Managment and Analysis of Physical Datasets – I
a.a. 2019-20

FPGA and VHDL

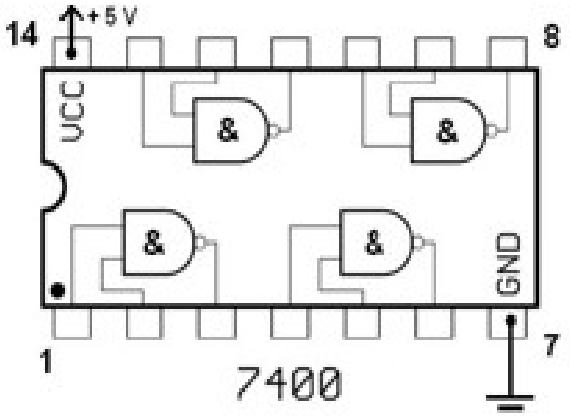
G.Collazuol

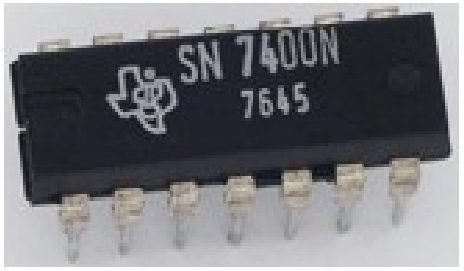
Overview

1) Introduction to FPGA and VHDL
2) VHDL Language elements
3) Concurrent vs Sequential Statements
4) Some relevant examples
5) Finite State Machines
6) Buses
7) Memories

Introduction to FPGA

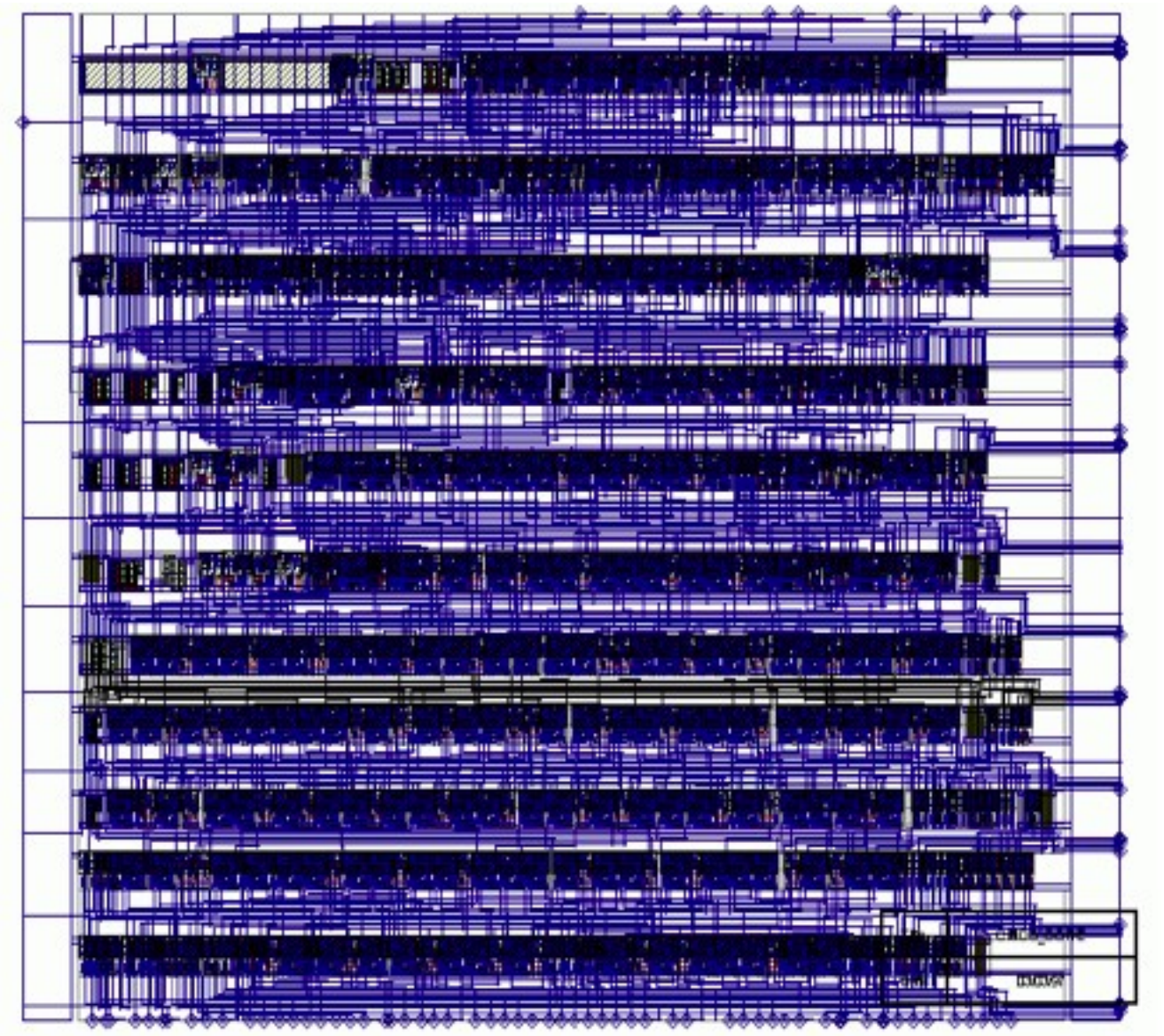
Old ways of implementing digital circuits





- Discrete logic – based on gates or small packages containing small digital building blocks (at most a 1-bit adder)
- De Morgan's theorem – theoretically we only need 2-input NAND or NOR gates to build anything
- Tedious, expensive, slow, prone to wiring errors

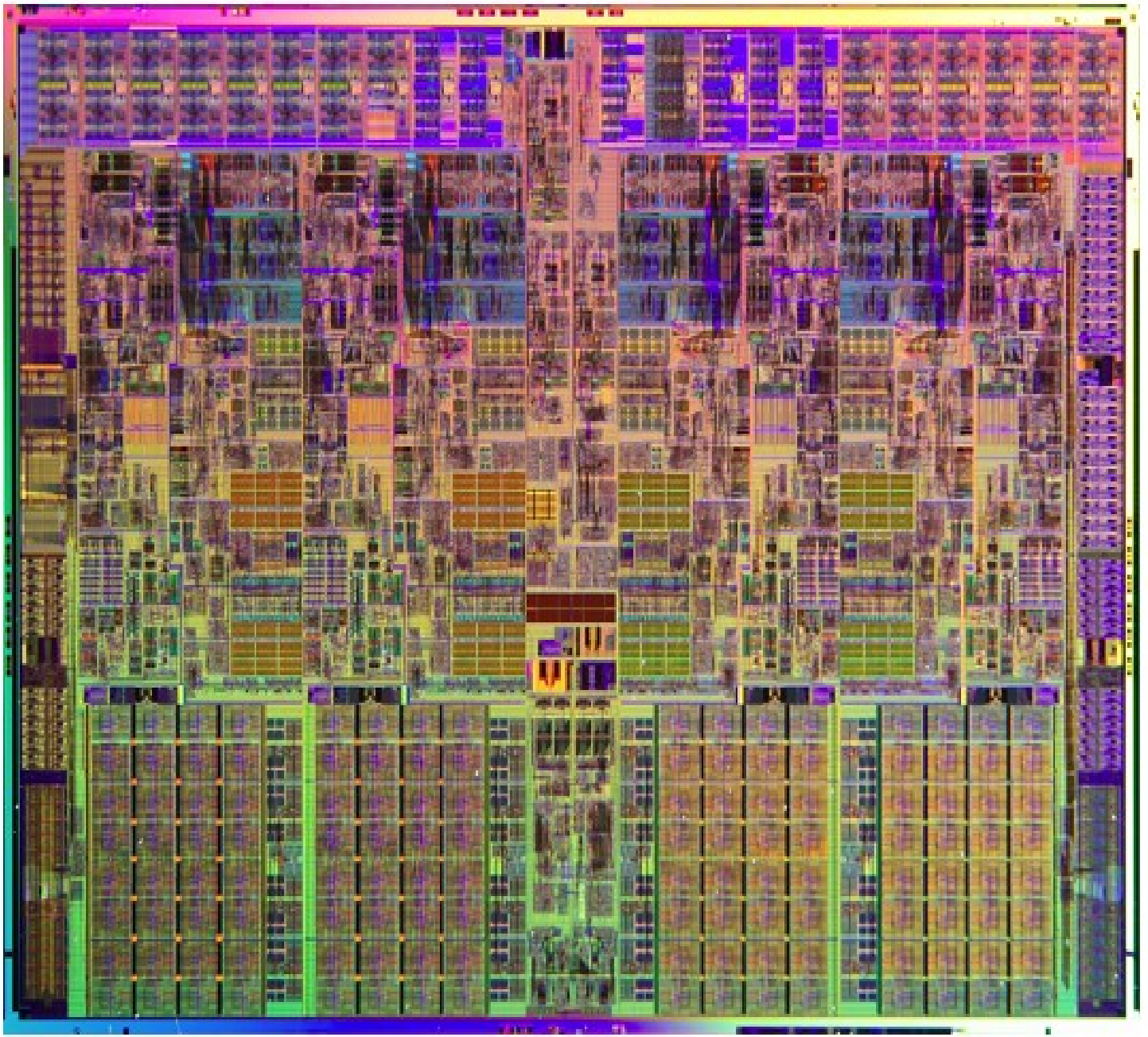
Old digital integrated circuits



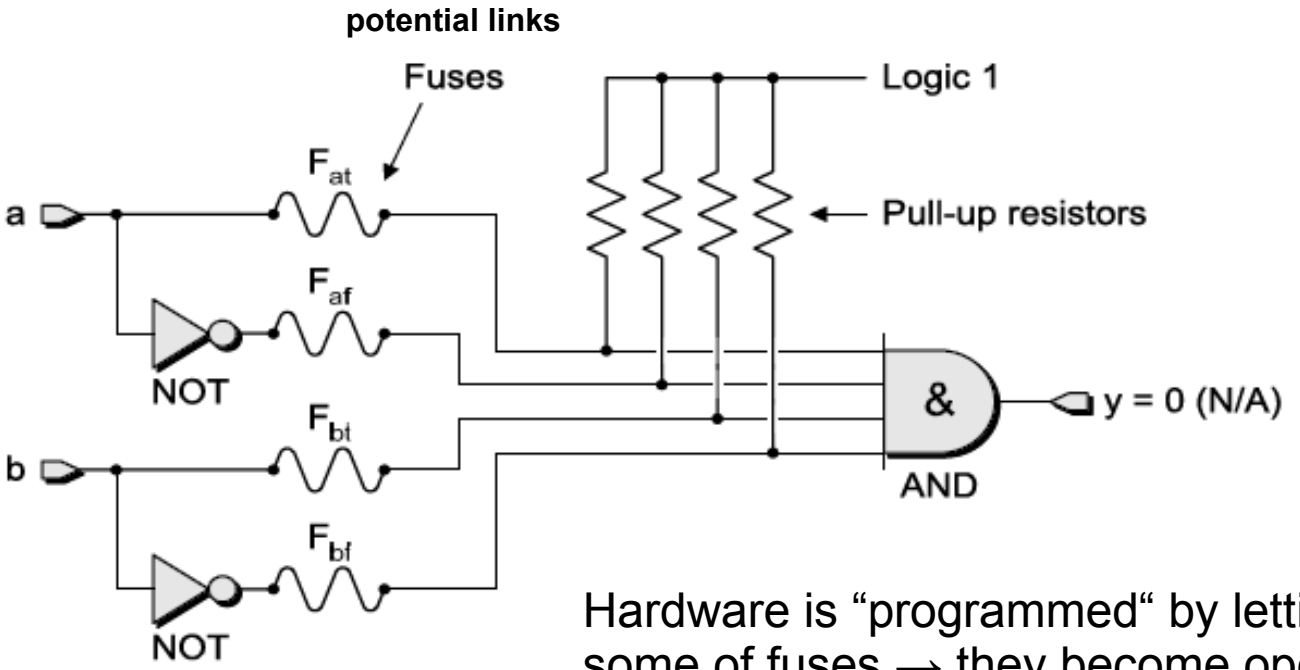
- Rows of gates – often identical in structure
- Connected to form customer specific circuits
- Can be full-custom (i.e. completely fabricated from scratch for a given design)
- Can be semi-custom (i.e. customisation on the metal layers only)
- Once fabricated, the design is fixed

Modern custom digital IC

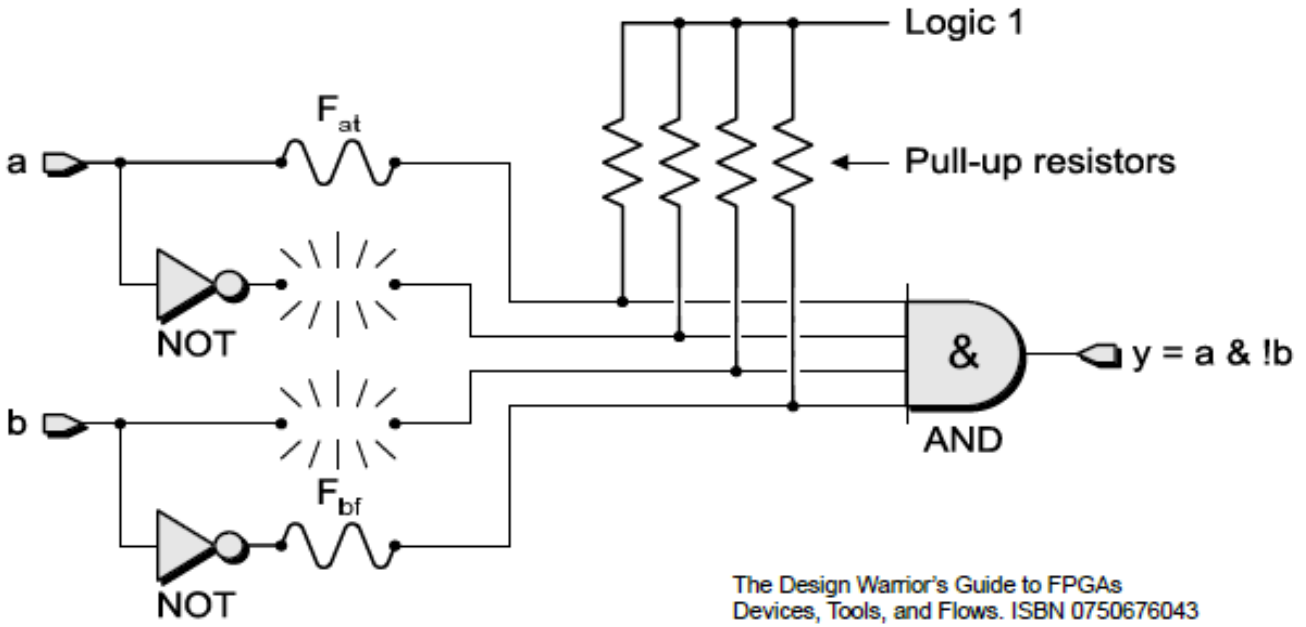
- Intel Core i7
- > ¾ billion trans.
- Very expensive to design
- Very expensive to manufacture
- Not viable unless the market is very large



Programmable Logic Devices



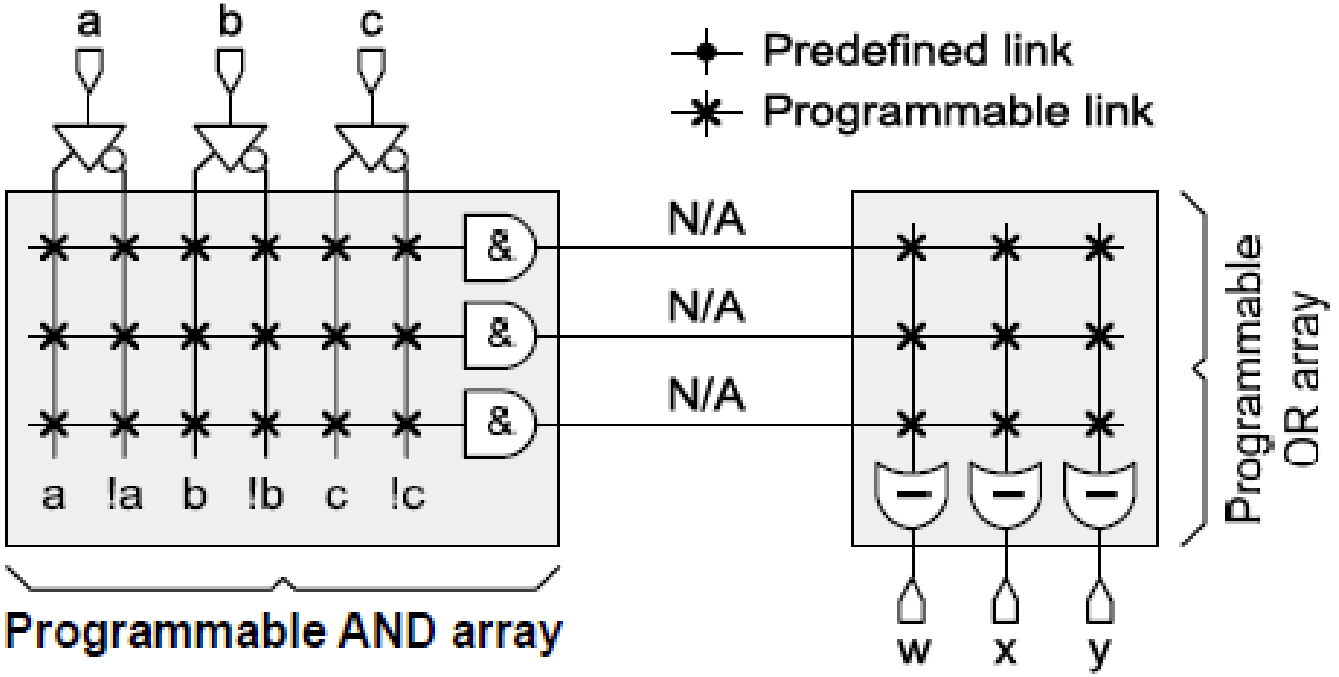
Hardware is “programmed“ by letting flow high current through some of fuses → they become open-circuits



Programmable Logic Arrays

A Programmable Logic Array (PLA) is a type of logic device that can be programmed to implement various kinds of combinational logic circuits. The device has a number of **AND** and **OR** gates which are linked together to give output or further combined with more gates or logic circuits

Reminder:
combinational logic block synthesis

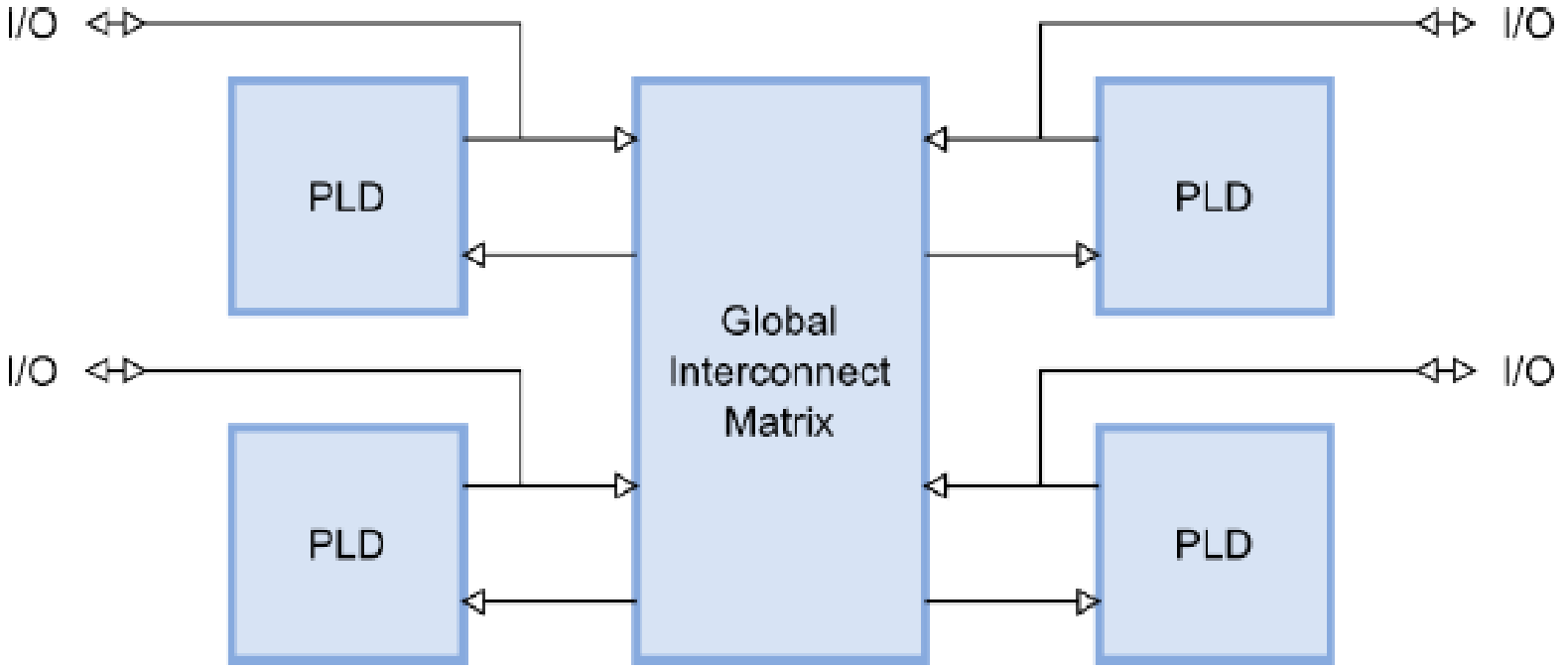


On a PLA is different logical functions can be combined as a sum-of-product or product-of-sum form: a PLA having N input buffers and M output buffers consists of 2N AND gates and M OR gates, each with programmable inputs from all of the AND gates.

PLAs have widely been acknowledged as compact and space-efficient solutions for many complicated circuits, especially in feedback and control systems where a number of factor variables must be involved for efficient functioning of the system

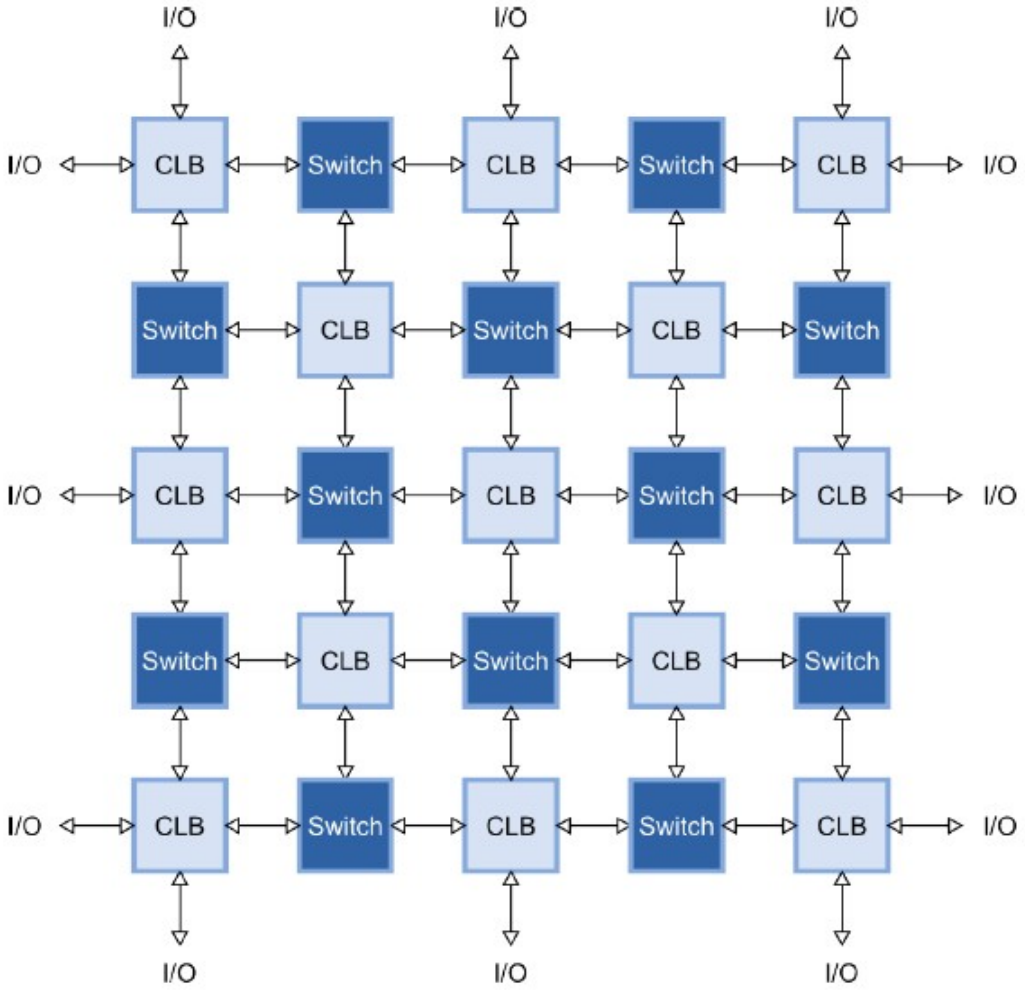
Complex Programmable Logic Device

A CPLD contains a bunch of PLD blocks like the one shown above, but their inputs and outputs are connected together by a *global interconnection matrix*. So a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.

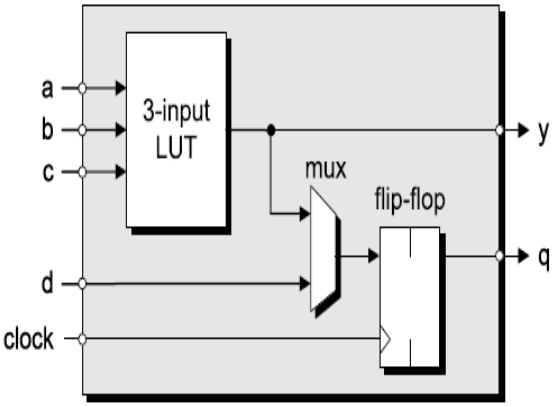


Field Programmable Gate Arrays - FPGA

An FPGA takes a different approach. It has a bunch of simple, *configurable logic blocks* (CLBs) interspersed within a *switching matrix* that can rearrange the interconnections between the them. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented.



