DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 14 - Resumen del Curso

Profesor: Germán Leandro Contreras Sagredo

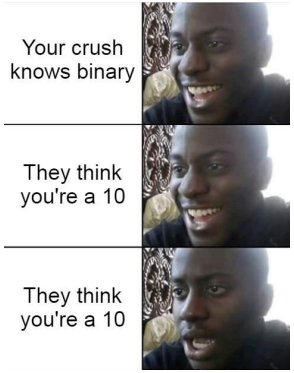
Propósito del material

- El objetivo de este material es que puedan realizar un estudio **rápido** de los contenidos más relevantes del curso.

- La idea es que con este puedan identificar **qué contenidos les falta profundizar**, de ser el caso, se espera que no solo se queden con lo que se encuentra aquí, sino que además acudan a las clases y apuntes correspondientes.

- También se espera que sirva como guía rápida mientras estén desarrollando ejercicios de los compilados.

Representaciones numéricas



- Contenido que encuentran en:
  - **Clase 1 - Representaciones Numéricas I**
  - **Clase 6 - Representaciones Numéricas II**
  - **01 - Representaciones Numéricas Parte 1 - Números Enteros** (Apuntes)
  - **02 - Representaciones Numéricas Parte 2 - Números Racionales** (Apuntes)




Representaciones numéricas - Números enteros

- En este curso, nos enfocamos en la **representación posicional binaria** por su utilidad en circuitos digitales y, posteriormente, componentes del computador básico.

- **Fórmula general de transformación de bases**

$$\sum_{k=0}^{n-1} s_k \times b^k$$

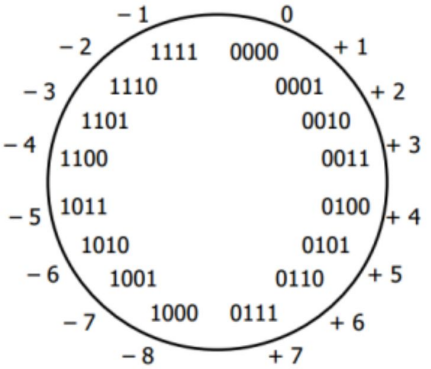
s = Símbolo (dígito).
n = Cantidad de dígitos en la secuencia.
b = Base numérica (o número de dígitos).
k = Posición del dígito en la secuencia, siendo 0 la posición del extremo derecho.

Representaciones numéricas - Números enteros

- **Complemento de 2:** Representación utilizada para números **enteros**. Se obtiene el complemento de 1 (complemento de cada dígito del número) y se suma una unidad al final. Esto asegura que $x + C_2(x) = 0$ (el bit de *carry* restante se descarta).

N	0101
-N	1011
N + (-N)	0000

- **Contras del complemento de 2**
  - Representación **desbalanceada** (un número negativo adicional).
  - ***Overflow*:** Operación cuyo resultado no es representable con la cantidad de bits disponible resultan en un valor erróneo.



Representaciones numéricas - Números enteros

- **Conversión entre base binaria y hexadecimal**
  - **Hexadecimal a binario:** Cada dígito hexadecimal se representa en su valor binario con cuatro dígitos. Se concatenan los resultados para el valor final.

$$0x9F2 = \left\{ \begin{array}{lcl} 0x9 & = & 1001b \\ 0xF & = & 1111b \\ 0x2 & = & 0010b \end{array} \right\} \rightarrow 0x9F2 = 100111110010b$$

  - **Binario a hexadecimal:** Se agrupan cuatro dígitos binarios y se representan en el valor hexadecimal. Se concatenan los resultados para el valor final.

$$100111110010b = (1001)(1111)(0010) = (0x9)(0xF)(0x2) = 0x9F2$$

* Se puede hacer lo mismo entre la base binaria y la base octal, haciendo las conversiones con 3 dígitos en vez de 4.

Representaciones numéricas - Números racionales

- Se pueden representar en base binaria, pero **números finitos en base decimal pueden ser infinitos en base binaria**. Esto hace que la cantidad de bits para representar estos números sea significativa para reducir el error.

- **Representaciones de números racionales en base binaria**
  - **Punto fijo:** Dados *N* bits, estos se dividen de forma **fija** para representar signo, parte entera y parte decimal.
    Ej: *N* = 8 bits = 1 bit signo (*s*) + 3 bits parte entera (*t*) + 4 bits fracción (*f*)
    10,110b = $0_s010_t0110_f$ -1,0011b = $1_s001_t0011_f$

  - **Punto flotante:** Dados *N* bits, los dividimos en un **significante/mantisa** y un **exponente**, ambos con signo.
    Ej: N = 8 bits = 1 bit signo significante (*ss*) + 3 bits significante (*s*) + 1 bit signo exponente (*se*) + 3 bits exponente (*e*) → -0,00011b = $-1{,}1 * 10^{-100} = 1_{ss}110_s1_{se}100_e$

Representaciones numéricas - Números racionales

- Usamos punto flotante por un **mayor rango** (en perjuicio de la precisión).

- Representación de punto flotante más utilizada: **Estándar IEEE754**
  - **Float (*Single Precision Floating Point*)**
    - 1 bit de signo.
    - 8 bits de exponente **desfasado** en 127 para obviar el bit de signo.
    - 23 bits de significante/mantisa **normalizado:** Tiene precisión de 24 bits ya que **asume que todo significante parte en 1**.
    - Exponente 00000000 reservado para representar el 0 (significante con solo bits de 0) y el exponente 11111111 para +-Infinity (significante con solo bits de 0) y NaN (significante con al menos un bit igual a 1).
    - $X = (-1)^{signo} * 1.significante * 10^{(exponente - 127)b}$
      Ej: $0.00101b = (-1)^{0} * 1.01 * 10^{01111100} =$
      $00111111000100000000000000000000_{IEEE754}$

Representaciones numéricas - Números racionales

- Representación de punto flotante más utilizada: **Estándar IEEE754**
  - **Double (*Double Precision Floating Point*)**
    - 1 bit de signo.
    - 11 bits de exponente **desfasado** en 1023 para obviar el bit de signo.
    - 52 bits de significante/mantisa **normalizado:** Tiene precisión de 53 bits ya que **asume que todo significante parte en 1**.
    - Presenta un error de precisión mucho menor a float (error de float igual a 10^{-23}; error de double igual a 10^{-52}), pero requiere del doble de bits para ser representado.

Representaciones numéricas - Números racionales

- Ejemplo de pérdida de precisión en floats: $2^{30} + 5$

$2^{30} \rightarrow (1)^0 * 1.0 * 10^{(157 - 127)} = 1 * 1.0 * 10^{157b} = 1^0 * 1.0 * 10^{10011101}$

$(2^{30})_{IEEE754} =$ 01001110100000000000000000000000$_{IEEE754}$

$5 \rightarrow (1)^0 * 1.01 * 10^{(129 - 127)} = 1 * 1.01 * 10^{129b} = 1^0 * 1.01 * 10^{10000001}$

$(5)_{IEEE754} =$ 01000000101000000000000000000000$_{IEEE754}$

- Para sumarlos, igualamos la potencia del número menor a la del mayor:

$5 = (1)^0 * 1.01 * 10^{(129 - 127)}$

$2^{30} + 5$

$= (1)^0 * 0.00000000000000000000000000101 * 10^{(157 - 127)}$

$= (1)^0 * 1.00000000000000000000000000101 * 10^{(157 - 127)}$

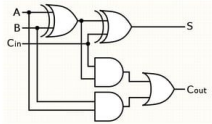
$= (1)^0 * 1.00000000000000000000000$**0000101** $* 10^{10000001}$

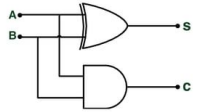
$=$ 01000000100000000000000000000000$_{IEEE754}$

* Estos bits se descartan porque la mantisa acepta hasta 23 bits.

