Lecture 1

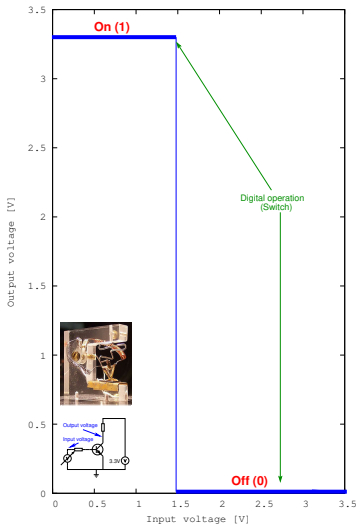
Digital system design

Introduction

CS173 - Conception de systèmes numériques
November 2016

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Digital Logic versus Analog circuits



Transistor:

- Discovered in 1947 by Shockley,Bardeen and Brattain
- Basis for almost all digital and analog circuits nowadays

Application:

- Analog as amplifier
- Possible values unlimited
- Digital as switch
- Only 2 values: on and off
- Digital: 2 valued digit

Digital Logic

We have seen: Digital: two valued digit
What about: Logic

Example:

When I'm **late and take a taxi or** I'm **not late and take the bus** then I'm **on time for the course**

- Each logic expression has variables
- Each logic expression has operators
- Each logic expression has one or more inputs
- Each logic expression has one output

Digital Logic is based on logic expressions of two valued (*binary*) variables

Basic Logic Operators

Operator:	Example:	Short:
NOT ▶ Complement ▶ Short notation: $\neg A$	If you can**not see** then you are **blind**	blind $=$ $\neg$see
AND ▶ Conjunction ▶ Short notation: $A \wedge B$	If you mix **flour and milk and eggs** then you have **pancakes**	pancakes $=$ flour $\wedge$ milk $\wedge$ eggs
OR ▶ Disjunction ▶ Short notation: $A \vee B$	If **John takes money or Marry takes money** then there is **less money**	less money $=$ John takes money $\vee$ Marry takes money

Basic Logic Operators (cont.)

What about:
If John is walking **or** John is biking then he is active

This is a *disjunction*
But a special one, as John cannot walk and bike at the same time!

This operator is called the *Exclusive OR* (**XOR**).

- Disjunction
- Short notation: $A \oplus B$
- Equivalence: $A \oplus B = (\neg A \wedge B) \vee (A \wedge \neg B)$

Truth Table

- Each logical expression has *n* inputs and one output
- Each input can **only** have two states, namely True (**T**) or False (**F**)
- For *n* inputs we can therefore have 2^n different input combinations
- We can write the *truth table* of a logical expression
- We can compress the *truth table* by using the **don't care symbol X** or -
- The - denotes that the variable can be either **T** or **F**

If **John takes money** **or** **Marry takes money** then there is **less money**

$Y = A \vee B$

A	B	Y
F	F	F
F	T	T
T	F	T
T	T	T

A	B	Y
F	F	F
F	T	T
T	-	T

Digital Logic $\stackrel{?}{=}$ Algebra

- We have seen a transistor giving in its digital operation the two values On (3.3V) and Off (0V)
- We have also seen logic expressions with their two values True and False
- Let us define that :
  1. Off = False = **0** and
  2. On = True = **1**.
- What happens than with the conjunction?
- We have a logic multiplication...

If you have **a password and a login** than **you can use the computer**

$C = P \wedge L$

P	L	C
0	0	0
0	1	0
1	0	0
1	1	1

$C = P \cdot L$

Digital Logic $\stackrel{?}{=}$ Algebra

- What happens than with the disjunction?
- Is it a logic addition?
- We cannot express **2**; we only have two quantities (**0** and **1**)
- Hence we have only two possibilities:
  1. **1**+**1**=**1**; we call this the logic addition
  2. **1**+**1**=**0**; we call this the logic exclusion

A	O	F
0	0	0
0	1	1
1	0	1
1	1	1

$F \stackrel{?}{=} A+O$

$0+0=0$

$0+1=1$

$1+0=1$

$1+1=1$

Digital Logic = Algebra

$Y=A+B$

Logic addition

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Y
0	0	0
-	1	1
1	-	1

$Y=A\oplus B$

Logic exclusion

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y=A\cdot B$

Logic multiplication

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

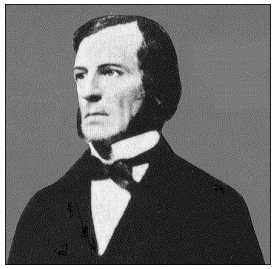
A	B	Y
0	-	0
-	0	0
1	1	1

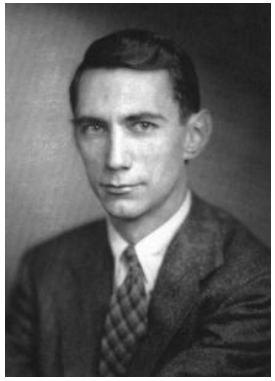
$Y=\overline{A}$

Logic complement

A	Y
0	1
1	0

Boolean Algebra





George Boole (1815–1864)

- Formulated in 1847 the *Mathematical Analysis of Logic*
- Based on mathematics restricted to two quantities, 0 and 1
- Known as Boolean Algebra

Claude Elwood Shannon (1916–2001)

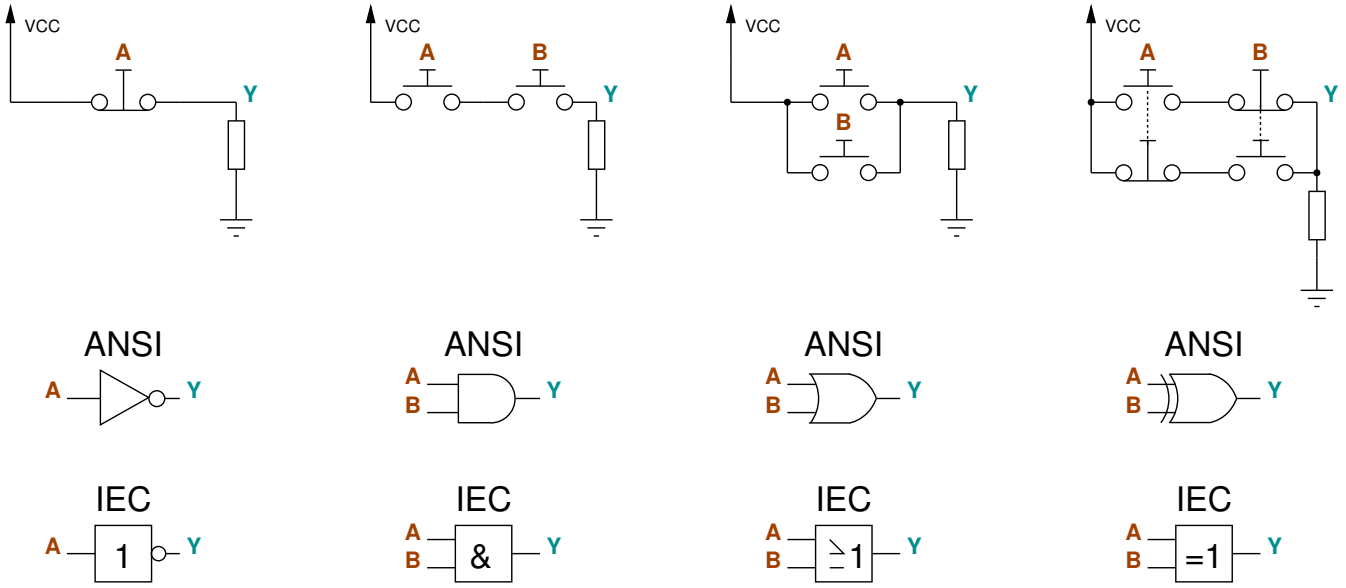
- Published in 1937 *A Symbolic Analysis of Relay and Switching Circuits*
- Proofs that Boolean Algebra could be used for digital circuit simplification
- Basis for all circuit design and simplification tools/practices

Boolean Algebra in Circuits

Mathematical Logic Operator:

NOT **AND** **OR** **XOR**

Schematic Equivalent and Gates:



The **ANSI** symbols are mainly used in the USA and ASIA, whilst the **IEC** ones are mainly seen in Europe

Lecture 2

Digital system design

Boolean algebra and optimizing logic functions

CS173 - Conception de systèmes numériques
November 2016

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Properties of Boolean Algebra

- We have seen that digital logic forms an algebra; the boolean algebra
- As each algebra, the boolean algebra has properties, we will review them quickly and without proof
- Elementary properties:
  1. $A \cdot A = A$
  2. $A + A = A$
  3. $A \oplus A = 0$
- Absorbtion properties:
  1. $A \cdot (A + B) = A$
  2. $A + (A \cdot B) = A$
- Constant properties:
  1. $A \cdot 0 = 0$
  2. $A \cdot 1 = A$
  3. $A + 0 = A$
  4. $A + 1 = 1$
  5. $A \oplus 0 = A$
  6. $A \oplus 1 = \overline{A}$

Properties of Boolean Algebra

- Complement properties:
  1. $A \cdot \overline{A} = 0$
  2. $A + \overline{A} = 1$
  3. $A \oplus \overline{A} = 1$
  4. $\overline{\overline{A}} = A$
- Commutative properties:
  1. $A \cdot B = B \cdot A$
  2. $A + B = B + A$
  3. $A \oplus B = B \oplus A$
- Distributive properties:
  1. $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
  2. $A + (B \cdot C) = (A + B) \cdot (A + C)$
  3. $A \cdot (B \oplus C) = (A \cdot B) \oplus (A \cdot C)$
- Associative properties:
  - $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
  - $A + (B + C) = (A + B) + C$
  - $A \oplus (B \oplus C) = (A \oplus B) \oplus C$

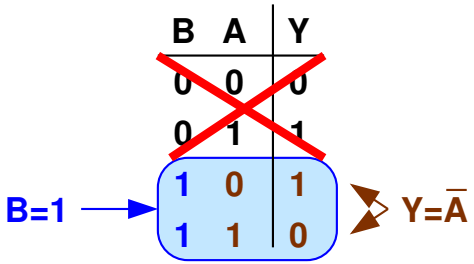
Exercise

Show that $\mathbf{A} \oplus \mathbf{1} = \overline{\mathbf{A}}$, using:
a) A truthtable.
b) Boolean algebra.

Solution:

We know that the XOR is defined by: $\mathbf{Y} = \mathbf{A} \oplus \mathbf{B}$

a) Truthtable



1. The truthtable for the XOR.
2. We take $\mathbf{B} = \mathbf{1}$.
3. We see directly that $\mathbf{A} \oplus \mathbf{1} = \overline{\mathbf{A}}$.

b) Boolean algebra

1. $\mathbf{Y} = \mathbf{A} \oplus \mathbf{B}$.
2. We rewrite the function in it's equivalent form.
3. $\mathbf{A} \oplus \mathbf{B} = \overline{\mathbf{A}} \cdot \mathbf{B} + \mathbf{A} \cdot \overline{\mathbf{B}}$.
4. We take $\mathbf{B} = \mathbf{1}$.
5. $\mathbf{A} \oplus \mathbf{1} = \overline{\mathbf{A}} \cdot \mathbf{1} + \mathbf{A} \cdot \mathbf{0}$.
6. $\mathbf{A} \oplus \mathbf{1} = \overline{\mathbf{A}} + \mathbf{0}$.
7. $\mathbf{A} \oplus \mathbf{1} = \overline{\mathbf{A}}$.

Exercise

Show that $\mathbf{A} \oplus (\mathbf{B} \cdot \mathbf{C}) \neq (\mathbf{A} \oplus \mathbf{B}) \cdot (\mathbf{A} \oplus \mathbf{C})$ using Boolean algebra:

Solution:

1. We start with the left function: $\mathbf{A} \oplus (\mathbf{B} \cdot \mathbf{C})$.
2. Using the definition: $\mathbf{D} \oplus \mathbf{E} = \mathbf{D} \cdot \overline{\mathbf{E}} + \overline{\mathbf{D}} \cdot \mathbf{E}$.
3. Hence: $\mathbf{A} \oplus (\mathbf{B} \cdot \mathbf{C}) = \mathbf{A} \cdot \overline{\mathbf{B} \cdot \mathbf{C}} + \overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C}$.

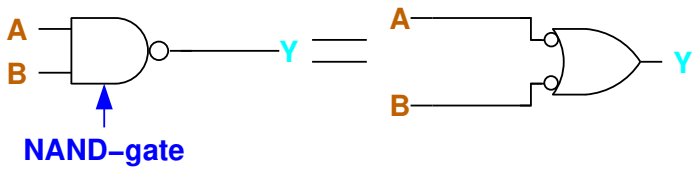
1. Now we take the right function: $(\mathbf{A} \oplus \mathbf{B}) \cdot (\mathbf{A} \oplus \mathbf{C})$.
2. Using the definition: $\mathbf{D} \oplus \mathbf{E} = \mathbf{D} \cdot \overline{\mathbf{E}} + \overline{\mathbf{D}} \cdot \mathbf{E}$.
3. Gives us: $(\overline{\mathbf{A}} \cdot \mathbf{B} + \mathbf{A} \cdot \overline{\mathbf{B}}) \cdot (\overline{\mathbf{A}} \cdot \mathbf{C} + \mathbf{A} \cdot \overline{\mathbf{C}})$
4. Multiplying it out gives: $\overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C} + \mathbf{0} \cdot \mathbf{B} \cdot \overline{\mathbf{C}} + \mathbf{0} \cdot \overline{\mathbf{B}} \cdot \mathbf{C} + \mathbf{A} \cdot \overline{\mathbf{B}} \cdot \overline{\mathbf{C}}$.
5. Hence: $(\mathbf{A} \oplus \mathbf{B}) \cdot (\mathbf{A} \oplus \mathbf{C}) = \mathbf{A} \cdot \overline{\mathbf{B}} \cdot \overline{\mathbf{C}} + \overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C}$

And: $\mathbf{A} \cdot \overline{\mathbf{B} \cdot \mathbf{C}} + \overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C} \neq \mathbf{A} \cdot \overline{\mathbf{B}} \cdot \overline{\mathbf{C}} + \overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C}$
(Hint: make a truthtable).

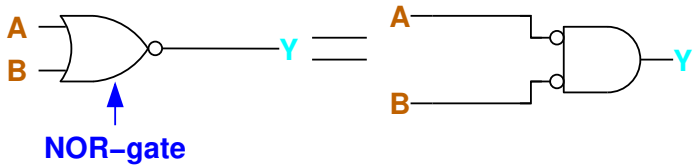
Theorems of Boolean Algebra

The theorems of *August de Morgan (1806-1871)*:

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$



$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

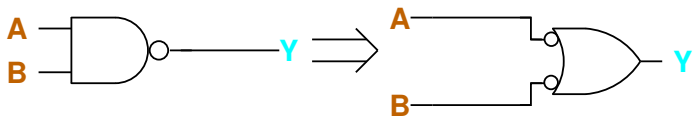


- De Morgen postulated two theorems.
- We can draw the gate-equivalent.
- The NOT gates can be merged.
- And the same for the second theorem.

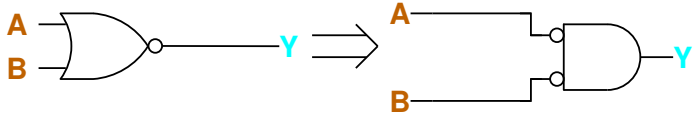
Theorems of De Morgan (cont.)

The theorems of *August de Morgan (1806-1871)*:

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$



$$\overline{A + B} = \overline{A} \cdot \overline{B}$$



- The theorems of De Morgan are very important as they show:
- Any logic expression can be formulated with **only OR** and **NOT**
- Any logic expression can be formulated with **only AND** and **NOT**

Theorems of De Morgan (Example)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$Y = A + B + C$$

- Enumerating all terms for which Y= 1 (minterms) leads to a sum of 7 products!
- Better: Enumerate all terms for which Y= 0 (maxterms) and use De Morgan

Theorems of De Morgan (Example 2)

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Y = \overline{\overline{\overline{A} \cdot B \cdot \overline{C}} \cdot \overline{A \cdot B \cdot \overline{C}} \cdot \overline{A \cdot B \cdot C}}$$

$$\overline{\overline{A \cdot B \cdot \overline{C}} + A \cdot B \cdot \overline{C} + A \cdot B \cdot C}$$

$$\overline{\overline{(A + \overline{B} + C) \cdot (\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{B} + \overline{C})}}$$

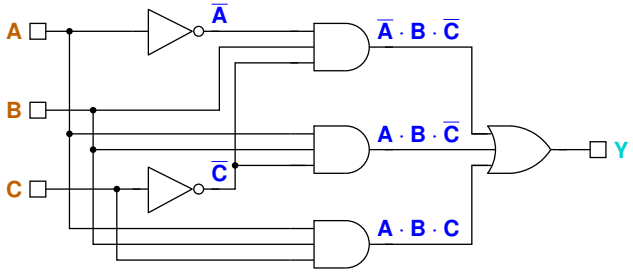
$$\overline{\overline{(A + \overline{B} + C)} + \overline{(\overline{A} + \overline{B} + C)} + \overline{(\overline{A} + \overline{B} + \overline{C})}}$$

- A logic equation can be formulated in the *disjunct* form; this form is also called *sum of products*
- A logic equation can be formulated in the *conjunct* form; this form is also called *product of sums*

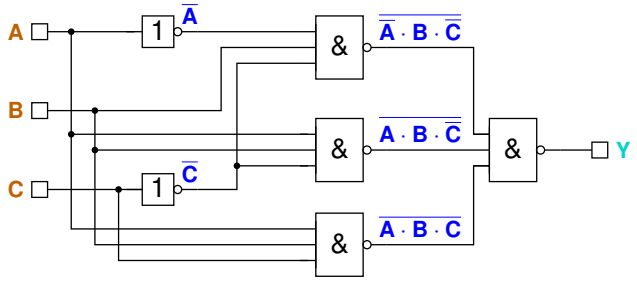
Theorems of De Morgan (Example 2 gate)

$$Y =$$

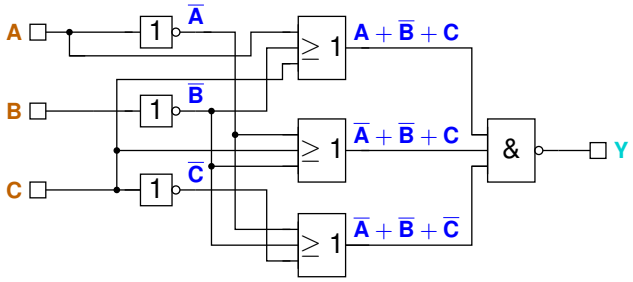
$$\overline{A} \cdot B \cdot \overline{C} + A \cdot B \cdot \overline{C} + A \cdot B \cdot C$$



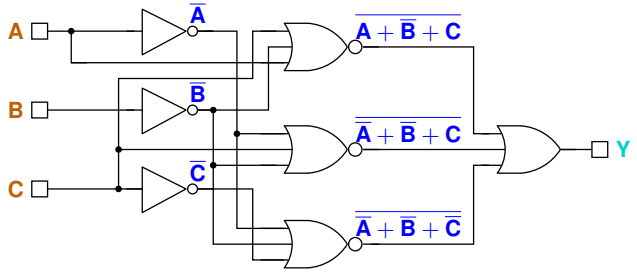
$$\overline{\left(\overline{A} \cdot B \cdot \overline{C}\right)} \cdot \overline{\left(A \cdot B \cdot \overline{C}\right)} \cdot \overline{(A \cdot B \cdot C)}$$



$$\overline{\left(A + \overline{B} + C\right)} \cdot \overline{\left(\overline{A} + \overline{B} + C\right)} \cdot \overline{\left(\overline{A} + \overline{B} + \overline{C}\right)}$$



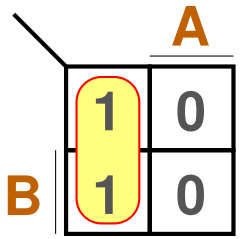
$$\overline{\left(A + \overline{B} + C\right)} + \overline{\left(\overline{A} + \overline{B} + C\right)} + \overline{\left(\overline{A} + \overline{B} + \overline{C}\right)}$$



Optimizing Logic Functions

A	B	Y
0	0	1
0	1	1
1	0	0
1	1	0

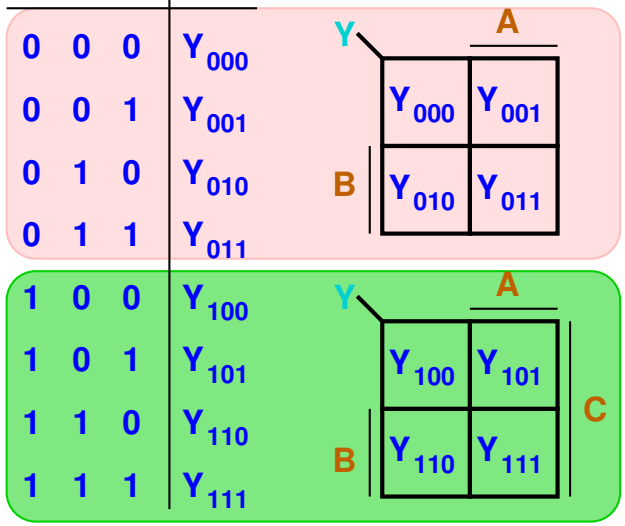
$$Y = \overline{A} \cdot \overline{B} + \overline{A} \cdot B$$

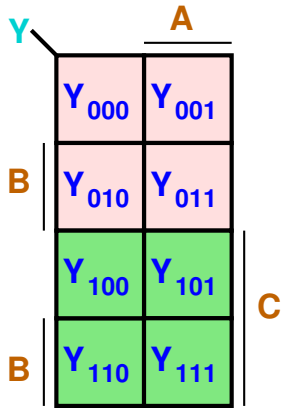


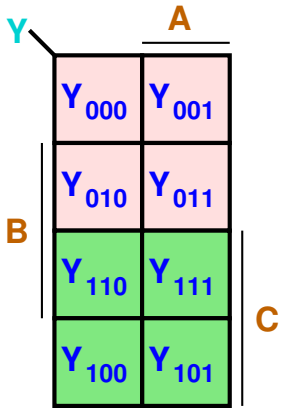
- The *disjunct* form does not always provide the smallest equation
- For this simple example, it can be seen in the truth table, but what about equations with five inputs?
- We can use the graphic optimizing method of Karnaugh, the Karnaugh diagram
- By selecting all groups of 2^m **1**'s we can eliminate variables: $Y = \overline{A}$

The Karnaugh diagram

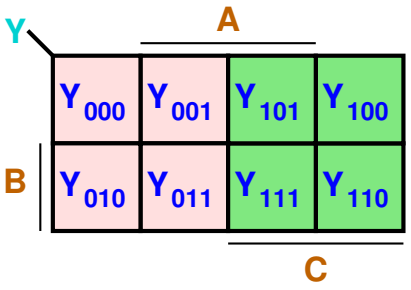
C	B	A	Y
0	0	0	Y_{000}
0	0	1	Y_{001}
0	1	0	Y_{010}
0	1	1	Y_{011}
1	0	0	Y_{100}
1	0	1	Y_{101}
1	1	0	Y_{110}
1	1	1	Y_{111}

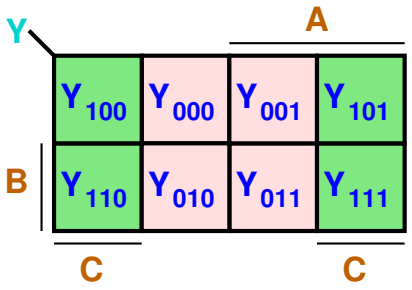






And we can continue for 4+ variables

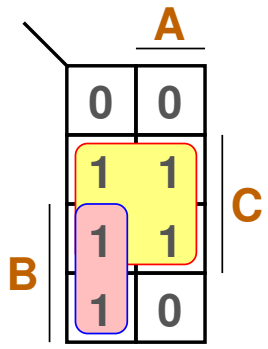




Optimizing Logic Functions with three Variables

$$Y = C + \overline{A} \cdot B$$

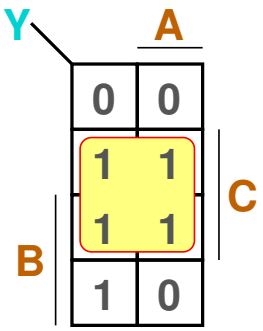
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



- We start selecting the biggest *group* of **1** 's (minterms)
- We continue until all minterms are selected

Valid Karnaugh groups

What is a valid group? In the Karnaugh diagram below we have a group of four minterms. We have 2^m minterms with m=2.



$$
\begin{aligned}
Y &= \overline{A}\cdot\overline{B}\cdot C + A\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot C + A\cdot B\cdot C \\
&= C\cdot\left(\overline{A}\cdot\overline{B} + A\cdot\overline{B} + \overline{A}\cdot B + A\cdot B\right) \\
&= C\cdot\left(\overline{A}\cdot\left(\overline{B}+B\right) + A\cdot\left(\overline{B}+B\right)\right) \\
&= C\cdot\left(\left(\overline{B}+B\right)\cdot\left(\overline{A}+A\right)\right) \\
&= C\cdot((1)\cdot(1))
\end{aligned}
$$

- We can put C outside of the brackets; the variable C is important for this group!
- We can also put A and $\overline{A}$ outside brackets. As easily can be seen: there are exactly 2^{m-1} minterms in the area where A=1 and the other 2^{m-1} minterms are in the area where A=0
- Finally we play with $\left(\overline{B}+B\right)$.
- Both A and B are *don't care* for this group as $\left(\overline{A}+A\right)=1$ and $\left(\overline{B}+B\right)=1$!

Valid Karnaugh groups

Hence we have a valid group when:

1. The group has exactly 2^m minterms or 2^m maxterms.
2. The group has exactly m-variables *don't care*.

A variable **E** is *don't care* when:

1. There are exactly 2^{m-1} minterms/maxterms from the group in the region where **E**=**1**.
2. There are exactly 2^{m-1} minterms/maxterms from the group in the region where **E**=**0** ($\overline{\mathbf{E}}$-region).

A variable **F** influences the function when either:

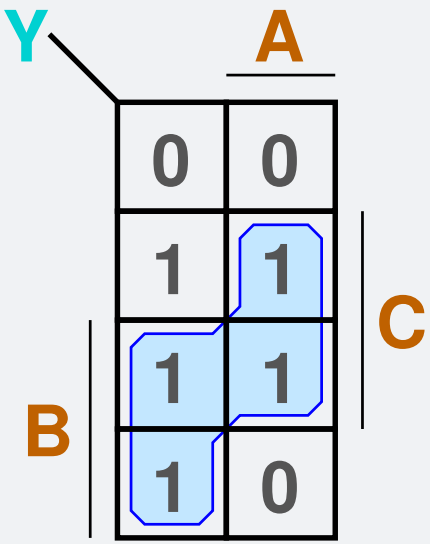
1. All 2^m minterms/maxterms from the group are in the region where **F**=**1**.
2. All 2^m minterms/maxterms from the group are in the region where **F**=**0** ($\overline{\mathbf{F}}$-region).

All m-variables have to be checked for the above rules.
Any violation of the above rules renders the group invalid!

Valid Karnaugh groups

Homework

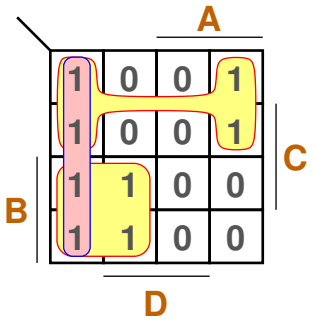
Given the group in the Karnaugh diagram below.



Is this a valid group?

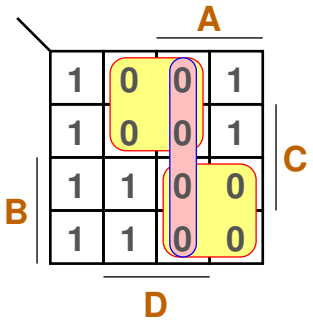
Optimizing Logic Functions with four Variables

$$Y = \overline{A} \cdot B + \overline{B} \cdot \overline{D}$$



- We start again with the biggest *groups* of minterms
- This group is redundant, as it is included in the other two!
- We are done

$$\overline{Y} = A \cdot B + \overline{B} \cdot D$$



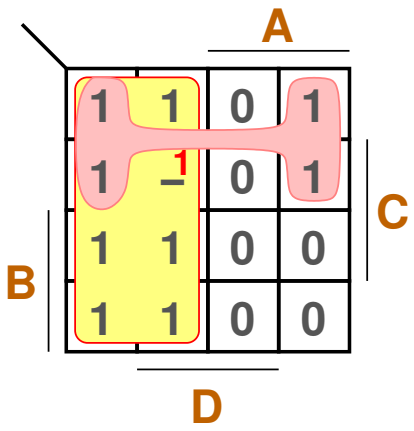
- We can also start with the biggest *groups* of maxterms
- This group is redundant, as it is included in the other two!
- We are done

Homework:

Show with boolean algebra that both functions are identical; what can you observe?

Incomplete Defined Functions

$$Y = \overline{A} + \overline{B} \cdot \overline{D}$$



- We start again with the biggest *groups* of minterms
- When assuming a **1** for the don't care we can find a group of 8!
- We continue until all **1**'s are covered
- We are done

- In mathematics logic functions are always completely defined: for each of the input combinations the function is always either **1** or **0**
- In practice a logic function can have input combinations where we as designer do not mind the outcome: the function is incomplete defined and shows one or multiple don't cares

Lecture 3

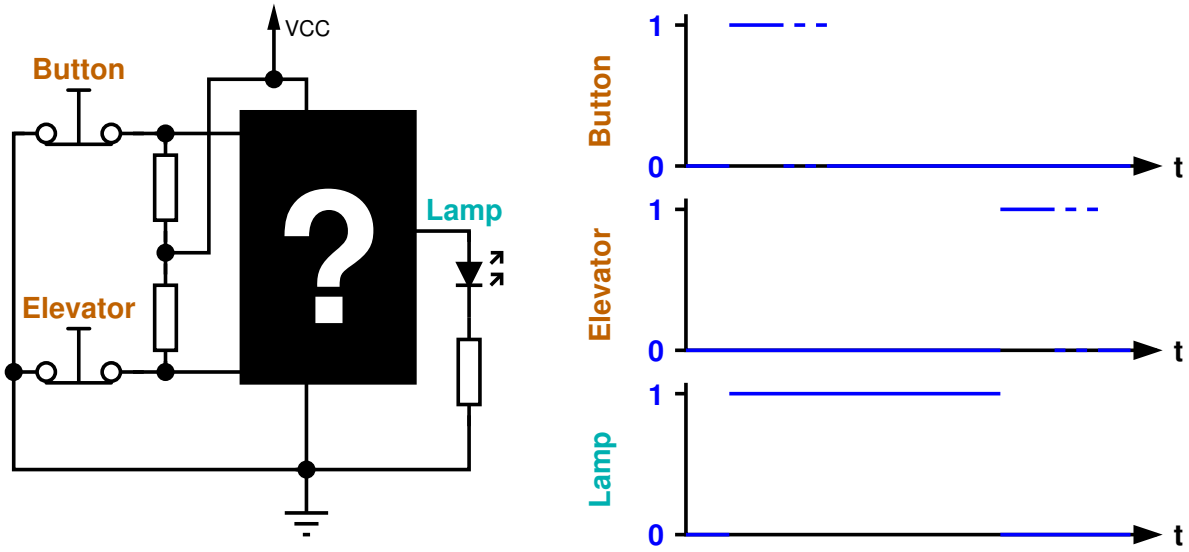
Digital system design

Memory

CS173 - Conception de systèmes numériques
October 2015

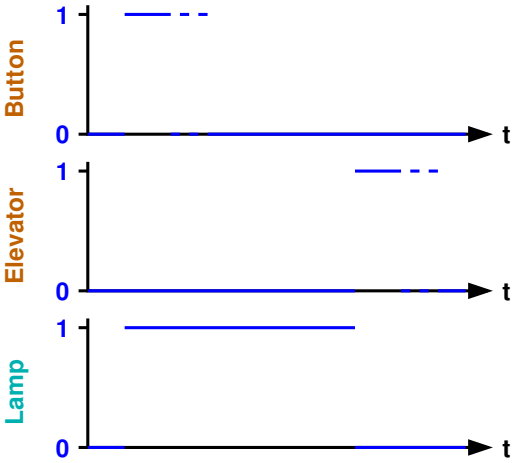
Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Elevator Intro

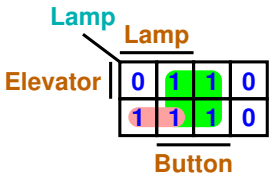


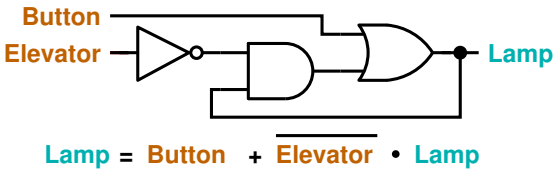
- Schematic of the elevator system with it's initial situation.
- Complete Voltage behavior over time for the given scenario.
- We transformed the scenario to a sequence of logic quantities.

Elevator Logic Function



Lamp	Elevator	Button	Lamp
0	0	0	0
–	–	1	1
1	0	0	1
–	1	0	0





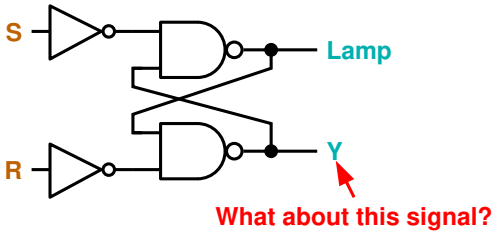
- What is the corresponding logic function and circuit?
- This circuit is called a *Set-Reset Latch (SR-Latch)*.
- This logic function has memory! It is called a *Sequential Logic Function*.
- All logic functions without memory are called *Combinational Logic Functions*

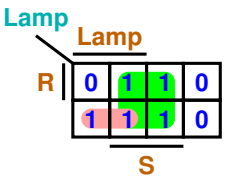
Set-Reset Latch

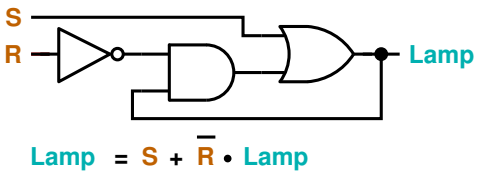
Let's play with De Morgan:

$$Lamp = \overline{\overline{S}} \cdot \overline{\overline{R} \cdot Lamp}$$

R	S	Lamp
0	0	Lamp
–	1	1
1	0	0



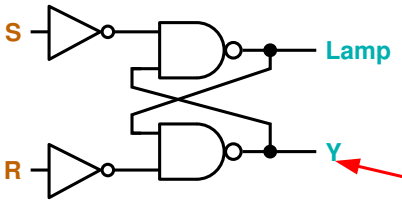




- The truthtable only contains two entries where the 'old' value of the output has an influence.
- We can use an alternate notation to shorten the thruthtable. The entry **Lamp** denotes that the circuit *remembers* its previous value.

Set-Reset Latch

Let's look into the details:



R	S	Lamp	Y
0	0	$\overline{Y}$	$\overline{Lamp}$
0	1	1	0
1	0	0	1
1	1	1	1

It is the inverse of Lamp , except for the situation where:
R = 1 and S = 1

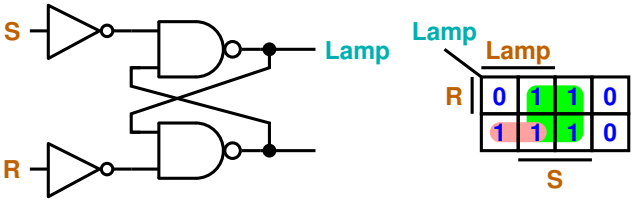
- When R=1 and S=1 then Lamp=1 and Y=1 !
- This situation is oftem referred to as *forbidden*.

Set-Reset Latch

Set–Reset Latch with Set priority:

R	S	Lamp
0	0	Lamp
–	1	1
1	0	0

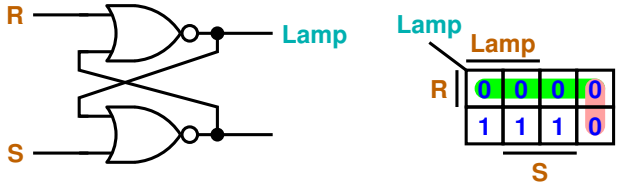
$$Lamp = S + \overline{R} \cdot Lamp$$



Set–Reset Latch with Reset priority:

R	S	Lamp
0	0	Lamp
0	1	1
1	–	0

$$\overline{Lamp} = R + \overline{S} \cdot \overline{Lamp}$$



- The latch we saw thus far is a Set-Reset Latch with a Set Priority. This means that if both **S** and **R** are **1** the Set wins.
- There exists also a Set-Reset Latch with a Reset Priority.

Lecture 4

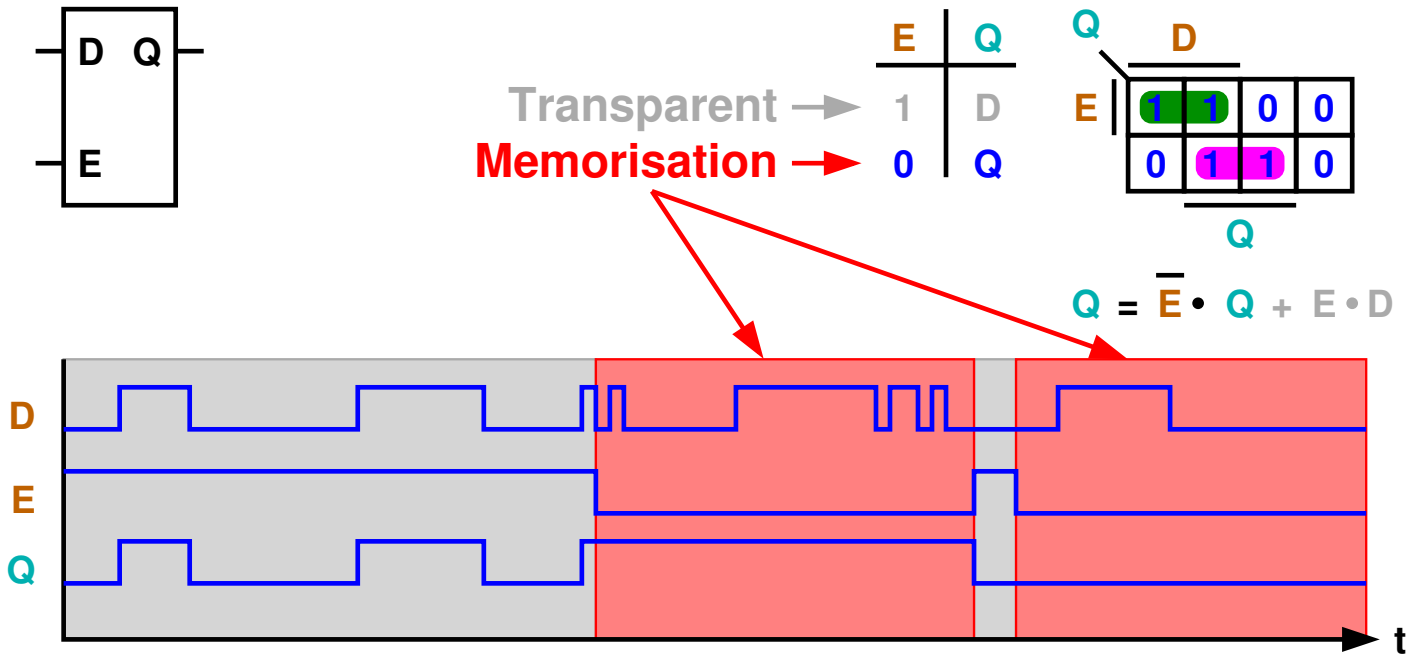
Digital system design

Flipflops and Latches

CS173 - Conception de systèmes numériques
October 2015

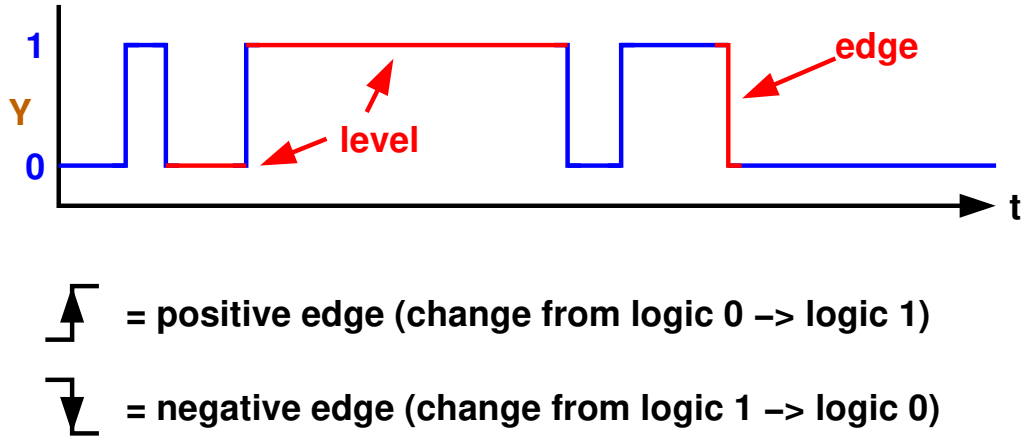
Prof. Dr. Theo Kluter
Bern University of Applied Sciences

D-Latch



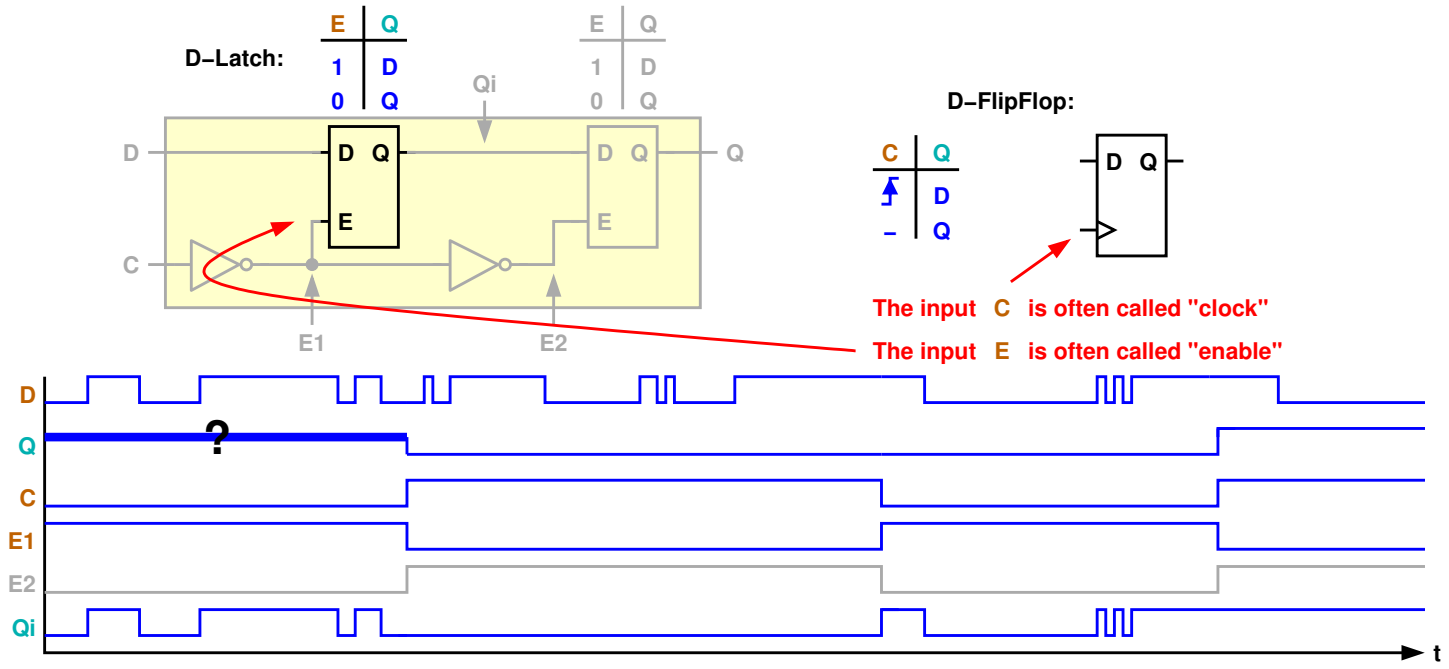
- The circuit retains the last value seen on the input **D** at the moment the input **E** changed from **1** to **0**.
- The D-Latch has its own electric symbol as shown above.
- In the transparent mode (**E**=**1**) the output follows the input **D** ; hence here the behavior of the output can be written as **Q**= **D**. Therefore, we can symplify the truthtable.

Levels and Edges



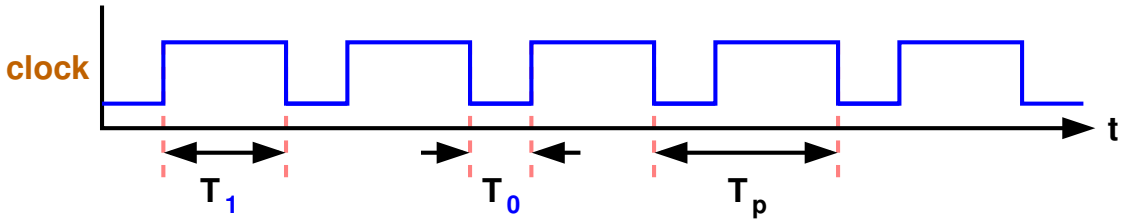
- We have seen that a logic function can have a logic level **0**.
- Or a logic level **1**.
- In time the function can change from a logic level **0** to a logic level **1**, we call this a *positive edge*.
- In time the function can also change from a logic level **1** to a logic level **0**, we call this a *negative edge*.
- Hence a logic function has edges ,
- and levels in time.

D-FlipFlop



What happens if we use 2 Latches and 2 Inverters like in the circuit above?

What is a clock?



