# The Elements of Computer Systems

## Chapter 2: Boolean Arithmetic

2022/04/21 kumo+bcali kekeho

# Contents of this chapter

**Implement logic gates for arithmetic operations and ALU**

- Introduction to code of Signed Integers
- **Adder**
  - Half-Adder, Full-Adder, 16bit Adder, Incrementer
- **ALU**

# 2.1 Background

## Binary numbers

- Convert $(10011)_{two}$ (base 2) to decimal value (base 10)

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

- Generalization

  $b$: base

$$(x_n\ x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i$$

# 2.1 Background

## Binary Addition

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | | (carry) | 1 | 1 | 1 | 1 | |
| | 1 | 0 | 0 | 1 | $x$ | | 1 | 0 | 1 | 1 |
| + | 0 | 1 | 0 | 1 | $y$ | + | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | $x + y$ | 1 | 0 | 0 | 1 | 0 |
| no overflow | | | | | | overflow | | | | |

- Same as 筆算

  1. Add each LSB

  2. Add resulting carry to sum of pair of LSB+1

  3. Continue until MSB are added

# 2.1 Background

## Signed Binary

- 2's complement method

$$\bar{x} = \begin{cases} 2^n - x & if\ x \neq 0 \\ 0 & otherwise \end{cases}$$

| Positive numbers | | Negative numbers | |
|---|---|---|---|
| 0 | 0000 | | |
| 1 | 0001 | 1111 | –1 |
| 2 | 0010 | 1110 | –2 |
| 3 | 0011 | 1101 | –3 |
| 4 | 0100 | 1100 | –4 |
| 5 | 0101 | 1011 | –5 |
| 6 | 0110 | 1010 | –6 |
| 7 | 0111 | 1001 | –7 |
| | | 1000 | –8 |

- Can code a total of $2^n$ signed numbers ($-2^{n-1} \sim 2^{n-1} - 1$)
- MSB of All positive numbers is 0
- MSB of All negative numbers is 1

# 2.1 Background

## Signed Binary

- **2's complement method**
  - To obtain the code of -x from +x:
    - Flip all the bits of +x, and add 1 to the result
  - Why 2's complement method often used?
    - Addition of any two signed numbers is the same as addition of positive numbers!

      $$(-2)+(-3)=(1110)_{two}+(1101)_{two}=(1011)_{two}=-5$$

| Positive numbers | | Negative numbers | |
|---|---|---|---|
| 0 | 0000 | | |
| 1 | 0001 | 1111 | −1 |
| 2 | 0010 | 1110 | −2 |
| 3 | 0011 | 1101 | −3 |
| 4 | 0100 | 1100 | −4 |
| 5 | 0101 | 1011 | −5 |
| 6 | 0110 | 1010 | −6 |
| 7 | 0111 | 1001 | −7 |
| | | 1000 | −8 |

# 2.2 Specification

## Adders

- Type of Adder
  - **Half-adder**(半加算器): designed to add two bits
  - **Full-adder**(全加算器): designed to add three bits
  - **Adder**(加算器): designed to add two n-bit numbers
  - **Incrementer**: designed to add 1 to a given n-bit numbers

# 2.2 Specification

## Adders | Half Adder: designed to add two bits

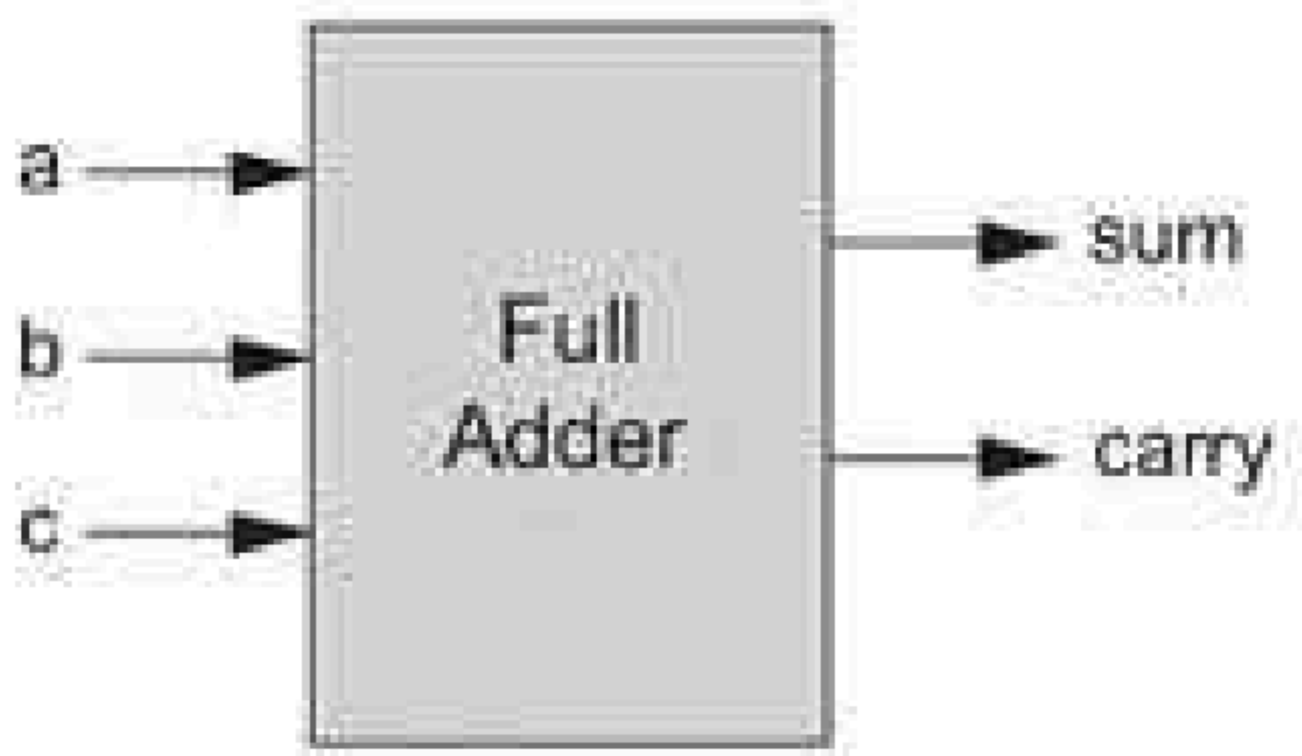
| Inputs | | Outputs | |
|---|---|---|---|
| a | b | carry | sum |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |


```
Chip name: HalfAdder
Inputs:    a, b
Outputs:   sum, carry
Function:  sum   = LSB of a + b
           carry = MSB of a + b
```

# 2.2 Specification

## Adders | Full Adder: designed to add two bits

| a | b | c | carry | sum |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |




```
Chip name: FullAdder
Inputs:    a, b, c
Outputs:   sum, carry
Function:  sum = LSB of a + b + c
           carry = MSB of a + b + c
```

