# Integrantes

Pablo José Cárdenas Meneses - 2170080

Juan Pablo Beleño Mesa - 2204656

Diego Alejandro Arévalo Quintero - 2220066

Ana Valeria Barreto Tellez - 2215630

Brayan Julián Barrera Hernández - 2220097

# Grupo BytesBuilders

1. ¿Qué consideraciones importantes debe tener en cuenta para trabajar con Nand2Tetris?

   Es útil tener una comprensión básica de lógica digital, álgebra booleana y sistemas binarios antes de comenzar además leer y entender la documentación de cada capítulo ya que la teoría que se proporciona es clave para comprender el propósito de cada componente y finalmente familiarizarse con él, diseñando y probando..

2. ¿Qué otras herramientas similares a Nand2Tetris existen?

   **Logisim**: Es un simulador de circuitos digitales que permite diseñar y simular circuitos lógicos con puertas, multiplexores, flip-flops y otros componentes básicos. Es ideal para aprender y experimentar con conceptos de diseño de hardware.

   **MIT's 6.004 Computation Structures**: Este es un curso del MIT que cubre temas similares a los de Nand2Tetris, incluyendo diseño de hardware y arquitectura de computadoras. Aunque no es tan accesible como Nand2Tetris, es una excelente fuente para aprender más.

   **Verilog/VHDL Tutorials**: Aprender lenguajes de descripción de hardware como Verilog o VHDL te permitirá diseñar circuitos digitales y sistemas, proporcionando una perspectiva diferente pero relacionada con el diseño de computadoras.

# NOT

```
CHIP Not {
    IN in;
    OUT out;
    PARTS:

    Nand(a= in, b= in, out= out);
}
```

**Not:**

| in | out |
|---|---|
| 0 | 1 |
| 1 | 0 |

Para el **Not** simplemente se utiliza un **Nand** con **in** en ambas entradas ya que en la tabla de verdad de **Nand** en este caso es equivalente al **Not.**

