Universidad Industrial de Santander

Arquitectura de computadores 2024-2

Práctica 2: Lógico aritmética

Nucleo Digital

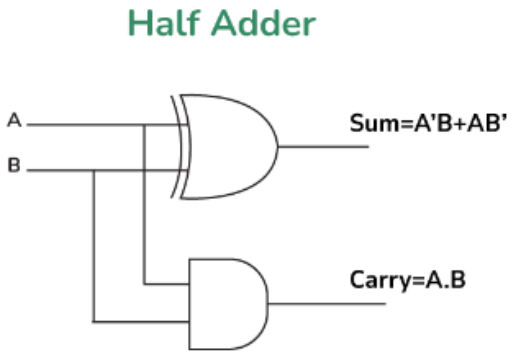
El proyecto 2 de Nand2Tetris, llamado aritmética booleana, consiste en la creación gradual de chips que permitirán la construcción de una ALU (Unidad Lógica-Aritmética), la cual es la pieza central de la CPU (Unidad Central de Procesamiento). Es posible utilizar los chips creados para el proyecto 1.

HalfAdder:
El chip está compuesto de dos entradas a y b se debe simular el proceso de suma de los dos bits de la siguiente manera.

Truth Table

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

Para conseguir este comportamiento usaremos dos compuertas una Xor y una And, en la primera su salida va a corresponder a la suma y la segunda es usada para el valor del carry al ser 1 y 1 se da como salida 1 es el único caso que necesitamos en el caso del carry.



Finalmente implementada en Nand2tetris queda de está manera.

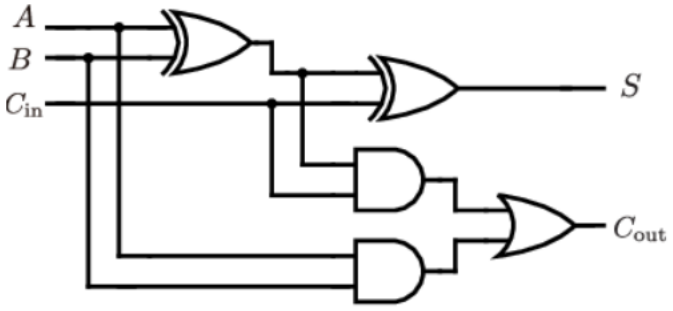
```
CHIP HalfAdder {
    IN a, b;    // 1-bit inputs
    OUT sum,    // Right bit of a + b
        carry;  // Left bit of a + b

    PARTS:
    Xor(a =a , b =b , out =sum );
    And(a=a , b=b, out=carry);
```

FullAdder:
El chip está compuesto por cinco compuertas dos Xor, dos And y una Or. Buscamos que se comporte de la siguiente manera

Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Para conseguirlo pasamos como en el HalfAdder las entradas por la compuerta Xor y su salida va a ir a otra Xor junto con la entrada de c que va a equivaler a la salida de la variable Sum. Por otro lado la entrada a y b tal cual como en el HalfAdder pasa por una compuerta And para el carry pero luego se debe usar una compuerta And para lo que podríamos ver como nuestro segundo carry y esta salida la llevamos a una compuerta tipo Or junto con la salida de la And anteriormente descrita y este resultado será la salida llamada Carry.



Finalmente implementada en Nand2tetris queda de está manera.

```
CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c

    PARTS:
    Xor(a =a , b = b, out =xorsalab );
    Xor(a =xorsalab , b = c, out =sum );
    And(a= a, b=b , out=andsalab );
    And(a= xorsalab, b= c, out=andcxor );
    Or(a=andcxor , b=andsalab , out=carry );
}
```

Add16:

El Add16 es un sumador de 16 bits que toma dos números binarios de 16 bits como entrada y los suma bit a bit. Utiliza una serie de sumadores completos (full adders) conectados en cascada, donde cada sumador suma los bits correspondientes de los dos números y lleva (carry) de la posición anterior. El resultado es la suma de los dos números, con una salida de 16 bits y un bit de acarreo final si es necesario.

```
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    FullAdder(a=a[0] , b=b[0] , c=false , sum=out[0] , carry=c1 );
    FullAdder(a=a[1] , b=b[1] , c=c1 , sum=out[1] , carry=c2 );
    FullAdder(a=a[2] , b=b[2] , c=c2 , sum=out[2] , carry=c3 );
    FullAdder(a=a[3] , b=b[3] , c=c3 , sum=out[3] , carry=c4 );
    FullAdder(a=a[4] , b=b[4] , c=c4 , sum=out[4] , carry=c5 );
    FullAdder(a=a[5] , b=b[5] , c=c5 , sum=out[5] , carry=c6 );
    FullAdder(a=a[6] , b=b[6] , c=c6 , sum=out[6] , carry=c7 );
    FullAdder(a=a[7] , b=b[7] , c=c7 , sum=out[7] , carry=c8 );
    FullAdder(a=a[8] , b=b[8] , c=c8 , sum=out[8] , carry=c9 );
    FullAdder(a=a[9] , b=b[9] , c=c9 , sum=out[9] , carry=c10 );
```

```
    FullAdder(a=a[10] , b=b[10] , c=c10 , sum=out[10] , carry=c11 );
    FullAdder(a=a[11] , b=b[11] , c=c11 , sum=out[11] , carry=c12 );
    FullAdder(a=a[12] , b=b[12] , c=c12 , sum=out[12] , carry=c13 );
    FullAdder(a=a[13] , b=b[13] , c=c13 , sum=out[13] , carry=c14 );
    FullAdder(a=a[14] , b=b[14] , c=c14 , sum=out[14] , carry=c15 );
    FullAdder(a=a[15] , b=b[15] , c=c15 , sum=out[15] , carry=c16 );
```