- What is a clock?
- A clock is a period $\mathbf{T_1}$ at a logic level **1**.
- a period $\mathbf{T_0}$ at a logic level **0**
- and repeats this pattern each $\mathbf{T_p} = \mathbf{T_1} + \mathbf{T_0}$
- The frequency of this signal is $\mathbf{f}_{\text{clock}} = \frac{1}{\mathbf{T_p}}$,
- And the duty-cycle is defined as $\frac{\mathbf{T_1}}{\mathbf{T_p}}$%.
- At a duty-cycle of 50% we have $\mathbf{T_1} = \mathbf{T_0}$

Lecture 5

Digital system design

Representations

CS173 - Conception de systèmes numériques
August 2014

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

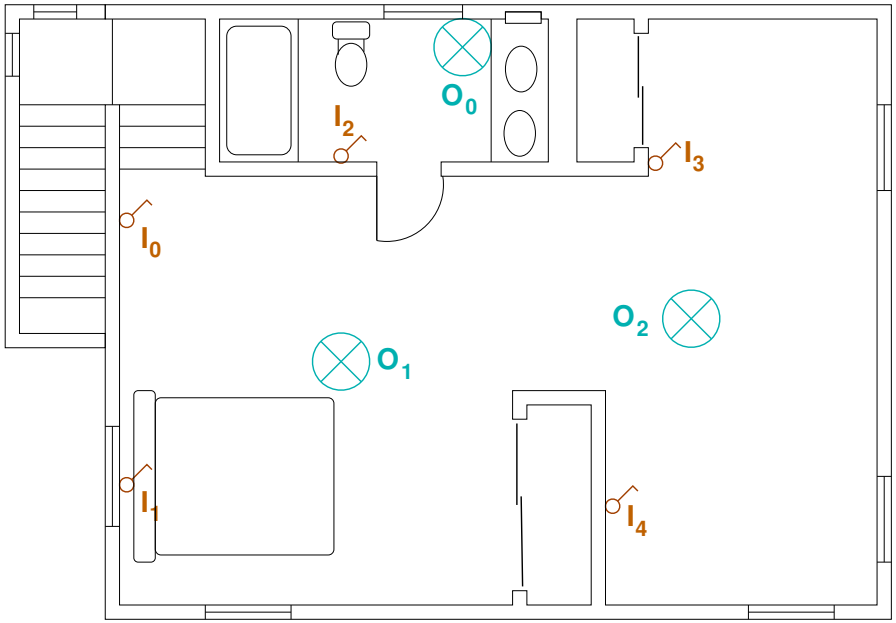
Representations

- We have seen that a circuit can have multiple (n) inputs, e.g. **A**, **B**, **C**, $\cdots$
- We have also seen that a circuit can have multiple (m) outputs, e.g. **X**, **Y**, **Z**, $\cdots$; or formulated differently, the circuit contains m logic functions
- Finally we have seen how to derive the smallest representation of those m logic functions
- The question is now: How do we interpret as designer those inputs and outputs?
- Each of the n inputs and m outputs represent a quantity that can be either **1** or **0**; we call this a **Bit**
- Multiple **Bits** can be grouped together to form one dimensional arrays; the well known groups are:
  - The **Nibble** as a group of 4 **Bits**, e.g. **1011**
  - The **Byte** as a group of 8 **Bits** (2 **Nibbles**), e.g. **10110011**

Ordering

- The groups are nice, but how do we form a **Nibble** from the inputs **A**, **B**, **C**, and **D**?
  1. **ABCD**?
  2. **BCDA**?
  3. **CBDA**?
  4. ...
- We need to define an order to be able to uniquely identify the inputs and outputs
- Let us define $\mathbf{I}_i \ \forall i \in [0..(n-1)]$ as the set of ordered inputs of a circuit, e.g. for n=8 the **Byte** formed by the inputs is: $\mathbf{I}_7\mathbf{I}_6\mathbf{I}_5\mathbf{I}_4\mathbf{I}_3\mathbf{I}_2\mathbf{I}_1\mathbf{I}_0$
- Let us define $\mathbf{O}_j \ \forall j \in [0..(m-1)]$ as the set of ordered outputs of a circuit, e.g. for m=4 the **Nibble** formed by the outputs is: $\mathbf{O}_3\mathbf{O}_2\mathbf{O}_1\mathbf{O}_0$
- The **Bits** $\mathbf{I}_0$ and $\mathbf{O}_0$ are called the **L**east **S**ignificant **B**its
- The **Bits** $\mathbf{I}_{n-1}$ and $\mathbf{O}_{m-1}$ are called the **M**ost **S**ignificant **B**its
- We always write : **MSB**$\cdots$**LSB**

Interpretation



- Here we have a house with 5 light switches (inputs) and 3 lamps (outputs). The interpretation of the inputs and outputs is in this case the switching of the light sources. This functions can be as complex as we want.
- There are various interpretations of the inputs and outputs of a digital system, numbers are one of them

Natural Numbers ($\mathbb{N}$)

- For the remainder of the discussion of the number systems we will use the interpretation of the set of ordered inputs I_i; the interpretation of the set of ordered outputs O_j is identical
- Assuming a byte of ordered inputs, than we could interpret the bit patterns as:

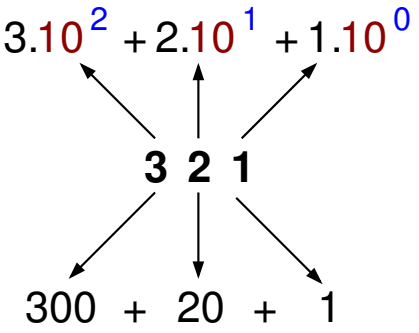
$I_7 I_6 I_5 I_4 I_3 I_2 I_1 I_0$	Decimal
00000000_b	0
00000001_b	1
00000010_b	2
00000100_b	3
00001000_b	4
00010000_b	5
00100000_b	6
01000000_b	7
10000000_b	8

This is the so called one-hot encoding

- However these encodings are not very efficient!

Natural Numbers ($\mathbb{N}$) in place-value notation

More compact and elegant are the positional systems like the decimal system we are used to:



- Each number consists of i digits
- The position of the digit denotes its “importance”
- A digit $d_p \in [0..(B-1)]$ at position $p \in [0..(i-1)]$ given the base B has the “importance” B^p
- The number $N \in \mathbb{N}$ is thereby defined by: $N = \sum_{p=0}^{i-1} d_p \cdot B^p$

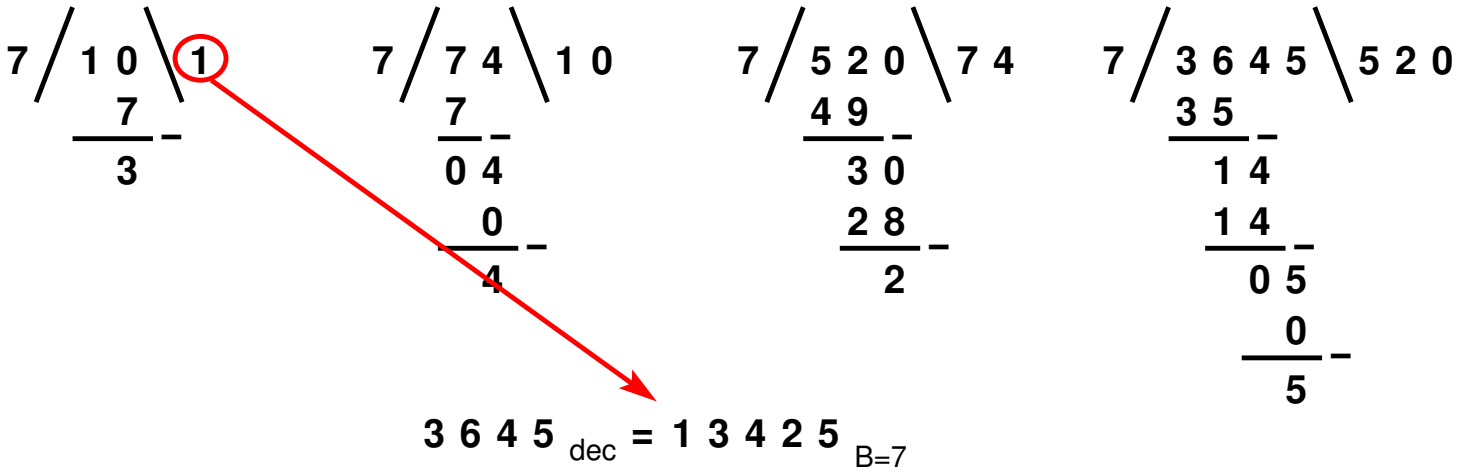
Natural Numbers ($\mathbb{N}$) in place-value notation

Other place-valued numbers:

- The binary number system:
  $I_{11}I_{10}I_9I_8I_7I_6I_5I_4I_3I_2I_1I_0 \equiv \mathbf{101001101001_b}$
  The base $B = 2$, and the digit range is $d_p \in [0, 1]$
  $\mathbf{101001101001_b} = 2^{11} + 2^9 + 2^6 + 2^5 + 2^3 + 2^0 = 2665$
- The octal number system:
  $I_{11}I_{10}I_9 \; I_8I_7I_6 \; I_5I_4I_3 \; I_2I_1I_0 \equiv \mathbf{101\ 001\ 101\ 001_b} \equiv \mathbf{5151_o}$
  The base $B = 8$, and the digit range is $d_p \in [0..7]$
  $\mathbf{5151_o} = 5 \cdot 8^3 + 1 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 = 2665$
- The hexadecimal number system:
  $I_{11}I_{10}I_9I_8 \; I_7I_6I_5I_4 \; I_3I_2I_1I_0 \equiv \mathbf{1010\ 0110\ 1001_b} \equiv \mathbf{0xA69}$
  The base $B = 16$, and the digit range is
  $d_p \in [0..9, A, B, C, D, E, F]$
  $\mathbf{0xA69} = 10 \cdot 16^2 + 6 \cdot 16^1 + 9 \cdot 16^0 = 2665$

Natural Numbers ($\mathbb{N}$) in place-value notation

How to transform 3645_{dec} to a place-value number with base $B = 7$?



- We start the division
- The rest is the digit at position 0, e.g. $3645 = 520 \cdot 7^1 + 5 \cdot 7^0$
- As $520 \notin [0..6]$ we have to continue the division
- Now $1 \in [0..6]$ we are done. Let's check:
  $3645 = 2401 + 1029 + 196 + 14 + 5$

The Addition

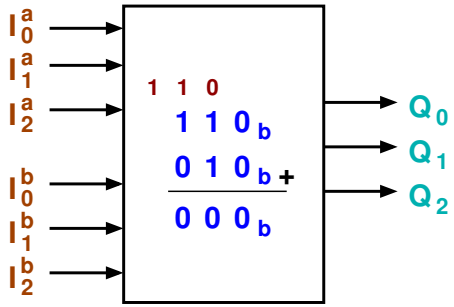
How do we add two place-valued numbers?

```
Carries:
0 1 0
  1 2 3
  5 9 2  +
---------
  7 1 5
```

- Assume we want to determine $N = 123 + 592$
- We start adding the digits at position $p = 0$, e.g. $S_0 = d_0^1 + d_0^2 = 3 + 2 = 5$, the resulting digit $d_0^r = \begin{cases} S_0 & S_0 < B \\ S_0 - B & S_0 \geq B \end{cases}$, and the carry $c_1 = \begin{cases} 0 & S_0 < B \\ 1 & S_0 \geq B \end{cases}$
- We continue to position $p = 1$; at all positions $p \geq 0$ the sum equals to $S_p = d_p^1 + d_p^2 + c_p$, note: $c_0 = 0$!
- What happens if carry $c_3 = 1$ (for example if $N = 110_b + 010_b$)?

The Addition; Limitations of Digital Systems

Assume following binary adder:



- We have a circuit that adds the numbers a and b
- In case $a = 110_b$ and $b = 010_b$ something strange happens, as $N = 110_b + 010_b \neq 000_b$!
- We call this an overflow situation
- An overflow can occur as any digital system has only a limited number of k-bits ($k = 3$ for this example)
- Given this limitation of k-bits, a digital system has a range in which it can represent numbers, e.g. for $N \in \mathbb{N}$ and $k = 3$ the representation range is $N \in [0..(2^k - 1)] \equiv N \in [0..7]$

The Substraction

Similar to the addition we can also substract two place-valued numbers:

Borrows:
0 1 0

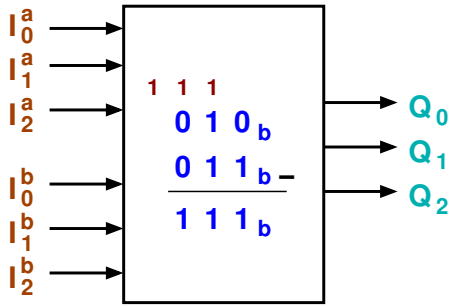
$$\begin{array}{r} 7\,1\,5 \\ 5\,9\,2 \;- \\ \hline 1\,2\,3 \end{array}$$

- Assume we want to determine $N = 715 - 592$
- We start at position $p = 0$, e.g. $S_0 = d_0^1 - d_0^2 = 5 - 2 = 3$, the resulting digit $d_0^r = \begin{cases} d_0^1 - d_0^2 & d_0^1 \geq d_0^2 \\ (B + d_0^1) - d_0^2 & d_0^1 < d_0^2 \end{cases}$, and the borrow $b_1 = \begin{cases} 0 & d_0^1 \geq d_0^2 \\ 1 & d_0^1 < d_0^2 \end{cases}$
- We continue to position $p = 1$; at all positions $p \geq 0$ the substraction equals to $S_p = \begin{cases} d_p^1 - d_p^2 - b_p & d_p^1 \geq d_p^2 + b_p \\ (B + d_p^1) - d_p^2 - b_p & d_p^1 < d_p^2 + b_p \end{cases}$

  Note: $b_0 = 0$!

The Substraction; No Negative Numbers

Assume following binary substractor:



- We have a circuit that substract the numbers a and b
- In case $a = 010_b$ and $b = 011_b$ something strange happens, as $N = 010_b - 011_b \neq 111_b$!
- We call this an underflow situation

Lecture 6

Digital system design

Representations

CS173 - Conception de systèmes numériques
August 2014

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Integers ($\mathbb{Z}$) sign-and-magnitude representation

- The set of integers $\mathbb{Z}$ contains the Natural Numbers $\mathbb{N} \subset \mathbb{Z}$ and their negatives ($-0, -1, -2, -3, -4, ...$)
- In the decimal system we use the sign-and-magnitude notation, e.g. **+**4 and **-**4 where the "**+**" is often omitted
- Important: the sign-and-magnitude notation has two 0's, namely +0 AND -0!
- In the binary system, we do not have a **+** and **-**, we only have **0** and **1**
- However, by the introduction of the **MSB** as sign-bit we can create a binary form of the sign-and-magnitude representation by interpreting **0**=**+** and **1**=**-**:
  $I_3\ I_2I_1I_0 \equiv \mathbf{0\ 010_b} \equiv +2 \equiv$ 0x2!
  $I_3\ I_2I_1I_0 \equiv \mathbf{1\ 010_b} \equiv -2 \equiv$ 0xA!

Addition and Substraction with Integers ($\mathbb{Z}$)

- A decimal substraction **Z=A-B** we normally rewrite to a decimal addition **Z=A+C** by forming the negate of the substractor, e.g. **C = -B**, this can be easily accomplished by changing the sign in a sign-and-magnitude system. For a binary sign-and-magnitude interpretation this operation would be an inversion of the sign-bit.
- In a sign-and-magnitude representation we can write this addition as: $s_z|Z| = s_a|A| + s_c|C| \Rightarrow -|-3| = -|-4| + (+|1|)$
- The addition is "complex":

s_a	s_c	Condition:	s_z	$\|Z\|$	Examples:
$+$	$+$	none	$+$	$\|A\| + \|C\|$	$1 + 2 = 3$
$-$	$-$	none	$-$	$\|A\| + \|C\|$	$-1 + (-2) = -3$
$+$	$-$	$\|A\| \geq \|C\|$	$+$	$\|A\| - \|C\|$	$2 + (-1) = 1$
		$\|A\| < \|C\|$	$-$	$\|C\| - \|A\|$	$1 + (-2) = -1$
$-$	$+$	$\|A\| \geq \|C\|$	$-$	$\|A\| - \|C\|$	$-2 + 1 = -1$
		$\|A\| < \|C\|$	$+$	$\|C\| - \|A\|$	$-1 + 2 = 1$

Integers ($\mathbb{Z}$) one's-complement representation

Addition and substraction in the sign-and-magnitude representation requires an adder and a substractor, can't we find another representation that requires less "complexity"?

- What if instead of inverting only the sign bit to represent a negative number, we invert (one's-complement) all bits:

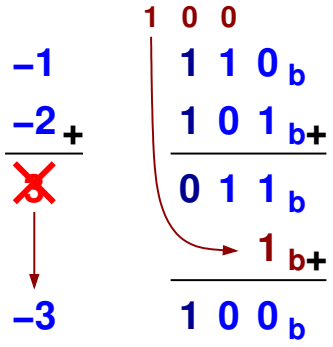
Dec.:	Inputs:	sign-and-magnitude:	1's-complement:
$+3$	$I_2\ I_1 I_0$	$\mathbf{0\ 11_b}$	$\mathbf{0\ 11_b}$
$+2$	$I_2\ I_1 I_0$	$\mathbf{0\ 10_b}$	$\mathbf{0\ 10_b}$
$+1$	$I_2\ I_1 I_0$	$\mathbf{0\ 01_b}$	$\mathbf{0\ 01_b}$
$+0$	$I_2\ I_1 I_0$	$\mathbf{0\ 00_b}$	$\mathbf{0\ 00_b}$
-0	$I_2\ I_1 I_0$	$\mathbf{1\ 00_b}$	$\mathbf{1\ 11_b}$
-1	$I_2\ I_1 I_0$	$\mathbf{1\ 01_b}$	$\mathbf{1\ 10_b}$
-2	$I_2\ I_1 I_0$	$\mathbf{1\ 10_b}$	$\mathbf{1\ 01_b}$
-3	$I_2\ I_1 I_0$	$\mathbf{1\ 11_b}$	$\mathbf{1\ 00_b}$

- NOTE: one's-complement representation also has 2 representations for 0!

Integers ($\mathbb{Z}$) one's-complement representation

Addition and subtracting one's-complement numbers is easier than doing the same in a sign-and-magnitude representation:

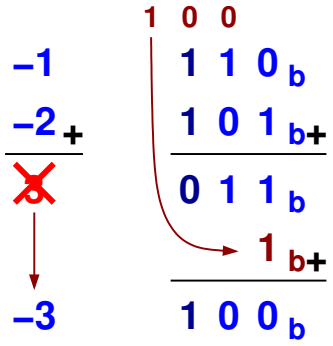
- A substraction **Z=A-B** is again transformed into an addition **Z=A+C** by forming the one's-complement **C=-B** (by inverting [complementing] all the bits of **B**).
- The addition **Z=A+C** can now be performed by a binary addition as for $N \in \mathbb{N}$, with the difference that carry c_k has to be added at the **LSB** position; this is the so called end-around carry addition:



One's–complement:

0 1 1 b +3	1 0 0 b –3
0 1 0 b +2	1 0 1 b –2
0 0 1 b +1	1 1 0 b –1
0 0 0 b +0	1 1 1 b –0

Integers ($\mathbb{Z}$) one's-complement representation

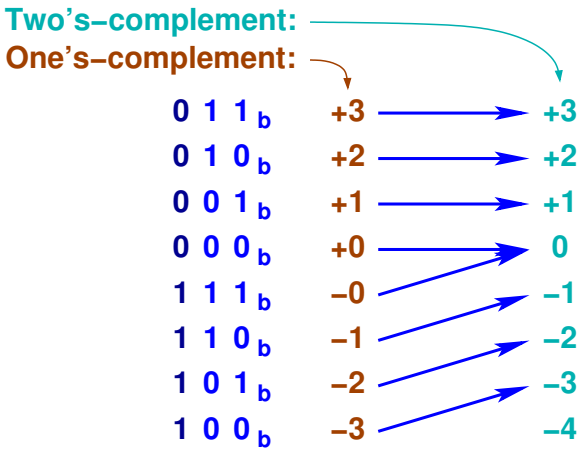


One's–complement:

0 1 1 b +3	1 0 0 b –3
0 1 0 b +2	1 0 1 b –2
0 0 1 b +1	1 1 0 b –1
0 0 0 b +0	1 1 1 b –0

- One's-complement is great, as for both the addition and substraction of integers we **only** require inverters and an end-around carry adder
- However, an end-around carry adder is slow!
- Observation: The end carry c_n is only **1** when one of the variables **A** or **C** in **Z=A+C** is negative!
- Why not add **1** to all negative numbers?

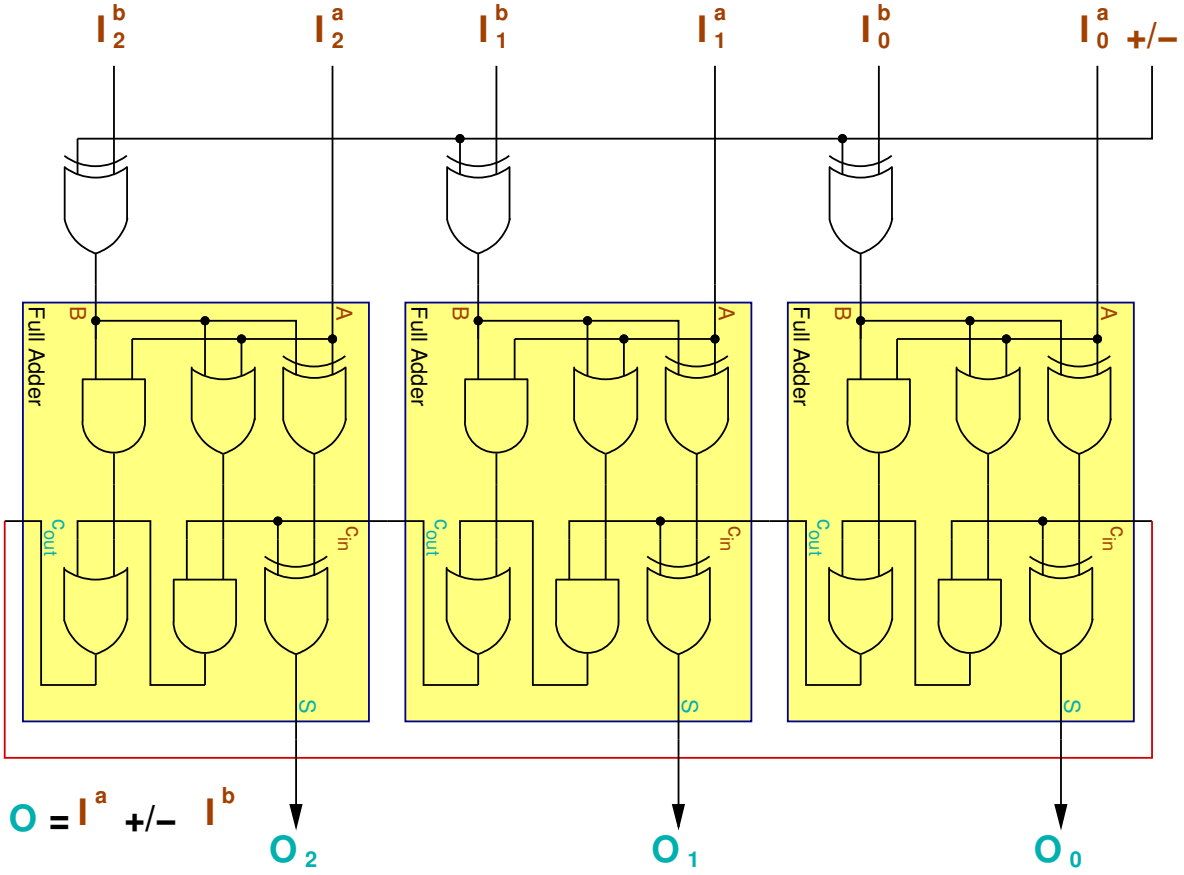
Integers ($\mathbb{Z}$) two's-complement representation



- The positive numbers in two's-complement notation have same bit encodings as the other presented number interpretation systems
- The **+0** and **-0** map on the same bit pattern in the two's-complement notation resulting in a single **0**!
- The remaining bit pattern is mapped to **-4**

Integers ($\mathbb{Z}$) one's versus two's complement

One's complement addition/substraction:



- One's complement is slow due to the end-around carry
- Two's complement is faster, and has only one **0**!

Integers ($\mathbb{Z}$) Excess-N representation

- The two's complement representation is commonly used to represent integers in for example Processors, Digital Signal Processors, and others
- However, there exists another important representation: the *Excess-N* representation, also called the biased representation
- The *Excess-N* representation uses a pre-specified number N as biasing value.
- The decimal number $-N$ is represented by all **0**'s
- Example: The three bit Excess-3 representation

$1\,1\,1_b$	4	$0\,1\,1_b$	0
$1\,1\,0_b$	3	$0\,1\,0_b$	–1
$1\,0\,1_b$	2	$0\,0\,1_b$	–2
$1\,0\,0_b$	1	$0\,0\,0_b$	–3

- The *Excess-N* representation is mainly used in Analog to Digital converters and the IEEE floating-point standard
- Performing calculations in the *Excess-N* representation is only in a few cases simple and out of the scope of this course!

Integers ($\mathbb{Z}$) Overflow and Underflow

- Did we solve the overflow and underflow problem?
- No, as we still have **only** *k*-bits to represent numbers!
- We just changed the ranges:

Representation:	Number:	Decimal range:
Binary	$x \in \mathbb{N}$	$0 \leq x \leq 2^k - 1$
Sign-and-Magn.	$x \in \mathbb{Z}$	$-(2^{k-1} - 1) \leq x \leq 2^{k-1} - 1$
One's-compl.	$x \in \mathbb{Z}$	$-(2^{k-1} - 1) \leq x \leq 2^{k-1} - 1$
Two's-compl.	$x \in \mathbb{Z}$	$-(2^{k-1}) \leq x \leq 2^{k-1} - 1$
Excess-N	$x \in \mathbb{Z}$	$-N \leq x \leq 2^k - (N+1)$

- NOTE: The two's complement and excess-N representations have an asymmetric range and do not have a $+0$ and -0; they have a single 0!
- In case the decimal calculation results in a value x outside the given ranges we phase an overflow or underflow problem!
- The detection of overflow and underflow is outside the scope of this course!

Summary ($\mathbb{N}$ and $\mathbb{Z}$)

- All number representations are noted as a series of bits from **MSB**...**LSB**: 10110010_b
- For a number representation of k-bits we can represent the decimal equivalents like:

Representation:	Number:	Decimal range:
Binary	$x \in \mathbb{N}$	$0 \leq x \leq 2^k - 1$
Sign-and-Magn.	$x \in \mathbb{Z}$	$-(2^{k-1} - 1) \leq x \leq 2^{k-1} - 1$
One's-compl.	$x \in \mathbb{Z}$	$-(2^{k-1} - 1) \leq x \leq 2^{k-1} - 1$
Two's-compl.	$x \in \mathbb{Z}$	$-(2^{k-1}) \leq x \leq 2^{k-1} - 1$
Excess-N	$x \in \mathbb{Z}$	$-N \leq x \leq 2^k - (N+1)$

- To convert a positive (negative) integer into a negative (positive) one we have to complement the representation as:

Representation:	Complement:
Sign-and-Magn.	Invert sign bit
One's-compl.	Invert all bits
Two's-compl.	Invert all bits and add 1
Excess-N	lookup in table

Multiplication

Remember:
The classic decimal pen and paper multiplication:

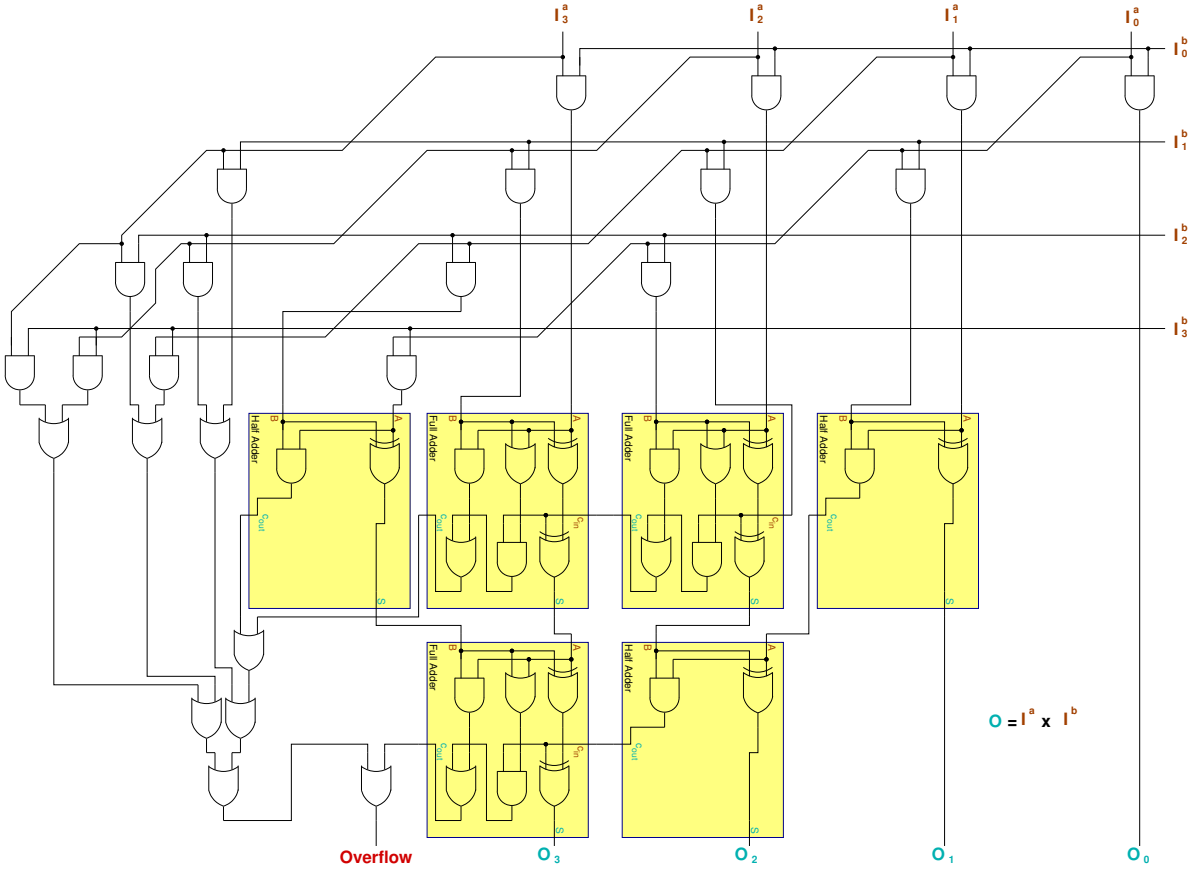
```
  2 4
  1 8   x
-------
0 1 0
  3 2
1 6 0
  4 0
2 0 0   +
-------
4 3 2
```

Binary multiplication:

```
    0 1 0 0 b
    0 0 1 1 b  x
-------------
0 0 0 0
  0 1 0 0 b
0 1 0 0 0 b
0 0 0 0 0 0 b
0 0 0 0 0 0 0 b  +
-----------------
1 1 0 0 b
```

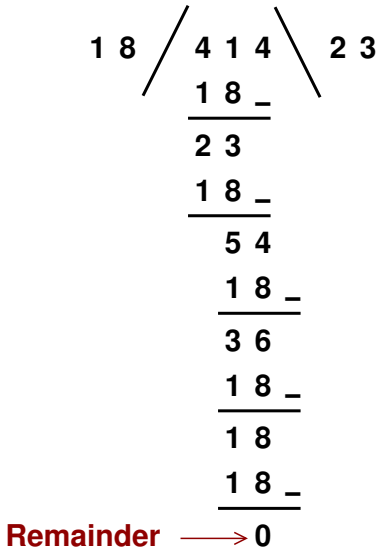
- Binary multiplication is performed similar, but much simpler!
- As: $1_b \times 1_b = 1_b$ and $1_b \times 0_b = 0_b$ (**Note:** This represent an **AND** gate!)

Multiplication

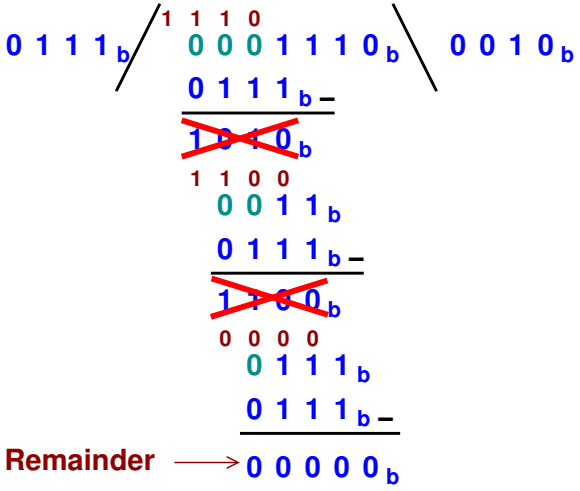


Division

Remember:
The classic decimal pen and paper division:

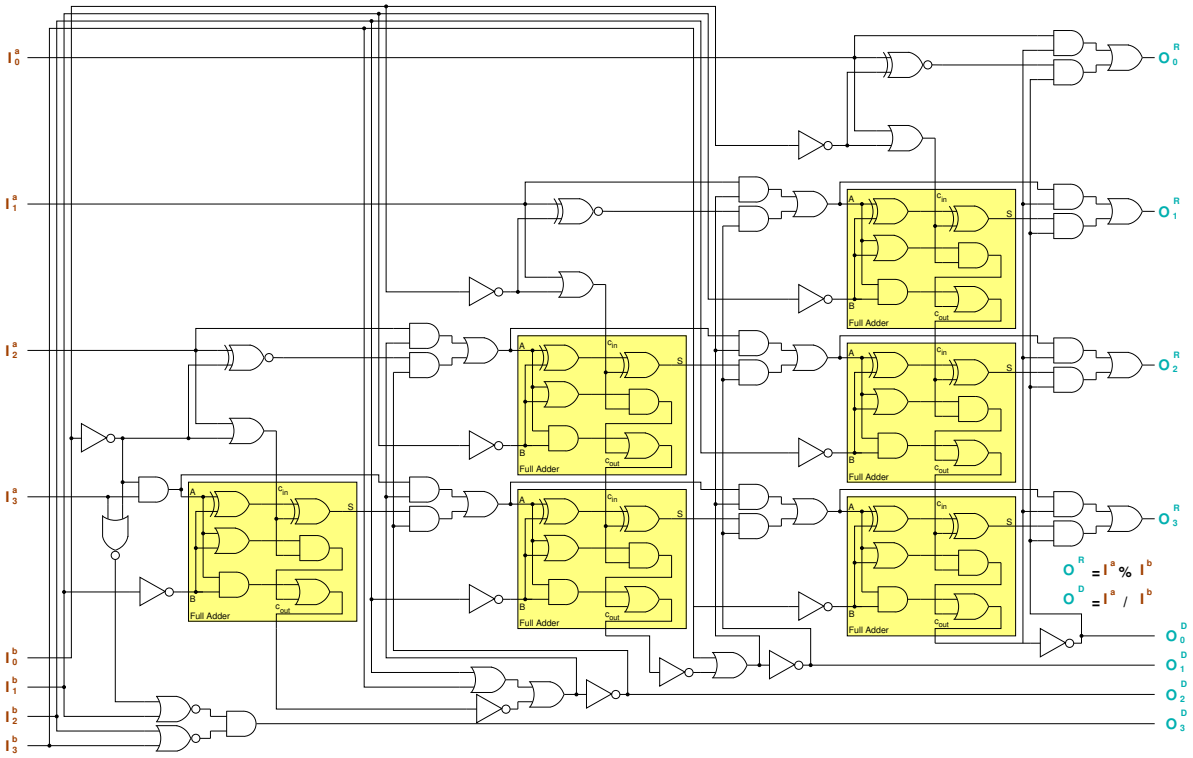


Binary division:



- Binary division is performed identical, but simpler!
- As for each resulting digit we only have to substract once!

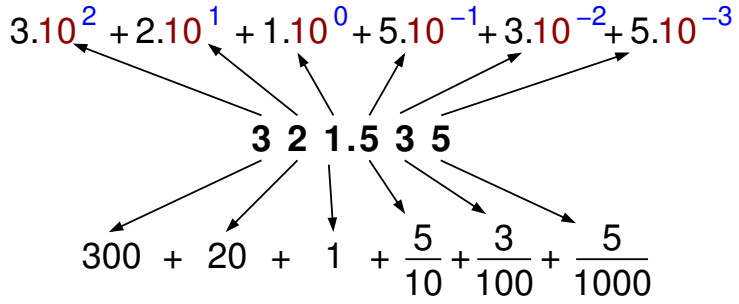
Division



- How can we represent the number when **Remainder** $\neq 0$, e.g. $\frac{3}{2} = 1\frac{1}{2}$?

Fixed Point representation ($\mathbb{R}$)

- The set on Real numbers $\mathbb{R}$ contains all integers $\mathbb{Z} \subset \mathbb{R}$ and all fractions $\frac{m}{n}$
- Real numbers are normally noted in the decimal system by placing a decimal point: **321.535**
- We call this notation the fixed point representation of a real number.
- Recall: Place-and-value notation



- Therefore: $N = \sum_{p=0}^{i-1} d_p \cdot B^p$ becomes $R = \sum_{p=-j}^{i-1} d_p \cdot B^p$
- We can do the same for all place-and-value notated numbers of any base B!

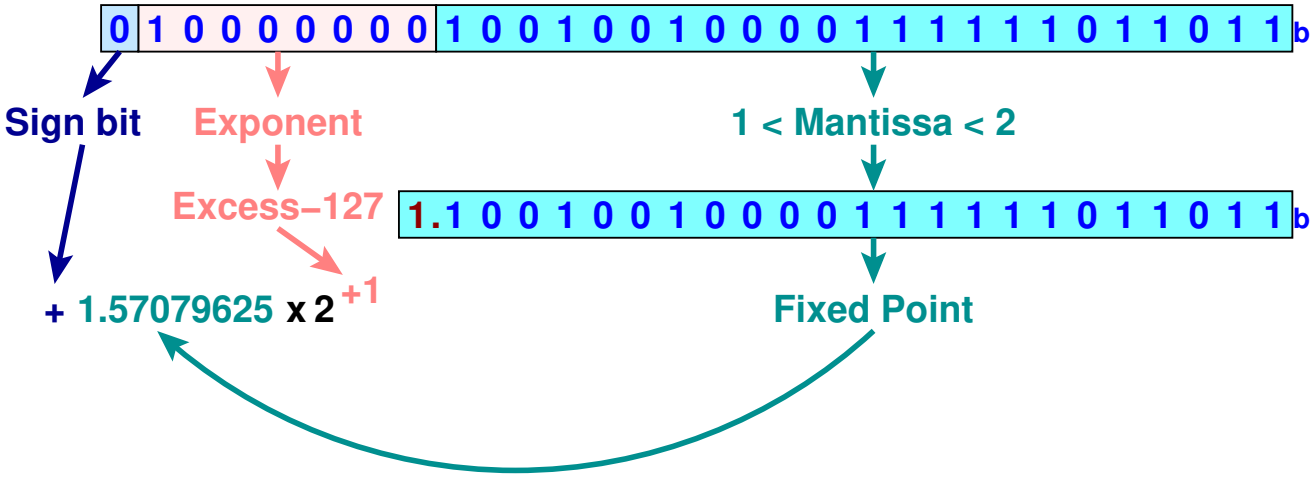
Floating Point representation ($\mathbb{R}$)

- What about very big and very small numbers?
- A big number like **845294294967295.85321** we normally write as 0.84529×10^{15} or $0.84529\,\mathrm{E}15$
- And small numbers line **0.00000000000000012** we similarly write as $0.1275245 \times 10^{-15}$ or $0.1275245\,\mathrm{E}-15$
- We call this the floating point representation, where a number R is represented by a mantissa and exponent. The mantissa is in a fixed-point representation, whilst the exponent is in a place-and-value representation:

$$R = 0.1275245 \times 10^{-15}$$
$$\textbf{mantissa} = 0.1275245 \quad \text{where:} \quad -1 < \textbf{mantissa} < 1$$
$$\textbf{exponent} = -15 \quad \text{where:} \quad -\infty < \textbf{exponent} < \infty$$

Floating Point representation ($\mathbb{R}$)

- We can also represent floating point numbers using a set of bits (IEEE 754 standard)
- This is for example used in every computer to represent the types float, double, etc.
- For a float often a single precision floating point representation is used:



Lecture 7

Digital system design

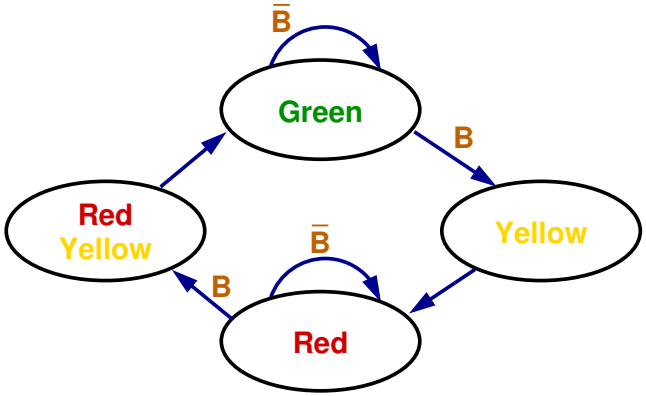
Finite State Machines

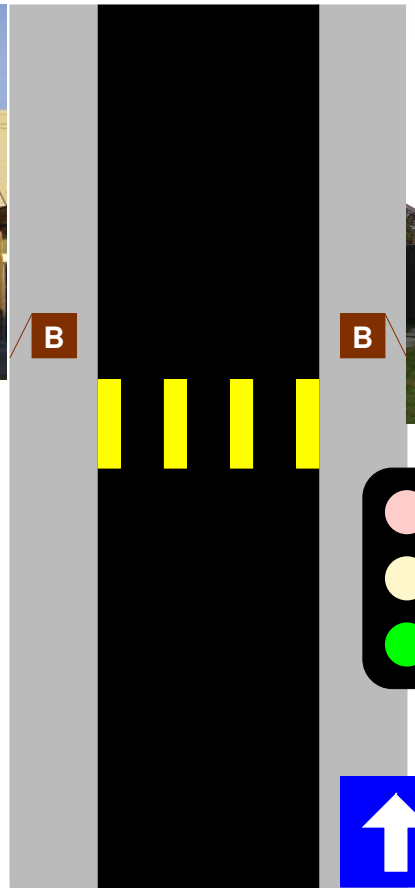
CS173 - Conception de systèmes numériques
November 2016

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Introduction

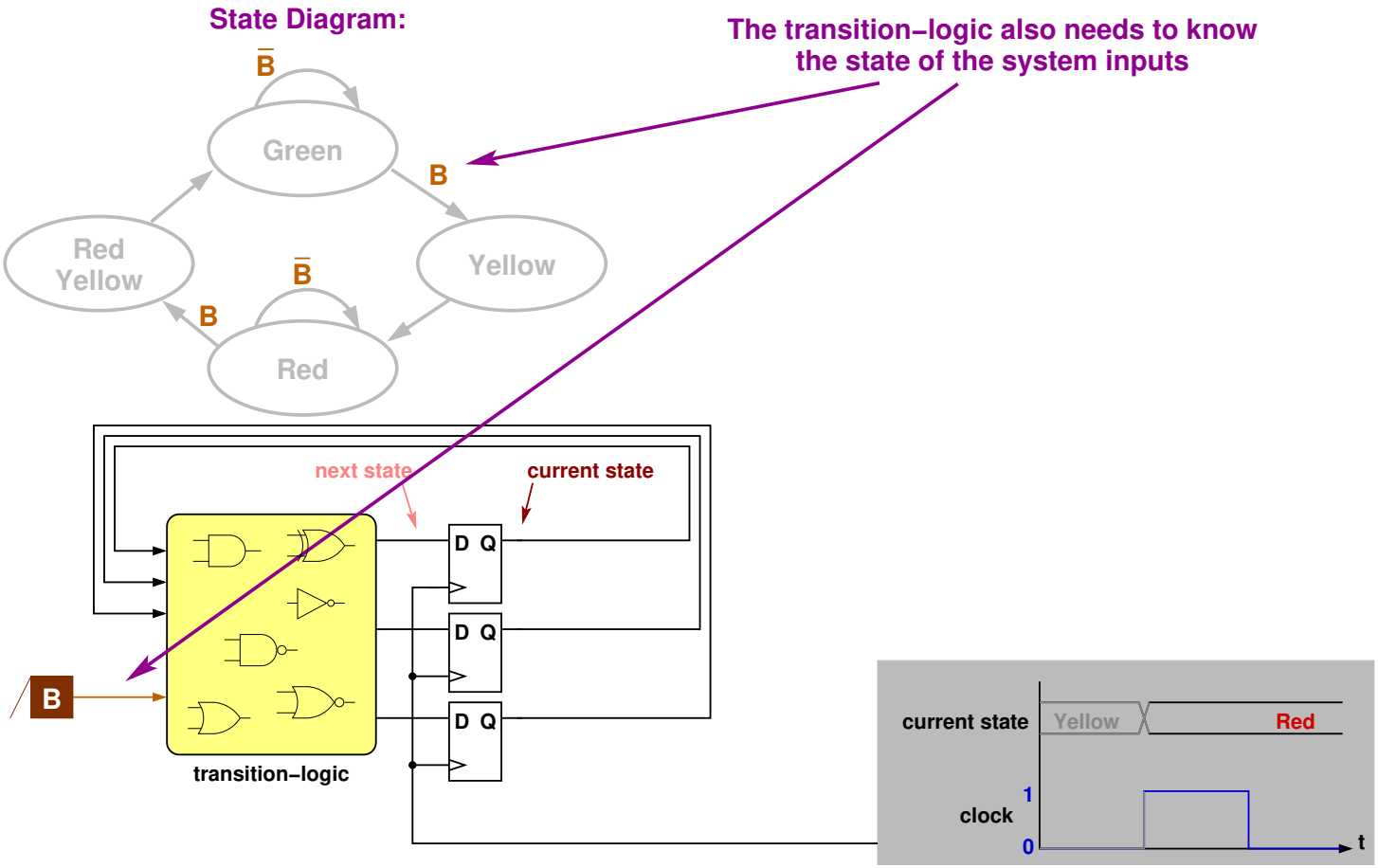
Sequence



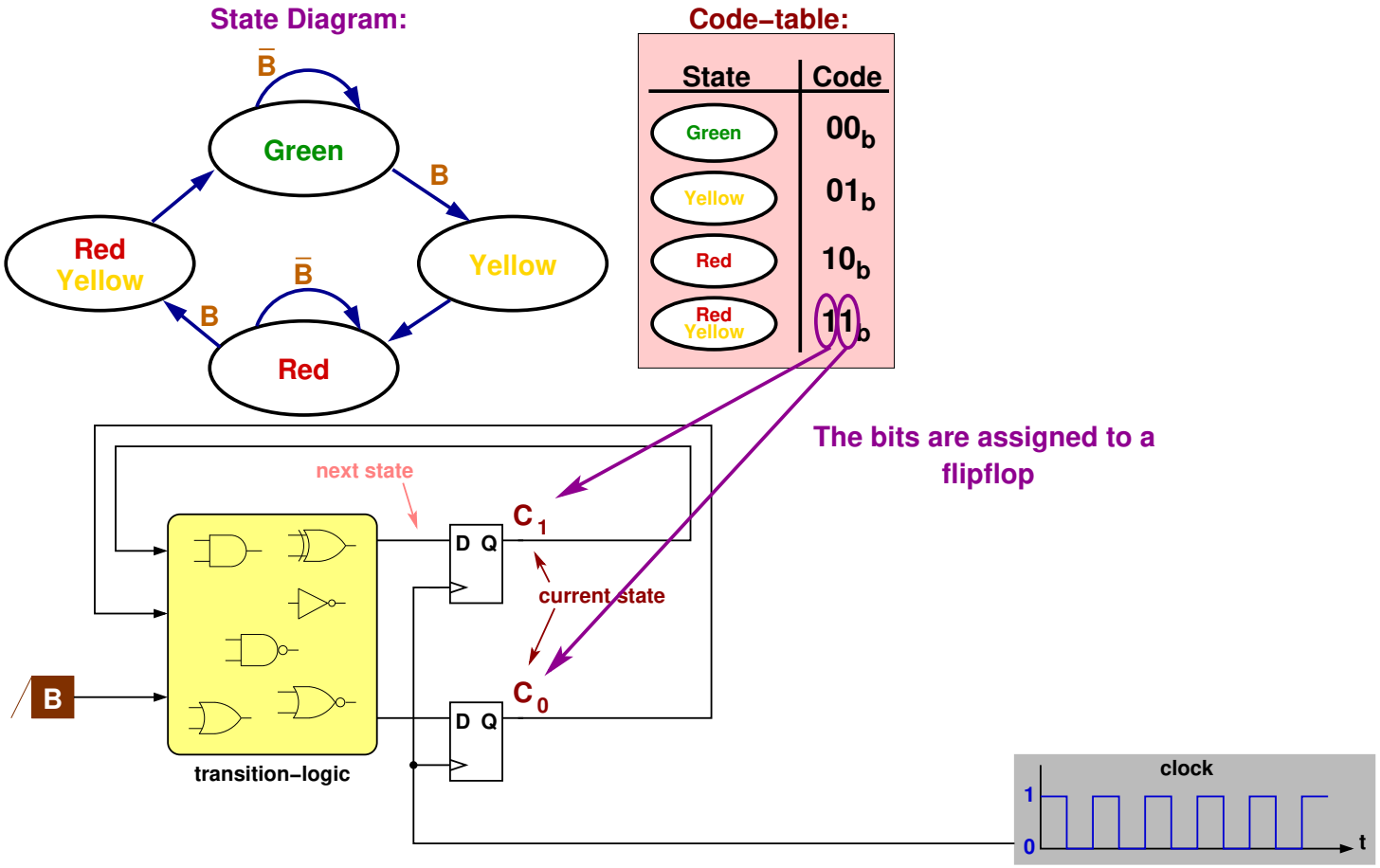


This sequence repeats itself

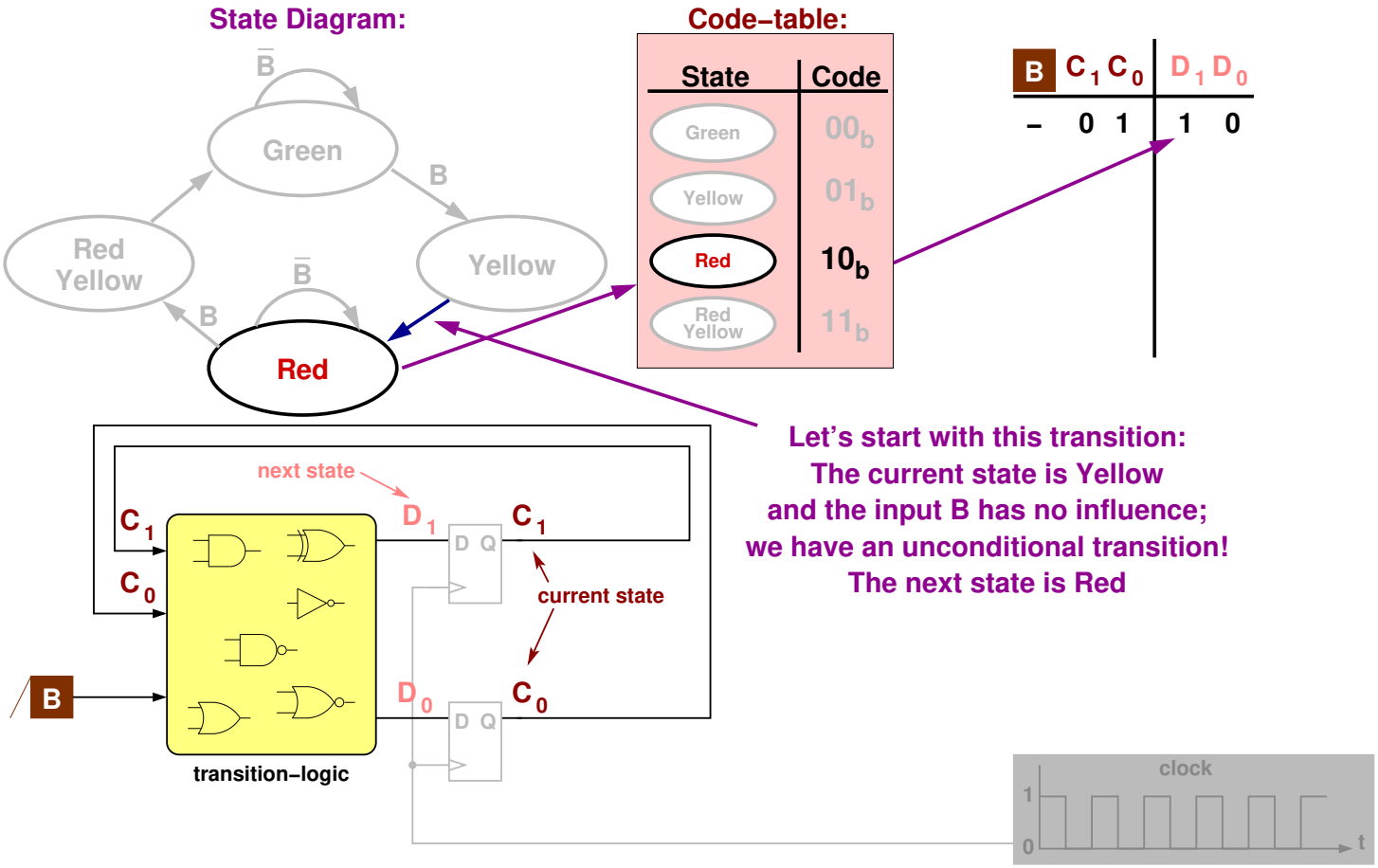
Basic Structure



Medvedev FSM

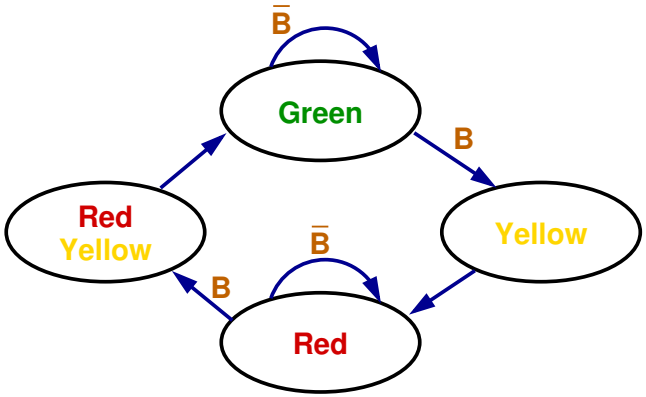


Medvedev FSM



Medvedev FSM

State Diagram:

