

VO Hardware Modeling – 182.696

HW Modeling - Design Entry

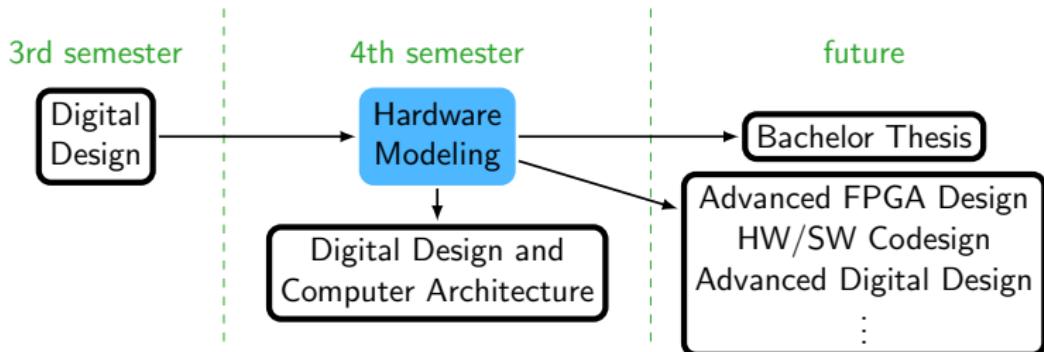
Jürgen Maier and Florian Huemer

{jmaier,fhuemer}@ecs.tuwien.ac.at

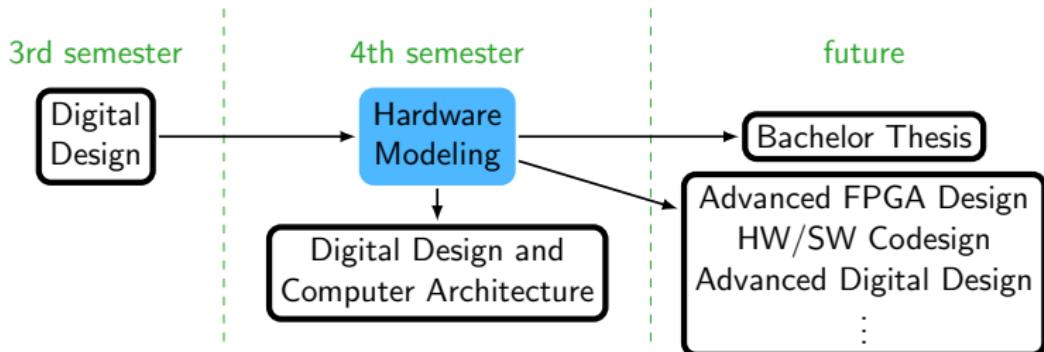
Institute of Computer Engineering
Embedded Computing Systems Group
TU Wien, Vienna, Austria

Summer Term 2019

Curriculum



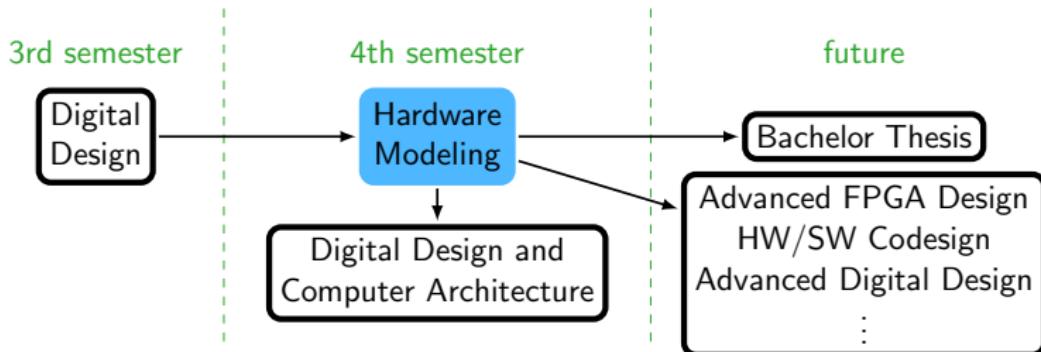
Curriculum



Requirements:

STEOP

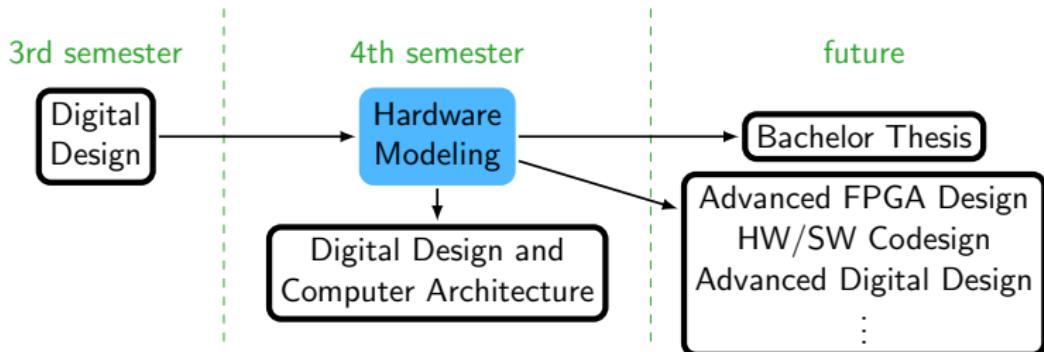
Curriculum



Expected Knowledge: **Digital Design**

- logic gates (OR, MUX, FF, ...)
- Mealy / Moore automata
- Y-diagram
- control flow concepts (if-then-else, loops, ...)
- synchronous circuit design
- hardware design flow including verification

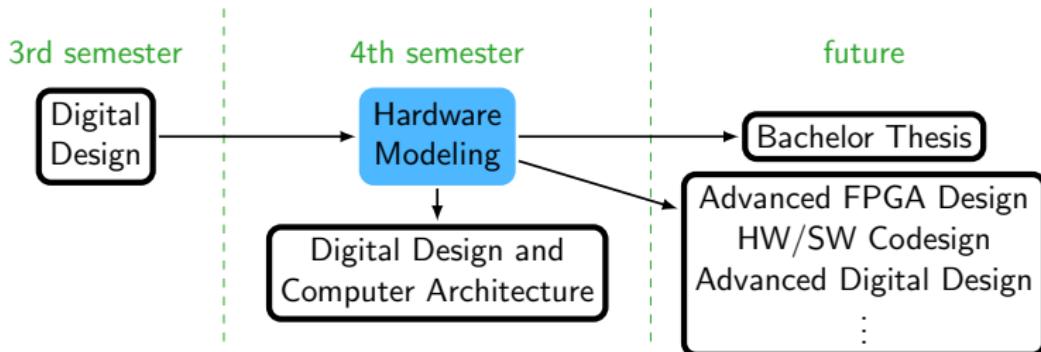
Curriculum



Accompanying courses:

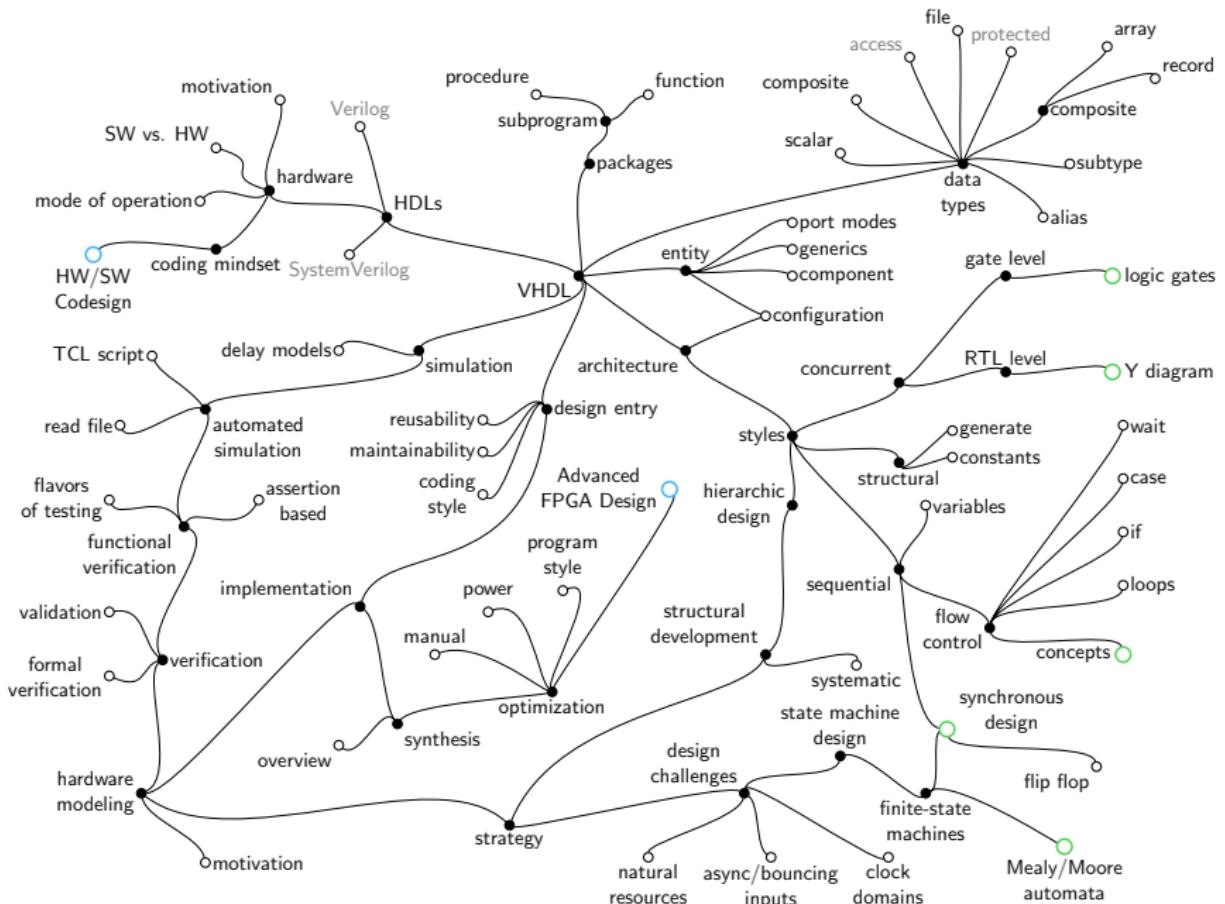
- 182.695 LU Digital Design and Computer Architecture

Curriculum



Continuative courses:

- Bachelor Thesis
- 182.756 VU Advanced FPGA Design
- 182.700 VU HW/SW Codesign
- 182.701 LU HW/SW Codesign



Lecture Organisation

- lecture will be blocked
 - from March 6th to April 12th
 - Wednesday, 9:15 to 10:45 in HS 17
 - Friday, 13:15 to 14:45 in EI 8
 - no lectures on **20.3.** and **3.4.**
- register in TISS to get access to TUWEL
- written exam (on paper, no PC)
 - duration: 60 minutes
 - registration in TISS required
- lecture revised
 - provide feedback!

Didactic Concept

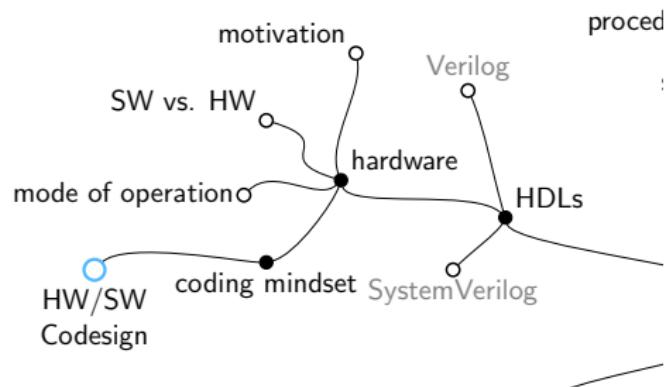
- lectures get recorded
 - audio & slides only
 - available for streaming in [TUWEL course](#) after lecture
 - code examples and slides available on [TUWEL](#) or [ownCloud](#) (weblinks on slides)
- features of lecture attendance
 - interactive learning with lecturer / other students
 - [online voting](#), BYOD (bring your own device)
 - possibility to ask / discuss with others
- independent practicing indispensable
 - tools presented during lecture ([Quartus](#), [Questa](#)([Model](#))[Sim](#))
 - accompanying examples in [TUWEL course](#)
 - self-exploration

Additional Ressources (not complete)

- several open source / free web resources
 - Intel VHDL Basics online course
 - try "VDHL Basics" on youtube
- books (excerpt)
 - VHDL-Synthese (german)
 - Lehrbuch Digitaltechnik (german)
 - The Designer's Guide to VHDL
 - Electronic Design Automation
- tools
 - QuestaSim and Intel Quartus (over ssh -X)
 - GHDL
 - Plug-Ins for terminal, vim, emacs, ...
- manuals
 - VHDL Standard 2008
 - Intel Quartus Prime Pro User Guides
 - QuestaSim User/Reference Manual

Hardware Design

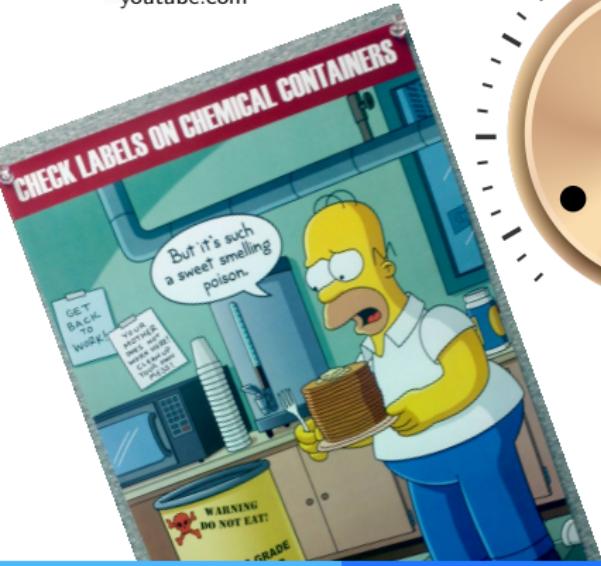
Motivation



Why Develop Hardware?

Why not just ...

- run everything on general purpose processor
- let few specialists design and optimize those
- concentrate on writing good software
- work on high abstraction level



Why Hardware Implementation?

- very high speed / high throughput / low latency
 - hardware designed for specific task
 - off-the-shelf (OTS) processor might be too slow
- high efficiency / low power consumption
 - low leakage due to small size
 - most of functionality of OTS processor not required
- safety / security
 - What is inside an OTS processor?
- custom optimization of above factors possible
 - can be adopted to current needs

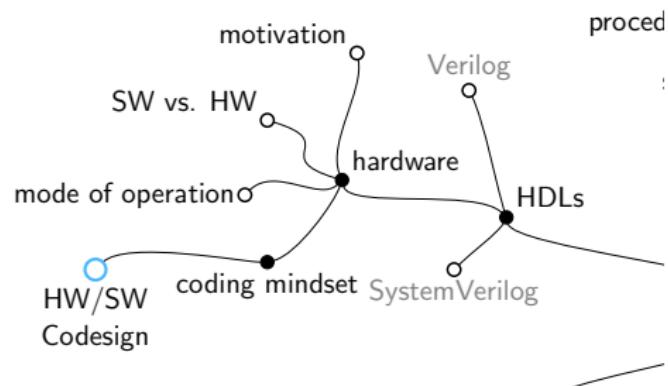
When use custom hardware design?

- specific requirements have to be met regarding
 - speed, throughput, latency, power, area, reliability, ...
- higher development costs/time can be tolerated
 - a lot higher compared to pure software development
- application reasonable
 - gain exceeds costs
 - task is precisely specified, encapsulated and executed frequently
- field of application
 - accelerators, Internet of Things (IoT), aerospace, ...
 - ASIPs, reconfigurable CPUs, ...



Hardware Design

From Software to Hardware Development

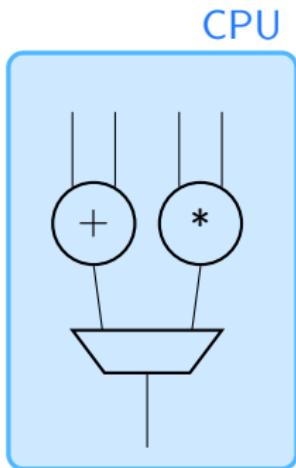


Software Development

```
1  if (x > y)  
2      a = x * y;  
3  else  
4      a = x + y;  
5  
6  return a;
```

- assumption
 - simple CPU
 - one adder, one multiplier
- execution sequential
 - CPU processes one after another
 - only required statements executed
- How about the CPU?
 - Which hardware components are active?
 - When are they active?

Hardware execution

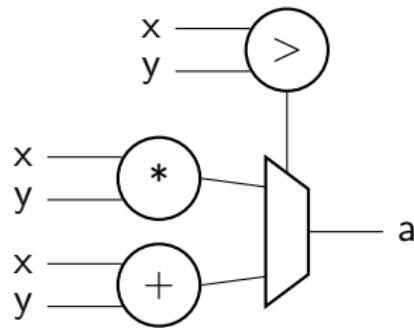


- What is active?
 - Everything!
 - can only be turned off by removing voltage source
- When is it active?
 - All the time!
 - result computed even when not required
- has to be considered when designing hardware

Hardware Development

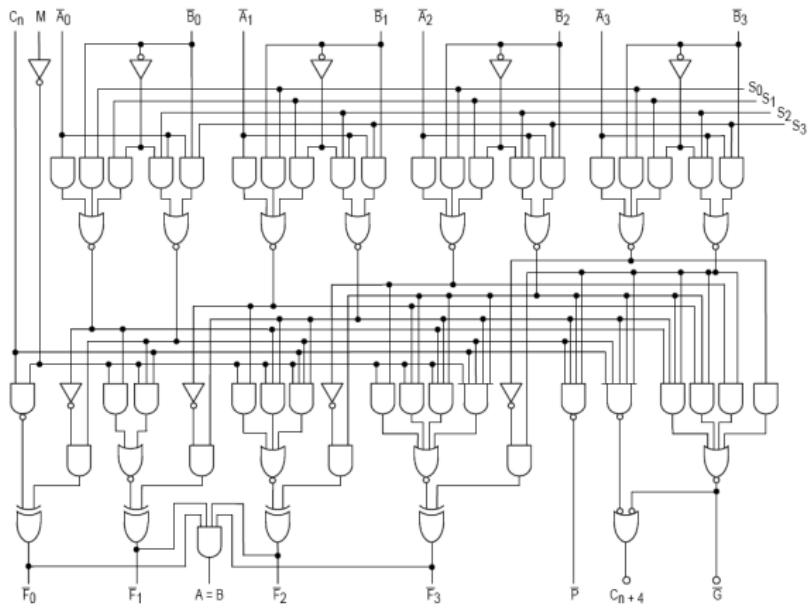
- this mindset used in hardware programming
 - only functionality described
 - everything done in parallel
 - sequential execution has to be explicitly introduced
- conversion of if-statement
 - both branches executed
 - selection via multiplexer
 - compare threads in software

```
1 if (x > y)
2   a = x * y;
3 else
4   a = x + y;
5
6 return a;
```



Hardware Development cont'd

- high parallelism
 - logic gates work at every time instant
 - chances of glitches and runts



blog.danyll.com

Coding in Hardware

- careful design necessary
 - effects on the generated hardware have to be predicted
 - parallel vs sequential execution
- code mere description of behavior
 - tools interpret and generate hardware
 - can be considered as code generators
 - verification required!

```
1  for (i=0; i<n; i++){  
2      x *= a;           ⇒      n multipliers in parallel  
3  }
```

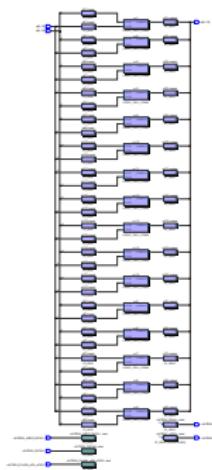
Coding in Hardware cont'd

- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?

$$N \text{ Bits} \Rightarrow N \times \begin{cases} \text{wires} \\ \text{memory cells} \\ \text{logic} \end{cases}$$

Coding in Hardware cont'd

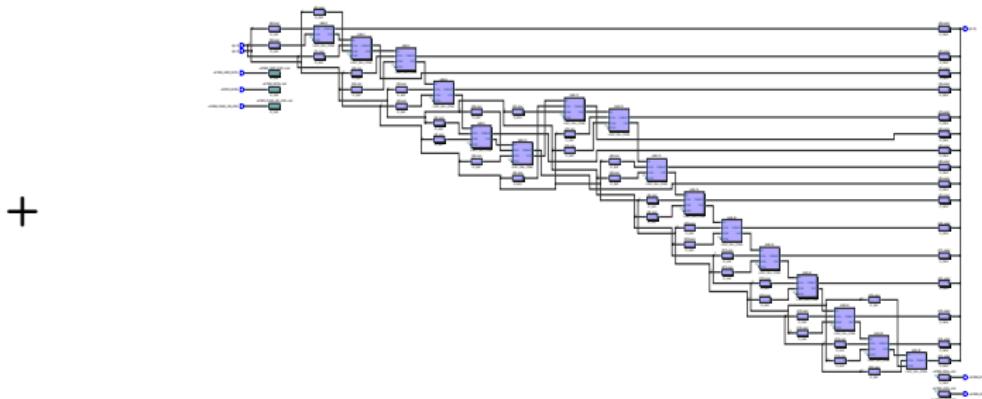
- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?



AND

Coding in Hardware cont'd

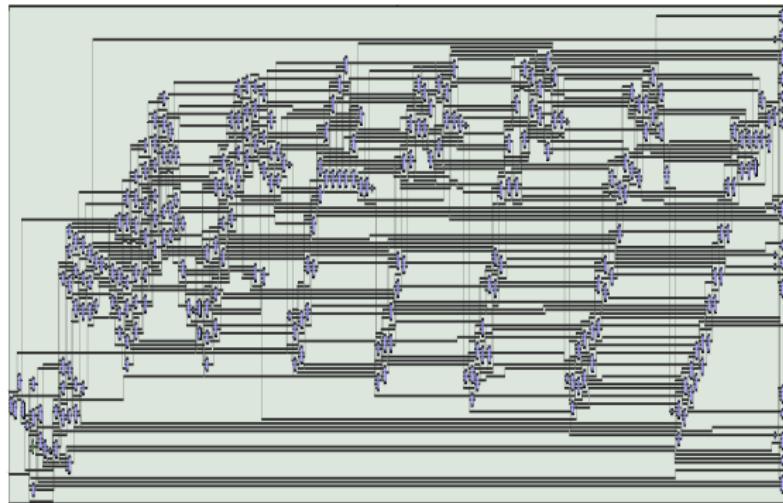
- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?



Coding in Hardware cont'd

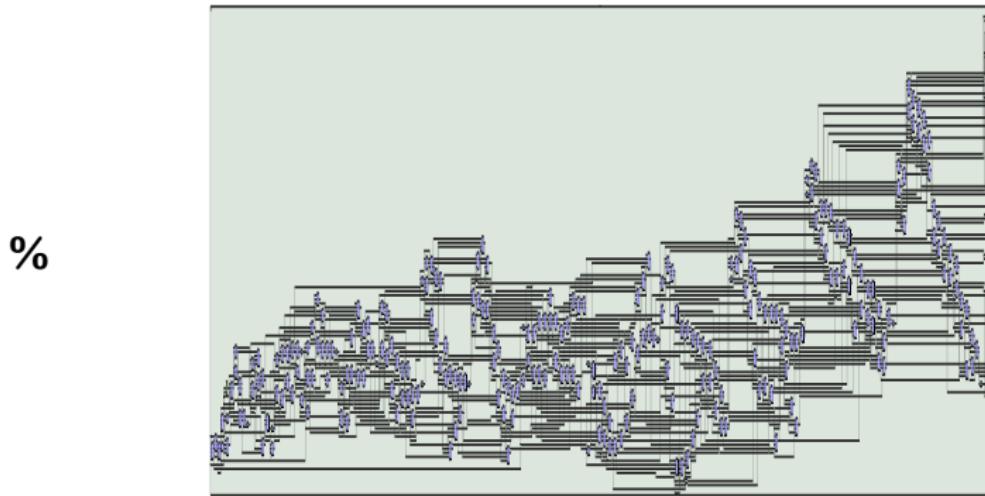
- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?

/



Coding in Hardware cont'd

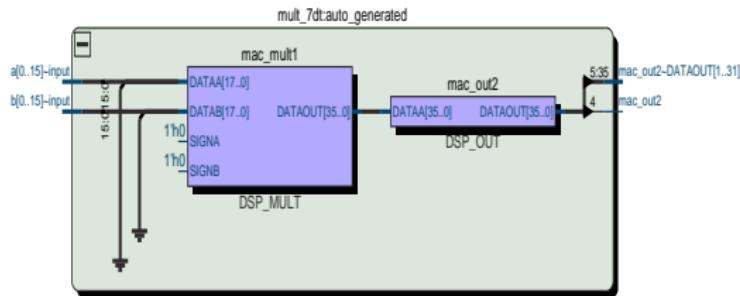
- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?



Coding in Hardware cont'd

- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?

*



Coding in Hardware cont'd

- different way of thinking required
 - how many resources are absolutely necessary?
 - how much hardware is generated by code?

operator	# LUTs on FPGA
AND	16
+	16
/	291
%	300
*	0

Software vs. Hardware

Hardware

Software

execution	concurrent	sequential
instructions increase	generated hardware	run time
limiting factor	available area	execution time
focus on	resulting hardware	computational complexity

Summary

Hardware

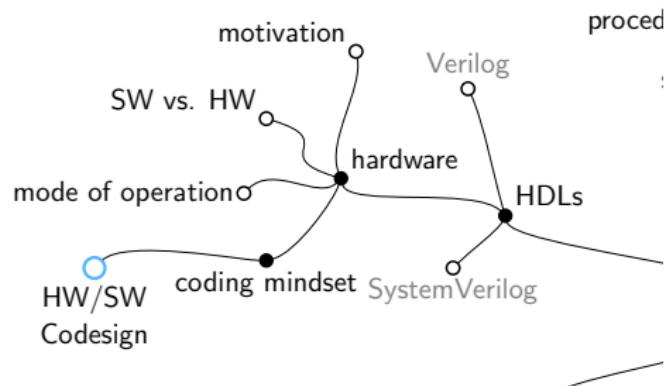
- required in various fields of application
- fundamentally different mode of operation
- mindset for coding has to be adapted

(Dis)Advantages

- highly optimized solutions possible
 - speed, throughput, latency, power, ...
- high development costs / time

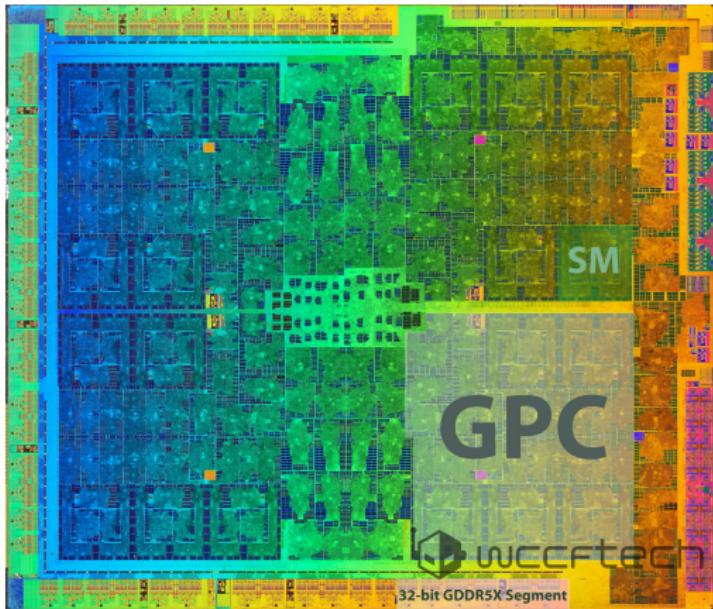
Hardware Design

Hardware Design Languages (HDL)



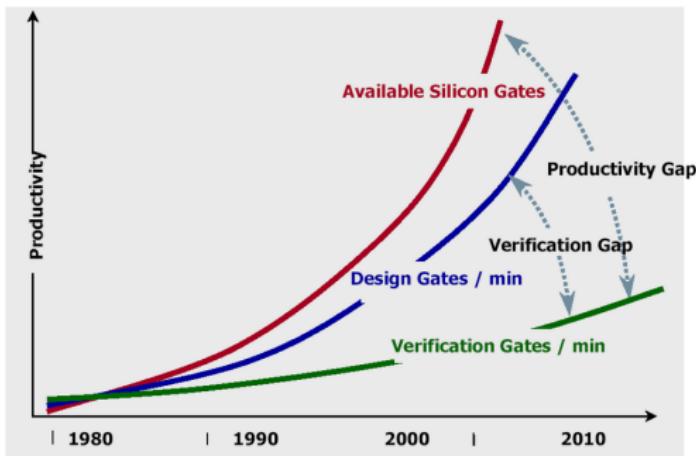
Circuit Complexity

- shrinking feature size allows more transistors on same area
- already several billions on a single chip
- time to market decreases



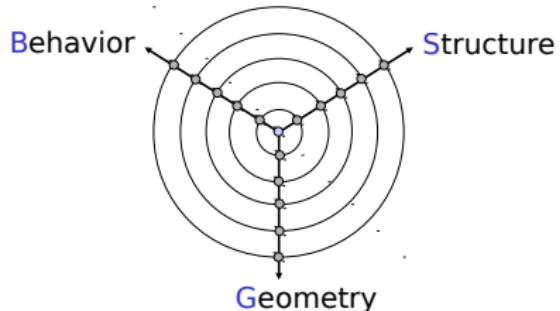
Design Productivity Gap

- code generated by humans rises slower than complexity
- however even shorter development time desired
- higher abstraction required \Rightarrow Hardware Design Languages



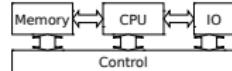
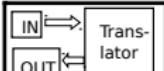
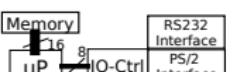
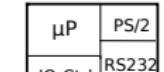
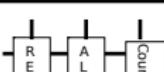
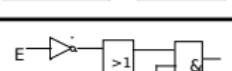
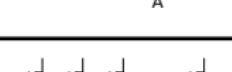
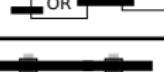
Hardware Design

1. System
2. Algorithmic
3. Register Transfer (RTL)
4. Gate
5. Transistor

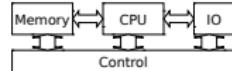
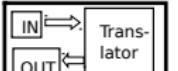
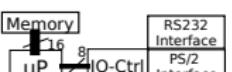
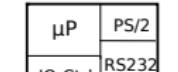
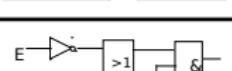
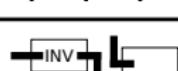
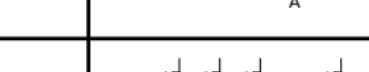
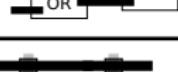


- all levels describe the same design
- higher levels have
 - more expressivity
 - less detail
- available for each view - circles in Y-diagram

Hardware Design cont'd

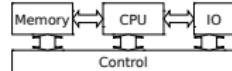
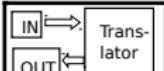
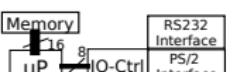
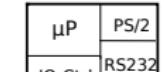
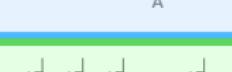
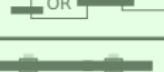
	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calulate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

Hardware Design cont'd

	Behavior	synthesis → Structure	place & route → Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calulate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

analysis ←

Hardware Design cont'd

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calulate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

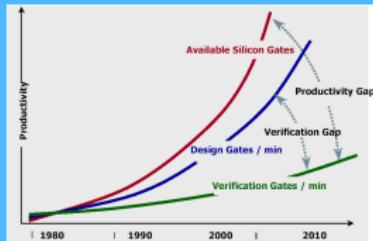
Tools

Hardware Design cont'd

- design from different angles and levels possible (Y-diagram)
- structural languages available but not well suited
 - hard to change design
- structural / behavioral description on RTL level desired for hardware design
- tools then convert it to structure (synthesis) and geometry (place & route)
- geometry and lowest levels completely handled by tools

Motivation for HDLs

Abstraction



Documentation



Communication



Motivation for HDLs cont'd

- allow higher abstraction
 - mask complexity
 - improves time-to-market
 - close design-productivity gap
- serve as documentation
 - initial purpose accurate description
 - readable by human and machine
 - improves reusability
- used for communication
 - natural language not suited well
 - to customers, other developers, ...

Hardware Design Languages

- multiple languages available
 - VHDL, Verilog, SystemC, SystemVerilog, ...
- VHDL and Verilog most popular

VHDL

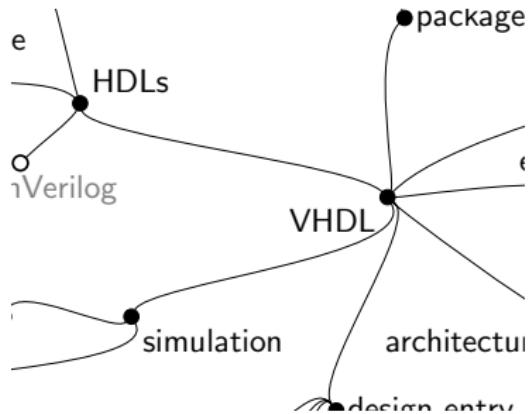
- Ada/Pascal
- strongly typed
- very stringent
- if runs likely to work
- at universities

Verilog

- C
- not that strongly typed
- easier to make hard to detect errors
- closely related to SystemVerilog
- at companies

VHDL

Introduction



Introduction

- VHDL (Very High Speed Integrated Circuit Hardware Description Language)
 - very popular all around the globe
- developed in the 80's for U.S. Department of Defense
 - based on Ada (strongly typed concept)
- initially solely used to document hardware
 - later extended by synthesis tools
 - essentially code generators
 - only subset of commands can be transferred to hardware

Introduction cont'd

- revisions 1987, 93, 2000 and 2002
- latest **VHDL Standard 2008** taught in this lecture
 - Attention: not all new concepts included in tools yet
 - has to be explicitly selected in tools

- watch out for  X ... page number

- lots of online resources available
 - tutorials, books, tools, ...

Language Properties

- case **insensitive**
 - variable = VARIABLE = VaRiAbLe
- commands terminated by ;'
- '--' single line comment
 - since 2008 multi line comments possible ('/*' and '*/')
- format used in lecture
 - **keywords**
 - DATATYPE
 - *comment*
 - everything else

Identifiers

229

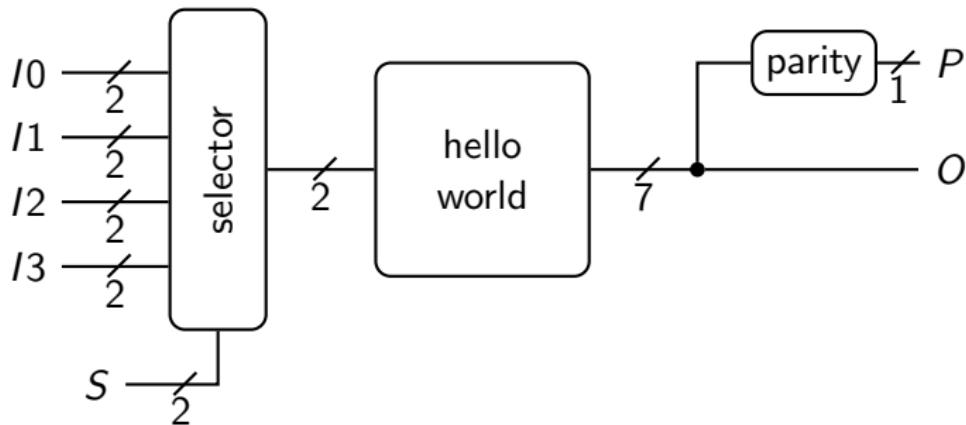
identifier := letter { [underline] letter_or_digit }

- first character must be letter
- underline not at end
- no two consecutive underlines

valid
in
in1
in10
in2_0

invalid
_in
0in
in_-
in__0

First Design – Hello World!

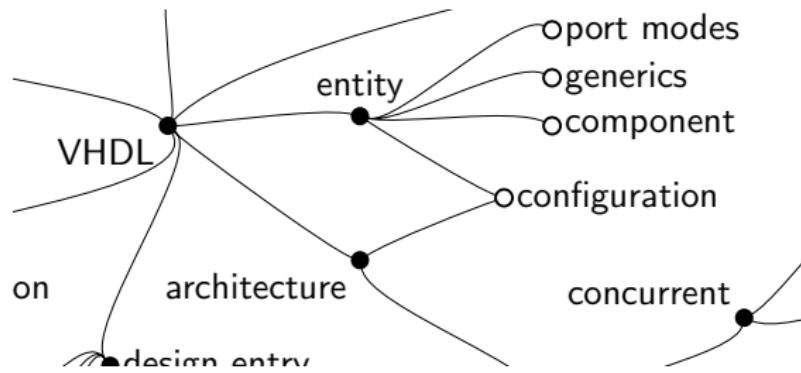


First Design – Hello World cont'd

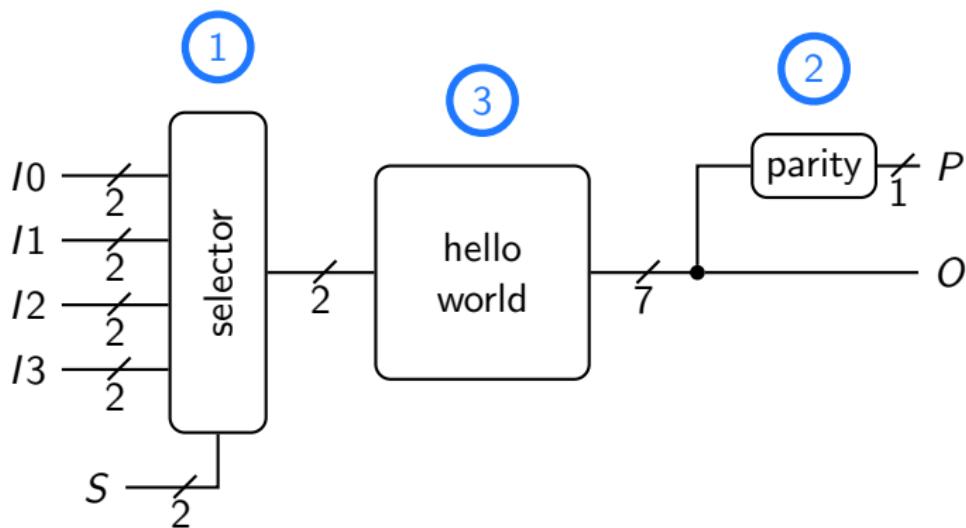
- show “HELLO” on 7-segment display one character at a time
 - 00 halt
 - 10 previous letter
 - 01 next letter
 - 11 not considered
- 8 Bit output
 - O : 7 Bit to display
 - P : parity Bit
- 4 control inputs $I_0 - I_3$
 - selection of active one by signal S

VHDL

Entity & Architecture

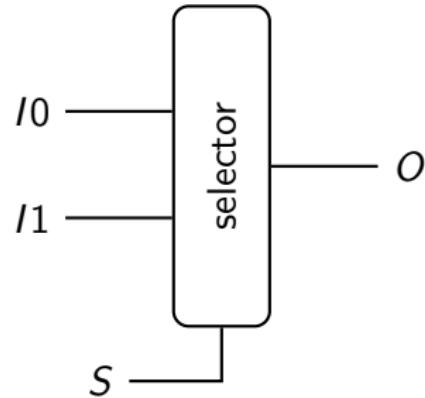


First Design – Hello World!



