

Informe Parcial 1

Matías Martínez, Emanuel Ossa y Jerónimo Restrepo

Universidad Eafit, Medellín

Organización de Computadores

Alberto Arias

Medellín

28 de febrero de 2025

Puntos desarrollados en clase con los fragmentos en HDL requeridos:

1.XOR

Parcial 1

Emmanuel Osan
Jesús Rodríguez
María Martínez

1. Diseñe una compuerta XOR con NAND

1.1 Tabla de verdad de compuerta XOR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

1.2 $(a \cdot \sim b) + (\sim a \cdot b)$

1.3 Using NAND

1.4

$$XOR(A,B) = NAND(NAND(A, NAND(A,B)), NAND(B, NAND(A,B)))$$

```
CHIP XOR {
  IN a, b;
  OUT out;
  PARTS:
    NAND(a=a, b=b, out=nand1);
    NAND(a=a, b=nand1, out=nand2);
    NAND(a=b, b=nand1, out=nand3);
    NAND(out=nand2, b=nand3, out=out);
}
```

```

CHIP Xor {
  IN a, b;
  OUT out;

  PARTS:
    Nand(a=a , b=b , out=nand1 );
    Nand(a=a , b=nand1 , out=nand2 );
    Nand(a=b , b=nand1 , out=nand3 );
    Nand(a=nand2 , b=nand3 , out=out );
}

```

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Chip Xor	
Input pins	
a	0
b	1
Output pins	
out	1
Internal pins	
nand1	1
nand2	1
nand3	0

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

2. Compuerta AND con NAND

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

  PARTS:
    Nand(a=a, b=b, out=nand);
    Nand(a=nand, b=nand, out=out);
}
```

Chip And			
Input pins			
a	1	b	0
Output pins			
out	0		
Internal pins			
nand	1		

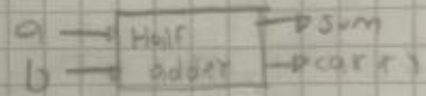
Simulation successful: The output file is identical to the compare file

3. Half Adder

3.1 HALF ADDER

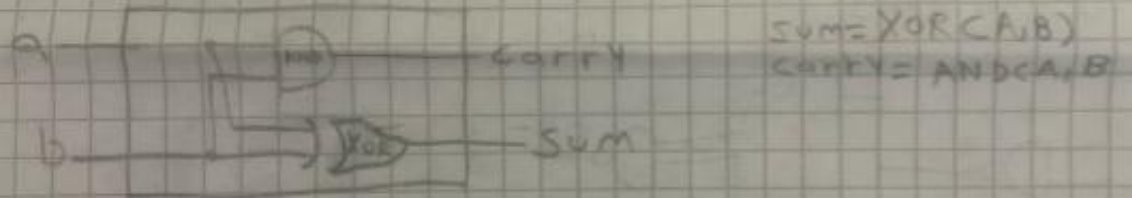
3.1

a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



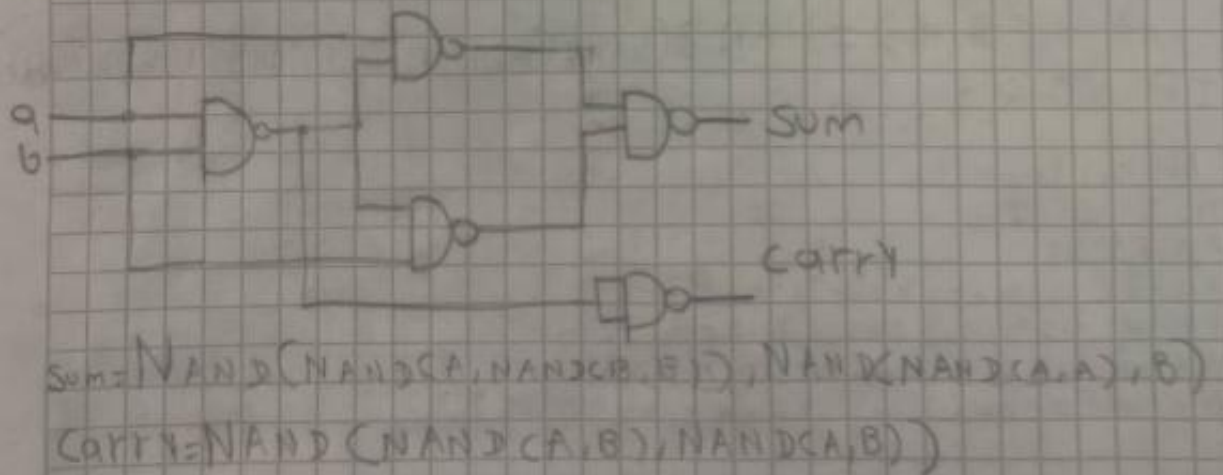
3.2

Diagrama con compuertas lógicas básicas



3.3

Diagrama usando NAND



