Unit 5. Arithmetic and logic units

Digital Electronic Circuits
E.T.S.I. Informática
Universidad de Sevilla

Jorge Juan-Chico <jjchico@dte.us.es> 2010-2020

Contents

- Introduction
- Binary arithmetic
- Basic adder circuits
- Magnitude adder
- Signed binary numbers
- Signed adder: overflow
- Adder/subtractor
- ALU

Bibliography

- Recommended readings
  - LaMeres, 2.3 y 2.4
    - Binary arithmetic and signed numbers.
  - LaMeres, 12.1 and 12.2
    - Adders and adder/subtractors.
    - Verilog examples (more detailed than in this unit).
- Reference bibliograpy
  - verilog-course.v, unit 5
    - Arithmetic circuits design examples in Verilog.

Bibliography

- To know more
  - Two's complement representation notes (in Spanish)
    - Two's complement representation properties with demonstrations.
  - LaMeres, 12.3 (multiplication) and 12.4 (division)
    - Multiplication and division is not part of the unit, but they are introduced here in a simple way.
  - Floyd, 2.4 to 2.6
    - Binary arithmetic and signed numbers.
    - Includes multiplication, division and floating point representation (not included in this unit).
  - Floyd, 6.1 to 6.3
    - Adder design
    - Practical examples with MSI devices (not included in this unit).

Recommended extra exercises

- Exercises from the course's collection 4 (in Spanish)
  - Binary (and other bases) arithmetic: 1, 6
  - Signed number arithmetic: 3, 4, 5
  - AU and ALU design: 8, 10, 12, 14
  - ALU expansion: 9
  - AU analysis: 11

Introduction

- Arithmetic circuits perform arithmetic operations on n-bits data:
  - addition (+), subtraction (-), product (*), division (/), etc.
- Arithmetic operations are the most important operations in computers (digital systems)
- Implementation of arithmetic (and other mathematical) operations:
  - Hardware: using dedicated circuits
  - Software: by programming, using basic operations in hardware

Implementation	Performance	Type of operation
Hardware	Very fast (compared to software)	Fixed set of operation, with fixed data format
Software	Very slow (compared to hardware)	Unlimited set of operation with any data format

Arithmetic hardware support in personal computers

- 1970-1980 (8 bit processors)
  - Only addition and subtraction of integer numbers
- 1980-1990 (16 bit processors)
  - Multipliers and dividers
  - Optional math co-processors: real numbers, complex functions, etc
- 1990-2000 (procesadores de 32 bits)
  - Co-procesadores integrados en la CPU
  - Múltiples unidades de enteros: varios cálculos a la vez
  - Operaciones de soporte multimedia
  - Operaciones para gráficos 2-D (en controladores gráficos)
- 2000- (procesadores de 64 bits)
  - Operaciones matemáticas avanzadas
    - Procesamiento digital, simulación física, etc.
  - Operaciones para gráficos 3D (en controladores gráficos)

Arithmetic hardware support in personal computers

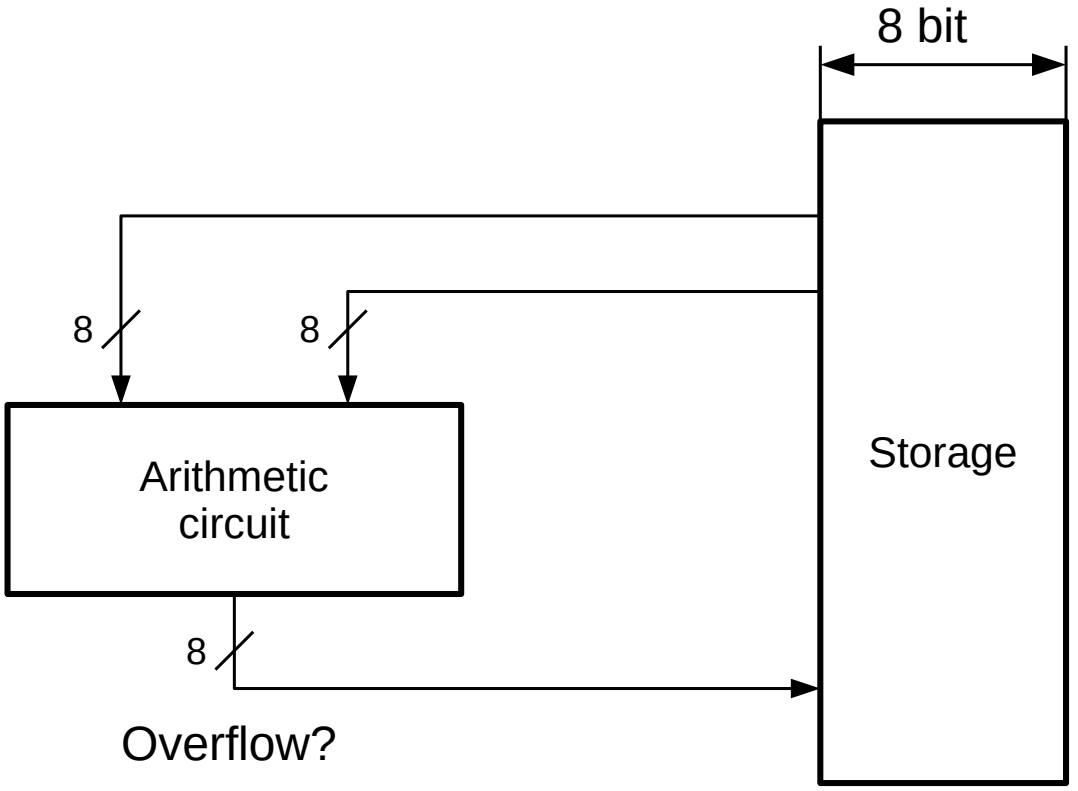
Date (data size)	Support	Applications
1970-1980 (8 bit)	• Addition and subtraction of integer numbers	• Complex operations done in software (slow)
1980-1990 (16 bit)	• Multipliers and dividers • Math co-processors (external chip)	• Real (floating point) arithmetic • Scientific calculations
1990-2000 (32 bit)	• Integrated co-processors • Multiple integer units • Multimedia support • 2-D graphics acceleration (in graphics controller)	• Faster calculations • Parallel calculations • Audio/video acceleration • Graphical User Interface
2000-now (64 bit)	• SIMD* instructions • 3-D graphics acceleration (CPU and GPU) • Codec support	• Faster graphics and multimedia processing • Video games • Video encoding/decoding

Contents

- Introduction
- **Binary arithmetic**
- Basic adder circuits
- Magnitude adder
- Signed binary numbers
- Signed adder: overflow
- Adder/subtractor
- ALU

Binary arithmetic

- Arithmetic in digital systems (computers)
- Base 2 numerical system
- Fixed number of bits



Binary arithmetic

- Consist in operating numbers using binary (base 2) representation.
- Follows the same rules than decimal (base 10) arithmetic.

Example 1

Do the following operations on numbers A and B using binary arithmetic.

a) A + B b) A - B

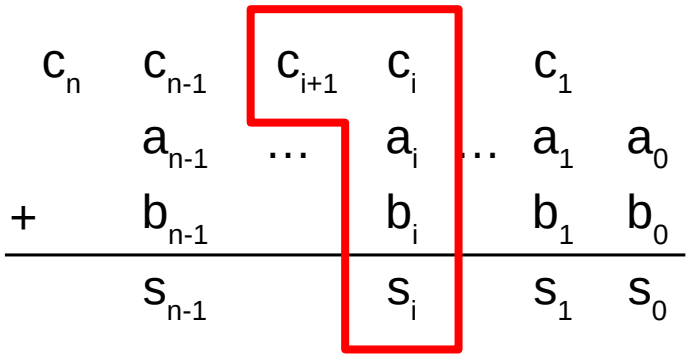
c) A * B d) A / B

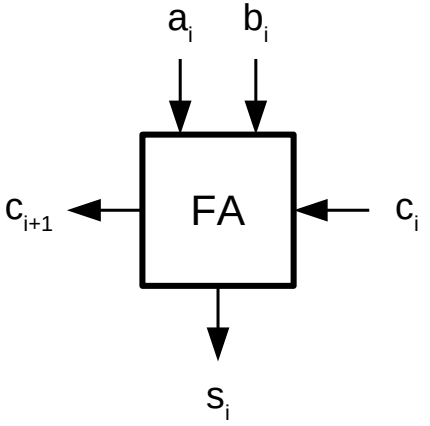
A = 100110, B = 1101

Contents

- Introduction
- Binary arithmetic
- **Basic adder circuits**
- Magnitude adder
- Signed binary numbers
- Signed adder: overflow
- Adder/subtractor
- ALU