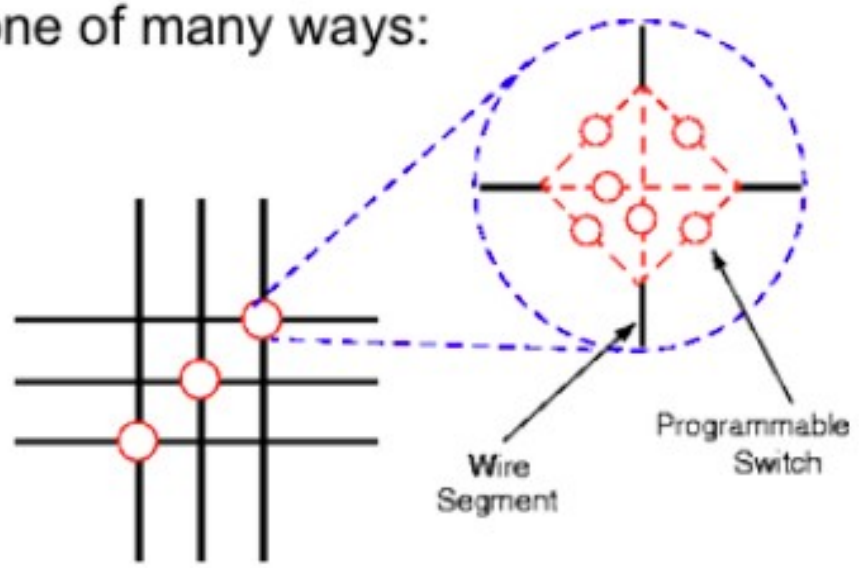
Example of CLB

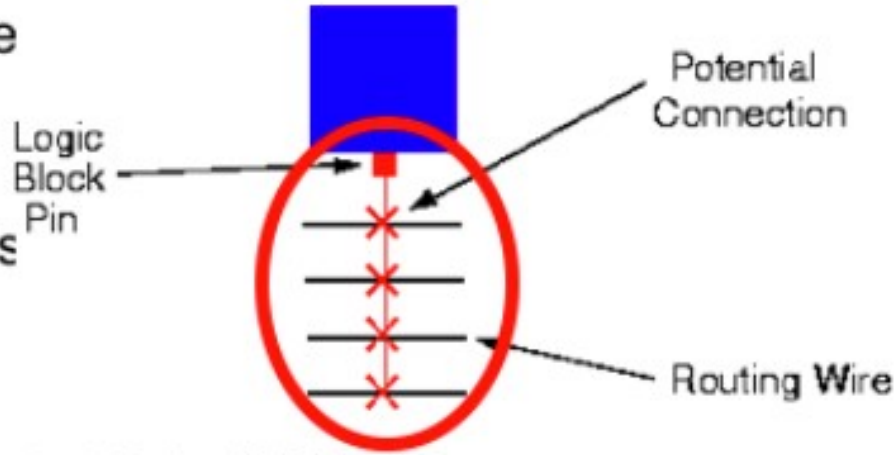


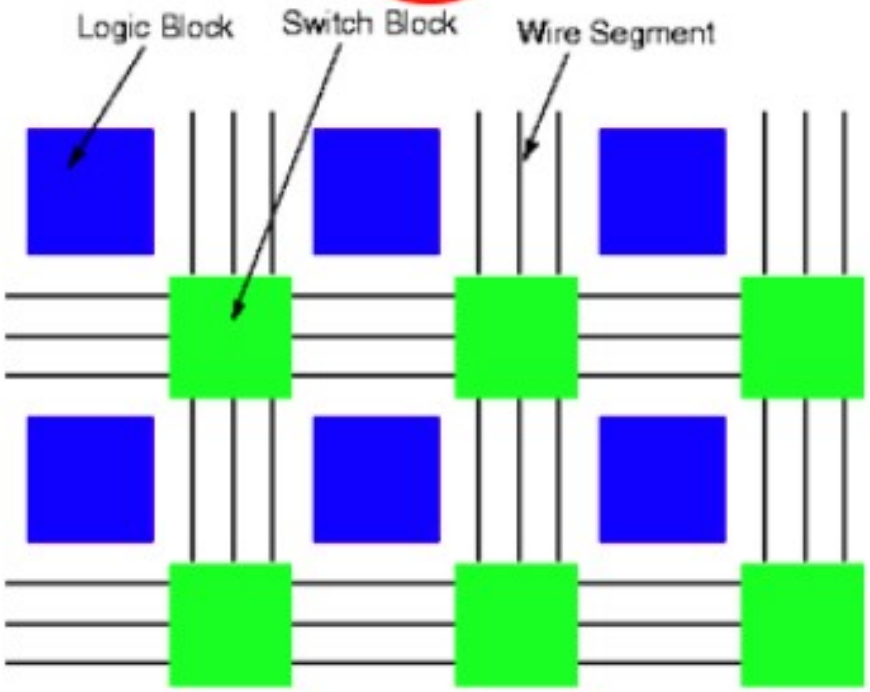
1) Look Up Table (LUT)
   → any combinatory function (3 input)
2) flip-flop (FF)
   → 1-bit memory
3) additional input
   → to combine large combinatory functions through multiplexer (MUX)

FPGA – Programmable Routing

- Between rows and columns of logic blocks are wiring channels
- These are programmable – a logic block pin can be connected to one of many wiring tracks through a programmable switch
- Xilinx FPGAs have dedicated switch block circuits for routing (more flexible)
- Each wire segment can be connected in one of many ways:

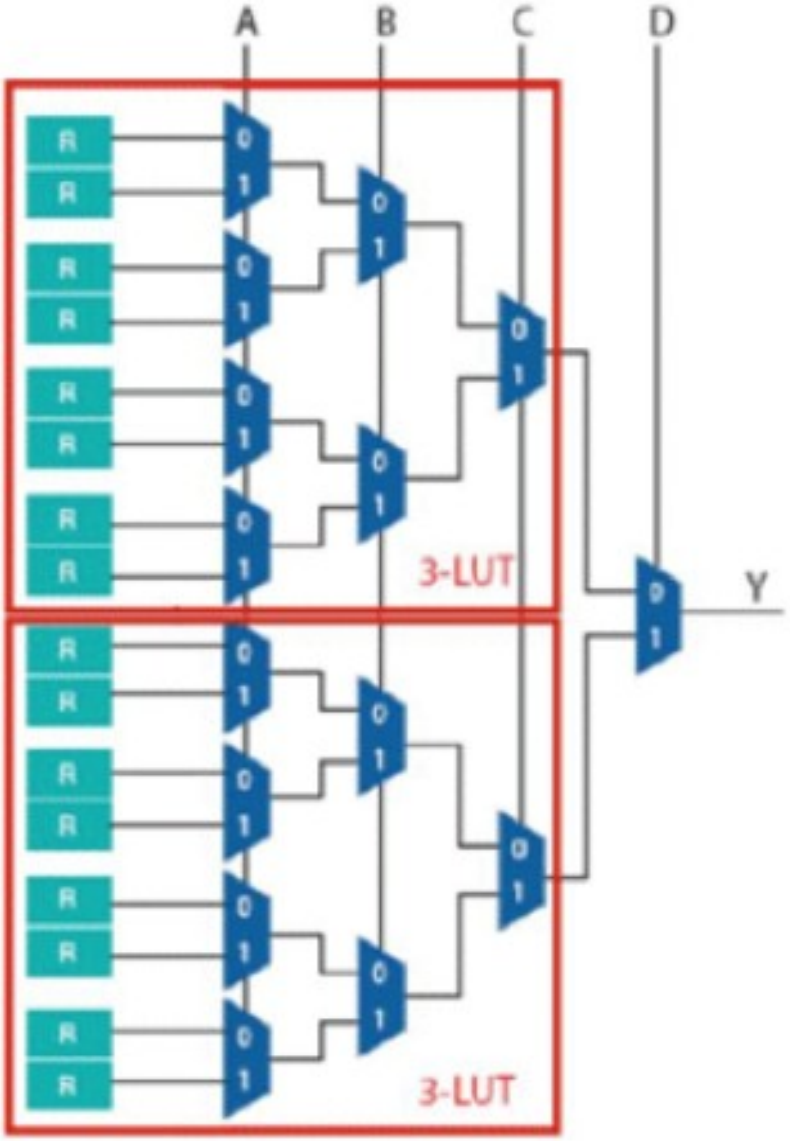






FPGA – Programming

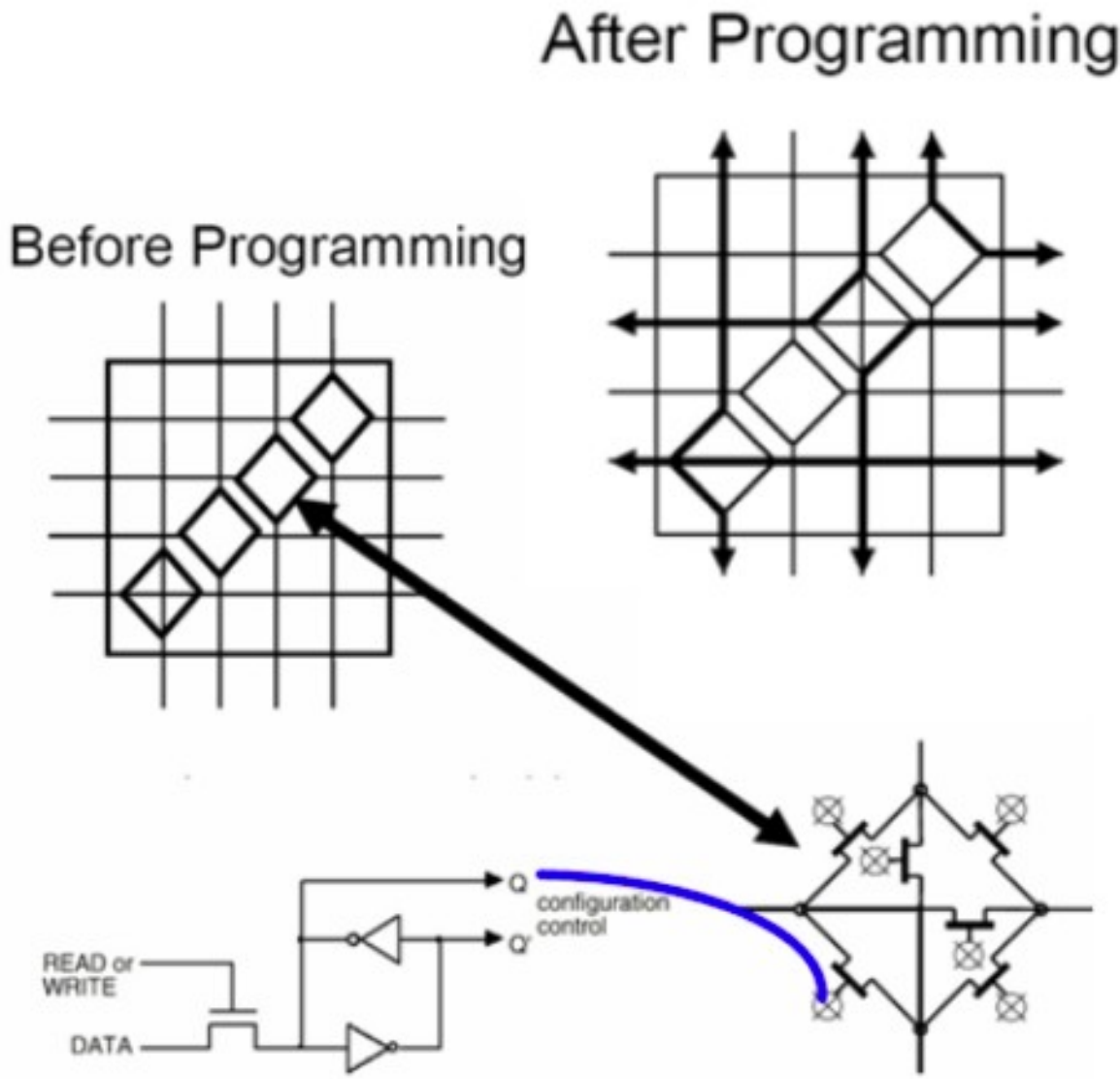
- Programming an FPGA is NOT the same as programming a microprocessor
- We download a **BITSTREAM** (not a program) to an FPGA
- Programming an FPGA is known as **CONFIGURATION**
- All LUTs are configured using the BITSTREAM so that they contain the correct values to implement the Boolean logic
- Shown here is a typical implementation of a 4-LUT circuit
  - ABCD are the FOUR inputs
  - There is four level of 2-to-1 multiplexer circuits
  - The 16-inputs to the mux tree determine the Boolean function to be implemented as in a truth-table
  - These 16 binary values are stored in registers (DFF)
  - Configuration = setting the 16 registers to 1 or 0



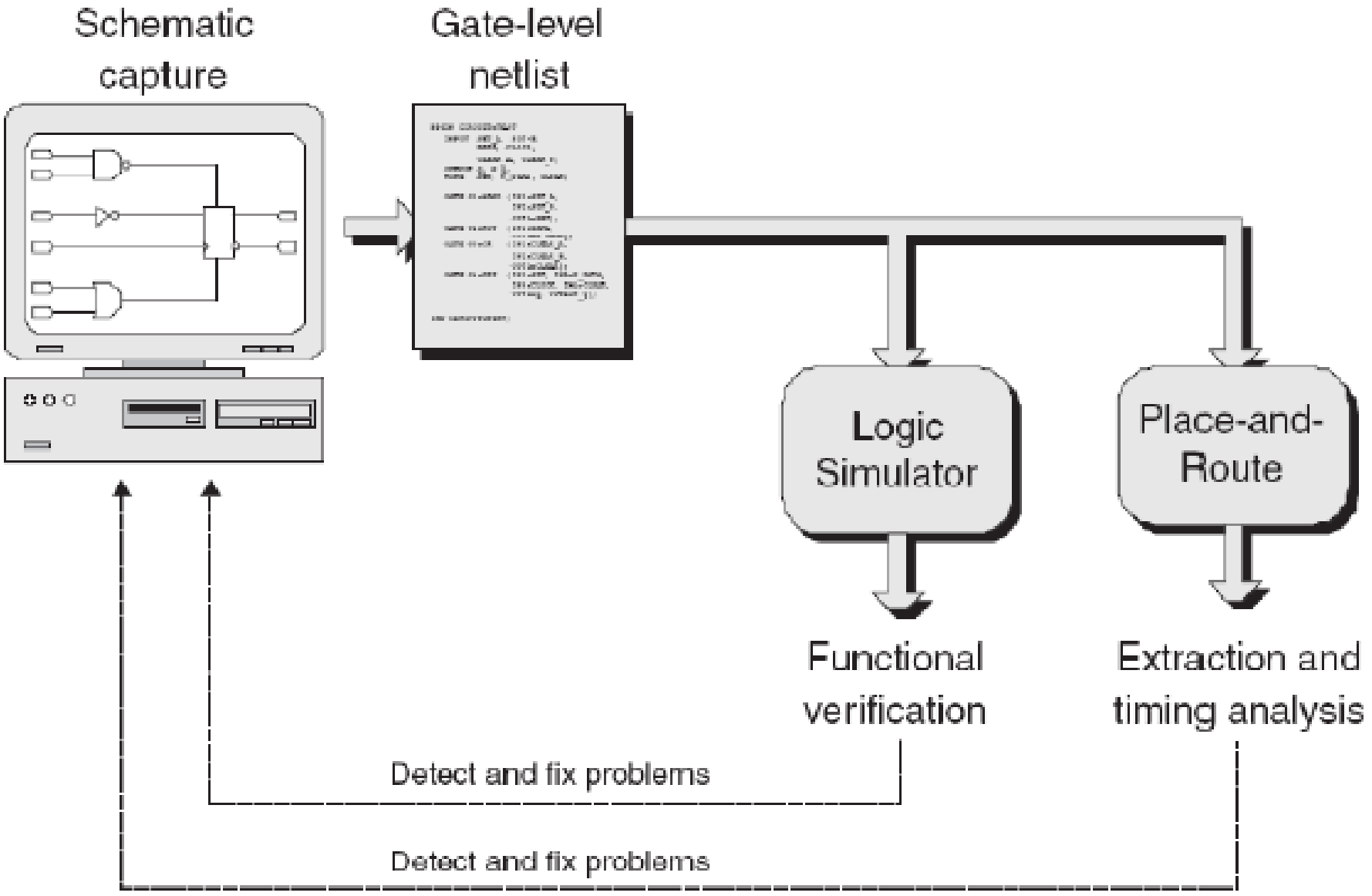
FPGA – Programming

- At each interconnect site, there is a transistor switch which is default OFF (not conducting)
- Each switch is controlled by the output of a 1-bit configuration register
- Configuring the routing is simply to put a '1' or '0' in this register to control the routing switches

- Bitstream is either stored on local flash memory or download via a computer
- Configuration happens on power-up



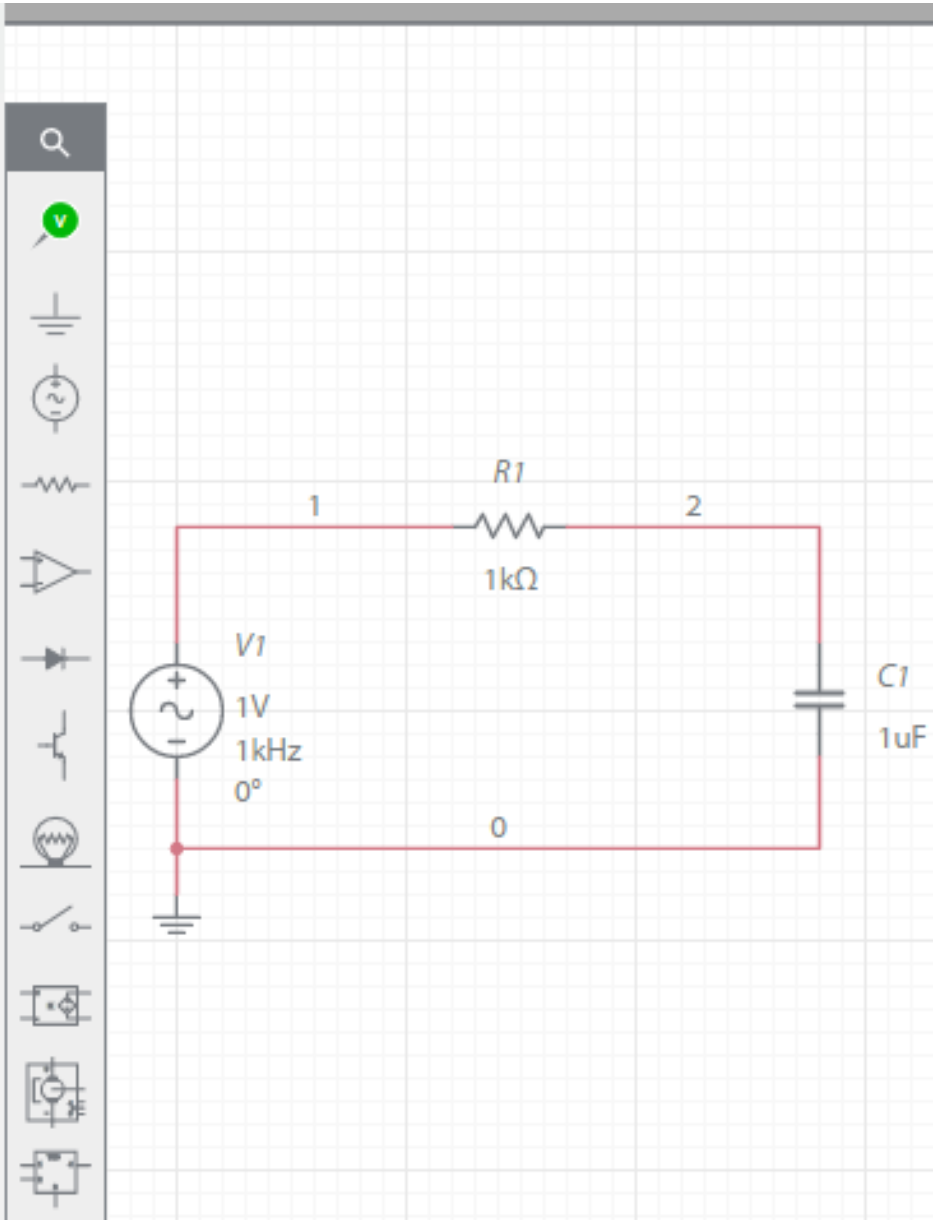
FPGA – Project flow



Note: Spice netlist
→ text-based representation of a circuit

SPICE Netlist

```
*
* Multisim Live SPICE netlist
*
*

* --- Circuit Topology ---

* Component: C1
cC1 2 0 0.000001

* Component: R1
rR1 1 2 1000 VIRTUAL_RESISTANCE_

* Component: V1
vV1 1 0 dc 0 ac 1 0
 + distof1 0 0
 + distof2 0 0
 + sin ( 0 1 1000 0 0 0 )


* --- Circuit Models ---

* R1 model
.model VIRTUAL_RESISTANCE_R1 r(
```



Introduction to VHDL

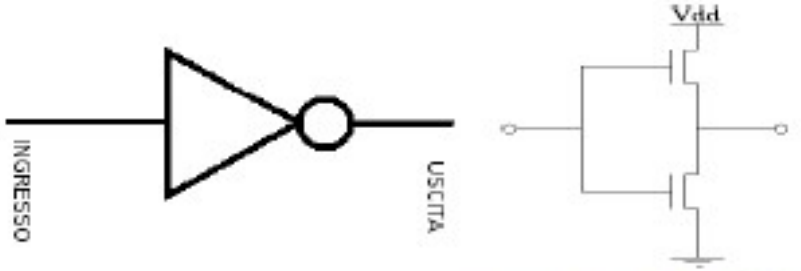
Introduction to HDL

Circuit description in the early ‘70-ies

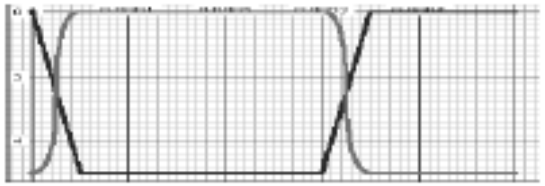
Description of the circuit with **Spice** netlist (**component** level)

Complete circuit characterization:

- Real models of the components
- extraction of Parasitic effects




```
CMOS Inverter
.lib 'models25.txt' TT
mn1 VSS IN OUT VSS nmos l=0.24u w=0.72u
mp1 VDD IN OUT VDD pmos l=0.24u w=0.72u
cLoad OUT VSS 50fF
vVDD VDD 0 2.5
vVSS VSS 0 0
vIN IN 0 pulse( 0 2.5 100ps 100ps 100ps 2ns 4ns )
.dc vIN start=0 stop=2.5 step=0.01
.tran 1ps 8ns
.option post
.end
```

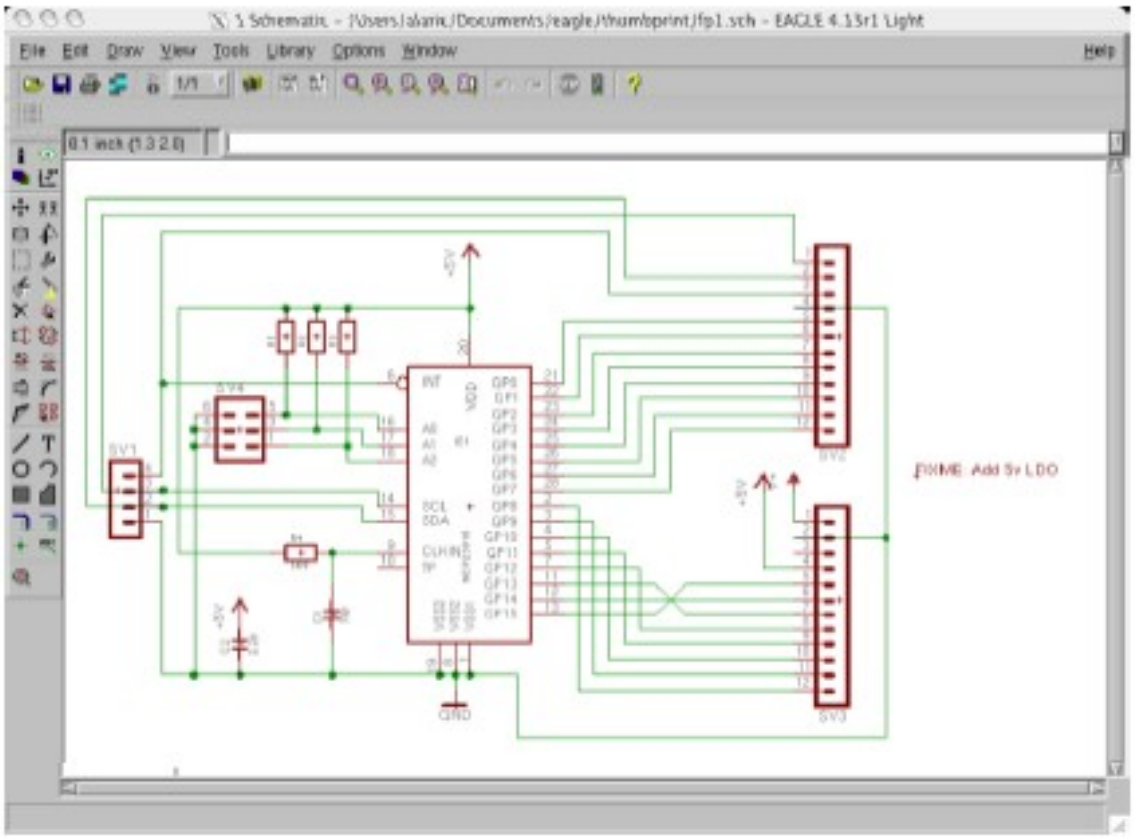


Introduction to HDL

Circuit description in the ‘80-ies

Description of the circuit **blocks** (**schematic** level)

- CAD → short developement time
- use of pre-built Libraries
- no need of full project for extracting the parasitic effects



Introduction to HDL

Circuit description in the ‘90-ies

Description of the circuit with **High Level Description Language** (**functional** level)

- use **functional blocks** described in detail

- allows for simulations with various level of accuracy

- synthesis

+ CAD
+ Libraries
+ …

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity ALU is
  port( A: in std_logic_vector(1 downto 0);
        B: in std_logic_vector(1 downto 0);
        Sel: in std_logic_vector(1 downto 0);
        Res: out std_logic_vector(1 downto 0));
end ALU;
architecture behv of ALU is begin
  process(A,B,Sel) begin
    case Sel is
      when "00" => Res <= A + B;
      when "01" => Res <= A + (not B) + 1;
      when "10" => Res <= A and B;
      when "11" => Res <= A or B;
      when others => Res <= "XX";
    end case;
  end process;
end behv;
```

Introduction to HDL

Two languages developed ad hoc

- **VHDL** (Very High speed integrated circuit – HDL)
  - developed for the US government / military
  - made standard by IEEE
  - nowadays widespread in EU industry

- **VERILOG**
  - developed by the electronics industry
  - made standard by IEEE
  - nowadays widespread in US industry

Other languages based on existing languages

- **System-C**
- System-VHDL, System-Verilog, etc...

a bit of VHDL history...

VHDL was originally developed at the behest of the US Department of Defense in order to **document** the behavior of the ASICs that supplier companies were including in equipment.
→ VHDL was developed as an alternative to huge, complex manuals which were subject to implementation-specific details

The idea of being able to **simulate** this documentation was so obviously attractive that logic simulators were developed that could read the VHDL files.
→ The next step was the development of logic **synthesis** tools that read the VHDL, and output a definition of the physical implementation of the circuit

Due to the Department of Defense requiring as much of the syntax as possible to be based on Ada, in order to avoid re-inventing concepts that had already been thoroughly tested in the development of Ada, VHDL borrows heavily from the Ada programming language in both concepts and syntax

The initial version of VHDL, designed to IEEE standard 1076-1987, included a wide range of data types, including numerical (integer and real), logical (bit and boolean), character and time, plus arrays of bit called bit_vector and of character called string…

… long history up to the most recent VHDL standard IEEE 1076-2008 published in January 2009

Introduction to VHDL

How can modern VHDL describe a circuit ? Choices are:

1. Schematic capture (graphical mode)
- draw the circuit blocks → instantiate the logical components
- synthesis tool → maps the circuit realized within the device

2. Hardware Description Language (code list mode)
- describe by appropriate language the operation of our system
- synthesis tool → translates the code into the component‘s structures

Mix of the two modes also possible ! → Mixed Mode

Introduction to VHDL

Looks like a “normal“ programming language (typically C):
- various types of data and objects (constants, variables, expressions)
- arithmetic / logical operators, sequential instructions (if, while, for ...)
- functions, sub-programs

Implement specific elements to efficiently model hardware blocks:
- allow to define components and instantiate them in a **hierarchical structure**
- support the representation of **concurrent events**
  (ie, operations that are not always activated in the same temporal order)

Allow to describe the system at different levels of abstraction
- from the highest level (system) up to the logical level (network of logic gates)
- traditional representation systems usually are specialized for one or two levels:
  - block diagrams ⇒ system / data-flow;
  - logical schemes ⇒ logical level;
  - electric schemes ⇒ electrical level;

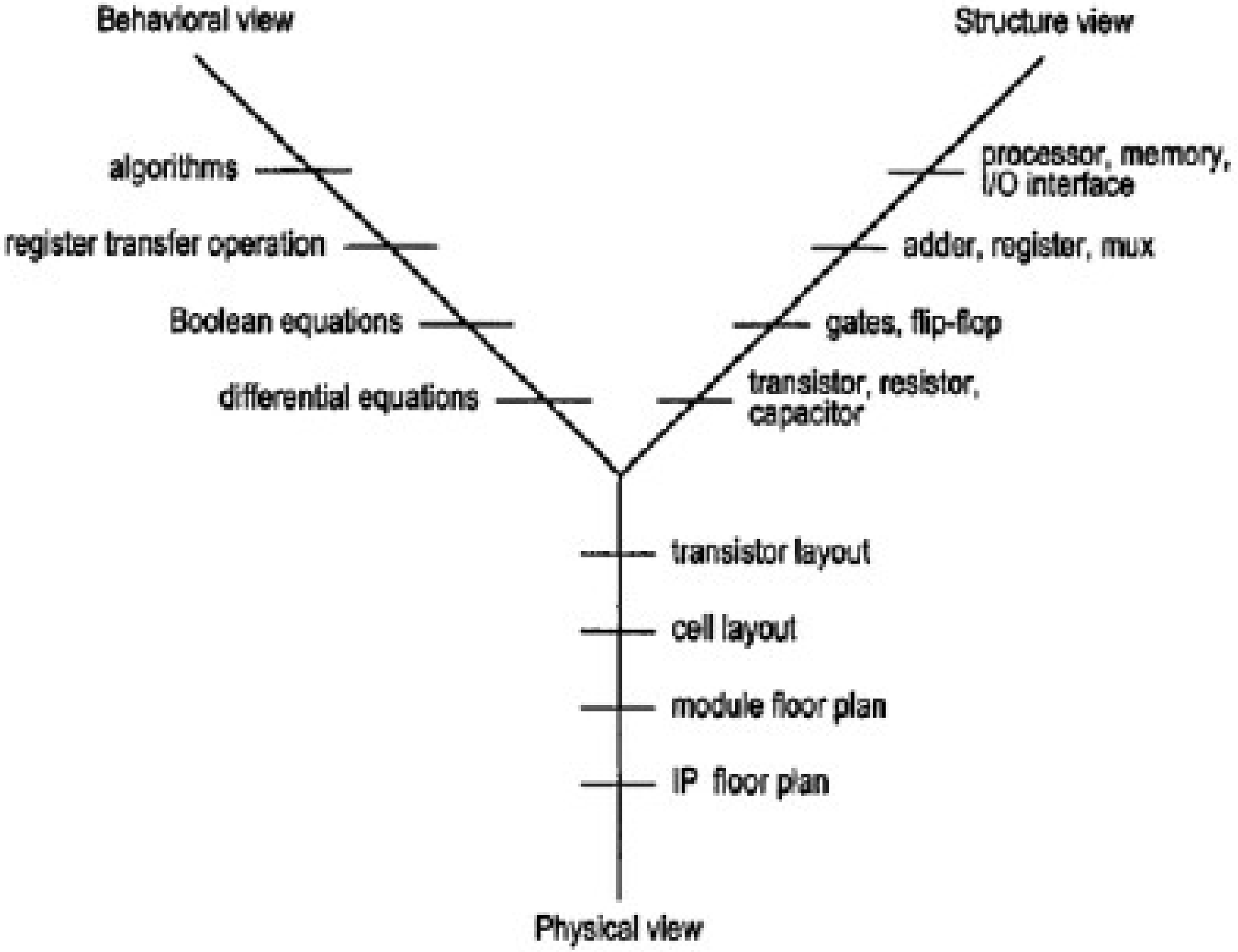
VHDL description modes

Behavioral
- system functionality description (by algorithms, conditions, loops)
  regardless of signals flow and implementation at elementary cells level
- correct operations sequence BUT no time information

Register Transfer Level (RTL or Data-Flow)
- description of the system in terms of signal flow among registers
  including connections (registers, combinational logic, bus, control units...)
- assignment of operations to a given clock cycle

Structural
- description of the system as a network of elementary components
  (described in behavioral level)
- detailed analysis of delay times possible

VHDL description modes



VHDL project flow

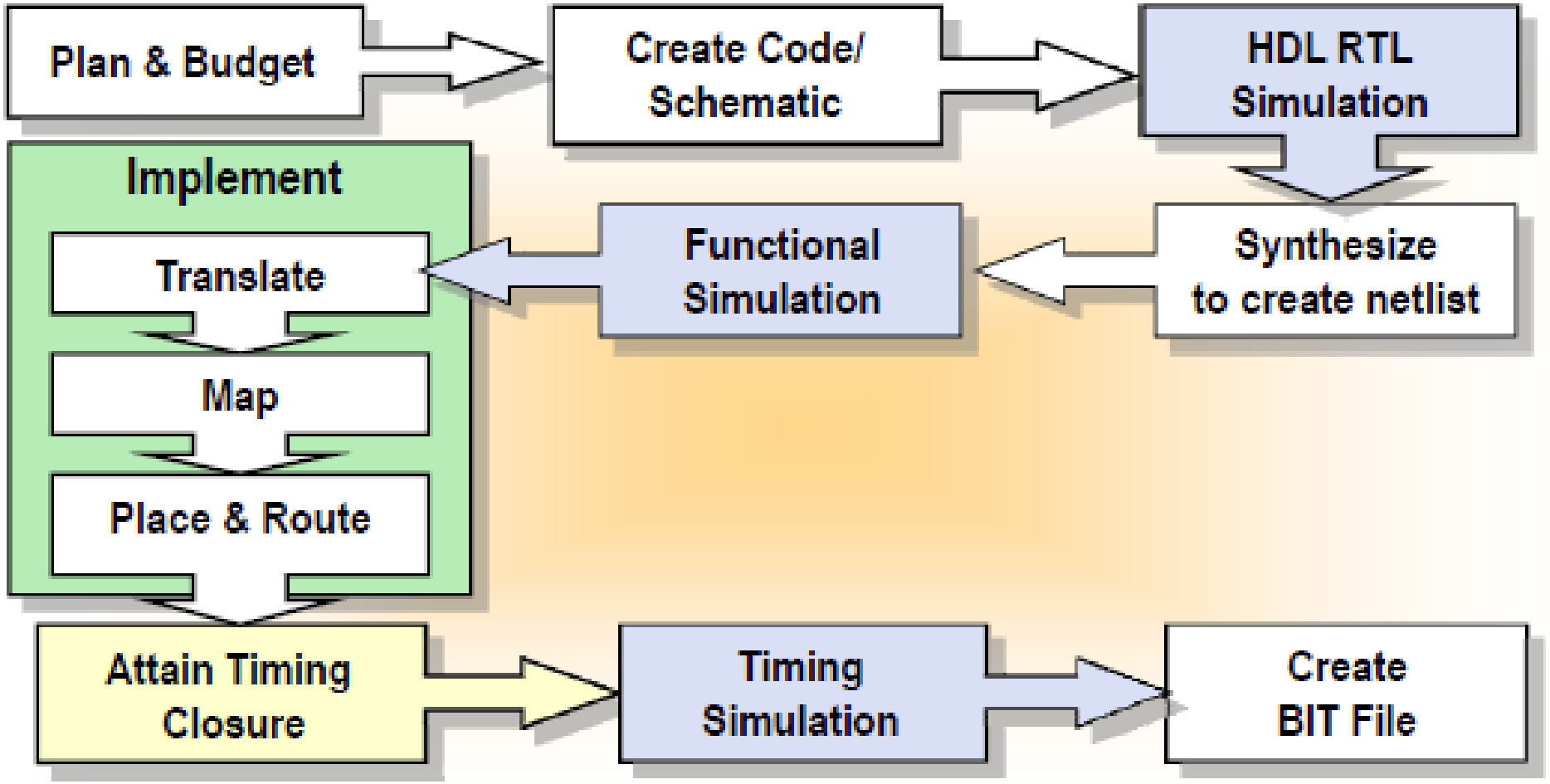
- VHDL supports all levels of the design flow
  development → simulation → synthesis → testing + documentation

- allows **description independent** of the implementation technology
  ⇒ portability of the project from one technology to another

- allows system event **simulation**
  validation of very complex systems

- allows automatic **synthesis**
  - from RTL to cell netlist (standard)
  - from behavioral to RTL (not standard)

VHDL project flow (Xilinx)



VHDL in practice…

Good practice rules:
- write your code considering that instructions are translated into circuits
- which are processing / reacting to real signals or events

I - VHDL basics

1. entity and architecture
2. signals and processes
3. simulation
4. synthesis

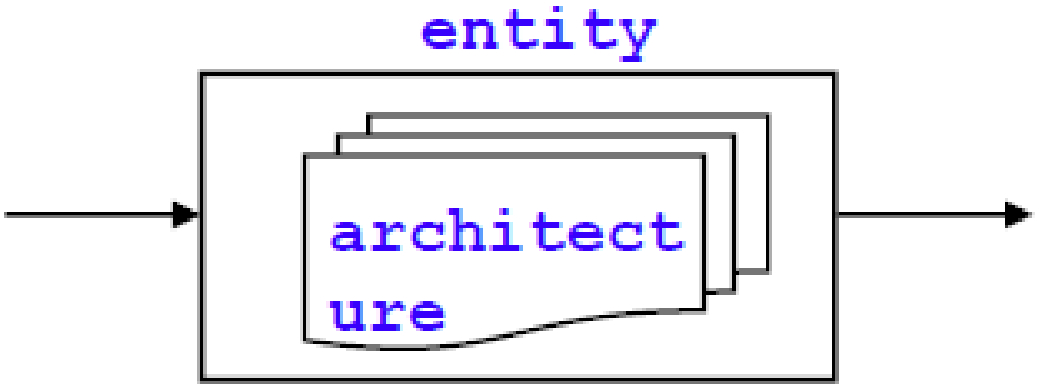
1 - VHDL description basics

Entity & Architecture

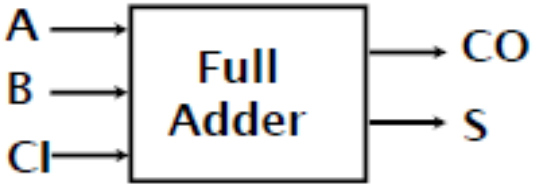
The description of a module in VHDL is made up of two main elements:

- an interface (entity)
  → defines the I / O terminals and the name of the circuit

- one or more implementations (architecture)
  → describe the behavior or the internal structure of the circuit



1 - VHDL description basics: Entity

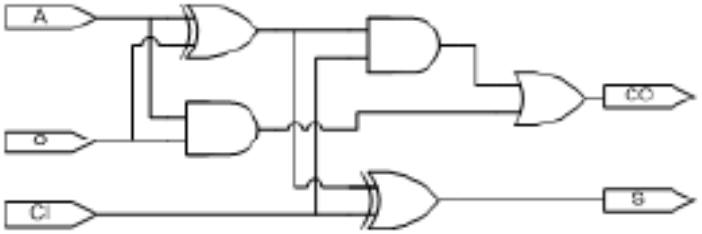


ENTITY

It defines the (black) box:
- name
- terminals w/ directon (in/out)
- data type