# 2.2 Specification

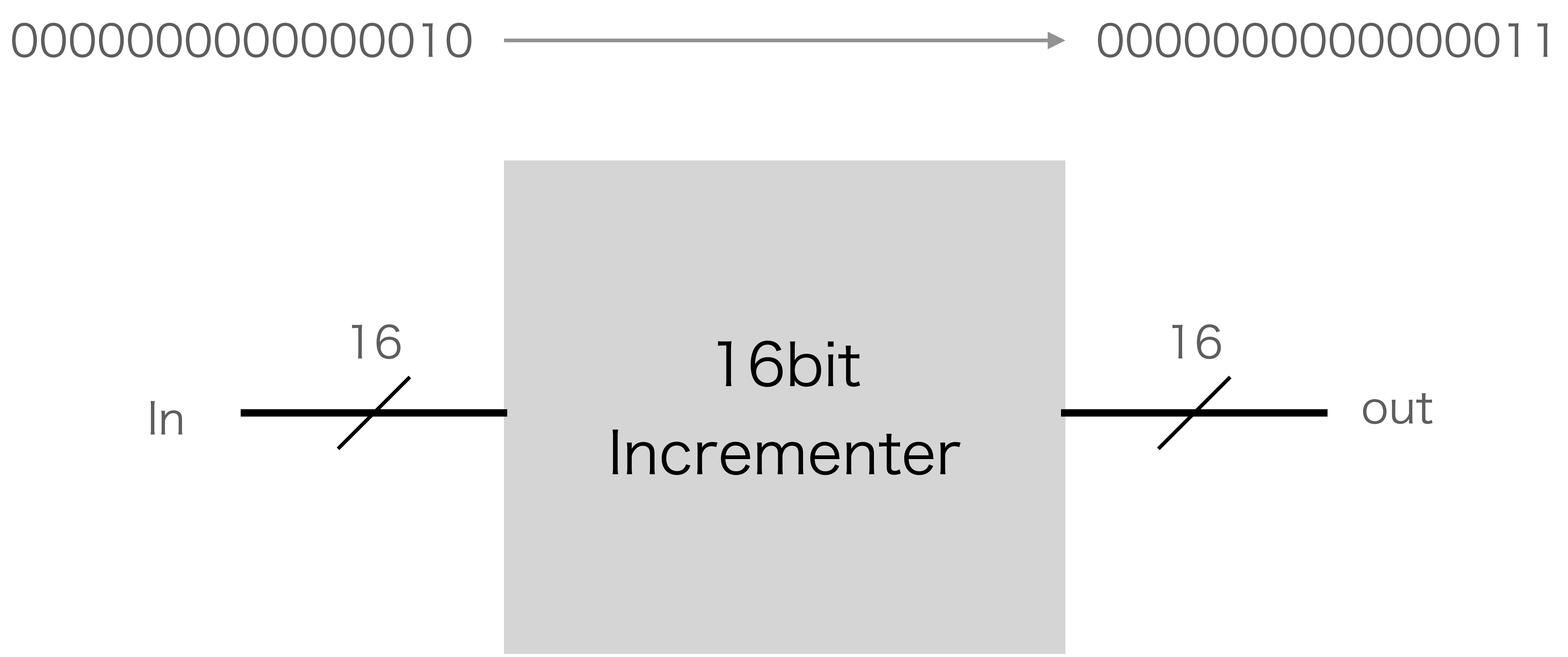
## Adders | 16-bit adder

```
... 1 0 1 1 a   +
... 0 0 1 0 b
-------------
... 1 1 0 1 out
```


```
Chip name: Add16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  out = a + b
Comment:   Integer 2's complement addition.
           Overflow is neither detected nor handled.
```

# 2.2 Specification

## Adders | Incrementer

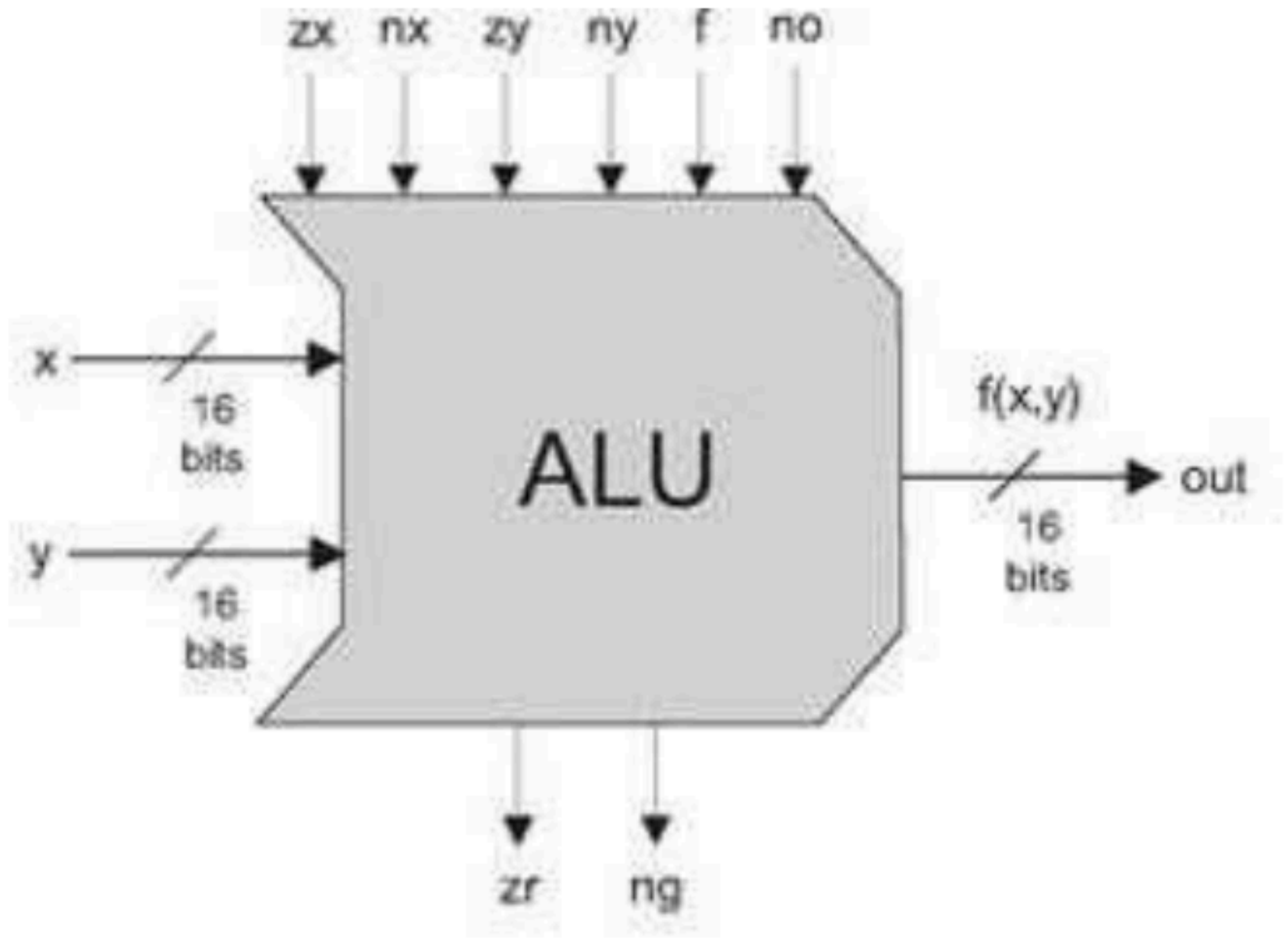



```
Chip name:  Inc16
Inputs:     in[16]
Outputs:    out[16]
Function:   out=in+1
Comment:    Integer 2's complement addition.
            Overflow is neither detected nor handled.
```

