

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

Proyecto 2

HalfAdder

La primera parte del proyecto 2 es hacer un sumador de dos números de 1 bit cada uno, gracias a la simplicidad de la tabla de verdad para suma y acarreo se puede escribir la expresión de cada uno terminando en:

```
//sum
Not(in= a, out= nota);
Not(in= b, out= notb);
And(a= nota, b= b, out= firsum);
And(a= a, b= notb, out= secsum);

Or(a= firsum, b= secsum, out= sum);

//carry
And(a= a, b= b, out= carry);
```

a	b	sum	car
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Add16

Necesitamos sumar dos inputs de 16 bits cada uno, entonces como indica la suma binaria, será necesario para la segunda suma en adelante, sumar 3 bits al tiempo (los dos de entrada y el de carry), entonces la primera suma es un HalfAdder y el carry será el input 'c' de el siguiente FullAdder, y así llevaremos los carry hasta terminar de sumar los 16 bits, y como indica el ejercicio, se desprecia el resultado del último carry.

AND2Tetris / Hardware Simulator

HDL Builtin Project 2 Add16 Chip Add16 Eval Reset Clock: 0 Test Edit Slow Fast

```

1 // This file is part of www.nand2tetris.org
2 // and the book "The Elements of Computing Systems"
3 // by Nisan and Schocken, MIT Press.
4 // File name: projects/2/Add16.hdl
5 /**
6  * 16-bit adder: Adds two 16-bit two's complement values.
7  * The most significant carry bit is ignored.
8  */
9 CHIP Add16 {
10     IN a[16], b[16];
11     OUT out[16];
12
13     PARTS:
14         /// Replace this comment with your code.
15         HalfAdder(a=a[0], b=b[0], sum=out[0], carry=carry0);
16         FullAdder(a=a[1], b=b[1], c=carry0, sum=out[1], carry=carry1);
17         FullAdder(a=a[2], b=b[2], c=carry1, sum=out[2], carry=carry2);
18         FullAdder(a=a[3], b=b[3], c=carry2, sum=out[3], carry=carry3);
19         FullAdder(a=a[4], b=b[4], c=carry3, sum=out[4], carry=carry4);
20         FullAdder(a=a[5], b=b[5], c=carry4, sum=out[5], carry=carry5);
21         FullAdder(a=a[6], b=b[6], c=carry5, sum=out[6], carry=carry6);
22         FullAdder(a=a[7], b=b[7], c=carry6, sum=out[7], carry=carry7);
23         FullAdder(a=a[8], b=b[8], c=carry7, sum=out[8], carry=carry8);
24         FullAdder(a=a[9], b=b[9], c=carry8, sum=out[9], carry=carry9);
25         FullAdder(a=a[10], b=b[10], c=carry9, sum=out[10], carry=carry10);
26         FullAdder(a=a[11], b=b[11], c=carry10, sum=out[11], carry=carry11);
27         FullAdder(a=a[12], b=b[12], c=carry11, sum=out[12], carry=carry12);
28         FullAdder(a=a[13], b=b[13], c=carry12, sum=out[13], carry=carry13);
29         FullAdder(a=a[14], b=b[14], c=carry13, sum=out[14], carry=carry14);
30         FullAdder(a=a[15], b=b[15], c=carry14, sum=out[15], carry=carry15);
31 }

```

Input pins
a: 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0
b: 1 0 0 1 1 0 0 0 0 1 1 1 0 1 1 0

Output pins
out: 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

Internal pins
carry0: 0
carry1: 0
carry2: 1
carry3: 0
carry4: 1
carry5: 1
carry6: 1
carry7: 0
carry8: 0
carry9: 0
carry10: 0
carry11: 0
carry12: 1
carry13: 0
carry14: 0
carry15: 0

Test Script Compare File Output File Diff Table

	a	b	out
0	0000000000000000	0000000000000000	0000000000000000
1	0000000000000000	1111111111111111	1111111111111111
2	1111111111111111	1111111111111111	1111111111111100
3	1000000000000000	0000000000000000	1111111111111111
4	0011100110000011	0000111111110000	0100110010010011
5	0001001000110100	1001100000110110	1010101010101010

Inc 16

Simplemente es un Add16 en el que su salida es igual a la entrada más uno.

