

# HDL Lecture: Verilog and VHDL



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**Summer Term 2013**

**Prof. Dr.-Ing. Klaus Hofmann  
M.Tech. Ashok Jaiswal**

<http://www.ies.tu-darmstadt.de>

# Organizational Information (I)

- This lecture is intended for students enrolled in the following departments:
    - Electrical Engineering and Information Technology (FB18)  
**ETiT**: Electrical Engineering and Information Technology
    - MEC**: Mechatronics
    - iKT**: Information and Communication Technology
    - ICE**: Information and Communication Engineering
  - Computer Science (FB20)
- 
- Mandatory requirements: Boolean algebra, one programming language (Java, Perl, VB, .net, C/C++, ...), digital logic.
  - Desired requirements: basics of computer architecture
  - You are perfectly prepared if you have attended or are attending “Advanced Digital Integrated Circuit Design”
  - The topics addressed in this lecture can be practiced in detail in the HDL lab

# Organizational Information (II)

Date & Place: Thursday, 9:50am -11:30am (Room S3/06/052)

Written exam: pls check in the online course overview. You must register for the lecture and exam using Tucan!

As a block lab, we are offering the „HDL lab“ in the summer break starting Aug 5 2013 (2wks of full time programming). Registration also works using Tucan. Note that seats are limited!

Programming tasks should be an integral part of this lecture. We strongly recommend you to do the programming tasks, so you are fit to pass the exam. You may do the programming tasks at home using a Verilog Freeware tool (Icarus + GTKWave). Runs on Windows + Linux + Mac machines.

Script: You may buy this script from City Copies (Holzstrasse 5, 64283 Darmstadt)

Teaching assistants: M. Tech. Ashok Jaiswal

Consultation: right after lecture, or upon request (if only 4-eye meeting wanted)

All contact data, download information, links to the lecture catalog, examination dates, news, announcements available under

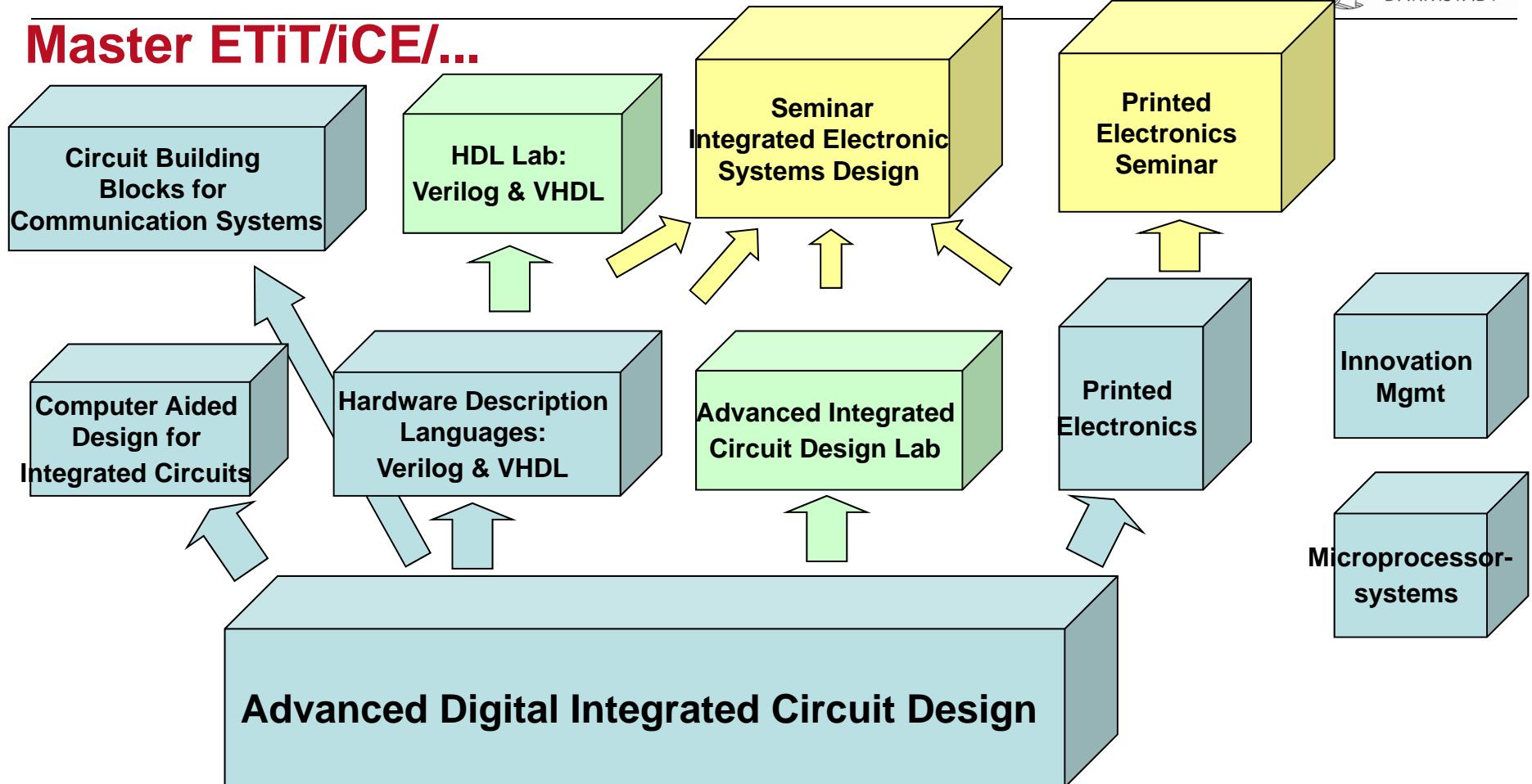
<http://www.ies.tu-darmstadt.de> and TUCAN

**Content of the exam will be everything that is either presented or orally told during the next weeks/months!**

On the exam: you must register for this exam using Tucan. On the location and time, please check relevant university pages or Tucan messages or link on <http://www.ies.tu-darmstadt.de> -> lectures -> HDL lecture

# IES further lectures

Master ETiT/iCE/...



Bachelor

Analog Integrated  
Circuit Design

Proseminar

The contents of the lectures are covered by the following book:

**[1] Thomas & Moorby's: The Verilog Hardware Description Language (Fifth Edition), Springer, 2002 (ISBN: 978-0-84930-0)**

Check out using <http://www.springerlink.com> for pdf of this book – you may also check out for other books on Verilog and/or VHDL

Further recommended texts:

[2] Hennessy, John L. and Patterson, David A.: Computer Architecture – A Quantitative Approach, Morgan Kaufman, 3. Auflage 2003

[3] Adel S. Sedra und Kenneth C. Smith: Microelectronic Circuits, Oxford University Press, 1991 (ISBN: 0-19-510369-6)

# Goal of this lecture

- Step 1: To give you a basic introduction to a hardware description language (Complete Verilog, plus a bit of VHDL), and the logic synthesis based design flow for synthesizing hardware

?

a very first step in the right  
direction



# Goal of this lecture

- Step 2: Practice the learned language constructs in your own speed: modeling and simulation of a simple, but rather complete microprocessor.



a first step in the right direction



# Contents

- Introduction & History of Verilog
- Semicustom Fundamentals (The digital design flow with Verilog (RTL -> GL -> F, P&R, Formal verification, STA, Advanced Verification))
- Structural Descriptions (GL-Netlist). Describes basic language constructs, shows how a simulation works (testbench, patterns)
- Syntax, Lexical Conventions, Data Types, Memory
- Structural and Behavioral Verilog description of combinational circuits (Example)
- Behavioral modeling (process model, if-then-else, functions and tasks, concurrent processes, events)
- Procedural modeling of Clocked Sequential Circuits (State-diagram example of FSM)
- Module hierarchy
- RTL Logic Synthesis (using examples, demonstrating procedural statements, sensitivity list, case statements, inferring sequential elements (Latch, D-FF), FSM, Datapath)

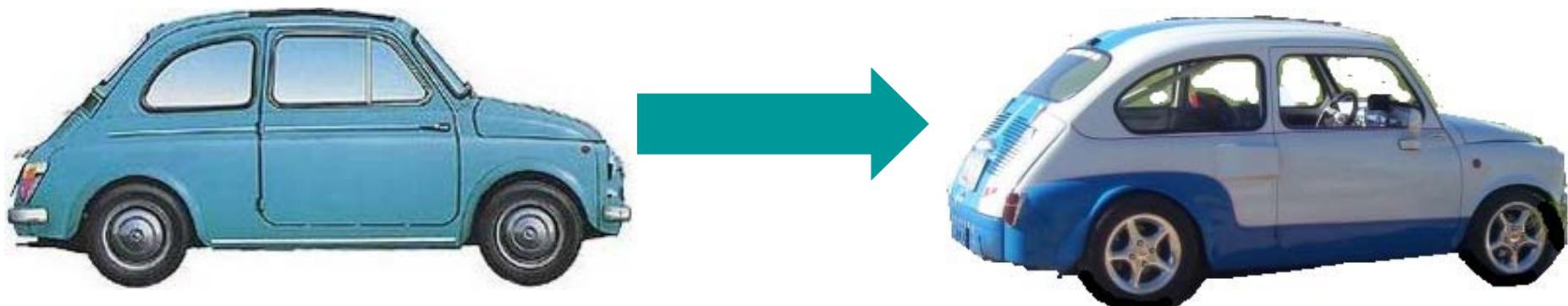
# Contents

- Advanced Hardware Modeling; Logic states (simple, user-defined primitives, strength definition), switch level modelling and simulation
- Coding Style
- Programming Language Interface (PLI), System Verilog (Outlook)
  
- Basic idea: all language constructs in Verilog, plus at the end „hands-on“ comparison Verilog vs. VHDL (using one example)

# Outlook

- If you liked this lecture, you may continue with our HDL Lab (starting August 6)
- In this lab, you will have the opportunity to practice your coding using an advanced project (pipelined microprocessor or pipelined digital signal processor)

Another step in the right direction



# From good to great

- If state-of-the-art hardware design and/or modeling is what you like, we have something for you ....



- IES Lab is always offering Bachelor/Master thesis topics based on Verilog/Verilog-A/VHDL/SystemC

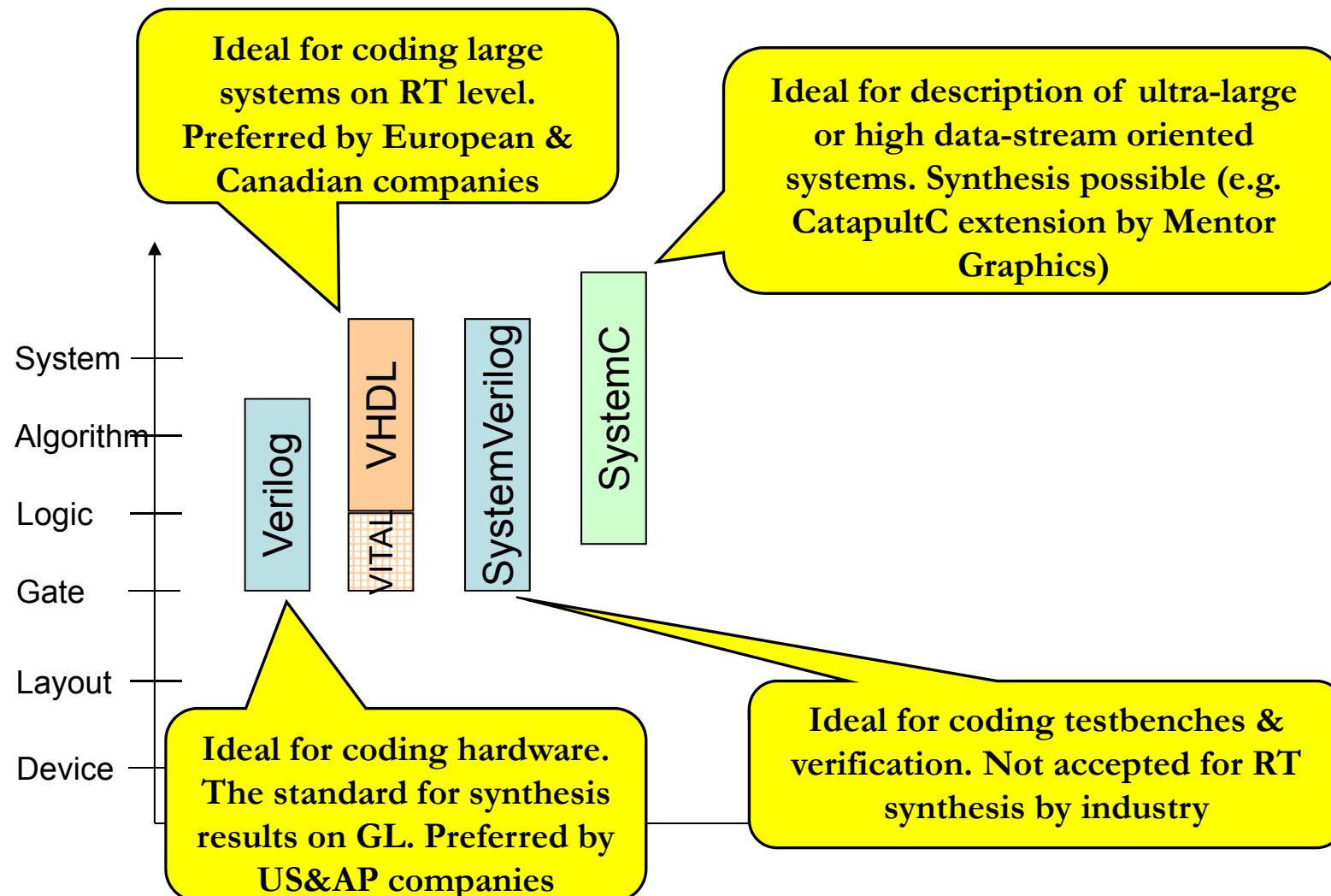
# History of Verilog

- 1982: HiLo is a popular hardware description language (by GenRad)
- 1984: Phil Moorby (who was co-developing HiLo) invents Verilog
- 1986: The Verilog-XL simulator (by Gateway) is the most-powerful simulator for digital circuits
- 1990: Cadence acquires Gateway and owns now Verilog and the Verilog-XL simulator. At the same time, Synopsys is pushing towards top-down logic synthesis.
- 1991: Cadence „opens“ the Verilog language by founding the OVI (Open Verilog International) initiative for developing and standardizing the HDL.
- 1992: Many companies offer Verilog simulators
- 1995: The Verilog LRM provided by OVI becomes the IEEE standard 1364.
- 2001: Latest version of Verilog is called Verilog-2001 (IEEE1364-2001)
- Several extensions are available towards verification and system description

# History of VHDL

- 1981: The United States Department of Defense recognizes the need for an HDL to „overcome the hardware life cycle crisis“. Sponsored with more than 30 Mio US\$
- 1983-85: Development of VHDL by IBM, Intermetrics and TI
- 1986: The DoD transfers all rights of VHDL to the IEEE
- 1987: Publication of an IEEE standard
- 1994: Publication of the VHDL-1993 standard
- 2000: Publication of the VHDL-2000 standard
- 2002: Publication of the VHDL-2002 standard
- 2007: Publication of the VHDL Procedural Language Application Interface standard (VHDL 1076c-2007)
- 2009: Revised standard (VHDL 1076-2008)

# Comparison of HD languages



# Comparison of HD languages

## Verilog HDL (with SystemVerilog extensions)

Verilog netlist instantiates the Verilog shell module that represents the SystemC model

```
module chip_top (...);  
    SC_alu_shell i1 (clk, a, b,  
                      opcode, result);  
    controller i2 (...);  
    RAM i3 (...);  
endmodule
```

The Verilog shell module directly calls the SystemC I/O function

```
SC_alu_shell (  
    input wire clk,  
    input real a, b,  
    input wire [7:0] opcode,  
    output real result );  
  
import "DPI" task SC_alu_run(...);  
  
always @(clk, a, b, opcode)  
    SC_alu_run(clk, opcode,  
               a, b, result,  
               $realtime);  
endmodule
```

## SystemC

I/O function to pass values and time to SystemC model

```
SC_alu_run (  
    int clk, opcode,  
    double a, b,  
    double* result,  
    double time) {  
  
    sc_start(time);  
  
    results = ...  
}
```

### SystemC Model

```
SC_alu_init() {  
    ...  
}  
  
SC_alu(...) {  
    ...  
}
```

### SystemC Kernel

# Comparison of HD languages

Verilog (with SystemVerilog extensions)	SystemC
<b>data types that are equivalent in both languages</b>	
byte	char
shortint	short int
int	int (32-bit), sc_int
longint	long long, sc_bigint
real	double
int unsigned	unsigned int (32 bit), sc_uint
longint unsigned	unsigned long long, sc_biguint
string	char*
structures	structures
unions	unions
enum (using default int type and values)	enum
chandle (not used within SystemVerilog, only used to store and pass C pointers)	void*

# Comparison of HD languages

similar data types that must be manually mapped between languages	
object handle	object handle
bit (2-state vector of arbitrary size)	sc_bit (2-state vector of arbitrary size)
logic, reg (4-state vector of arbitrary size)	sc_logic (4-state vector of arbitrary size)
unpacked arrays, 1 or more dimensions	unpacked arrays, 1 or more dimensions
dynamic arrays, associative arrays	dynamic arrays, associative arrays
data types unique to SystemC (must be manually converted to SystemVerilog)	
	sc_int<>, sc_bint<>, ... (sized integers)
	sc_fixed, sc_ufixed, ... (fixed point)
data types unique to SystemVerilog (must be manually converted to SystemC)	
packed arrays with more than 1 dimension	
queues	
integer, time (4-state, arbitrary size)	
wire, wand, wor, tri, triand, trior, trireg, pullup, pulldown (net types with strength)	

# Comparison of both languages

	VHDL	Verilog
Compilation	Multiple design-units (entity/architecture pairs), residing in the same system file, may be separately compiled if so desired.	Interpreted language! Compilation is a means of speeding up simulation, but has not changed the original nature of the language.
Data Types	A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.	Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, (e.g. wire), and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects, that are signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of its simplicity.

# Comparison of both languages

	VHDL	Verilog
Compilation	Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them	There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive
Efficiency	Due to the strongly typed nature, VHDL is less efficient than Verilog	Due to its robust architecture, very high efficiency.
Learning	Needs more practice and thorough understanding	Very easy to learn

# Comparison of both languages

	VHDL	Verilog
Extensions	Has an attribute called 'foreign' that allows architectures and subprograms to be modeled in another language	The Programming Language Interface (PLI) is an interface mechanism between Verilog models and Verilog software tools. For example, a designer, or more likely, a Verilog tool vendor, can specify user defined tasks or functions in the C programming language, and then call them from the Verilog source description. Use of such tasks or functions make a Verilog model nonstandard and so may not be usable by other Verilog tools. Their use is not recommended.
Libraries	A library is a store for compiled entities, architectures, packages and configurations. Useful for managing multiple design projects	There is no concept of a library in Verilog. This is due to its origins as an interpretive language.
Low level constructs	Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR.	The Verilog language was originally developed with gate level modeling in mind, and so has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries. Examples include User Defined Primitives (UDP), truth tables and the specify block for specifying timing delays across a module.

# Comparison of both languages

	VHDL	Verilog
Large designs	Configuration, generate, generic and package statements all help manage large design structures.	There are no statements in Verilog that help manage large designs.
Operators	About the same – except unary operators	About the same – plus some powerful unary operators
Procedures and tasks	concurrent procedure calls are allowed	concurrent procedure calls are not allowed

# Comparison of both languages

	VHDL	Verilog
Readibility	Depends on education – ADA programmers like VHDL more	C programmers like Verilog more.
Testbenches	Generic and configuration statements are helpful for writing testbenches	No concept such as generic or configuration
Verboseness	Because VHDL is a very strongly typed language models must be coded precisely with defined and matching data types. This may be considered an advantage or disadvantage. However, it does mean models are often more verbose, and the code often longer, than its Verilog equivalent.	Signals representing objects of different bits widths may be assigned to each other. The signal representing the smaller number of bits is automatically padded out to that of the larger number of bits, and is independent of whether it is the assigned signal or not. Unused bits will be automatically optimized away during the synthesis process. This has the advantage of not needing to model quite so explicitly as in VHDL, but does mean unintended modeling errors will not be identified by an analyzer.

# Why still Verilog?

SystemVerilog by Accellera available for unified hardware description and verification (IEEE-1800).

It was developed originally by Accellera to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips. SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system-level design flow. SystemVerilog has been adopted by 100's of semiconductor design companies – except for RTL synthesis.

# Conclusion

Hardware structure can be modeled equally effectively in both VHDL and Verilog. When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI, or by using SystemVerilog.

The choice of which to use is not therefore based solely on technical capability but on:

- personal preferences (education, Europe vs. Rest of world (U.S., Asia-Pacific))
- EDA tool availability (Verilog tools are always 6months ahead)
- commercial, business and marketing issues (executable model)

# Semi Custom Fundamentals



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Contents

*Motivation*

*Gajski Y-diagram*

*Semi vs. full custom*

*Semi custom flow*

*Implementation flow details*

*Motivation*

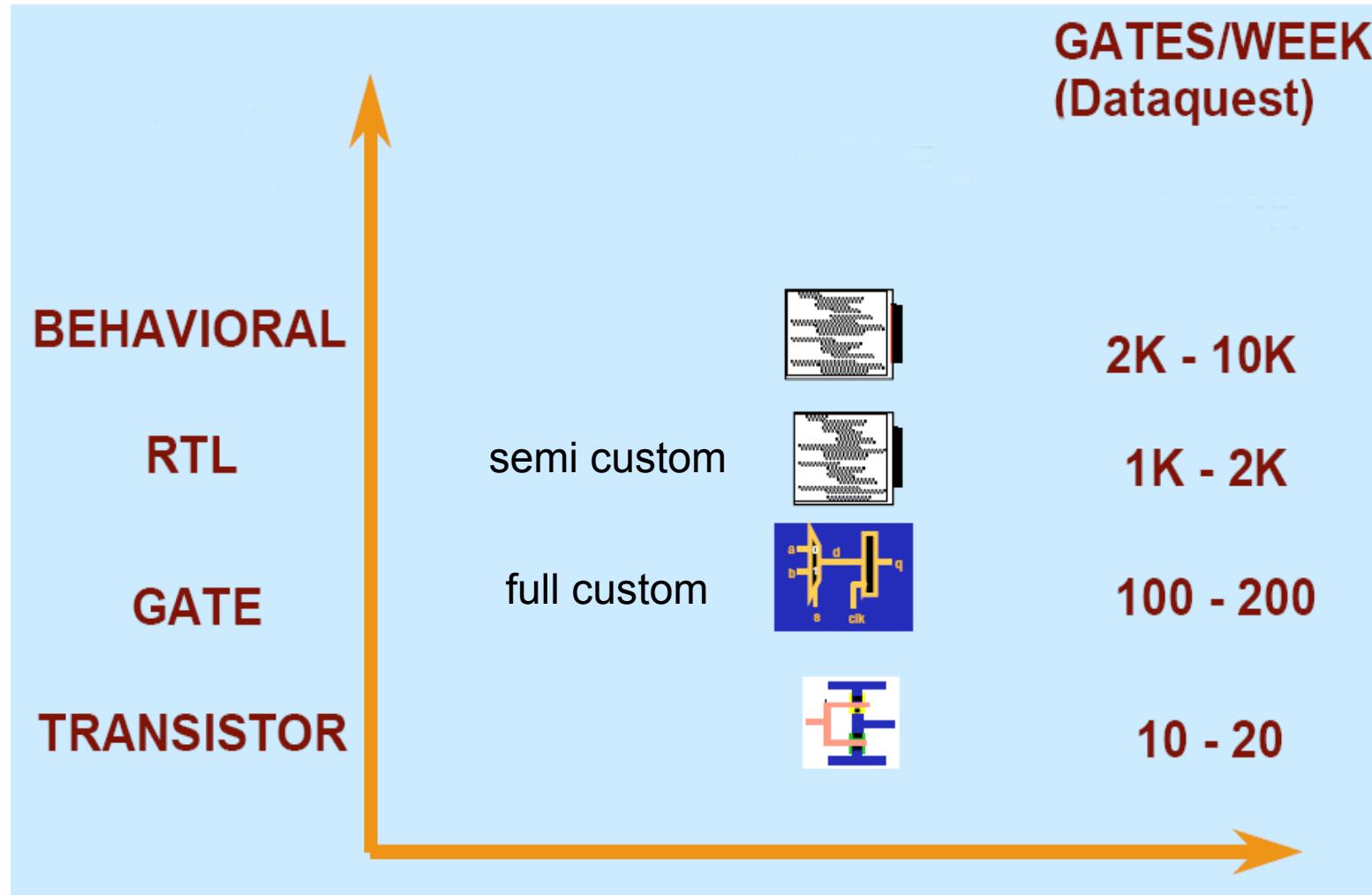
*Gajski Y-diagram*

*Semi vs. full custom*

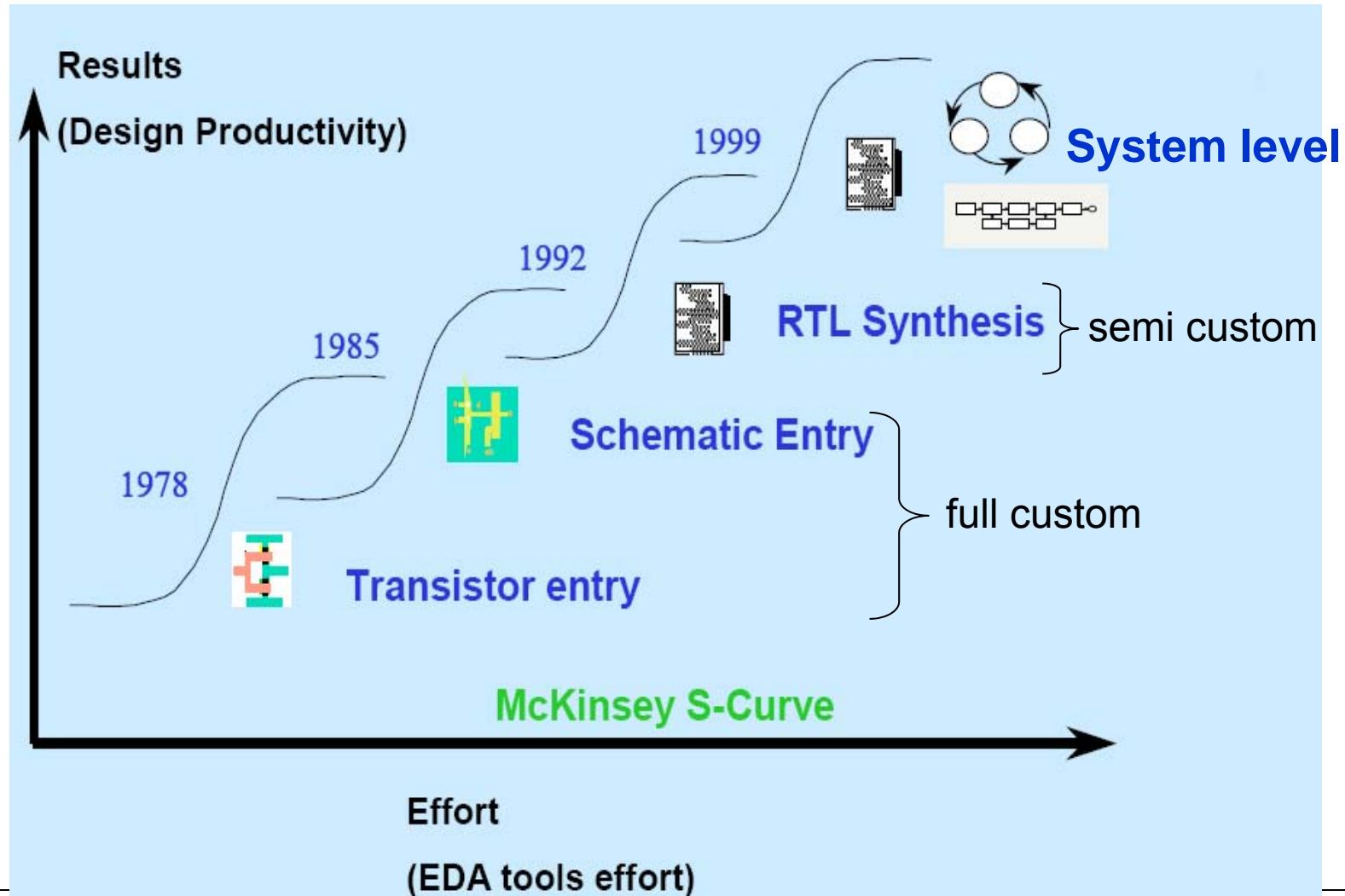
*Semi custom flow*

*Implementation flow details*

# Productivity estimation (from Dataquest)



# Design method history (McKinsey S-Curve)



*Motivation*

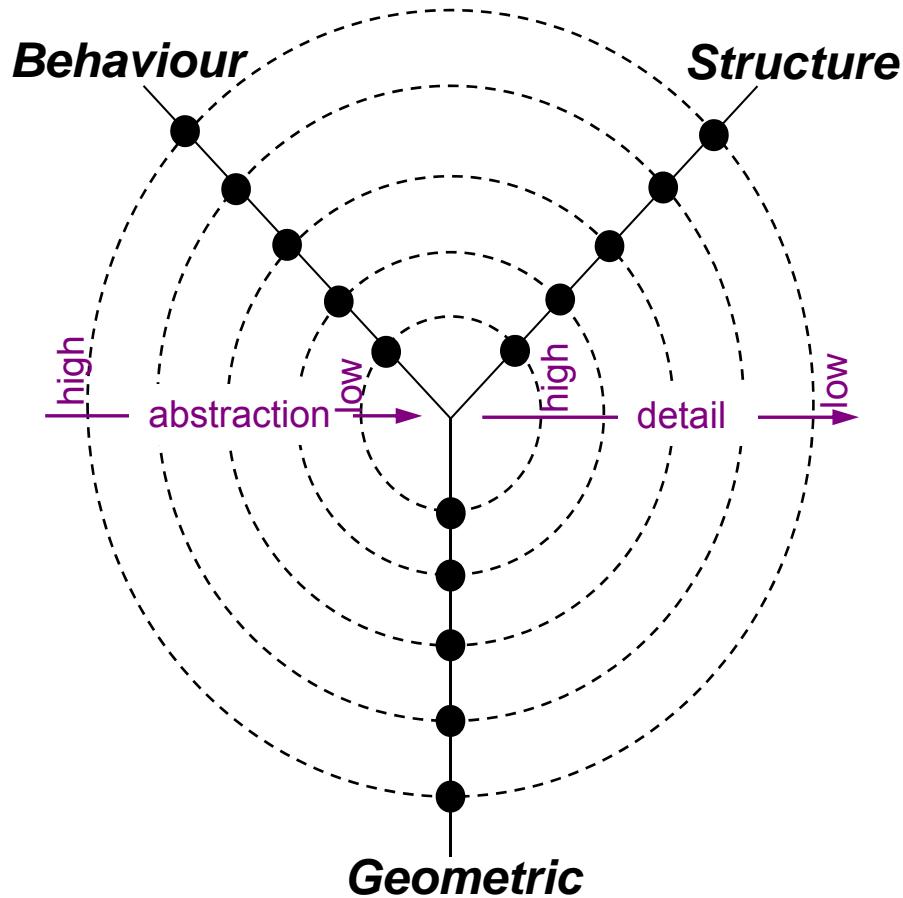
*Gajski Y-diagram*

*semi vs. full custom*

*semi custom flow*

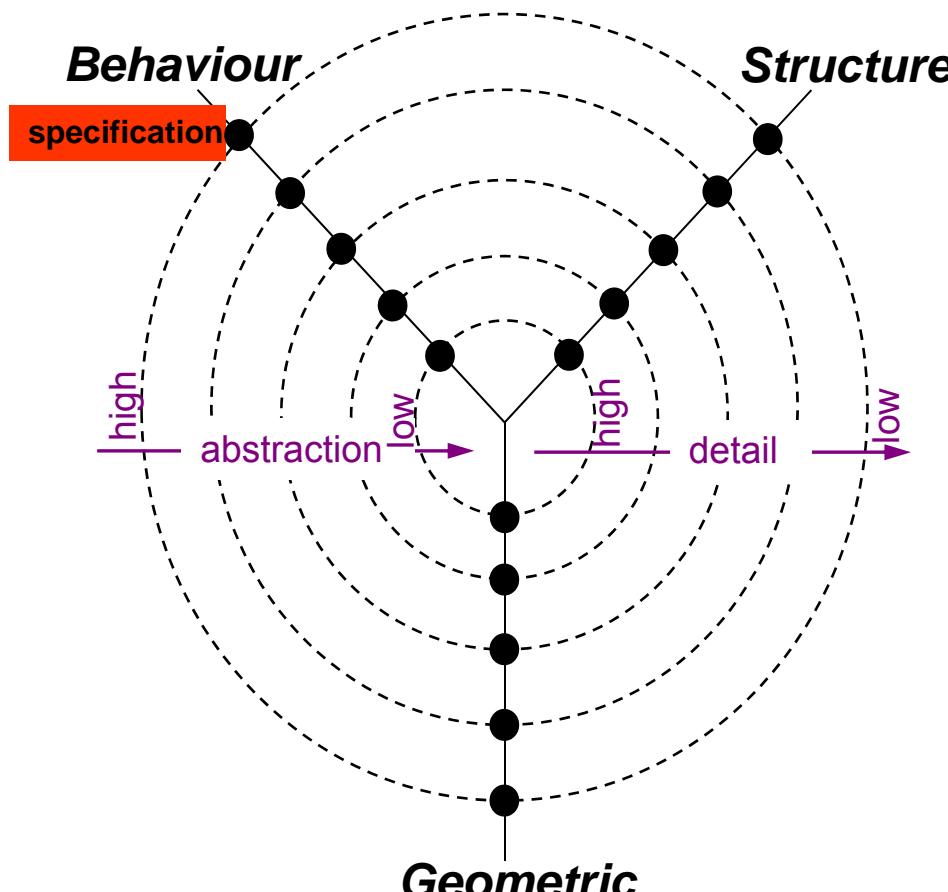
*Implementation flow details*

# Gajski Y-diagram

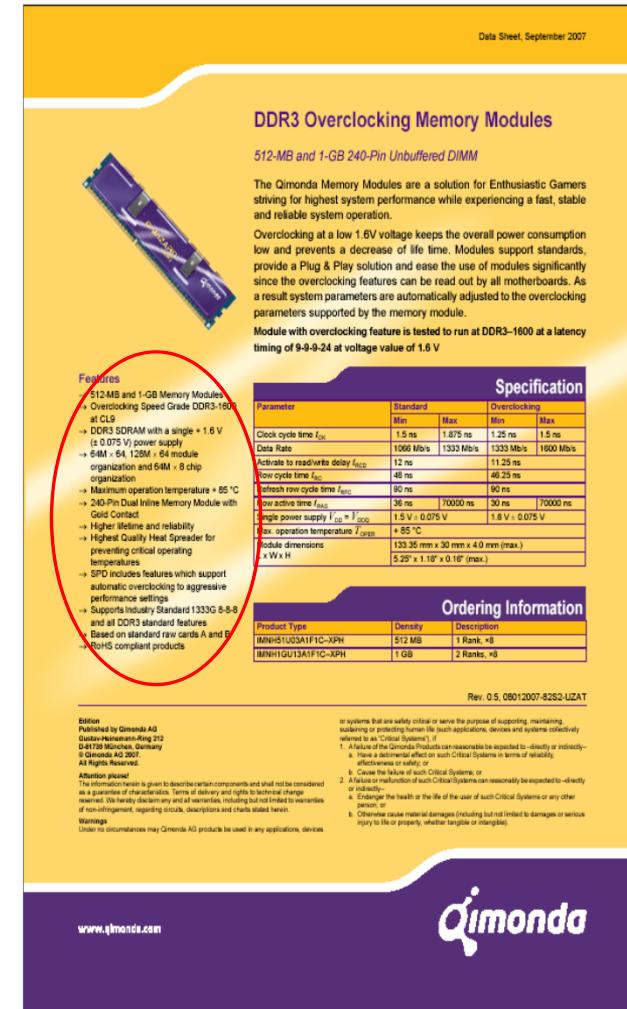


- 3 axis represent 3 forms of chip description
- outside: high level of abstraction, low detail level
- inside: low level of abstraction, high detail level

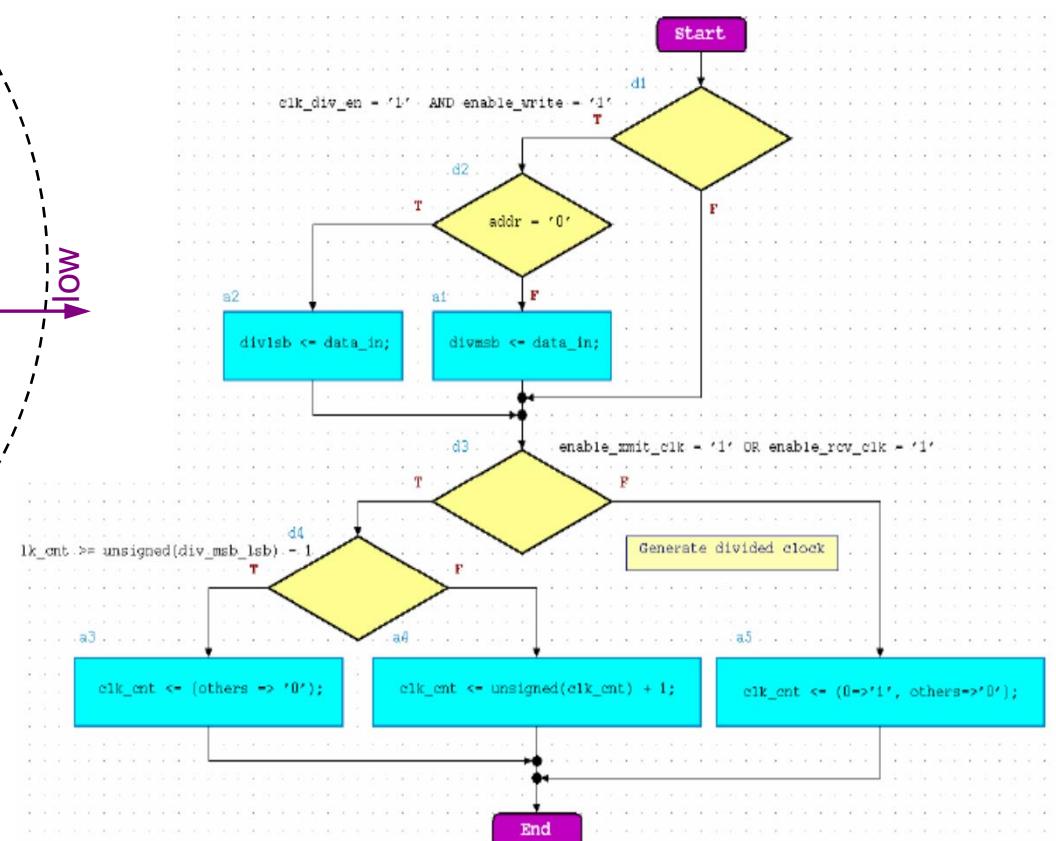
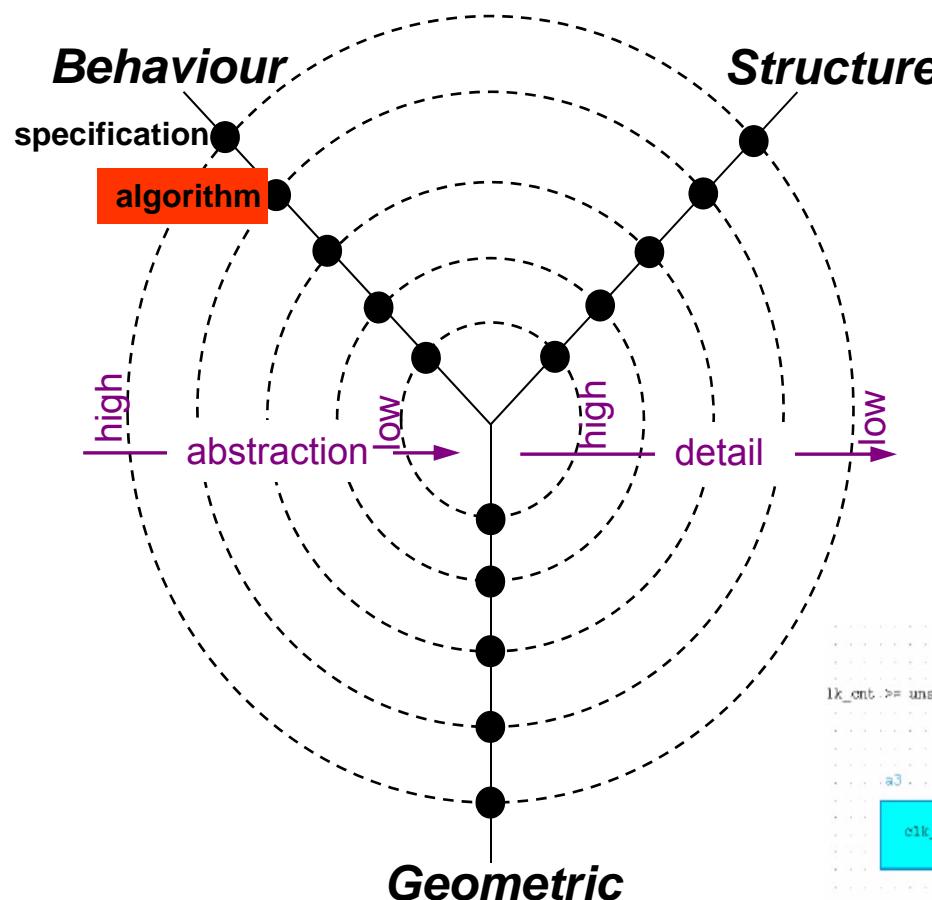
# Behaviour: Specification



- system description in natural language
- high level of abstraction, low detail level
- designer has to translate this description in a more exact and detailed form