Selector

- propagate I_0 / I_1 based on S
 - consider data width of 1 Bit for now
- interface
 - 3 inputs (I_0 , I_1 , S)
 - 1 output (O)
- behavior
 - if $S = 0 \rightarrow O = I_0$
 - if $S = 1 \rightarrow O = I_1$



Compare to Human

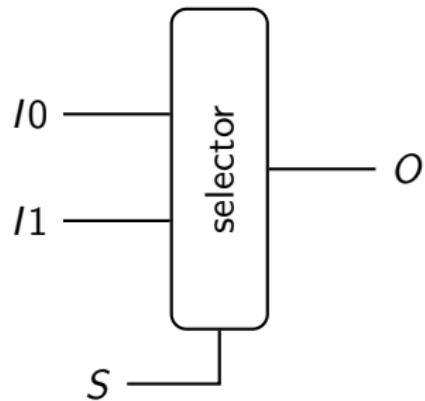
- how to characterize a person
- appearance
 - color of eyes / hair, size, ...
 - no knowledge of person required
 - in *VHDL* defined by **entity**
- behavior
 - hobbies, likes / dislikes, ...
 - knowledge of person required
 - in *VHDL* defined by **architecture**



Entity

7

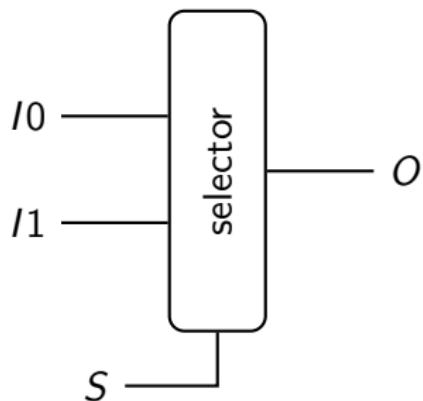
```
1 entity selector is
2 port
3 (
4     I0, I1, S: in STD_LOGIC;
5     O : out STD_LOGIC
6 );
7 end entity;
```



Entity

7

```
entity selector is
port
(
    I0, I1, S: in STD_LOGIC;
    O : out STD_LOGIC
);
end entity;
```

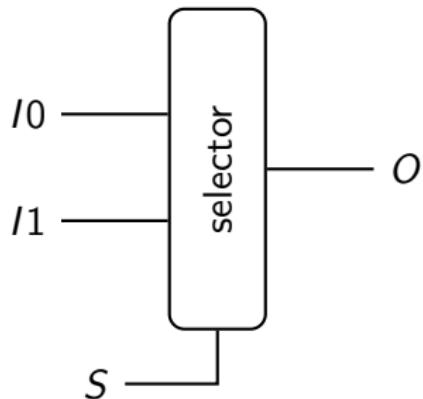


Entity

7

```
1 entity selector is
2 port
3 (
4   I0, I1, S in STD_LOGIC;
5   O: out STD_LOGIC
6 );
7 end entity;
```

port names

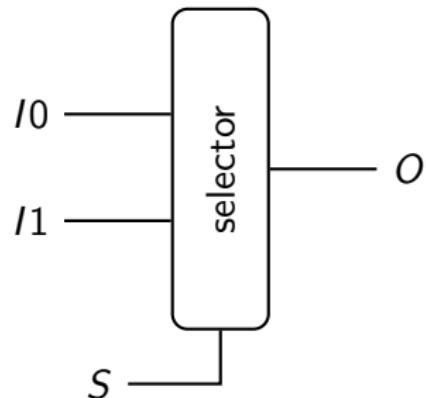


Entity

7

```
1 entity selector is
2 port
3 (
4     I0, I1, S: in STD_LOGIC;
5     O : out STD_LOGIC
6 );
7 end entity;
```

port modes

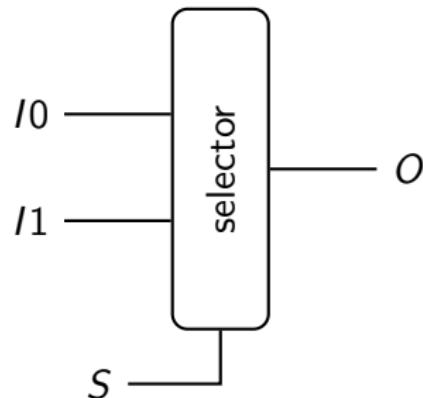


Entity

7

```
1 entity selector is
2 port
3 (
4     I0, I1, S: in STD_LOGIC;
5     O : out STD_LOGIC
6 );
7 end entity;
```

data types



Port Modes

73

- define direction of a port
 - port ... signal connecting inside of **entity** to outside
- different types available
 - **in** (input) and **out** (output) commonly used
 - **inout**: written and read from in- and outside
 - **buffer**: written and read from inside / only read from outside

	read	write	comment
in	x		
out	(x)	x	readable with restrictions
inout / buffer	x	x	very few restrictions
linkage	x	x	

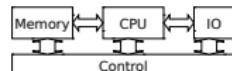
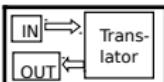
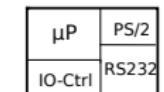
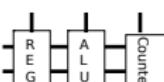
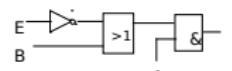
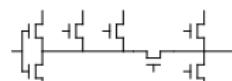
Data Type STD_LOGIC

- nine valued logic
 - '0' and '1' classical values
- allows modeling of various situations
 - 'Z' ... bus access
 - 'H' ... pull up
 - 'L' ... pull down

values

- '0': Strong low
- '1': Strong high
- 'X': Strong unknown
- '-' : Don't care
- 'Z': High impedance
- 'W': Weak unknown
- 'L': Weak low
- 'H': Weak high
- 'U': Uninitialized

Architecture

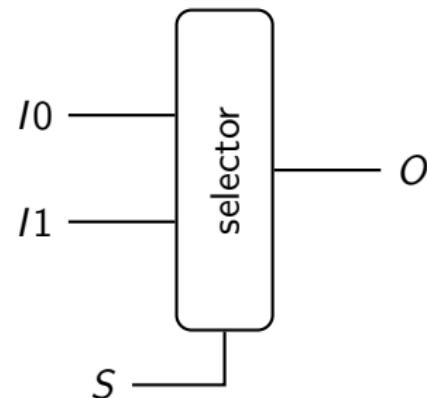
	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

Architecture cont'd

- models actions of **entity**
 - what are the outputs for given inputs in a certain state
 - asynchronous / synchronous
- different descriptions possible
 - behavioral and structural view
 - RTL and logic level
- based on **architecture** hardware is generated
- can be defined in
 - same file as **entity**
 - separate one

Selector Architecture

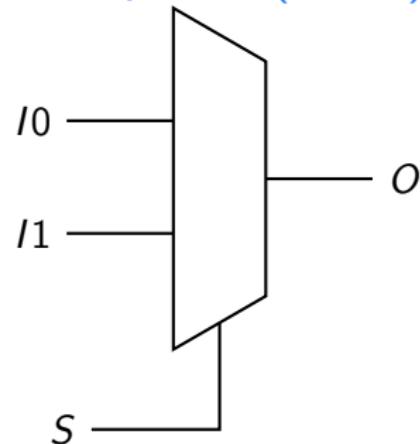
- implement our selector unit
 - behavioral view
 - first logic then RTL level
- for a start
 - reduce data width to 1 Bit
 - only use two input signals
- basic circuit that implements desired behavior
 - What is it called?
 - What is its gate-level representation?



Selector Architecture

- implement our selector unit
 - behavioral view
 - first logic then RTL level
- for a start
 - reduce data width to 1 Bit
 - only use two input signals
- basic circuit that implements desired behavior
 - What is it called?
 - What is its gate-level representation?

Multiplexer (MUX)



Selector Architecture

- implement our selector unit
 - behavioral view
 - first logic then RTL level
- for a start
 - reduce data width to 1 Bit
 - only use two input signals
- basic circuit that implements desired behavior
 - What is it called?
 - What is its gate-level representation?

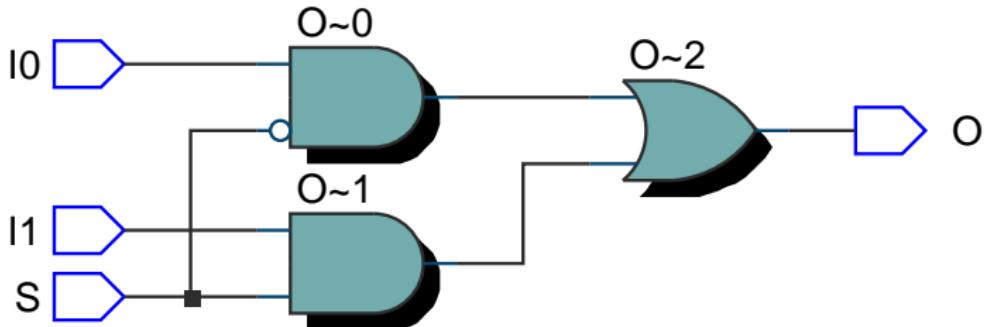
```
22 entity MUX21 is
23 port(
24   I0, I1, S : in std_logic;
25   O : out std_logic
26 );
```

Selector Architecture

- implement our selector unit
 - behavioral view
 - first logic then RTL level
- for a start
 - reduce data width to 1 Bit
 - only use two input signals
- basic circuit that implements desired behavior
 - What is it called?
 - What is its gate-level representation?

$$O = (I0 \wedge (\neg S)) \vee (I1 \wedge S)$$

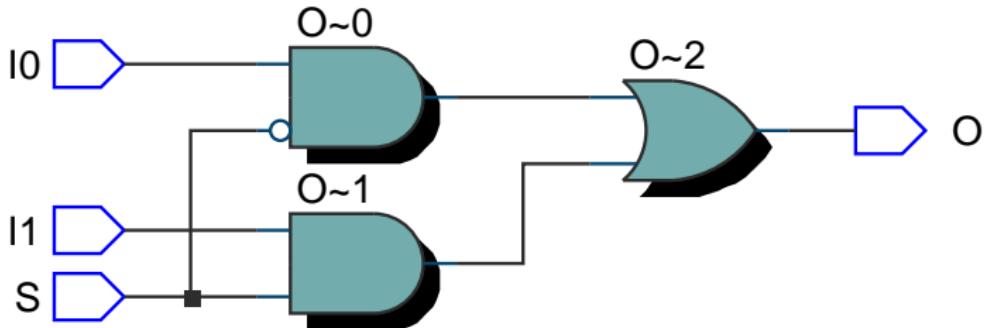
MUX21 Gate Level



```
29 architecture concurrent of MUX21 is
30 begin
31     O <= (I0 and (not S)) or (I1 and S);
32 end architecture;
```

[eaMUX21_concurrent.vhd](#)

MUX21 Gate Level

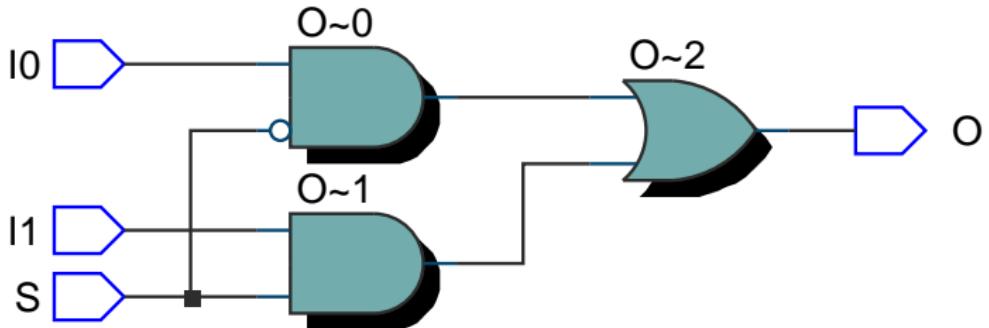


architecture name

```
29 architecture concurrent of MUX21 is
30 begin
31     O <= (I0 and (not S)) or (I1 and S);
32 end architecture;
```

eaMUX21_concurrent.vhd

MUX21 Gate Level

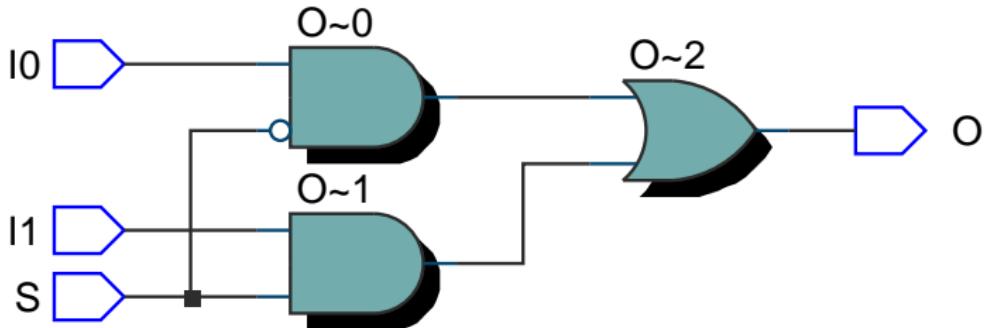


entity name

```
29 architecture concurrent of MUX21 is
30 begin
31     O <= (I0 and (not S)) or (I1 and S);
32 end architecture;
```

eaMUX21_concurrent.vhd

MUX21 Gate Level



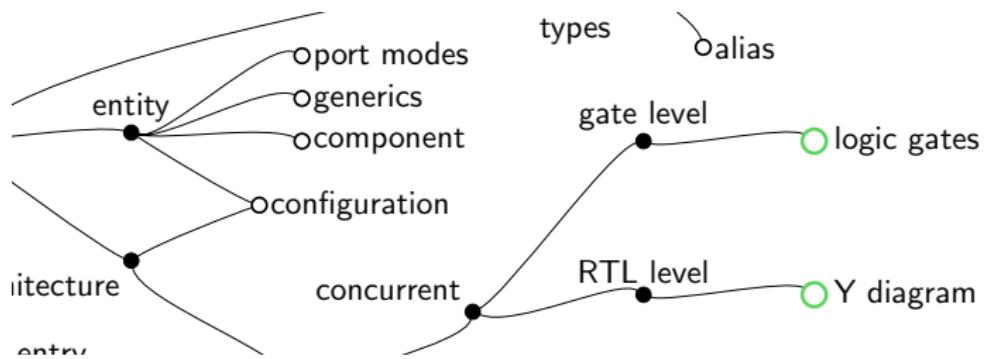
```
29 architecture concurrent of MUX21 is
30 begin
31     O <= (I0 and (not S)) or (I1 and S);
32 end architecture;
                                         assignment
                                         eaMUX21_concurrent.vhd
```

Quartus

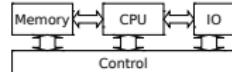
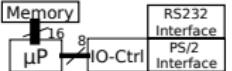
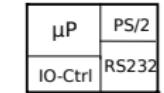
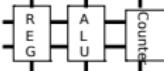
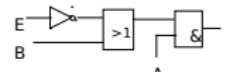
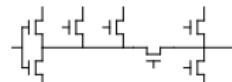
- written code can now be fed to tools
 - in our case Quartus
 - do not forget to activate standard 2008
- translates
 - behavioral description to structural (synthesis)
 - structural description to geometric (place & route)

VHDL

Concurrent Design Style



Repetition

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

Parallel Assignments

- assignment operator ' $<=$ '
- complicated problem can be split in simpler terms
 - execution in parallel, order not important

```
29  architecture concurrent of MUX21 is
30  begin
31      O <= (I0 and (not S)) or (I1 and S);
32  end architecture;
```

[eaMUX21.concurrent.vhd](#)

Parallel Assignments

- assignment operator ' $<=$ '
- complicated problem can be split in simpler terms
 - execution in parallel, order not important

```
1 architecture concurrent of MUX21 is
2   begin
3     O <= (I0 and (not S));
4     O <= (I1 and S);
5   end architecture concurrent;
```

Parallel Assignments

- assignment operator ' $<=$ '
- complicated problem can be split in simpler terms
 - execution in parallel, order not important
- WARNING: no multiple drivers allowed

```
1 architecture concurrent of MUX21 is
2 begin
3   O <= (I0 and (not S));
4   O <= (I1 and S);
5 end architecture concurrent;
```

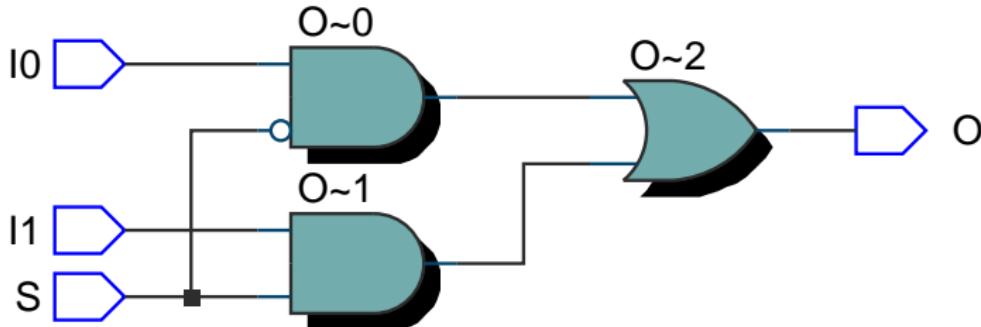
Parallel Assignments

- assignment operator ' $<=$ '
- complicated problem can be split in simpler terms
 - execution in parallel, order not important
- WARNING: no multiple drivers allowed
- add internal wire (**signal**)

```
29  architecture concurrent_sig of MUX is
30    signal A0, A1 : STD_LOGIC;
31  begin
32    A0 <= (I0 and (not S));
33    A1 <= (I1 and S);
34    O <= A0 or A1;
35  end architecture;
```

eaMUX21_concurrent_signals.vhd

Parallel Assignments



```
29 architecture concurrent_sig of MUX is
30   signal A0, A1 : STD_LOGIC;
31 begin
32   A0 <= (I0 and (not S));
33   A1 <= (I1 and S);
34   O <= A0 or A1;
35 end architecture;
```

eaMUX21_concurrent_signals.vhd

Are we done?

We can already build every combinational circuit!

Are we done?

We can already build every combinational circuit!

problem

- very high complexity
- very low abstraction level (logic level)
- does not close design productivity gap

Are we done?

We can already build every combinational circuit!

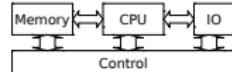
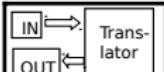
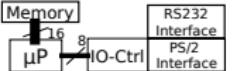
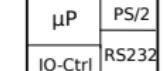
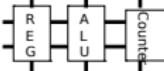
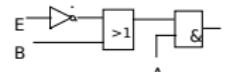
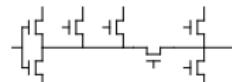
problem

- very high complexity
- very low abstraction level (logic level)
- does not close design productivity gap

solution

- design on Register Transfer Level (RTL)
 - Reminder: Y-Diagram

Y Diagram

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dl}{dt} + \frac{l}{C} + L \frac{d^2l}{dt^2}$		

Selected Signal Assignments

- corresponds to case statement in C
- execution still concurrent
- all possibilities have to be covered
 - always use **others** at end
 - otherwise incomplete specification (cp. STD_LOGIC)
 - leads to memory (latch) here

```
29  architecture selected of MUX21 is
30  begin
31      with S select
32          O <= I0 when '0',
33              I1 when others;
34  end architecture;
```

eaMUX21_selected.vhd

Incomplete Specification

Assume the following task description at your math exam

Calculate y for $x = 100$ where $y = 2 \cdot x$ if $x < 10$.

What are you going to do?

1. ask lecturer
 - what if noone is available (tool can not ask)
2. do nothing
 - not really an option
3. do what you like best
 - $y = x$, $y = 0$, ...
 - that is exactly what tools do (e.g. make memory)

Conditional Signal Assignments

- corresponds to if-else chain in C
 - choose first branch that evaluates to true
- very similar to selected signal assignment
 - more freedom regarding expression to check
- all possibilities should be covered
 - always use final **else**
 - not mandatory but behavior arbitrary otherwise

```
29  architecture conditional of MUX21 is
30  begin
31      O <= I0 when S='0' else
32          I1 ;
33  end architecture;
```

eaMUX21_conditional.vhd

Conditional vs. Selected Assignment

- for simple 2-to-1 MUX no difference
 - let's extend it to 4-to-1
- how does the entity change?

```
1 entity MUX21 is
2 port
3 (
4     I0, I1, S: in STD_LOGIC;
5     O : out STD_LOGIC
6 );
7 end entity;
```

Conditional vs. Selected Assignment

- for simple 2-to-1 MUX no difference
 - let's extend it to 4-to-1
- how does the entity change?
- use two wires for *S*, combine in single **signal** array
 - called STD_LOGIC_VECTOR
 - for size use either **to** or **downto**

```
1 entity MUX41 is
2 port (
3     I0, I1, I2, I3: in STD_LOGIC;
4     S0, S1 : in STD_LOGIC;
5     O : out STD_LOGIC
6 );
7 end entity;
```

Conditional vs. Selected Assignment

- for simple 2-to-1 MUX no difference
 - let's extend it to 4-to-1
- how does the entity change?
- use two wires for *S*, combine in single **signal** array
 - called STD_LOGIC_VECTOR
 - for size use either **to** or **downto**

```
1 entity MUX41 is
2 port (
3     I0, I1, I2, I3: in STD_LOGIC;
4     S : in STD_LOGIC_VECTOR(1 downto 0);
5     O : out STD_LOGIC
6 );
7 end entity;
```

4-to-1 Multiplexer

- logic level description already infeasible

```
O <= (I0 and ( (not S(1)) and (not S(0)) ) ) or ...
```

- RTL level similar to 2-to-1 MUX
 - more cases have to be checked
 - use double quotes for vectors

conditional

```
architecture conditional of MUX41 is
begin
    O <= I0 when S="00" else
        I1 when S="01" else
        I2 when S="10" else
        I3;
end architecture;
```

[eaMUX41_conditional.vhd](#)

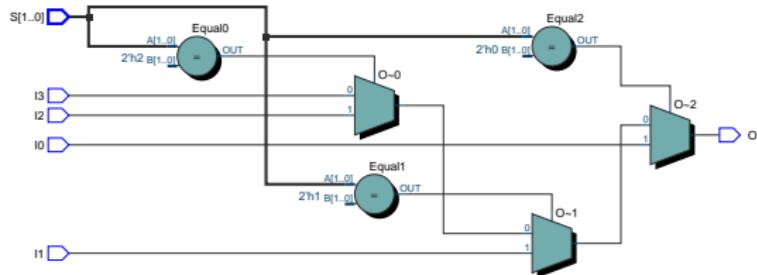
selected

```
architecture selected of MUX41 is
begin
    with S select
        O <= I0 when "00",
        I1 when "01",
        I2 when "10",
        I3 when others;
end architecture;
```

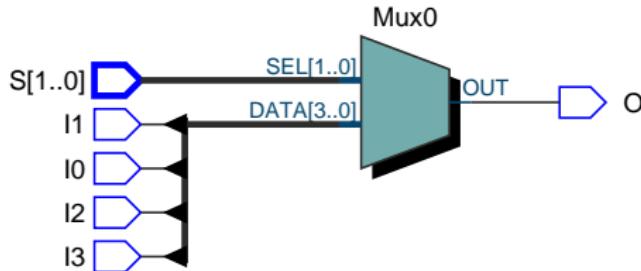
[eaMUX41_selected.vhd](#)

4-to-1 Multiplexer cont'd

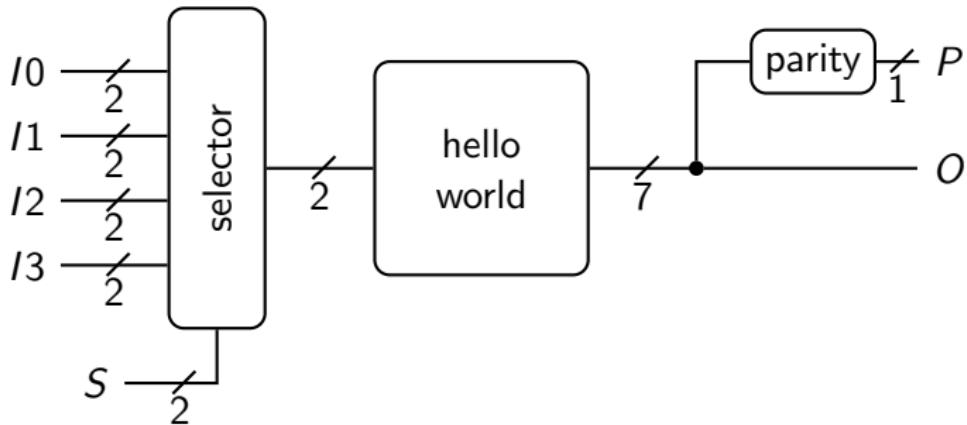
selected



conditional



Lookback



2 Bit Input Width

- recall hello world application
 - 2 Bit inputs to our selector
- What has to be changed to account for this change?

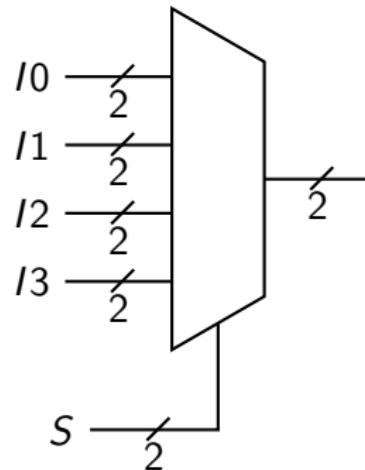
2 Bit Input Width

- recall hello world application
 - 2 Bit inputs to our selector
- What has to be changed to account for this change?
 - only the **entity**!

```
1 entity MUX41 is
2 port (I0, I1, I2, I3: in STD_LOGIC_VECTOR(1 downto 0);
3     S : in STD_LOGIC_VECTOR(1 downto 0);
4     O : out STD_LOGIC_VECTOR(1 downto 0));
5 end entity;
```

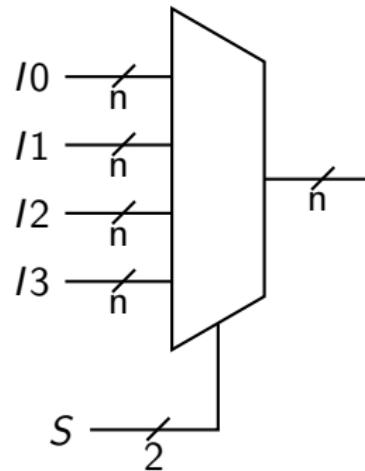
Variable Input Width

- 2-Bit data width done
- how about width of 3, 4, ...
 - **entity** has to be changed again
- managing each separately cumbersome
 - separate entity for each data width required
 - very bad maintainability
- parameterization desirable
 - create single unit
 - define data width by parameter



Variable Input Width

- 2-Bit data width done
- how about width of 3, 4, ...
 - **entity** has to be changed again
- managing each separately cumbersome
 - separate entity for each data width required
 - very bad maintainability
- parameterization desirable
 - create single unit
 - define data width by parameter



Generics

- allow configuration of **entity**
 - can be adapted to needs
- **generic** defines parameters
 - data type has to be specified (in this case NATURAL)
 - ':=' sets default value (optional)

```
22 entity MUX41 is
23   generic ( width : NATURAL := 2);
24   port (I0, I1, I2, I3: in STD_LOGIC_VECTOR(width-1 downto 0);
25     S : in STD_LOGIC_VECTOR(1 downto 0);
26     O : out STD_LOGIC_VECTOR(width-1 downto 0));
27 end entity;
```

[eaMUX41_selected_generic.vhd](#)

Generics

- allow configuration of **entity**
 - can be adapted to needs
- generic** defines parameters
 - data type has to be specified (in this case NATURAL)
 - ':=' sets default value (optional)

```
22 entity MUX41 is
23 generic (width : NATURAL := 2);
24 port (I0, I1, I2, I3: in STD_LOGIC_VECTOR(width-1 downto 0);
25      S : in STD_LOGIC_VECTOR(1 downto 0);
26      O : out STD_LOGIC_VECTOR(width-1 downto 0));
27 end entity;
```

eaMUX41_selected_generic.vhd

(Complete) Entity Definition

7

```
1 entity identifier is
2   [formal_generic_clause]
3   [formal_port_clause]
4   {entity_declarative_item}
5   [begin entity_statement_part]
6   end [entity] [identifier];
```

(Complete) Architecture Definition

10

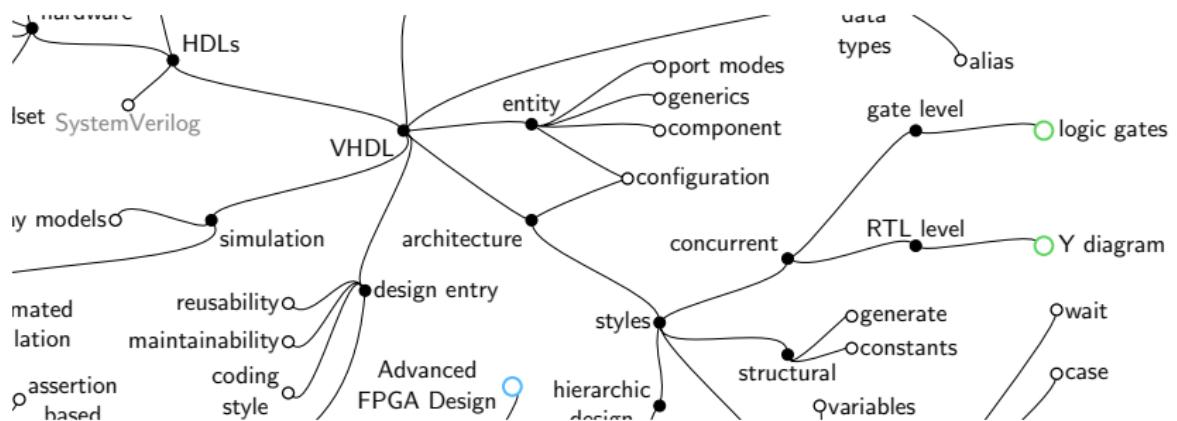
```
1 architecture identifier of entity_name is
2   {block_declarative_item}
3 begin
4   {concurrent_statement}
5 end [architecture] [identifier];
```

Summary

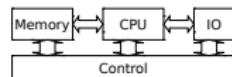
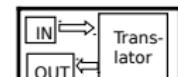
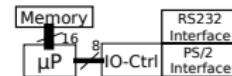
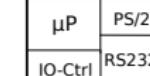
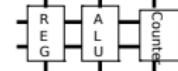
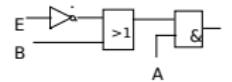
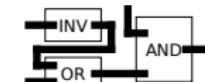
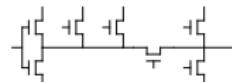
- statements executed in parallel
 - statement order not important
 - multiple drivers not allowed
- logic and RTL level supported
- logic level
 - directly specify the desired gates
- RTL level
 - conditional / selected assignments

VHDL

Structural Design Style



Y diagram

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dl}{dt} + \frac{l}{C} + L \frac{d^2l}{dt^2}$		

Structural Design Style

- previously behavioral view used
 - how does the **entity** react to inputs
- structural view also very important
 - wire already defined **entities** together
 - compare connecting SMD blocks
- create complex circuits from basic blocks
- also mandatory for testing

Testing

- until now only syntactical checks done
- but: does the design fulfill the task we want it to?
 - run module in simulation and check results
- various flavors available
 - providing input and analyse output
 - run against reference and compare results
 - read/write from/to files
- more on that follows later in the lecture

Testbench

- testing carried out in special **entity** called testbench
 - has to model outside world completely (no in- and outputs)
 - very minimalistic
- unit under test (UUT) has to be called
 - component declaration resp. instantiation
 - configuration
- to test UUT testbench has to
 - provide input stimuli to the UUT
 - check outputs regarding correctness

```
22  entity testbench is  
23  end;
```

[simulation.l/testbench.vhd](#)

Component Declaration

- **component** defines the interface of the **entity** to import
 - port names may differ from **entity**
 - good practice to choose the same
- different locations possible
 - packages (similar to header files, more on that later)
 - *block_declarative_item* in the architecture
- cp. to socket definition when designing a circuit board using surface mounted devices

```
31      component MUX41 is
32          generic ( width : NATURAL );
33          port (I0, I1, I2, I3: in STD_LOGIC_VECTOR(width-1 downto 0);
34                  S : in STD_LOGIC_VECTOR(1 downto 0);
35                  O : out STD_LOGIC_VECTOR(width-1 downto 0));
36      end component;
```

[simulation_1/testbench.vhd](#)

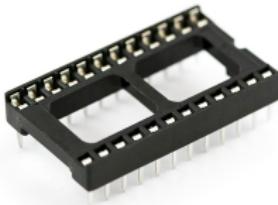
Component Declaration

- **component** defines the interface of the **entity** to import
 - port names may differ from **entity**
 - good practice to choose the same
- different locations possible
 - packages (similar to header files, more on that later)
 - *block_declarative_item* in the architecture
- cp. to socket definition when designing a circuit board using surface mounted devices

```
1  component identifier [ is ]
2    [ local_generic_clause ]
3    [ local_port_clause ]
4  end component [ identifier ];
```

Component Declaration

- **component** defines the interface of the **entity** to import
 - port names may differ from **entity**
 - good practice to choose the same
- different locations possible
 - packages (similar to header files, more on that later)
 - *block_declarative_item* in the architecture
- cp. to socket definition when designing a circuit board using surface mounted devices



sparkfun.com

Component Instantiation

- instantiate **component** in testbench
- connect in-/outputs to internal signals of testbench
- positional vs. named mapping
 - better to use names
- cp. to soldering socket to board

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  begin
5  ...
6
7  UUT: MUX41
8  generic map(2)
9  port map (TB_I0,
10           TB_I1,
11           TB_I2,
12           TB_I3,
13           TB_S,
14           TB_O);
15
16  ...
```

Component Instantiation

- instantiate **component** in testbench
- connect in-/outputs to internal signals of testbench
- positional vs. named mapping
 - better to use names
- cp. to soldering socket to board

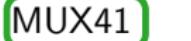
```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  begin
5  ...
6
7  UUT MUX41
8  generic map(2)
9  port map (TB_I0,
10           TB_I1,
11           TB_I2,
12           TB_I3,
13           TB_S,
14           TB_O);
15
16  ...
```



Component Instantiation

- instantiate **component** in testbench
- connect in-/outputs to internal signals of testbench
- positional vs. named mapping
 - better to use names
- cp. to soldering socket to board

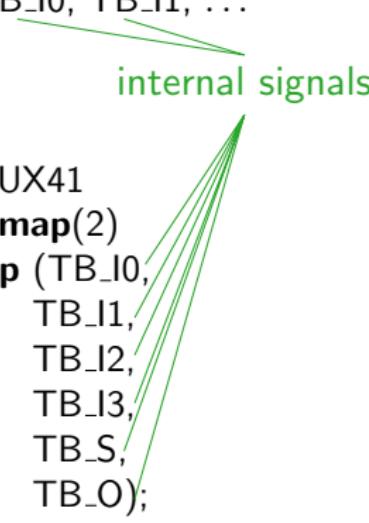
```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  begin
5      ...
6
7      UUT: MUX41
8      generic map(2)
9      port map (TB_I0,
10          TB_I1,
11          TB_I2,
12          TB_I3,
13          TB_S,
14          TB_O);
15
16      ...
```



Component Instantiation

- instantiate **component** in testbench
- connect in-/outputs to internal signals of testbench
- positional vs. named mapping
 - better to use names
- cp. to soldering socket to board

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  begin
5  ...
6
7  UUT: MUX41
8  generic map(2)
9  port map (TB_I0,
10   TB_I1,
11   TB_I2,
12   TB_I3,
13   TB_S,
14   TB_O);
15
16  ...
```



internal signals

Component Instantiation

- instantiate **component** in testbench
- connect in-/outputs to internal signals of testbench
- positional vs. named mapping
 - better to use names
- cp. to soldering socket to board

```
1  architecture ...
2  constant width ...
3  signal TB_I0, TB_I1, ...
4  begin
5  ...
6
7  UUT: MUX41
8  generic map(width => 2)
9  port map(I0 => TB_I0,
10           I1 => TB_I1,
11           I2 => TB_I2,
12           I3 => TB_I3,
13           S => TB_S,
14           O => TB_O);
15
16  ...
```

Component Instantiation

- instantiate **component** in testbench
- connect in-/outputs to internal signals of testbench
- positional vs. named mapping
 - better to use names
- cp. to soldering socket to board



sparkfun.com

Constants

- width defined for each MUX
 - hard to change
 - bad maintainability
- better to use constant values for generics
 - can not be altered afterwards
- defined by **constant**
 - provide name and type
 - ':=' assigns value

```
1  architecture ...
2  constant width ...
3  signal TB_I0, TB_I1, ...
4  begin
5  ...
6
7  UUT: MUX41
8  generic map(width => 2)
9  port map(I0 => TB_I0,
10           I1 => TB_I1,
11           I2 => TB_I2,
12           I3 => TB_I3,
13           S => TB_S,
14           O => TB_O);
15
16  ...
```

Constants

- width defined for each MUX
 - hard to change
 - bad maintainability
- better to use constant values for generics
 - can not be altered afterwards
- defined by **constant**
 - provide name and type
 - ':=' assigns value

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width : NATURAL := 2;
5  begin
6
7      ...
8
9  UUT: MUX41
10 generic map(width => width)
11 port map(I0 => TB_I0,
12           I1 => TB_I1,
13           I2 => TB_I2,
14           I3 => TB_I3,
15           S => TB_S,
16           O => TB_O);
17     ...
```

Constants

- width defined for each MUX
 - hard to change
 - bad maintainability
- better to use constant values for generics
 - can not be altered afterwards
- defined by **constant**
 - provide name and type
 - ':=' assigns value

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width: NATURAL := 2;
5  begin
6
7  ...
8
9  UUT: MUX41
10 generic map(width => width)
11 port map(I0 => TB_I0,
12           I1 => TB_I1,
13           I2 => TB_I2,
14           I3 => TB_I3,
15           S => TB_S,
16           O => TB_O);
17 ...
```

Constants

- width defined for each MUX
 - hard to change
 - bad maintainability
- better to use constant values for generics
 - can not be altered afterwards
- defined by **constant**
 - provide name and type
 - ':=' assigns value

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width : NATURAL := 2;
5  begin
6      data type
7      ...
8
9  UUT: MUX41
10 generic map(width => width)
11 port map(I0 => TB_I0,
12           I1 => TB_I1,
13           I2 => TB_I2,
14           I3 => TB_I3,
15           S => TB_S,
16           O => TB_O);
17   ...
```

Configurations

- **configuration** determines entity and architecture to use
 - only interface determined by **component**
- recall multiplexer implementation
 - multiple architectures for single entity
- can be neglected if entity / architecture uniquely defined
- compare to placing chip in soldered socket



Configurations

- **configuration** determines entity and architecture to use
 - only interface determined by **component**
- recall multiplexer implementation
 - multiple architectures for single entity
- can be neglected if entity / architecture uniquely defined
- compare to placing chip in soldered socket



sparkfun.com

sparkfun.com

Configurations cont'd

can be placed at different locations

1. at instantiation
 - bad maintainability
 - not recommended
2. in architecture header
3. separately outside architecture
 - can become very complex
 - not explained in detail here

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width ...
5  begin
6  ...
7
8  UUT: entity work.MUX41 (selected)
9  generic map(width)
10 port map(I0 => TB_I0,
11           I1 => TB_I1,
12           I2 => TB_I2,
13           I3 => TB_I3,
14           S => TB_S,
15           O => TB_O);
16
17 ...
```

Configurations cont'd

can be placed at different locations

1. at instantiation
 - bad maintainability
 - not recommended
2. in architecture header
3. separately outside architecture
 - can become very complex
 - not explained in detail here

```

1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width ...
5  begin
6  ...
7
8  UUT: entity work.MUX41(selected)
9  generic map(width)
10 port map(I0 => TB_I0,
11           I1 => TB_I1,
12           I2 => TB_I2,
13           I3 => TB_I3,
14           S => TB_S,
15           O => TB_O);
16
17 ...

```



entity name

Configurations cont'd

can be placed at different locations

1. at instantiation
 - bad maintainability
 - not recommended
2. in architecture header
3. separately outside architecture
 - can become very complex
 - not explained in detail here

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width ...
5  begin
6      ...          architecture name
7
8      UUT: entity work.MUX41 (selected)
9      generic map(width)
10     port map(I0 => TB_I0,
11                 I1 => TB_I1,
12                 I2 => TB_I2,
13                 I3 => TB_I3,
14                 S => TB_S,
15                 O => TB_O);
16
17     ...
```

Configurations cont'd

can be placed at different locations

1. at instantiation
 - bad maintainability
 - not recommended
2. in architecture header
3. separately outside architecture
 - can become very complex
 - not explained in detail here

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width ...
5  for UUT:MUX41 use entity work.MUX41
6      (selected);
7  begin
8      ...
9      UUT: MUX41
10     generic map(width)
11     port map(I0 => TB_I0,
12                 I1 => TB_I1,
13                 I2 => TB_I2,
14                 I3 => TB_I3,
15                 S => TB_S,
16                 O => TB_O);
17     ...
```

Configurations cont'd

can be placed at different locations

1. at instantiation
 - bad maintainability
 - not recommended
2. in architecture header
3. separately outside architecture
 - can become very complex
 - not explained in detail here

```
1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width ...
5  for UUT:MUX41 use entity work.MUX41
6  (selected);
7  begin
8  ...
9  UUT: MUX41
10 generic map(width)
11 port map(I0 => TB_I0,
12           I1 => TB_I1,
13           I2 => TB_I2,
14           I3 => TB_I3,
15           S => TB_S,
16           O => TB_O);
17 ...
```

entity name

Configurations cont'd

can be placed at different locations

1. at instantiation
 - bad maintainability
 - not recommended
2. in architecture header
3. separately outside architecture
 - can become very complex
 - not explained in detail here

```

1  architecture ...
2  component ...
3  signal TB_I0, TB_I1, ...
4  constant width ...
5  for UUT:MUX41 use entity work.MUX41
6  selected);
7 begin
8   ...           architecture name
9   UUT: MUX41
10  generic map(width)
11  port map(I0 => TB_I0,
12            I1 => TB_I1,
13            I2 => TB_I2,
14            I3 => TB_I3,
15            S => TB_S,
16            O => TB_O);
17 ...