# 2.2 Specification

## ALU (Arithmetic Logic Unit)

- Design ALU for Hack computer platform
- Given a control signal, can perform various calculations on the input.
- $out = f(x, y)$
  - Select function by {zx, nx, zy, ny, f, no}




```
Chip name: ALU
Inputs:    x[16], y[16],      // Two 16-bit data inputs
           zx,                // Zero the x input
           nx,                // Negate the x input
           zy,                // Zero the y input
           ny,                // Negate the y input
           f,                 // Function code: 1 for Add, 0 for And
           no                 // Negate the out output
Outputs:   out[16],           // 16-bit output
           zr,                // True iff out=0
           ng                 // True iff out<0
Function:  if zx then x = 0         // 16-bit zero constant
           if nx then x = !x        // Bit-wise negation
           if zy then y = 0         // 16-bit zero constant
           if ny then y = !y        // Bit-wise negation
           if f then out = x + y    // Integer 2's complement addition
                else out = x & y    // Bit-wise And
           if no then out = !out    // Bit-wise negation
           if out=0 then zr = 1 else zr = 0  // 16-bit eq. comparison
           if out<0 then ng = 1 else ng = 0  // 16-bit neg. comparison
Comment:   Overflow is neither detected nor handled.
```

# 2.2 Specification

## ALU (Arithmetic Logic Unit)

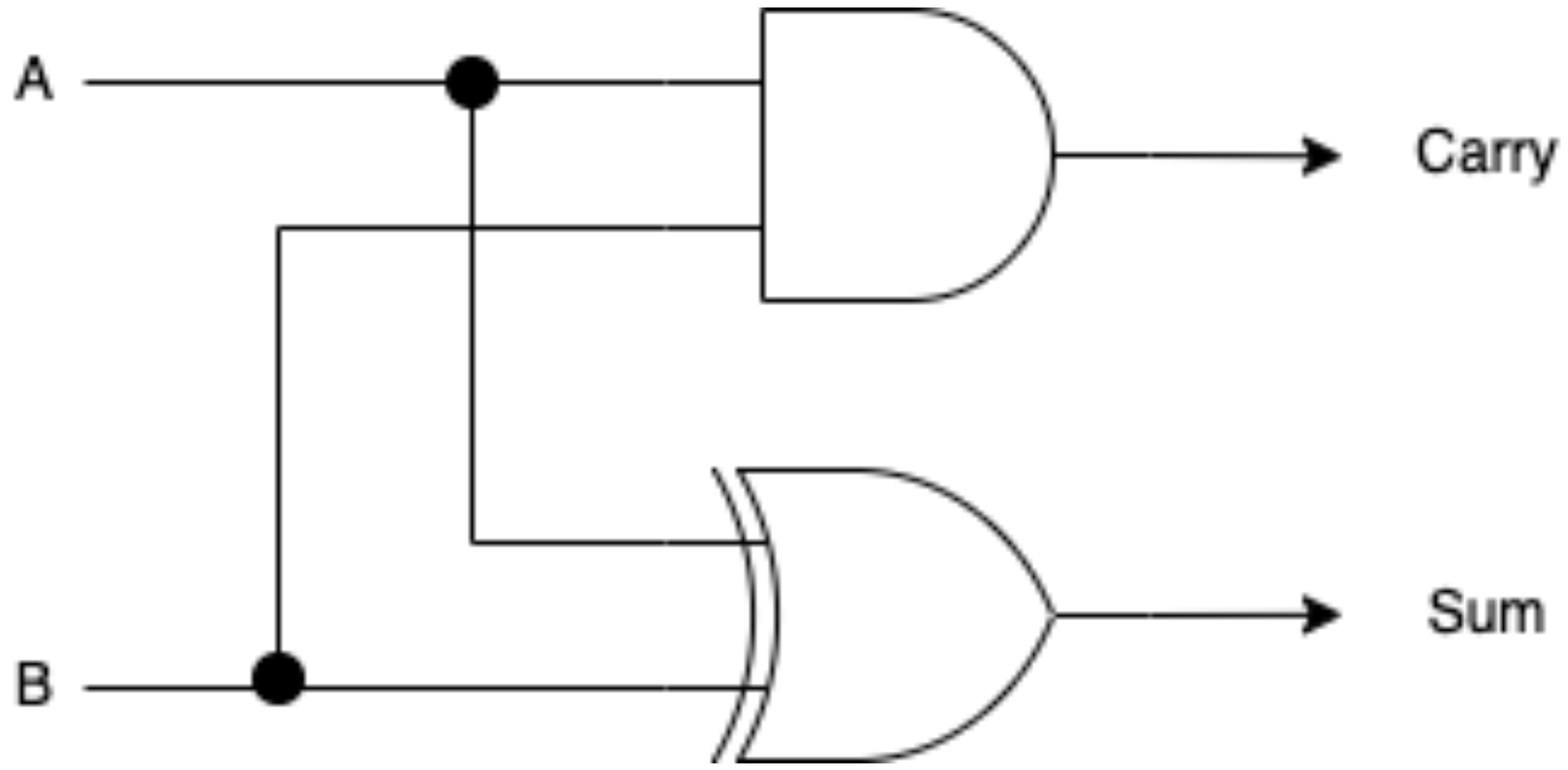
- $out = f(x, y)$
  - Select function by {zx, nx, zy, ny, f, no}

| These bits instruct how to preset the x input | | These bits instruct how to preset the y input | | This bit selects between + / And | This bit inst. how to postset out | Resulting ALU output |
|---|---|---|---|---|---|---|
| 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 |