```

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

    PARTS:
        Nand(a=a , b=a , out=nota);
        Nand(a=nota , b=b , out=n1 );
        Nand(a=b , b=b , out=notb );
        Nand(a=a , b=notb , out=n2 );
        Nand(a=n1 , b=n2 , out=sum );

        Nand(a=a , b=b , out=n3 );
        Nand(a=n3 , b=n3 ,out=carry);
}

```

					Input pins	
a	b	sum	car		a	1
0	0	0	0		b	1
0	1	1	0		Output pins	
1	0	1	0		sum	0
1	1	0	1		carry	1
					Internal pins	
					nota	0
					n1	1
					notb	0
					n2	1
					n3	0

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

Half adder con compuertas básicas:

Handwritten code for a Half Adder using basic gates:

```

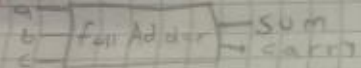
CHIP Halfadder {
    IN a, b;
    OUT Sum, carry;

    PARTS:
        XORC(a=a, b=b, out=Sum); // Sum
        ANDC(a=a, b=b, out=carry); // carry
}

```

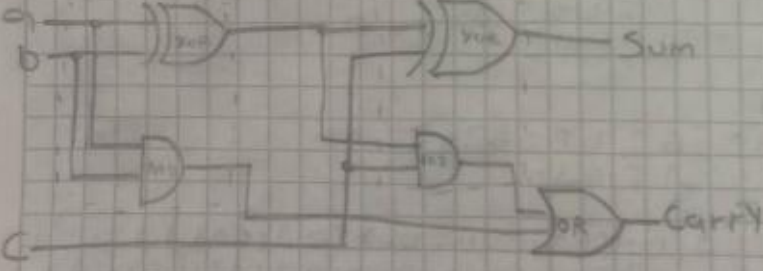
4. Full Adder

4.1 Full Adder



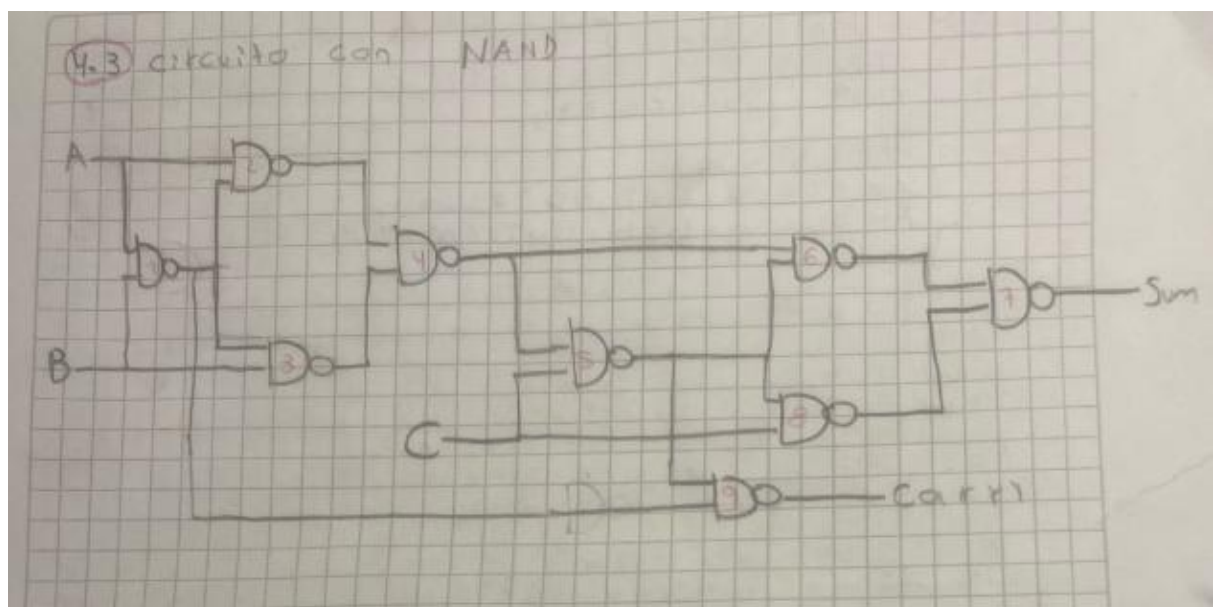
a	b	c	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

4.2 Diagrama con compuertas lógicas básicas



$$S = \text{XOR}(\text{XOR}(A, B), C_{in})$$

$$C = \text{OR}(\text{AND}(A, B), \text{AND}(C, \text{XOR}(A, B)))$$