HDL Builtin Project 2 Inc16 Chip Inc16 Eval Reset Clock: 0 Test Edit Slow Fast

```

1 /**
2  * 16-bit incrementer:
3  * out = in + 1
4  */
5 CHIP Inc16 {
6     IN in[16];
7     OUT out[16];
8
9     PARTS:
10         Add16(a=in, b[0]=true, out=out);
11 }
12 }

```

Input pins
in: 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1

Output pins
out: 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0

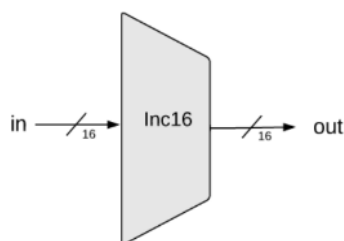
Test Script Compare File Output File Diff Table

```

1 // This file is part of www.nand2tetris.org
2 // and the book "The Elements of Computing Systems"
3 // by Nisan and Schocken, MIT Press.
4 // File name: projects/2/Inc16.tst
5
6 output-list in%B1.16.1 out%B1.16.1;
7
8 set in %B0000000000000000 // in = 0
9 eval,
10 output;
11
12 set in %B1111111111111111 // in = -1
13 eval,
14 output;
15
16 set in %B00000000000000101 // in = 5
17 eval,
18 output;
19
20 set in %B1111111111111011 // in = -5
21 eval,
22 output;

```

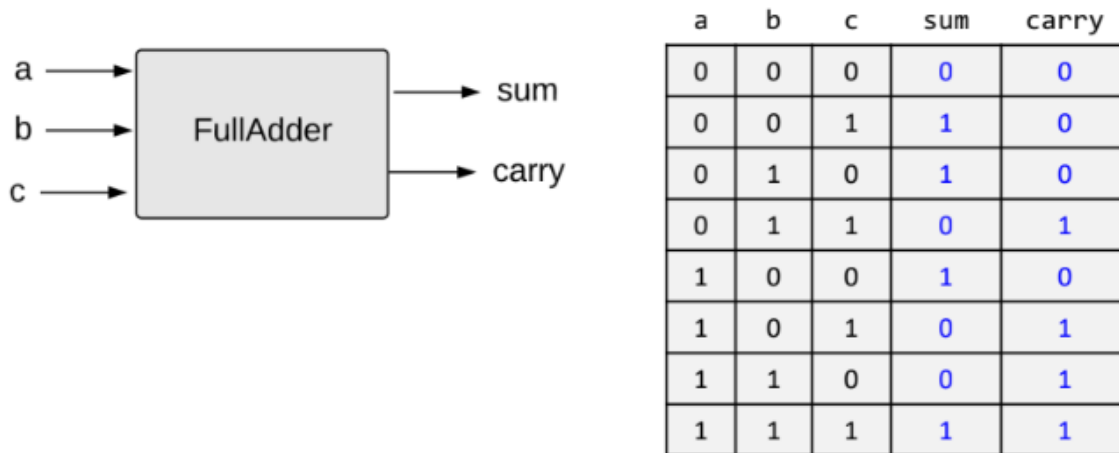
Simulation successful: The output file is identical to the compare file



FullAdder

En el caso del sumador completo, se puede aplicar la propiedad *asociativa* de la suma: La cual propone que $(a + b) + c = a + (b + c)$. Por ende se puede hacer la primera adición con un medio sumador, su resultado almacenado en la variable *aux Sum* será sumado nuevamente (usando otro medio sumador) a la entrada *c* restante

y de ahí se obtiene el resultado final de la suma, sin embargo cada medio sumador utilizado tiene su propio acarreo (auxCarry1 y auxCarry2 respectivamente), por lo que usando una compuerta Xor, siempre que estos 2 sean diferentes la salida Out será igual a 1, lo que significa que llevará un acarreo.