Operaciones aritméticas y lógicas

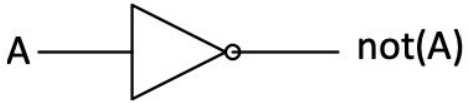
- Contenido que encuentran en:
  - **Clase 2 - Operaciones Aritméticas y Lógicas**
  - **03 - Operaciones Aritméticas y Lógicas**

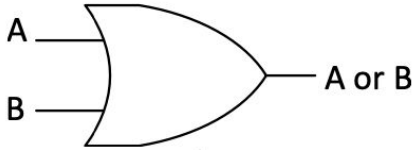


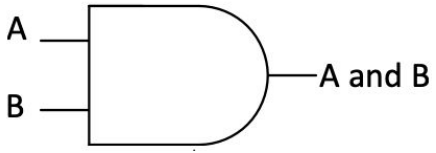


Operaciones aritméticas y lógicas - Compuertas lógicas

- **Compuertas lógicas:** Base de todos los componentes del computador básico: `NOT`, `OR`, `AND`.







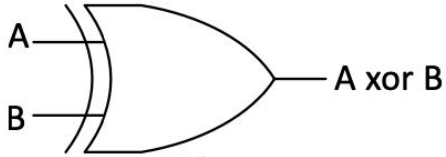
A	not(*A*)
1	0
0	1

A	*B*	*A* or *B*
1	1	1
1	0	1
0	1	1
0	0	0

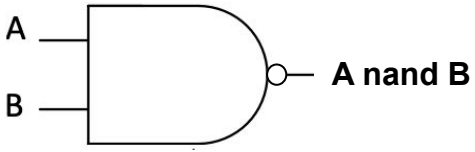
A	*B*	*A* and *B*
1	1	1
1	0	0
0	1	0
0	0	0

Operaciones aritméticas y lógicas - Compuertas lógicas

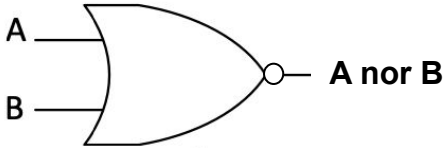
- **Compuertas lógicas:** Base de todos los componentes del computador básico: `XOR`, `NAND`, `NOR`, `XNOR`.



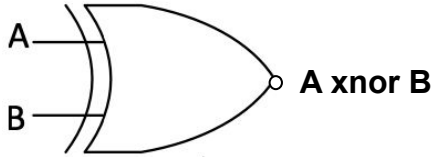
A	*B*	*A* xor *B*
1	1	0
1	0	1
0	1	1
0	0	0



A	*B*	*A* nand *B*
1	1	0
1	0	1
0	1	1
0	0	1



A	*B*	*A* nor *B*
1	1	0
1	0	0
0	1	0
0	0	1



A	*B*	*A* xnor *B*
1	1	1
1	0	0
0	1	0
0	0	1

Operaciones aritméticas y lógicas - *Minterms* y *maxterms*

- Técnicas utilizadas para obtener la expresión lógica equivalente a una tabla de verdad.

- La expresión resultante se puede traducir directamente a un circuito eléctrico al utilizar solo conectivos lógicos `AND`, `OR`, `NOT`.

- Los *minterms* se basan en las condiciones (señales de entrada) que derivan en un valor de verdad igual a 1; los *maxterms* en las que derivan en un valor de verdad igual a 0. Conviene usar *minterms* cuando menos de la mitad de las filas de la tabla de verdad tienen *output* igual a 1; en otro caso, conviene usar *maxterms*. Si se tiene la misma cantidad de filas con *output* igual a 0 y 1, entonces se puede usar cualquiera de los dos.

Operaciones aritméticas y lógicas - *Minterms* y *maxterms*

- ***Minterms*:** Solo se consideran las filas cuyo valor de verdad sea 1. Las condiciones de cada fila de la tabla se conectan con conectivos AND; las condiciones falsas (iguales a 0) se niegan. La expresión de cada fila se conecta con conectivos OR.

(NOT(A) AND B AND NOT(C)) OR (A AND NOT(B) AND C) OR (A AND B AND C)

$$\equiv$$

$$(\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) + (A \cdot B \cdot C)$$

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Operaciones aritméticas y lógicas - *Minterms* y *maxterms*

- ***Maxterms*:** Solo se consideran las filas cuyo valor de verdad sea 0. Las condiciones de una fila de la tabla se conectan con conectivos OR; las condiciones verdaderas (iguales a 1) se niegan. La expresión de cada fila se conecta con conectivos AND.

$$(A \text{ OR } B \text{ OR } C) \text{ AND } (A \text{ OR } B \text{ OR NOT}(C)) \text{ AND } (A \text{ OR NOT}(B) \text{ OR NOT}(C)) \text{ AND } (\text{NOT}(A) \text{ OR } B \text{ OR } C) \text{ AND } (\text{NOT}(A) \text{ OR NOT}(B) \text{ OR } C)$$

$$\equiv$$

$$(A+B+C) \cdot (A+B+\overline{C}) \cdot (A+\overline{B}+\overline{C}) \cdot (\overline{A}+B+C) \cdot (\overline{A}+\overline{B}+C)$$

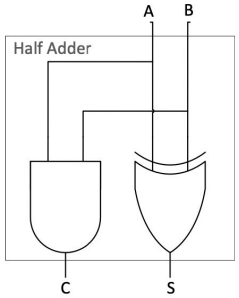
A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

* También se puede construir usando el mismo formato de *minterms*, pero aplicando una negación al final y aplicando Ley De Morgan.

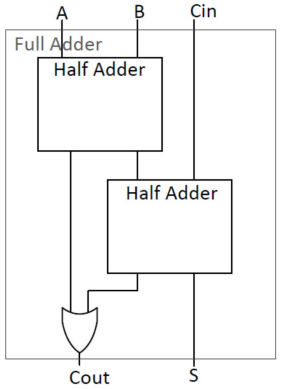
Operaciones aritméticas y lógicas - Componentes aritméticas

- ***Half-Adder***: "Medio sumador", suma dos bits sin considerar bit de *carry*.

A	*B*	*S*	*C*
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0



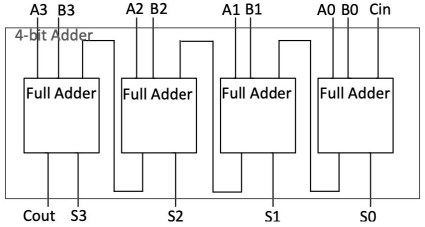
- ***Full-Adder***: Sumador completo, suma dos bits y considera bit de *carry* (señal C_{in}).

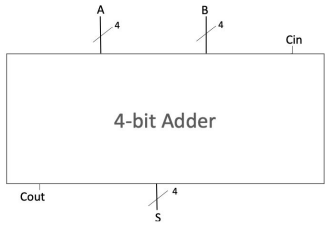


*** El *carry* de salida puede ser generado con la suma de *A* y *B* o con la suma entre dicho resultado y el *carry* de entrada, razón por la que C_{out} es la compuerta OR de estos dos resultados**

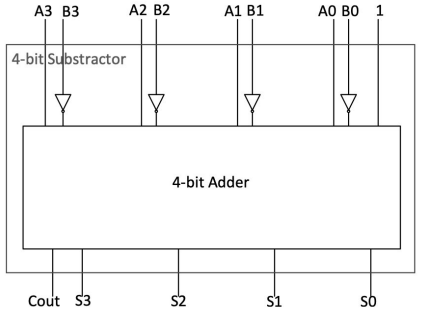
Operaciones aritméticas y lógicas - Componentes aritméticas

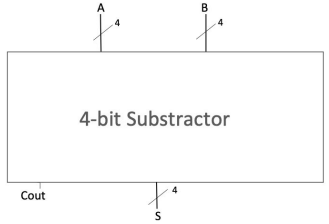
- **Sumador de 4 bits:** *Full-Adders* conectados a través de la señal C_{in}. Se extiende a más bits de la misma forma.