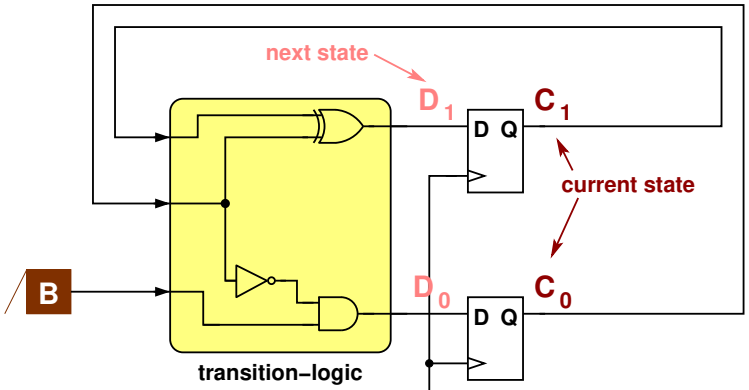


Code–table:

State	Code
Green	00_b
Yellow	01_b
Red	10_b
Red Yellow	11_b

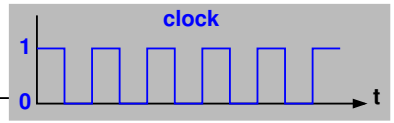
Transition–table:

B	C_1	C_0	D_1	D_0
–	0	1	1	0
–	1	1	0	0
0	1	0	1	0
1	1	0	1	1
0	0	0	0	0
1	0	0	0	1

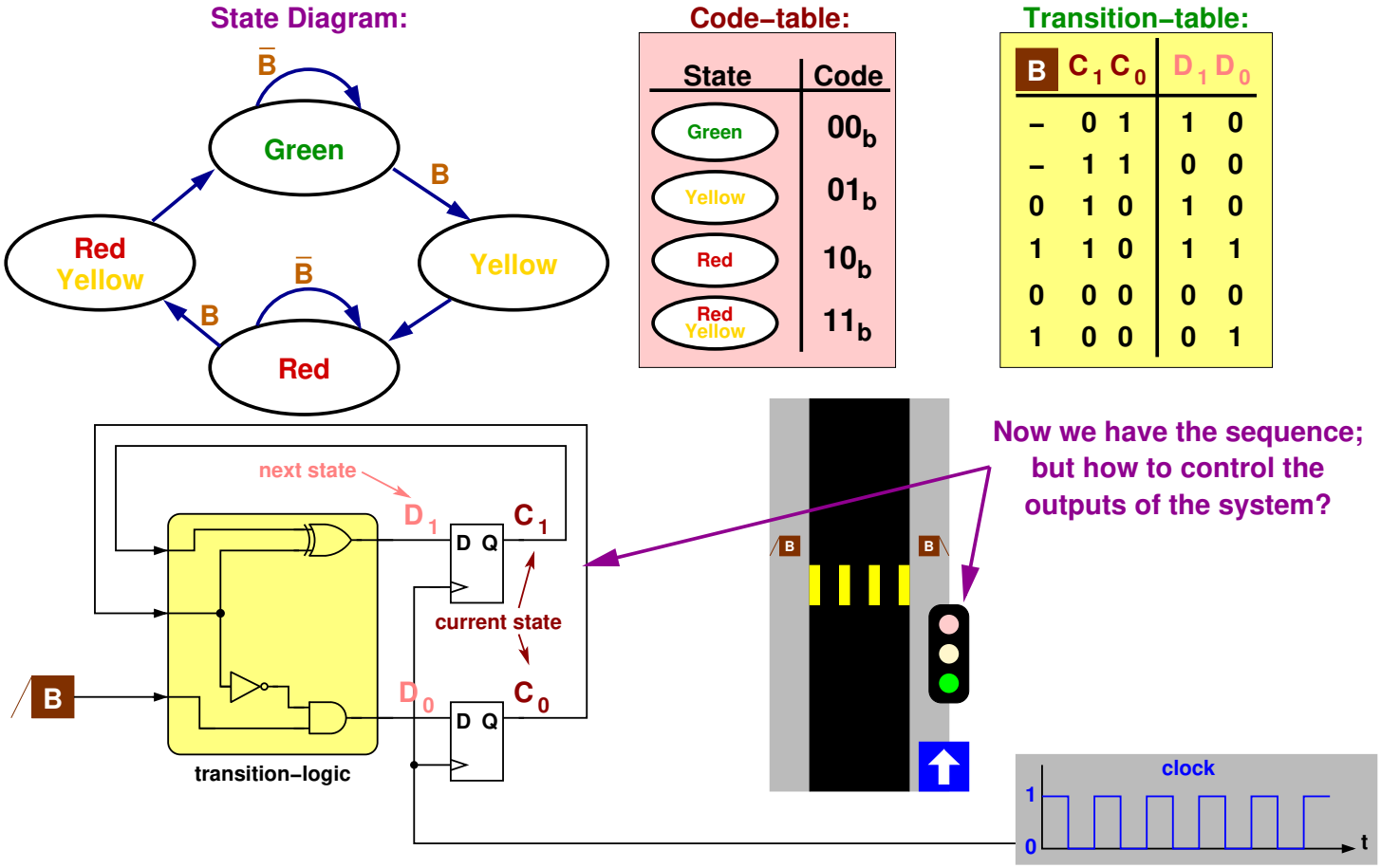


This type of Finite State Machine (FSM) is called a:

Medvedev FSM

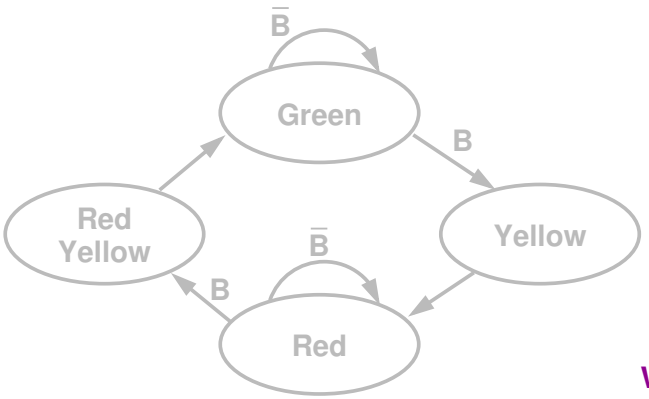


Moore FSM


State Diagram:
B̄
Green
B
Red
Yellow
Yellow
B̄
B
Red
Code–table:
State
Code
Green
00b
Yellow
01b
Red
10b
Red
Yellow
11b
Transition–table:
B C1 C0 D1 D0
– 0 1 1 0
– 1 1 0 0
0 1 0 1 0
1 1 0 1 1
0 0 0 0 0
1 0 0 0 1
next state
D1
C1
D Q
current state
B
D0
C0
D Q
transition–logic
Now we have the sequence; but how to control the outputs of the system?
clock
1
0
t

Moore FSM

State Diagram:



Code–table:

State	Code
Green	00_b
Yellow	01_b
Red	10_b
Red Yellow	11_b

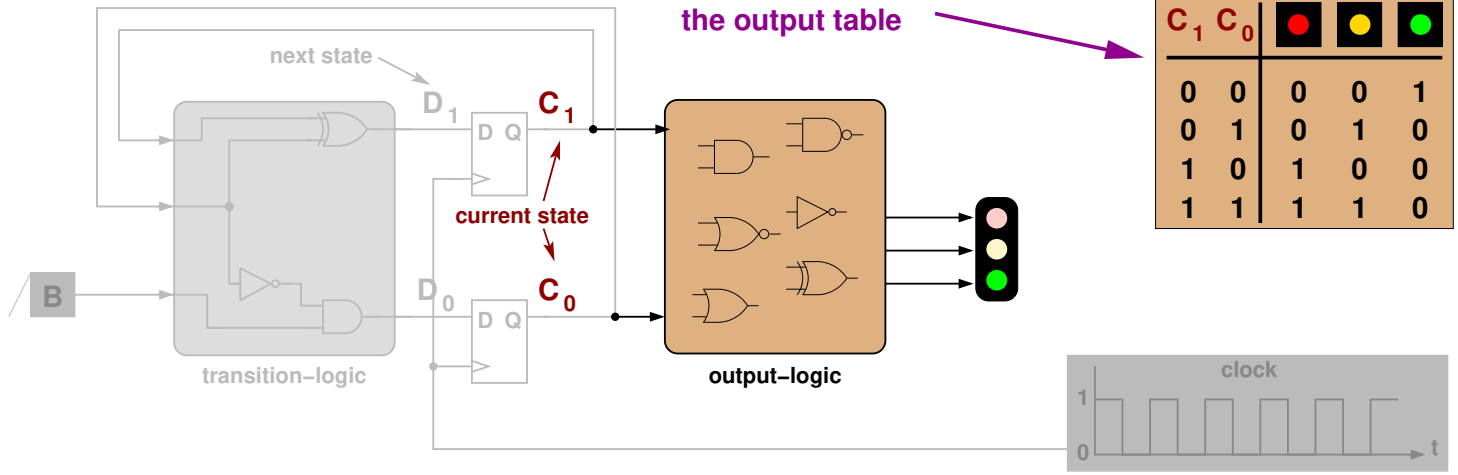
Transition–table:

B	C_1	C_0	D_1	D_0
–	0	1	1	0
–	1	1	0	0
0	1	0	1	0
1	1	0	1	1
0	0	0	0	0
1	0	0	0	1

We call this truthtable the output table

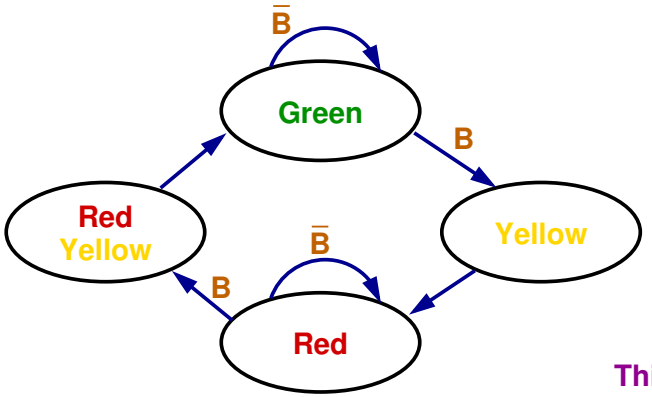
Output–table:

C_1	C_0			
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	1	1	0



Moore FSM

State Diagram:



Code–table:

State	Code
Green	00_b
Yellow	01_b
Red	10_b
Red Yellow	11_b

Transition–table:

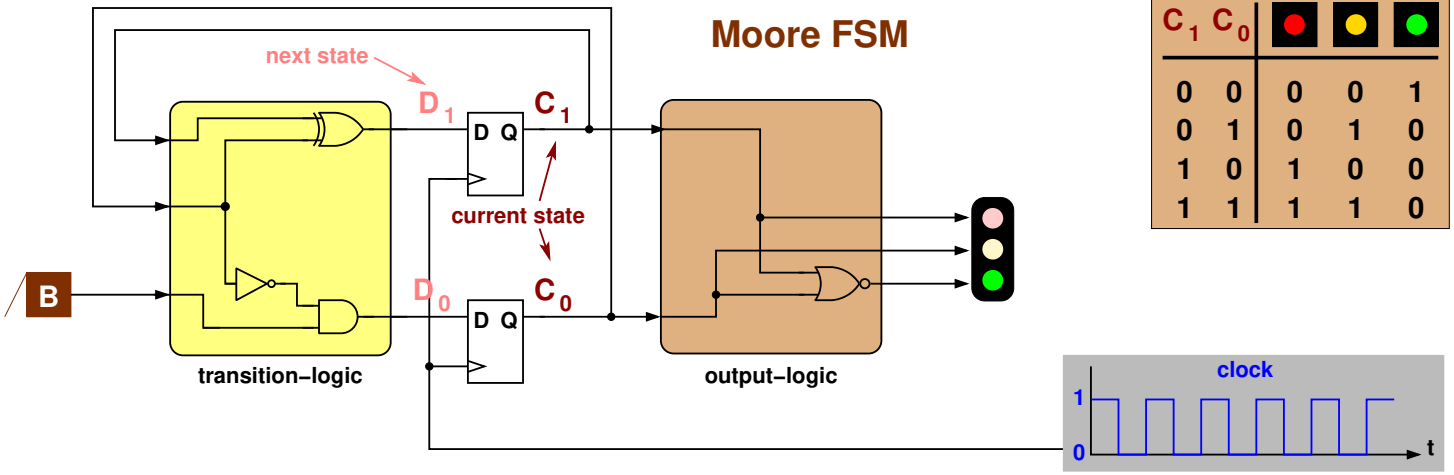
B	C_1	C_0	D_1	D_0
–	0	1	1	0
–	1	1	0	0
0	1	0	1	0
1	1	0	1	1
0	0	0	0	0
1	0	0	0	1

This type of FSM is called a:

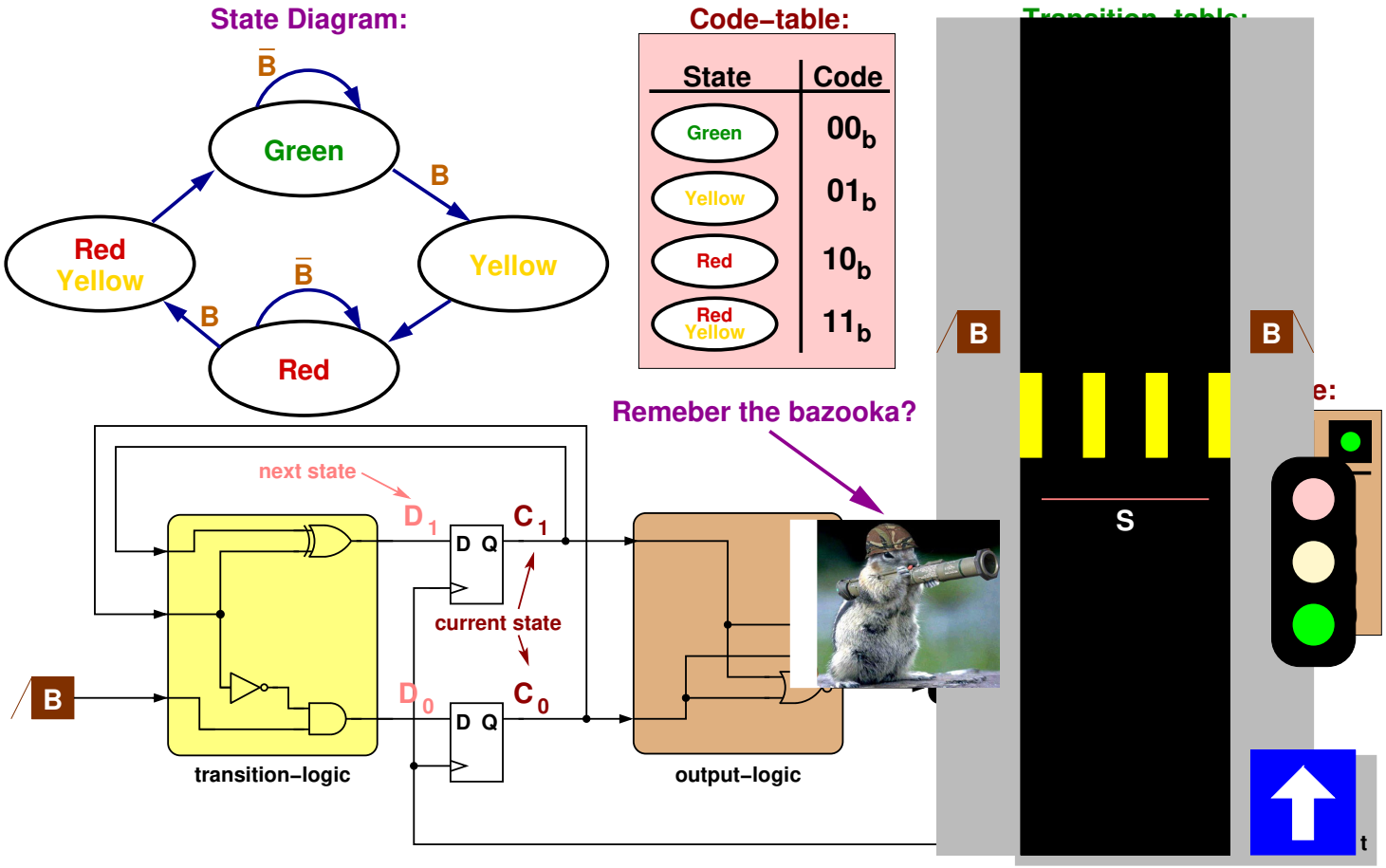
Moore FSM

Output–table:

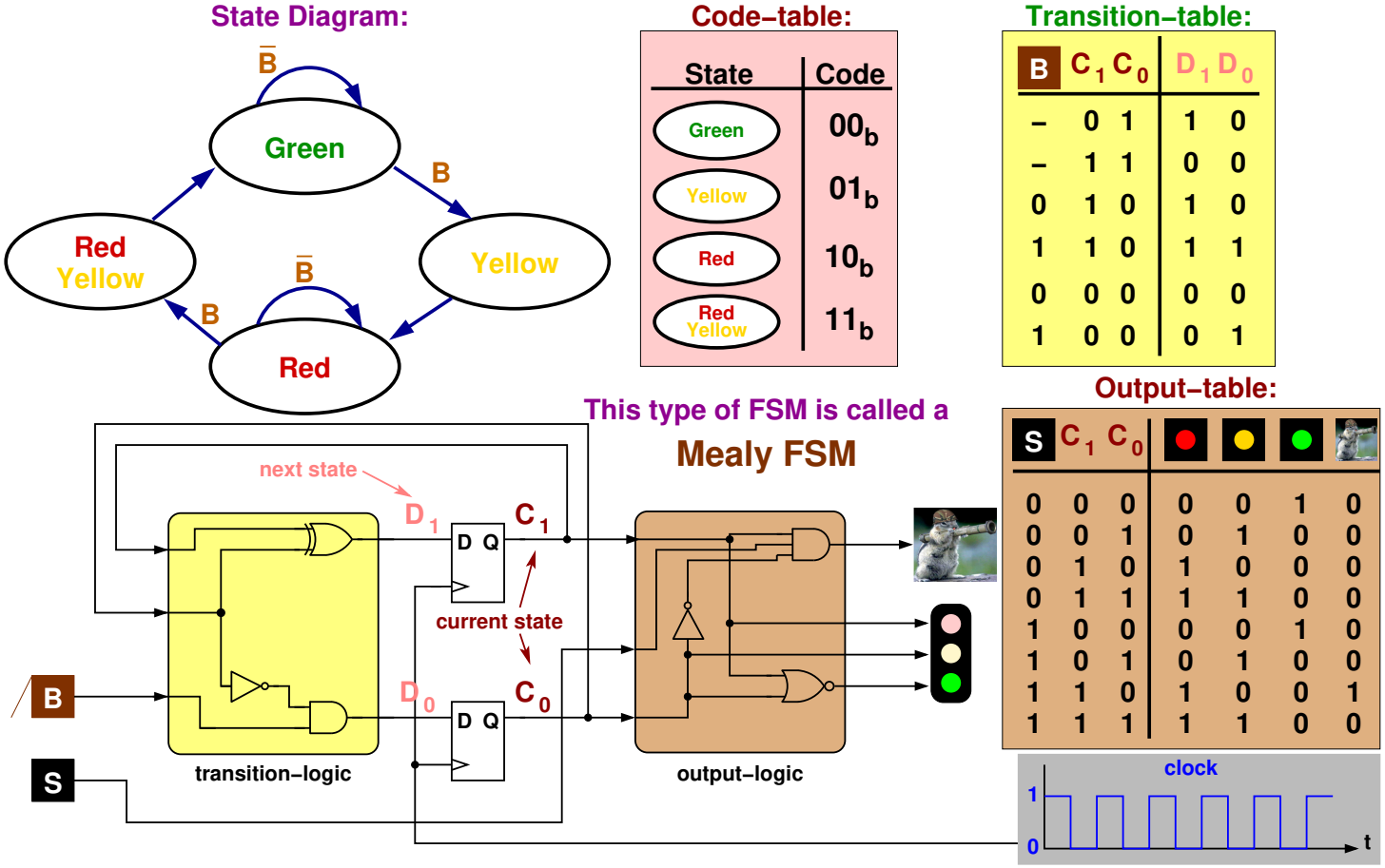
C_1	C_0			
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	1	1	0



Mealy FSM


State Diagram:
Green
Yellow
Red
Red Yellow
B̄
B
Code–table:
State
Code
Green
00b
Yellow
01b
Red
10b
Red Yellow
11b
Remeber the bazooka?
next state
D1
C1
D Q
current state
D0
C0
B
transition–logic
output–logic
S
B
B

Mealy FSM


State Diagram:
B̄
Green
B
Red
Yellow
Yellow
B̄
B
Red
Code–table:
State
Code
Green
00b
Yellow
01b
Red
10b
Red
Yellow
11b
Transition–table:
B C1 C0 D1 D0
– 0 1 1 0
– 1 1 0 0
0 1 0 1 0
1 1 0 1 1
0 0 0 0 0
1 0 0 0 1
This type of FSM is called a
Mealy FSM
Output–table:
S C1 C0
0 0 0 0 0 1 0
0 0 1 0 1 0 0
0 1 0 1 0 0 0
0 1 1 1 1 0 0
1 0 0 0 0 1 0
1 0 1 0 1 0 0
1 1 0 1 0 0 1
1 1 1 1 1 0 0
next state
D1
C1
D Q
current state
D0
C0
D Q
B
S
transition–logic
output–logic
clock
1
0
t

Lecture 8

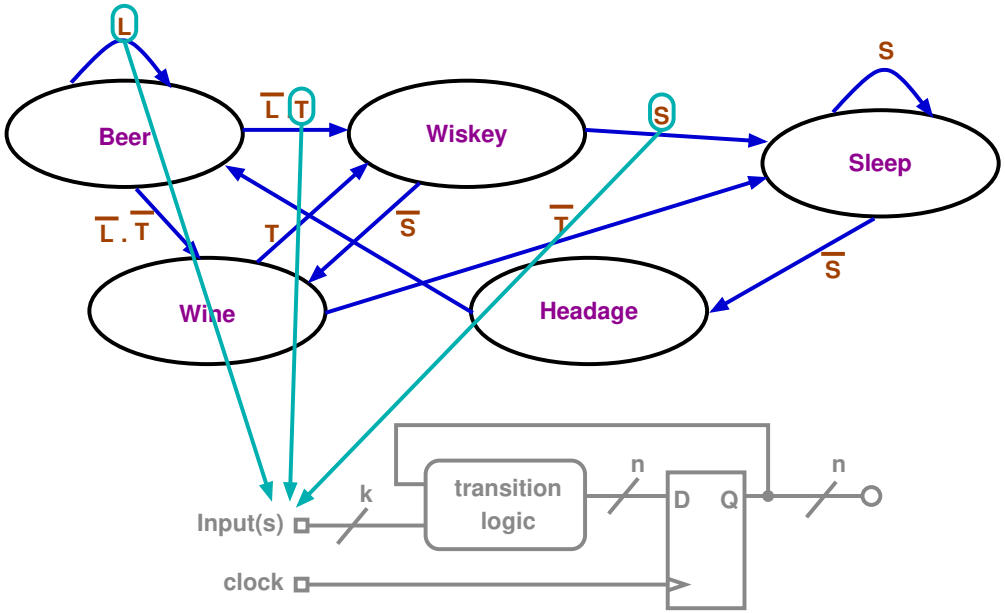
Digital system design

Finite State Machines (2)

CS173 - Conception de systèmes numériques
February 2017

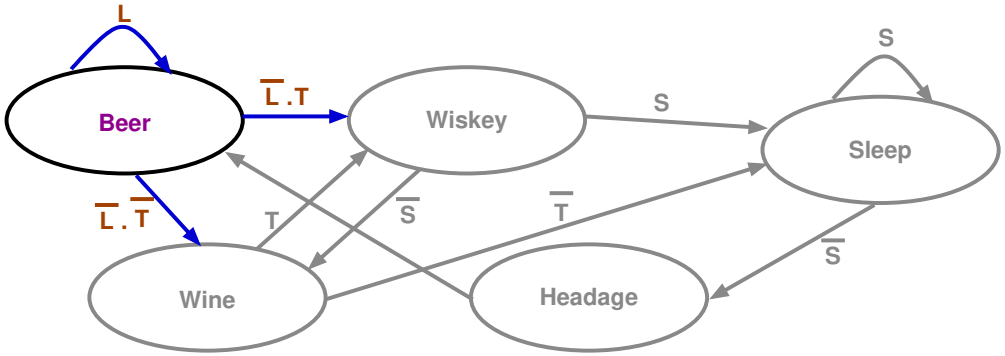
Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Howto design an FSM: step 1



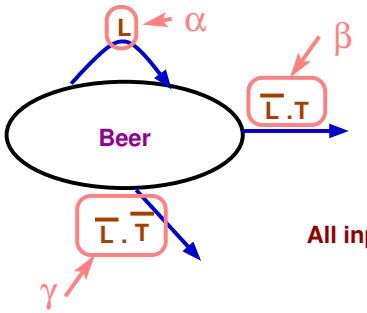
- We take as example a Medvedev state machine
- First we define the states of the state machine
- Than we define the different transitions and their conditions
- Please note that the conditions are the inputs of the system
- Also note that the clock is not used as condition

Howto design an FSM: step 2



- To be able to realize this state-diagram we have to make sure that:
- For each of the states that they are:
- **Complete**
- and
- **Consistent**

Completeness



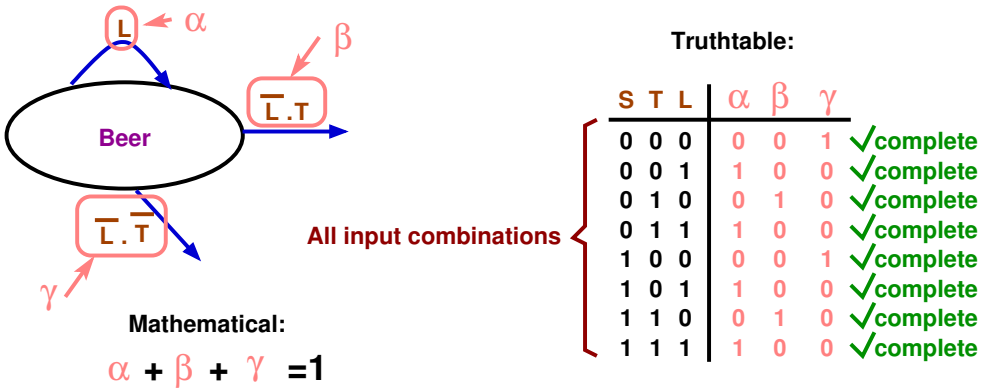
All input combinations

Truthtable:

S	T	L	α	β	γ	
0	0	0	0	0	1	✓complete
0	0	1	1	0	0	✓complete
0	1	0	0	1	0	✓complete
0	1	1	1	0	0	✓complete
1	0	0	0	0	1	✓complete
1	0	1	1	0	0	✓complete
1	1	0	0	1	0	✓complete
1	1	1	1	0	0	✓complete

- A state is complete when *for all possibilities of the inputs* there is *at least one transition* active.
- Let's name the transition conditions.
- We can check by using a truthtable.
- We can fill in the transition conditions.
- We can now check on completeness.

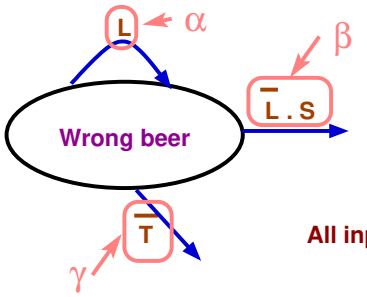
Completeness



- A state is complete when *for all possibilities of the inputs* there is *at least one transition* active.
- We can also do the check with boolean algebra
- By performing the OR of the transition conditions:

$$\begin{aligned}\alpha + \beta + \gamma &= L + \overline{L} \cdot T + \overline{L} \cdot \overline{T} \\ &= L + \overline{L} \cdot (T + \overline{T}) \\ &= L + \overline{L} \\ &= 1\end{aligned}$$

Example of an incomplete state



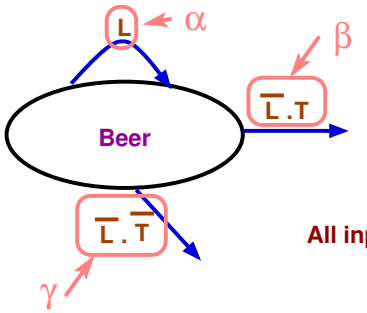
All input combinations

Truthtable:

S	T	L	α	β	γ	
0	0	0	0	0	1	✓complete
0	0	1	1	0	1	✓complete
0	1	0	0	0	0	✗incomplete
0	1	1	1	0	0	✓complete
1	0	0	0	1	1	✓complete
1	0	1	1	0	1	✓complete
1	1	0	0	1	0	✓complete
1	1	1	1	0	0	✓complete

- A state is complete when *for all possibilities of the inputs* there is *at least one transition* active.
- Let's look at this state.
- We start again with the truthtable method.
- And check all possibilities.
- We found already 1 entry that is incomplete, we can stop here!

Consistency



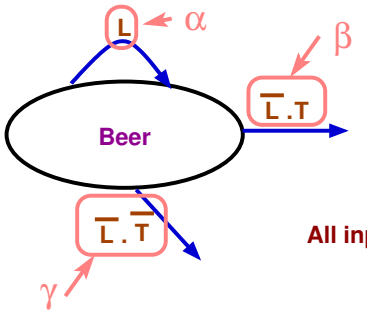
All input combinations

Truthtable:

S	T	L	α	β	γ	
0	0	0	0	0	1	√consistent
0	0	1	1	0	0	√consistent
0	1	0	0	1	0	√consistent
0	1	1	1	0	0	√consistent
1	0	0	0	0	1	√consistent
1	0	1	1	0	0	√consistent
1	1	0	0	1	0	√consistent
1	1	1	1	0	0	√consistent

- A state is consistent when *for all possibilities of the inputs* there is *at most one transition* active.
- Let's name the transition conditions.
- We can check by using a truthtable.
- We can now check on consistency.

Consistency



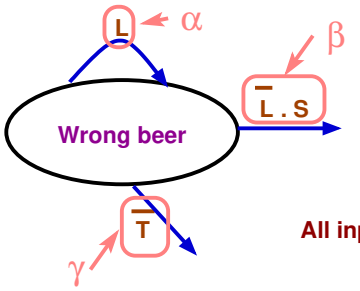
All input combinations

Truthtable:

S	T	L	α	β	γ	
0	0	0	0	0	1	✓consistent
0	0	1	1	0	0	✓consistent
0	1	0	0	1	0	✓consistent
0	1	1	1	0	0	✓consistent
1	0	0	0	0	1	✓consistent
1	0	1	1	0	0	✓consistent
1	1	0	0	1	0	✓consistent
1	1	1	1	0	0	✓consistent

- A state is consistent when *for all possibilities of the inputs* there is *at most one transition* active.
- We can also use the boolean algebra:
  1) $\alpha \cdot \beta = (L) \cdot (\overline{L} \cdot T) = L \cdot \overline{L} \cdot T = 0$
  2) $\alpha \cdot \gamma = (L) \cdot (\overline{L} \cdot \overline{T}) = L \cdot \overline{L} \cdot \overline{T} = 0$
  3) $\beta \cdot \gamma = (\overline{L} \cdot T) \cdot (\overline{L} \cdot \overline{T}) = \overline{L} \cdot T \cdot \overline{T} = 0$

Example of an inconsistent state



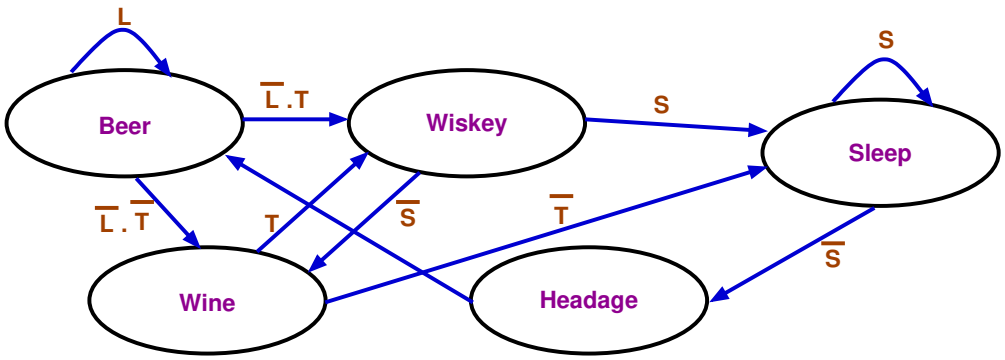
All input combinations

Truthtable:

S	T	L	α	β	γ	
0	0	0	0	0	1	✓consistent
0	0	1	1	0	1	✗inconsistent
0	1	0	0	0	0	✓consistent
0	1	1	1	0	0	✓consistent
1	0	0	0	1	1	✗inconsistent
1	0	1	1	0	1	✗inconsistent
1	1	0	0	1	0	✓consistent
1	1	1	1	0	0	✓consistent

- A state is consistent when *for all possibilities of the inputs* there is *at most one transition* active.
- We start again with the truthtable method.
- And check all possibilities.
- We found already 1 entry that is inconsistent, we can stop here!

Howto design an FSM: step 2



Now that we determined that the state diagramm is **complete** and **consistent** we can continue. Otherwise:

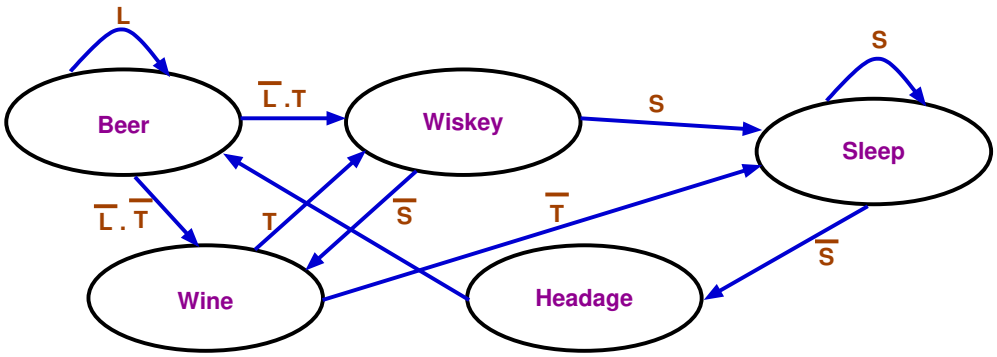
To make a state complete we have to:

Add to each incomplete line in the truthtable one **1** for one of the conditions

To make a state consistent we have to:

Remove from each inconsistent line in the truthtable as many **1**'s such that only one of the conditions is active

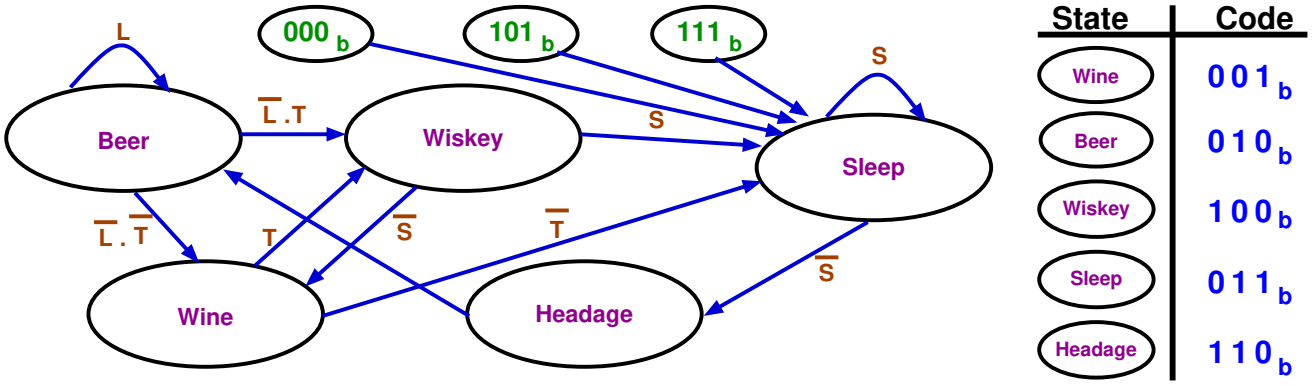
Howto design an FSM: step 3



State	Code
Wine	001_b
Beer	010_b
Wiskey	100_b
Sleep	011_b
Headage	110_b

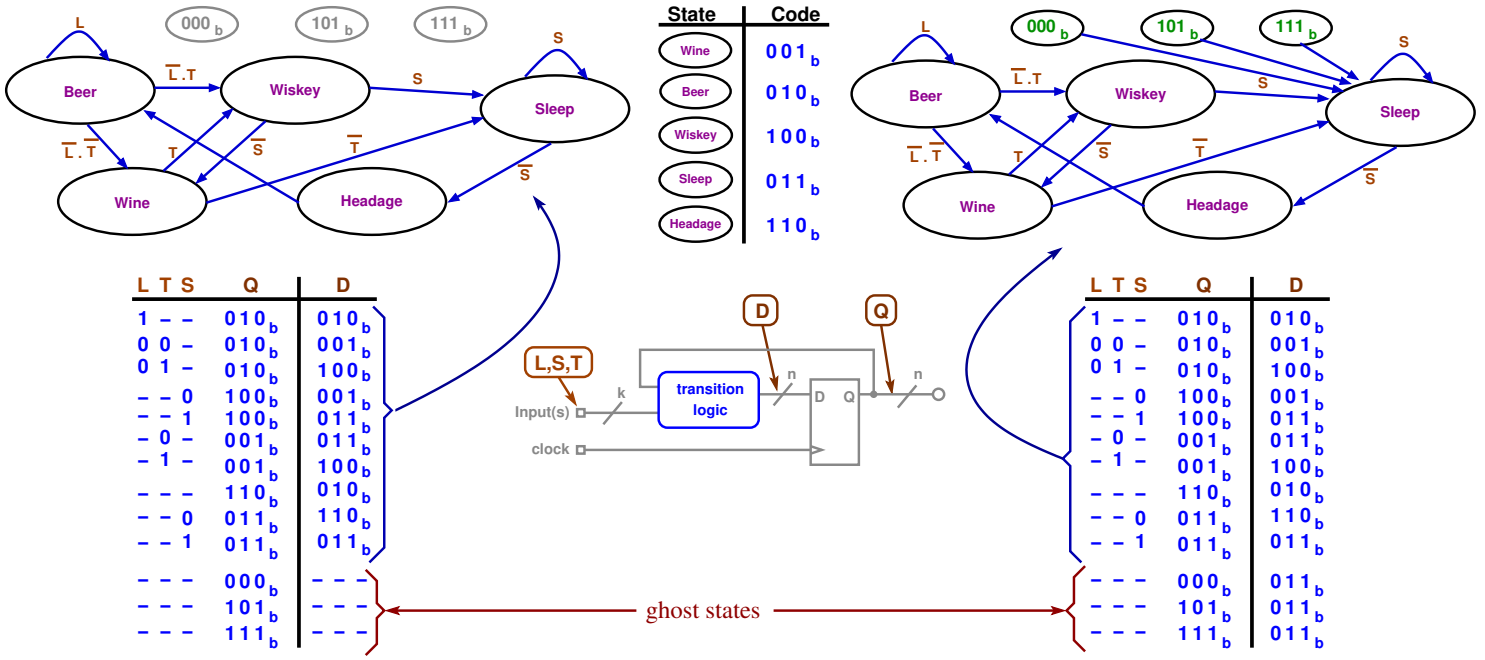
- The next step is to define the code table.
- But how many bits **n** do we require?
- To code **I** states we require at least $\mathbf{n} \geqslant \left\lceil \frac{ln(\mathbf{I})}{ln(2)} \right\rceil$ bits.
- Hence we could code with 5 bits like this.
- The minimum number of bits (**n**) required here is 3, like for example this.
- Note: The coding can impose that you have to use a Moore machine, or that you can use a Medvedev machine.

Ghost states



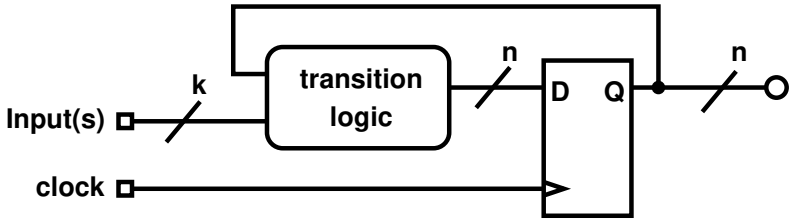
- From the 8 possible codes we only used 5, what about the other 3?
- These unused codes represent the *ghost states* of our state machine.
- There are two ways to deal with these *ghost states*:
  1. We can just ignore them and treat them as *don't care*. This is good for non-critical applications.
  2. We can make sure that they point to a *save state*. This is good for critical applications like medical implants.

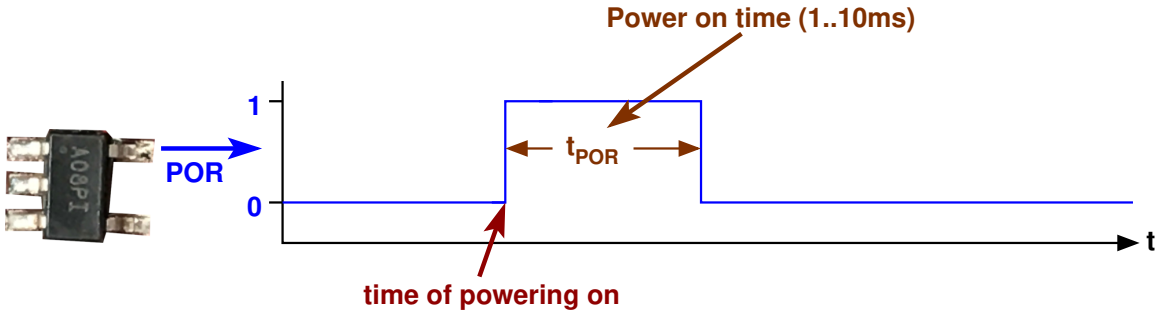
Transition table



- Now we can create the transition table.
- We start with the version with the ignored ghost states.
- As comparision the version with the “save” ghost states.