```
entity FULL_ADDER is
   port (A, B, CI: in bit;
         CO, S: out bit);
end FULL_ADDER;

architecture BHV of FULL_ADDER is
  signal P, G: bit;
begin
  P  <= A xor B;
  G  <= A and B;
  S  <= P xor CI;
  CO <= (P and CI) or G;
end BHV;
```

1 - VHDL description basics: Architecture



ARCHITECTURE

Defines the box functionality:

```
architecture … of … is
```

Includes the following parts:

- **Declaration** (berore **`begin`**)
  decl. internal signals/variables

- **Assertions** (**`begin`** to **`end`**)
  describes circuit behaviour by using
  - **operations** among signals (internal and/or I/O) and
  - **assignement** (<=)

```
entity FULL_ADDER is
   port (A, B, CI: in bit;
         CO, S: out bit);
end FULL_ADDER;

architecture BHV of FULL_ADDER is
  signal P, G: bit;
begin
  P  <= A xor B;
  G  <= A and B;
  S  <= P xor CI;
  CO <= (P and CI) or G;
end BHV;
```

Note: these are not “instructions“ to be executed once: they are descriptions of operations that are always ready to be activated, into the described hardware

2 - VHDL main model elements: signal & process

Main elements of a VHDL model: signals and processes

- a **process** is a block of code that describes the operation of a logic module (**sequential** or **combinatorial**)

- different **processes communicate** with each other through **signals**

- a **process is activated** and executes its function in **response to an event**
ie, to a change in value of one of the signals to which it is sensitive
(indicated in an appropriate list appropriate)

- in some cases, the function represented by a process can be described with one concurrent **assignment** simple or conditioned to a signal

```
architecture BHV2 of AOI is
begin
  Z <= C nor (A and B);
end BHV;
```

simple assignment

```
architecture BHV3 of AOI is
begin
  Z <= '0' when C = '1' else
       A and B;
end BHV;
```

concurrent condition

```
architecture BHV1 of AOI is
begin
  process(A,B,C) begin
    if C = '1' then
      Z <= '0';
    else
      Z <= A nand B;
    end if;
  end process;
end BHV;
```

sequential condition

2 - VHDL main model elements: signals

Signals are data structures able
to represent (digital) waveforms in time

The **signal** instruction can be **declared**
- in the **architecture** with the signal statement or
- in the clause **port** of the **entity**

```
signal P, G: bit;
...
P  <= A xor B;
G  <= A and B;
S  <= P xor CI;
...
```

• signals are associated with a **type** (e.g bit) and may have an **initial value**

• in **architecture** assertive part there are
  - operations between signals (**xor, and, ...**) and
  - assignments (**<=**);

operators can be primitive of the VHDL or user-defined functions and procedures

• to the left of the **assignment** sign (**<=**) is the **target** (which must be a signal) (the output)
  to the right the **signal driver** (the input)

• the **assignment** establishes a link between
its inputs (the signals contained in the driver) and its output (the target) so it is **activated** only when there is an event (ie a change in value) of one of the inputs

• consequently, the **order** with which assignments are written inside
of the architecture **does not matter**

3 - VHDL simulation

VHDL simulator is a program capable of applying
a temporal succession of the inputs (input waveforms),
and determine the time sequence of the outputs (waveform)
of a circuit described by a VHDL model

It is an event driven simulator, whose ingredients are:
1) internal model of time;
2) signals update;
3) execution of the processes and the drivers (event processing)

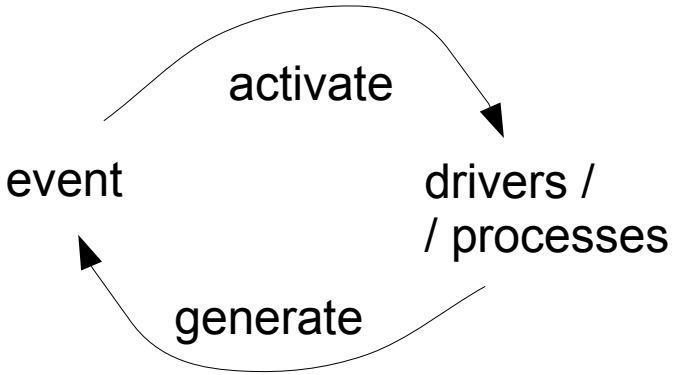
Note: an event is the change in the value of a signal, to which it is associated
an internal time which represents the operating time of the simulated circuit
(… it is NOT the time that the computer takes to run the simulation)

3 - VHDL simulation

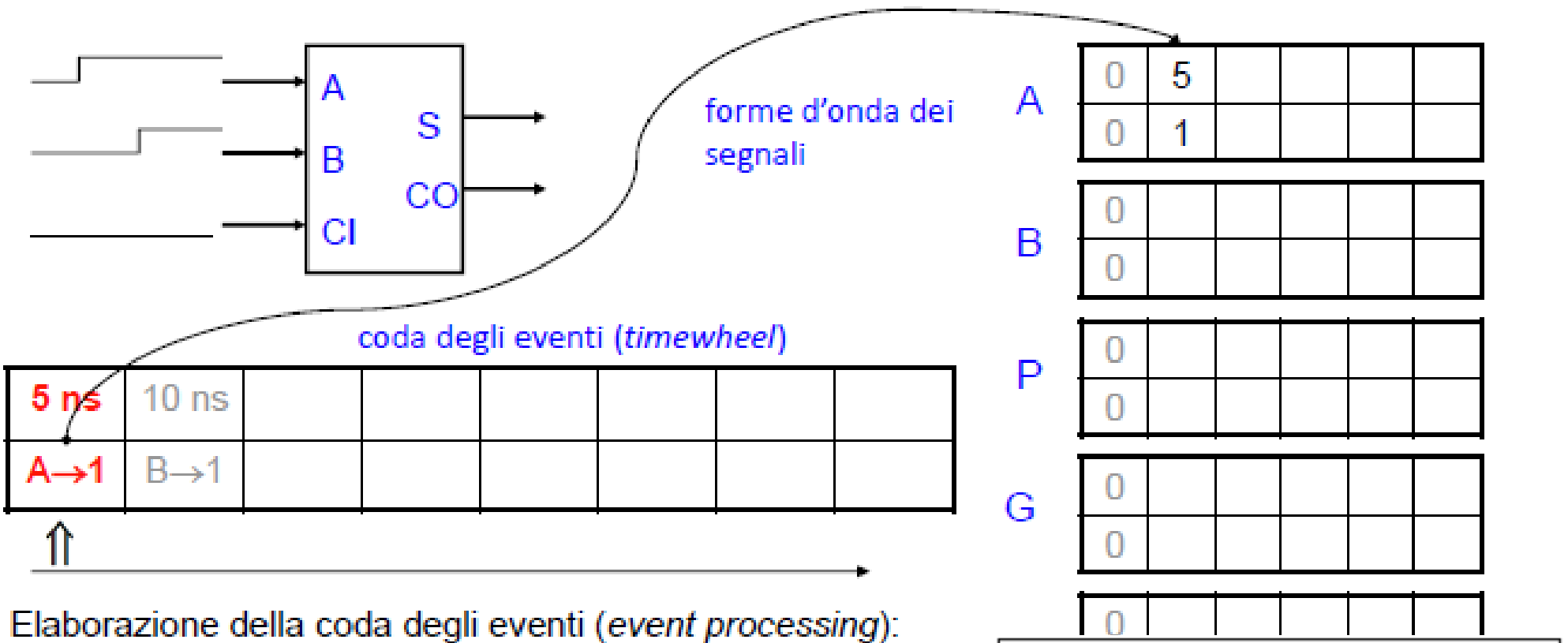
The simulation proceeds by jumps from one event to the next one
→ jumps along the queue of input events

1.a each event activates the drivers and processes that are sensitive to the signal triggered the event
1.b the drivers calculate the new target signal value
1.c the new value ("transaction") is inserted in the queue of the events
→ new transactions (potential events) inserted within the queue
The execution of all drivers sensitive to the event causes the insertion of one or more transactions in the simulation event queue

2. when the driver execution phase is over, the simulation jumps to the next transaction in the queue and update the value of the related signal:
→ if the updated value is different from the previous value we have an event → a new phase of execution of the drivers sensitive to it
→ otherwise the simulator jumps to the next transaction in the queue



3 - VHDL simulation: example

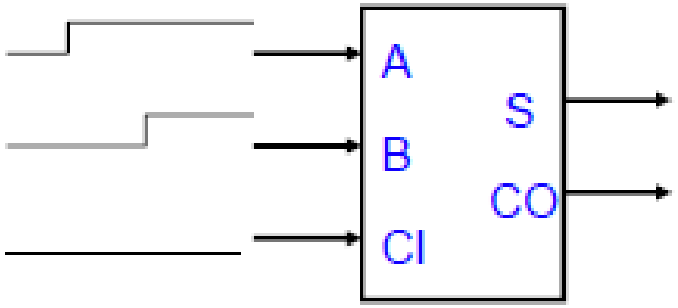


Elaborazione della coda degli eventi (*event processing*):

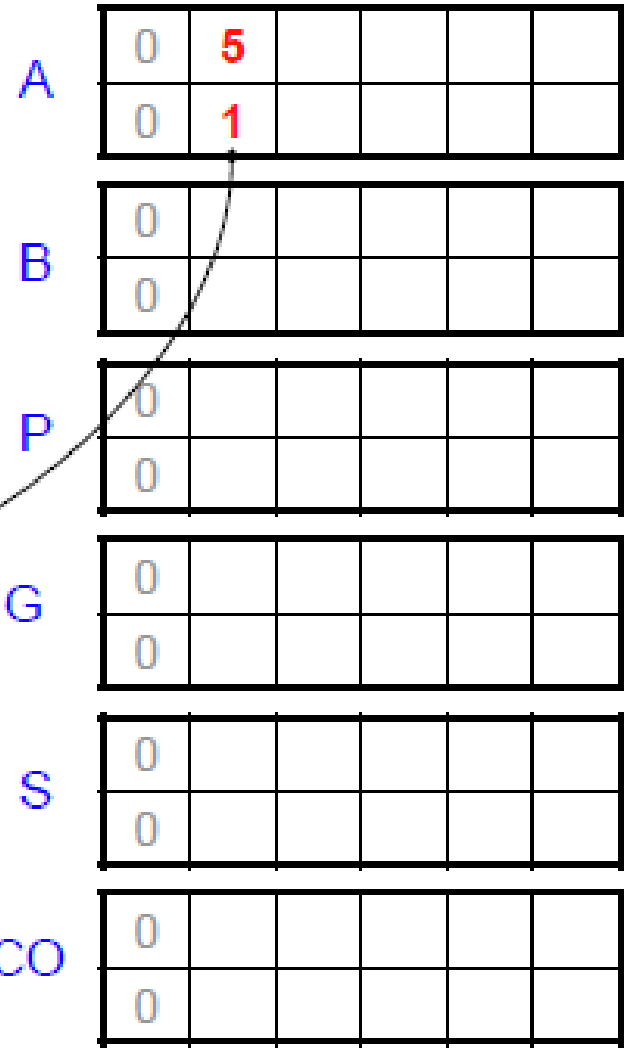
il simulatore carica la prima transazione della coda (A →1), confronta il nuovo valore di A (1) con il vecchio (0), e, se sono diversi, inserisce un evento nella forma d'onda del segnale

```
entity FULL_ADDER is
   port (A, B, CI: in bit;
         CO, S: out bit);
end FULL_ADDER;

architecture BHV of FULL_ADDER is
  signal P, G: bit;
begin
  P  <= A xor B;
  G  <= A and B;
  S  <= P xor CI;
  CO <= (P and CI) or G;
end BHV;
```

3 - VHDL simulation: example



forme d'onda dei segnali



coda degli eventi (*timewheel*)

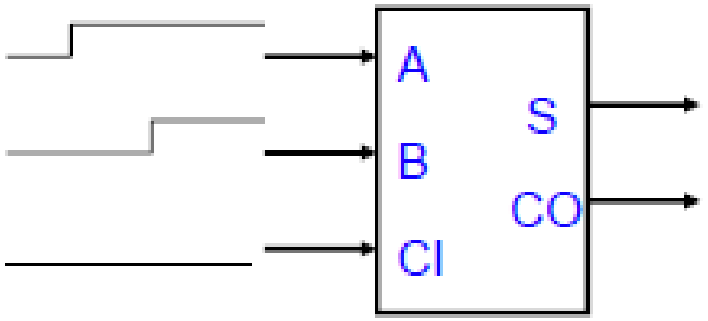
5 ns	5 ns	5 ns	10 ns				
A→1	P→1	G→0	B→1				

```
P <= A xor B;

G <= A and B;
```

esecuzione dei processi e dei driver

driver sensibili al segnale A

3 - VHDL simulation: example



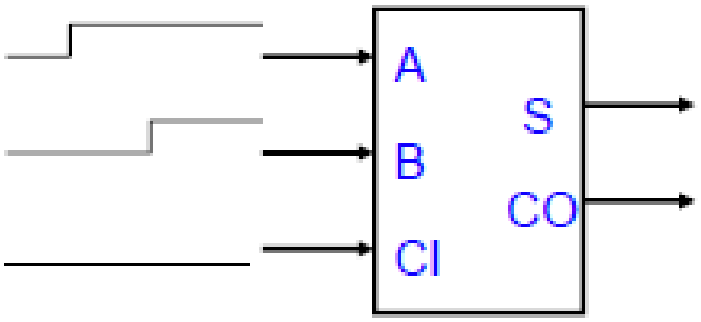
5 ns	5 ns	5 ns	10 ns				
A→1	P→1	G→0	B→1				

A	0	5					
	0	1					
B	0						
	0						
P	0	5					
	0	1					
G	0						
	0						
S	0						
	0						
CO	0						
	0						

Elaborazione della coda degli eventi (*event processing*):

il simulatore carica la transazione successiva (P →1), confronta il nuovo valore di P (1) con il vecchio (0), e, se sono diversi, inserisce un evento nella forma d'onda del segnale

3 - VHDL simulation: example



forme d'onda dei segnali

A	0	5				
	0	1				
B	0					
	0					
P	0	5				
	0	1				
G	0					
	0					
S	0					
	0					
CO	0					
	0					

coda degli eventi (*timewheel*)

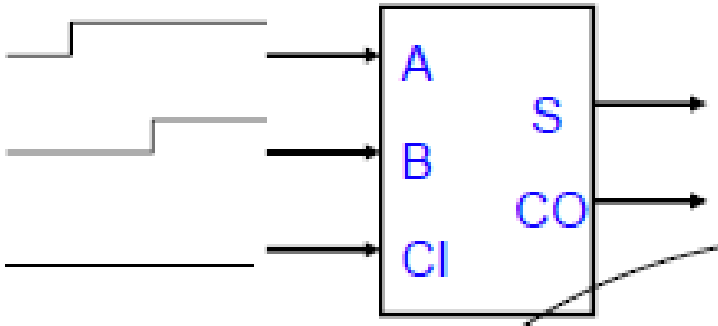
5 ns	5 ns	5 ns	5 ns	5 ns	10 ns		
A→1	P→1	G→0	S→1	CO →0	B→1		

```
S <= P xor CI;

CO <= (P and CI) or G;
```

esecuzione dei processi e dei driver

driver sensibili al segnale P

3 - VHDL simulation: example



il valore di G non è cambiato: NO evento

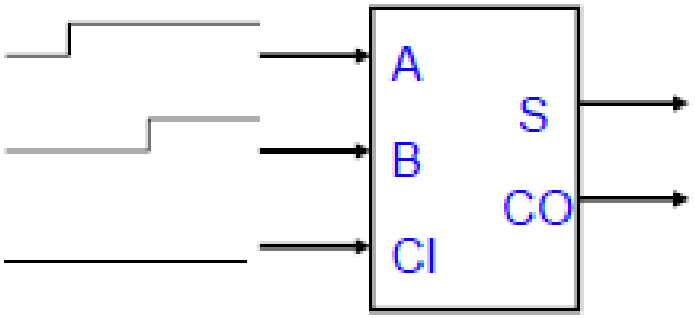
5 ns	5 ns	5 ns	5 ns	5 ns	10 ns		
A→1	P→1	G→0	S→1	CO →0	B→1		

A	0	5				
	0	1				
B	0					
	0					
P	0	5				
	0	1				
G	0					
	0					
S	0					
	0					
CO	0					
	0					

Elaborazione della coda degli eventi (*event processing*):

il simulatore carica la transazione successiva (G →0), confronta il nuovo valore di G (0) con il vecchio (0), e, siccome il valore di G non è cambiato, non inserisce nessun evento nella forma d'onda; i driver e i processi sensibili a G **non si attivano**

3 - VHDL simulation: example



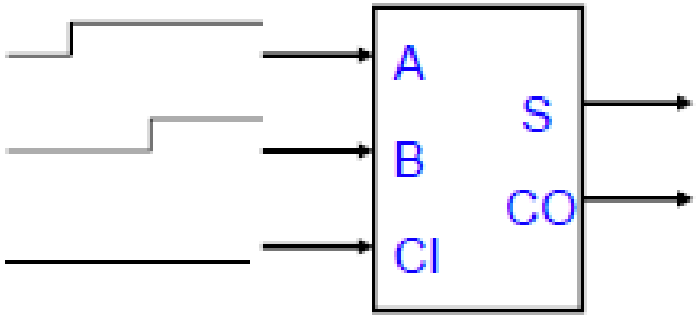
5 ns	5 ns	5 ns	**5 ns**	5 ns	10 ns		
A→1	P→1	G→0	**S→1**	CO →0	B→1		

A	0	5				
	0	1				
B	0					
	0					
P	0	5				
	0	1				
G	0					
	0					
S	0	5				
	0	1				
CO	0					
	0					

Elaborazione della coda degli eventi (*event processing*):

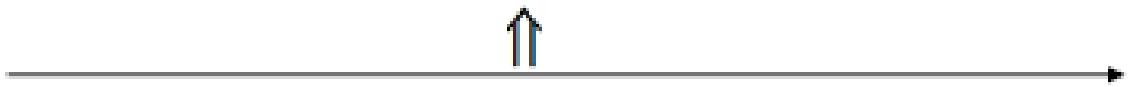
il simulatore carica la transazione successiva (S →1), confronta il nuovo valore di S (1) con il vecchio (0), e inserisce il nuovo valore nella forma d'onda; non ci sono driver e/o processi sensibili a S, per cui il simulatore passa alla transazione successiva

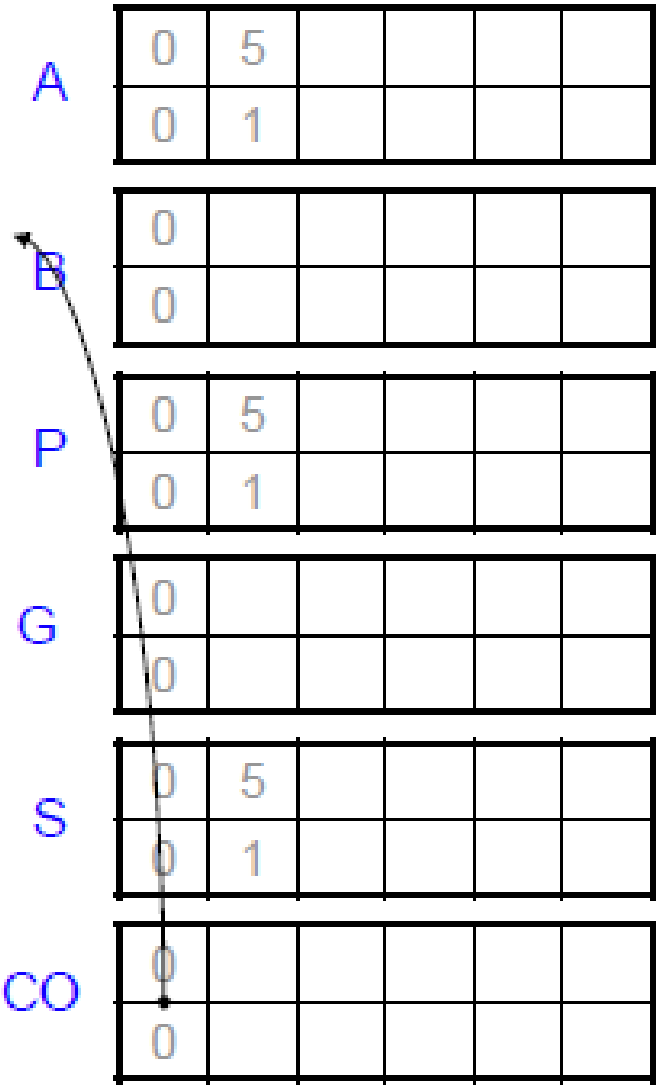
3 - VHDL simulation: example



il valore di CO non è cambiato: NO evento

5 ns	5 ns	5 ns	5 ns	**5 ns**	10 ns		
A→1	P→1	G→0	S→1	**CO →0**	B→1		

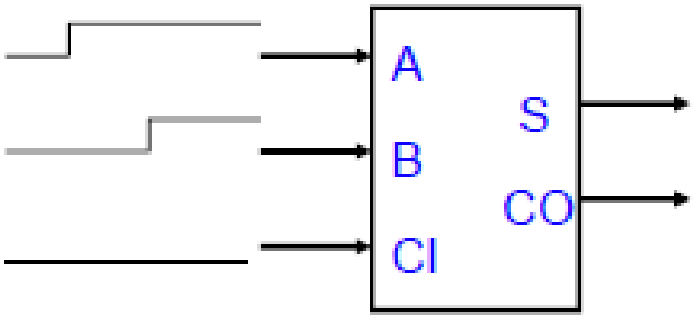




Elaborazione della coda degli eventi (*event processing*):

CO non cambia valore, per cui non si ha un evento; il simulatore passa alla transazione successiva nella coda

3 - VHDL simulation: example



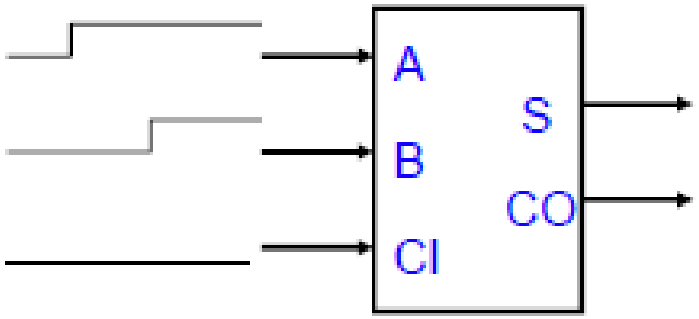
5 ns	5 ns	5 ns	5 ns	5 ns	10 ns		
A→1	P→1	G→0	S→1	CO →0	B→1		

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5			
	0	1			
G	0				
	0				
S	0	5			
	0	1			
CO	0				
	0				

Elaborazione della coda degli eventi (*event processing*):

B va a 1, il che causa l'inserimento di un evento nella sua forma d'onda

3 - VHDL simulation: example



…	5 ns	5 ns	10 ns	10 ns	10 ns		
…	S→1	CO→0	B→1	P→0	G→1		

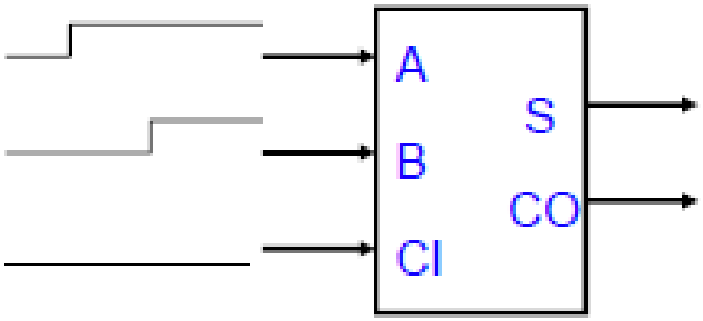
```
P <= A xor B;

G <= A and B;
```

driver sensibili al segnale B

l'evento di B causa transazioni di P e G

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5			
	0	1			
G	0				
	0				
S	0	5			
	0	1			
CO	0				
	0				

3 - VHDL simulation: example



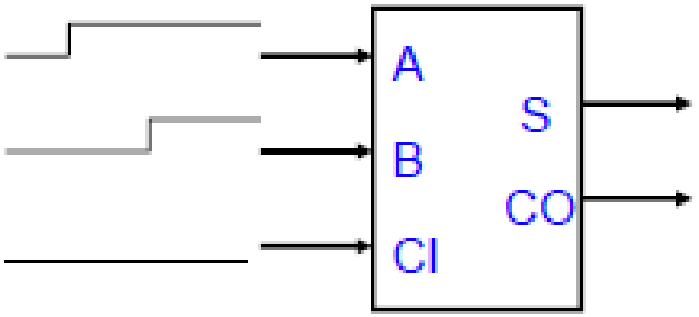
…	5 ns	5 ns	10 ns	**10 ns**	10 ns		
…	S→1	CO→0	B→1	**P→0**	G→1		

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0				
	0				
S	0	5			
	0	1			
CO	0				
	0				

Elaborazione della coda degli eventi (*event processing*):

P va a 0, il che causa l'inserimento di un evento nella sua forma d'onda

3 - VHDL simulation: example



…	10 ns	10 ns	10 ns	10 ns	10 ns		
…	B→1	P→0	G→1	S→0	CO→0		

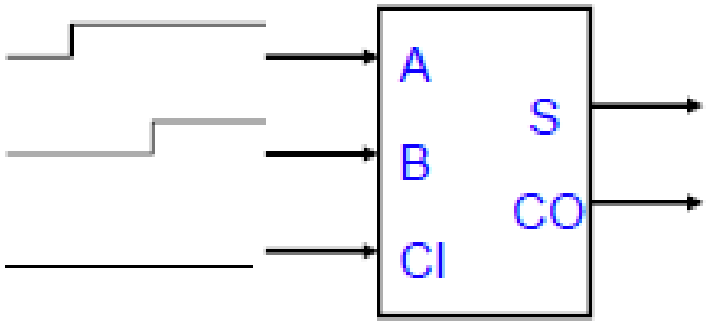
```
S <= P xor CI;