- **Restador de 4 bits:** Sumador de 4 bits pero con entrada *B* transformada a su complemento de dos.

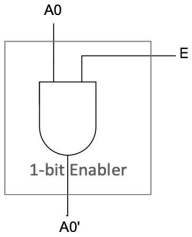




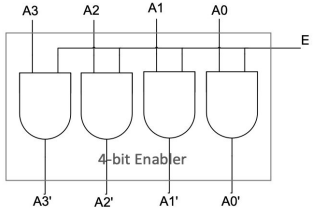
*** Notación de buses utilizada para facilitar la abstracción de la cantidad de señales.**

Operaciones aritméticas y lógicas - Componentes aritméticas

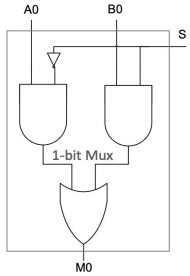
- ***Enabler***: Componente que habilita o no el paso de una señal.



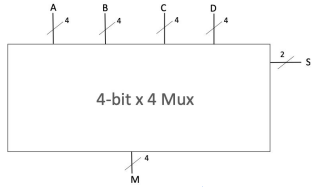
E	A_0'
0	0
1	A_0

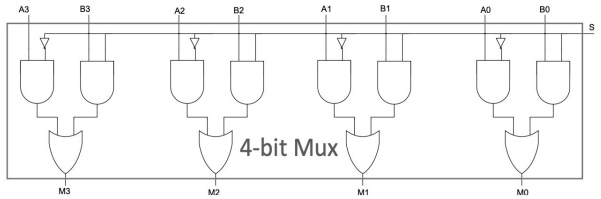


- **Multiplexor/Mux:** Componente que selecciona como salida una de un conjunto de señales de entrada.



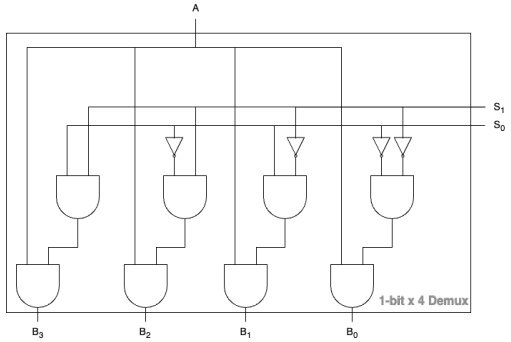
S	M_0
0	A_0
1	B_0





Operaciones aritméticas y lógicas - Componentes aritméticas

- **De-Multiplexor/Demux:**
  Componente que transmite una señal de entrada a una de múltiples salidas. En el resto de salidas no seleccionadas se transmite un 0.

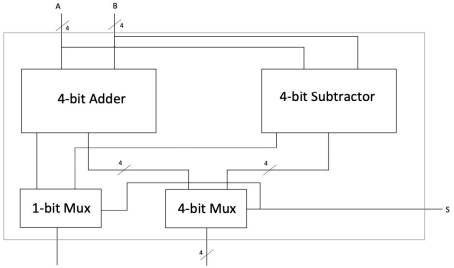


S_1	S_0	Output
0	0	$B_0 = A$
1	0	$B_1 = A$
0	1	$B_2 = A$
1	1	$B_3 = A$

Operaciones aritméticas y lógicas - Componentes aritméticas

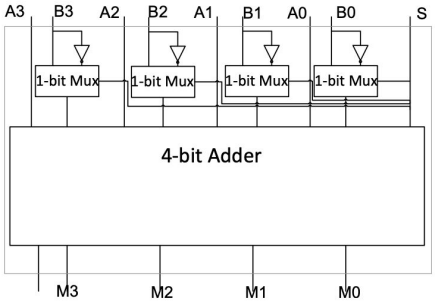
Sumador-Restador

- No optimizado.



S	M	C_{out}
0	A+B	A+B C_{out}
1	A-B	A-B C_{out}

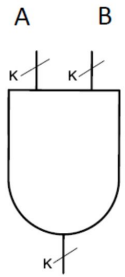
- Optimizado.



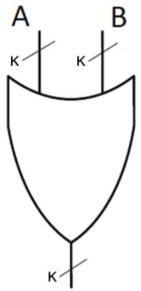
S	M	Cout
0	A+B	A+B C_{out}
1	A-B	A-B C_{out}

Operaciones aritméticas y lógicas - Componentes aritméticas

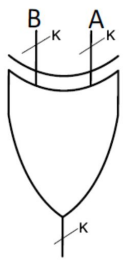
- **Operadores *bitwise*:** Componentes que extienden las operaciones de compuertas lógicas a señales de más de 1 bit.



A and B

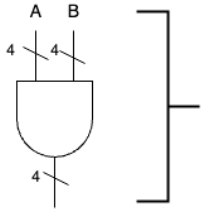


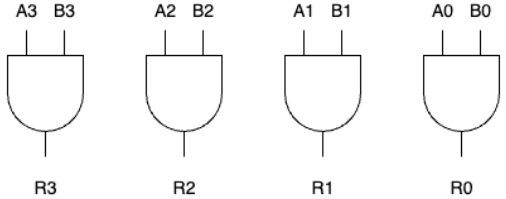
A or B



A xor B

not(A)



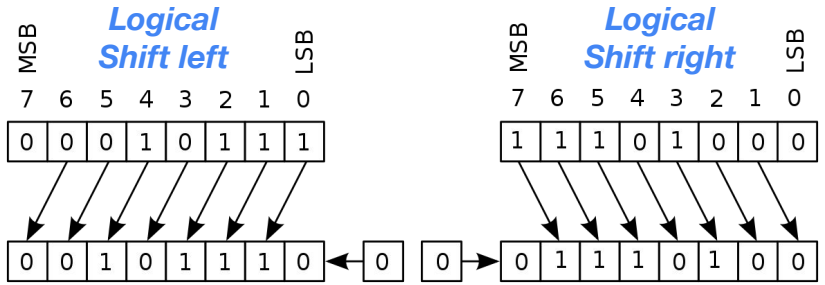


R = (A3 AND B3)(A2 AND B2)(A1 AND B1)(A0 AND B0) = R3R2R1R0

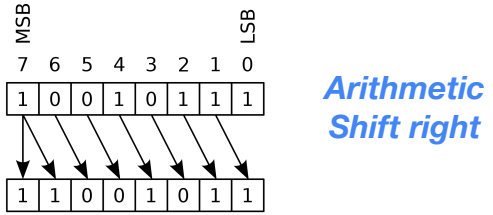
* Ejemplo de construcción de componente *bitwise* AND.