```

Configurations cont'd

```
1 configuration v1 of testbench is
2   for beh
3     for SEL : MUX41
4       use entity MUX41(selected);
5     end for;
6     for COND : MUX41
7       use entity MUX41(concurrent);
8     end for;
9   end for;
10 end configuration v1;
```

Configurations cont'd

```
1   configuration v1 of testbench is  
2     for beh           instance name  
3       for SEL : MUX41    — component name  
4         use entity MUX41(selected); — architecture name  
5       end for;  
6       for COND : MUX41    entity name  
7         use entity MUX41(concurrent);  
8       end for;  
9     end for;  
10    end configuration v1;
```

Summary

- imagine a circuit board that shall be equipped with socket mounted chips
- **component** declaration
 - Which sockets are required for the chips?
 - e.g. 10, 12 and 14 pin chips shall be mounted
- **component** instantiation
 - define interaction between single chips
 - connect pins of sockets by wires
- **configuration**
 - plug suitable chips into sockets

Formal, Local and Actual

- describe ports of unit at different location

Formal ports in entity

Local ports in component

Actual signals in architecture

- instantiating component

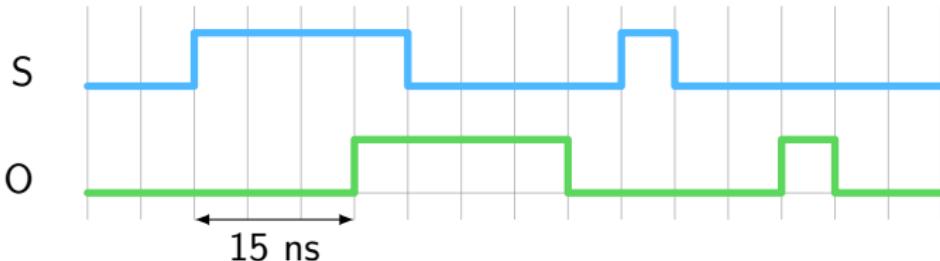
- connecting Local to Actual
- attach wire on board to socket

- configuring component

- connecting Formal to Local
- put chip in socket

Delay Models

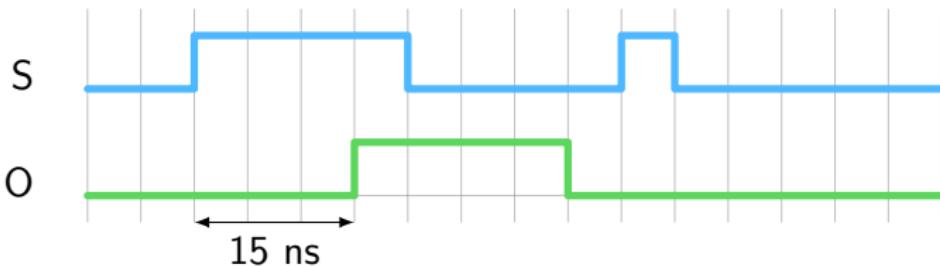
- signals on chip are not infinitely fast
 - approximately 2/3 the speed of light ($\approx 20 \text{ cm/ns}$)
- delays can not be synthesized (only for simulation)
- pure delay
 - no cancellation effects
 - used to model transmission lines
 - $O \leq \text{transport } S \text{ after } 15 \text{ ns};$



Delay Models

149

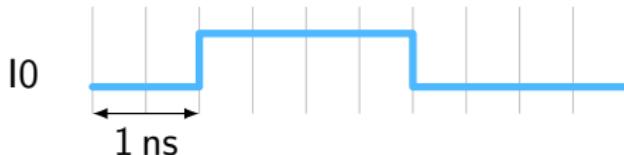
- inertial delay (default method)
 - short pulses are removed → model switching circuits
- $O \leq \text{reject } 10 \text{ ns}$ **inertial S after 15 ns**;
 - pulses shorter than **reject** (10 ns) are removed
- **inertial S after 15 ns**; → **reject 15 ns inertial S after 15 ns**;
 - if **reject** not specified first delay value taken
 - equivalent to **S after 15 ns**;
- more sophisticated ones also possible



Delay Models cont'd

- useful to generate signals in testbench
 - one of several methods (stay tuned)
 - applied to unit under test
- multiple values can be defined at once
 - delays for all counted from start

```
1 architecture
2 ...
3 signal I0 : STD_LOGIC;
4 begin
5   I0 <= transport '0', '1' after 1 ns, '0' after 3 ns;
6 ...
7 end architecture;
```



Complete Testbench

```
19 library IEEE;
20 use IEEE.std_logic_1164.all;
21
22 entity testbench is
23 end;
24
25 architecture beh of testbench is
26   constant width : NATURAL := 2;
27   signal TB_I0, TB_I1, TB_I2, TB_I3 : STD_LOGIC_VECTOR(width-1 downto 0);
28     signal TB_SEL, TB_COND : STD_LOGIC_VECTOR(width-1 downto 0);
29   signal TB_S : STD_LOGIC_VECTOR(1 downto 0);
30
31   component MUX41 is
32     generic ( width : NATURAL );
33     port (I0, I1, I2, I3: in STD_LOGIC_VECTOR(width-1 downto 0);
34           S : in STD_LOGIC_VECTOR(1 downto 0);
35           O : out STD_LOGIC_VECTOR(width-1 downto 0));
36   end component;
37
38   for SEL : MUX41 use entity work.MUX41(selected);
39   for COND : MUX41 use entity work.MUX41(conditional);
40 begin
```

[simulation_I/testbench_multi.vhd](#)

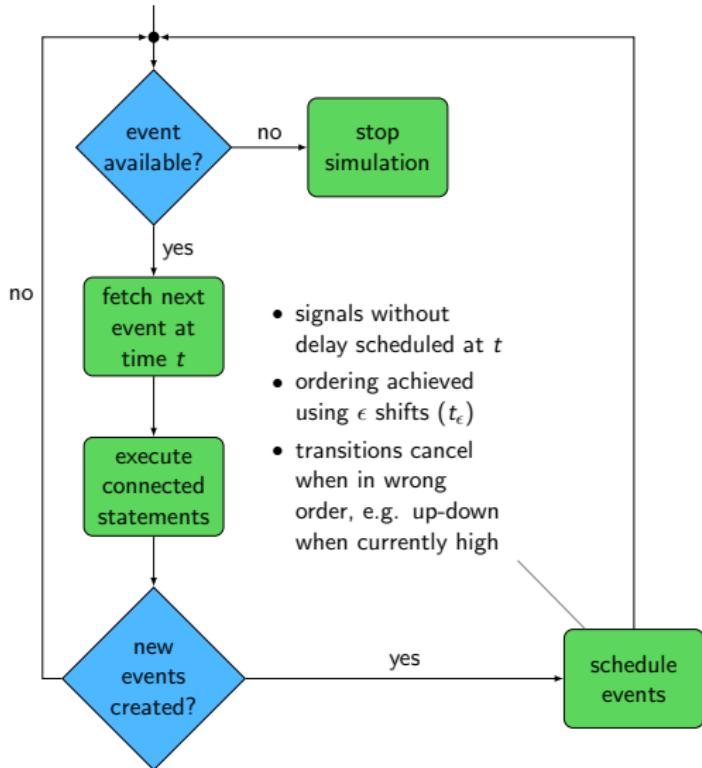
Complete Testbench cont'd

```
40 begin
41
42     TB_I0 <= (others => '0');
43     TB_I1 <= "10";
44     TB_I2 <= transport "01" after 0 ns, "11" after 4 ns;
45     TB_I3 <= "10";
46
47     TB_S <= transport "00" after 0 ns, "01" after 1 ns, "10" after 3 ns, "11" after 6 ns;
48
49 SEL : MUX41
50 generic map(width => width)
51 port map(I0 => TB_I0,
52           I1 => TB_I1,
53           I2 => TB_I2,
54           I3 => TB_I3,
55           S => TB_S,
56           O => TB_SEL);
57
58 COND : MUX41
59 generic map(width => width)
60 port map(I0 => TB_I0,
61           I1 => TB_I1,
62           I2 => TB_I2,
63           I3 => TB_I3,
64           S => TB_S,
65           O => TB_COND);
66
67 end architecture;
```

[simulation_1/testbench_multi.vhd](#)

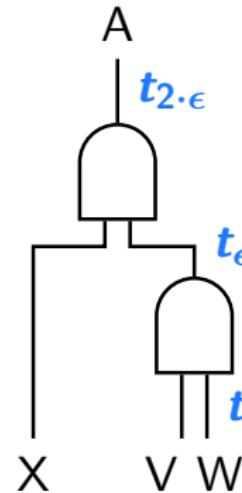
Simulation Algorithm

- event driven simulation
 - transition triggers evaluation
 - long simulation times possible
- new values are scheduled in future
 - may cancel previously scheduled ones
 - events at same time instant are separated by ϵ (very small) step



Simulation Algorithm

- event driven simulation
 - transition triggers evaluation
 - long simulation times possible
- new values are scheduled in future
 - may cancel previously scheduled ones
 - events at same time instant are separated by ϵ (very small) step



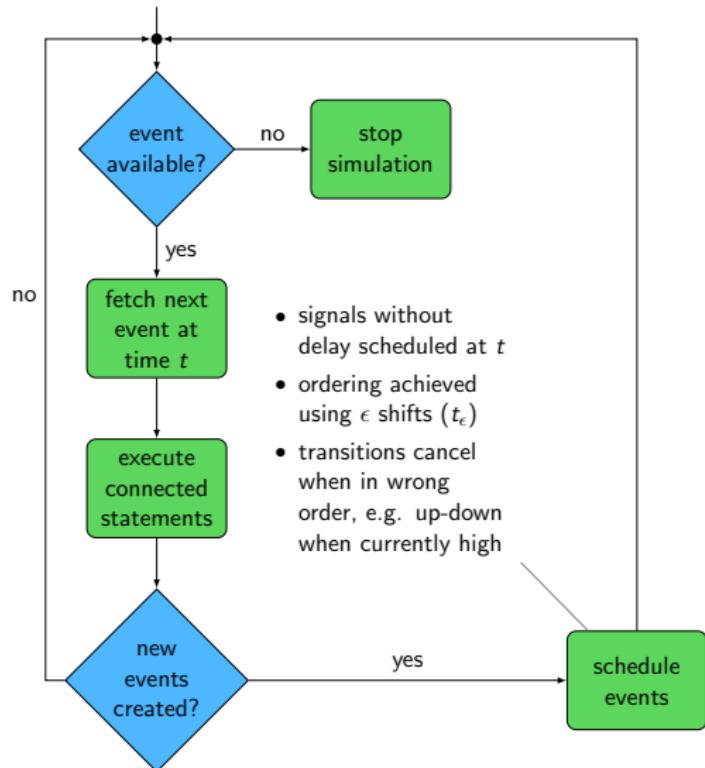
Simulation Algorithm

How is this code simulated?

testbench:

```
B <= '1';
```

```
1 architecture ...
2 ...
3 begin
4   A <= not B;
5   B <= not A;
6 end architecture;
```



Simulation Algorithm

How is this code simulated?

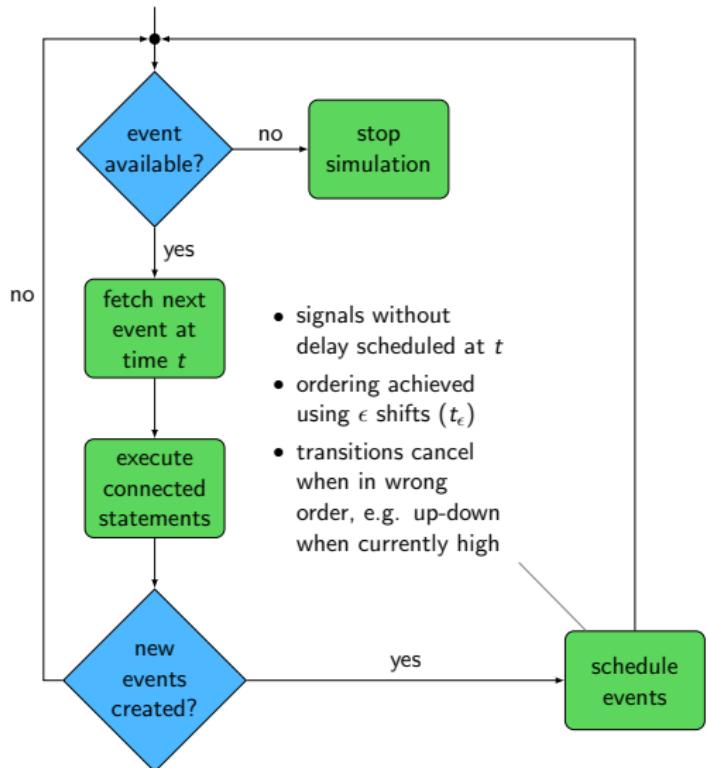
testbench:

```
B <= '1';
```

```

1  architecture ...
2  ...
3  begin
4      A <= not B;
5      B <= not A;
6  end architecture;
```

**Iteration limit
reached at time
0 ps.**



Generate

- N Bit MUX by N parallel 1 Bit MUX
 - very cumbersome to write
- simplified by **generate**
 - **for range generate**
 - **if condition generate**
 - **case expression generate**

```
45      MUXES: for i in width-1 downto 0 generate
46          UUT:MUX41_1
47              port map(I0 => TB_I0(i),
48                          I1 => TB_I1(i),
49                          I2 => TB_I2(i),
50                          I3 => TB_I3(i),
51                          S => TB_S,
52                          O => TB_O(i));
53      end generate;
```

[simulation_I/testbench_1Bit.vhd](#)

Generate

182

- N Bit MUX by N parallel 1 Bit MUX
 - very cumbersome to write
- simplified by **generate**
 - **for range generate**
 - **if condition generate**
 - **case expression generate**

```
58 configuration v1 of testbench is
59   for beh
60     for MUXES
61       for all:MUX41_1 use entity work.MUX41_1(selected); end for;
62     end for;
63   end for;
64 end configuration v1;
```

simulation_1/testbench_1Bit.vhd

Open Output Ports

sometimes output signals
not required

1. create dummy signal

- overhead
- not elegant
- not recommended

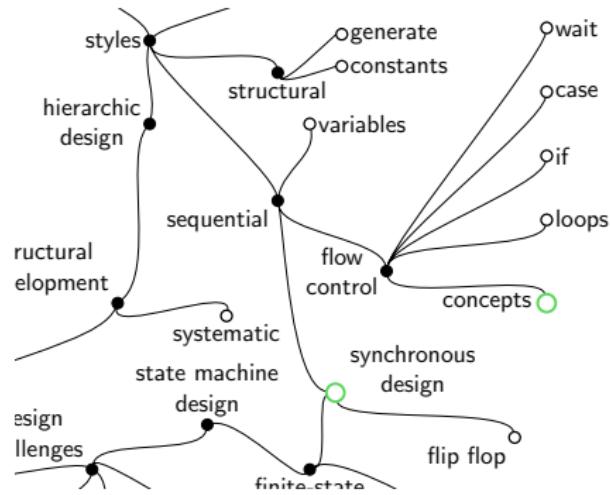
2. leave port unconnected

- declared by **open**
- only possible for output

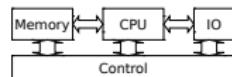
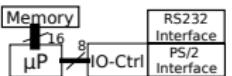
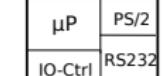
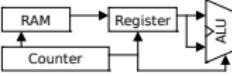
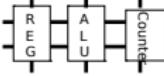
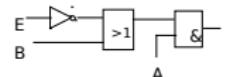
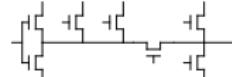
```
6      ...
7
8      UUT: MUX41
9      generic map(width => 2)
10     port map(I0 => TB_I0,
11                 I1 => TB_I1,
12                 I2 => TB_I2,
13                 I3 => TB_I3,
14                 S => TB_S,
15                 O => open);
16
17     ...
```

VHDL

Sequential Design Style



Y-diagram

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

Sequential Design Style

concurrent and structural style very powerful

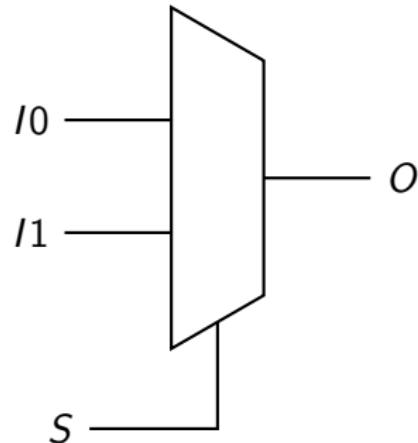
- every combinational circuit can be implemented

why use additional style?

- known control flow statements (if, loop, ...) can not be used
- tools specialized to detect units only in sequential style (e.g. memory, state machines)
- required for synchronous design

Sequential Design Style cont'd

- **process** allows sequential eradication of statements
 - concurrent statements still applicable
 - loops, branches, ... possible
 - hardware realization still concurrent
- process loops infinitely
 - paused solely when **wait** encountered
 - only then signal assignments carried out
 - amount of **wait** ≥ 1



Sequential Design Style cont'd

- **process** allows sequential eradication of statements
 - concurrent statements still applicable
 - loops, branches, ... possible
 - hardware realization still concurrent
- **process** loops infinitely
 - paused solely when **wait** encountered
 - only then signal assignments carried out
 - amount of **wait** ≥ 1

```
1  architecture ...
2  ...
3  begin
4
5  output : process
6  begin
7
8  O <= (I0 and (not S)) or (I1 and S);
9  wait on I0, I1, S;
10
11 end process;
12
13 end architecture;
```

Sequential Design Style cont'd

- **process** allows sequential eradication of statements
 - concurrent statements still applicable
 - loops, branches, ... possible
 - hardware realization still concurrent
- process loops infinitely
 - paused solely when **wait** encountered
 - only then signal assignments carried out
 - amount of **wait** ≥ 1

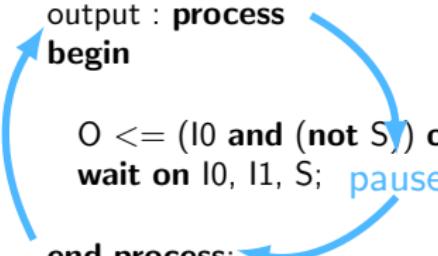
```
1  architecture ...
2  ...
3  begin
4
5  output: process
6  begin
7
8  O <= (I0 and (not S)) or (I1 and S);
9  wait on I0, I1, S;
10
11 end process;
12
13 end architecture;
```



Sequential Design Style cont'd

- **process** allows sequential eradication of statements
 - concurrent statements still applicable
 - loops, branches, ... possible
 - hardware realization still concurrent
- process loops infinitely
 - paused solely when **wait** encountered
 - only then signal assignments carried out
 - amount of **wait** ≥ 1

```
1  architecture ...
2  ...
3  begin
4
5  output : process
6  begin
7
8  O <= (I0 and (not S)) or (I1 and S);
9  wait on I0, I1, S; pause
10
11 end process;
12
13 end architecture;
```



Wait Statements

- are not synthesized, just for simulation
- **wait on signal list;**
 - continue when one of signals in the list changes
- **wait until condition;**
 - continue when *condition* becomes true
- **wait for time;**
 - continue after *time* has past
- **wait;**
 - never continue (wait forever)
 - stops execution of testbenches

Sensitivity list

- **wait** can be replaced by sensitivity list
 - adds automatically **wait** at end
 - no additional waits possible!
- signals triggering simulation step have to be added
 - common mistake that signals forgotten
 - simulation and reality differ significantly
- since standard 2008 keyword **all** possible
 - adds all signals that are read
- ignored during synthesis (warning when not complete)

```
output : process (I0, I1, S)
begin
```

$O \leq (I0 \text{ and } \dots$



```
output : process
begin
```

$O \leq (I0 \text{ and } \dots$
wait on I0, I1, S;

```
end process;
```

```
end process;
```

Sensitivity list

- **wait** can be replaced by sensitivity list
 - adds automatically **wait** at end
 - no additional waits possible!
- signals triggering simulation step have to be added
 - common mistake that signals forgotten
 - simulation and reality differ significantly
- since standard 2008 keyword **all** possible
 - adds all signals that are read
- ignored during synthesis (warning when not complete)

output : process (I0, I1, S)
begin

O <= (I0 and ...

end process;

output : process
begin

O <= (I0 and ...
wait on I0, I1, S;

end process;

Process Execution

```
1 architecture beh of abc is
2     signal s1, s2, s3 : BIT;
3     begin
4         p1 : process(s1)
5             begin
6                 s2 <= s1;
7                 s3 <= s2;
8             end process;
9     end architecture beh;
```

Assumptions:

At start all signals '0'

Process Execution

```
1 architecture beh of abc is
2   signal s1, s2, s3 : BIT;
3   begin
4     p1 : process(s1)
5       begin
6         s2 <= s1;
7         s3 <= s2;
8       end process;
9   end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

Process Execution

```
1 architecture beh of abc is
2   signal s1, s2, s3 : BIT;
3   begin
4     p1 : process(s1)
5       begin
6         s2 <= s1;
7         s3 <= s2;
8       end process;
9   end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

s2: marked to get '1'
(s1's value)

Process Execution

```
1 architecture beh of abc is
2   signal s1, s2, s3 : BIT;
3   begin
4     p1 : process(s1)
5       begin
6         s2 <= s1;
7         s3 <= s2;
8       end process;
9   end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

s2: marked to get '1'
(s1's value)

s3: marked to get '0'
(s2's old value!)

Process Execution

```
1 architecture beh of abc is
2   signal s1, s2, s3 : BIT;
3   begin
4     p1 : process(s1)
5       begin
6         s2 <= s1;
7         s3 <= s2;
8       end process;
9   end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

s2: marked to get '1'
(s1's value)

s3: marked to get '0'
(s2's old value!)

Implicit wait (sensitivity list)

new values assigned to s2 and s3 ⇒
s2 → '1', s3 stays '0'!

Process Execution

```
1 architecture beh of abc is
2   signal s1, s2, s3 : BIT;
3   begin
4     p1 : process(s1)
5       begin
6         s2 <= s1;
7         s3 <= s2;
8       end process;
9   end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

s2: marked to get '1'
(s1's value)

s3: marked to get '0'
(s2's old value!)

Implicit wait (sensitivity list)

new values assigned to s2 and s3 ⇒
 $s2 \rightarrow '1'$, $s3$ stays '0'!

$s2 \neq s3$! How can we assure that also $s3$ is updated ???

Process Execution

```
1 architecture beh of abc is
2     signal s1, s2, s3 : BIT;
3 begin
4     p1 : process(s1, s2)
5         begin
6             s2 <= s1;
7             s3 <= s2;
8         end process;
9     end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

s2: marked to get '1'
(s1's value)

s3: marked to get '0'
(s2's old value!)

Implicit wait (sensitivity list)

new values assigned to s2 and s3 ⇒
 $s2 \rightarrow '1'$, $s3$ stays '0'!

$s2 \neq s3$! How can we assure that also $s3$ is updated ???

Add s2 to sensitivity list!

Process Execution

```
1 architecture beh of abc is
2   signal s1, s2, s3 : BIT;
3   begin
4     p1 : process(all)
5       begin
6         s2 <= s1;
7         s3 <= s2;
8       end process;
9   end architecture beh;
```

Assumptions:

At start all signals '0'

s1: '0' → '1' ⇒ start process

s2: marked to get '1'
(s1's value)

s3: marked to get '0'
(s2's old value!)

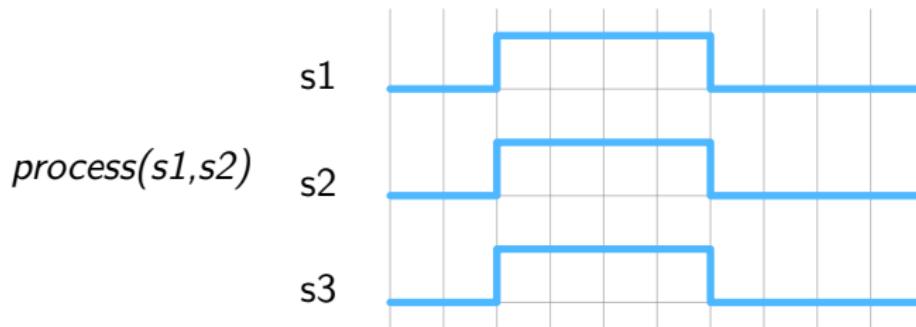
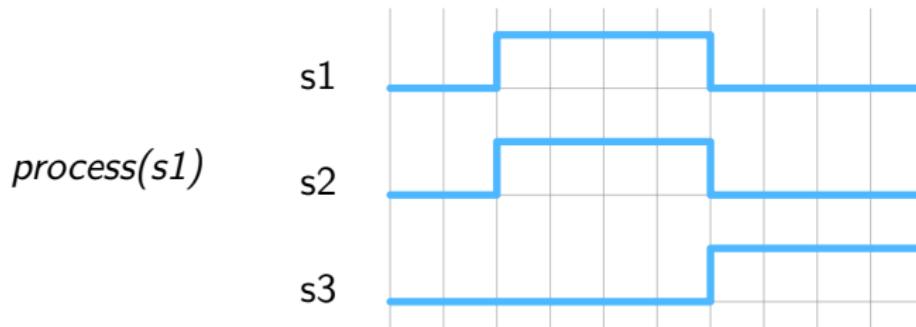
Implicit wait (sensitivity list)

new values assigned to s2 and s3 ⇒
 $s2 \rightarrow '1'$, $s3$ stays '0'!

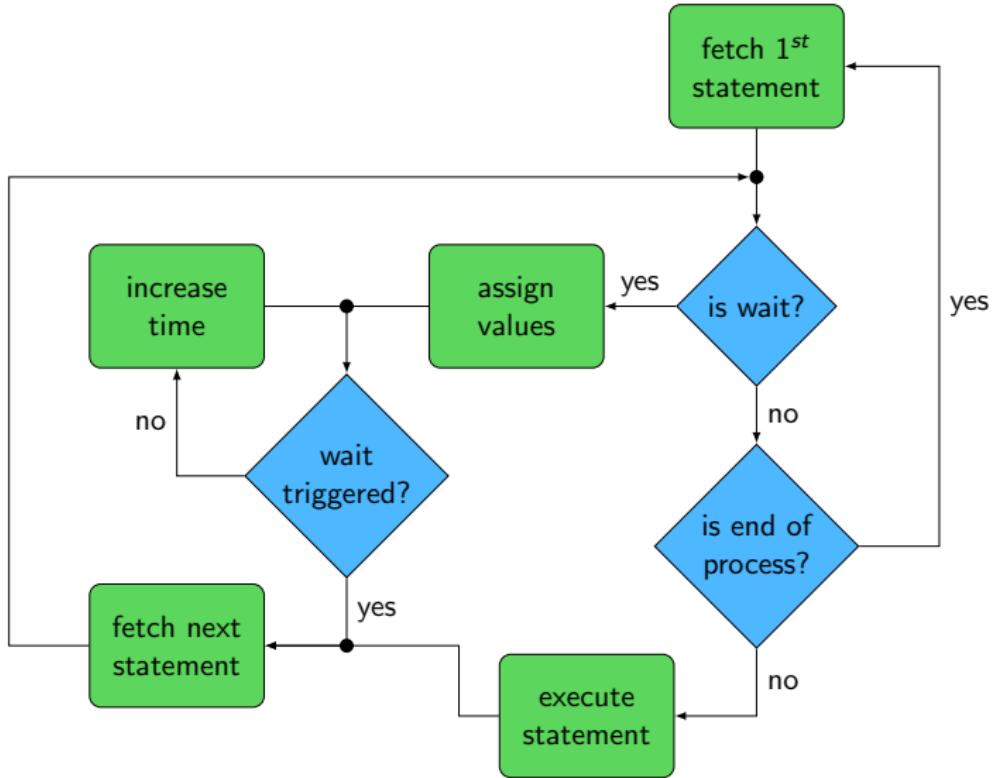
$s2 \neq s3$! How can we assure that also $s3$ is updated ???

Add s2 to sensitivity list!

Process Execution cont'd



Process Simulation



Process

- statements executed sequentially in infinite loop
 - generated hardware still parallel
- execution paused at **wait**
 - only then signals assigned
 - automatically created by sensitivity list
- multiple **process** in single architecture
 - are executed in parallel
 - activity can trigger execution of same/other **process**
- can be combined with concurrent and structural design

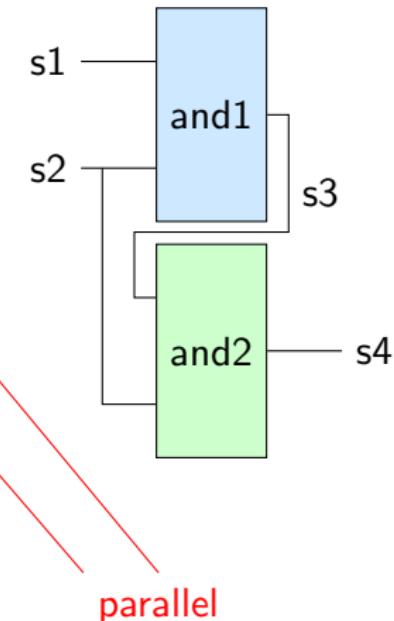
```
1 [ label ] : [ postponed ] process [ (sensitivity_list) ] [ is ]
2     process_declarative_part
3 begin
4     process_statement_part
5 end [ postponed ] process [ label ];
```

Example: Architecture with 2 Processes

```
architecture beh of abc is
    signal s1, s2, s3, s4: STD_LOGIC;
begin
    and1 : process(s1, s2)
        if s1 = '1' and s2 = '1' then
            s3 <= '1';
        else
            s3 <= '0';
        end if;
    end process and1;
    and2 : process(s2, s3)
        if s2 = '1' and s3 = '1' then
            s4 <= '1';
        else
            s4 <= '0';
        end if;
    end process and2;
end architecture beh;
```

sequential

sequential



Example: Architecture with 2 Processes

```
architecture beh of abc is
    signal s1, s2, s3, s4: STD_LOGIC;
begin
    and1 : process(s1, s2)
        if s1 = '1' and s2 = '1' then
            s3 <= '1';
        else
            s3 <= '0';
        end if;
    end process and1;
    and2 : process(s2, s3)
        if s2 = '1' and s3 = '1' then
            s4 <= '1';
        else
            s4 <= '0';
        end if;
    end process and2;
end architecture beh;
```

Initial Values:

s1 = 0

s2 = 1

⇒ s3 = 0

⇒ s4

Example: Architecture with 2 Processes

```
architecture beh of abc is
    signal s1, s2, s3, s4: STD_LOGIC;
begin
    and1 : process(s1, s2)
        if s1 = '1' and s2 = '1' then
            s3 <= '1';
        else
            s3 <= '0';
        end if;
    end process and1;
    and2 : process(s2, s3)
        if s2 = '1' and s3 = '1' then
            s4 <= '1';
        else
            s4 <= '0';
        end if;
    end process and2;
end architecture beh;
```

Initial Values:

s1 = 0

s2 = 1

⇒ s3 = 0

⇒ s4

Simulation:

s1: 0 → 1

Example: Architecture with 2 Processes

```
architecture beh of abc is
    signal s1, s2, s3, s4: STD_LOGIC;
begin
    and1 : process(s1, s2)
        if s1 = '1' and s2 = '1' then
            s3 <= '1';
        else
            s3 <= '0';
        end if;
    end process and1;
    and2 : process(s2, s3)
        if s2 = '1' and s3 = '1' then
            s4 <= '1';
        else
            s4 <= '0';
        end if;
    end process and2;
end architecture beh;
```

Initial Values:

s1 = 0
s2 = 1
⇒ s3 = 0
⇒ s4

Simulation:

s1: 0 → 1
⇒ s3: 0 → 1

Example: Architecture with 2 Processes

```
architecture beh of abc is
    signal s1, s2, s3, s4: STD_LOGIC;
begin
    and1 : process(s1, s2)
        if s1 = '1' and s2 = '1' then
            s3 <= '1';
        else
            s3 <= '0';
        end if;
    end process and1;
    and2 : process(s2, s3)
        if s2 = '1' and s3 = '1' then
            s4 <= '1';
        else
            s4 <= '0';
        end if;
    end process and2;
end architecture beh;
```

Initial Values:

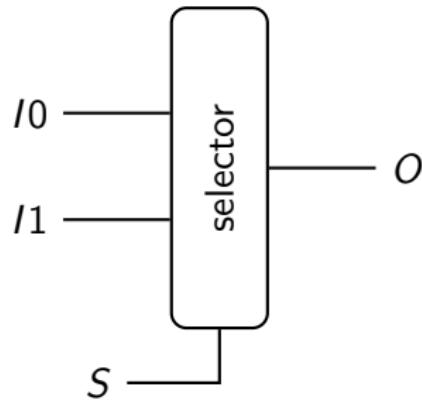
s1 = 0
s2 = 1
⇒ s3 = 0
⇒ s4

Simulation:

s1: 0 → 1
⇒ s3: 0 → 1
⇒ s4: 0 → 1

Recap

- remember our task
 - implement selector circuit,
a.k.a. multiplexer
- logic level implementations
 - selected signal assignment
 - conditional signal assignment
- How can we do that on RTL level?



If Statement

- same as in software programming language
- corresponds to conditional signal assignment
- how can we implement our MUX with that?

```
1 [ label: ] if condition then
2   sequence_of_statements
3 {elsif condition then
4   sequence_of_statements}
5 [else
6   sequence_of_statements]
7 end if [ label ];
```

If Statement

- same as in software programming language
- corresponds to conditional signal assignment
- how can we implement our MUX with that?

sequential

```

ITE : process (all)
begin
    if S = "00" then
        O <= I0;
    elsif S = "01" then
        O <= I1;
    elsif S = "10" then
        O <= I2;
    else
        O <= I3;
    end if;
end process;

```

[eaMUX41_if.vhd](#)

concurrent

```

architecture conditional of MUX41 is
begin
    O <= I0 when S="00" else
    I1 when S="01" else
    I2 when S="10" else
    I3;
end architecture;

```

[eaMUX41_conditional.vhd](#)

Common Pitfall

- What is wrong with this code?

```
architecture ITE of MUX41 is
begin
```

```
    ITE : process (all)
        begin
```

```
            if S = "00" then
```

```
                O <= I0;
```

```
            elsif S = "01" then
```

```
                O <= I1;
```

```
            elsif S = "10" then
```

```
                O <= I2;
```

```
        end if;
```

```
    end process;
```

```
end architecture;
```

Common Pitfall

- What is wrong with this code?
 - No default value!
 - remember math exam example
- introduces latch
 - message “inferring latch”
 - almost never desired in synchronous logic
- always use default assignment and / or default branch

```

architecture ITE of MUX41 is
begin
  ITE : process (all)
  begin
    if S = "00" then
      O <= I0;
    elsif S = "01" then
      O <= I1;
    elsif S = "10" then
      O <= I2;
    end if;
  end process;
end architecture;

```

```

INFO: Only one process was detected - disabling parallel compilation
Info: Found 2 design units, including 1 entities, in source file latch.vhd
Info: Elaborating entity "latch1" for the top level hierarchy
Warning (10631): VHDL Process Statement warning at latch.vhd(14): inferring latch(es) for signal or variable "o", w
Info (10041): Inferred latch for "o" at latch.vhd(14)
Info: Implemented 6 device resources after synthesis - the final resource count might be different

```

Case Statement

- pick first matching argument
- corresponds to selected signal assignment
- how can we implement our MUX with that?

```
1 [ label: ] case [ ? ] expression is
2   when choices =>
3     sequence_of_statements
4   {when choices =>
5     sequence_of_statements}
6 end case [ ? ] [ label ];
```

Case Statement

- pick first matching argument
- corresponds to selected signal assignment
- how can we implement our MUX with that?

sequential

```
ITE : process (all)
begin
    case S is
        when "00" =>
            O <= I0;
        when "01" =>
            O <= I1;
        when "10" =>
            O <= I2;
        when others =>
            O <= I3;
    end case;
end process;

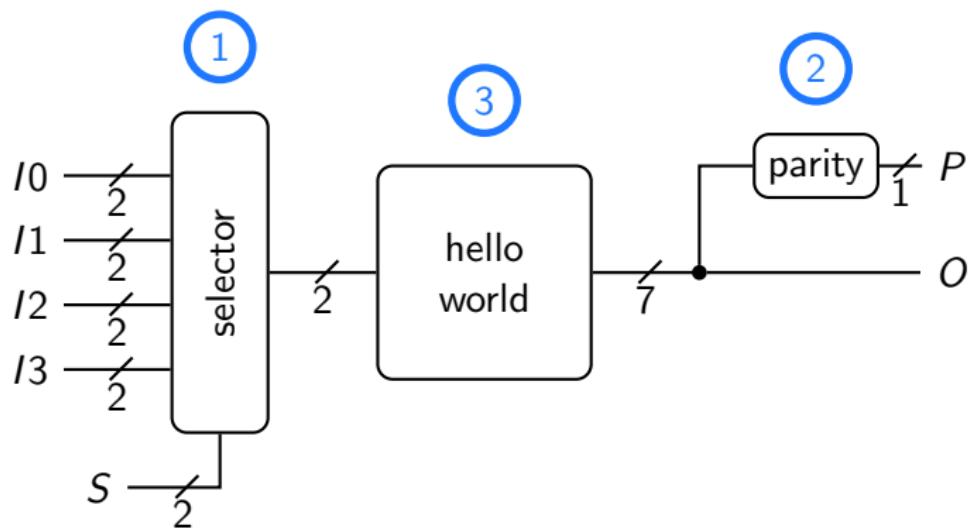
eaMUX41_case.vhd
```

concurrent

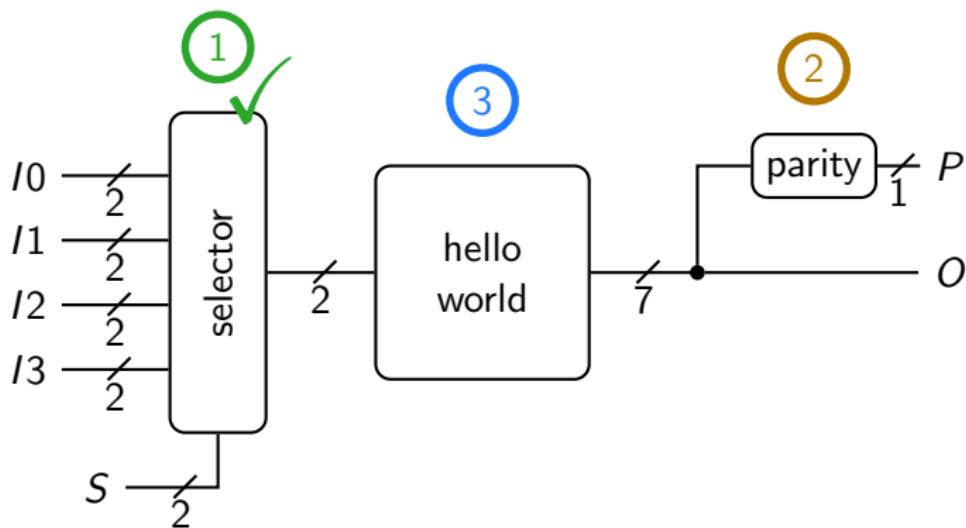
```
architecture selected of MUX41 is
begin
    with S select
        O <= I0 when "00",
        I1 when "01",
        I2 when "10",
        I3 when others;
end architecture;

eaMUX41_selected.vhd
```

First Design – Hello World!



First Design – Hello World!



Parity Unit

- assures even number of 1s
 - output 0 if amount of input 1s even
 - 1 otherwise
- realization: connect all inputs to **xor** gate
- first consider small example
 - assume data width of 3 Bits
 - create parity unit in **VHDL**

entity

```
entity parity is
port (
    I : in STD_LOGIC_VECTOR(2 downto 0);
    O : out STD_LOGIC);
end entity;
```

[eaParity.vhd](#)

architecture

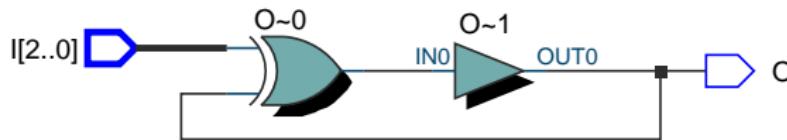
```
architecture beh of parity is
begin
    calc : process (all)
    begin
        O <= I(0) xor I(1);
        O <= I(2) xor O;
    end process;
end architecture;
```

[eaParity.vhd](#)



Parity Unit cont'd

Synthesis gives



Why does it not work?