CO <= (P and CI) or G;
```

driver sensibili al segnale P

esecuzione dei processi e dei driver

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0				
	0				
S	0	5			
	0	1			
CO	0				
	0				

3 - VHDL simulation: example



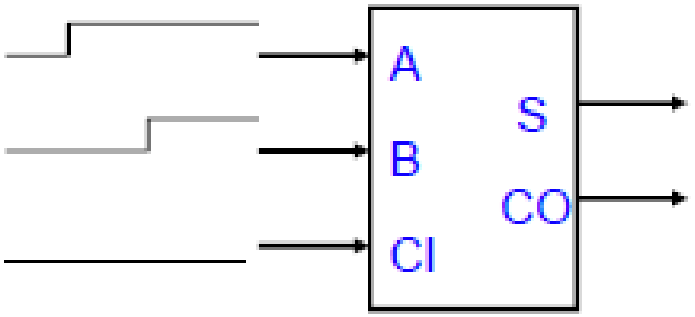
…	10 ns	10 ns	**10 ns**	10 ns	10 ns		
…	B→1	P→0	**G→1**	S→0	CO→0		

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0	10			
	0	1			
S	0	5			
	0	1			
CO	0				
	0				

Elaborazione della coda degli eventi (*event processing*):

G va a 1, il che causa l'inserimento di un evento nella sua forma d'onda

3 - VHDL simulation: example



…	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns
…	B→1	P→0	G→1	S→0	CO→0	CO→1

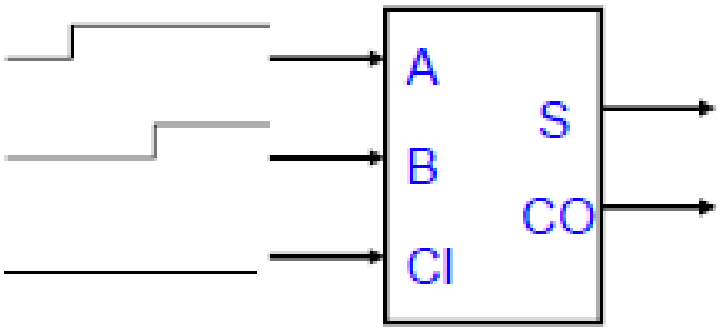
```
CO <= (P and CI) or G;
```

driver sensibili al segnale G

esecuzione dei processi e dei driver

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0	10			
	0	1			
S	0	5			
	0	1			
CO	0				
	0				

3 - VHDL simulation: example



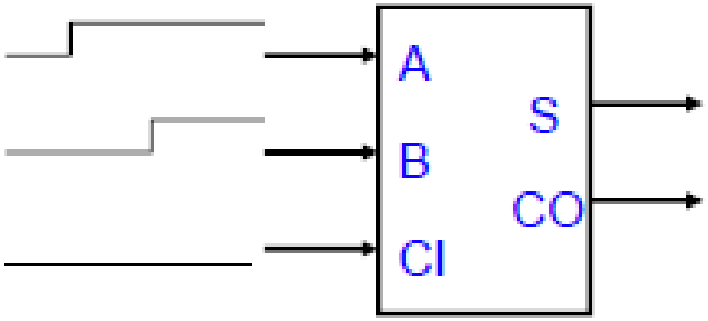
…	10 ns	10 ns	10 ns	**10 ns**	10 ns	10 ns
…	B→1	P→0	G→1	**S→0**	CO→0	CO→1

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0	10			
	0	1			
S	0	5	10		
	0	1	0		
CO	0				
	0				

Elaborazione della coda degli eventi (*event processing*):

S va a 0, il che causa l'inserimento di un evento nella sua forma d'onda; non ci sono driver e/o processi sensibili a S, per cui il simulatore passa alla transazione successiva

3 - VHDL simulation: example



il valore di CO non è cambiato: NO evento

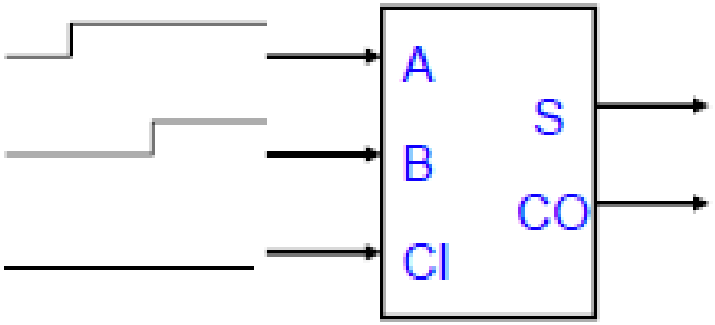
…	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	
…	B→1	P→0	G→1	S→0	CO→0	CO→1	

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0	10			
	0	1			
S	0	5	10		
	0	1	0		
CO	0				
	0				

Elaborazione della coda degli eventi (*event processing*):

CO non cambia valore, per cui non si ha un evento; il simulatore passa alla transazione successiva nella coda

3 - VHDL simulation: example



…	10 ns	10 ns	10 ns	10 ns	10 ns	**10 ns**
…	B→1	P→0	G→1	S→0	CO→0	**CO→1**

A	0	5			
	0	1			
B	0	10			
	0	1			
P	0	5	10		
	0	1	0		
G	0	10			
	0	1			
S	0	5	10		
	0	1	0		
CO	0	10			
	0	1			

Elaborazione della coda degli eventi (*event processing*):

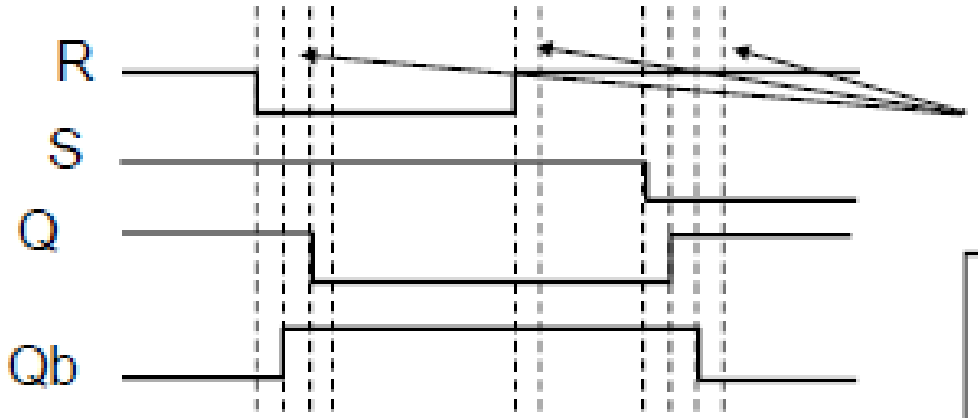
CO va a 1, il che causa l'inserimento di un evento nella sua forma d'onda; non ci sono driver e/o processi sensibili a CO; non ci sono altri eventi nella coda, per cui la simulazione termina

3 - VHDL simulation: example #2

- suppose that initially Q = R = S = '1', Qb = '0',
  then that at a certain moment, in the events queue, a transaction of R to '0' appears

- when the simulator arrives at the transaction, this becomes an event because
  the value of R changes, then the following sequence starts:
  - driver: activates D2, which is sensitive to R and
    → inserts a transaction to '1' of Qb in the queue with null delay
  - events: the simulator advances a delta time (null), meets the transaction of Qb
    update its value → have an event
  - driver: the event on Qb activates D1 which inserts a transaction to '0' of Q
  - events: the simulator advances another delta, updates Q → have a new event
  - driver: the event on Q activates D2 which inserts a transaction to '1' of Qb
  - events: the simulator advances another delta, updates Qb but does not change value;
    the events are over

  → the drivers D1 and D2 remain inactive until any other events on R or on S



cicli ***delta*** di durata nulla; si fermano quando non succede più niente

```
D1: Q <= S nand Qb;

D2: Qb <= R nand Q;
```

4 - VHDL synthesis

VHDL was born as a language for **description** and **simulation**:
→ the synthesis process must interpret the language to infer the corresponding logic circuit
→ not all VHDL constructs can be **synthesized**

The synthesis process ("compilation") can be described in 3 phases:

1. **translation**: model transformed into a network of elementary operations
   (simple combinatorial gates, MUX/DEMUX, registers)
2. **optimization**: the logical network is optimized using various algorithms to satisfy
   the constraints on propagation times and minimize the area (and/or power) of the circuit
3. **technological mapping**: the optimized network is transformed to be
   achievable with the cells contained in a technological library
   (library of standard cells or cells that can be implemented on FPGA)

→ the result is a network of physically feasible cells

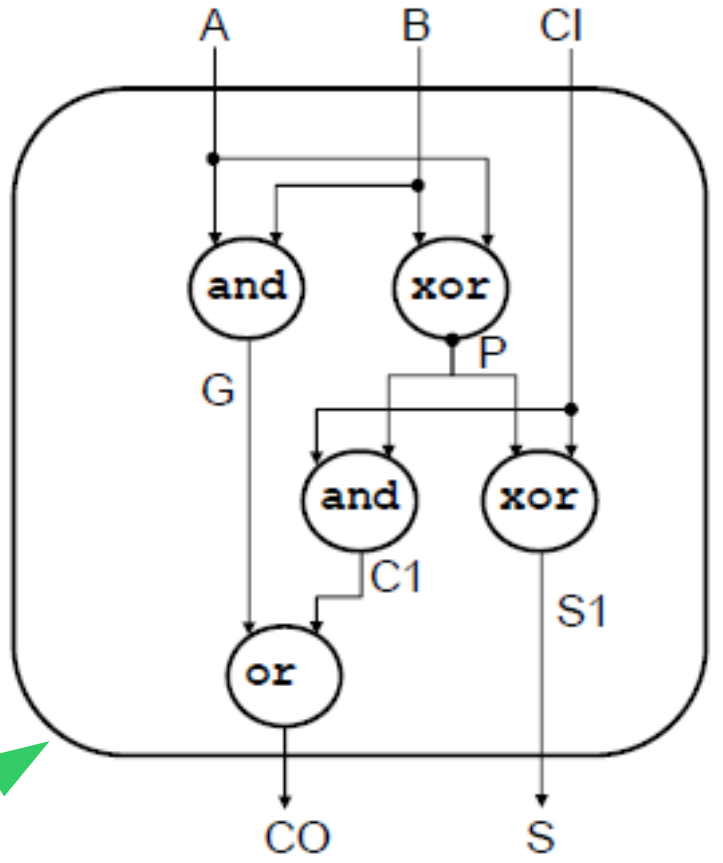
4 - VHDL synthesis

… not all VHDL constructs can be synthesized

```
-- File hello_world.vhd

entity hello_world is
end entity hello_world;

architecture arc of hello_world is
begin
assert false report "Hello world!" severity note;
end architecture arc;
```

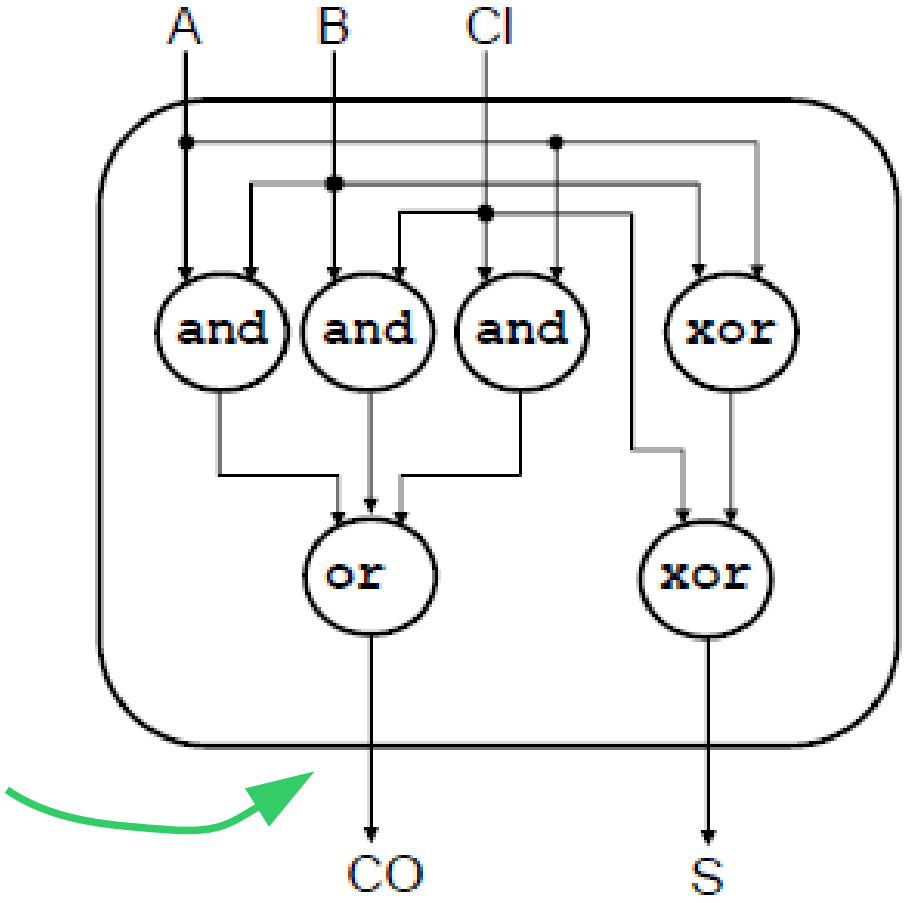
4 - VHDL synthesis: example

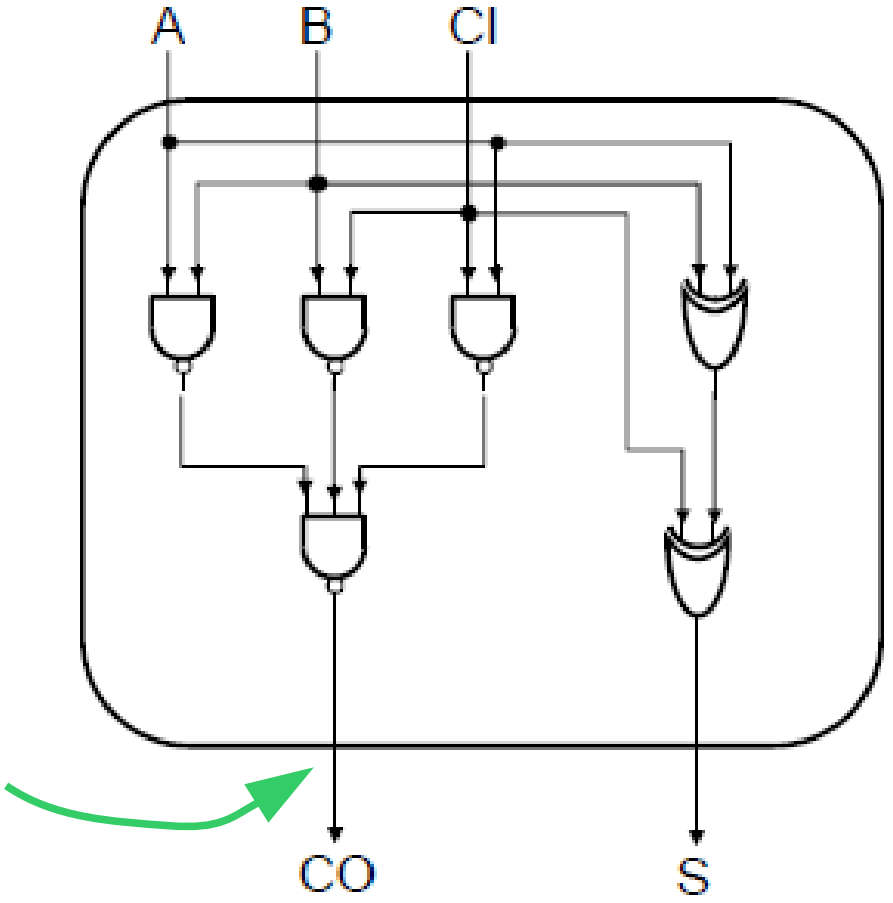
```
entity FULL_ADDER is
   port (A, B, CI: in bit;
         CO, S: out bit);
end FULL_ADDER;

architecture BHV of FULL_ADDER is
  signal P, S1, G, C1: bit;
begin
  P  <= A xor B;
  G  <= A and B;
  S1 <= P xor CI;
  C1 <= P and CI;
  S  <= S1;
  CO <= C1 or G;
end BHV;
```



Model translated into intermediate format
using elementary operations and generic logics
(ie not associated with cells of a technological library)

4 - VHDL synthesis: example





Optimized model:
the netword was modified in order to minimize the time delay of CO

Mapped model:
the model was mapped onto a library of standard or pre-built cells

4 - VHDL synthesis: example

VHDL description of the mapped model (netlist)

→ NAND2, NAND3, XOR2 describe the I/O terminals and the connectivity of the cells used

```
architecture STR of FULL_ADDER is
-- dichiarazione dei componenti
  component NAND2
    port(A, B: in bit;
         Z: out bit);
  end component;
  component NAND3
    port(A, B, C: in bit;
         Z: out bit);
  end component;
  component XOR2
    port(A, B: in bit;
         Z: out bit);
  end component;
-- connessioni interne
  signal X1, X2, X3, X4: bit;
```

```
begin
-- gate netlist (rete di celle)
  G1: NAND2 port map(A => A,
      B => B, Z => X1);
  G2: NAND2 port map(A => B,
      B => CI, Z => X2);
  G3: NAND2 port map(A => CI,
      B => A, Z => X3);
  G4: NAND3 port map(A => X1,
      B => X2, C => X3, Z => CO);
  G5: XOR2 port map(A => A,
      B => B, Z => X4);
  G6: XOR2 port map(A => CI,
      B => X4, Z => S);
end STR;
```

4 - VHDL synthesis: example

The netlist must be completed by associating
the components to their VHDL models, which are contained in the
technological library used for the synthesis (“binding“ step)

```
configuration C1 of FULL_ADDER is
begin
-- binding collettivi
  for all: NAND2 use entity
    TECHLIB.NA2(BHV);
  for all: NAND3 use entity
    TECHLIB.NA3(BHV);
-- binding individuali
  for G5: XOR2 use entity
    TECHLIB.XO2(BHV);
  for G6: XOR2 use entity
    TECHLIB.WXO2(BHV);
end C1;
```

```
-- libreria tecnologica
…
entity NA2 is
  generic(TP: time := 0.2ns);
  port(A,B: in bit; Z: out bit);
end NA2;
architecture BHV of NA2 is
begin
-- assegnazione con ritardo TP
  Z <= A nand B after TP;
end BHV;

entity NA3 is
  …
```

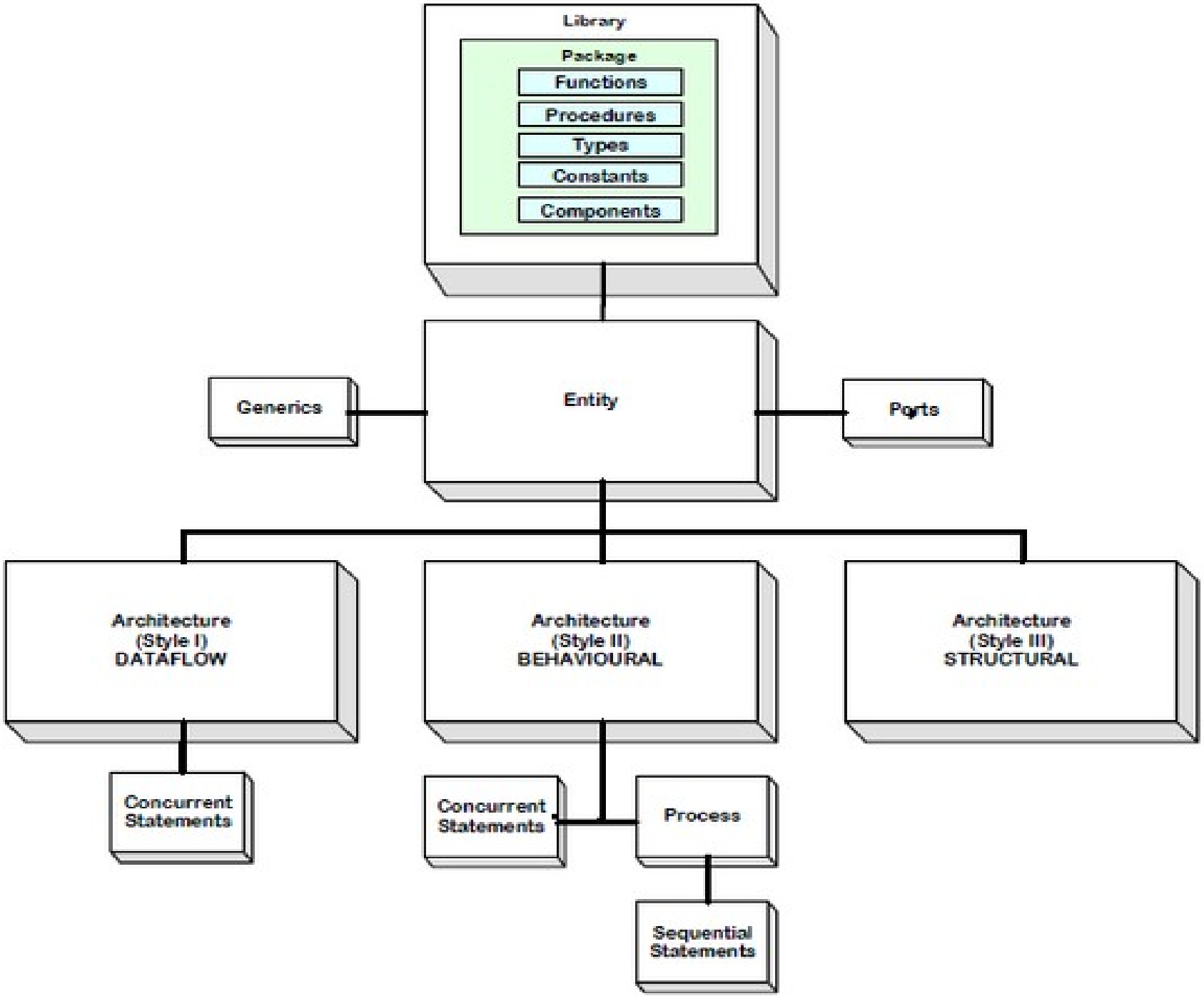
Note: there are more effective ways for binding
than using the **`configuration`**

II - VHDL language elements

1. Code anathomy – design units
2. Syntax
3. Data types
4. Object types and arrays
5. Operators and multi value logic
6. Packages and libraries → standard logic lib
7. Instructions: sequential and concurrent
   → signals vs variables
8. Procedures and Functions

… Examples

VHDL design anathomy



VHDL design units

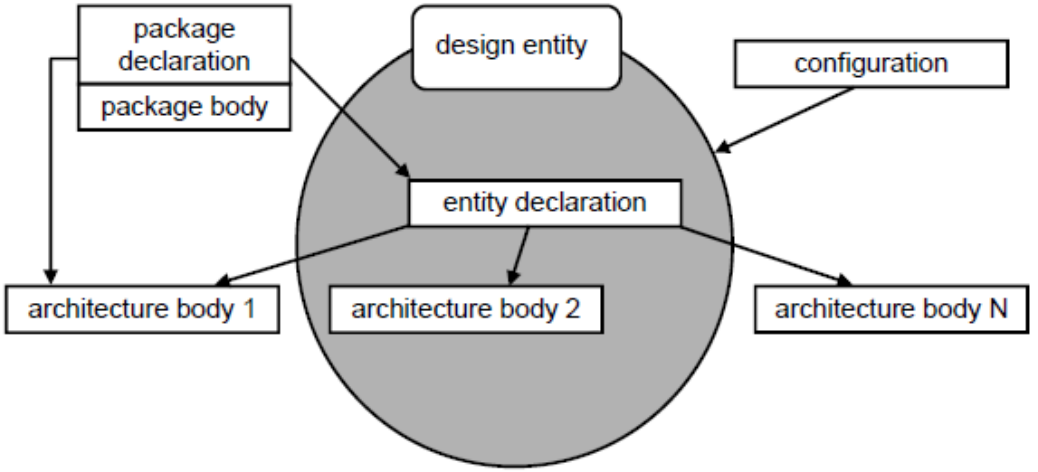
- **ENTITY**: describes the interface (I / O terminals) of each module of the model

- **ARCHITECTURE**: associated with an **ENTITY**; describes what the module does

- **PACKAGE**: contains the declarations of data types, functions and procedures

- **PACKAGE BODY**: contains the definitions of the functions e procedures declared in a **PACKAGE**

- **CONFIGURATION**: defines the link between the instances of components in a structural description, and their actual implementation



VHDL lexicon

• VHDL does not distinguish between upper and lower case
→ "**`architecture`**", "**`ARCHITECTURE`**" and "**`aRCHiTeCtUrE`**" are the same thing

• a **comment** begins with **`--`** (double dash) and extends to the end of the line
- Structure similar to "//" used in C ++
- DOES NOT EXIST a way to interpose comments within the lines of code
(in C we use "/*...*/")

• an **identifier** for signals, variables, functions, ..., starts with a alphabetic character and may contain letters, numbers and the character _

- Use simple names, which recall their meaning
- Try not to use similar names for objects not related to each other
- Do not use names of predefined identifiers, such as BIT or TIME
- Two consecutive underscels (__) are not allowed
- Names that do not comply with these rules are also allowed: it sufficies to enclose them with "\" (Eg. \ 0% $ # & __ \). They can start and contain any character. Distinction between uppercase and lower case... please DO NOT USE them

VHDL lexicon

• **numbers** may be integers or real (if they include a dot):
- `0 1 435_197 623E4` -- integers
- `0.0 1.0 251.436 12.3E-4` -- real numbers
- `2 # 1100_0100 # 16 # C4 # 4 # 301 # E1` -- integer 196
- `2 # 1.1111_111_111 # E + 11 16 # F.FF # E2` -- real number 4095.0

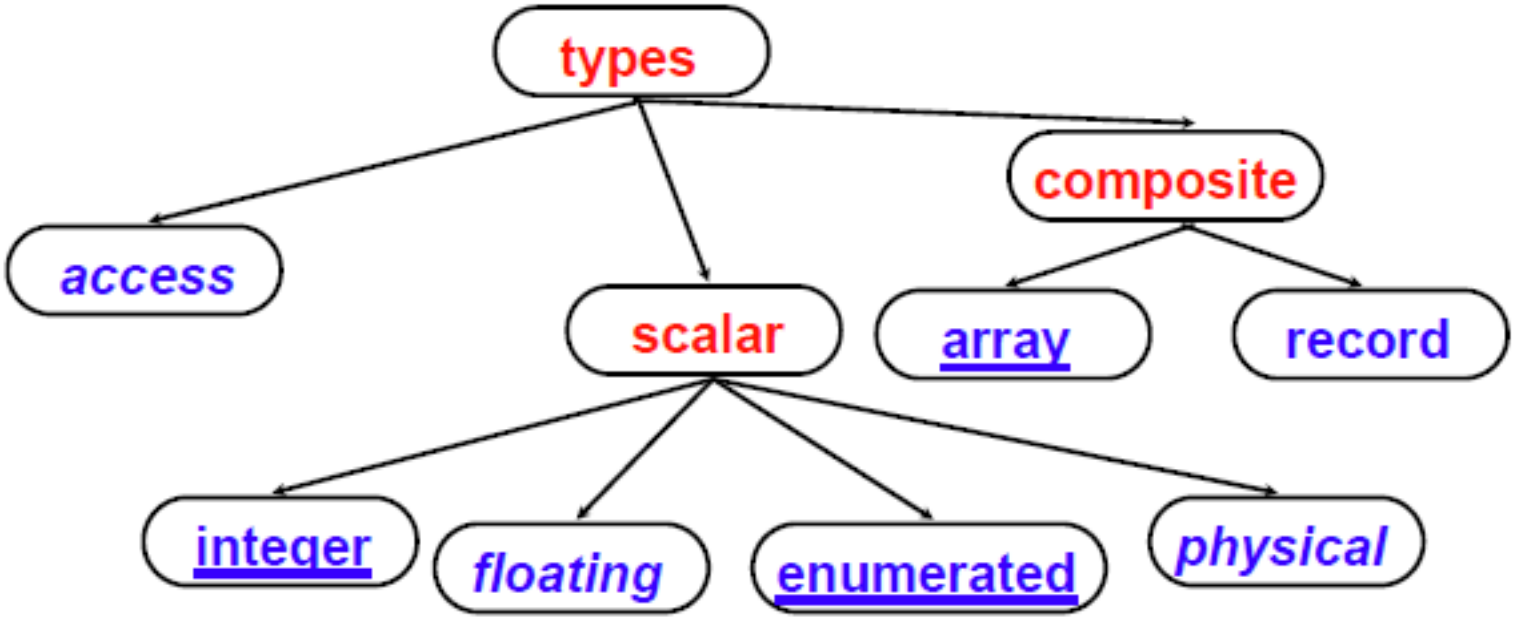
• **characters** must be enclosed in single quotes:
- `'A'  'a'  '*'  '1'  '0'`

• a **string** of characters is enclosed in double quotes:
- `"string"  ""`

• bit strings allow you to easily specify values for vectors (**arrays**):
- `B"101100110101"` -- binary
- `O"5465"` -- octal
- `X"B35"` -- hexadecimal

VHDL types of data

All signals and variables must be declared of a defined type (or sub-type)



`packages` are design units (like entities and architecture) used to group definitions of data types and standard or commonly used operators (functions and procedures)

VHDL standard types

- definiti nel package **STANDARD** contenuto nella libreria **STD**
- presenti implicitamente in ogni simulatore VHDL

```
-- enumerazioni
TYPE boolean IS (false, true);
TYPE bit IS ('0', '1');
TYPE character IS (NUL, SOH,
     ..., 'a', 'b', 'c', ...);

-- interi
TYPE integer IS RANGE -2147483647
  TO + 2147483647;
SUBTYPE positive IS integer RANGE
        1 to integer'HIGH;
SUBTYPE natural IS integer RANGE
        0 to integer'HIGH;
```

```
-- reali
TYPE real IS RANGE -1E38 to 1E38;
-- fisici
TYPE time IS RANGE 0 TO NNN
  UNITS
    fs;
    ps = 1000 fs;
    ...
    hr = 60 min;
  END UNITS;
-- array
TYPE string IS ARRAY (POSITIVE
 RANGE <>) OF characters;
TYPE bit_vector IS ARRAY (NATURAL
 RANGE <>) OF bit;
```

VHDL data types: definition examples