Operaciones aritméticas y lógicas - Componentes aritméticas

- ***Logical shifting components***: Componentes que desplazan los bits de un número a la derecha o a la izquierda, equivalente a dividir o multiplicar por 2 de forma respectiva.
  - shift_left(0100b) = 1000b
  - shift_left(1001b) = 0010b
  - shift_right(0100b) = 0010b
  - shift_right(1001b) = 0100b

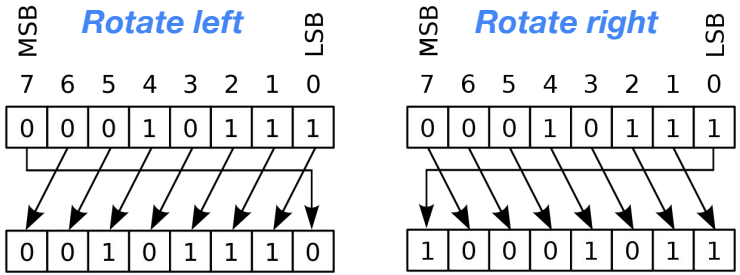


- ***Arithmetic shift right***: Componente que realiza *shift right* manteniendo el signo (bit más significativo).
  - shift_arithmetic_right(1100b) = 1110b
  - shift_arithmetic_right(1001b) = 1100b



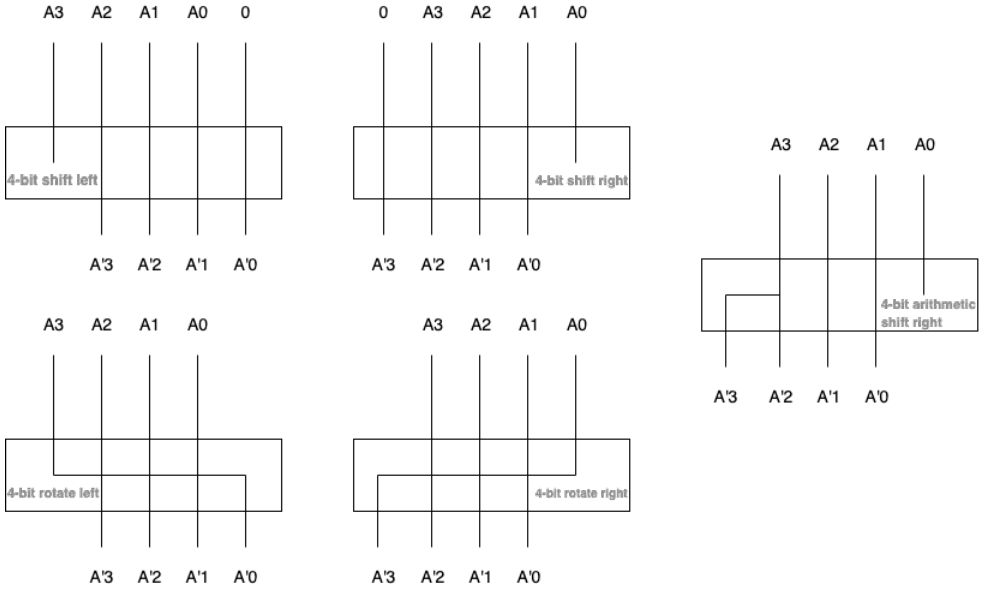
Operaciones aritméticas y lógicas - Componentes aritméticas

- ***Rotating components***: Componentes que desplazan los bits de un número a la derecha o a la izquierda, pero que rotan el bit "descartado" al bit más o menos significativo respectivamente.
  - rotate_left(1000b) = 0001b
  - rotate_left(0101b) = 1010b
  - rotate_right(0100b) = 0010b
  - rotate_right(0011b) = 1001b



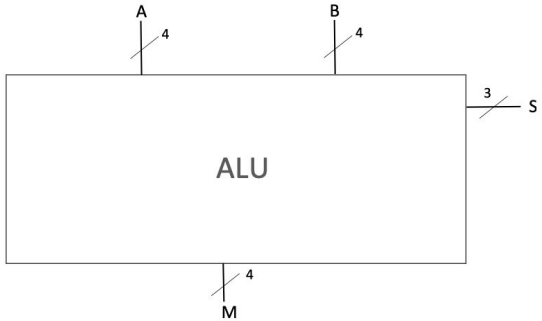
Operaciones aritméticas y lógicas - Componentes aritméticas

Diagramas de componentes de *shift* y *rotate*



Operaciones aritméticas y lógicas - Componentes aritméticas

- ***Arithmetic Logic Unit* (ALU):** Realiza las operaciones aritmético-lógicas con los componentes antes vistos y selecciona la operación mediante multiplexores (que reciben el resultado de todas las operaciones). Será la **unidad de ejecución del computador básico**.

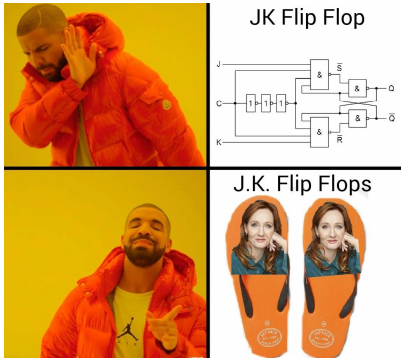


S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

* Solo realiza *shifts* lógicos, no aritméticos.

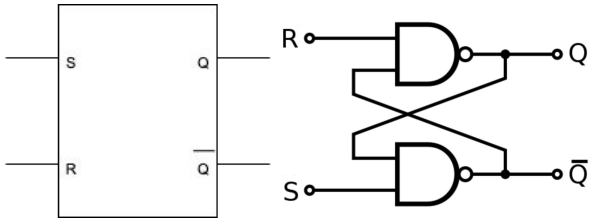
Almacenamiento de datos

- Contenido que encuentran en:
  - **Clase 3 - Almacenamiento de Datos**
  - **04 - Almacenamiento de Datos** (Apuntes)



Almacenamiento de datos - Latches

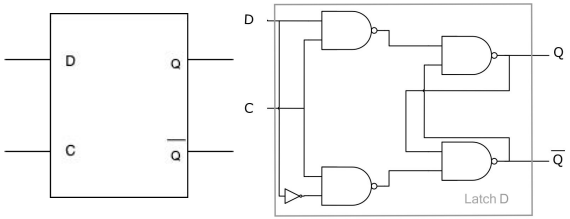
- **Latch RS:** Componente que puede almacenar un estado o cambiarlo mediante las señales *R*(*eset*) y *S*(*et*) a través de un **circuito secuencial**.



S	R	Q(t+1)
0	0	-
0	1	0
1	0	1
1	1	Q(t)

* Notar el uso de compuertas NAND.

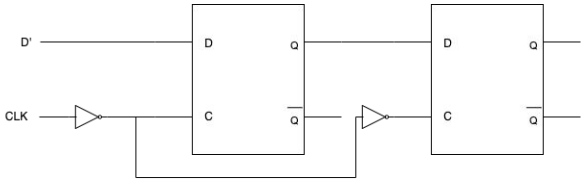
- **Latch D:** Componente que se construye sobre un Latch RS para permitir el almacenamiento de una señal *D* a partir de una señal de control *C*.

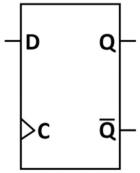


C	D	Q(t+1)
0	0	Q(t)
0	1	Q(t)
1	0	0
1	1	1

Almacenamiento de datos - Flip-flops

- **Flip-flop D:** Componente que permite guardar el estado anterior de una señal **en un instante dado**.

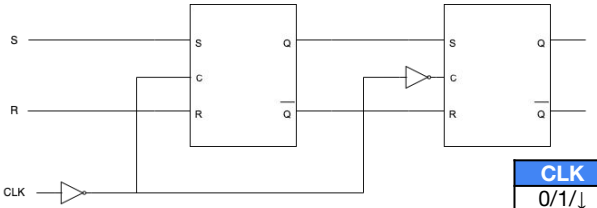


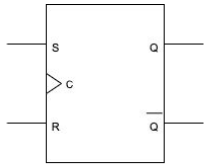


↑ = Flanco de subida (CLK de 0 a 1).
↓ = Flanco de bajada (CLK de 1 a 0).

