Implementing Binary Arithmetic

Cezar Ionescu
cezar.ionescu@th-deg.de

WS 2023-24

TECHNISCHE HOCHSCHULE DEGGENDORF

Outline

Admin

Questions?

Review

Binary arithmetic

- Addition (carry is 0 or 1):

```
1111     (carry)
 10110
+01110
------
100100
```

- Multiplication

```
  1110
x 1001
------
  1110
 0000
0000
1110
-------
1111110
```

Logic gates

- NOT, OR, AND, XOR, NAND, NOR
- sum-of-terms expressions
- Karnaugh maps

Implementing binary arithmetic

Reference

We follow Chapter 4 of *Foundations of Computer Science*, A.J.T. Colin, Macmillan 1980, which is also the source of the following images.

One-bit addition

- Addition on natural numbers is a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
  - If $n + n = 0$, then $n = 0$
- If we consider addition on $\mathbb{B}$ to be of the analog type $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, we have a loss of information:
  - $1 + 1 = 0$ (therefore $1 = 0$?)
- To avoid loss of information, we can extend the output type: $\mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}$
  - $1 + 1 = (0, 1) \neq 0$
  - alternative notation:

```
add₀₅ : B² -> B²
add₀₅ (x, y) = (sum, carry)
where sum   = (x + y) mod 2
      carry = (x + y) div 2
```

- This function is called **half-adder**.

One-bit addition

- Addition on natural numbers is a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
  - If $n + n = 0$, then $n = 0$
- If we consider addition on $\mathbb{B}$ to be of the analog type $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, we have a loss of information:
  - $1 + 1 = 0$ (therefore $1 = 0$?)
- To avoid loss of information, we can extend the output type: $\mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}$
  - $1 + 1 = (0, 1) \neq 0$
  - alternative notation:

```
add₀₅ : B² -> B²
add₀₅ (x, y) = (sum, carry)
where sum   = (x + y) mod 2
      carry = (x + y) div 2
```

- This function is called **half-adder**.

One-bit addition

- Addition on natural numbers is a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
  - If $n + n = 0$, then $n = 0$
- If we consider addition on $\mathbb{B}$ to be of the analog type $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, we have a loss of information:
  - $1 + 1 = 0$ (therefore $1 = 0$?)
- To avoid loss of information, we can extend the output type: $\mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}$
  - $1 + 1 = (0, 1) \neq 0$
  - alternative notation:

```
add₀₅ : B² -> B²
add₀₅ (x, y) = (sum, carry)
where sum   = (x + y) mod 2
      carry = (x + y) div 2
```

- This function is called **half-adder**.

One-bit addition

- Addition on natural numbers is a function of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
    - If $n + n = 0$, then $n = 0$
- If we consider addition on $\mathbb{B}$ to be of the analog type $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, we have a loss of information:
    - $1 + 1 = 0$ (therefore $1 = 0$?)
- To avoid loss of information, we can extend the output type: $\mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}$
    - $1 + 1 = (0, 1) \neq 0$
    - alternative notation:

```
add₀₅ : B² -> B²
add₀₅ (x, y) = (sum, carry)
where sum   = (x + y) mod 2
      carry = (x + y) div 2
```

- This function is called **half-adder**.

One-bit addition

- ```
  add₀₅ : B² -> B²
  add₀₅ (x, y) = (sum, carry)
  where sum   = (x + y) mod 2
        carry = (x + y) div 2
  ```
- We want to use `add05` to add two-digit binary numbers, e.g. `01 + 11`

```
    0 1
    1 1
  ------
  c₁r₁r₀

  add₀₅(1, 1) = (sum₀, car₀) = (0, 1) -- ok
  add₀₅(0, 1) = (sum₁, car₁) = (1, 0) -- ok
```

But the result is not `car1sum1sum0 = [1, 0]` (first element of the results of `add05`), because we've lost `car0`.
```

One-bit addition

- ```
  add₀₅ : B² -> B²
  add₀₅ (x, y) = (sum, carry)
  where sum   = (x + y) mod 2
        carry = (x + y) div 2
  ```
- We want to use add₀₅ to add two-digit binary numbers, e.g. `01 + 11`

```
  0 1
  1 1
------
c₁r₁r₀

add₀₅(1, 1) = (sum₀, car₀) = (0, 1) -- ok
add₀₅(0, 1) = (sum₁, car₁) = (1, 0) -- ok
```

But the result is not $car_1 sum_1 sum_0$ = [1, 0] (first element of the results of add₀₅), because we've lost car_0.
```

One-bit addition

- ```
  add05 : B² -> B²
  add05 (x, y) = (sum, carry)
  where sum   = (x + y) mod 2
        carry = (x + y) div 2
  ```
- We want to use `add05` to add two-digit binary numbers, e.g. `01 + 11`

```
    0 1
    1 1
  ------
  c1r1r0

  add05(1, 1) = (sum0, car0) = (0, 1) -- ok
  add05(0, 1) = (sum1, car1) = (1, 0) -- ok
```

But the result is not `car1sum1sum0 = [1, 0]` (first element of the results of `add05`), because we've lost `car0`.
```

One-bit addition

To accomodate the carry, we need to extend the input type as well.