# 2.3 Implementation

## Half-Adder

- Carry: And
- Sum: Xor




And Xor

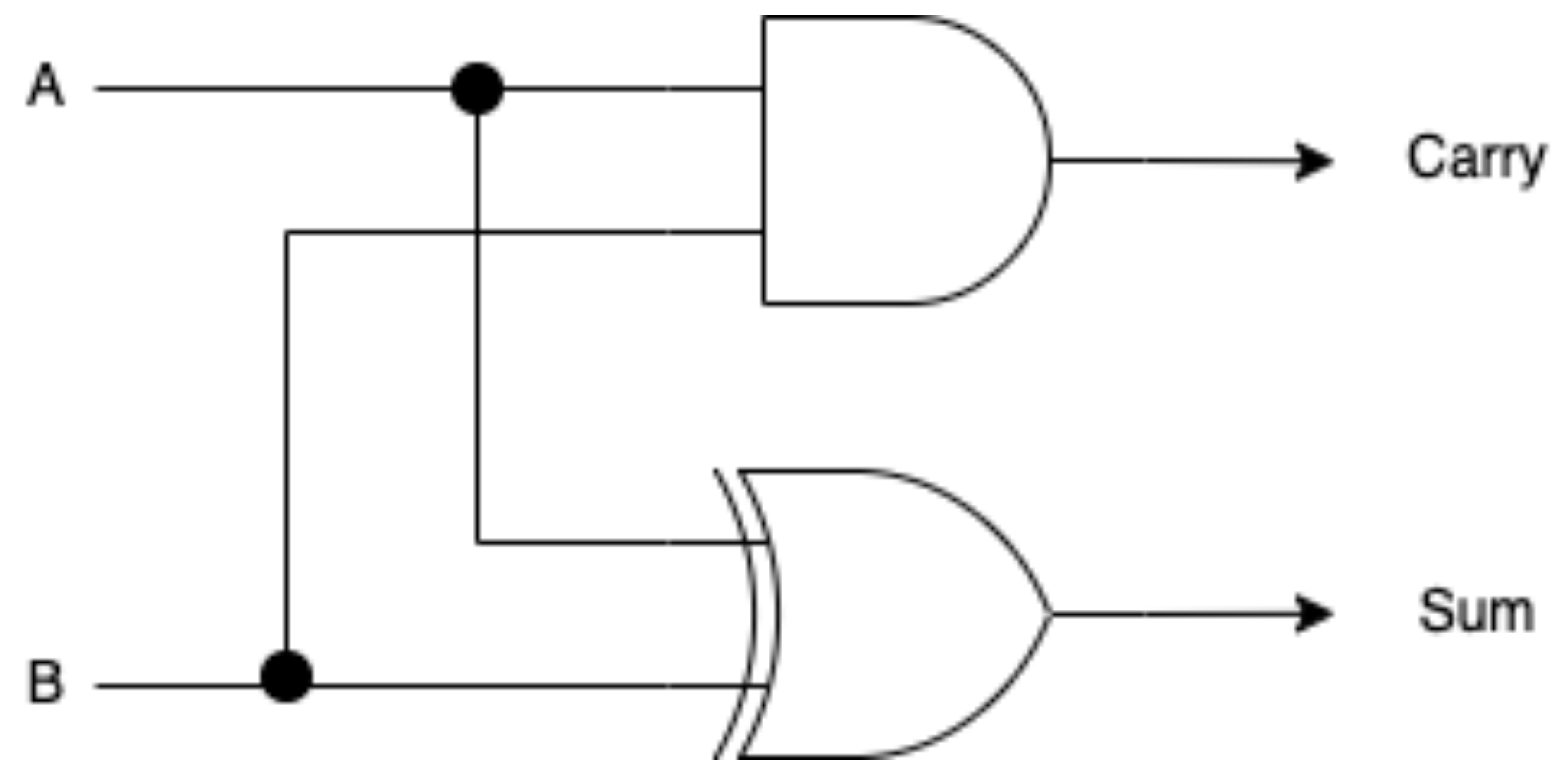
| Inputs | | Outputs | |
|---|---|---|---|
| a | b | carry | sum |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |


```
Chip name: HalfAdder
Inputs:    a, b
Outputs:   sum, carry
Function:  sum   = LSB of a + b
           carry = MSB of a + b
```