Basic adder circuits. Full adder





Full Adder

a_i	b_i	c_i	c_{i+1}	s_i
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

$$s_i = a_i \oplus b_i \oplus c_i$$

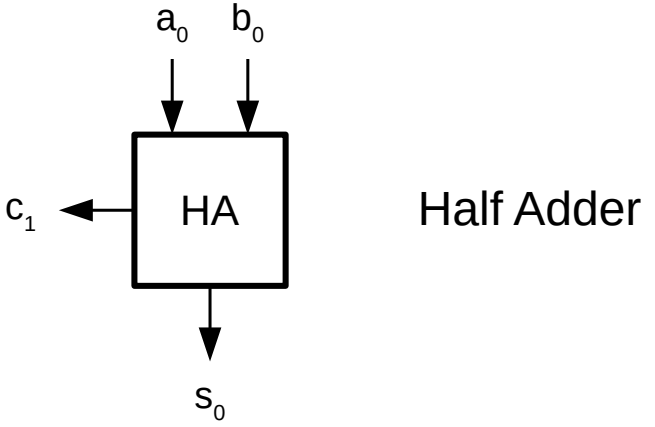
$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

$$c_{i+1} = a_i b_i + (a_i \oplus b_i) c_i$$

Full adder circuit

Basic adder circuits. Half adder

$$
\begin{array}{cccccccc}
 & c_n & c_{n-1} & & c_{i+1} & c_i & & c_1 & \\
 & & a_{n-1} & \cdots & & a_i & \cdots & a_1 & a_0 \\
+ & & b_{n-1} & & & b_i & & b_1 & b_0 \\
\hline
 & & s_{n-1} & & & s_i & & s_1 & s_0
\end{array}
$$



Half Adder

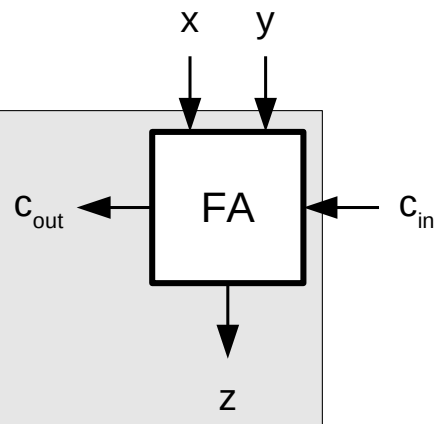
a_0	b_0	c_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$s_0 = a_0 \oplus b_0$$
$$c_1 = a_0 b_0$$

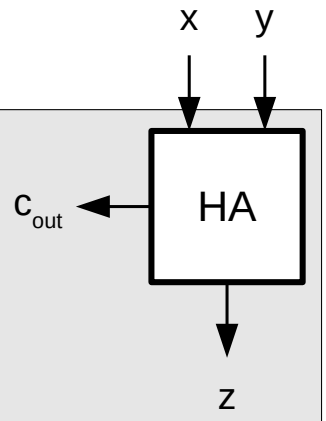
Half adder circuit

FA and HA. Verilog descriptions

```
module fa(
    input x,
    input y
    input cin,
    output z,
    output cout
    );

    assign z = x ^ y ^ cin;
    assign cout = x&y | x&cin | y&cin;
endmodule // fa
```




```
module ha(
    input x,
    input y
    output z,
    output cout
    );

    assign z = x ^ y;
    assign cout = x&y;
endmodule // ha
```

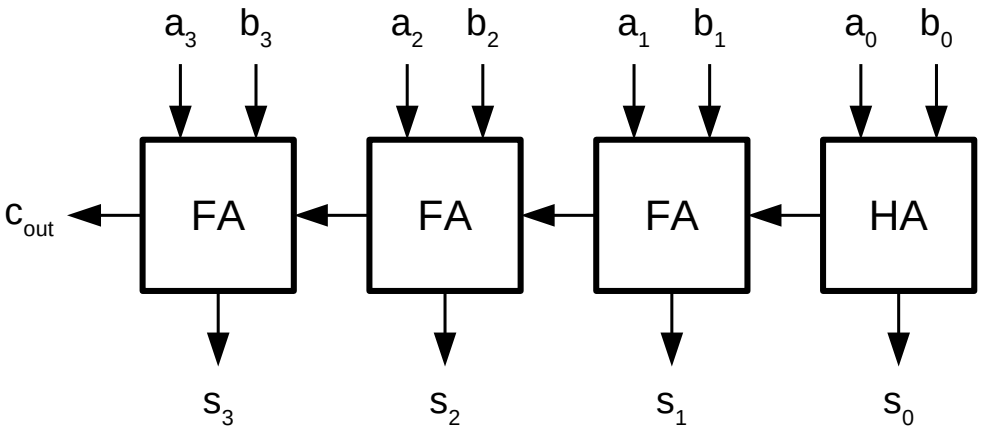


Contents

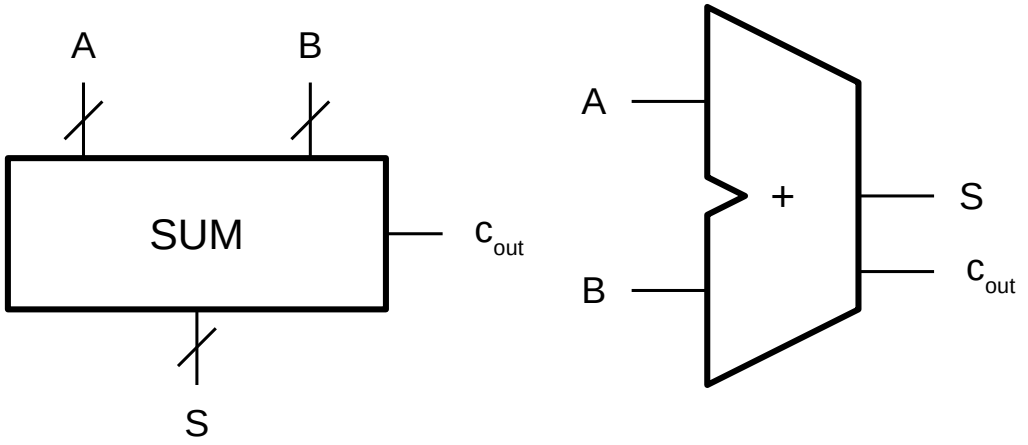
- Introduction
- Binary arithmetic
- Basic adder circuits
- **Magnitude adder**
- Signed binary numbers
- Signed adder: overflow
- Adder/subtractor
- ALU

n-bit magnitude adder

	c_{out}	c_3	c_2	c_1	
		a_3	a_2	a_1	a_0
+		b_3	b_2	b_1	b_0
		s_3	s_2	s_1	s_0



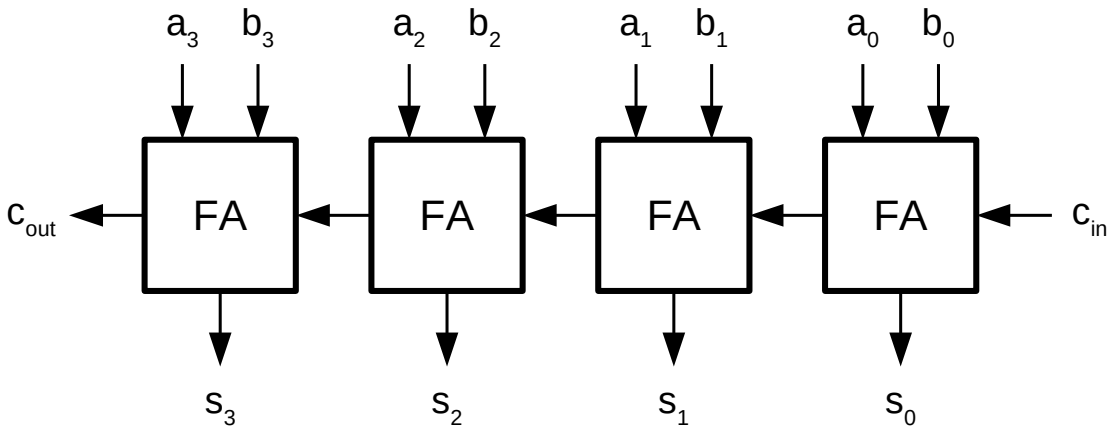
$S = (A + B) \bmod 2^n$



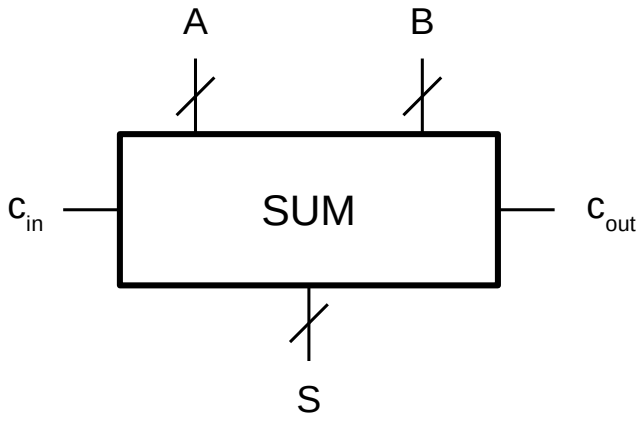
c_{out} is an “overflow” indicator: the result cannot be represented with the available output bits (n).