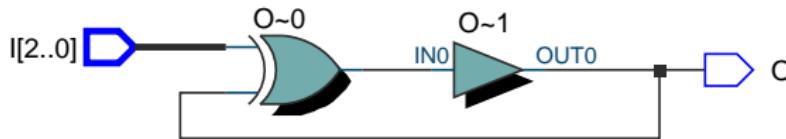
```
28 architecture beh of parity is
29 begin
30   calc : process (all)
31   begin
32     O <= I(0) xor I(1);
33     O <= I(2) xor O;
34   end process;
35 end architecture;
```

eaParity.vhd



Parity Unit cont'd

Synthesis gives



Why does it not work?

```
28 architecture beh of parity is
29 begin
30   calc : process (all)
31   begin
32     O <= I(0) xor I(1);
33     O <= I(2) xor O;
34   end process;
35 end architecture;
```

eaParity.vhd

Assignments only
at **wait!**

Parity Unit cont'd

- recall: signals only assigned at **wait**
 - signal O not updated inside **process**
 - in second assignment O still has initial value
 - first statement overwritten
- use **variable** for immediate effect
 - value gets updated right away
 - not allowed in sensitivity list

signal

```
calc : process (all)
begin
  O <= I(0) xor I(1);
  O <= I(2) xor O;
end process;
```

eaParity.vhd

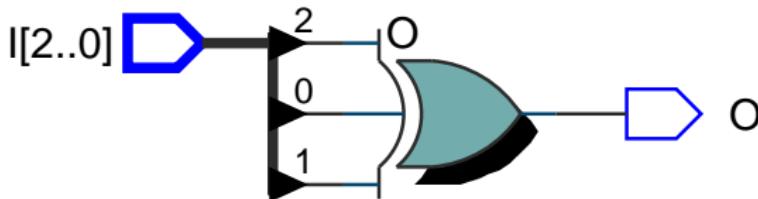
variable

```
calc : process (all)
  variable parity : STD_LOGIC;
begin
  parity := I(0) xor I(1);
  O <= I(2) xor parity;
end process;
```

eaParity.variable.vhd

Parity Unit cont'd

Synthesis gives



Now everything looks fine!

```
30      calc : process (all)
31          variable parity : STD_LOGIC;
32      begin
33          parity := I(0) xor I(1);
34          O <= I(2) xor parity;
35      end process;
```

[eaParity_variable.vhd](#)

Variables vs. Signals

signal

```
architecture sig of app is
    signal W, X, Y, Z : STD_LOGIC;
begin
    process (all)

        begin
            X <= A;
            Y <= B;
            Z <= X and Y;
            Y <= C;
            W <= X and Y;
        end process;
    end architecture;
```

variable

```
architecture var of app is
    signal W, Z : STD_LOGIC;
begin
    process (all)
        variable X, Y : STD_LOGIC;
        begin
            X := A;
            Y := B;
            Z <= X and Y;
            Y := C;
            W <= X and Y;
        end process;
    end architecture var;
```

Initial Values: A=1, B=1, C=1; W=? , X=? , Y=? , Z=? ; C: 1 → 0

Iteration #	signal values at end
1	
2	

Iteration #	signal values at end
1	
2	

Variables vs. Signals

signal

```
architecture sig of app is
    signal W, X, Y, Z : STD_LOGIC;
begin
    process (all)

        begin
            X <= A;
            Y <= B;
            Z <= X and Y;
            Y <= C;
            W <= X and Y;
        end process;
    end architecture;
```

variable

```
architecture var of app is
    signal W, Z : STD_LOGIC;
begin
    process (all)
        variable X, Y : STD_LOGIC;
        begin
            X := A;
            Y := B;
            Z <= X and Y;
            Y := C;
            W <= X and Y;
        end process;
    end architecture var;
```

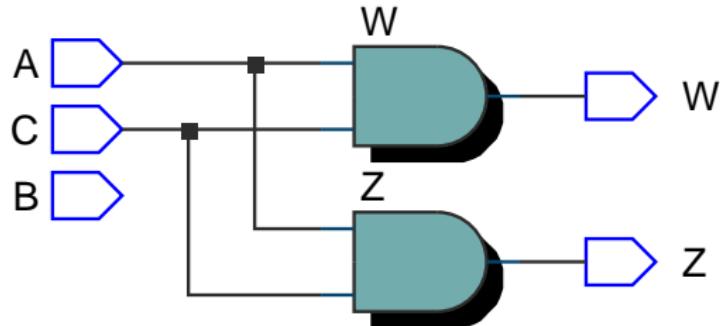
Initial Values: A=1, B=1, C=1; W=1, X=1, Y=1, Z=1; C: 1 → 0

Iteration #	signal values at end
1	W = 1, X = 1 Y = 0, Z = 1
2	W = 0, X = 1 Y = 0, Z = 0

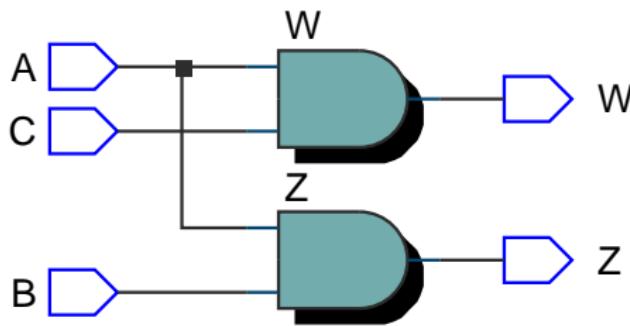
Iteration #	signal values at end
1	W = 0, X = 1 Y = 0, Z = 1
2	-

Variables vs. Signals cont'd

signals



variables



Parity Unit cont'd

- currently only 3 inputs
 - extend it to 16
- bad maintainability of current implementation
 - changing input width cumbersome
 - solution not generic
- same statement is called again and again
 - loops can be used to shorten code

```
1 architecture beh of parity is
2 begin
3 calc : process (all)
4 variable parity: STD_LOGIC;
5 begin
6 parity := I(0) xor I(1);
7 parity := I(2) xor parity;
8 ...
9 O := I(width-1) xor parity;
10 end process;
11 end architecture;
```

Loops

- simplify and shorten code
 - essentially code generation units
 - ATTENTION: use carefully
- program in generic fashion
 - what if no generics available?
 - alternative: attributes
 - **I'RANGE \Leftrightarrow 2 downto 0**

```
30      calc : process (all)
31          variable parity : STD_LOGIC;
32      begin
33          parity := '0';
34          for idx in 2 downto 0 loop
35              parity := l(idx) xor parity;
36          end loop;
37          O <= parity;
38      end process;
```

eaParity_for.vhd

Loops

- simplify and shorten code
 - essentially code generation units
 - ATTENTION: use carefully
- program in generic fashion
 - what if no generics available?
 - alternative: attributes
 - **I'RANGE** \Leftrightarrow 2 **downto** 0

```
31      calc : process (all)
32          variable parity : STD_LOGIC;
33      begin
34          parity := '0';
35          for idx in I'RANGE loop
36              parity := I(idx) xor parity;
37          end loop;
38          O <= parity;
39      end process;
```

eaParity_for_range.vhd

Loops cont'd

- different loop types available
- iteration_schemes
 - **for** *identifier* **in** *range*
 - **while** *condition*
- **next** [*loop_label*] [**when** *condition*]
 - complete current iteration of loop
- **exit** [*loop_label*] [**when** *condition*]
 - complete execution of loop

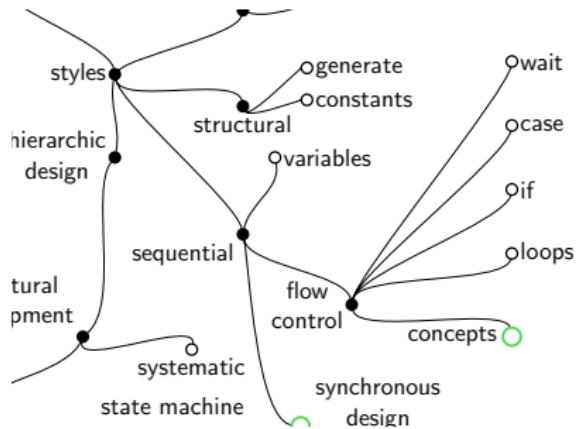
```
1  [loop_label:]
2    [iteration_scheme] loop
3      sequence_of_statements
4    end loop [loop_label];
```

Summary Sequential Design Style

- powerful concept of **process**
 - allows usage of common control flow elements (if, loops, ...)
 - when bottom hit starts from top
- commands are executed sequentially
 - new values only assigned at next wait
 - implicitly introduced by sensitivity list
- multiple **process** possible
 - are executed in parallel (concurrent style)
 - actions in one can trigger execution of other
- variables vs. signal

VHDL

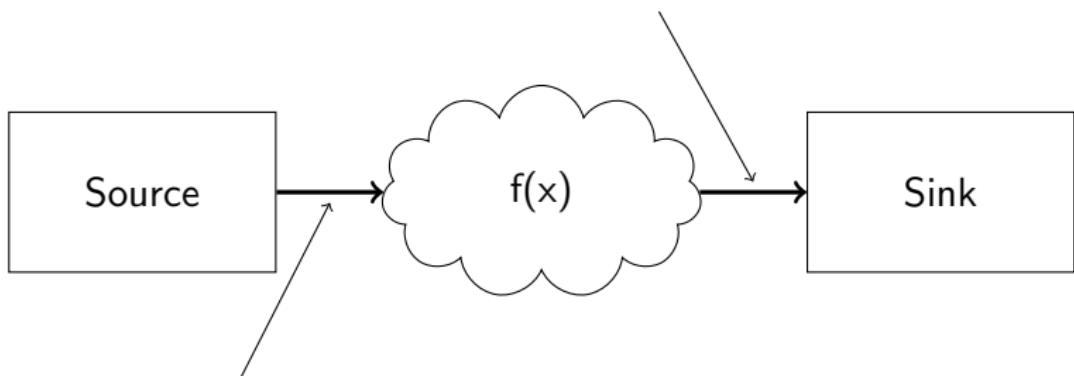
Synchronous Circuit Design



Synchronous Circuit Design

When is the data valid and consistent?

When may the sink use the data?



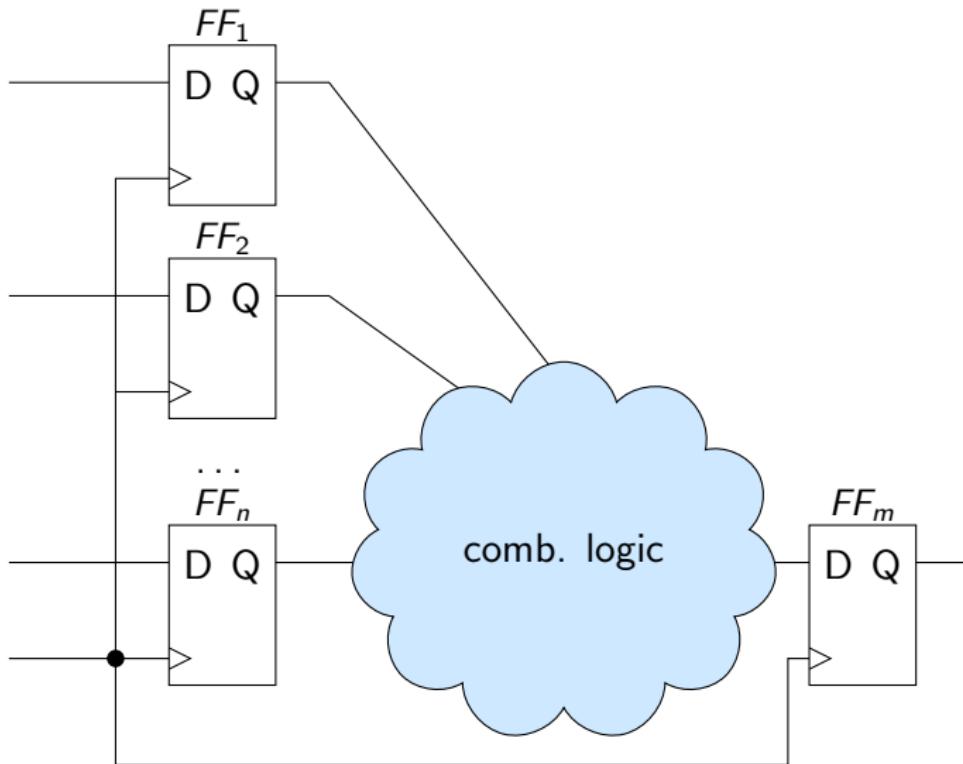
When to issue the next data item?

When has the sink consumed the data?

Synchronous Circuit Design cont'd

- solution: create common, discrete time base
 - key feature: central clock
 - all events triggered by this signal
- very powerful toolchains available
 - nearly all commercial circuits synchronous
- change memory values only at rising clock edge
 - suitable memory elements called flip-flop
 - connected by (asynchronous) logic
- problems
 - complex timing analysis
 - slowest path determines clock
 - ...

Synchronous Circuit Design cont'd



D-Flip Flop

- change memory values only at rising clock edge
 - solely use those in synchronous circuits
- only specify behavior at clock edge
 - not specifying other cases results in memory
- is mapped to appropriate elements in FPGA
- Why not use all in sensitivity list?

entity

```
entity flip_flop is
port (
    clk : in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF.vhd

architecture

```
architecture beh of flip_flop is
begin
    sync : process (clk)
    begin
        if rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end architecture;
```

eaFF.vhd

D-Flip Flop

- change memory values only at rising clock edge
 - solely use those in synchronous circuits
- only specify behavior at clock edge
 - not specifying other cases results in memory
- is mapped to appropriate elements in FPGA
- Why not use all in sensitivity list? no unnecessary simulations

entity

```
entity flip_flop is
port (
    clk : in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF.vhd

architecture

```
architecture beh of flip_flop is
begin
    sync : process (clk)
    begin
        if rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end architecture;
```

eaFF.vhd

Clock Enable

- what if update on clock edge shall be prevented?
- gated clock
 - bad design style (in FPGAs)
- separate enable signal

entity

```
entity flip_flop is
  port
  (
    clk, en, D : in STD_LOGIC;
    Q : out STD_LOGIC
  );
end entity;
```

eaFF_en.vhd

architecture

```
architecture rtl of flip_flop is
begin
  sync : process (clk, en)
  begin
    if rising_edge(clk) then
      if en = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
```

eaFF_en.vhd

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here?

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if res='0' then
            if rising_edge(clk) then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here? **None of both!**

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if res='0' then
            if rising_edge(clk) then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here?

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if rising_edge(clk) then
            Q <= 'D';
        elsif rising_edge(clk) and res='0' then
            Q <= 0;
        end if;
    end process;

end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here? **None of both!**

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if rising_edge(clk) then
            Q <= 'D';
        elsif rising_edge(clk) and res='0' then
            Q <= 0;
        end if;
    end process;

end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here?

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res1 of flip_flop is
begin
    sync : process (clk, res)
    begin
        if res = '0' then
            Q <= '0';
        elsif rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end architecture;
```

eaFF_res1.vhd

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here? **synchronous** / **asynchronous**

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res1 of flip_flop is
begin
    sync : process (clk, res)
    begin
        if res = '0' then
            Q <= '0';
        elsif rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end architecture;
```

eaFF_res1.vhd

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here?

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;

eaFF_res1.vhd
```

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if rising_edge(clk) then
            if res='0' then
                Q <= '0';
            end if;
        else
            Q <= D;
        end if;
    end process;
end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here? **None of both!**

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if rising_edge(clk) then
            if res='0' then
                Q <= '0';
            end if;
        else
            Q <= D;
        end if;
    end process;
end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here?

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;

eaFF_res1.vhd
```

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if rising_edge(clk) then
            if res='0' then
                Q <= '0';
            end if;
            Q <= D;
        end if;
    end process;
end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here? **None of both!**

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res0 of flip_flop is
begin
    sync:process(clk, res)
    begin
        if rising_edge(clk) then
            if res='0' then
                Q <= '0';
            end if;
            Q <= D;
        end if;
    end process;
end architecture;
```

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here?

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

architecture

```
architecture res2 of flip_flop is
begin
    sync : process (clk, res)
    begin
        if rising_edge(clk) then
            if res = '0' then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end architecture;
```

eaFF_res2.vhd

Initialization

- proper initialization important
 - otherwise undefined behaviour at startup ('X')
- use reset signal
 - synchronous and asynchronous versions
 - What do you see here? **synchronous** / **asynchronous**

entity

```
entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;
```

eaFF_res1.vhd

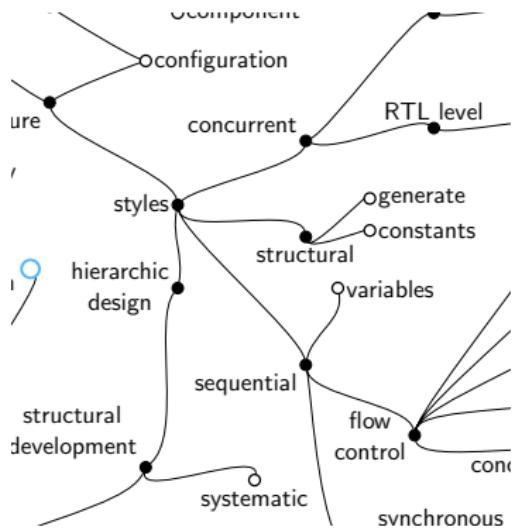
architecture

```
architecture res2 of flip_flop is
begin
    sync : process (clk, res)
    begin
        if rising_edge(clk) then
            if res = '0' then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end architecture;
```

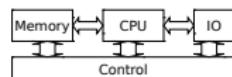
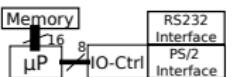
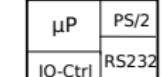
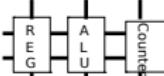
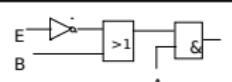
eaFF_res2.vhd

VHDL

Design Styles Comparison



Y-diagram

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B := B+1 else B := B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{I}{C} + L \frac{d^2I}{dt^2}$		

Concurrent Design Style

- all commands executed in parallel
 - order not important
- assignment operator ' $<=$ '
 - selected and conditional signal assigment
- specify all cases
 - otherwise latches may be inferred

```
1   O <= A and B;  
2  
3   P <= C when S = '1' else  
4       D when T = '0' else  
5       E;  
6  
7   with S select  
8     R <= F when '1' else  
9       G;
```

Structural Design Style

- instantiate entities and connect them
 - signals connected by position or by name
- component declaration required
 - **configuration** assignes **architecture-entity** to **component** declaration/instantiation

```
1  UUT: MUX41
2  generic map(width => width)
3  port map(I0 => TB_I0,
4          I1 => TB_I1,
5          I2 => TB_I2,
6          I3 => TB_I3,
7          S => TB_S,
8          O => TB_O);
```

Sequential Design Style

- sequential execution of statements inside **process**
 - signals assigned at next **wait**
 - if no assignment current value stored → inferred latches!
- for immediate changes **variable** required
 - keeps value between executions
- all read signals should be in sensitivity list
 - covered by **all** (not necessary for synchronous processes)

```
30      calc : process (all)
31          variable parity : STD_LOGIC;
32      begin
33          parity := I(0) xor I(1);
34          O <= I(2) xor parity;
35      end process;
```

[eaParity_variable.vhd](#)

When to use what?

- structural design style
 - top level
 - define structure of the implementation
 - assemble single components
- concurrent design style
 - drive output ports from internal signals
 - only simple combinational logic used
- sequential design style
 - define actual behavior of unit
 - synchronous design, state machines, memory, ...
- mixing possible
 - error prone → careful design required

Multiple Design Styles

Structural

```
architecture mixed of abc is
    signal s1, s2, s3, s4, s5 : STD_LOGIC;
begin
    and1 : one
generic map(DEPTH => 10)
port map(a => s1, b => s2, c => s3);
process(s3)
begin
```

Behavioral RTL level

```
    if s3='1' then
        s4 <= '0';
    else
        s4 <= '1';
    end if;
```

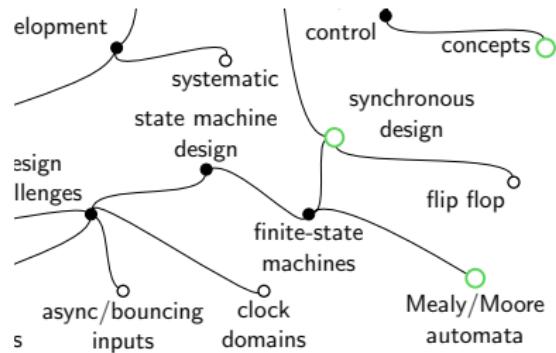
```
end process;
```

Behavioral logic level

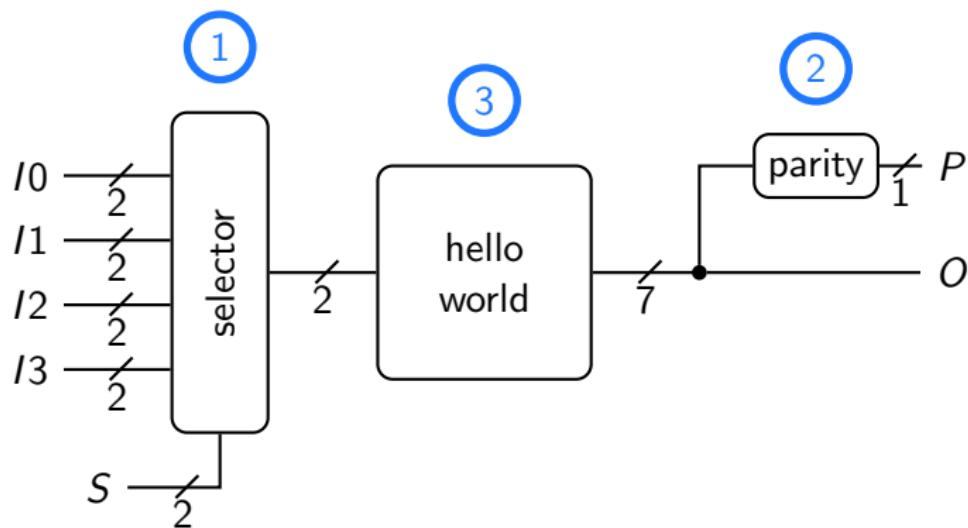
```
    s5 <= s2 xor s4;
end architecture mixed;
```

VHDL

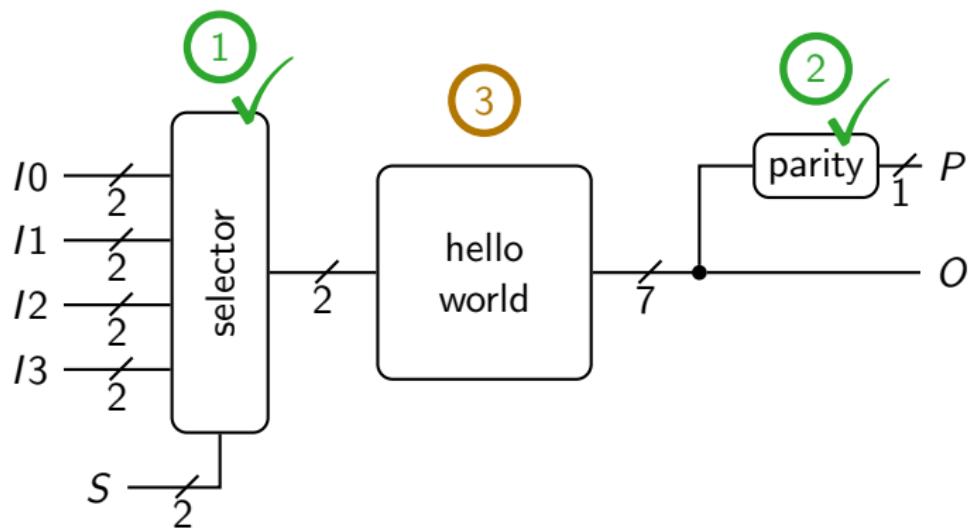
Finite State Machines (FSM)



First Design – Hello World!



First Design – Hello World!



Hello World Entity

inputs

- mode (2 Bits)
- clock (1 Bit)
- reset (1 Bit)

output

- display lines (7 Bits)

```
23  entity hello_world is
24    generic ( width : NATURAL := 1);
25    port (
26      clk, res : in STD_LOGIC;
27      mode : in STD_LOGIC_VECTOR(1 downto 0);
28      disp : out STD_LOGIC_VECTOR(width-1 downto 0)
29    );
30  end entity;
```

[hello_world/src/eaHello_world.vhd](#)

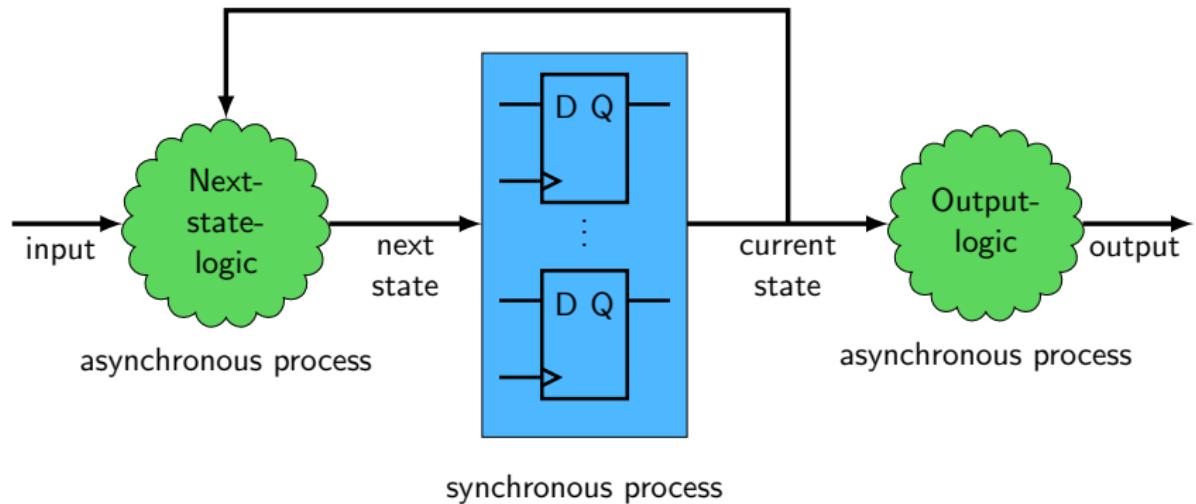
First Design – Hello World Behavior

- show “HELLO” on 7-segment display one character at a time
- run through word with following modes
 - 00 halt
 - 10 previous letter
 - 01 next letter
 - 11 not considered
- changes have to happen synchronous to clock
- inherent temporal order
 - inputs have different effect based on when they happen
 - circuit has internal state
- use state machines to model/implement that

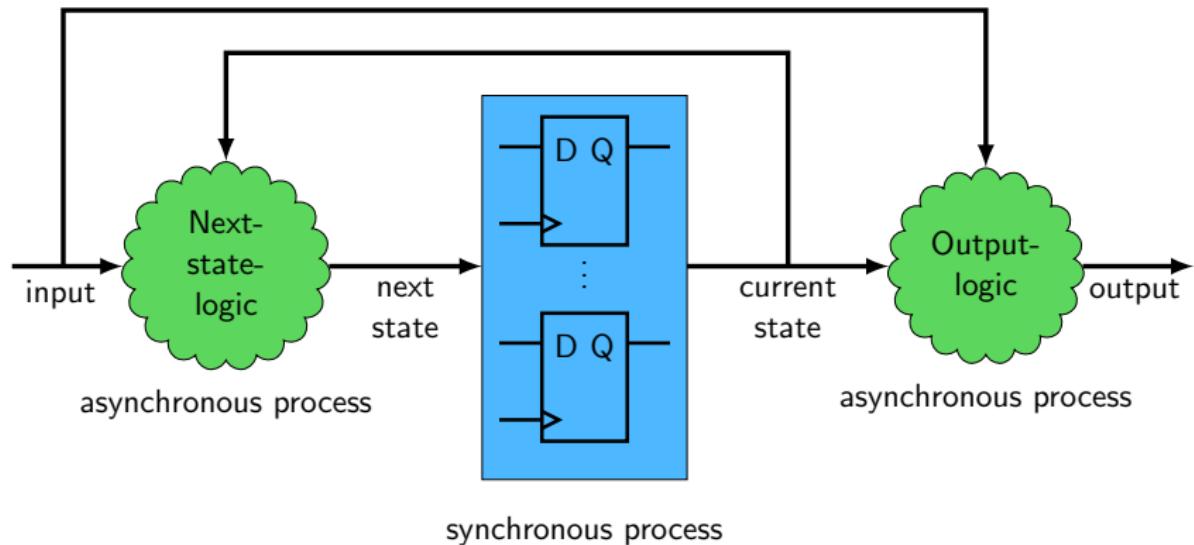
State Machines

- history of input important
 - which input vectors appeared in which order
 - represented by internal state
- states switched synchronous to clock
- next state depends on current state and input values
- output depends on
 - current state alone (Moore state machine)
 - o takes one cycle until inputs take effect
 - current state and input (Mealy state machine)
 - o outputs may change at any point in time

Moore State Machine



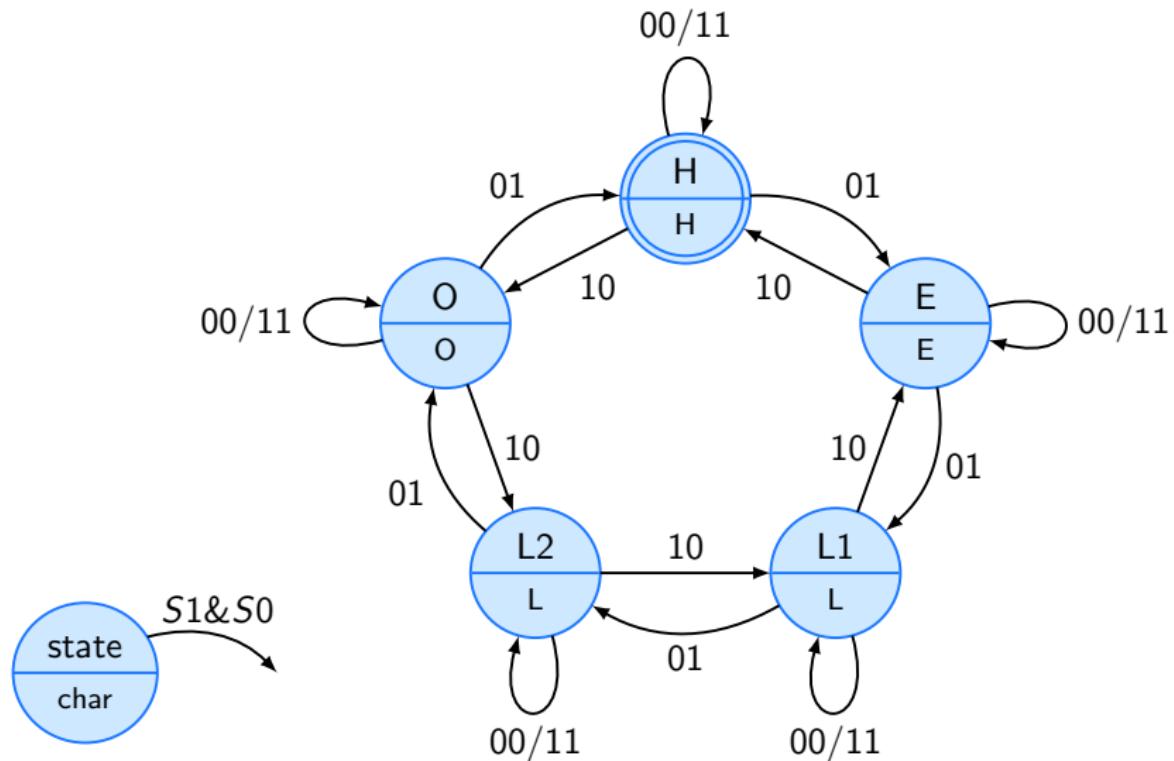
Mealy State Machine



Modeling State Machines

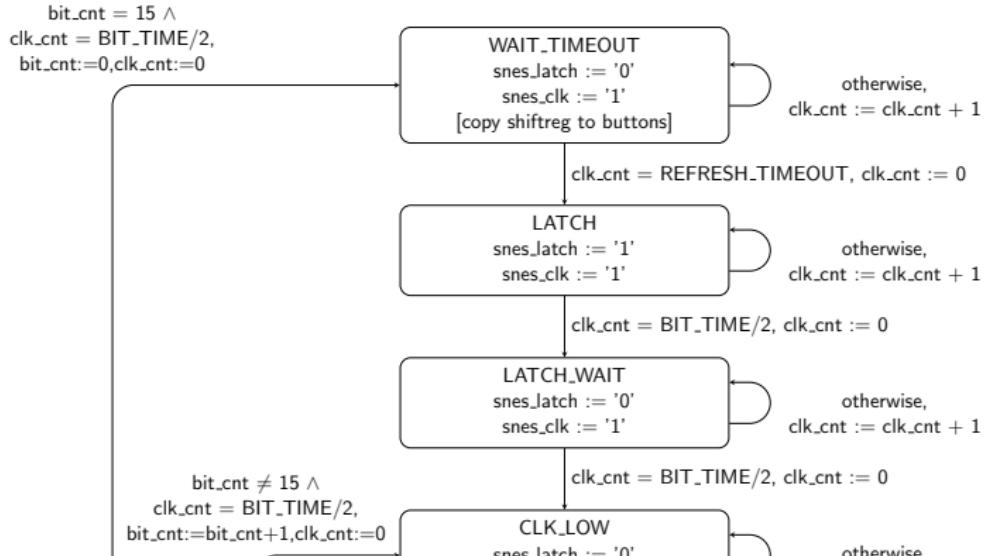
- model behavior as graph
 - always state notation of nodes and edges
 - make sure that all input combinations covered
- for Moore automata
 - node represent internal states & corresponding output
 - edge represent transitions based on input
- for Mealy automata
 - node represent internal states
 - edge represent transitions based on input and corresponding output

Modeling State Machines cont'd



Modeling State Machines cont'd

- model describes behavior of state machine
 - not unique neither in structure nor description
- graph representation used in lecture / exam
- compare to more detailed graph in LU
 - provides additional information about internal mechanisms



Enumeration Data Type

state encoding

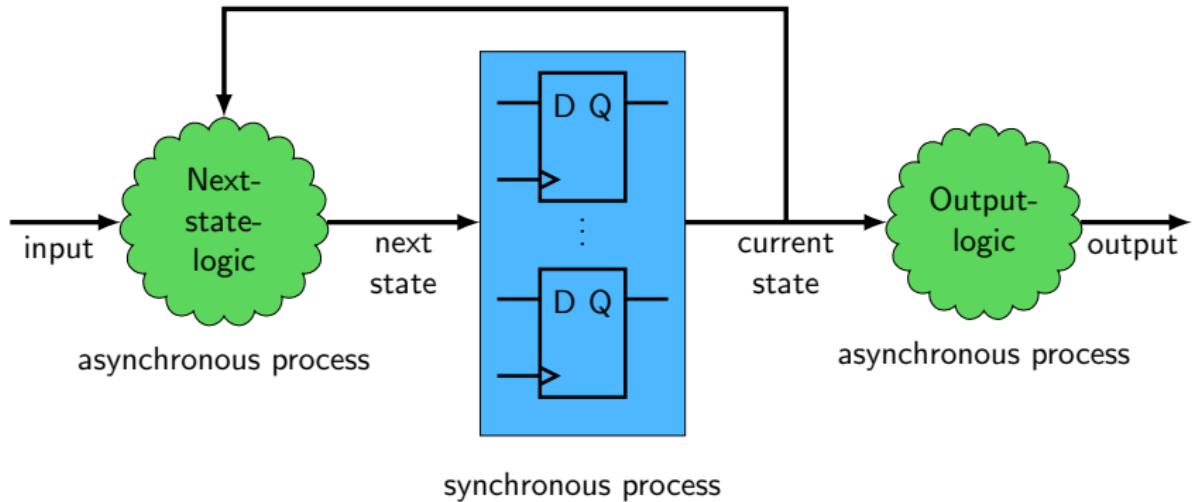
- states have to be encoded
- defining constants cumbersome and not optimal

enumeration

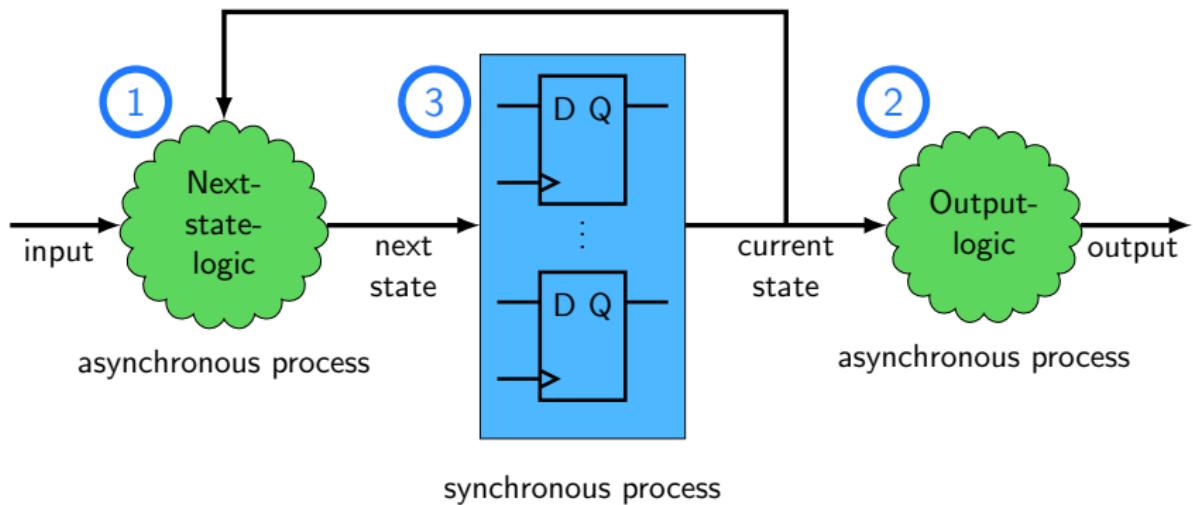
- define allowed values: **type name is (possibilities);**
- internally binary encoded
- handled by tool → optimizations

```
1 architecture beh of hello_world is
2   type STATE_TYPE is (H, E, L1, L2, O);
3   signal state : STATE_TYPE;
4 begin
5   ...
```

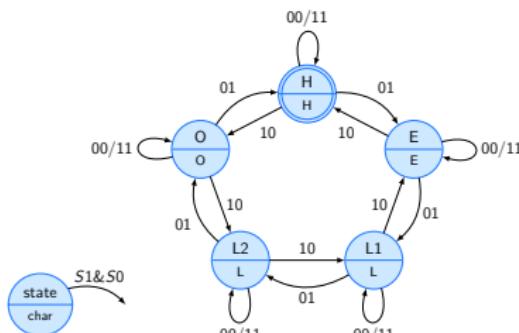
Moore State Machine



Moore State Machine



Next State Logic

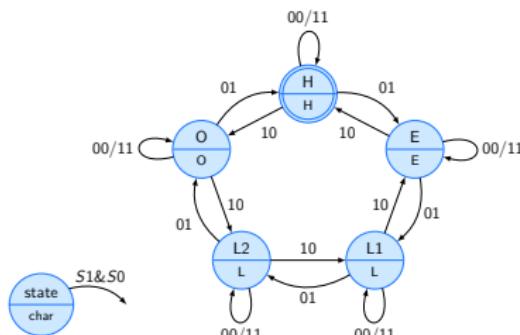


- next state based on input and current state
 - often also current state
- state switch only on clock edge
 - use next state signal

```

1   next_state : process (all)
2   begin
3
4   -- what is missing?
5
6   case state is
7   when H =>
8     if S = "10" then
9       state_next <= O;
10    elsif S = "01" then
11      state_next <= E;
12    end if;
13  when E =>
14    if S = "10" then
15      state_next <= H;
16    elsif S = "01" then
17      state_next <= L1;
18    end if;
19  when L1 =>
20    ...
21  end case;
22 end process;
  
```

Next State Logic



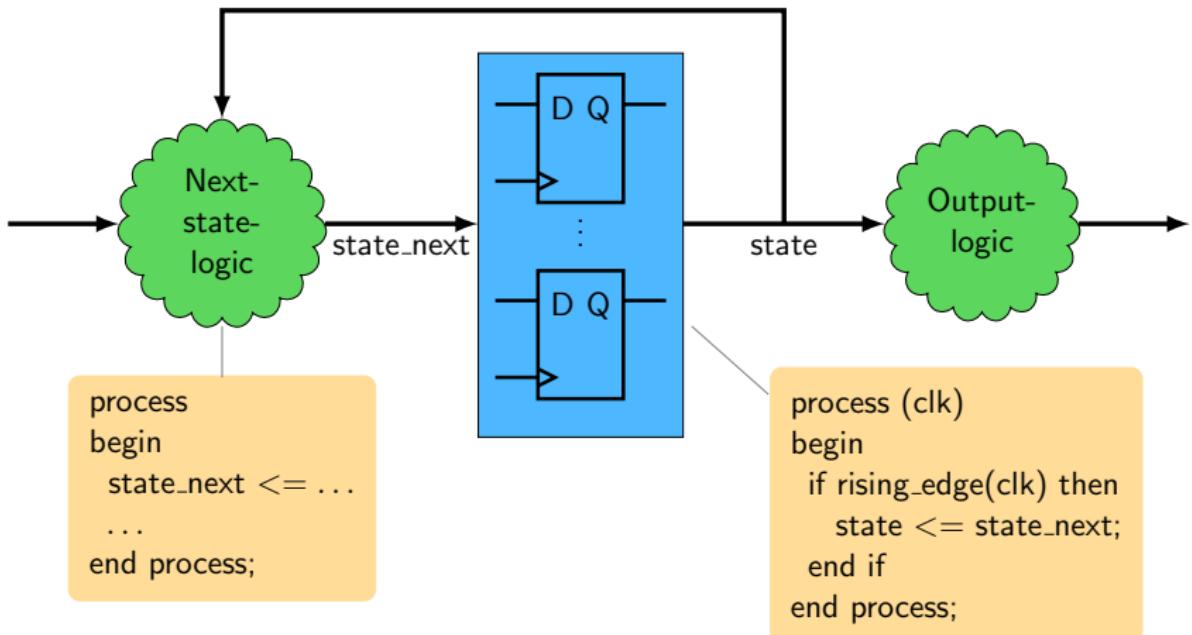
- next state based on input and current state
 - often also current state
 - state switch only on clock edge
 - use next state signal

