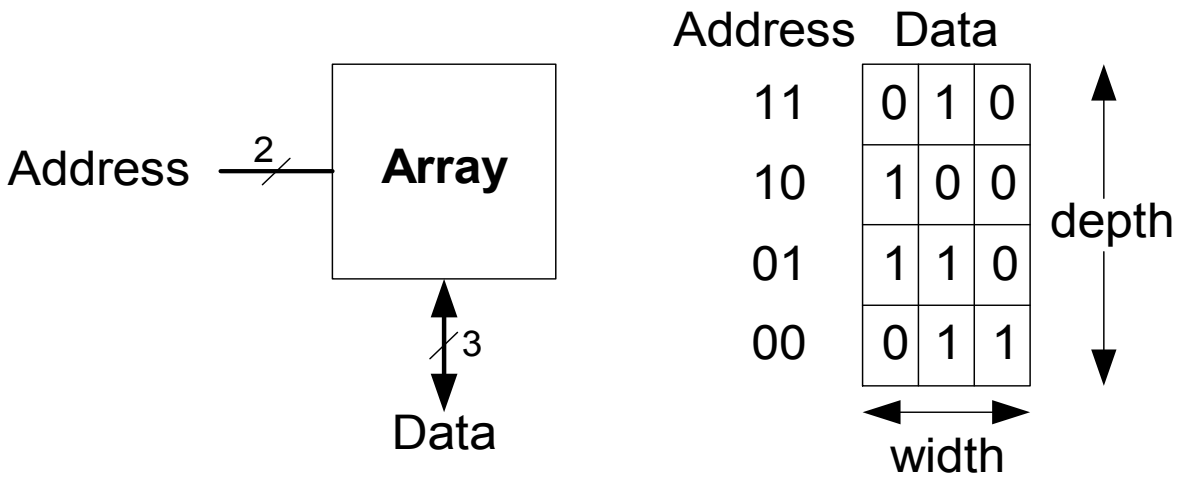




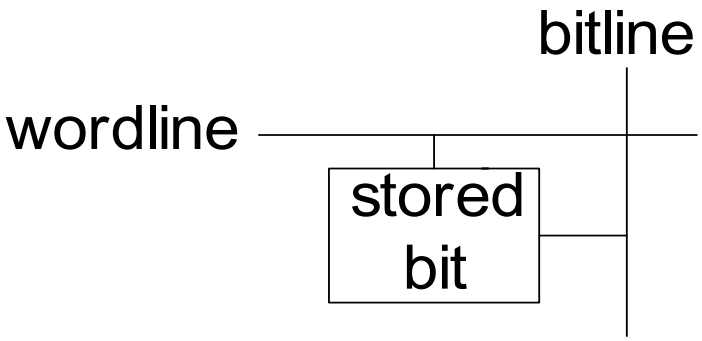


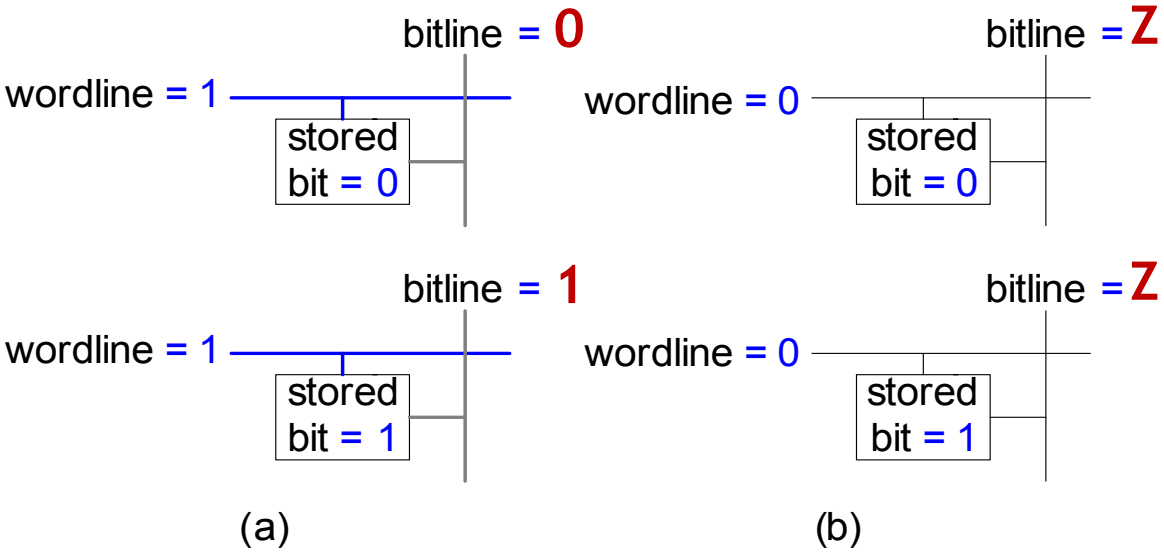
Memoria, Ejemplo

- ✓ 2^2 x 3 bits.
- ✓ Número de palabras: 4.
- ✓ Tamaño de la palabra: 3 bits.
- ✓ El contenido de la dirección 10 es 100.