```
TYPE byte_in IS RANGE 0 TO 255
TYPE s_word IS RANGE -32768 TO 32767
TYPE resistence IS RANGE 0 TO 1E8
  UNITS
    ohms;
    kohms = 1000 ohms;
    Mohms = 1000 kohms;
  END UNITS;
TYPE probability IS RANGE 0.0 TO 1.0
TYPE bit IS ('0', '1');
TYPE word IS ARRAY (0 TO 31) OF bit;
TYPE vector IS ARRAY (integer RANGE
     <>) OF real;
SUBTYPE bcd IS integer RANGE 0 TO 9;
```

integer types

physical types

floating types

enumeration types

composite types

subtypes

VHDL object types, arrays

- the index variation range of an array can be ascending or descending:
  - `SIGNAL a: bit_vector (3 downto 0); --` discendente
  - `SIGNAL b: bit_vector (0 to 3); --` ascendente

- array assignments are based on location:
  - `b <= a; --` significa: `b(0) <= a(3);…; b(3) <= a(0);`
  - `b(1 downto 0) <= a(3 downto 2); --` assegnazione parziale, con inversione dell'ordine predefinito di `b`; il risultato è `b(1) <= a(3); b(0)<= a(2);`
  - `a <= ('1','0','1','0'); --` assegnazione tramite aggregato
  - `a <= B"1010" --` assegnazione tramite stringa di bit
  - `a <= X"A" --` come sopra, ma in formato esadecimale
  - `a <= (OTHERS => '1'); --` assegnazione dello stesso valore a tutti gli *elementi di* `a`
  - `b & a --`produce `(b(0),b(1),b(2),b(3),a(3),a(2),a(1),a(0))`

VHDL object types, arrays attributes

in the case of arrays, attributes provide additional information relative to the signal index

```
SIGNAL x: bit_vector(0 TO 3);
SIGNAL y: bit_vector(3 DOWNTO 0);
SIGNAL i: integer;
-- attributi di valore:
forniscono il valore minimo e
massimo dell'indice, e il numero
di elementi dell'array
i <= x'LOW -- i = 0
i <= y'LOW -- i = 0
i <= x'HIGH -- i = 3
i <= y'HIGH -- i = 3
i <= x'LENGTH -- i = 4
i <= y'LENGTH -- i = 4
```

```
-- attributi di posizione
i <= x'LEFT; -- i = 0
i <= y'LEFT; -- i = 3
i <= x'RIGHT; -- i = 3
i <= y'RIGHT; -- i =0
x'RANGE -- 0 TO 3
y'RANGE -- 3 DOWNTO 0
x'REVERSE_RANGE -- 3 DOWNTO 0
y'REVERSE_RANGE -- 0 TO 3
```

VHDL standard operators and expressions

Operatori presenti nel package **STANDARD**:

- booleani: **not, and nand, or, nor, xor, xnor**
- relazionali: **=, /=, <, <=, >, >=**
- shift: **sll, srl, sla, sra, rol, ror**
  - poco usati, hanno un comportamento non convenzionale
- aritmetici: **+, -, abs, +, -, *, /, mod, rem, ****
  - segno operazioni
- concatenazione: **&**
  - concatena due array, mettendo l'operando di destra in coda a quello di sinistra

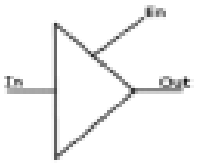
VHDL tri-state logic

- La tipologia `bit` va molto bene per descivere un hardware digitale ideale, in cui tutti i segnali hanno il valore logico '0' o '1'.
- Esistono situazioni nelle quali i due stati logici non sono più sufficienti.
- Un esempio di questa necessità si ha nella descrizione di una porta logica tri-state nella quale il segnale di uscita può assumere tre valori: '0', '1' e 'Z' (per indicare l'alta impedenza).
- Definiamo quindi un nuovo tipo:

```
type tri is ('0', '1', 'Z');
```

e quindi andare a definire porte e segnali:

```
signal a,b,c : tri;
```



VHDL tri-state logic

Oltre a definire la tipologia, però dobbiamo andare a definire tutte le varie funzioni in modo che abbiano ancora senso assegnazioni del tipo:

```
a <= '0' and '1';
b <= a or c;
```

Ad esempio la funzione **and** dovrà rispettare le regole dettate dalla tabella di verità qui a fianco

AND	0	1	Z
0	0	0	0
1	0	1	1
Z	0	1	1

Visto che questa esigenza è comune a tutti i progettisti la IEEE ha creato un pacchetto standard che permette di superare la limitazione della tipologia bit, fornendo anche tutte le relazioni fra i segnali.

VHDL Standard Logic Packet **STD_LOGIC_1164**

IEEE ha creato due package per tipi di dati (e relativi operatori) logici e aritmetici:

- **STD_LOGIC_1164**
  - sistema a 9 stati logici
  - **TYPE std_logic IS ('U','X','0','1','Z','W','L','H','-');**

    "U" Non definito (a cui non è mai stato dato un valore)

    "X" Sconosciuto (il cui valore non è determinabile)

    "0" 0 logico

    "1" 1 logico

    "Z" Alta impedenza

    "W" Segnale debole, senza un valore determinabile

    "L" Segnale debole che probabilmente andrà a 0

    "H" Segnale debole che probabilmente andrà a 1

    "–" Indifferente (*Don't care*)
  - **TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;**

VHDL Arithmetic Logic Packets

Oltre al pacchetto **STD_LOGIC_1164** con tutte le funzioni logiche si è visto che è utile andare a definire anche tutte le funzioni aritmetiche su questo tipo di dati; per questo la IEEE ha creato due diversi pacchetti (che possono essere utilizzati in alternativa).

- **NUMERIC_STD**
  - **TYPE unsigned IS ARRAY (NATURAL RANGE <>) OF std_logic;**
  - **TYPE signed IS ARRAY (NATURAL RANGE <>) OF std_logic;**
  - `unsigned` interpretato come rappresentazione binaria di un intero senza segno;
  - `signed` interpretato come rappresentazione in complemento a 2 di un intero
- **STD_LOGIC_ARITH**
  - alternativa (preferita) a **NUMERIC_STD**

VHDL Packet: **STD_LOGIC_(UN)SIGNED**

- la libreria IEEE contiene anche i due package **STD_LOGIC_UNSIGNED** e **STD_LOGIC_SIGNED**
- sono in alternativa l'uno all'altro e introducono operatori e funzioni che interpretano i dati di tipo **STD_LOGIC_VECTOR** come numeri binari rispettivamente senza e con segno:

```
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
  return STD_LOGIC_VECTOR;
function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return
  STD_LOGIC_VECTOR;
...
function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return
  INTEGER;
```

VHDL Libraries

- Una libreria è una collezione di *design units* il riferimento ad una libreria avviene attraverso **nomi logici**
  - l'associazione tra i nomi logici e le directory che contengono le librerie è contenuta in file di setup
  - il nome logico **WORK** è normalmente usato per la libreria che memorizza le *design units* create dall'utente
  - la libreria predefinita **STD** contiene i packages **STANDARD** e **TEXTIO**
  - la libreria **IEEE** contiene, tra gli altri, i package **STD_LOGIC_1164** e **STD_LOGIC_ARITH**
- per accedere a una libreria e usare le *design units* in essa contenute:

```
LIBRARY IEEE;  -- permette di accedere alla libreria attraverso il nome
               -- logico IEEE
USE IEEE.STD_LOGIC_1164.ALL;  -- carica tutto (ALL) il contenuto del
                              -- package STD_LOGIC_1164
USE IEEE.STD_LOGIC_ARITH.ALL;  -- carica tutto (ALL) il contenuto del
                               -- package STD_LOGIC_ARITH
```

VHDL Concurrent Instructions

Concurrent instructions
- used within the architecture to describe the signal behavior
- activated (or executed) only after an event on one of the signals to which they are sensitive

→ the order of execution does not depend from the order with which they appear in the model
→ **"parallel" signal processing**

• inside a process

```
--with sensitivity list
P1: PROCESS(a,b)
BEGIN
z <= a NOR b;
END PROCESS;
--with wait instruction
P2: PROCESS
BEGIN
z <= a NOR b;
WAIT ON a, b;
END PROCESS;
```

• conditional assignments

```
-- not in a process
ARCHITECTURE rtl OF ...
BEGIN
-- simplest
z <= a NOR b;
-- priority conditional assignment
a <= b WHEN sel=B"00" ELSE c WHEN
    sel=B"01" ELSE d;
a <= b WHEN sel=B"00" ELSE c WHEN
    sel=B"01" ELSE UNAFFECTED;
-- conditional without priority
WITH sel SELECT
    a <= b WHEN B"00",
    c WHEN B"01",
    d WHEN OTHERS;
```

VHDL Sequential Instructions

Sequential instructions
- they are user only inside a process,
- they are executed following the order with which they appear in the model
  → **"serial" signal processing**

• wait

```
WAIT ON x UNTIL y=0 FOR 12 NS;
```

• esecuzioni condizionate

```
-- choices with priority
IF sel = B"00" THEN
    z_out := a;
ELSIF sel = B"01" THEN
    z_out := b;
ELSE
    z_out := c;
ENDIF
-- choices without priority
CASE sel IS
    WHEN B"00" => z_out := a;
    WHEN B"01" => z_out := b;
    WHEN OTHERS => z out := c;
END CASE;
```

• loop

```
WHILE i < nmax LOOP
    z(i) := x(i);
    i := i + 1;
END LOOP;
FOR k IN 1 TO n LOOP
    table(k) := 0;
END LOOP
```

• NEXT, EXIT, NULL

```
NEXT; -- interrompe l'iterazione
-- corrente in un loop e passa
-- alla successiva
EXIT; -- interrompe il loop
NULL; -- non fa niente; utile
-- nei CASE e IF se si desidera
-- che a certe scelte non
--corrisponda alcuna azione
```

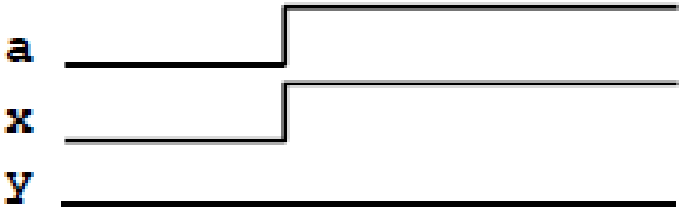
VHDL Variables vs Signals

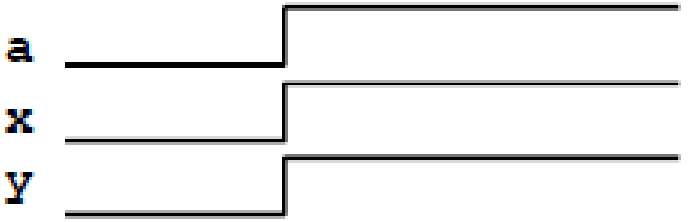
→ sequential/ concurrent

variables: sequential alternative for signal processing

```
ARCHITECTURE test1 OF mux IS
  SIGNAL a,x,y : BIT := '0';
BEGIN
    PROCESS (a)

      BEGIN
        x <= a;
        y <= x;
      END PROCESS;
END test1;
```

```
ARCHITECTURE test2 OF mux IS
    SIGNAL a,y : BIT := '0';
BEGIN
    PROCESS (a)
      VARIABLE x : BIT := '0';
    BEGIN
      x := a;
      y <= x;
    END PROCESS;
END test2;
```





The process is executed when **a** value changes (from 0 to 1, in the example above);

- In the first case, **x** takes the value of **a** at the end of the process, whereby **y** is assigned to the original value of **x** (i.e. 0) (concurrent/parallel execution)

- In the second case, **x** takes the value of **a** when the assignment **x:=a** is executed (sequential/serial execution) then **y** assumes the new value of **x**

VHDL Procedures and Functions

- dichiarazione

```
PROCEDURE p_and(CONSTANT a, b: IN bit; SIGNAL c: OUT bit);
```

classe direzione tipo

  - direzione IN, OUT, INOUT; classe CONSTANT, VARIABLE, SIGNAL
  - non ritorna parametri

```
FUNCTION f_and(CONSTANT a, b: IN bit) RETURN bit;
```

  - solo direzione IN (non può modificare i parametri)

- definizione (*body*)

```
PROCEDURE p_and(CONSTANT a, b: IN bit; SIGNAL c: OUT bit) IS
  BEGIN  c <= a AND b; END;
FUNCTION f_and(CONSTANT a, b: IN bit) RETURN bit IS
  BEGIN  RETURN a AND b; END
```

- chiamata

```
p_and(x1, x2, y); -- scrive direttamente il risultato in y
y <= f_and(x1, x2); -- ritorna il risultato che poi viene assegnato a y
   esternamente alla funzione
```

III – Concurrent vs Sequential

Sequential statements are executed in the order in which they appear within the process or subprogram
(as in programming languages)

Concurrent statements are executed in parallel
(the operations in real systems are executed concurrently)

Signals represent the interface between
- the Concurrent domain VHDL model and
- the Sequential domain within processes

Sequential statements

Sequential statements:

- Processes are composed of sequential statements …
… but process declarations are concurrent statements

- Sequential Signal assignment

- Variable assignment

- if statement, case statement, loop statements
(loop, while loop, for loop, next,exit)

- sequential assert statement

- procedure call statement

- return statement from a procedure or function

Processes


```
[name:] process [(sensitivity_list)]
    [type_declarations]
    [constant_declarations]
    [variable_declarations]
    [subprogram_declarations]
begin
    sequential_statements
end process [name];
```

Note:
no Signal declaration
allowed in a process

When the sensitivity list is missing, the process will be run continuously

In this case, the process must contain a `wait` statement to suspend the process and to activate it when an event occurs or a condition becomes true

When the sensitivity list is present, the process cannot contain `wait` statements

Processes


```
proc1: process (a, b, c)
begin
   x <= a and b and c;
end process proc1;

proc2: process (a, b)
begin
   x <= a and b and c;
end process proc2;
```

```
proc3: process
begin
   x <= a and b and c;
   wait on a, b, c;
end process proc3;
```

Wait Statement Forms

There are three forms of the wait statement:

```
wait on sensitivity_list;
wait until conditional_expresion;
wait for time_expression;
```

When the sensitivity list is missing, the process will be run continuously

In this case, the process must contain a `wait` statement to suspend the process and to activate it when an event occurs or a condition becomes true

When the sensitivity list is present, the process cannot contain `wait` statements

Processes

`Wait` statement examples

```
wait until signal = value;
wait until signal'event and signal = value;
wait until not signal'stable and signal = value;
```

Synchronous operations

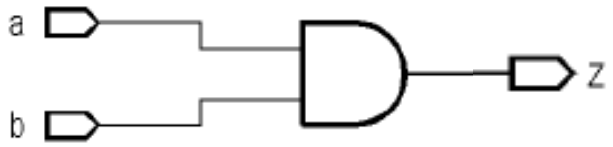
```
wait until clk = '1';
wait until clk'event and clk = '1';
wait until not clk'stable and clk = '1';
```

Processes
Combinatorial vs Sequential

Both combinational and sequential processes are interpreted in the same way, the only difference being that
for sequential processes the output signals are stored into registers

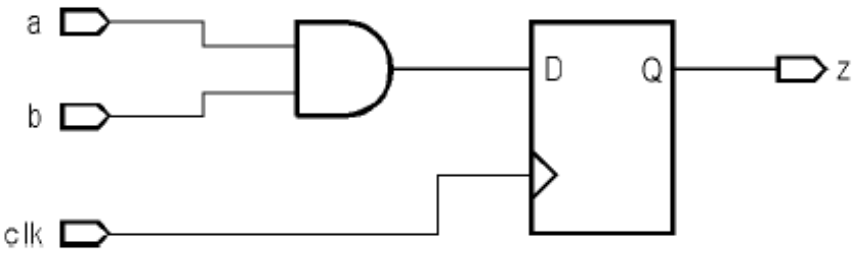
combinational

```
proc5: process
begin
   wait on a, b;
   z <= a and b;
end process proc5;
```



sequential

```
proc6: process
begin
   wait until clk = '1';
   z <= a and b;
end process proc6;
```



Seq. Signal Assigment

Signals represent the interface between
- the Concurrent domain VHDL model and
- the Sequential domain within processes

Signal assignment may be performed using
a sequential statement or a concurrent statement:
→ The sequential statement may only appear inside a process
→ The concurrent statement may only appear outside processes

The sequential signal assignment has a single form
(the simple one i.e. unconditional assignment)

The concurrent signal assignment has, in addition to its simple form two other forms:
- the conditional assignment and
- the selective assignment

Seq. Signal Assigment


```
signal <= expression
```

As a result of executing this statement in a process, the expression on the right-hand side of the assignment symbol is evaluated and an event is scheduled to change the value of the signal (see simulator example)

The simulator will only change the value of a signal when the process suspends. Therefore, in a process the signals will be updated only after executing all the statements of the process or when a wait statement is encountered

Consequence I: only **last assignment** to same signal will be executed

```
proc7: process (a)
begin
   z <= '0';
   z <= a;
end process proc7;
```

```
proc8: process (a)
begin
   z <= a;
end process proc8;
```

Seq. Signal Assigment

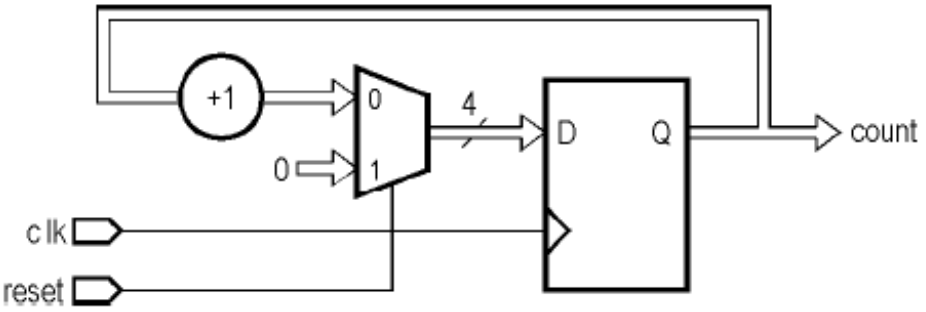

```
signal <= expression
```

As a result of executing this statement in a process, the expression on the right-hand side of the assignment symbol is evaluated and an event is scheduled to change the value of the signal (see simulator example)

The simulator will only change the value of a signal when the process suspends. Therefore, in a process the signals will be updated only after executing all the statements of the process or when a wait statement is encountered

Consequence II: **feedback** !!!

```
library ieee;
use ieee.numeric_bit.all;

signal count: unsigned (3 downto 0);
process
begin
   wait until clk = '1';
   if reset = '1' then
      count <= "0000";
   else
      count <= count + "0001";
   end if;
end process;
```



Seq. Signal Assigment

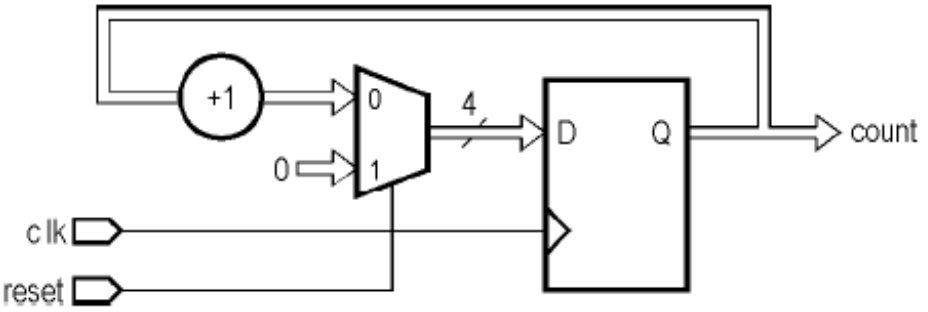
Another consequence of the way in which signal assignments are executed is that any reading of a signal that is also assigned to in the same process will return the value assigned to the signal in the previous execution of the process (reading a signal and assigning a value to it in the same process is equivalent to a *feedback)*

In a combinational process, the previous value is an output of the combinational logic and so the **feedback is asynchronous**

In a sequential process, the previous value is the value stored in a latch or register, so the **feedback is synchronous**.

Consequence II: feedback !!!

```
library ieee;
use ieee.numeric_bit.all;

signal count: unsigned (3 downto 0);
process
begin
   wait until clk = '1';
   if reset = '1' then
      count <= "0000";
   else
      count <= count + "0001";
   end if;
end process;
```



Variable Assigment

Signal drawbacks:
- can only hold the last value assigned to them → they cannot be used to store intermediary results within a process
- new values are not assigned to signals when the assignment statement executes, but only after the process execution suspends

Variables:
- can be declared inside (and only inside) processes: local to processes (only in sequential domani, not in architectures)
- can store intermediate results → many assignment are possible
- are updated immediately (immediate execution of assignment)

Assigned

```
variable := expression;
```

Declared

```
variable a, b, c: bit;
variable x, y: integer;
variable index range 1 to 10 := 1;
variable cycle_t: time range 10 ns to 50 ns := 10 ns;
variable mem: bit_vector (0 to 15);
```

Variable Assigment


```
entity add_1 is
   port (a, b, cin: in bit;
         s, cout: out bit);
end add_1;

architecture functional of add_1 is
begin
   process (a, b, cin)
      variable s1, s2, c1, c2: bit;
   begin
      s1 := a xor b;
      c1 := a and b;
      s2 := s1 xor cin;
      c2 := s1 and cin;
      s <= s2;
      cout <= c1 or c2;
   end process;
end functional;
```

full-adder made of two half-adders
(not the optimal way...)

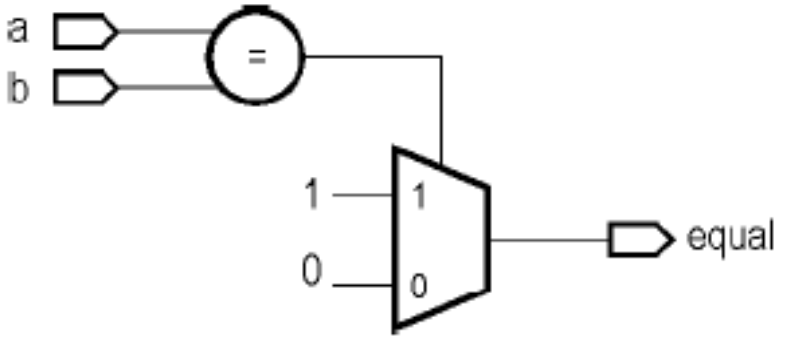
If statement


```
if condition then statement_sequence
    [elsif condition then statement_sequence...]
    [else statement_sequence]
end if;
```

Conditions are Boolean expressions (`TRUE` / `FALSE`)

If statemets synthesised as Multiplexer

```
library ieee;
use ieee.numeric_bit.all;
entity comp is
   port (a, b: in unsigned (7 downto 0);
         equal: out bit);
end comp;

architecture functional of comp is
begin
   process (a, b)
   begin
      if a = b then
         equal <= '1';
      else
         equal <= '0';
      end if;
   end process;
end functional;
```

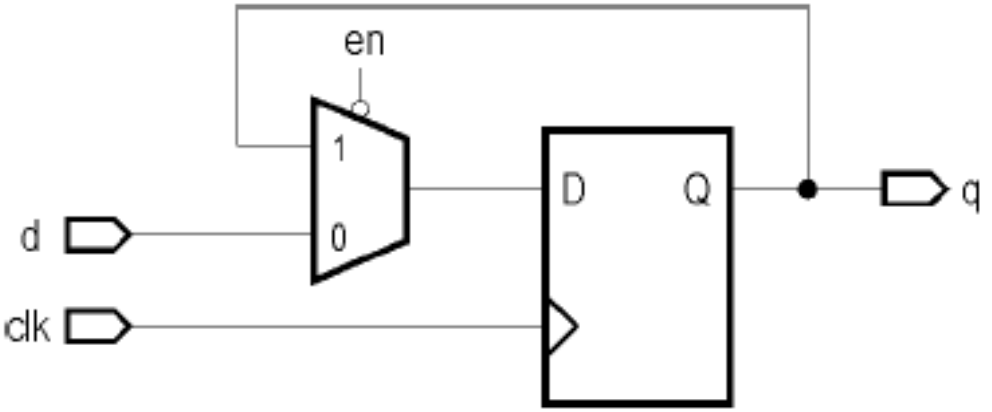


If statement

Warning:
describe every possible condition !

Incomplete If statemets result in **feedback**

```
process
begin
    wait until clk = '1';
    if en = '1' then
        q <= d;
    end if;
end process;
```



implicit : **else**

```
        q <= q;
    end if;
```

feedback !

If statement

Variables can be used in IF statements, allowing uses of feedback

```
process
   variable count: unsigned (7 downto 0);
begin
   wait until clk = '1';
   if reset = '1' then
      count := "00000000";
   else
      count := count + 1;
   end if;
   result <= count;
end process;
```

Case statement

Case statement is like If statement but:
- allows selection based on expression (not just boolean)
- all choices depend on the same input and are mutually exclusive

```
case expression is
    when options_1 =>
        statement_sequence
    ...
    when options_n =>
        statement_sequence
    [when others =>
        statement_sequence]
end case;
```

Case statemets synthesised as Multiplexer

Loop statements

Loop statements allow repeated execution of a statement sequence
→ eg processing each element of an array

```
[label:] for counter in range loop
   statement_sequence
end loop [label];
```

```
for i in 1 to 10 loop
   i_square (i) <= i * i;
end loop;
```

```
for i in integer range 1 to 10 loop
```

Loop statements


```
entity match_bits is
   port (a, b: in bit_vector (7 downto 0);
         matches: out bit_vector (7 downto 0));
end match_bits;

architecture functional of match_bits is
begin
   process (a, b)
   begin
      for i in 7 downto 0 loop
         matches (i) <= not (a(i) xor b(i));
      end loop;
   end process;
end functional;
```

This loop is sinthesised as a set of 1-bit comparators to compare bits of the same order of vectors a and b
→ equivalent to:

(order of the executions not relevant …
… concurrent execution)

```
process (a, b)
begin
   matches (7) <= not (a(7) xor b(7));
   matches (6) <= not (a(6) xor b(6));
   matches (5) <= not (a(5) xor b(5));
   matches (4) <= not (a(4) xor b(4));
   matches (3) <= not (a(3) xor b(3));
   matches (2) <= not (a(2) xor b(2));
   matches (1) <= not (a(1) xor b(1));
   matches (0) <= not (a(0) xor b(0));
end process;
```

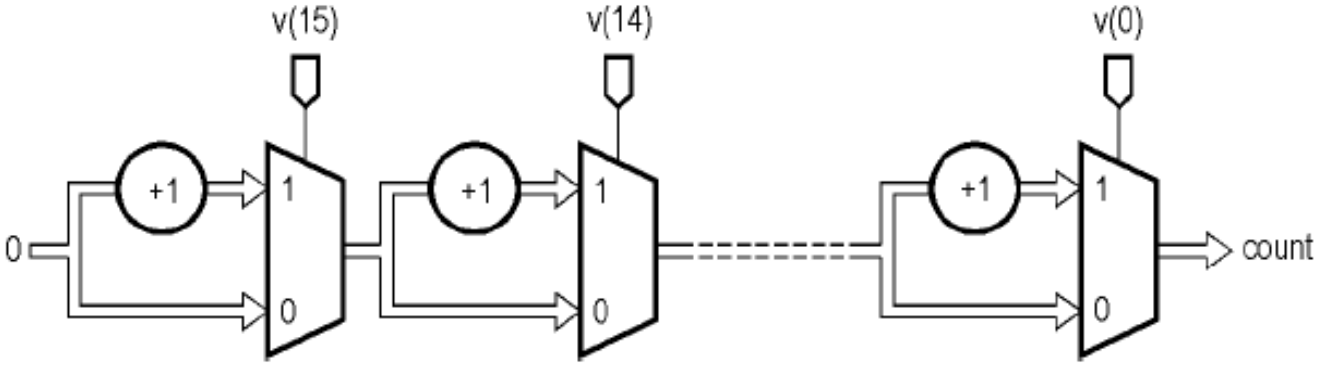
Loop statements


```
library ieee;
use ieee.numeric_bit.all;
entity count_ones is
   port (v: in bit_vector (15 downto 0);
         count: out signed (3 downto 0));
end count_ones;

architecture functional of count_ones is
begin
   process (v)
      variable result: signed (3 downto 0);
   begin
      result := (others => '0');
      for i in 15 downto 0 loop
         if v(i) = '1' then
            result := result + 1;
         end if;
      end loop;
      count <= result;
   end process;
end functional;
```

This is a combinational circuit which counts the number of bits in vector v that are set to '1'

The result is accumulated during the execution of the process in a variable called result and then assigned to the output signal count at the end of the process

The loop body – if statement containing a variable assignment – represents a block containing a multiplexer and an adder, which will be generated by synthesis for each iteration of the loop. The output of a block becomes the input result of the next block...



If statement

Variables can be used in IF statements

If a variable is assigned to only in some branches of the if statement
→ previous value is preserved by feedback !!! Unlike the case when a signal is used, the reading and writing of a variable in the same process will result in feedback only if the read occurs before the write
In this case, the value read is the previous value of the variable

→ this may be used to create registers or counters using variables

Remember that a sequential process is interpreted by synthesis by placing a flip-flop or register on every signal assigned to in the process. This means that normally variables are not written to flip-flops or registers **unless** there is feedback of a previous variable value, then this feedback is implemented via a flip-flop or register to make the process synchronous

Concurrent statements

Concurrent statements:

- Processe declarations are concurrent statements
- Concurrent Signal assignment
- Block statements
- concurrent assert statement
- concurrent procedure call statement
- component instantiation statement
- generate statement

Architecture structure and execution

An architecture definition has two parts: declarative and statement
- declarative part: objects internal to the architecture may be defined
- statement part: contains concurrent statements which define the processes or interconnected blocks that describe the operation
or the global structure of the system

All **processes** in an architecture are executed **concurrently** with each other, but the **statements** within a process are executed **sequentially**

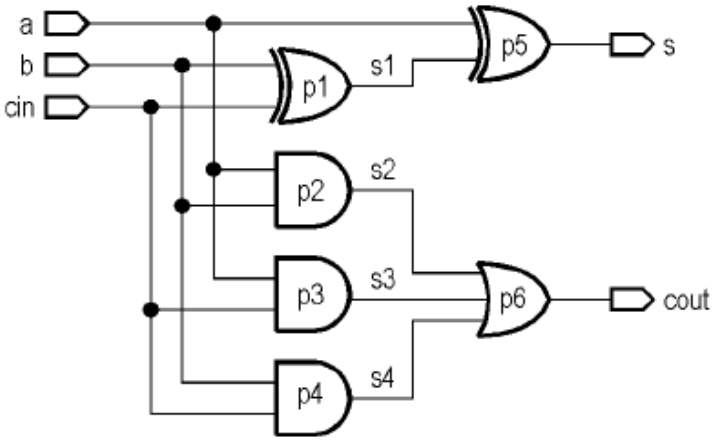
A suspended process is activated again when one of the signals in its sensitivity list changes its value. When there are multiple processes in an architecture, if a signal changes its value then all processes that contain this signal in their sensitivity lists are activated. The statements within the activated processes are executed sequentially, but independently from the statements in other processes

Communication among processes is achieved using
concurrent signal assignment statements
(can be used eg for synchronization). Note: variables cannot be used

Architecture structure and execution