Proyecto 3

Bit

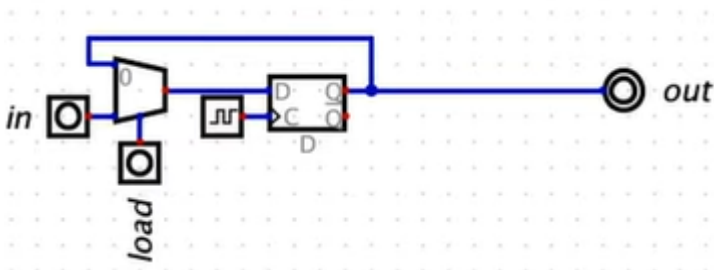
Aquí por primera vez en el proyecto se usan flip flops, en este caso el tipo D el cual se usa para guardar la entrada de D sea esta 1 o 0 en el momento que haya diferencia de voltaje en el reloj, además del flip flop se usa un multiplexor el cual funciona para plantear una condición donde si el selector es positivo, "out" muestra lo que haya en "b" y si es 0 lo de "a".

En el caso del componente Bit lo que se busca es mantener una entrada y esta poder cambiarla cada que se desee, aquí se usa la variable "load" que cuando sea positiva, lo que haya en "in" se carga y se muestra y este resultado se mantiene hasta que se cambie "in" y a la vez "load" vuelva a ser positivo. Si "load" es negativo se muestra el Bit guardado

```
// si load es positivo el multiplexor elige in,
// si no elige el valor anterior
// usamos in para definir si queremos un 0 o 1
// y load caundo queramos cambiar a ese bit elegido

// en este caso el Mux sirve como if
Mux(a= otroout, b= in, sel= load, out= dffout);
// el flipflop como memoria
DFF(in= dffout, out= out, out = otroout);
```

Aquí se puede observar como cuando "load" = 0 el multiplexor elige la salida anterior, en el caso de que "load" = 1 se elige el valor de "in" y este entra al flip flop manteniéndose.



Register

El bit implementado previamente, también puede ser llamado como registrador de un solo bit, por lo que en este chip en específico lo que se hace es generalizarlo a 16 bits, hacerlo no es muy complicado, simplemente se toman cada uno de los 16 bits de la entrada y se almacenan en su respectiva compuerta Bit, para generar un Out específico para ese bit (observar la imagen), out[0], out[1], out[2], ... , out[15].

```
CHIP Register {
    IN in[16], load;
    OUT out[16];

    PARTS:
        Bit(in=in[0], load=load, out=out[0]);
        Bit(in=in[1], load=load, out=out[1]);
        Bit(in=in[2], load=load, out=out[2]);
        Bit(in=in[3], load=load, out=out[3]);
        Bit(in=in[4], load=load, out=out[4]);
        Bit(in=in[5], load=load, out=out[5]);
        Bit(in=in[6], load=load, out=out[6]);
        Bit(in=in[7], load=load, out=out[7]);
        Bit(in=in[8], load=load, out=out[8]);
        Bit(in=in[9], load=load, out=out[9]);
        Bit(in=in[10], load=load, out=out[10]);
        Bit(in=in[11], load=load, out=out[11]);
        Bit(in=in[12], load=load, out=out[12]);
        Bit(in=in[13], load=load, out=out[13]);
        Bit(in=in[14], load=load, out=out[14]);
        Bit(in=in[15], load=load, out=out[15]);
}
```

RAM 8

Se usa DMux por que es necesario direccionar hacia qué Register va a dirigirse la información, puesto que cualquiera de los ocho Register puede ser la entrada para el Mux (el Mux se debe usar dado que el load=1 indica si el valor de address se carga a la memoria almacenada o no), es ahí cuando se ve conveniente el uso del Mux8Way16, 8 entradas y una salida de 16 bits, además del select de tres bits (address).

Entonces, cada Register va a almacenar el valor de el bit correspondiente, y quien ordena que Register se activa según la entrada de load, va a ser un DMux8Way.

The screenshot shows the NAND2Tetris Hardware Simulator interface. The top panel displays the HDL code for the RAM8 chip, which uses a DMux8Way component to route data from 8 registers to a 16-bit output based on a 3-bit address. The right panel shows the chip's configuration with input pins (in, load, address) and output pins (out) set to specific values. The bottom panel shows the test script and the successful simulation result.