```

1 next_state : process (all)
2 begin
3
4     state_next <= state;
5
6         case state is
7             when H =>
8                 if S = "10" then
9                     state_next <= O;
10                elsif S = "01" then
11                    state_next <= E;
12                end if;
13                when E =>
14                    if S = "10" then
15                        state_next <= H;
16                    elsif S = "01" then
17                        state_next <= L1;
18                    end if;
19                when L1 =>
20
21                    ...
22
23         end case;
24
25     end process;

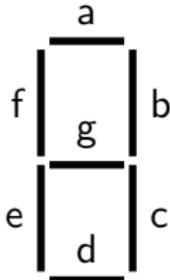
```

Next State Logic cont'd



Output Logic

- pure logic that determines output value
 - depends solely on current state
- letters have to be encoded for display
 - 0 means segment is lightened
 - 1 means segment dark
- $O \leq abcdefg$



```
54     output : process (all)
55 begin
56 case state is
57 when H =>
58     disp <= "1001000";
59 when E =>
60     disp <= "0110000";
61 when L1 =>
62     disp <= "1110001";
63 when L2 =>
64     disp <= "1110001";
65 when O =>
66     disp <= "0000001";
67 when others =>
68     disp <= "1111111";
69 end case;
70 end process;
```

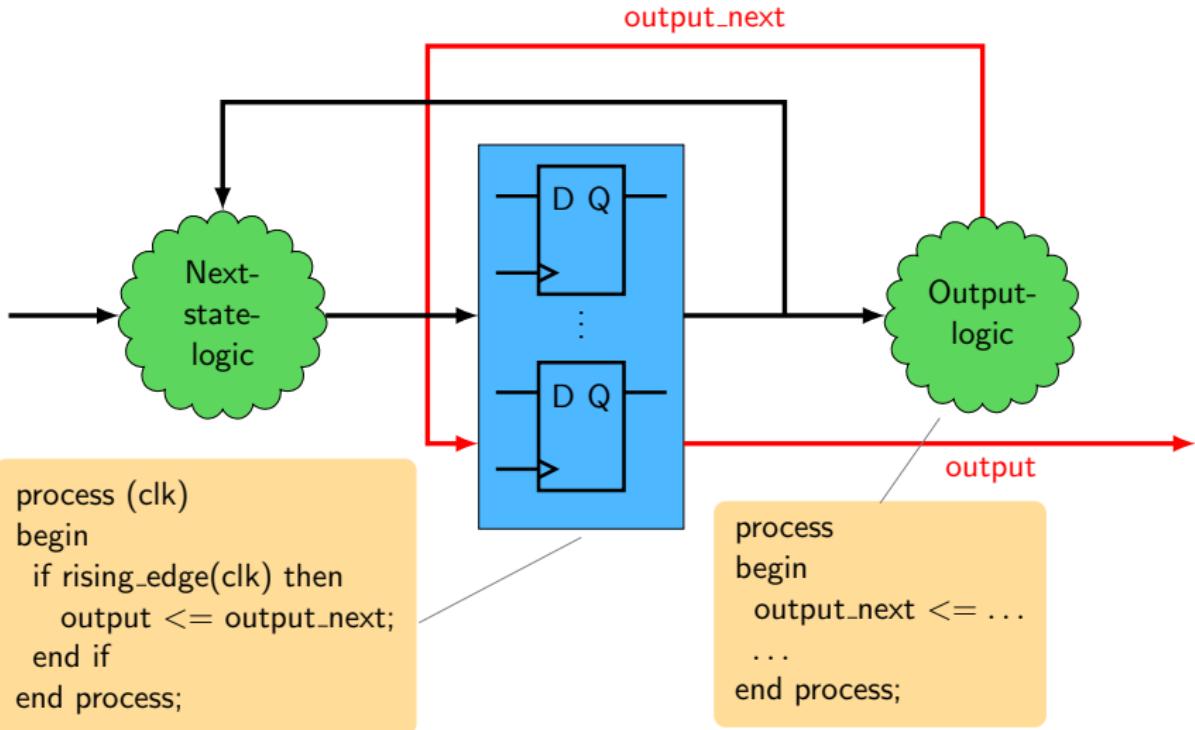
[hello_world/src/eaHello_world.vhd](#)

Output Logic cont'd

Why no next signals here?

Output Logic cont'd

Why no next signals here?



Synchronous Logic

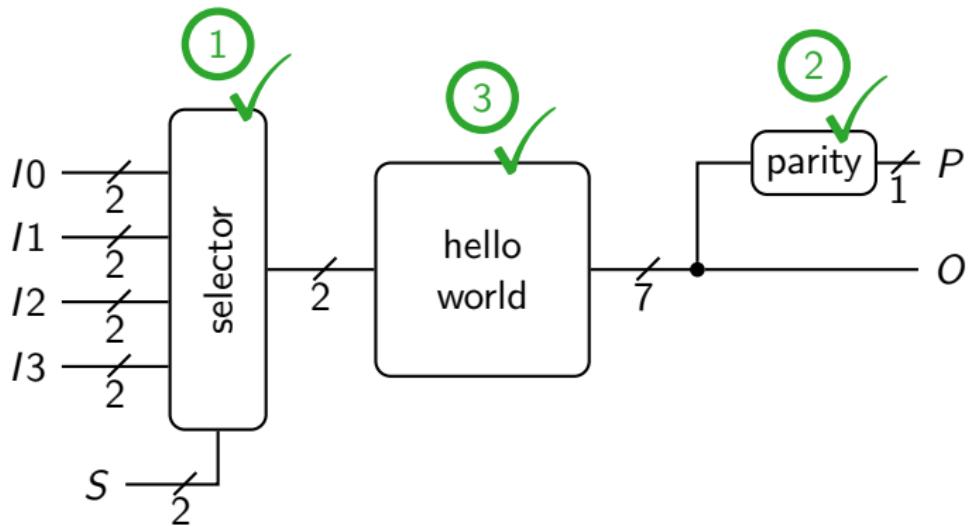
- state changes synchronous to clock
 - simply creates a register
- state_next signal prepared in next state logic
 - simply assign it to state at clock edge
- proper initialization required
 - low active asynchronous reset used

```
1  sync: process (all)
2  begin
3    if res = '0' then
4      state <= H;
5    elsif rising_edge(clk) then
6      state <= state_next;
7    end if;
8  end process;
```

Overview

- three processes required
 - synchronous, output and next state logic
 - called the “three process method”
- combining them may be beneficial
 - two process: merge output and next state logic
 - one process: merge all three
- for start two or three process method recommended
 - direct mapping of process to state machine
- requirements have impact on chosen style
 - registered / unregistered outputs
 - will be elaborated in detail later

First Design – Running Light



Complete Application

```
22 entity top is
23   port (
24     I0, I1, I2, I3 : in STD_LOGIC_VECTOR(1 downto 0);
25     sys_clk, sys_res : in STD_LOGIC;
26     S : in STD_LOGIC_VECTOR(1 downto 0);
27     O : out STD_LOGIC_VECTOR(6 downto 0);
28     P : out STD_LOGIC
29   );
30 end entity;
```

[hello_world/src/top.vhd](#)

Complete Application cont'd

```
32 architecture beh of top is
33
34 signal selection : STD.LOGIC.VECTOR(1 downto 0);
35 constant mode_width : NATURAL := 2;
36 constant data_width : NATURAL := 7;
37 signal hello_world_O : STD.LOGIC.VECTOR(data_width-1 downto 0);
38
39 component MUX41 is
40 generic ( width : NATURAL := 1);
41 port (I0, I1, I2, I3: in STD.LOGIC.VECTOR(width-1 downto 0);
42 S : in STD.LOGIC.VECTOR(1 downto 0);
43 O : out STD.LOGIC.VECTOR(width-1 downto 0));
44 end component;
45
46 component parity is
47 generic ( width : NATURAL := 1);
48 port (
49 I : in STD.LOGIC.VECTOR(width-1 downto 0);
50 O : out STD.LOGIC);
51 end component;
52
53 component hello_world is
54 generic ( width : NATURAL := 1);
55 port (
56 clk, res : in STD.LOGIC;
57 mode : in STD.LOGIC.VECTOR(1 downto 0);
58 disp : out STD.LOGIC.VECTOR(width-1 downto 0)
59 );
60 end component;
61
62 begin
```

[hello_world/src/top.vhd](#)

Complete Application cont'd

```
62  begin
63
64      M : MUX41
65      generic map(width => mode_width)
66      port map(
67          I0 => I0,
68          I1 => I1,
69          I2 => I2,
70          I3 => I3,
71          S => S,
72          O => selection);
73
74      RUN : hello_world
75      generic map (width => data_width)
76      port map (
77          clk => sys_clk,
78          res => sys_res,
79          mode => selection,
80          disp => hello_world_O);
81
82      PAR : parity
83      generic map (width => data_width)
84      port map(
85          I => hello_world_O,
86          O => P);
87
88      O <= hello_world_O;
89
90  end architecture;
```

[hello.world/src/top.vhd](#)

Problem

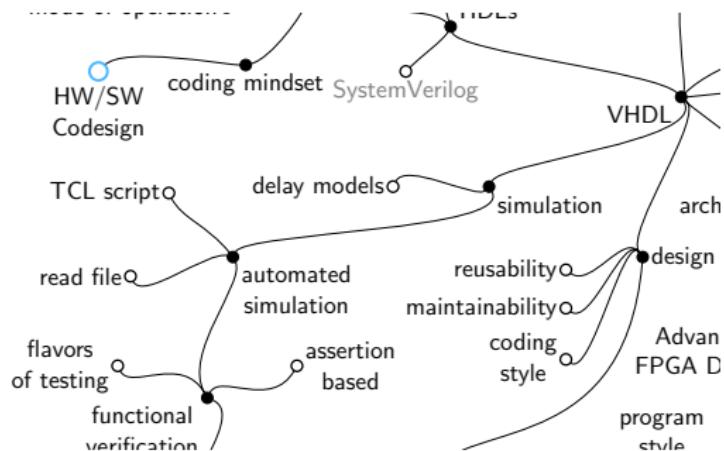
- letters change far too quickly
 - clock frequency of 50 MHz
- easy solution: counter
 - increment at each clock tick
 - will see other methods later

```
38     sync: process (clk, res)
39         variable cnt : unsigned (25 downto 0) := (others => '0');
40 begin
41     if res = '0' then
42         state <= H;
43     elsif rising_edge(clk) then
44
45         if cnt(25) = '1' then
46             state <= state_next;
47             cnt := (others => '0');
48         else
49             cnt := cnt + 1;
50         end if;
51     end if;
52 end process;
```

[hello_world/src/eaHello_world.vhd](#)

VHDL

Simulation



Improved Testbenches

Before:

```
TB_S <= transport "00" after 0 ns, "01" after 1 ns, "10" after 3 ns, "11" after 6 ns;
```

[simulation_I/testbench.vhd](#)

Now:

- processes can be used to enhance testbenches
 - previously concurrent statements used to generate input signals
 - alternative: assignments combined with **wait**
- typically the following in repetition
 - apply input data
 - **wait for time**;

```
44      process
45        begin
46          TB_S <= "00";
47          wait for 1 ns;
48
49          TB_S <= "01";
50          wait for 2 ns;
51
52          TB_S <= "10";
53          wait for 3 ns;
54
55          TB_S <= "11";
56          wait;
57      end process;
```

[simulation_II/testbench.vhd](#)

Clock Generation

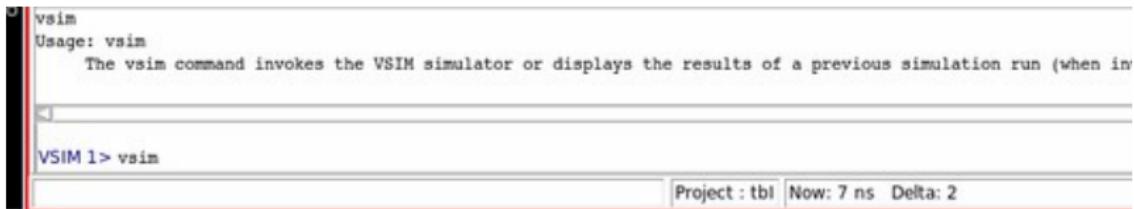
- clock generation required for synchronous designs
- good practice to use constants for clock period
 - **constant** clk_period : TIME := *time*;
- method #1: concurrent assignment
 - `clk <= not clk after clk_period / 2;`
- method #2: process
 - assignments combined with **wait for**

```
59      process
60        begin
61          TB_clk <= '1';
62          wait for clk_period / 2;
63          TB_clk <= '0';
64          wait for clk_period / 2;
65      end process;
```

[simulation_II/testbench.vhd](#)

TCL scripts

- everything in Questa can be run with commands / automated
 - TCL language used
- for more details see
 - Questa User/Reference Manual
 - Transcript window at bottom (gives suggestions)
- run commands ...
 - directly in Transcript window
 - from file
- store commands in .do file and add to project
 - run with right click → run



```
vsim
Usage: vsim
      The vsim command invokes the VSIM simulator or displays the results of a previous simulation run (when in

VSIM 1> vsim
Project : tbl Now: 7 ns Delta: 2
```

TCL scripts cont'd

- simple script to run testbench for MUX
 - compile project
 - define testbench
 - add signals to wave
 - zoom on interesting range
 - run simulation
- lots of options for **add wave**
 - radix: data representation
 - label: signal name
- signals can be set from script
 - see **force**

```
2  project compileall
3
4  vsim -novopt work.testbench
5
6  add wave TB_I0
7  add wave -label I1 TB_I1
8  add wave TB_I2
9  add wave -radix decimal TB_I3
10
11 add wave TB_S
12 add wave TB_SEL
13 add wave TB_COND
14 add wave TB_clk
15
16 wave zoom range {0 ps} {7000 ps}
```

[simulation.ll/run.do](#)

Report

- write messages to Transcript window
 - adds automatically time and instance

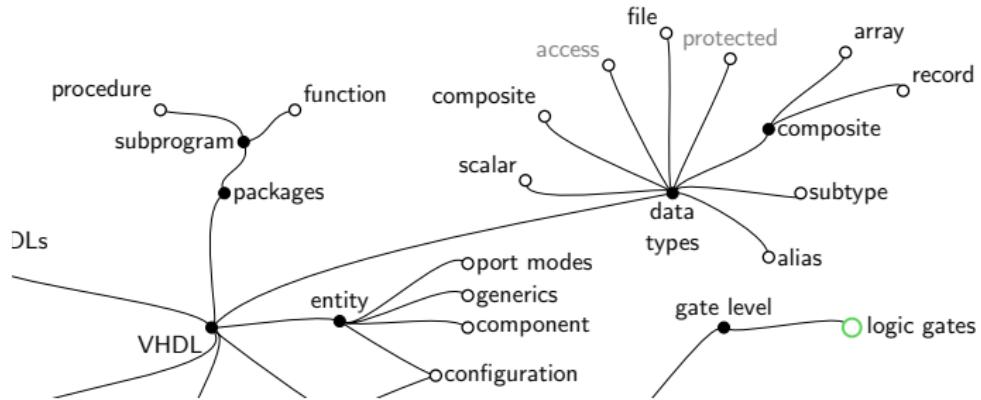
```
# ** Note: Test Message
```

```
# Time: 3 ns Iteration: 0 Instance: /testbench
```

- **report** *string* [**severity** *string*];
 - **report** “Test Message” **severity** WARNING ;
- **severity** sets criticality of message
 - behavior of tools can be adjusted
- Transcript window automatically written to file *transcript*
 - enables automatic post processing of simulation
 - stay tuned for more on that

VHDL

VHDL Internals



Data Types

- strong type system
 - due to strong correlation to Ada
- basic data type categories
 - logic
 - boolean
 - numeric
 - enumeration
 - character
 - physical
 - composite
- easy to create subtypes or new types

Logic Types

277

- STD_LOGIC, STD_ULOGIC, BIT
 - STD_LOGIC is STD_ULOGIC with resolve function

'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'	
'U'									
'U'	'X'								
'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'	'0'
'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'	'1'
'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'	'Z'
'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'	'W'
'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'	'L'
'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'	'H'
'U'	'X'	'-'							

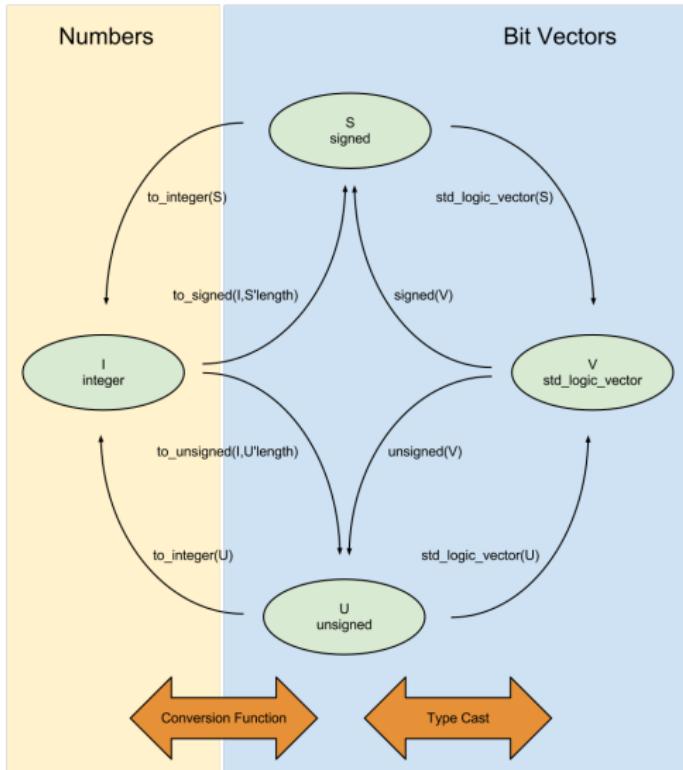
Logic Types cont'd

- constants
 - '0', '1'
 - "01001", b"11001", x"1A", o"31", d"31"
- possible to define as vector (append _VECTOR)
 - range defined either by **downto** or **to**
- assignment by
 - constant: C <= "00010011";
 - position: C <= (0 => '1', **others** => '0');
- access parts of vector
 - single bit: C (5)
 - range: C (5 **downto** 2)

Logic Types cont'd

- concatenation with '&' possible
 - $(A,B) \leq "01";$
 - $C \leq A \& B \& "0110";$
- watch out when comparing!
 - $"110" > "1010"$
 - similar to strings in SW
- just bits → interpretation required
 - signed or unsigned
 - casting required: `signed(vector)`
 - can be used for arithmetic operations
- clock signal restricted to (subtypes) of logic types

Type Conversion



www.bitweenie.com

Type Conversion cont'd

- very important for computations on logic values
 - logic values have no numerical meaning
 - have to be converted first
- first convert to signed/unsigned
 - `signed(.)`, `unsigned(.)`
 - already enough for calculations
 - no actual conversion to number required
- only in second step conversion to integer
 - `to_integer(.)`
 - when converting back length has to be provided

Boolean Types

- **type** BOOLEAN is (TRUE,FALSE);
 - essentially enumeration
- result of comparing operators
 - demanded where *conditions* are required
- constants
 - TRUE, FALSE
- similar to logic types
 - BOOLEAN_VECTOR defined
 - compare to **type** BIT is ('0','1');

Numeric Types

- INTEGER, REAL
 - 2, 250_000, 3E6
 - 2.0, 2.6E-5, 3_123.56
- value range
 - INTEGER: 32 Bit, -2^{31} to $2^{31} - 1$
 - REAL: host dependent
- can be limited to certain range
 - **signal A : INTEGER range 3 to 50;**
 - **subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;**
- consider size when subtyping
 - **subtype S1 is INTEGER range 10 to 16;** → 5 Bits
 - **subtype S2 is INTEGER range 0 to 6;** → 3 Bits

Enumeration Types

- **type** *name* **is** (*possibilities*);
- textually specify allowed values
 - increases readability and maintainability
- values internally represented in binary format
 - encoding is variable
 - in FPGA one-hot often better
- encoding can be specified manually
 - using attributes (more on that later)
 - not recommended as tools try optimizations

Character Types

- CHARACTER, STRING
- constants
 - 'a', 'n'
 - "text"
- concatenation with &
- required for example by **report**
- conversion of other types to string
 - function **to_string()** not supported by every type
 - attribute 'IMAGE'

Examples

```
report "delay: " & time'IMAGE(delay);
```

Physical Types

- allow usage of unit with value
 - ns, us, ms, sec ... for time
 - um, mm, cm, m ... for length
- internally integer value stored
 - multiple of primary unit
- other units use as abbreviations
 - apply appropriate multiplication

```
1 type DURATION is range -1E18 to 1E18
2 units
3   fs; --femtosecond
4   ps = 1000 fs; --picosecond
5   ns = 1000 ps; --nanosecond
6   us = 1000 ns; --microsecond
7   ms = 1000 us; --millisecond
8   sec = 1000 ms; --second
9   min = 60 sec; --minute
10 end units;
```

Composite Types

Array

- group multiple elements of same type
 - **type NAME is array (range) of TYPE_NAME;**
- multi-dimensional arrays possible
- access elements as in vectors (actually vectors are arrays)

Examples

```
type MY_BYTE is array (0 to 7) of BIT;  
type MY_WORD is array (INTEGER range <>) of BIT;  
type MY_MEMORY is array (INTEGER range <>, 0 to 31)  
of STD_LOGIC;
```

Composite Types cont'd

Record

- group arbitrary types together
 - **type NAME is record** *types* **end record;**
- access single elements by name

Examples

```
type DATE is
  record
    DAY : INTEGER range 1 to 31;
    MONTH : INTEGER range 1 to 12;
    YEAR : INTEGER range 0 to 4000;
  end record;
```

```
signal d : DATE;
d.DAY <= 15;
```

Attributes

95

- properties of signals/variables/types
 - invoked by '
- possibilities depend on applied value/type
- range based attributes
 - LEFT, RIGHT, LOW, HIGH, RANGE
- signal based attributes
 - EVENT, STABLE
- user defined attributes also possible

Examples

```
if clk'EVENT and clk = '1' then  
  ...  
end if;
```

Attributes cont'd

signal A : STD_LOGIC_VECTOR(3 **downto 0)**

A'LEFT → 3 A'LOW → 0 A'RANGE → 3 **downto** 0
A'RIGHT → 0 A'HIGH → 3 A'LENGTH → 4

signal B : STD_LOGIC_VECTOR(0 **to 3)**

B'LEFT → 0 B'LOW → 0 B'RANGE → 0 **to** 3
B'RIGHT → 3 B'HIGH → 3 B'LENGTH → 4

Alias

- create acronym for (parts of) variable /signal
 - comparable to pointer in other programming languages
- also shortcut for longer names

Examples

```
variable real_number : BIT_VECTOR (0 to 31);  
  
alias sign : BIT is real_number(0);  
alias mantissa : BIT_VECTOR (23 downto 0) is  
    real_number (8 to 31);  
  
alias "nor" is STD.STANDARD."nor" [STD.STANDARD.BIT,  
    STD.STANDARD.BIT  
    return STD.STANDARD.BIT];
```

Operators

118

precedence	category	operators
1)	logical	and or nand nor xor xnor
2)	relational	= / = < <= > >=
3)	shift	sll srl sla sra rol ror
4)	adding	+ - &
5)	sign	+ -
6)	multiplying	* / mod rem
7)	miscellaneous	** abs not

higher precedence operations get executed first

Subprograms

- remember parity function for arbitrary length
- might be required in other unit as well
 - has to be recoded in every unit
 - bad maintainability, very error prone
- better do it once and reuse code
 - functions & procedures
 - may be defined in entity, architecture, package, ...

```
31      calc : process (all)
32          variable parity : STD_LOGIC;
33      begin
34          parity := '0';
35          for idx in l'RANGE loop
36              parity := l(idx) xor parity;
37          end loop;
38          O <= parity;
39      end process;
```

[eaParity_for_range.vhd](#)

Function

- zero, one or multiple arguments
- returns exactly one value
- can be used in places where its return type may be used
- explicit return statement required

```
29  architecture beh of parity is
30      function get_parity(I : STD_LOGIC_VECTOR) return STD_LOGIC is
31          variable parity : STD_LOGIC;
32      begin
33          parity := '0';
34          for idx in I'RANGE loop
35              parity := I(idx) xor parity;
36          end loop;
37          return parity;
38      end function;
39  begin
```

[eaParity.function.vhd](#)

Function

- zero, one or multiple arguments
- returns exactly one value
- can be used in places where its return type may be used
- explicit return statement required

```
40      calc : process (all)
41      begin
42          O <= get_parity(I);
43      end process;
```

[eaParity_function.vhd](#)

Function

- zero, one or multiple arguments
- returns exactly one value
- can be used in places where its return type may be used
- explicit return statement required

```
1 function identifier [(parameter_list)] return type is
2   declarations
3 begin
4   function_body
5 end [ function ] [ identifier ];
```

Procedure

- bidirectional arguments (call by reference)
- can be used in places where statement may be used
- terminates at return / end of function

```
29  architecture beh of parity is
30    procedure calc_parity(
31      signal l : in STD_LOGIC_VECTOR;
32      variable parity : inout STD_LOGIC) is
33    begin
34      parity := '0';
35      for idx in l'RANGE loop
36        parity := l(idx) xor parity;
37      end loop;
38    end procedure;
39  begin
```

[eaParity_procedure.vhd](#)

Procedure

- bidirectional arguments (call by reference)
- can be used in places where statement may be used
- terminates at return / end of function

```
40      calc : process (all)
        variable parity : STD_LOGIC;
        begin
            calc_parity(I, parity);
            O <= parity;
        end process;
```

[eaParity_procedure.vhd](#)

Procedure

- bidirectional arguments (call by reference)
- can be used in places where statement may be used
- terminates at return / end of function

```
1 procedure identifier [(parameter_list)] is
2   declarations
3   begin
4     procedure_body
5   end [ procedure ] [ identifier ];
```

Subprograms Comparison

	Function	Procedure
argument direction	input	input, output or both
argument types	constants, signals	constants, signals, variables
return value(s)	exactly one	arbitrary many
usable where ...	its return type may be used	a statement may be used
return statement	necessary	optional

Packages

- defining **component** in each architecture cumbersome
 - collect in single place
 - import wherever needed (cp. header files in C)
- **package** consists of
 - package declaration
 - package body (required for implementations of subprograms)
- may contain various information
 - constants, components, subprograms, custom datatypes, ...
- make visible with **use** work.name.all;

```
1 package identifier is
2   package_header
3   package_declarative_part
4 end [ package ] [ identifier ];
```

Packages

- defining **component** in each architecture cumbersome
 - collect in single place
 - import wherever needed (cp. header files in C)
- **package** consists of
 - package declaration
 - package body (required for implementations of subprograms)
- may contain various information
 - constants, components, subprograms, custom datatypes, ...
- make visible with **use** work.name.all;

```
1 package body identifier is
2   package_body_declarative_part
3 end [ package body ] [ identifier ];
```

Package Example

```
22 package pMUX41 is
23
24     component MUX41 is
25         generic ( width : NATURAL := 1);
26         port (I0, I1, I2, I3: in STD_LOGIC_VECTOR(width-1 downto 0);
27               S : in STD_LOGIC_VECTOR(1 downto 0);
28               O : out STD_LOGIC_VECTOR(width-1 downto 0));
29     end component;
30
31     constant width : NATURAL := 7;
32
33     type STATE_TYPE is (H, E, L1, L2, O);
34
35     function get_parity(I : STD LOGIC VECTOR) return STD_LOGIC;
36
37 end package ;
```

pMUX41.vhd

Package Example

```
39 package body pMUX41 is
40
41     function get_parity(I : STD_LOGIC_VECTOR) return STD_LOGIC is
42         variable parity : STD_LOGIC;
43     begin
44         parity := '0';
45         for idx in I'RANGE loop
46             parity := I(idx) xor parity;
47         end loop;
48         return parity;
49     end function;
50
51 end package body;
```

[pMUX41.vhd](#)

Libraries

- contain (precompiled) VHDL units, e.g., packages
- have to be made visible before usage
 - **library** *library_name*;
 - i.e. **library** IEEE;
- elements also have to be made visible
 - **use** *library_name.package_name.element_name*
 - select all by choosing *element_name* = all

Examples

```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
use IEEE.numeric_std. "+";
```

Standard Library (std)

package standard

- visible by default
- basic types
 - BIT, BOOLEAN, INTEGER, NATURAL, ...
- operation for basic types
 - **and, or, not**, ...
 - =, / =, <, ...

package textio

- not visible by default
- textmode file operations

IEEE Library

package std_logic_1164

- types: STD_LOGIC, STD_ULOGIC, ...
- operators: **and**, **or**, **not**, ...
- functions: **rising_edge(.)**, **falling_edge(.)**, ...

package numeric_std

- types: SIGNED, UNSIGNED
- operators: +, -, *, <, >, ...
- conversion functions: **to_integer(.)**, ...
- use instead of std_logic_arith

VHDL vs. Verilog

VHDL

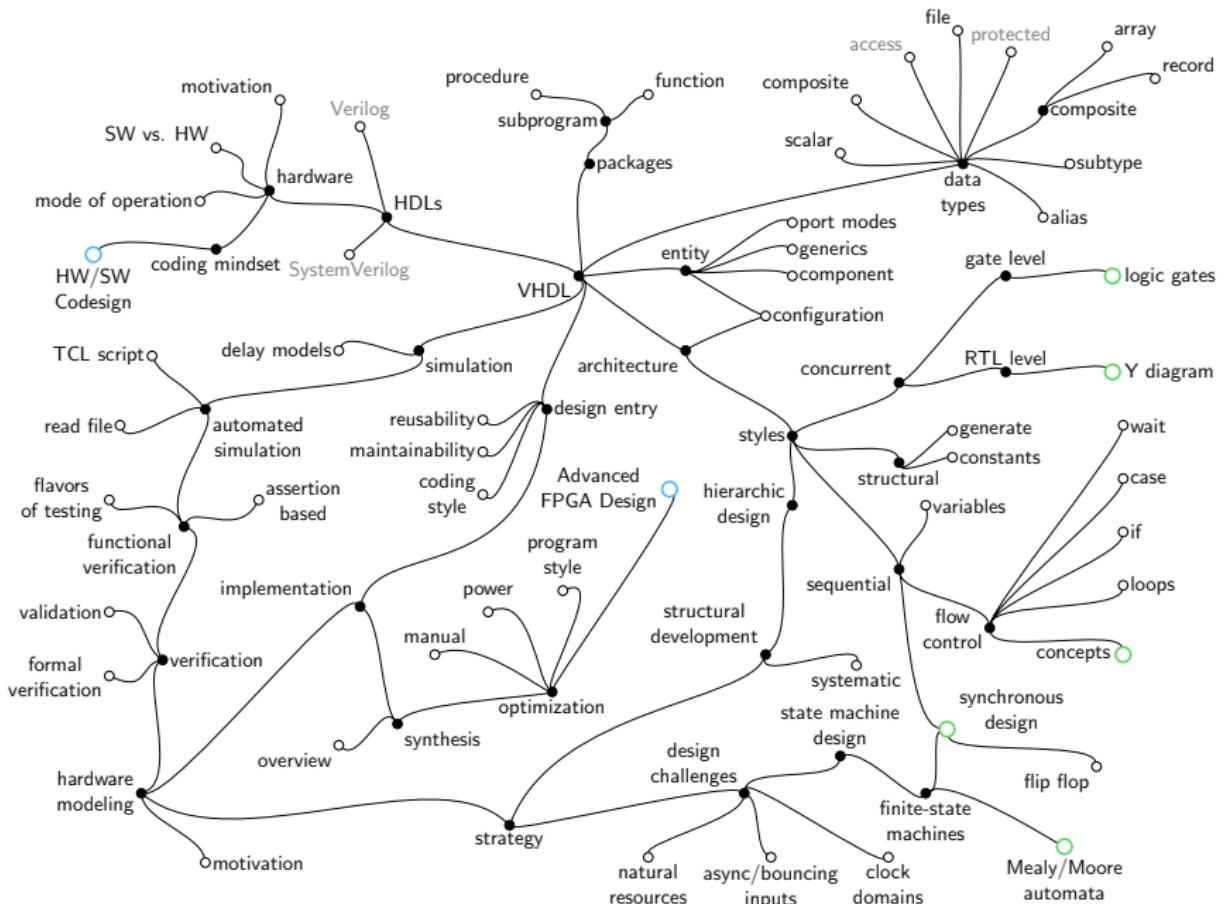
```
library IEEE;
use IEEE.std_logic_1164.all;

entity flip_flop is
port (
    clk , res: in STD_LOGIC;
    D : in STD_LOGIC;
    Q : out STD_LOGIC);
end entity;