**Nand:**

| a | b | out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

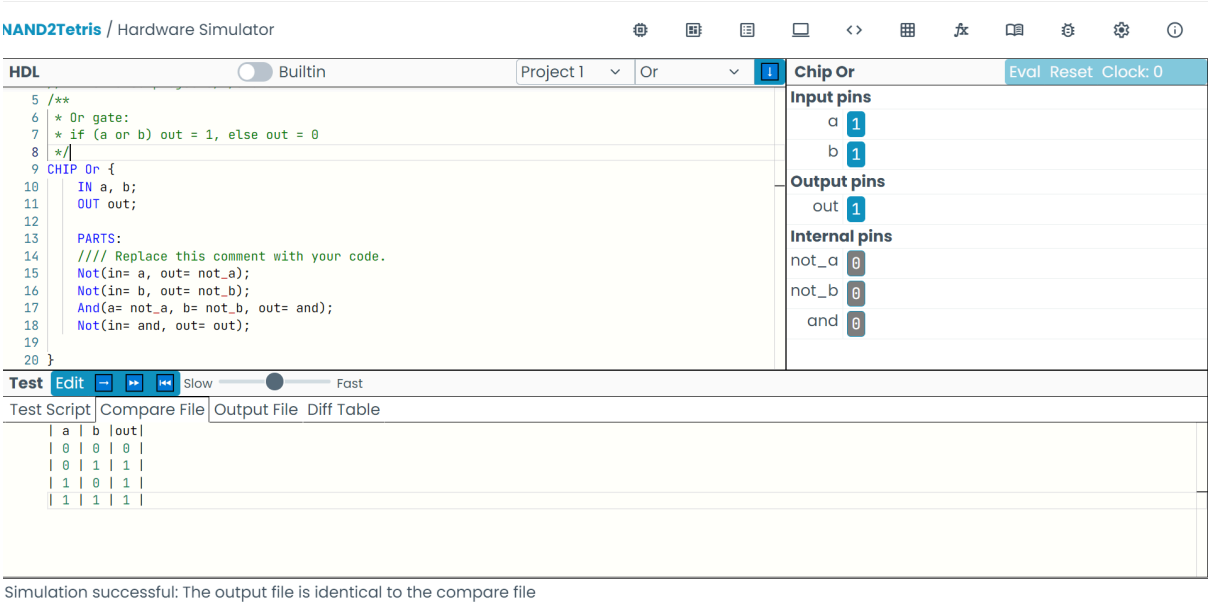
# AND

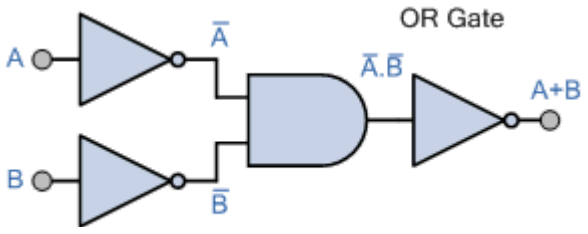
```
9  CHIP And{
10     IN a, b;
11     OUT out;
12
13     PARTS:
14     Nand(a= a, b=b , out= x);
15     Not(in=x , out=out );
16 }
```

| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Combinamos dos compuerta la NAND y la NOT las cuales nos ayudarán a generar la AND, le damos las entradas a y b a la compuerta NAND que nos genera una salida x esta salida la invertimos con la compuerta NOT y nos genera la AND.

# OR



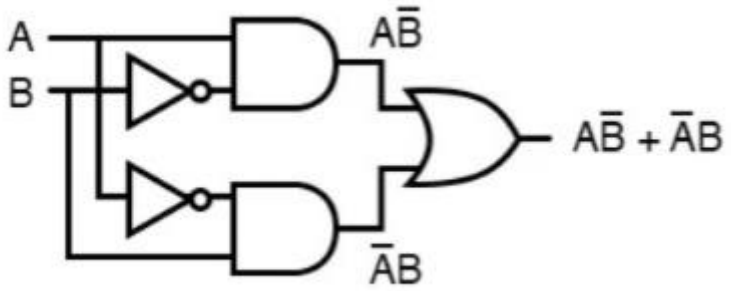

Para él OR se tienen dos opciones, o bien se puede construir usando tres compuertas NAND o en está implementación, usando tres NOT y una NAND. En el caso de esta implementación, la negación de ambas entradas se introduce al AND y la salida del NAND se niega, y por álgebra de Boole el resultado es equivalente.

# Xor

```
CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not(in=a, out=Not1);
    Not(in=b, out=Not2);
    And(a=Not1, b=b, out=And1);
    And(a=Not2, b=a, out=And2);
    Or(a=And1, b=And2, out= out);
}
```

| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




Viendo la tabla de verdad de la Xor nos damos cuenta que al haber dos valores iguales de entrada da como resultado 0, de esto podemos concluir la función que representa esta compuerta como **a¬b + ¬ab** donde "¬" representa la negación de la entrada.

# Mux

```
 9 CHIP Mux {
10     IN a, b, sel;
11     OUT out;
12
13     PARTS:
14     Not(in=sel, out=notSel);
15     And(a=a, b=notSel, out=outA);
16     And(a=b, b=sel, out=outB);
17     Or(a=outA, b=outB, out=out);
18 }
```

Sabiendo que la compuerta Mux, teniendo 3 entradas de 1 bit, dos convencionales (a y b) y una de selección (sel), cuando sel tenga el valor 0, el valor de la salida será el mismo que de la entrada a, y en caso contrario el valor de la salida será el mismo de la entrada b. Primero, en la línea 14 negamos el selector para poder obtener en la línea siguiente los casos en que la entrada a es 1 y el sel es 0, permitiendo asignar dicho valor a una salida outA. Se obtienen los casos en que la entrada b y el sel tienen el valor de 1 para asignarlos a una salida outB. Finalmente se unen las salidas outA y outB para obtener nuestra salida de la compuerta Mux.

| a | b |sel|out|
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

# DMUX

```
CHIP DMux {
    IN in, sel;
    OUT a, b;

    PARTS:
    Not(in= sel, out= notsel);
    And(a= in, b= notsel, out= a);
    And(a= in, b= sel, out= b);
}
```