HDL Code:

```
12 IN in[16], load, address[3];
13 OUT out[16];
14
15 PARTS:
16     /// Replace this comment with your code.
17     DMux8Way(in= load, sel= address, a= dmuxA, b= dmuxB,
18     c= dmuxC, d= dmuxD, e= dmuxE, f= dmuxF, g= dmuxG, h= dmuxH);
19
20     Register(in= in, load= dmuxA, out= A);
21     Register(in= in, load= dmuxB, out= B);
22     Register(in= in, load= dmuxC, out= C);
23     Register(in= in, load= dmuxD, out= D);
24     Register(in= in, load= dmuxE, out= E);
25     Register(in= in, load= dmuxF, out= F);
26     Register(in= in, load= dmuxG, out= G);
27     Register(in= in, load= dmuxH, out= H);
28
29     Mux8Way16(a= A, b= B, c= C, d= D, e= E, f= F, g= G, h= H, sel= address, out= out);
30 }
```

Chip RAM8 Configuration:

- Input pins: in = 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1, load = 0, address = 1 1 1
- Output pins: out = 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
- Internal pins: dmuxA = 0, dmuxB = 0, dmuxC = 0, dmuxD = 0

Test Script:

```
547 output;
548 set address 5,
549 eval,
550 output;
551 set address 6,
552 eval,
553 output;
554 set address 7,
555 eval,
556 output;
557
```

Simulation successful: The output file is identical to the compare file

RAM 64

Podemos construir una RAM de 64 registros a partir de un arreglo de ocho chips de RAM de 8 registros, para seleccionar un registro en particular de la memoria usamos una dirección de 6 bits.

8

IN in[16], load, address[6];

9

OUT out[16];

10

11

PARTS:

12

DMux8Way(in=load, sel=address[3..5], a=load1, b=load2, c=load3, d=load4, e=load5, f=load6, g=load7, h=load8);

13

RAM8(in=in, load=load1, address=address[0..2], out=out1);

14

RAM8(in=in, load=load2, address=address[0..2], out=out2);

15

RAM8(in=in, load=load3, address=address[0..2], out=out3);

16

RAM8(in=in, load=load4, address=address[0..2], out=out4);

17

RAM8(in=in, load=load5, address=address[0..2], out=out5);

18

RAM8(in=in, load=load6, address=address[0..2], out=out6);

19

RAM8(in=in, load=load7, address=address[0..2], out=out7);

20

RAM8(in=in, load=load8, address=address[0..2], out=out8);

21

Mux8Way16(a=out1, b=out2, c=out3, d=out4, e=out5, f=out6, g=out7, h=out8, sel=address[3..5], out=out);

22

Chip RAM64

Eval

Reset

Clock: 81

Input pins

in 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 de

load 0

address 1 1 1 0 1 dec

Output pins

out 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 de

Internal pins

load1 0

load2 0

Test

Edit

Slow

Fast

Test Script

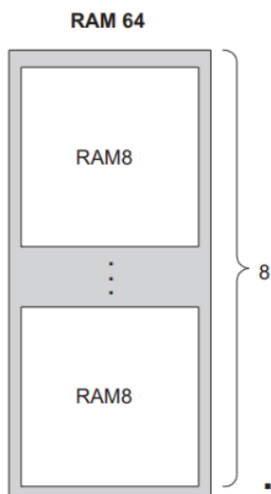
Compare File

Output File

Diff Table

time	in	load	address	out
0+	0	0	0	0
1	0	0	0	0
1+	0	1	0	0
2	0	1	0	0
2+	1313	0	0	0
3	1313	0	0	0
3+	1313	1	13	0

Simulation successful: The output file is identical to the compare file



RAM4K

En este caso el proceso anterior vuelve a repetirse, en donde el primer demultiplexor sirve como selector de entre las 8 particiones principales usando para este fin los 3 primeros y más significativos dígitos de "address", después con RAM512 que recibe 9 dígitos se le pasan los 9 menos significativos del actual "address", logrando en este proceso seleccionar el correcto entre las 4096 posibilidades que existen.