```
entity add_1 is
   port (a, b, cin: in bit;
         s, cout: out bit);
end add_1;

architecture processes of add_1 is
   signal s1, s2, s3, s4: bit;
begin

   p1: process (b, cin)
   begin
      s1 <= b xor cin;
   end process p1;

   p2: process (a, b)
   begin
      s2 <= a and b;
   end process p2;

   p3: process (a, cin)
   begin
      s3 <= a and cin;
   end process p3;

   p4: process (b, cin)
   begin
      s4 <= b and cin;
   end process p4;

   p5: process (a, s1)
   begin
      s <= a xor s1;
   end process p5;

   p6: process (s2, s3, s4)
   begin
      cout <= s2 or s3 or s4;
   end process p6;

end processes;
```

full-adder



Process (summary)

- It is executed in parallel with other processes;
- It cannot contain concurrent statements;
- It defines a region of the architecture where statements are executed sequentially;
- It must contain an explicit sensitivity list or a wait statement;
- It allows functional descriptions, similar to the programming languages;
- It allows access to signals defined in the architecture in which the process appears and to those defined in the entity to which the architecture is associated.

Concurrent signal assign.
→ simple version

This statement is the concurrent version of the sequential signal assignment statement and has the same form with this

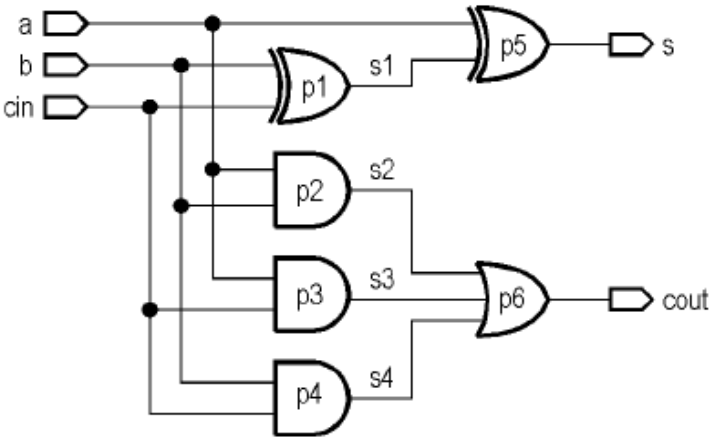
As the sequential version, the concurrent assignment
defines a new driver for the assigned signal

A concurrent assignment statement appears outside a process, within an architecture. A concurrent assignment statement represents a **simplified form of writing a process**
→ it is equivalent to a process that contains a single sequential assignment statement

Concurrent signal assign.
→ simple version

```
entity add_1 is
   port (a, b, cin: in bit;
         s, cout: out bit);
end add_1;

architecture concurrent of add_1 is
   signal s1, s2, s3, s4: bit;
begin
      s1 <= b xor cin;
      s2 <= a and b;
      s3 <= a and cin;
      s4 <= b and cin;
      s  <= a xor s1;
      cout <= s2 or s3 or s4;
end concurrent;
```

!!! full-adder



Nota: the order in which the statements are written is not relevant

Concurrent signal assign. → conditional version

The conditional assignment statement is functionally equivalent to the `If` conditional statement:

```
signal <= [expression when condition else ...]
          expression;
```

The differences are:

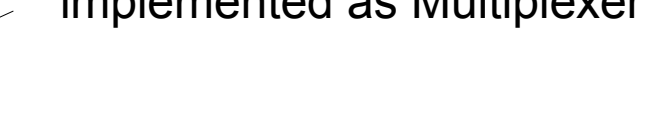
• the conditional assignment statement is a concurrent statement and therefore it can be used in an architecture,
while the If statement is a sequential statement and can be used only inside a process

• the conditional assignment statement can only be used to assign values to signals, while the If statement can be used to execute any sequential statement

Note: the hardware behind is different:
- all possible drivers for the same signal are prepared
- though they are evaluated with some order

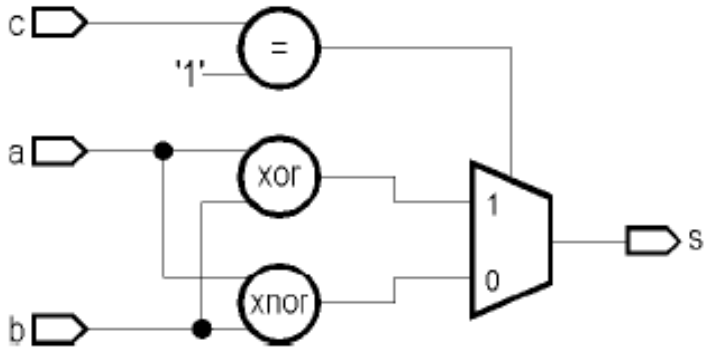
Concurrent signal assign. → conditional version

```
entity xor2 is
   port (a, b: in bit;
         x: out bit);
end xor2;

architecture arch1_xor2 of xor2 is
begin
   x <= '0' when a = b else
        '1';
end arch1_xor2;

architecture arch2_xor2 of xor2 is
begin
   process (a, b)
   begin
      if a = b then x <= '0';
               else x <= '1';
      end if;
   end process;
end arch2_xor2;
```

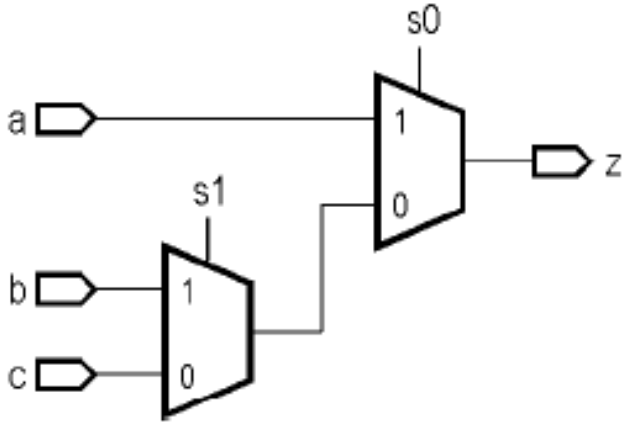


Concurrent signal assign.
→ conditional version

```
s <= a xor b when c = '1' else
     not (a xor b);
```




```
z <= a when s0 = '1' else
     b when s1 = '1' else
     c;
```



Concurrent signal assign. → selected version

```
with selection_expression select
   signal <= expression_1 when options_1,
             ...
             expression_n when options_n,
             [expression when others];
```

Like the conditional signal assignment statement, the selected signal assignment statement allows to select a source expressions based on a condition. The difference is that the selected signal assignment statement uses a single condition to select between several options

This statement is functionally equivalent to the case sequential statement

Block Statement

A `block` statement defines a group of concurrent statements.

This statement is useful to hierarchically organize the concurrent statements or to partition a list of structural interconnections in order to improve readability of the description

```
label: block [(guard_expression)]
   [declarations]
begin
   concurrent_statements
end block [label];
```

The mandatory label identifies the block
In the declaration part, local objects of the block may be declared
The possible declarations are those that may appear in the declarative part of an architecture

- `Use` clauses;
- Port and generic declarations, as well as port map and generic map declarations;
- Subprogram declarations and bodies;
- Type and subtype declarations;
- Constant, variable, and signal declarations;
- Component declarations;
- File, attribute, and configuration declarations.

The order of concurrent statemets is not relevant …

Reminder

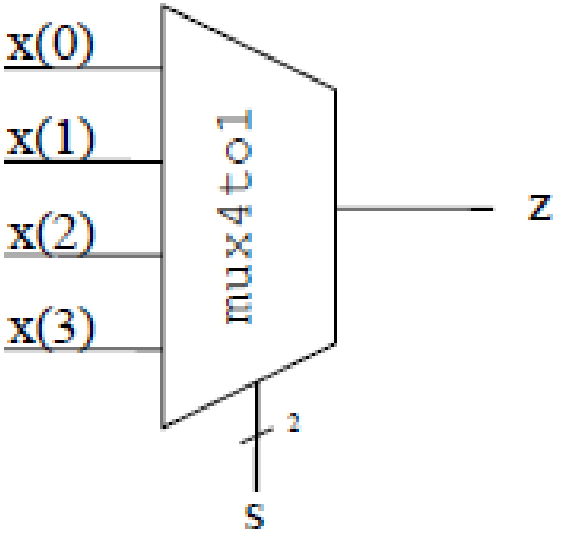
- Asynchronous sequential circuits: Latches
- Synchronous circuits: flip flops, counters, registers
- Systhesis infrence rules

IV - VHDL examples

- Multiplexers
- Adders
- Memory elements: latch and flip-flop
- Registers

VHDL examples: MULTIPLEXER 4→1

esempio con istruzione concorrente WITH SELECT




```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY mux4to1 IS
  PORT(x: IN std_logic_vector(3
          DOWNTO 0);
       s: IN unsigned(1 DOWNTO 0);
       z: OUT std_logic);
END mux4to1;

ARCHITECTURE bhv1 OF mux4to1 IS
BEGIN
  WITH s SELECT
    z <= x(0) WHEN "00",
    z <= x(1) WHEN "01",
    z <= x(2) WHEN "10",
    z <= x(3) WHEN OTHERS;
END bhv1;
```

VHDL examples: MULTIPLEXER 4→1

esempio con istruzione sequenziale `CASE` all'interno di un processo

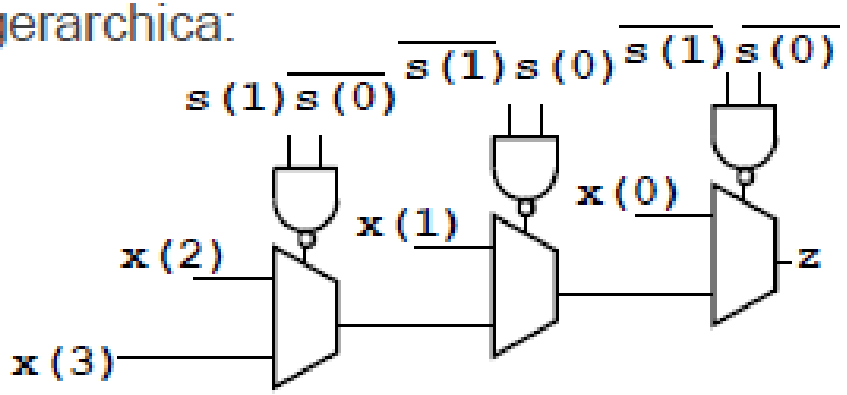
```
ARCHITECTURE bhv2 OF mux4to1 IS
BEGIN
-- il processo è concorrente: viene
-- eseguito solo in seguito a eventi
-- su s o x
  PROCESS(s,x) BEGIN
-- le istruzioni contenute nel
-- processo vengono eseguite in
-- sequenza; in questo esempio c'è
-- una sola istruzione (CASE ...
-- END CASE;)
    CASE s IS
      WHEN "00" => z <= x(0);
      WHEN "01" => z <= x(1);
      WHEN "10" => z <= x(2);
      WHEN OTHERS => z <= x(3);
    END CASE;
  END PROCESS;
END bhv2;
```

VHDL examples: MULTIPLEXER 4→1

esempio con istruzione sequenziale **IF ...THEN ... ELSE**

```
ARCHITECTURE bhv4 OF mux4to1 IS
BEGIN
  PROCESS(s,x)
  BEGIN
    IF s = "00" THEN
      z <= x(0);
    ELSIF s = "01" THEN
      z <= x(1);
    ELSIF s = "10" THEN
      z <= x(2);
    ELSE
      z <= x(3);
    END IF;
  END PROCESS;
END bhv4;
```

è un uso improprio di **IF ...THEN ... ELSE** perché questa istruzione, in fase di sintesi, ha una interpretazione gerarchica:




```
-- soluzione alternativa con
-- conversione di s
ARCHITECTURE bhv3 OF mux4to1 IS
BEGIN
  z <= x(conv_integer(s));
END bhv3;
```

VHDL examples: ADDER (behav. architecture)

esempio di architettura di tipo comportamentale (*behavioral*): descrive solo la sequenza di operazioni necessarie a ottenere le uscite dagli ingressi

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

ENTITY adder IS
  GENERIC(N: integer := 16);
  PORT(ci : IN  std_logic;
       x,y: IN  signed(N-1 DOWNTO 0);
       s  : OUT signed(N-1 DOWNTO 0);
       co : OUT std_logic);
END adder;

ARCHITECTURE bhv OF adder IS
  SIGNAL sum: signed(N DOWNTO 0);
BEGIN
  sum <= ('0' & x) + y + ci;
  s   <= sum(N-1 DOWNTO 0);
  co  <= sum(N);
END bhv;
```

concatenazione: porta il numero di bit a 17, come sum

VHDL examples: ADDER (ripple-carry)

il processo con il ciclo **FOR** crea una architettura di tipo ripple-carry

```
ARCHITECTURE rtl OF adder IS
BEGIN
  PROCESS (x,y,ci)
-- l'array carry è una variable quindi il suo valore viene aggiornato nel
-- momento stesso in cui viene eseguita l'assegnazione carry(i) := ...
    VARIABLE carry: std_logic_vector(N-1 DOWNTO 0);
  BEGIN
    FOR i IN x'REVERSE_RANGE LOOP
      IF i=0 THEN
        carry(i):= (x(i) AND y(i)) OR (x(i) AND ci) OR (y(i) AND ci);
        s(i) <= x(i) XOR y(i) XOR ci;
      ELSE
        carry(i):= (x(i) AND y(i)) OR (x(i) AND carry(i-1)) OR
                   (y(i) AND carry(i-1));
        s(i) <= x(i) XOR y(i) XOR carry(i-1);
      END IF;
    END LOOP;
    co  <= carry(N-1);
  END PROCESS;
END rtl;
```

VHDL examples: ADDER (ripple-carry)

generate è un'istruzione concorrente usata per creare repliche di istruzioni concorrenti (quindi anche di processi); può creare un numero fissato di repliche (**FOR** … **GENERATE**) oppure può sottostare ad una condizione (**IF** … **GENERATE**)

```
ARCHITECTURE rtl2 OF adder IS
  SIGNAL carry: std_logic_vector(N-1 downto 0);
BEGIN
  G1: FOR i IN x'RANGE GENERATE
    G2: IF i = 0 GENERATE
      carry(i) <= (x(i) AND y(i)) OR (x(i) AND ci) OR (y(i) AND ci);
      s(i)     <= x(i) XOR y(i) XOR ci;
    END GENERATE;
    G3: IF i > 0 GENERATE
      carry(i) <= (x(i) AND y(i)) OR (x(i) AND carry(i-1)) OR
                  (y(i) AND carry(i-1));
      s(i)     <= x(i) XOR y(i) XOR carry(i-1);
    END GENERATE;
  END GENERATE;
  co  <= carry(N-1);
END rtl2;
```

VHDL examples: ADDER (ripple-carry)

generate viene anche usato all'interno di descrizioni strutturali per creare una sequenza di istanze dello stesso componente; supponiamo ad esempio di voler realizzare un MUX 16-1 usando 5 MUX 4-1 come quello visto prima:

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

ENTITY mux16to1 IS
  PORT(x: IN std_logic_vector(15
          DOWNTO 0);
       s: IN unsigned(3
          DOWNTO 0);
       z: OUT std_logic);
END mux16to1;

ARCHITECTURE str OF mux16to1 IS
  SIGNAL m: std_logic_vector(3
                             DOWNTO 0);
```

```
-- dichiaro il componente che uso
-- nella netlist
  COMPONENT mux4to1 PORT(
  x: IN std_logic_vector(3 DOWNTO 0);
  s: IN unsigned(1 DOWNTO 0);
  z: OUT std_logic);
BEGIN
  G1: FOR i IN 0 TO 3 GENERATE
    level1: mux4to1 PORT MAP(
    x(4*i+3),x(4*i+2),x(4*i+1),
    x(4*i),s(1 DOWNTO 0),m(i));
  END GENERATE;
  level2: mux4to1 PORT MAP(
    m(3),m(2),m(1),m(0),
    s(3 DOWNTO 2),z);
END str;
```

VHDL examples: latch and flip-flop

ci sono diversi modi per descrivere il funzionamento di un latch e di un flip-flop in VHDL; tuttavia, solo rispettando certe regole, il codice viene riconosciuto in modo corretto in fase di sintesi

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

ENTITY latch IS
  PORT(d,clk: IN  std_logic;
       q    : OUT std_logic);
END latch;

ARCHITECTURE bhv OF latch IS
BEGIN
  PROCESS(d,clk)
  BEGIN
    IF clk = '1' THEN
      q <= d;
    END IF;
  END PROCESS;
END bhv;
```

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

ENTITY dff IS
  PORT(d,clk: IN  std_logic;
       q    : OUT std_logic);
END dff;

ARCHITECTURE bhv OF dff IS
BEGIN
  PROCESS(clk)
  BEGIN
   IF clk'EVENT AND clk='1' THEN
     q <= d;
   END IF;
  END PROCESS;
END bhv;
```

VHDL examples: flip-flop w/ synchronous and asynchronous reset

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

ENTITY dffr IS
  PORT(d,clk,res: IN  std_logic;
       q: OUT std_logic);
END dffr;
ARCHITECTURE bhv OF dffr IS
BEGIN
  PROCESS(res,clk)
  BEGIN
    IF res = '1' THEN
      q <= '0';
    ELSIF clk'EVENT AND clk='1' THEN
      q <= d;
    END IF;
  END PROCESS;
END bhv;
```

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

ENTITY dffr IS
  PORT(d,clk,res: IN  std_logic;
       q: OUT std_logic);
END dffr;
ARCHITECTURE bhv2 OF dffr IS
BEGIN
  PROCESS BEGIN
    WAIT UNTIL clk'EVENT AND clk='1';
    IF res = '1' THEN
      q <= '0';
    ELSE
      q <= d;
    END IF;
  END PROCESS;
END bhv2;
```

VHDL examples: n-bit register

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

ENTITY regn IS
  GENERIC(n: INTEGER := 16);
  PORT(d: IN std_logic_vector(n-1 DOWNTO 0);
       res,clk: IN std_logic;
       q: OUT std_logic_ vector(n-1 DOWNTO 0));
END regn;
ARCHITECTURE bhv OF regn IS
BEGIN
  PROCESS(res,clk)
  BEGIN
    IF res = '1' THEN
      q <= (OTHERS =>'0');
    ELSEIF clk'EVENT AND clk='1'THEN
      q <= d;
    END IF;
  END PROCESS;
END bhv;
```

valore di default; può essere cambiato con **generic map** quando il registro viene istanziato

non sapendo quanti bit ha **q**, usiamo **others** per assegnare a tutti il valore '0'

VHDL examples: shift register

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY shift IS
  GENERIC(n: INTEGER := 16);
  PORT(r: IN std_logic_vector(n-1 DOWNTO 0);
       load,clk,w: IN std_logic;
       q: OUT std_logic_ vector(n-1 DOWNTO 0));
END shift;
ARCHITECTURE bhv OF shift IS
SIGNAL qi: std_logic_ vector(n-1 DOWNTO 0);
BEGIN
  PROCESS(clk)
  BEGIN
    IF clk'EVENT AND clk='1'THEN
      IF load = '1' THEN
        qi <= r;
      ELSE
          qi(qi'RIGHT) <= w;
        FOR i IN 1 TO q'LEFT LOOP
          qi(i) <= qi(i-1);
        END LOOP;
      END IF;
    END IF;
  END PROCESS;
  q <= qi;
END bhv;
```

qi(i) viene aggiornato al termine del processo, quindi per tutto il ciclo FOR i segnali qi(i) conservano il valore iniziale

V - Finite State Machines

A system that
→ jumps from one state to the next
→ within a pool of finite states
→ upon clock edges and input transitions

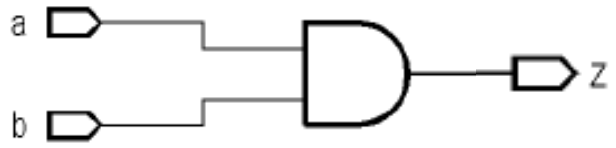
Examples: traffic light, digital watch, CPU, ...

Reminder: Combinatorial vs Sequential

Both combinational and sequential processes are interpreted in the same way, the only difference being that
for sequential processes the output signals are stored into registers

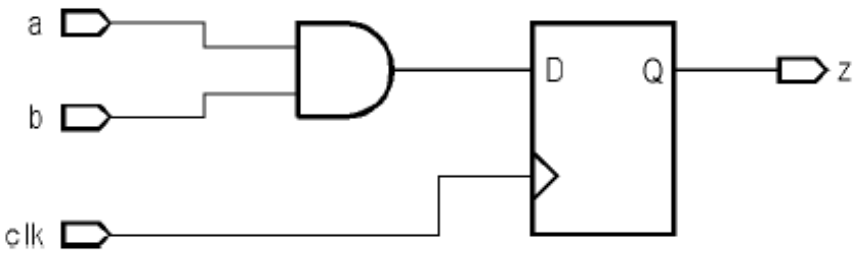
Combinatorial

```
proc5: process
begin
   wait on a, b;
   z <= a and b;
end process proc5;
```



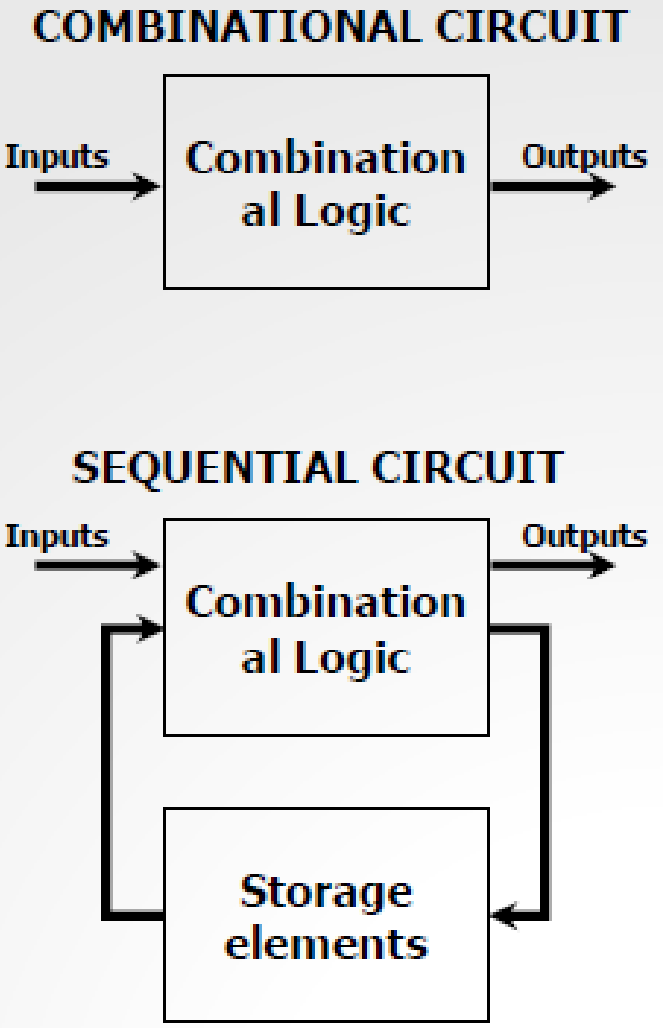
Sequential