CLK	*D*	*Q(t+1)*
0/1/↓	0/1	*Q(t)*
↑	0	0
↑	1	1

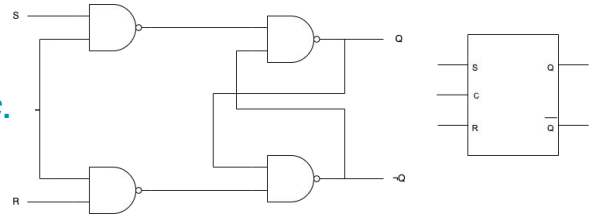
- **Flip-flop RS:** Flip-flop construido con latches RS que incluyen señal de control *C*.





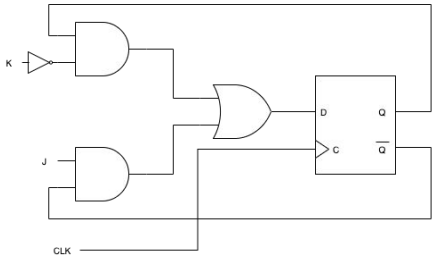
CLK	*S*	*R*	*Q(t+1)*
0/1/↓	0/1	0/1	*Q(t)*
↑	0	0	*Q(t)*
↑	0	1	0
↑	0	1	1
↑	1	1	-

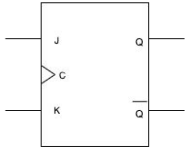
* Latch RS con señal de control *C*.



Almacenamiento de datos - Flip-flops

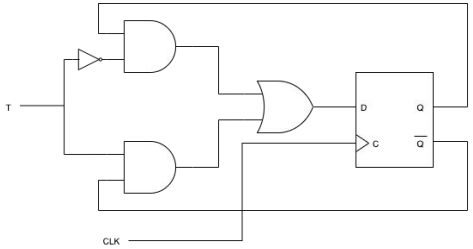
- **Flip-flop JK:** Similar al flip-flop RS, pero sin estado inválido.

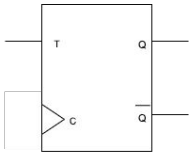




CLK	*J*	*K*	*Q(t+1)*
0/1/↓	0/1	0/1	*Q(t)*
↑	0	0	*Q(t)*
↑	0	1	0
↑	1	0	1
↑	1	1	¬*Q(t)*

- **Flip-flop T:** Flip-flop que a partir de una señal *T*(*oggle*) invierte el estado.

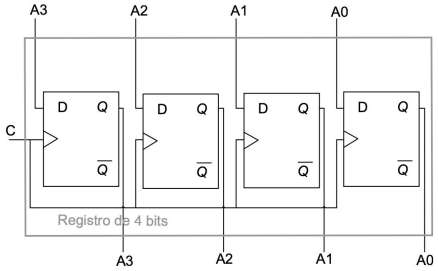


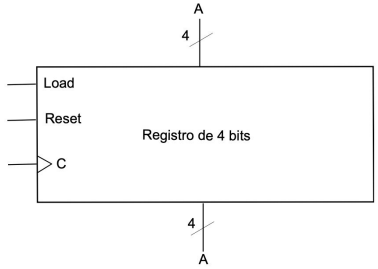


CLK	*T*	*Q(t+1)*
0/1/↓	0/1	*Q(t)*
↑	0	*Q(t)*
↑	1	¬*Q(t)*

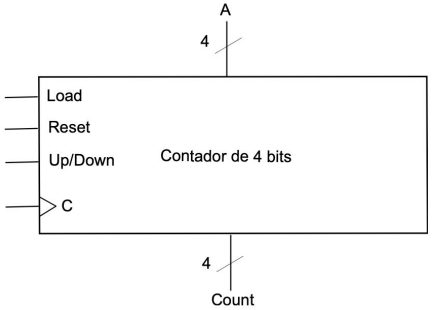
Almacenamiento de datos - Registros y contadores

- **Registro:** Conjunto de flip-flops que almacenan cada bit de un valor numérico. Posee señales *Reset* para dejar el valor en 0 y *Load* para cargar una señal de entrada.





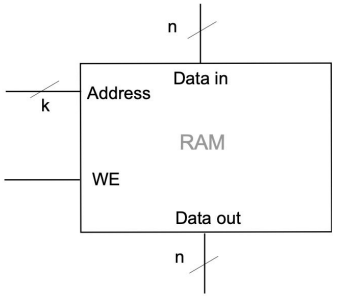
- **Contador:** Registro con señales de incremento/decremento para aumentar o disminuir en una unidad el valor almacenado.

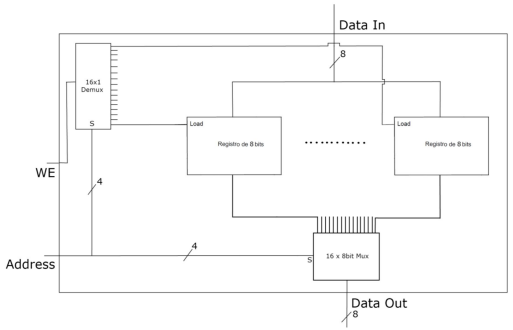


*** De ahora en adelante, todo componente que en su interior posea flip-flops debe estar conectado al *clock* del sistema para asegurar sincronización.**

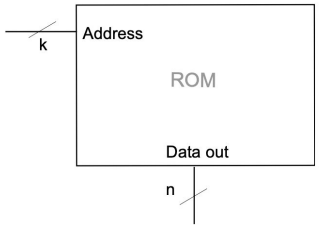
Almacenamiento de datos - Memorias

- **RAM:** Extensión de los registros para leer o escribir **palabras de memoria** (generalmente de 8 bits - 1 byte). Si se poseen *k* bits de direccionamiento, la memoria posee 2^k palabras de memorias (2^k bytes).





- **ROM:** Memoria de “solo lectura” (dependiendo del caso, pero pueden asumirla así).



*** El uso del Demux es fundamental para el funcionamiento de la memoria RAM y la propagación de la señal WE (*Write Enable*) solo al registro que corresponda.**

Almacenamiento de datos - Tipos de dato

- **Variables:** Valor que puede cambiar durante la ejecución de un programa y cuyo tipo de dato incide en la forma en que se almacena.

Tipo de dato	Codificación	Interpretación	#Bits de representación
char	base 2 sin signo	carácter o entero positivo	8
signed char	base 2 con signo en complemento de 2	entero positivo o negativo	8
short	base 2 con signo en complemento de 2	entero positivo o negativo	16
unsigned short	base 2 sin signo	entero positivo	16
int	base 2 con signo en complemento de 2	entero positivo o negativo	32
unsigned int	base 2 sin signo	entero positivo	32
long	base 2 con signo en complemento de 2	entero positivo o negativo	64
unsigned long	base 2 sin signo	entero positivo	64
long long	base 2 con signo en complemento de 2	entero positivo o negativo	128
unsigned long long	base 2 sin signo	entero positivo	128
float	punto flotante de precisión simple	Racionales y casos especiales	32
double	punto flotante de precisión doble	Racionales y casos especiales	64
long double	punto flotante de precisión cuádruple	Racionales y casos especiales	128

Almacenamiento de datos - *Endianness*

- ***Endianness*:** Orden en el que se almacenan la secuencia de un dato.
  - ***Big Endian*:** La palabra más significativa de la secuencia se almacena en la dirección **menor**.
  - ***Little Endian*:** La palabra menos significativa de la secuencia se almacena en la dirección **menor**.