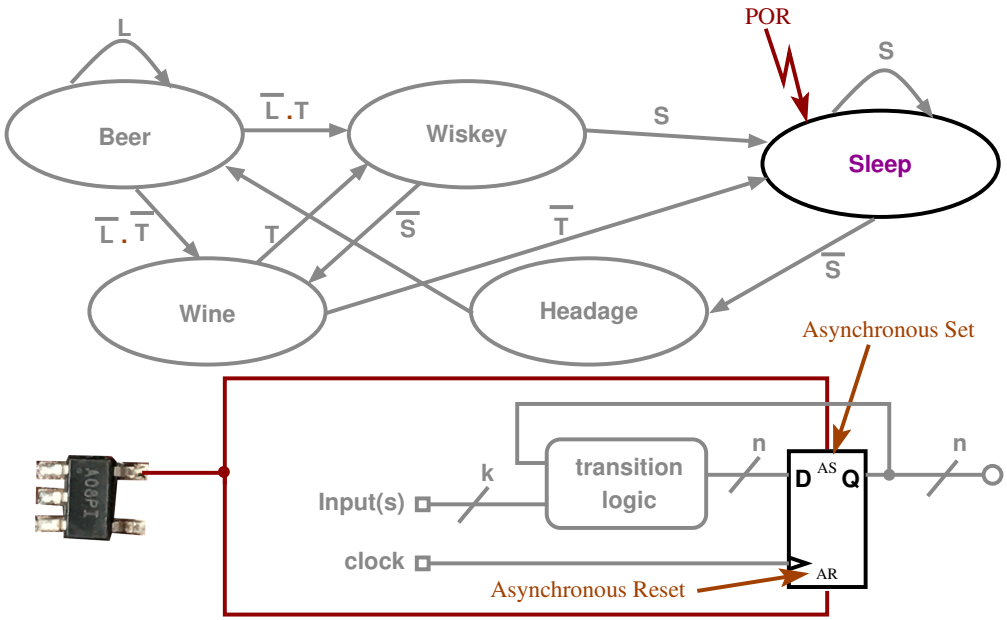
Power On Reset





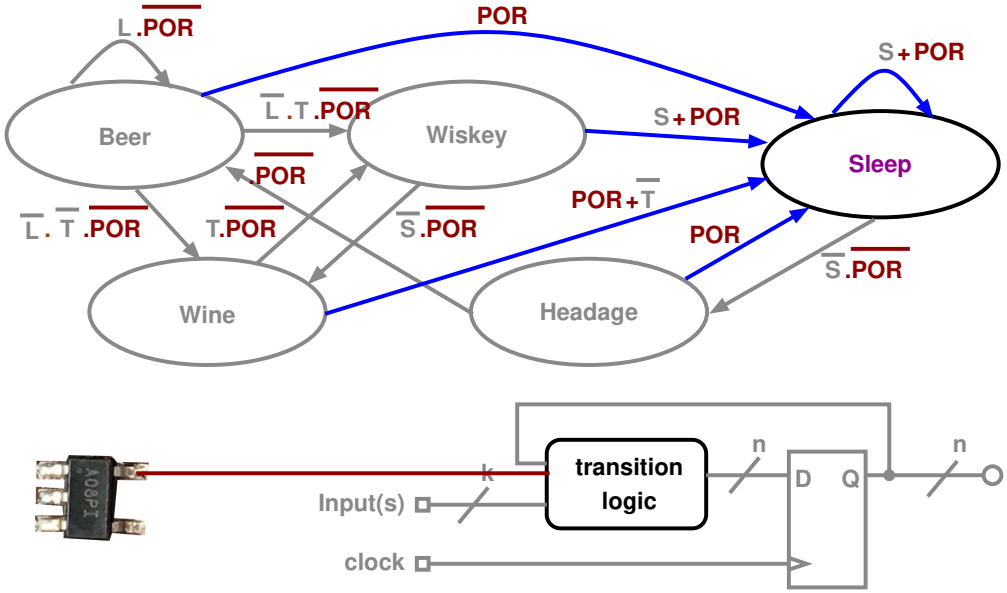
- What happens with my FSM when powering on?
- The state at power on is random.
- To be able to have a defined state we use a Power-On-Reset (POR).

Asynchronous Power On Reset



- How to use the POR?
- We identify the *Reset State*.
- We can use now the asynchrounous Set and asynchronous reset inputs of a flipflop.
- This is a asynchronous POR, we note it with a flash in the state-diagram.

Synchronous Power On Reset



- We can also feed the POR into the transition logic.
- We call this a synchronous POR.
- We have to modify the state-diagram with the reset transitions.
- And finally we have to make the state-diagram complete and consistent.

Lecture 9

Digital system design

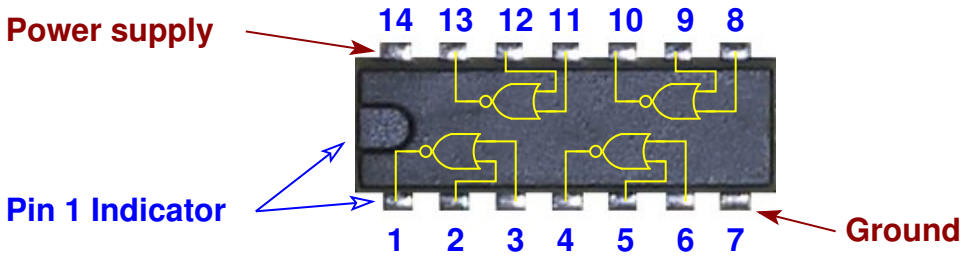
None Ideal Behavior

CS173 - Conception de systèmes numériques
March 2017

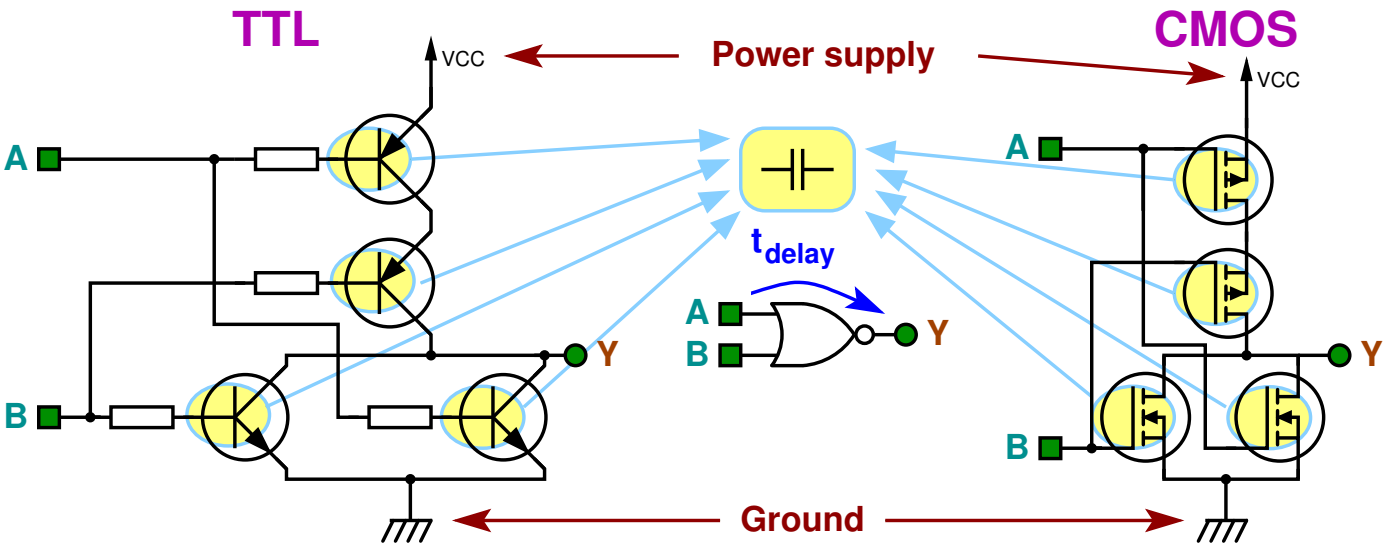
Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Technology

- Today all gates are implemented in semiconductor material
- However, the transistors used are different
- The two famous families are:
- **T**ransistor to **T**ransistor **L**ogic
- **C**omplementary **M**etal **O**xide **S**ilicon
- The gates come in the form of an **I**ntegrated **C**ircuit

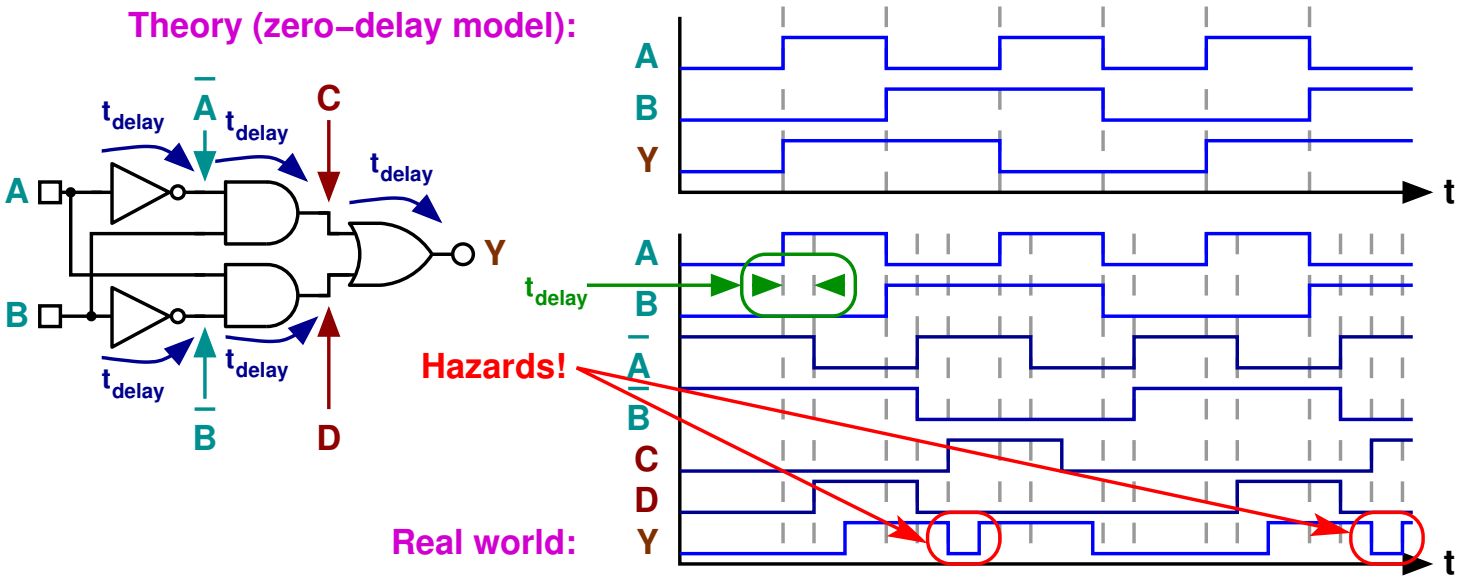


Technology



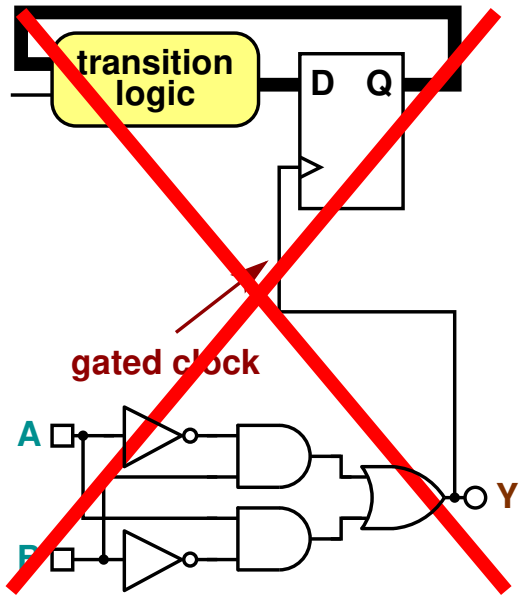
- Let's take a closer look at the NOR-gate.
- In TTL-technology it is build up like this, and in CMOS-technology like this.
- In both cases we need a power suply, otherwise they won't work!
- Everywhere there are capacitors, resulting in a delayed reaction of the gate!

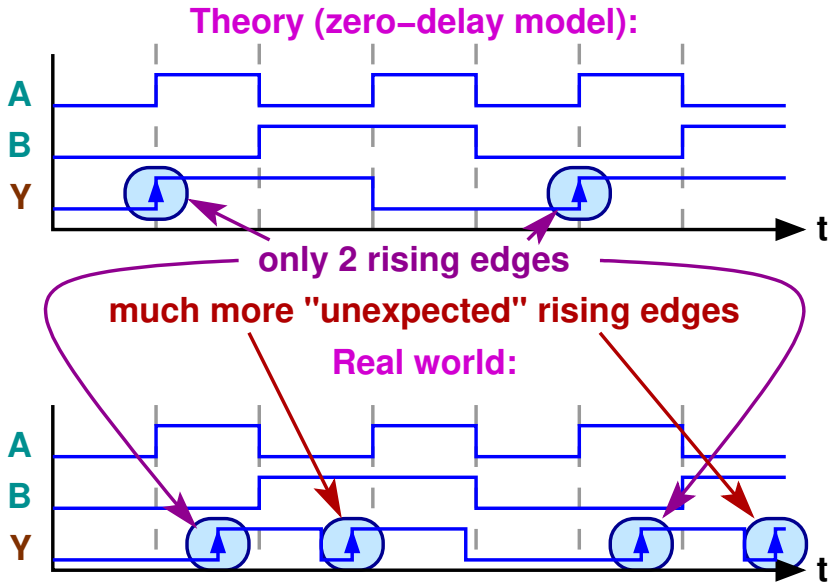
Hazards



- What happens with this circuit?
- This is the theoretical behavior, also called the *zero-delay model*; what about reality?
- Each gate has a gate-delay. In this example we simplify and assume that all the gates have the same delay $\mathbf{t_{delay}}$.
- We get due to the gate delays unexpected level changes! These delay-caused level changes are called *hazards* which cannot be avoided!

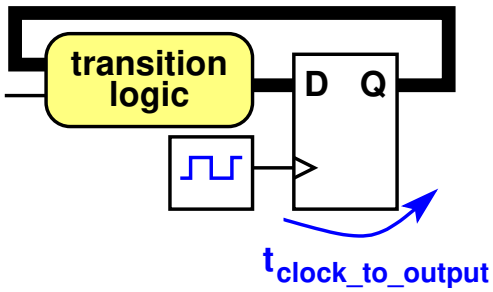
Gated clocks

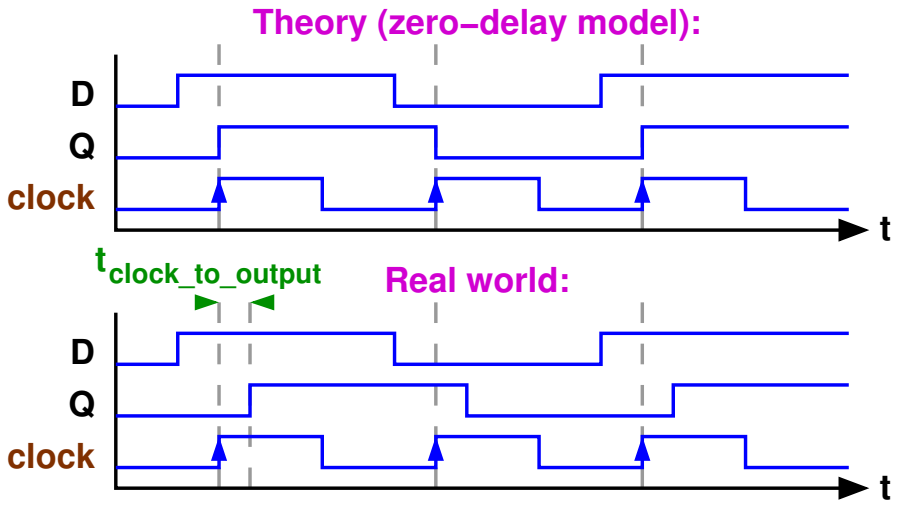




- A situation which seems often to be “logical”:
- We call these kind of circuits *gated-clock* circuits.
- This is a very **bad** idea; we think there are only two rising edges.
- But in the real circuit there can be an awfull lot more rising edges due to *hazards*!
- So don’t do this, unless you know what you are doing! Use the signal Y as input to the transition-logic!

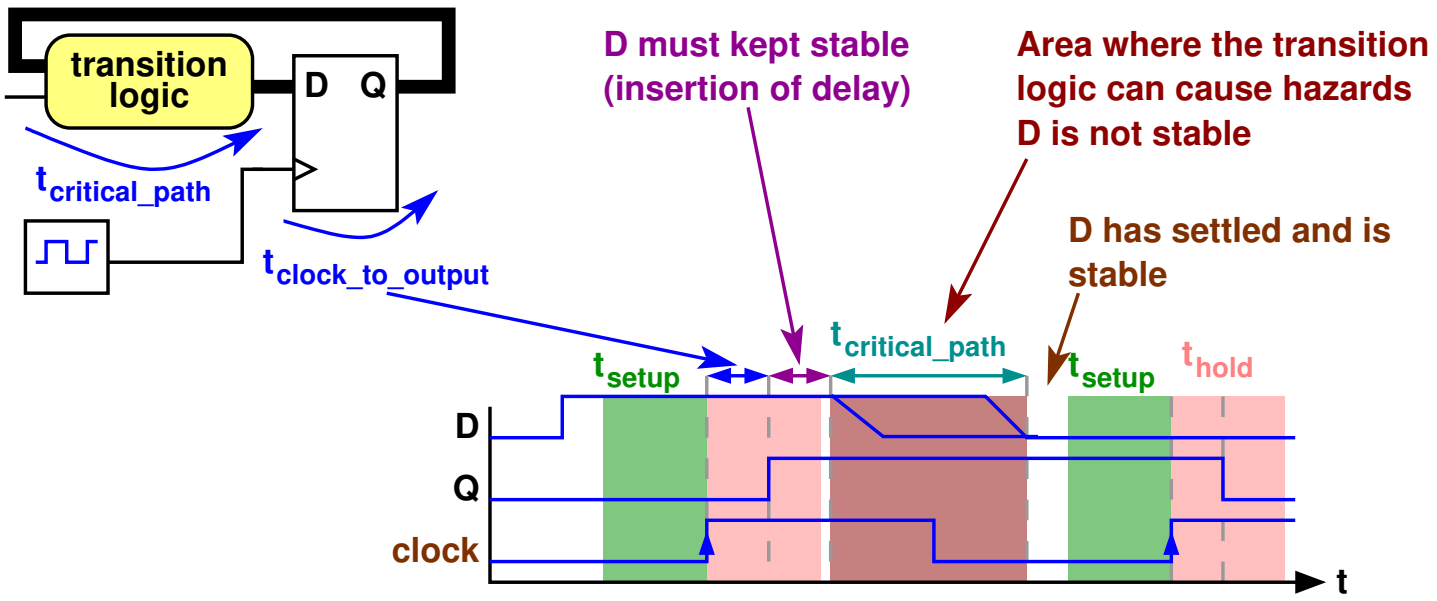
Real DFF's





- So we schould always connect like this. But what about the real behavior of the d-flipflop?
- In the zero-delay model we know it works like this.
- In reality the output of the flipflop reacts a time $t_{clock\_to\_output}$ after the rising edge of the clock.
- And it becomes even worst.....

Metastability



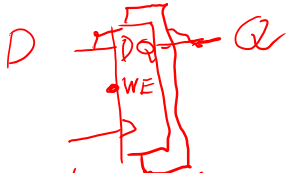
- Let's look at one positive edge.
- There are two time regions, the setup time t_{setup}, and the hold time t_{setup}; D must kept stable in this period, otherwise:
- The D-flipflop goes in metastability, and it settles at either **1** or **0**, independent of the value of **D**!

TODO

- critical path
- max. frequency
- synchronisation

$$Q = \alpha \cdot \overline{E_1} + \beta \cdot E_1$$

D	WE	α	
0	0	Q	←
0	1	ϕ	
1	0	Q	←
1	1	1	



1) diagram d'état pour 1 bit

2) D et Q ont 2 bit dresser le circuit correspondant

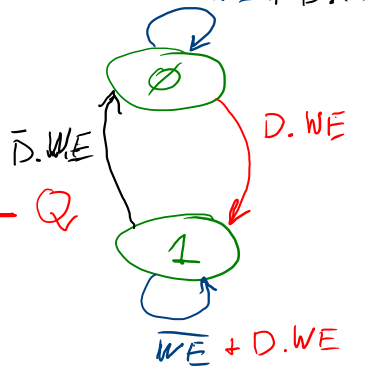
table fonctionnelle :

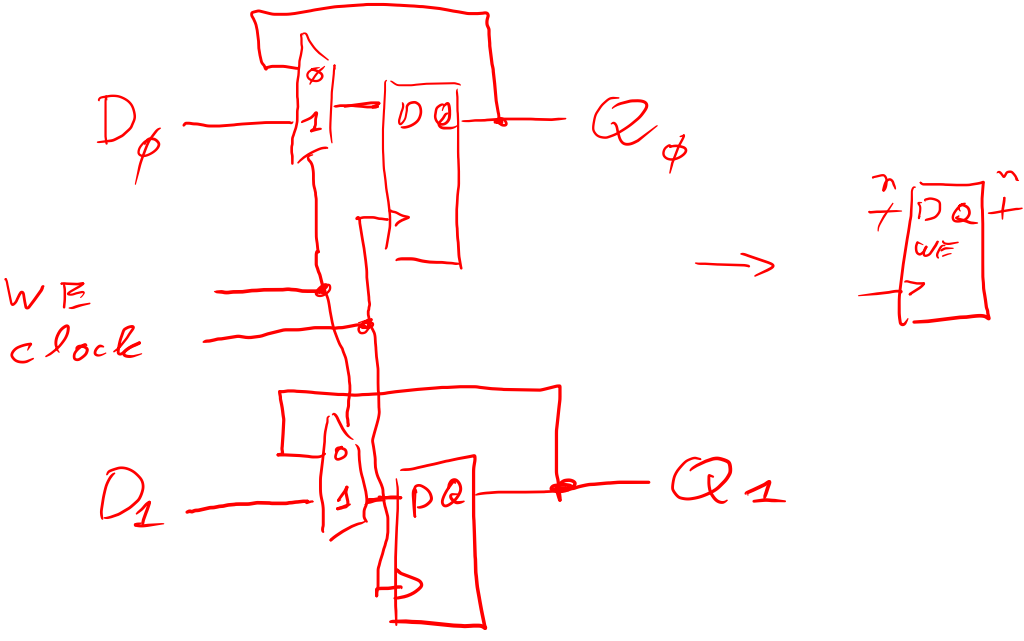
WE	D	Q	α
ϕ	-	ϕ	ϕ
ϕ	-	1	1
1	ϕ	-	ϕ
1	1	-	1

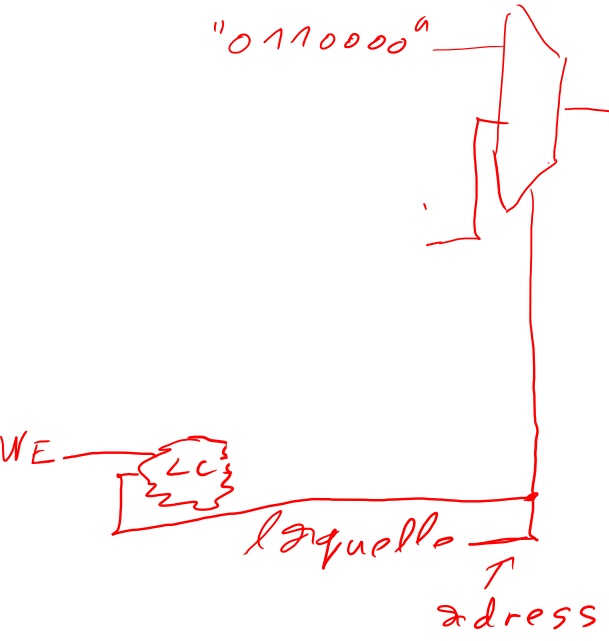
WE \ Q	ϕ	1
ϕ	ϕ	1
1	ϕ	1

$\alpha = WE.D + \overline{WE}.Q$

WE	Q	α
ϕ	Q	
1	D	




D_ϕ
0
1
D
Q
Q_ϕ
WE
clock
D_1
0
1
D
Q
Q_1
n
D
Q
WE
n



RAM

Random
Access
Memory

ROM → Read only memory

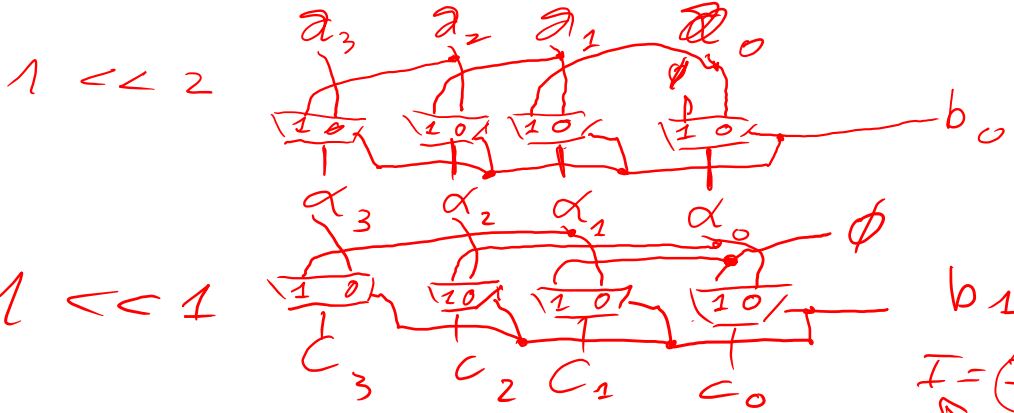
$a << b$ $\rightarrow$ a = 4 bit

b = 2 bit

$c = a << b$

$= a << (b_1 \cdot 2 + b_0)$ $b_1\ b_0$

$= a << (b_1 \cdot 2) + a << b_0$



I = I + 1;

Lecture 11

Digital system design

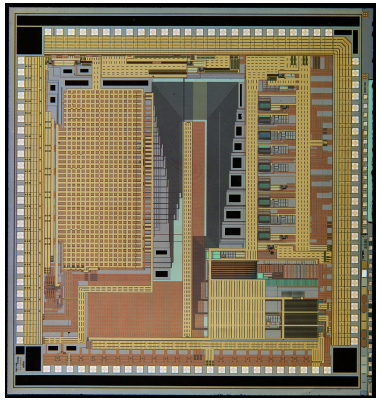
Programmable Logic (1)

CS173 - Conception de systèmes numériques
April 2017

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Introduction

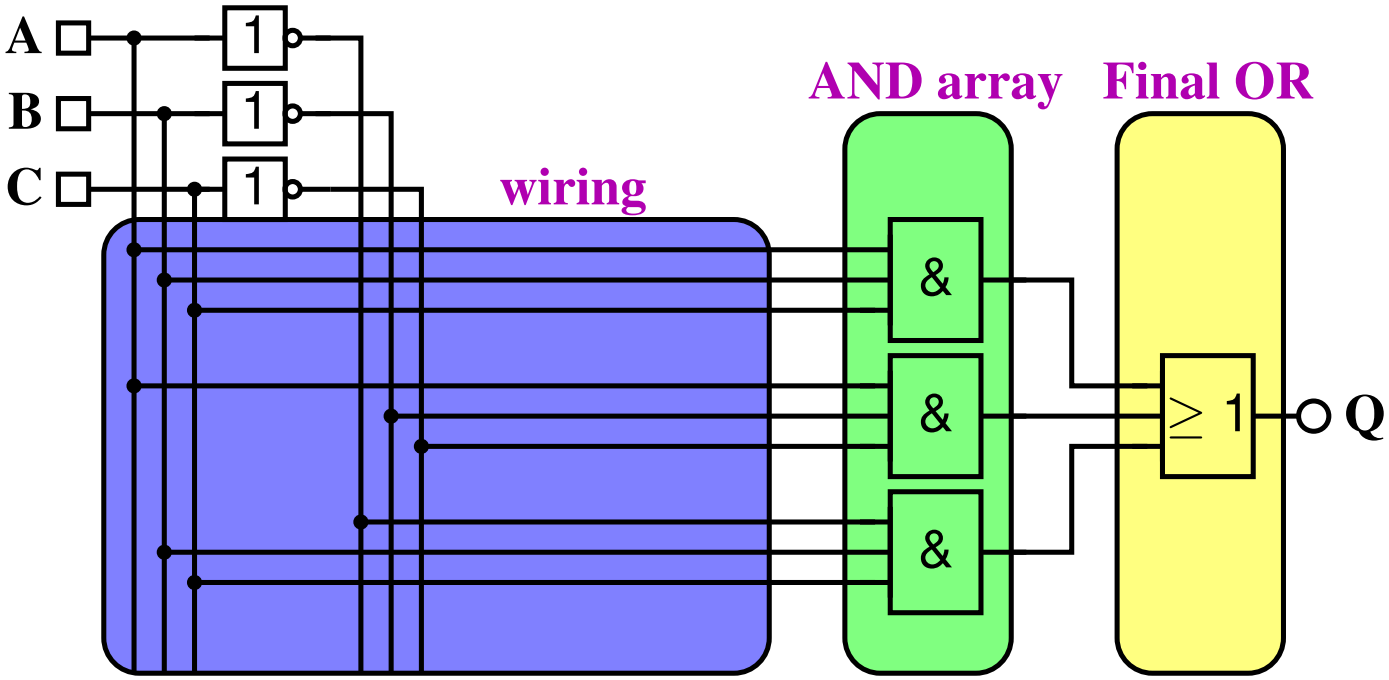
- Today all digital (and analog) circuits are realized as ASIC.
- The ASIC (*Application Specific Integrated Circuit*) contains billions of transistors.
- Making an ASIC is very expensive as:
  - The design process takes more than a year.
  - The initial cost starts from $100,000.= and can go as high as $5,000,000.=
- Therefore, it is only interesting for high-volume market.
- This is one of the reasons why *programmable logic devices* were introduced.
- Of course a programmable logic device is an ASIC.



Programmable Array Logic

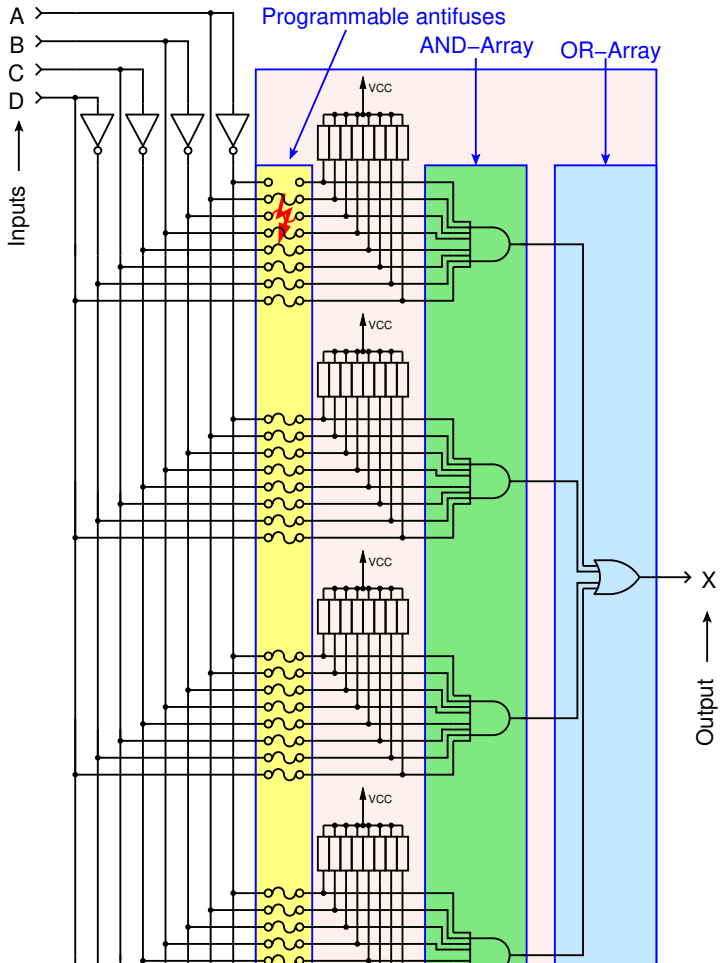
- The first of the family of programmable logic devices was introduced in 1978.
- In 1978 the company MMI (*Monolithic Memories Inc.*) introduced the PAL.
- PAL stands for *Programmable Array Logic*.
- The basic concept of a PAL is based on the kanonical form in which a logic function can be written:

$$\mathbf{Q} = \mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} + \overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C} + \mathbf{A} \cdot \overline{\mathbf{B}} \cdot \overline{\mathbf{C}}$$



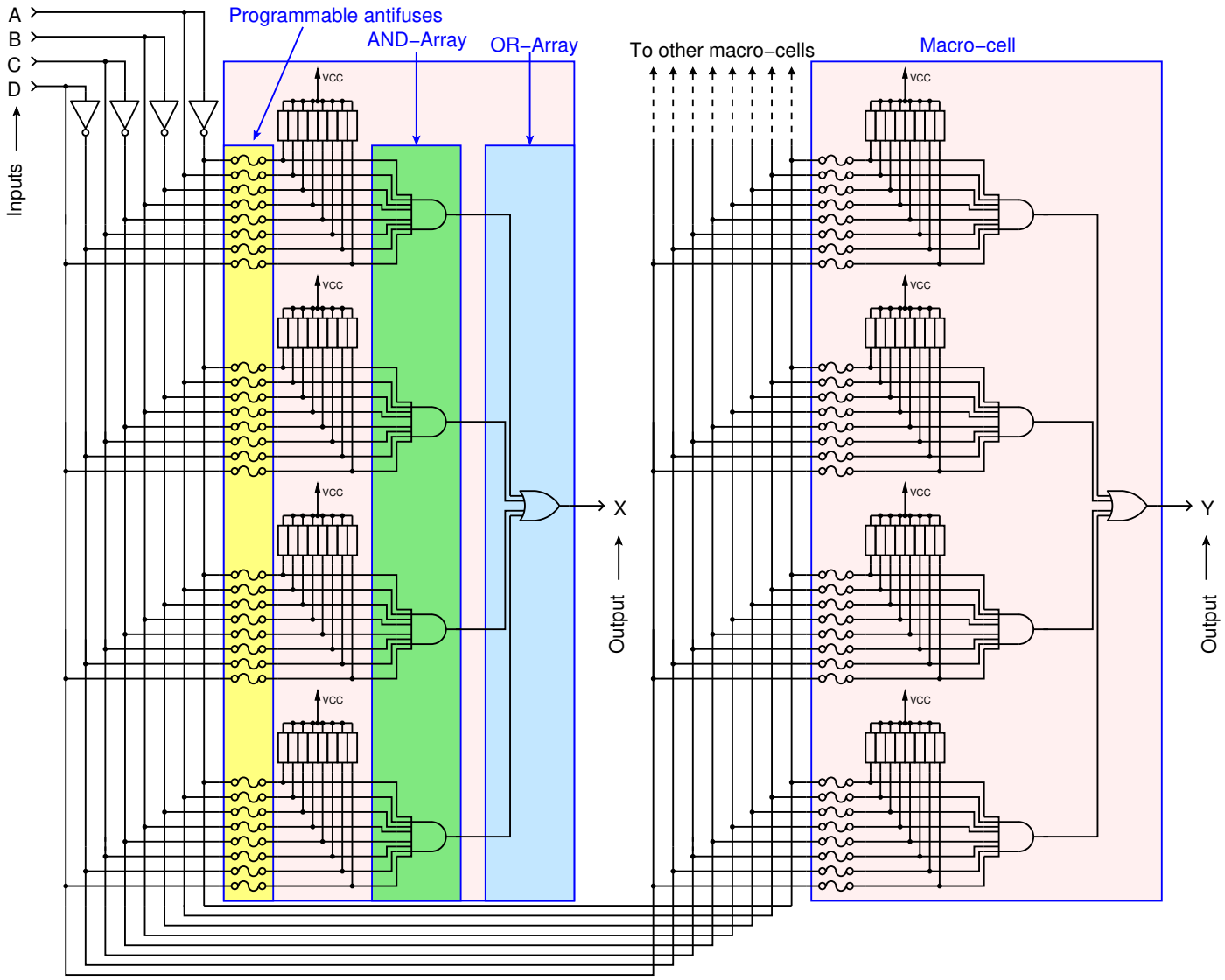
Programmable Array Logic

- Using the *sum of products* form always results in the same two-stage logic.
- The only thing that differs is the wiring. This is the *programmable* part.
- In the PAL, the wiring is realised by *antifuses*. Programming is performed by *“blowing the fuses”*



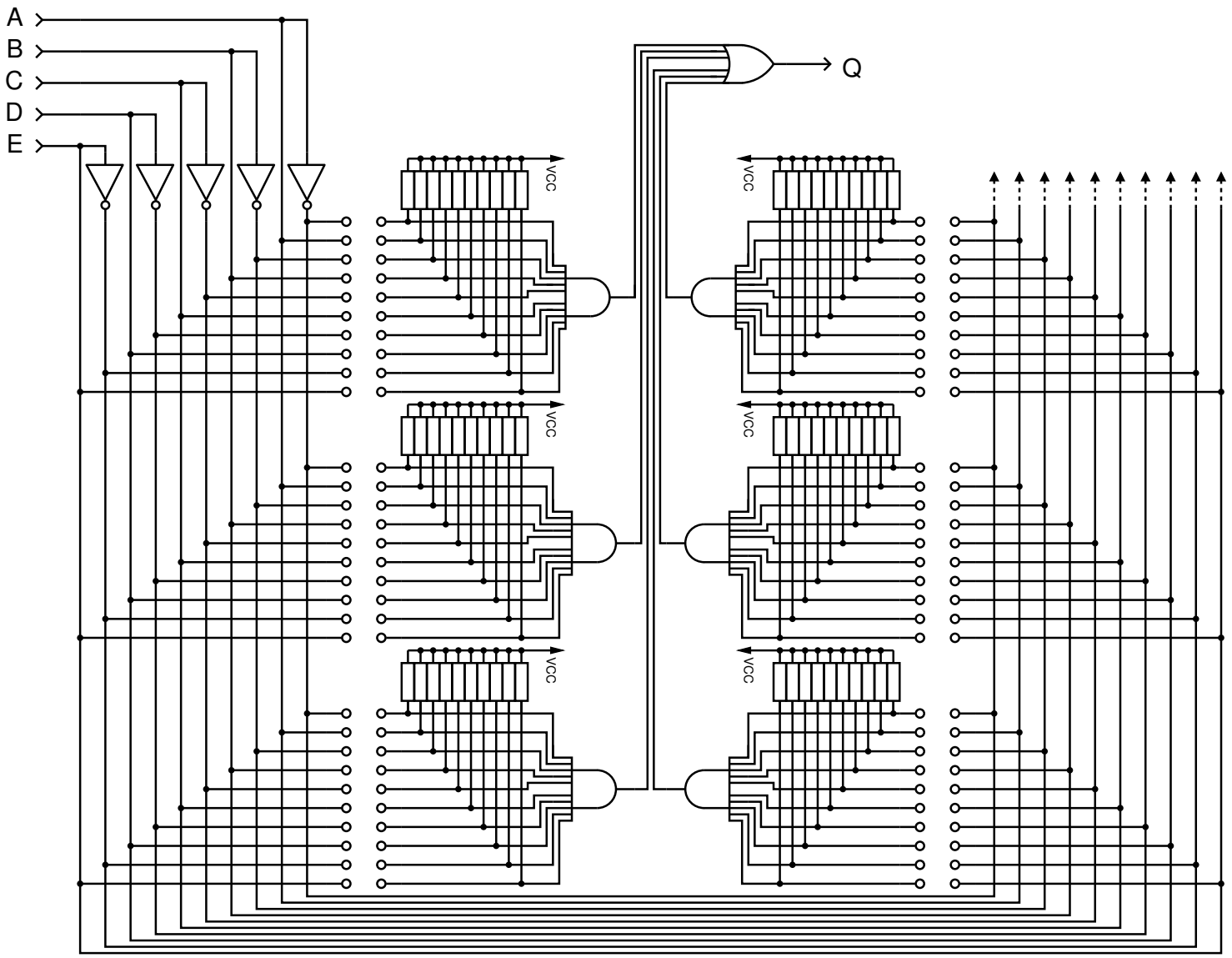
Programmable Array Logic

- One such arrangement is called a *macro-cell*.
- A PAL contains multiple of them.



Exercise

Realize the function $\mathbf{Q} = \mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} + \overline{\mathbf{A}} \cdot \mathbf{B} \cdot \mathbf{C} + \mathbf{A} \cdot \overline{\mathbf{B}} \cdot \overline{\mathbf{C}}$ on the following macro-cell:

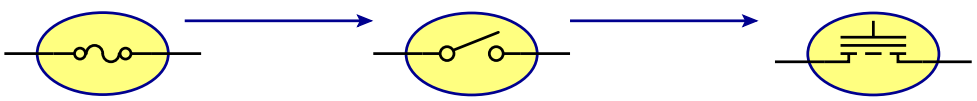


Programmable Array Logic

- A PAL was very succesfull as:
  1. It allowed for any logic function to be realized.
  2. The input-to-output delay is fixed (namely: $\mathbf{t}_{not} + \mathbf{t}_{and} + \mathbf{t}_{or}$)
  3. Low power.
  4. Once programmed, even when the power is switched-off, the programming stays.
     We call this *non-volatile*.
- The PAL has one draw-back:
  It can only be programmed once. We call this *OTP (=One Time Programmable)*.
  When the designer made a mistake :

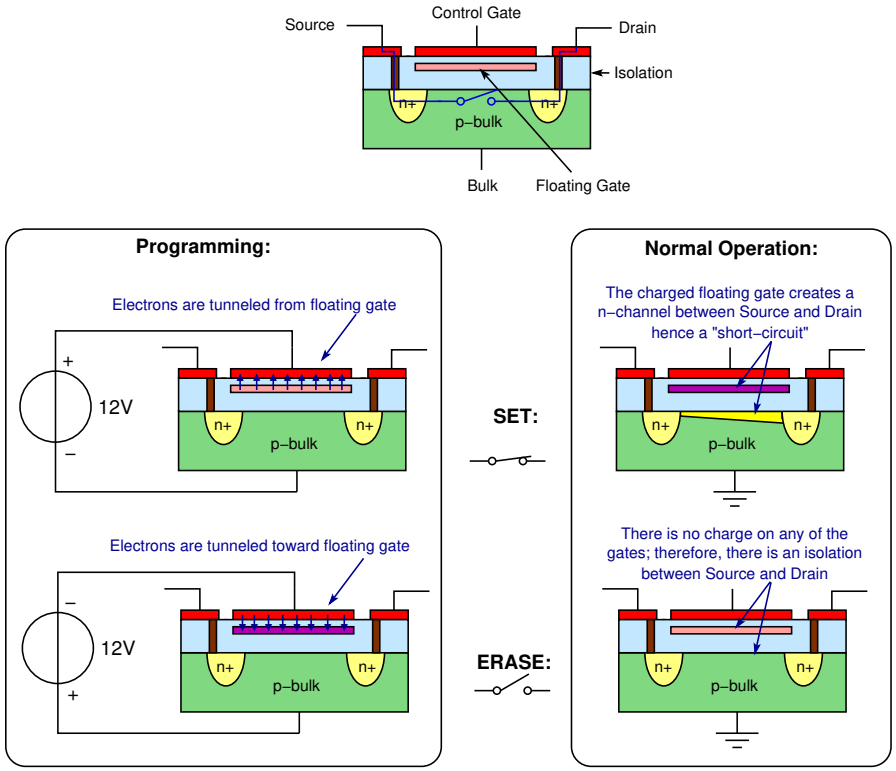
Generic Array Logic

- To overcome the problems of *OTP*, *Lattice Semiconductors* introduced in 1985 the GAL.
- GAL stands for *Generic Array Logic*
- Lattice had the great idea to replace the *antifuses* of the PAL by E^2(=Electrical Erasable)-switches.



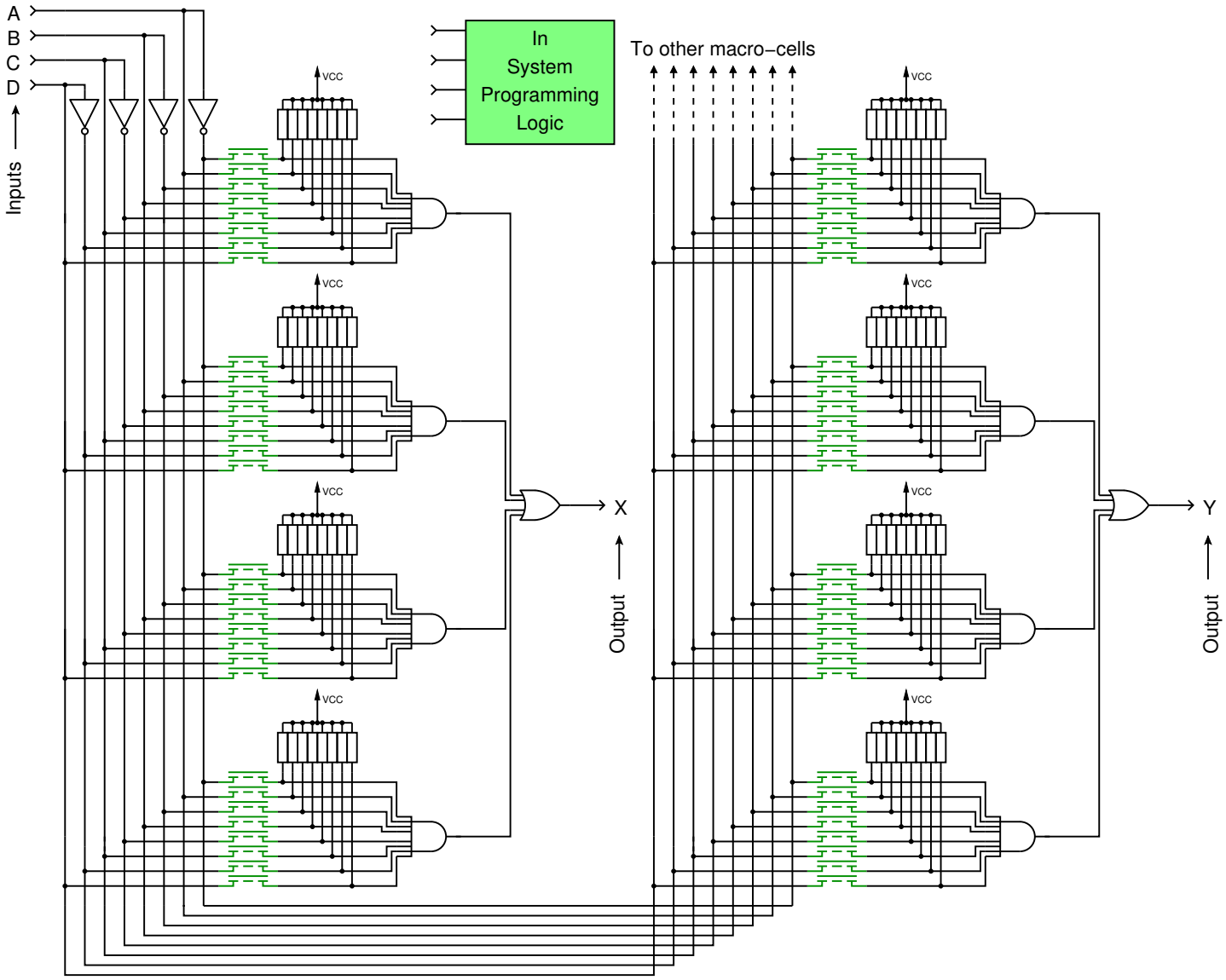
- These E^2-switches are realized by *floating-gate-transisotrs*, which are still the basis of nowadays used FLASH devices.