**Dmux:**

| in | sel | a | b |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Para el **Demultiplexor** simplemente se toman los casos donde hay 1 o positivo, y se genera el caso devolviendo el correspondiente, esto se hace dos veces para **a** y para **b** entonces para a se toma el caso donde **in** es positivo y **sel** es negativo, y para **b** cuando tanto **in** como **sel** son positivos, se necesita que ocurran los dos casos al mismo tiempo, por eso se usa **And**

# NOT16

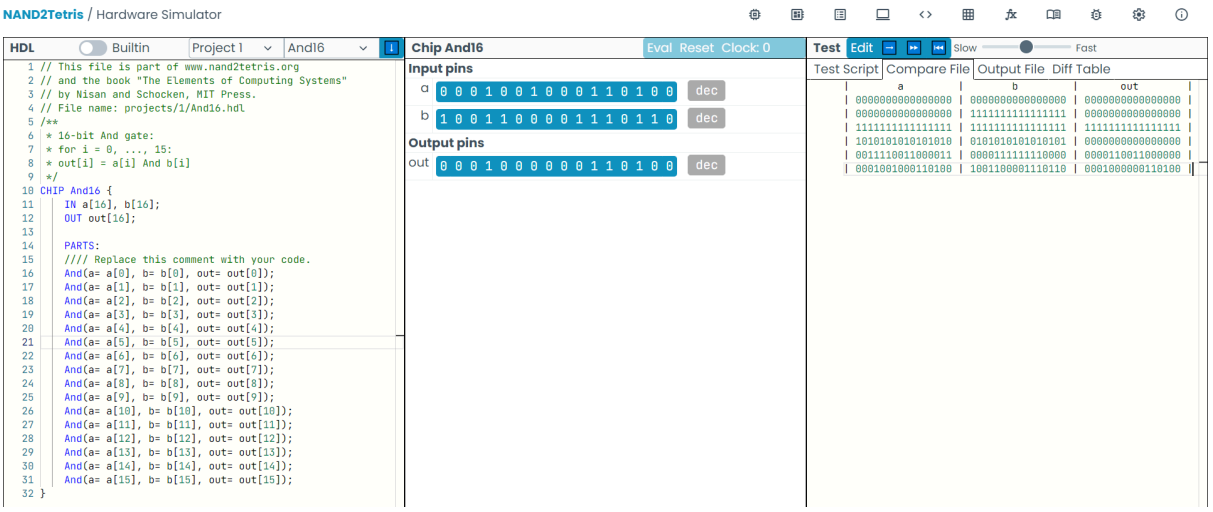
```
CHIP Not16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Not(in=in[0],
        out=out[0]);
    Not(in=in[1],
        out=out[1]);
    Not(in=in[2],
        out=out[2]);
    Not(in=in[3],
        out=out[3]);
    Not(in=in[4],
        out=out[4]);
    Not(in=in[5],
        out=out[5]);
    Not(in=in[6],
        out=out[6]);
    Not(in=in[7],
        out=out[7]);
    Not(in=in[8],
        out=out[8]);
    Not(in=in[9],
        out=out[9]);
    Not(in=in[10],
        out=out[10]);
    Not(in=in[11],
        out=out[11]);
    Not(in=in[12],
        out=out[12]);
    Not(in=in[13],
        out=out[13]);
    Not(in=in[14],
        out=out[14]);
    Not(in=in[15],
        out=out[15]);
```

| in | out |
|---|---|
| 0000000000000000 | 1111111111111111 |
| 1111111111111111 | 0000000000000000 |
| 1010101010101010 | 0101010101010101 |
| 0011110011000011 | 1100001100111100 |
| 0001001000110100 | 1110110111001011 |