*** Ejemplos de *endianness* para el almacenamiento del dato 0000111101010101, que se separa en los bytes 00001111 y 01010101**

Dirección de memoria (hexa)	Palabra almacenada (*big endian*)
0x00	00001111
0x01	01010101
0x02	-

Dirección de memoria (hexa)	Palabra almacenada (*little endian*)
0x00	01010101
0x01	00001111
0x02	-

Almacenamiento de datos - Arreglos y matrices

- **Arreglo:** Tipo de dato que además posee un **largo** y una **dirección de memoria de inicio**.

- **Matriz:** Arreglo de arreglos. Se puede almacenar según la convención de **filas** o **columnas**.

```
let variables: number[] = [1, 3, 5, 7];
```

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000011
0x02	00000101
0x03	00000111
0x04	-

```
let variablesMatrix: number[][] = [
  [1, 3, 5],
  [2, 4, 6]
];
```

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000011
0x02	00000101
0x03	00000010
0x04	00000100
0x05	00000110

Dirección de memoria (hexa)	Palabra almacenada
0x00	00000001
0x01	00000010
0x02	00000011
0x03	00000100
0x04	00000101
0x05	00000110

Direccionamiento en convención por filas: $dir(matriz[i,j]) = dir(\text{matriz}) + i * sizeof(matriz[i,j]) * \#columnas + j * sizeof(matriz[i,j])$

Direccionamiento en convención por columnas: $dir(matriz[i,j]) = dir(\text{matriz}) + j * sizeof(matriz[i,j]) * \#filas + i * sizeof(matriz[i,j])$

Computador básico

- Contenido que encuentran en:
  - **Clase 4 - Programabilidad**
  - **Clase 5 - Saltos y Subrutinas**
  - **Clase 7 - Arquitecturas de Computadores**
  - **05 - Programabilidad** (Apuntes)
  - **06 - Saltos y Subrutinas** (Apuntes)
  - **07 - Arquitecturas de Computadores** (Apuntes)

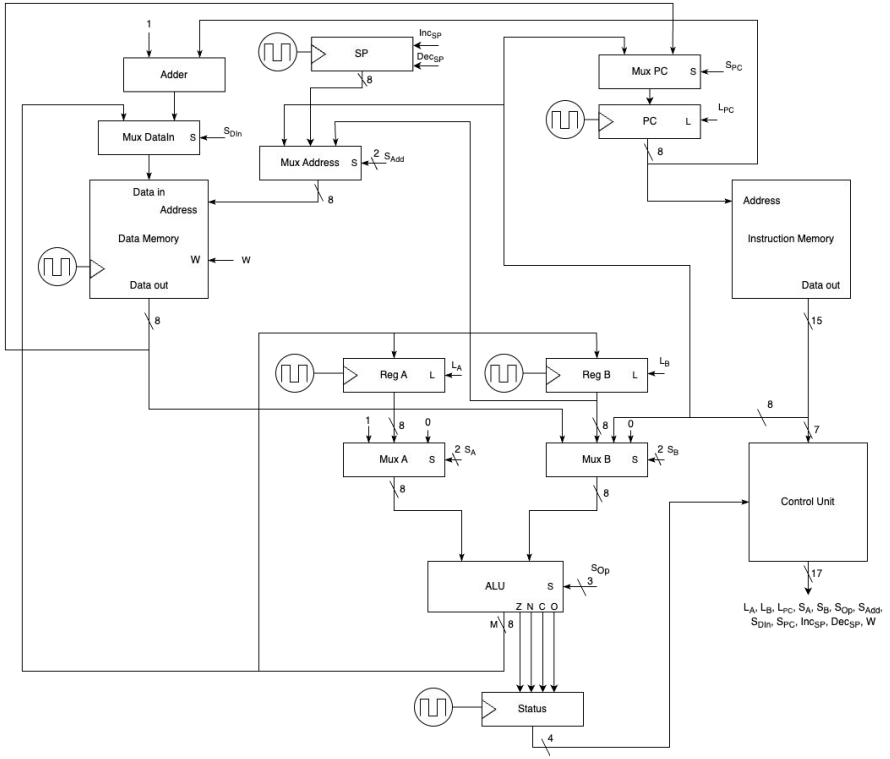




Computador básico - Microarquitectura

- **Diagrama del computador básico**

Diagrama completo. Iremos destacando las conexiones y componentes relevantes que habilitan cada funcionalidad.

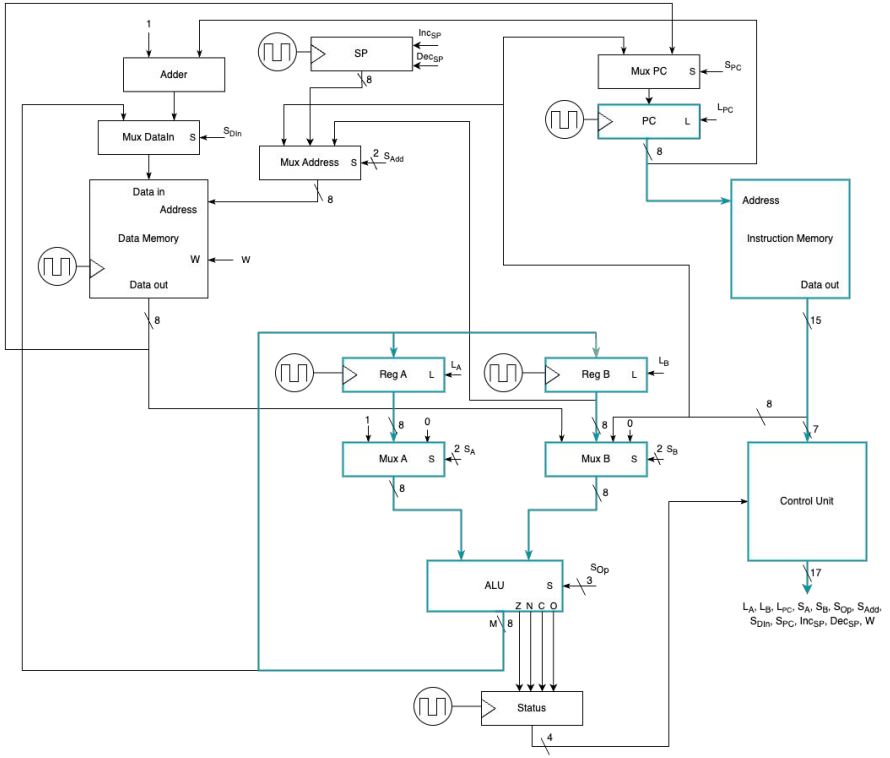


Computador básico - Microarquitectura

- Contador PC (*Program Counter*) permite ejecutar de forma secuencial un programa almacenado en la memoria ROM *Instruction Memory*. Por cada flanco de subida incrementa su valor y, así, cambia la instrucción en ejecución.