```
add₁ : B³ -> B²
add₁ (x, y, c₀) = (sum, carry) where
    sum   = (x + y + c₀) mod 2
    carry = (x + y + c₀) div 2
```

Example: Using `add₁` to implement *two-bit* addition.

```
a₁a₀ + b₁b₀ = c₁s₁s₀ where
    (s₀, c₀) = add₁(a₀, b₀, 0)  -- initial carry is 0
    (s₁, c₁) = add₁(a₁, b₁, c₀) -- intermediate carry is c₀
```

One-bit addition

To accomodate the carry, we need to extend the input type as well.

```
add₁ : B³ -> B²
add₁ (x, y, c₀) = (sum, carry) where
    sum   = (x + y + c₀) mod 2
    carry = (x + y + c₀) div 2
```

Example: Using `add₁` to implement *two-bit* addition.

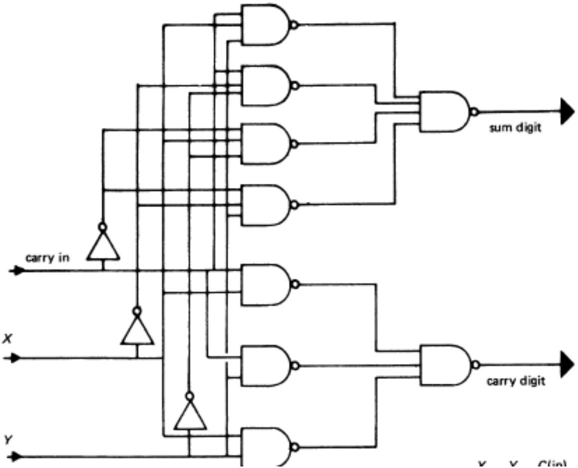
```
a₁a₀ + b₁b₀ = c₁s₁s₀ where
     (s₀, c₀) = add₁(a₀, b₀, 0)  -- initial carry is 0
     (s₁, c₁) = add₁(a₁, b₁, c₀) -- intermediate carry is c₀
```

Implementing one-bit addition

x	y	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

The one-bit adder

A one-bit adder is a circuit that implements one-bit addition. For example



Alternative implementation

The half-adder function can also be used to implement a one-bit adder:

```
add₁ (x, y, c₀) = (sum, carry) where
  (s₁, c₁)   = add₀₅(x, y)
  (sum, c₂)  = add₀₅(s₁, c₀)
  carry      = c₁ OR c₂
```

The advantage of this formulation is that the half-adder is a very simple circuit to implement:

```
add₀₅(x, y) = (x XOR y, x AND y)
```

Alternative implementation

The half-adder function can also be used to implement a one-bit adder:

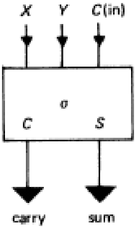
```
add₁ (x, y, c₀) = (sum, carry) where
  (s₁, c₁)   = add₀.₅(x, y)
  (sum, c₂)  = add₀.₅(s₁, c₀)
  carry      = c₁ OR c₂
```

The advantage of this formulation is that the half-adder is a very simple circuit to implement:

```
add₀.₅(x, y) = (x XOR y, x AND y)
```

Block representation of a one-bit adder

We can represent the one-bit adder as a *black box*, hiding the implementation:

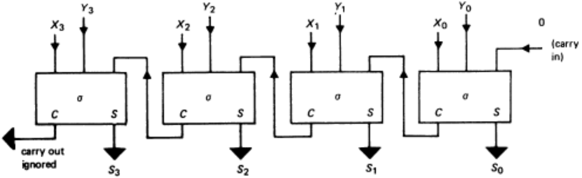


Four-bit adders