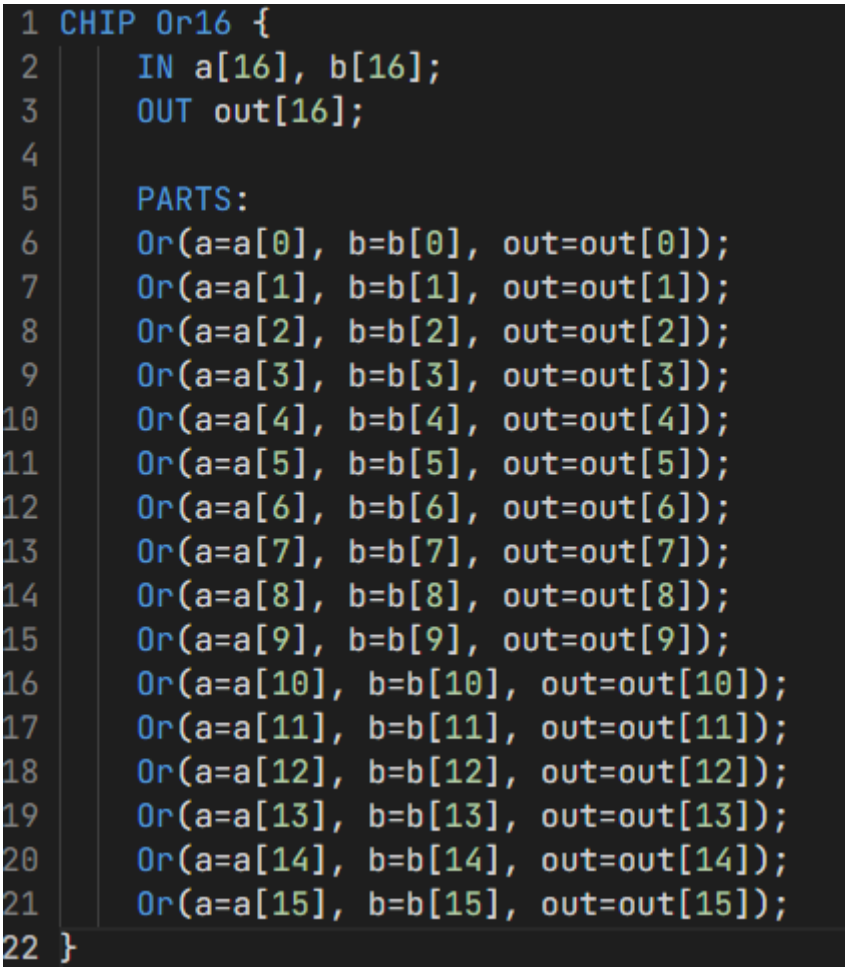
La compuerta NOT toma el primer bit de la entrada (in[0]) y lo invierte. El resultado se asigna al primer bit de la salida (out[0]) esto se repite para 16 bits desde 0 hasta 15.

# AND16

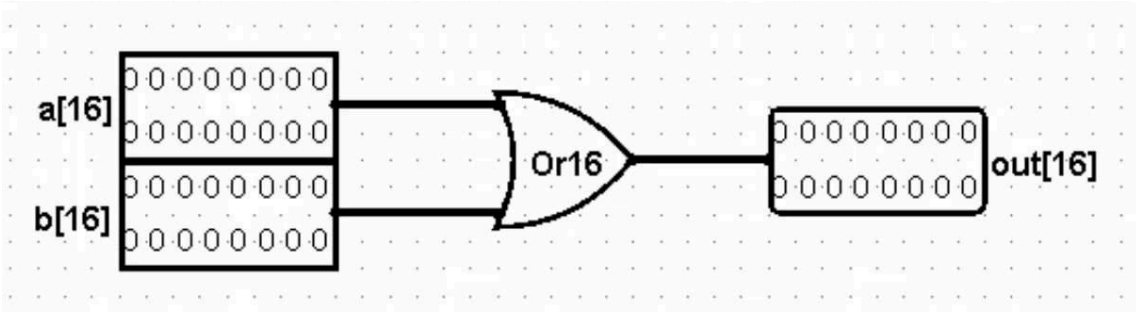



Para el AND16 se compara ambos array, y se compara ambas entradas, a y b para que tengan el mismo elemento en el espacio correspondiente, en el caso de que sea el mismo, se agrega dicho valor en el array de salida.

# Or16



| a | b | out |
|---|---|---|
| 0000000000000000 | 0000000000000000 | 0000000000000000 |
| 0000000000000000 | 1111111111111111 | 1111111111111111 |
| 1111111111111111 | 1111111111111111 | 1111111111111111 |
| 1010101010101010 | 0101010101010101 | 1111111111111111 |
| 0011110011000011 | 0000111111110000 | 0011111111110011 |
| 0001001000110100 | 1001100001110110 | 1001101001110110 |




Simplemente está formada por Or de un solo bit y se realiza la operación bit a bit correspondiente de las entradas con las salidas.