```

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:
        HalfAdder(a=a,b=b,sum=ab,carry=cab);
        HalfAdder(a=c,b=ab,sum=sum,carry=s);
        Or(a=cab,b=s,out=carry);
}

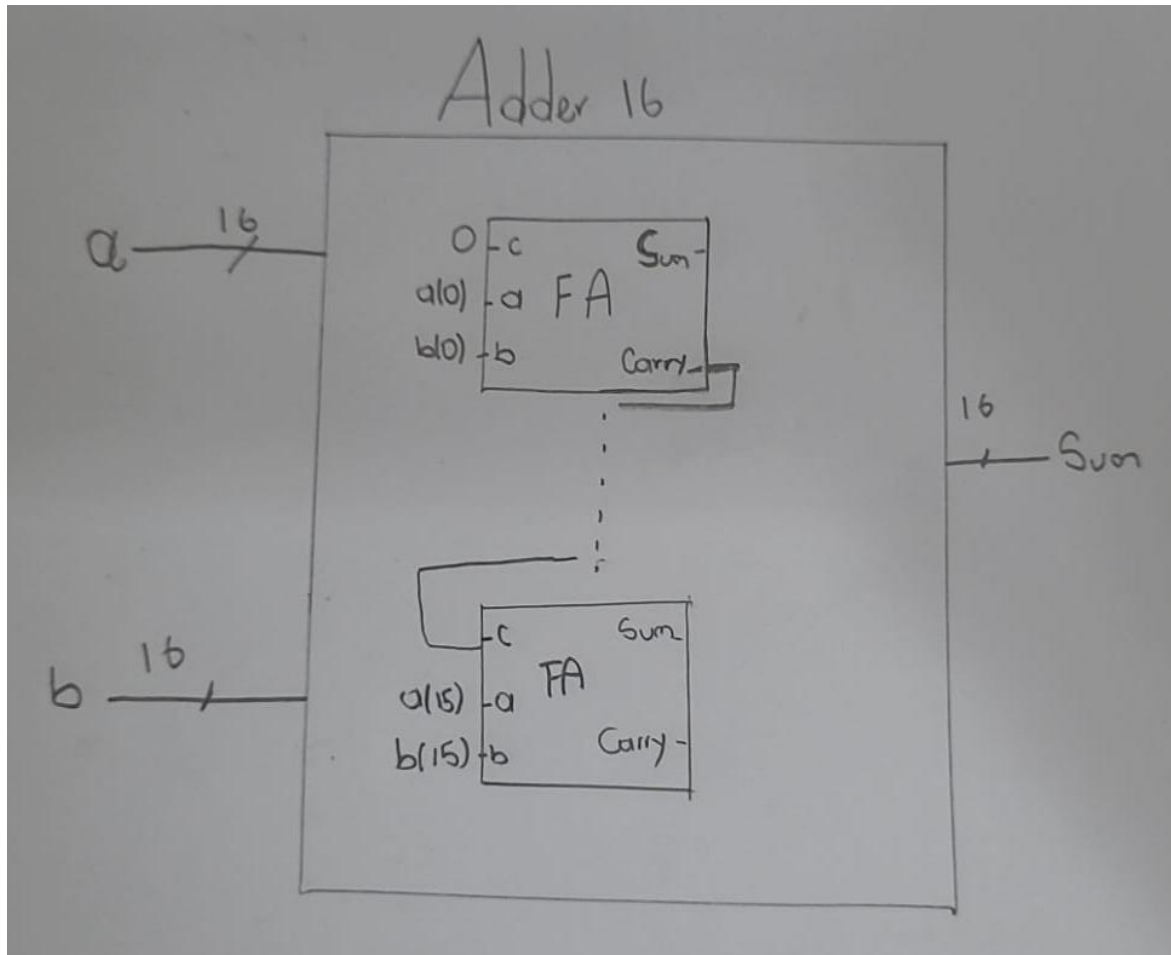
```

a	b	c	sum	carry	Input pins
0	0	0	0	0	a 1
0	0	1	1	0	b 1
0	1	0	1	0	c 0
0	1	1	0	1	Output pins
1	0	0	1	0	sum 0
1	0	1	0	1	carry 1
1	1	0	0	1	Internal pins
1	1	1	1	1	ab 0
					cab 1
					s 0

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

5. Add 16

- Diseñe un ADDER de 16 bits utilizando los diagramas de los FULL ADDER (No incluya las compuertas internas, solamente el esquema de entradas y salidas de cada FULL ADDER).