architecture res2 of flip_flop is
begin
    sync : process (clk, res)
    begin
        if rising_edge(clk) then
            if res = '0' then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end architecture;
eaFF_res2.vhd
```

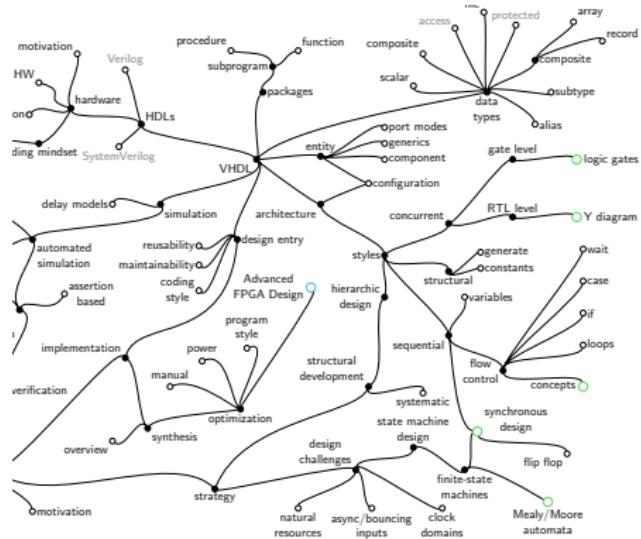
Verilog

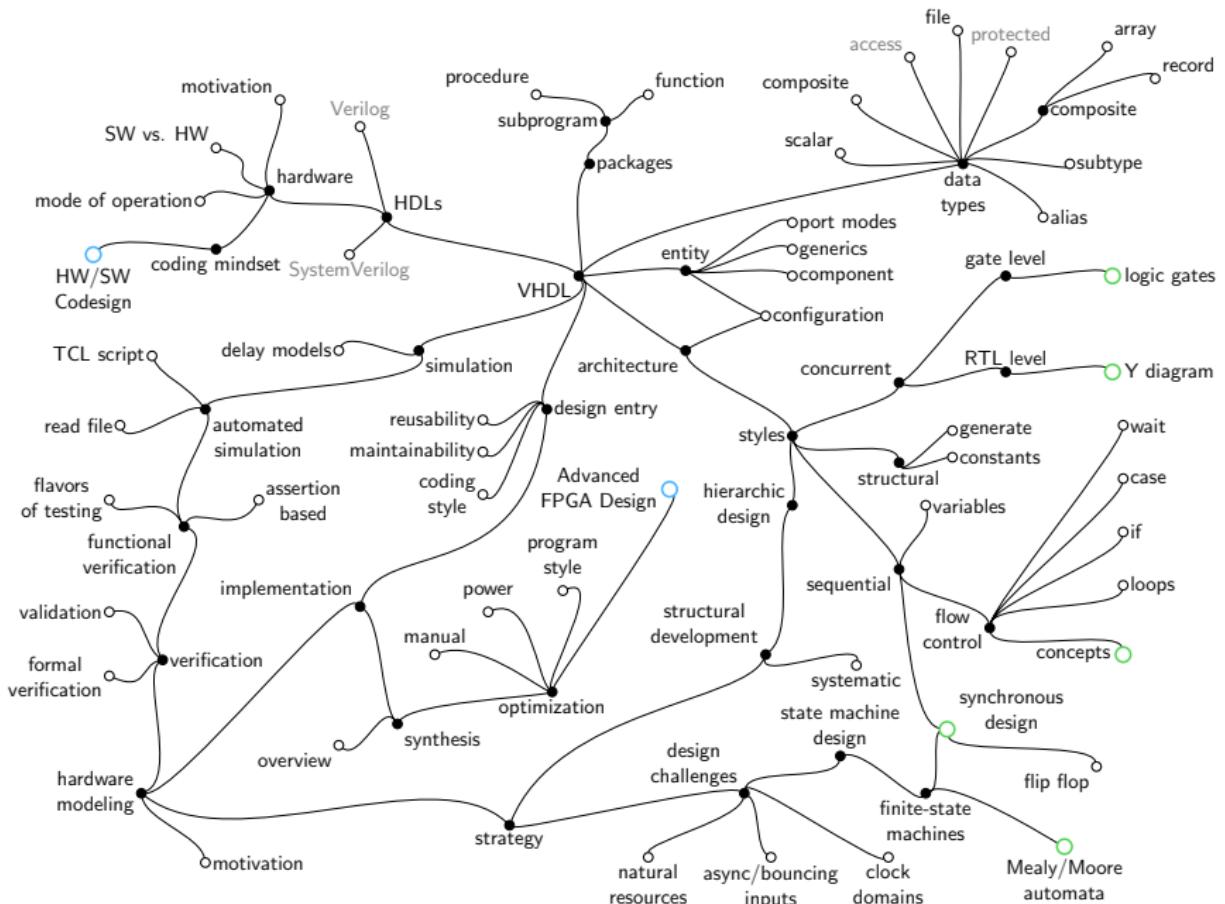
```
moduledff_sync_rst (data, clk, reset, q);
    input data, clk, reset;
    output q;
    reg q;
    always @ (posedge clk)
        if (~reset)
            q = 1'b0;
        else q = data;
endmodule
```



Hardware Modeling

A look back





Safety Check

writing code in VHDL

1. How comfortable do I feel regarding this topic?
 - 0 worst, 10 best

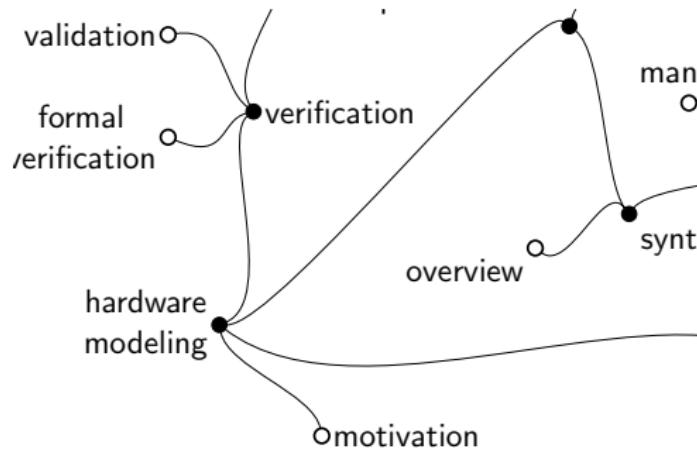
Safety Check

writing code in VHDL

1. How comfortable do I feel regarding this topic?
 - 0 worst, 10 best
2. Why?
3. What do I need to get further towards 10?

Hardware Modeling

Motivation



Hardware modeling

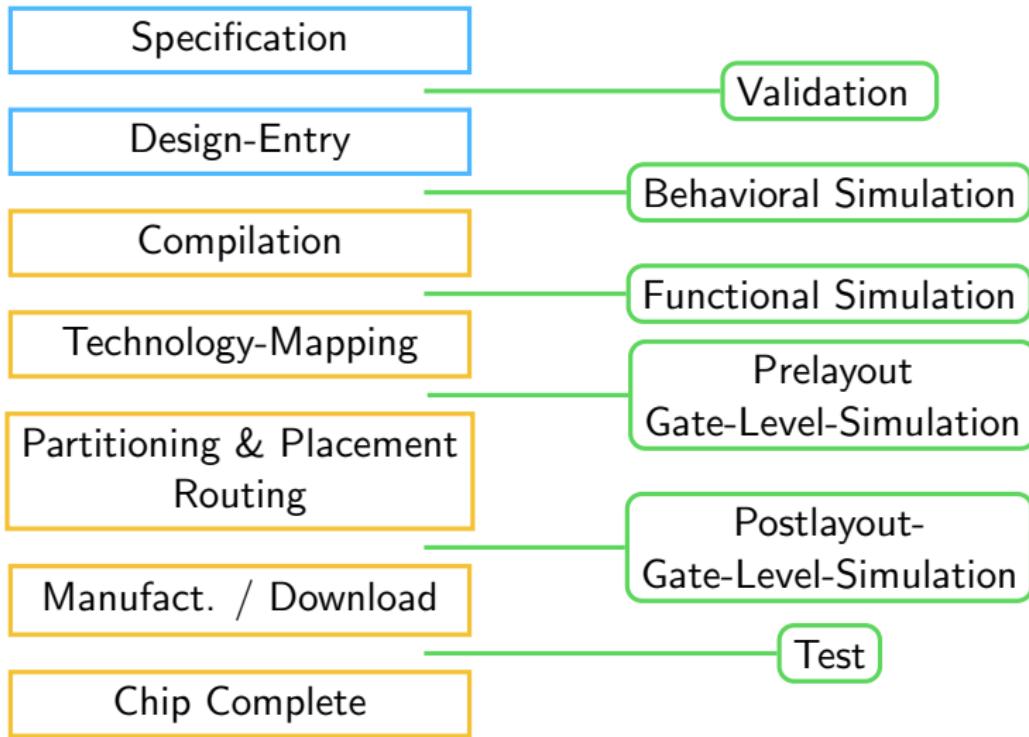
so far: how implement design (VHDL)

- concurrent / structural / sequential design style
- process, control flow statements, variables
- state machines, data types, subprograms
- ...

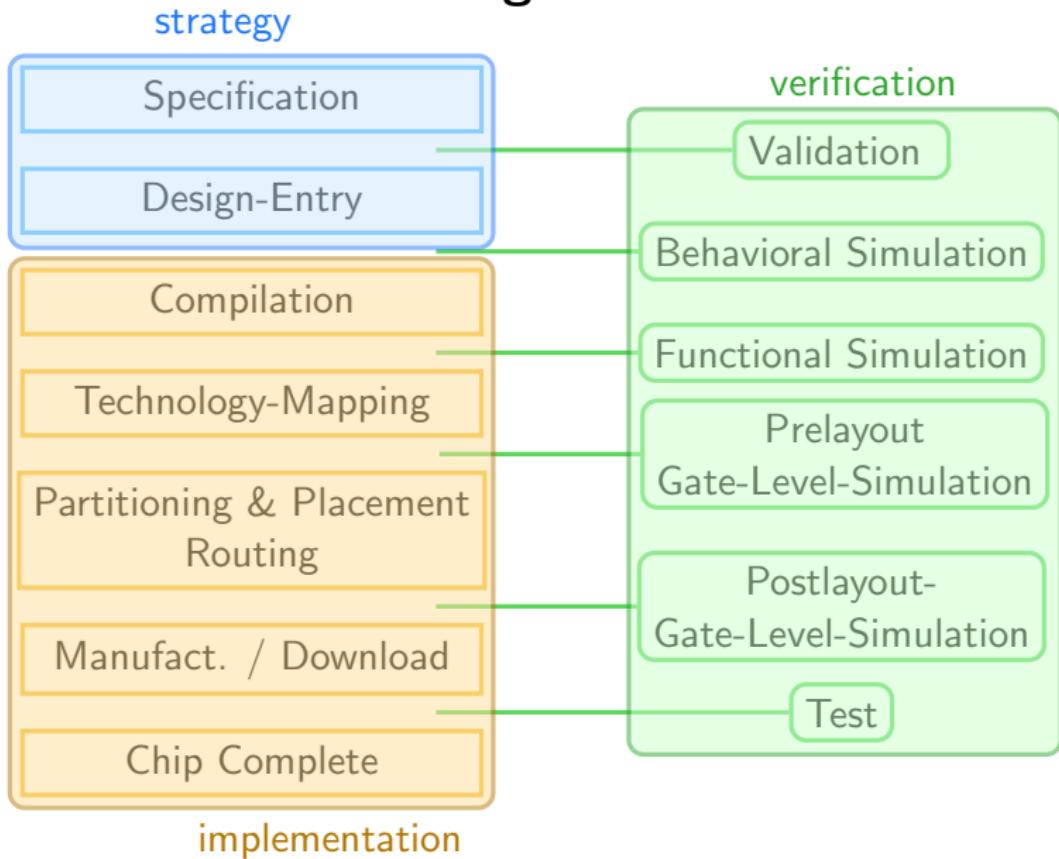
now: how achieve proper result

- strategy
- implementation
- verification

Design Flow



Design Flow



The Problem

- strategy
 - analysis of task and how it can be realized
- implementation
 - coding (VHDL, Verilog, ...)
- verification
 - validation, functional / formal verification

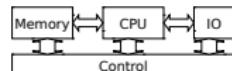
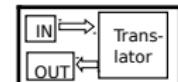
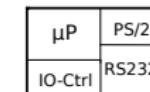
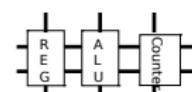
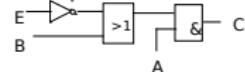
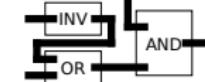
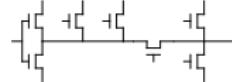
The Problem

- strategy
 - analysis of task and how it can be realized
 - should be **1-5 %** effort
- implementation
 - coding (VHDL, Verilog, ...)
 - should be **25-30 %** effort
- verification
 - validation, functional / formal verification
 - should be **70-80 %** effort

The Problem

- strategy
 - analysis of task and how it can be realized
 - should be **1-5 %** effort
 - usually is **1-5 %**
- implementation
 - coding (VHDL, Verilog, ...)
 - should be **25-30 %** effort
 - usually is **85-90 %**
- verification
 - validation, functional / formal verification
 - should be **70-80 %** effort
 - usually is **5-10 %**

The Problem cont'd

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

The Problem cont'd

- projects often started in complete wrong fashion
 - eager in the beginning
 - start coding immediately without careful analysis
- leads almost certainly to problems
 - the later the more complicated and harder to resolve
 - ends up in high frustration
- result is very complicated code
 - error prone
 - difficult to maintain
- systematic design approach indispensable
 - improves quality
 - reduces costs

Goal: Systematic Design

- applicable to all kind of development tasks (SW and HW)
- will be required in later stages of your studies
 - e.g. HW/SW Codesign
- tools are code generators
 - very advanced, can help a lot but:
 - o lots of parameters
 - o have to be handled appropriately
- real world is not optimal
 - unfavorable properties have to be handled appropriately
- poor verification leads to bugs
 - prominent examples, e.g. FDIV
 - is often disregarded part in design

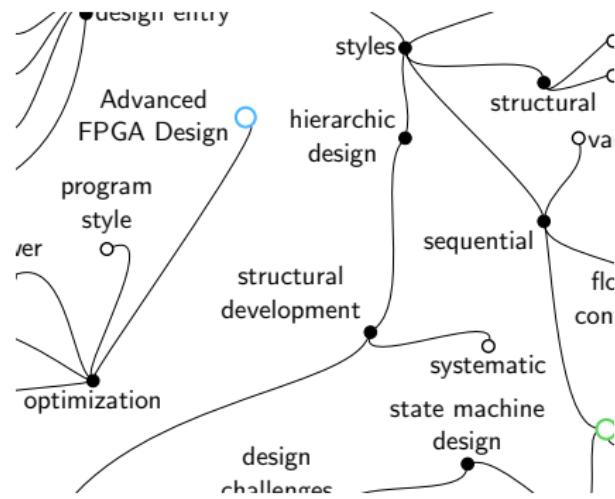
Case Study

- Snake
 - addictive mobile phone game in 1990s-2000s
 - constant movement of snake
 - single dots have to be consumed
 - length increases with each dot
 - with / without borders
- Goal
 - collect as many dots
 - do not hit own tail (respectively border)

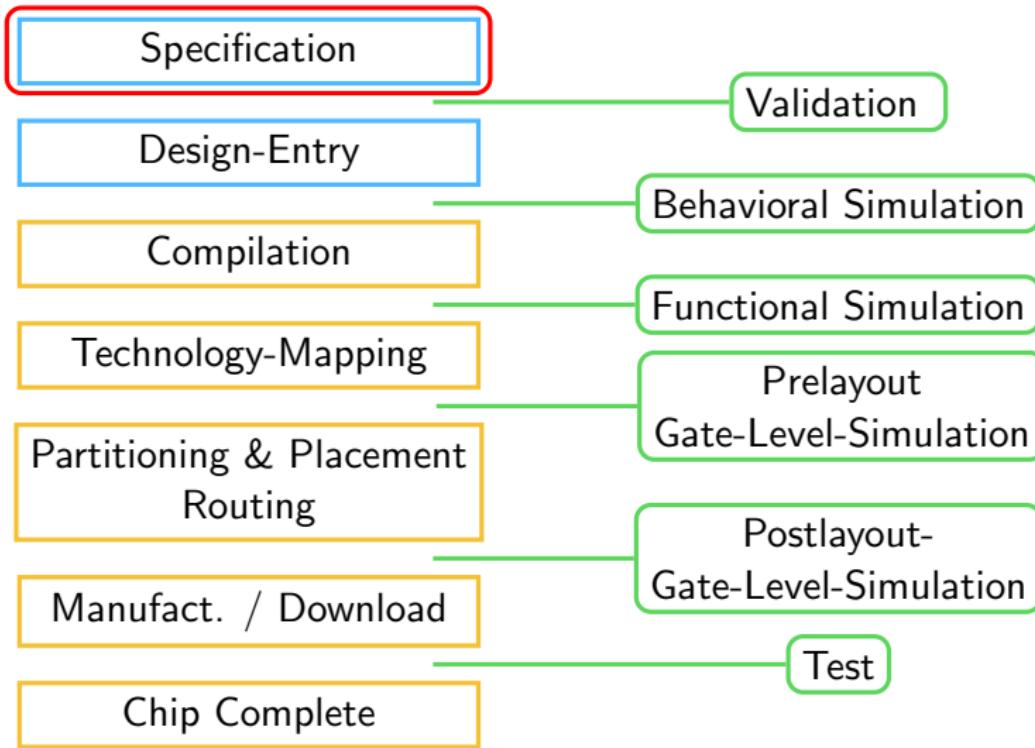


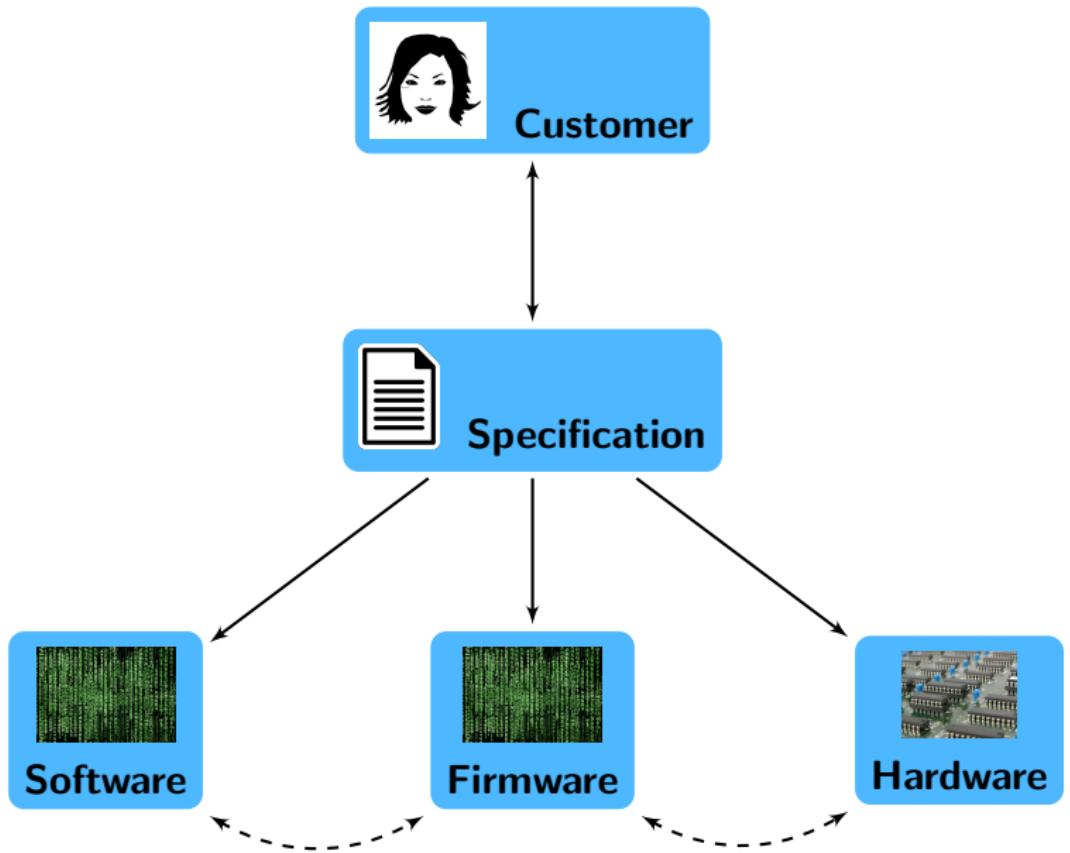
Hardware Modeling

Structural Development



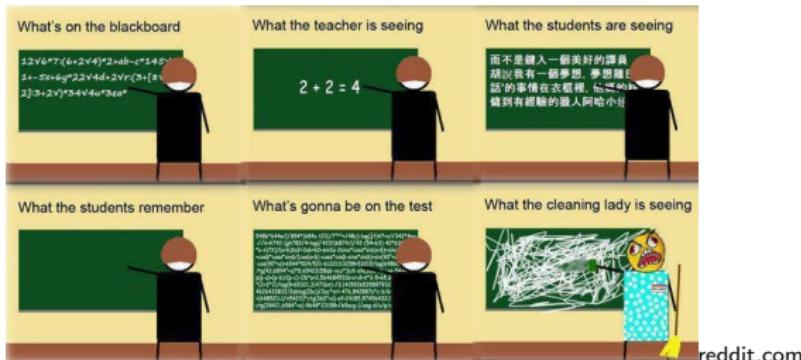
Design Flow





Customer

- assume no knowledge
 - makes communication very cumbersome
 - hard to make statements bullet proof
- has its own ideas and goals
 - task of developer to find out what customer wants
 - map to possible solutions
- important to find common ground
 - especially important for specification



Specification

- description of task to solve
- clearly state problem
 - used to communicate with customer / colleague
 - should be commonly understandable but unambiguous
 - How can this be done?
- reference for later development steps
 - validate that product according to specification
- be very specific
 - functionality, user interface, capabilities...
 - do not include implementation details
- indispensable part of design process

Specification Case Study

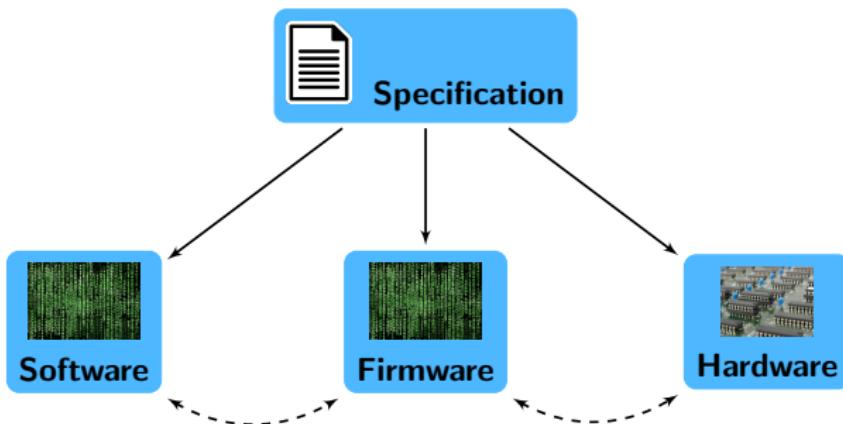
- implement snake on FPGA
- steer with four keys
 - up, down, left and right
 - accept input from keyboard, UART and on-board buttons
- show game on display
 - walls, snake and background have different colors
 - draw snake as '*' and dots as 'o'
- implement replay functionality
 - store game progress in memory
 - possibility to watch later
- snake speed should be adjustable
- when wall hit enter at opposite side

Specification Case Study

- implement snake on FPGA
- steer with four keys
 - up, down, left and right
 - accept input from keyboard, UART and on-board buttons
- show game on display
 - walls, snake and background have different colors
 - draw snake as '*' and dots as 'o'
- implement replay functionality
 - store game progress in memory
 - possibility to watch later
- snake speed should be adjustable
- when wall hit enter at opposite side

bad specification

HW/SW Codesign



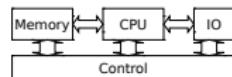
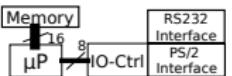
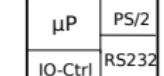
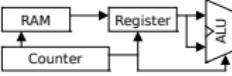
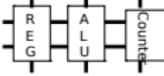
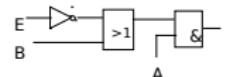
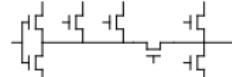
- partitioning hard task
 - will be covered in lecture [182.700 VU HW/SW Codesign](#)
- assume everything done in hardware
 - concentrate on implementation and verification

How to proceed ... ?

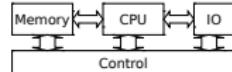
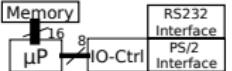
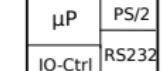
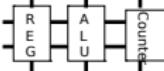
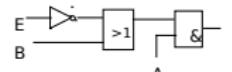
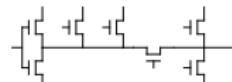
- How do we get a proper implementation for given task?
- wrong development process leads to
 - bad code quality
 - frustration and exhaustion
 - failing to implement the task
 - ...
- learning by doing painful
- generally applicable to all kinds of development tasks



... Jump right in

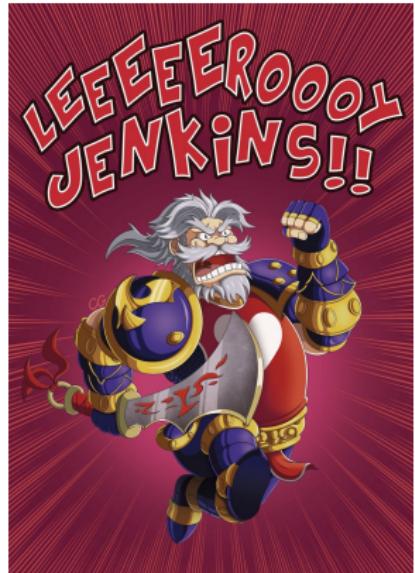
	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B:= B+1 else B:= B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dl}{dt} + \frac{l}{C} + L \frac{d^2l}{dt^2}$		

... Jump right in

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	<pre> if A='1' then B:= B+1 else B:= B end if </pre>		
Logic Level	$D = \text{NOT } E$ $C = (D \text{ OR } B) \text{ AND } A$		
Circuit Level	$\frac{dU}{dt} = R \frac{dl}{dt} + \frac{l}{C} + L \frac{d^2l}{dt^2}$		

... Jump right in cont'd

- very enthusiastic in the beginning
 - no initial planning → start coding right away
 - immediately working on RTL and logic level
- not a good way to do things
 - no clear structure results in complicated code
 - bad readability / maintainability
- most certainly heavy problems experienced later
 - leads to compromises / redesigns
 - the later detected the higher the costs
- better systematic approach



twitter.com

Systematic Approach

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read "Schilling" Calulate Euro Display „Euro“		
Register Transfer Level (RTL)	if A='1' then B := B+1 else B := B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

Systematic Approach cont'd

- start at high abstraction level
 - behavioral and structural view in parallel
 - geometry of no importance
- analyze task in detail on different abstraction levels
 - what are the challenges?
 - which tradeoffs have to be made?
- gradually refine design (divide and conquer)
 - split function blocks in smaller, simpler subblocks
 - guided by specification and constraints
- only explorative coding in this phase

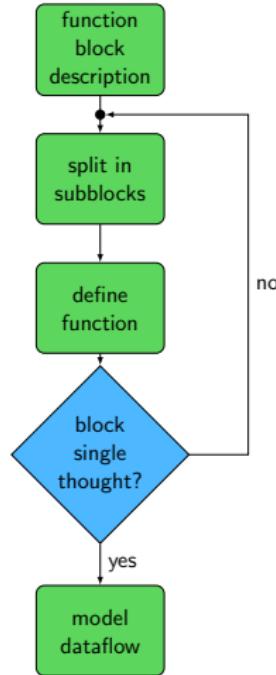
Flat Structure

- implement everything in single entity
- confusing code leads to
 - bad readability and maintainability
 - deeply nested (spaghetti) code
 - hard verification (equivalence of generated hardware)
- no code reusability on entity level
 - appropriate sections of code have to be exported
- **Not recommended at all!**
- use hierarchy of entities

top

Hierarchical Structure

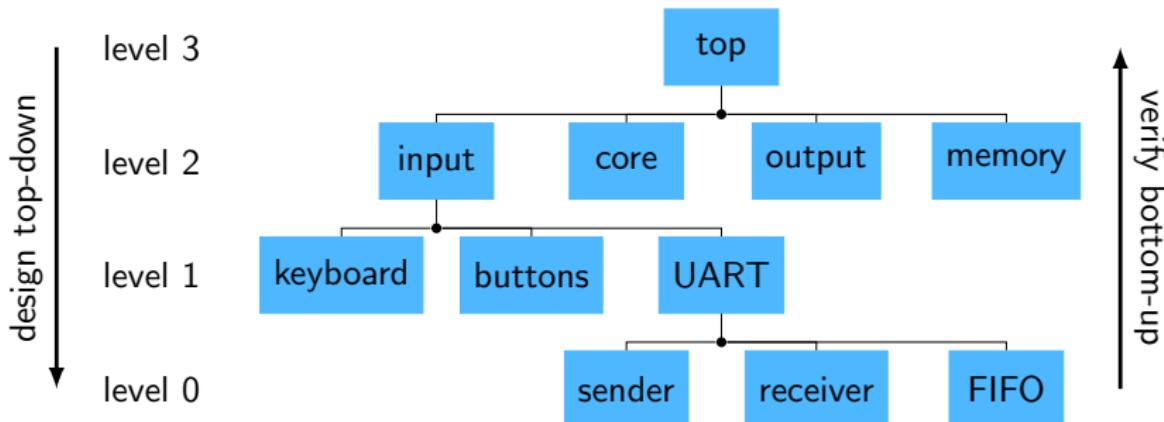
- very natural structure in VHDL
- every function block is entity
 - subblocks are separate entities
 - combined using structural design style
 - interfaces can already be specified
- partitioning repeated until entities “single thought”
 - describe behavior in single sentence
 - do one thing and do it well (cp. [Unix philosophy](#))
- interesting idea: model dataflow by comments in architecture
 - may serve already for documentation





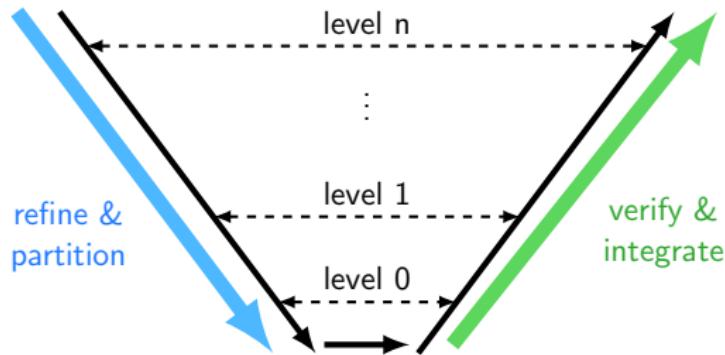
Hierarchical Structure cont'd

- example for our running example (very coarse)
 - in real application much more information necessary
- design top-down
- implement main parts on lowest level
- verification bottom-up



Implementation Strategy

- single, encapsulated blocks with precisely defined interfaces
 - multiple developers can work in parallel
 - verification independently possible
- limited complexity due to partitioning
 - verification simplified → incremental verification possible
 - good code reusability
- verification carried out on each level (V-diagram)



Summary

Design Process

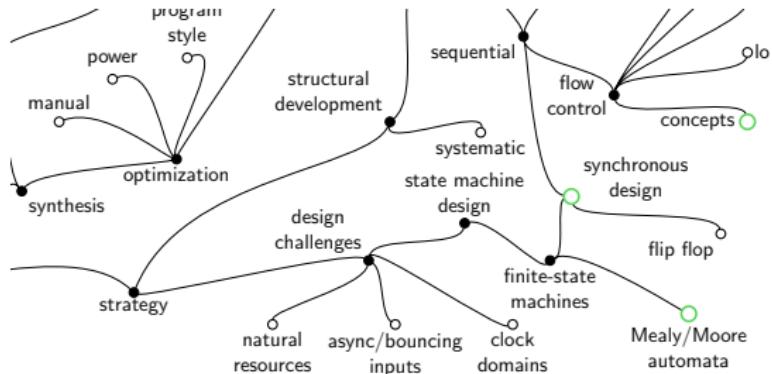
- careful analysis key at start
- create hierachic structure
- refine & partition top-down (divide and conquer)
- implement & verify bottom-up

Advantages

- lower complexity due to smaller size
- lower design effort
- high reusability (libraries)
- simpler implementation, verification and simulation

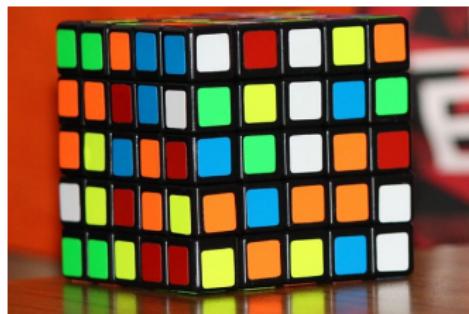
Hardware Modeling

Design Challenges



Design Challenges

- physical reality not optimal
- different challenges that have to be considered at design time
- handling inputs
 - asynchronous events to a synchronous system
- bouncing inputs
 - oscillations of input signal
- differing operation speed of single parts
 - clock domain crossings
- resource requirements
- state machine design



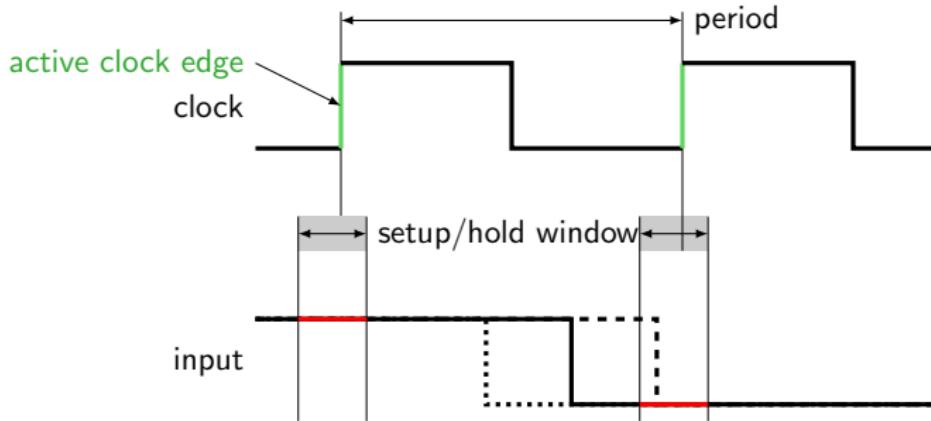


Handling Inputs

- Why are asynchronous inputs a problem?

Handling Inputs

- Why are asynchronous inputs a problem?
- may violate capture / hold time of flip flops
 - metastability possible
- proper resolution of these cases required

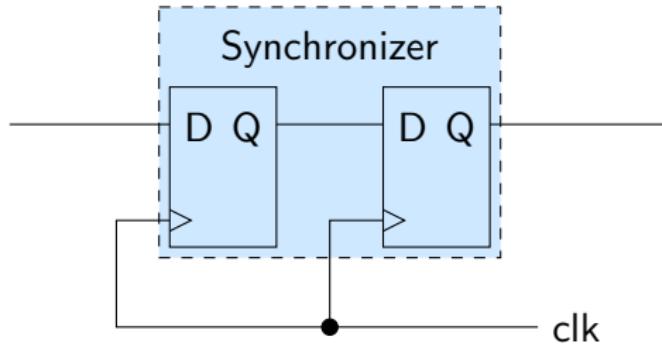


Handling Inputs cont'd

- How can we handle asynchronous inputs properly?

Handling Inputs cont'd

- How can we handle asynchronous inputs properly?
- have to be synchronized to clock
 - give metastability time to resolve (does not prevent it)
- realized by synchronizers
 - essential chain of N flip-flops
 - for uncritical system in general $N = 2$

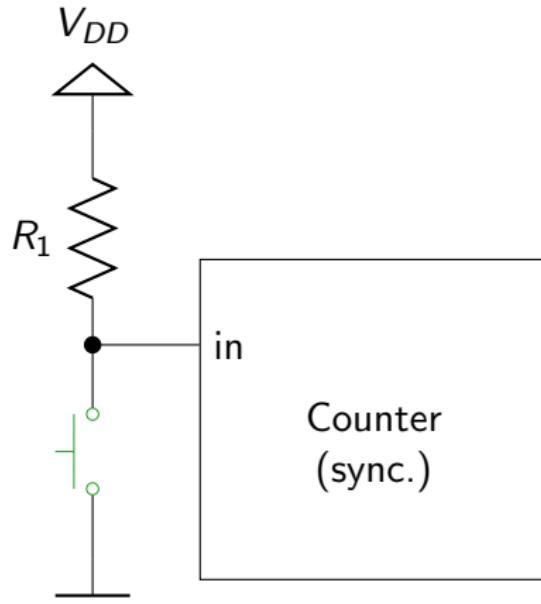


Handling Inputs cont'd