# 2.3 Implementation

## Half-Adder

```
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);
}
```

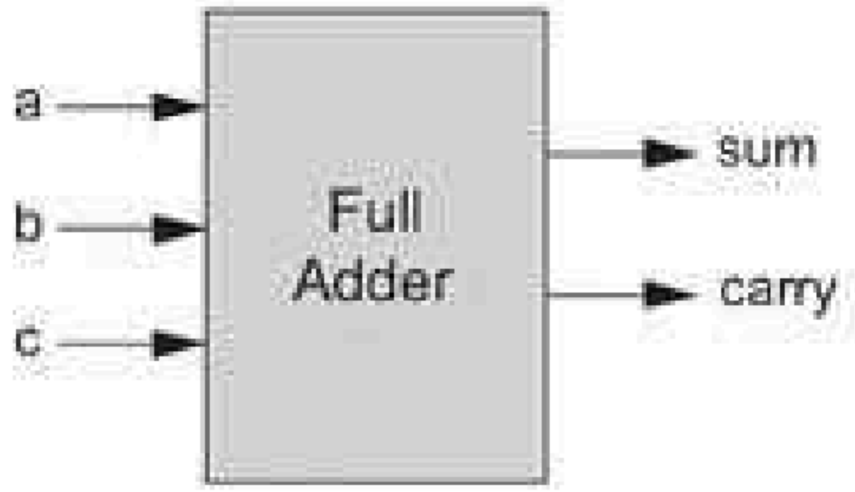


# 2.3 Implementation

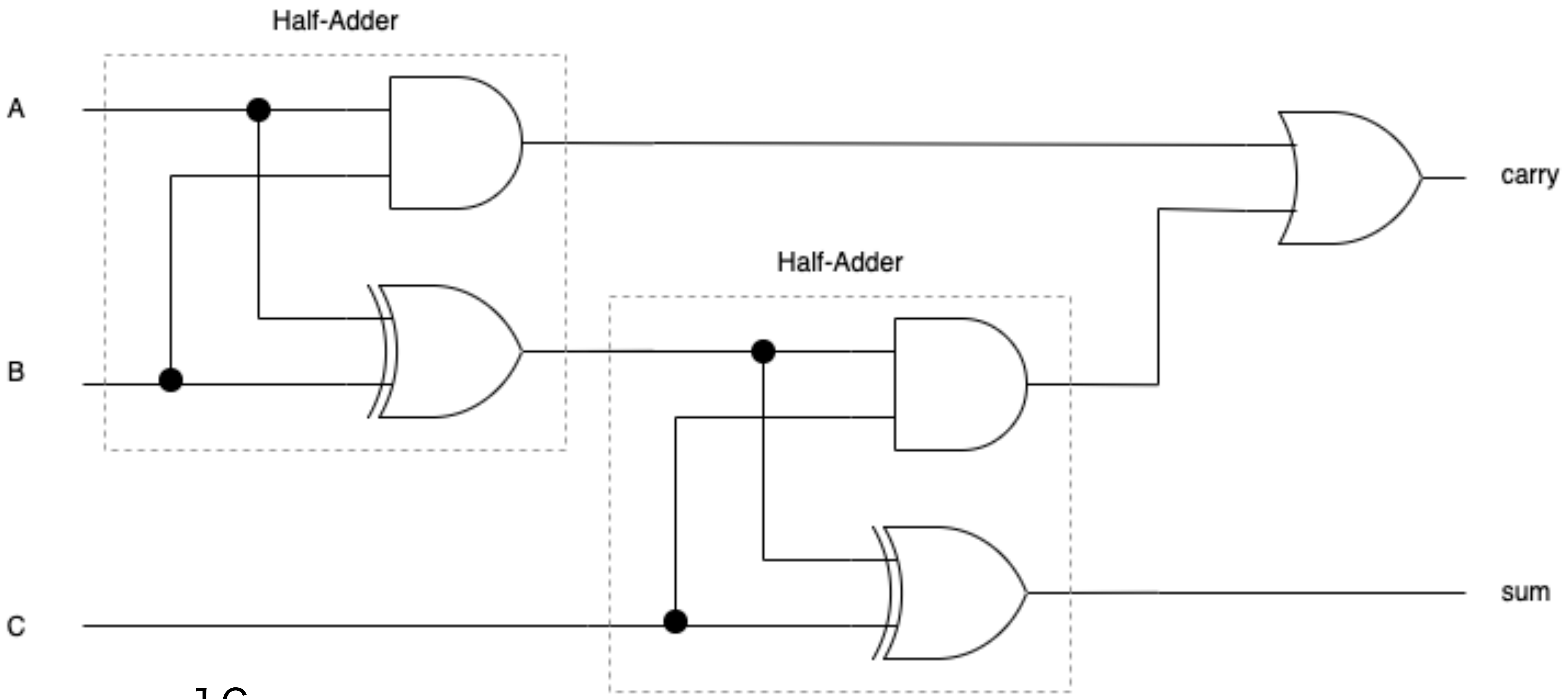
## Full-Adder

- Output sum = sum(
    sum(a,b),
    c
  )
- Output carry = Or(
    carry(a, b),
    carry(sum(a, b), c)
  )

| a | b | c | carry | sum |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

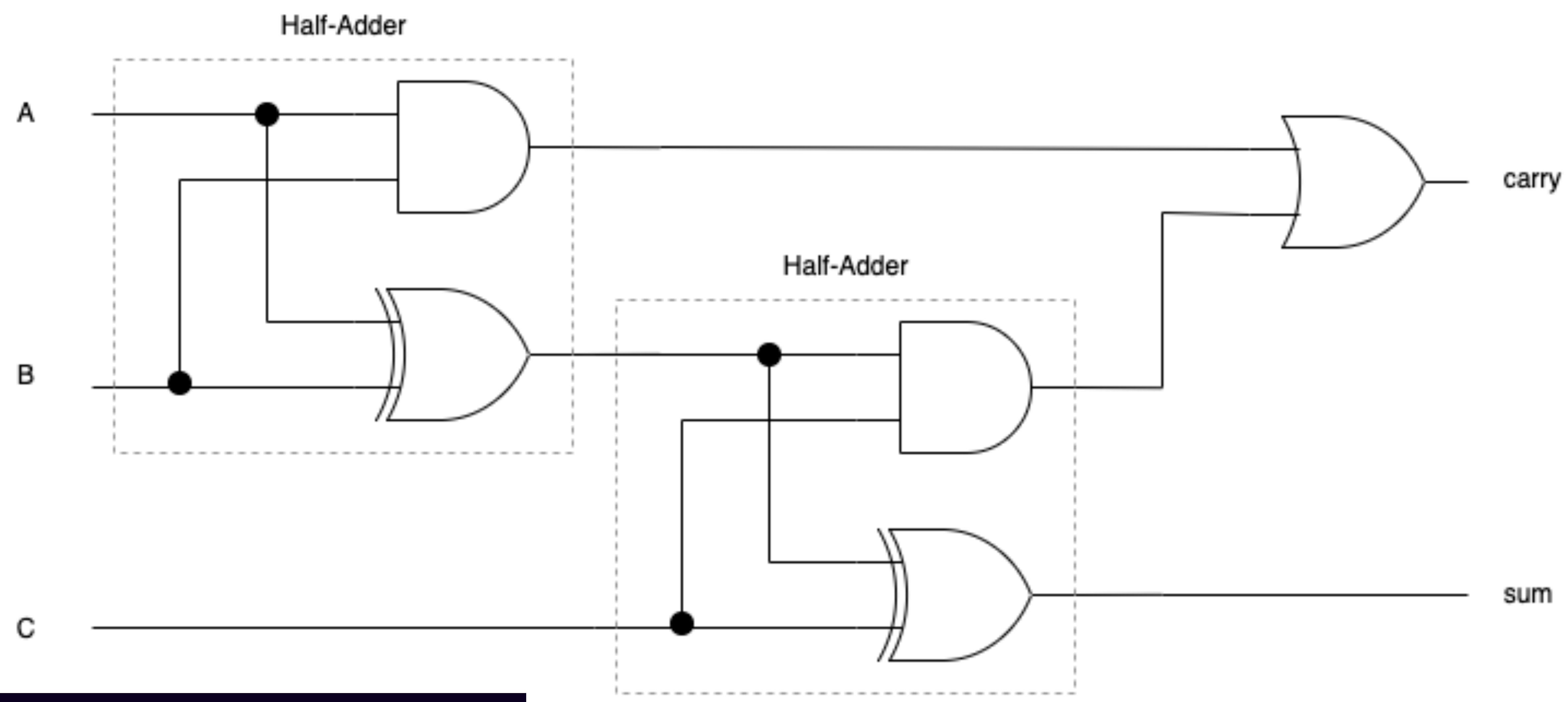



```
Chip name: FullAdder
Inputs:    a, b, c
Outputs:   sum, carry
Function:  sum = LSB of a + b + c
           carry = MSB of a + b + c
```