Generic Array Logic



- The advantage of this device is that programming and normal-operation are orthogonal, making ISP(=*In System Programming*), also refered to as ICP(=*In Circuit Programming*), possible.

Generic Array Logic

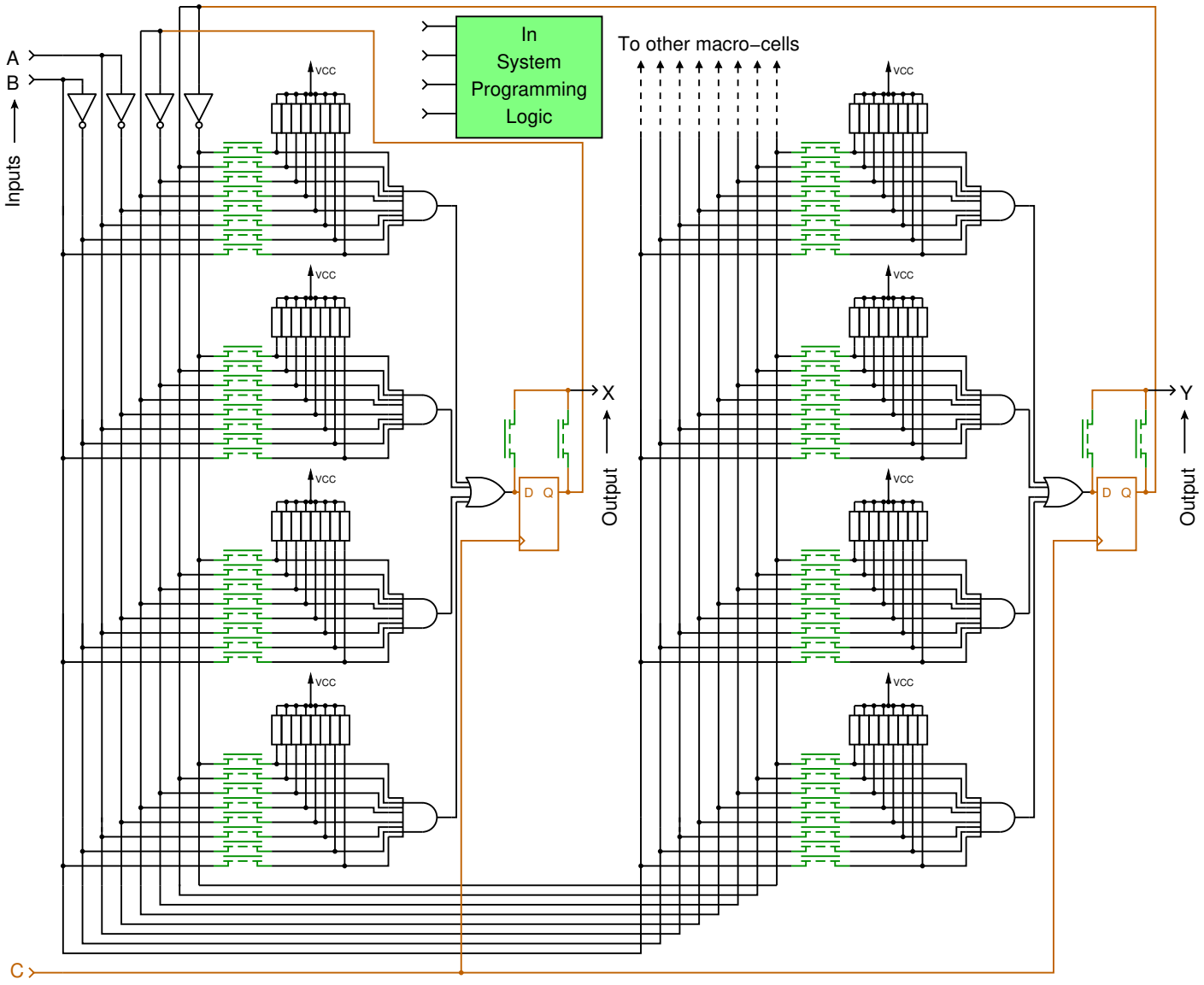


- GAL superseeded the PAL.

Generic Array Logic

- The GAL is in many aspects better than a PAL, however the E^2-switch technology has some restrictions:
  1. It can only be reprogrammed about 10.000 times.
  2. The programming stays for about 20 years.
- Another disadvantages of PAL's and GAL's is that only combinational functions can be realized.
- This has lead to a second generation of GAL's that incorporated in each macro-cell a D-flipflop:

Generic Array Logic



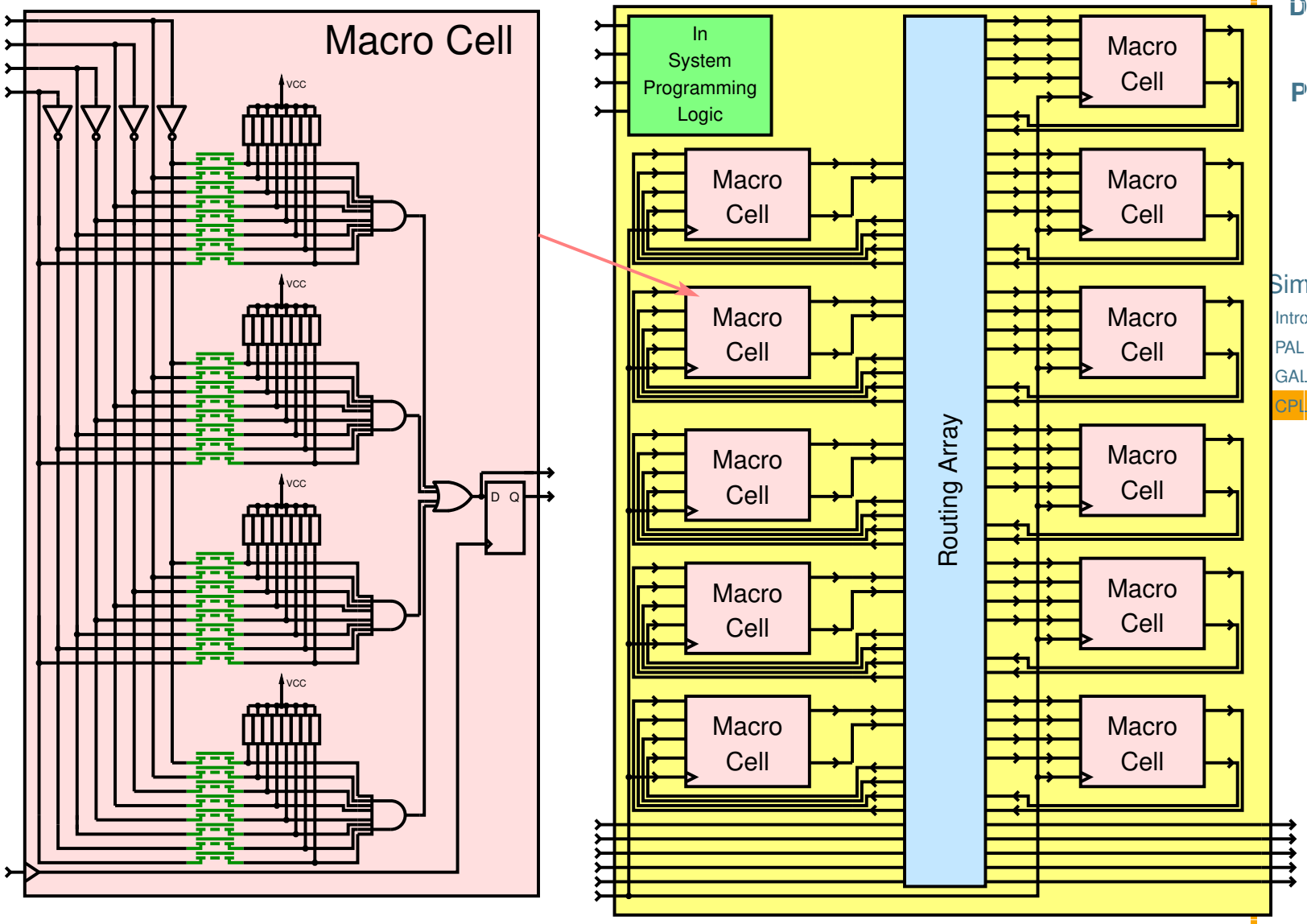
- The second generation GAL, also refered to as PLD(=*Programmable Logic Device*).

Complex Programmable Logic Device

As compexity of digital system increased,
and technology shrink smaller,
a new area of PLD's emerged,
the CPLD(=*Complex Programmable Logic Device*) was born.
The *complex* was not related to the programming,
but to the size of the device

the CPLD can be seen as several tens to hundreds of GAL's in one package.

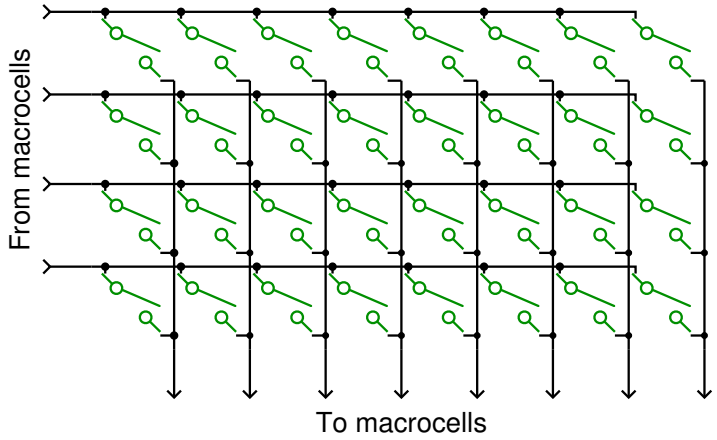
Complex Programmable Logic Device



- The CPLD uses still the same macro-cell, but connect them internally by a routing array.

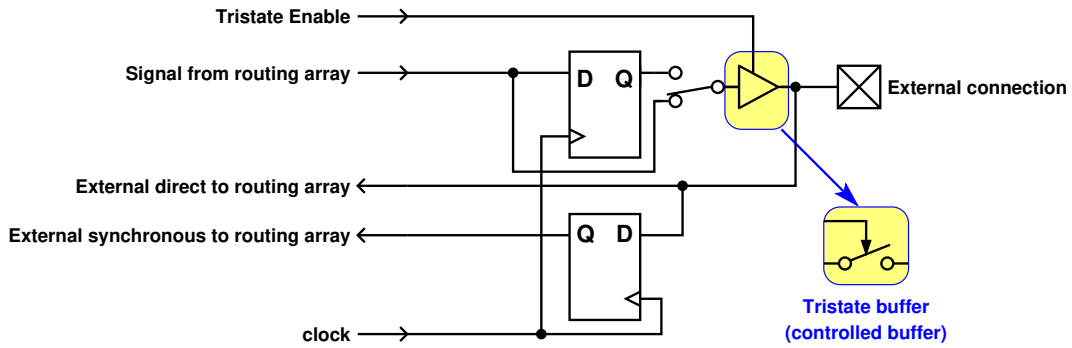
Complex Programmable Logic Device

- The “heart” of the CPLD is the routing array.
- The routing array consists of a matrix of connections.
- The horizontal lines are the outputs of the macro-cells.
- The vertical lines provide the inputs to the macro-cells.
- On each cross-point an E^2-switch is placed.
- In this way any connection can be made internal to the CPLD.



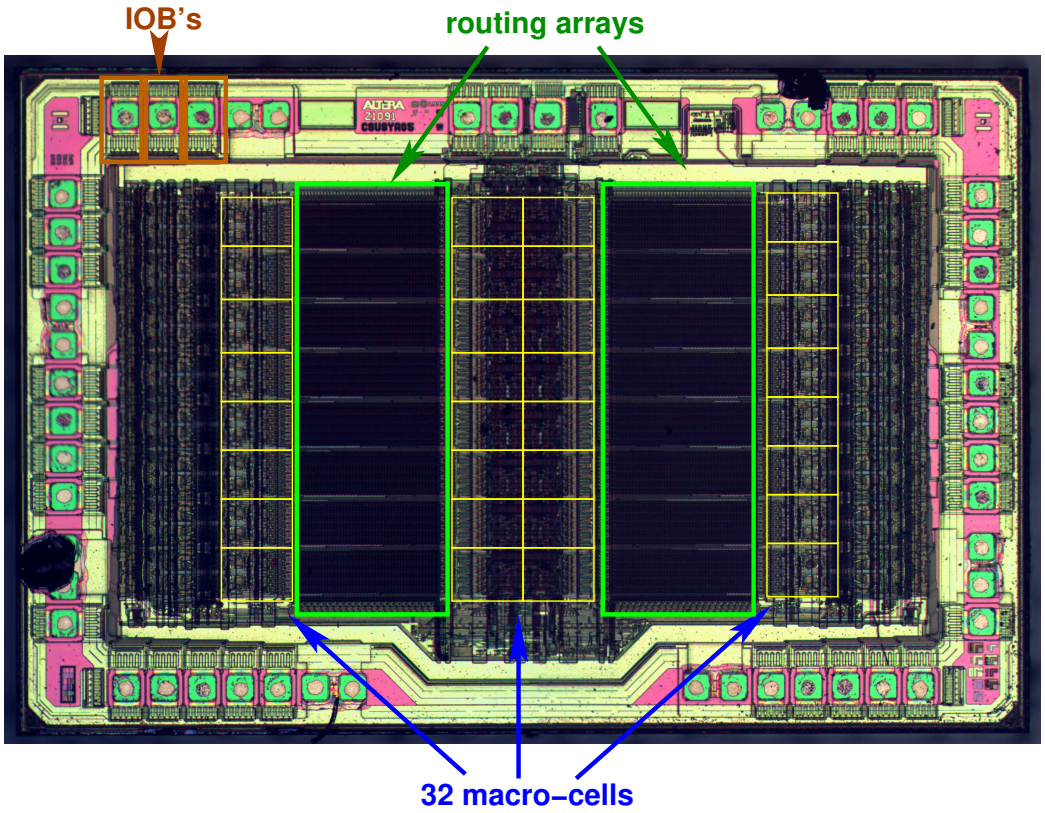
Complex Programmable Logic Device

- But how to connect to the outside?
- The CPLD introduced a programmable connection.
- This connection is called IOB(=*Input Output Block*).
- It allows to use a connection as Input
- Or as Output
- Or bi-directional



Complex Programmable Logic Device

- Example: Altera's/Intel's EPM7032 CPLD



Lecture 12

Digital system design

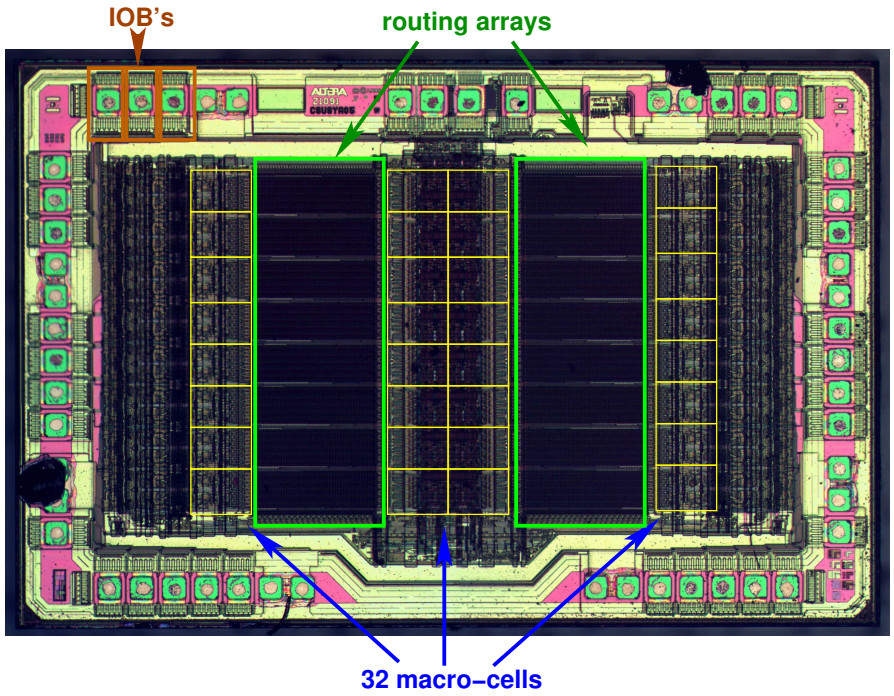
Programmable Logic (2)

CS173 - Conception de systèmes numériques
May 2018

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

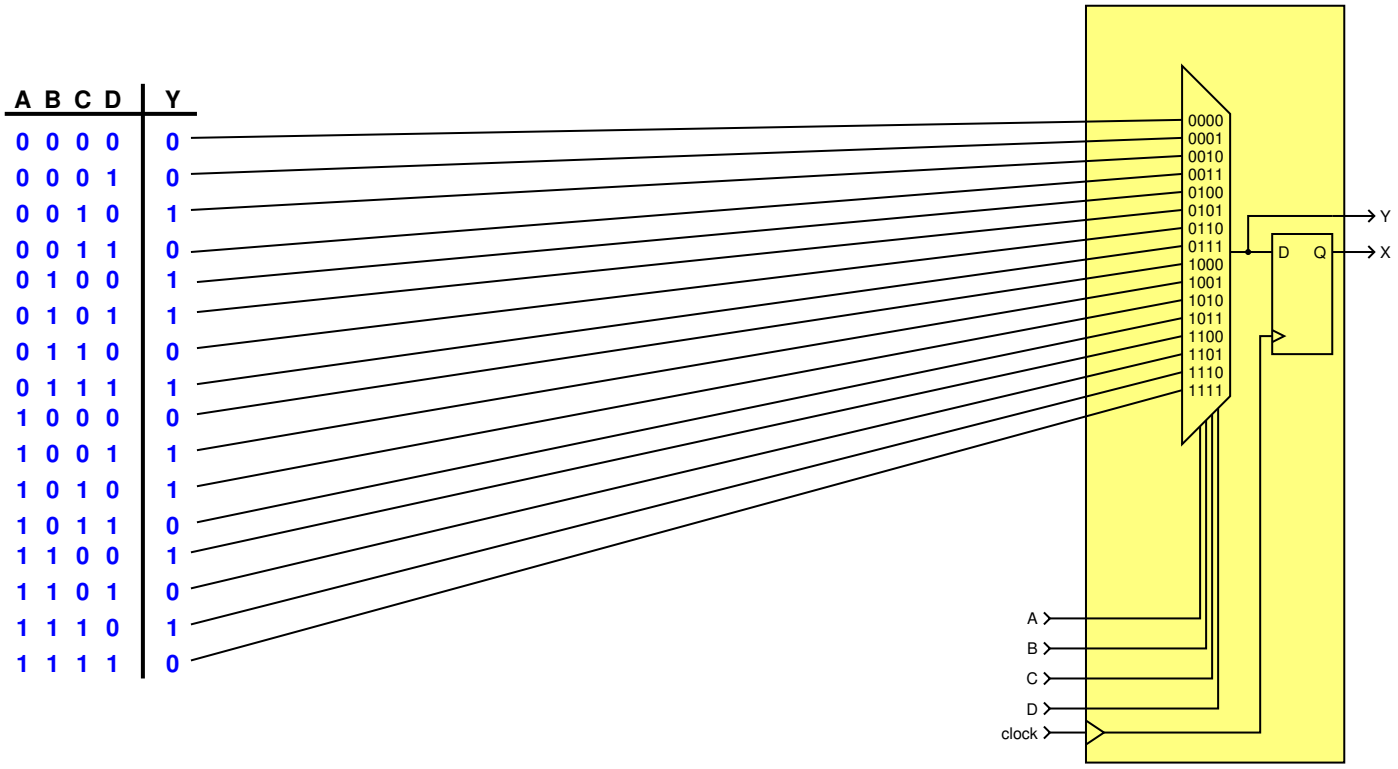
From CPLD to FPGA

- We have seen the three basic components of a CPLD.
- We have also seen that the scalability of the CPLD is restricted by it's routing array.
- To generate more complex designs, bigger devices are required.
- This has lead to the introduction of the FPGA.
- FPGA stand for *Field Programmable Gate Array*
- Where the CPLD used the *Sum-of-Products* representation of a logic function, the FPGA takes another approach.



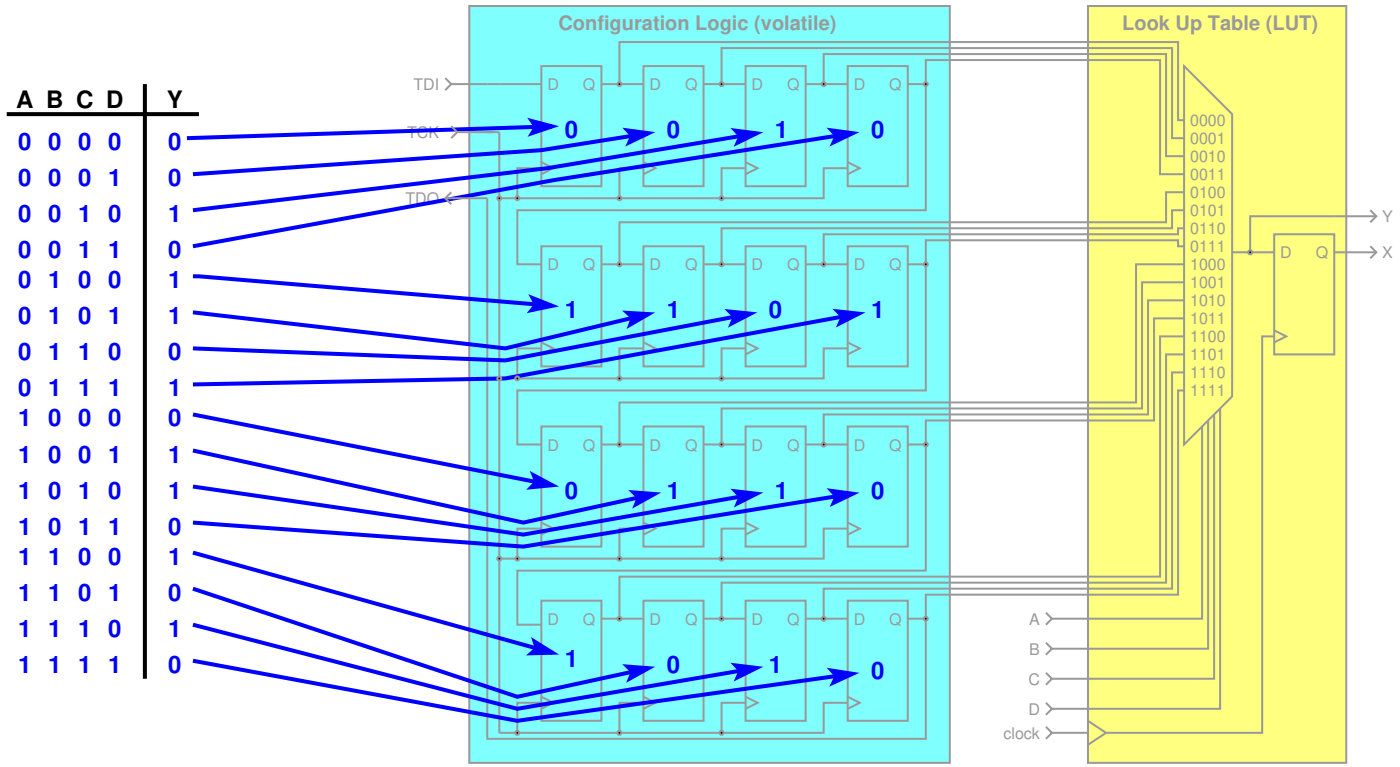
Look Up Table

- The FPGA takes the truthtable as basis of representing a logic function.
- Taking a MUX one can select each of the entries.
- With a D-flipflop we have the possibility to make FSM's.
- We call this structure (in yellow) a LUT (=*Look Up Table*).



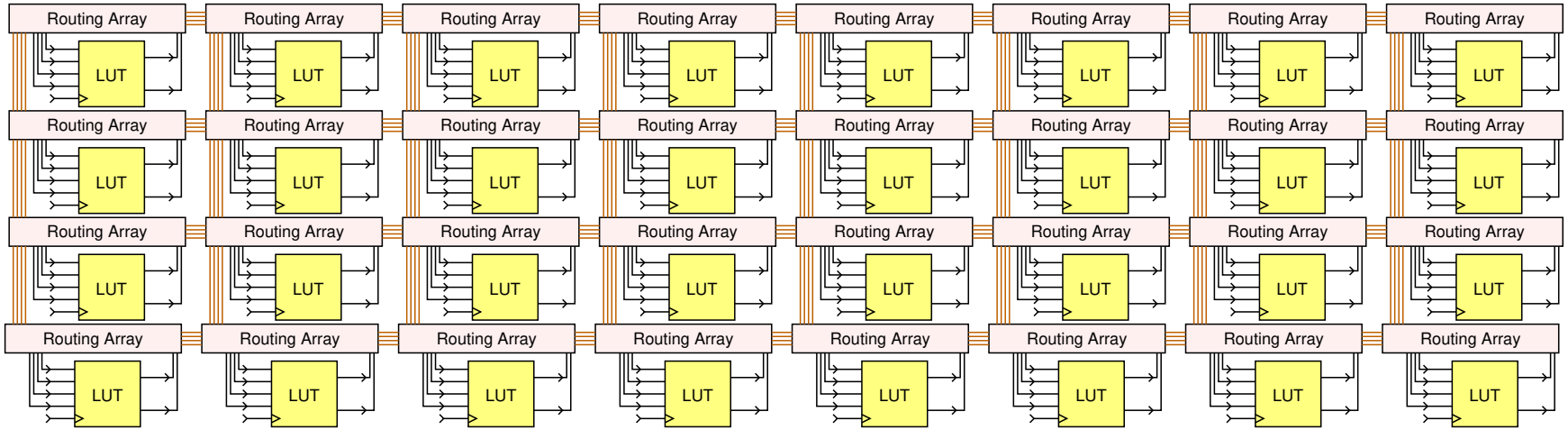
Look Up Table

- To store each of the entries in the truth-table we need to add memory.
- This memory-pane is the configuration part.
- It consists of D-flipflops in a shift-register configuration.
- During configuration the configuration pane is loaded with the values.



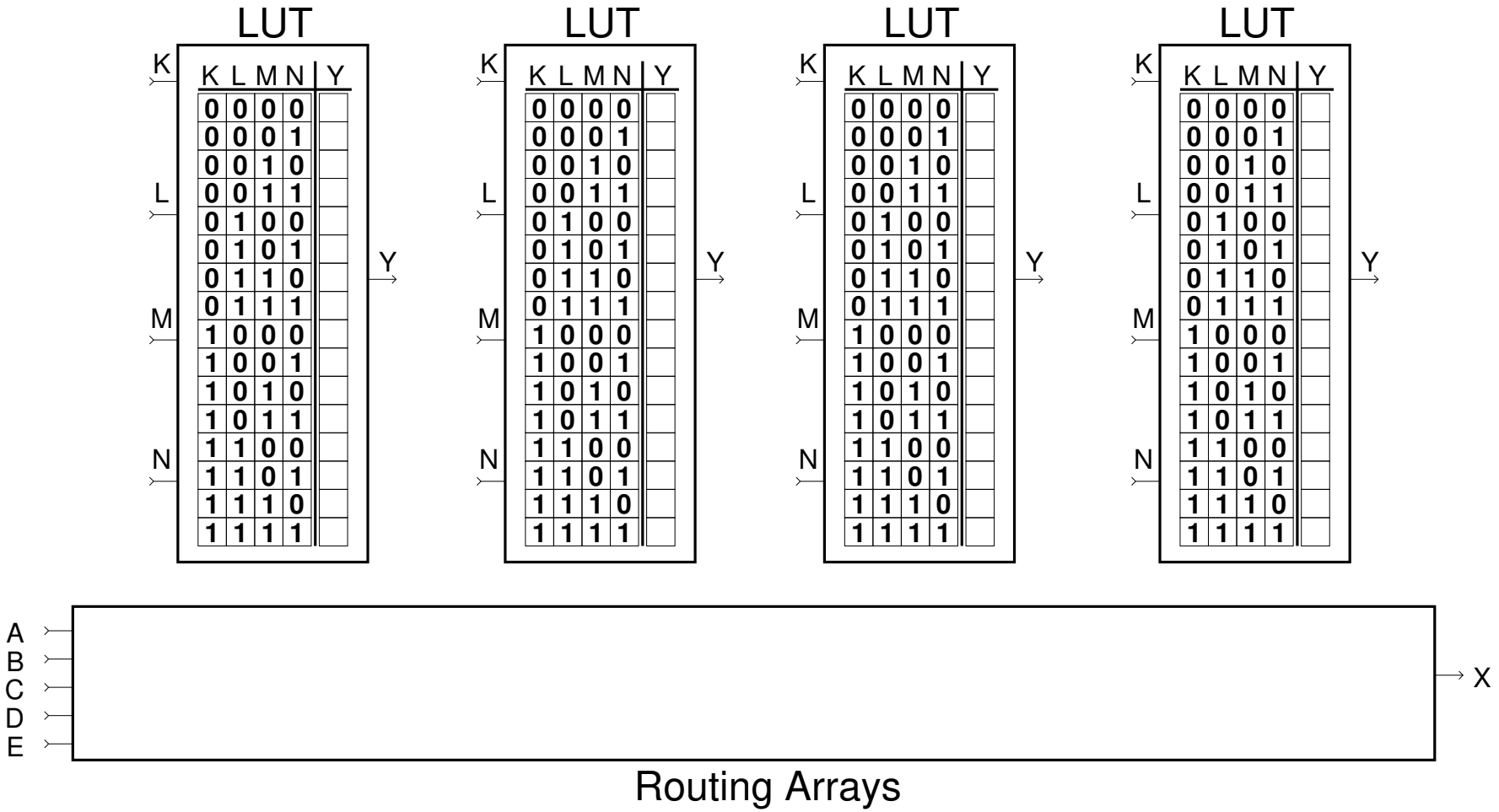
Architecture

- To overcome the scaling-problem of the CPLD, in the FPGA each LUT has it's own routing-array (*distributed routing*).
- This structure can be copied horizontally and vertically.
- To provide interconnects, the routing arrays are connected horizontally and vertically.



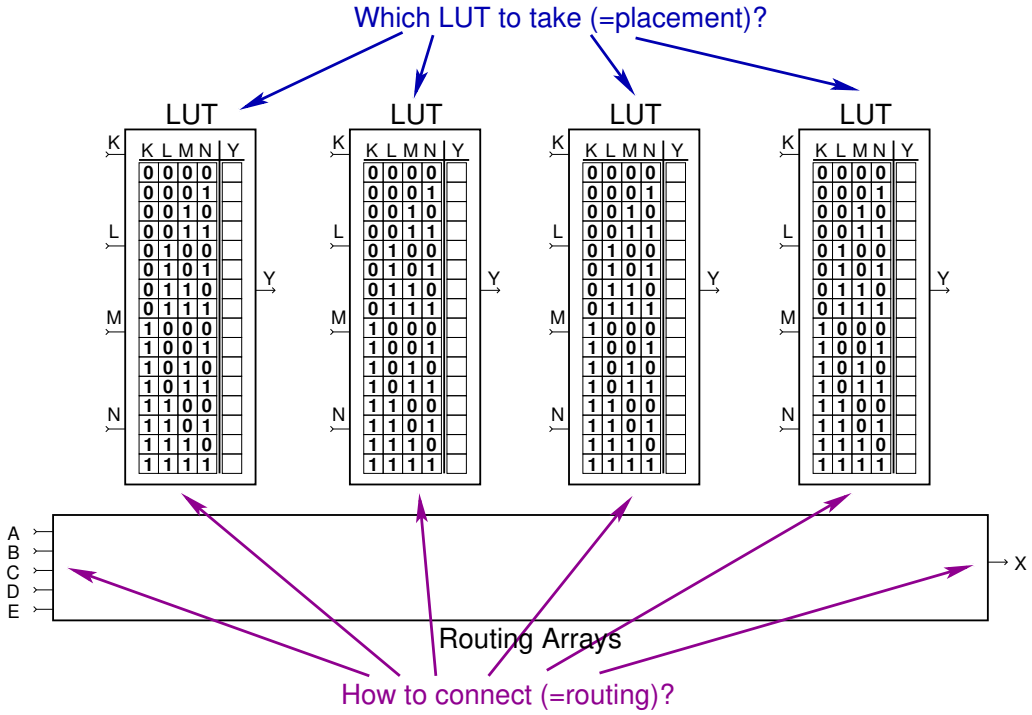
Exercise:

Implement the function $\mathbf{X} = \overline{\mathbf{A}} \cdot (\mathbf{B} \oplus \mathbf{C}) + \mathbf{D} \cdot \overline{\mathbf{C}}$ on the below shown FPGA-structure:



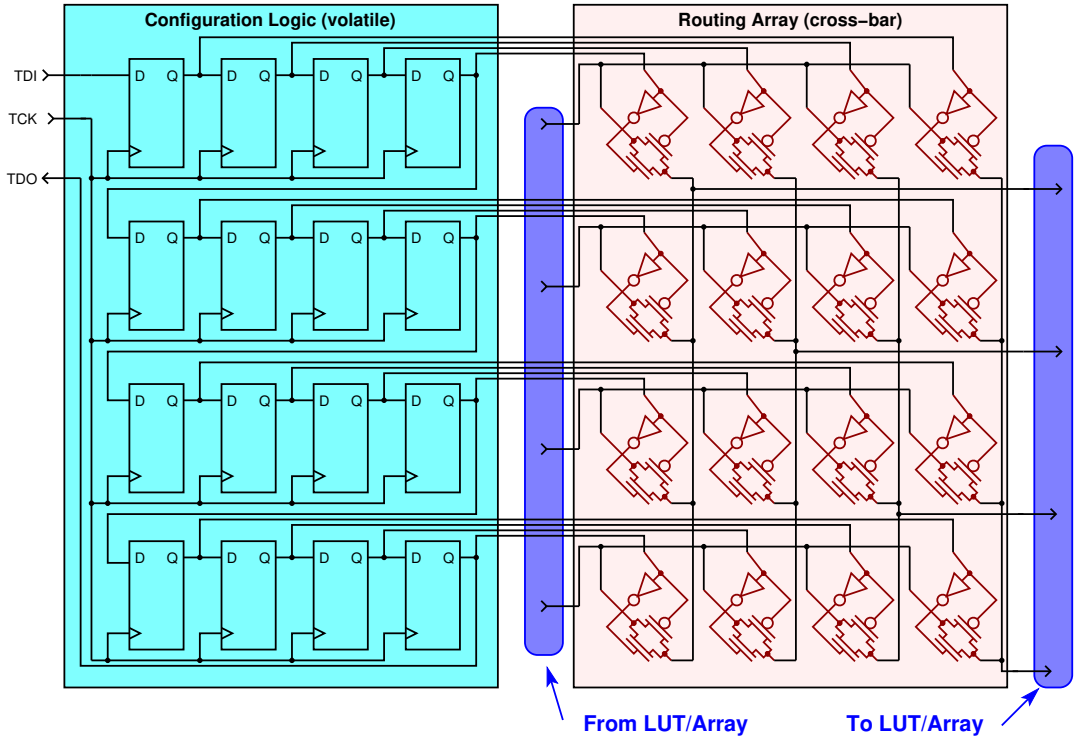
Place and Route

- In the previous exercise you had two problems, namely:
  1. On which LUT to map the function, we call this the *placement*.
  2. How to connect the inputs to the selected LUT and how to connect the LUT to the output, we call this the *routing*.
- Both actions are NP-hard problems. However, we have software that solve these problems, which are called the *Place and Route software* (P&R-software). Altera/Intel's Quartus performs this task.



Routing Array

- But how does the distributed routing array look like? Very simular to the CPLD, we use the same structure, namely a *cross-bar*.
- However, we replace the E^2-switches by pass-gates (*tri-state buffers*).
- let's look into the details, we have again the *use-pane* and the *programming* pane.
- As with the LUT we have to make a difference between configuration and *programming*.



Only four inputs?

This is all great, but each LUT is limited to four inputs!

This is quite limiting, isn't it?

At least in the CPLD we had macro-cells with 22+ inputs...

How do we implement functions with more than four inputs on an FPGA?

Let's see:

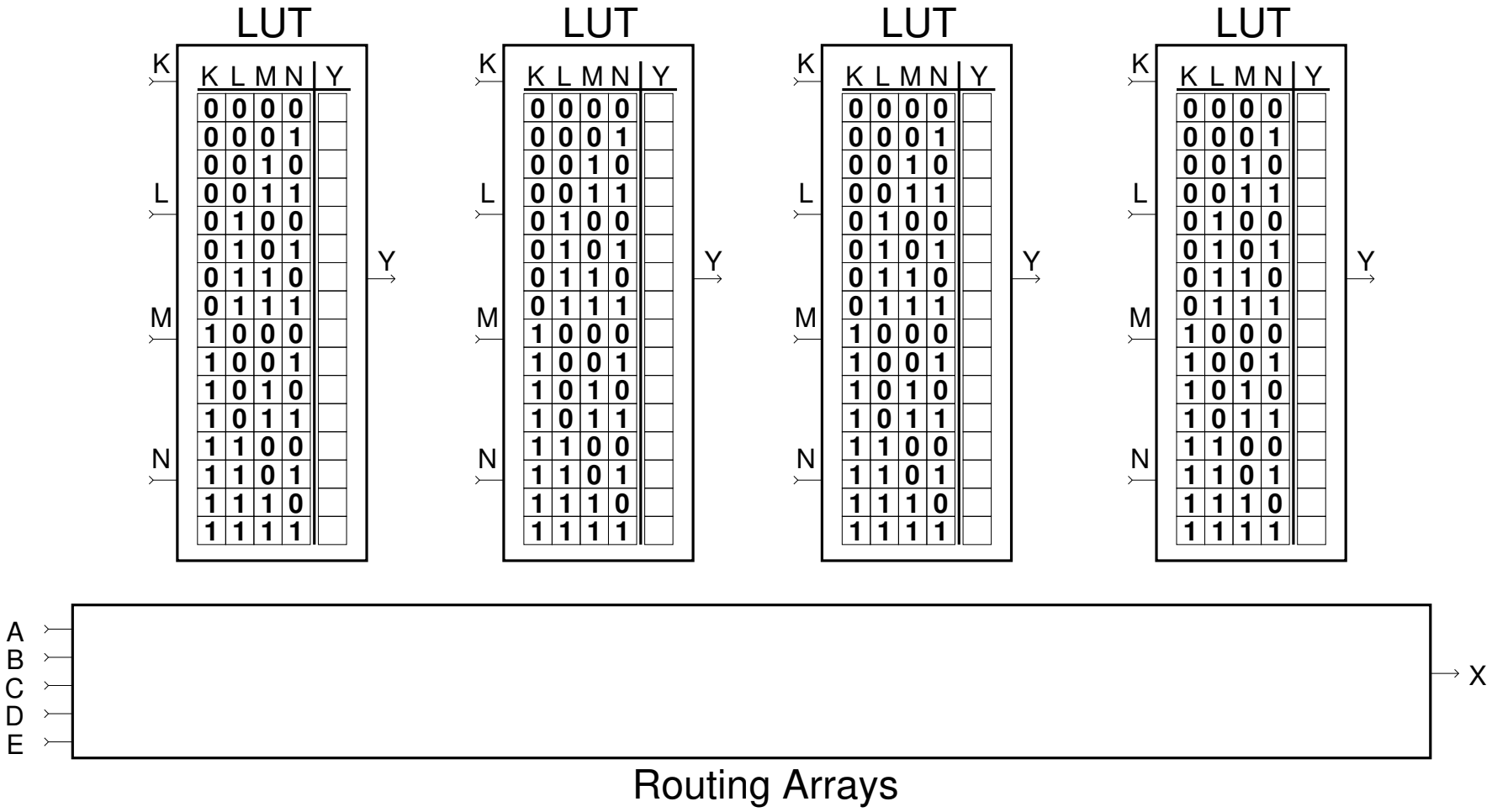
Exercise:

Given is the function X below. Realize this function on an FPGA.

E	D	C	B	A	X
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0

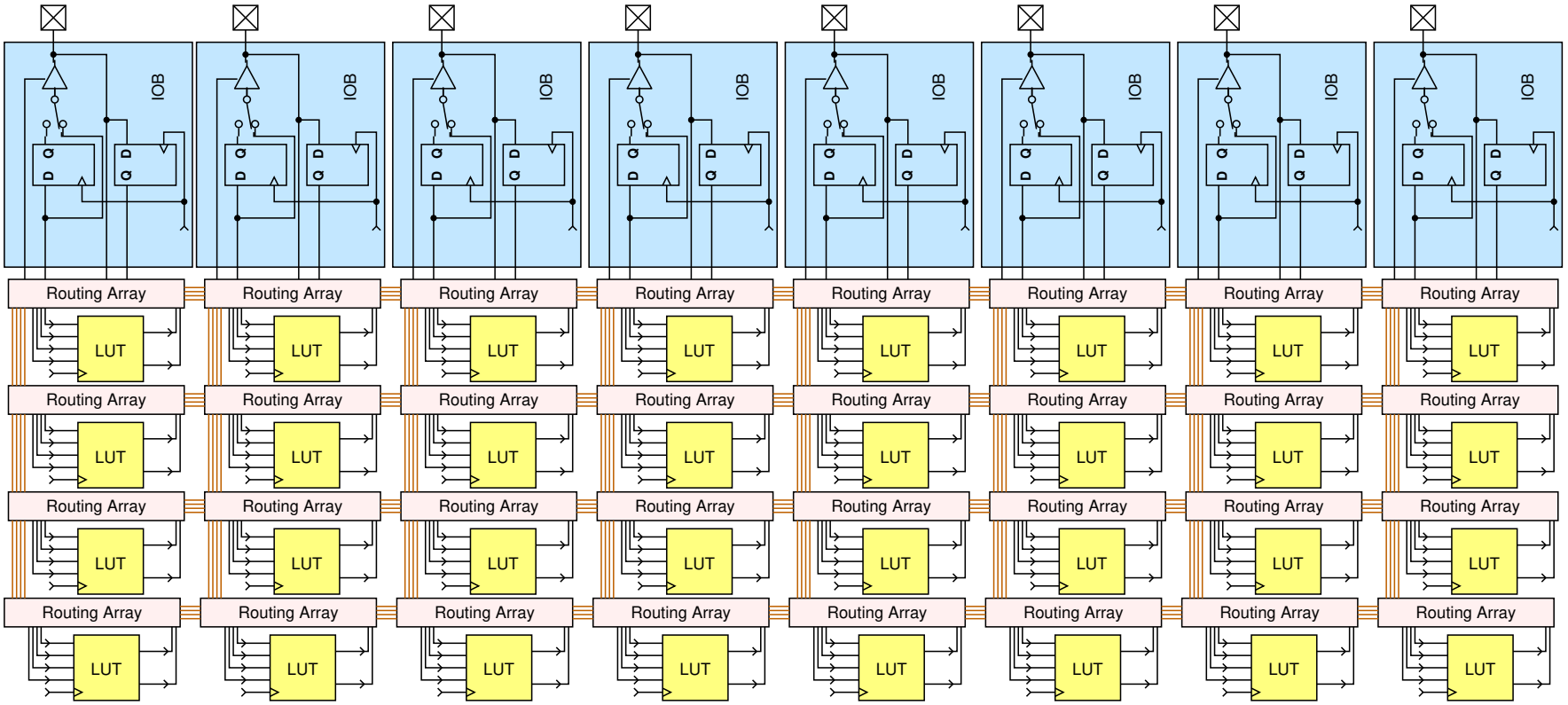
E	D	C	B	A	X
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	1
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	1

Exercise:


LUT
LUT
LUT
LUT
K
L
M
N
Y
K L M N Y
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
A
B
C
D
E
X
Routing Arrays

Input and Output

- But how do we connect the inputs and outputs to the outside world.
- Remeber the IOB(=*Input Output Block*) from the CPLD?
- The FPGA uses them also, they are connected to the “outer” routing arrays.
- Of course in the FPGA the E^2-switches are replaced by multiplexers and tri-state buffers.

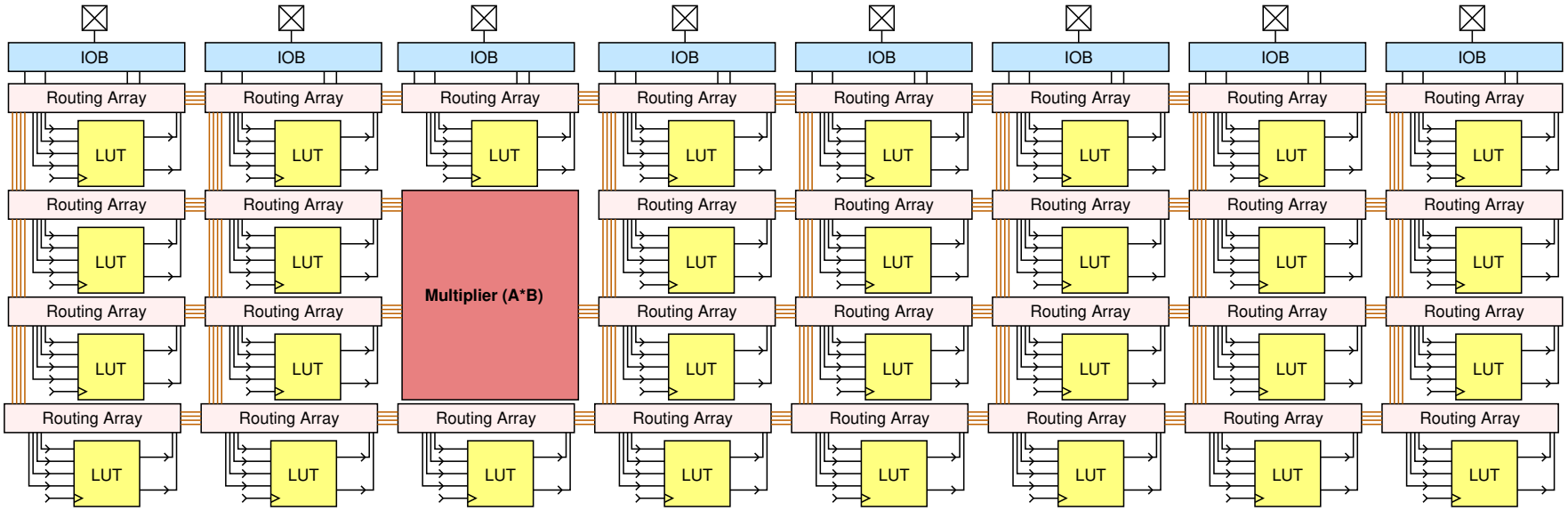


Summary

- So we see that the FPGA is very scalable (devices with 454240 LUT's are no exceptions).
- Contrary to the CPLD, the FPGA is *volatile* (remember the configuration pane)
- This means that if the power is switched off, the FPGA looses its functionality.
- Each time the FPGA is turned on we have to configure it. This is done by means of a *bit-file*.
- The bit-file contains the content for each of the D-flipflop in the configuration-pane. It also contains the "initial values" of the D-flipflops in the LUTs.
- The bit-file is generated by the P&R(Place and Route)-software.
- When we look at the time-delays inside an FPGA, it shows that due to the distributed routing-network, most of the delay is generated by the routing. In "big"-designs about 80% of the delay is routing delay, and 20% is LUT-delay.
- But is that all there is?