```
29 architecture beh of sync is
30     signal stages : STD.LOGIC_VECTOR(1 downto 0);
31 begin
32
33     process(clk)
34         begin
35         if rising_edge(clk) then
36             stages(0) <= l;
37             stages(1) <= stages(0);
38         end if;
39     end process;
40
41     0 <= stages(1);
42
43 end architecture;
```

[eaSynchronizer.vhd](#)

Buttons and Switches



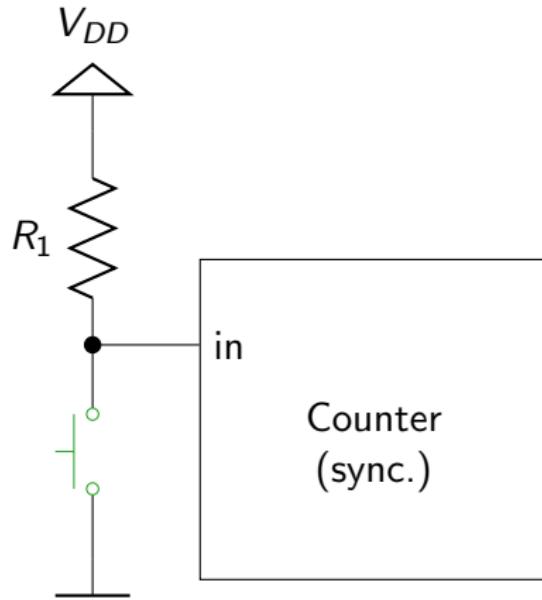
consider the shown circuit

- weak pull-up resistor
- push button connecting *in* to ground

How many transitions does the counter see when the button is pressed?

- A: 0
- B: 1
- C: ≥ 1

Buttons and Switches



consider the shown circuit

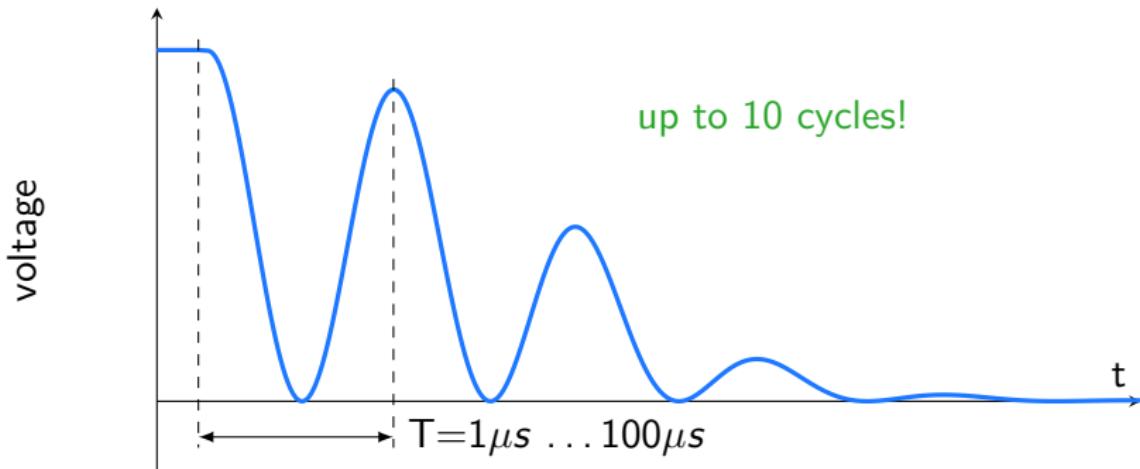
- weak pull-up resistor
- push button connecting *in* to ground

How many transitions does the counter see when the button is pressed?

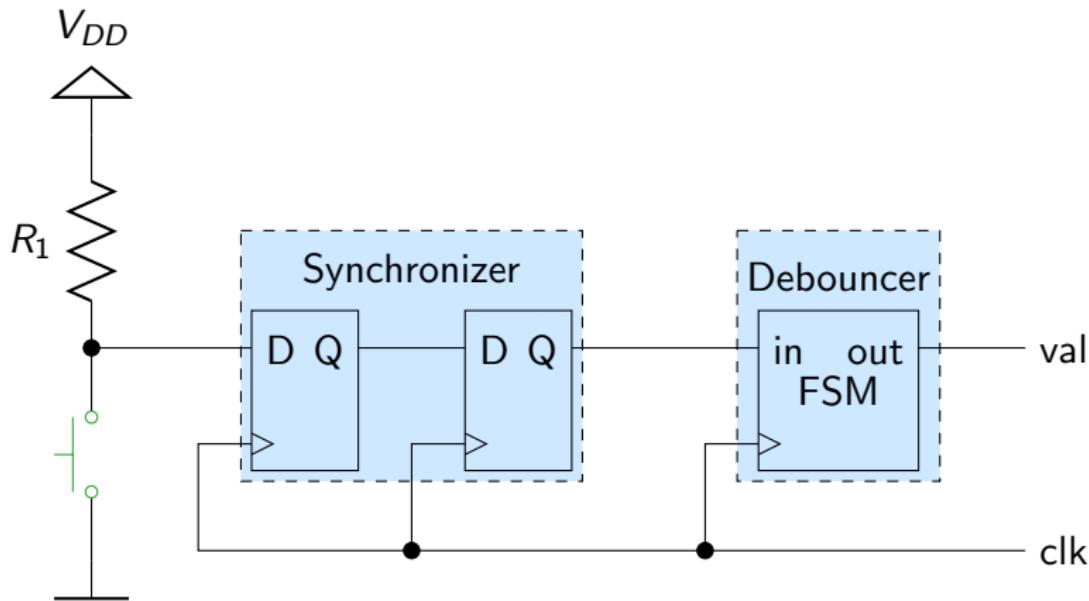
- A: 0
B: 1
C: ≥ 1 **correct!**

Bouncing Inputs

- results of bouncing inputs
 - one push on button results in multiple input transitions
- debouncing required
 - consider change only if stable for several clock periods
 - block input for several clock cycles after change
- either in hardware or software



Handling Bouncing Inputs



Running Hello World

- recall toy example “Hello World”
 - no synchronizers / debouncers used
 - neglected to keep it simple
- why is it working in the first place?
 - metastability
 - switches for direction
 - reset button

Running Hello World

- recall toy example “Hello World”
 - no synchronizers / debouncers used
 - neglected to keep it simple
- why is it working in the first place?
 - metastability **very low chances**
 - switches for direction
 - reset button

Running Hello World

- recall toy example “Hello World”
 - no synchronizers / debouncers used
 - neglected to keep it simple
- why is it working in the first place?
 - metastability **very low chances**
 - switches for direction **low sampling frequency**
 - reset button

Running Hello World

- recall toy example “Hello World”
 - no synchronizers / debouncers used
 - neglected to keep it simple
- why is it working in the first place?
 - metastability **very low chances**
 - switches for direction **low sampling frequency**
 - reset button **multiple resets no problem**
- exercise
 - extend “Hello World” by synchronizers / debouncers

Clock Domains

- single parts may run at different speed
 - compare CPU and RAM in PC
- speed may also be reduced on purpose
 - not always full speed required
 - lower frequency → less power consumption
 - $P \propto V^2f$
- synchronization on borders required
 - lots of different mechanisms possible
 - active field of research
 - easy solutions are e.g. synchronizers, dual clocked FIFO

Phase Locked Loops (PLL)

- remember synchronous process in “Hello World” example
 - speed reduction realized by counter
 - only viable in interfaces (no clock domain crossings)
- do not generate clock in logic (e.g. inverter chain)
 - signal not stable (PVT variations)!
- FPGAs support clock generation by PLLs
 - alter input clock in a given ratio (rational number)
 - Cyclone IV: ratio = m/n ; $m, n \in \{1, 2, \dots, 512\}$
 - multiple clocks from one PLL possible
- problem: startup time
 - takes some time until clock stable
- ATTENTION: works in simulations only if time set to ps

Resource Requirements

- what is required to implement design
 - based on finished design descriptions
 - has to be considered in design proceed if picked early
- hardware
 - size, functionality, additional equipment, special building blocks, ...
- software
 - tool chain
- selection based on
 - financial constraints
 - availability
 - knowledge
- always include some safety margins!

Resource Requirements cont'd

Case Study

- FPGA
 - capabilities to connect periphery
- periphery
 - keyboard (at least five keys)
 - push buttons (at least five)
 - USB port
 - display
- memory
 - sufficient to store complete games
- tools
 - VHDL 2008
 - support for chosen FPGA

State Machine Design

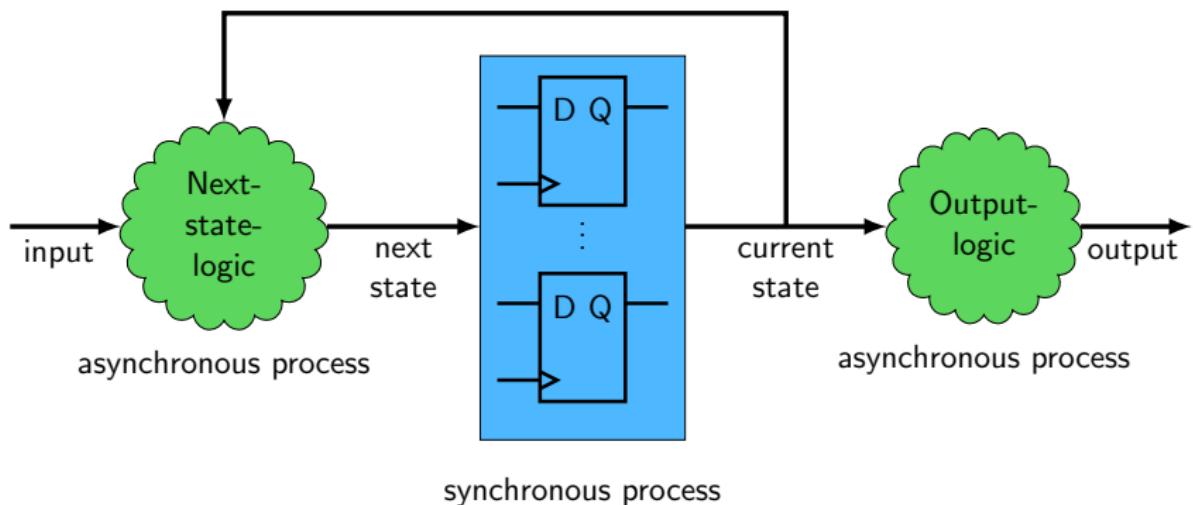
- state machine necessary if temporal order important
 - effect of input resp. output depend on internal state
 - frequently used in actual design
- can be represented in various ways (graph, table, ...)
- up until now graph translated to VHDL
 - how can this graph be derived from specification?
 - start from textual description in iterative fashion

Task Description

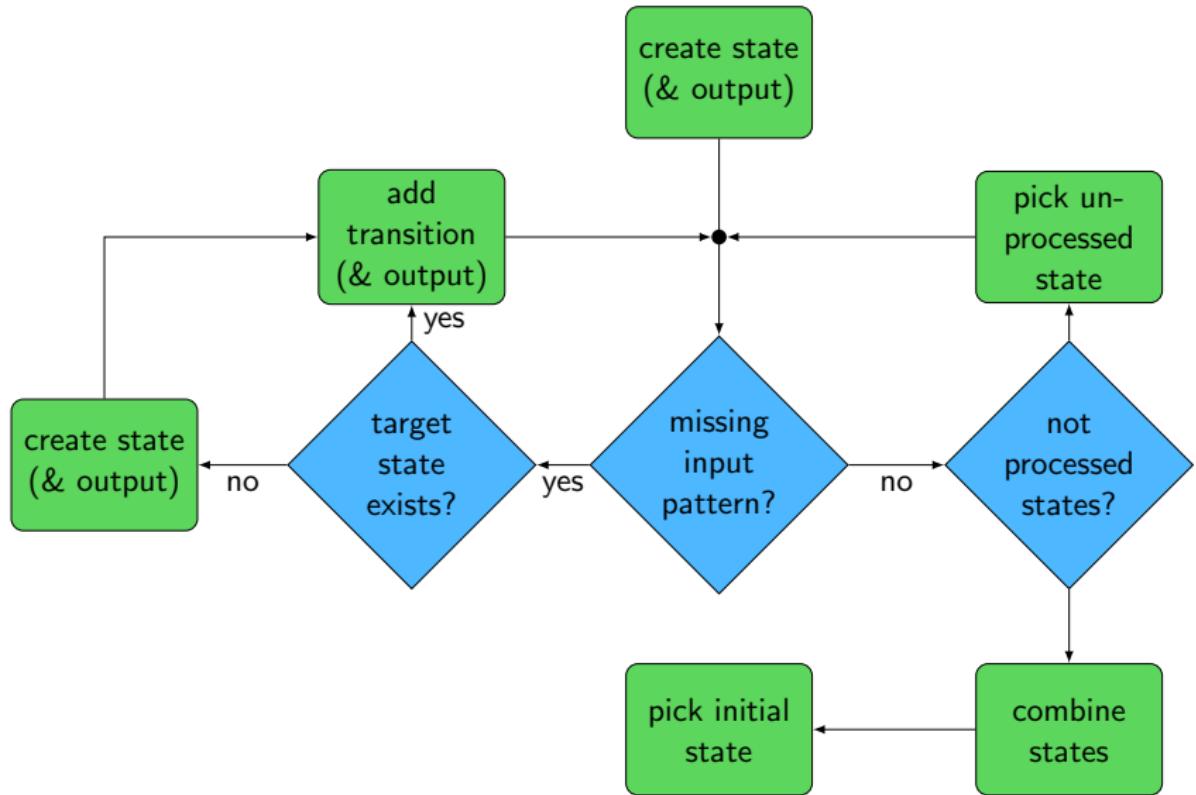
Model the snake control whose entity has four steering inputs (up, down, left, right) and four outputs (mu, md, ml, mr) which indicate the movement direction (active '1'). If the snake moves up/down only the inputs left/right have an effect and vice versa. Left has superiority over right, up over down. Initially the snake moves up.

State Machine Design cont'd

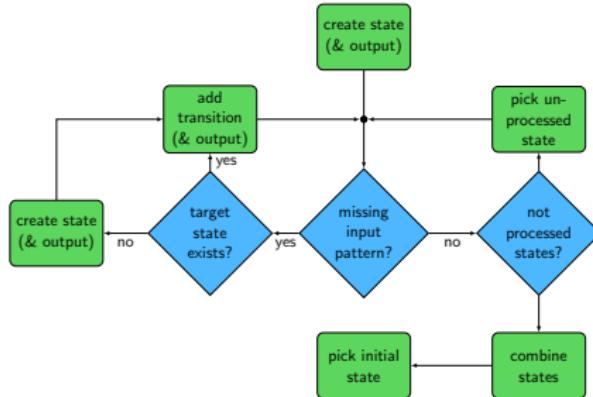
- focus on *Moore* state machines



State Machine Design cont'd

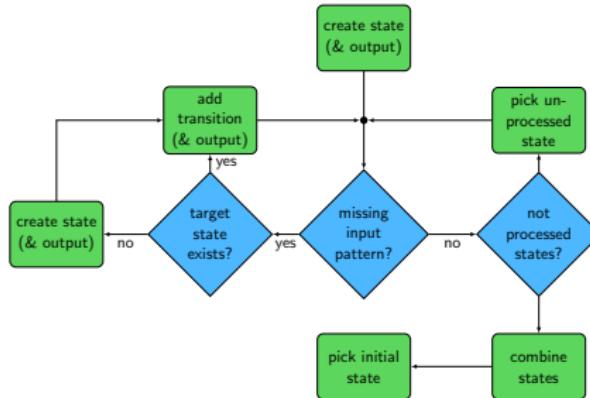


State Machine Design cont'd



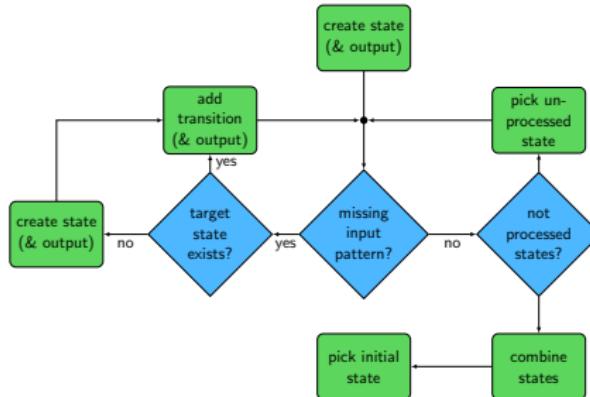
state	mu & md & ml & mr	old	u & d & l & r	new

State Machine Design cont'd



state	mu & md & ml & mr	old	u & d & l & r	new
U	1000			

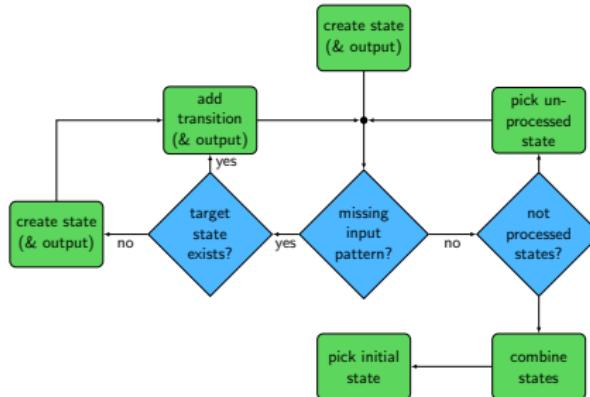
State Machine Design cont'd



state	mu & md & ml & mr
U	1000

old	u & d & l & r	new
U	--1--	L

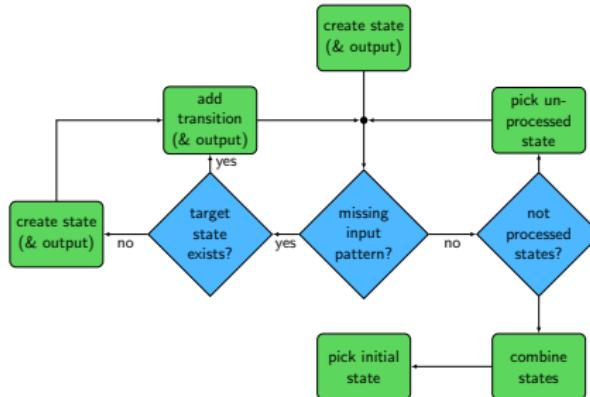
State Machine Design cont'd



state	mu & md & ml & mr
U	1000
L	0010

old	u & d & l & r	new
U	--1--	L

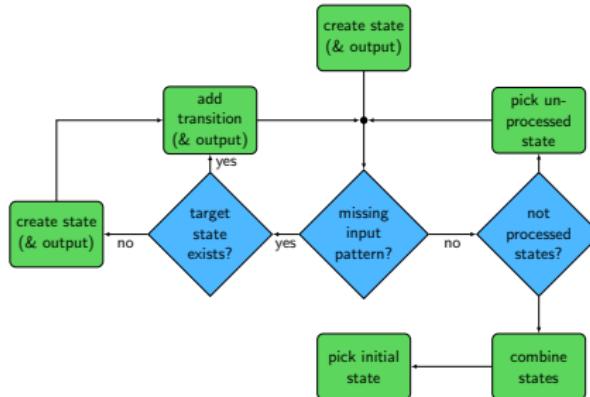
State Machine Design cont'd



state	mu & md & ml & mr
U	1000
L	0010
R	0001

old	u & d & l & r	new
U	--1--	L
U	--01	R

State Machine Design cont'd



state	mu & md & ml & mr
<i>U</i>	1000
<i>L</i>	0010
<i>R</i>	0001

old	u & d & l & r	new
<i>U</i>	--1--	<i>L</i>
<i>U</i>	--01	<i>R</i>
<i>U</i>	--00	<i>U</i>

State Machine Design cont'd

states

state	mu & md & ml & mr
U	1000
L	0010
R	0001

state transitions

old	u & d & l & r	new
U	--1-	L
U	--01	R
U	--00	U

State Machine Design cont'd

states

state	mu & md & ml & mr
U	1000
L	0010
R	0001
D	0100

state transitions

old	u & d & l & r	new
U	--1-	L
U	--01	R
U	--00	U
L	1---	U
L	01--	D
L	00--	L

State Machine Design cont'd

states

state	mu & md & ml & mr
U	1000
L	0010
R	0001
D	0100

state transitions

old	u & d & l & r	new
U	--1-	L
U	--01	R
U	--00	U
L	1---	U
L	01--	D
L	00--	L
R	1---	U
R	01--	D
R	00--	R

State Machine Design cont'd

states

state	mu & md & ml & mr
U	1000
L	0010
R	0001
D	0100

state transitions

old	u & d & l & r	new
U	--1-	L
U	--01	R
U	--00	U
L	1---	U
L	01--	D
L	00--	L
R	1---	U
R	01--	D
R	00--	R
D	--1-	L
D	--01	R
D	--00	D

State Machine Design cont'd

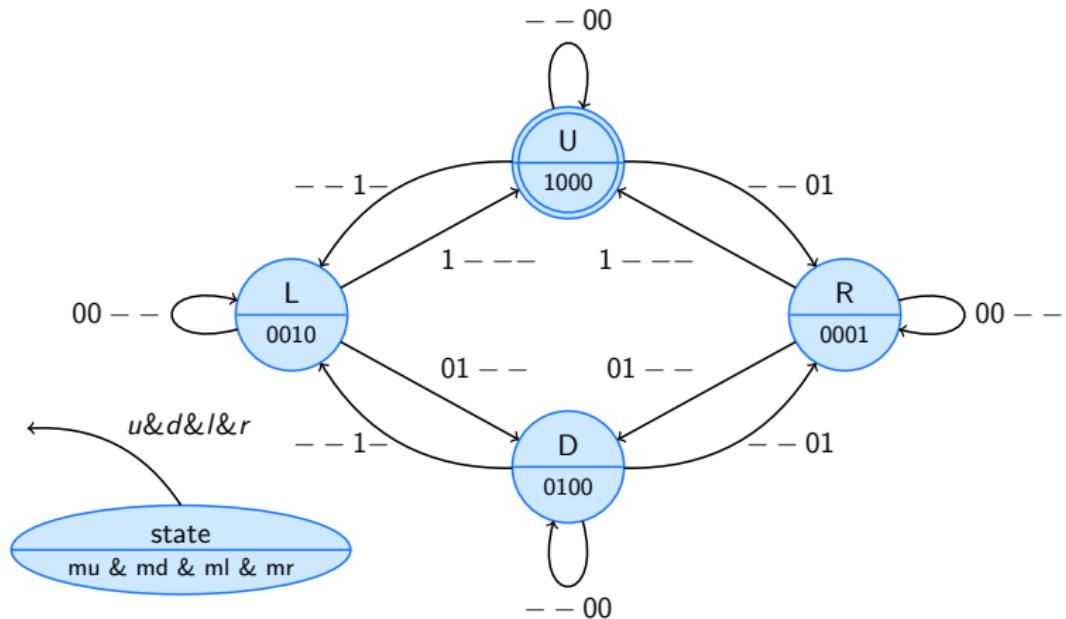
states

state	mu & md & ml & mr
U^*	1000
L	0010
R	0001
D	0100

state transitions

old	u & d & l & r	new
U	--1-	L
U	--01	R
U	--00	U
L	1---	U
L	01--	D
L	00--	L
R	1---	U
R	01--	D
R	00--	R
D	--1-	L
D	--01	R
D	--00	D

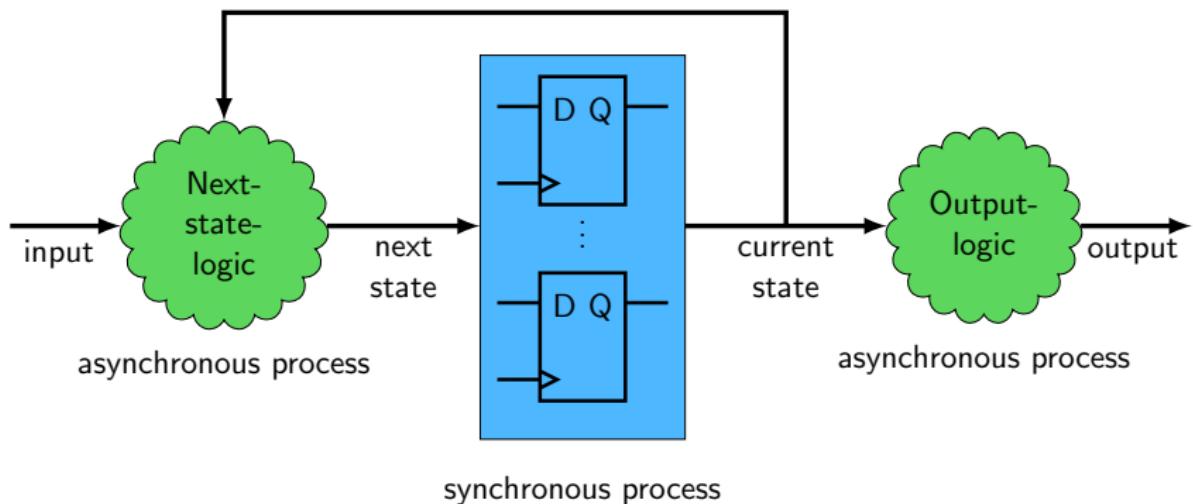
State Machine Design cont'd



State Machine Design Summary

- describe temporal behavior of unit
 - may be necessary to include internal processes
- achieve state machine design in iterative process
 - (1) identify state (& output)
 - (2) determine all transitions (& output) from that state
 - (3) start over with newly discovered states
- do not forget to
 - define format of output / transition
 - determine initial state
- further optimizations maybe necessary
 - split task to increase clock frequency
 - basic concept behind pipelining
 - for more details see lecture [Computer Organization and Design](#)

State Machine Design cont'd



Specification Summary

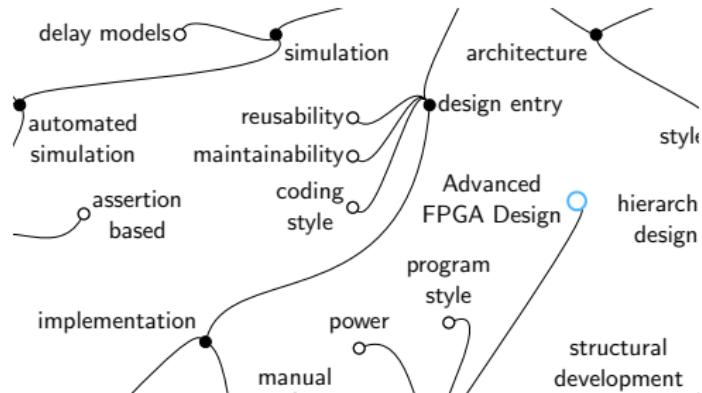
- specification indispensable at beginning
- thorough analysis of task serves
 - to detect pitfalls and misdesigns early
 - to guide and speed up implementation
- different methodologies used
 - top-down approach for design
 - bottom-up for implementation
- non-ideal behavior of real hardware has to be considered
- a designer should also consider
 - which computation to perform where
 - general data flow (time vs. location)
 - critical paths (clock frequency)

Check Your Knowledge

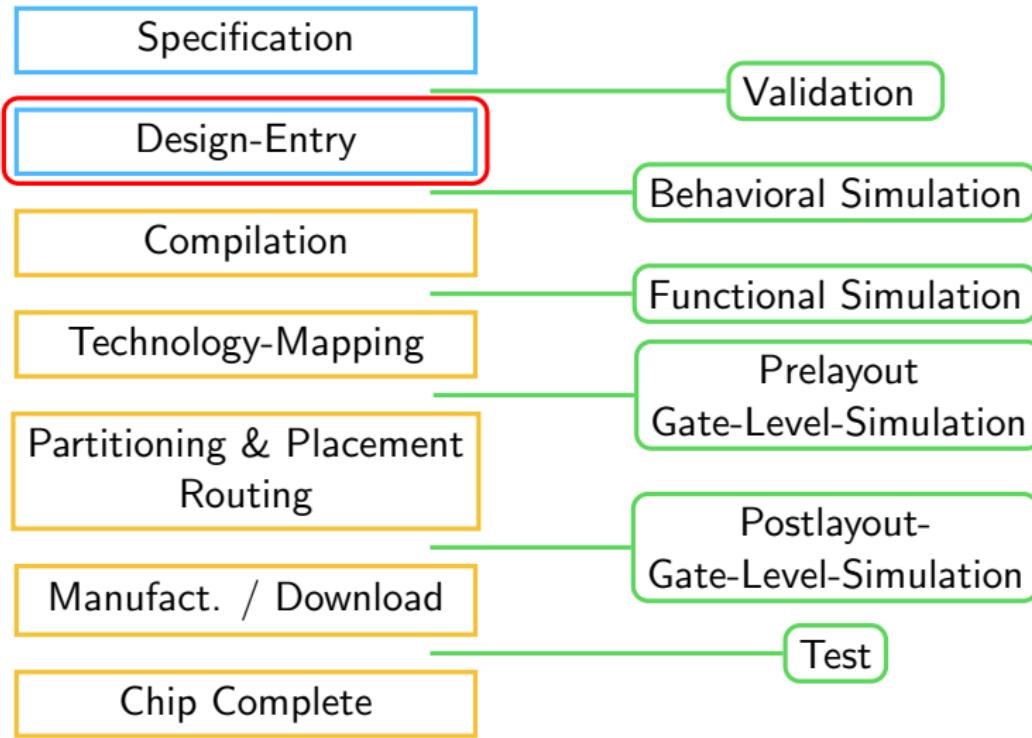
- take again a look at exercises in LU
 - investigate structure
 - which units are connected?
- try to retrace made decision
 - what is the purpose of the single units?
 - would it be possible to remove some?
 - would you further split or merge certain entities?
 - would you have chosen the same structure?
 - can the presented state machine be improved?

Hardware Modeling

Design Entry



Design Flow

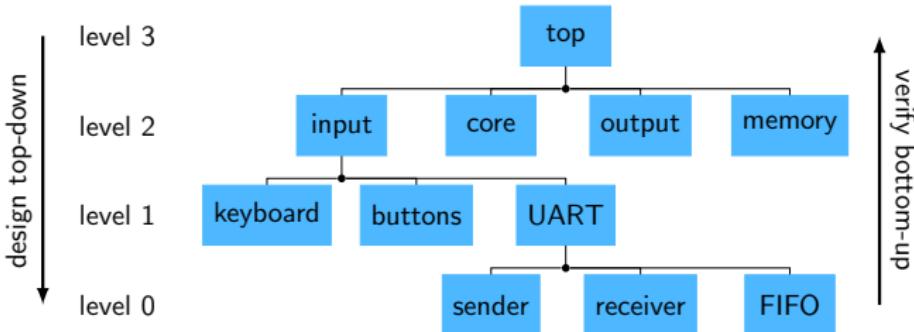


Design Entry

- VHDL only description language
 - code generated automatically
 - little changes might have huge impact
- mapping on FPGA not optimal
 - memory realized in logic blocks
 - special blocks like multipliers not used
 - state machines not detected
- hard to write really good code
 - e.g. [Intel Quartus Prime Pro User Guides](#) contains “Design Recommendations”
 - contains hints on what to consider

Design Entry cont'd

- detailed structure of entities available
 - defined in specification
 - already contains coarse behavior
- bottom-up implementation / verification
 - start at lowest abstraction level
 - implement and immediately test each entity separately
 - leads to hierachic system → good reusability
- more on verification later



Reusability

- highest goal when writing code
 - tested and verified entities can be used in future projects
 - just additional connections have to be checked
 - reduces implementation / verification effort
- certain conditions have to be fulfilled
 - final partitioning (entity can not be divided further)
 - good documentation (also benefits yourself)
 - usage of packages, components, generics
- use copy & paste!
 - except exercises and exams ... ;)

Reusability cont'd

- not restricted to own code
- well tested code available
 - even optimized for synthesis
 - automatically detected and mapped to DSP blocks
- provided by vendors
 - IP Catalog
 - Templates
- open source software / modules
- gives further insight on problem solving
 - learn unknown language features

Templates

- generates piece of code solving problem
 - can then be altered according to needs
 - CAUTION: slight changes might already affect synthesis
- various designs in various languages available
 - (System)Verilog, VHDL, TCL, ...
 - full designs, specific units, logic, ...
- example memory: if done wrongly mapped to LUTs
 - e.g. reset signal
- advantages
 - high portability (just recompile with new tool, library, ...)
- disadvantage
 - has to be handled carefully to not destroy optimizations

IP Catalog

- vendor provides complete modules
 - memory, controller, communication interfaces, ...
 - delivered as precompiled library
- highly configurable
 - define desired properties during creation
 - alter signal width, enable signals, clocks, ...
- in Quartus “IP Catalog” used
 - previously called Mega-Wizard (Megafunctions)
- consider documentation for temporal control
 - temporal behavior of input and outputs
 - accessible from within “IP Catalog”
- not so flexible as templates
 - when tools changed module maybe not available any more

Records on Interfaces

- specify interface types as records
 - outputs are inputs of another unit
 - readability increased and dataflow easier to follow
- optimally implemented in package
 - easily available to other entities

Package

```
package interface_pkg is

    type in_type is record
        data : std_logic_vector(15 downto 0);
        en : std_logic;
    end record;

    type out_type is record
        data : std_logic_vector(15 downto 0);
        en : std_logic;
        full : std_logic;
    end record;

end interface_pkg;
```

Entity/Architecture

```
use work.interface_pkg.all;

entity interface is
    port
    (
        input : in in_type;
        output : out out_type
    );
end entity;

architecture rtl of interface is
begin
    output.data <= input.data;
end architecture;
```

Benefits

- less error prone
- readability, maintainability and extendability greatly improved

(1) Adding a port

Traditional

add port in

- entity declaration
- component declaration
- sensitivity list
- instantiation

Records

- add element in the interface record

Benefits

- less error prone
- readability, maintainability and extendability greatly improved

(2) Adding a register

Traditional

- add signal declaration
- alter sensitivity list of synchronous process
- add driving statement

Records

- add definition in register record

Benefits

- less error prone
- readability, maintainability and extendability greatly improved

(3) Tracing signals during debugging

Traditional

- determine appropriate signals
- add signals individually to trace
- repeat every time on change

Records

- add interface / internal record
- all relevant signals grouped together
- alteration of record automatically propagated

Records cont'd

- also useful for internal signals
 - summarize all signals written in process
- assign at start of process to next signal
 - prevents that no values assigned
 - no more “inferred latches”

```
1  architecture rtl of use_records is
2    type INTERNAL_SIGNALS is record
3      data : std_logic_vector(15 downto 0);
4      en : std_logic;
5      done : std_logic;
6    end record;
7
8    signal sig, sig_next : INTERNAL_SIGNALS;
9  begin
```

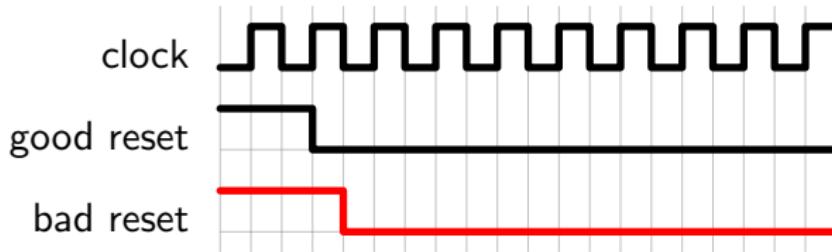
Records cont'd

- also useful for internal signals
 - summarize all signals written in process
- assign at start of process to next signal
 - prevents that no values assigned
 - no more “inferred latches”

```
11      output : process (all)
12      begin
13          sig_next <= sig;
14
15      if ... then
16          sig_next.en <= '1';
17      end if;
18      ...
19  end;
20
21  sync : process (all)
22  begin
23      if rising_edge(clk) then
24          sig <= sig_next;
25      end if;
26  end;
27
28  sig_out1 <= sig.en;
29  sig_out2 <= sig.done;
30  --sig_out <= sig;
```

Reset Signals

- important to set initial state of circuit
 - e.g. starting point of state machines
- handle with care
 - may lead to undesired synthesis, e.g. with memory
- different characteristics
 - synchronous vs. asynchronous
 - active low vs. active high
- has to be released in appropriate distance to clock
 - otherwise metastability possible



State Machine Implementation

- different possibilities encountered
 - 1 / 2 / 3 **process** method
- more than 3 not reasonable
 - same signal written from different **process**

1. How do they differ?

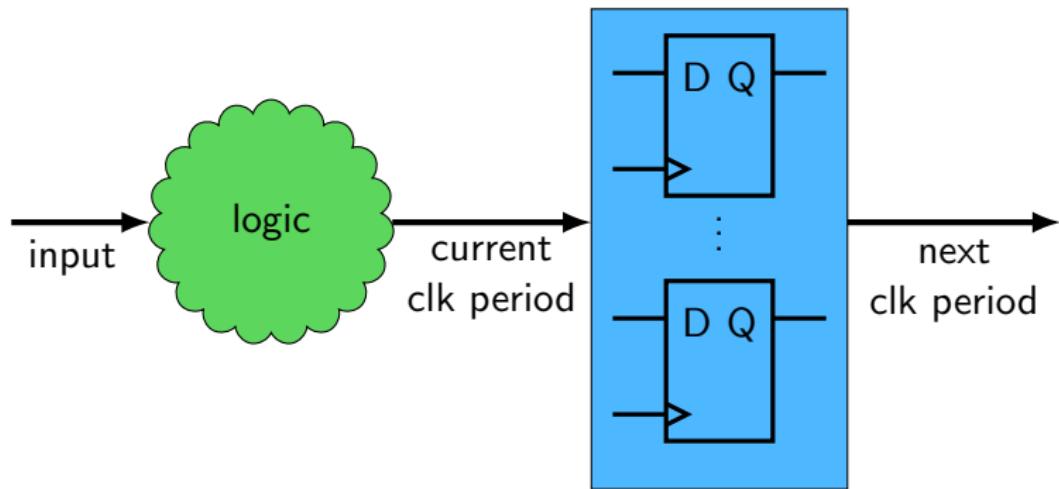
2. When to use what?

1 process Method

- only single process used
 - everything solely evaluated at rising clock edge
 - no next signals required
 - clock and enable in sensitivity list sufficient
- output automatically registered
 - no direct output possible with this method

```
1 architecture ...
2 ...
3 begin
4
5 sync: process(clk, en)
6 begin
7 if en = '1' then
8 ...
9 elsif rising_edge(clk) then
10 case state is
11 when ... =>
12     state <= ...
13     output <= ...
14 when ... =>
15 ...
16 end case;
17 end if;
18 end process;
19
20 end architecture;
```

1 process Method cont'd

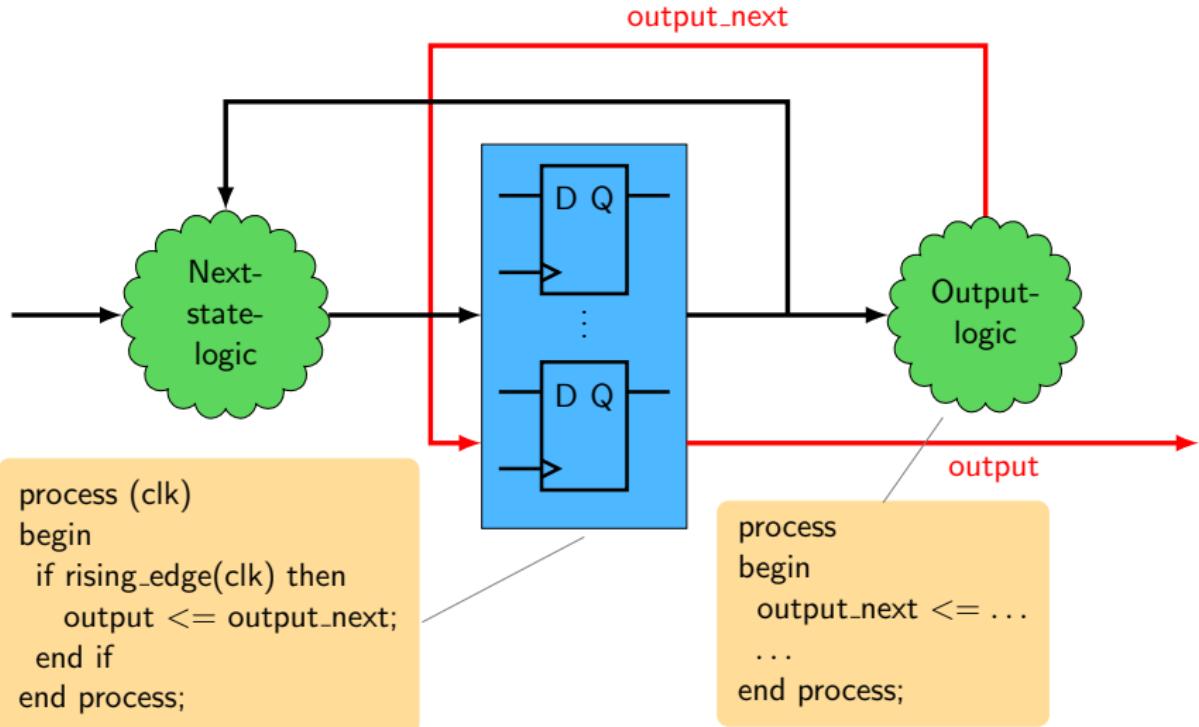


2/3 process Method

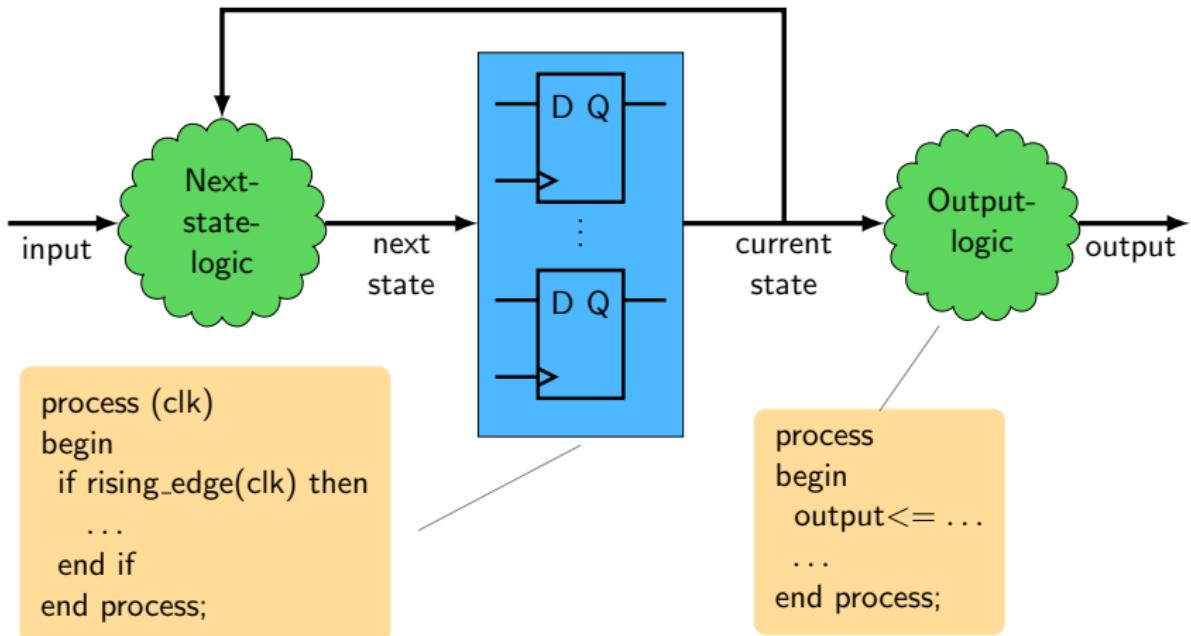
- two process utilized
 - asynchronous one preparing next signal values
 - synchronous one assigning next signal values at clock edge
- output registered and not registered possible
 - write directly on output if no register desired
- in 3 process method write output and next state in separate process

```
1  architecture ...
2  ...
3  begin
4      sync: process(clk, en)
5          begin
6              if en = '1' then
7                  ...
8              elsif rising_edge(clk) then
9                  output <= output_next;
10                 state <= state_next;
11             end if;
12         end process;
13
14         async : process (all)
15     begin
16         case state is
17             when ... =>
18                 state.next <= ...
19                 output.next <= ...
20             when ... =>
21                 ...
22         end case;
23     end process;
24 end architecture;
```

2/3 process Method cont'd



2/3 process Method cont'd



process Method comparison

- Methods equally powerful?
 - 1 process enforces registers
 - more versatile with 2 and 3 process method
- When to use what if one has choice?
 - largely based on personal preferences
 - consider corporate guidelines
- impact of register at output
 - new value issued synchronous to clock
 - higher clock frequency may be possible

Useful Language Constructs

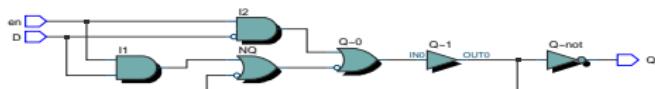
- short conditional assignment
 - `<signal> <= <bool> ? <true> : <false>;`
- assign multiple values at once
 - `(a,b,c) <= "0010110"`
 - length of left and right hand side must be equal
- std_match, ?=, ?<
 - compare operations that allow non-boolean results
 - return type is std_ulogic
 - simulation and synthesis the same

Style Guidelines

- combinational loops without registers
 - may start to oscillate
- memory elements built with combinational logic
 - not detected → not properly synthesized
 - oscillatory behavior possible
- inferred latches

```
31 architecture rtl of d_latch is
32     signal NQ, I1, I2 : STD_LOGIC;
33 begin
34
35     I1 <= D and en;
36     I2 <= (not D) and en;
37     Q <= I2 nor NQ;
38     NQ <= I1 nor Q;
39
40 end architecture;
```

eaLatch_comb.vhd

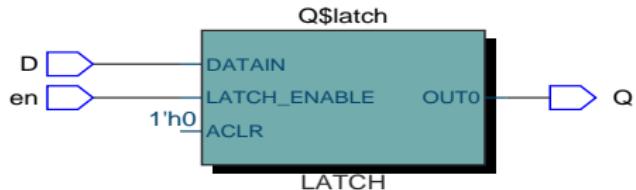


Style Guidelines

- combinational loops without registers
 - may start to oscillate
- memory elements built with combinational logic
 - not detected → not properly synthesized
 - oscillatory behavior possible
- inferred latches

```
31 L : process (en, D)
32 begin
33   if en = '1' then
34     Q <= D;
35   end if;
36 end process;
```

eaDLatch.vhd



Style Guidelines

- combinational loops without registers
 - may start to oscillate
- memory elements built with combinational logic
 - not detected → not properly synthesized
 - oscillatory behavior possible
- inferred latches

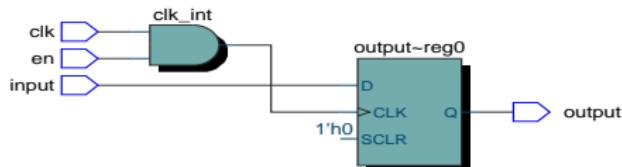
```
Info: Only one processor detected - using single parallel compilation
Info: Found 2 design units, including 1 entities, in source file latch.vhd
Info: Elaborating entity "latch1" for the top level hierarchy
Warning (10631): VHDL Process Statement warning at latch.vhd(14): inferring latch(es) for signal or variable "o", w
Info (10041): Inferred latch for "o" at latch.vhd(14)
Info: Implemented 6 device resources after synthesis - the final resource count might be different
```

Style Guidelines cont'd

- do not implement gates on clock paths
 - introduce additional delay
 - registers switch at different points in time
 - use enable signals on data path