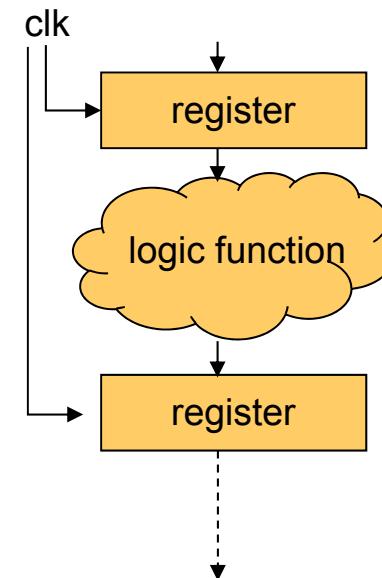
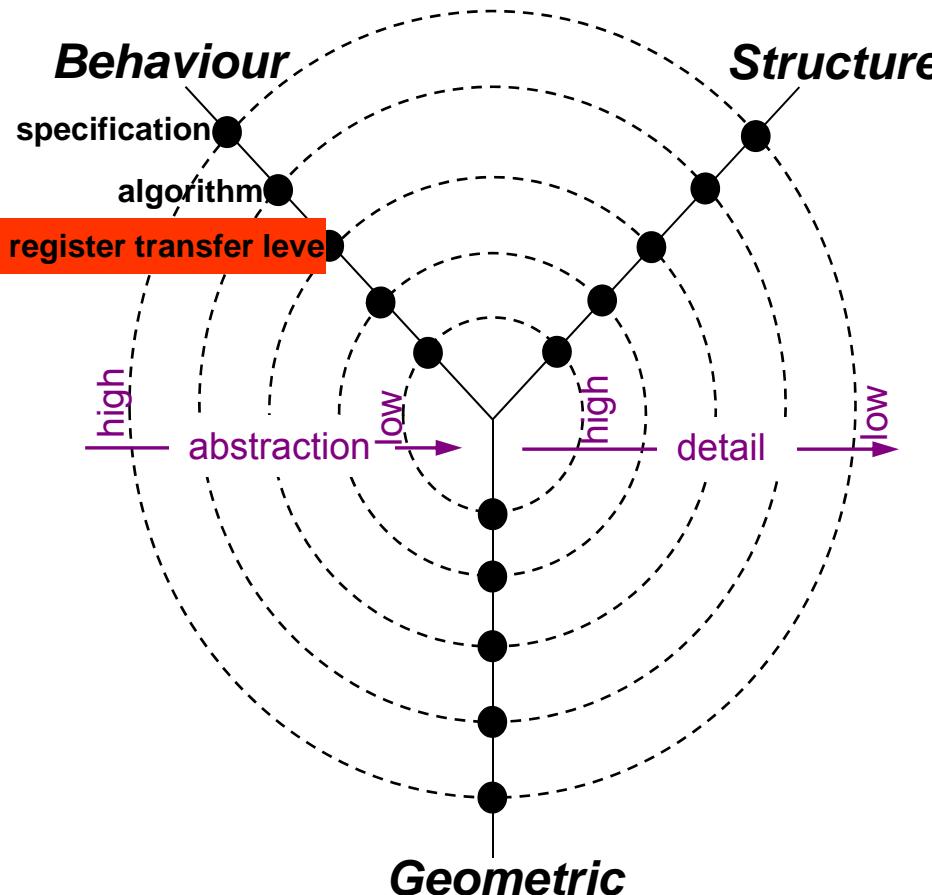


# Behaviour: Algorithm



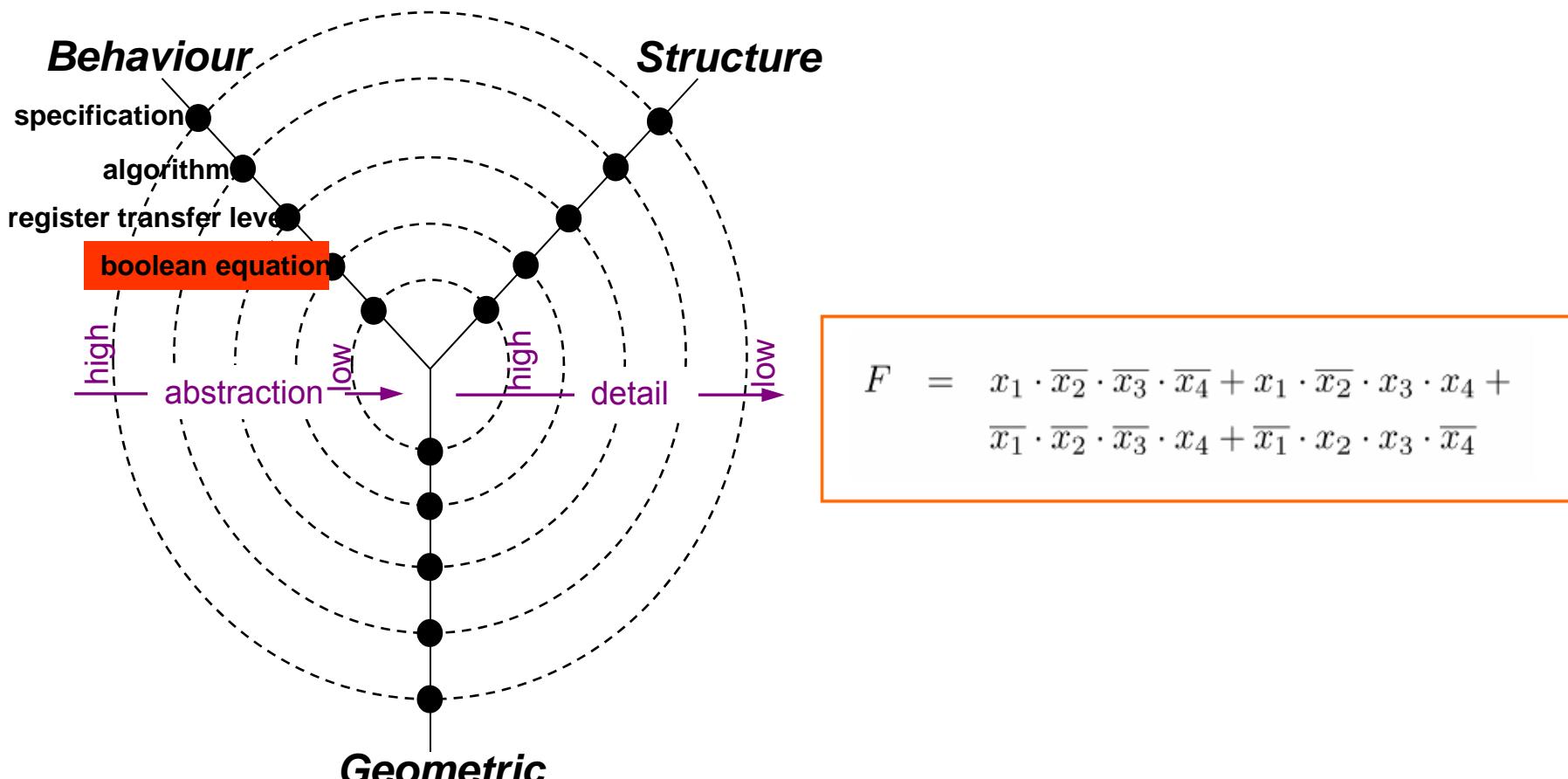
- important step:  
development of first algorithm which  
reflects system behaviour

# Behaviour: Register Transfer Level (RTL)



- most important form of HDL description
- provides enough detail level for automation of all following transformation steps
- connects the algorithm with the physical elements (register respectively flip-flops)
- first description including timing information

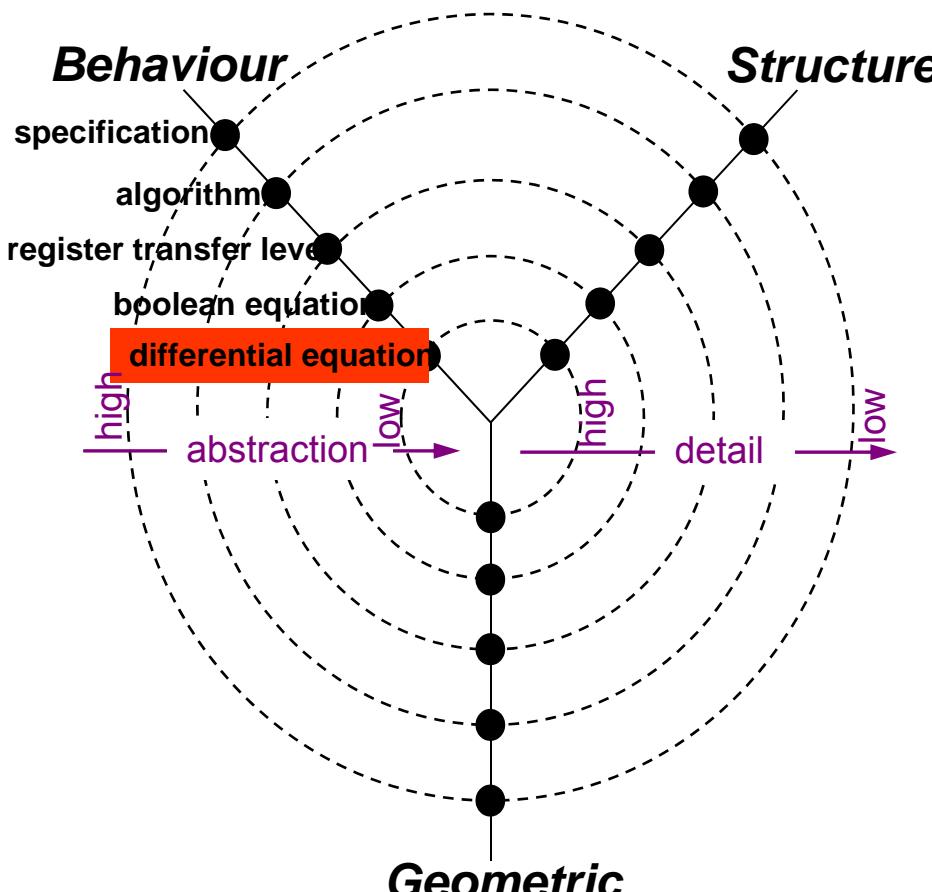
# Behaviour: Boolean Equation



$$\begin{aligned} F = & x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot \overline{x_2} \cdot x_3 \cdot x_4 + \\ & \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4 + \overline{x_1} \cdot x_2 \cdot x_3 \cdot \overline{x_4} \end{aligned}$$

- RTL to Boolean converting effort would be very intensive
- no benefit for designer of that high detail level
- in former time a common description form for digital systems

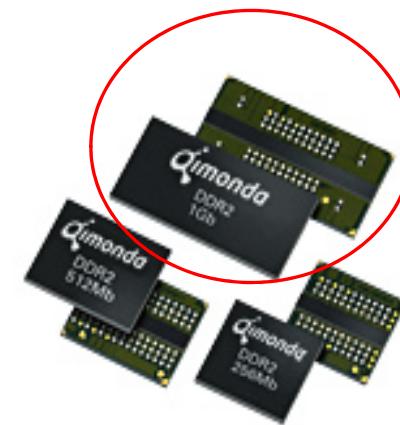
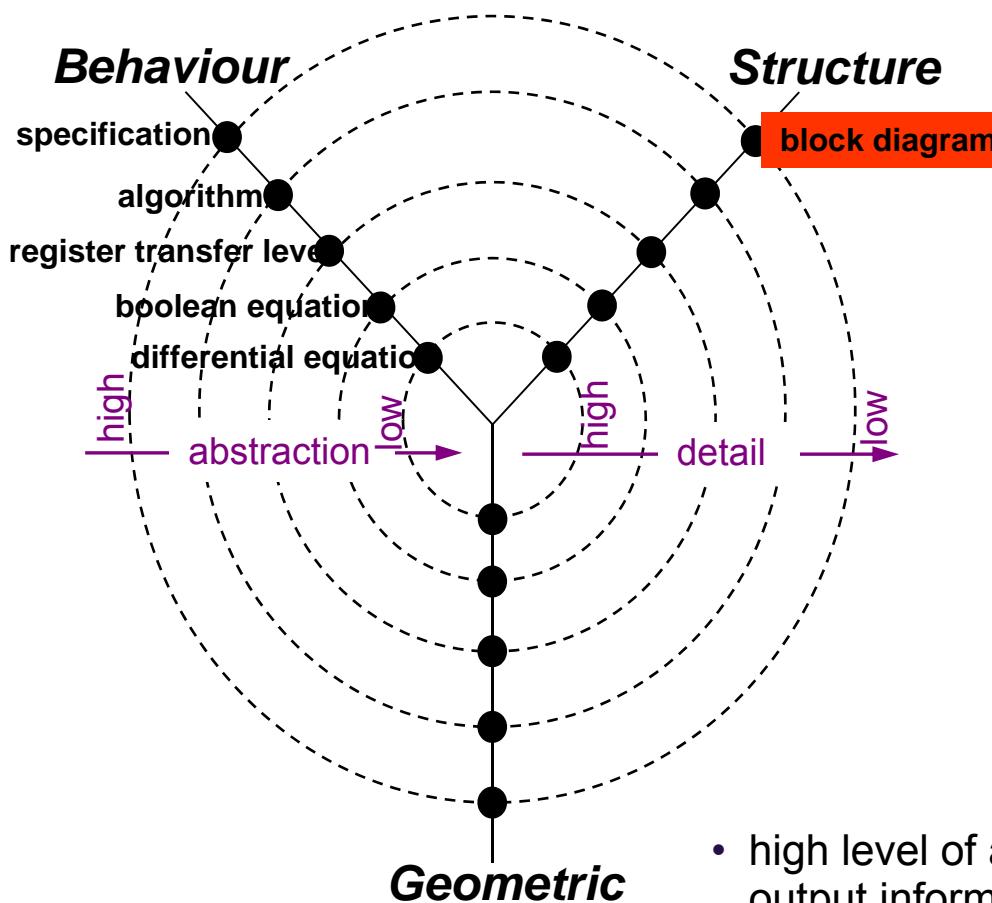
# Behaviour: Differential Equation



$$L \frac{d^2}{dt^2} I(t) + R \frac{d}{dt} I(t) + \frac{1}{C} I(t) = \frac{d}{dt} U_e(t)$$

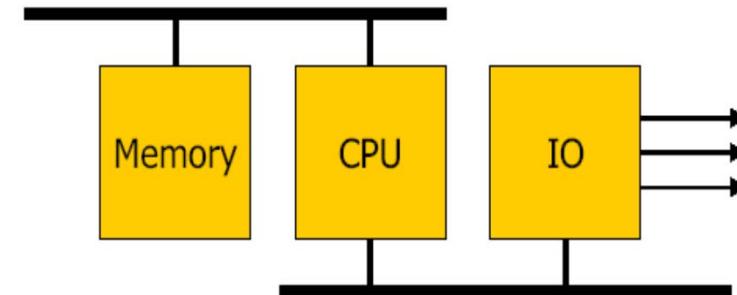
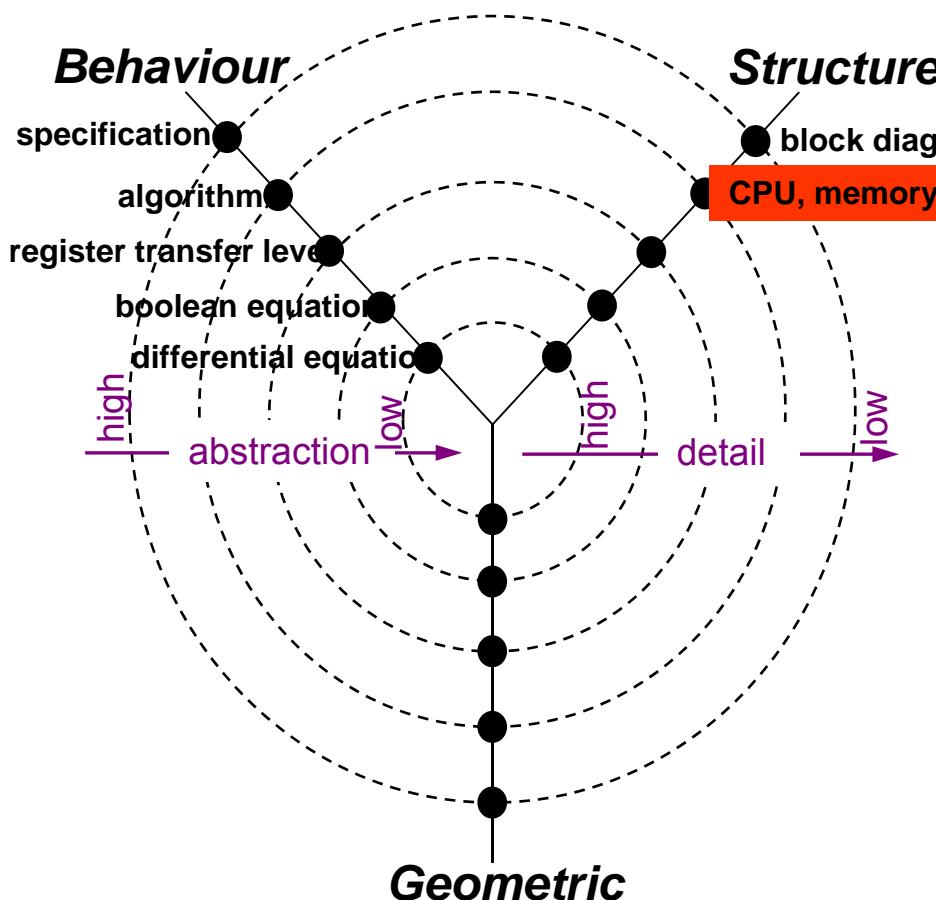
- no practical use for digital systems
- analog SPICE simulations are done on this level
- generation would be nearly impossible, too much details

# Structure: Block Diagram



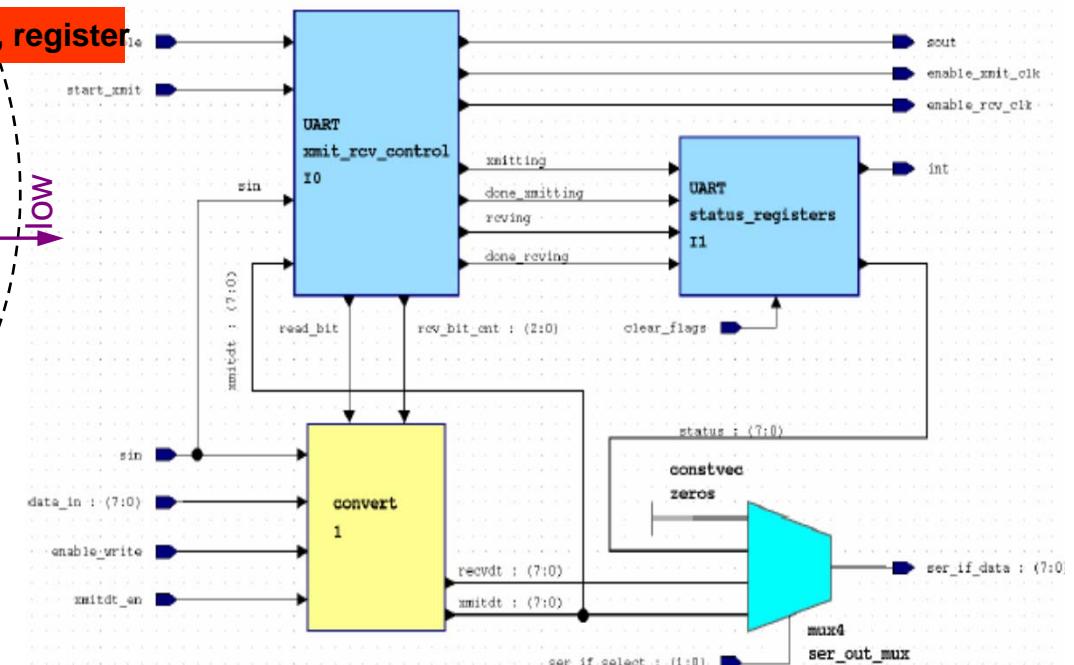
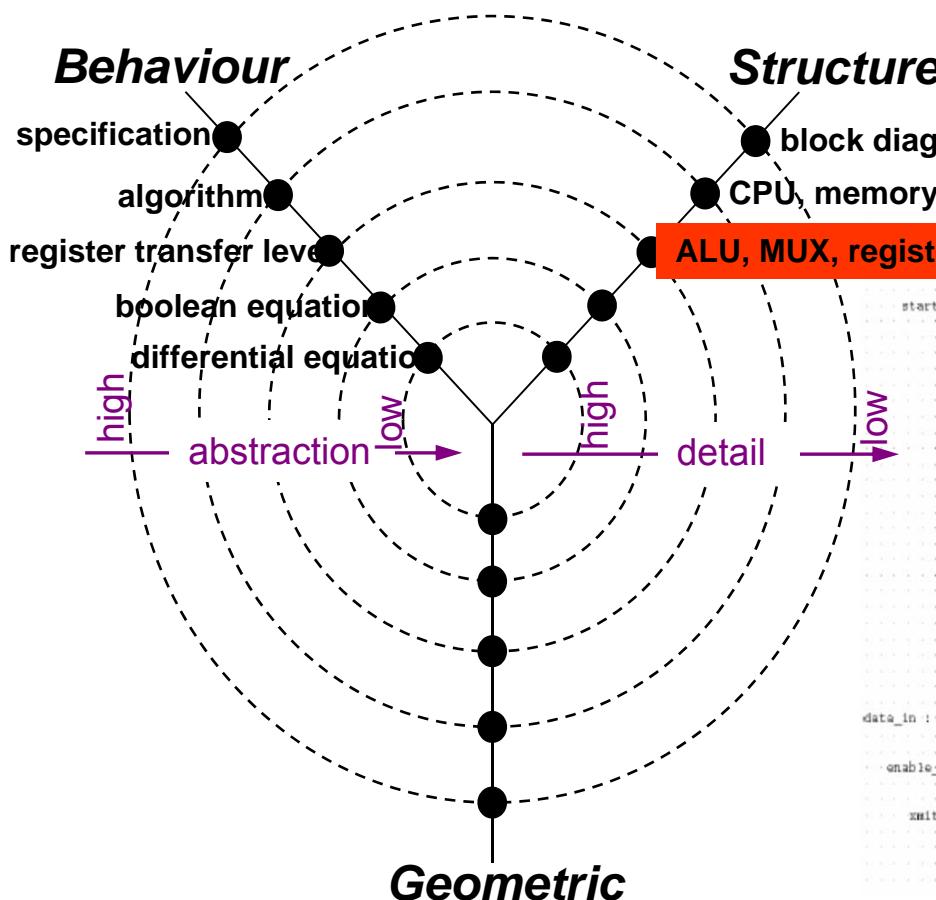
- high level of abstraction, only pin, input and output information available
- description form as often used in customer manuals

# Structure: CPU, memory



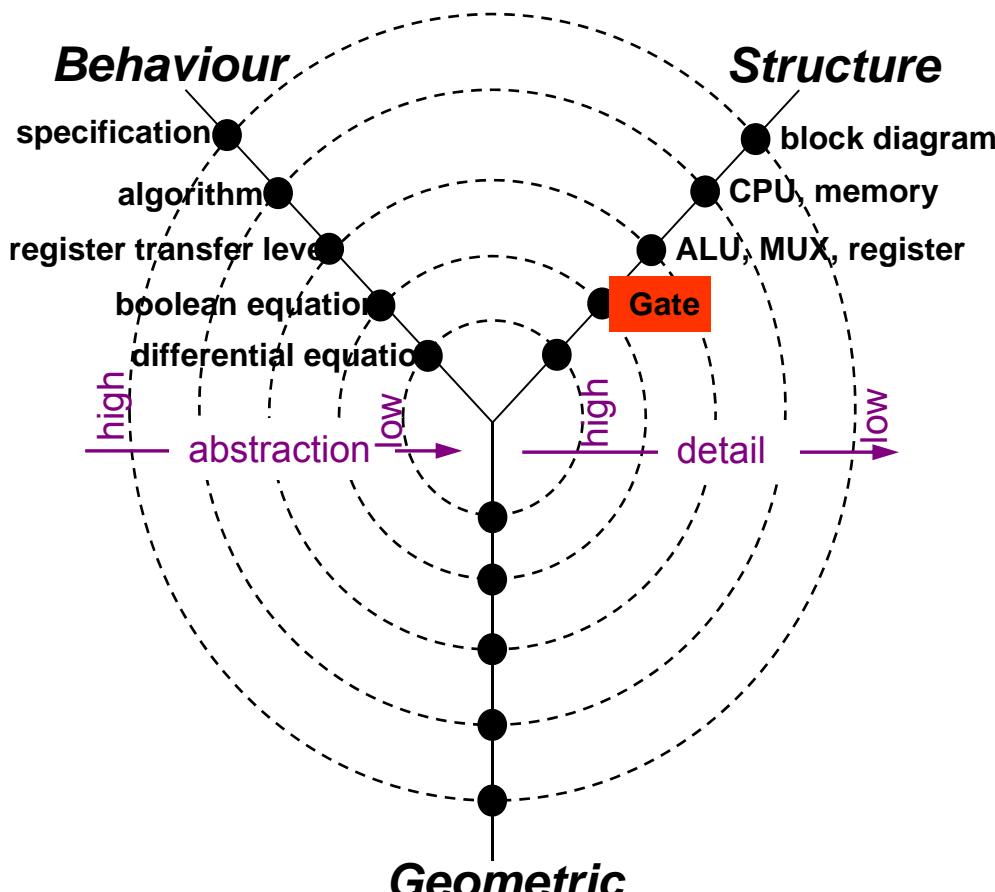
- further breakdown of the block diagram into sub modules
- e.g. PC (main board, CPU, memory, ...)

# Structure: ALU, MUX, register

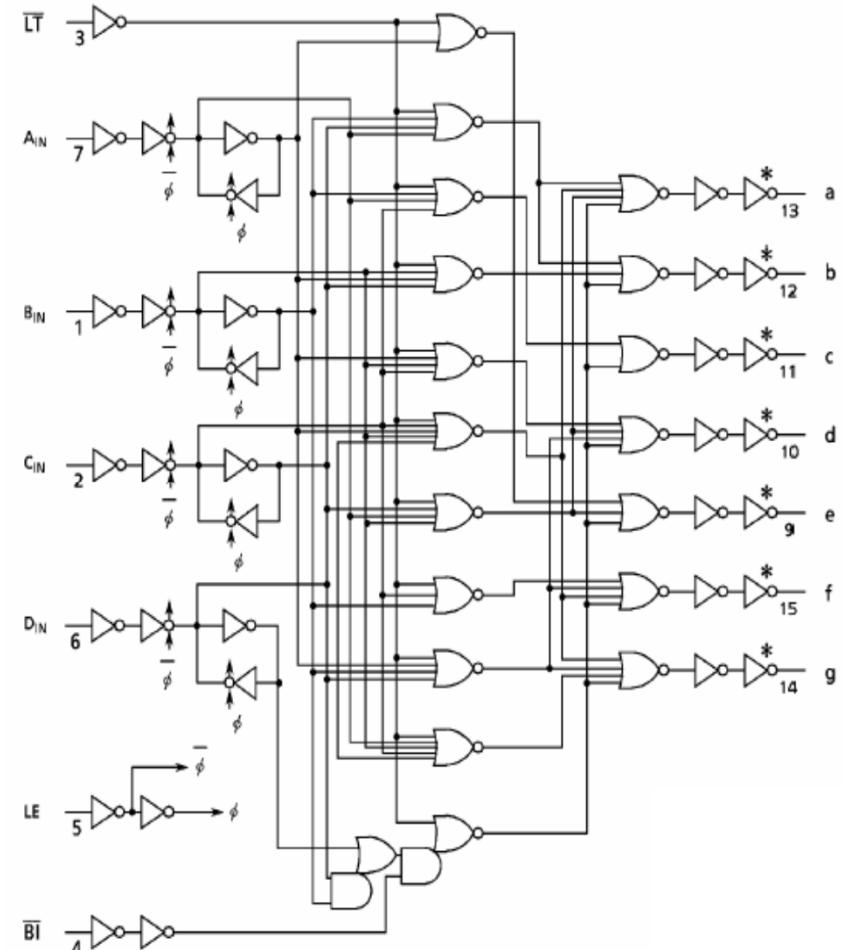


- represents the level of schematic entry as often used for the description of digital systems

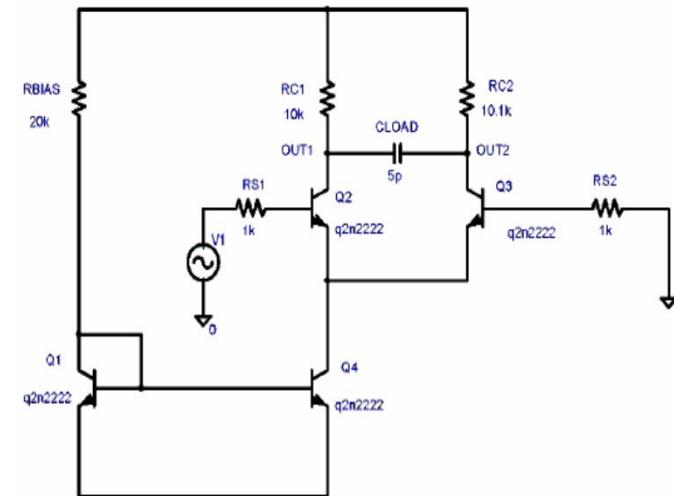
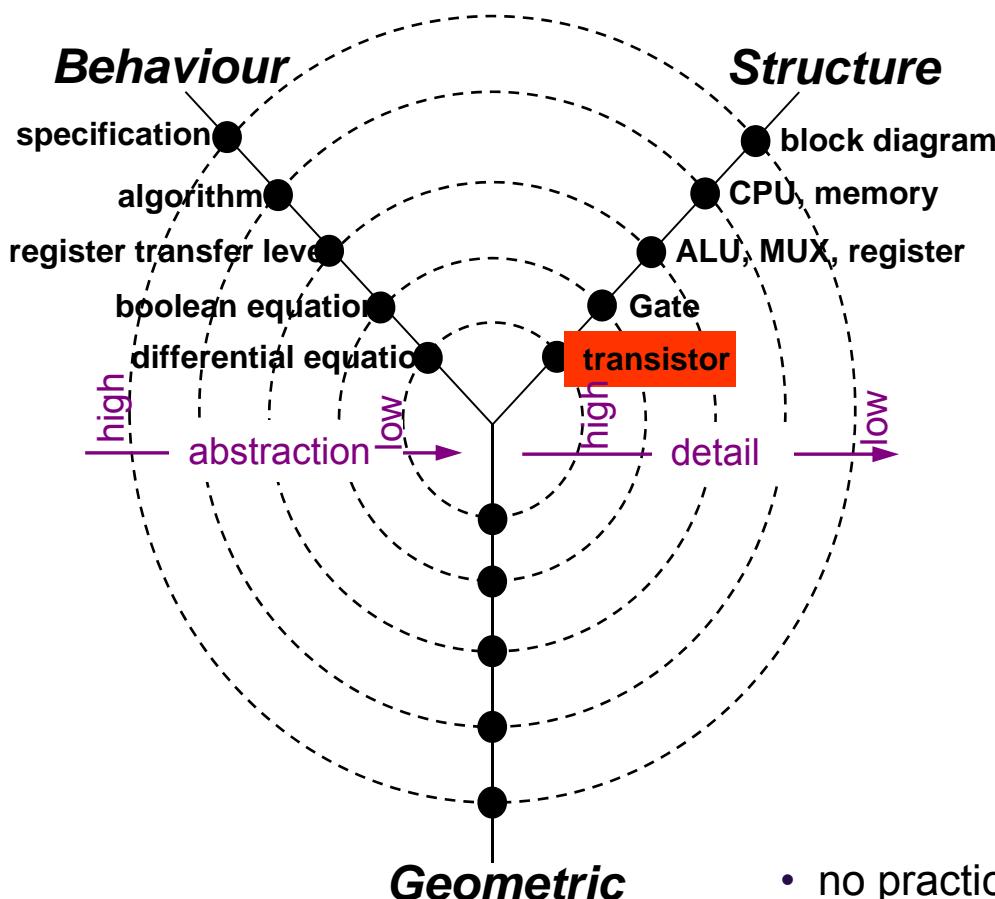
# Structure: Gate



- comparable level of detail as for boolean equation
- high effort for designer, but still practical for small systems
- very good control of the used resources

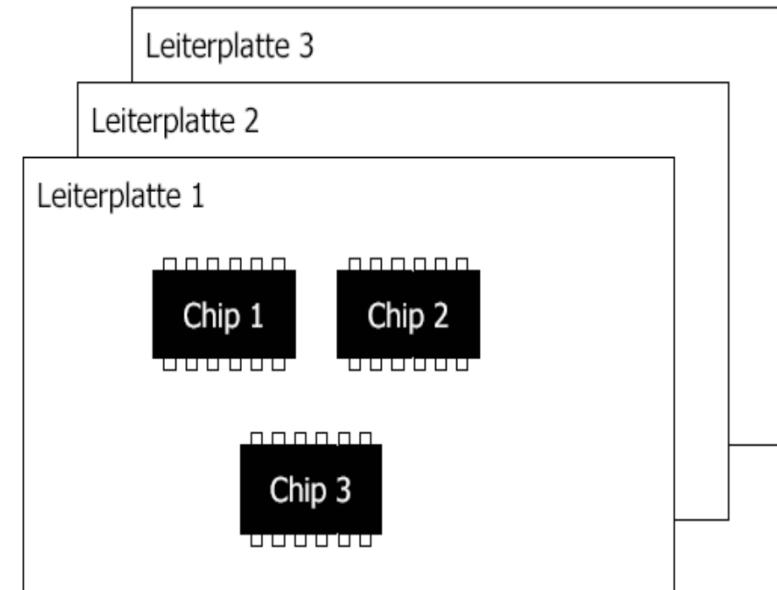
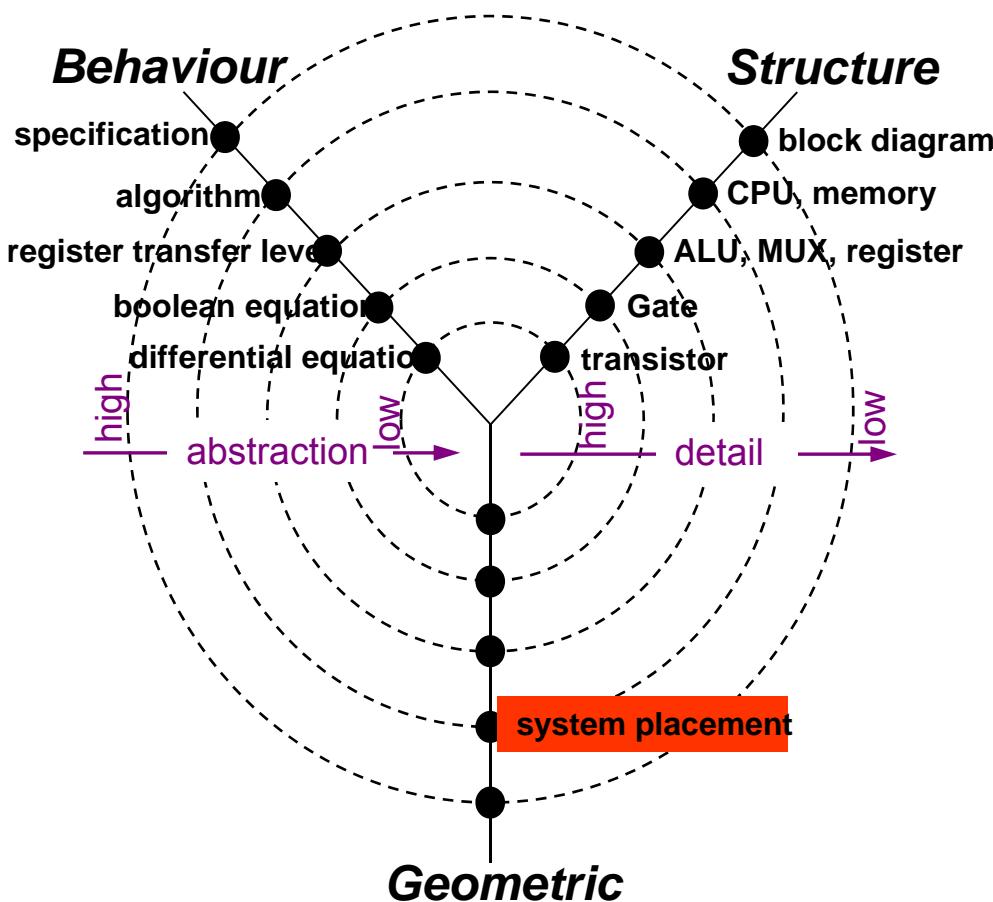


# Structure: Transistor



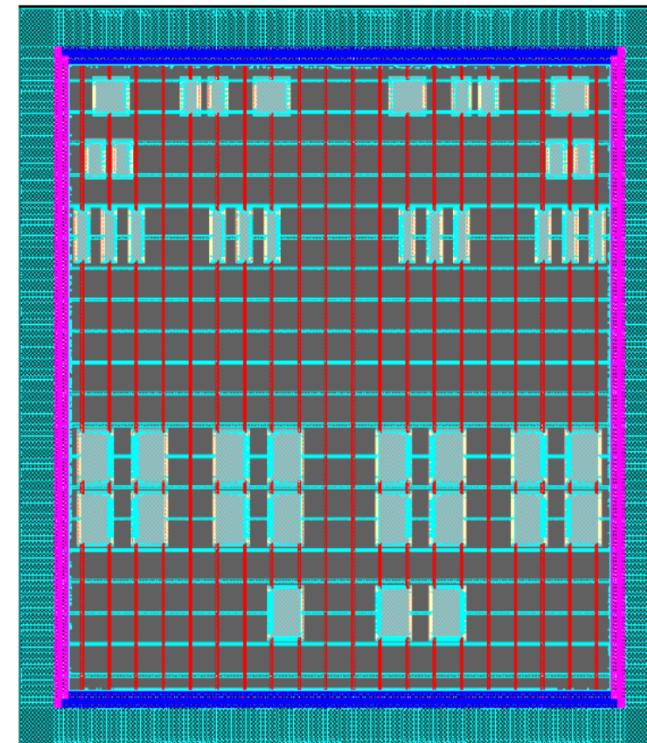
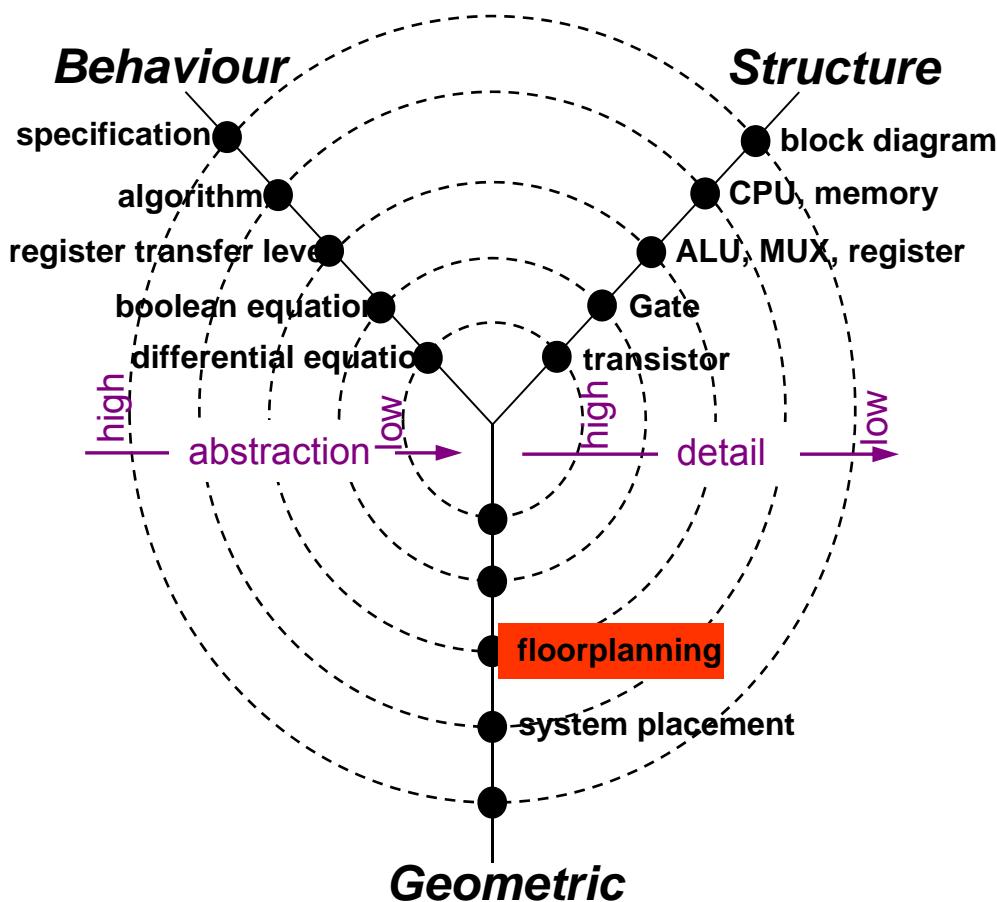
- no practical use for digital system design
- exception: development of standard cell libraries
- Important and used for analog circuit development