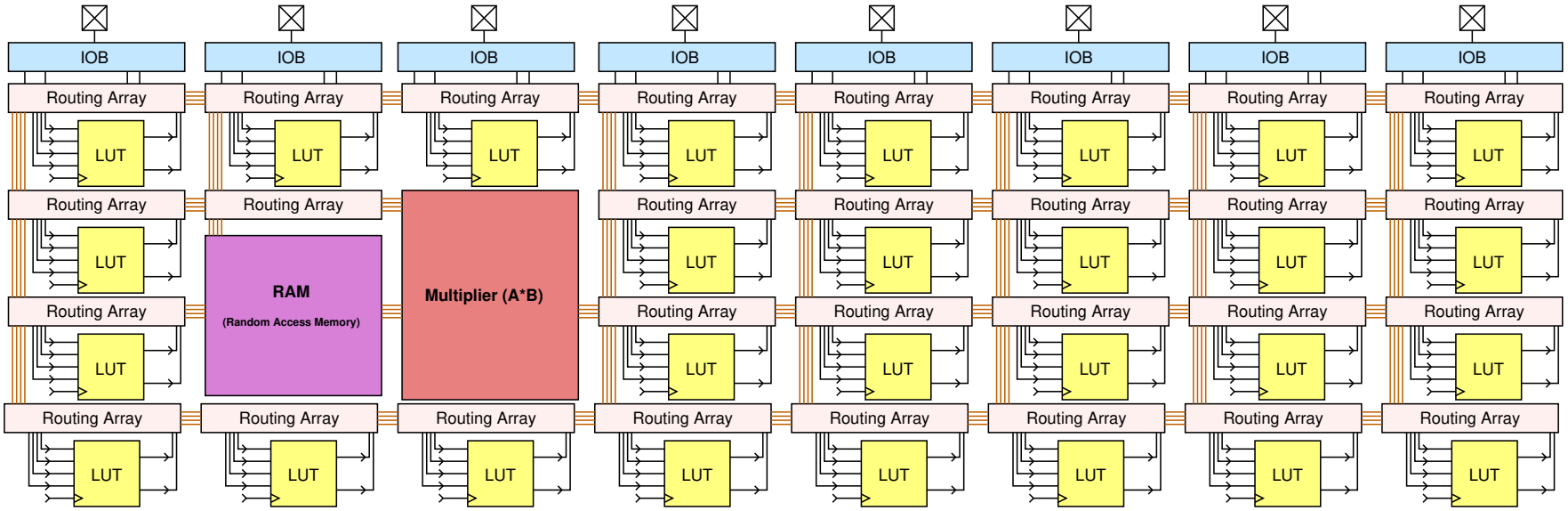
The evolving FPGA

- Of course not. In the 1990's this was about how an FPGA looked like.
- But soon the FPGA was used to implement calculations.
- Due to the routing- and LUT-delays these calculations were slow.
- Hence hardware (fast) multipliers were added to the FPGA.



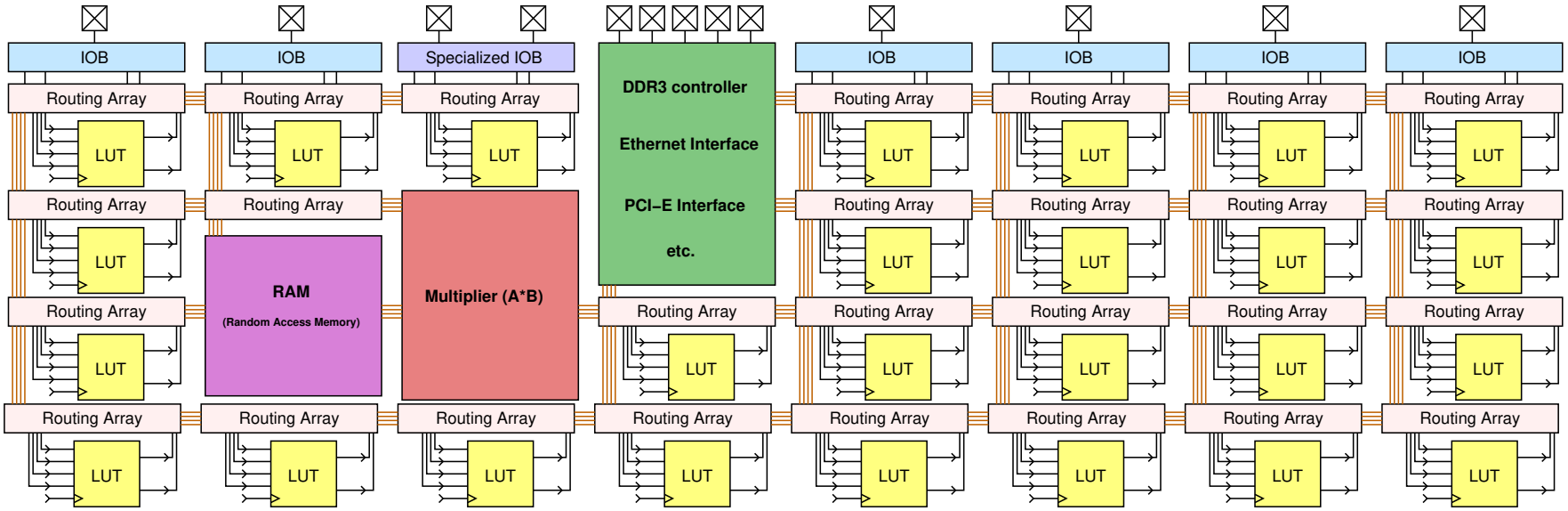
The evolving FPGA

- Having multipliers need's data.
- Using the flipflops in the LUT's is a waste of resources.
- So RAM(=*Random Access Memory*)'s were added to the FPGA.
- This architecture provided very fast MAC(=*Multiply ACumulate*)'s.



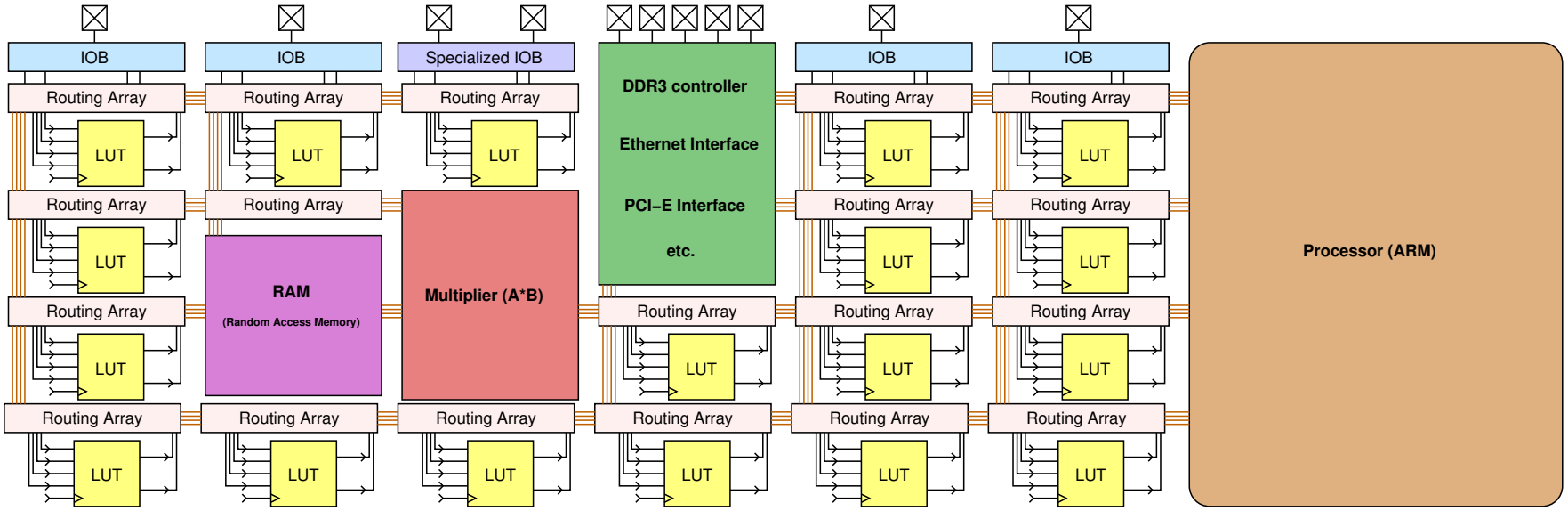
The evolving FPGA

- As technology evolved, the IOB's were not good enough.
- There was a requirement for faster IO.
- This has let to the integration of hardware IP(=*Intellectual Property*) blocks.
- And integration of specialized IO-blocks.



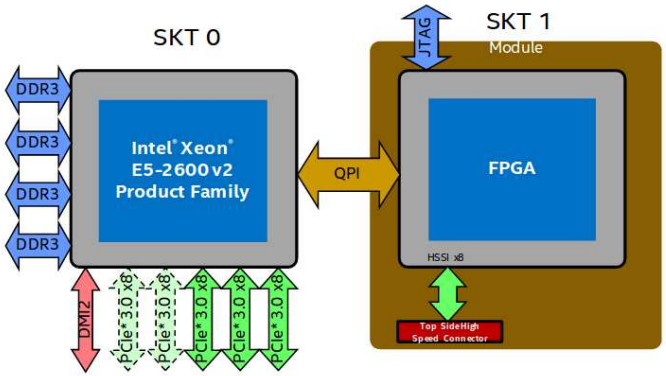
The evolving FPGA

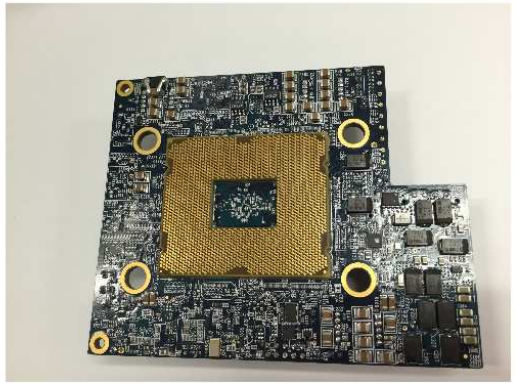
- Why putting a processor outside?
- We can also integrate it in the FPGA.
- The SOC(=*System On Chip*) was born.
- Examples of these SOC's are Xilinx's Zynq FPGA's and Altera/Intel's Cyclone V FPGA's.



Why FPGA

- This is all great, but why is the FPGA so successfull.
- Intel lanced in 2016 the HARP-program. They combined a Xeon processor with an FPGA.
- There must be a good reason for this.
- Intel even purchaised Altera, one of the two leaders in the FPGA-market.



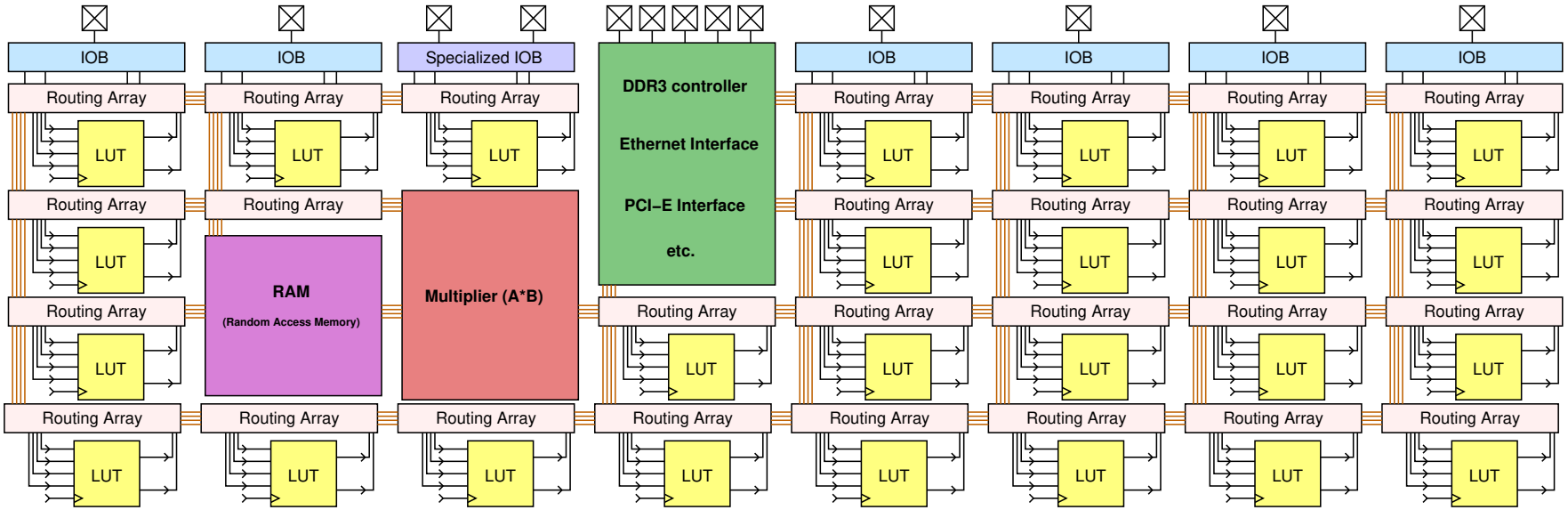


Why FPGA

- Let's see, an FPGA runs at approx. 300MHz-600MHz.
- That's slow, a Xeon runs upto a factor 10 faster.....
- Ah, on an FPGA I can do massive parallelism, so I gain.
- A GPU(=*Graphic Processing Unit*) does this also, but much faster....
- Ah, I got it, an FPGA can do massive parallelism and can be tuned to each task/thread.
- Indeed, the FPGA can be reconfigured during task/thread switching and can be tailored completely to given operations. Both GPU and CPU are fixed architecturs and are therefore limited.

How to use an FPGA

- But how to use all these resources....
- Doing it in a graphic-entry program like logisim is not very productive
- We need the means to design systems at a higher level
- For this reason nowadays a HDL(=*Hardware Description Language*) is used.
- There exists two flavours, namely (1) VHDL and (2) Verilog.
- From next week we will look into the details of VHDL.



Lecture 13

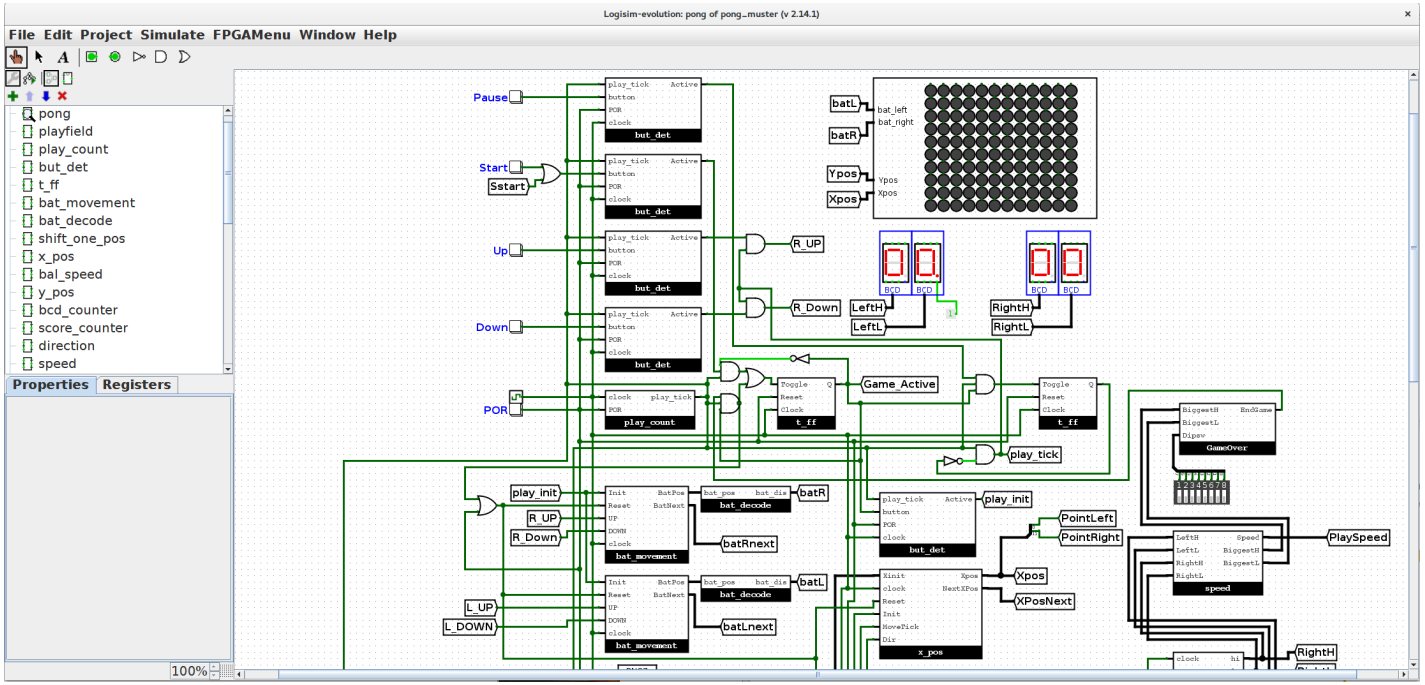
Digital system design

VHDL Introduction

CS173 - Conception de systèmes numériques
May 2018

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

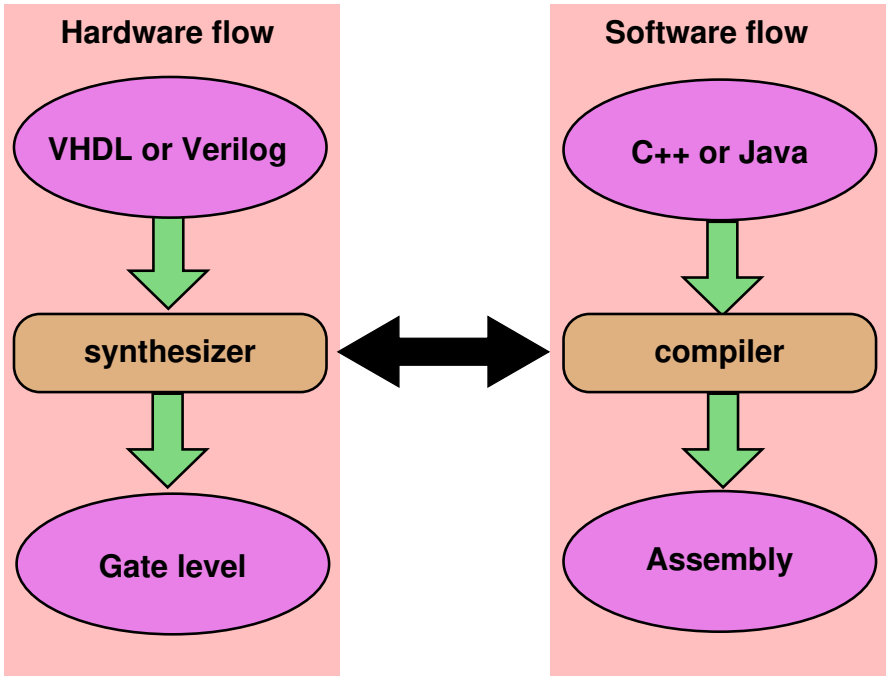
Introduction



- Is this the way to design a digital system?
- Yes, upto about 1985 this was used to design systems.
- For simple designs this method works, but as complexity increases new methodologies are required.
- Compare gate-level designing with writing programs in assembly language.
- We are not going to use gate-level design methods for 1 billion transistors!

Design flow

- So how to go about it, let's compare a software flow to a hardware flow.
- We use a high-level language to describe the functionality, than use a compiler that generates the assembly.
- At the same level as assembly we can think in hardware the gates.
- Hence we need a "compiler", which is called a synthesizer that can translate a hardware description to gates.
- These Hardware Description Languages(HDL's) are VHDL and Verilog.



VHDL

- We will look into the hardware description language VHDL.
- VHDL stands for: **V**ery high-speed integrated circuits **H**ardware **D**escription **L**anguage
- VHDL is a formal language for specifying digital systems, equally good at structural as behavioral level.
- VHDL is used for:
  - System description.
  - Simulation.
  - Conceptual modeling.
  - Documentation.
- VHDL's main characteristics are:
  - Hierarchical.
  - Event-driven simulation.
  - Modular.
  - Extensible.
  - General language, strongly typed, similar to Ada.

VHDL's history

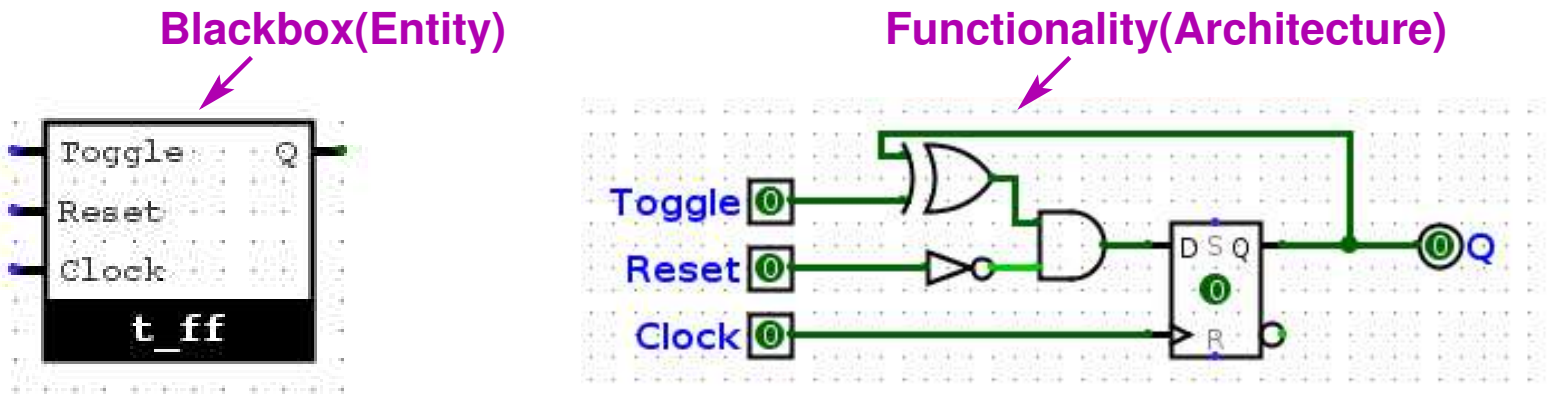
- **1980:**
  Beginning of the project, financed by the American Department of Defense(=DoD) with 400M $US.
- **1982:**
  Contracts for Intermetrics, IBM et Texas.
- **1985:**
  Version 7.2 released public domain.
- **1987:**
  Standarisation by the standard IEEE 1076 (VHDL-87).
- **1993:**
  New version of the standard by adding the standard IEEE 1164 (VHDL-93).
  Most commonly used nowadays.
- **2008:**
  Release of VHDL 4.0 by the standard IEEE 1076-2008.
  Many simplifications and *supported by most tools*.

VHDL's usage

- VHDL is a very rich language and provides syntactical elements for describing:
  - Synthesizable digital systems.
  - Functional description of complex components (processors, DSPs, etc.).
  - Description of libraries of elementary gates with internal delays rise and fall times, etc.
  - ...
- Many of VHDL's features were invented for system simulation.
- For hardware design only a small subset of the language is used, namely: the description of *synthesizable zero-delay digital systems*.
- VHDL's syntactical rules are:
  - VHDL is a **case insensitive** language.
  - VHDL has free formatting.
  - Each command sequence is **terminated by a ;** (note: there are exceptions).
  - A remark can be placed by using **-- in front of the remark**. A remark ends at the end of the line (<EOL>).

VHDL's elements

- VHDL consists of two main items:
  1. The **black-box** view of the digital circuit. This is described in the syntactical element called *Entity*.
  2. The **functionality** of the digital circuit. This is described in the syntactical element called *Architecture*.
- Let's start in looking into the **black-box** view.



The black-box view

- Each VHDL-description contains the view of the circuit as black-box. This is indicated by the **ENTITY** .
- Each Entity has some syntactical elements, namely:
  - ➥ A library definition. We will always use the one showed below (similar to **#include <stdlib.h>** in C/C++). Note: Each Entity requires it, as it is not inhereted in hierarchy like in C/C++.
  - ➥ A unique name . This name has to start with a letter `(A..Z,a..z)` and may *only* contain letters `(A..Z,a..z)`, numbers `(0..9)`, and underscores `(_)`.
  - ➥ Optionally a **PORT** section to define the input(s) and output(s) of the black-box.

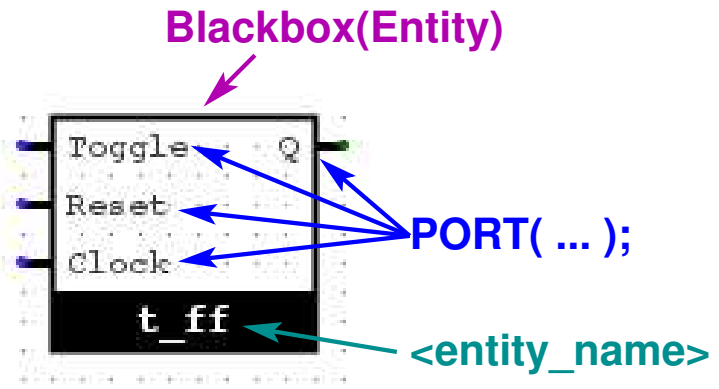
VHDL syntax definition:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY <entity_name> IS
    PORT ( ...  );
END <entity_name>;
```

Example

VHDL snippet:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY t_ff IS
    PORT (
        Toggle : IN std_logic;
        Reset  : IN std_logic;
        Clock  : IN std_logic;
        Q      : OUT std_logic);
END t_ff;
```



Ports

- The PORT-section contains all connections to the outside world.
- Each line in the PORT-section contains:
  - A unique name, or a list of unique names that have the same `<port_type>` and `<signal_type>` (as shown in the second line).
  - A type of the connection, valid port-types are **IN** , **OUT** , and **INOUT**. We will seldom (maybe never...) use the port type **INOUT**.
  - A signal type. We will look into details of the signal type a bit later.
- Important: As always there are exceptions, also here, the last line in the PORT-section is **not terminated** by a **;**

VHDL syntax definition:

```
PORT (
    <port_name>                 :   <port_type> <signal_type>;
    <port_name>,<port_name>     :   <port_type> <signal_type>;
        ...
    <port_name>                 :   <port_type> <signal_type>
    );
```

Signal types

- VHDL knows various types, like:
  - Real
  - Integer
  - Time
  - File
  - Character
  - ...

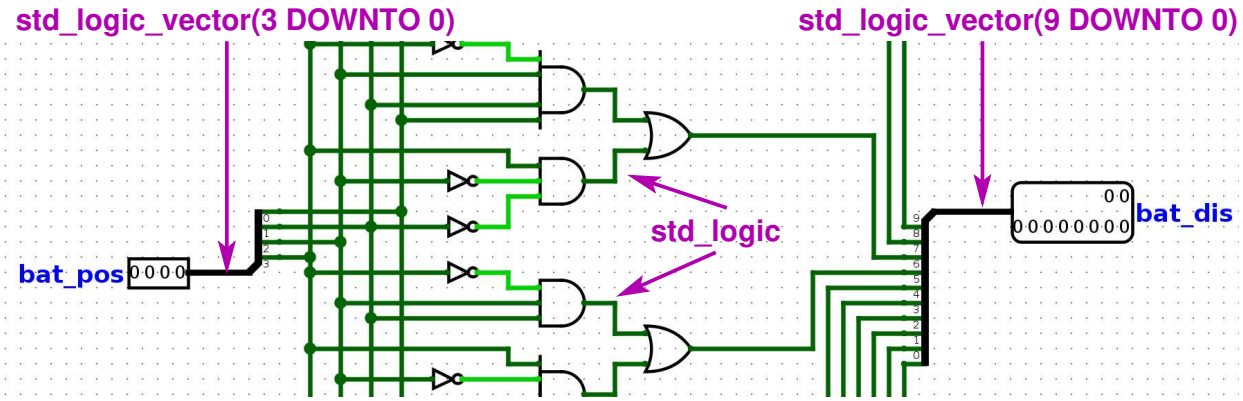
  However, these types have no real meaning in *synthesizable digital systems*.
- As we have seen in the first half of the course, there are two possible quantities, namely:
  - A **bit**. This is represented in VHDL by the type:

    std_logic
  - A **set of n-bits** (also called a bus). This is represented in VHDL by the type:

    std_logic_vector((n-1) DOWNTO 0)

  These quantities do not contain any interpretation (we will see later in the course that there also exists types that contain an interpretation).

Signal types

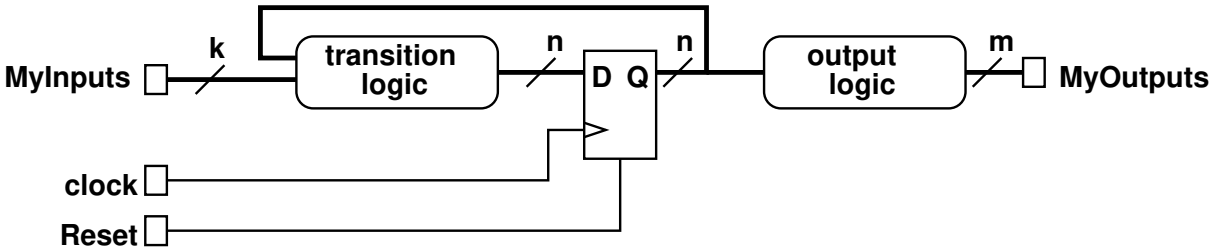


- A **std_logic**, and each bit of a **std_logic_vector** can hold nine values:
  - → '**0**' = logic 0.
  - → '**1**' = logic 1.
  - → '**U**' = Undefined state (for example of a D-flipflop).
  - → '**X**' = Unknow (often caused by a short circuit).
  - → '**-**' = Don't care (used in not completely defined functions).
  - → '**Z**' = high impedance (output of tri-state buffer).
  - → '**L**' = weak 0 (pull-down).
  - → '**H**' = weak 1 (pull-up).
  - → '**W**' = weak unknown (neither H nor L).

  The last four we will not see/use in this course.

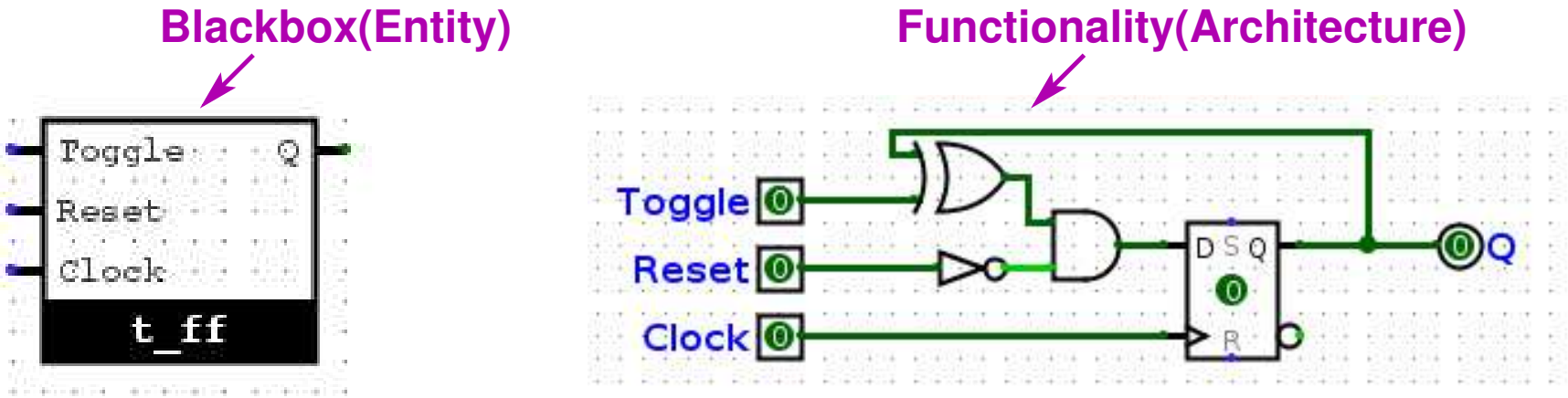
Exercise

Write the complete entity for the following state-machine called **moore**:



What's next

- We now have seen the first item of a VHDL-description, the **ENTITY**.



- Let's look into the details of the second part, the **ARCHITECTURE**.

Lecture 14

Digital system design

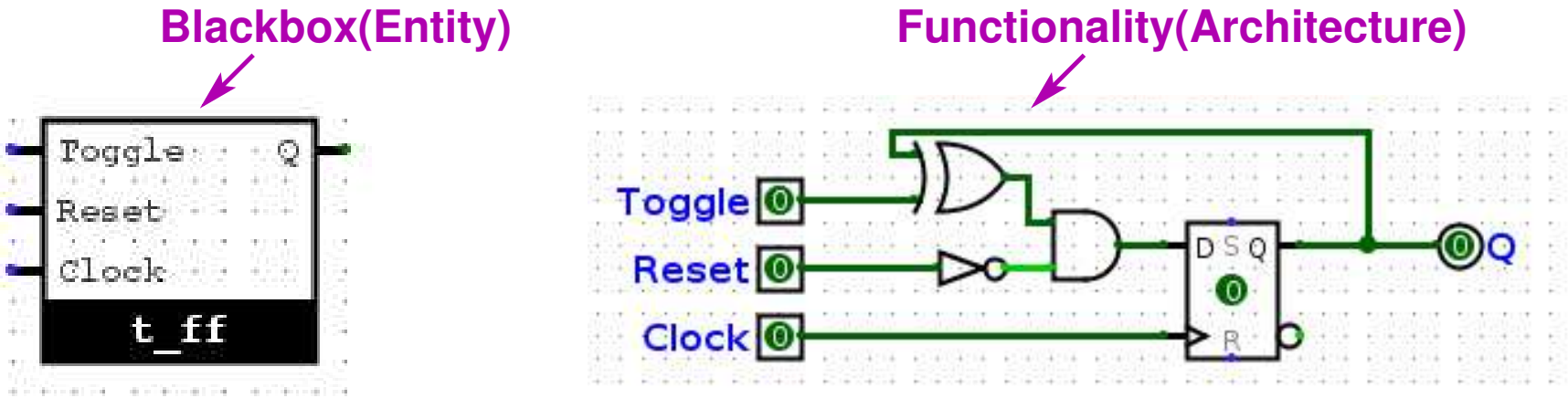
VHDL Behavior(1)

CS173 - Conception de systèmes numériques
May 2018

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Architecture

- Last time we saw the first item of a VHDL-description, the **ENTITY**.



- Let's look into the details of the second part, the **ARCHITECTURE**.

Architecture and entity relationship

VHDL snippet:

```
ENTITY exp IS
   PORT ( ... );
END example;
```

VHDL snippet:

```
ARCHITECTURE im1 OF exp IS
BEGIN
   ...
END example;
```

VHDL snippet:

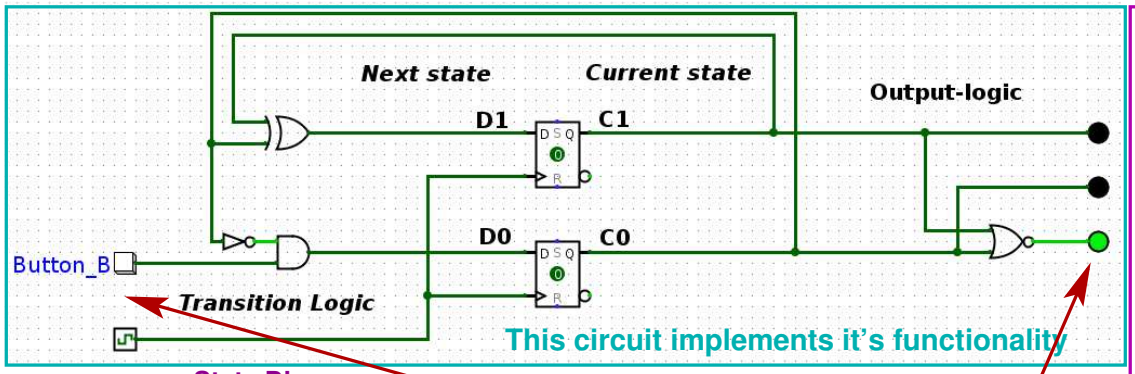
```
ARCHITECTURE im2 OF exp IS
BEGIN
   ...
END example;
```

VHDL snippet:

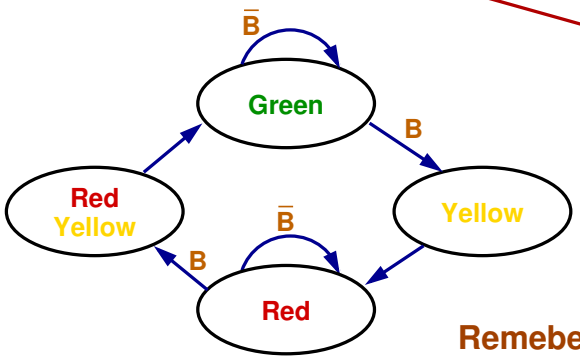
```
ARCHITECTURE im3 OF exp IS
BEGIN
   ...
END example;
```

- Each entity has at least one functional description (architecture).
- But an entity can have multiple functional descriptions (architectures).

Architecture and entity relationship (example)

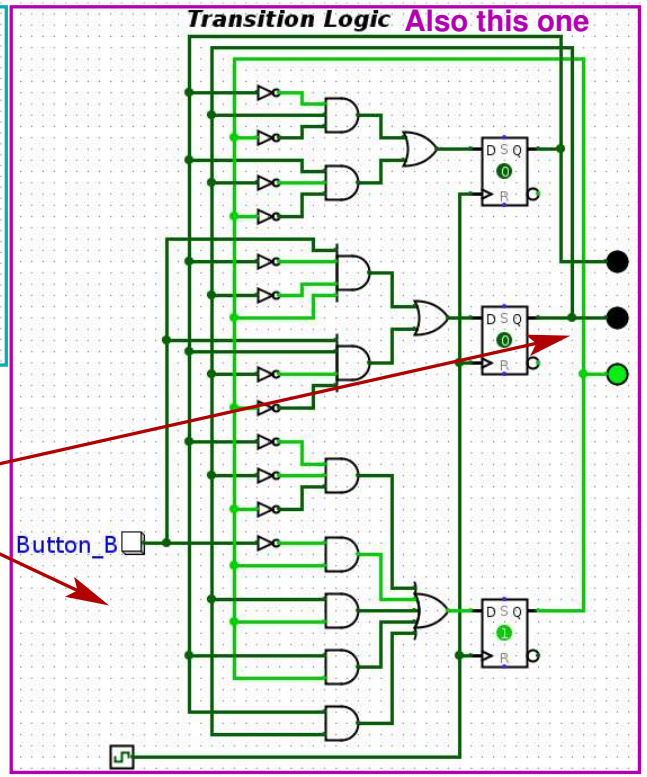


State Diagram:



Both have the same inputs and outputs (entity) but they have different implementations (architectures)

Remeber this one?



Architecture

VHDL syntax definition:

```
ARCHITECTURE <id_name> OF <entity_name> IS

    »declaration section«

BEGIN

    »description section«

END <id_name>;
```

- The architecture body is defined as above.
- Each architecture has a unique name (`<id_name>`).
- Each architecture has a reference to it's corresponding entity (`<entity_name>`).
- Each architecture has a declaration section.
- Each architecture has a description section.

Architecture: »declaration section«

- The *»declaration section«* of an **ARCHITECTURE** can contain:
  - **SIGNAL** declarations. Signals are the “wires” or “state” of the circuit.
  - **CONSTANT** definitions. Constants are fixed value “wires” or fixed value “state” within the circuit.
  - **COMPONENT** declarations. Components give us the possibility to design hierarchical.
  - **FUNCTION** definitions. Functions can be used for actions which are often required in the functional description.
  - **PROCEDURE** definitions. Similar to Functions also procedures can be used.
- In this course we will only use the **SIGNAL** , **CONSTANT** , and **COMPONENT** syntactical elements.
- Each of this topics will be explained later on in this course.

Architecture: »description section«

- The *»desciption section«* of an **ARCHITECTURE** contains:
  - *Implicite processes* also called *“concurrent statements”*.
  - *Explicite processes*.
  - Component instantiations.
- Each of these parts execute in parallel (in contrast with most software programming languages where instructions are executed sequentially)!
- Most VHDL code written and used nowadays by designers is written following the RTL (=Register Transfer Level) principle.
- The RTL principle consists of separating the sequential logic (e.g., memory, flip-flops and latches) from the combinatorial logic. Finally we connect the different parts by wires.
- How do we do this:
  - Identify the combinatorial and sequential elements.
  - Write the VHDL description for all these elements.
  - Control that no memory action is introduced in the combinatorial elements (to be elaborated later on in this course).
  - Connect the elements by using wires (e.g., signals).

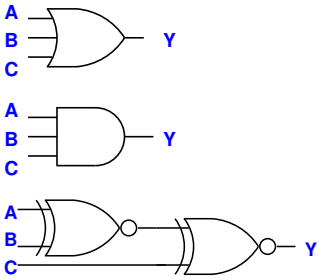
Architecture: »description section«; let's start simple: gates

Assume following entity:

VHDL snippet:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY gate IS
PORT ( A : IN std_logic;
   B : IN std_logic;
   C : IN std_logic;
   Y : OUT std_logic);
END gate;
```

How do we describe a gate in the *»description section»* of the corresponding architecture?



Y <= A **OR** B **OR** C;

Y <= A **AND** B **AND** C;

Y <= (A **XNOR** B) **XNOR** C;

- We can easily describe gates.
- Known gates are **AND**, **NAND**, **OR**, **NOR**, **NOT**, **XOR**, and **XNOR**.
- NOTE: The output of the gate is on the left hand side of the **<=** operator; the inputs of the gate are on the right hand side of the **<=** operator.

Exercise

Draw the circuit that is described by the below VHDL-description:

VHDL snippet:

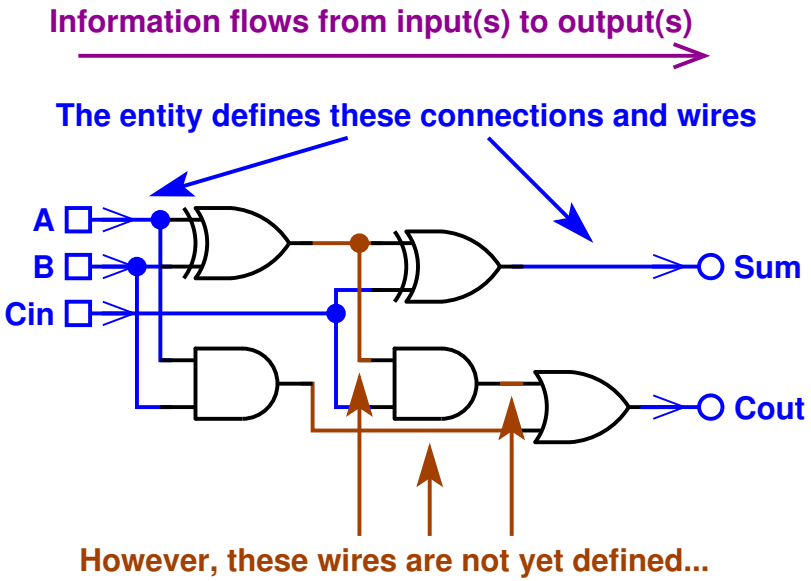
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY exercise IS
PORT (
    A,B,C : IN std_logic_vector(1 DOWNTO 0);
    X : OUT std_logic_vector(1 DOWNTO 0) );
END exercise;

ARCHITECTURE example OF exercise IS
BEGIN
    X  <= (A AND NOT (B)) NOR C;
END example;
```

Exercise

Architecture: more gates

VHDL snippet:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY full_adder IS
PORT ( A,B,Cin :  IN std_logic;
    Sum,Cout :  OUT std_logic);
END full_adder;
```

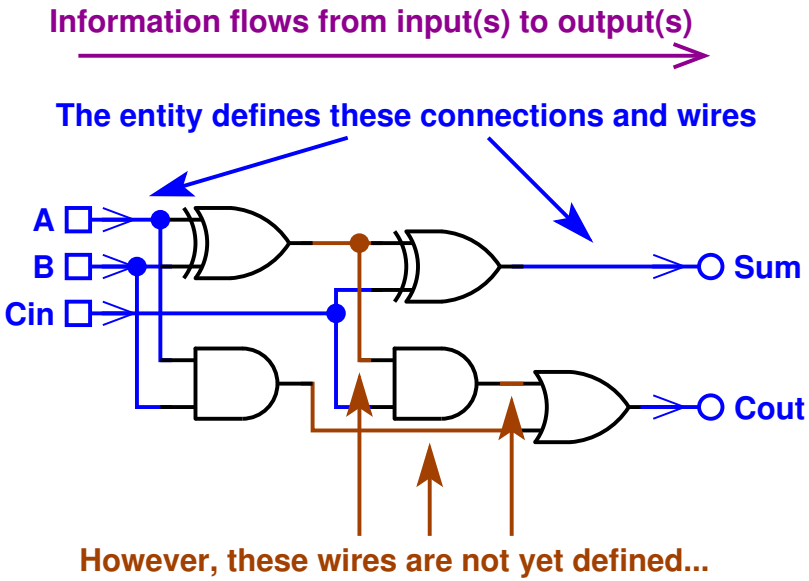


- Let's take a look at a more "complex" circuit.
- The information "flows" always from the inputs through the gates to the outputs; the inputs "write" the logic values and the outputs "read" the result.
- The entity defines the shown connection and wires in the architecture.
- However, there are wires in the architecture that are not visible at entity level, we have to define them!

Architecture: »declaration section«; signals and constants

VHDL snippet:

```
ARCHITECTURE gate OF full_adder IS

    »declaration section«

BEGIN
    ...
END gate;
```



- Each of these wires need to be defined in the *»declaration section«* of the architecture.
- Wires are represented in VHDL by **signals**, or in case of fixed-value wires by **constant**.

Architecture: »declaration section«; signals and constants

VHDL syntax definition:

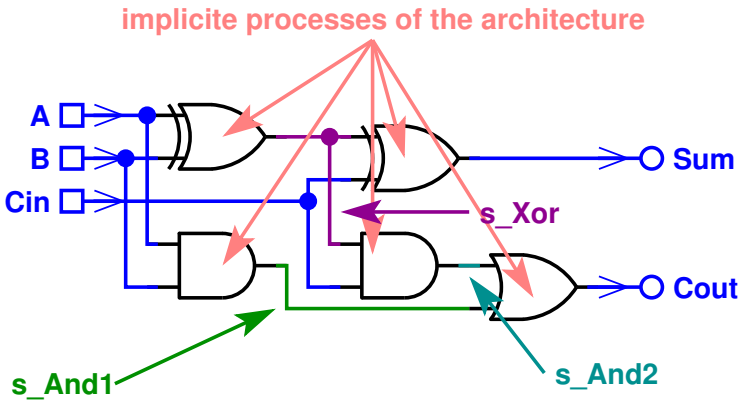
```
SIGNAL <signal_name>[,<signal_name>,..]  :  <signal_type>;
```

- Each signal has a unique name (<signal_name>) that has to start with a letter (a..z,A..Z), may contain numbers (0..9), letters (a..z,A..Z), and underscores (_).
- It is good practice to start each signal name with the prefix **s_**.
- It is also good practice to use camel-case for all names used in VHDL. E.g. the signal called “this is my nice signal” would be written as *s_ThisIsMyNiceSignal*.
- Each signal has to have a type (<signal_type>), e.g. std_logic or std_logic_vector (we will see others later on).
- Signals of the same <signal_type> can be concatinated by seperating their name by a comma.
- **Note:** Signals do not have a pre-defined signal flow (hence no **IN** or **OUT** after the **:**).

Architecture: more gates

VHDL snippet:

```
ARCHITECTURE gate OF full_adder IS

SIGNAL s_Xor,s_And1,s_And2 :  std_logic;

BEGIN
    Cout <= s_And1 OR s_And2;
    s_And1 <= A AND B;
    Sum <= s_Xor XOR Cin;
    s_And2 <= s_Xor AND Cin;
    s_Xor <= A XOR B;
END gate;
```



- Let's name the wires.
- We can combine the signal definitions also like this.
- Now we can describe the system.
- **Note:** The order in which we put the implicite processes is of no importance as harware is parallel!

Architecture: more gates

VHDL snippet:

```
ARCHITECTURE gate1 OF full_adder IS

SIGNAL s_Xor,s_And1,s_And2 :  std_logic;

BEGIN
    Cout <= s_And1 OR s_And2;
    s_And1 <= A AND B;
    Sum <= s_Xor XOR Cin;
    s_And2 <= s_Xor AND Cin;
    s_Xor <= A XOR B;
END gate1;
```

VHDL snippet:

```
ARCHITECTURE gate2 OF full_adder IS

SIGNAL s_Xor,s_And1,s_And2 :  std_logic;

BEGIN
    Cout <= s_And1 OR s_And2;
    s_And1 <= A AND B;
    s_And2 <= s_Xor AND Cin;
    s_Xor <= A XOR B;
    Sum <= s_Xor XOR Cin;
END gate2;
```

VHDL snippet:

```
ARCHITECTURE gate3 OF full_adder IS

SIGNAL s_Xor,s_And1,s_And2 :  std_logic;

BEGIN
    s_And2 <= s_Xor AND Cin;
    s_Xor <= A XOR B;
    Cout <= s_And1 OR s_And2;
    s_And1 <= A AND B;
    Sum <= s_Xor XOR Cin;
END gate3;
```

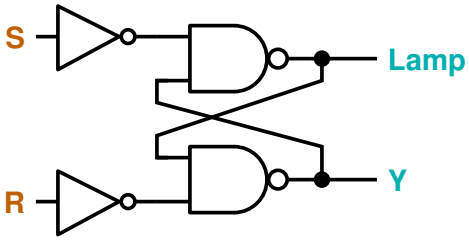
VHDL snippet:

```
ARCHITECTURE gate4 OF full_adder IS

BEGIN
    Cout <= (A AND B) OR
        (Cin AND (A XOR B));
    Sum <= (A XOR B) XOR Cin;
END gate4;
```

- All these descriptions are correct. We can suppress wires (but is not required).

Exercise

Remember this one?



R	S	Lamp	Y
		$\overline{Y}$	$\overline{Lamp}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	1	1

Write the complete VHDL description of this latch.

Pitfalls (1)

Entity:

VHDL snippet:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY RS_LATCH IS
PORT ( S,R : IN std_logic;
    Lamp,Y : OUT std_logic);
END RS_Latch;
```

Architecture:

VHDL snippet:

```
ARCHITECTURE wrong OF RS_LATCH IS

BEGIN
    Lamp  <= (NOT (S))NAND Y;
    Y   <= (NOT (R))NAND Lamp;
END wrong;
```

- We could think that this is correct. However, Lamp and Y are outputs and their value can only be "assigned", not read!
- We have to introduce two signals, s_lamp and s_Y.