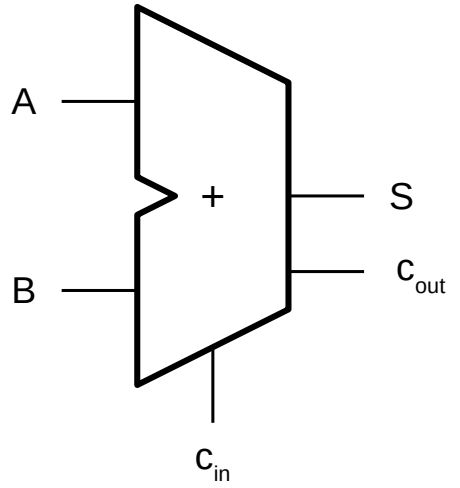
n-bit magnitude adder with carry input

	c_{out}	c_3	c_2	c_1	c_{in}
		a_3	a_2	a_1	a_0
+		b_3	b_2	b_1	b_0
		s_3	s_2	s_1	s_0



$$S = (A + B + C_{in}) \bmod 2^n$$

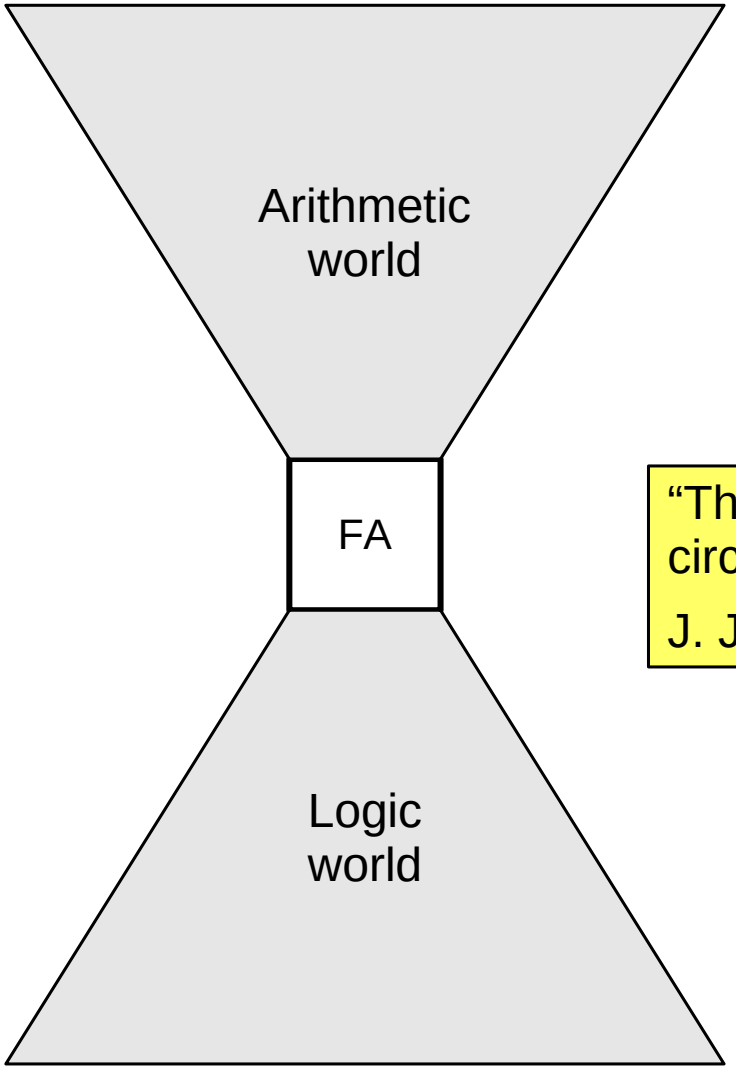




c_{in} is useful to connect various adders together to sum bigger numbers (more than n bits).

4-bit magnitude adder circuit

Full adder relevance



- The full adder does the basic arithmetic operation (addition of three bits) by using only logic operations.

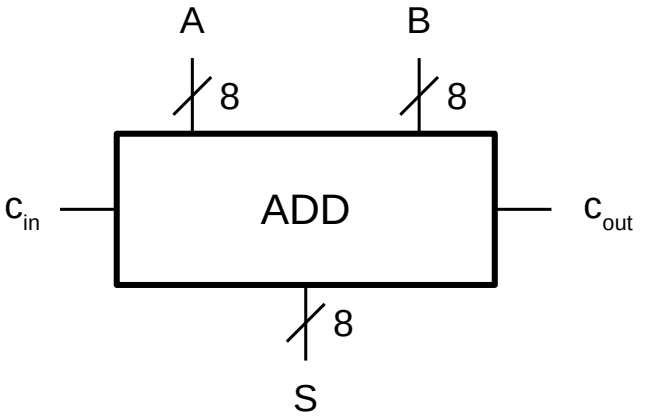
"The full adder is where the logic world of digital circuits meets the arithmetic world of computers"

J. Juan-Chico

Verilog examples

Using Full Adders

```
module adder8_fa(
    input wire [7:0] a,
    input wire [7:0] b,
    input wire cin,
    output wire [7:0] s,
    output wire cout
    );

    // auxiliary signal
    wire [7:1] c;

    fa fa0 (a[0], b[0],  cin, s[0], c[1]);
    fa fa1 (a[1], b[1], c[1], s[1], c[2]);
    fa fa2 (a[2], b[2], c[2], s[2], c[3]);
    fa fa3 (a[3], b[3], c[3], s[3], c[4]);
    fa fa4 (a[4], b[4], c[4], s[4], c[5]);
    fa fa5 (a[5], b[5], c[5], s[5], c[6]);
    fa fa6 (a[6], b[6], c[6], s[6], c[7]);
    fa fa7 (a[7], b[7], c[7], s[7], cout);

endmodule   // adder8_fa
```



Using arithmetic operators

```
module adder8(
    input [7:0] a,
    input [7:0] b,
    input cin,
    output [7:0] s,
    output cout
    );

    assign
        {cout, s} = a+b+cin;

endmodule // adder8
```

Example

Example 2

Input signal x and output signal z are 8-bits wide. Design circuits to perform the following operations considering two alternatives: 1) using magnitude adders, 2) using basic adder blocks (FA and HA).

a) z = x + 73 b) z = 2 * x c) z = 5 * x

Basic adder blocks (FA and HA) and magnitude adders are new tools in our combinational subsystems toolbox.

Contents

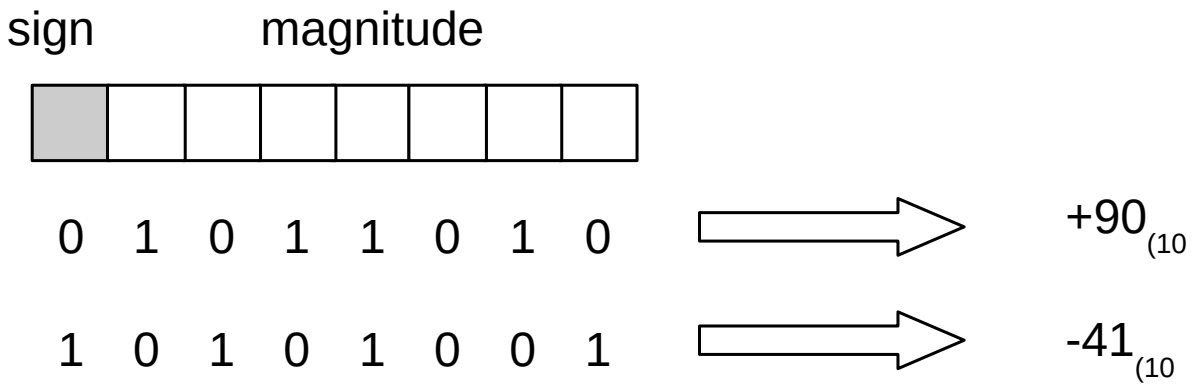
- Introduction
- Binary arithmetic
- Basic adder circuits
- Magnitude adder
- **Signed binary numbers**
- Signed adder: overflow
- Adder/subtractor
- ALU

What about negative numbers? Signed binary numbers

- In digital circuits there is no “sign”, just '0' and '1'.
- Sign must be coded with bits inside the word representing the number.
- Various alternative signed numbers representations:
  - Sign-and-magnitude
  - Excess-e representation
  - Complement representations
    - Ones' complement
    - Two's complement

Sign-and-magnitude representation

- Uses one bit for the sign and the rest for the magnitude
  - Sign: 0(+), 1(-)
  - n-bit different numbers: 2^n-1
  - Two “0” representations: 00000000, 10000000



$$-\left(2^{n-1}-1\right) \leqslant x \leqslant 2^{n-1}-1$$

Sign-and-magnitude representation with n bits

- The good
  - Easy to understand
  - Opposite easy calculation
- The bad
  - Arithmetic operations are perform differently depending on the sign of the operands.
  - Complex arithmetic circuit design.
- Usage in digital systems (computers)
  - Not used in practice for integer numbers.
  - Similar concept used in floating point representation (real numbers).

Binary	Unsigned	S-M
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	0
1001	9	-1
1010	10	-2
1011	11	-3
1100	12	-4
1101	13	-5
1110	14	-6
1111	15	-7

Excess-e representation

- Given a number x and an excess e, Excess-e representation uses the natural binary representation of x+e to represent x.
- For the representation to be possible, x+e should yield a positive integer that can be expressed in natural binary with n bits.
- With n bits, a convenient value for the excess is $e=2^{n-1}$
  - Represent about the same number of positive and negative numbers.
  - First bit of the representation identifies the sign (0 → negative, 1 → positive).
- Eg: excess-128 (n=8 → $2^{n-1}=128$)
  - $-35_{(10} \rightarrow -35+128 = 93 = 01011101_{(2}$
  - $-35_{(10} = 01011101_{exc-128}$

$$0 \leqslant x+e < 2^n$$

$$e = 2^{n-1}$$

$$0 \leqslant x+2^{n-1} < 2^n$$

$$-2^{n-1} \leqslant x < 2^{n-1}$$

Excess-e representation

- The good
  - Easy conversion from the number to its representation.
- The bad
  - Calculating the opposite requires arithmetic operations.
  - Arithmetic circuit design is moderately complex.
- Usage in digital systems (computers)
  - Used to encode the exponent in standard floating point representation.