```
31 architecture clk_enable of flip_flop is
32   signal clk_int : STD_LOGIC;
33 begin
34
35   clk_int <= clk and en;
36
37   sync : process (clk_int)
38   begin
39     if rising_edge(clk_int) then
40       Q <= D;
41     end if;
42   end process;
43
44 end architecture;
```

eaFF_clk.vhd

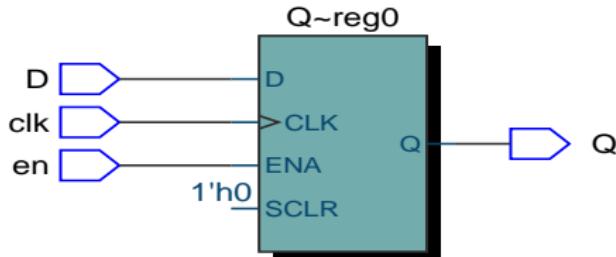


Style Guidelines cont'd

- do not implement gates on clock paths
 - introduce additional delay
 - registers switch at different points in time
 - use enable signals on data path

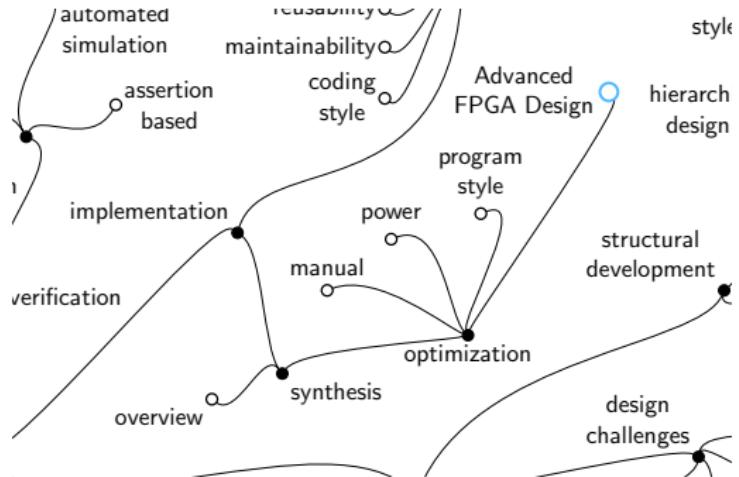
```
31 architecture rtl of flip_flop is
32 begin
33 sync : process (clk, en)
34 begin
35 if rising_edge(clk) then
36   if en = '1' then
37     Q <= D;
38   end if;
39   end if;
40 end process;
41 end architecture;
```

eaFF_en.vhd

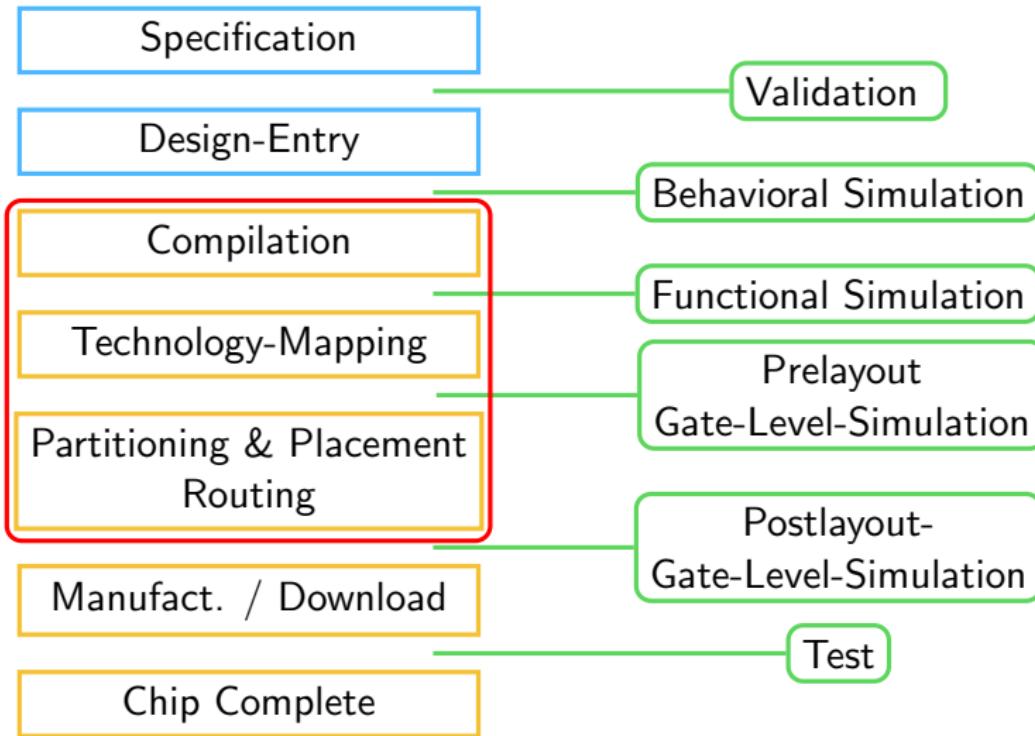


Hardware Modeling

Synthesis



Design Flow



Synthesis Overview

- compilation
 - convert code to netlist with generic components
 - not interesting if such a circuit available
 - number of in- and outputs, clock signal, sync / async reset, ...
- technology mapping
 - map generic components to gates (standard cells) from library
 - not every possibility available
 - o use available units to recreate functionality
- place & route
 - choose position for library cells and connect them
 - two (nearly) independent procedures
 - heuristics used to reduce run time
- compare to lecture [Digital Design](#)

Physically Aware Synthesis

- consider physical considerations much earlier in synthesis process
- pick library cells depending on location and connections
 - driving strength, speed, area, ...
- estimate interconnects by preliminary placement during synthesis
- floorplan and pin locations necessary
- iterative process
 1. logic synthesis
 2. placement and routing estimation
 3. interconnect property extraction
 4. optimization

Constraints

- sometimes tools assume world too optimal
 - e.g. clocks are in phase
 - estimated values fit badly
 - design passes timing analysis but does not work in reality
- tell tool about real world properties
- examples
 - input / output delays
 - clock uncertainties
- very important in industry!
- added in *.sdc* file

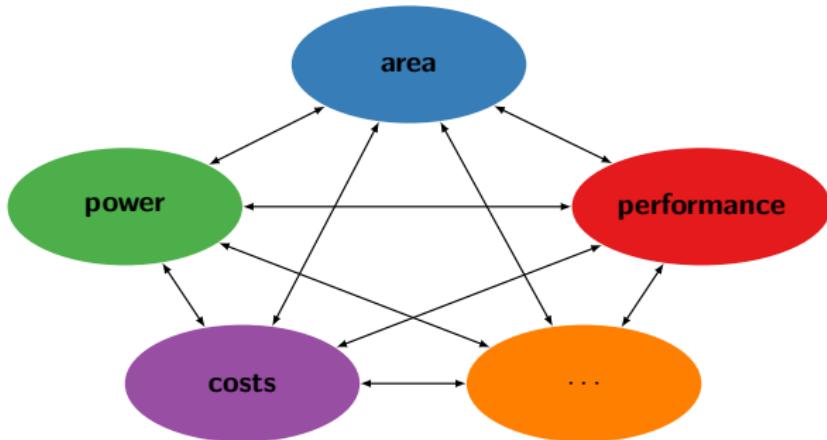


Optimizations

- extensive optimizations in every step
 - lots of different optimization procedures implemented
 - explains sometimes longer run times
- models from library for suitable timing / area / power
- available for different process / voltage / temperature (PVT) conditions (corners)
 - best case, typical case, worst case, worst leakage
- special nets
 - are not optimized during synthesis
 - instead timing is adapted during routing
 - e.g. clock net, reset signals

Optimizations cont'd

- optimization goal to fulfill constraints
 - e.g. regarding performance / area / power / costs / ...
 - not possible to optimize all at the same time
 - o higher speed → higher area and power
 - slower solution chosen to reduce power consumption if possible

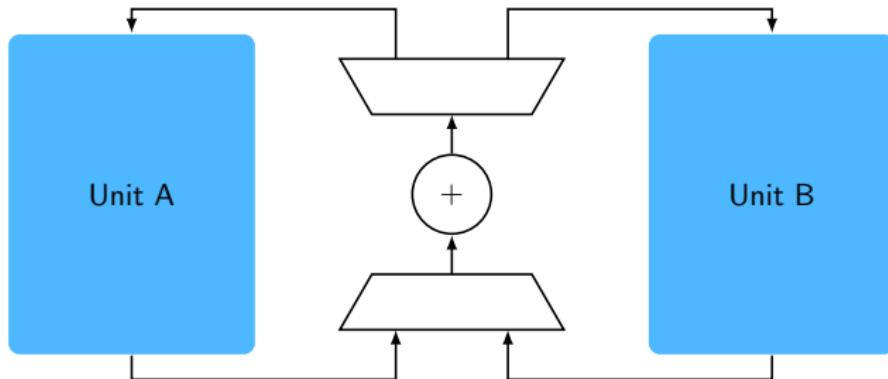


Optimizations cont'd

- characterize optimization by goal
 - area
 - speed
 - power
 - ...
- characterize optimization by method
 - tool based: automatically done
 - manual: user defined optimizations
 - programming style: coding patterns
- just short overview given
 - separate lecture [182.756 VU Advanced FPGA Design](#)

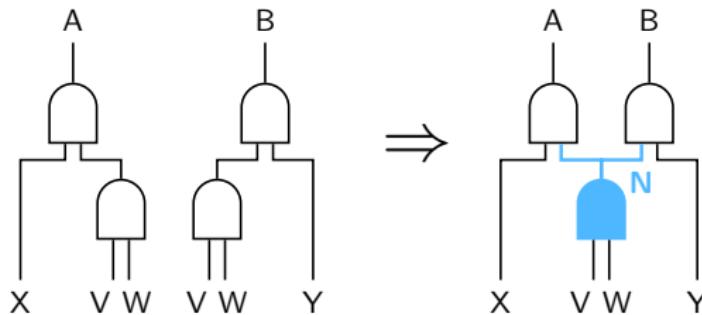
Optimization procedures

- operator resource sharing
 - use same hardware in different modules
 - only if access mutually exclusive (time multiplexing)
 - higher utilization of units



Optimization procedures cont'd

- common sub-expression elimination
 - consider $A = V \& W \& X$; $B = V \& W \& Y$;
 - share common term $V \& W$
 - convert to $N = V \& W$; $A = N \& X$; $B = N \& Y$;
 - hard to detect → heuristics used

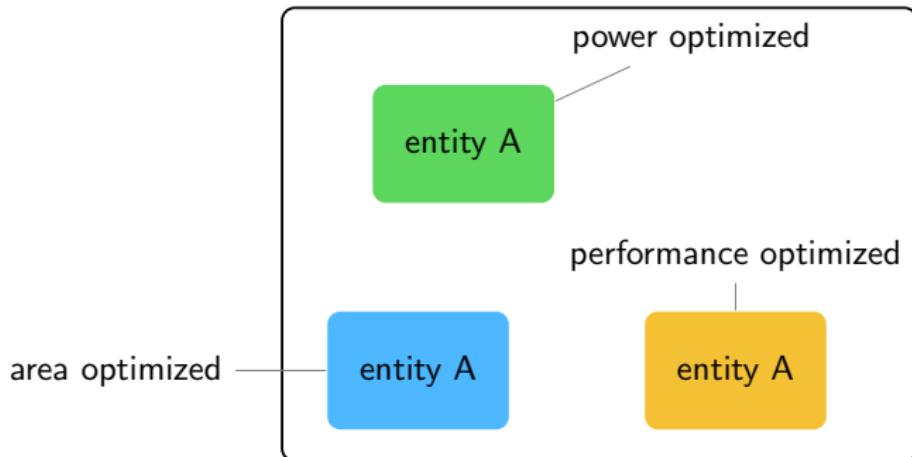


Optimization procedures cont'd

- managing hierarchy
 - change structure to allow better optimization
- ungrouping
 - break up group affiliation of single units
 - combine them in one large logic and optimize at once
- flattening
 - comparable to ungrouping only on different hierachic levels
- boundary optimization
 - do not completely dissolve units
 - only optimizations at boundaries allowed

Optimization procedures cont'd

- unification
 - multiple instantiations of single entity
 - each is independently optimized



Power Optimization

- two types of power consumption
 - static (leakage)
 - dynamic (during switching)
- highest saving potential at high levels of abstraction
 - determine where highest switching activities are
 - global connections only detectable at high abstraction
- algorithmic / architectural choices
 - optimize highly active gates
 - requires early power estimation
- switching behavior largely influences synthesis
 - different realizations depending on input

Power Optimization cont'd

- design options
 - pipelining
 - o cp. State Machine Design
 - o more stages vs. bigger logic
 - encoding (state machine states, bus)
 - o one hot encoding may be more efficient
 - memory banking
 - o cut off unused parts of memory
 - pre-computation
 - o compute result and shift in pipeline vs. in place computation
 - clock gating
 - o cut off clock signal to subtree
 - o specific Integrated Clock Gating (ICG) cells available

Power Optimization cont'd

- automated optimization techniques
 - operand isolation
 - o disable units if output not required
 - o e.g. data gating (AND gates / latches)
 - pin swapping
 - o power consumption on input pins differs
 - factoring
 - o move high activity nets into more complex cells
 - o results in shorter interconnects
 - multi threshold voltage
 - o lower threshold → faster but more leakage

Tool Based Optimization Overview

- highly developed tools
 - according to manual designer should trust tools
 - reasonable to completely rely on tool?
- various automatic optimization steps
 - regarding area, power, performance, costs, ...
 - are responsible for large amount of synthesis time
 - only subset shown in this lecture
- still designer can control / enhance optimization
 - assign '-' (don't care) instead of arbitrary value to unused signals
 - gives compiler more freedoms

Attributes

- contain additional information
 - predefined already shown, e.g. I'RANGE
 - user-defined also possible
- tool specific, standard for most basic ones
- used to control synthesis
 - only possible this way from within VHDL
 - details e.g. [Intel Quartus Prime Pro User Guides](#), [The Designer's Guide to VHDL](#), Chapter 21
- example: encoding of enumeration type (enum_encoding, FSM_STATE)
 - “sequential”, “gray”, “johnson”, “one-hot”, “default”

```
type state is (U,D,L,R);
attribute enum_encoding : string;
attribute enum_encoding of state : type is "1000_0100_0010_0001";
```

Attributes

- contain additional information
 - predefined already shown, e.g. I'RANGE
 - user-defined also possible
- tool specific, standard for most basic ones
- used to control synthesis
 - only possible this way from within VHDL
 - details e.g. [Intel Quartus Prime Pro User Guides, The Designer's Guide to VHDL, Chapter 21](#)
- example: encoding of enumeration type (enum_encoding, FSM_STATE)
 - “sequential”, “gray”, “johnson”, “one-hot”, “default”

attribute name : type;

attribute name of entity_name_list : entity_class is expression;
entity_class ::= entity | type | signal | variable | ...

Metacomments

- control statements inside regular comments
 - case insensitive
 - are interpreted by tools
 - very powerful commands possible
 - again tool and company specific
- lots of different commands available
- *--coverage on / --coverage off*
 - enable / disable coverage analysis
 - useful for assert statements
- *--rtl_synthesis on / --rtl_synthesis off*
 - enable / disable synthesis
 - skip monitoring processes

-- coverage off

assert a>0 report "a_not_set_correctly" severity ERROR

-- coverage on

Manual Optimization

- sometimes human intervention necessary
 - tools do not always predict usage correctly
- example: inverter delay chain
 - gets reduced by tool to single inverter / buffer
- possible to prevent optimization of tools
 - set user defined attribute *keep* to true

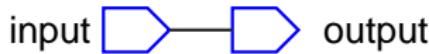
```
23  entity delay_line is
24    generic (
25      count : NATURAL := 5
26    );
27    port
28    (
29      input : in STD_LOGIC;
30      output : out STD_LOGIC
31    );
32  end entity;
```

[eaDelay_line_opt.vhd](#)

Prevent Optimization

```
34 architecture inverters of delay_line is
35     signal inv_chain : STD_LOGIC_VECTOR(count-1 downto 0);
36 begin
37     chain : for i in count-1 downto 1 generate
38         inv_chain(i) <= not(inv_chain(i-1));
39     end generate;
40
41     inv_chain(0) <= input;
42     output <= inv_chain(count-1);
43 end architecture;
```

eaDelay_line_opt.vhd



Prevent Optimization

```
34  architecture inverters of delay_line is
35      signal inv_chain : STD_LOGIC_VECTOR(count-1 downto 0);
36      attribute keep : BOOLEAN;
37      attribute keep of inv_chain : signal is TRUE;
38  begin
39      chain : for i in count-1 downto 1 generate
40          inv_chain(i) <= not(inv_chain(i-1));
41      end generate;
42
43      inv_chain(0) <= input;
44      output <= inv_chain(count-1);
45  end architecture;
```

eaDelay.line.keep.vhd



Programming Style Optimization

- sometimes order of commands influences hardware complexity

version (1)

```
architecture inverters of delay_line is
begin
    process (all)
    begin
        if x = '1' then
            o <= i+1;
        else
            o <= i-1;
        end if;
    end process;
end architecture;
```

eaTwo.adders.vhd

version (2)

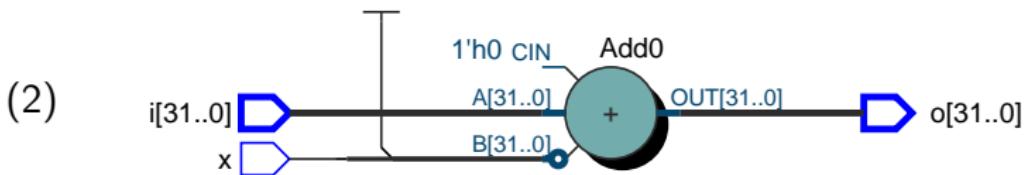
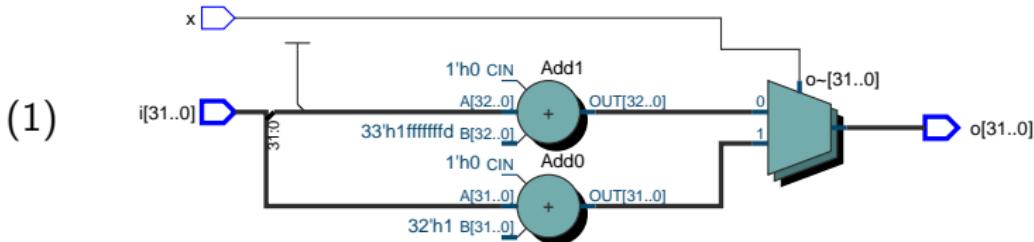
```
architecture inverters of delay_line is
    signal tmp : integer;
begin
    process (all)
    begin
        if x = '1' then
            tmp <= 1;
        else
            tmp <= -1;
        end if;
    end process;
    o <= i + tmp;
end architecture;
```

eaOne.adder.vhd



Programming Style Optimization

- sometimes order of commands influences hardware complexity



Timing Analysis

- delay estimation crucial task in synchronous circuits
 - how do signals propagate through a circuit?
 - setup and hold time violations → metastability
 - glitches & runts
 - clock domain crossings
 - ...
- only very crude estimations possible
 - huge safety margins required
 - always worst case corner has to be considered
- cell libraries contain various information
 - input-to-output delay, setup and hold time, ...
 - models for delay, power, noise
 - design rules regarding transition times, fanout, ...

Timing Analysis cont'd

- path delay = cell delay + interconnect delay
- cell delay calculated from library
 - usually lookup table based
 - for current technologies more elaborate current source models (CSM) used
 - o high characterization effort
 - o only full range switchings considered
- interconnect delay
 - dominant in modern technologies (up to over 80 % possible)
 - also lookup table based (block size/ fanout)
 - more precise models available (Elmore delay)

Synthesis of Data Types

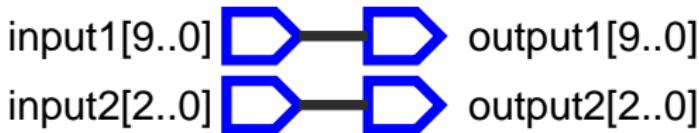
- take care when using range data types
 - 0 always included
 - sometimes better to start at 0 and add constant value
- tools handle this in general automatically

```
23  entity data_width is
24    port
25    (
26      wide_in : in INTEGER range 512 to 519;
27      narrow_in : in INTEGER range 0 to 7;
28      wide_out : out INTEGER range 512 to 519;
29      narrow_out : out INTEGER range 0 to 7
30    );
31  end entity;
32
33  architecture rtl of data_width is
34  begin
35    wide_out <= wide_in;
36    narrow_out <= narrow_in;
37  end architecture;
```

[eaData_width.vhd](#)

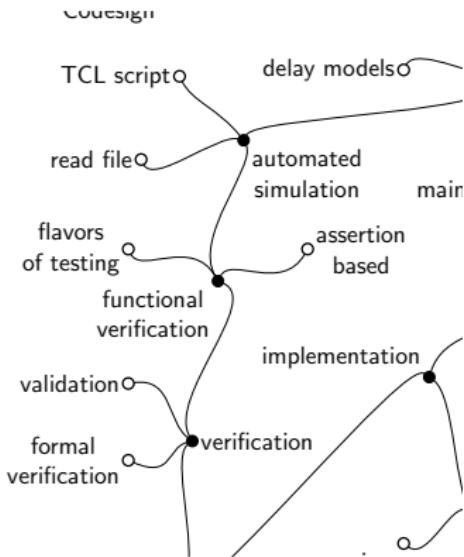
Synthesis of Data Types

- take care when using range data types
 - 0 always included
 - sometimes better to start at 0 and add constant value
- tools handle this in general automatically



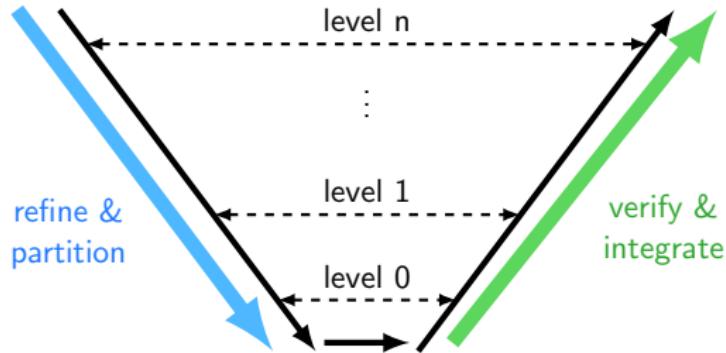
Hardware Modeling

Verification

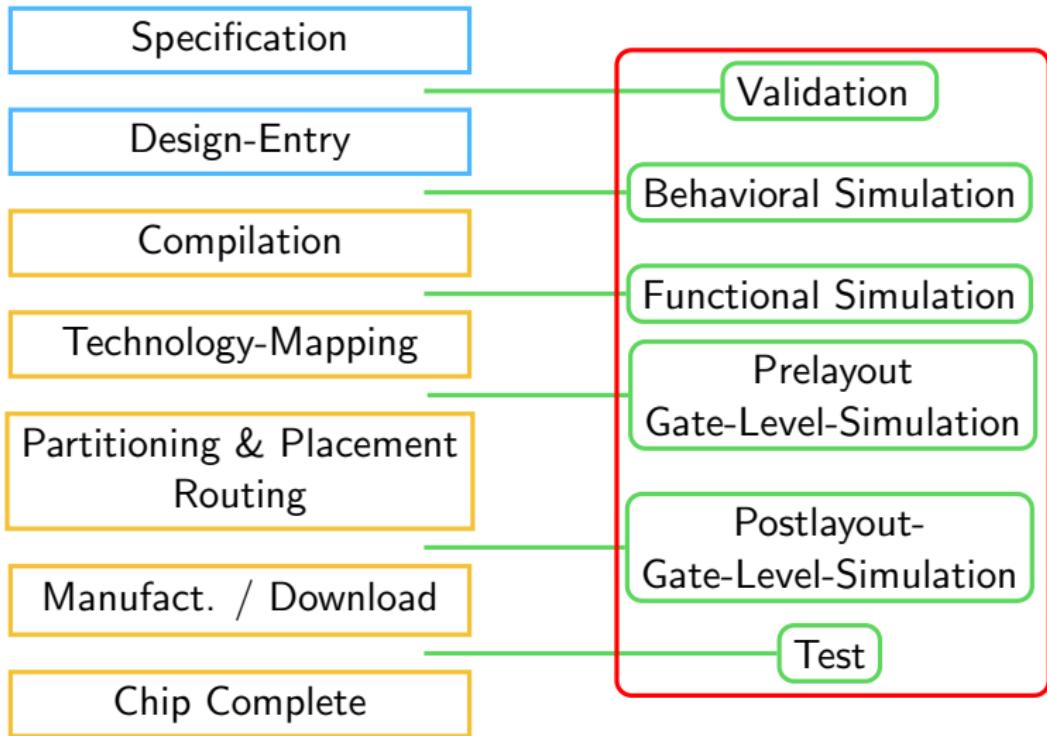


Verification

- required in every design phase
 - different abstraction levels
 - different methods
- recall: supposed to take **70-80 %** of overall time



Design Flow

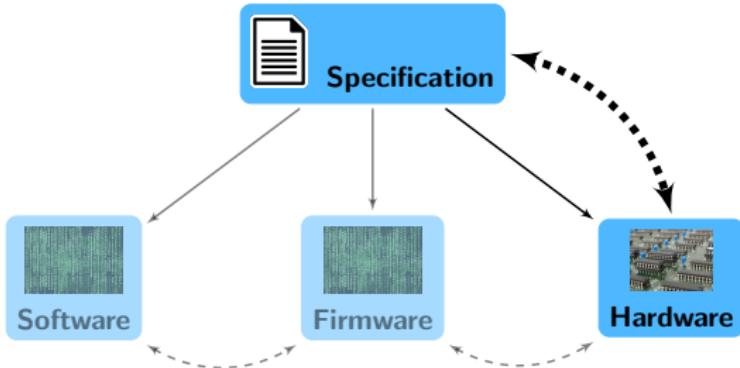


Verification

- in this lecture three types distinguished
- validation
 - specification vs. implementation
 - human factor very important
- functional verification
 - simulation in software
 - test of actual hardware
 - high degree of automation possible
- formal verification
 - equivalence checking – comparing models
 - model checking – verifying properties
 - young field that gets more and more important

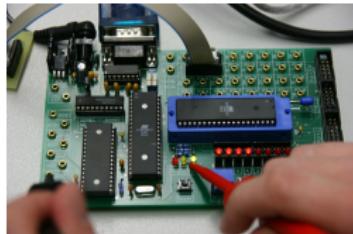
Validation

- check specification vs. implementation
 - did we build the correct system?
- human factor very important
 - cumbersome and error prone
- automatization hard but desired
 - active field of research



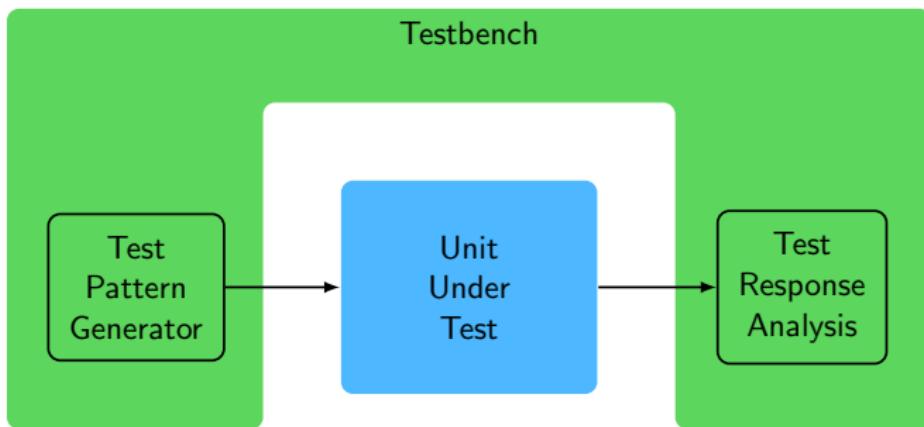
Functional Verification

- check if behavior of design as expected
 - did we build the system correct?
- simulation in software
 - each basic module extensively tested in isolation
 - connected modules again verified
 - lots of test cases required
 - luckily automatization possible (tool support available)
- measurements of actual hardware
 - final stage of functional verification
 - logic analyzer / Signal Tap (Quartus)



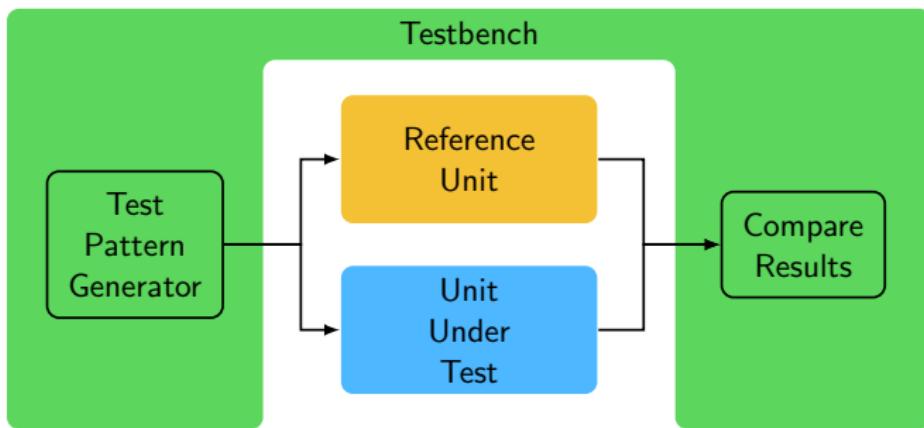
Flavors of Testing

- test response analysis inside testbench
- results compared to reference implementation
- input and output read / written from / to file



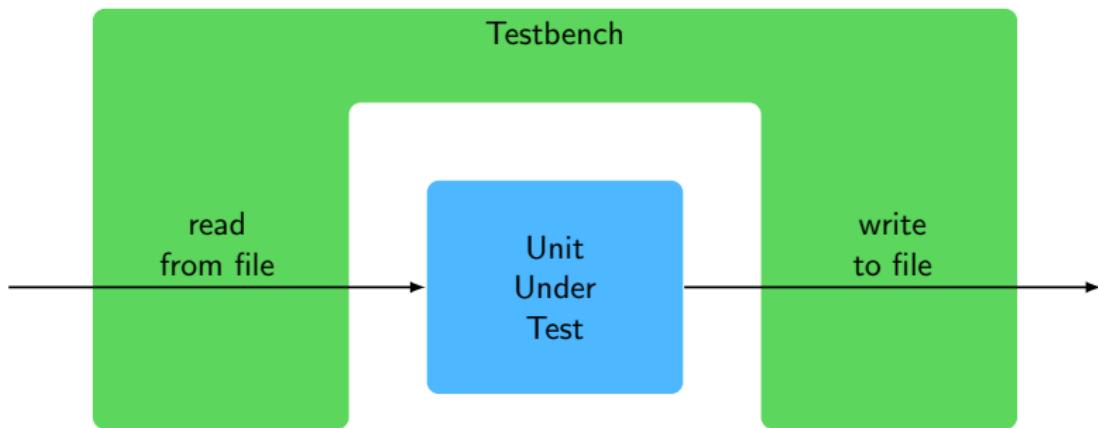
Flavors of Testing

- test response analysis inside testbench
- results compared to reference implementation
- input and output read / written from / to file



Flavors of Testing

- test response analysis inside testbench
- results compared to reference implementation
- input and output read / written from / to file



Automated Simulation

- in general lots of test vectors required
- until now each signal applied by hand
 - `i <= '0'; wait for 1 ns; ...`
 - `force i 0; run 1 ns; ...`
 - very cumbersome procedure for large test sets
- output observed in wave viewer
 - very time consuming procedure
 - not applicable for many signals / long traces
- test automation desired
 - inputs (read files, functions / procedures)
 - results (assert, report, write to file, functions / procedures)
 - tool chain (command line, TCL scripts)

Handling Inputs

- pseudo-random methods
 - for example linear feedback shift registers (LFSR)
 - hard to get good coverage of all test cases
- functions / procedures
 - generate more complex data in subprograms
 - well suited for recurring structures (e.g. in communication protocols)
- read data from file
 - easy to apply high volume of test vectors
 - possible to provide switching time and values
 - process reads data and applies them automatically
 - files usually generated by automatic test pattern generators (ATPG)
 - may also be used for complex configurations

Handling Inputs cont'd

```
1  # time [fs] value
2  0 0
3  999950 1
4  1029050 0
5  1066702 1
6  1113059 0
7  1159994 1
8  1196568 0
9  1228063 1
10 1248844 0
11 1270283 1
12 1299323 0
13 1328835 1
14 1352647 0
15 1395668 1
16 1419062 0
17 1436748 1
18 1469295 0
19 1490386 1
20 1529913 0
21 USE std.textio.ALL;
22 generate_input: PROCESS
23   FILE vector_file : text;
24   VARIABLE l : line;
25   VARIABLE vector_time : time;
26   VARIABLE r : integer;
27   VARIABLE good_number : boolean;
28 BEGIN
29   file_open(vector_file, "vectors", READ_MODE);
30   WHILE NOT endfile(vector_file) LOOP
31     readline(vector_file, l);
32     -- read the time from the beginning of the line
33     -- skip the line if it doesn't start with a number
34     read(l, r, good => good_number);
35     NEXT WHEN NOT good_number;
36
37   vector_time := r * 1 fs;
38   -- convert real number to time
39
40   IF (now < vector_time) THEN
41     -- wait until the vector time
42     WAIT FOR vector.time - now;
43   END IF;
```

read_file.vhd

Handling Inputs cont'd

```
1  # time [fs] value          44  FOR i IN l'RANGE LOOP
2  0 0                         45  CASE l(i) IS
3  999950 1                     46    WHEN '0' => -- Drive 0
4  1029050 0                   47      input <= '0';
5  1066702 1                   48    WHEN '1' => -- Drive 1
6  1113059 0                   49      input <= '1';
7  1159994 1                   50    WHEN ' ' | ht => -- Skip white space
8  1196568 0                   51      NEXT;
9  1228063 1                   52    WHEN OTHERS =>
10 1248844 0                  53      -- Illegal character
11 1270283 1                  54      ASSERT false
12 1299323 0                  55      REPORT "Illegal_character_in_vector_file:" & l(i);
13 1328835 1                  56      EXIT;
14 1352647 0                  57      END CASE;
15 1395668 1                  58      END LOOP;
16 1419062 0                  59
17 1436748 1                  60      END LOOP;
18 1469295 0                  61      REPORT "Test_complete";
19 1490386 1                  62      file_close(vector_file);
20 1529913 0                  63      WAIT;
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64  END PROCESS;
```

read_file.vhd

Alias II

- access signals inside (nested) entities using alias
 - **alias DONE_SIG is <<signal .TOP.DUT.DONE: BIT>>;**
- deliberately introduce errors
 - simulate transient / stuck-at faults
 - more fine grained testing possible

```
<signal> <= force [ in | out ] expression;  
<signal> <= release [ in | out ] ;
```

File I/O

- package

```
use std.textio.all;  
use ieee.std_logic_textio.all;
```

- file objects

```
file <name> : TEXT;
```

- strings

```
variable <name> : LINE;  
variable <name> : STRING (N downto 1);
```

- open / close

```
file_open(<text_var>, <file_name>, <mode>);  
file_close(<text_var>);
```

- read

```
readline(<text_var>, <line_var>);  
read(<line_var>, <sink_var>);  
write in the same fashion (conversion to string!)
```

Result Handling

- check if simulation deliver expected results
 - do outputs fit to provided input data?
did we build the system right?
 - is output according to specification?
did we build the right system?
- write results and check later
 - transcript window content written to *transcript*
 - waveform can be written to .wlf file (binary!)
 - value change dump file (.vcd) better processible (text format)
 - tools available to display contents
- compare to reference inside testbench
 - read from file or co-simulate known correct design
 - functions / procedure
 - assert / report

Assert

- if boolean *condition* not true
 - display error message “Assertion violation.” in transcript
 - if **report** specified *expression* is printed instead
- based on chosen **severity** different effects
 - can be configured in tools
 - possible values *NOTE*, *WARNING*, *ERROR*, *FAILURE*
 - default value *ERROR*

```
TB_I1 <= "10";
...
[ label: ] assert condition
  [ report expression ]
  [ severity expression ] ;
    TB_S <= "01";
    wait for 2 ns;
assert TB_O = "10"
  report "output_not_equal_to_10!"
  severity ERROR;
...

```

Assertion Based Verification

- for long waves cumbersome to check each signal
 - use assertions to indicate wrong behavior
- concurrently to code development
 - early detection of bugs possible
 - errors found close to their source
- designer has to think about wrong behavior early
 - gain additional insights
 - flaws in the specification and detected earlier
- can catch unintended changes by later modifications

Tool Chain



- tools can be run from command line
 - no graphical user interface (GUI) invoked
 - everything controllable from scripts
- process output in succeeding tools
- advantages
 - enables highly automated verification tool chain
 - work speed increases significantly
- Modelsim
 - start with `vsim -c <tcl script>`
 - slight modifications to scripts required

TCL script

```
1 # delete work directory if exists
2 if {[file isdirectory work]} {file delete -force work};
3 vlib work
4
5 # compile VHDL file
6 vcom -work work -2008 test_file.vhd
7 vcom -work work -2008 testbench.vhd
8
9 vsim work.TB -t fs
10
11 # write .vcd file
12 vcd file test_output.vcd
13 run -all
14
15 # quit simulation and Modelsim
16 quit -sim
17 quit -f
```

vsim.tcl

Measurements

- last verification step
 - necessary as accuracy of simulations limited
- measure signals on hardware / FPGA
- reuse test patterns from simulations
- logic analyzer
 - forward signals to output pins
 - not exactly same signals as deep inside chip
- Signal Tap (Quartus)
 - trigger on events and store traces
 - limited by available resources on FPGA
 - adds additional logic (design altered)

Coverage

- gives hints about completeness of testcases
 - the higher the better
- requirement coverage
 - manual check if every requirement in specification covered
 - results in important documentation for customers
- code coverage
 - how many lines of the code have been checked
 - supported by tools (Modelsim)
 - problems in two/three process method
 - o lines visited but not actually executed

Code Coverage

- typically used for behavioral simulation
- gives statistics about execution of lines of code
- with multiple testcases, the statistics need to be combined
- coverage below 100% indicates
 - incomplete test
 - unreachable code
- for code that must never be executed, insert assertions and exclude from coverage statistics:

```
1 counter.next <= counter + 1;  
2 -- coverage off  
3 assert counter < 12  
4 report "Counter_Overflow" severity error;  
5 -- coverage on
```

Design for Testability

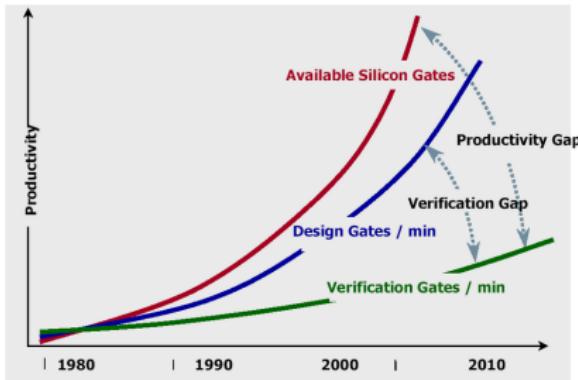
- implement separate logic used for testing
 - implementation is altered!
- Signal Tap (Quartus)
 - adds memory on chip to record behavior
- Scan Chain
 - add connect flip flops which record internal state
 - content can shifted out via separate pins
- built-in-self-test (BIST)
 - chip is tested during run time
 - error case is indicated

Formal Verification

- Problem of functional verification
 - how can one be sure that all faults covered?
 - when stop testing?
- alternative approach: formal verification
 - provides proofs for desired properties
 - require accurate model
- model checking
 - conditions to verify specified e.g. by **assert**
- equivalence checking
 - comparison of behavior desirable
 - mostly only structural available

Formal Verification cont'd

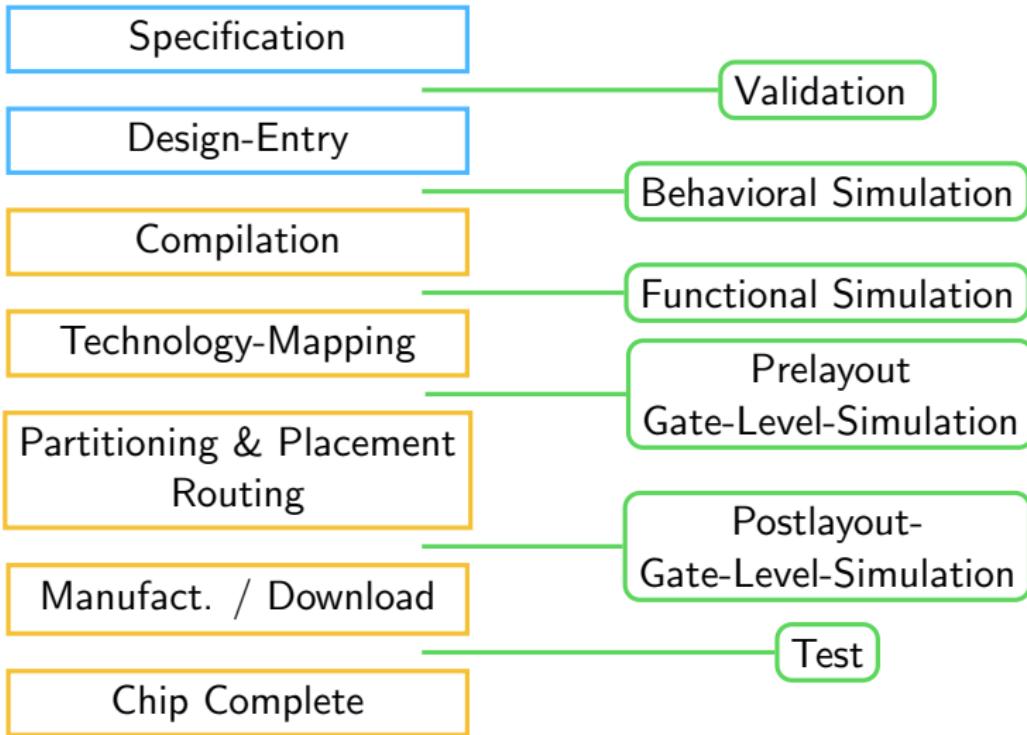
- becomes more and more important
 - growing demand for verification engineers
 - complexity that can be handled grows slowly
- historically very late in design process
 - in general only small parts can be checked
 - used for critical parts of circuit
- nowadays already integration in earlier development stages
 - compare to assertion based verification



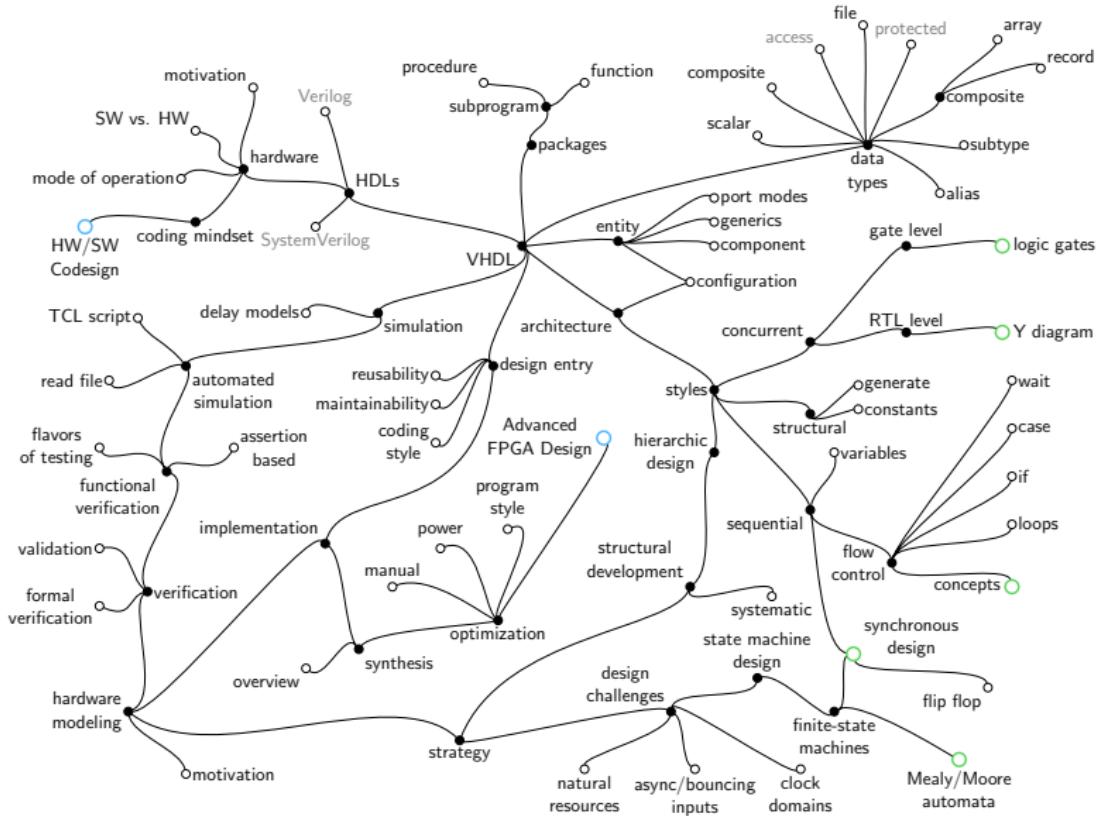
Formal Verification in HDLs

- in general formal verification at end
 - should also be done at design time
 - tight integration in HDLs desirable
- write formal properties directly in code
 - **assert**, comments, ...
- Property Specification Language (PSL)
 - not part of VHDL, handled by separate tool
 - always ($a \rightarrow \text{next}[3] b$)
- **YoSys**
 - developed by alumni of TU Wien (Clifford Wolf)
 - currently only available for Verilog

Design Flow



The End



The End

things left to do:

- program, program, program
 - practice, practice, practice
 - [LU Digital Design and Computer Architecture](#) very good opportunity
- solve examples provided in TUWEL
 - help you understand content better
 - show you where you still lack knowledge
 - are very similar to exam questions ;)
- attend the exam
 - dates regularly available (registration via TISS)
 - cheat sheet will be provided

The End cont'd

- if you liked hardware design keep practicing
 - visit succeeding lectures
 - explore VHDL in more detail
 - yet a lot left to learn
- rate lecture in TISS, provide feedback in TUWEL
 - both anonymous
 - please tell us what you did not understand well
 - help to improve the quality of the lecture
- if you really loved the lecture
 - there is a teaching award *hint* ;)

Good Luck