# Geometric: System Placement



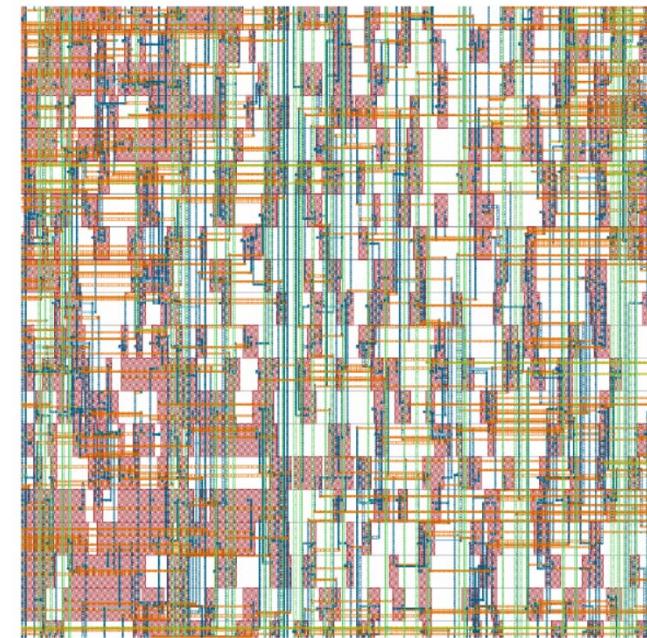
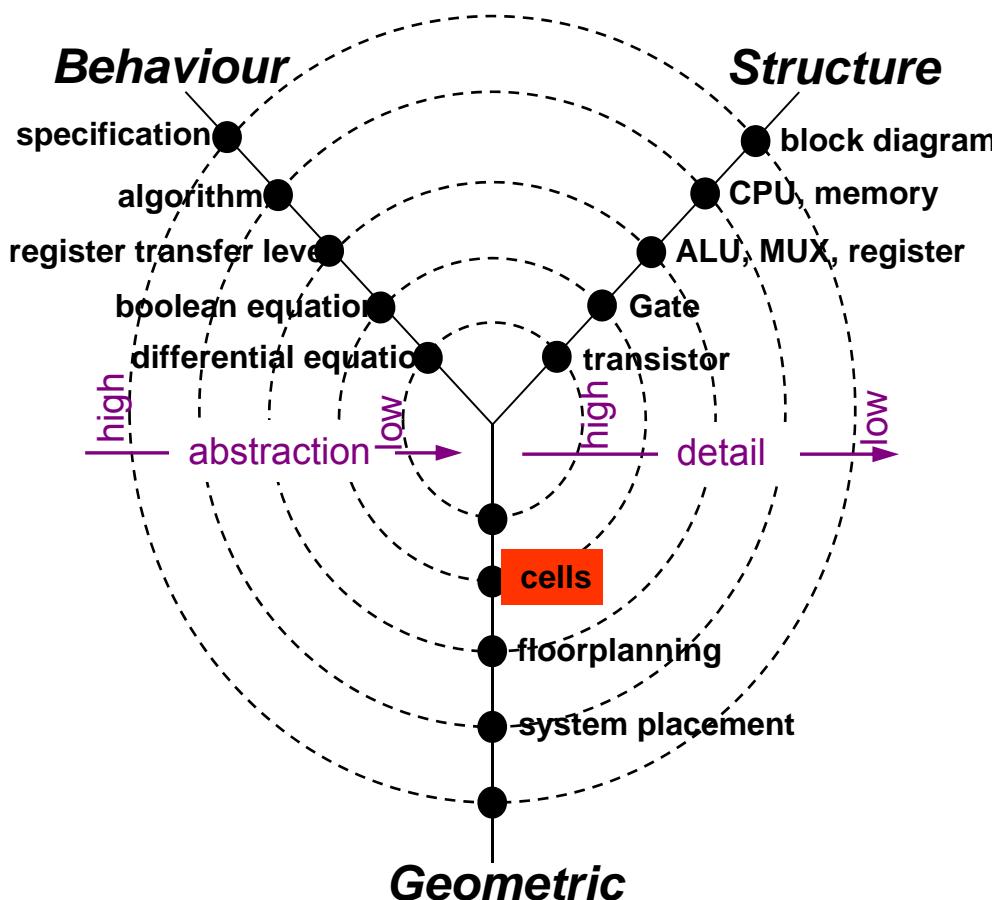
- breakdown the system on different chips or boards

# Geometric: Floorplanning



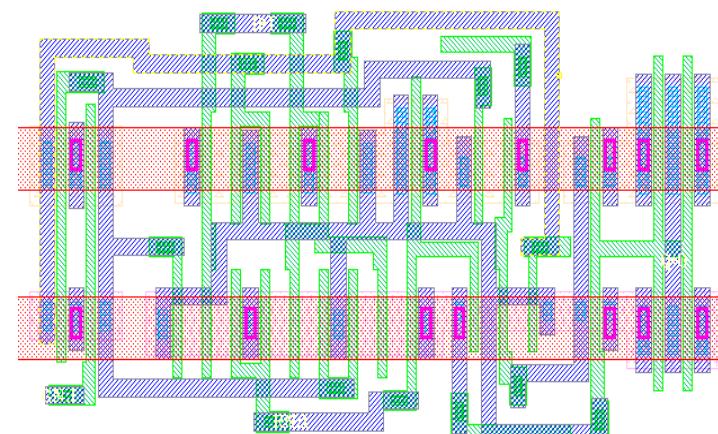
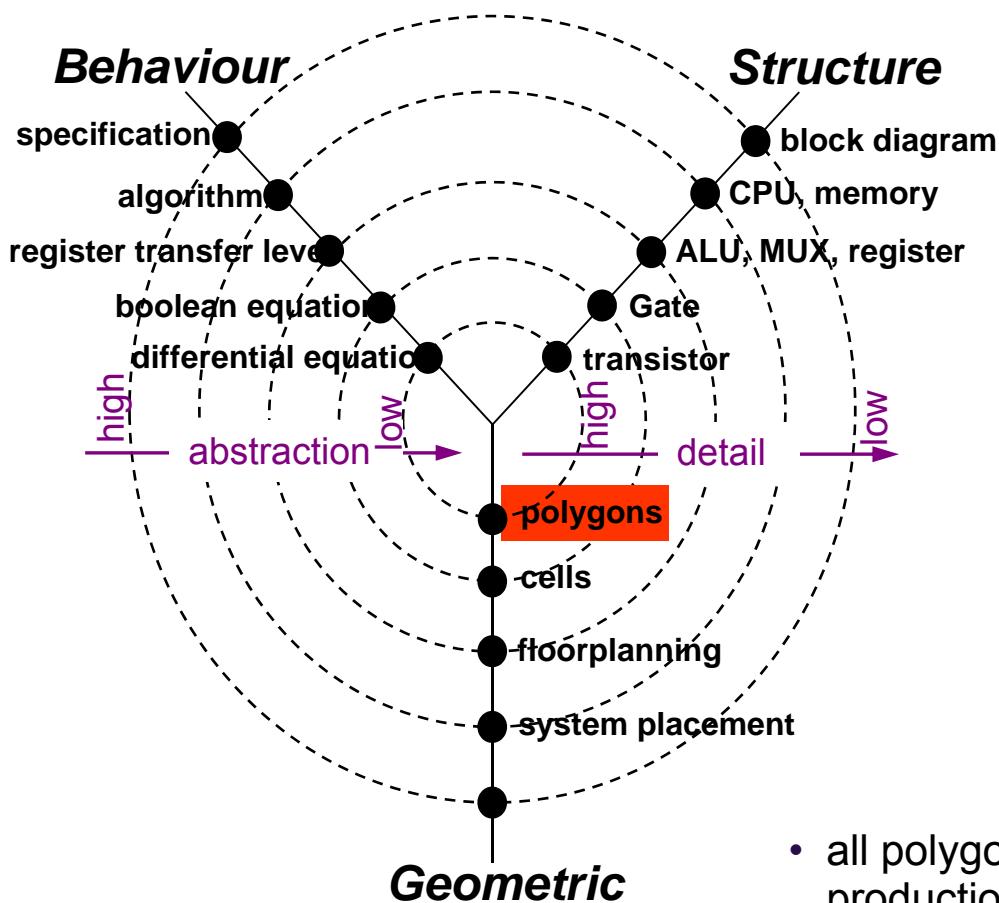
- some sub modules can be places on fix positions like Pads, array, macros ...

# Geometric: Cells



- place and route of cells and macros

# Geometric: Polygons



- all polygons represent the layout for mask production
- layout describes the chip completely
- level of polygon is the development target!

*Motivation*

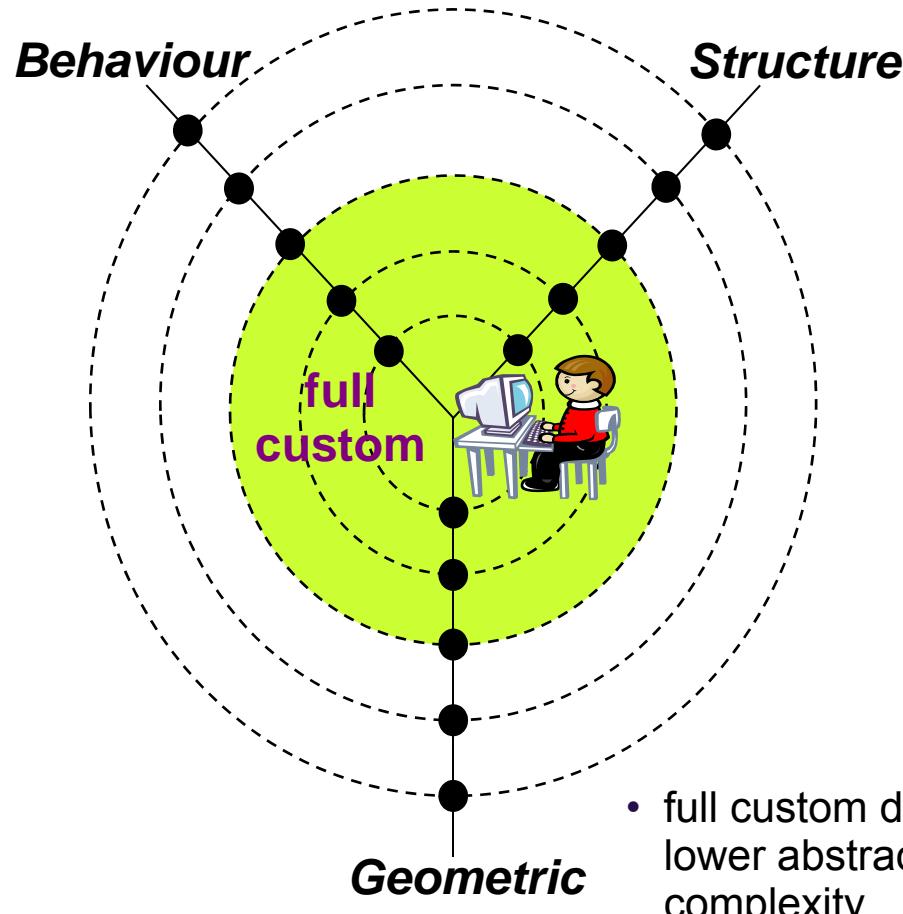
*Gajski Y-diagram*

*Semi vs. full custom*

*Semi custom flow*

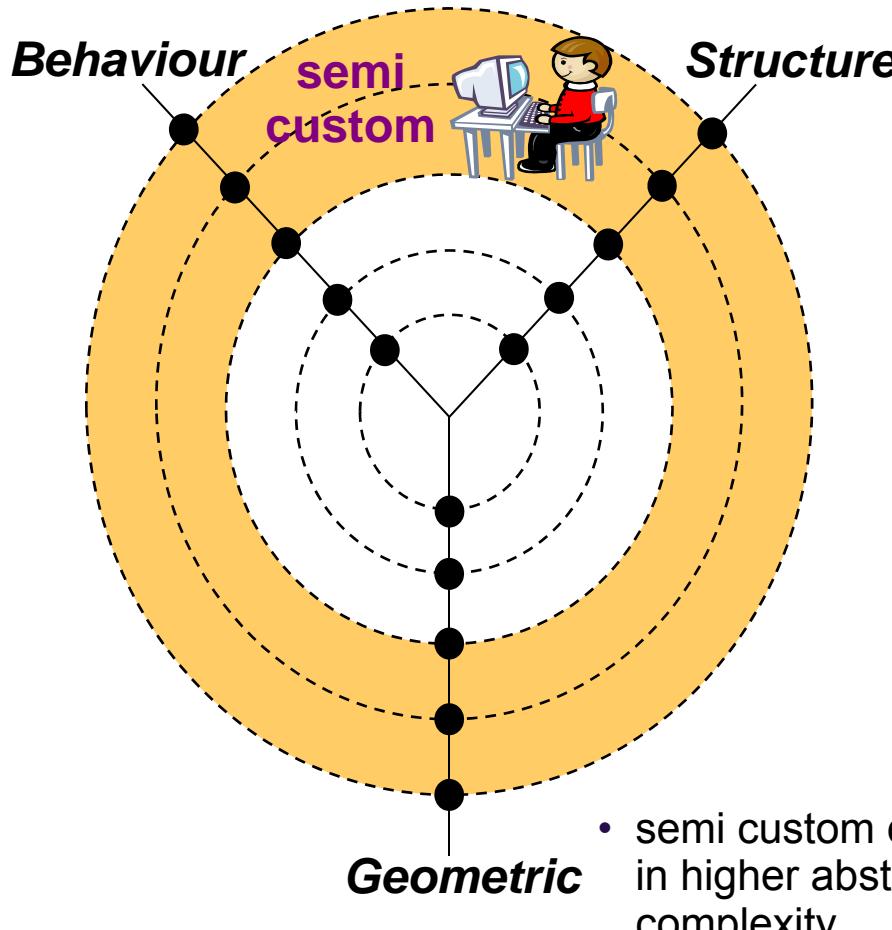
*Implementation flow details*

# full custom designers are tweaking transistors

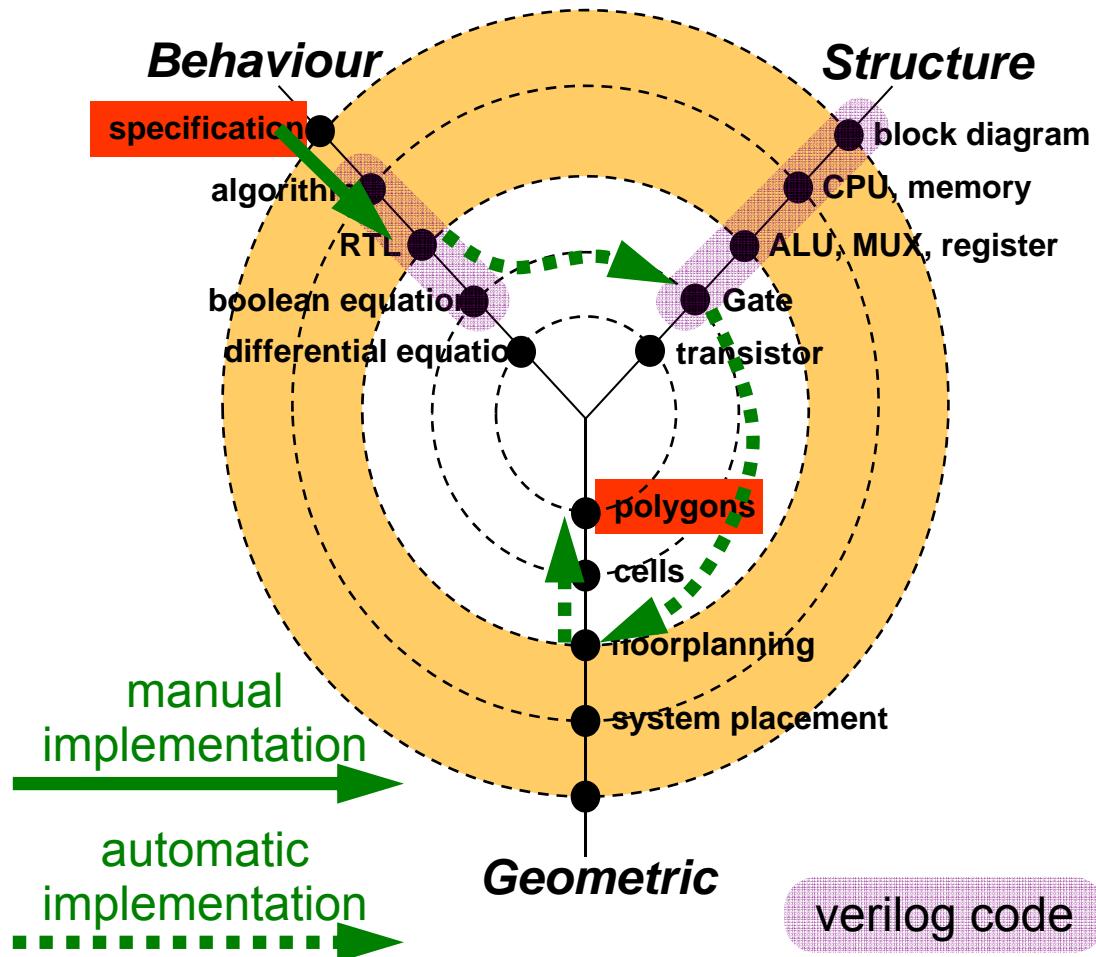


- full custom designers spend their main efforts in lower abstract levels and have to handle higher complexity

# Semi custom designers are tweaking tools

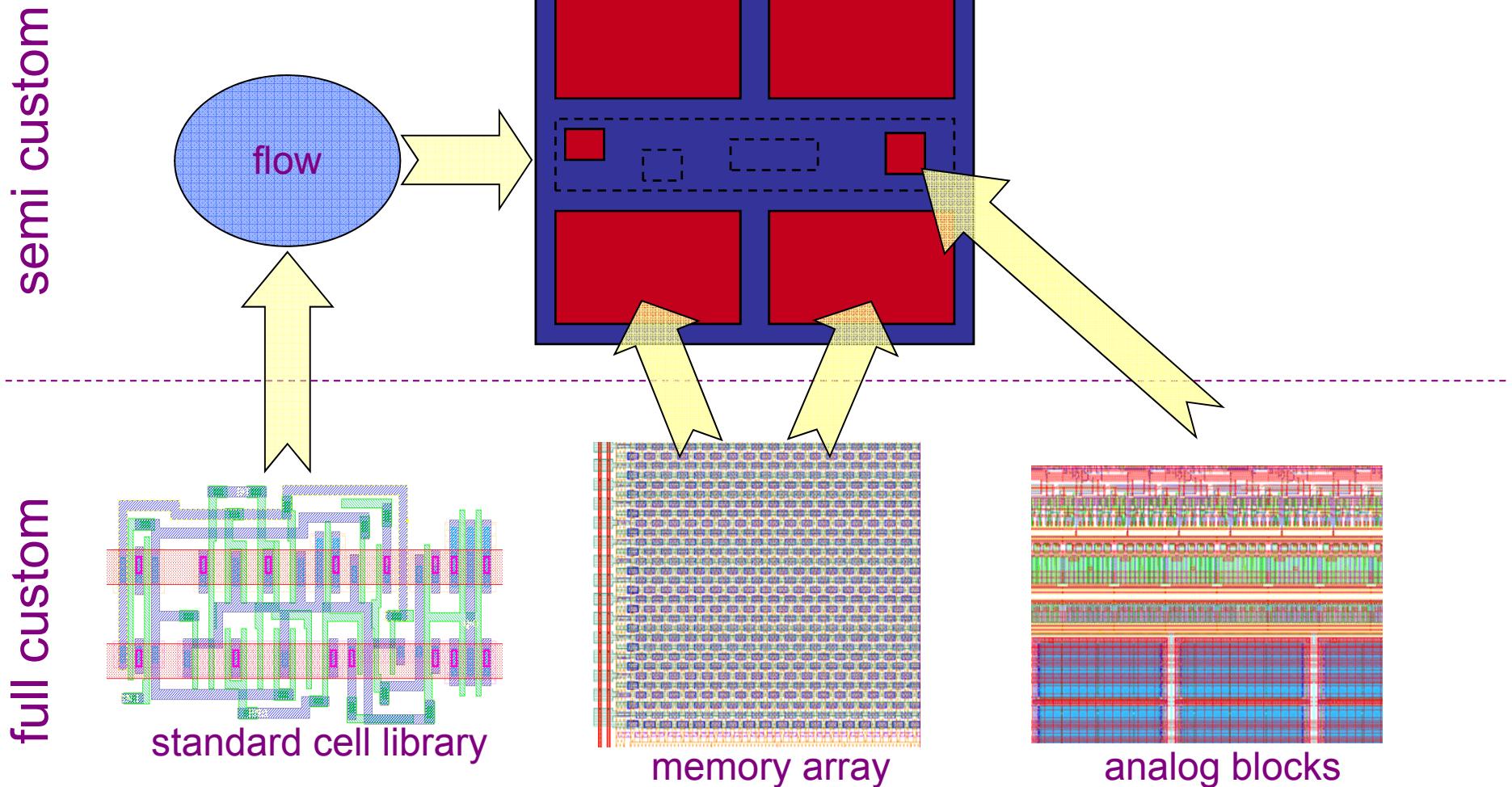


# Semi Custom Design in Y-Diagram



- with HDLs behavioural and structural descriptions are possible, no geometrical description
- Front-end design flow: from specification to gate level
- Back-end design flow: from gate to polygon level
- Synthesis: RTL code transformation into gate level netlist
- Design has to support the automatic tools little by little with further information

# Fields of application



*Motivation*

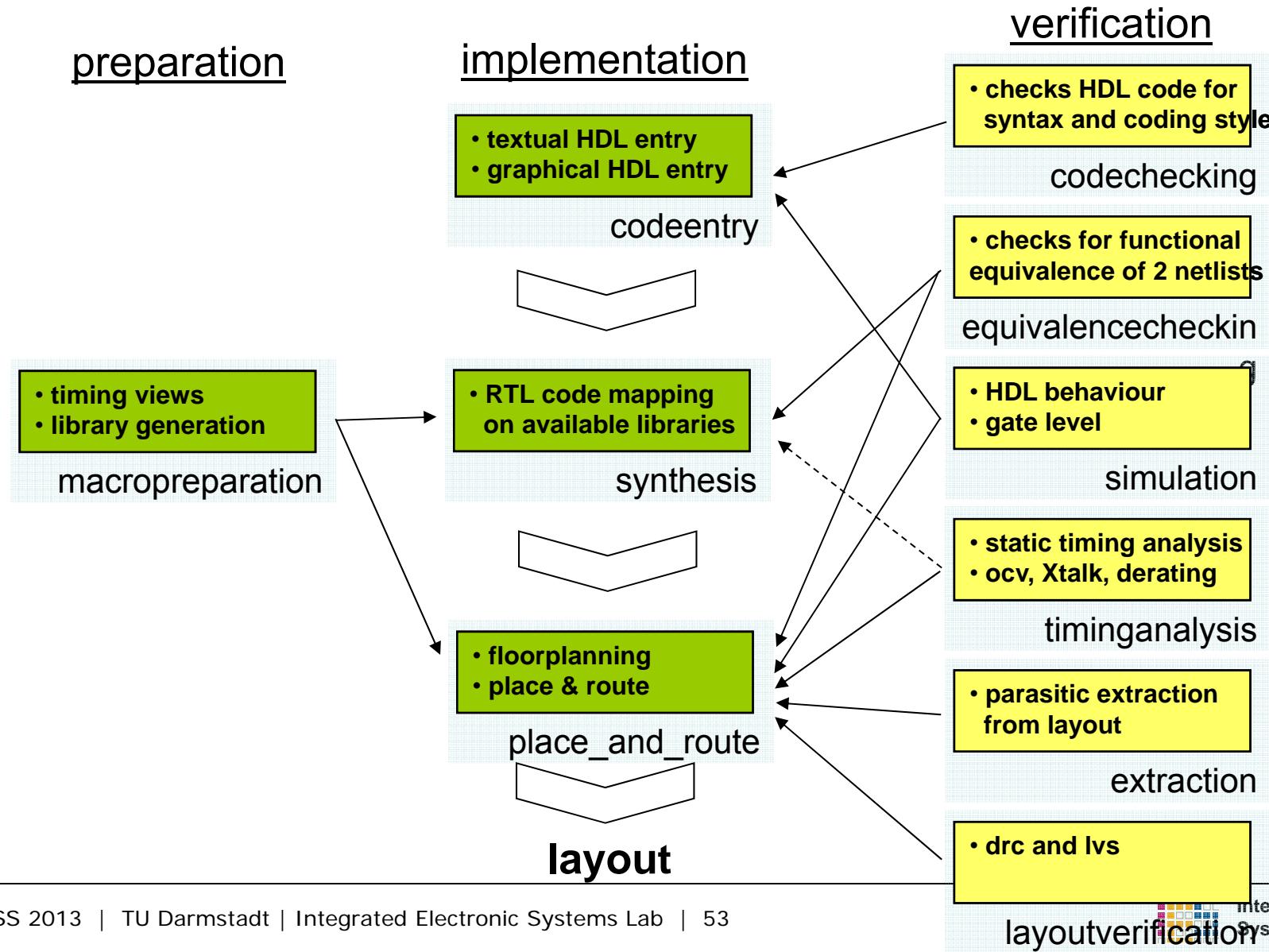
*Gajski Y-diagram*

*Semi vs. full custom*

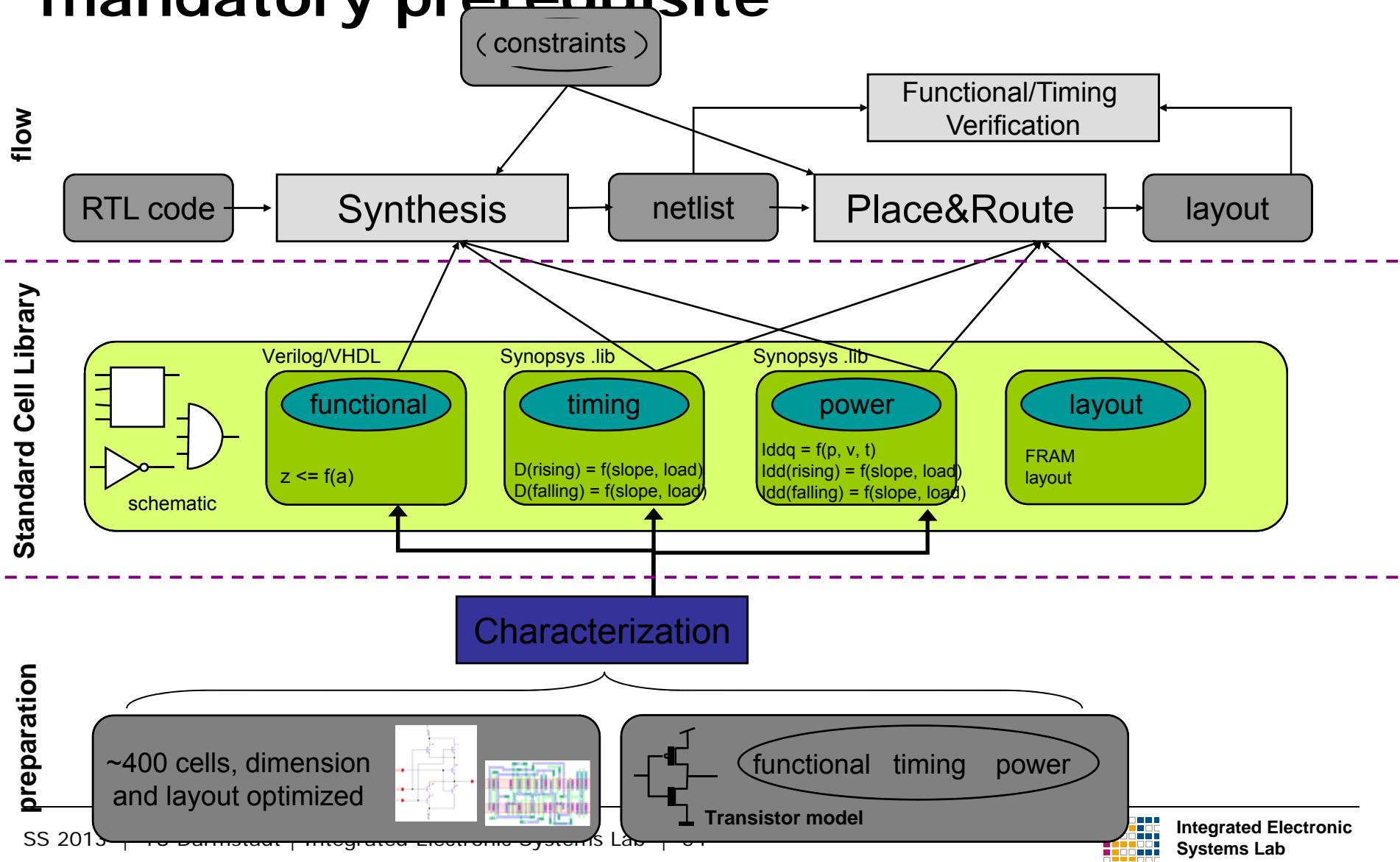
*Semi custom flow*

*Implementation flow details*

# Semi Custom flow



# Standard cell library: mandatory prerequisite



*Motivation*

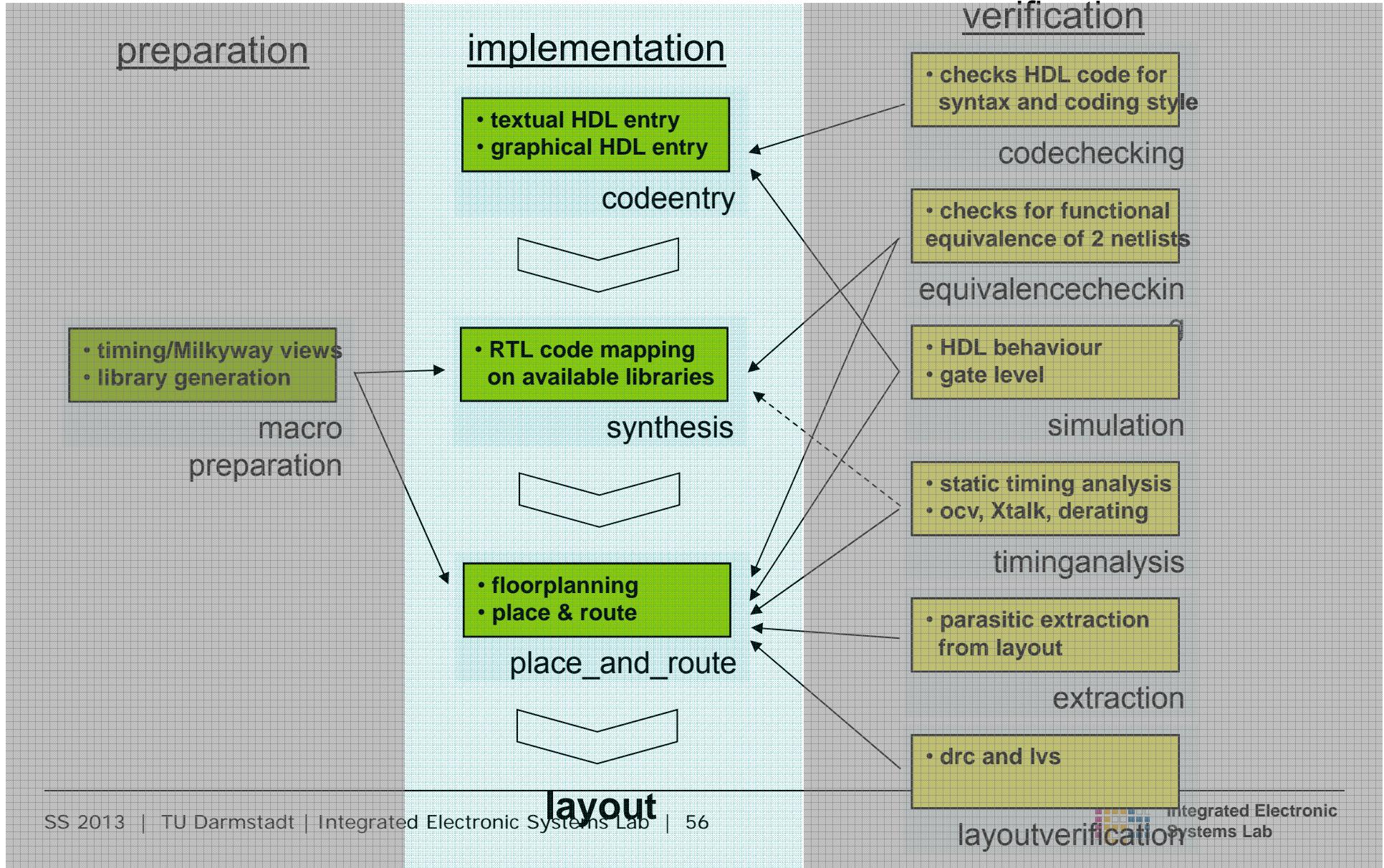
*Gajski Y-diagram*

*Semi vs. full custom*

*Semi custom flow*

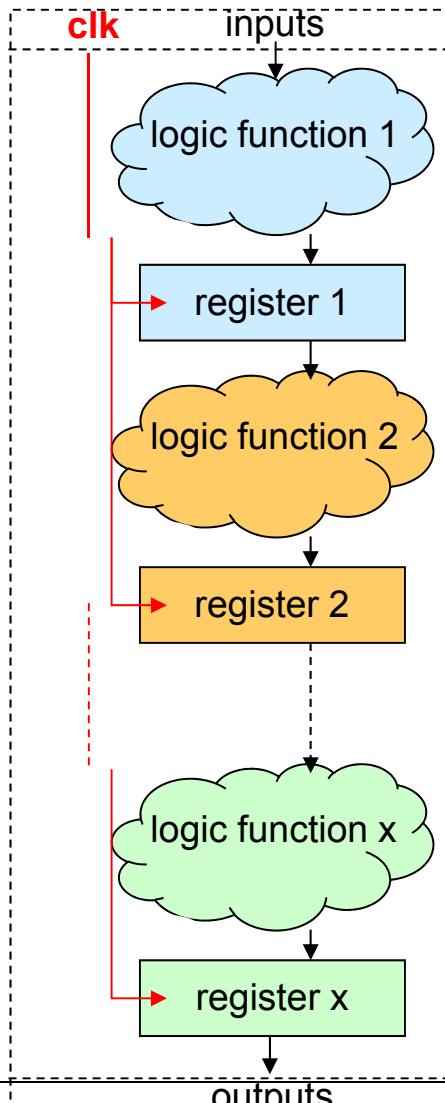
*Implementation flow details*

# Semi Custom flow



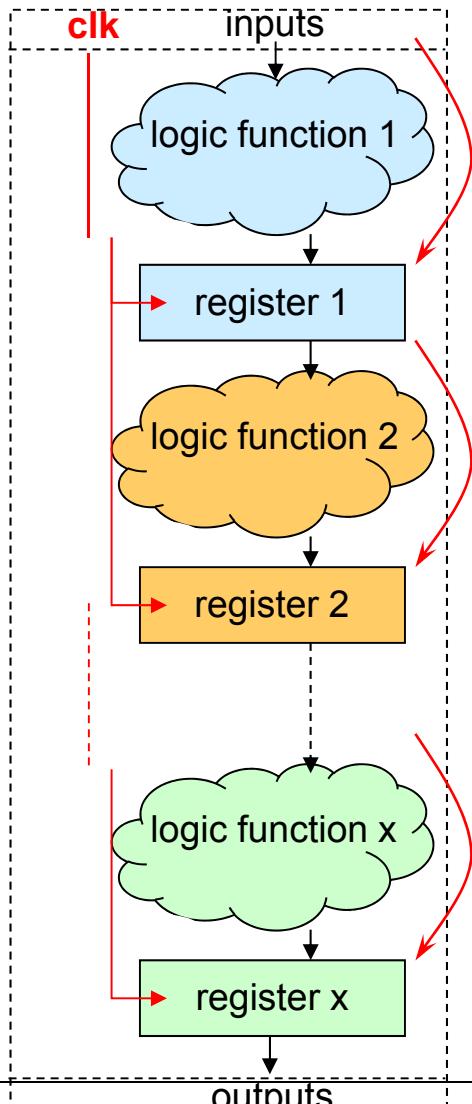
# RTL code and synchronous design

(Register Transfer Level)



# RTL code and synchronous design

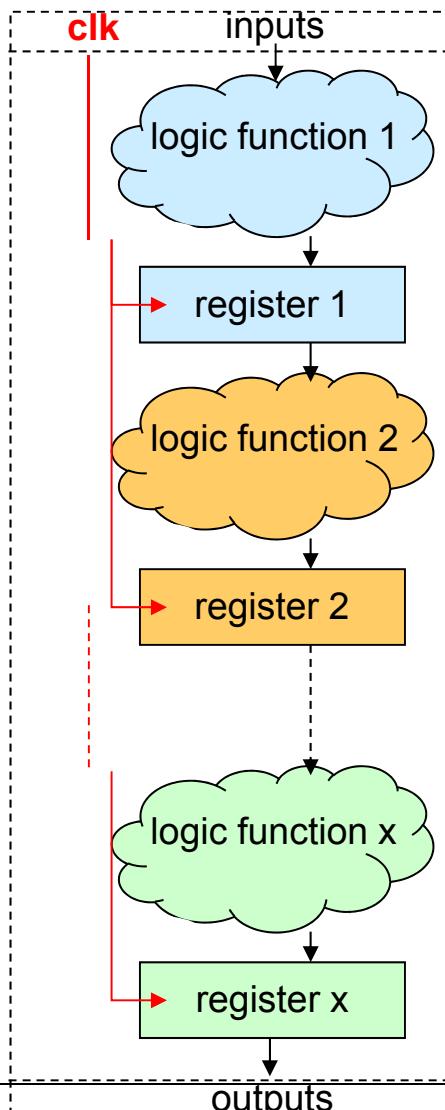
(Register Transfer Level)



synchronous (concurrent)  
data propagate  
from each register  
to the next,  
trigger by the clock signal

# RTL code and synchronous design

(Register Transfer Level)



```
module (...)  
  inputs clk, ...  
  outputs ...  
  always @(posedge clk)  
    begin  
      function 1  
    end  
  always @(posedge clk)  
    begin  
      function 2  
    end  
  .  
  .  
  always @(posedge clk)  
    begin  
      function x  
    end  
end module
```

# Synthesis

## RTL description

```
module multiplier (clk, resetn, a, b,
z);
  input clk, resetn;
  input [3:0] a, b;
  output reg [7:0] z;

  always @ (posedge clk or negedge
resetn)
  begin
    if (~resetn)
      z <= 8'b00000000;
    else
      z <= a * b;
  end
endmodule
```

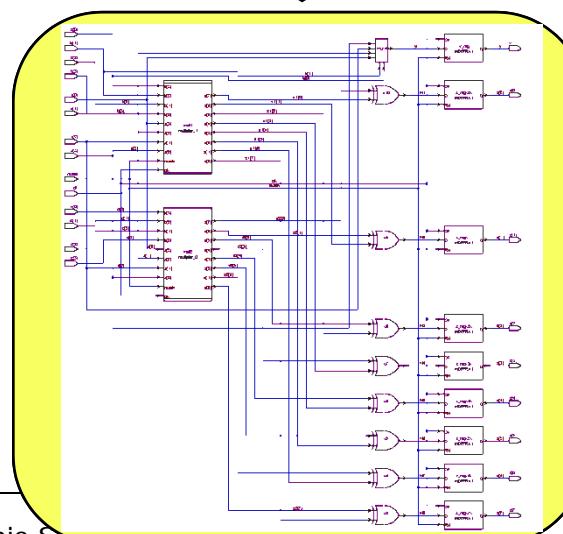
## Library

- Function
- Timing
- Power

## Synthesis

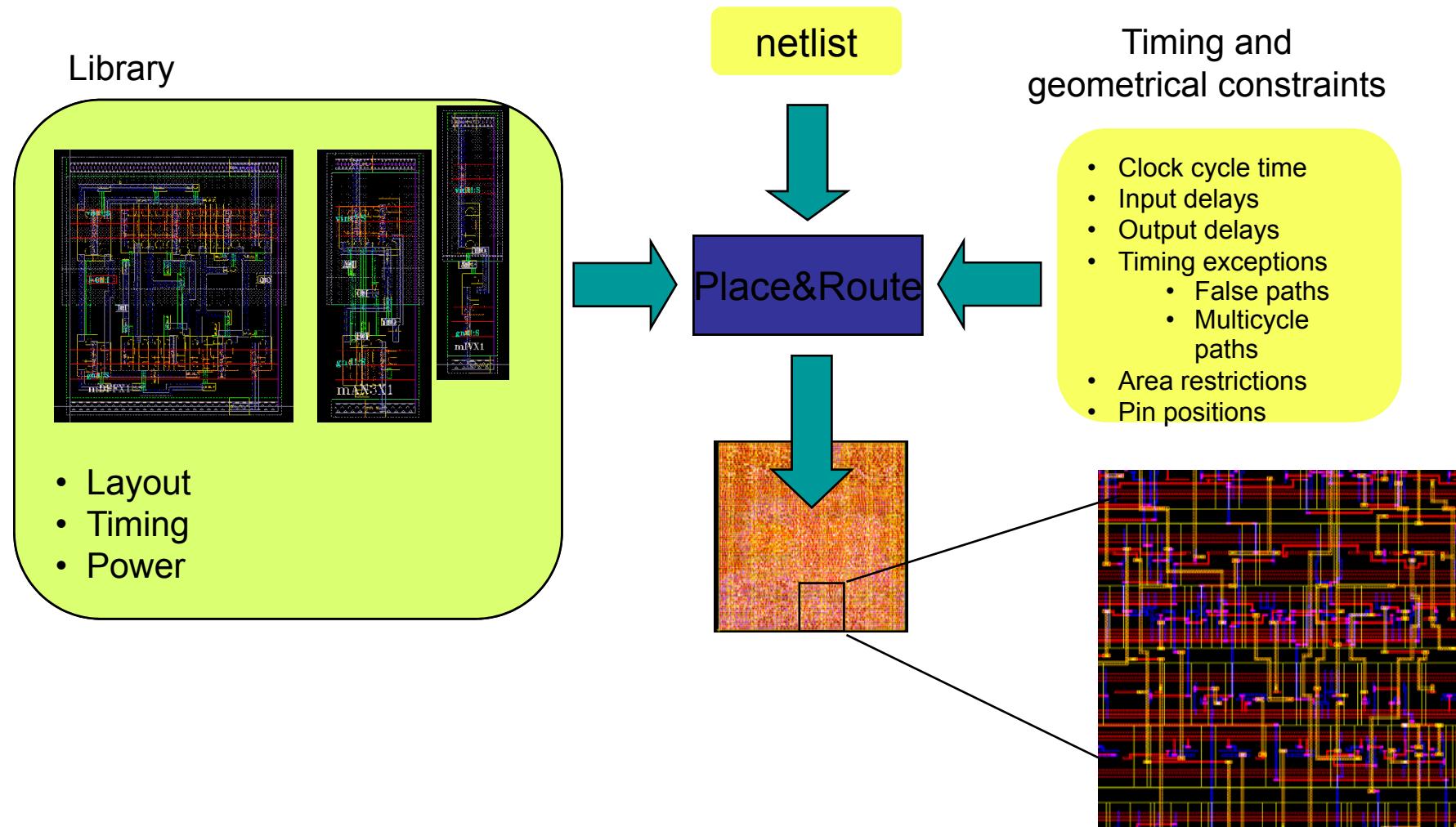
## Timing constraints

- Clock cycle time
- Input delays
- Output delays
- Timing exceptions
  - False paths
  - Multicycle paths

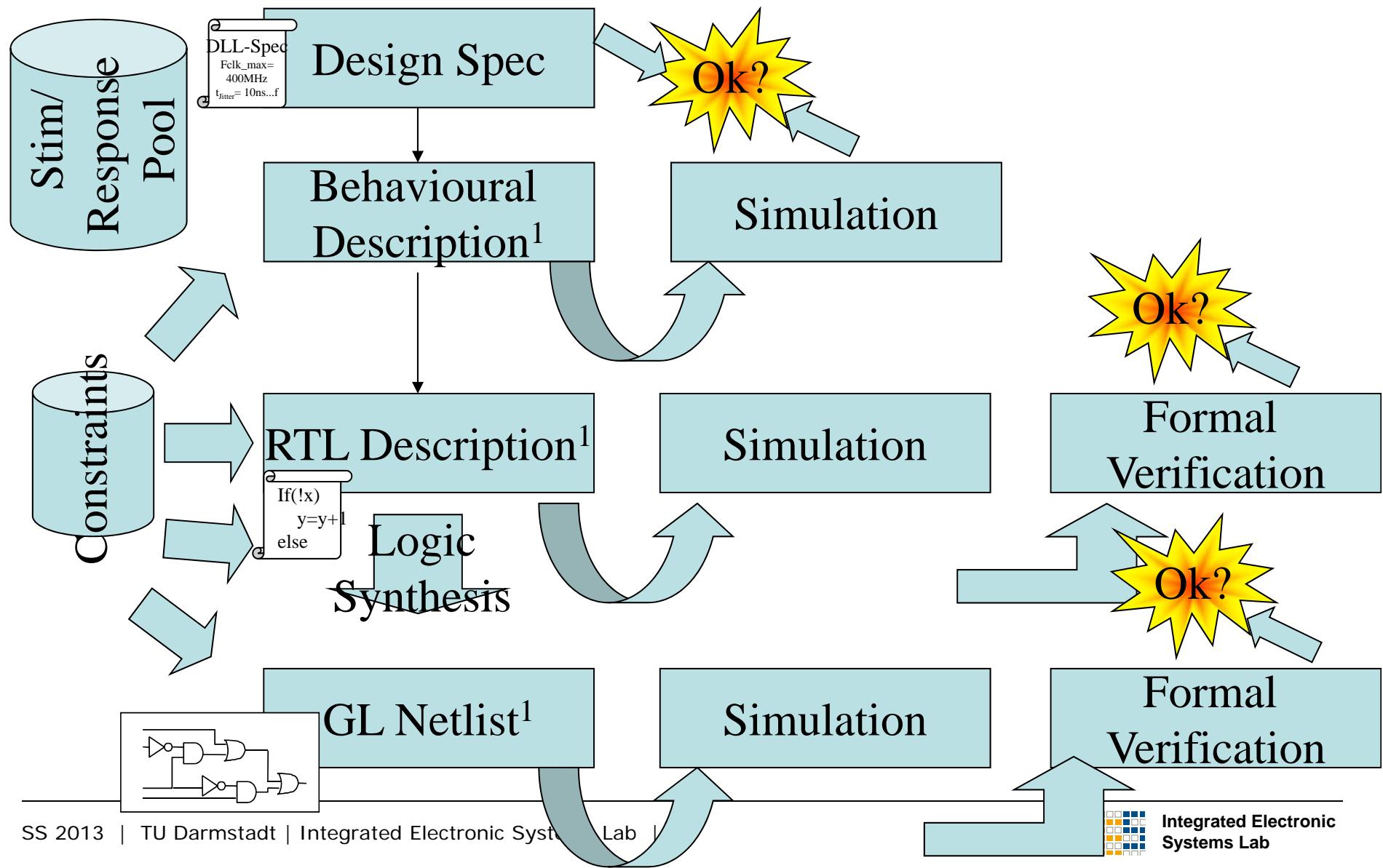


+ netlist

# Place & Route



# The digital design and verification flow



# The digital design and verification flow

- Design Spec: mostly given by Technical Marketing, or simply using the properties of the previous product. Ideally: System-, Subsystem- and Blockspecification
- Stimulus/Response Pool: contains all known input stimulus and response patterns. Determines the coverage of the simulation
- Constraints: This is an abstraction of the data/voltage sources to your system/block (rise/fall times, clock frequency, timing), and of the capacitive loads of the outputs
- Testbench: Environment of your system/block model (e.g. providing pattern, checking/comparing for correctness, ...)