```
proc6: process
begin
   wait until clk = '1';
   z <= a and b;
end process proc6;
```

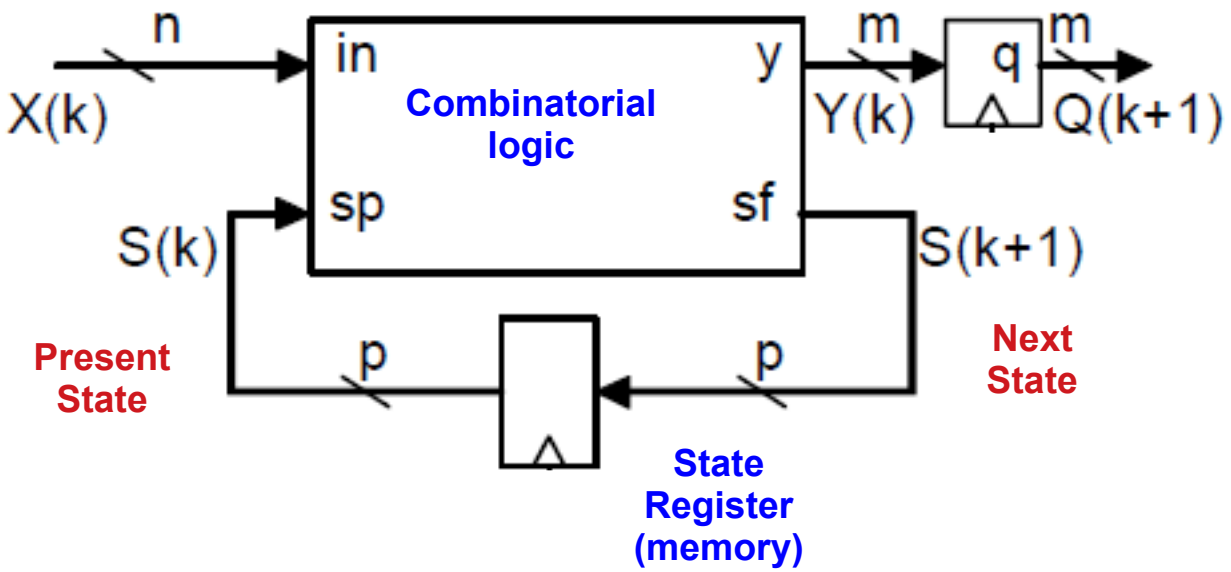


Reminder: Combinatorial vs Sequential

- In combinational circuits, the output only depends upon the present input values.
- There exist another class of logic circuits whose outputs not only depend on the present input values but also on the past values of inputs, outputs, and/or internal signal. These circuits include storage elements to store those previous values.
- The content of those storage elements represents the *circuit state*. When the circuit inputs change, it can be that the circuit stays in certain state or changes to a different one. Over time, the circuit goes through a sequence of states as a result of a change in the inputs. The circuits with this behavior are called *sequential circuits*.



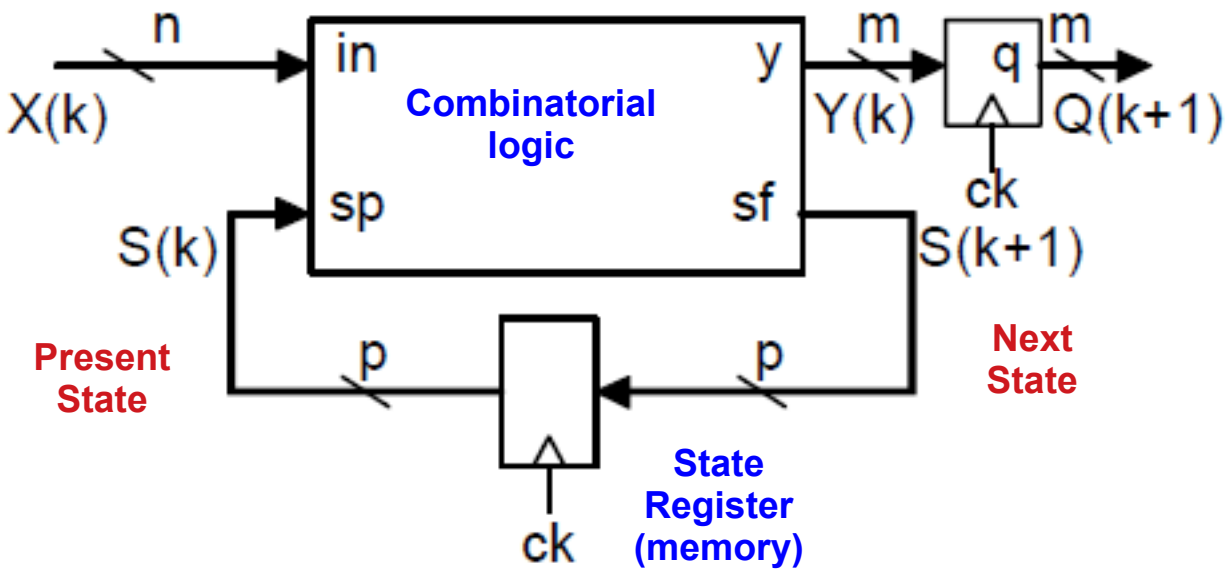
General Sequential Systems



General Sequential systems are such that
past input values either are stored explicitly or
cause the system to enter a particular state

→ working principles are feed-forward and feed-back

General Sequential Systems

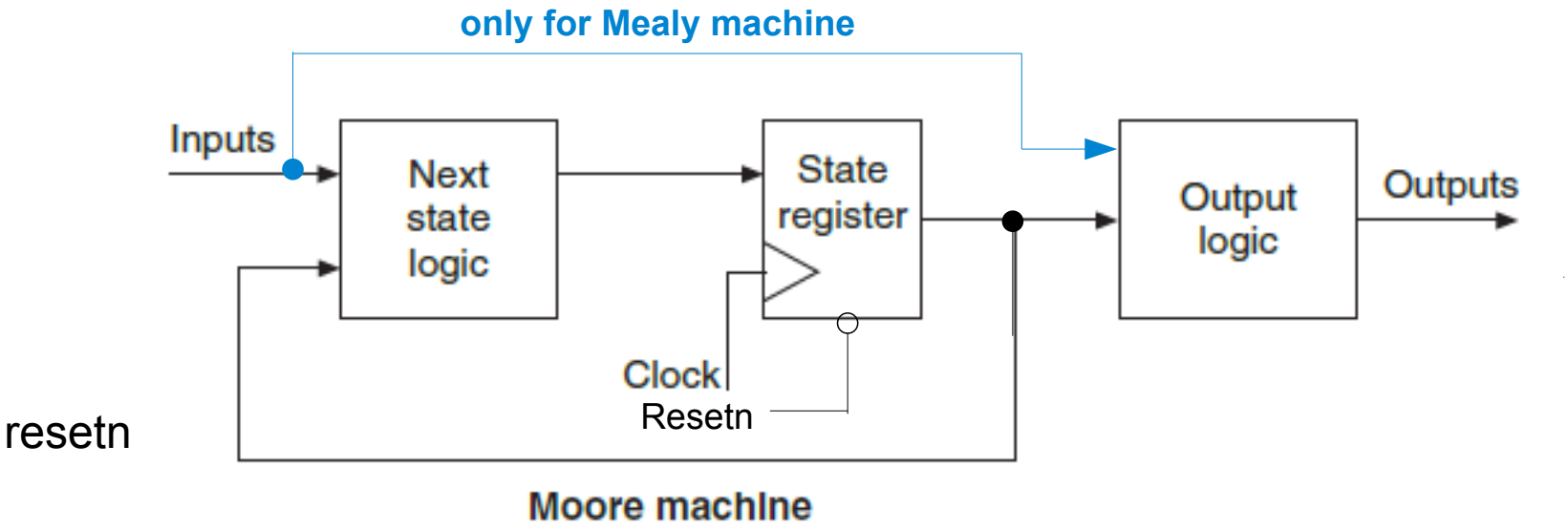


Sequential
synchronous

The present state of the system can be updated
- either as soon as the next state changes → asynchronous
- or only when a clock signal changes → synchronous

Synchronous systems are simpler → start with them

Finite State Machine



Two common models of synchronous sequential systems

- **Moore Machine**: Outputs depend only on the current state of the circuit
- **Mealy Machine**: Outputs depend on the current state of the circuit as well as the inputs of the circuit

Note: the circuit must always start from an initial state

→ there should be a ‘resetn’ signal

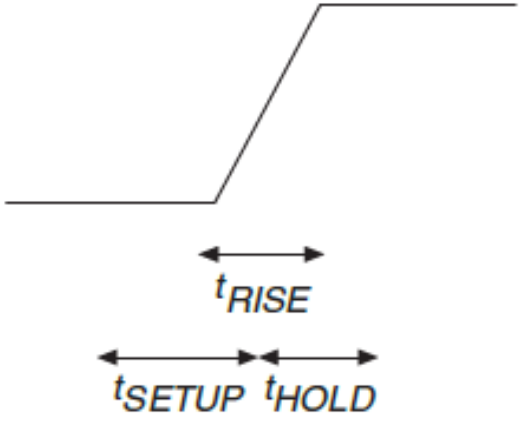
About FSM registers

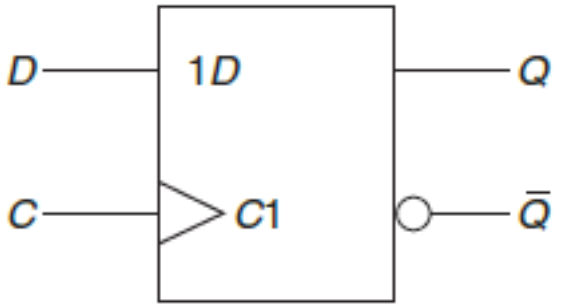
Combinational logic can contain hazards ! The next state logic of the Moore and Mealy machines is simply a block of combinational logic with a number of inputs and a number of outputs. The existence of hazards in the F next state logic could cause the system to go to an incorrect state. There are two ways to avoid that:

1) either the next state logic should include the redundant logic needed to suppress the hazard

2) the state machine should be designed such that a hazard is allowed to occur, but is ignored ← this approach is usually adopted

To ensure that sequential systems are able to ignore hazards,
→ a clock is used to synchronize data



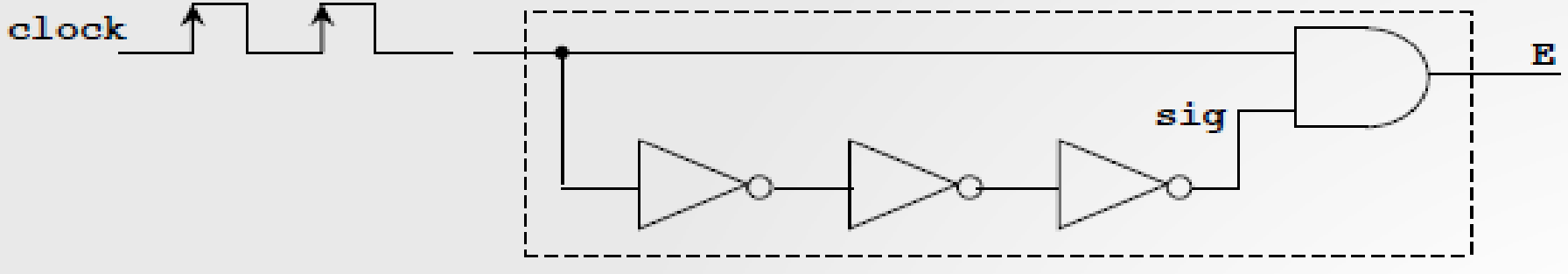


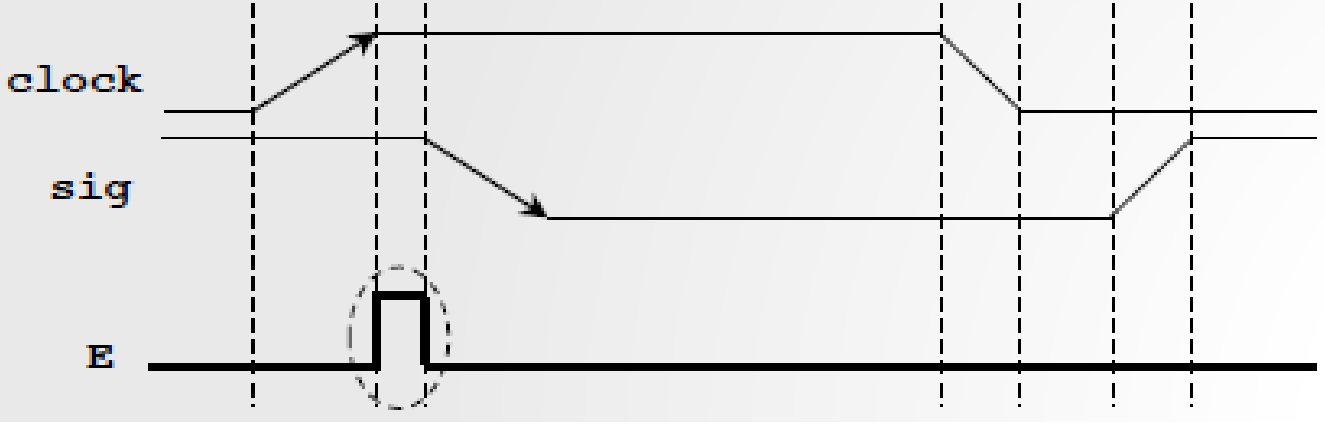
D	C	Q^+	$\bar{Q}^+$
0	↑	0	1
1	↑	1	0
–	0	Q	$\bar{Q}$
–	1	Q	$\bar{Q}$

About edge triggered FF

Flip Flops:

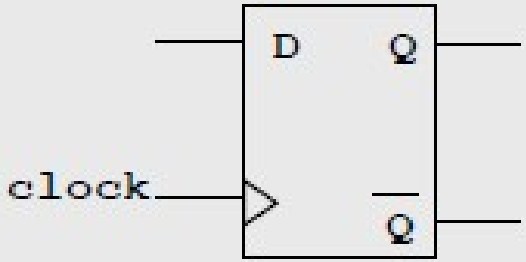
- The edge detector circuit generates E='1' during the edge (rising or falling). We will work with circuits activated by either rising or falling edge. We will not work with circuits activated by both edges.
- An example of a circuit that detects a rising edge is shown below. The redundant NOT gates cause a delay that allows a pulse to be generated during a rising edge (or positive edge).





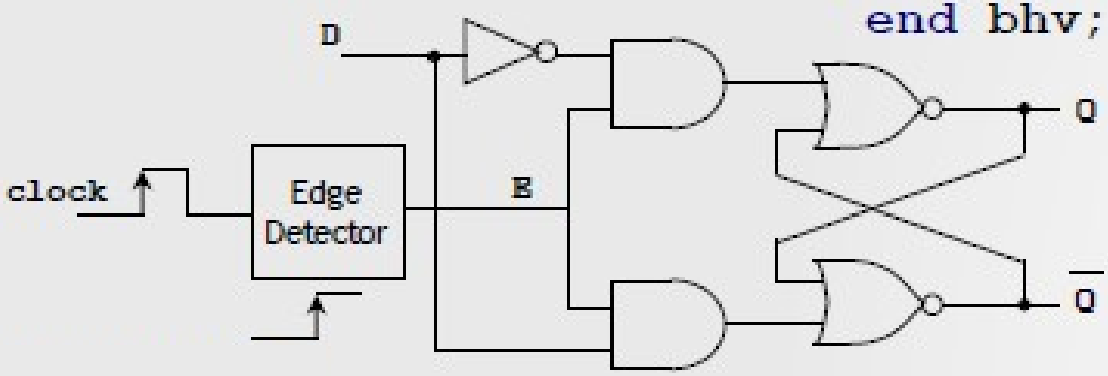
About edge triggered FF

- **D Flip Flop**

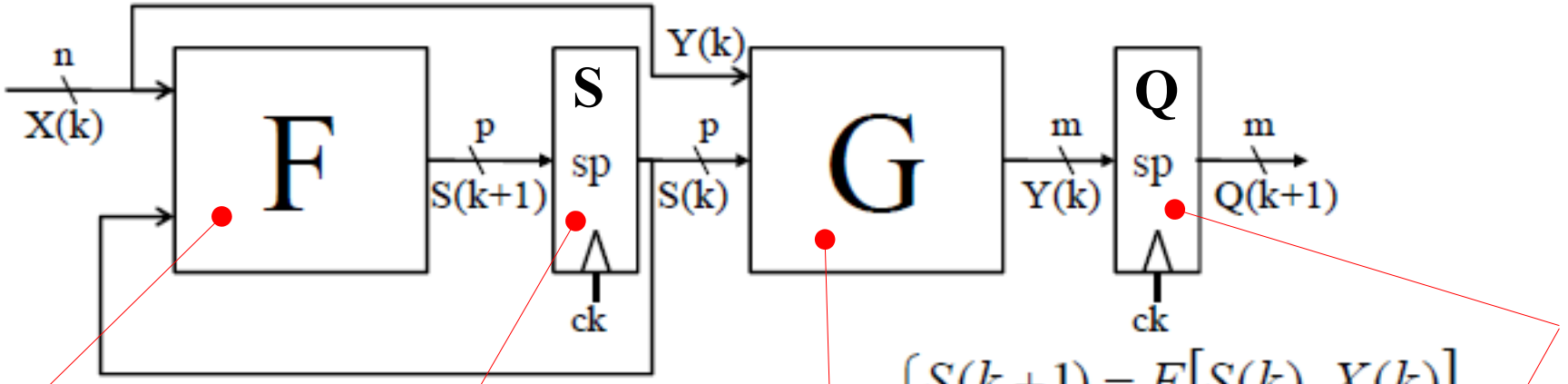


clock	D	Q_{t+1}
↑	0	0
↑	1	1

```
library ieee;
use ieee.std_logic_1164.all;

entity ff_d is
 port ( d, clock: in std_logic;
        q, qn: out std_logic);
end ff_d;

architecture bhv of ff_d is
  signal qt,qnt: std_logic;
begin
  process (d,clock)
  begin
    if (clock'event and clock='1') then
       qt<=d;
    end if;
  end process;
  q <= qt; qn <= not(qt);
end bhv;
```



Finite State Machine



$$\begin{cases} S(k+1) = F\left[S(k), X(k)\right] \\ Y(k) \quad = G\left[S(k), X(k)\right] \end{cases}$$

F
Evaluates
Future
State

S
State
Register
(memory)

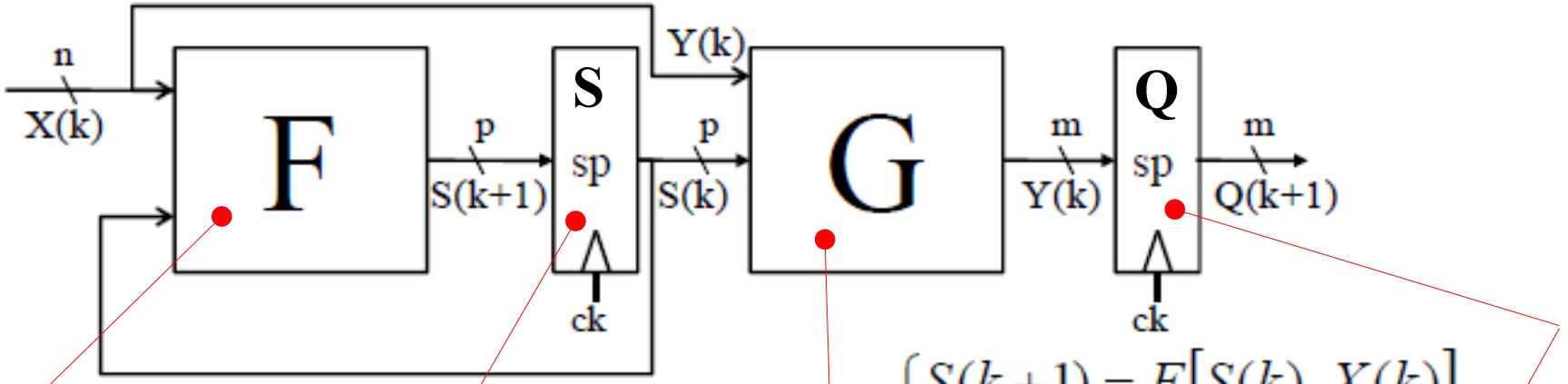
G
Evaluates
Outputs

Q
Outout
Memory
(optional)

Various model options:

- 2 combinatorial (F,G) + 2 clock driven processes (S,Q)
- 2 processes: clock driven (F+S) + combinatorial (G)
- 2 processes: combinatorial (F+G) + clock driven (S+Q)

Finite State Machine



$$\begin{cases} S(k+1) = F\left[S(k), X(k)\right] \\ Y(k) \quad = G\left[S(k), X(k)\right] \end{cases}$$

F
Evaluates
Future
State

S
State
Register
(memory)

G
Evaluates
Outputs

Q
Outout
Memory
(optional)

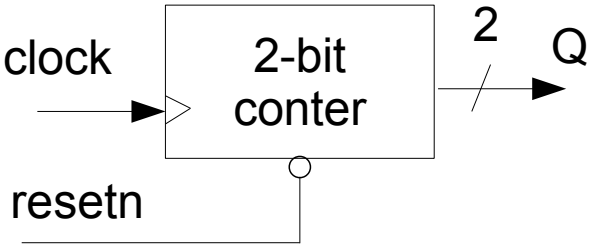
Descriptions

- X,Y,Q usually → std_logic_vector
- S states → ad hoc enumeration type (systhesys tool)

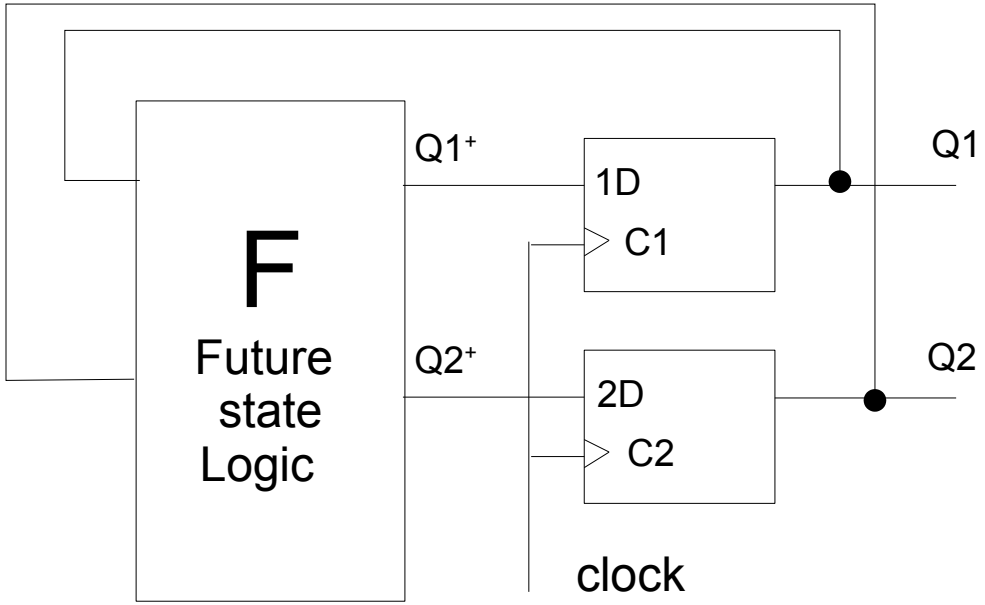
Example of FSM: 2-bit counter

- Moore type FSM
- Functional descr: COUNT = 00 → 01 → 10 → 11 → 00 → …

Block description



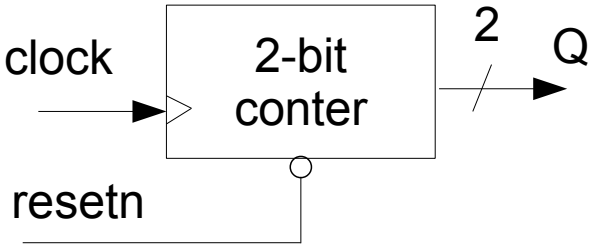
Functional description



Example of FSM: 2-bit counter

- Moore type FSM
- Functional descr: COUNT = 00 → 01 → 10 → 11 → 00 → …

Block description

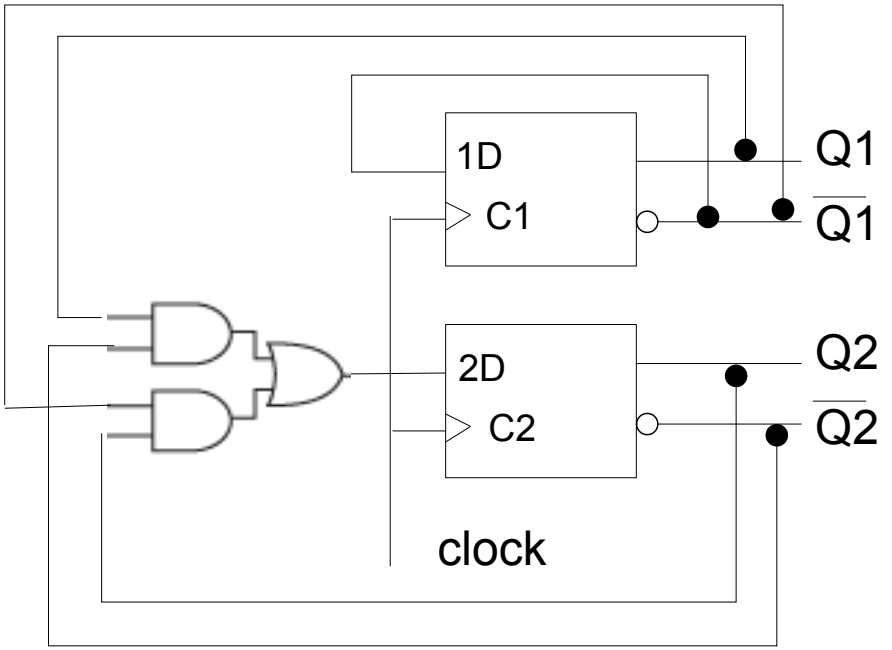


Truth Table description

Q2 Q1	$Q2^+$ $Q1^+$
0 0	0 1
0 1	1 0
1 0	1 1
1 1	0 0

$Q1^+$= NOT Q1
$Q2^+$= Q1 XOR Q2

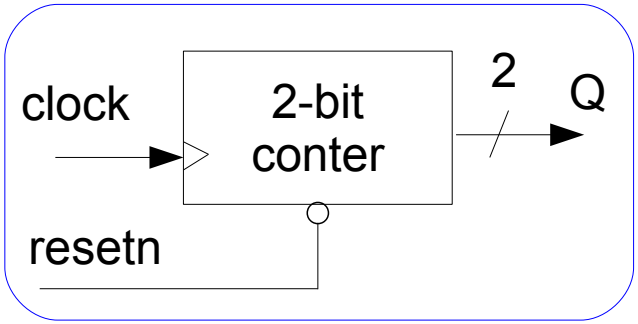
Circuit description



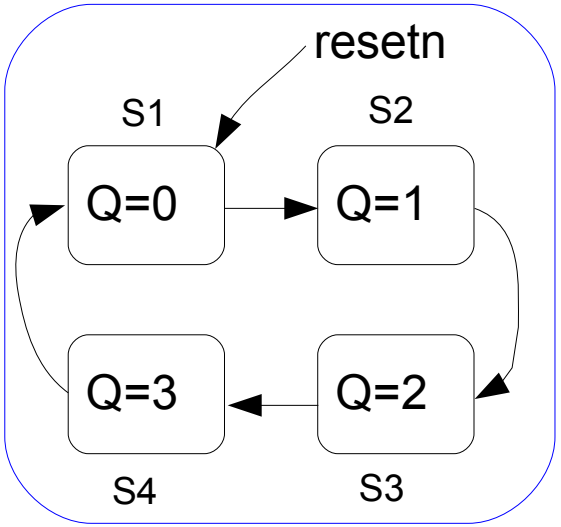
Example of FSM: 2-bit counter

- Moore type FSM
- Functional descr: COUNT = 00 → 01 → 10 → 11 → 00 → …

Block description



Graph type description



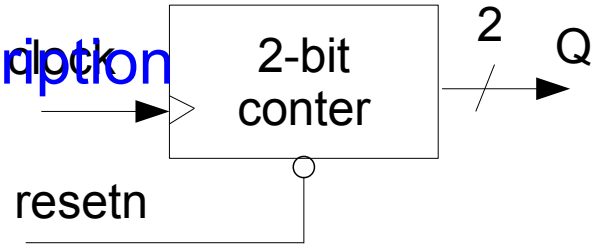
→ Algorithymic type description:
very useful starting point for generic FSM sythesis

Example of FSM: 2-bit counter

- Moore type FSM
- Functional descr: COUNT = 00 → 01 → 10 → 11 → 00 → …

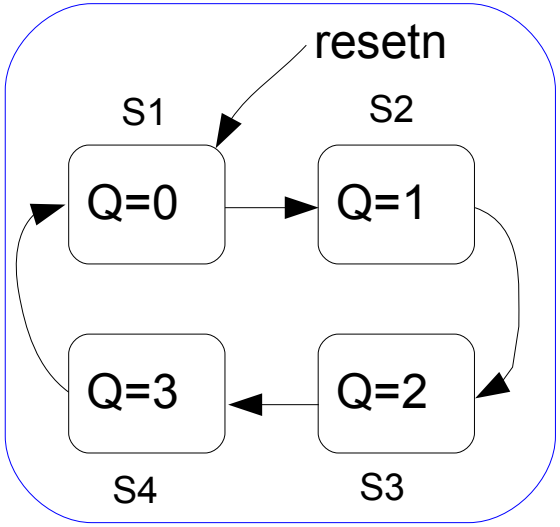
VHDL description

Entity description




```
library ieee;
use ieee.std_logic_1164.all;

entity my_2bitcounter is
    port ( clock, resetn: in std_logic;
           Q: out std_logic_vector (1 downto 0));
end my_2bitcounter;

architecture bhv of my_2bitcounter is
    type state is (S1,S2,S3,S4);
    signal y: state;
begin
    Transitions: process (resetn, clock)
    begin
        if resetn = '0' then -- asynchronous signal
            y <= S1; -- initial state
…
```

custom definition of signal y as ad hoc type state with 4 possible values (S1, …, S4)



process defining the state transition

Example of FSM: 2-bit counter

```
Transitions: process (resetn, clock)
begin
    if resetn = '0' then -- asynchronous signal
            y <= S1; -- initial state
    elsif (clock'event and clock='1') then
        case y is
            when S1 => y <= S2;
            when S2 => y <= S3;
            when S3 => y <= S4;
            when S4 => y <= S1;
        end case;
    end if;
end process;
```

Process defining the state transitions

Note: transitions only occur on the rising edge

```
Outputs: process (y)
begin
    case y is
        when S1 => Q <= "00";
        when S2 => Q <= "01";
        when S3 => Q <= "10";
        when S4 => Q <= "11";
    end case;
end process;
end bhv;
```

Process defining the outputs

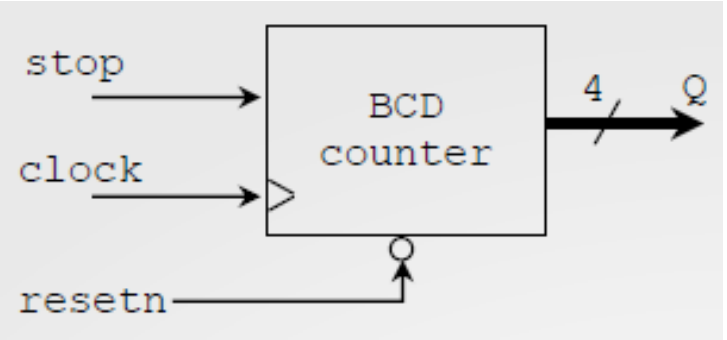
Note 1) the output is not controlled by the clock edge
(only the current state

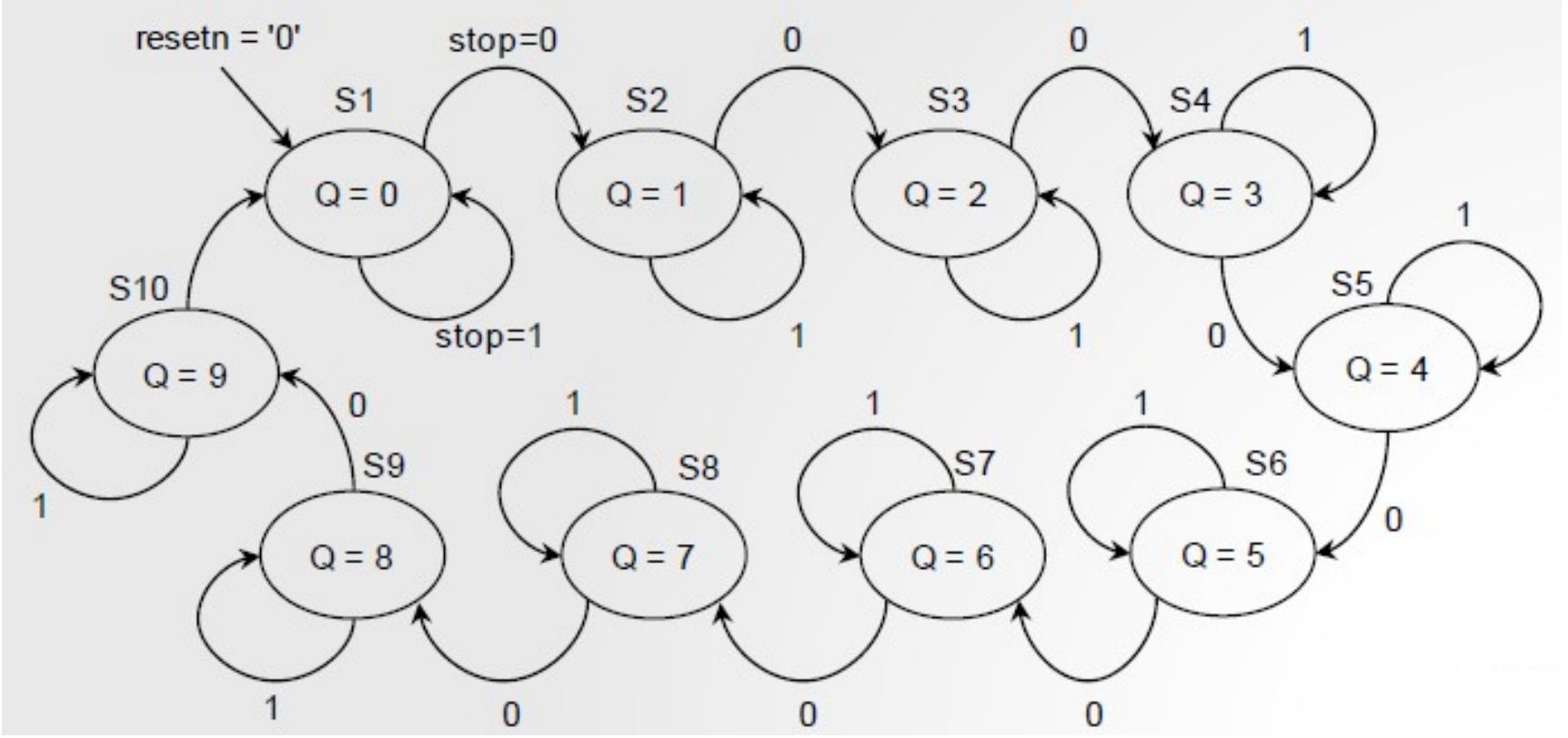
Note 2) the outputs only depend on the current state
→ Moore type FSM

Example of FSM: BCD counter w/ stop signal

- Moore-type FSM
- If the 'stop' signal is asserted, the count stops. If 'stop' is not asserted, the count continues.

BCD = Binary Coded Decimal





Example of FSM: BCD counter w/ stop signal

VHDL description

```
library ieee;
use ieee.std_logic_1164.all;

entity bcd_count is
 port ( clock, resetn, stop: in std_logic;
        Q: out std_logic_vector (3 downto 0));
end bcd_count;

architecture bhv of bcd_count is

  type state is (S1,S2,S3,S4,S5,S6,S7,S8,S9,S10);
  signal y: state;

begin

  Transitions: process (resetn, clock, stop)
  begin
    if resetn = '0' then -- asynchronous signal
       y <= S1; -- initial state
...
```

Custom datatype definition: 'state' with 10 possible values: S1 to S10

Definition of signal 'y' of type 'state'.

Process that defines the state transitions

Example of FSM: BCD counter w/ stop signal

```
...
    elsif (clock'event and clock='1') then
       case y is
          when S1 =>
             if stop='1' then y<=S1; else y<=S2; end if;
          when S2 =>
             if stop='1' then y<=S2; else y<=S3; end if;
          when S3 =>
             if stop='1' then y<=S3; else y<=S4; end if;
          when S4 =>
             if stop='1' then y<=S4; else y<=S5; end if;
          when S5 =>
             if stop='1' then y<=S5; else y<=S6; end if;
          when S6 =>
             if stop='1' then y<=S6; else y<=S7; end if;
          when S7 =>
             if stop='1' then y<=S7; else y<=S8; end if;
          when S8 =>
             if stop='1' then y<=S8; else y<=S9; end if;
          when S9 =>
             if stop='1' then y<=S9; else y<=S10; end if;
          when S10 =>
             if stop='1' then y<=S10; else y<=S1; end if;
       end case;
    end if;
  end process;
...
```

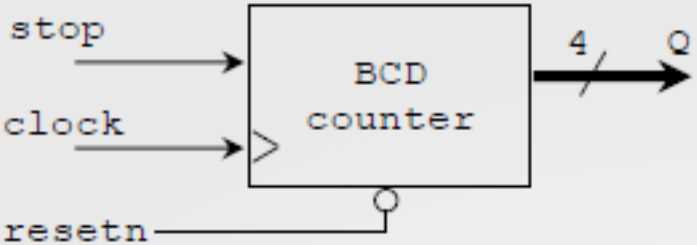
Note that the state transitions depend on the stop signal 'stop'

Process that defines the state transitions

Note that the state transitions only occur on the rising clock edge

Example of FSM: BCD counter w/ stop signal