Binary	Unsigned	Excess-8
0000	0	-8
0001	1	-7
0010	2	-6
0011	3	-5
0100	4	-4
0101	5	-3
0110	6	-2
0111	7	-1
1000	8	0
1001	9	1
1010	10	2
1011	11	3
1100	12	4
1101	13	5
1110	14	6
1111	15	7

Complement representation with n bits

- Complement representations uses a “complement” operation to represent negative numbers.
- The complement operation is built so that the most-significant bit (MSB) is 0 for positive numbers and 1 for negative numbers.
- Positive number representation
  - Represented in natural binary.
  - MSB of the result must be '0' for the number to be “representable”.
- Negative numbers
  - Represented by making the “complement” of the opposite number.
  - MSB of the result must be '1' for the number to be “representable”.
- Sign change
  - Sign is changed by applying the “complement” operation.
- Typical complement operations/representations
  - Ones' complement
  - Two's complement

Ones' complement representation (OCR)

- First bit determines the sign of the number.
- The good
  - Easy to calculate the opposite.
  - More simple arithmetic circuits.
- The bad
  - Two representation for the '0'.
- Usage in digital systems (computers)
  - Used in some early computers.

Binario	Positivo	RC1
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-7
1001	9	-6
1010	10	-5
1011	11	-4
1100	12	-3
1101	13	-2
1110	14	-1
1111	15	0

$$-\left(2^{n-1}-1\right) \leqslant x \leqslant 2^{n-1}-1$$

Ones' complement representation

- Definition (Ones' complement operation with n bits): Let x be a positive integer number, $x \leq 2^n - 1$, the ones' complement operation with n bits on x, $OC_n(x)$, is defined as the number resulting from complementing all the bit of x when expressed in base 2.
- Definition (Ones' complement representation with n bits): Let x be an integer, $-2^{n-1} < x\ 2^{n-1}$, the ones' complement representation of x with n bits, $OCR_n(x)$, is an n-bit binary word which magnitude is:
  - x, if $0 \leq x \leq 2^{n-1} - 1$
  - $OC_n(-x)$, if $-2^{n-1} - 1 \leq x < 0$
- Definition (Ones' complement representativity): Let x be a positive integer. If $x < -(2^{n-1} - 1)$ or $x > 2^{n-1} - 1$, it is said that x is not representable in ones' complement with n bits.

Two's complement representation (TCR) Intuitive idea

- Takes advantage of the way magnitude adders add:
  - $s = (a+b) \bmod 2^n$
- We can add a negative number using a positive number. Eg:
  - 5 + (-3) = 2
  - (5 + 13) mod 16 = 18 mod 16 = 2
- In general, if x<0, it is enough to use $x+2^n$
- We must set limits to tell positive from negative numbers
  - $0 \ldots 2^{n-1}-1 \rightarrow$ positive (msb=0)
  - $2^{n-1} \ldots 2^n-1 \rightarrow$ negative (msb=1)
- It works for all combinations of a and b except overflow!

Binario	Positivo	Negativo	RC2
0000	0	0	0
0001	1	-15	1
0010	2	-14	2
0011	3	-13	3
0100	4	-12	4
0101	5	-11	5
0110	6	-10	6
0111	7	-9	7
1000	8	-8	-8
1001	9	-7	-7
1010	10	-6	-6
1011	11	-5	-5
1100	12	-4	-4
1101	13	-3	-3
1110	14	-2	-2
1111	15	-1	-1

Two's complement representation
More formal definition

- **Definition (Two's complement operation with n bits)**: Let x be an integer number, $0 \leq x \leq 2^n - 1$, the two's complement operation with n bits on x, $TC_n(x)$, is defined as:
  - $TC_n(x) = 2^n - x$
- **Definition (Two's complement representation with n bits)**: Let x be an integer, $-2^{n-1} \leq x \leq 2^{n-1} - 1$. The Two's complement representation of x with n bits, $TCR_n(x)$, is an n-bit binary word which magnitude is:
  - x, if $0 \leq x \leq 2^{n-1} - 1$
  - $TC_n(-x)$, if $-2^{n-1} \leq x < 0$
- **Definition (Two's complement representativity)**: Let x be a positive integer. If $x < -2^{n-1}$ or $x > 2^{n-1} - 1$, it is said that x is not representable in two's complement with n bits.

Demonstrations (in Spanish)

Two's complement representation
Sign properties

- **Theorem (Sign bit)**: If x is an integer representable in Two's complement with n bits, the most significant bit of the representation is '0' if, and only if, $x \geq 0$.
- **Theorem (Opposite calculation)**: If x is an integer representable in Two's complement with n bits, the magnitude of the $TCR_n(-x)$ is the two's complemente of $TCR_n(x)$, provided that -x is representable in two's complement. That is:
  - $TCR_n(-x) = TC_n(TCR_n(x))$
- **Corollary**: the TCR of the opposite is calculated just by applying the two's complement operation to the TCR of the number.

Two's complement and ones' complement relation

- Theorem: the ones' complement with n bits of x can be calculated as:
  - $OC_n(x) = 2^n - x - 1$
- Corollary:
  - $OC_n(x) = TC_n(x) - 1$
  - $TC_n(x) = OC_n(x) + 1$

Rule 1 (fast TC calculation)

$TC_n(x)$ can be calculated by complementing all the bits of x and adding 1 to the result.

Rule 2 (super-fast TC calculation)

$TC_n(x)$ can be calculated by keeping the bits of x that are '0' starting with the LSB, including the first bit at '1', and complementing the rest of the bits.

Signed numbers representation examples

Example 3

Represent the following number in S-M, OC and TC notation with 8 bits.

a) 32, b) -13, c) 115, d) -140, e) 128, f) -128

Example 4

Obtain the minimum number of bits to represent the following number in S-M, OC and TC notation and represent them:

a) 32, b) -13, c) 115, d) -140, e) 128, f) -128

Example 5

Calculate the decimal value of the following words when interpreted in S-M, OC and TC representation with 8 bits.

a) 01001100, b) 11110000

Two's complement representation
How to add

- **Theorem (Sum rule)**: Let a and b be two integer number so that a, b and a+b are representable in two's complement with n bits. The two's complement representation of a+b, $TCR_n(a+b)$, can be calculated as:
  - $TCR_n(a+b) = [TCR2_n(a) + TCR_n(b)] \bmod 2^n$

  That is, the TCR with n bits of a+b can be obtained by adding the TCRs of a and b with n bits, discarding any carry bit.
- **Corolary**: A n-bit magnitude adder that takes the TCRs of a and b with n bits as inputs, will produce the correct TCR with n bits of a+b, as long as a+b is representable in two's complement with n bits.

Two's complement representation
How to detect overflow

- **Definition (Two's complement overflow)**: Let a and b be two integer number that are representable in two's complement with n bits. It is said that the sum a+b produces overflow in two's complement if a+b is not representable in two's complement.
- **Corolary (overflow rule)**: Let a and b be two integer number that are representable in two's complement with n bits. The sum a+b produces overflow if, and only if, a and b have the same sign (or at least one of them is 0) and this sign is different from the sign bit of the sum of the TCRs of a and b, modulo 2^n.

Two's complement representation
Sum and overflow

```
 1001 = -7
 0101 = +5
-----
 1110 = -2
```

```
 1100 = -4
 0100 = +4
-----
10000 =  0
```

```
 0011 = +3
 0100 = +4
-----
 0111 = +7
```

```
 1100 = -4
 1111 = -1
-----
11011 = -5
```

```
 0101 = +5
 0100 = +4
-----
 1001 = -7
```

```
 1001 = -7
 1010 = -6
-----
10011 = +3
```

Overflow!

Two's complement representation as a weighted code

- **Theorem**: Let x be an integer representable in two's complement with n bits, which TC representation is given by the binary digits $\{x_0,x_1,\ldots,x_{n-1}\}$. It is verified that:

$$X = -2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} + \ldots + 2x_1 + x_0$$

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-128	64	32	16	8	4	2	1

1	0	1	1	0	1	1	0	→	-74
1	1	1	1	1	1	1	0	→	-2
0	1	0	0	0	0	0	1	→	65

Two's complement sign extension

- **Theorem (Sign extension of the two's complement)**: Let x be an integer number representable in two's complement with n bits, $TCR_n(x)$ the magnitude of its two's complement representation with n bits and s the sign bit of the two's complement representation. It is verified that:
  - $TCR_{n+1}(x) = s\ 2^n + TCR_n(x)$

  It is to say that, in order to obtain the TCR of a number with one more bit it is enough to add one bit on the left side that is equal to the sign bit.
- **Corollary**: If x is an integer number representable in two's complement with n bits, it will be representable with n-1 bits if, and only if, the two most significant bits of the two's complement representation with n bits are equal.