Rule of thumb **1**: HDL-Coding of system/block takes only  
**20%** of the overall effort in previous page

Rule of thumb **2**: Writing the specs & testbench, deriving constraints and collecting patterns takes **weeks** before writing the first line of HDL-code!

# How to include timing

- Some design houses accept GL Netlist + constraints and produce a chip for you
- For critical timing, logic only information is not sufficient. How to overcome missing physical (location) information?
- Traditional approach (from 1985 on, until 2004): use statistical information of loading and fanout of circuits for modeling the interconnect delay
- Modern approach (standard from 2004 on): Synthesis, Floorplanning and Routing cannot be separated from each other

# The old approach: Wireload Models

- Part of the technology library
- ```
wire_load("4000") { capacitance : 0.000240 ;
  resistance : 0.321e-3;
  area : 0.01;
  slope : 85.05 ;
  fanout_length(1,12.978);
  fanout_length(2,25.83);
  fanout_length(3,41.202);
  fanout_length(4,73.71);
  fanout_length(5,90.877);
  fanout_length(6,112.203);
  fanout_length(7,121.275);
  fanout_length(8,194.04);
  fanout_length(9,199.71);
  fanout_length(10,248.85);
  fanout_length(11,316.89);
  fanout_length(12,355.32);
  fanout_length(13,360.36);
  fanout_length(14,376.11);
  fanout_length(15,396.90);
  fanout_length(16,413.28);
  fanout_length(17,446.04);
  fanout_length(18,512.19);
  fanout_length(19,584.01);
  fanout_length(20,669.06);
}
```

# The old approach: Wireload Models

```
dc_shell-xg-t> read_file my_lib.db
```

```
Example Wire Load Models Report
```

```
*****  
Report : library  
Library: my_lib  
Version: Y-2006.06  
Date : Mon May 1 10:56:49 2006  
*****
```

```
...  
Wire Loading Model:
```

```
Name : 05x05  
Location : my_lib  
Resistance : 0  
Capacitance : 1  
Area : 0  
Slope : 0.186  
Fanout Length Points Average Cap Std Deviation
```

```
-----  
1 0.39
```

```
Name : 10x10
```

```
Location : my_lib  
Resistance : 0  
Capacitance : 1  
Area : 0  
Slope : 0.311  
Fanout Length Points Average Cap Std Deviation
```

```
-----  
1 0.53
```

```
...
```

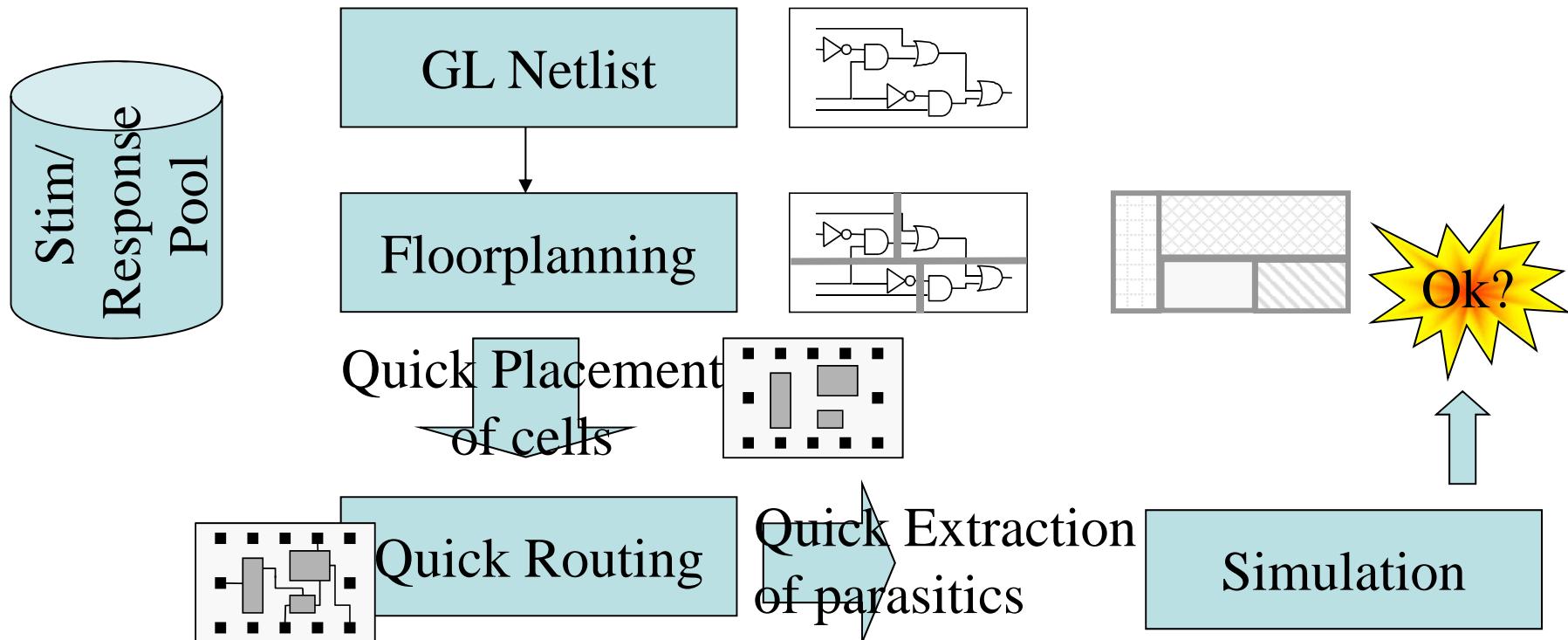
## Specifying Wire Load Models and Modes

The `default_wire_load` library attribute identifies the default wire load model for a technology library. To change the wire load model or mode specified in a technology library, use the `set_wire_load_model` and `set_wire_load_mode` commands.

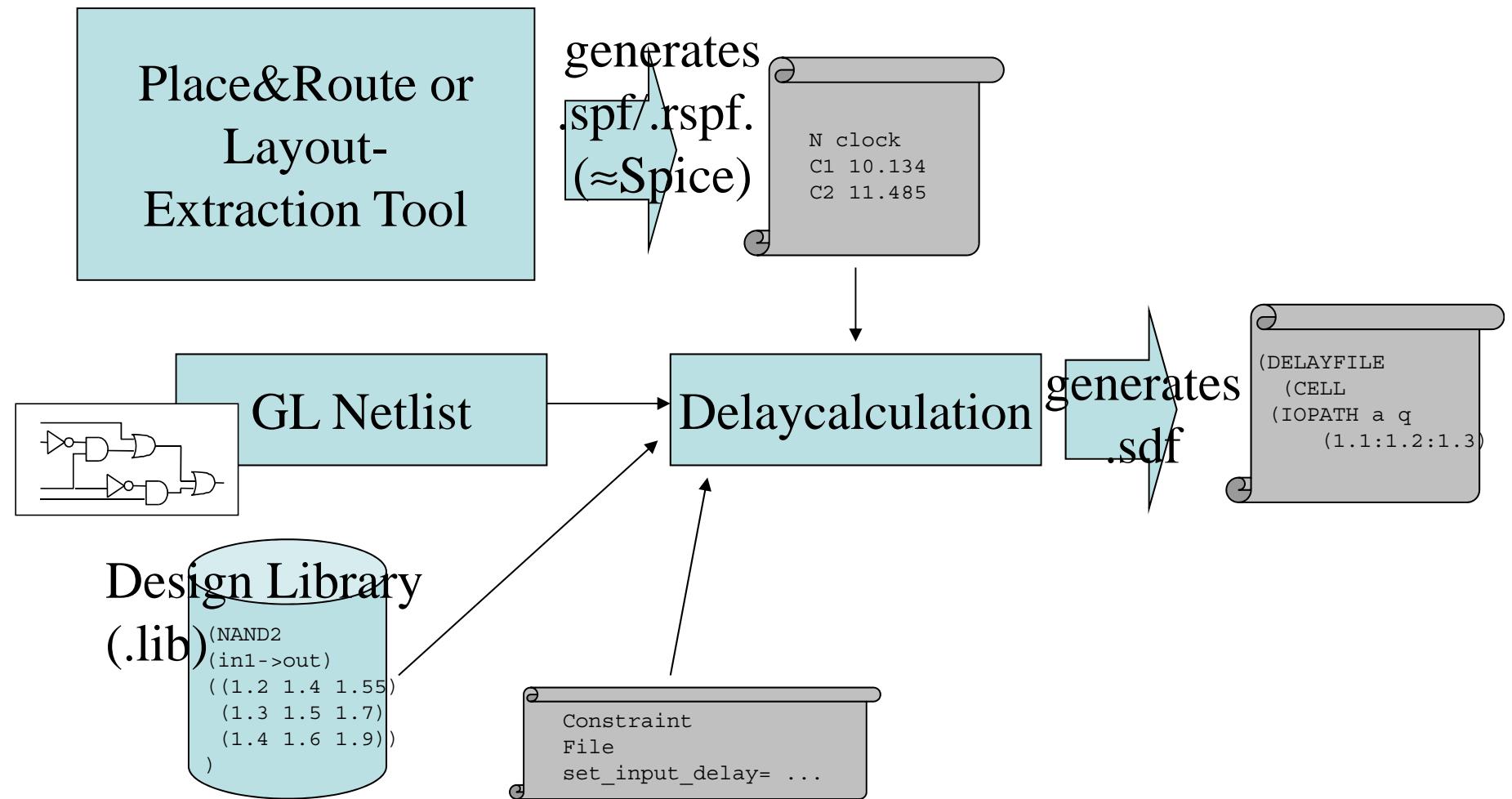
eg:

```
dc_shell-xg-t> set_wire_load_model "10x10"
```

# The new approach



# Calculation of Timing



# Syntax and Lexical Conventions of Verilog



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Verilog

- The good news: close to C language
- Comments:
  - Single line: //
  - More than one line (block comment): /\* ..... \*/
- Comments cannot be nested
  
- Whitespace is defined as blanks, tabs, newlines and formfeeds
- All whitespaces is ignored, except in strings

```
module shiftregister;  
    reg shiftin;    // register to drive value into circuit
```

# Numbers

- Constant numbers can be specified as decimal, hexadecimal, octal or binary. They may optionally start with a + or – sign in front. In general, there are two ways to specify a constant number:
  - 1) Unsized decimal number. Simply use the digits from 0..9. Even if you do not specify the size, Verilog will calculate the size for use in an expression. Usually the size chosen is then the one of the other operand. Otherwise, Verilog calculates the appropriate size (see next page):

# Numbers

| Expression              | Bit Length                      | Comment                                                                                                                   |
|-------------------------|---------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Unsized constant number | Same as integer<br>(usually 32) |                                                                                                                           |
| Sized constant number   | As given                        |                                                                                                                           |
| $i \text{ OP } j$       | $\max(L(i), L(j))$              | OP is $+, -, /, *, \%, \&,  ,$<br>$\wedge, \sim \wedge$                                                                   |
| $\text{OP } j$          | $L(j)$                          | OP is $+, -, \sim$                                                                                                        |
| $i \text{ OP } j$       | 1 bit                           | OP is $==, !=, \text{!}==, \text{!}=, \&\&,   , <, >, <=, >=, \&, \sim \&,  , \sim  , \wedge \sim, \sim \wedge, \text{!}$ |
| $i \text{ OP } j$       | $L(i)$                          | OP is $>>, <<, **, <<<, >>>$                                                                                              |
| $i ? j : k$             | $\max(L(i), L(j))$              |                                                                                                                           |
| $\{i, \dots, j\}$       | $L(i) + \dots + L(j)$           |                                                                                                                           |
| $\{i \{j, \dots, k\}\}$ | $i * (L(j) + \dots + L(k))$     |                                                                                                                           |

2) The second form specifies the size of the constant. Examples:

- 2 b 00 (a two bit number, both bits are zero)
- 8 h f1 (an eight bit number: 11110001)
- In general: aa...a sf nn..n , whereby
  - aa...a is the size in bits of the constant. The size is specified as an unsigned decimal number
  - `sf is the base format. The f is either d (decimal), h (hexadecimal), o (octal) or b (binary). s optionally specifies that the value is to be considered a signed number (e.g. -6 `d 5 is a six-bit constant with the value of -5 (i.e. 111011)). No whitespace is allowed between ` and f.
  - nn..n is the value of the constant specified in the given base with allowable digits. For the hexadecimal base, the letters a through f may also be capitalized. Letters are case insensitive. If s was specified in the base format, then the constant will be treated as a signed value in all calculations.

## Some examples:

- 12 b 1x1x01100zx0 (a 12 bit number)
- 12 b 1x1x\_0110\_0zx0 (same as before, but better readable)
- 12 b \_1x1x\_0110\_0zx0 (illegal, \_ not allowed as first character)
- 345 (a decimal number)
- 3f5 (illegal, a hexadecimal number must be specified with h)
- 12 h 3f5 (now this is a legal 12 bit number)
- 11 d 18 (a eleven bit constant with the value 18)
- -6 d 5 (a six-bit constant with the value of -5 (i.e. 111011))
- 6 d -5 (illegal)
- 3 d 7 (a three bit constant with the value 7 (111))
- 3 sd 7 (a three bit signed constant with the value -1 (111))
- 4 sh F (a four bit signed constant with the value -1 (1111))

# Strings

A string is a sequence of characters enclosed by double quotes. It must be contained on a single line. Special characters may be specified in a string using the „\“ escape character:

- \n : The newline character (similar to the „return“ key)
- \t : The tab character. Same as the „tab“ key
- \\ : The \ character itself
- \\* : The \* character itself
- \ddd : An ASCII-character specified in one to three octal digits

# Identifiers, System Names & Keywords

Identifiers are names that are given to elements (modules, registers, ports, wires, instances, begin-end blocks). An identifier is any sequence of letters, digits and underscore symbols, except that there is a limitation to max 1024 characters, and the first character must not be a number.

Upper and lower case characters are considered different!

System tasks and functions start with a dollar (\$) symbol. Since system tasks and functions are simulator and OS dependent, we always recommend to have a look in the simulator manual for the correct syntax. In the following slides we would like to give some insight in useful system functions.

# System Tasks and Functions

- 1) Display and Write Tasks: \$display and \$write are almost the same, except that \$display always prints out a newline character at the end.

Example:

```
$display(„Hello world!\n”) ; // prints out „Hello world!“  
// with 2 newline characters  
parameter plz=64283;  
$display(„The zip code of Merckstrasse 25 is %d“,plz);
```

# System Tasks and Functions

The following formats are allowed in the format control specification:

- h, H: display in hexadecimal
- d, D: display in decimal
- o, O: display in octal
- b, B: display in binary
- c, C: display ASCII character
- v, V: display net signal strength
- m, M: display hierarchical name
- s, S: display string

For h, d and o, you might suppress leading zeros with the „%0d“ command

# System Tasks and Functions

2) Continuous monitoring: Similar to the \$display function, \$monitor is useful to print out information whenever there is a change in one or more specified values. The printout is always done at the end of the current time.

Note that only one \$monitor display list may be active one at a time.

3) Strobed monitoring: Similar to the \$display function, \$strobe is useful to print out information just before the simulation time is about to advance. In this way, \$strobe insures that all of the changes that were made at the current simulation time have been made, and thus will be printed.

# System Tasks and Functions

- 4) File output: The \$display, \$write, \$monitor and \$strobe functions have closely related functions that write to a file. Like in the C-language, you first have to have a descriptor (32 bit value returning from an \$fopen(„filename“) command).

The syntax is then:

```
$fdisplay(descriptor, parameters as for $display)  
$fwrite(descriptor, parameters as for $write)  
$fmonitor(descriptor, parameters as for $monitor)  
$fstrobe(descriptor, parameters as for $strobe)
```

After the job is done, you should close the file using

```
$fclose(descriptor)
```

# System Tasks and Functions

- 5) Simulation time: you might find it useful to print out the simulation time using \$time (64 bit value) or \$stime (32 bit value).

Example:

```
$monitor($time,,,"regA = ", regA);
```

# System Tasks and Functions

- 6) Stopping and finishing simulation. \$stop and \$finish tasks stop simulation. \$stop returns to the simulators command interpreter, while \$finish returns to the OS.

You might provide optionally one parameter to \$stop or \$finish, e.g. 0 prints nothing, 1 gives simulation time and location (this is the default, same as not giving a parameter to the two functions, 2 same as 1 plus a few lines of run statistics).

Example:

```
$stop(2); // prints out simulation time and location,  
plus a few lines of run statistics
```

# System Tasks and Functions

- 7) Random numbers: The \$random system function provides a random number mechanism, return a random number each time the function is called.

Example:

```
parameter seed = 42;  
reg [31:0] vector;  
always @(posedge clock)  
    vector = $random(seed);
```

# System Tasks and Functions

- 8) Reading data from disk: With \$readmemb and \$readmemh you may load binary or hexadecimal numbers stored in disk files to Verilog memories.

Usage:

```
$readmemb(“filename”, <memname>, <<start_addr>
           <, <finish_addr>>?>?) ;
```

<start\_addr> optionally specifies the starting address of the data. If none is given, the left-hand address given in the memory declaration is used.

<finish\_addr> is then the last address.

Verilog has also access to several C-like I/O system function calls, such as \$fgetc, \$fungetc, \$fgets, \$fscanf, \$ftell, \$fseek, \$rewind, \$ferror, \$fflush.

# Arithmetic Operators

- Arithmetic Operators perform arithmetic operations (what else..)
- + and – can be either used as unary or binary operators

| Operator Symbol | Name           | Definition                        | Comment                                                                                                                                       |
|-----------------|----------------|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| +               | Addition       | Sums two operands                 | Register and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown, the result will be unknown |
| -               | Unary minus    | Changes the sign of its operand   | See comment on top                                                                                                                            |
| -               | Subtraction    | Finds difference between operands | See comment on top                                                                                                                            |
| *               | Multiplication | Multiply two operands             | See comment on top                                                                                                                            |
| /               | Division       | Divide two operands               | See comment on top. Additionally, a division by zero produces an „x“                                                                          |
| %               | Modulus        | Find remainder                    | See comment on top                                                                                                                            |

# Arithmetic Operators (Examples)

```
parameter m = 5;  
reg[3:0] a,b,c,d,count;  
c=a+b;  
d=b-m;  
count=(count+1)%16; // counts 0 through 15
```

# Operators

- Relational operators compare two operands and return a single bit

| Operator Symbol | Name                  | Definition                         | Comment                                                                                                                                       |
|-----------------|-----------------------|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <               | Less than             | Determines relative value          | Register and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown, the result will be unknown |
| <=              | Less than or equal    | Determines relative value          | See comment on top                                                                                                                            |
| >               | Greater than          | Determines relative value          | See comment on top                                                                                                                            |
| >=              | Greater than or equal | Determines relative value          | See comment on top                                                                                                                            |
| ==              | Logical equality      | Compares two values for equality   | See comment on top                                                                                                                            |
| !=              | Logical inequality    | Compares two values for inequality | See comment on top                                                                                                                            |

# Relational Operators (Examples)

```
if (x == y) e = 1;  
else e = 0;
```

```
e = (x == y); // this is equivalent to the line above  
// Compare in 2's complement; a > b  
reg[3:0] a,b;  
if(a[3] == b[3]) a[2:0] > b[2:0];  
else b[3];
```

# Operators

- Bit wise operators do a bit-by-bit comparison between two operands.

| Operator Symbol                   | Name             | Definition                                         | Comment                                                                           |
|-----------------------------------|------------------|----------------------------------------------------|-----------------------------------------------------------------------------------|
| $\sim$                            | Bitwise negation | Complements each bit in the operand                | Each bit of the operand is complemented. The complement of $x$ is $\sim x$        |
| $\&$                              | Bitwise AND      | Produces the bitwise AND of two operands           | See truthtables on following pages                                                |
| $ $                               | Bitwise OR       | Produces the bitwise OR of two operands            | See truthtables on following pages                                                |
| $^$                               | Bitwise XOR      | Produces the bitwise exclusive OR of two operands  | See truthtables on following pages                                                |
| $==$                              | Bitwise equality | Compares two values for equality                   | Bitwise comparison includes $x$ and $z$ values – all bits must match for equality |
| $\sim \wedge$ or<br>$\wedge \sim$ | Equivalence      | Produces the bitwise exclusive NOR of two operands | See truthtables on following pages                                                |

# Bit-wise Operators (Examples)

```
module and2 (a,b,c);  
    input [1:0] a, b;  
    output [1:0] c;  
    assign c = a & b;  
endmodule
```

# Truthtables of Logic Gates

| AND | 0 | 1 | x | z |
|-----|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 |
| 1   | 0 | 1 | x | x |
| x   | 0 | x | x | x |
| z   | 0 | x | x | x |

| NAND | 0 | 1 | x | z |
|------|---|---|---|---|
| 0    | 1 | 1 | 1 | 1 |
| 1    | 1 | 0 | x | x |
| x    | 1 | x | x | x |
| z    | 1 | x | x | x |

| OR | 0 | 1 | x | z |
|----|---|---|---|---|
| 0  | 0 | 1 | x | x |
| 1  | 1 | 1 | 1 | 1 |
| x  | x | 1 | x | x |
| z  | x | 1 | x | x |

| NOR | 0 | 1 | x | z |
|-----|---|---|---|---|
| 0   | 1 | 0 | x | x |
| 1   | 0 | 0 | 0 | 0 |
| x   | x | 0 | x | x |
| z   | x | 0 | x | x |

# Truthtables of Logic Gates

| XOR | 0 | 1 | x | z |
|-----|---|---|---|---|
| 0   | 0 | 1 | x | x |
| 1   | 1 | 0 | x | x |
| x   | x | x | x | x |
| z   | x | x | x | x |

| XNOR | 0 | 1 | x | z |
|------|---|---|---|---|
| 0    | 1 | 0 | x | x |
| 1    | 0 | 1 | x | x |
| x    | x | x | x | x |
| z    | x | x | x | x |

# Operators

- Logical operators return a single bit (1 or 0). They can work in expressions, integers or groups of bits, and treat all nonzero values as „1“.

| Operator Symbol | Name             | Definition              | Comment                                                                                                           |
|-----------------|------------------|-------------------------|-------------------------------------------------------------------------------------------------------------------|
| !               | Logical negation | Unary complement        | Converts a non-zero value (TRUE) into a zero, a zero value (FALSE) into one, and an ambiguous truth value into x. |
| &&              | Logical AND      | ANDs two logical values | Used as a logical connective in statements                                                                        |
|                 | Logical OR       | ORs two logical values  | Used as a logical connective in statements                                                                        |

# Logical Operators (Examples)

```
wire[7:0]  x,y,z; // x, y and z are multibit variables
reg a;
if ((x == y) && (z)) a = 1 // a = 1 if x equals y, and
                           // z is nonzero
else a = !x              // a = 0 if x is anything
                           // but nonzero
```

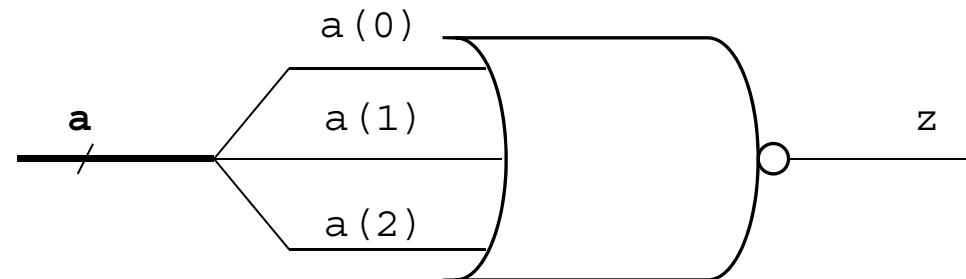
# Operators

- Reduction operators operate on all the bits of an operand vector and return a single bit value. They can be viewed as the unary form of the bit-wise operators before.

| Operator Symbol | Name                 | Definition                                                             | Comment                                                                   |
|-----------------|----------------------|------------------------------------------------------------------------|---------------------------------------------------------------------------|
| &               | Unary reduction AND  | Produces the single bit AND of all of the bits of the operand          | Unary reduction and binary bitwise operators are distinguished by syntax. |
|                 | Unary reduction OR   | Produces the single bit inclusive OR of all of the bits of the operand | See comment on top                                                        |
| ~&              | Unary reduction NAND | Produces the single bit NAND of all of the bits of the operand         | See comment on top                                                        |
| ~               | Unary reduction NOR  | Produces the single bit NOR of all of the bits of the operand          | See comment on top                                                        |
| ^               | Unary reduction XOR  | Produces the single bit XOR of all of the bits of the operand          | See comment on top                                                        |
| ~^ or<br>^~     | Unary reduction XNOR | Produces the single bit XNOR of all of the bits of the operand         | See comment on top                                                        |

# Reduction Operators (Examples)

```
module check_zero (a, z);
    input [2:0] a;
    output z;
    assign z = ~|a; // reduction NOR
endmodule
```



# Operators

- Shift operators shift the first operand by the number of bits specified by the second operand. Inserted positions are filled with zeros.

| Operator Symbol       | Name        | Definition                                                                                 | Comment                                     |
|-----------------------|-------------|--------------------------------------------------------------------------------------------|---------------------------------------------|
| <code>&lt;&lt;</code> | Left shift  | Shift the left operand left by the number of bit positions specified by the right operand  | Vacated bit positions are filled with zeros |
| <code>&gt;&gt;</code> | Right shift | Shift the left operand right by the number of bit positions specified by the right operand |                                             |

# Shift Operators (Examples)

```
assign c = a << 2 /* c = a shifted left 2 bits;  
vacant positions are filled with 0 s */
```

# Operators

- Concatenation operators combine two or more operands to form a larger vector

| Operator Symbol | Name          | Definition                                                       | Comment                                                               |
|-----------------|---------------|------------------------------------------------------------------|-----------------------------------------------------------------------|
| { }             | Concatenation | Joins together bits from two or more comma-separated expressions | Constants must be sized. Alternate form uses a repetition multiplier. |

# Concatenation Operators (Examples)

```
wire[1:0]  a,b; wire[2:0]  x; wire[3:0]  y,z;  
assign x = {1 b0, a}; // x[2]=0, x[1]= a[1], x[0]=a[0]  
assign y = {a,b}; /* y[3]=a[1], y[2]=a[0], y[1]=b[1],  
                 y[0]=b[0] */  
assign {cout,y} = x + z; // concatenation of a result
```

# Operators

- The replication operator makes multiple copies of an item.

| Operator Symbol | Name        | Definition                                | Comment |
|-----------------|-------------|-------------------------------------------|---------|
| {n<br>{item} }  | Replication | Joins together repetitively an expression |         |

# Replication Operator (Examples)

```
wire[1:0]  a,b;  
wire[4:0]  x,y;  
assign x = {3{1'b0}, a}; // equivalent to x = {0,0,0,a}  
assign y = {2{a}, 3{b}}; //equivalent to y = {a,a,b,b,b}
```

# Operators

- The conditional operator is the same like in C/C++. One of the two expressions is evaluated based on a condition.

| Operator Symbol                                                          | Name        | Definition | Comment |
|--------------------------------------------------------------------------|-------------|------------|---------|
| (cond) ?<br>(result if<br>cond<br>true):<br>(result if<br>cond<br>false) | Conditional | Assign     |         |

# Conditional Operator (Examples)

```
assign a = (g) ? x : y;  
assign a = (inc == 2) ? a+1: a-1;  
/* if (inc equals 2), a=a+1, else a=a-1 */
```

# Operator Precedence

This table shows the precedence of operators from highest to lowest. Operators on the same level evaluate from left to right. It is strongly recommended to use parentheses to define the order of precedence and improve the readability of your code.

| Operator Symbol         | Name                                    |
|-------------------------|-----------------------------------------|
| [ ]                     | Bit-select or part select               |
| ( )                     | Parentheses                             |
| !, ~                    | Logical and bit-wise NOT                |
| &,  , ~&, ~ , ^, ~^, ^~ | Reduction AND, OR, NAND, NOR, XOR, XNOR |
| + , -                   | Unary (sign) plus, minus                |
| { }                     | concatenation                           |
| { { } }                 | replication                             |

# Operator Precedence (II)

| Operator Symbol         | Name                                    |
|-------------------------|-----------------------------------------|
| [ ]                     | Bit-select or part select               |
| ( )                     | Parentheses                             |
| !, ~                    | Logical and bit-wise NOT                |
| &,  , ~&, ~ , ^, ~^, ^~ | Reduction AND, OR, NAND, NOR, XOR, XNOR |
| + , -                   | Unary (sign) plus, minus                |
| { }                     | concatenation                           |
| { { } }                 | replication                             |

# Registers

- Registers are the simplest form of storage devices found in digital systems. In Verilog, they are defined with the keyword **reg**, and you might give them optionally a size or bit width.

```
reg mybit; // a one bit register (scalar)
reg [15:0] mynumber; // a 16 bit register (vector)
reg [2:0] a; // a 3 bit register
a = mynumber[14:12]; // 3 bits loaded in to a
reg [0:15] mynumber; // also a 16 bit register, but now
                     // mynumber[0] is the most
                     // significant bit
reg signed [15:0] mynumber; // a signed 16 bit register
reg [-2:2] mynumber; // unusual, but a legal 5-bit vector
```

# Memories

- Memories are 2- or higher-dimensional registers

```
reg [15:0] my_mem [3:0]; // four 16 bit vectors
my_mem[2] = 117;
reg [15:0] my_mem [3:0][0:7]; 3-dim memory
reg [7:0] a;
a = my_mem[2][3][15:8]; // accessing 8 bits of my_mem
```

# Integer and time

- Registers and Memories should only been used for modeling of hardware, never as general purpose storage containers that may change over time. For this Verilog offers integer and time variables. Usually, an integer variable consumes 32 bits and is supposed to store data in the 2´s complement. A time variable will use 64 bits of data and is supposed to be used with the \$time function.

```
integer a,b,c; // three integer numbers
integer d[100:0]; // an array of integers
time t, deltat; // two time variables
time elapsed_time[15:0]; // 16 time variables
```

# Structural and Behavioral Verilog

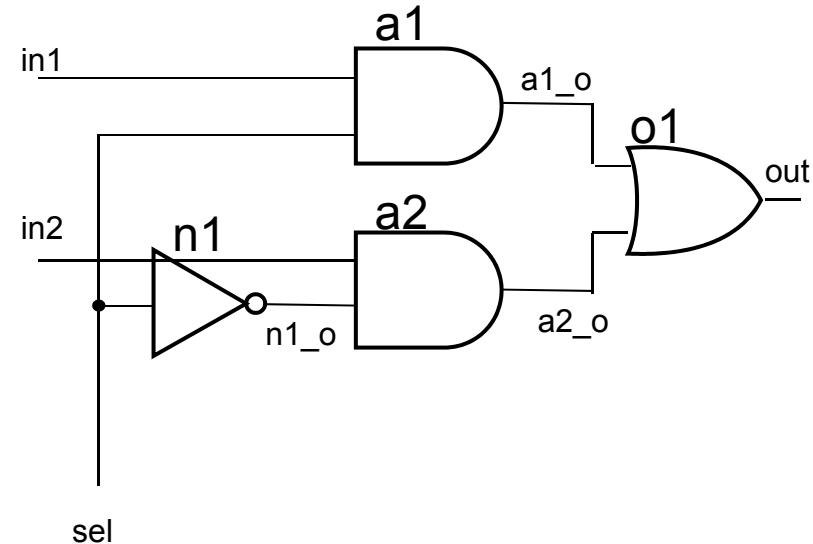


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Gate-level Verilog

- Verilog is perfectly suited for the description of gate-level netlists. Therefore de-facto all tool-written netlists on gate-level are in the Verilog language. But it is also possible to design on Gate-level in Verilog:

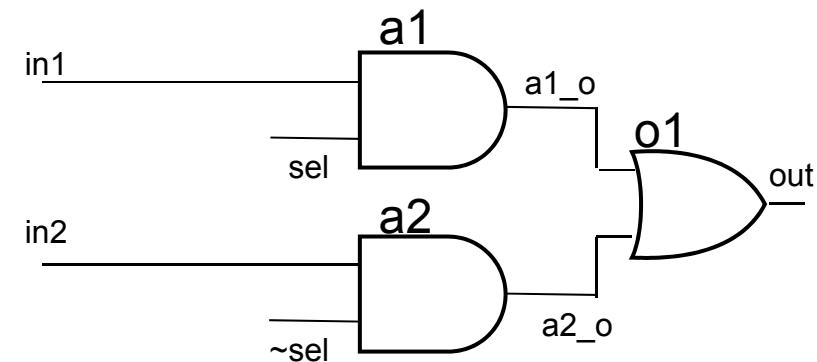
```
module mux2 (in1, in2, sel, out);  
    output out;  
    input in1, in2, sel;  
  
    and a1(a1_o, in1, sel);  
    not n1(n1_o, sel);  
    and a2(a2_o, in2, n1_o);  
    or o1(out, a1_o, a2_o);  
endmodule
```



# Structural Verilog

- You may also combine gate-level primitives with unary operators:

```
module mux2 (in1, in2, sel, out);  
    output out;  
    input in1, in2, sel;  
  
    and a1(a1_o, in1, sel);  
    and a2(a2_o, in2, ~sel);  
    or  o1(out, a1_o, a2_o);  
endmodule
```



# How to test ...

- You may add lines of code inside the module to test the behavior....

```
module mux2_with_test (in1, in2, sel, out);
    output out;
    input in1, in2, sel;

    and #1 a1(a1_o, in1, sel);
    and #1 a2(a2_o, in2, ~sel);
    or  #2 o1(out, a1_o, a2_o);

initial
begin
    $monitor($time, ,
              "in1 = %b in2 = %b sel = %b out = %b",
              in1, in2, sel, out);
    #10 in1 = 0; in2 = 1; sel = 0;
    #10 sel = 1;
    #5  in1 = 1; in2 = 0;
    #8  $finish;
end
endmodule
```

# The result ....

- Results of simulation

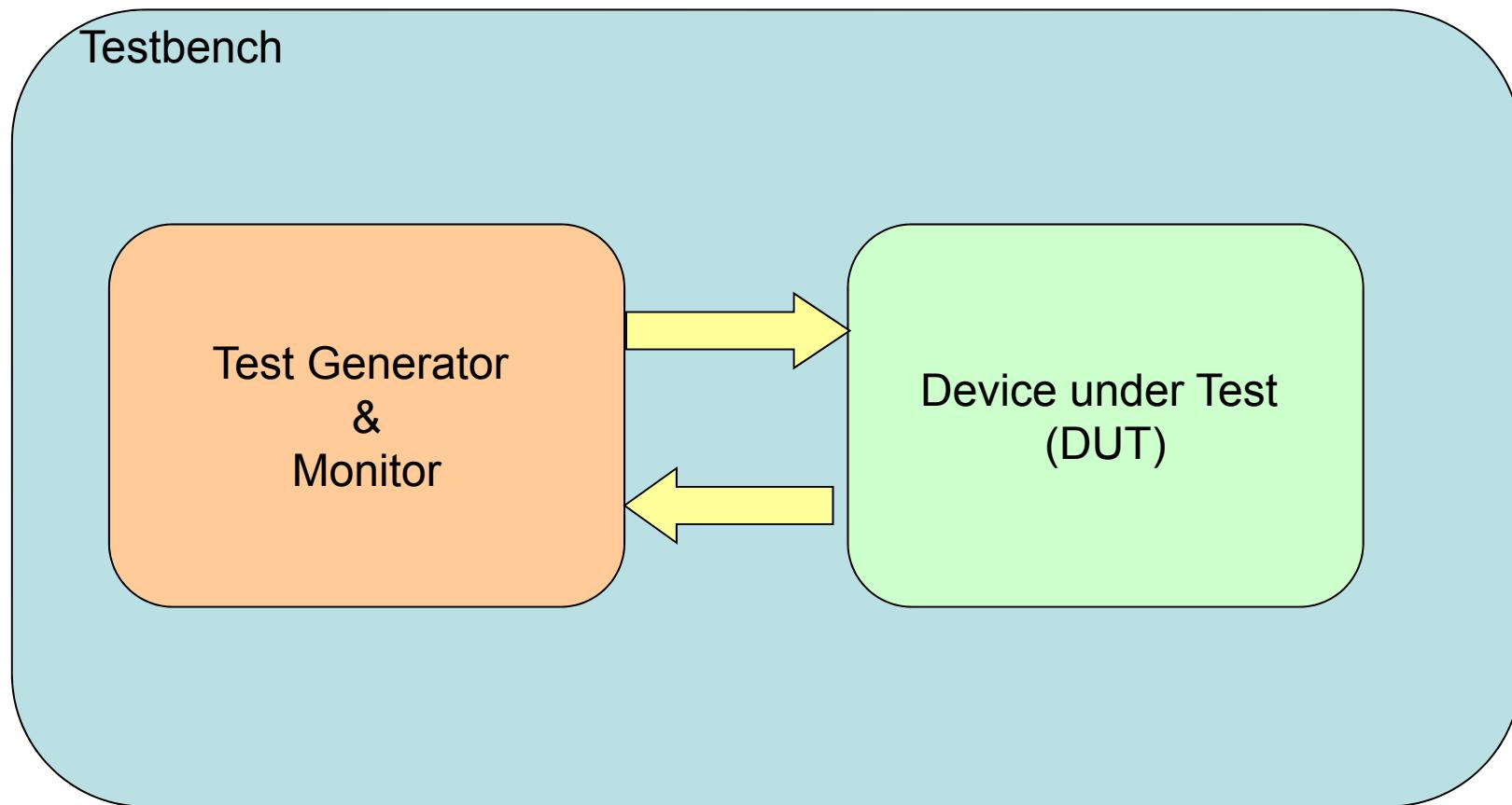
```
0  in1 = x  in2 = x  sel = x  out = x
10 in1 = 0  in2 = 1  sel = 0  out = x
13 in1 = 0  in2 = 1  sel = 0  out = 1
20 in1 = 0  in2 = 1  sel = 1  out = 1
23 in1 = 0  in2 = 1  sel = 1  out = 0
25 in1 = 1  in2 = 0  sel = 1  out = 0
28 in1 = 1  in2 = 0  sel = 1  out = 1
```

- Works ....