```

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

  PARTS:
    FullAdder(a=a[0],b=b[0],sum=out[0],carry=c);
    FullAdder(a=a[1],b=b[1],c=c,sum=out[1],carry=d);
    FullAdder(a=a[2],b=b[2],c=d,sum=out[2],carry=e);
    FullAdder(a=a[3],b=b[3],c=e,sum=out[3],carry=f);
    FullAdder(a=a[4],b=b[4],c=f,sum=out[4],carry=g);
    FullAdder(a=a[5],b=b[5],c=g,sum=out[5],carry=h);
    FullAdder(a=a[6],b=b[6],c=h,sum=out[6],carry=i);
    FullAdder(a=a[7],b=b[7],c=i,sum=out[7],carry=j);
    FullAdder(a=a[8],b=b[8],c=j,sum=out[8],carry=k);
    FullAdder(a=a[9],b=b[9],c=k,sum=out[9],carry=l);

    FullAdder(a=a[10],b=b[10],c=l,sum=out[10],carry=m);
    FullAdder(a=a[11],b=b[11],c=m,sum=out[11],carry=n);
    FullAdder(a=a[12],b=b[12],c=n,sum=out[12],carry=o);
    FullAdder(a=a[13],b=b[13],c=o,sum=out[13],carry=p);
    FullAdder(a=a[14],b=b[14],c=p,sum=out[14],carry=q);
    FullAdder(a=a[15],b=b[15],c=q,sum=out[15],carry=drop);
}

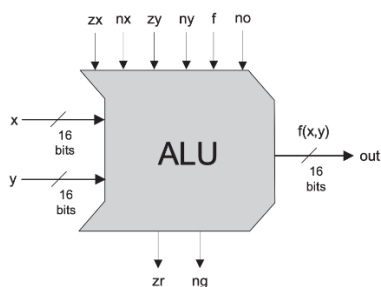
```

a	b	out
0000000000000000	0000000000000000	0000000000000000
0000000000000000	1111111111111111	1111111111111111
1111111111111111	1111111111111111	1111111111111110
1010101010101010	0101010101010101	1111111111111111
0011110011000011	0000111111110000	0100110010110011
0001001000110100	1001100001110110	1010101010101010

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

Input pins		Input pins	
a	0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0	a	4660
b	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	b	48
Output pins		Output pins	
out	0 0 0 1 0 0 1 0 0 1 1 0 0 1 0 0	out	4708

6. Implementación de una ALU en HDL



zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x y