Two's complement representation examples

Example 6

Represent the following number in TCR with 8 bits using a weights table:

a) 32, b) -13, c) 115, d) -140, e) 128, f) -128

Example 7

Do the following operations in binary form with numbers in two's complement representation by first extending the TC representations to 8 bits. Check the results repeating the operations in decimal.

a) 011110 + 10100, b) 0111 + 10, c) 11111010 + 100001, d) 10001 + 0111111

Signed numbers summary

x	S-M	Exc-2^{n-1}	OCR	TCR
-8	-	0000	-	1000
-7	1111	0001	1000	1001
-6	1110	0010	1001	1010
-5	1101	0011	1010	1011
-4	1100	0100	1011	1100
-3	1011	0101	1100	1101
-2	1010	0110	1101	1110
-1	1001	0111	1110	1111
0	0000/1000	1000	0000/1111	0000
1	0001	1001	0001	0001
2	0010	1010	0010	0010
3	0011	1011	0011	0011
4	0100	1100	0100	0100
5	0101	1101	0101	0101
6	0110	1110	0110	0110
7	0111	1111	0111	0111

Contents

- Introduction
- Binary arithmetic
- Basic adder circuits
- Magnitude adder
- Signed binary numbers
- **Signed adder: overflow**
- Adder/subtractor
- ALU

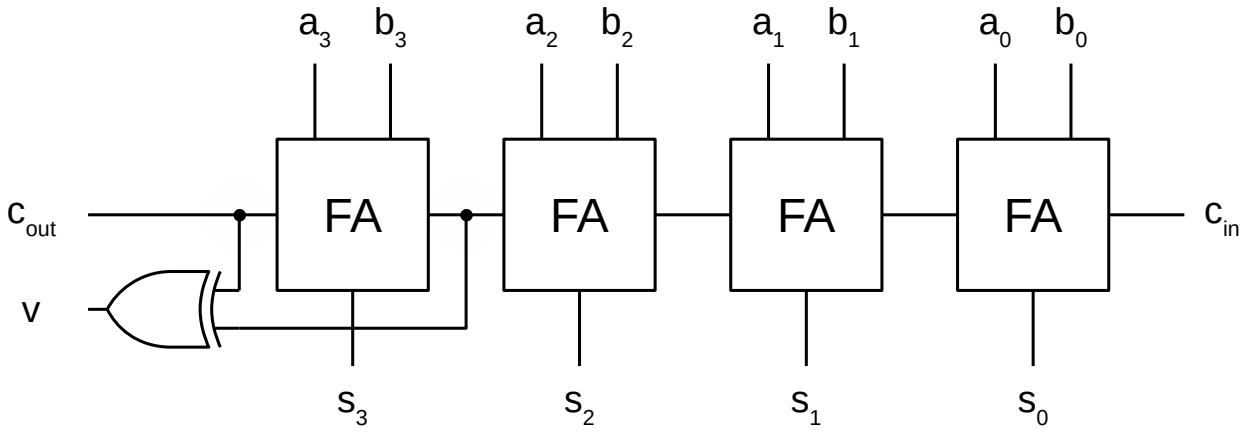
Signed adder: overflow

- Same adder used for magnitudes (TCR property).
- Carry bit is not a valid overflow indicator for TC addition.
  - We need a new “overflow” flag for TC addition.
  - Overflow rule: sign of result different to sign of operands

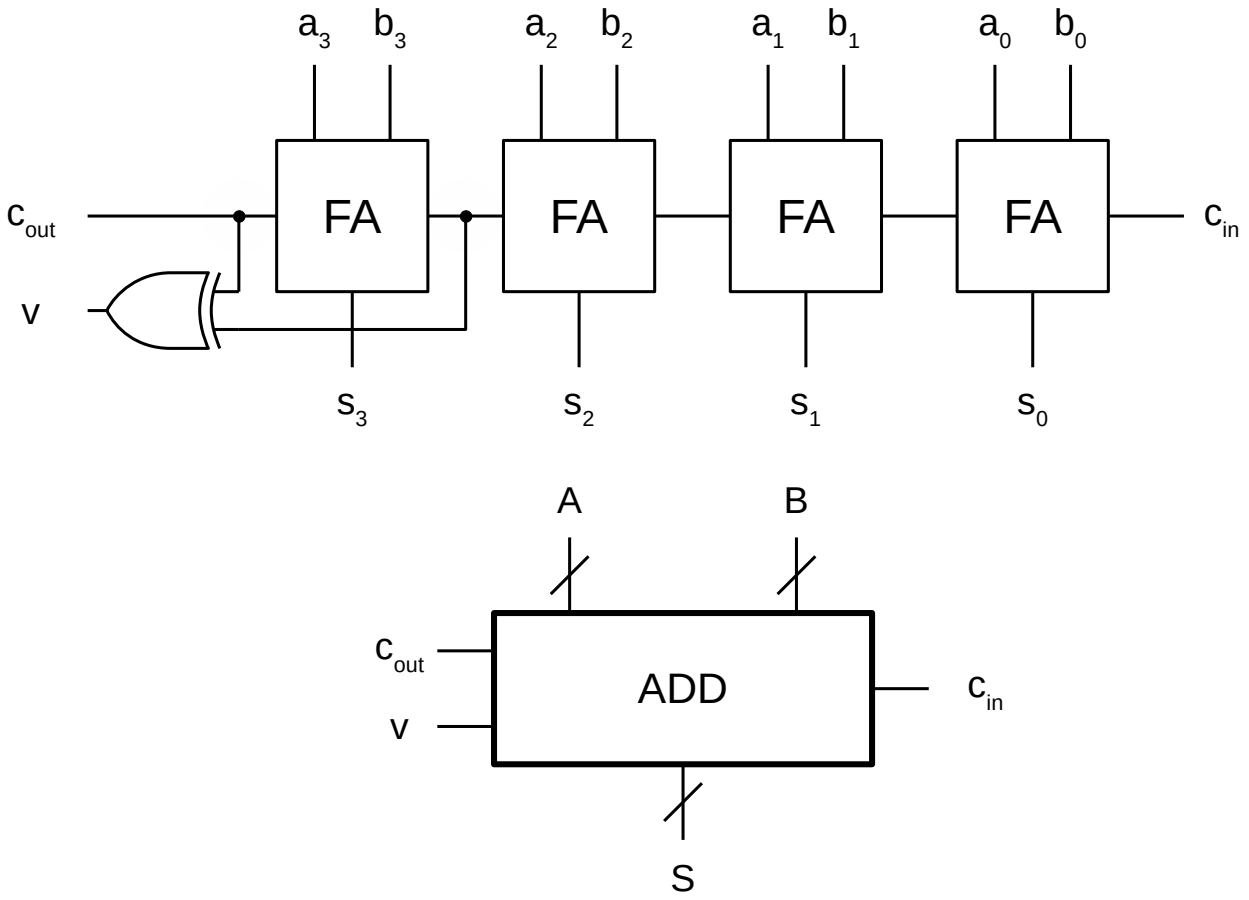
```
   0  1
      0  ...
+     0  ...
-------------
      1  ...

   1  0
      1  ...
+     1  ...
-------------
      0  ...
```

$$v = \overline{a}_{n-1}\,\overline{b}_{n-1}\,s_{n-1} + a_{n-1}\,b_{n-1}\,\overline{s}_{n-1}$$

$$v = c_n \oplus c_{n-1}$$



Signed adder: overflow



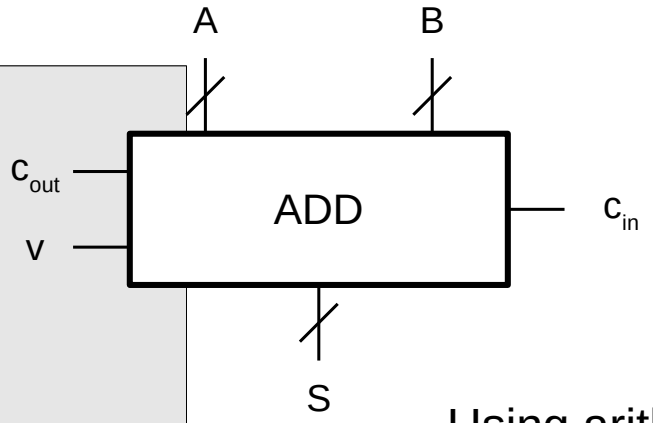
Exercise

Add an overflow output to our magnitude adder in circuitjs and check its operation.

Signed adder. Verilog examples

Using full adders

```
module adder8_fa(
    input wire [7:0] a,
    input wire [7:0] b,
    input wire cin,
    output wire [7:0] s,
    output wire cout, v
    );

    // auxiliary signal
    wire [7:1] c;

    fa fa0 (a[0], b[0],  cin, s[0], c[1]);
    fa fa1 (a[1], b[1], c[1], s[1], c[2]);
    fa fa2 (a[2], b[2], c[2], s[2], c[3]);
    fa fa3 (a[3], b[3], c[3], s[3], c[4]);
    fa fa4 (a[4], b[4], c[4], s[4], c[5]);
    fa fa5 (a[5], b[5], c[5], s[5], c[6]);
    fa fa6 (a[6], b[6], c[6], s[6], c[7]);
    fa fa7 (a[7], b[7], c[7], s[7], cout);

    assign v = c[7] ^ cout;

endmodule   // adder8_fa
```



Using arithmetic operators

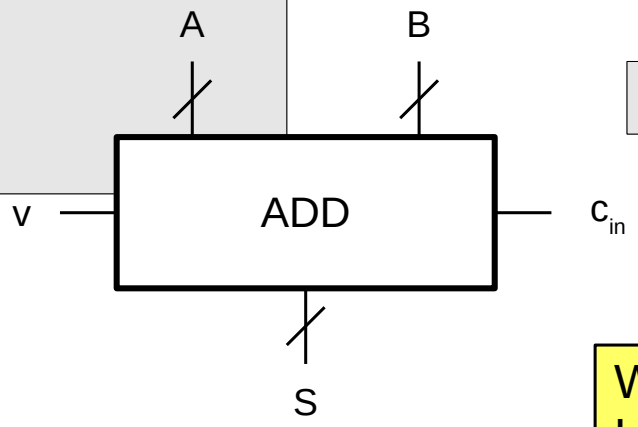
```
module adder8(
    input wire [7:0] a,
    input wire [7:0] b,
    input wire cin,
    output wire [7:0] s,
    output wire cout, v
    );

    assign {cout, s} = a+b+cin;

    assign v = ~a[7] & ~b[7] & s[7]
             | a[7] & b[7] & ~s[7];

endmodule // adder8
```