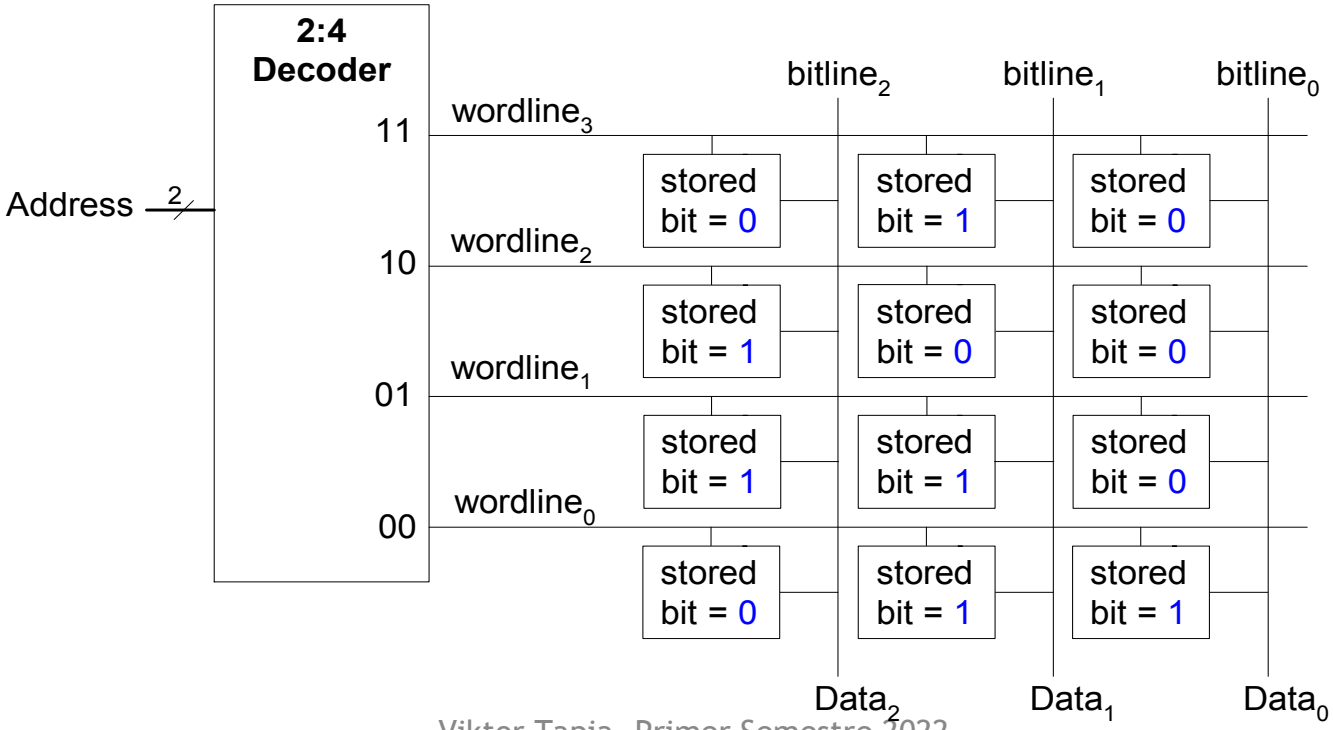
Memoria, Arreglo de celdas de Bits





Memoria, Arreglo de celdas de Bits

- ✓ Wordline:
  - ✓ Una única fila en el arreglo de memoria con única dirección.
  - ✓ Solamente una línea activa a la vez.



Memoria, Tipos de Memoria

- ✓ Random Access Memory (RAM):
  - ✓ Volátil: Pierde el contenido de la memoria cuando se apaga el computador.
  - ✓ Lectura y escritura rápida.
  - ✓ Corresponde a la memoria principal de nuestros computadores.
  - ✓ Se le denomina “random” por la facilidad de acceder a cualquier palabra en memoria.

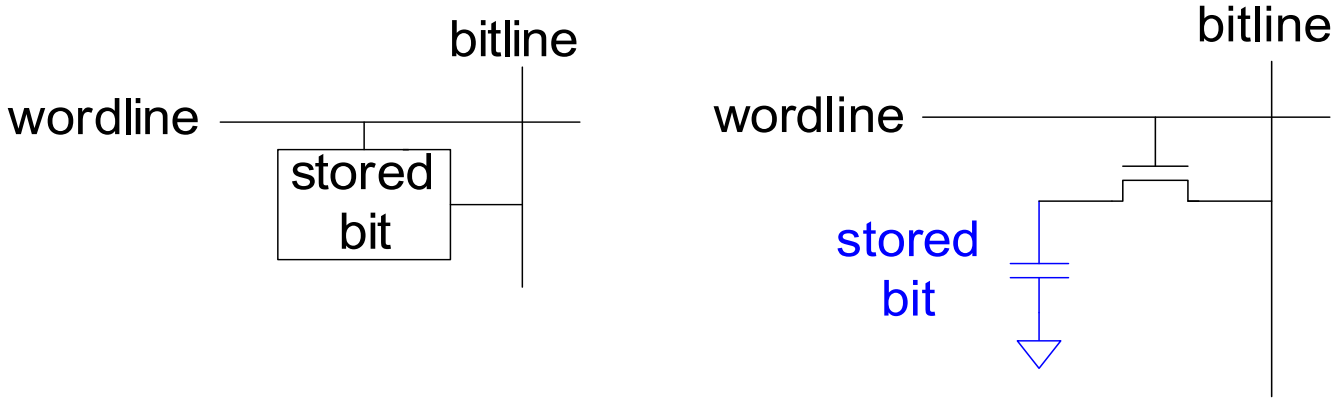
Memoria, Tipos de Memoria

- Read only Memory (ROM):
  - No - Volátil: Cuando se apaga el computador mantiene el contenido de la memoria.
  - Lectura es muy rápida pero escritura es casi imposible o muy lento.
  - Ejemplos: Memorias flash en las cámaras, cámaras digitales.