Inc16:

El chip Inc16 consiste en que a una entrada de 16 bits se le suma 1. Por lo tanto, el procedimiento para generarlo fue a la entrada sumarle 0000000000000001 (15 ceros y 1 uno) con el chip Add16 generado en este mismo proyecto. Para poder ingresar este número se tiene en cuenta que el bit 0 (menos significativo) es 1 (true), y los demás bits (del 1 al 15) son 0 (false).

```
CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Add16(a = in, b[0]= true, b[1..15]= false, out = out);
}
```

ALU:

Para realizar la ALU, en primer lugar se tuvo que hacer las modificaciones correspondientes para X y Y dependiendo de los valores de ZX, NX, ZY y NY.

Para convertir X en ceros, en caso de que ZX sea 1, se utilizó un multiplexor de 16 bits (Mux16), donde la primera entrada es X, la segunda entrada serán 16 ceros (false) y el selector será ZX. De esta forma si ZX es 1, X será reemplazada por ceros, y sino, se mantendrá intacta.

Para poder negar X según NX sea 1, primero se negó la salida del primer multiplexor, que es X luego de su primera modificación, con Not16. Posteriormente, se pasa este X y su versión negada como las entradas 1 y 2 respectivamente, de un nuevo multiplexor, donde el selector será NX, de esta forma X se negará sólo si NX es 1.

Estos procedimientos se repetirán con Y, ZY y NY.

Para seleccionar la operación a realizar (AND o suma) se realizan ambas con los chips And16 y Add16. Posteriormente se genera un multiplexor que tenga como entradas dichas operaciones y el selector será F. De forma que si F es 0, se realizará la operación AND, y si es 1, se sumarán.

Para negar la salida en caso de que NO sea 1, se negó la salida de la operación con Not16. Luego, se generó un multiplexor que tendrá como entradas la salida de la operación y su versión negada, y el selector será NO. Por lo que, si NO es 0, la salida se mantendrá intacta. Y si es 1, se negará la salida. Cabe aclarar que este multiplexor se realizó dos veces para generar dos salidas: OUT que es la salida directa del ALU, y SALIDA que se generó para poder ser utilizada para las operaciones posteriores sin que se confunda con sus salidas “out”.

La salida ZR será 1 si la salida es completamente 0. Por lo tanto, se utilizaron los chips Or8Way y Or. De forma que con por lo menos un 1 en la salida, se obtendrá un 1, y si es completamente 0, saldrá un 0. Por lo que luego habrá que negar la salida del chip Or para poder obtener ZR.

La salida NG determina si la salida es negativa (bit más significativo igual a 1). Por lo que se utilizó un chip And, donde la primera entrada será el bit 15 de la salida y la segunda será 1. De forma que si salida[15] es 0, osea positivo, NG será 0. Y si salida[15] es 1, osea negativo, NG será 1.

```
CHIP ALU {
    IN
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute (out = x + y) or (out = x & y)?
        no; // negate the out output?
    OUT
        out[16], // 16-bit output
        zr,      // if (out == 0) equals 1, else 0
        ng;      // if (out < 0)  equals 1, else 0
```

```
    PARTS:
    // Modificaciones en X
    Mux16(a= x, b= false, sel= zx, out= outZX);
    Not16(in= outZX, out= outNX);
    Mux16(a= outZX, b= outNX, sel= nx, out= outX);

    // Modificaciones en Y
    Mux16(a= y, b= false, sel= zy, out= outZY);
    Not16(in= outZY, out= outNY);
    Mux16(a= outZY, b= outNY, sel= ny, out= outY);

    // Selección de operación
    And16(a= outX, b= outY, out= andXY);
    Add16(a = outX, b = outY, out = sumXY);
    Mux16(a= andXY, b= sumXY, sel= f, out= funXY);

    // Negación de la salida
    Not16(in= funXY, out= notfunXY);
    Mux16(a= funXY, b= notfunXY, sel= no, out= out);
    Mux16(a= funXY, b= notfunXY, sel= no, out= salida);

    // Comprobar si la salida es 0
    Or8Way(in= salida[0..7], or= orOut1);
    Or8Way(in= salida[8..15], or= orOut2);
    Or(a= orOut1, b= orOut2, out= orOut);
    Not(in=orOut, out=zr);

    // Determinar si la salida es negativa
    And(a=salida[15], b=true, out=ng);
}
```

Explique las principales diferencias entre la lógica aritmética y la lógica secuencial.

La **lógica aritmética** se enfoca en realizar operaciones matemáticas como sumas, restas, multiplicaciones, y divisiones en circuitos digitales, utilizando compuertas lógicas que manipulan valores binarios para resolver cálculos.

La **lógica secuencial**, por otro lado, involucra circuitos que dependen no solo de entradas actuales, sino también de su estado previo. Esto significa que almacena y recuerda información a través de elementos como flip-flops, y se usa en la creación de sistemas que requieren memoria, como contadores o registros.

Bibliografia

ResearchGate - Full Adder Circuit Diagram and Truth Table:
ResearchGate. (n.d.). *Full adder circuit diagram and truth table, where A, B, and C-in are binary inputs*. Retrieved from https://www.researchgate.net/figure/Full-adder-circuit-diagram-and-truth-table-where-A-B-and-C-in-are-binary-inputs_fig2_349727409