# Mux16

```
10 CHIP Mux16 {
11     IN a[16], b[16], sel;
12     OUT out[16];
13
14     PARTS:
15     Mux(a=a[0], b=b[0], sel=sel, out=out[0]);
16     Mux(a=a[1], b=b[1], sel=sel, out=out[1]);
17     Mux(a=a[2], b=b[2], sel=sel, out=out[2]);
18     Mux(a=a[3], b=b[3], sel=sel, out=out[3]);
19     Mux(a=a[4], b=b[4], sel=sel, out=out[4]);
20     Mux(a=a[5], b=b[5], sel=sel, out=out[5]);
21     Mux(a=a[6], b=b[6], sel=sel, out=out[6]);
22     Mux(a=a[7], b=b[7], sel=sel, out=out[7]);
23     Mux(a=a[8], b=b[8], sel=sel, out=out[8]);
24     Mux(a=a[9], b=b[9], sel=sel, out=out[9]);
25     Mux(a=a[10], b=b[10], sel=sel, out=out[10]);
26     Mux(a=a[11], b=b[11], sel=sel, out=out[11]);
27     Mux(a=a[12], b=b[12], sel=sel, out=out[12]);
28     Mux(a=a[13], b=b[13], sel=sel, out=out[13]);
29     Mux(a=a[14], b=b[14], sel=sel, out=out[14]);
30     Mux(a=a[15], b=b[15], sel=sel, out=out[15]);
31 }
```

Siguiendo la lógica de la compuerta Mux, que se ha explicado con anterioridad, se procede a aplicarse a cada pareja de entradas (a y b) a lo largo del bus de 16 bits, manteniendo la entrada de selección de un solo bit ya que seguimos manteniendo dos entradas a y b.

| a | b | sel | out |
|---|---|---|---|
| 0000000000000000 | 0000000000000000 | 0 | 0000000000000000 |
| 0000000000000000 | 0000000000000000 | 1 | 0000000000000000 |
| 0000000000000000 | 0001001000110100 | 0 | 0000000000000000 |
| 0000000000000000 | 0001001000110100 | 1 | 0001001000110100 |
| 1001100001110110 | 0000000000000000 | 0 | 1001100001110110 |
| 1001100001110110 | 0000000000000000 | 1 | 0000000000000000 |
| 1010101010101010 | 0101010101010101 | 0 | 1010101010101010 |
| 1010101010101010 | 0101010101010101 | 1 | 0101010101010101 |

# Or8Way

```
CHIP Or8Way {
    IN in[8];
    OUT out;

    PARTS:
    Or(a= in[0], b= in[1], out= o1);
    Or(a= o1, b= in[2], out= o2);
    Or(a= o2, b= in[3], out= o3);
    Or(a= o3, b= in[4], out= o4);
    Or(a= o4, b= in[5], out= o5);
    Or(a= o5, b= in[6], out= o6);
    Or(a= o6, b= in[7], out= out);
}
```

**Or8Way:**

| in | out |
|---|---|
| 00000000 | 0 |
| 11111111 | 1 |
| 00010000 | 1 |
| 00000001 | 1 |
| 00100110 | 1 |

Para **Or8Way** siendo este 8 entradas anidadas, donde si alguna de esas entradas es positiva, **out** es positiva, se utilizan 7 **Or's** en serie, donde en el caso en el que alguno de ellos de como salida positiva, ya todos los siguientes **Or's** serán positivos, consigo el último out.