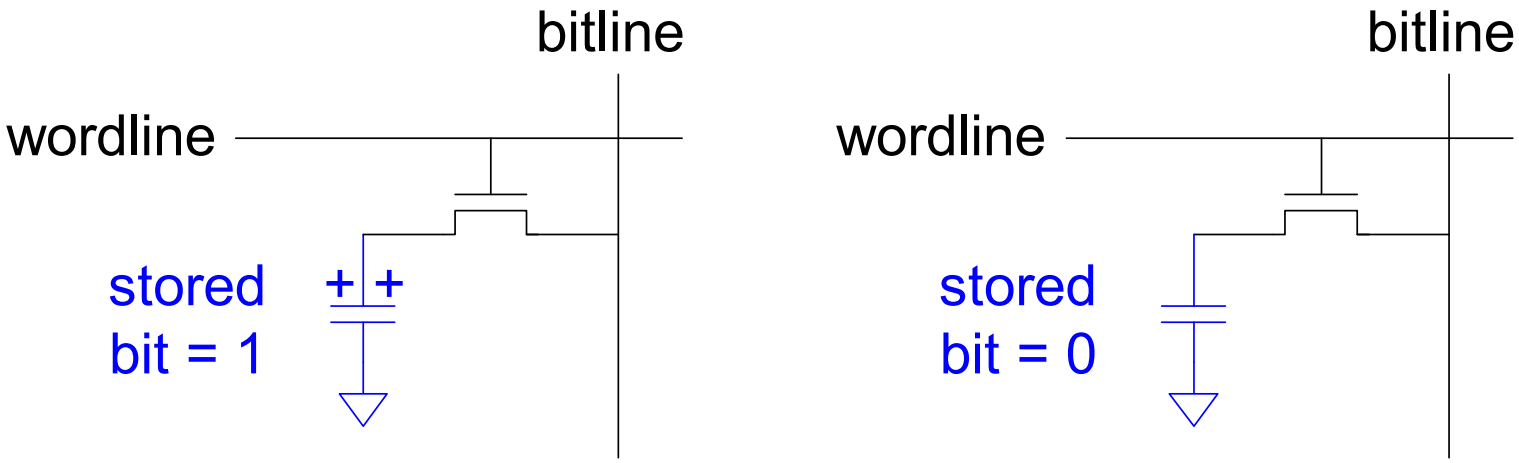
- ¿Cómo almacenan data?
  - DRAM: Utiliza capacitores.
  - SRAM: Utiliza inversores cruzados.

Memoria, DRAM

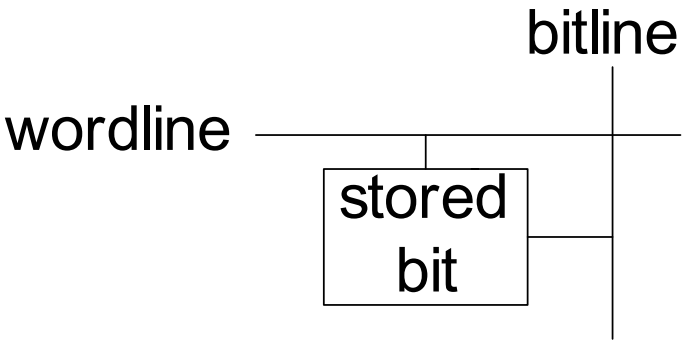
- ✓ Bits almacenados en capacitores.
- ✓ Dinámica debido a que el valor necesita ser refrescado periódicamente y luego leído.