... But mixing the stimulus and the structure is not a good idea ...

# Creating a testbench

- Splitting the description and the test code is much better:

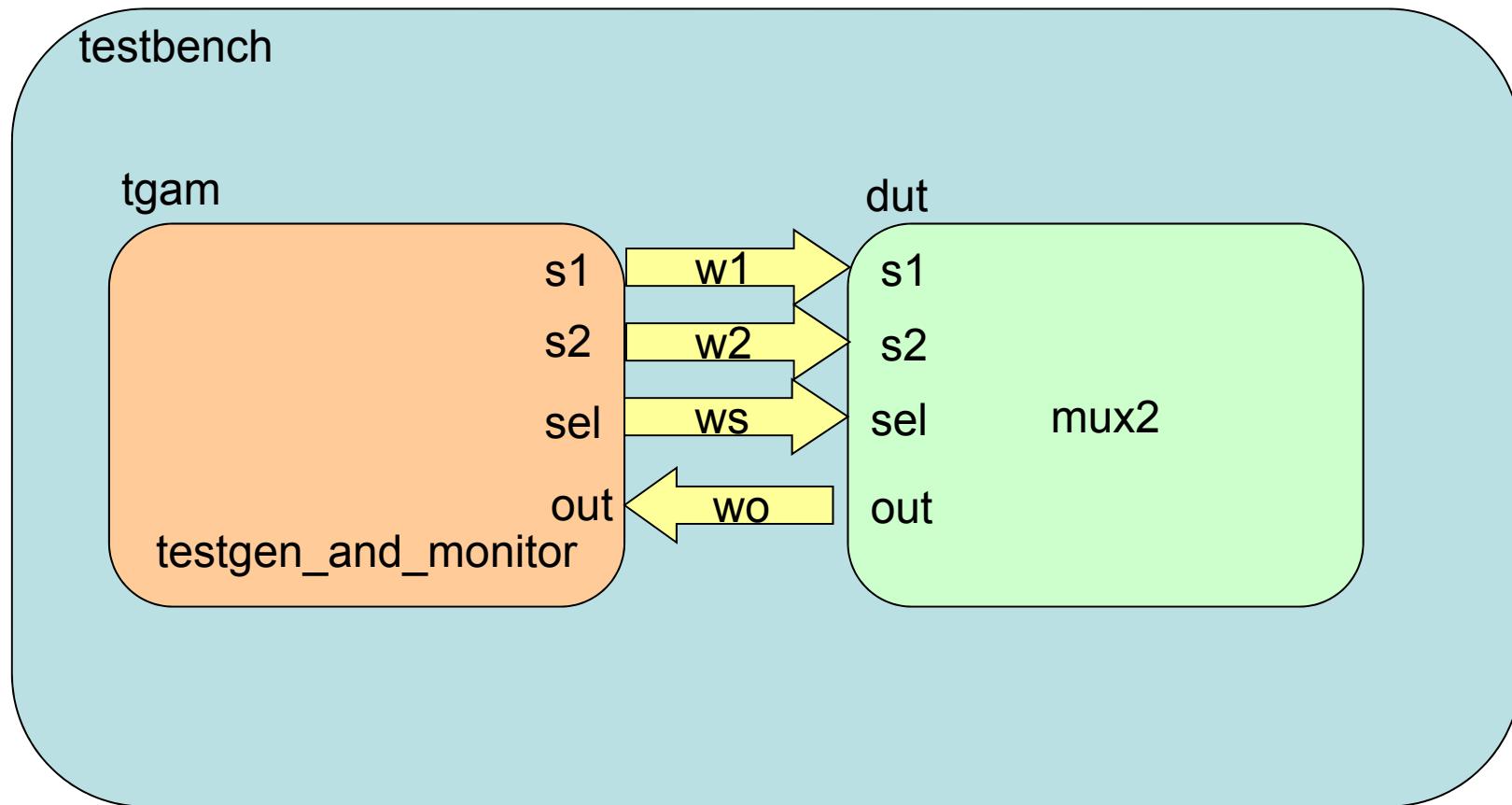


# How to test better ...

- Simply use the module mux2 defined some pages ago ... Plus:

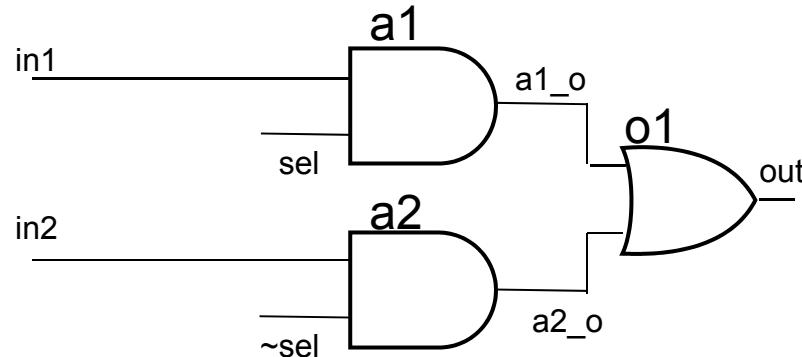
```
module testbench;  
  
    wire w1, w2, ws, wo;  
  
    mux2 dut (w1, w2, ws, wo);  
    testgen_and_monitor tgam (w1, w2, ws, wo);  
endmodule  
  
module testgen_and_monitor (output in1, in2, sel,  
                           input out);  
initial  
begin  
    $monitor($time,,,  
            "in1 = %b in2 = %b sel = %b out = %b",  
            in1, in2, sel, out);  
    #10 in1 = 0; in2 = 1; sel = 0;  
    #10 sel = 1;  
    #5 in1 = 1; in2 = 0;  
    #8 $finish;  
end  
endmodule
```

# Graphically ...



# Behavioral Verilog

- The behavior of a block can also be expressed in terms of procedural statements, rather than gate-level primitives.



| in1 | in2 | sel | out |
|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   |
| 0   | 1   | 0   | 1   |
| 1   | 0   | 0   | 0   |
| 1   | 1   | 0   | 1   |
| 0   | 0   | 1   | 0   |
| 0   | 1   | 1   | 0   |
| 1   | 0   | 1   | 1   |
| 1   | 1   | 1   | 1   |

# Behavioral Verilog

- The behavior of a block can also be expressed in terms of procedural statements, rather than gate-level primitives

```
module mux2_behav (in1, in2, sel, out);  
    output reg out;  
    input in1, in2, sel;  
  
    always @ (in1, in2, sel) begin  
        if(sel)  
            out = in1;  
        else  
            out = in2;  
    end  
endmodule
```

The “=” inside a procedural statement must be made to a register

The “@” is an event control statement, followed by a sensitivity list.

If a change occurs in one of the signals in the sensitivity list, the part between begin...end is executed

**Don't get confused: if the synthesis tool finds out we do not need a register, it is not created!**

# Processes

- The statement always @(in1, in2, sel) tells the simulator to suspend execution if there is no change in at least one of the three inputs
- The always continuously repeats its statement, never exiting or stopping.
- Don't get confused that „out“ is now a reg. Looking from the outside, the circuit is clearly still a combinational block.
- The basic essence of a behavioral model is the process:
  - Independent thread of control
  - Think of a complex system as a large set of independent, but communicating processes
- In contrast to always : The initial construct is executed only once, otherwise similar

Processes are a group of sequential statements; indicated either by initial, always statements or of „continuous“ nature

# Concurrency

- Concurrent processes „live“/„happen“ at the same time: not one after the other.
- One model waits for an event that happens (concurrently) in another model
- What stops a process? Only a delay (#10), wait, or @ statement. If this is missing, the process will run forever.

# A few reflections on time

- In a behavioral model, time is not existent! Behavioral statements (if, loop, while, ....) take zero time to execute
- Time only advances in a process, if a wait, @ or delay is executed
- An initial block will only execute once. It will always start at time=0 (even if there are more than one initial block in the system).
- An always process will execute forever

# Continuous assignments

- Continuous assignments are always active:

```
module oneBitFullAdder (output cOut, sum,  
                        input: in1, in2, in3);  
    assign sum = in1 ^ in2 ^ in3,           XOR of the three inputs  
           cOut = (in1 & in2) | (in1 & in3) | (in2 & in3);  
endmodule  
Majority of the three inputs
```

# Blocking vs. Nonblocking assignment

- Blocking assignment done with „=“
- Only inside a process
- Only to reg
- Value of the left-hand-side (lhs) changes immediately

```
reg x1, qbar;

initial
begin
    qbar = 0;    // blocking assignment
    #100;        // wait for 100ns
    qbar = 1;
end
```

# Blocking vs. Nonblocking assignment

- Nonblocking assignment done with „`<=`“
- Only inside a process
- Only to reg
- Value of the left-hand-side (lhs) changes only after rhs has been evaluated
- A time may be given additionally as the assignment delay

```
reg q;  
  
always @ (posedge clk)  
  q <= #10 d; // non-blocking
```

Doing Synthesis, this  
will always create a  
flip-flop (register)

@ is perfectly suited for  
synthesis of  
synchronous systems!

# Continuous assignment (rep `d)

- Continuous assignment done with „=“
- Only to wire
- Always sensitive to changes on the rhs

```
assign x1 = a & b; // continuous assignment
```

# Blocking a process

- Processes can be blocked the following way:

- Wait for time to elapse:

```
#300; // waits for 300ns
```

- Wait for a signal edge

```
@(posedge clk)
```

- Wait for a signal to change

```
@(sel_bar);
```

- Wait for a logical value

```
wait(sel_bar);
```

# Behavioral Verilog

- Some nice & useful procedural Verilog statements:

```
...
input      go;
always begin
    // initialize your block
    wait (go);
    // do some computations
...
    wait (~go);
end
endmodule
```

The execution stops here, unless `go=1`. Then, the following code is executed once.



Wait is not used for synthesis!

The execution stops here, unless `go=0`. Then, the module becomes inactive again.

# Behavioral Verilog

- The conditional statement: If ....else

```
...  
if ((a > b) && (c < b))  
    // then-statement goes here  
else if (a > d)  
    // else-if statement goes here  
else  
    // else clause goes here
```

Else / else if is  
a) Optional  
b) Always matched to  
the nearest “if”

# Control structures

- Control structures are very much like C
  - if, while, for, forever, repeat, case, fork/join

```
i = 16;  
while(i)  
begin  
    // do something useful here  
    i = i - 1;  
end
```

A “zero” would be  
interpreted as “false”

Condition must change  
inside your process!

# Control structures

- Control structures are very much like C
  - if, while, for, forever, repeat, case, fork/join

```
module a_very_abstract_dram;  
  
    always  
        begin  
            read_spd; // read timing/latency settings  
            forever  
                begin  
                    get_commands_from_mem_ctrl;  
                end  
            end  
        end  
    endmodule
```

# Control structures

- Control structures are very much like C
  - if, while, for, forever, repeat, case, fork/join

```
begin:break
  for(i=0; i<n;i=i+1)
    begin:continue
      if(a==0)
        disable continue;
      ... // other statements
      if(a==b)
        disable break;
      ... // other statements
    end
  end
```

Proceed with  $i = i + 1$ ;  
but stay in loop

Exit the loop

# Control structures

- Comparison of if and case

```
reg[15:0] signed dram[0:8191] // signed 8192 x 16 bit memory
reg[15:0] ir; // 16 bit instruction register
```

```
always
```

```
begin
```

```
    @posedge(clk)
```

```
        ir <= dram[pc]; // get the ir from a dram address
```

```
    @posedge(clk)
```

```
        if(ir[15:13] == 3'b000) // begin decoding
```

```
            pc <= dram[ir[12:0]]; // and executing
```

```
        else if(ir[15:13] == 3'b001)
```

```
            pc <= pc + dram[ir[12:0]];
```

```
        else if(ir[15:13] == 3'b010)
```

```
            acc <= -dram[ir[12:0]];
```

```
        ...
```

```
        pc <= pc + 1
```

```
    end
```

Very powerful and general comparisons possible – but exact evaluations needed

But readability suffers a bit ....

# Control structures

- Comparison of if and case

```
reg[15:0] ir; // 16 bit instruction register

always
begin
    @posedge(clk)
        ir <= dram[pc]; // get the ir from a dram address
    @posedge(clk)
        case (ir[15:13])
            3'b000:      pc <= dram[ir[12:0]];
            3'b001:      pc <= pc + dram[ir[12:0]];
            3'b010:      acc <= -dram[ir[12:0]];
            ...
        endcase
        pc <= pc + 1
end
```

Much better readable,  
or?

# Control structures

- The case statement may even include unknown and high-impedance values

```
reg  reg1; // a one bit register
begin
    case (reg1)
        1bz: $display(„reg1 is high impedance“);
        1bx: $display(„reg1 is unknown“);
        default: $display(„reg1 has the following value: %b“, reg1);
    endcase
end
```

# Control structures

- Casex and casez even allow high-impedance or unknown values in case-statements:

```
module decoder;  
    reg [7:0] reg1;  
  
    always  
        begin  
            reg1 = 8'bx1x0x1x0;  
            casex (reg1)  
                8'b001100xx: display(„Case 1“);  
                8'b1100xx00: display(„Case 2“);  
                8'b00xx0011: display(„Case 3“);  
                8'bxx001100: display(„Case 4“);  
            endcase  
        end  
endmodule
```

Case 2 ...

- Note that casez treats z and x as don't care values

# Functions and tasks

- Using modules you are able to partition large pieces of code. But modules imply structural boundaries. If this is not the case, you may use functions and tasks. Example (see next page):

```
module advanced_decoder_with_function;
    reg [15:0] signed m [0:8191];// signed 8192 x 16 bit memory
    reg [12:0] signed pc; // signed 13 bit program counter
    reg [12:0] signed acc; // signed 13 bit accumulator
    reg ck; // a clock signal
    always
        begin: executeInstructions
            reg [15:0] ir; // 16 bit instruction register
            @(posedge ck)
                ir <= m [pc];
            @(posedge ck)
                case (ir [15:13]) // as before
                    3'b111: acc <= multiply(acc, m [ir [12:0]]);
                endcase
                pc <= pc + 1;
        end
end
```

# Functions and tasks

```
function signed [12:0] multiply
    (input signed [12:0] a,
     input signed [15:0] b);
begin: serialMult
    reg [5:0] mcnd, mpy;
    mpy = b[5:0];
    mcnd = a[5:0];
    multiply = 0;
    repeat (6)
        begin
            if (mpy[0])
                multiply = multiply + {mcnd, 6'b000000};
            multiply = multiply >> 1;
            mpy = mpy >> 1;
        end
    end
endfunction
endmodule
```

# Functions and tasks

```
module advanced_decoder_with_task;
    reg [15:0] signed m [0:8191];// signed 8192 x 16 bit memory
    reg [12:0] signed pc; // signed 13 bit program counter
    reg [12:0] signed acc; // signed 13 bit accumulator
    reg ck; // a clock signal
    always
        begin: executeInstructions
            reg [15:0] ir; // 16 bit instruction register
            @ (posedge ck)
                ir <= m [pc];
            @ (posedge ck)
                case (ir [15:13]) // as before
                    3'b111 : multiply (acc, m [ir [12:0]]);
                endcase
                pc <= pc + 1;
        end
end
```

# Functions and tasks

```
task multiply
    (inout [12:0] a,
     input [15:0] b);
    begin: serialMult
        reg [5:0] mcnd, mpy; //multiplicand and multiplier
        reg [12:0] prod; //product
        mpy = b[5:0];
        mcnd = a[5:0];
        prod = 0;
        repeat (6)
            begin
                if (mpy[0])
                    prod = prod + {mcnd, 6'b000000};
                prod = prod >> 1;
                mpy = mpy >> 1;
            end
            a = prod;
        end
    endtask
endmodule
```

# Functions and tasks: comparison

- We learned, that functions and tasks are similar to software functions and procedures. Their main goal is to make code more readable. However, there are differences you must know as a programmer:

| Category | Task                                                                                            | Function                                                                                                                                                                                                           |
|----------|-------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Calling  | A task call is a separate procedural statement. It cannot be called from a continuous statement | A function call is an operand in an expression. It is called from within the expression and returns a value used in the expression. Functions may be called within procedural and continuous assignment statements |
| I/O      | Can have zero or more arguments of any kind.                                                    | Has at least one input, but no inouts or outputs. At least one value is returned.                                                                                                                                  |

# Functions and tasks: comparison

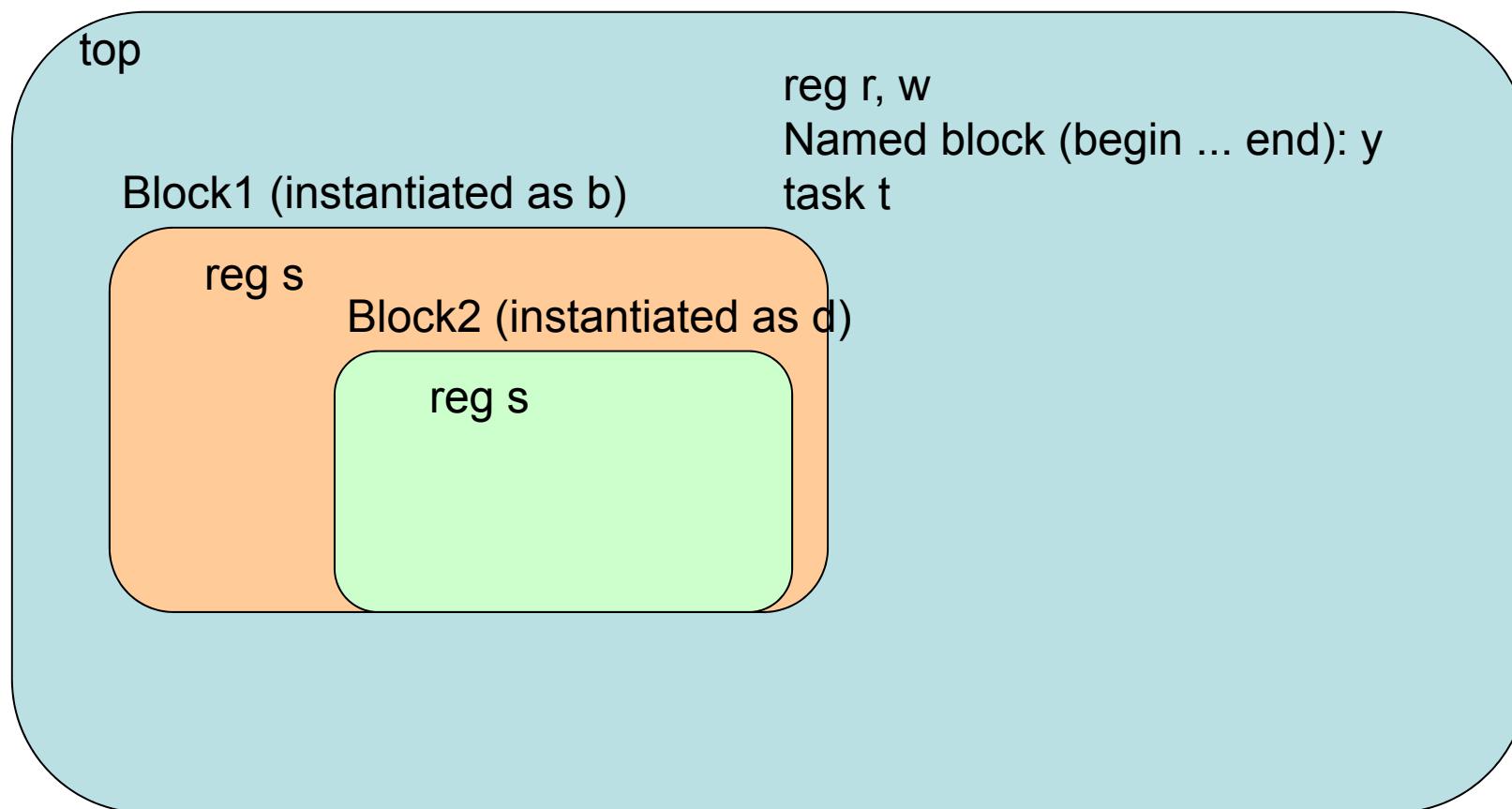
| Category                          | Task                                                                                                                                                                                                                           | Function                                                                                                                                                                        |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Timing and event control          | A task can contain timing and event control (#, @ and wait). It can therefore be concurrently active if called from concurrent always/initial blocks.                                                                          | Functions do not contain these statement. They are executed in zero time.                                                                                                       |
| Calling others tasks or functions | A task may enable other tasks and functions                                                                                                                                                                                    | A function can enable other functions, but not other tasks.                                                                                                                     |
| Storage                           | Storage of the inputs, outputs and internally declared variables is static – concurrent calls share the storage.<br>Exception: if the task is declared automatic, then the storage is dynamic and each call gets its own copy. | Storage of the inputs and internally declared variables is static. If the function is declared automatic, then the storage is dynamic and recursive calls get their own copies. |

# Functions and tasks: comparison

| Category        | Task                                                                                                                            | Function                                                                                                                                                 |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Returned values | A task does not return values. If inout or output ports are changed, then this is copied back at the end of the task execution. | A function returns a single value to the expression that called it. The value to be returned is assigned to the function identifier within the function. |

# Scope and Hierarchy

- Building up system hierarchically is essentially needed for mastering the complexity. We need to understand how to address variables and where data are known across hierarchies.



# Scope and Hierarchy

- Look at the following code:

```
module top;
    reg r;           //hier. name is top.r
    wire w;          //hier. name is top.w

    b instance1();

    always
        begin: y
            reg q;      //hier. name is top.y.q
        end

    task t;
        begin: c  // hier. name is top.t.c
            reg q;  // hier. name is top.t.c.q
            disable y; //OK
        end
    endtask
endmodule
```

```
module b;
    reg s; /* hierarchical name is
top.instance1.s */

    always
        begin
            t; //OK
            disable y; //OK
            disable c; //Nope, c is not known
            disable t.c; //OK
            s = 1; //OK
            r = 1; //Nope, r is not known
            top.r = 1; //OK
            t.c.q = 1; //OK
            y.q = 1; /* OK, a different q
                        than t.c.q */
        end
    endmodule
```

# Scope and Hierarchy

- Registers and wires are not forward referencing, can only be accessed in the local scope!
- Since module b is instantiated in module top, a procedural statement in b can enable tasks, functions and named blocks defined in the local scope of module top.
  
- Consequence:
  - Functions and tasks used in many parts of your design should be defined in the top module.
  - Be aware, in principle anything can access anything else through hierarchical naming, but this is not good design style.

# Finite State Machines

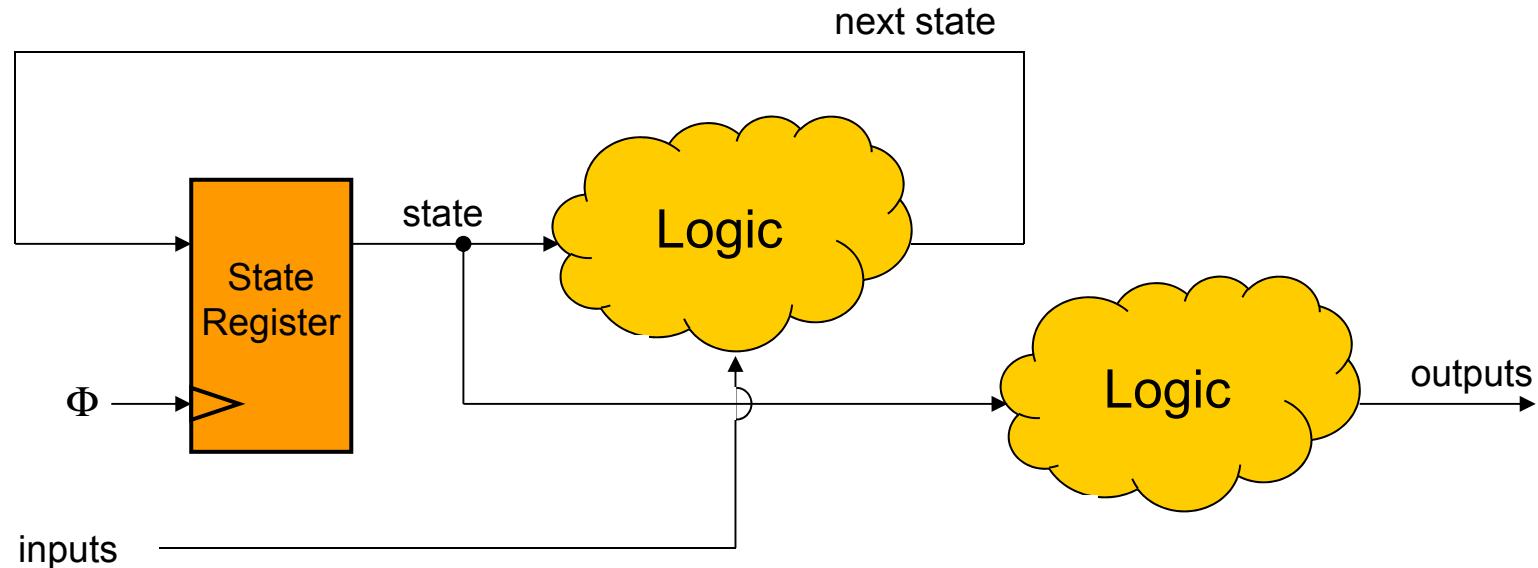


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Finite State Machines

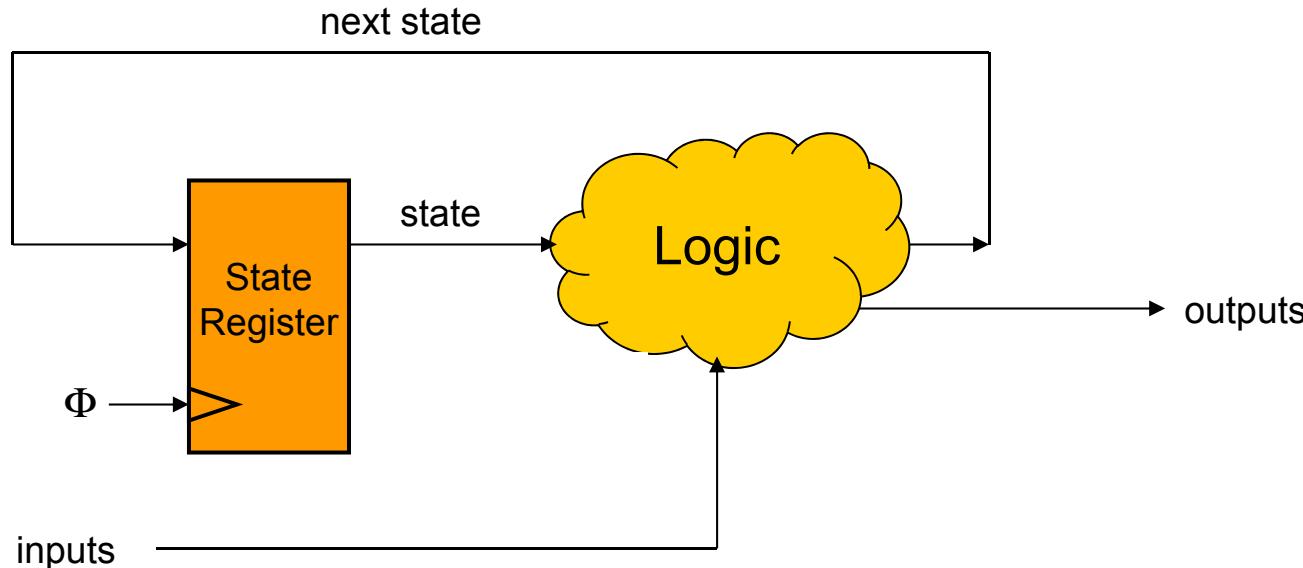
- Combinational logic and sequential elements can be combined to the specification of a Finite State Machine. In general, Finite State Machines (FSMs) can be divided into 2 classes:
  - Moore Machines
    - The outputs depend only on the current state
    - The next state depends on current state and inputs
  - Mealy Machines
    - The outputs depend on current state and inputs
    - The next state depends on current state and inputs

# FSM: Moore machine



- Characteristics of a Moore Machine:
- Outputs depend only on the current state
- Next state depends on current state and inputs

# FSM: Mealy machine



- Characteristics of a Mealy Machine:
- Outputs and next state both depend on current state and inputs

# Table Notation

- FSMs can be represented as a State Transition Table
  - The table exactly defines the values for the next state and all outputs (right side of the table) depending on the current state and the inputs (left side)
  - Logic functions can be easily derived from the table, e.g.

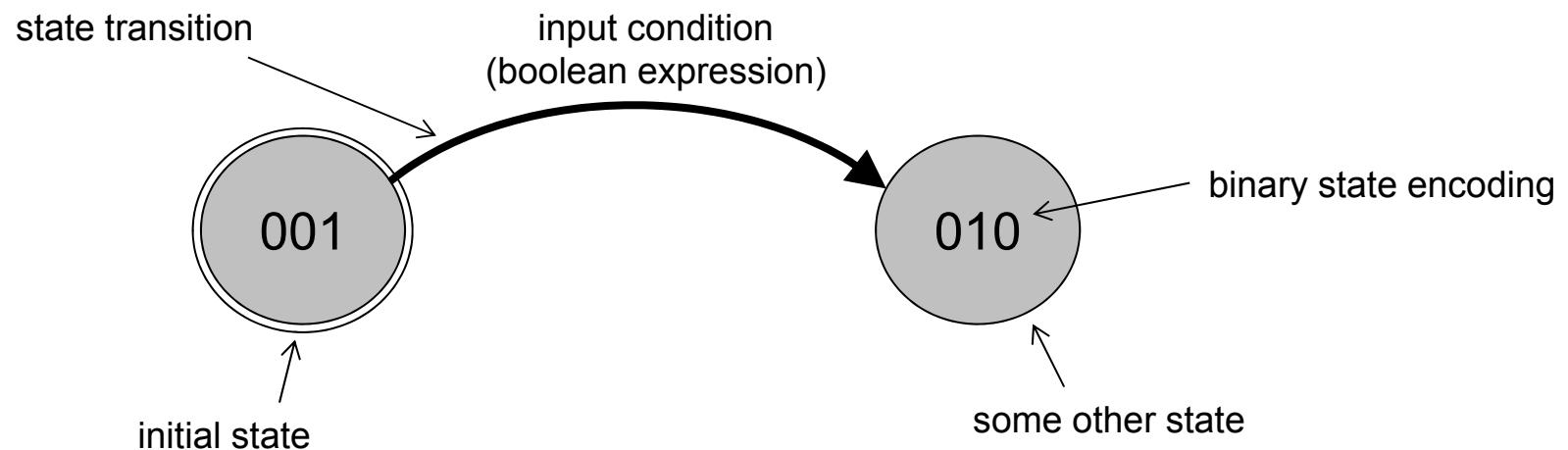
$$S_0' = \overline{S_2 S_1 S_0} ab + \overline{S_2 S_1 S_0} a + \dots$$

- Current state and next state are encoded binary (in the example: 3 bits)
- “Don’t cares” in the input conditions are indicated by an ‘x’
- In each state, every possible combination of input values should be covered by exactly one line in the table (not more, not less)

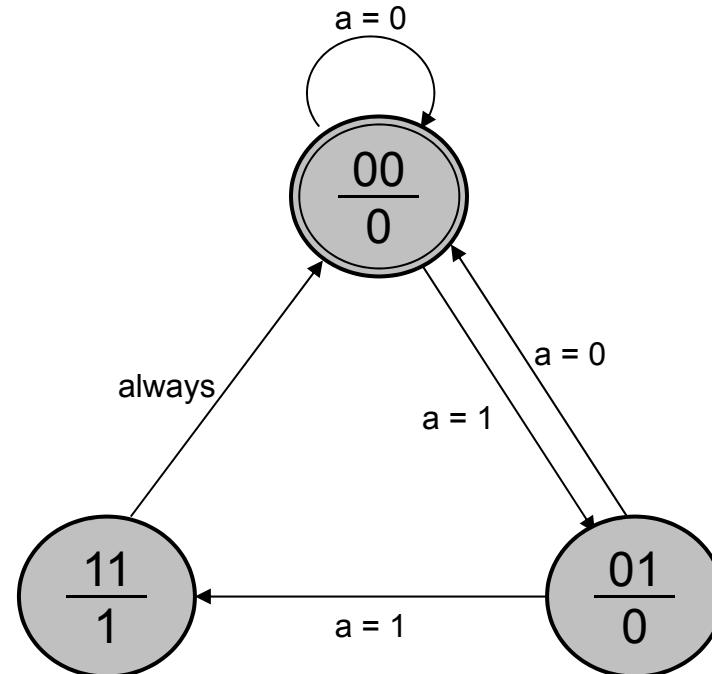
| current state | inputs | next state       | outputs |
|---------------|--------|------------------|---------|
| $S_2 S_1 S_0$ | a b    | $S_2' S_1' S_0'$ | x y     |
| 0 0 0         | 0 0    | 0 0 0            | 0 0     |
| 0 0 0         | 0 1    | 0 0 1            | 0 0     |
| 0 0 0         | 1 x    | 1 0 1            | 0 0     |
| 0 0 1         | 1 0    | 0 1 0            | 0 1     |
| ...           | ...    | ...              | ...     |

# Graph Notation

- FSMs can also be represented as a graph
  - Every state is a node in the graph
  - Every state transition is an edge (arrow)
    - The arrows indicate which state is taken in the next cycle, depending on the inputs and the current state
  - State encoding is displayed inside the nodes

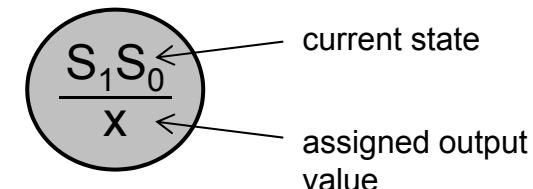


# Example for a Moore Machine

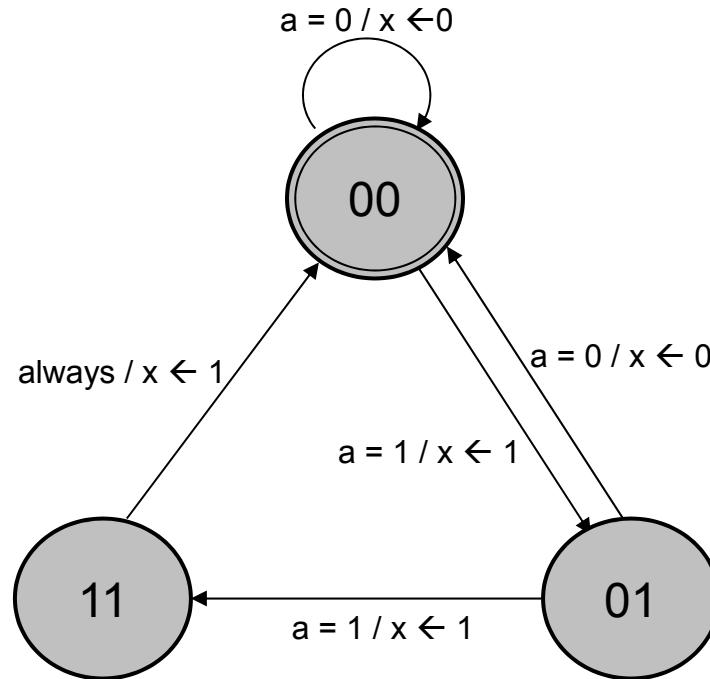


| current state<br>$S_1S_0$ | inputs<br>a | next state<br>$S_1'S_0'$ | outputs<br>x |
|---------------------------|-------------|--------------------------|--------------|
| 0 0                       | 0           | 0 0                      | 0            |
| 0 0                       | 1           | 0 1                      | 0            |
| 0 1                       | 0           | 0 0                      | 0            |
| 0 1                       | 1           | 1 1                      | 1            |
| 1 1                       | x           | 0 0                      | 0            |

Notation:



# Example for a Mealy Machine



| current state<br>$S_1S_0$ | inputs<br>a | next state<br>$S_1'S_0'$ | outputs<br>x |
|---------------------------|-------------|--------------------------|--------------|
| 0 0                       | 0           | 0 0                      | 0            |
| 0 0                       | 1           | 0 1                      | 1            |
| 0 1                       | 0           | 0 0                      | 0            |
| 0 1                       | 1           | 1 1                      | 1            |
| 1 1                       | x           | 0 0                      | 0            |

- Because the outputs of a Mealy Machine also depend on the inputs, the values assigned to them are annotated at the transitions
- The notation is:

input condition / output assignment



# State Encoding

- The encoding of the states plays a key role for the implementation of a FSM
  - It influences the complexity of the logic functions, the hardware costs of the circuits, timing issues, power, etc.
- Therefore, several common coding styles with different features exist
  - regular encoding
  - „one hot“ encoding
  - ...
- The optimum choice depends on the used technology (ASIC, PLA, FPGA, etc.) as well as on the given design goals

## ■ Regular Encoding

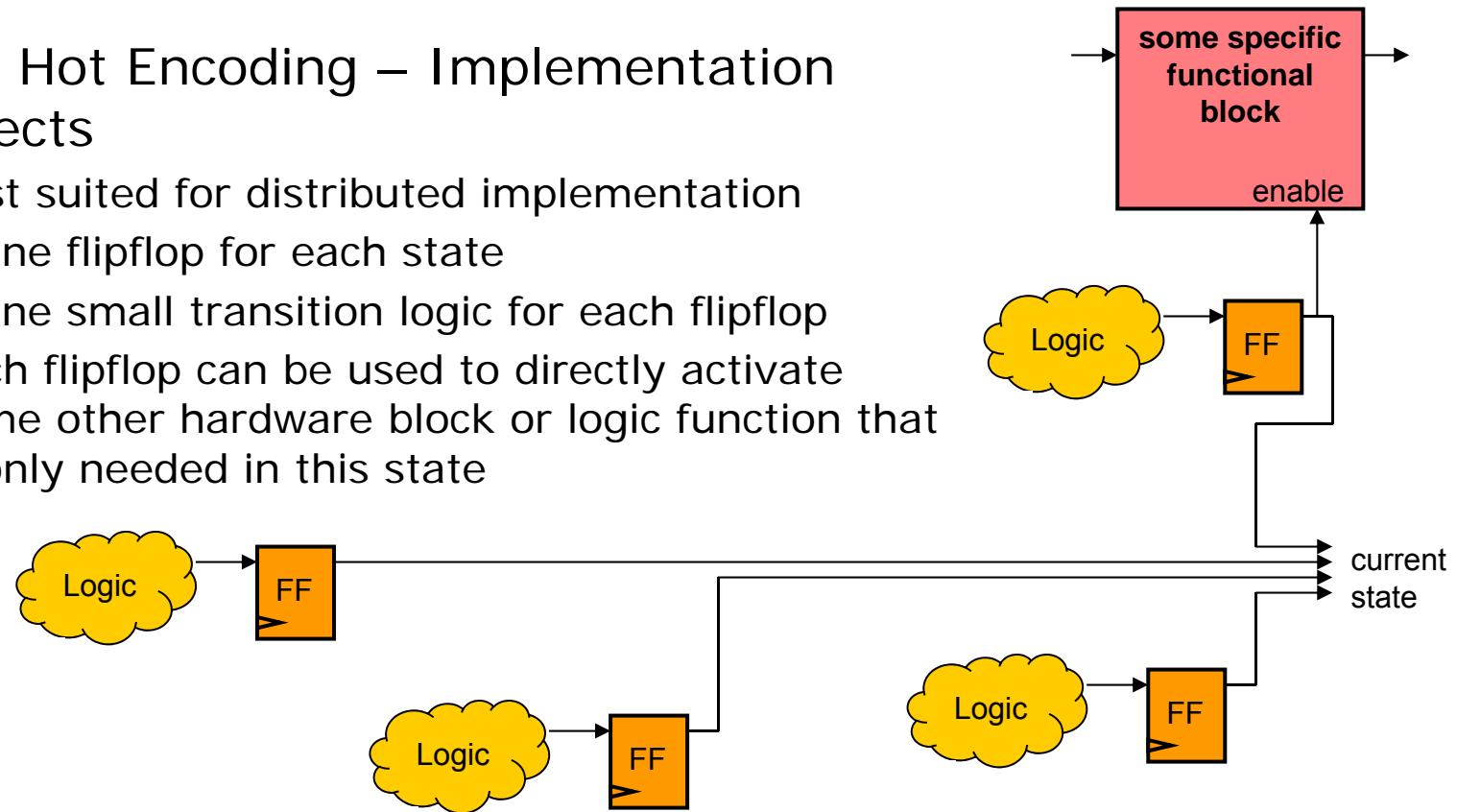
- The minimum number of bits is used to encode the states
  - At least  $N$  bits are required to encode up to  $2^N$  states
- Codes can be assigned to states arbitrarily or according to certain rules (e.g., in order to minimize complexity of the logic)
- Advantages:
  - Minimum number of flipflops required
- Disadvantages:
  - Due to the compactness of the state encoding, the logic functions for calculating the next state and the outputs can become more complex
  - On average, many bits switch when the state changes
    - Higher power consumption
    - Glitches can occur

- One Hot Encoding

- N bits are used to encode N states
  - In each state, exactly one bit is '1', all others are '0'
  - → therefore the name "one hot" encoding
- Advantages:
  - In many cases, less logic is required
    - many small logic functions are used instead of few complex functions
    - particularly advantageous for FPGA implementations
  - Low switching activity, resulting in ...
    - lower power consumption
    - less glitches
- Disadvantages:
  - The number of required flipflops grows linearly with the number of states
    - High hardware costs for large FSMs

# State Encoding

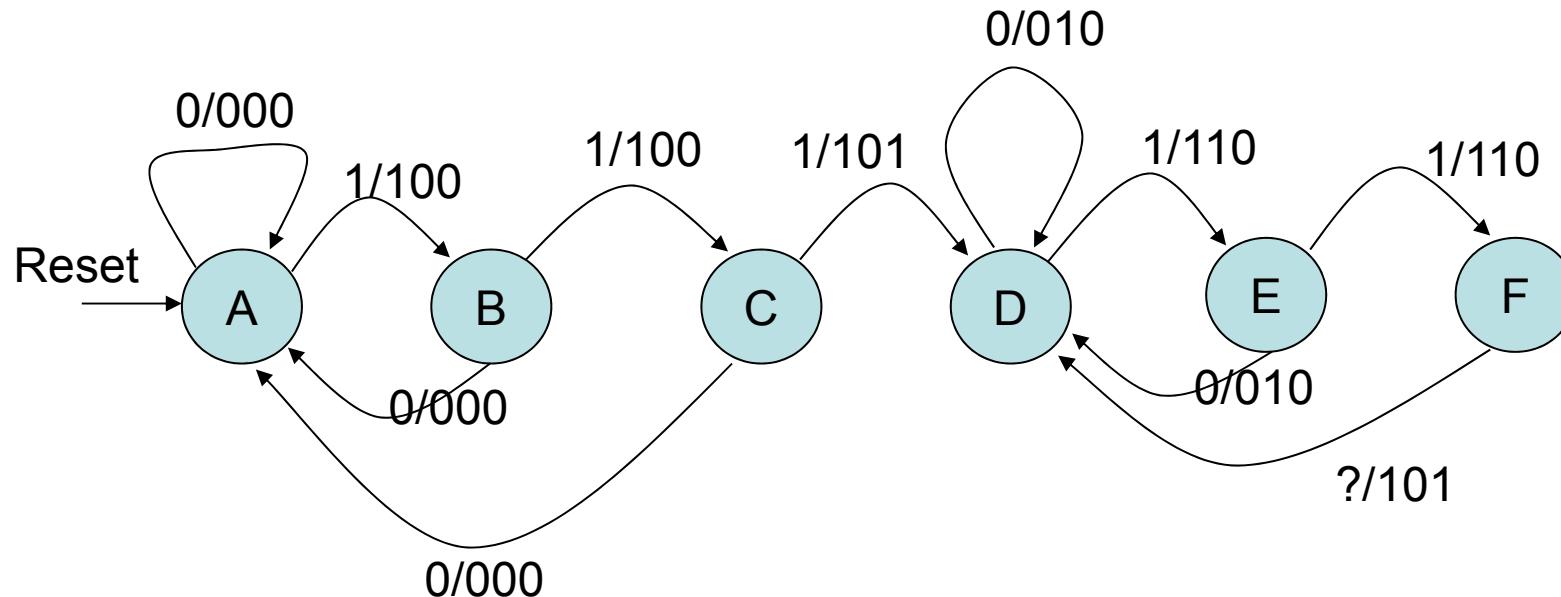
- One Hot Encoding – Implementation Aspects
  - Best suited for distributed implementation
    - One flipflop for each state
    - One small transition logic for each flipflop
  - Each flipflop can be used to directly activate some other hardware block or logic function that is only needed in this state



- From an abstract point of view, all N flipflops together can also be seen as one single state register of size N

# Verilog coding of a FSM

- There is more than one way to model a FSM. We will show one example of explicit style modeling of a FSM using case statements.
- The following example uses 6 states, using one input and three output bits.