Seguido de esto como en los anteriores se guarda en "out" lo que haya en "in" si es que "load" fue positivo, caso contrario RAM512 no hubiera cambiado su "out" y consigo tampoco RAM4K

```

DMux8Way(in= load, sel= address[0..2], a= l1, b= l2, c= l3,
d= l4, e= l5, f= l6, g= l7, h= l8);

RAM512(in= in, load= l1, address= address[3..11], out= first);
RAM512(in= in, load= l2, address= address[3..11], out= second);
RAM512(in= in, load= l3, address= address[3..11], out= thirth);
RAM512(in= in, load= l4, address= address[3..11], out= fourth);
RAM512(in= in, load= l5, address= address[3..11], out= fifth);
RAM512(in= in, load= l6, address= address[3..11], out= sixth);
RAM512(in= in, load= l7, address= address[3..11], out= seventh);
RAM512(in= in, load= l8, address= address[3..11], out= eighth);

Mux8Way16(a= first, b= second, c= thirth, d= fourth,
e= fifth, f= sixth, g= seventh, h= eighth,
sel= address[0..2], out= out);

```

RAM16K

En este caso, se reutiliza la estructura de las 4 memorias RAM construidas anteriormente, pero se simplifica levemente, pues solo se utilizan 4 memorias RAM4K para alcanzar el valor deseado de 16k, quedando la implementación de la siguiente manera:

```

CHIP RAM16K {
    IN in[16], load, address[14];
    OUT out[16];

    PARTS:
        DMux4Way(in=load, sel=address[0..1], a=dA, b=dB, c=dC, d=dD) ;

        RAM4K(in=in, load=dA, address=address[2..13],out=memA);
        RAM4K(in=in, load=dB, address=address[2..13],out=memB);
        RAM4K(in=in, load=dC, address=address[2..13],out=memC);
        RAM4K(in=in, load=dD, address=address[2..13],out=memD);

        Mux4Way16(a=memA, b=memB, c=memC, d=memD,
        |         |         |         |
        |         |         |         |
        sel=address[0..1], out=out);
}

```

La estructura y el funcionamiento lógico son similares en ambos niveles: tanto para los chips RAM512 individuales como para la RAM4k que los agrupa. Tanto el demultiplexor como el multiplexor (simplificados, siendo un DMux4Way y un Mux4Way16 en este caso) son utilizados para seleccionar la dirección de memoria a interactuar, utilizando los bits de la entrada address. Luego, el valor de la señal load se envía a esa dirección

seleccionada. De esta manera, cada chip RAM512 dentro de la RAM4K realiza su función según el valor de la señal load. Es importante destacar que los demultiplexores (DMux) y multiplexores (Mux) operan con los bits menos significativos (en este caso solo necesitan 2, el address[0] y el address[1]) de la variable address, mientras que los chips RAM4K utilizan los bits más significativos.

PC (Program Counter)

La idea del PC es implementar un contador, con la funcionalidad de resetear el conteo, iniciarlo donde se desee, e incrementarlo, incluso dejar el valor anterior si no se desea incrementar en un pulso dado.

Para lograr esta implementación, como la funcionalidad nos dice que hay tres if, necesitaremos tres Multiplexores, en este caso de 16 bits, dado que ese es el input, codificamos cada una de las funcionalidades anteriormente descritas, lo más importante es tener enlazadas estas funcionalidades, por ejemplo, el Multiplexor encargado de resetear el conteo va a recibir como entrada 'a' el valor previo de la salida del Mux. de load, y a su vez este tiene en su entrada la salida del Mux. de incremento; todo esto debido a que podemos dar varias instrucciones y debemos poder contar con la salida de la funcionalidad anterior.

NAND2Tetris / Hardware Simulator

Project 3 PC Chip PC

```

1 // This file is part of www.nand2tetris.org
2 // and the book "The Elements of Computing Systems"
3 // by Nisan and Schocken, MIT Press.
4 // File name: projects/3/PC.hdl
5 /**
6  * A 16-bit counter.
7  * if reset(t): out(t+1) = 0
8  * else if load(t): out(t+1) = in(t)
9  * else if inc(t): out(t+1) = out(t) + 1
10 * else out(t+1) = out(t)
11 */
12 CHIP PC {
13     IN in[16], reset, load, inc;
14     OUT out[16];
15
16     PARTS:
17         // Replace this comment with your code.
18         Register(in= reset_mux_out, load= true, out= out, out =
19         Inc16(in= register_out, out= inc_out);
20
21         Mux16(a= load_mux_out, b[0..15]= false, sel= reset, out=
22         Mux16(a= inc_mux_out, b= in, sel= load, out= load_mux_out
23         Mux16(a= register_out, b= inc_out, sel= inc, out= inc_mux
24 }