# 2.3 Implementation

## Full-Adder

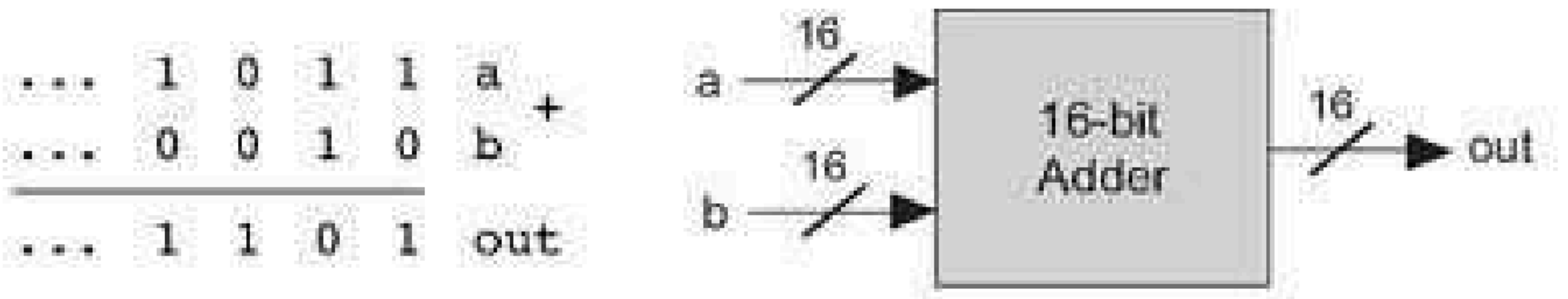



```
CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c

    PARTS:
    HalfAdder(a=a, b=b, sum=absum, carry=abcarry);
    HalfAdder(a=absum, b=c, sum=sum, carry=ccarry);
    Or(a=abcarry, b=ccarry, out=carry);
}
```

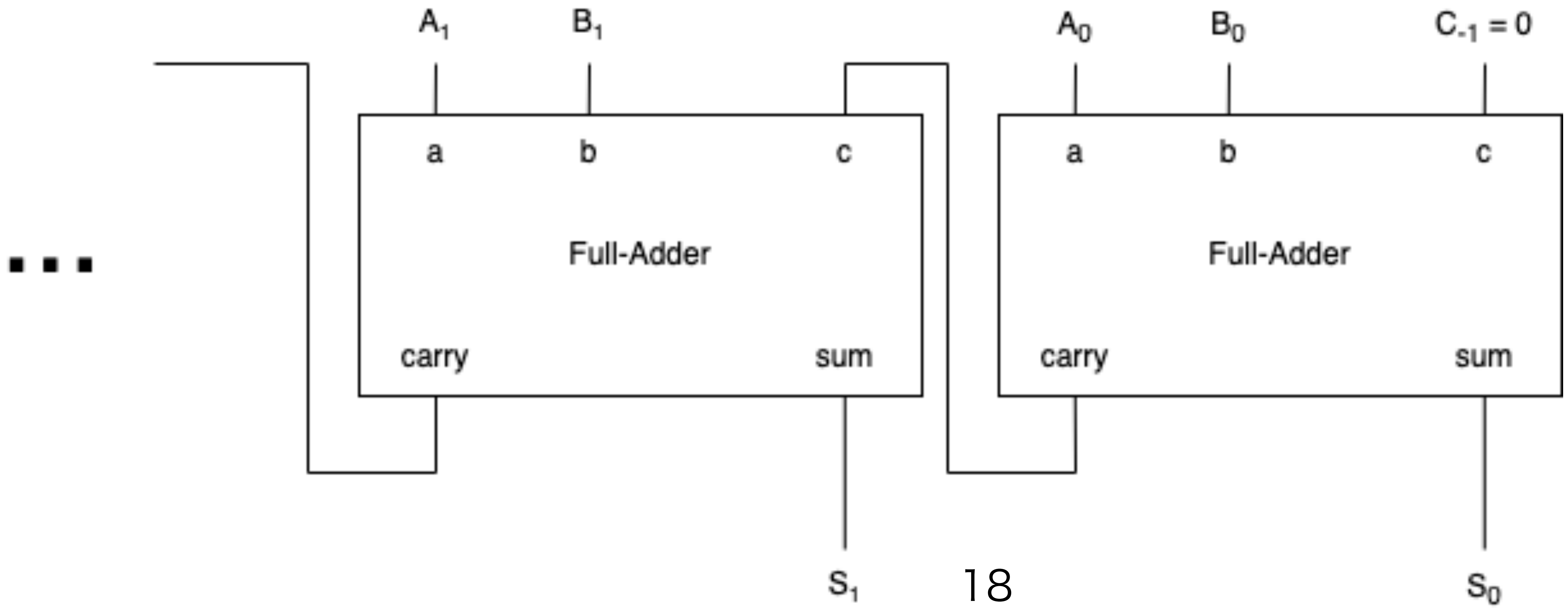
# 2.3 Implementation

## 16bit adder

- 16x Full-adder



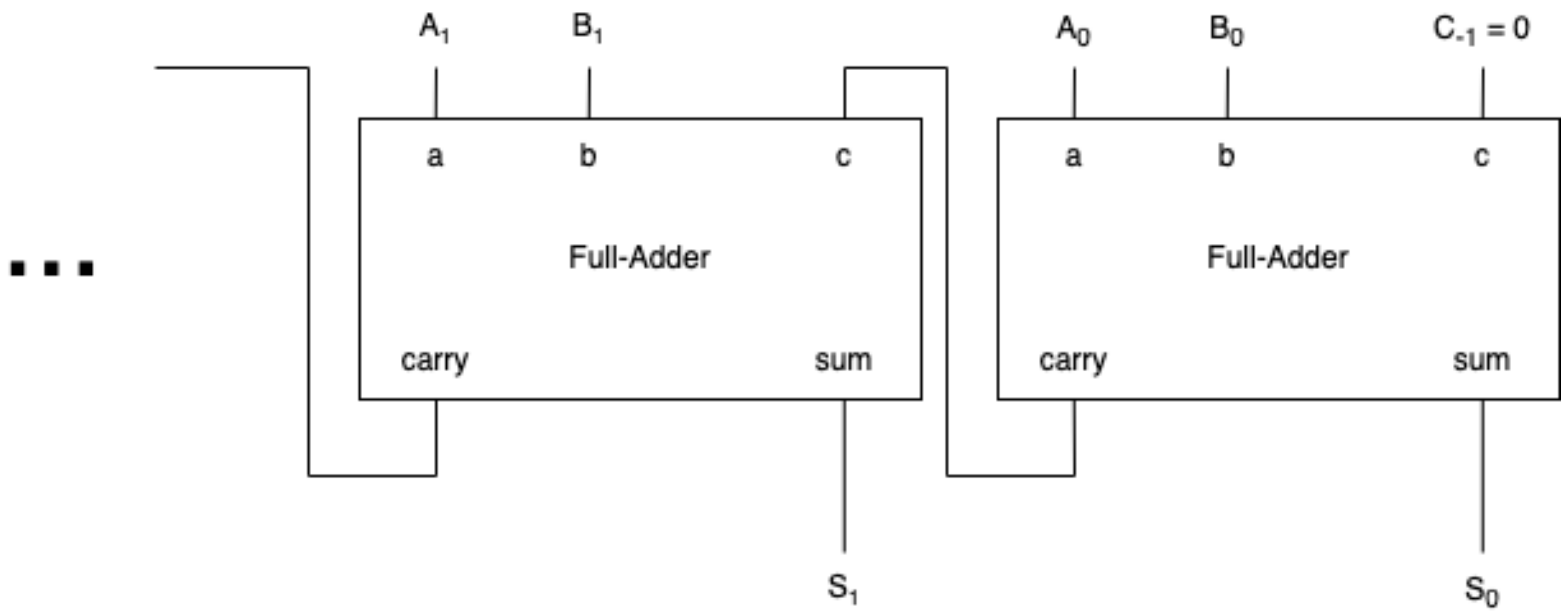

```
Chip name: Add16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  out = a + b
Comment:   Integer 2's complement addition.
           Overflow is neither detected nor handled.
```