Signed adder. Verilog examples

```
module adder8 #(parameter N = 8)(
    input wire signed [N-1:0] a,
    input wire signed [N-1:0] b,
    output reg signed [N-1:0] z,
    output reg v
    );

    reg signed [N:0] f;

    always @* begin
        f = a + b;

        if (f[N] != f[N-1])
            v = 1;
        else
            v = 0;

        z = f[N-1:0];
    end
endmodule // adder8
```

- Type “signed” handles automatically TC representation:
  - Negative constants
  - Sign extension
  - ...

```
if (f[N] != f[N-1])
    v = 1;
else
    ov = 0;
```




```
v = (f[N]==f[N-1])? 0: 1;
```

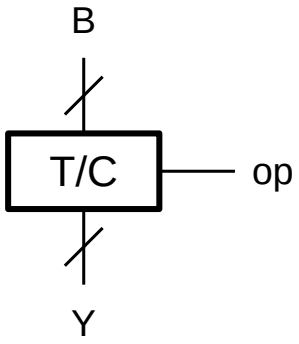
```
v = f[N] ^ f[N-1]);
```

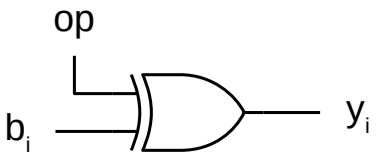
Why there is not a Cout output? How can we add one?

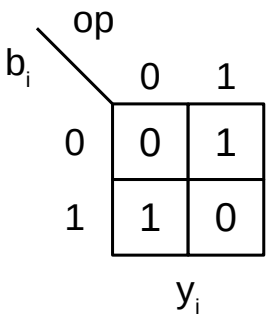
Contents

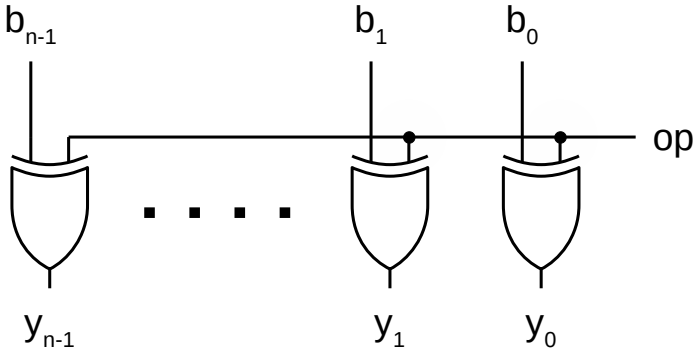
- Introduction
- Binary arithmetic
- Basic adder circuits
- Magnitude adder
- Signed binary numbers
- Signed adder: overflow
- **Adder/subtractor**
- ALU

Adder/subtractor
Transfer/complement block









$$Y = \overline{B} = OC_n(B) = 2^n\text{-}B\text{-}1 = TC_n(B)\text{-}1$$

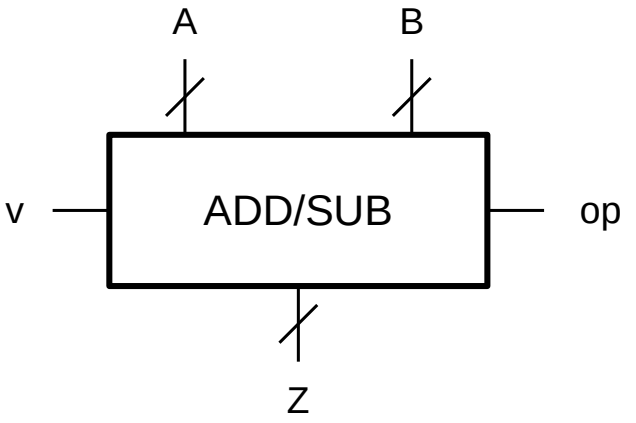
If $B=TCR_n(b)$, hence:

$$Y+1 = TC_n(B) = TCR_n(-b)$$

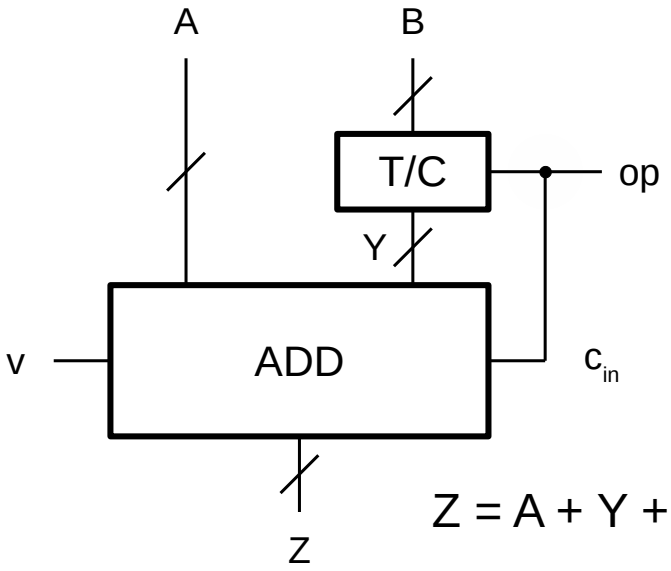
Notation: $X = TCR_n(x)$

Adder/subtractor

- Two's complement adder/subtractor
  - $A=TCR_n(a)$, $B=TCR_n(b)$, $Z=TCR_n(z)$
  - v: overflow output (z is not representable in two's complement)



op	z	Z
0	a + b	A + B
1	a - b	A+TC(B)



$Z = A + Y + cin$

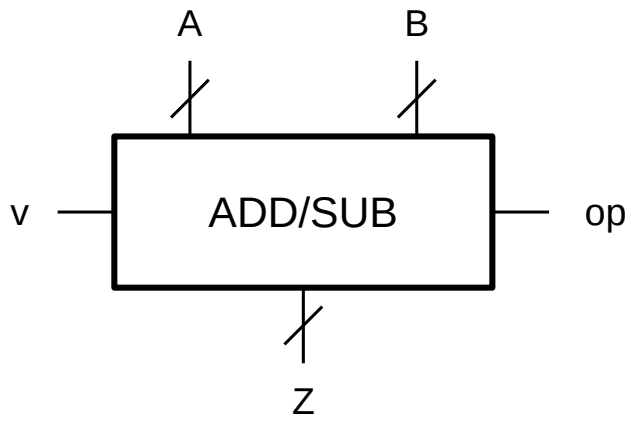
op	Y	c_{in}	Z	z
0	B	0	A + B	a + b
1	$\overline{B}$	1	A+$\overline{B}$+1	a - b

$TC_n(B) = OC_n(B)+1 = B+1$

Exercise

Design a 4-bit adder/subtractor in circuitjs and check its operation (use our magnitude adder circuit as starting point).

Adder/subtractor
Verilog description



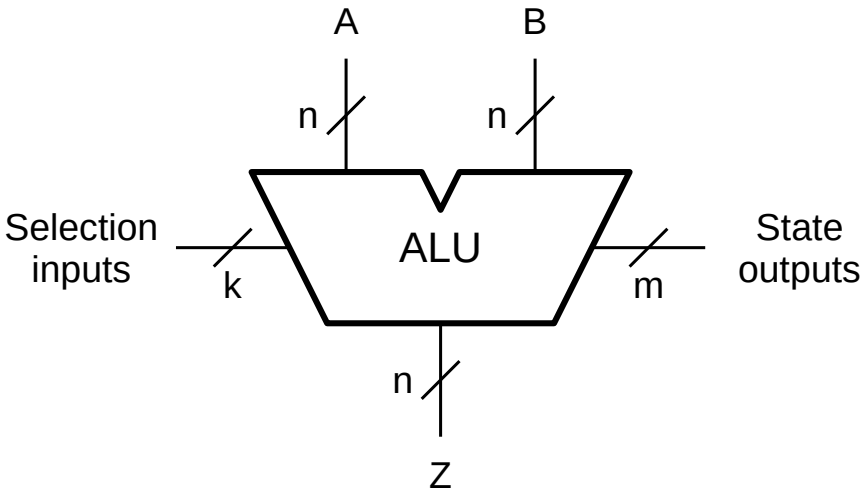

```
module addsub #(parameter N = 8)(
    input wire signed [N-1:0] a,
    input wire signed [N-1:0] b,
    input wire op,
    output reg signed [N-1:0] z,
    output reg v
    );

    reg signed [N:0] f;

    always @* begin
        case (op)
          0:
            f = a + b;
          default:
            f = a - b;
        endcase

        v = f[N] ^ f[N-1]);

        z = f[N-1:0];
    end
endmodule // addsub
```

Contents

- Introduction
- Binary arithmetic
- Basic adder circuits
- Magnitude adder
- Signed binary numbers
- Signed adder: overflow
- Adder/subtractor
- **ALU**