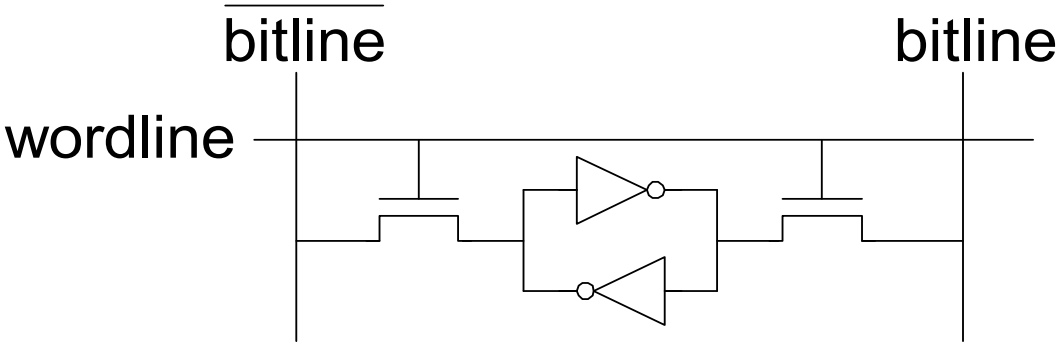


Memoria, DRAM

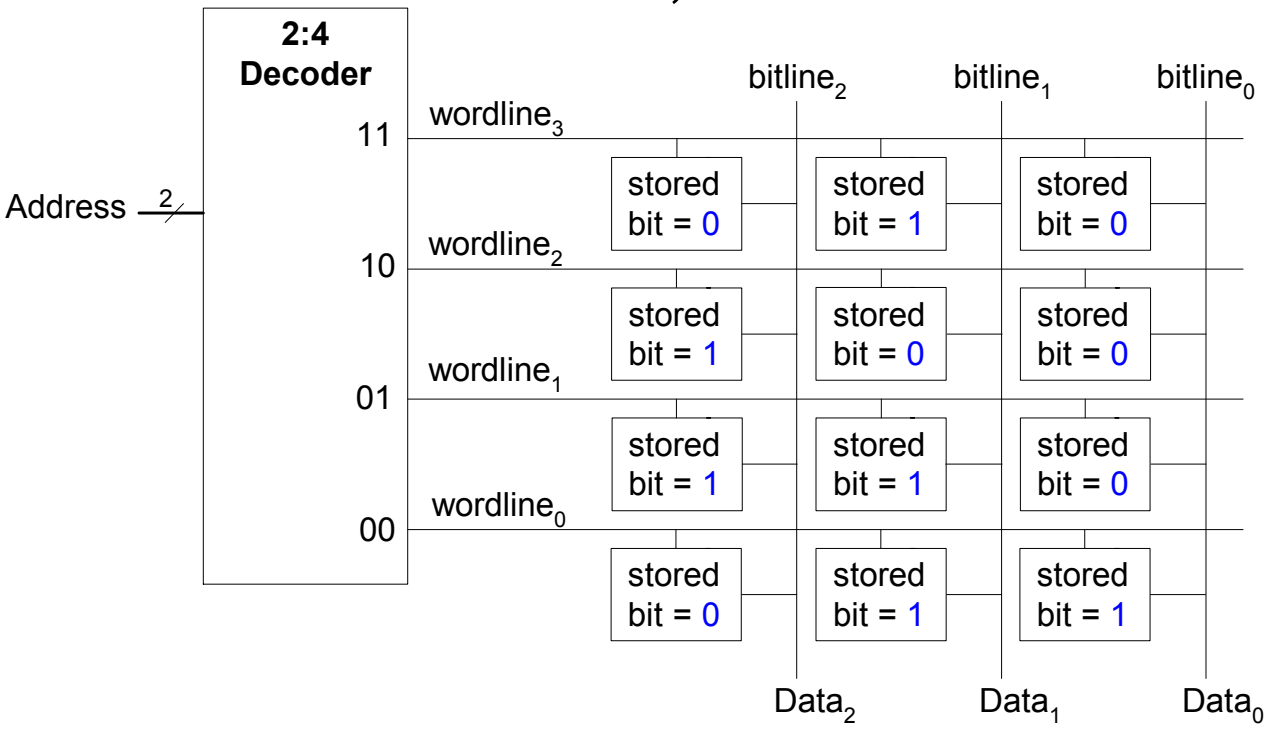


Memoria, SRAM

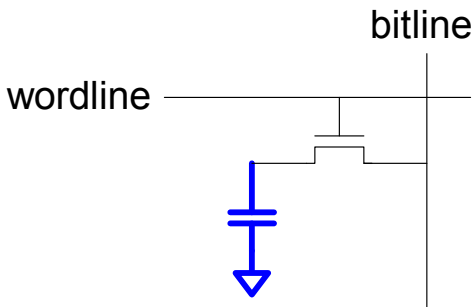




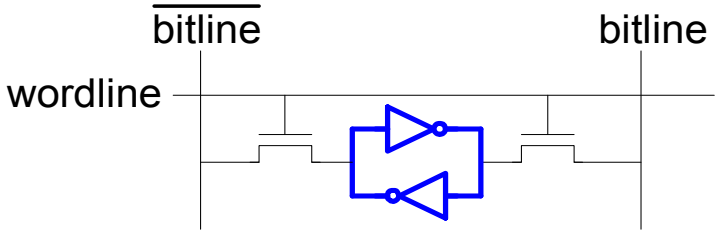
Memoria, Resumen



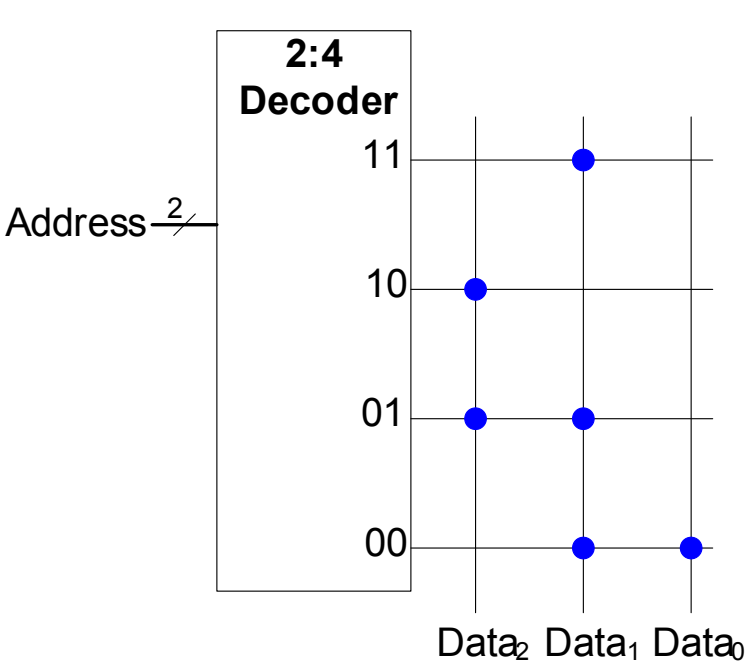
DRAM bit cell:

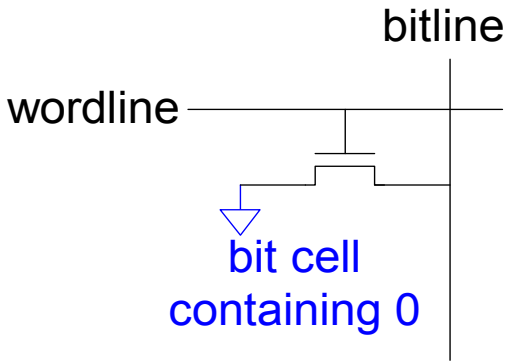


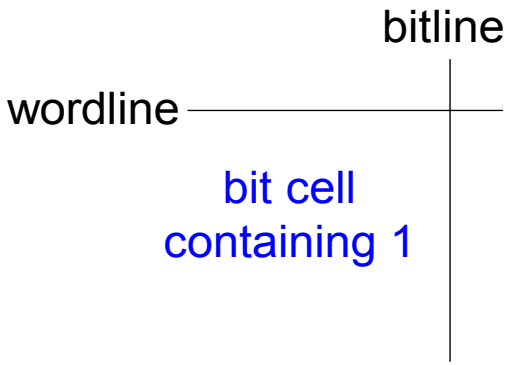
SRAM bit cell:



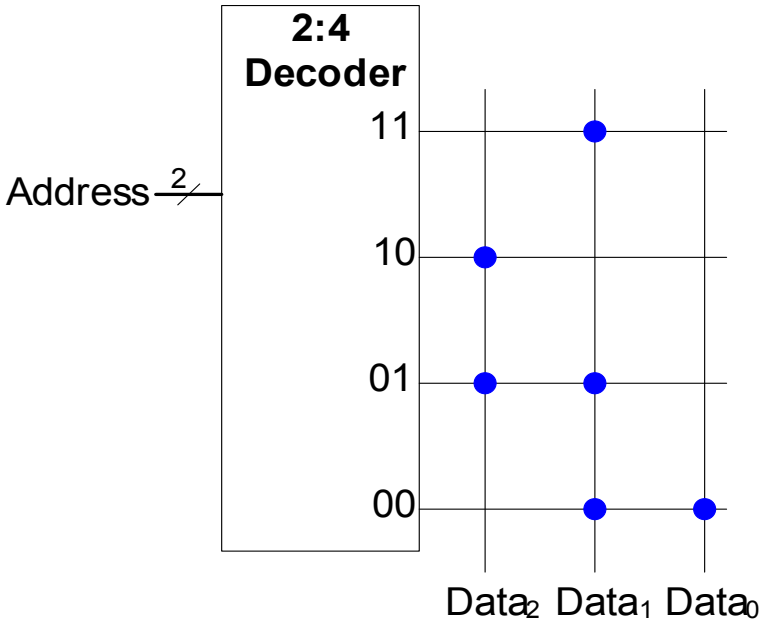
Memoria, Notación para ROM

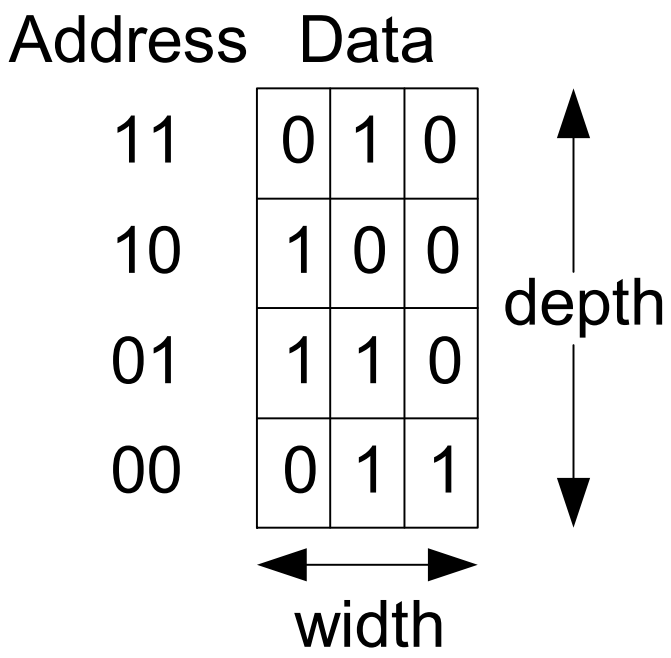




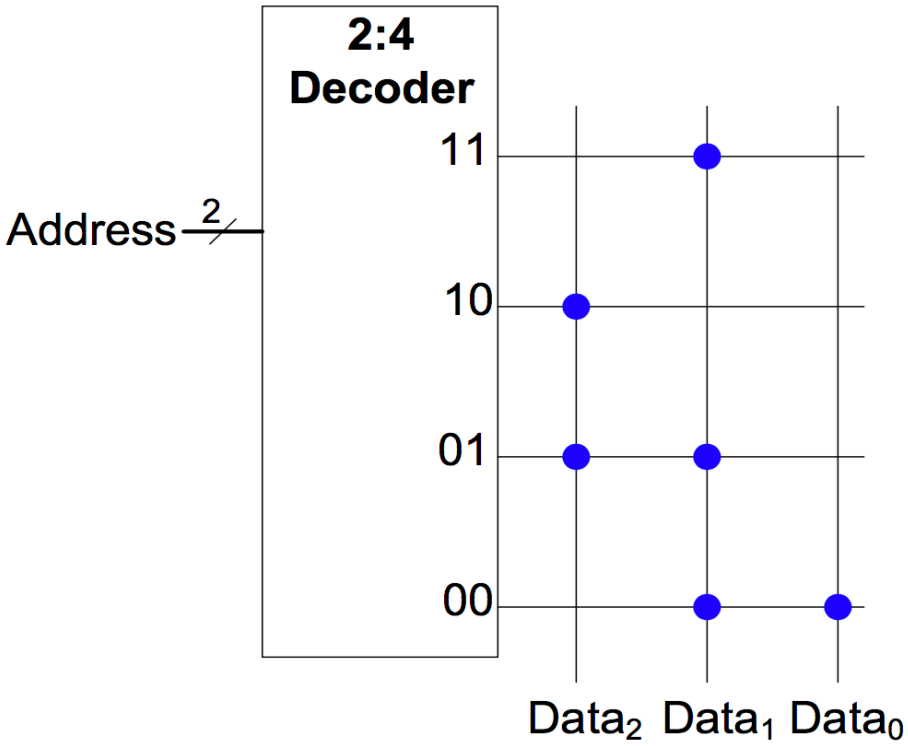


Memoria, Notación para ROM





Memoria, Lógica ROM



$$Data_2 = A_1 \oplus A_0$$

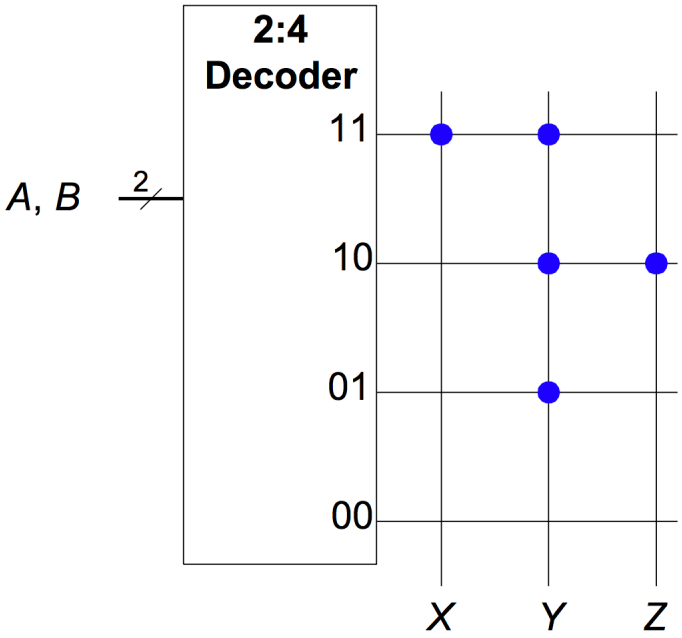
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

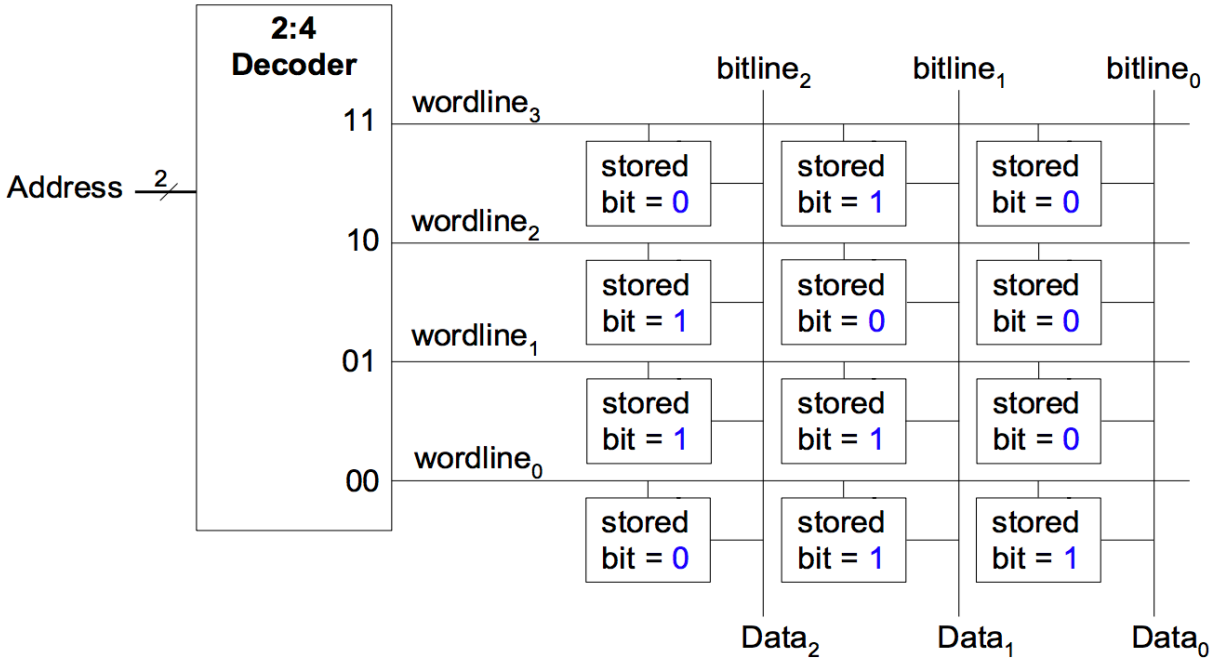
Memoria, Lógica ROM

Implement the following logic functions using a $2^2 \times 3$-bit ROM:

- $X = AB$
- $Y \equiv A + B$
- $Z = A\,B$



Memoria, Lógica



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A}_1 + A_0$$

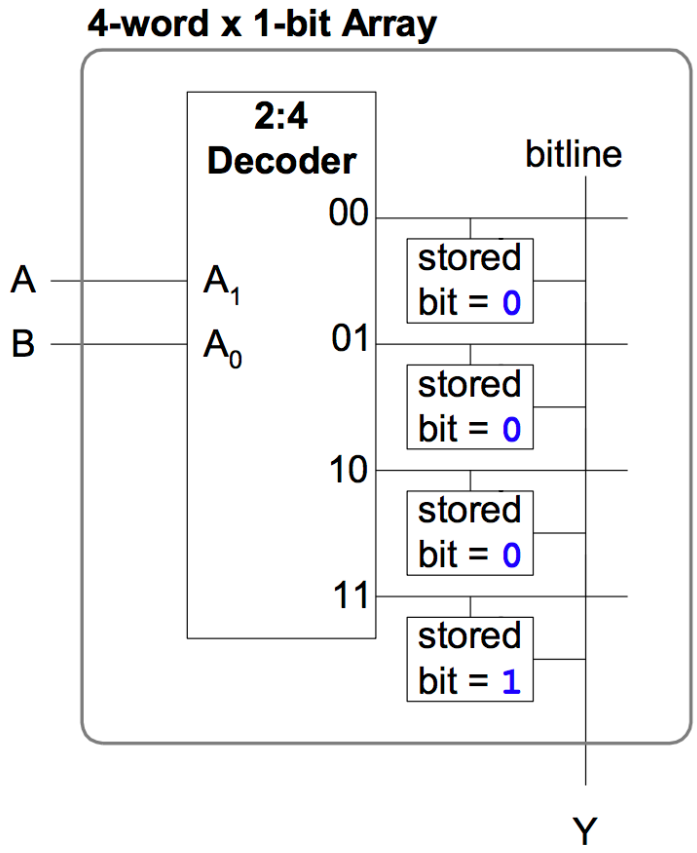
$$Data_0 = \overline{A}_1 \overline{A}_0$$

Memoria, Lógica

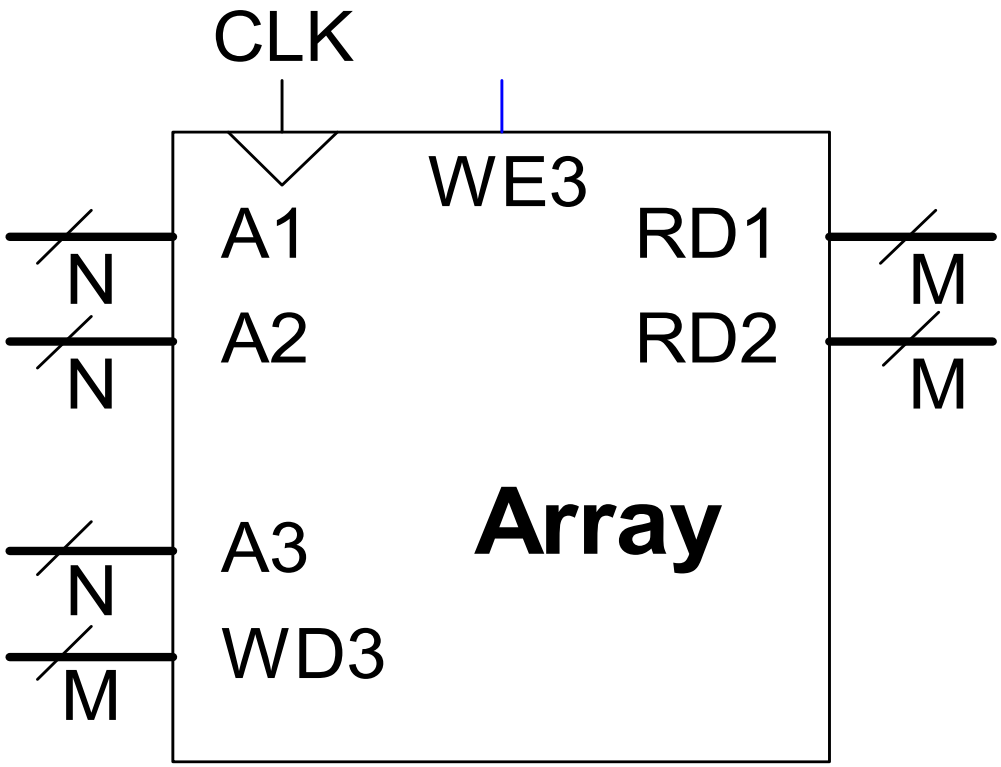
Called *lookup tables* (LUTs): look up output at each input combination (address)