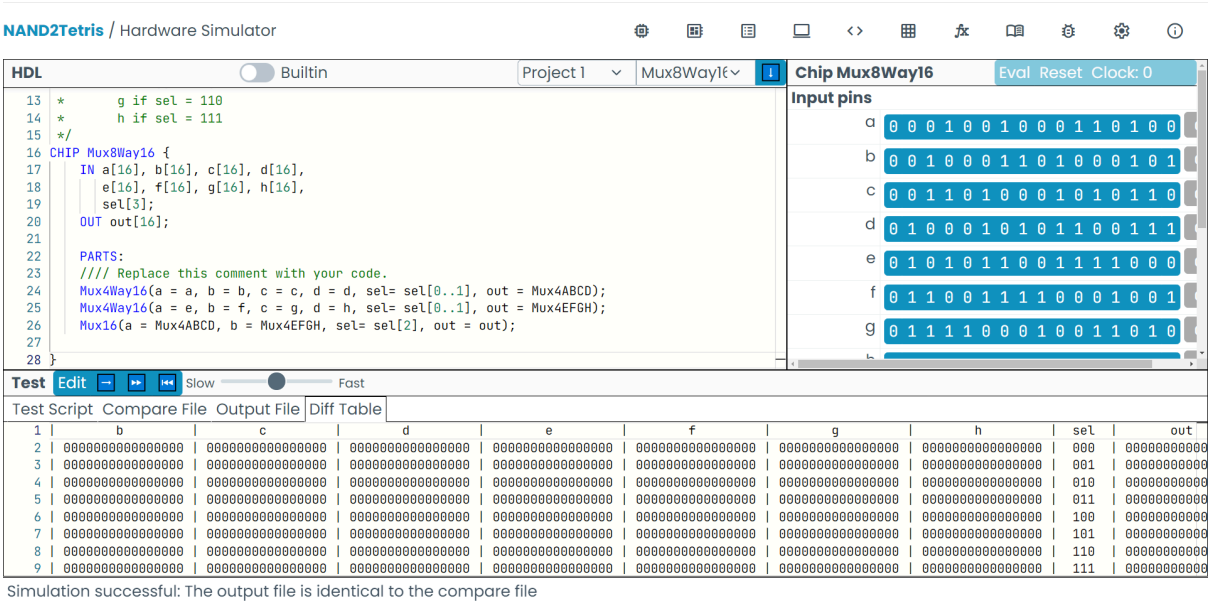
# MUX4WAY16

```
12 CHIP Mux4Way16 {
13     IN a[16], b[16], c[16], d[16], sel[2];
14     OUT out[16];
15
16     PARTS:
17      Mux16(a[0..15]=a[0..15], b[0..15]=b[0..15], sel=sel[0], out[0..15]=first2);
18     Mux16(a[0..15]=c[0..15], b[0..15]=d[0..15], sel=sel[0], out[0..15]=last2);
19     Mux16(a[0..15]=first2, b[0..15]=last2, sel=sel[1], out[0..15]=out[0..15]);
20 }
```

| a | b | c | d | sel | out |
|---|---|---|---|---|---|
| 0000000000000000 | 0000000000000000 | 0000000000000000 | 0000000000000000 | 00 | 0000000000000000 |
| 0000000000000000 | 0000000000000000 | 0000000000000000 | 0000000000000000 | 01 | 0000000000000000 |
| 0000000000000000 | 0000000000000000 | 0000000000000000 | 0000000000000000 | 10 | 0000000000000000 |
| 0000000000000000 | 0000000000000000 | 0000000000000000 | 0000000000000000 | 11 | 0000000000000000 |
| 0001001000110100 | 1001100001110110 | 1010101010101010 | 0101010101010101 | 00 | 0001001000110100 |
| 0001001000110100 | 1001100001110110 | 1010101010101010 | 0101010101010101 | 01 | 1001100001110110 |
| 0001001000110100 | 1001100001110110 | 1010101010101010 | 0101010101010101 | 10 | 1010101010101010 |
| 0001001000110100 | 1001100001110110 | 1010101010101010 | 0101010101010101 | 11 | 0101010101010101 |

Permite seleccionar una de las cuatro entradas de 16 bits (a, b, c, d) y dirigirla a la salida out, utilizando una señal de selección de 2 bits (sel). La selección se realiza en dos niveles: primero se seleccionan pares de entradas y luego se elige entre los dos resultados intermedios.