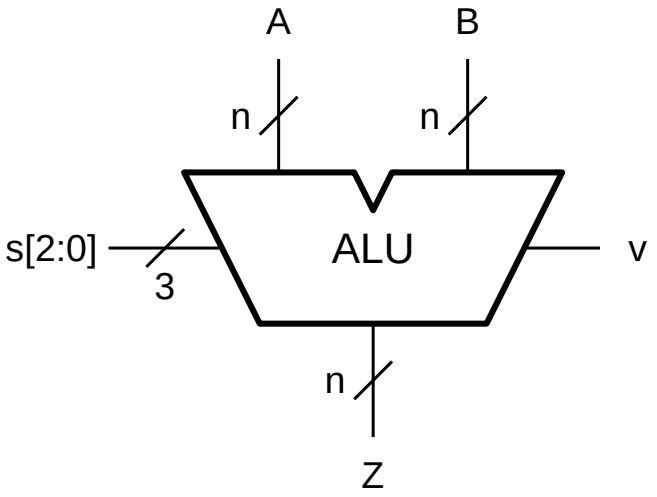
ALU

- Data processing operations concentrated in a single device
  - Logic operations
  - Arithmetic operations
- One of the most important parts of a computer



Sample ALU

- Arithmetic-Logic Unit using two's complement representation
  - A = $TCR_n(a)$, B = $TCR_n(b)$, Z = $TCR_n(z)$
  - v: overflow output (z is not representable in two's complement

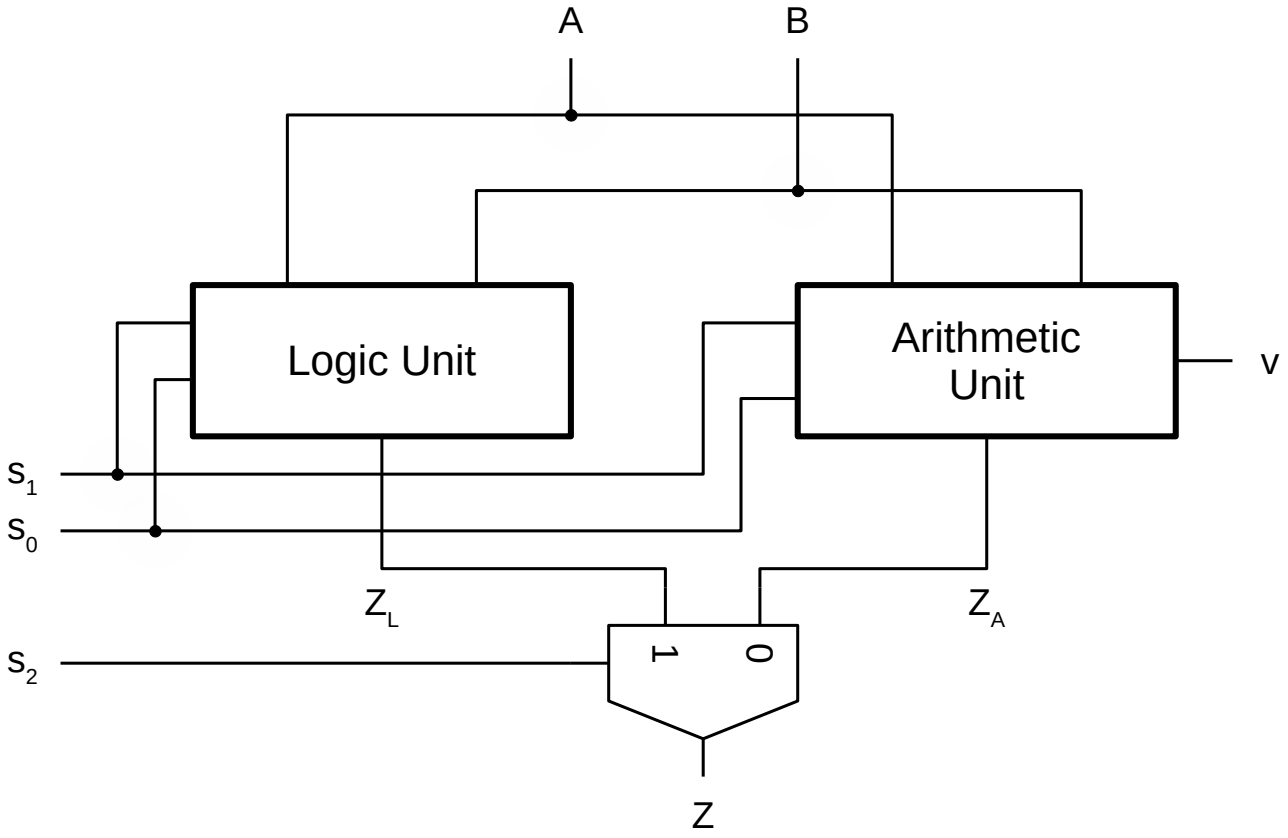


$s_2s_1s_0$	z	Z	
000	a + b	A + B	Arithmetic (s_2=0)
001	a - b	A+TC(B)	
010	a + 1	A + 1	
011	a - 1	$A+2^n-1$	
100	A AND B		Logic (s_2=1)
101	A OR B		
110	A XOR B		
111	NOT A		

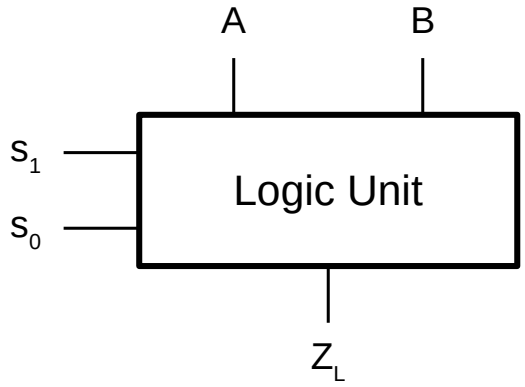
ALU design strategy

- Divide and conquer! (again)
  - Design a logic unit and an arithmetic unit independently. Select with s_2 (multiplexer?)
- Logic unit
  - Select the appropriate operation with s_1 and s_0 (multiplexer?)
- Arithmetic unit
  - Start with a magnitude's adder
  - Calculate adder's inputs (B and Cin) to obtain the desired result.
  - Select the appropriate B and Cin with s_1 and s_0 (multiplexer?)