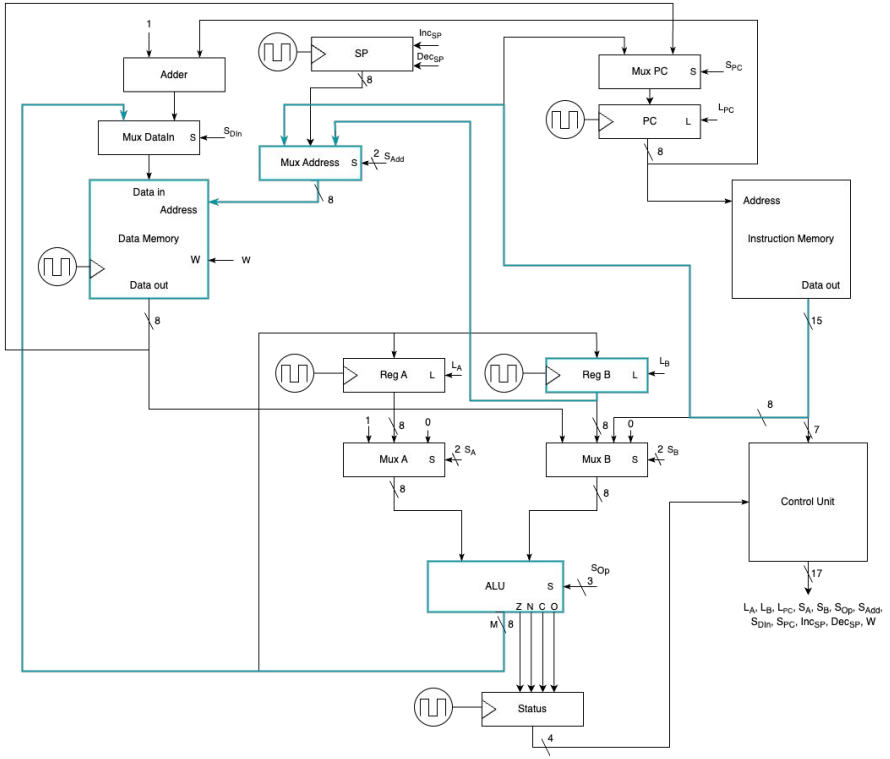
- *Control Unit* decodifica el *opcode* de la instrucción en señales de control que ejecutan la operación deseada.

- Registros *A* y *B* permiten realizar operaciones en la ALU y almacenar el resultado temporalmente. Muxes *A* y *B* permiten ampliar las operaciones a realizar.



Computador básico - Microarquitectura

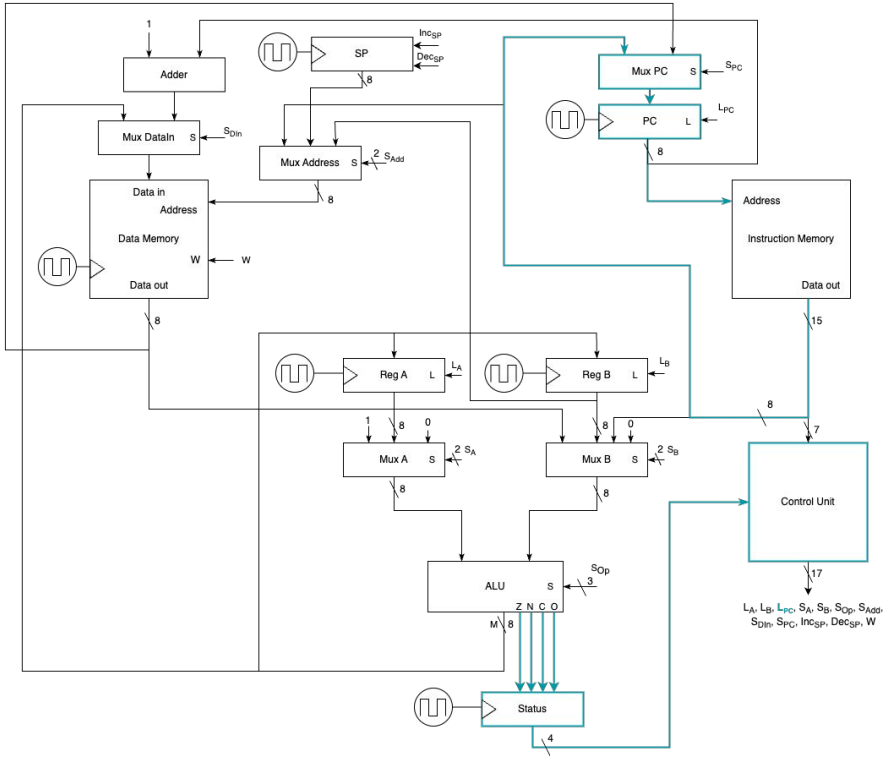
- Valor literal (numérico) asociado a la instrucción puede ser utilizado para realizar más operaciones a través del Mux *B*.
- Los resultados de las operaciones se pueden almacenar en la memoria RAM *Data Memory* (donde también se almacenan las variables del programa).
- La dirección donde se lee o escribe un dato puede ser dada por el literal mismo (direccionamiento directo) o por el valor del registro *B* (direccionamiento indirecto).



Computador básico - Microarquitectura

- La conexión entre el literal y el contador `PC` (a través de su Mux) permite la ejecución de **instrucciones de salto**, cargando en el contador la dirección de memoria de la instrucción deseada.

- Si se quiere realizar un salto **condicional**, se hace uso de las *flags* de estado que otorgan información de la instrucción anterior (`Z` = cero; `N` = negativo; `C` = *carry*; `O` = *overflow*). Si se ejecuta una instrucción de salto que cumple la condición, la *Control Unit* habilita la señal $\mathbf{L_{PC}}$.

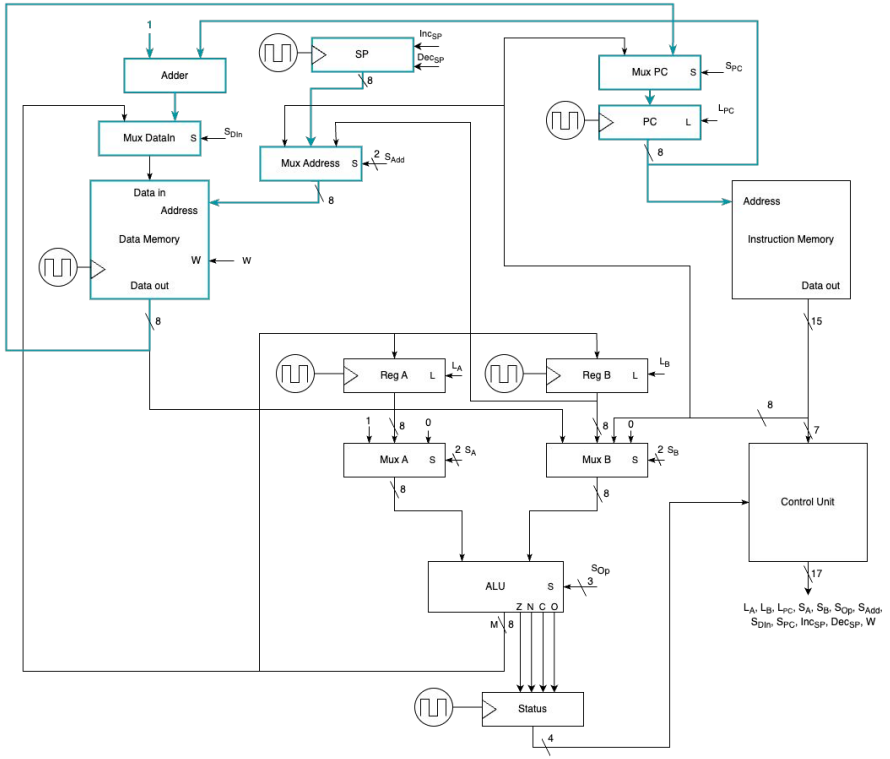


* Usamos la instrucción `CMP` antes del salto para tener certeza del estado esperado, pero en estricto rigor se pueden realizar saltos condicionales siempre.