# 2.3 Implementation

## 16bit adder

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

    PARTS:
    // Simple Add16
    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);
}
```

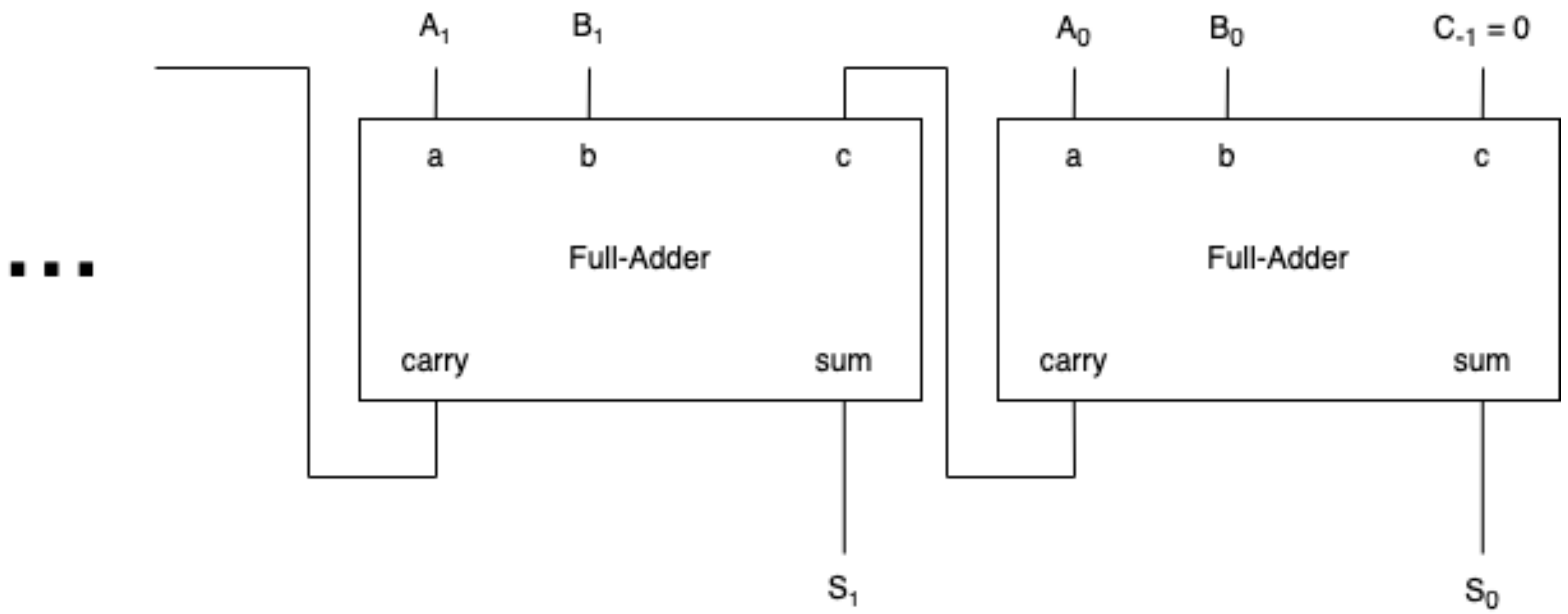


# 2.3 Implementation

## 16bit adder

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

    PARTS:
    // Simple Add16
    FullAdder(a=a[0], b=b[0], c=fal
    FullAdder(a=a[1], b=b[1], c=c1,
    FullAdder(a=a[2], b=b[2], c=c2,
    FullAdder(a=a[3], b=b[3], c=c3,
    FullAdder(a=a[4], b=b[4], c=c4,
    FullAdder(a=a[5], b=b[5], c=c5,
    FullAdder(a=a[6], b=b[6], c=c6,
    FullAdder(a=a[7], b=b[7], c=c7,
    FullAdder(a=a[8], b=b[8], c=c8,
    FullAdder(a=a[9], b=b[9], c=c9,
    FullAdder(a=a[10], b=b[10], c=c
    FullAdder(a=a[11], b=b[11], c=c
    FullAdder(a=a[12], b=b[12], c=c
    FullAdder(a=a[13], b=b[13], c=c
    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);
}
```

O(n), Too Slow!



# 2.3 Implementation

## 16bit adder with 4bit Carry Look-ahead Adder

```
CHIP Add4CLA {
    // Carry Look-ahead Adder
    IN a[4], b[4], c;
    OUT sum[4], carry;

    PARTS:
    // g, q
    And(a=a[0], b=b[0], out=g0); Xor(a=a[0], b=b[0], out=q0);
    And(a=a[1], b=b[1], out=g1); Xor(a=a[1], b=b[1], out=q1);
    And(a=a[2], b=b[2], out=g2); Xor(a=a[2], b=b[2], out=q2);
    And(a=a[3], b=b[3], out=g3); Xor(a=a[3], b=b[3], out=q3);

    And(a=q0, b=c, out=t0); Or(a=g0, b=t0, out=c0);  // c0
    And(a=q1, b=c0, out=t1); Or(a=g1, b=t1, out=c1); // c1
    And(a=q2, b=c1, out=t2); Or(a=g2, b=t2, out=c2);  // c2
    And(a=q3, b=c2, out=t3); Or(a=g3, b=t3, out=carry);  // carry(c3)

    // sum
    Xor(a=c, b=q0, out=sum[0]); Xor(a=c0, b=q1, out=sum[1]);
    Xor(a=c1, b=q2, out=sum[2]); Xor(a=c2, b=q3, out=sum[3]);        You
}
```

http://ifdl.jp/akita/class_old/old/07/lsi2/02.html

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

    PARTS:
    Add4CLA(a=a[0..3], b=b[0..3], c=false, sum=out[0..3], carry=c4);
    Add4CLA(a=a[4..7], b=b[4..7], c=c4, sum=out[4..7], carry=c8);
    Add4CLA(a=a[8..11], b=b[8..11], c=c8, sum=out[8..11], carry=c12);
    Add4CLA(a=a[12..15], b=b[12..15], c=c12, sum=out[12..15], carry=c16);
}
```

# 2.3 Implementation

## 16bit Incrementer

- Just add 1