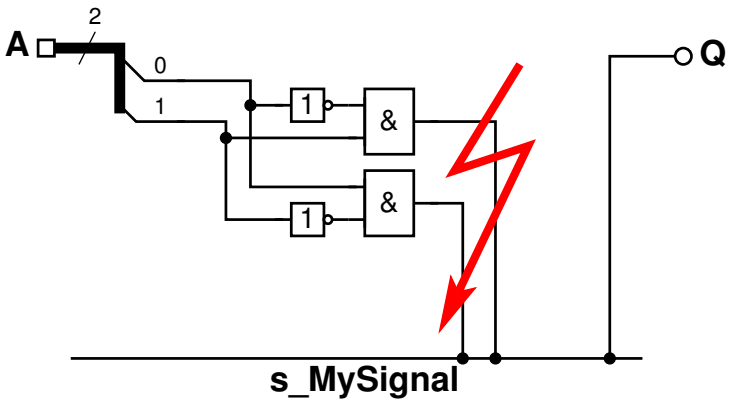
- The correct version is than:

VHDL snippet:

```
ARCHITECTURE Correct OF RS_LATCH IS
SIGNAL s_Lamp,s_Y : std_logic;
BEGIN
    s_Lamp  <= (NOT (S))NAND s_Y;
    s_Y   <= (NOT (R))NAND s_Lamp;
    -- Don't forget to assign the outputs!
    Y   <= s_Y;
    Lamp  <= s_Lamp;
END Correct;
```

Exercise (2)

Given following VHDL description:



VHDL snippet:

```
ARCHITECTURE common OF DoesItWork IS

SIGNAL s_MySignal :  std_logic;

BEGIN
    s_MySignal  <= NOT ( A(0) ) AND A(1);
    Q  <= s_MySignal;
    s_MySignal  <= A(0) AND NOT ( A(1) );
END common;
```

Draw the corresponding circuit:

- The entity gives us this. The »declaration section« of the architecture adds a wire.
- We can select each bit of a std_logic_vector by using the bit index in **( )**. Therefore the first *implicite process* in the architecture adds this circuit.
- The second *implicite process* adds this circuit.
- And finally the third *implicite process* adds this circuit.
- **Boooom**. As the signal *s_MySignal* appears twice on the left hand side of the **<=** operator a short circuit is generated! This is a common pitfall.

Architecture: signals and constants

VHDL syntax definition:

```
CONSTANT <constant_name> :  <constant_type>  := <value>;
```

- We can also define constants.
- It is good practice to start each constant name with the prefix **c_**.
- Each constant has to have a type (<constant_type>), e.g. std_logic or std_logic_vector (we will see others later on).
- A signal/constant of type std_logic is assigned a constant value by using ', example:
  - **CONSTANT** c_example1 : std_logic **:=** '1'; *--(in the »declaration section«)*
  - s_MySuperSignal **<=** 'Z'; *--(in the »description section«)*
- A signal/constant of type std_logic_vector is assigned a constant value by using **" "**, example:
  - **CONSTANT** c_example2 : std_logic_vector(2 **DOWNTO** 0) **:=** "110";
  - s_MySuperBus **<=** "0100";

Architecture: signals and constants

- Assume following signal defition in the »declaration section« of an architecture:

```
SIGNAL s_MyExampleBus : std_logic_vector( 23 DOWNTO 0 );
```

- To assign a constant to it we can do it like:

```
s_MyExampleBus  <= "100000000000000000000000";
```

- We can simplify this by using the hexadecimal representation like this:

```
s_MyExampleBus  <= X"800000";
```

  IMPORTANT: When using the hexadecimal representation each *digit* represents exactly 4 bits!

- We can also use a combination between the two by using concatination (**&** as glue element):

```
s_MyExampleBus  <= X"80"&"0000"&X"000";
```

  This is equal to:

```
s_MyExampleBus(23 DOWNTO 16)  <= X"80";
s_MyExampleBus(15 DOWNTO 12)  <= "0000";
s_MyExampleBus(11 DOWNTO 0)  <= X"000";
```

- Or we can use the macro **OTHERS**:

```
s_MyExampleBus  <= ( 23  => '1', OTHERS => '0' );
```

Lecture 15

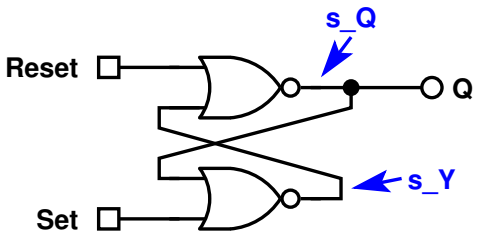
Digital system design

VHDL Behavior(2)

CS173 - Conception de systèmes numériques
May 2018

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

System



VHDL snippet:

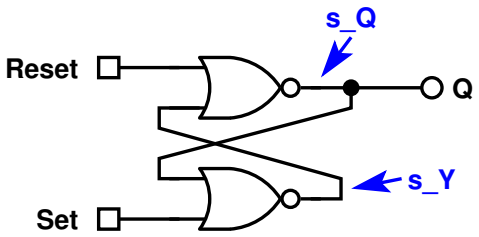
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY SR_LATCH IS
PORT ( Set,Reset :  IN std_logic;
   Q : OUT std_logic);
END SR_LATCH;
```

VHDL snippet:

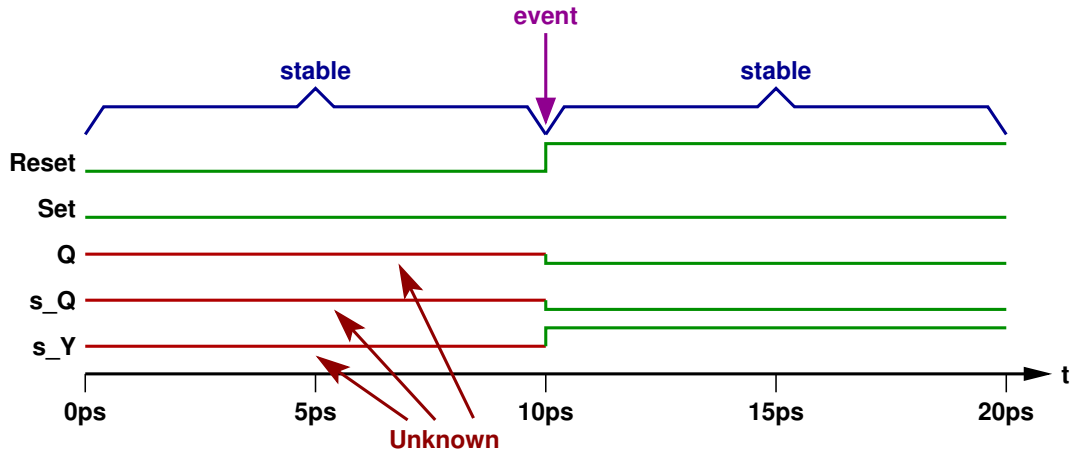
```
ARCHITECTURE simple OF SR_LATCH IS
SIGNAL s_Q,s_Y : std_logic;
BEGIN
   Q  <= s_Q;
   s_Q  <= reset NOR s_Y;
   s_Y  <= set NOR s_Q;
END simple;
```

- We can describe the above system with three *implicite processes*.
- We know that hardware is parallel. We also know that a computer is sequential.
- So how do we simulate a parallel system on a sequential machine...

Simulation

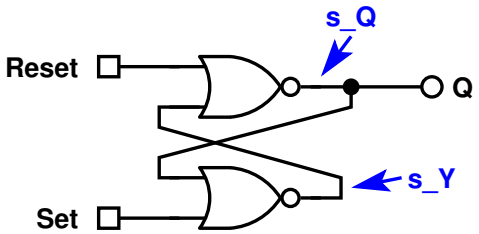


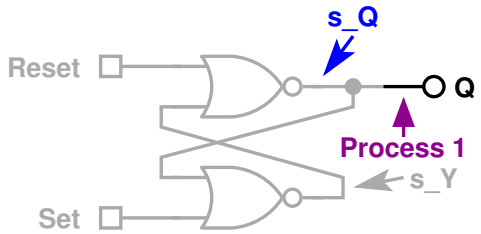

```
force reset 0 0 , 1 10
force set 0
run 20
```



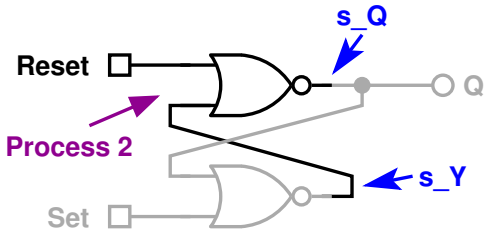
- Lets look into the simulation.
- Of course in the beginning most signals are undefined (**U**); remember logisim also.
- Most of the time the signals are stable, hence no new values need to be calculated.
- Only if a signal changes (we call this an *event*) the state of the system may change.
- For simulation purposes we use hence an *event-driven* approach.

Simulation

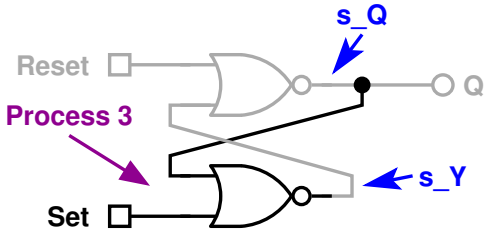




Q <= s_Q;



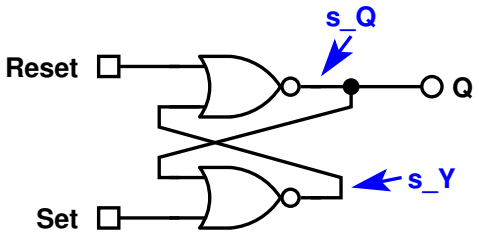
s_Q <= reset **NOR** s_Y;

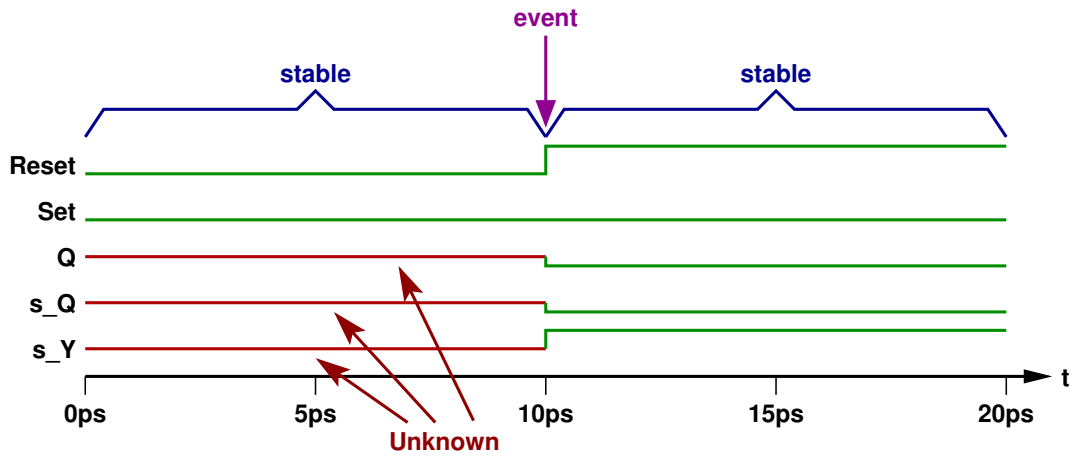


s_Y <= set **NOR** s_Q;

- To explain *event-driven* simulation the notion of *sensitivity* is introduced.
- The output Q of process 1 is sensitive to a change on s_Q.
- Hence each process is *sensitive* to the signals right of the <= symbol.

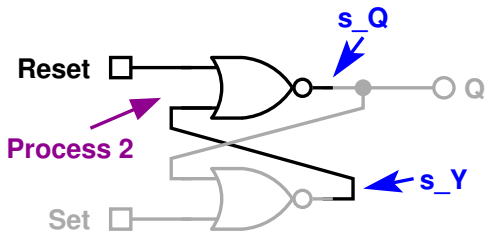
Simulation

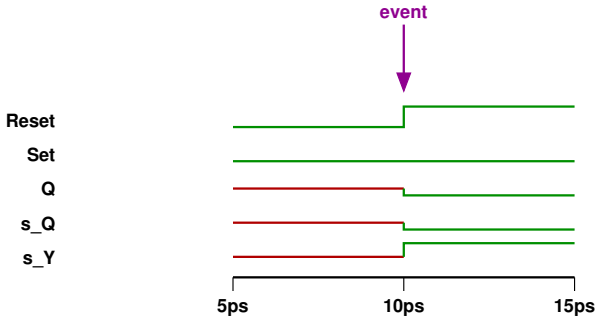




- So in case of an event, only the processes that are *sensitive* to this event need to be taken into account.
- However, this event may trigger another event, trigering process 1 and process 2!
- And this process continues until a new “stable” situation is found; hence no more *resulting events* are generated.
- **Note:** There is in this event-triggering no notion of time (as we look at the zero-delay model); it is an effect of simulating a parallel system on a sequential machine.

Simulation: δ-iterator





δ	s_Q$_\delta$
0	**u**
1	**0**
2	**0**

- At the event process 2 is triggered as it is sensitive to reset.
- At this moment two things happen, namely:
  - A sequential δ-iterator is initialised for all signals in the process left of the **<=** operator.
  - These signals get now two values, namely the value seen before the event, and the resulting δ-value.
- One δ-step is taken, determining the resulting value of the event.
- As s_Q$_{\delta=1} \neq$ s_Q$_{\delta=0}$ a resulting event is “fired” triggering process 1 and 3.
- Another δ-step is taken. As now s_Q$_{\delta=2} =$ s_Q$_{\delta=1}$ a new stable situation is found, and s_Q is assigned the value of s_Q$_{\delta=2}$.

Simulation: δ-iterator

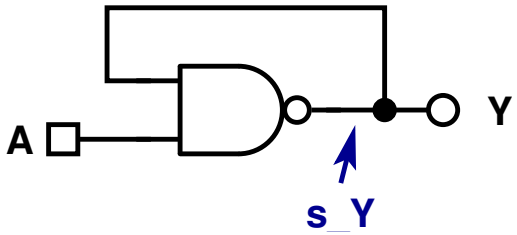
Tr=*Triggered by an event*, Fi=*Fire an event*, St=*Stable*

δ	Q <= s_Q;				s_Q <= reset **NOR** s_Y;				s_Y <= set **NOR** s_Q;			
	Tr	Q_δ	Fi	St	Tr	s_Q_δ	Fi	St	Tr	s_Y_δ	Fi	St
0		**u**		**X**	**X**	**u**				**u**		**X**
1				**X**		**0**	**X**					**X**
2	**X**					**0**		**X**	**X**			
3		**0**	**X**					**X**		**1**	**X**	
4		**0**		**X**	**X**					**1**		**X**
5				**X**		**0**		**X**				**X**

- The initial situation at the event for the whole system.
- Process 2 fires a resulting event.
- Process 1 and 3 get triggered by the resulting event.
- Process 2 gets triggered by the resulting event of process 3.
- A new stable situation is found, hence the last values assigned are the resulting values.

Exercise

Given circuit below, fill out the δ table below




```
force a 0 0 , 1 10
run 20
```

δ	Y <= s_Y;				s_Y <= a **NAND** s_Y;			
	Tr	Y_δ	Fi	St	Tr	s_Y_δ	Fi	St
0								
1								
2								
3								
4								
5								
6								
7								
8								

Simulation

- As demonstrated in the previous exercise, not all zero-delay systems can be simulated.
- In case of a non-zero delay model (hence the gates have a gate-delay) the *fire* is delayed by the gate-delay time.
- Non-zero delay VHDL description, however, are not synthesizable.
- Simulating *implicite processes* is very calculation intensive as they are *sensitive* to all input signals of the process.
- In the time VHDL was developed, the computers were not very fast, hence a simulation could take hours to complete.
- To reduce the calculation time the concept of the *explicite process* was introduced.

Explicite Process

VHDL syntax definition:

```
<process_identifier>:  PROCESS ( <sensitivity_list>) IS
BEGIN
    »process body«
END PROCESS <process_identifier>;
```

- An *explicite process* is used simular as an *implicite process* in the »description section«of an **ARCHITECTURE** .
- Each explicite process may have a unique identifier.
- Each explicite process has a sensitivity list.
- The functionality is described in the process body.
- Each implicite process can be writen in it's explicite form:

Implicite form:

VHDL snippet:

```
Z  <= A NAND B;
```

Explicite form:

VHDL snippet:

```
test :  PROCESS ( A,B ) IS
BEGIN
   Z  <= A NAND B;
END PROCESS test;
```

Explicite Process

Implicite form:

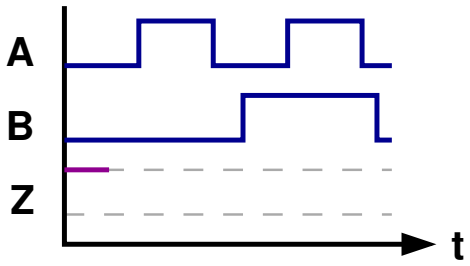
VHDL snippet:

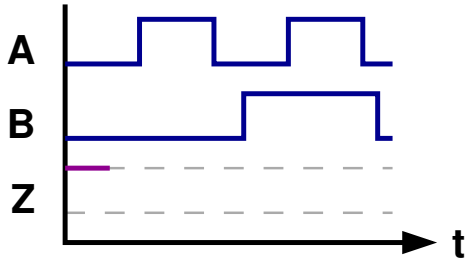
```
Z  <= A NAND B;
```

Explicite form:

VHDL snippet:

```
test1 :  PROCESS (B) IS
BEGIN
   Z  <= A NAND B;
END PROCESS test1;
```





- *Explicite processes* are particular in the sense that in simulation it will **only** trigger on events of the signals in it's sensitivity list.
- In the above vhdl-snippets, the implicite process will trigger on events on A and B.
- Whereas the explicite process will *only* trigger on B, as only B is in it's sensitivity list.
- Exercise: Complete the simulation of the above two processes.

Explicite Process

Implicite form:

VHDL snippet:

```
Z  <= A NAND B;
```

Explicite form:

VHDL snippet:

```
test1 :  PROCESS (A,B) IS
BEGIN
   Z  <= A NAND B;
END PROCESS test1;
```

- The explicite process in the previous exercise does *not* simulate as being a NAND-gate!
- However, both processes describe for the synthesizer a NAND-gate, and hence we have a difference between simulation and the hardware realized!
- To allign simulation with the hardware realized we have to put all “inputs” of the logic in the sensitivity list of an explicite process! Hence in this case we have to add A.
- Okay, but why then using the explicite form, it is way more work to write it.....
- It has to do with the behaviour of the explicit process’s body, let’s look into detail.

Explicite Process's body

- To understand the behaviour of an explicite process we have to remember the δ-iterator.
- What happened in simulation:
  - An event triggers the simulator.
  - All signals are replaced by their $\delta = 0$-representatives (e.g. $A_{\delta=0}$).
  - The simulator takes δ-iteration steps determining eventual new values for the δ-signal-representatives (e.g. A_δ).
  - This δ-iteration continues until a new *stable* situation is found.
  - All signals are assigned the value of their latest δ-representative (e.g. $A = A_\delta$).
- The sequential nature of this δ-evaluation is used in the body of an explicite process.
- Huh.... I do not understand.... Let's look at an example:

Explicite Process's body

VHDL snippet:

```
example :  PROCESS (A,B) IS
BEGIN
    Z  <= NOT (A);
    Z  <= '0';
    Z  <= NOT (A) AND B;
    Z  <= A NOR B;
END PROCESS example;
```

- Which hardware is described in the above explicite process?
- I know: **Booooooooooooom**!
- Actually: no, the above explicite process describes a NOR-gate.
- So how does it work:

Explicite Process's body

VHDL snippet:

```
example :  PROCESS (A,B) IS
BEGIN
(1)    Z_δ   <= NOT (A_δ=0);
(2)    Z_δ   <= '0';
(3)    Z_δ   <= NOT (A_δ=0) AND B_δ=0;
(4)    Z_δ   <= A_δ=0 NOR B_δ=0;
END PROCESS example;
```

- To understand the behaviour of the process we have to put it in the δ-iterator form.
- The moment we have an event on either A or B all three signals are transformed into their $\delta = 0$-representatives (e.g. $A_{\delta=0}$, $B_{\delta=0}$, and $Z_{\delta=0}$).
- On each δ-iteration the simulator will determine eventual new values of the δ-representatives (in this case Z_δ).
- In an explicite process this calculation is done by execution one line after the other (hence first line (1), then (2), etc.).
- Finally when the stable situation is reached the signal is assigned the value of it's δ-representative (e.g. $Z=Z_\delta$).

Explicite Process's body

VHDL snippet:

```
example :  PROCESS (A,B) IS
BEGIN
(1)    Z_δ   <= NOT (A_δ=0);
(2)    Z_δ   <= '0';
(3)    Z_δ   <= NOT (A_δ=0) AND B_δ=0;
(4)    Z_δ   <= A_δ=0 NOR B_δ=0;
END PROCESS example;
```

- There are two observations to be made:
  - The values seen on the right hand side of the **<=** operator are always the values at δ-iteration 0 (hence the values seen at the event).
  - The lines in an explicite process are evaluated in sequential order, hence their position does matter for the functionality.
- In this example the first three lines the process are redundant, as they do not describe any functionality. Only the last line describes the functionality, and hence the hardware!

Exercise

Which hardware is described by the following explicite process?

VHDL snippet:

```
exercise :  PROCESS (A,B,C) IS
BEGIN
    IF (A = '1') THEN
        X  <= B;
        Y  <= C;
    ELSE
        X  <= C;
    END IF;
END PROCESS exercise;
```

Memory or no memory

VHDL snippet:

```
exercise :  PROCESS (A,B,C) IS
BEGIN
(1)    IF (A_δ=0 = '1') THEN
(2)        X_δ  <= B_δ=0;
(3)        Y_δ  <= C_δ=0;
(4)    ELSE
(5)        X_δ  <= C_δ=0;
(6)    END IF;
END PROCESS exercise;
```

- Let us review this exercise.
- We put it in the δ-iterator representation.
- We see that X_δ is always assigned a value. Either in line (2) when $A_{\delta=0}$='1', or in line (5) when $A_{\delta=0}$='0'. X represents, therefore, combinational logic.
- We see also that Y_δ is only assigned a value in line (3) when $A_{\delta=0}$='1'. If $A_{\delta=0}$='0', Y_δ is not assigned a new value meaning that Y <= $Y_{\delta=0}$; the signal retains it's "old" value, we have memory. In this case we have the behaviour of a D-latch.

Exercise

Which hardware is described by the following explicite process?

VHDL snippet:

```
exercise :  PROCESS (A,B) IS
BEGIN
    Y  <= '0';
    IF (A = '1') THEN
        Y  <= B;
        X  <= B;
    END IF;
    X  <= '0';
END PROCESS exercise;
```

Exercise

Which hardware is described by the vhdl description?

VHDL snippet:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Example IS
PORT ( A,B : IN std_logic,
    Y : OUT std_logic);
END Example;

ARCHITECTURE test OF example IS
BEGIN

    Y  <= A AND B;

    proc1 :  PROCESS (A,B) IS
    BEGIN
        IF (A = '1') THEN
            Y  <= B;
        END IF;
    END PROCESS proc1;
END test;
```

Does it matter in which order the processes are placed?

Lecture 16

Digital system design

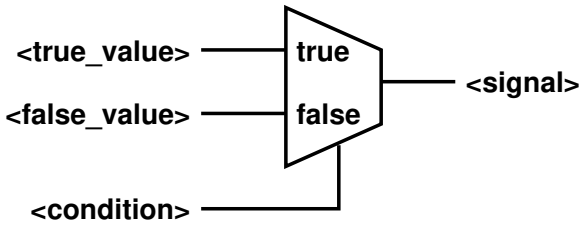
VHDL Behavior(3)

CS173 - Conception de systèmes numériques
May 2018

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Multiplexers

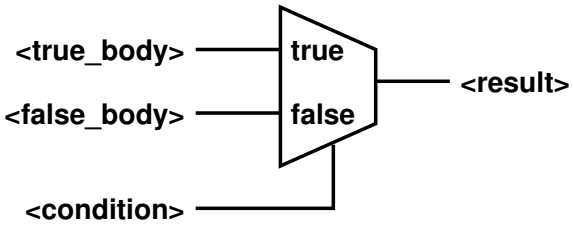
Implicite form:



VHDL syntax definition:

```
<signal>  <= <true_value>
                WHEN <condition> ELSE
             <false_value>;
```

Explicite form:



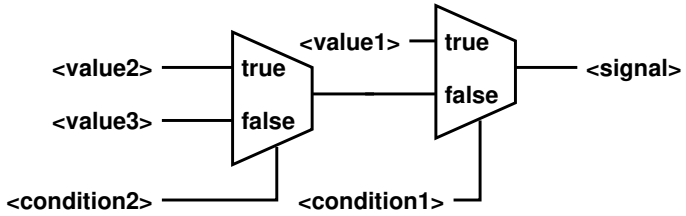
VHDL syntax definition:

```
IF <condition> THEN
   <true_body>
END IF;
```

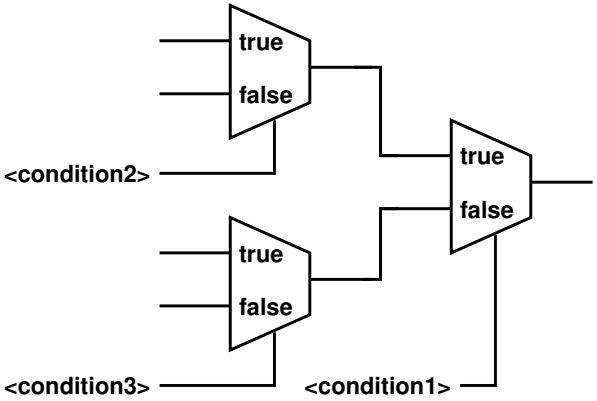
- In the implicite form only one signal can be defined.
- The explicite form allows for multiple statements in both the `<true_body>` as well as in the `<false_body>`.
- The implicite form **requires** the **ELSE** , whereas the explicite form allows for only the `<true_body>` realizing possibly one/multiple memory(es).
- The condition is evaluated “boolean”.

Multiplexers

Implicite form:



Explicite form:



VHDL syntax definition:

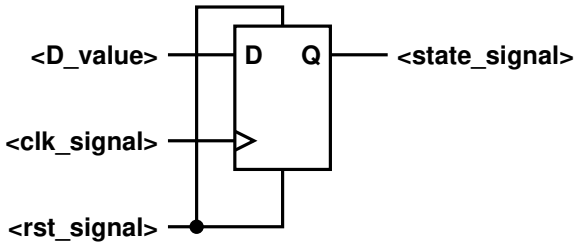
```
<signal>  <= <value1>
                WHEN <condition1> ELSE
             <value2>
                WHEN <condition2> ELSE
             <value3>;
```

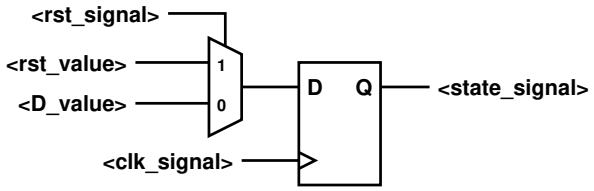
VHDL syntax definition:

```
IF <condition1> THEN
   IF <condition2> THEN ...
                   ELSE ...
   END IF;
               ELSE
   IF <condition3> THEN ...
                   ELSE ...
   END IF;
END IF;
```

- We can nest the multiplexers (prioritized muxes).

D-flipflops





VHDL snippet:

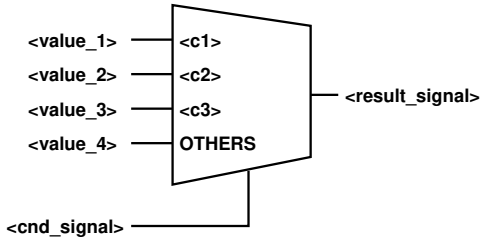
```
dff:PROCESS (<rst_signal>,<clk_signal>) IS
BEGIN
    IF (<rst_signal> = '1') THEN
        <state_signal>  <= <rst_value>;
    ELSIF (rising_edge(<clk_signal>)) THEN
        <state_signal>  <= <d_value>;
    END IF;
END PROCESS dff;
```

VHDL snippet:

```
dff:PROCESS (<clk_signal>) IS
BEGIN
    IF (rising_edge(<clk_signal>)) THEN
        IF (<rst_signal> = '1') THEN
            <state_signal>  <= <rst_value>;
        ELSE
            <state_signal>  <= <D_value>;
        END IF;
    END IF;
END PROCESS dff;
```

- Can describe D-flipflops by using the macro **rising_edge()** or **falling_edge()**.
- The left explicite process describes a D-flipflop with asynchronous reset, as the <rst_signal> has “priority” over the **rising_edge()**.
- The right explicite process describes a D-flipflop with synchronous reset as the **rising_edge()** has “priority” over the <rst_signal>.

Unconditional multiplexers



Implicite form:

VHDL syntax definition:

```
WITH <cnd_signal> SELECT <result_signal>  <=
   <value_1> WHEN <c1>,
   <value_2> WHEN <c2>,
   <value_3> WHEN <c3>,
   <value_4> WHEN OTHERS;
```

Explicite form:

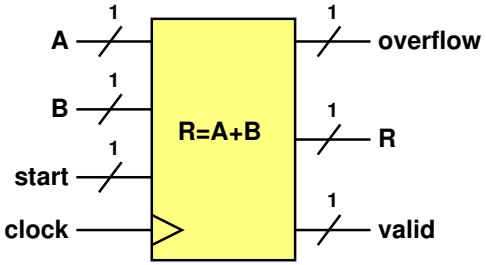
VHDL syntax definition:

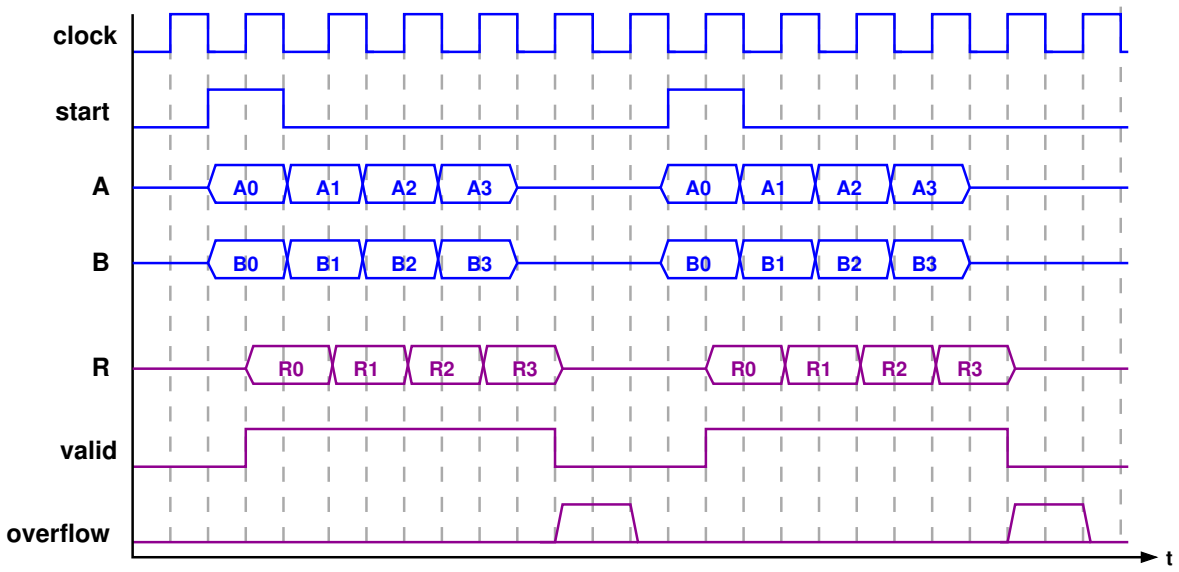
```
CASE <cnd_signal> IS
   WHEN <c1>  => <body_c1>;
   WHEN <c2>  => <body_c2>;
   WHEN <c3>  => <body_c3>;
   WHEN OTHERS  => <body_default>;
END CASE ;
```

- We can also use unconditional multiplexers. Note the **OTHERS**-statement.
- <body_c1> could be for example: <result_signal> **<=** <value_1>;
- Each selection in the explicite form can contain multiple statements (even **IF** .. **THEN** .. **ELSE** .. **END IF**;).

Design example

- We now know all constructs to describe a digital system.
- Let's put it into practice; realize below functionality in VHDL:
- Of course you need to know the coding: it is binairy.





Lecture 17

Digital system design

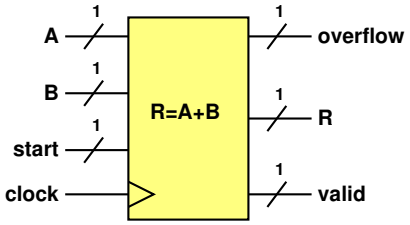
VHDL Behavior(4)

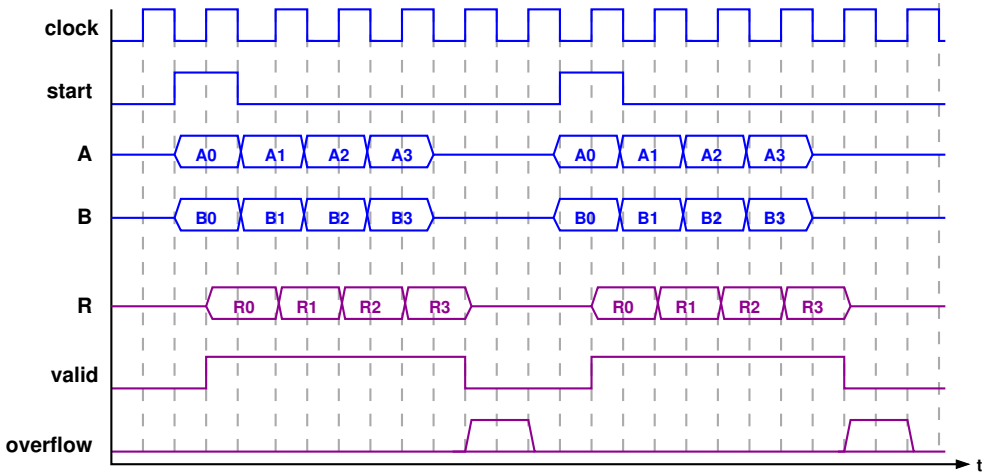
CS173 - Conception de systèmes numériques
Mai 2017

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

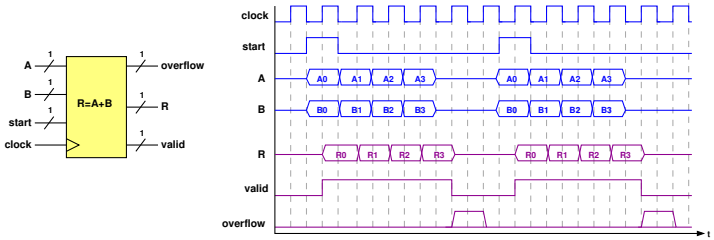
Design example

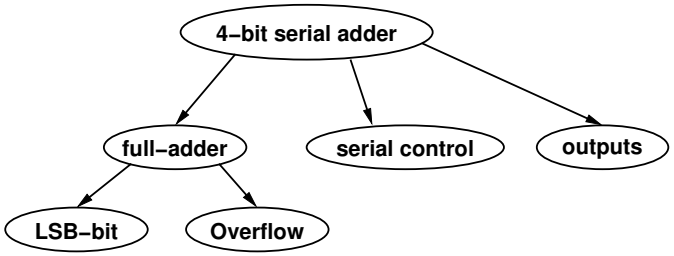
- We now know all constructs to describe a digital system.
- Let's put it into practice; realize below functionality in VHDL:
- Of course you need to know the coding: it is binairy.





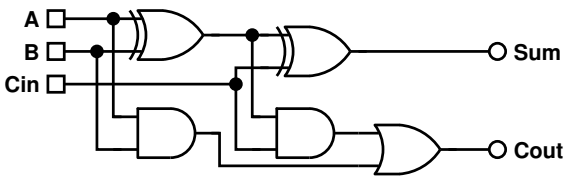
Divide and Conquer





- We can divide this problem in 3 sub-problems, namely:
  1. The addition of one bit, this is a full-adder already seen in an earlier TP.
  2. The serial control, as the bits come in one after the other, we have to keep track if we are processing bits, and when, which bit we are processing.
  3. The output control, as seen in the timing diagramm all are synchronous to the rising edge of the clock.

Full adder



- Let's start with the simple part, the full-adder.
- Write the complete VHDL-description for this full-adder:

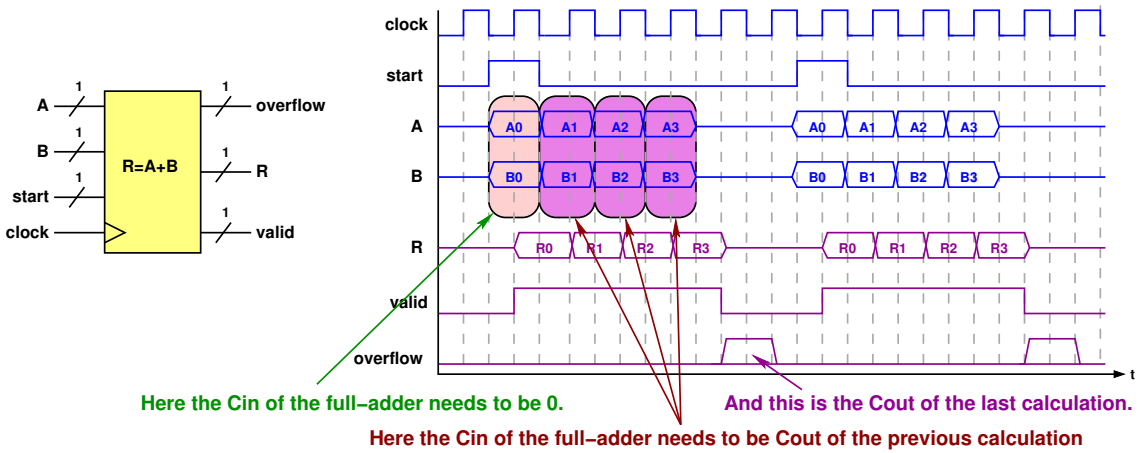
VHDL snippet:

```
LIBRARY ieee;
USE
ieee.std_logic_1164.all;

ENTITY FullAdder IS
```

VHDL snippet:

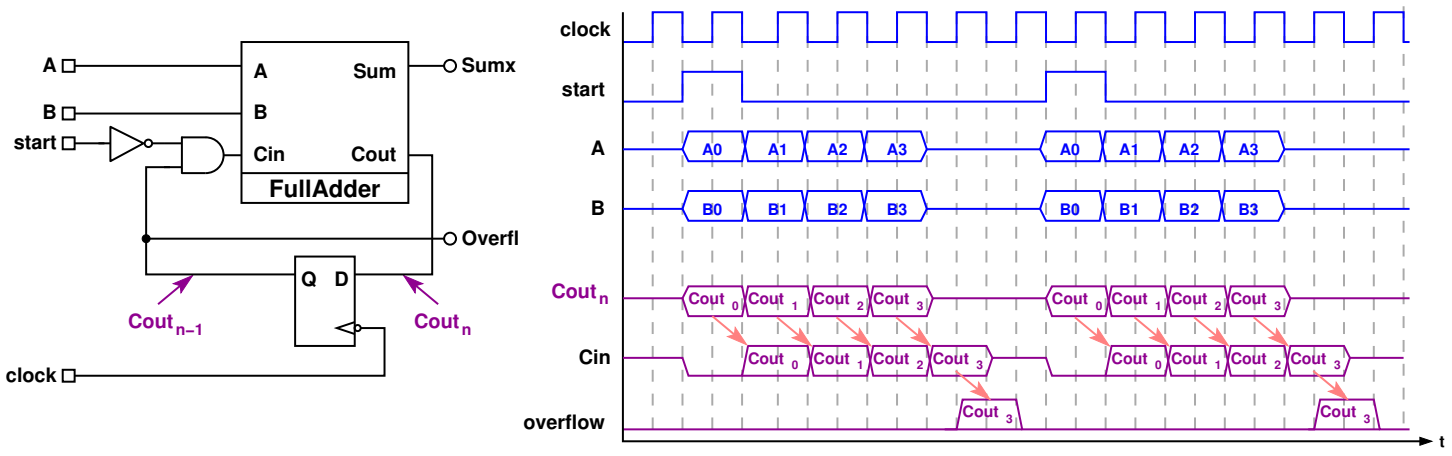
```
ARCHITECTURE dataflow OF
FullAdder IS
```

Full adder



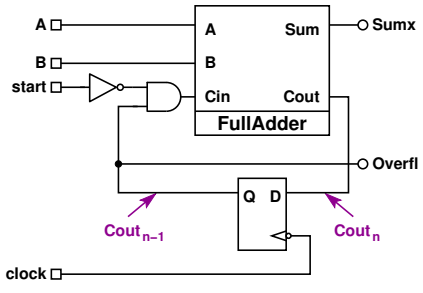
- The full-adder is super, however, there are some particulaties in this system.
- When the LSB arrives, the Cin of the full-adder needs to be zero, hence: $Cin_0 = 0$.
- For the following bits we need to have the carry-out of the previous calculation. Hence: $Cin_n = Cout_{n-1}$.
- The overflow is the carry-out of the last calculation, hence: $Overflow = Cout_n$.

Carry-in and Carry-out



- How to solve the problem: $Cin_n = Cout_{n-1}$?
- We want the timly behavior as shown in the timing diagram.
- We can observe that Cin is the one clock cycle delayed version of Cout; we can use a D-flipflop to delay Cout. It gives us directly the overflow at the right position.
- However, Cin should be 0 at the LSB-position (when start=1), so we add following logic function.

Datapath



- Write the complete VHDL-description of this system.

VHDL snippet:

```
LIBRARY ieee;
USE
ieee.std_logic_1164.all;

ENTITY SerialAdd IS
```

VHDL snippet:

```
ARCHITECTURE datapath OF
SetialAdd IS
```

- I could again describe the circuit of the full-adder, but is there not a way of using the already created VHDL-description?

Components

VHDL snippet:

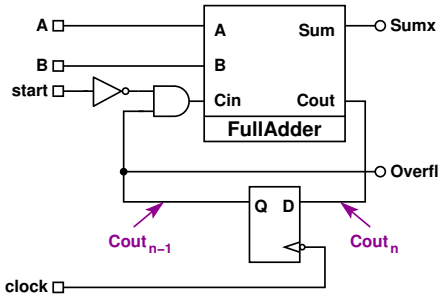
```
LIBRARY ieee;
USE
ieee.std_logic_1164.all;

ENTITY FullAdder IS
PORT ( A,B,Cin :  IN
std_logic;
       Sum,Cout :  OUT
std_logic);
END FullAdder;
```

VHDL snippet:

```
COMPONENT FullAdder IS
PORT ( A,B,Cin :  IN
std_logic;
       Sum,Cout :  OUT
std_logic);
END COMPONENT ;
```

- Yes, we can. We do this by using components.
- In the «declaration section» of an architecture we can reference the usage of the FullAdder by copying the entity description and embed it in the **COMPONENT** syntax.
- This indicates that we are going to use one or multiple instances of this hardware in the «description section» of the architecture.

Components

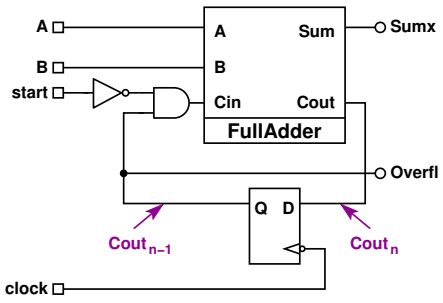


- With an implicte process the combinational logic is described.
- With an explicite process the D-flipflop is described.
- Finally the component is instantiated.