# Verilog coding of a FSM

```
module fsm
  (input i, clock, reset,
   output reg [2:0] out);

  reg [2:0] currentState,
            nextState;
  localparam [2:0]
    A = 3'b000,
    // The state labels and their assignments
    B = 3'b001,
    C = 3'b010,
    D = 3'b011,
    E = 3'b100,
    F = 3'b101;

  always @(*)
    // The combinational logic
    case (currentState)
      A: begin
        nextState = (i == 0) ? A : B;
        out = (i == 0) ? 3'b000 : 3'b100;
      end
      B: begin
        nextState = (i == 0) ? A : C;
        out = (i == 0) ? 3'b000 : 3'b100;
      end
      C: begin
        nextState = (i == 0) ? A : D;
        out = (i == 0) ? 3'b000 : 3'b101;
      end
    endcase
  endmodule
```

# Verilog coding of a FSM

```
D: begin
nextState = (i == 0) ? D : E;
out = (i == 0) ? 3'b010 :
      3'b110;
end
E: begin
nextState = (i == 0) ? D : F;
out = (i == 0) ? 3'b010 :
      3'b110;
End
F: begin
nextState = D;
out = 3'b101;
end

default: begin
// oops, undefined states.
// Go to state A
nextState = A;
out = (i == 0) ? 3'bxxxx :
      3'bxxxx;
end
endcase
always @(posedge clock or
         negedge reset) // The state
register

if (~reset)
currentState <= A;
// the reset state
else
currentState <= nextState;

endmodule
```

# Concurrent Processes



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Concurrent Processes

- Processes should interact with each other. You may formulate every system as one process, but readability and modularity are very poor.
- We need a mechanism of interactions between processes – events and wait-statements

```
module dEdgeFF (output reg q, input clock, data);  
  
    always @ (negedge clock) q <= data;  
  
endmodule
```

# Event control statements

- General form of event control statements (see also BNF for Verilog):

## <event\_control>

```
::= @ <identifier>
||= @ ( <event_expression> )
```

## <event\_expression>

```
::= <expression>
||= posedge <scalar_event_expression>
||= negedge <scalar_event_expression>
||= <event_expression> or <event_expression>
```

# Event control statements

- Example of two flip-flops

```
module toplevel(input clock, reset, output flop1, flop2);  
  
always @ (posedge reset or posedge clock)  
  if (reset)  
    begin  
      flop1 <= 0;  
      flop2 <= 1;  
    end  
  else  
    begin  
      flop1 <= flop2;  
      flop2 <= flop1;  
    end  
endmodule
```

# Fibonacci numbers

- A more abstract form of event control is the named event statement.  
It allows a trigger to be sent to another part of the design. We will use a Fibonacci number generator as one example.
- Quick recap: Fibonacci numbers or sequence is a series of numbers which follow the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with  $F_0=0$  and  $F_1=1$ .

- The first numbers of the Fibonacci Sequence are:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

# Fibonacci number generators: Named Events

```
module topFib;
    wire [15:0] number, numberOut;
    numberGen ng (number);
    fibNumCalc fnc (number, numberOut);
endmodule

module numberGen(number) ;
    output reg [15:0] number = 0;

    event ready; //declare the event
    always
        begin
            #50 number = number + 1;
            #50 -> ready; //generate event signal
        end
    endmodule
```

# Fibonacci number generators: Named Events

```
module fibNumCalc(startingValue, fibNum);
    input [15:0] startingValue;
    output [15:0] fibNum;
    reg [15:0] count, fibNum, oldNum, temp;

    always
        begin
            @ng.ready          //wait for event signal
            count = startingValue;
            oldNum = 1;
            for (fibNum = 0; count != 0; count = count - 1)
                begin
                    temp = fibNum;
                    fibNum = fibNum + oldNum;
                    oldNum = temp;
                end
            $display ("%d, fibNum=%d", $time, fibNum);
        end
    endmodule
```

# VHDL-Fibonacci number generators

```
-- Fib.vhd
--
-- Fibonacci number sequence generator
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Fibonacci is port (
    Reset : in std_logic;
    Clock : in std_logic;
    Number : out unsigned(31 downto 0) );
end entity Fibonacci;
```

# VHDL-Fibonacci number generators

architecture Rcingham of Fibonacci is

```
signal Previous : natural;
signal Current : natural;
signal Next_Fib : natural;
begin
    Adder:
        Next_Fib <= Current + Previous;
    Registers:
        process (Clock, Reset) is
            begin
                if Reset = '1' then
                    Previous <= 1;
                    Current <= 1;
                elsif
                    rising_edge(Clock) then
                        Previous <= Current;
                        Current <= Next_Fib;
                    end if;
            end process Registers;
        Number <= to_unsigned(Previous, 32);
end architecture Rcingham;
```

# Wait a moment ....

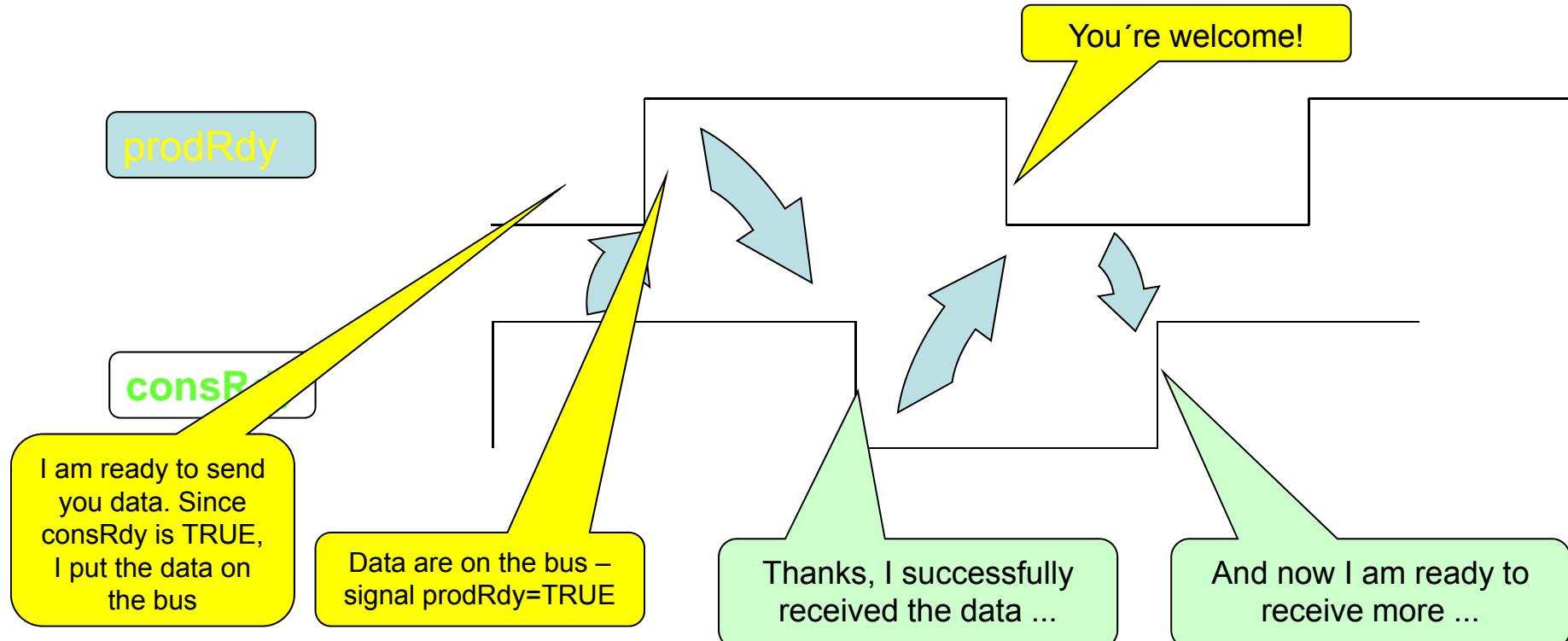
- The wait statement is a level sensitive statement. The execution of a process stops, if the value of the conditional is FALSE. It continues, if the value evaluates to TRUE:

```
module consumer (input [7:0] dataIn, input ready);

reg [7:0] in;
always
begin
    wait(ready);
    in = dataIn;
    // do something smart here ...
end
endmodule
```

# Consumer/Producer Handshake

- The previous example may lead to problems, since the consumer never signals to the producer that it successfully accepted the data. A consumer/producer handshake mechanism overcomes this deficiency:



# Consumer/Producer Handshake

- Here we go with the code:

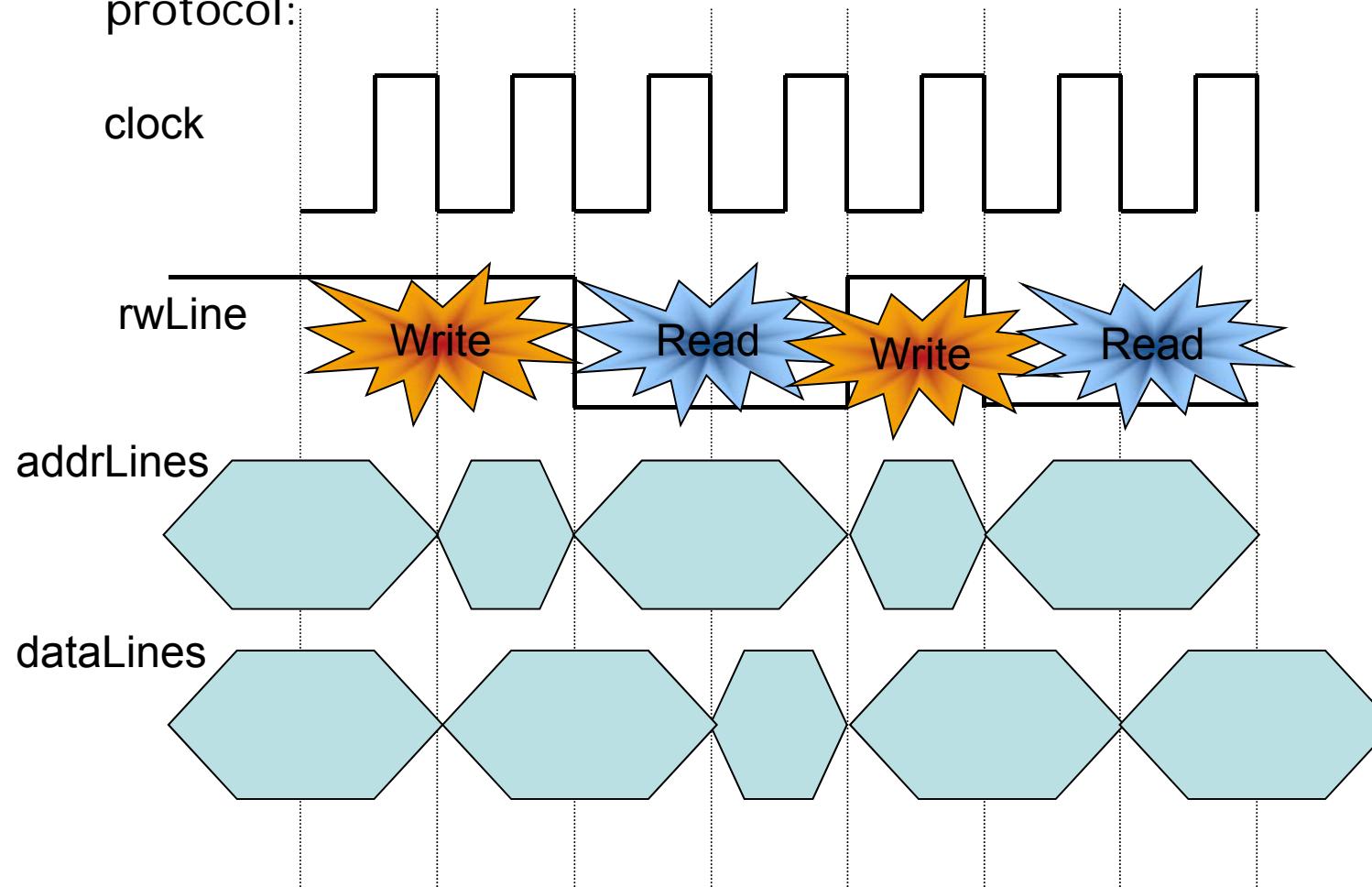
```
module ProducerConsumer;
  reg consReady, prodReady;
  reg [7:0] dataInCopy, dataOut;
  always // The consumer process
  begin
    consReady = 1; /* indicate
                     consumer ready */
  forever
    begin
      wait (prodReady)
      dataInCopy = dataOut;
      consReady = 0; /* indicate
                      value consumed */
    //... Do some smart stuff here ...
    wait (!prodReady)
      // complete handshake
    consReady = 1;
  end
end

always // The producer process
begin
  prodReady = 0; /* indicate
                  nothing to transfer */
forever
  begin
    /* ...produce data and put into
       "dataOut" */
    wait (consReady) /* wait for
                      consumer ready */
    dataOut = $random;
    prodReady = 1; /*indicate
                    ready to transfer */
    wait (!consReady) /* finish
                       handshake */
    prodReady = 0;
  end
end
endmodule
```

# A concurrent process example

## (Memory Controller)

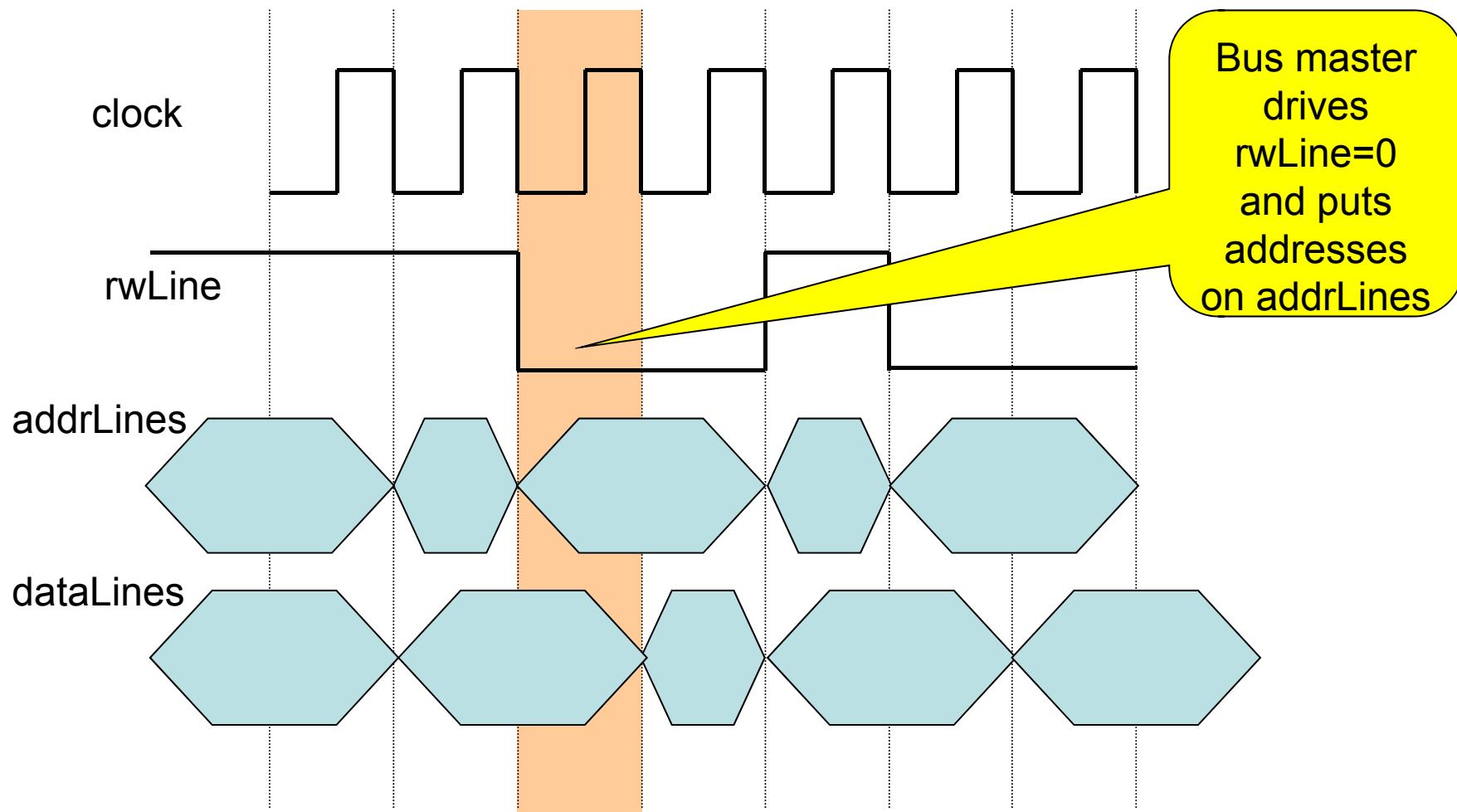
- Let's continue with a more complex example, a synchronous bus protocol:



# A concurrent process example

## (Memory Controller)

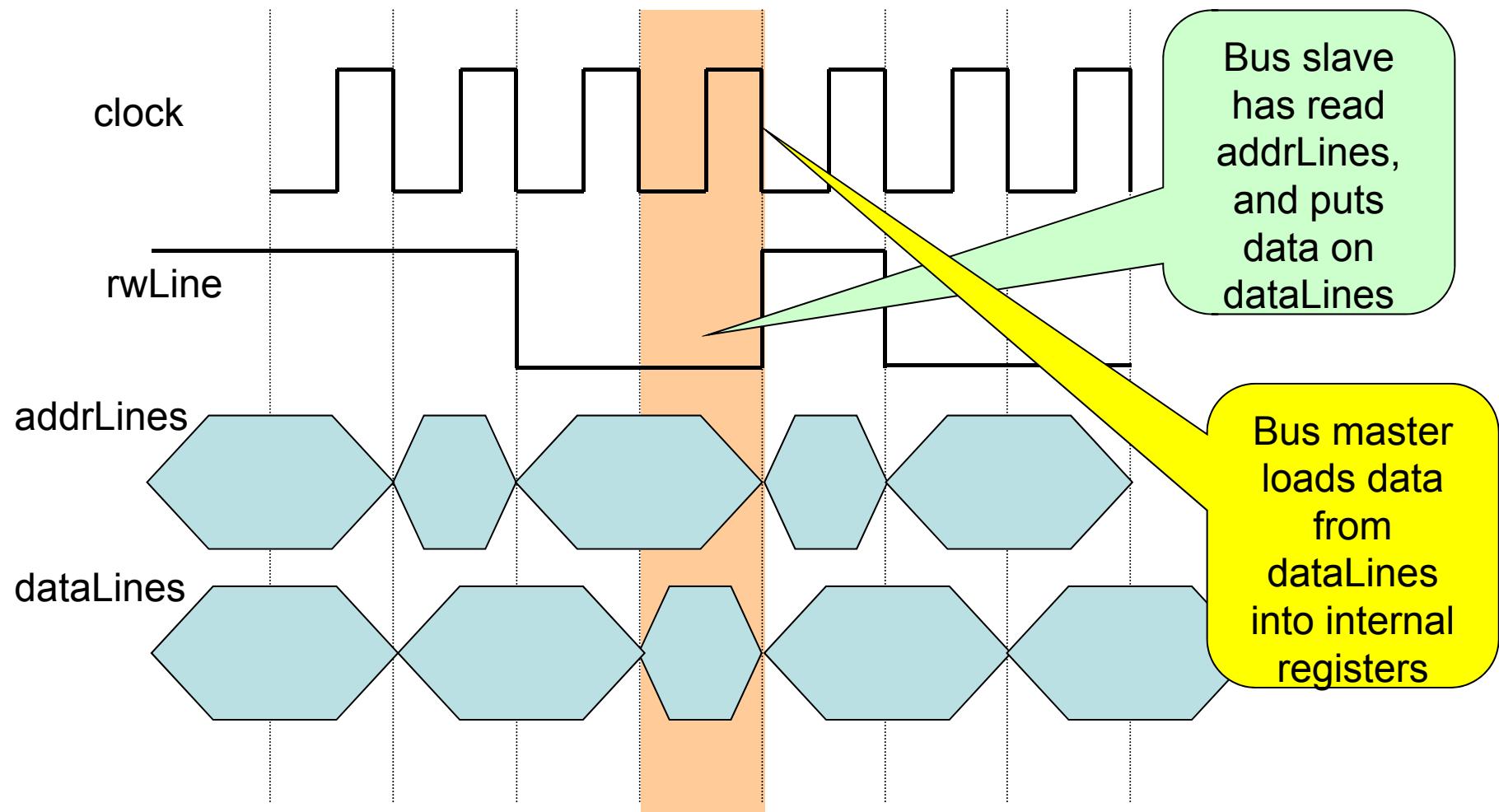
- A read cycle takes two clock periods to complete:



# A concurrent process example

## (Memory Controller)

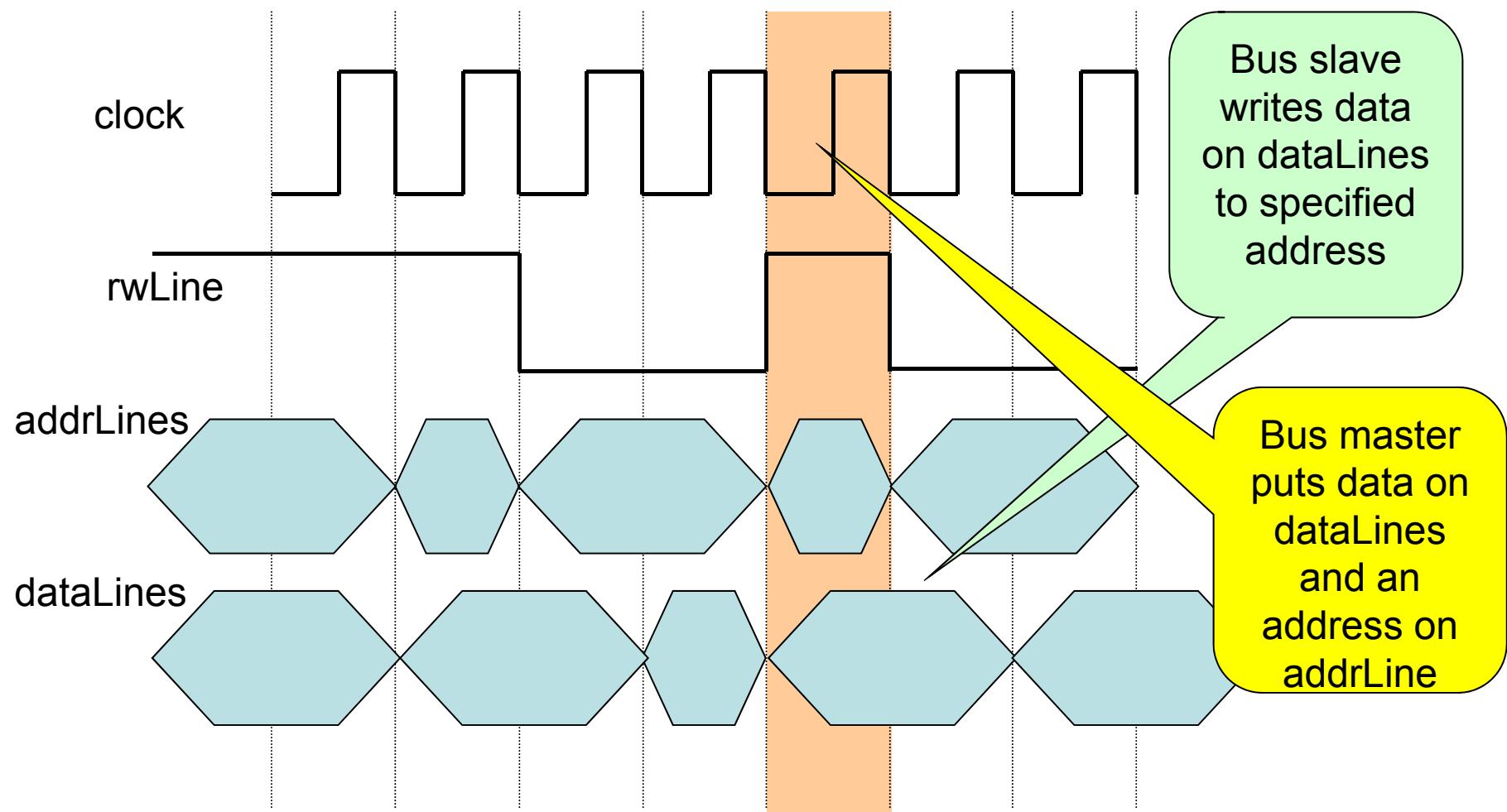
- A read cycle takes two clock periods to complete:



# A concurrent process example

## (Memory Controller)

- A write cycle takes one clock period to complete:



# Memory Controller

```
`define READ 0
`define WRITE 1

module sbus;
    parameter tClock = 20;
    reg clock;
    reg[15:0] m[0:31];
        //32 16-bit words
    reg[15:0] data;
    // registers names xLine
    // model the bus lines using
    // global registers
    reg rwLine; //write = 1, read = 0
    reg [4:0] addressLines;
    reg [15:0] dataLines;

initial
begin
$readmemh ("memory.data", m);
clock = 0;
$monitor ("rw=%d, data=%d, addr=%d
at time %d",
rwLine, dataLines, addressLines,
$time);
end
```

# Memory Controller

```
always
#tClock clock = !clock;

initial // bus master end
begin
#1
wiggleBusLines ( `READ, 2, data);
wiggleBusLines ( `READ, 3, data);
data = 5;
wiggleBusLines ( `WRITE, 2, data);
data = 7;
wiggleBusLines ( `WRITE, 3, data);
wiggleBusLines ( `READ, 2, data);
wiggleBusLines ( `READ, 3, data);
$finish;
end
```

```
task wiggleBusLines
(input readWrite,
input [5:0] addr,
inout [15:0] data);

begin
rwLine <= readWrite;
if (readWrite) begin
// write value
addressLines <= addr;
dataLines <= data;
end
else begin //read value
addressLines <= addr;
@(negedge clock);
end
@(negedge clock);
if (~readWrite)
data <= dataLines;
// value returned during read cycle
end
endtask
```

# Memory Controller

```
always // bus slave end
begin
    @(negedge clock);
    if (~rwLine) begin //read
        dataLines <= m[addressLines];
        @(negedge clock);
        end
    end
    else //write
        m[addressLines] <= dataLines;
end
endmodule
```

# Advanced Hardware Modeling

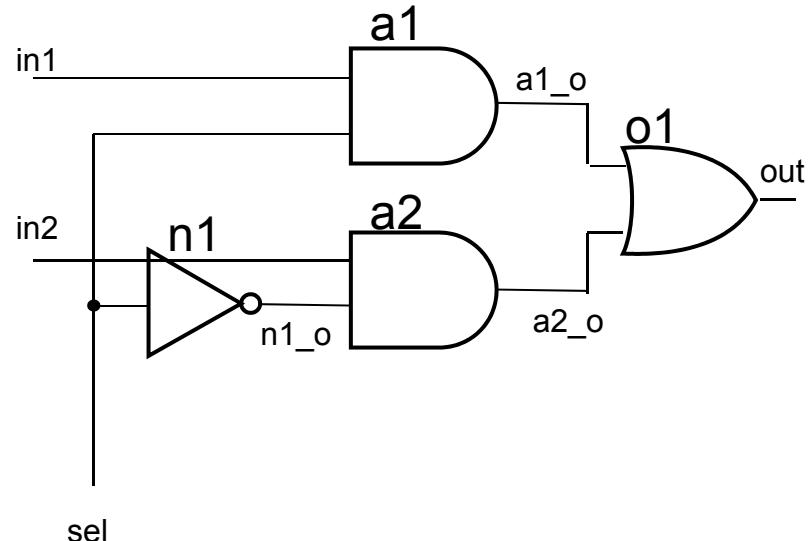


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Another deep look into our mux2

- We have already seen examples for hardware description using primitive logic gates

```
module mux2 (in1, in2, sel, out);  
    output out;  
    input in1, in2, sel;  
  
    and a1(a1_o, in1, sel);  
    not n1(n1_o, sel);  
    and a2(a2_o, in2, n1_o);  
    or o1(out, a1_o, a2_o);  
endmodule
```



- The question is now, which gate and switch level primitives are built into Verilog?

# Gate and switch level primitives

| N-input gates | N-output gates | Tristate gates | Pull gates | MOS switches | Bi-directional switches |
|---------------|----------------|----------------|------------|--------------|-------------------------|
| and           | buf            | bufif0         | pullup     | nmos         | tran                    |
| nand          | not            | bufif1         | pulldown   | pmos         | tranif0                 |
| nor           |                | notif0         |            | cmos         | tranif1                 |
| or            |                | notif1         |            | rmos         | rtran                   |
| xor           |                |                |            | rpmos        | rtranif0                |
| xnor          |                |                |            | rcmos        | rtranif1                |

# User defined primitives

- You may define your own primitives, such as:

```
primitive carry
```

```
(output carryOut,  
input carryIn, aIn, bIn);
```

```
table
```

|   |    |   |    |
|---|----|---|----|
| 0 | 00 | : | 0; |
| 0 | 01 | : | 0; |
| 0 | 10 | : | 0; |
| 0 | 11 | : | 1; |
| 1 | 00 | : | 0; |
| 1 | 01 | : | 1; |
| 1 | 10 | : | 1; |
| 1 | 11 | : | 1; |

```
endtable
```

```
endprimitive
```

# User defined primitives

- Also possible in a shorter way ...

```
primitive smart_carry
  (output carryOut,
   input carryIn, aIn, bIn);
```

```
table
```

|   |    |   |    |
|---|----|---|----|
| 0 | 0? | : | 0; |
| 0 | ?0 | : | 0; |
| ? | 00 | : | 0; |
| ? | 11 | : | 1; |
| 1 | ?1 | : | 1; |
| 1 | 1? | : | 1; |

```
endtable
```

```
endprimitive
```

# User defined primitives

- UDPs are not only limited to logical primitives:

```
primitive latch
```

```
(output reg q,  
input clock, data);
```

```
table
```

```
//   clock  data   state  output  
    0      1      : ? :  1;  
    0      0      : ? :  0;  
    1      ?      : ? :  -;
```

```
endtable
```

```
endprimitive
```

# User defined primitives

- Edge sensitive UDPs are also possible:

```
primitive dEdgeFF
  (output reg q,
   input clock, data);

  table
    //  clock  data   state  output
    (01)  0       : ?   : 0;
    (01)  1       : ?   : 1;
    (0x)  1       : 1   : 1;
    (0x)  0       : 0   : 0;
    (?0)  ?       : ?   : -;
    ?      (??)  : ?   : -;

  endtable
endprimitive
```

# User defined primitives

User defined primitives (UDP) have strict rules:

- Only exact one output port allowed, but multiple inputs
- The output port must be the first port to be listed
- All primitive ports are scalar – sorry, no vectors please
- Only logic values of 0, 1 and x can be treated on inputs and output.  
The z-value cannot be specified, but if put on an input, it is equivalent to x

# Shorthand notations for UDPs

| Symbol | Interpretation     | Comments                                                        |
|--------|--------------------|-----------------------------------------------------------------|
| 0      | Logic 0            |                                                                 |
| 1      | Logic 1            |                                                                 |
| x      | unknown            |                                                                 |
| ?      | Can be 0, 1, and x | Cannot be used in the output field                              |
| b      | Can be 0 and 1     | Cannot be used in the output field                              |
| -      | No change          | May only be given in the output field of a sequential primitive |

# Shorthand notations for UDPs

| Symbol | Interpretation                   | Comments                           |
|--------|----------------------------------|------------------------------------|
| (vw)   | Change from a value v to w       | v and w can be one of 0, 1, x or b |
| *      | Same as (??)                     | Any value change on input          |
| r      | Same as (01)                     |                                    |
| f      | Same as (10)                     |                                    |
| p      | Iteration of (01), (0x) and (x1) | Positive edge including x          |
| n      | Iteration of (10), (1x) and (x0) | Negative edge including x          |

# Modeling interconnect

- Wires (or nets or interconnect) are simply used to connect ports or signals. Usually they do not store values, but transmit values that are driven on them by structural elements such as gate outputs, assign statements and registers in a behavioral model.
- In Verilog, you can declare wires using the keyword wire, and also assign a delay with this interconnect:

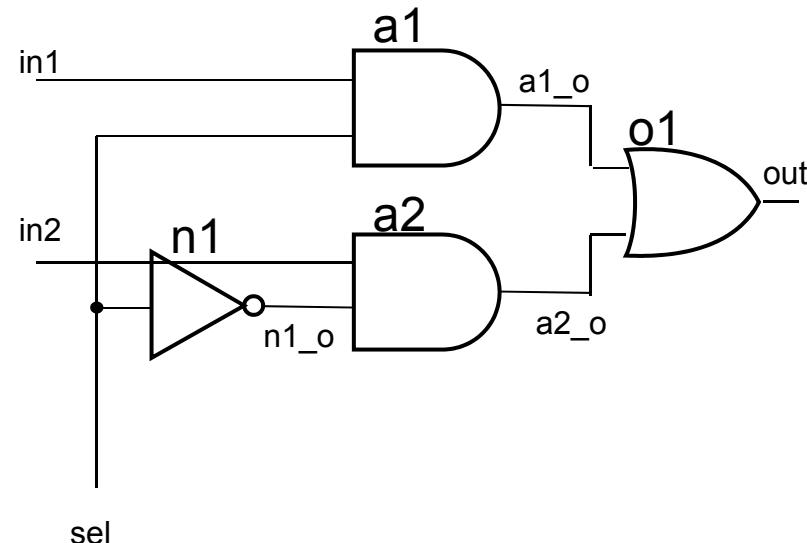
```
wire #3 x1; // a wire named x1 with a delay of 3
```

```
wire #(3,5) x2; // a wire named x2 with a rising  
// delay of 3, and a falling delay of 5
```

# Oh no, again mux2 ...

- Have a look again at mux2 – why did 't we have to declare a1\_o, a2\_o, n1\_o?

```
module mux2 (in1, in2, sel, out);  
    output out;  
    input in1, in2, sel;  
  
    and a1(a1_o, in1, sel);  
    not n1(n1_o, sel);  
    and a2(a2_o, in2, n1_o);  
    or o1(out, a1_o, a2_o);  
endmodule
```



- Wires can be declared implicitly. If an identifier appears in the port list of an instance of a gate primitive, module instantiation, or on the LHS of a continuous assignment, it will be implicitly declared as a net.