```
Chip name: Inc16
Inputs:    in[16]
Outputs:   out[16]
Function:  out=in+1
Comment:   Integer 2's complement addition.
           Overflow is neither detected nor handled.
```

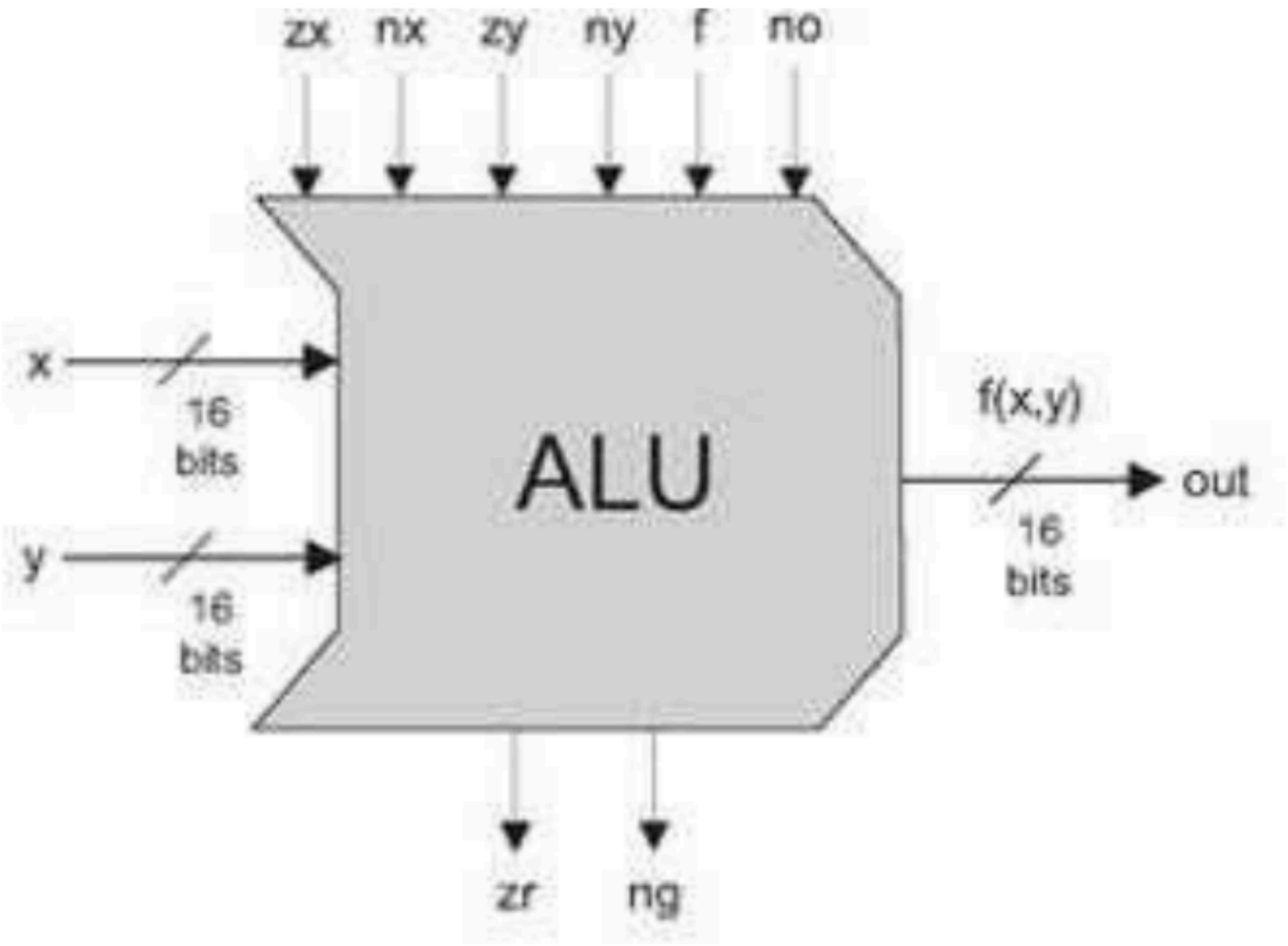
```
CHIP Inc16 {
    IN in[16];
    OUT out[16];

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

# 2.3 Implementation

## ALU

- Just implements…

```
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 (if 1) or x & y (if 0)
        no; // negate the out output?

    OUT
        out[16], // 16-bit output
        zr, // 1 if (out == 0), 0 otherwise
        ng; // 1 if (out < 0),  0 otherwise

    PARTS:
    // zx
    Mux16(a=x, b[0..15]=false, sel=zx, out=zxout);
    // zy
    Mux16(a=y, b[0..15]=false, sel=zy, out=zyout);

    // nx
    Not16(in=zxout, out=notzxout);
    Mux16(a=zxout, b=notzxout, sel=nx, out=nxout);
    // ny
    Not16(in=zyout, out=notzyout);
    Mux16(a=zyout, b=notzyout, sel=ny, out=nyout);

    // f
    Add16(a=nxout, b=nyout, out=fadd);
    And16(a=nxout, b=nyout, out=fand);
    Mux16(a=fand, b=fadd, sel=f, out=fout);

    // no
    Not16(in=fout, out=foutneg);
    Mux16(a=fout, b=foutneg, sel=no, out=out, out[0..7]=outl, out[8..15]=outm, out[15]=ng);

    // zr
    Or8Way(in=outl, out=orl);
    Or8Way(in=outm, out=orm);
    Or(a=orl, b=orm, out=zrsel);
    Not(in=zrsel, out=zr);
}
```




```
Chip name: ALU
Inputs:    x[16], y[16],    // Two 16-bit data inputs
           zx,              // Zero the x input
           nx,              // Negate the x input
           zy,              // Zero the y input
           ny,              // Negate the y input
           f,               // Function code: 1 for Add, 0 for And
           no               // Negate the out output
Outputs:   out[16],         // 16-bit output
           zr,              // True iff out=0
           ng               // True iff out<0
Function:  if zx then x = 0        // 16-bit zero constant
           if nx then x = !x       // Bit-wise negation
           if zy then y = 0        // 16-bit zero constant
           if ny then y = !y       // Bit-wise negation
           if f then out = x + y   // Integer 2's complement addition
                else out = x & y   // Bit-wise And
           if no then out = !out   // Bit-wise negation
           if out=0 then zr = 1 else zr = 0  // 16-bit eq. comparison
           if out<0 then ng = 1 else ng = 0  // 16-bit neg. comparison
Comment:   Overflow is neither detected nor handled.
```