VHDL snippet:

```
ARCHITECTURE datapath OF
SerialAdd IS

COMPONENT FullAdder
IS
PORT ( A,B,Cin :  IN
std_logic;
       Sum,Cout :  OUT
std_logic);
END COMPONENT ;

SIGNAL
s_Cin,s_Cout,s_Cout_del :
std_logic;

BEGIN
    s_Cin <= s_Cout_del
AND NOT (start);

    dff :  PROCESS (
clock ) IS
    BEGIN
       IF
(falling_edge(clock))
THEN
          s_Cout_del  <=
s_Cout;
       END IF;
    END PROCESS ;

    add :  FullAdder
    PORT MAP ( A  => A,
               B  => B,
               Cin  =>
s_Cin,
```

Components

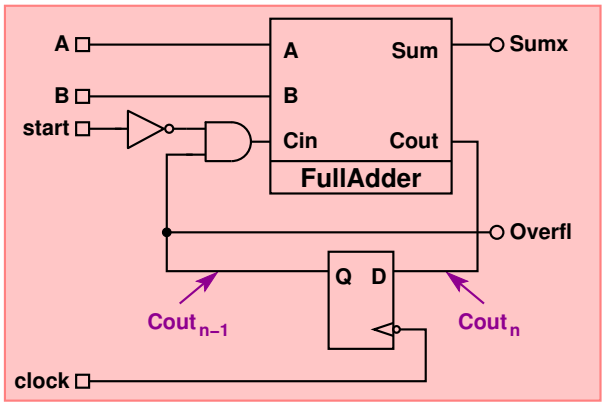


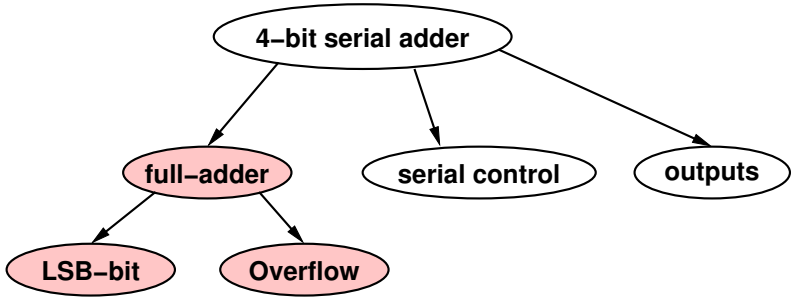
VHDL snippet:

```
add :  FullAdder
PORT MAP (A   => A,
          B   => B,
          Cin  =>
s_Cin,
          Sum  =>
Sumx,
          Cout =>
s_Cout);
```

- How does it work:
- Each instance of a component needs a unique name.
- Followed by the <entity_name> of the component.
- Then the connections are made with a **PORT MAP** .
- The names left of the **=>** operator are the names used in the entity of the component.
- The names right of the **=>** operator are the names of the signals in the current architecture.
- All “maps” are seperated by a comma.

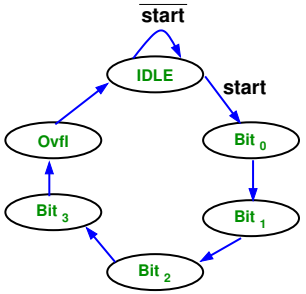
Divide and conquer

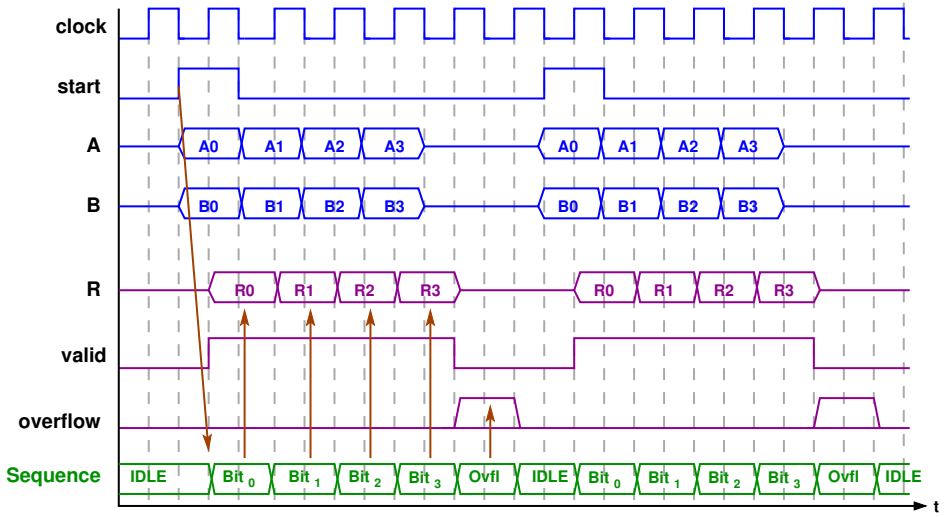




- We now solved the first sub-problem of our serial adder.
- This part is called the *datapath* of the system.
- Let's continue with the control part.

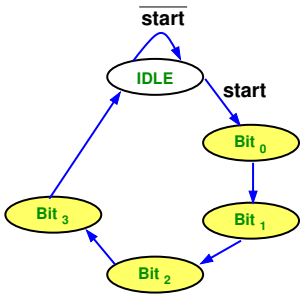
Determining the sequence

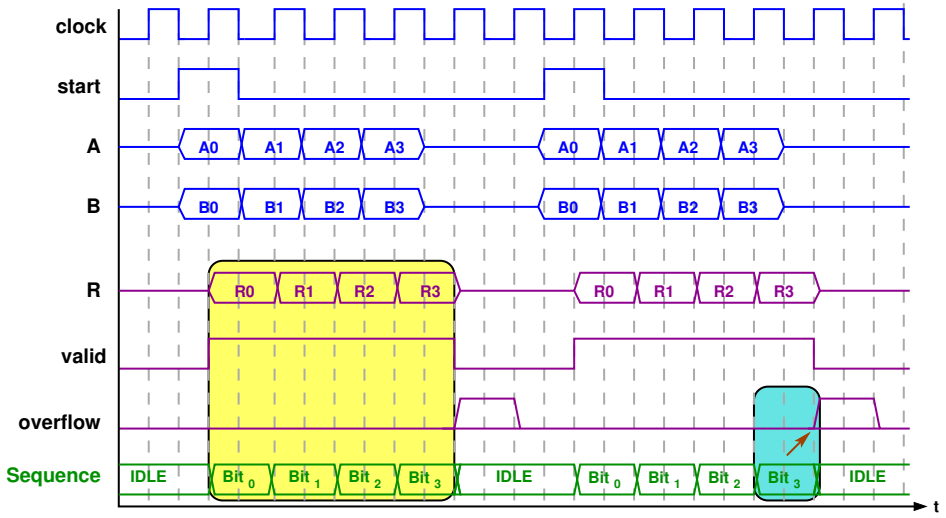




- Let us determine the state diagram.
- As long as there is no start signal there is nothing to do (we are in idle).
- After the start the LSB is calculated. And one by one the next bits.
- The sequence has ended and we are in idle again.
- Upto the moment another start indicates a new series of bits.

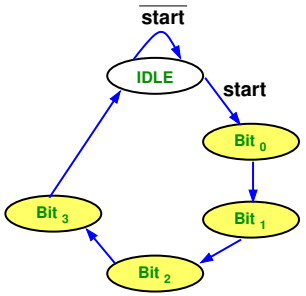
What about the outputs

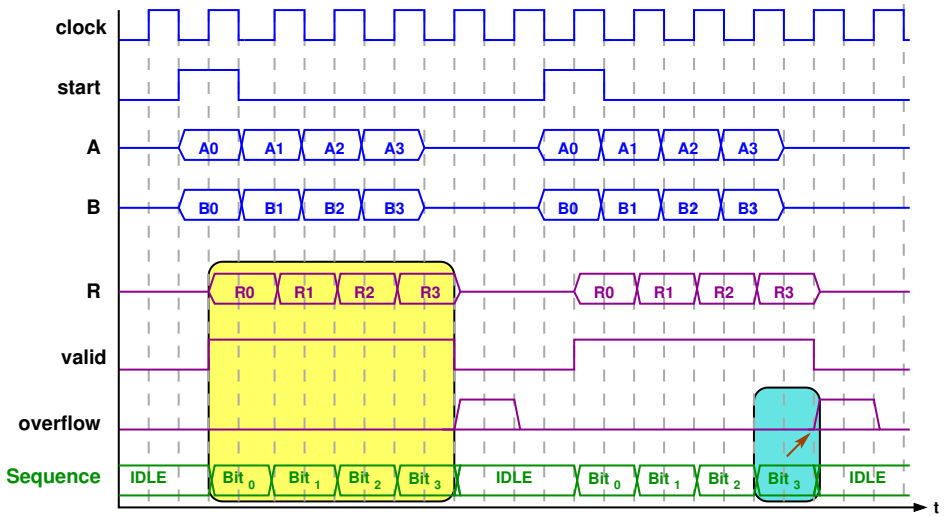




- What about the outputs of this finite state machine.
- We can observe that the valid is high during the states Bit_0 ... Bit_3. So this signal is a direct output of the state machine.
- We can also observe that state Bit_3 can be used to enable the storage of the overflow bit.
- We can even optimize the sequence a bit, as the state `Ovl` has no use, so we can jump directly from state Bit_3 in state `IDLE`.

Exercise





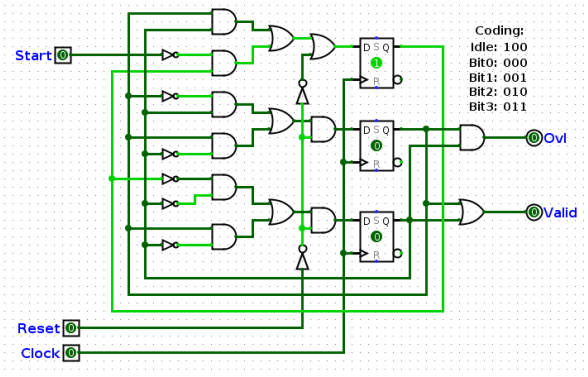
- Write the complete VHDL description for this state machine.
- The Entity is given by:

VHDL snippet:

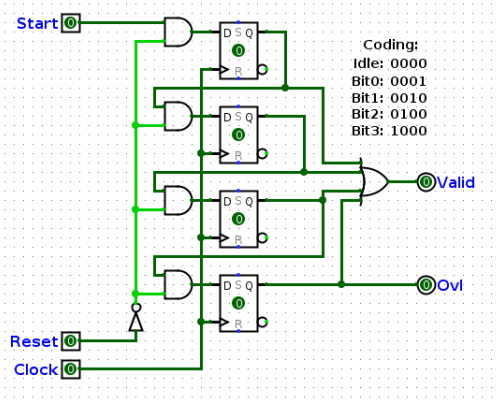
```
LIBRARY ieee;
USE
ieee.std_logic_1164.all;

ENTITY ControlMachine
IS
PORT ( clock,start,reset :
IN std_logic;
       valid,ovl :  OUT
std_logic);
END ControlMachine;
```

Many possibilities

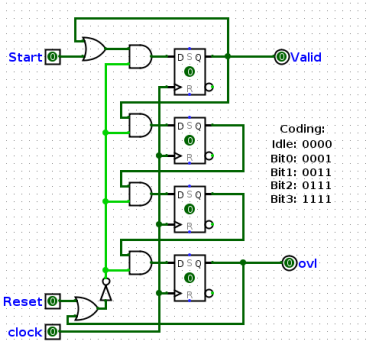
Binairy coding:



One–hot coding:



Thermometer coding:



- There are many possible implementations of this functionality.
- Do we really care about the implementation?
- Can’t we just describe the functionality and let the synthesizer “pick” a implementation?
- Yes we can!

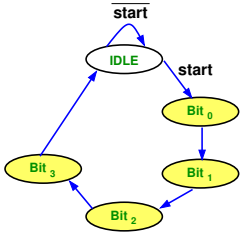
Types

VHDL syntax definition:

```
TYPE <type_name> IS
(<name_1>,...,<name_n>);
```

- We can define types in VHDL in the «declaration section»of an architecture.
- The synthesizer will automatically generate a code-table for the entires included in the type definition.
- Each type needs a unique name.
- Each type is local to it's architecture.
- A type can have as many entries as required, seperated by a comma.
- The entry names must be unique and may not be a VHDL keyword.

One description

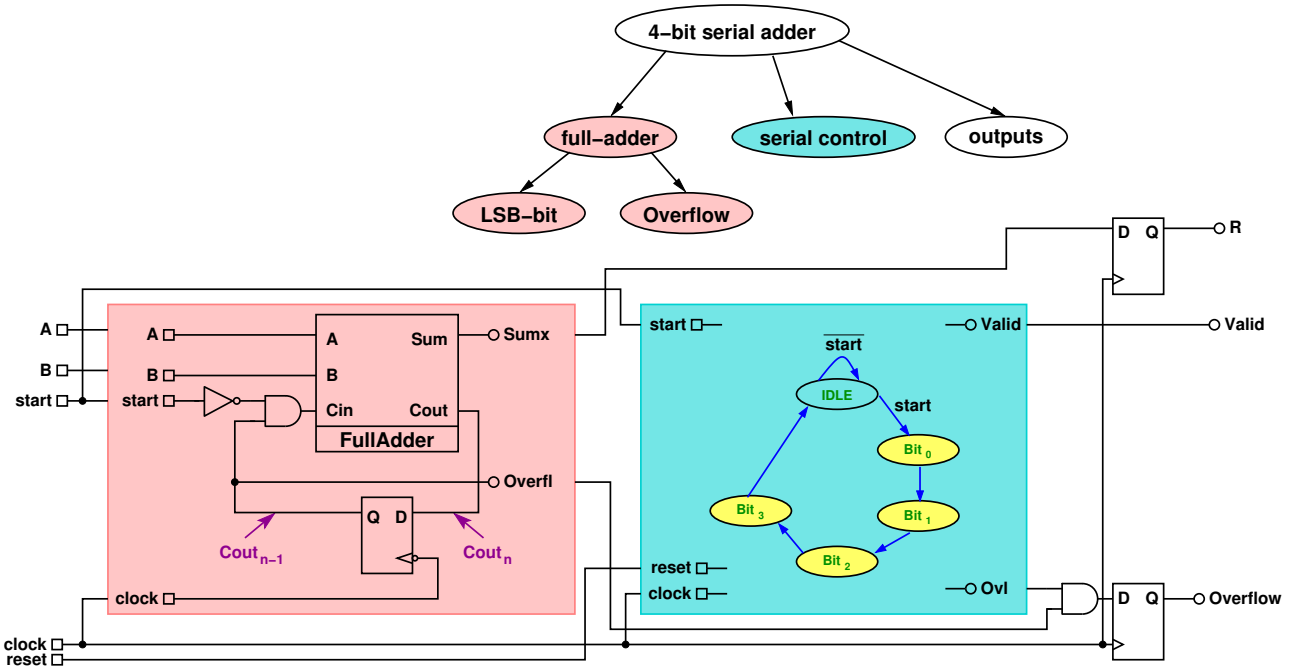


- Let's see how it works.
- We define a type with the state names.
- We define the current and next state signals of this type.
- We assign the outputs.
- We infer the state

VHDL snippet:

```
ARCHITECTURE UseType OF
ControlMachine IS
   TYPE StateType IS
(IDLE,BIT0,BIT1,BIT2,BIT3);
   SIGNAL s_cur_state,
s_next_state :  StateType;
BEGIN
   Ovl  <= '1' WHEN s_cur_state =
BIT3 ELSE '0';
   Valid  <= '1' WHEN s_cur_state
= BIT0 OR
                     s_cur_state
= BIT1 OR
                     s_cur_state
= BIT2 OR
                     s_cur_state
= BIT3 ELSE '0';
   dff :  PROCESS ( clock )
   BEGIN
      IF (rising_edge(clock))
THEN
         IF (reset = '1') THEN
s_cur_state  <= IDLE;
                  ELSE
s_cur_state  <= s_next_state;
         END IF;
      END IF;
   END PROCESS dff;
   transition :  PROCESS (
start,s_cur_state) IS
   BEGIN
      CASE (s_cur_state) IS
         WHEN IDLE  => IF (start
= '1') THEN
                                 s_next_state
```

The complete system



- Now we have solved the second problem.
- And this is how it looks like when we put all together.
- This combination is called a *FSM-D* and a simple version of a processor.

Lecture 19

Digital system design

VHDL Behavior(5)

CS173 - Conception de systèmes numériques
Mai 2017

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Introduction

- In the last lecture we have seen how to make a state machine by using types.
- But what about state machines with 500+ states?
- or comparisons,
- or additions ....
- The types std_logic and std_logic_vector have no interpretation....
- Would it not be nice to have a binairy or two's complement representation?

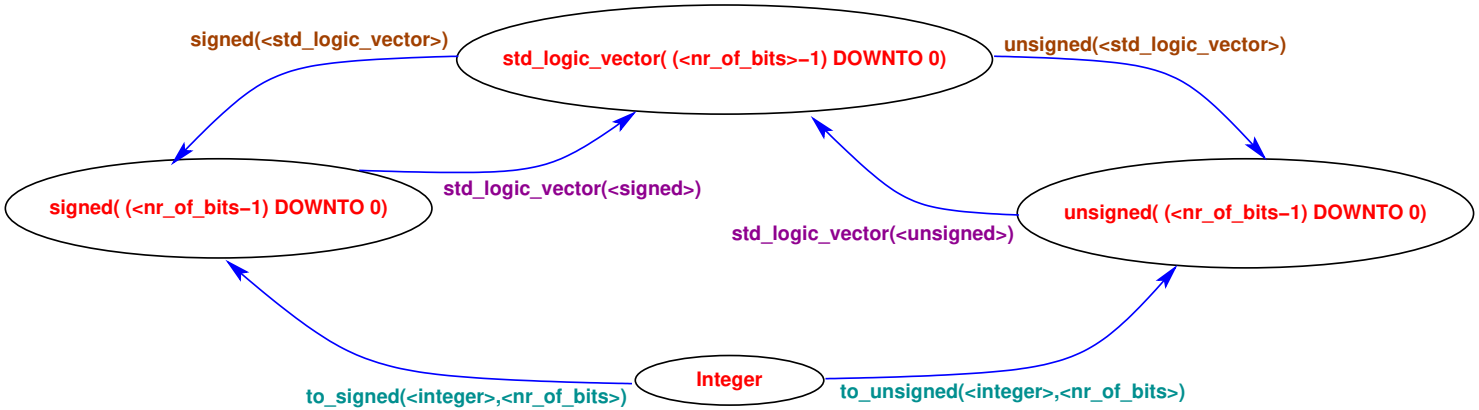
Numbers

- We have, in the **numeric_std** package.
- To use this package we have to define before the **ENTITY** :

  LIBRARY ieee;
  USE ieee.std_logic_1164.all;
  USE ieee.numeric_std.all;

- The **numeric_std** package defines two types, namely:
  - **unsigned**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a binairy interpretation.
  - **signed**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a two's complement interpretation.
- But how to use.....

Conversions

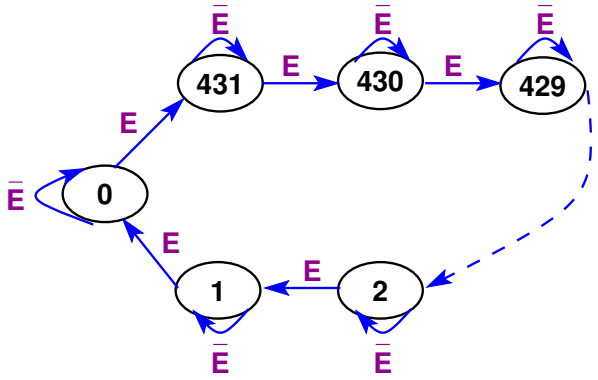


- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.
- But what about the std_logic_vector?
- We can do "*type-casting*".

Operators

- So which operators exist:
  - **+**: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - **-**: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).
  - **>**: Bigger than.
  - **>=**: Bigger or equal.
  - **<**: Smaller than.
  - **=<**: Smaller or equal.
  - ***: Multiplication: Never use this operator, as not all synthesisers know which hardware to generate!
  - /: Division: Never use this operator, as all synthesisers don't know which hardware to generate!
  - **sll**,**srl**: Shift logic left and shift logic right.
  - **sla**,**sra**: Shift arithmetic left and shift arithmetic right.
  - **rol**,**ror**: rotate left and rotate right.
    Important: Never use the above shift functions as their implementations may result in unexpected behavior (use manual concatination instead).
- Note the order of the **=** and the **</>** symbols in the comparision, they have to be in this order!
- *Important:* Of course the types left and right of the operator need to be of same *type* and need to have the same *number of bits*!

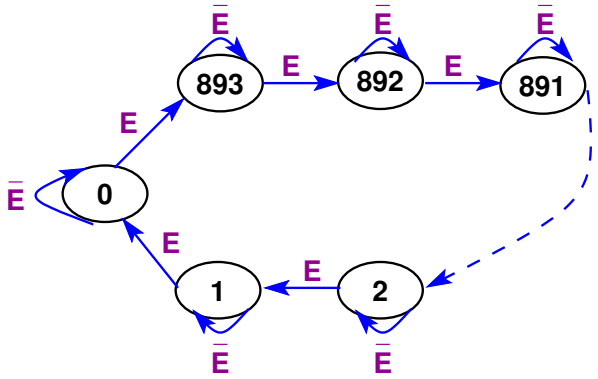
Exercise



- We want to design a counter with 432 states.
- The counter is reset synchronously in state 356.
- It's state diagram is given above.
- The output **X** of this state-machine is active for the states 0..10.
- Create the complete VHDL-description for this counter.

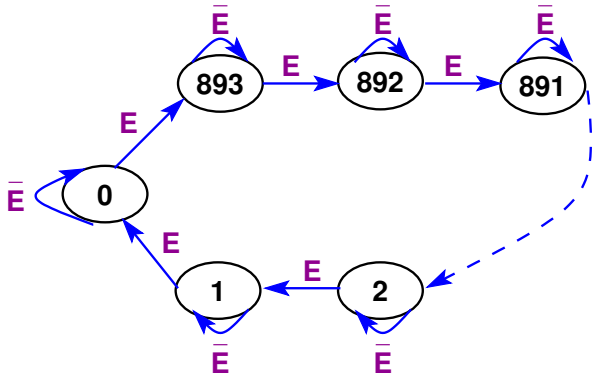
Exercise

Another one



- We want to design a counter with 894 states.
- The counter is reset synchronously in state 513.
- It's state diagram is given above.
- The output **X** of this state-machine is active for the states 0..50.
- Create the complete VHDL-description for this counter.
- Not again, can't we do something more intelligent?

Another one



- Where are the differences between this counter and the previous one:
  - The number states.
  - The number of bits required.
  - The reset state.
  - The number of states required for the output.
- Would it not be nice to be able to parameterize a component?

Generics

VHDL syntax definition:

```
ENTITY <name> IS
    GENERIC ( <list> );
    PORT ( <list> );
END <name>;
```

- Yes we can, we can insert a **GENERIC** block in the **ENTITY** definition.
- In the **GENERIC** block we can define the parameters.
- And as such we can describe a parameterized circuit description.
- Note that the parameters are *fixed* the moment a component is used, as there exist no such thing as dynamic hardware.
- So how to use this for our counter?

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;
              ResetState :  INTEGER;
              OutState :  INTEGER);
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:
  - The number states.
  - The number of bits required.
  - The reset state.
  - The number of states required for the output.
- **Note :** As in the port declaration also the last line in the generic declaration is not ended with a **;**!

Exercise

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;
              ResetState :  INTEGER;
              OutState :  INTEGER);
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- Modify your architecture such that it uses the above parameters.

Exercise

How to use

VHDL snippet:

```
Example1 :  GenericCounter
    GENERIC MAP ( MaxState
=> 432,
                  NrOfBits
=> 9,
                  ResetState
=> 356,
                  OutState
=> 10 )
    PORT MAP ( ...  );
```

VHDL snippet:

```
Example2 :  GenericCounter
    GENERIC MAP ( MaxState
=> 894,
                  NrOfBits
=> 10,
                  ResetState
=> 513,
                  OutState
=> 50 )
    PORT MAP ( ...  );
```

- We can now instantiate in an architecture the parameterized versions of the generic counter as shown above.
- **Important:** After the closing) of the **GENERIC MAP** and the start of the **PORT MAP** there is no **;**!

Lecture 19

Digital system design

VHDL Behavior(5)

CS173 - Conception de systèmes numériques
May 2018

Prof. Dr. Theo Kluter
Bern University of Applied Sciences

Introduction

- In the last lecture we have seen how to make a state machine by using types.

Introduction

- In the last lecture we have seen how to make a state machine by using types.
- But what about state machines with 500+ states?

Introduction

- In the last lecture we have seen how to make a state machine by using types.
- But what about state machines with 500+ states?
- or comparisons,

Introduction

- In the last lecture we have seen how to make a state machine by using types.
- But what about state machines with 500+ states?
- or comparisons,
- or additions ....

Introduction

- In the last lecture we have seen how to make a state machine by using types.
- But what about state machines with 500+ states?
- or comparisons,
- or additions ....
- The types std_logic and std_logic_vector have no interpretation....

Introduction

- In the last lecture we have seen how to make a state machine by using types.
- But what about state machines with 500+ states?
- or comparisons,
- or additions ....
- The types std_logic and std_logic_vector have no interpretation....
- Would it not be nice to have a binairy or two's complement representation?

arithmetic

- We have, in the **numeric_std** package.

Numbers

- We have, in the **numeric_std** package.
- To use this package we have to define before the **ENTITY** :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

Numbers

- We have, in the **numeric_std** package.
- To use this package we have to define before the **ENTITY** :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

- The **numeric_std** package defines two types, namely:

Numbers

- We have, in the **numeric_std** package.
- To use this package we have to define before the **ENTITY** :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

- The **numeric_std** package defines two types, namely:
  - **unsigned**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a binairy interpretation.

Numbers

- We have, in the **numeric_std** package.
- To use this package we have to define before the **ENTITY** :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

- The **numeric_std** package defines two types, namely:
  - **unsigned**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a binairy interpretation.
  - **signed**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a two's complement interpretation.

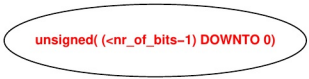
Numbers

- We have, in the **numeric_std** package.
- To use this package we have to define before the **ENTITY** :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

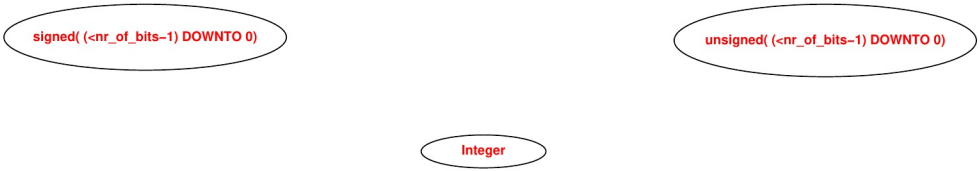
- The **numeric_std** package defines two types, namely:
  - **unsigned**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a binairy interpretation.
  - **signed**(<nr_of_bits-1> **DOWNTO** 0)
    This defines a set of *nr_of_bits* bits with a two's complement interpretation.
- But how to use.....

Conversions



- We have the **unsigned** and **signed** vectors with the same number of bits.

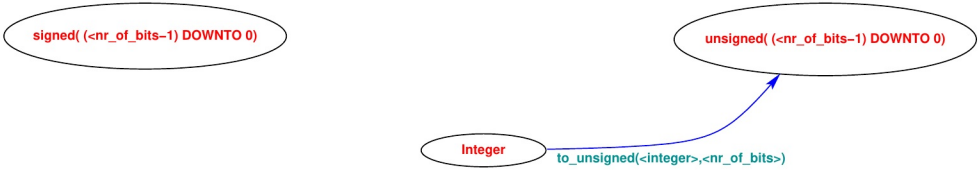
Conversions



- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**.

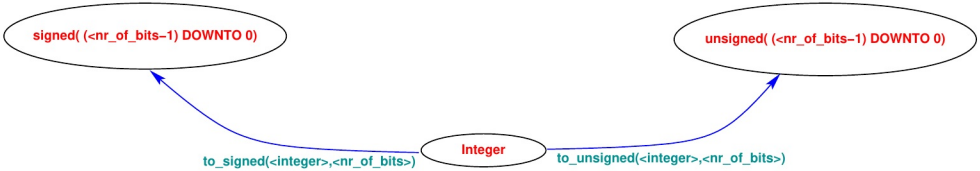
Conversions

```
to_unsigned(127, 3);
to_unsigned(-3, 3);
```



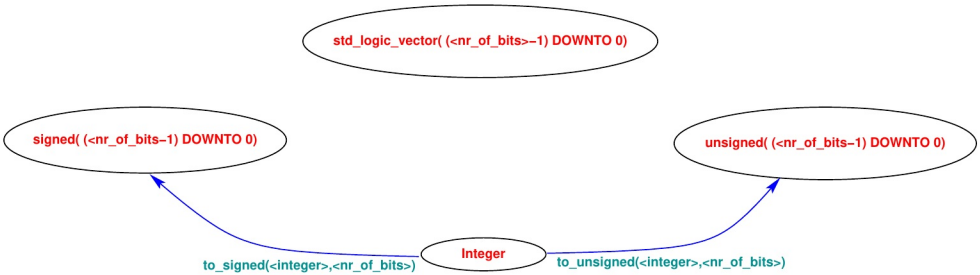
- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.

Conversions



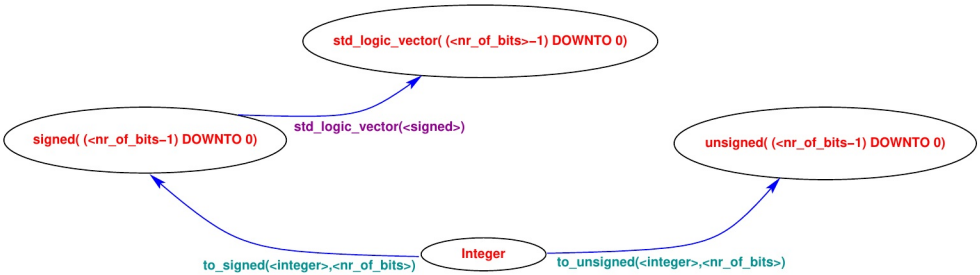
- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.

Conversions



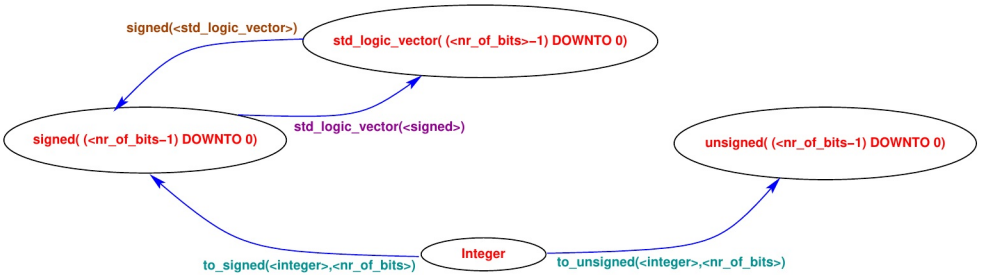
- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.
- But what about the std_logic_vector?

Conversions



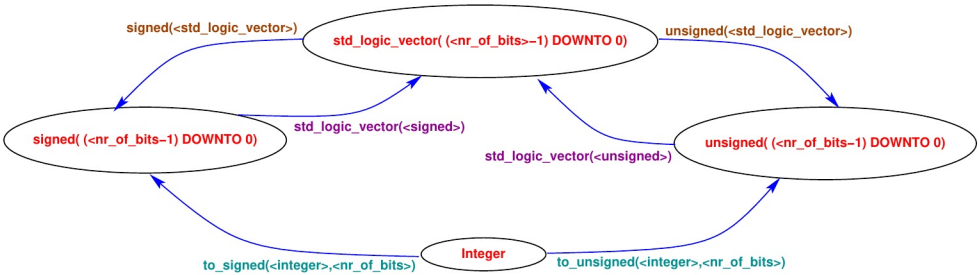
- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.
- But what about the std_logic_vector?
- We can do "*type-casting*".

Conversions



- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.
- But what about the std_logic_vector?
- We can do "*type-casting*".

Conversions



- We have the **unsigned** and **signed** vectors with the same number of bits.
- We are used to use **integers**. With macros we can convert the **integers** to their binairy or two's complement representation.
- But what about the std_logic_vector?
- We can do "*type-casting*".

Operators

- So which operators exist:

Operators

- So which operators exist:
  - +: Addition. The synthesiser will realize a chain of full-adders (think about TP2).

Operators

- So which operators exist:
  - +: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - -: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).

Operators

- So which operators exist:
  - `+`: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - `-`: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).
  - `>`: Bigger than.
  - `>=`: Bigger or equal.
  - `<`: Smaller than.
  - `=<`: Smaller or equal.

Operators

- So which operators exist:
  - +: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - -: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).
  - >: Bigger than.
  - >=: Bigger or equal.
  - <: Smaller than.
  - =<: Smaller or equal.
  - *: Multiplication: Never use this operator, as not all synthesisers know which hardware to generate!
  - /: Division: Never use this operator, as all synthesisers don't know which hardware to generate!

Operators

- So which operators exist:
  - **+**: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - **-**: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).
  - **>**: Bigger than.
  - **>=**: Bigger or equal.
  - **<**: Smaller than.
  - **=<**: Smaller or equal.
  - ***: Multiplication: Never use this operator, as not all synthesisers know which hardware to generate!
  - **/**: Division: Never use this operator, as all synthesisers don't know which hardware to generate!
  - **sll**,**srl**: Shift logic left and shift logic right.
  - **sla**,**sra**: Shift arithmetic left and shift arithmetic right.
  - **rol**,**ror**: rotate left and rotate right.
    Important: Never use the above shift functions as their implementations may result in unexpected behavior (use manual concatination instead).

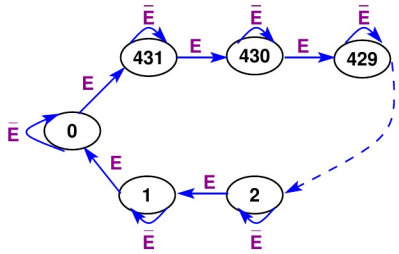
Operators

- So which operators exist:
  - **+**: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - **-**: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).
  - **>**: Bigger than.
  - **>=**: Bigger or equal.
  - **<**: Smaller than.
  - **=<**: Smaller or equal.
  - ***: Multiplication: Never use this operator, as not all synthesisers know which hardware to generate!
  - **/**: Division: Never use this operator, as all synthesisers don't know which hardware to generate!
  - **sll**,**srl**: Shift logic left and shift logic right.
  - **sla**,**sra**: Shift arithmetic left and shift arithmetic right.
  - **rol**,**ror**: rotate left and rotate right.
    Important: Never use the above shift functions as their implementations may result in unexpected behavior (use manual concatination instead).
- Note the order of the **=** and the **</>** symbols in the comparision, they have to be in this order!

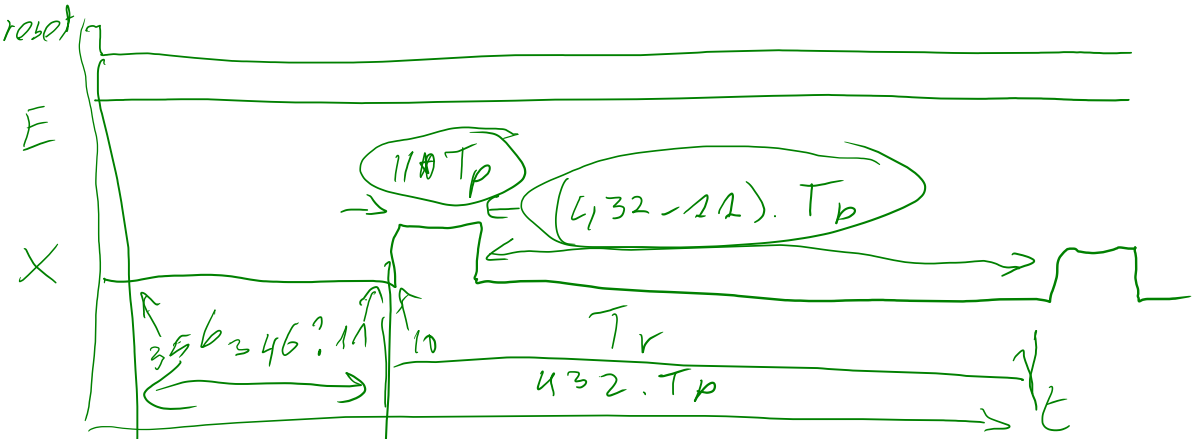
Operators

- So which operators exist:
  - **+**: Addition. The synthesiser will realize a chain of full-adders (think about TP2).
  - **-**: Subtraction. The synthesiser will realize a chain of full-subtractors (think about TP2).
  - **>**: Bigger than.
  - **>=**: Bigger or equal.
  - **<**: Smaller than.
  - **=<**: Smaller or equal.
  - ***: Multiplication: Never use this operator, as not all synthesisers know which hardware to generate!
  - **/**: Division: Never use this operator, as all synthesisers don't know which hardware to generate!
  - **sll**,**srl**: Shift logic left and shift logic right.
  - **sla**,**sra**: Shift arithmetic left and shift arithmetic right.
  - **rol**,**ror**: rotate left and rotate right.
    Important: Never use the above shift functions as their implementations may result in unexpected behavior (use manual concatination instead).
- Note the order of the **=** and the **</>** symbols in the comparision, they have to be in this order!
- *Important:* Of course the types left and right of the operator need to be of same *type* and need to have the same *number of bits*!

Exercise



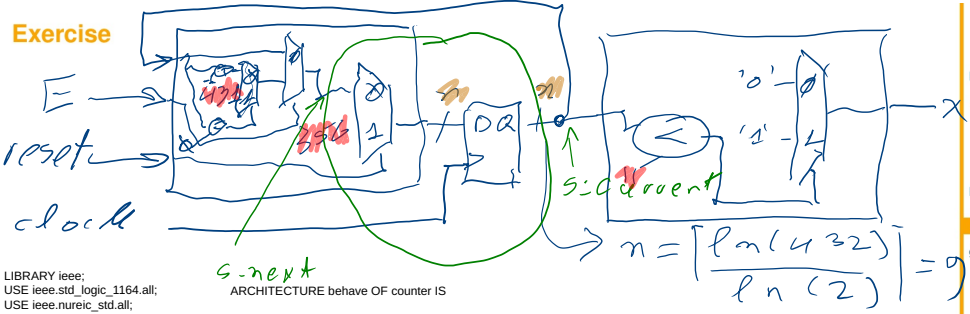
- We want to design a counter with 432 states.
- The counter is reset synchronously in state 356.
- It's state diagram is given above.
- The output **X** of this state-machine is active for the states 0..10.
- Create the complete VHDL-description for this counter.



Horloge = 50 MHz Tp = 20ns

T0 = 10ns T1 = 10ns PC = 50%

Exercise

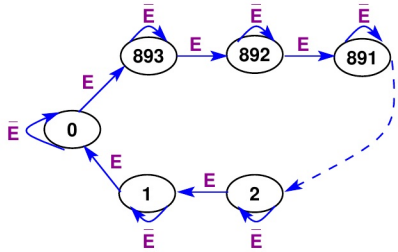



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.nureic_std.all;

ENTITY counter IS
  GENERIC( NrBits : INTEGER;
           NrDesEtats : INTEGER;
           ResetState : INTEGER;
           ValueSortie : INTEGER );
  PORT( E,reset,clock : IN std_logic;
        X : OUT std_logic );
END counter;
```

```
ARCHITECTURE behave OF counter IS

  SIGNAL s_next,s_current : unsigned( (NrBits-1) DOWNTO 0 );

BEGIN
  -- output logic:
  X <= '1' WHEN s_curent < to_unsigned(ValueSortie,NrBits) ELSE '0';

  -- transition logic
  s_next <= s_current WHEN E = '0' ELSE
            to_unsigned(NrDesEtats-1,NrBits) WHEN s_current = to_unsigned(0,NrBits) ELSE
            s_current - to_unsigned(1,NrBits);

  -- Flipflops:
  dff : PROCESS ( clock) IS
  BEGIN
    IF (rising_edge(clock)) THEN
      IF (reset = '1') THEN s_current <= to_unsigned(ResetState,NrBits);
                       ELSE s_current <= s_next;
      END IF;
    END IF;
  END PROCESS dff;
END behave;
```

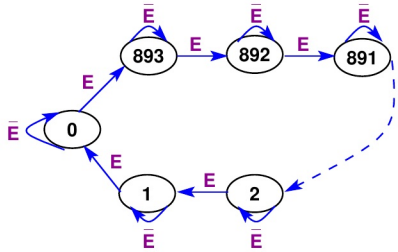
```
cnt1: counter
  GENERIC MAP ( NrBits => 9,
                NrDesEtats => 432,
                ResetState => 356,
                ValueSortie => 11)
  PORT MAP ();
```

Another one



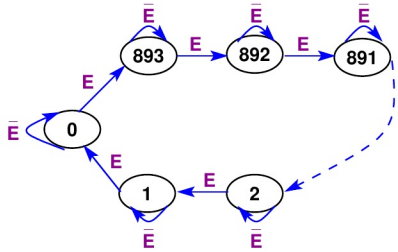
- We want to design a counter with 894 states.
- The counter is reset synchronously in state 513.
- It's state diagram is given above.
- The output **X** of this state-machine is active for the states 0..50.
- Create the complete VHDL-description for this counter.

Another one



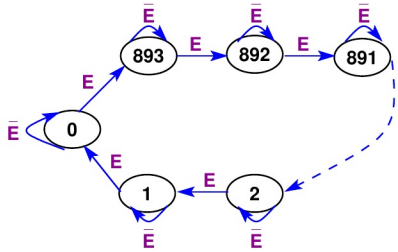
- We want to design a counter with 894 states.
- The counter is reset synchronously in state 513.
- It's state diagram is given above.
- The output **X** of this state-machine is active for the states 0..50.
- Create the complete VHDL-description for this counter.
- Not again, can't we do something more intelligent?

Another one



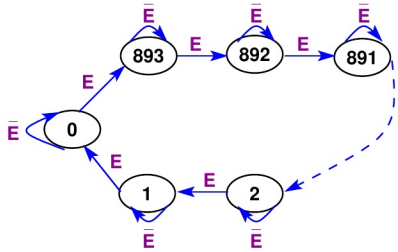
- Where are the differences between this counter and the previous one:

Another one



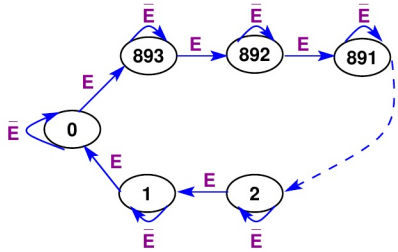
- Where are the differences between this counter and the previous one:
  - The number states.

Another one



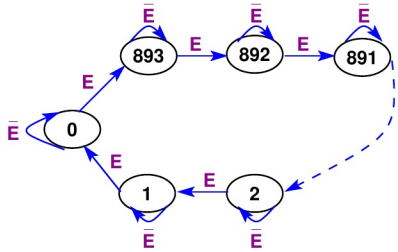
- Where are the differences between this counter and the previous one:
  - The number states.
  - The number of bits required.

Another one



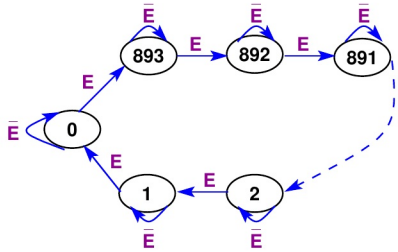
- Where are the differences between this counter and the previous one:
  - The number states.
  - The number of bits required.
  - The reset state.

Another one



- Where are the differences between this counter and the previous one:
  - The number states.
  - The number of bits required.
  - The reset state.
  - The number of states required for the output.

Another one



- Where are the differences between this counter and the previous one:
  - The number states.
  - The number of bits required.
  - The reset state.
  - The number of states required for the output.
- Would it not be nice to be able to parameterize a component?

Generics

VHDL syntax definition:

```
ENTITY <name> IS
    GENERIC ( <list> );
    PORT ( <list> );
END <name>;
```

- Yes we can, we can insert a **GENERIC** block in the **ENTITY** definition.

Generics

VHDL syntax definition:

```
ENTITY <name> IS
    GENERIC ( <list> );
    PORT ( <list> );
END <name>;
```

- Yes we can, we can insert a **GENERIC** block in the **ENTITY** definition.
- In the **GENERIC** block we can define the parameters.

Generics

VHDL syntax definition:

```
ENTITY <name> IS
    GENERIC ( <list> );
    PORT ( <list> );
END <name>;
```

- Yes we can, we can insert a **GENERIC** block in the **ENTITY** definition.
- In the **GENERIC** block we can define the parameters.
- And as such we can describe a parameterized circuit description.

Generics

VHDL syntax definition:

```
ENTITY <name> IS
    GENERIC ( <list> );
    PORT ( <list> );
END <name>;
```

- Yes we can, we can insert a **GENERIC** block in the **ENTITY** definition.
- In the **GENERIC** block we can define the parameters.
- And as such we can describe a parameterized circuit description.
- Note that the parameters are *fixed* the moment a component is used, as there exist no such thing as dynamic hardware.

Generics

VHDL syntax definition:

```
ENTITY <name> IS
    GENERIC ( <list> );
    PORT ( <list> );
END <name>;
```

- Yes we can, we can insert a **GENERIC** block in the **ENTITY** definition.
- In the **GENERIC** block we can define the parameters.
- And as such we can describe a parameterized circuit description.
- Note that the parameters are *fixed* the moment a component is used, as there exist no such thing as dynamic hardware.
- So how to use this for our counter?

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC (


                );
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;


              );
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:
  - The number states.

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;

              );
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:
  - The number states.
  - The number of bits required.

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;
              ResetState :  INTEGER;
              );
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:
  - The number states.
  - The number of bits required.
  - The reset state.

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;
              ResetState :  INTEGER;
              OutState :  INTEGER);
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:
  - The number states.
  - The number of bits required.
  - The reset state.
  - The number of states required for the output.

Generics

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;
              ResetState :  INTEGER;
              OutState :  INTEGER);
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- What were the parameters:
  - The number states.
  - The number of bits required.
  - The reset state.
  - The number of states required for the output.
- **Note :** As in the port declaration also the last line in the generic declaration is not ended with a **;**!

Exercise

VHDL snippet:

```
ENTITY GenericCounter IS
    GENERIC ( MaxState :  INTEGER;
              NrOfBits :  INTEGER;
              ResetState :  INTEGER;
              OutState :  INTEGER);
    PORT ( clock,reset,e :  IN std_logic;
           X : OUT std_logic);
END GenericCounter;
```

- Modify your architecture such that it uses the above parameters.

Exercise

How to use

VHDL snippet:

```
Example1 :  GenericCounter
   GENERIC MAP ( MaxState  => 432,
                 NrOfBits  => 9,
                 ResetState  => 356,
                 OutState  => 10 )
   PORT MAP ( ...  );
```

VHDL snippet:

```
Example2 :  GenericCounter
   GENERIC MAP ( MaxState  => 894,
                 NrOfBits  => 10,
                 ResetState  => 513,
                 OutState  => 50 )
   PORT MAP ( ...  );
```

- We can now instantiate in an architecture the parameterized versions of the generic counter as shown above.

How to use

VHDL snippet:

```
Example1 :  GenericCounter
   GENERIC MAP ( MaxState  => 432,
                 NrOfBits  => 9,
                 ResetState  => 356,
                 OutState  => 10 )
   PORT MAP ( ... );
```

VHDL snippet:

```
Example2 :  GenericCounter
   GENERIC MAP ( MaxState  => 894,
                 NrOfBits  => 10,
                 ResetState  => 513,
                 OutState  => 50 )
   PORT MAP ( ... );
```

- We can now instantiate in an architecture the parameterized versions of the generic counter as shown above.
- **Important:** After the closing **)** of the **GENERIC MAP** and the start of the **PORT MAP** there is no **;**!