```

CHIP ALU {
  IN
    x[16], y[16], // Entradas de 16 bits
    zx,          // Bit de control: si zx=1, la entrada x se
                  // convierte en 0; si zx=0, x se mantiene sin cambios
    nx,          // Bit de control: si nx=1, se aplica NOT a x
                  // (después de zx); si nx=0, x no se niega
    zy,          // Bit de control: si zy=1, la entrada y se
                  // convierte en 0; si zy=0, y se mantiene sin cambios
    ny,          // Bit de control: si ny=1, se aplica NOT a y
                  // (después de zy); si ny=0, y no se niega
    f,           // Bit de control: si f=1, la operación es suma (x2
                  // + y2); si f=0, la operación es AND (x2 & y2)
    no;          // Bit de control: si no=1, se aplica NOT al
                  // resultado final; si no=0, el resultado no se niega
  OUT
    out[16],     // Salida de 16 bits: resultado final de la
                  // operación de la ALU

```

```

    zr,          // zr=1 si out es exactamente 0 (todos los bits son
0); zr=0 si out no es 0
    ng;          // ng=1 si out es negativo (bit 15=1 en complemento
a 2); ng=0 si no es negativo

```

PARTS:

```

// *Preprocesamiento de x: Se ajusta x según los bits de control zx y
nx
// - Mux16 selecciona entre x y 0 basado en zx
Mux16(a=x, b=false, sel=zx, out=x1);
    // Si zx=1, selecciona b=false (0), por lo que x1=0. Si zx=0,
selecciona a=x, por lo que x1=x.
// - Not16 calcula la negación de x1 para usarla si nx lo requiere
Not16(in=x1, out=negatedX);
    // Aplica NOT bit a bit: negatedX[i] = NOT(x1[i]) para cada bit de
x1.
// - Mux16 decide si usar x1 o su negación según nx
Mux16(a=x1, b=negatedX, sel=nx, out=x2);
    // Si nx=1, selecciona b=negatedX, por lo que x2=NOT(x1). Si nx=0,
selecciona a=x1, por lo que x2=x1.

// *Preprocesamiento de y**: Se ajusta y según los bits de control zy y
ny
// - Mux16 selecciona entre y y 0 basado en zy
Mux16(a=y, b=false, sel=zy, out=y1);
    // Si zy=1, selecciona b=false (0), por lo que y1=0. Si zy=0,
selecciona a=y, por lo que y1=y.
// - Not16 calcula la negación de y1 para usarla si ny lo requiere
Not16(in=y1, out=negatedY);
    // Aplica NOT bit a bit: negatedY[i] = NOT(y1[i]) para cada bit de
y1.
// - Mux16 decide si usar y1 o su negación según ny
Mux16(a=y1, b=negatedY, sel=ny, out=y2);
    // Si ny=1, selecciona b=negatedY, por lo que y2=NOT(y1). Si ny=0,
selecciona a=y1, por lo que y2=y1.

// *Operación principal: Se calcula el resultado según el bit de
control f
// - Add16 realiza la suma de x2 y y2
Add16(a=x2, b=y2, out=added);
    // Suma bit a bit: added[i] = x2[i] + y2[i] + acarreo, generando un
resultado de 16 bits.
// - And16 realiza la operación AND entre x2 y y2
And16(a=x2, b=y2, out=xyAnd);
    // AND bit a bit: xyAnd[i] = x2[i] AND y2[i] para cada bit.
// - Mux16 selecciona entre AND y suma según f
Mux16(a=xyAnd, b=added, sel=f, out=result);
    // Si f=1, selecciona b=added(suma), por lo que result=x2+y2. Si
f=0, selecciona a=xyAnd (AND), por lo que result=x2&y2.

// Se ajusta el resultado final según el bit de control no
// - Not16 calcula la negación del resultado para usarla si no lo
requiere
Not16(in=result, out=negatedResult);
    // Aplica NOT bit a bit: negatedResult[i] = NOT(result[i]) para
cada bit de result.

```

```

    // - Mux16 decide si usar result o su negación según no, y genera la
    salida final
    Mux16(a=result, b=negatedResult, sel=no, out=out, out[15]=firstOut,
    out[0..7]=finalLeft, out[8..15]=finalRight);
    // Si no=1, selecciona b=negatedResult, por lo que out=NOT(result).
    Si no=0, selecciona a=result, por lo que out=result.
    // Además, extrae: firstOut=out[15] (bit más significativo),
    finalLeft=out[0..7], finalRight=out[8..15].