```

Input pins

in: 0 1 0 1 0 1 1 0 1 1 0 0 1 1 1 0 dec

reset: 1

load: 0

inc: 0

Output pins

out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 dec

Internal pins

reset_mux_out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 dec

register_out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 dec

inc_out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 dec

load_mux_out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 dec

inc_mux_out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 dec

Visualization

Register: 0

Test Script Compare File Output File Diff Table

time	in	reset	load	inc	out
0+	0	0	0	0	0
1	0	0	0	0	0
1+	0	0	0	1	0
2	0	0	0	1	1
2+	-32123	0	0	1	1
3	-32123	0	0	1	2
3+	-32123	0	1	1	2
4	-32123	0	1	1	-32123
4+	-32123	0	0	1	-32123
5	-32123	0	0	1	-32122
5+	-32123	0	0	1	-32122
6	-32123	0	0	1	-32121
6+	12345	0	1	0	-32121
7	12345	0	1	0	12345
7+	12345	1	1	0	12345
8	12345	1	1	0	0
8+	12345	0	1	1	0
9	12345	0	1	1	12345
9+	12345	1	1	1	12345
10	12345	1	1	1	0
10+	12345	0	0	1	0
11	12345	0	0	1	1
11+	12345	1	0	1	1
12	12345	1	0	1	0
12+	0	0	1	1	0
13	0	0	1	1	0
13+	0	0	0	1	0
14	0	0	0	1	1
14+	22222	1	0	0	1
15	22222	1	0	0	0

Simulation successful: The output file is identical to the compare file

¿Cuál es el objetivo de cada uno de esos proyectos con sus palabras y qué se debe hacer para desarrollarlo?

En el proyecto 2 se busca construir gradualmente un conjunto de chips que realizan sumas aritméticas, culminando en la construcción del chip ALU de la computadora Hack. Partimos del conjunto de puertas lógicas que se construyeron en el capítulo 1 y finalizamos con una Unidad Lógica Aritmética completamente funcional. ALU es el componente central encargado de ejecutar todas las operaciones lógicas y aritméticas de la computadora. Por lo tanto, desarrollar la funcionalidad de la ALU es un paso clave para entender el funcionamiento de la Unidad Central de Procesamiento (CPU) y del sistema informático en general.

En el proyecto 3 se busca construir gradualmente una unidad RAM utilizando las compuertas lógicas para almacenar bits a lo largo del tiempo y así mismo localizar el registro de memoria sobre el que queremos operar.

¿Qué tipo de unidades aritmético lógicas existen?

ALU de propósito general:

Estas ALU están diseñadas para realizar operaciones aritméticas (suma, resta, multiplicación, división) y lógicas (AND, OR, XOR, NOT).

ALU de propósito específico:

Estas ALU están diseñadas para realizar un conjunto limitado de operaciones específicas a una tarea o aplicación particular. Por ejemplo, pueden estar optimizadas para procesamiento de señales o gráficos.

ALU de bit único (bit-slice ALU):

Se construyen para operar sobre un solo bit de los operandos a la vez. Un procesador puede usar varias ALU de bit-slice en paralelo para manejar palabras de varios bits. Estas ALU permiten la creación de procesadores personalizables.

ALU de enteros:

Estas ALU se encargan de operaciones aritméticas y lógicas con números enteros. Suelen ser usadas en la mayoría de los procesadores para manejar datos enteros y direccionamiento de memoria.


ALU de coma flotante (FPU - Floating Point Unit):

Estas ALU están diseñadas para trabajar con números en coma flotante (decimales) y son muy utilizadas en cálculos científicos, gráficos y otras aplicaciones que requieren precisión matemática.

ALU SIMD (Single Instruction, Multiple Data):

Este tipo de ALU permite realizar la misma operación en múltiples datos simultáneamente, lo cual es útil en aplicaciones como procesamiento multimedia o gráficos.

Referencias

1. The Elements of Computing Systems (2nd ed.), Nisan and Schocken, MIT Press.
2.  **NAND To Tetris 5a: Creating RAM and Memory lab**
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>