Truth Table

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



Memoria, Componente



Arreglos Lógicos

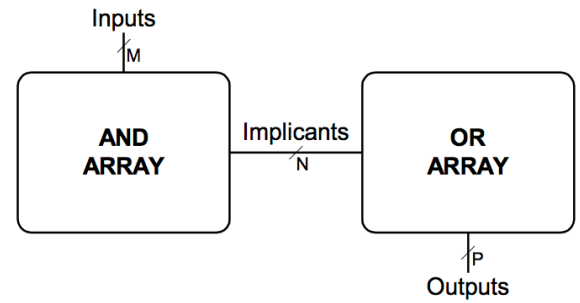
- ✓ PLA's (Arreglos programables)
  - ✓ Compuestos por arreglos de AND's y luego de OR's
  - ✓ Corresponden a circuitos combinacionales

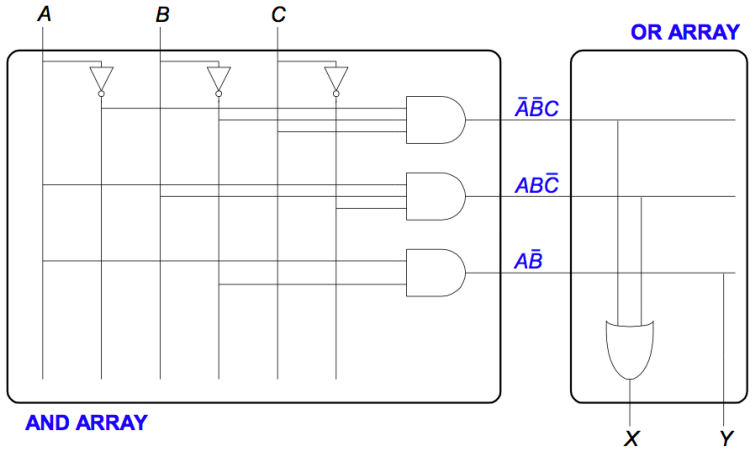
- ✓ FPGA's (Field programmable gate arrays)
  - ✓ Arreglos de elementos estructurados
  - ✓ Incluye sistemas secuenciales

Arreglos Lógicos

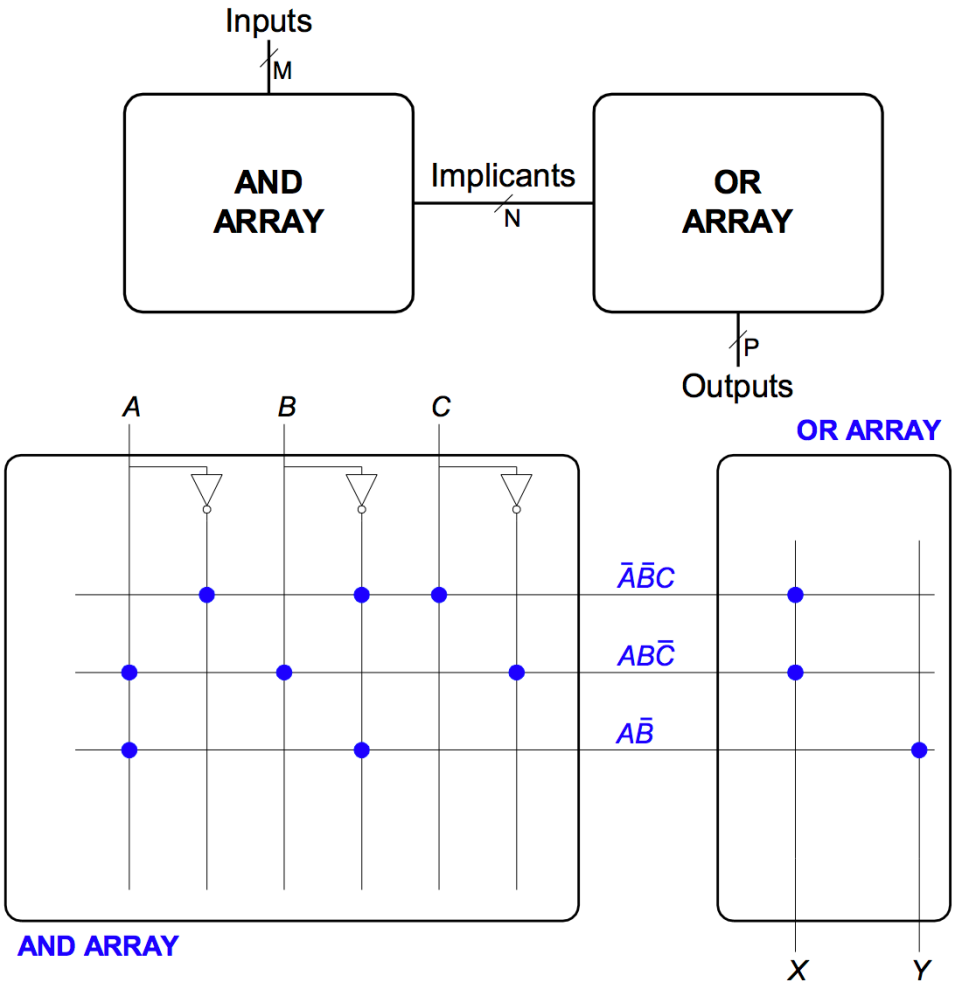
$$X = \bar{A}\bar{B}C + AB\bar{C}$$

$$Y = A\bar{B}$$





Arreglos Lógicos



UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
CAMPUS SAN JOAQUÍN

Arquitectura y Organización de Computadores

Primer Semestre 2022