    // *Cálculo de ng (negativo): Determina si el resultado es negativo
    // - Usa el bit más significativo (out[15]) como indicador
    And(a=firstOut, b=true, out=ng);
    // ng = firstOut AND true = firstOut. Si out[15]=1, ng=1
    (negativo). Si out[15]=0, ng=0 (no negativo).

    // *Cálculo de zr (cero): Determina si el resultado es igual a 0
    // - Or8Way verifica si hay algún 1 en los primeros 8 bits (out[0..7])
    Or8Way(in=finalLeft, out=zrl);
    // zrl = OR(out[0], out[1], ..., out[7]). Si algún bit es 1, zrl=1;
    si todos son 0, zrl=0.
    // - Or8Way verifica si hay algún 1 en los últimos 8 bits (out[8..15])
    Or8Way(in=finalRight, out=zrr);
    // zrr = OR(out[8], out[9], ..., out[15]). Si algún bit es 1,
    zrr=1; si todos son 0, zrr=0.
    // - Or combina los resultados para determinar si hay algún 1 en todo
    out
    Or(a=zrl, b=zrr, out=nzr);
    // nzr = zrl OR zrr. Si zrl=1 o zrr=1, nzr=1 (out no es 0). Si
    ambos son 0, nzr=0 (out es 0).
    // - Not invierte nzr para obtener zr
    Not(in=nzr, out=zr);
    // zr = NOT(nzr). Si nzr=0 (out es 0), zr=1. Si nzr=1 (out no es
    0), zr=0.
}

```

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

7. Explique el funcionamiento de la ALU cuando el resultado es un valor

negativo

La ALU incluye una salida especial llamada ng (negative), que indica si el resultado (out[15:0]) es negativo:

$ng = 1$ si out[15] (MSB OF THE OUTPUT)= 1 (resultado negativo).

$ng = 0$ si $out[15]$ (MSB OF THE OUTPUT) = 0 (resultado no negativo).

Esto puede suceder por ejemplo al usar las señales de control para realizar $x - y$ si $x < y$ o aplicar $-x$ a un numero positivo

8. Explique el funcionamiento de la ALU cuando salida es cero (los 16 bits de salida)

La ALU tiene una salida especial llamada zr (zero), que indica si todos los 16 bits del resultado ($out[15:0]$) son 0:

$zr = 1$ si $out = 0000000000000000$.

$zr = 0$ si al menos un bit de out es 1.

Esto puede ocurrir por ejemplo con resta de números iguales y AND con cero

9. - ¿Para qué necesita un computador una ALU?

La ALU es el núcleo del procesamiento en una computadora; sin ella, la CPU no podría realizar cálculos ni tomar decisiones, lo que imposibilitaría la ejecución de cualquier algoritmo. Por ejemplo, en la ejecución de un bucle, la ALU se encarga de incrementar el contador de iteraciones y evaluar la condición de salida mediante operaciones de suma y comparación lógica. Esto es fundamental para tareas como el recorrido de listas o el procesamiento de datos, ya que permite a la máquina ejecutar operaciones secuenciales de forma precisa y eficiente

- ¿Por qué no se hacen operaciones Aritméticas y Lógicas directamente en procesador?

Las operaciones aritméticas y lógicas se ejecutan en la ALU porque está diseñada para procesarlas de manera rápida y eficiente. Separarla de la unidad de control permite

modularidad, optimización del hardware y ejecución en paralelo, mejorando el rendimiento del procesador sin sobrecargar su arquitectura principal.

- ¿Las ALUs de los computadores modernos procesan solamente dos entradas de 16 bits cada una?

Las ALUs en computadores modernos procesan datos de distintos anchos según la arquitectura, desde 16 hasta 64 bits en CPUs comunes y hasta 512 bits en ALUs vectoriales (SIMD). Aunque una ALU básica opera con dos entradas, las modernas pueden manejar múltiples operandos en paralelo en unidades avanzadas como SIMD y FPU