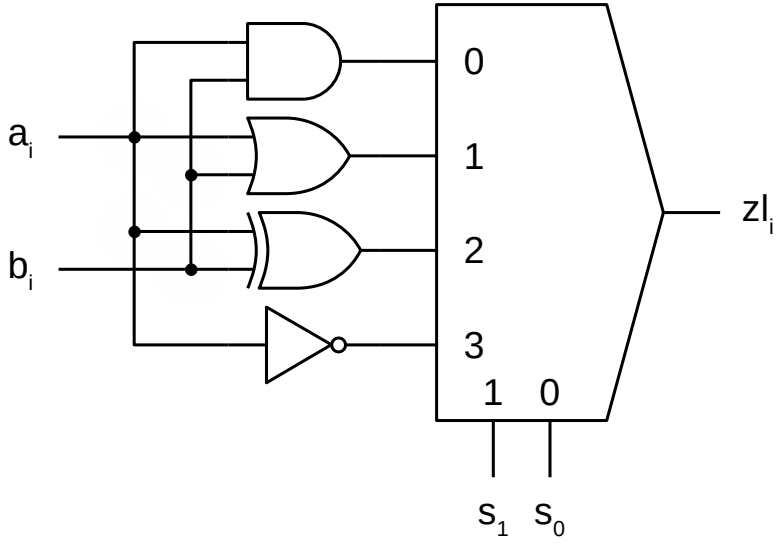
ALU design



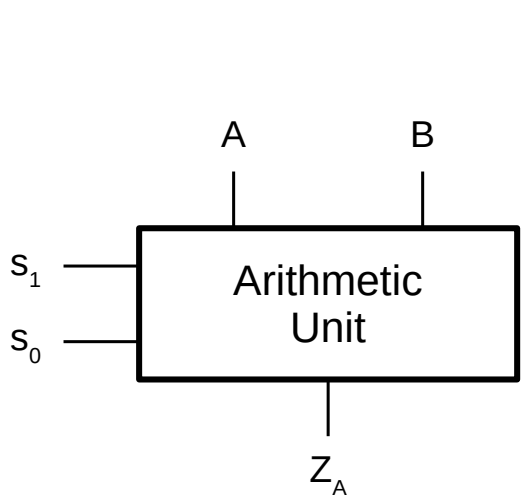
ALU
Diseño de la unidad lógica

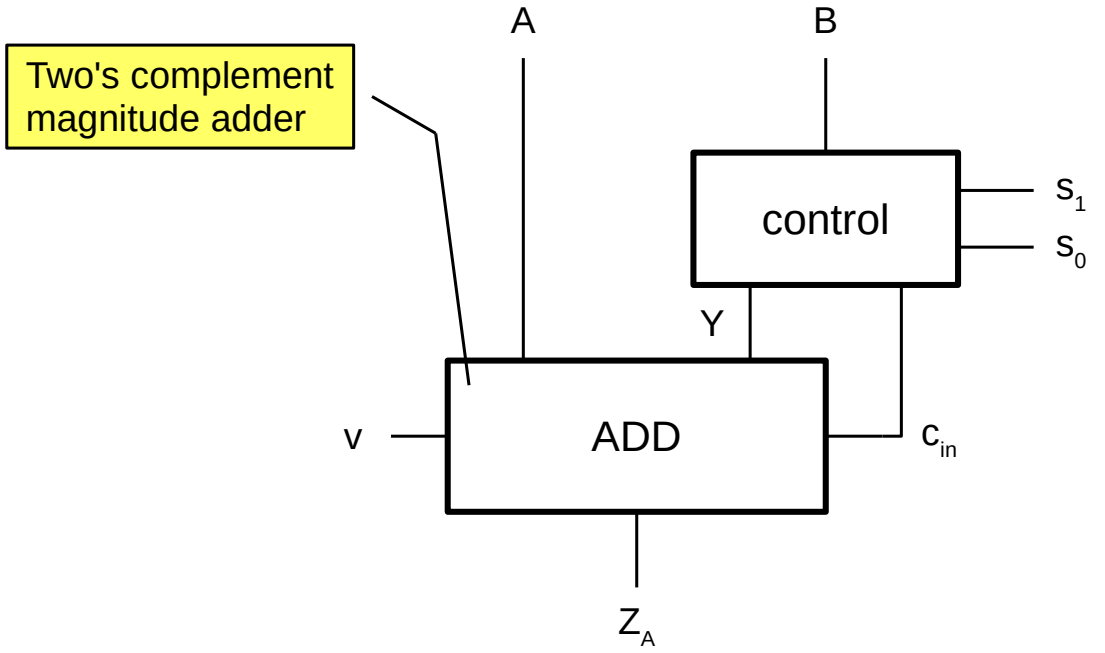


s_1s_0	zl	zl_i
00	A AND B	a_i AND b_i
01	A OR B	a_i OR b_i
10	A XOR B	a_i XOR b_i
11	NOT A	NOT a_i



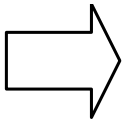
ALU design. Arithmetic unit





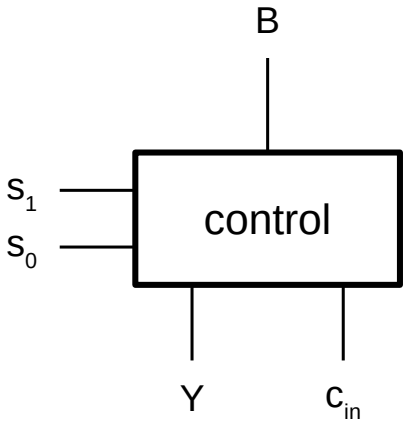
s_1s_0	z_A	Z_A
00	a + b	A + B
01	a – b	A + $\overline{B}$ + 1
10	a + 1	A + 1
11	a – 1	A + 2^n-1

$Z_A = A + Y + c_{in}$

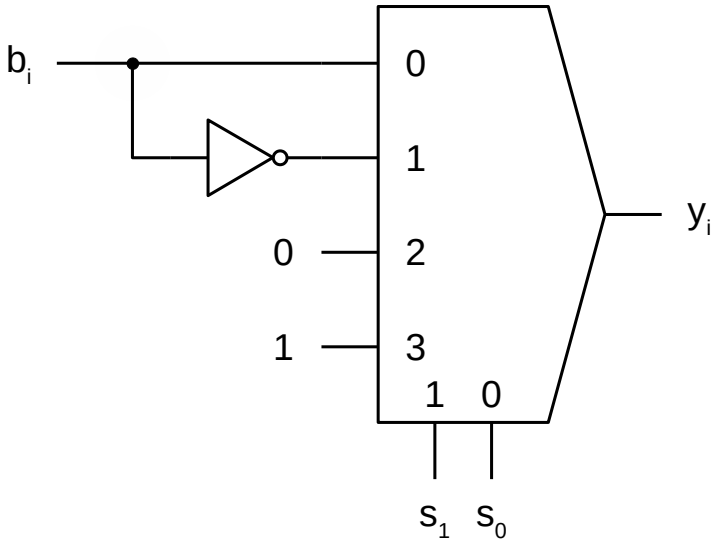


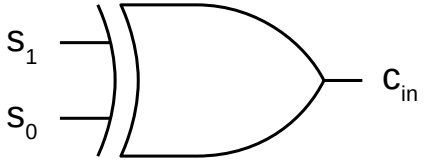
s_1s_0	Y	y_i	c_{in}
00	B	b_i	0
01	$\overline{B}$	$\overline{b_i}$	1
10	0	0	1
11	2^n-1	1	0

ALU design. Arithmetic unit

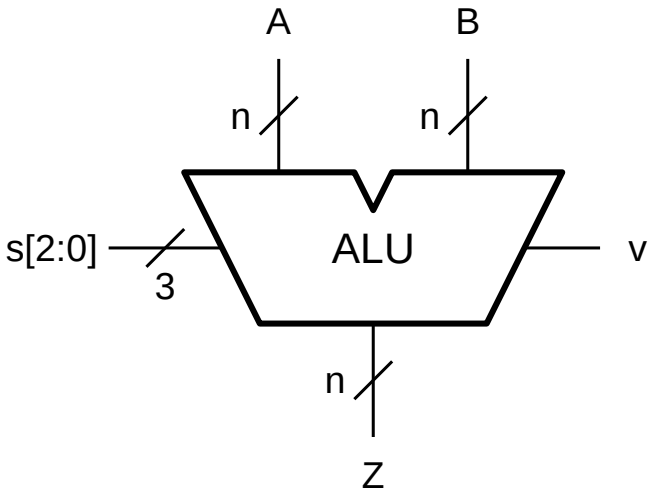


s_1s_0	Y	y_i	c_{in}
00	B	b_i	0
01	$\overline{B}$	$\overline{b_i}$	1
10	0	0	1
11	2^n-1	1	0





ALU. Verilog description




```
module alu #(parameter N = 8)(
  input signed [N-1:0] a,
  input signed [N-1:0] b,
  input [2:0] s,
  output reg signed [N-1:0] z,
  output reg v
  );

  reg signed [N:0] f;
```

```
  always @* begin
    ov = 0;

    if (s[2] == 0) begin  // Arithmetic
      case (s[1:0])
        2'b00:  f = a + b;
        2'b01:  f = a - b;
        2'b10:  f = a + 1;
        2'b11:  f = a - 1;
      endcase

      v = (f[N] == f[N-1])? 0: 1;

      z = f[N-1:0];

    end else                  // Logic
      case (s[1:0])
        2'b00:  z = a & b;
        2'b01:  z = a | b;
        2'b10:  z = a ^ b;
        2'b11:  z = ~a;
      endcase
  end // always

endmodule // alu
```

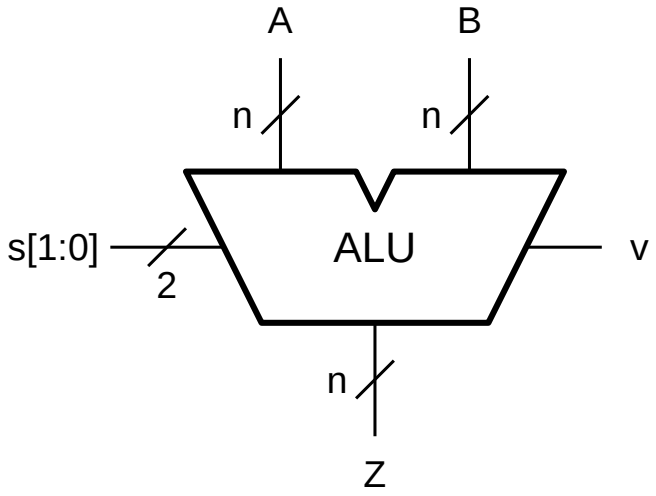
ALU design

Example 8

Design an n bit ALU according to the operation table and figure below. Arithmetic operations use two's complement representation. The ALU has an overflow (v) output.

a) Base the design in a magnitude adder with an overflow output.

b) Write a Verilog description.



$s_1 s_0$	z
00	a + 2
01	a - b
10	NOT a
11	NOT b

ALU design

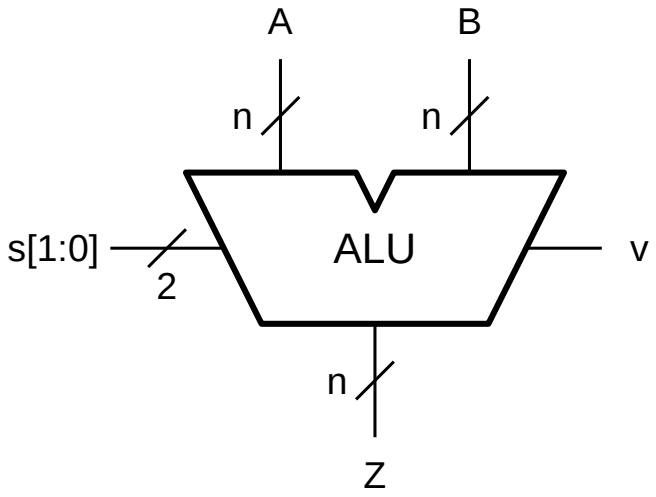
Example 9

Design an n bit ALU according to the operation table and figure below. Arithmetic operations use two's complement representation. The ALU has an overflow (v) output.

a) Base the design in a magnitude adder with an overflow output.

b) Write a Verilog description.

(NOTE: be careful when calculating the overflow)



$s_1 s_0$	z
00	a + 2b
01	a - 4
10	NOT a
11	a XOR b