```
add₄ (x₃x₂x₁x₀, y₃y₂y₁y₀) = c₃s₃s₂s₁s₀
     (s₀, c₀) = add₁(x₀, y₀, 0)  -- initial carry is 0
     (s₁, c₁) = add₁(x₁, y₁, c₀)
     (s₂, c₂) = add₁(x₂, y₂, c₁)
     (s₃, c₃) = add₁(x₃, y₃, c₂)
```

Four-bit adders

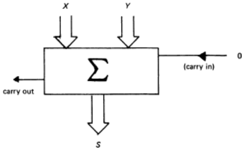
Implementation:



Four-bit systems

In an *n-bit* system, circuits will usually work on groups of n bits in size (plus the occasional *control* bits like the carry).

Therefore, a more compact representation is adopted for such groups. For example:



This is similar to the introduction of vectors in linear algebra, allowing us to use $\mathbf{x} \in \mathbb{R}^n$ to represent $(x_{n-1}, ..., x_0)$, $x_i \in \mathbb{R}$.

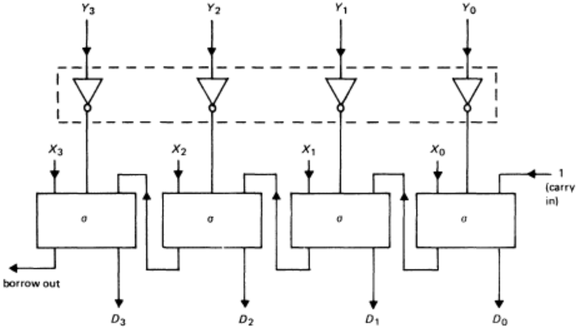
Subtraction on n bit numbers

- Let xs and ys be n bit numbers. Then

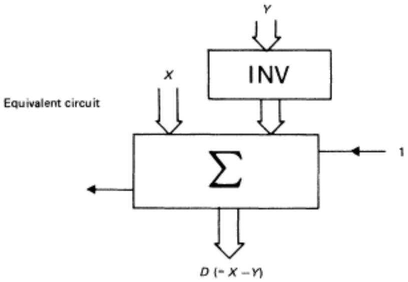
```
xs - ys = xs + (-ys) = xs + (flipₙ(ys) + 1)
```

where $\texttt{flip}_n$ inverts every bit of an n bit input.

Implementing two's complement



Implementing subtraction



The multiplexer

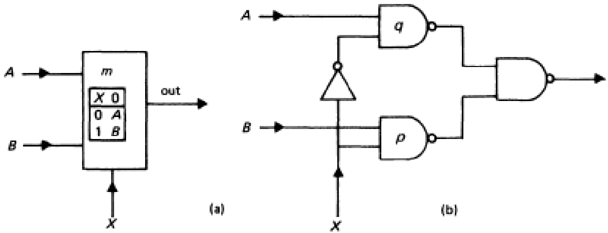
The two-way multiplexer selects one of the two input lines depending on the value of the control input (`sel`).

sel	x	y	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

A more compact description of the multiplexer:

sel	out
0	x
1	y

Implementing a multiplexer



A four-way multiplexer

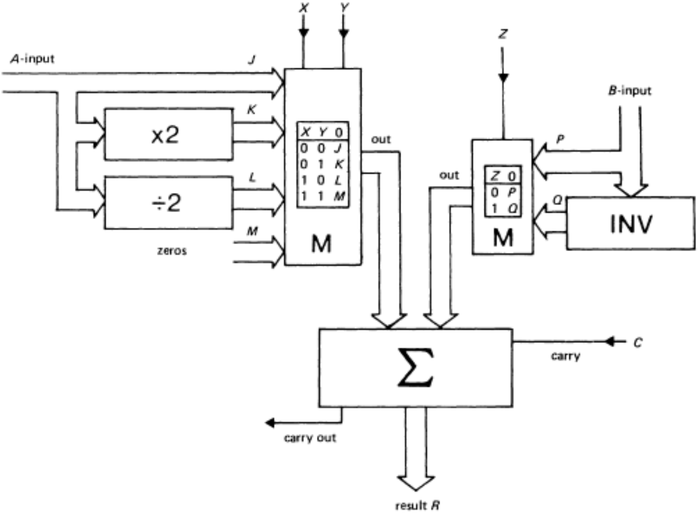
We can also build multiplexers that select one of more than two (usually 2^n) alternatives. In this case, we need more than one bit to represent the selection. For example, a four-way multiplexer selects one out of four signals u, v, x, y.

sel_0 sel_1	out
00	u
01	v
10	x
11	y

Shifting binary numbers

- Given a binary number $x_1x_2...x_n$, the operation *left shift* produces the (truncated) result of the multiplication with 2: $x_2x_3...x_n0$.
- Similarly, the *right shift* produces the (truncated) result of dividing the number by 2: $0x_1x_2...x_{n-1}$.

An arithmetic unit



Functionality table

X	Y	Z	C	Effective function
0	0	0	0	$R = A + B$
0	0	0	1	$R = A + B + 1$
0	0	1	0	$R = A - B - 1$
0	0	1	1	$R = A - B$
0	1	0	0	$R = 2A + B$
0	1	0	1	$R = 2A + B + 1$
0	1	1	0	$R = 2A - B - 1$
0	1	1	1	$R = 2A - B$
1	0	0	0	$R = \frac{1}{2}A + B$
1	0	0	1	$R = \frac{1}{2}A + B + 1$
1	0	1	0	$R = \frac{1}{2}A - B - 1$
1	0	1	1	$R = \frac{1}{2}A - B$
1	1	0	0	$R = B$
1	1	0	1	$R = B + 1$
1	1	1	0	$R = B - 1$
1	1	1	1	$R = B$