# MUX8WAY16



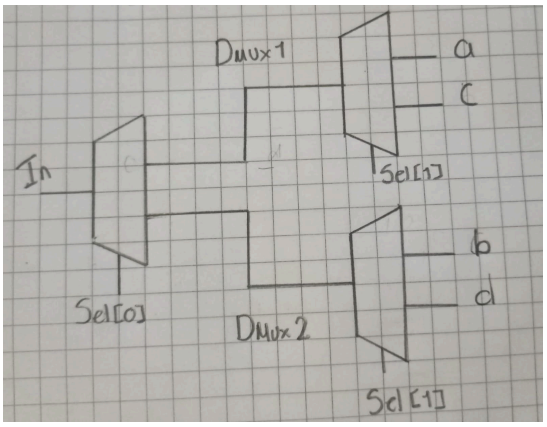
Para la construcción de esta compuerta, nos apoyamos en la Mux4Way16, ya que tienen el mismo funcionamiento, únicamente cambia la cantidad de entradas y por ende la cantidad de entradas del ‘sel’, que en este caso es de tres bits. Se usan dos Mux4Way16 para operar cada uno de a cuatro entradas, y la salida de estas se opera en un Mux16, ya que solamente tenemos estas nuevas dos entradas de 16 bits.

# DMux4Way

```
CHIP DMux4Way {
    IN in, sel[2];
    OUT a, b, c, d;

    PARTS:
    DMux(in=in, sel=sel[0], a=Dmux1, b=Dmux2);
    DMux(in=Dmux1, sel=sel[1], a=a, b=c);
    DMux(in=Dmux2, sel=sel[1], a=b, b=d);
}
```

| in | sel | a | b | c | d |
|---|---|---|---|---|---|
| 0 | 00 | 0 | 0 | 0 | 0 |
| 0 | 01 | 0 | 0 | 0 | 0 |
| 0 | 10 | 0 | 0 | 0 | 0 |
| 0 | 11 | 0 | 0 | 0 | 0 |
| 1 | 00 | 1 | 0 | 0 | 0 |
| 1 | 01 | 0 | 1 | 0 | 0 |
| 1 | 10 | 0 | 0 | 1 | 0 |
| 1 | 11 | 0 | 0 | 0 | 1 |



Se utilizan **3 Dmux**, el primero recibe la entrada y usa el primer bit del **sel** como el selector, al indicar **0** como primer bit del sel pasarán **a y c**, en caso de ser **1**, **b y d** respectivamente como posibles opciones para los siguiente 2 **Dmux** que dependiendo del último dígito del **sel** decidirá la salida final.

# DMux8Way

```
16 CHIP DMux8Way {
17     IN in, sel[3];
18     OUT a, b, c, d, e, f, g, h;
19
20     PARTS:
21     DMux(in=in, sel=sel[0], a=outA, b=outB);
22     DMux(in=outA, sel=sel[1], a=outA2, b=outC);
23     DMux(in=outA2, sel=sel[2], a=a, b=e);
24     DMux(in=outC, sel=sel[2], a=c, b=g);
25     DMux(in=outB, sel=sel[1], a=outB2, b=outD);
26     DMux(in=outB2, sel=sel[2], a=b, b=f);
27     DMux(in=outD, sel=sel[2], a=d, b=h);
28
29 }
```

Tenemos una compuerta DMux de 8 salidas de 1 bit, por lo tanto, debemos utilizar una entrada de 3 bits que nos permita asignar un valor a cada salida. Dado que la compuerta trabaja con pares, debemos realizar varias bifurcaciones, inicialmente para el valor de sel --0/--1, asignando un valor a salidas temporales para poder pasar al siguiente bit, donde tomamos uno de estos valores como entrada para analizar los casos, ya sea para el valor de sel -00/-10 o -01/-11, y realizamos el mismo proceso, de manera que podemos obtener cada salida que corresponda con el valor de sel.

| in | sel | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 010 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 011 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 101 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 110 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 000 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 001 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 010 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 011 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 100 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 101 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 110 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

# Referencias


1. The Elements of Computing Systems (2nd ed.), Nisan and Schocken, MIT Press.

2. Electronics Tutorials: Universal Logic Gates. https://www.electronics-tutorials.ws/logic/universal-gates.html

3. Build a Modern Computer from First Principles: From Nand to Tetris (Project-Centered Course). Hebrew University of Jerusalem. https://www.coursera.org/learn/build-a-computer