# More on nets ...

- By default, an implicitly declared net will be of type „wire“, and will have the default width of the connected module (or gate primitive port).
- You may change or even turn off this default by setting the compiler directive

```
default_nettype none // now an undeclared net will be
                     // flagged as an error
```

- Note that this is a fundamental difference between Verilog and VHDL – in VHDL you must declare everything that you use, Verilog is much more „get the work done“. Be aware, typing errors in port-names (e.g. O instead of o) do not automatically lead to an error in Verilog

# Net types

| Net Type                 | Modeling Usage                                                                                                              |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| wire and tri             | Used to model connections with no logic function. Only difference is the name. Use appropriate name for readability         |
| wand, wor, triand, trior | Used to model the wired logic function. Only difference between wire and tri version of the same logic function is the name |
| tri0, tri1               | Used to model connections with a resistive pull to the given supply                                                         |
| supply0, supply1         | Used to model the connection to the power supply                                                                            |
| trireg                   | Used to model charge storage on a net                                                                                       |

# Examples for wired logic functions

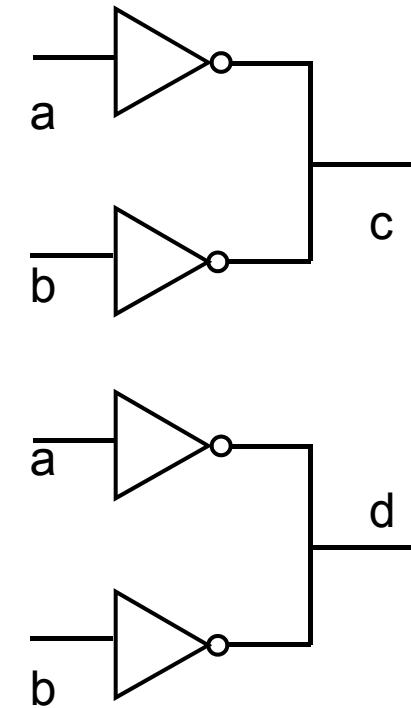
- With a net declared as `wand`, its output will be 0 if any of its drivers will be 0. Or, the output `c` will be 1 if both `a` and `b` are 0. Output `d` will be unknown if any input is unknown or `z`, or if `a = !b`

```
module wired_and_example
  (input a, b, output wand c, output d);

  not (c,a);
  not (c,b);

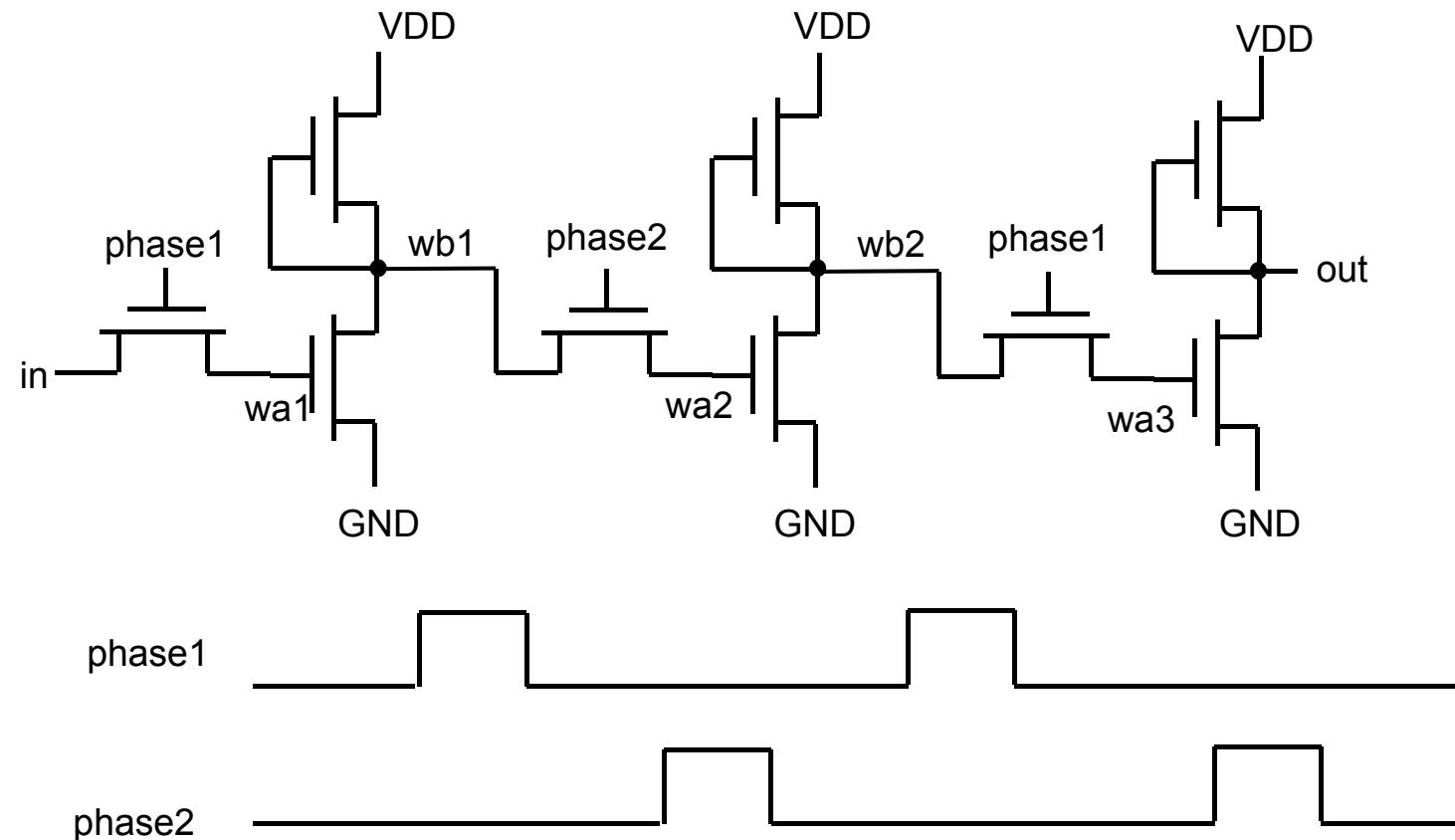
  not (d,a);
  not (d,b);

endmodule
```



# Switch level modeling

- Verilog enables you to even model hardware on the transistor level, but of course still only for digital signals. Have a look at the following MOS shift register:



# Switch Level Modeling

```
module shreg
/* IO port declarations, where 'out' is the inverse
   of 'in' controlled by the dual-phased clock */
(output tri out, //shift register output
 input in, //shift register input
 phase1, //clocks
 phase2);

tri wb1, wb2; //tri nets pulled up to VDD
pullup (wb1), (wb2), (out); //depletion mode pullup devices
trireg (medium) wa1, wa2, wa3; //charge storage nodes
supply0 gnd; //ground supply

nmos #3 //pass devices and their interconnections
      a1(wa1, in, phase1),    b1(wb1, gnd, wa1),
      a2(wa2, wb1, phase2),    b2(wb2, gnd, wa2),
      a3(wa3, wb2, phase1),    gout(out, gnd, wa3);
endmodule
```

# Switch Level Modeling

```
module waveShReg;  
    wire shiftout; //net to receive circuit output value  
    reg shiftin; //register to drive value into circuit  
    reg phase1, phase2; //clock driving values  
    parameter d = 100; //define the waveform time step  
  
    shreg cct (shiftout, shiftin, phase1, phase2);  
  
initial  
begin :main  
    shiftin = 0; //initialize waveform input stimulus  
    phase1 = 0;  
    phase2 = 0;  
  
    setmon; // setup the monitoring information
```

# Switch Level Modeling

```
repeat(2) //shift data in
    clockcct;
end

task setmon; //display header and setup monitoring
begin
    $display(" time clks in out wa1-3 wb1-2");
    $monitor ($time,,,phase1, phase2,,,,,shiftin,,, shiftout,,,
              cct.wa1, cct.wa2, cct.wa3,,,cct.wb1, cct.wb2);
end
endtask

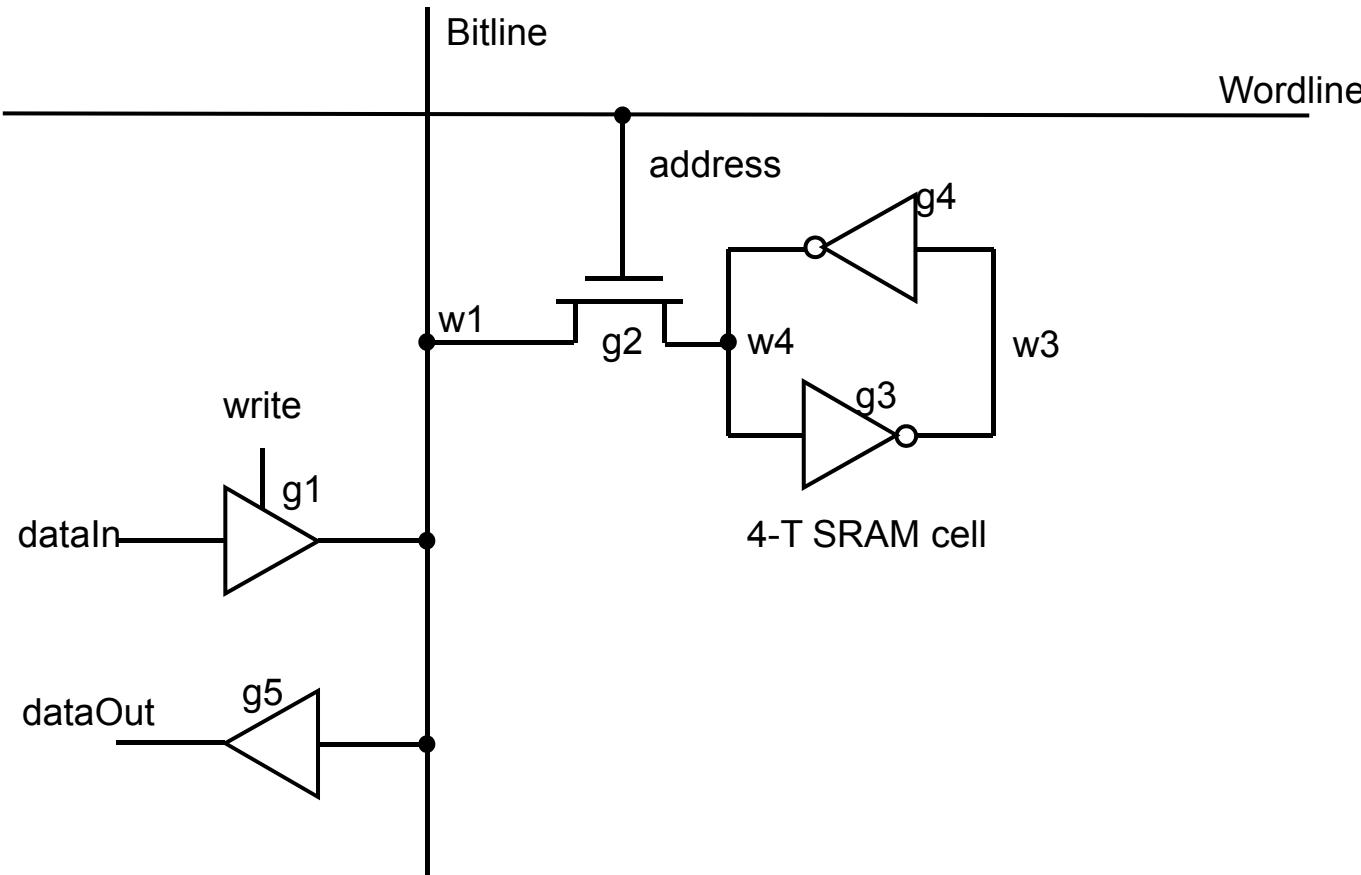
task clockcct; //produce dual-phased clock pulse
begin
    #d phase1 = 1; //time step defined by parameter d
    #d phase1 = 0;
    #d phase2 = 1;
    #d phase2 = 0;
end
endtask
```

# Result of simulation

| time | clks | in | out | wa1-3 | wb1-2 |
|------|------|----|-----|-------|-------|
| 0    | 00   | 0  | x   | xxx   | xx    |
| 100  | 10   | 0  | x   | xxx   | xx    |
| 103  | 10   | 0  | x   | 0xx   | xx    |
| 106  | 10   | 0  | x   | 0xx   | 1x    |
| 200  | 00   | 0  | x   | 0xx   | 1x    |
| 300  | 01   | 0  | x   | 0xx   | 1x    |
| 303  | 01   | 0  | x   | 01x   | 1x    |
| 306  | 01   | 0  | x   | 01x   | 10    |
| 400  | 00   | 0  | x   | 01x   | 10    |
| 500  | 10   | 0  | x   | 01x   | 10    |
| 503  | 10   | 0  | x   | 010   | 10    |
| 506  | 10   | 0  | 1   | 010   | 10    |
| 600  | 00   | 0  | 1   | 010   | 10    |
| 700  | 01   | 0  | 1   | 010   | 10    |

# Strength definitions

- Sometimes hardware is designed to overwrite signals by stronger drivers. Verilog enables you to model this. Have a look at the following SRAM example:



# SRAM cell: here `s the code

```
module sram
(output dataOut,
input address, dataIn, write);
tri w1, w3, w4;

bufif1 g1(w1, dataIn, write);
tranif1 g2(w4, w1, address);
not (pull0, pull1) g3(w3, w4), g4(w4, w3);
buf g5(dataOut, w1);
endmodule
```

# SRAM cell: here `s the code

```
module wave_sram //waveform for testing the static RAM cell
#(parameter d = 100);
    wire dataOut;
    reg address, dataIn, write;

    sram cell (dataOut, address, dataIn, write);

initial begin
    #d dis;
    #d address = 1;      #d dis;
    #d dataIn = 1;       #d dis;
    #d write = 1;         #d dis;
    #d write = 0;         #d dis;
    #d write = 'bx;       #d dis;
    #d address = 'bx;     #d dis;
    #d address = 1;       #d dis;
    #d write = 0;         #d dis;
end
```

# SRAM cell: here `s the code

```
task dis; //display the circuit state
    $display($time,, "addr=%v d_In=%v write=%v d_out=%v",
             address, dataIn, write, dataOut,
             " (134)=%b%b%b", cell.w1, cell.w3, cell.w4,
             " w134=%v %v %v", cell.w1, cell.w3, cell.w4);
endtask
endmodule
```

# Result of simulation

| time | addr | d_in | wr | d_out | 134 | Comment          |
|------|------|------|----|-------|-----|------------------|
| 100  | x    | x    | x  | x     | xxx |                  |
| 300  | 1    | x    | x  | x     | xxx |                  |
| 500  | 1    | 1    | x  | x     | xxx |                  |
| 700  | 1    | 1    | 1  | 1     | 101 | Write function   |
| 900  | 1    | 1    | 0  | 1     | 101 | Read function    |
| 1100 | 1    | 1    | x  | 1     | 101 |                  |
| 1300 | x    | 1    | x  | x     | x01 | SRAM holds value |
| 1500 | 1    | 1    | x  | 1     | 101 |                  |
| 1700 | 1    | 1    | 0  | 1     | 101 | Read function    |

# Strength definition

| Strength name    | Strength level | Element modeled                         | Declaration abbreviation | Printed abbreviation |
|------------------|----------------|-----------------------------------------|--------------------------|----------------------|
| Supply Drive     | 7              | Power supply                            | supply                   | Su                   |
| Strong Drive     | 6              | Default gate and assign output strength | strong                   | St                   |
| Pull Drive       | 5              | Gate and assign output strength         | pull                     | Pu                   |
| Large Capacitor  | 4              | Size of trireg net capacitor            | large                    | La                   |
| Weak Drive       | 3              | Gate and assign output statement        | weak                     | We                   |
| Medium Capacitor | 2              | Size of trireg net capacitor            | medium                   | Me                   |
| Small Capacitor  | 1              | Size of trireg net capacitor            | small                    | Sm                   |
| High Impedance   | 0              | Not applicable                          | highz                    | Hi                   |

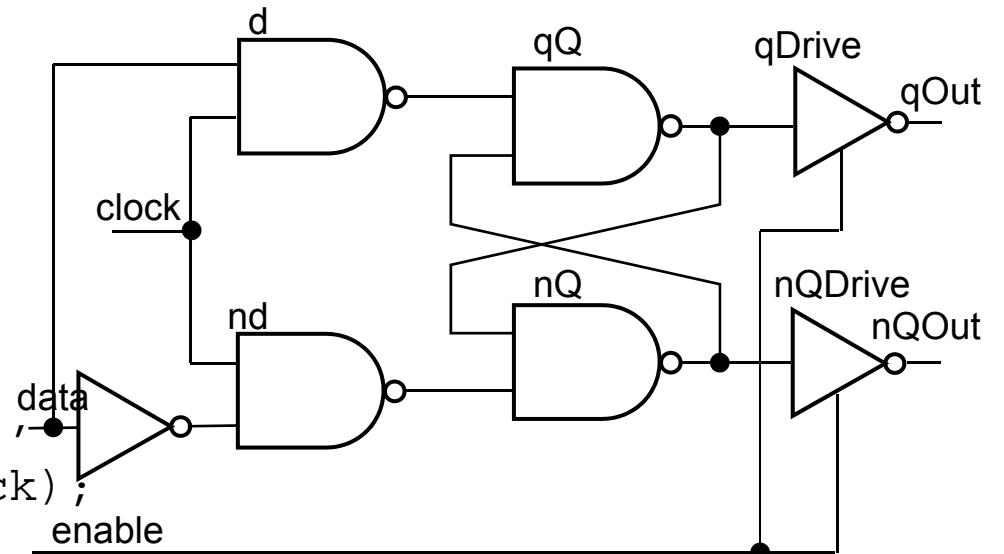
# Logic delay modeling

- The realistic delay of a circuit can only be modeled if each component can be assigned a realistic logic delay. Verilog offers a wide variation of options to model hardware to a so-called „sign-off“ point.

```
module triStateLatch
  (output qOut, nQOut,
   input clock, data, enable);
  tri qOut, nQOut;

  not #5 (ndata, data);
  nand #(3,5) d(wa, data, clock), nd(wb, ndata, clock);
  nand #(12, 15) qQ(q, nq, wa),
                 nQ(nq, q, wb);
  bufif1 #(3, 7, 13) qDrive (qOut, q, enable),
                     nQDrive(nQOut, nq, enable);

endmodule
```



# Specifying time units

- We have used the time units up to now without knowing the meaning of it. There is a compiler directive `timescale which specifies the time unit and the time precision

```
`timescale 10ns / 1ns
```

```
#7 // The delay is now 7*time_unit = 70ns
    // Time is maintained to the scale of 1ns
#7.748 // The delay is 77ns
```

```
`timescale 10ns / 10ns
#7.5 // The delay is rounded to 80ns
```

- You may use s for seconds, ms for milliseconds, us for microseconds, ns for nanoseconds, ps for picoseconds and fs for femtoseconds

# Module Hierarchy

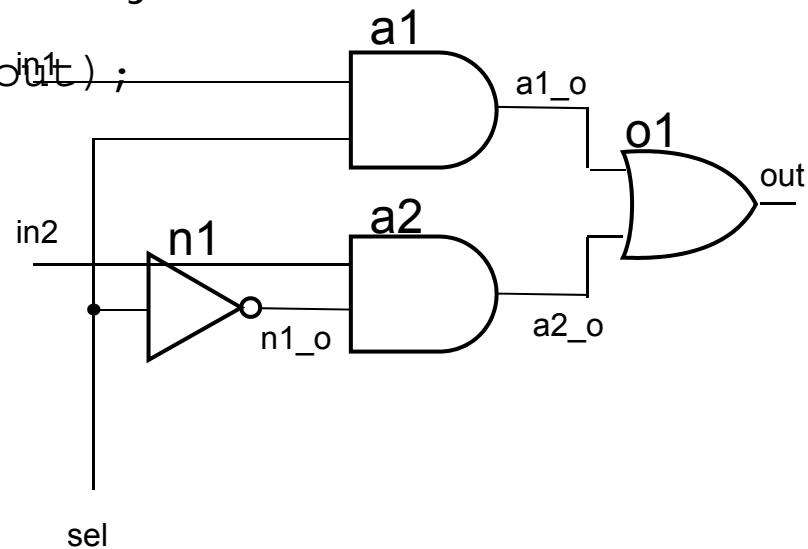


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# A deeper look into Modules

- The port of a module are a link to the outside world.
- The input port specifies the internal name for a vector or scalar driven by an external entity
- An output port specifies the internal name for a vector or scalar driven by an internal entity.
- An inout port specifies the internal name for a vector or scalar driven either by an internal or external entity

```
module mux2 (in1, in2, sel, out1out);  
  output out;  
  input in1, in2, sel;  
  
  and a1(a1_o, in1, sel);  
  not n1(n1_o, sel);  
  and a2(a2_o, in2, n1_o);  
  or o1(out, a1_o, a2_o);  
endmodule
```



# Connecting Modules

- Using a module instance in a higher level module can be done using the following way (see below). Note that a, b, my\_sel and out must be declared as wires
- Using the period („.“) indicates that the port name as defined in the module is connected to the wire name in the brackets.

```
mux2 m1 (a, b, my_sel, out);
```

```
mux2 m2a (.a(in1), .b(in2), .my_sel(sel), .out(out));
```

```
mux2 m2b (.out(out), .a(in1), .b(in2), .my_sel(sel));
```

# Module Parameters

- Parameters are an indispensable help for any kind of subprogram or module. Example of a somehow tedious instantiation of eight xors:

```
module xor8 (output [1:8] xout, input [1:8] xin1, xin2);  
  
    xor    (xout[8], xin1[8], xin2[8]),  
            (xout[7], xin1[7], xin2[7]),  
            (xout[6], xin1[6], xin2[6]),  
            (xout[5], xin1[5], xin2[5]),  
            (xout[4], xin1[4], xin2[4]),  
            (xout[3], xin1[3], xin2[3]),  
            (xout[2], xin1[2], xin2[2]),  
            (xout[1], xin1[1], xin2[1]);  
  
endmodule
```

# Module Parameters

- Can be done much smarter:

```
module xorx
    #(parameter width = 8,
      delay = 10)
    (output [1:width] xout, input [1:width] xin1, xin2);

    assign #(delay) xout = xin1 ^ xin2;

endmodule
```

- Module parameters can be anything: signed, sized (with a range), integer, real, realtime and time.

# Local Module Parameters

- Local module parameter values can be overridden during the instantiation of the module itself.

```
module toplevel_xorx
  (output [3:0] a1, a2);

  reg [3:0]   b1, c1, b2, c2;

  xorx #(4,15) a(a1, b1, c1),
           b(a2, b2, c2);
endmodule
```

- During the instantiation with new parameters the local parameters for width and delay (8 and 10) are overridden with 4 and 15

# Local Module Parameters

- If you want to be sure which parameter overwrites which local one, you might explicitly name the parameter names:

```
module toplevel_xorx
  (output [3:0] a1, a2);

  reg [3:0]   b1, c1, b2, c2;

  xorx #( .width(4), .delay(15) ) a(a1, b1, c1),
                b(a2, b2, c2);
endmodule
```

# More Overwriting...

- You might also use the defparam way of overriding parameters.  
We use again the module definition for xorx, but now we change toplevel\_xor:

```
module toplevel_xorx_v2
    (output [3:0]  a1, a2);

    reg[3:0]      b1, c1, b2, c2;

    xorx      a(a1, b1, c1),
              b(a2, b2, c2);

endmodule

module annotate
    defparam
        toplevel_xorx_v2.a.width = 4;
        toplevel_xorx_v2.b.width = 4;
        toplevel_xorx_v2.b.delay = 15;
endmodule
```

# Arrays of Instances

- The previous way of instantiating eight xors was quite cumbersome.... A nicer way is:

```
module xor8 (output [1:8] xout, input [1:8] xin1, xin2);  
  
xor a[1:8] (xout, xin1, xin2);  
  
endmodule
```

# Arrays of Instances

- Defining register banks is now quite easy, even using scalar inputs for all dff:

```
module register_bank
  (output [7:0] q,
   input [7:0] d,
   input clock, clear);

  dff  r[7:0] (q, d, clock, clear);

endmodule
```

# Generate Blocks

- Arrays of instances are only limited to quite simple repetitive structures. Much more power has the generate command:

```
module xorGen
  #(parameter width = 4,
    delay = 10)
  (output [1:width] xout,
   input [1:width] xin1, xin2);

  generate
    genvar i;
    for(i=1; i<=width; i=i+1)begin :xi
      assign #delay xout[i] = xin1[i] ^ xin2[i];
    end
  endgenerate
endmodule
```

# A complex example: n-bit adder

- Consider modeling an n-bit adder (with  $n > 1$ ) that also has condition code outputs to indicate if the result was negative, produced a carry, or produced a 2's complement overflow.
- In this case, not all generated instances of the adder are connected the same
- You may then use if-then-else and case-statements in the for-loop to generate these differences.

# Backtrack: Theory of Adders

## Basic Adder Cells

- Half Adder:

- Can be used to calculate the sum of two bits  $A_1$  and  $A_2$ .

$$C = A_1 A_2$$

$$S = A_1 \oplus A_2$$

- Full Adder:

$$C_{out} = C_{in} (A_1 + A_2) + A_1 A_2$$

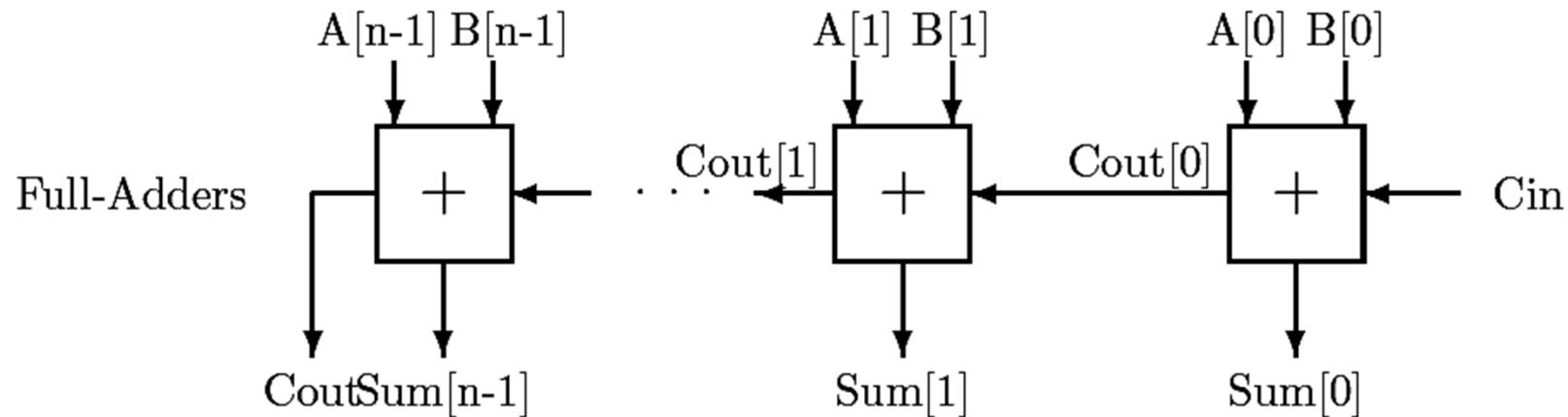
$$S_{out} = A_1 \oplus A_2 \oplus C_{in}$$

- For adding binary numbers having a bitwidth of more than one single bit.
  - These equations can be realized either by logic gates (AND, OR, XOR) or by two half-adders and an OR gate.

# Adders / Subtractors for Binary Coded Integers

## Parallel Adders

- Ripple Carry Adder:



- Chained full-adders where the carry „ripples“ through the whole chain from the LSB to the MSB.
- The addition time depends on the wordlength of the operands.

# A complex example: n-bit adder

```
module adderWithConditionCodes
  #(parameter width = 1)
  (output reg [width-1:0] sum,
   output reg cOut, neg, overFlow,
   input  [width-1:0] a, b,
   input  cIn);

  reg [width -1:0] c;
```

# A complex example: n-bit adder

```
generate
    genvar i;
    for (i = 0; i <= width-1; i=i+1) begin: stage
        case(i)
            0: begin
                always @(*) begin
                    sum[i] = a[i] ^ b[i] ^ cIn;
                    c[i] = a[i] & b[i] | b[i]& cIn | a[i] & cIn;
                end
            end
        end
```

# A complex example: n-bit adder

```
width-1: begin
    always @(*) begin
        sum[i] = a[i] ^ b[i] ^ c[i-1];
        cOut = a[i]&b[i] | b[i]&c[i-1] | a[i] & c[i-1];
        neg = sum[i];
        overFlow = cOut ^ c[i-1];
    end
end
default: begin
    always @(*) begin
        sum[i] = a[i] ^ b[i] ^ c[i-1];
        c[i] = a[i]&b[i] | b[i] & c[i-1] | a[i] & c[i-1];
    end
end
endcase
end
endgenerate
endmodule
```

# Advanced Timing



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Verilog Timing Model

- Welcome to the black-belt section of this Verilog lecture!
- Let´s have a look on how the Verilog timing models really works.
- Using Gate level primitives, the complete model is always sensitive to input changes. In other words: inputs are always evaluated and determine the output to change.
- Any input change at *any* time will cause the gate instance to execute the evaluation of its output.

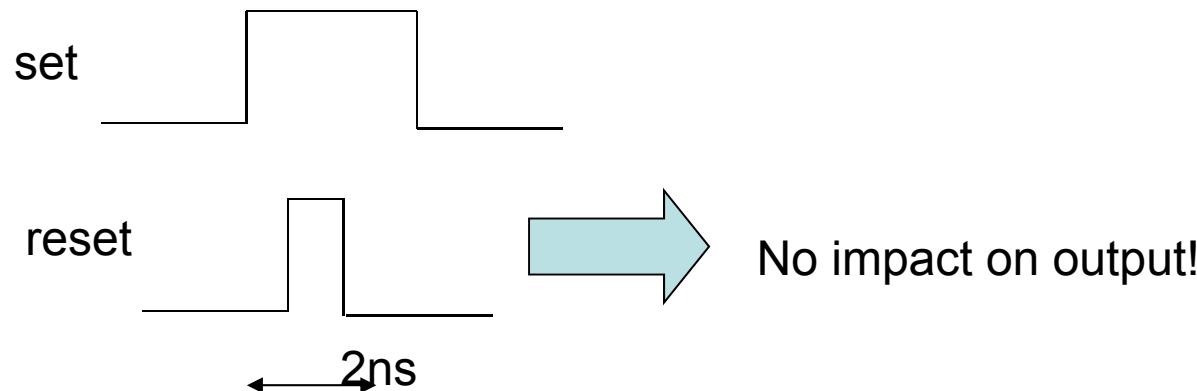
```
module nandLatch (output q, qBar,
    input set, reset );

    nand #2 (q, qBar, set),
        (qBar, q, reset);

endmodule
```

# Verilog Timing Model

- Let's assume that a scheduled event in our gate level timing example has not yet been executed (the model was still busy executing a scheduled event – e.g. due to the 2 timeunits delay). If now a new event is generated for the output of that element, the previously scheduled event will be cancelled and the new one will be put in the event queue instead.



- If a pulse is shorter than the propagation time at a gates input, the output of the gate will not change
- Inertial delay is the minimum time a set of inputs must be present for a change in the output to be seen
- Verilog gate models have – by definition – inertial delays just greater than their propagation delay – so watch out!

# Verilog Timing Model

- Using behavioral models, you have to specify the sensitivity list yourself
- The sensitivities are context dependent – you decide to which input your model is sensitive!
- The model below is only sensitive to clockedges which are more than 5 timeunits apart

```
module dff (output reg q,
             input d, clock );
    always @ (posedge clock)
        #5 q <= d;
endmodule
```

# Verilog Timing Model

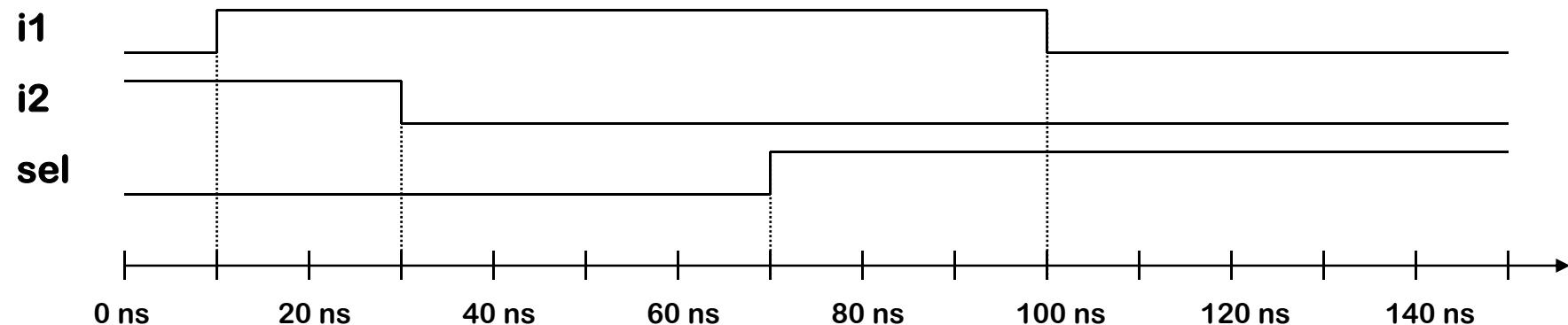
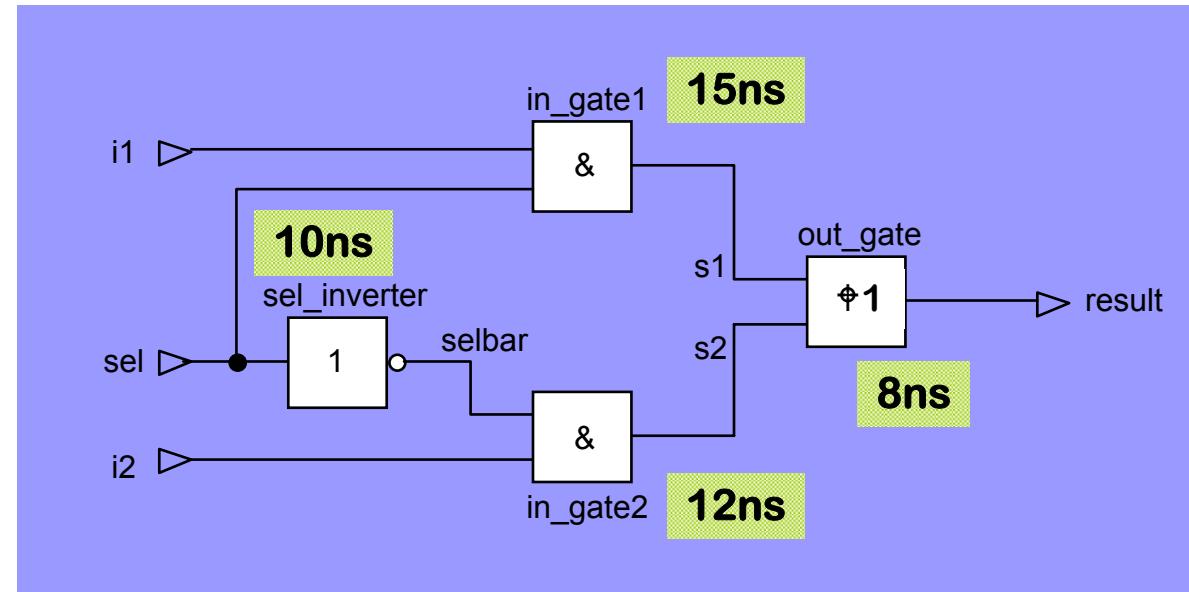
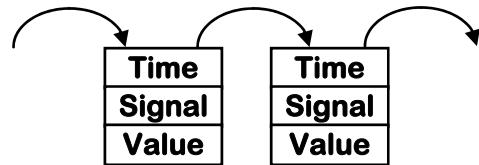
- The procedural timing model of Verilog does not cancel events in the event queue.
- If there are multiple events scheduled for the same time, the execution order is indeterminate. Therefore you must avoid writing such bad code:

```
moduledff (output reg q,  
           input d1, d2, clock );  
  
  always @ (posedge clock)  
    begin  
      q <= d1;  
      q <= d2;  
    end  
endmodule
```



# Event queue

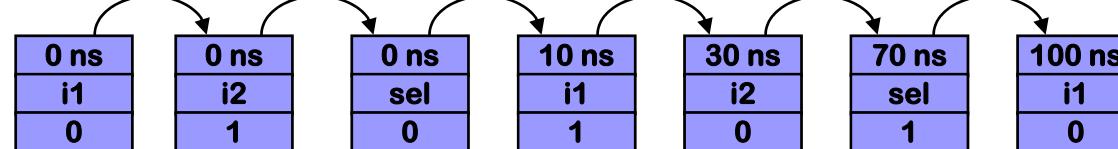
- Event Queue Example



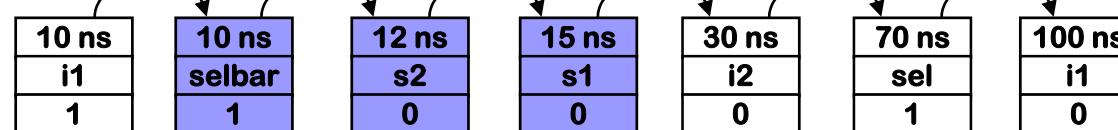
# Event queue

- Event Queue Example (cont.)

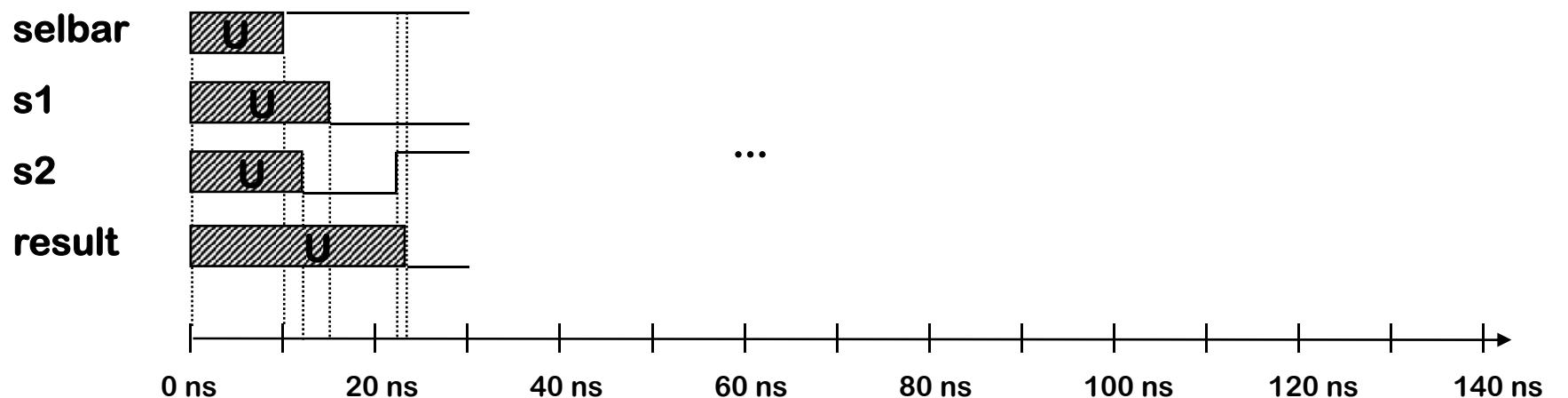
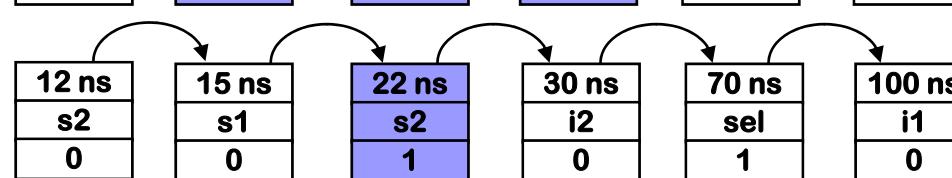
- Event queue before Initialization



- Event queue for t = 0 ns

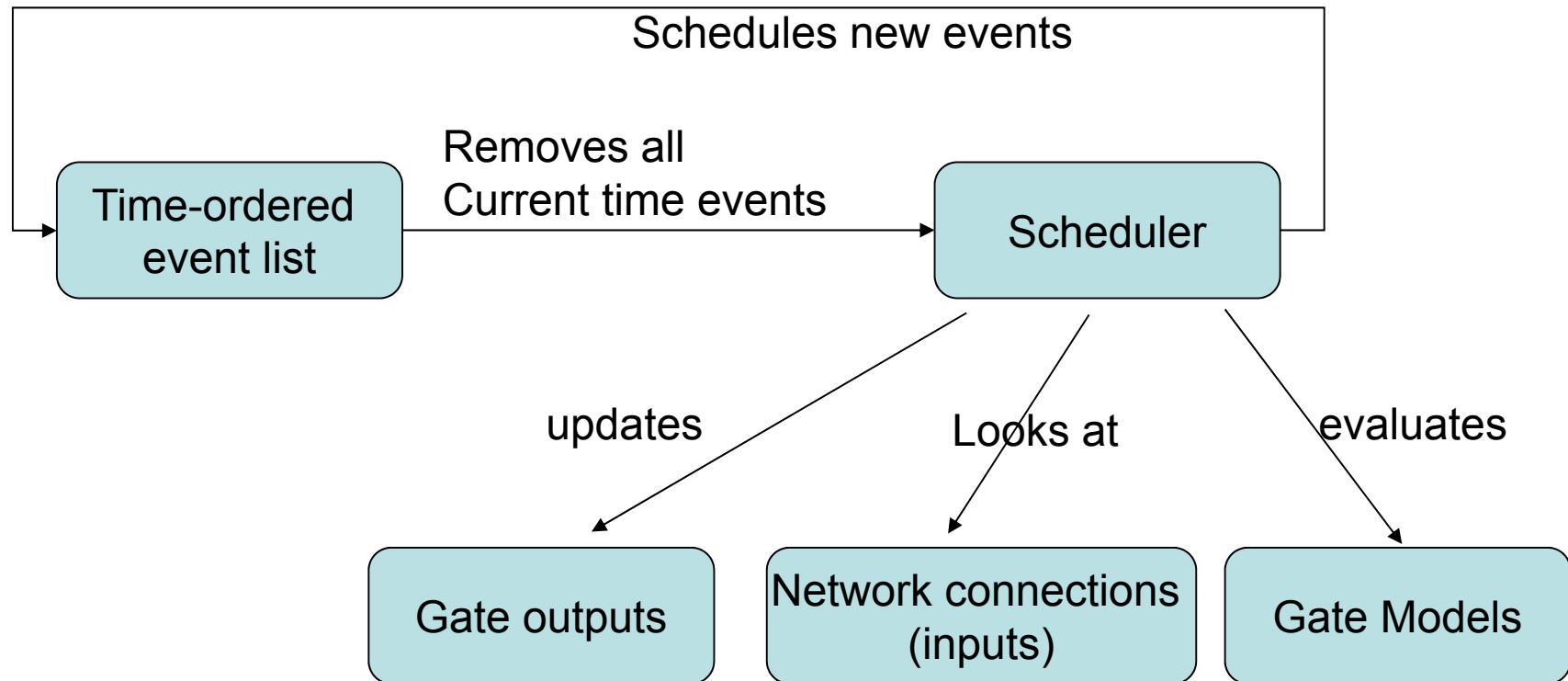


- Event queue for t = 10 ns

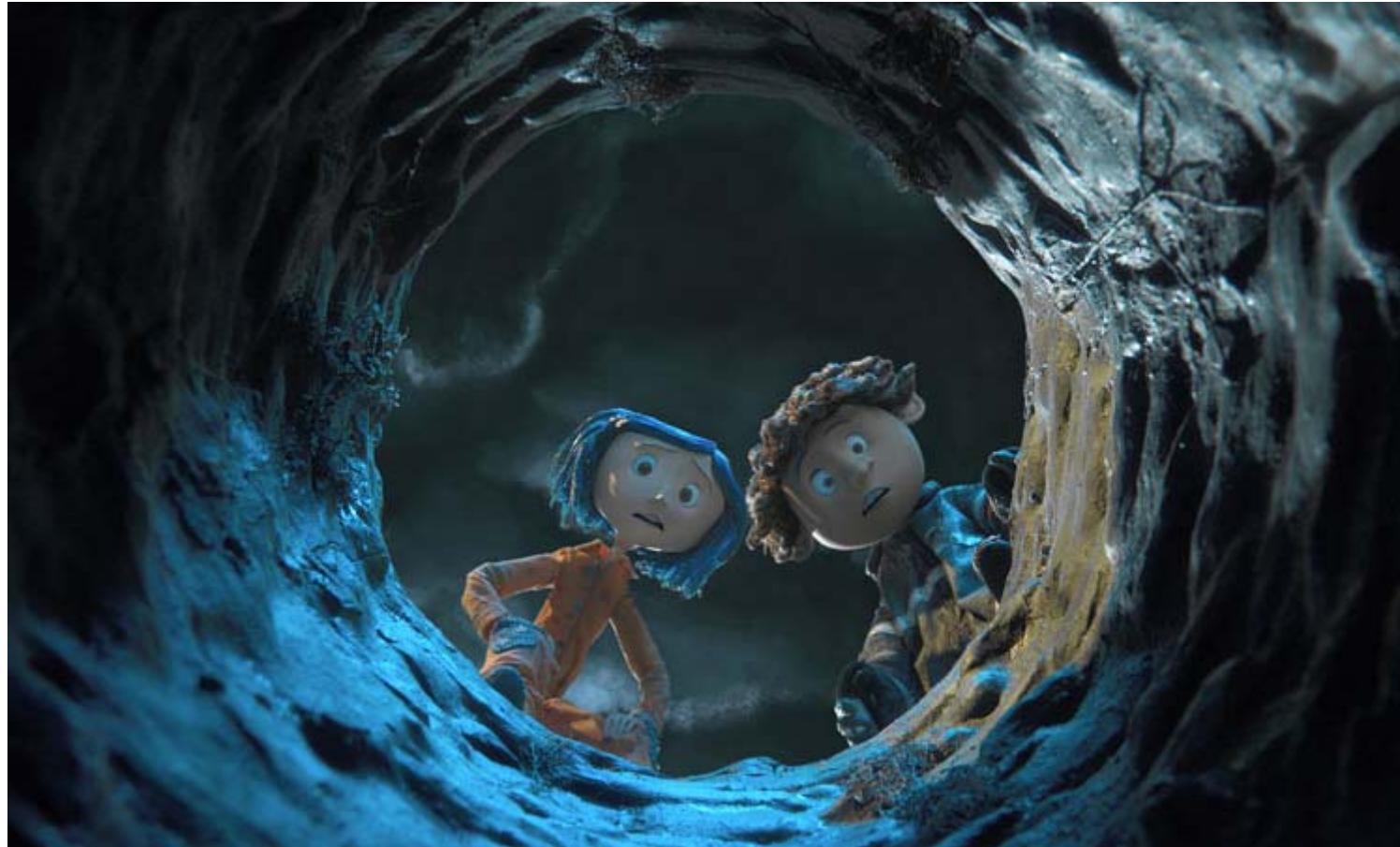


# Event driven simulator

- The following figure shows the basic elements of an event driven simulator



# A view to a black hole: $\leq$ and $=$



# A view to a black hole

- Consider again block and nonblocking assignment below:
- In isolation,  $a = b$  and  $a \leq b$  will perform the same function – they will assign the value currently in  $b$  to the register  $a$
- This even holds true for the statements #3  $a = b$ ; and #3  $a \leq b$ ;
- The difference is how the assignment is made, and what consequences it has on other variable assignments
  
- Look at  $a = \#4 b$ ; and  $a \leq \#4 b$ ;
- The first statement is identical to
  - $bTemp = b;$
  - $\#4 a = bTemp;$
- The second statement calculates the value of  $b$ , schedules an update event at 4 time units in the future, and continues executing the process in the current time

# A view to a black hole

- Another example on the difference (assume initially  $b=1$ ,  $a=0$ ):

```
begin
    a = #2 b;
    c = #2 a;
end
```

```
begin
    a <= #2 b;
    c <= #2 a;
end
```

- for the left example, b is taken, stored into a temporary variable, and delays for 2 timeunits the update event for a. When this update event is executed ( $a = 1$ ), the process continues. Same for c, so c gets the value 1 after 4 timeunits.
- for the right example, in the first line b is evaluated, an event is scheduled 2 timeunits in the future, and the process continues. Another event (for c) is scheduled also 2 timeunits in the future. Therefore, c is 0 after 2 timeunits.