Computador básico - Microarquitectura

- Contador SP (*Stack Pointer*) permite añadir **memoria de *stack*** (ingreso de elementos desde la última dirección de la memoria).

- A través de la memoria de *stack* podemos implementar **subrutinas**. La conexión del PC y la *Data Memory* permite almacenar la dirección de retorno (PC+1) en el *stack* y cargarla de vuelta a través de la conexión entre la salida de la *Data Memory* y PC a través de su Mux.



* Por construcción, SP siempre apunta una dirección sobre el tope. Por ende, para leer el valor del tope es necesario invertir una iteración en decrementarlo en una unidad.

Computador básico - ISA

- ***Instruction Set Architecture*:** Indica la **forma** en la que se escriben los programas de una arquitectura y las operaciones que soporta, indicando a su vez las señales requeridas para ello, su formato, etc.

Instrucción	Operandos	*Opcode*	Condición	L_{PC}	L_A	L_B	$S_A0,1$	$S_B0,1$	$S_{Op}0,1,2$	$S_{Add}0,1$	S_{DIn}	S_{PC}	W	Inc_{SP}	Dec_{SP}
MOV	A,B	0000000		0	1	0	ZERO	B	ADD	-	-	-	0	0	0
	B,A	0000001		0	0	1	A	ZERO	ADD	-	-	-	0	0	0
	A,Lit	0000010		0	1	0	ZERO	LIT	ADD	-	-	-	0	0	0
	B,Lit	0000011		0	0	1	ZERO	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0000100		0	1	0	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	B,(Dir)	0000101		0	0	1	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	(Dir),A	0000110		0	0	0	A	ZERO	ADD	LIT	ALU	-	1	0	0
	(Dir),B	0000111		0	0	0	ZERO	B	ADD	LIT	ALU	-	1	0	0
	A,(B)	0001000		0	1	0	ZERO	DOUT	ADD	B	-	-	0	0	0
	B,(B)	0001001		0	0	1	ZERO	DOUT	ADD	B	-	-	0	0	0
	(B),A	0001010		0	1	0	A	ZERO	ADD	B	ALU	-	1	0	0

Parte 1 de la ISA del computador básico

Computador básico - ISA

ADD	A,B	0001011	0	1	0	A	B	ADD	-	-	-	0	0	0
	B,A	0001100	0	0	1	A	B	ADD	-	-	-	0	0	0
	A,Lit	0001101	0	1	0	A	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0001110	0	1	0	A	DOUT	ADD	LIT	-	-	0	0	0
	A,(B)	0001111	0	1	0	A	DOUT	ADD	B	-	-	0	0	0
	(Dir)	0010000	0	0	0	A	B	ADD	LIT	ALU	-	1	0	0
SUB	A,B	0010001	0	1	0	A	B	SUB	-	-	-	0	0	0
	B,A	0010010	0	0	1	A	B	SUB	-	-	-	0	0	0
	A,Lit	0010011	0	1	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0010100	0	1	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0010101	0	1	0	A	DOUT	SUB	B	-	-	0	0	0
	(Dir)	0010110	0	0	0	A	B	SUB	LIT	ALU	-	1	0	0
AND	A,B	0010111	0	1	0	A	B	AND	-	-	-	0	0	0
	B,A	0011000	0	0	1	A	B	AND	-	-	-	0	0	0
	A,Lit	0011001	0	1	0	A	LIT	AND	-	-	-	0	0	0
	A,(Dir)	0011010	0	1	0	A	DOUT	AND	LIT	-	-	0	0	0
	A,(B)	0011011	0	1	0	A	DOUT	AND	B	-	-	0	0	0
	(Dir)	0011100	0	0	0	A	B	AND	LIT	ALU	-	1	0	0
OR	A,B	0011101	0	1	0	A	B	OR	-	-	-	0	0	0
	B,A	0011110	0	0	1	A	B	OR	-	-	-	0	0	0
	A,Lit	0011111	0	1	0	A	LIT	OR	-	-	-	0	0	0
	A,(Dir)	0100000	0	1	0	A	DOUT	OR	LIT	-	-	0	0	0
	A,(B)	0100001	0	1	0	A	DOUT	OR	B	-	-	0	0	0
	(Dir)	0100010	0	0	0	A	B	OR	LIT	ALU	-	1	0	0

Parte 2 de la ISA del computador básico

Computador básico - ISA

NOT	A,A	0100011	0	1	0	A	-	NOT	-	-	-	0	0	0
	B,A	0100100	0	0	1	A	-	NOT	-	-	-	0	0	0
	(Dir)	0100101	0	0	0	A	B	NOT	LIT	ALU	-	1	0	0
XOR	A,B	0100110	0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0100111	0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101000	0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101001	0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101010	0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101011	0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101100	0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101101	0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0101110	0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0101111	0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110000	0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0110001	0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0110010	0	0	1	ONE	B	ADD	-	-	-	0	0	0
	(B)	0110011	0	0	1	ONE	DOUT	ADD	B	ALU	-	1	0	0
	(Dir)	0110100	0	0	1	ONE	DOUT	ADD	LIT	ALU	-	1	0	0

Parte 3 de la ISA del computador básico

Computador básico - ISA

CMP	A,B	0110101		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0110110		0	0	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0110111		0	0	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0111000		0	0	0	A	DOUT	SUB	B	-	-	0	0	0
JMP	Dir	0111001		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111010	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111011	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	0111100	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	0111101	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	0111110	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	0111111	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000000	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000001	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000010		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1000100		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1000101		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
	B	1000110		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1000111		0	1	0	ZERO	DOUT	ADD	SP	-	-	0	0	0
POP	B	1000011		0	0	0	-	-	-	-	-	-	0	1	0
		1001000		0	0	1	ZERO	DOUT	ADD	SP	-	-	0	0	0
NOP		1001001		0	0	0	-	-	-	-	-	-	0	0	0

Parte 4 de la ISA del computador básico

Computador básico - ISA

- Ejemplo de código con la ISA del computador básico. Cabe destacar que se definen dos segmentos: `DATA` para las variables (incluyendo arreglos) y `CODE` para las instrucciones.

- El ***assembler*** es el programa encargado de transformar este código en lenguaje de máquina (binario) y de asegurar que existan instrucciones que almacenen las variables en memoria, así como también *mapear* correctamente las direcciones de los *labels* a los literales correctos.

```
DATA:
  var 1
  arr 1
      3
      5
CODE:
  MOV B,arr
  MOV A,(B)
  SHL A,A
  MOV (B),A
  INC B
  MOV A,(B)
  SHL A,A
  MOV (B),A
  INC B
  MOV A,(B)
  SHL A,A
  MOV (B),A
```

Computador básico - Clasificaciones

- **Paradigmas según ISA**
  - **RISC:** ***R****educed* ***I****nstruction* ***S****et* ***C****omputer*. Instrucciones pequeñas y simples. Su diseño permite simplificar el *hardware*, poniendo énfasis en el *software*.
  - **CISC:** ***C****omplex* ***I****nstruction* ***S****et* ***C****omputer*. Muchas instrucciones y con complejidad alta. Énfasis en un *hardware* más complejo para poder ejecutarlas.

- **Paradigmas según memoria**
  - **Harvard:** Memoria de datos separada de la memoria de instrucciones.
  - **Von Neumann:** Datos e instrucciones en una única unidad de memoria.

*** Bajo estos paradigmas, el computador básico presenta una microarquitectura Harvard y una ISA RISC.**

DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 14 - Resumen del Curso

Profesor: Germán Leandro Contreras Sagredo