```
...
    Outputs: process (y)
    begin
        case y is
            when S1 => Q <= "0000";
            when S2 => Q <= "0001";
            when S3 => Q <= "0010";
            when S4 => Q <= "0011";
            when S5 => Q <= "0100";
            when S6 => Q <= "0101";
            when S7 => Q <= "0110";
            when S8 => Q <= "0111";
            when S9 => Q <= "1000";
            when S10 => Q <= "1001";
        end case;
    end process;
end bhv;
```



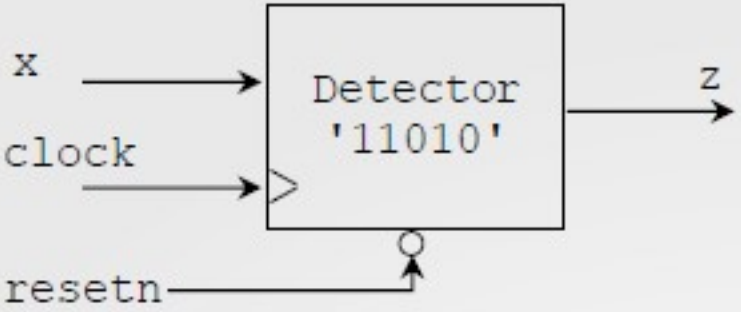
Note that the outputs only depend on the current state, hence this is a Moore machine

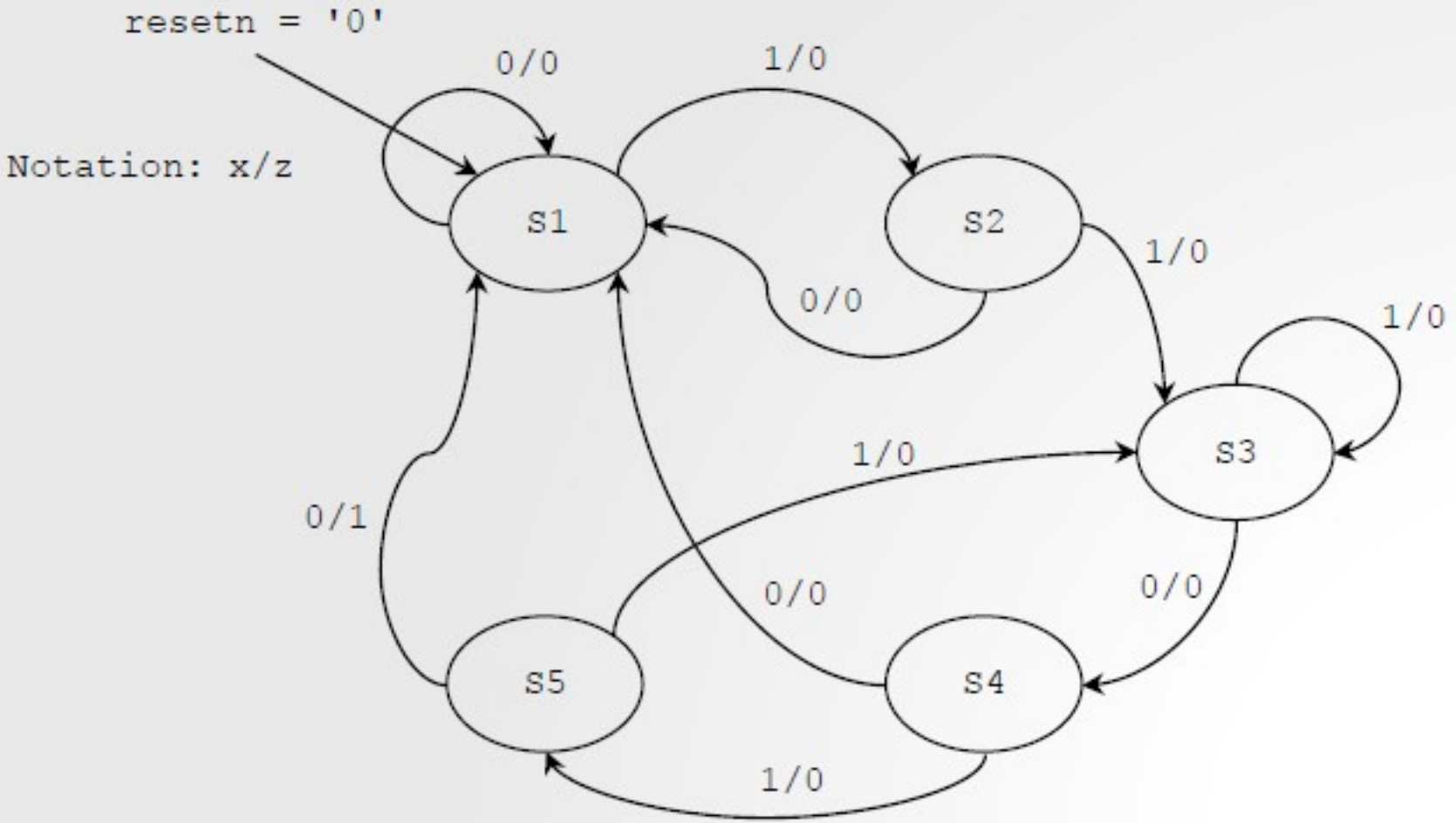
Process that defines the outputs

Note that the output is not controlled by the rising clock edge, only by the current state.

Example of FSM: sequence detector w/ overlap

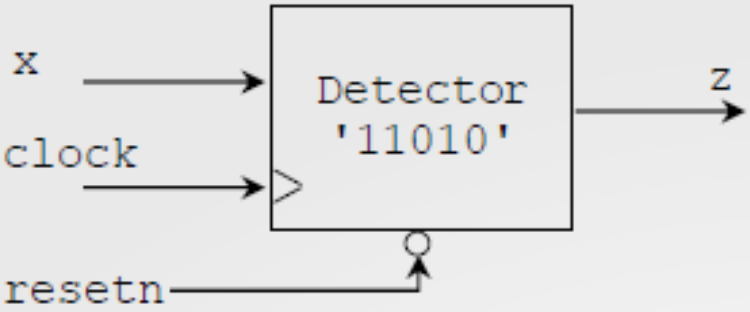
- Mealy-type FSM
- It detects the sequence 11010
- State Diagram: 5 states





Example of FSM: sequence detector w/ overlap

- VHDL Code: Mealy FSM




```
library ieee;
use ieee.std_logic_1164.all;

entity my_seq_detect is
 port ( clock, resetn, x: in std_logic;
        z: out std_logic);
end my_seq_detect;

architecture bhv of my_seq_detect is

  type state is (S1,S2,S3,S4,S5);
  signal y: state;

begin

  Transitions: process (resetn, clock, x)
  begin
    if resetn = '0' then -- asynchronous signal
       y <= S1; -- initial state
```

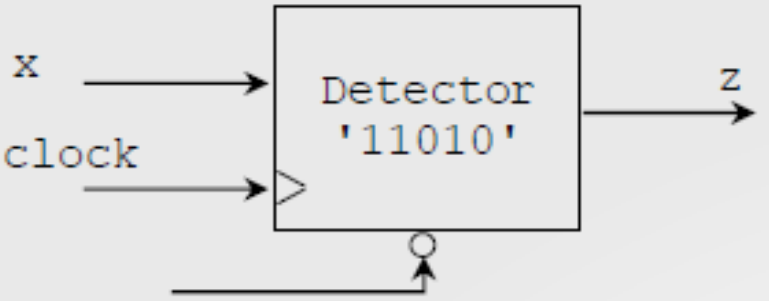
Custom datatype definition: 'state' with 5 possible values: S1 to S5

Definition of signal 'y' of type 'state'.

Process that defines the state transitions

Example of FSM: sequence detector w/ overlap

- VHDL Code: Mealy FSM




```
...
    elsif (clock'event and clock='1') then
       case y is
          when S1 =>
             if x = '1' then y<=S2; else y<=S1; end if;
          when S2 =>
             if x = '1' then y<=S3; else y<=S1; end if;
          when S3 =>
             if x = '1' then y<=S3; else y<=S4; end if;
          when S4 =>
             if x = '1' then y<=S5; else y<=S1; end if;
          when S5 =>
             if x = '1' then y<=S3; else y<=S1; end if;
       end case;
    end if;
  end process;
...
```

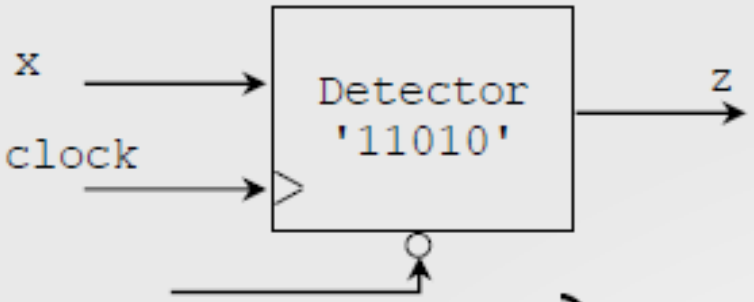
Note that the state transitions depend on the input signal 'x'

Process that defines the state transitions

Note that the state transitions only occur on the rising clock edge

Example of FSM: sequence detector w/ overlap

- VHDL Code: Mealy FSM



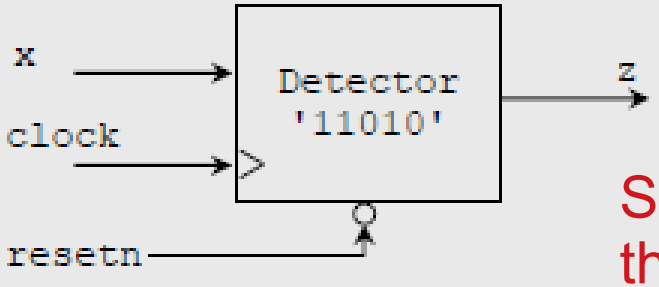

```
...
   Outputs: process (x,y)
   begin
      z <= '0';
      case y is
         when S1 =>
         when S2 =>
         when S3 =>
         when S4 =>
         when S5 =>
            if x = '0' then z <= '1'; end if;
       end case;
  end process;
end bhv;
```

Note that the output depends on the current state and the input 'x', hence this is a Mealy machine.

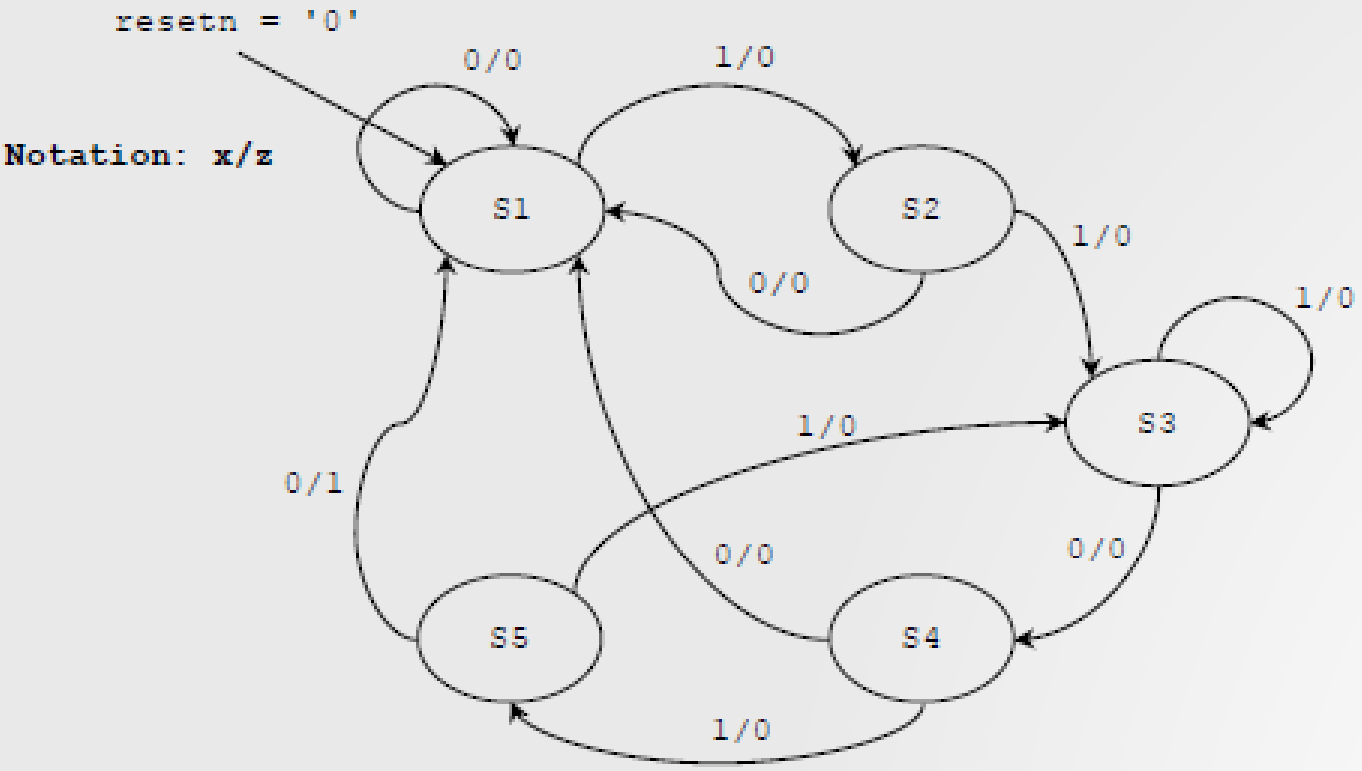
Process that defines the outputs

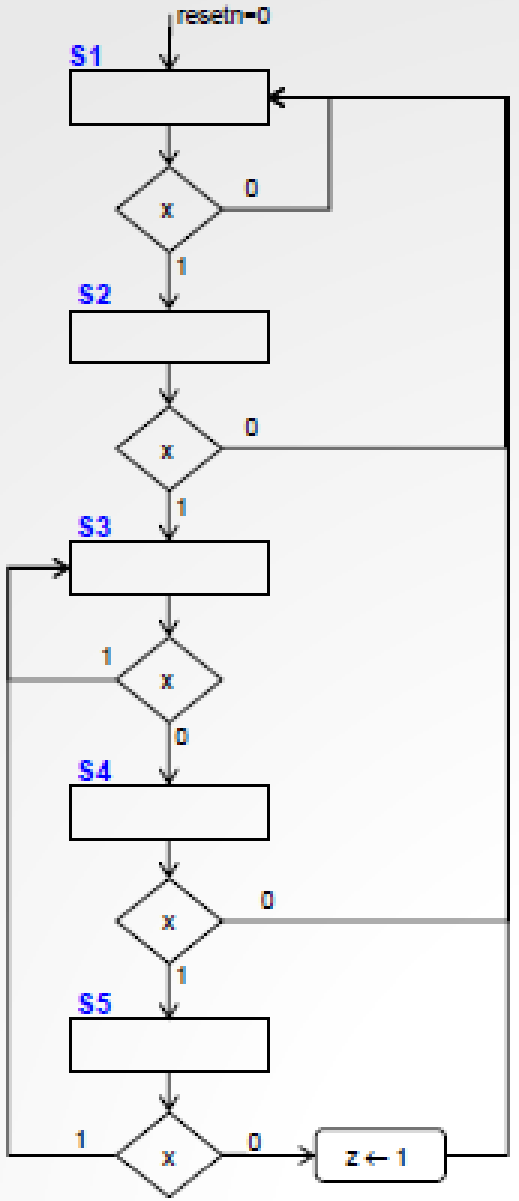
Algorithmic State Machine (ASM) representation

- This is an efficient way to represent Finite State Machines.
- We use the 11010 detector as an example here.



Starting points for building the VHDL description





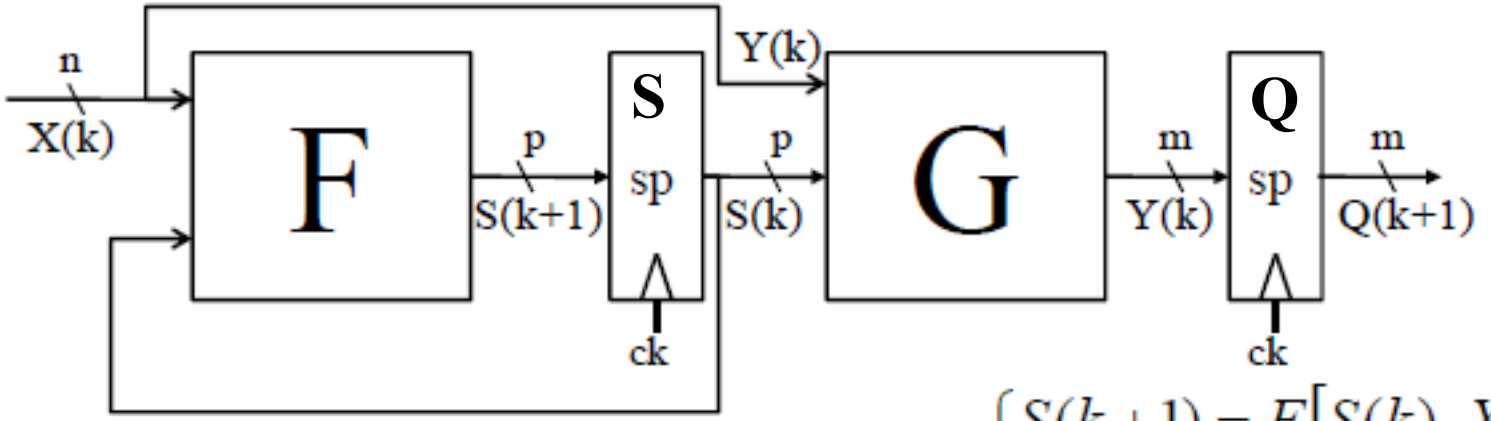
FMS models

F Evaluates Future State

G Evaluates Outputs

Q Outout Memory (optional)

S State Register (memory)



$$\begin{cases} S(k+1) = F\left[S(k), X(k)\right] \\ Y(k) \quad = G\left[S(k), X(k)\right] \end{cases}$$

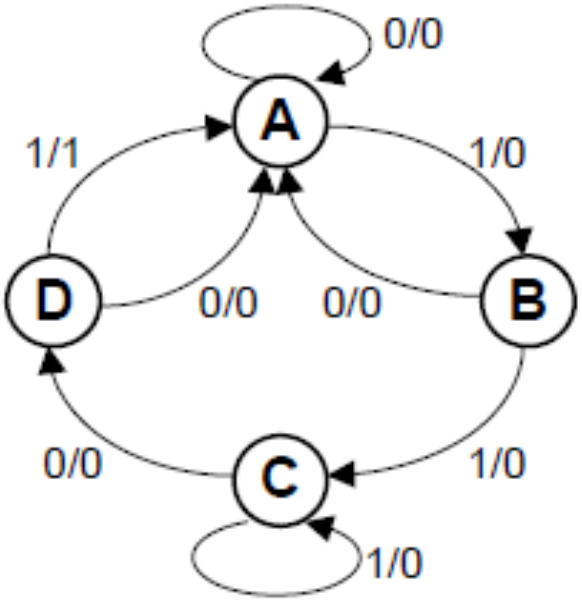
Various model options:

- 2 combinatorial (F,G) + 2 clock driven processes (S,Q)
- 2 processes: clock driven (F+S) + combinatorial (G)
- 2 processes: combinatorial (F+G) + clock driven (S+Q)

up to now this was the model, let‘s check alternative

FMS – sequence detector “1101“

Mealy type




```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity MSF1 is
port(MSF_IN, CK, R: in std_logic;
MSF_OUT: out std_logic);
end MSF1;

architecture RTL of MSF1 is
-- STATO e` un'enumerazione; i dati di tipo STATO possono
-- assumere solo i valori A, B, C, D. I due segnali stato_pres
-- e stato_fut sono dichiarati di questo tipo
type STATO is (A, B, C, D);
signal STATO_PRES, STATO_FUT: STATO := A;
signal Y: std_logic;

begin
-- Il processo P1 modella le funzioni F e G della macchina,
-- quindi si attiva quando c'e` un evento su STATO_PRES o
-- sul segnale di ingresso MSF_IN
P1: PROCESS(STATO PRES MSF
STATO_PRES, MSF_IN)
begin
```

Process P1 modeling F and G

FMS – sequence detector “1101“

```
-- le funzioni F e G sono
-- descritte con istruzioni
-- case e if...then...else
    case STATO_PRES is
      when A =>
        if MSF_IN = '0' then
          STATO_FUT <= A;
          Y <= '0';
        else
          STATO_FUT <= B;
          Y <= '0';
        end if;
      when B =>
        if MSF_IN = '0' then
          STATO_FUT <= A;
          Y <= '0';
        else
          STATO_FUT <= C;
          Y <= '0';
        end if;
      when C =>
        if MSF_IN = '0' then
          STATO_FUT <= D;
          Y <= '0';
        else
          STATO_FUT <= C;
          Y <= '0';
        end if;
      when D =>
        if MSF_IN = '0' then
          STATO_FUT <= A;
          Y <= '0';
        else
          STATO_FUT <= A;
          Y <= '1';
        end if;
    end case;
  end process P1;
-- continua
```

FMS – sequence detector “1101“

```
-- Il processo P2 modella i
-- registri della macchina
P2: process(CK)
  begin
    if CK'event and CK = '1' then
      if R = '0' then
        STATO_PRES <= A;
        MSF_OUT <= '0';
      else
-- STATO_FUT proviene dal
-- dal processo P1
        STATO_PRES <= STATO_FUT;
-- il registro di uscita è
-- opzionale; elimina i glitch ma
-- ritarda di un ciclo l'uscita
        MSF_OUT <= Y;
      end if;
    end if;
  end process P2;
end architecture rtl;
```

Process P2 modeling S and Q

Note: Q register useful against glitches but result in delayed output

FMS – sequence detector “1101“

Moore alternative

An alternative Moore type model consists in exploiting the state signal for representing/ encoding also the output (Then the output logic G is no more necessary just Q is needed)

output = state

Description: start from Mealy model and split A state into
- A0 (transition from D when MSF_IN=0) and
- A1 (from D when MSF_IN=1)

Output coding:
Y=0 for states A0, B, C, D
Y=1 for state A1

→ sequence: **A0:**000 **B:**001 **C:**010 **D:**011 **A1:**111

… MSB is the output Y bit

FMS – sequence detector “1101“

Moore alternative

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity MSF1 is
  port(MSF_IN, CK, R: in
   std_logic;
        MSF_OUT: out std_logic);
end MSF1;
architecture RTL of MSF1 is
 type STATO is (A0,A1,B,C,D);
 attribute ENUM_ENCODING: STRING;
 attribute ENUM_ENCODING of STATO:
  type is "000 111 001 010 011";
 signal STATO_PRES, STATO_FUT:
  STATO := A0;
 signal Y: std_logic;
begin
```

```
  P1: PROCESS(STATO_PRES, MSF_IN)
  begin
    case STATO_PRES is
      when A0 =>
        Y <= '0';
        if MSF_IN = '0' then
          STATO_FUT <= A0;
        else
          STATO_FUT <= B;
        end if;
      when A1 =>
        Y <= '1';
        if MSF_IN = '0' then
          STATO_FUT <= A0;
        else
          STATO_FUT <= B;
        end if;
      when B =>
        Y <= '0';
        ...
      when D =>
        Y <= '0';
        if MSF_IN = '0' then
          STATO_FUT <= A0;
        else
          STATO_FUT <= A1;
        end if;
    end case;
  end process P1;
```

Process P1 modeling F and S

```
  P2: process(CK)
  begin
    if CK'event and CK = '1' then
      if R = '0' then
        STATO_PRES <= A0;
        MSF_OUT <= '0';
      else
        STATO_PRES <= STATO_FUT;
        MSF_OUT <= Y;
      end if;
    end if;
  end process P2;
end architecture rtl;
```

Process P2 modeling Q

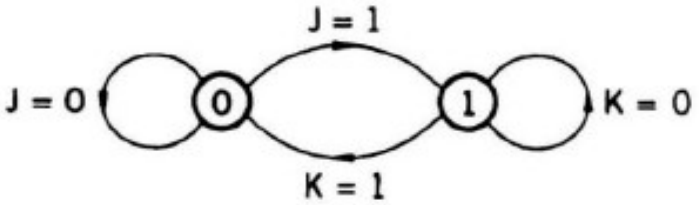
Notes: State Machines

Edge-triggered flip-flops, synchronous counters, and shift registers are simple examples of state machines → which are synchronous systems that consist only of edge-triggered flip-flops and combinational circuits

• The inputs to the flip-flops in a state machine are logical functions only of the external inputs (if any) and of the outputs of the flip-flops themselves
External inputs are assumed to be synchronized to the system clock

• The state of the machine between clock transitions is completely determined by the state of the flip-flops. In any given state the next state is decided by the combinational logi

• One orderly way of describing a state machine is a state diagram, in which the states are nodes and the transitions between states are indicated by directed traces labeled with the conditions tha enable these transitions.



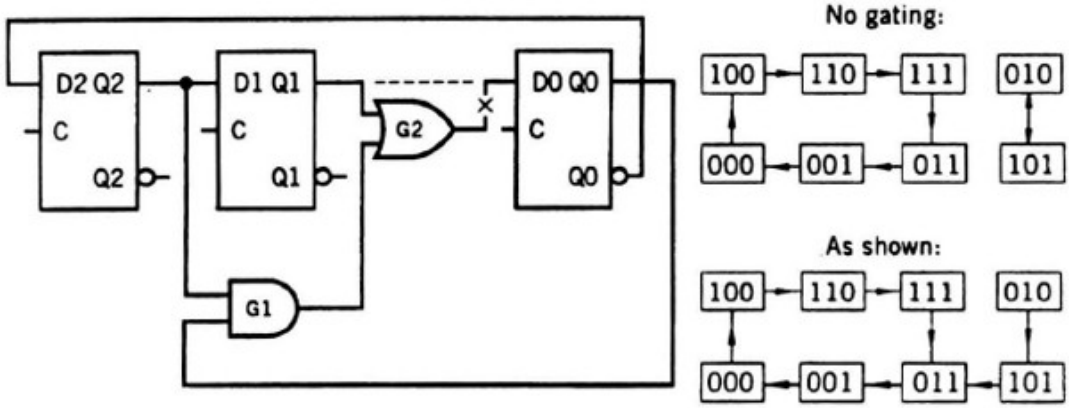
Example :
state diagram of a JK flip-flop
The two states are labeled according to the value of Q
If Q=0, Q will remain at 0 if J=0, and it will go to 1 if J=1
If Q=1, Q will remain at 1 if K=0, and it will go to 0 if K=1

Notes: State Machines – initial state

A state machine with N flip-flops has 2N possible states → state machines generally have more states than are needed for a given task

Excess states → consideration of what action to take if the machine falls into one of these states (either at power-up or because of noise)

Example: the divide-by-6 Johnson counter



The counter has three flip-flops → eight possible states

The normal sequence is 000,100,110,111,011,001,000 and so on

In the absence of gating, (that is if G2 is removed and Q1 is connected to DO) states 010 and 101 form a disjoint group
→ to avoid this possibility, G1 detects state 101 and makes D0 equal to 1, thus simulating state 111 and forcing the counter into state 011

Notes: State Machines – advantages

1) state machines are insensitive to races in their combinational logic
because races can occur only between clock transitions

Example: in the shift register D3 will exhibit a short pulse if S/$\underline{L}$ goes 1→0
when DP3 and DH are both 1; but S/$\underline{L}$ is synchronized to the clock and can
thus change only immediately after a clock transition, so that the pulse has
no effect on Q3

2) state machines show reduced sensitivty to inductively or capacitively coupled transients
generated by changes of state, due to the fact that outputs change
well after the decision to change is firm

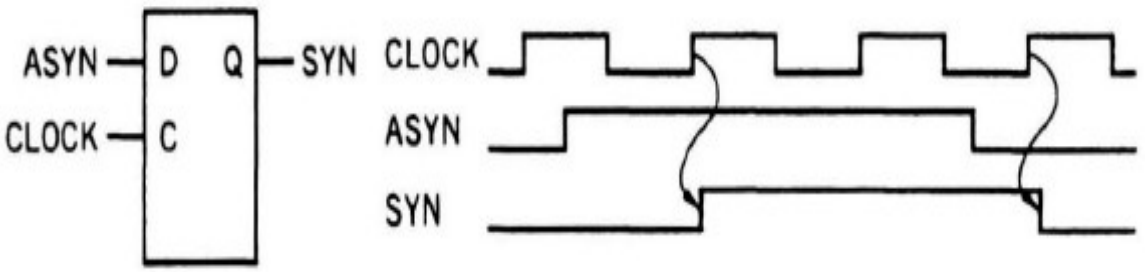
This advantage is particularly valuable when changes of state trigger
high-power circuits and there are inputs connected to the outside world
by long unshielded lines

Synchronizers → for Async FSM

External inputs to state machines are assumed to be synchronous…
… but asynchronous inputs do exist (except in the rarest of cases)
→ must somehow be converted into synchronous versions

Example: the “standard synchronizer“

Asynchronous input ASYN drives
the D input of a D flip-flop
→ a change of state in D is
synchronously reflected at
output SYN at the
first following clock transition

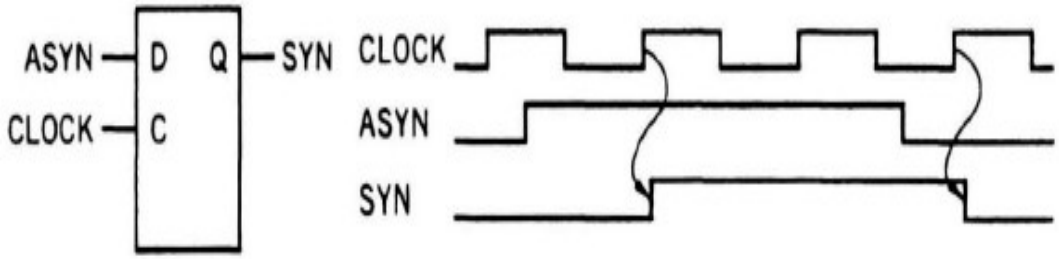


Note: there exists a chance of synchronization failure (as in RS flip-flops)
ie a change in ASYN might violate setup or hold requirements
→ the flip-flop becomes metastable

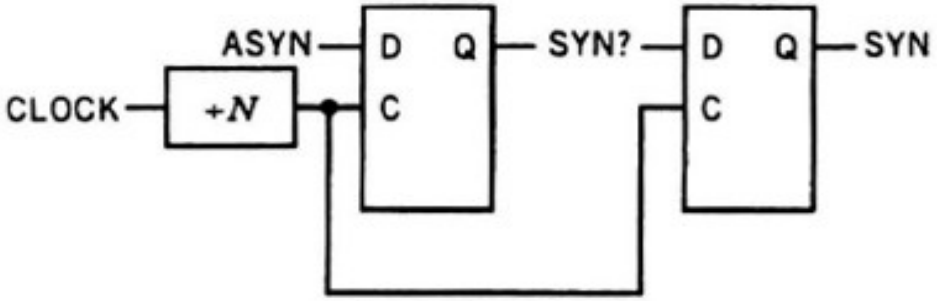
Synchronizers – failure and fundamental rule

There are several procedures to reduce the probability P of synchronization failure:
of which we will mention only two; both result in a longer response time to ASYN

(1) driving the standard synchronizer with a submultiple f/N of the clock frequency f
→ the frequency at which ASYN is sampled (and thus P) is reduced by a factor N



(2) a much more effective procedure is illustrated by the two-stage synchronizer
Both ASYN and the intermediate output SYN? are sampled at a frequency f/N
→ this allows a time T = N/f for SYN? to stabilize before it is copied to SYN
Since P decreases exponentially with T, it can be reduced by orders of magnitude beyond a factor N



Note: a fundamental rule about synchronization is that it must be done only once for a given asynchronous input because different synchronizers might arrive at different decisions even in the case that they do not become metastable:
Example: an otherwise acceptable clock skew might make one synchronizer recognize a change that another synchronizer ignores

Additional material