# How the event scheduler deals with $<=$ and $=$

- In fact, the event scheduler deals differently with „regular“ and non-blocking events. The following table tries to shed some light onto this:

|           |                                                                                                                                                                                                                                                                              |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $a = b;$  | b is calculated and used immediately to update a. Note that the next statement in the behavioral process that uses a will use this new value. If a is an output of the process, elements on a´s fanout list are scheduled in the current time as a regular evaluation event. |
| $a <= b;$ | b is calculated and a non-blocking update event is scheduled for a during the current time. Execution of the process continues. This new value for a will not be seen by other elements, until the non-blocking update event is executed                                     |

# How the event scheduler deals with $<=$ and $=$

|               |                                                                                                                                                                                                                                      |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $a = \#0 b;$  | b is calculated and an update event is scheduled as a regular event in the current time. The current process will be blocked until the next simulation cycle when the update of a will occur and the process will continue executing |
| $a <= \#0 b;$ | This is identical to $a <= b;$                                                                                                                                                                                                       |
| $a = \#4 b;$  | This is like $a = \#0 b;$ except that the update event and the continuation of the process is scheduled 4 time units in the future.                                                                                                  |
| $a <= \#4 b;$ | This is like $a <= \#0 b;$ except that a will not be updated (using a non-blocking update event) until 4 time units in the future.                                                                                                   |

# How the event scheduler deals with $<=$ and $=$

|                |                                                                                                                                   |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------|
| #4 a = b;      | Wait 4 time units before doing the action for $a = b$ ;<br>The value assigned to $a$ will be the value of $b$ 4 time units later. |
| #4 a $\leq$ b; | Wait 4 time units before doing the action for $a \leq b$ ; The value assigned to $a$ will be the value of $b$ 4 time units later. |

# A view to another black hole

- Mixing gate level instances and behavioral code might make you loose your head ...

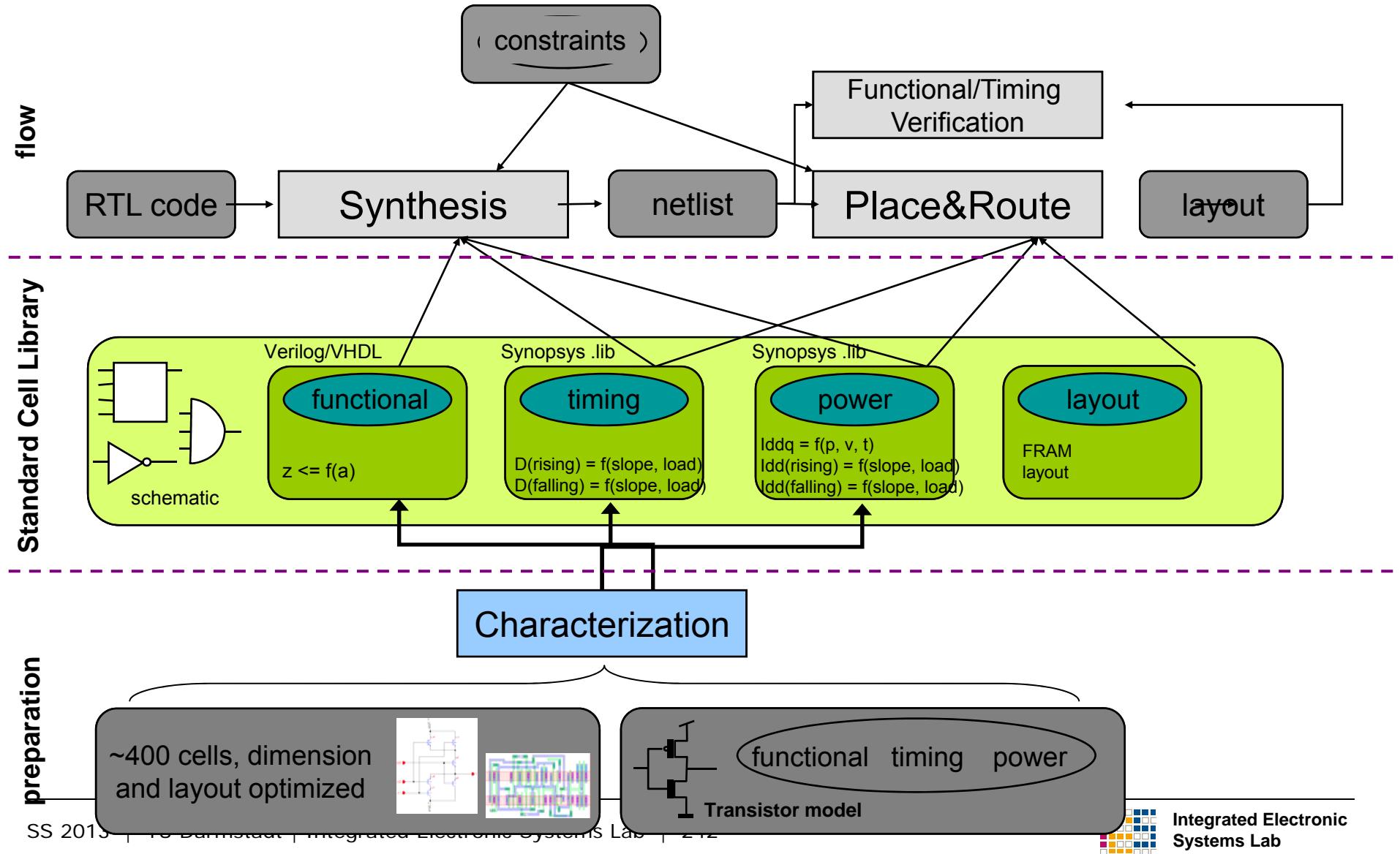
```
module blackhole
    (output reg f,
     input a, b);
    reg q;
initial
    f = 0;
always
    @ (posedge a)
        #10 q = b;
not (qbar, q);
always
    @ q
        f = qbar;
endmodule
```

# Logic Synthesis from RTL descriptions

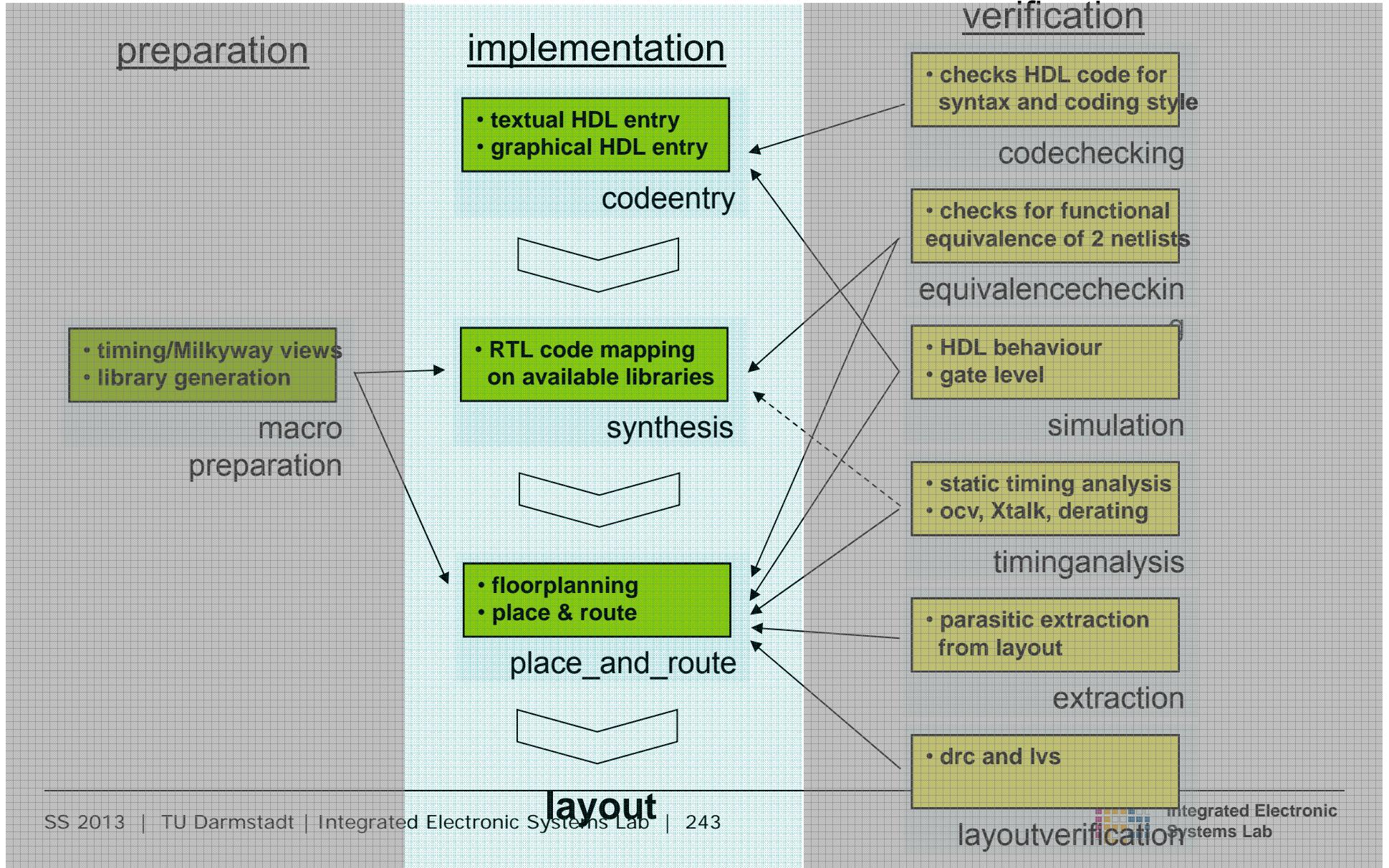


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Logic Synthesis Flow revisited

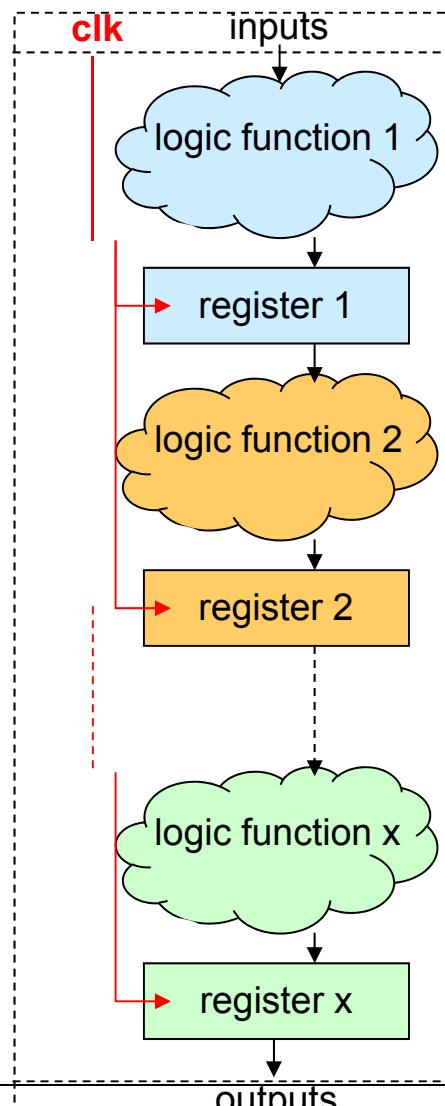


# Logic Synthesis Flow revisited



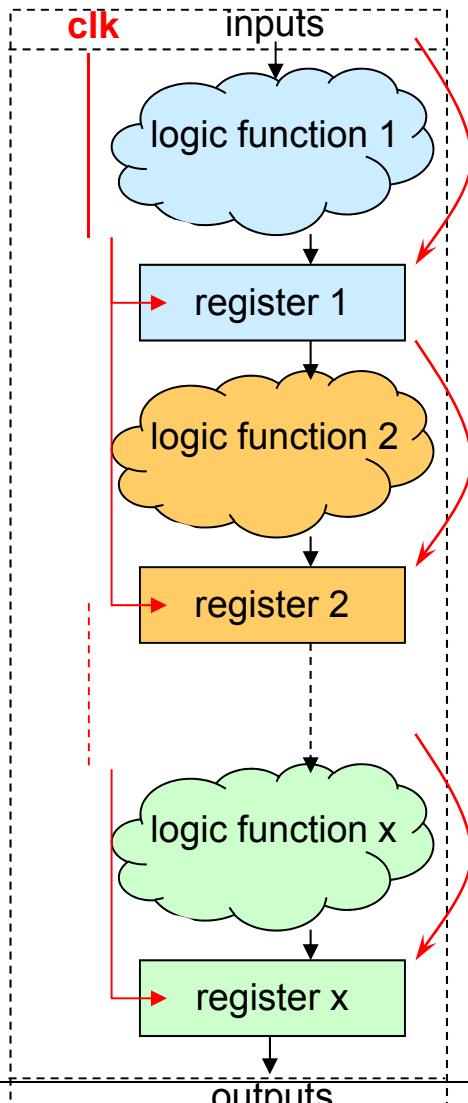
# RTL code and synchronous design

(Register Transfer Level)



# RTL code and synchronous design

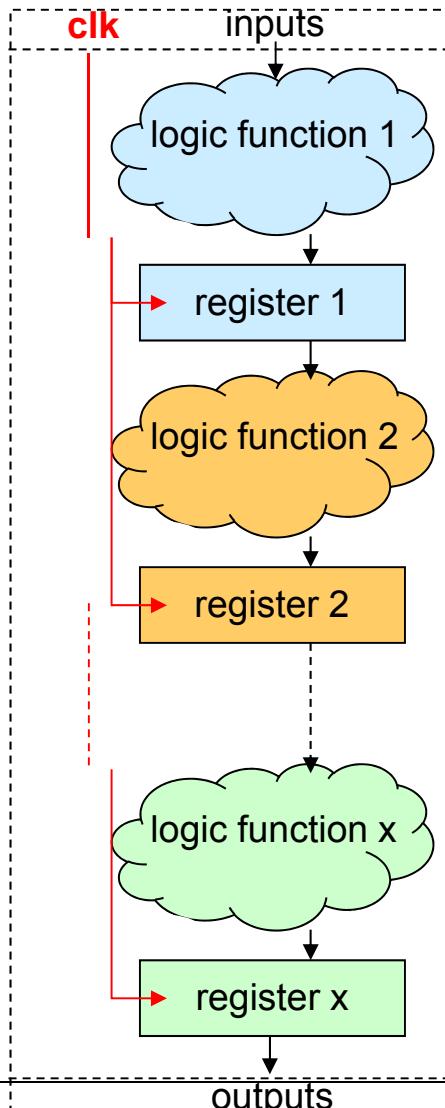
(Register Transfer Level)



synchronously (concurrently)  
data propagate  
from each register  
to the next,  
triggered by the clock signal

# RTL code and synchronous design

(Register Transfer Level)



```

module (...)

  inputs clk, ...
  outputs ...

  always @(posedge clk)
    begin
      function 1
    end

  always @(posedge clk)
    begin
      function 2
    end

  .
  .

  always @(posedge clk)
    begin
      function x
    end

end module

```

# Synthesis

## RTL description

```
module multiplier (clk, resetn, a, b,
z);
  input clk, resetn;
  input [3:0] a, b;
  output reg [7:0] z;

  always @(posedge clk or negedge
resetn)
  begin
    if (~resetn)
      z <= 8'b00000000;
    else
      z <= a * b;
  end
endmodule
```

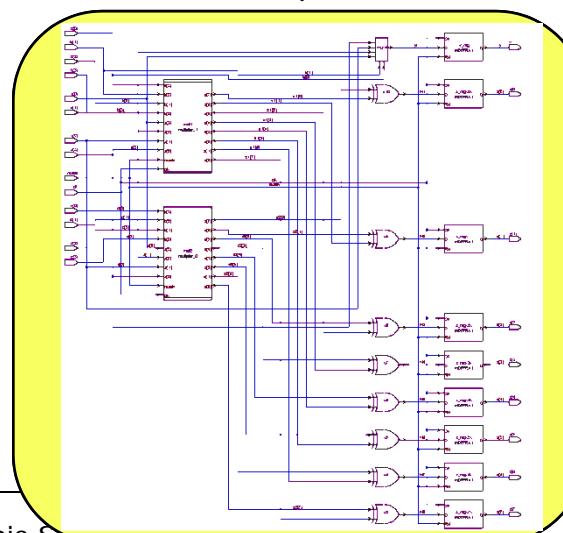
## Library

- Function
- Timing
- Power

## Synthesis

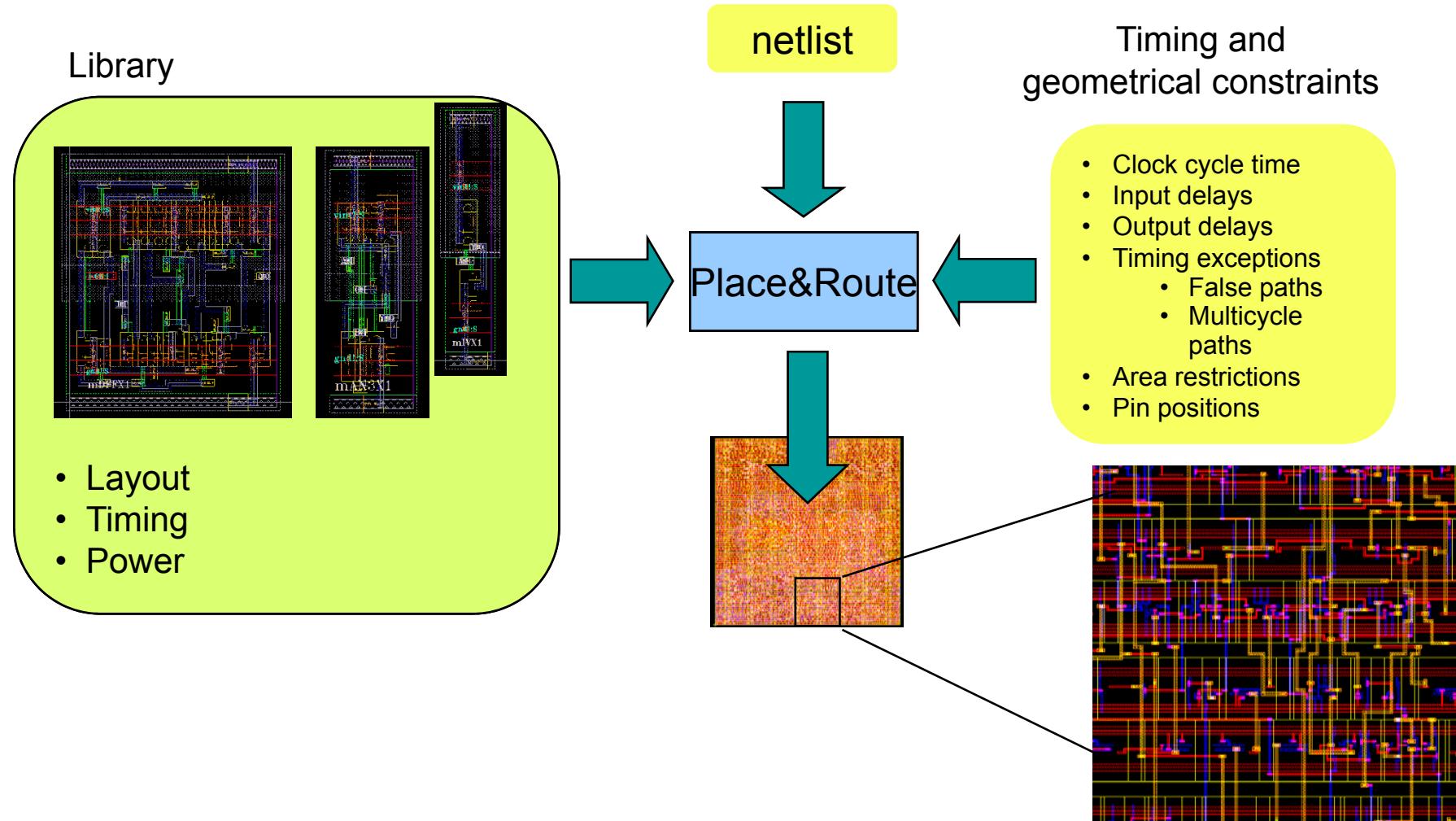
## Timing constraints

- Clock cycle time
- Input delays
- Output delays
- Timing exceptions
  - False paths
  - Multicycle paths

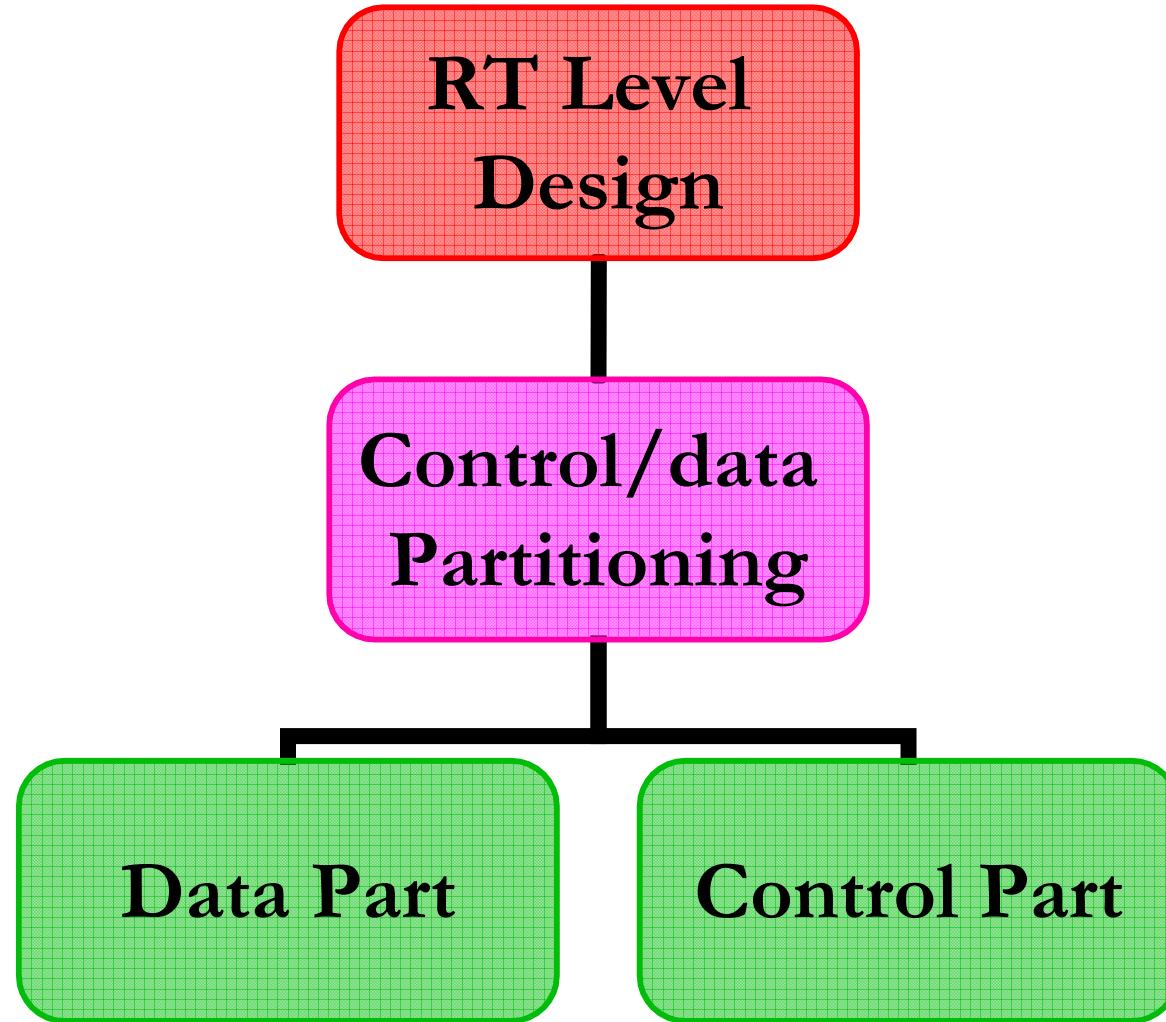


+ netlist

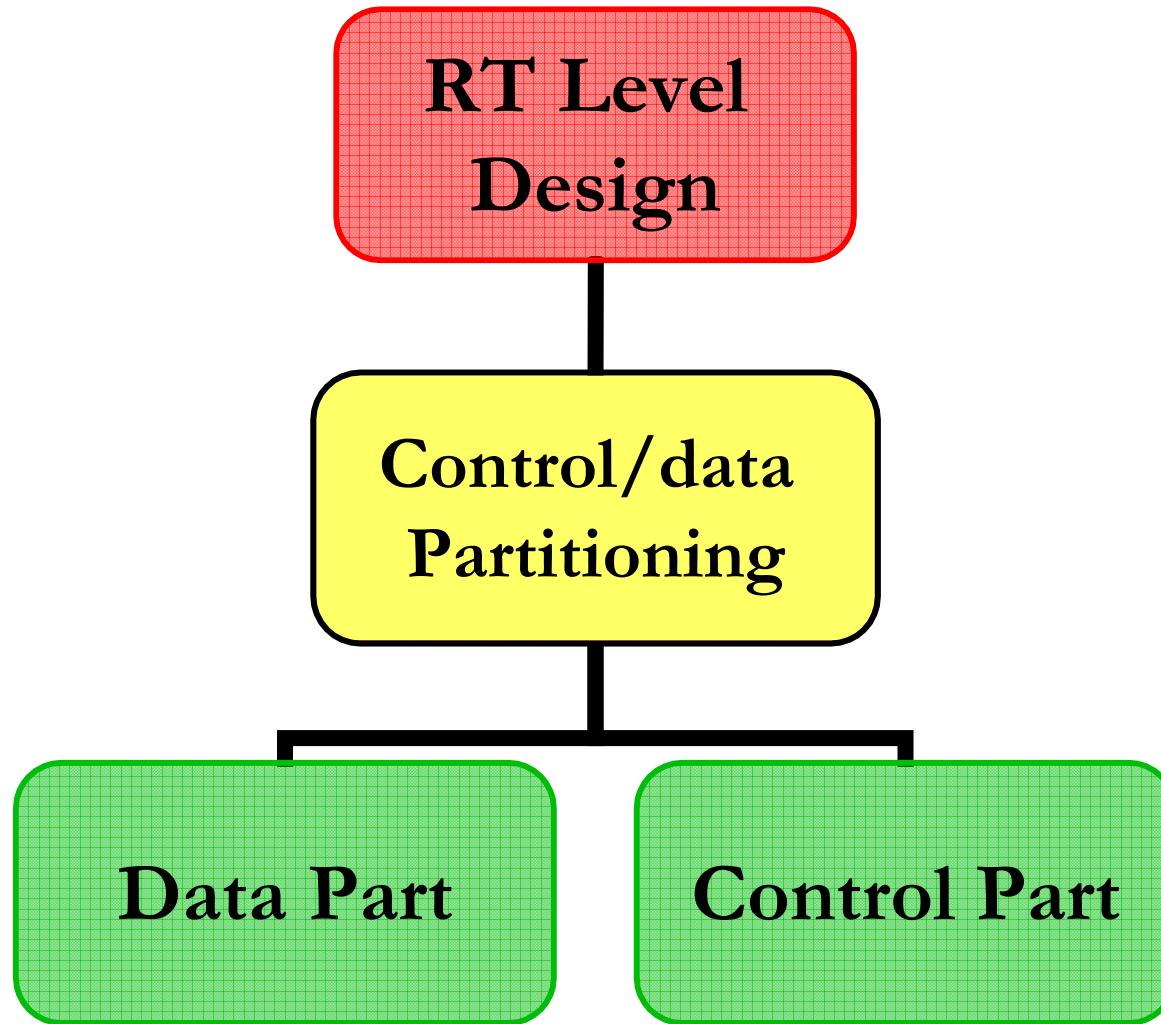
# Place & Route



- **RT level design:**
  - Taking a high level description of a design
  - Partitioning
  - Coming up with an architecture
  - Designing the bus structures
  - Describing and implementing various components of the architecture
- **Steps in RT level design:**
  - Control/Data Partitioning
  - Data Part Design
  - Control Part Design

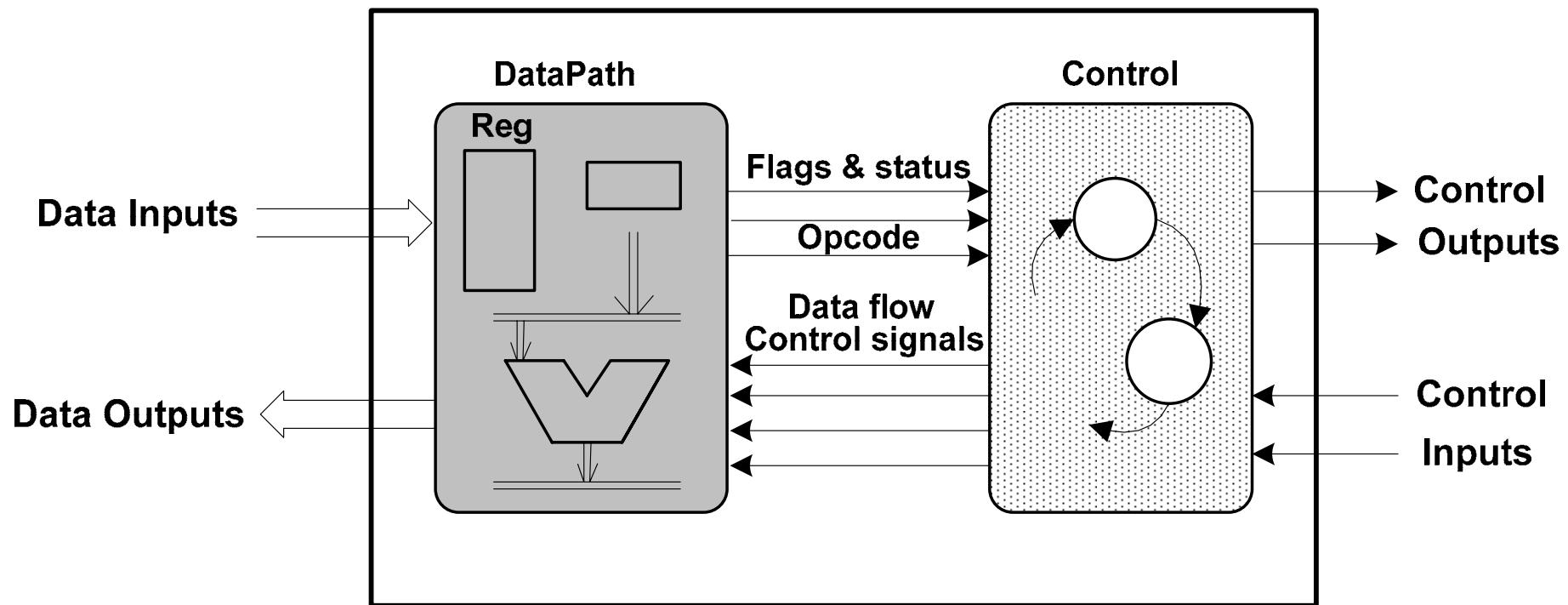


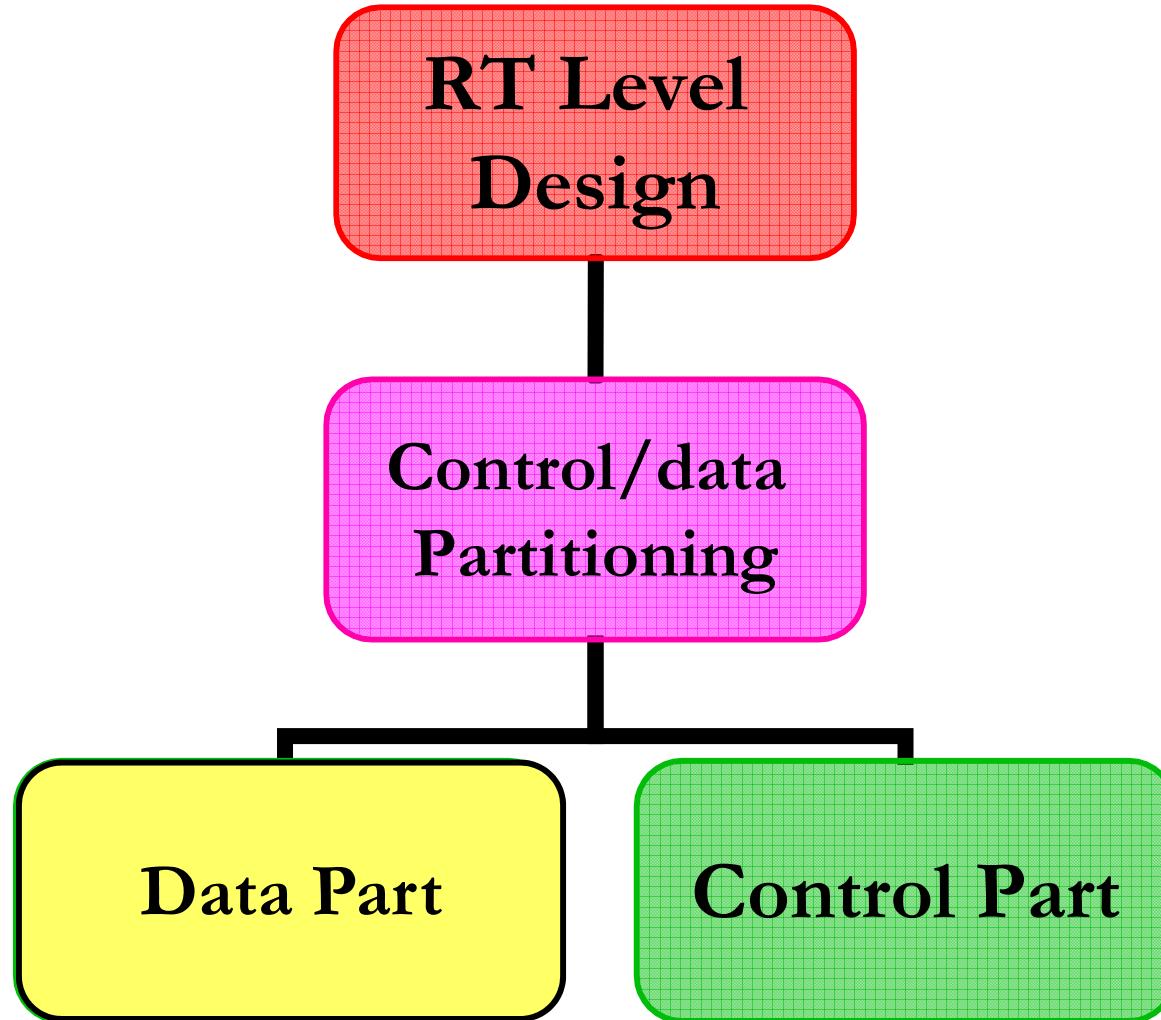
# Control/Data Partitioning



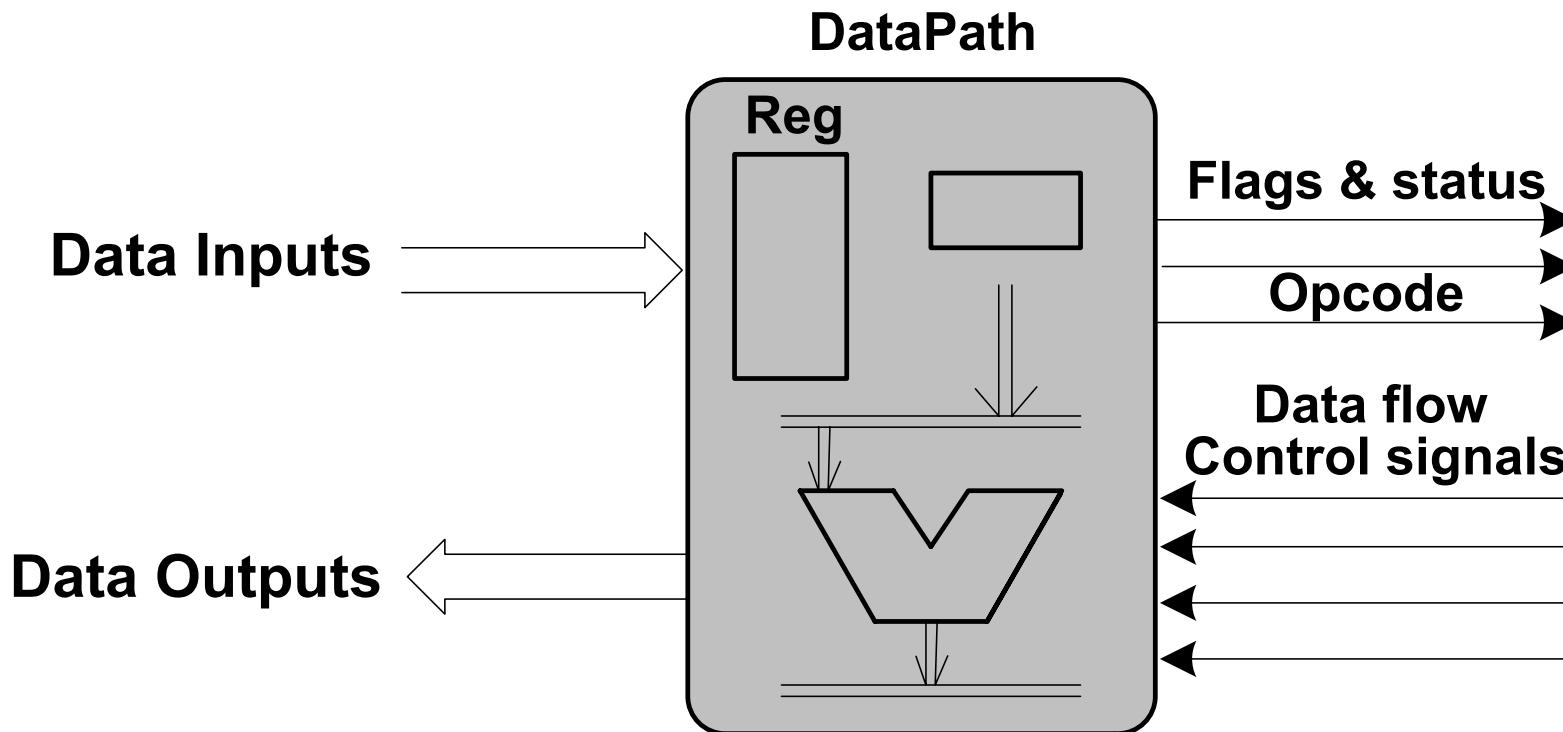
# Control/Data Partitioning

## RT Level Design





# Data Part



# Data Part

```
module DataPath
  (DataInput, DataOutput, Flags, Opcodes,
   ControlSignals);

  input [15:0] DataInputs;
  output [15:0] DataOutputs;
  output Flags, ...;
  output Opcodes, ...;
  input ControlSignals, ...,
    // instantiation of data components
    // ...
    // interconnection of data components
    // bussing specification
endmodule
```

Output Signals: Going to the control part, provide flags and status of the data

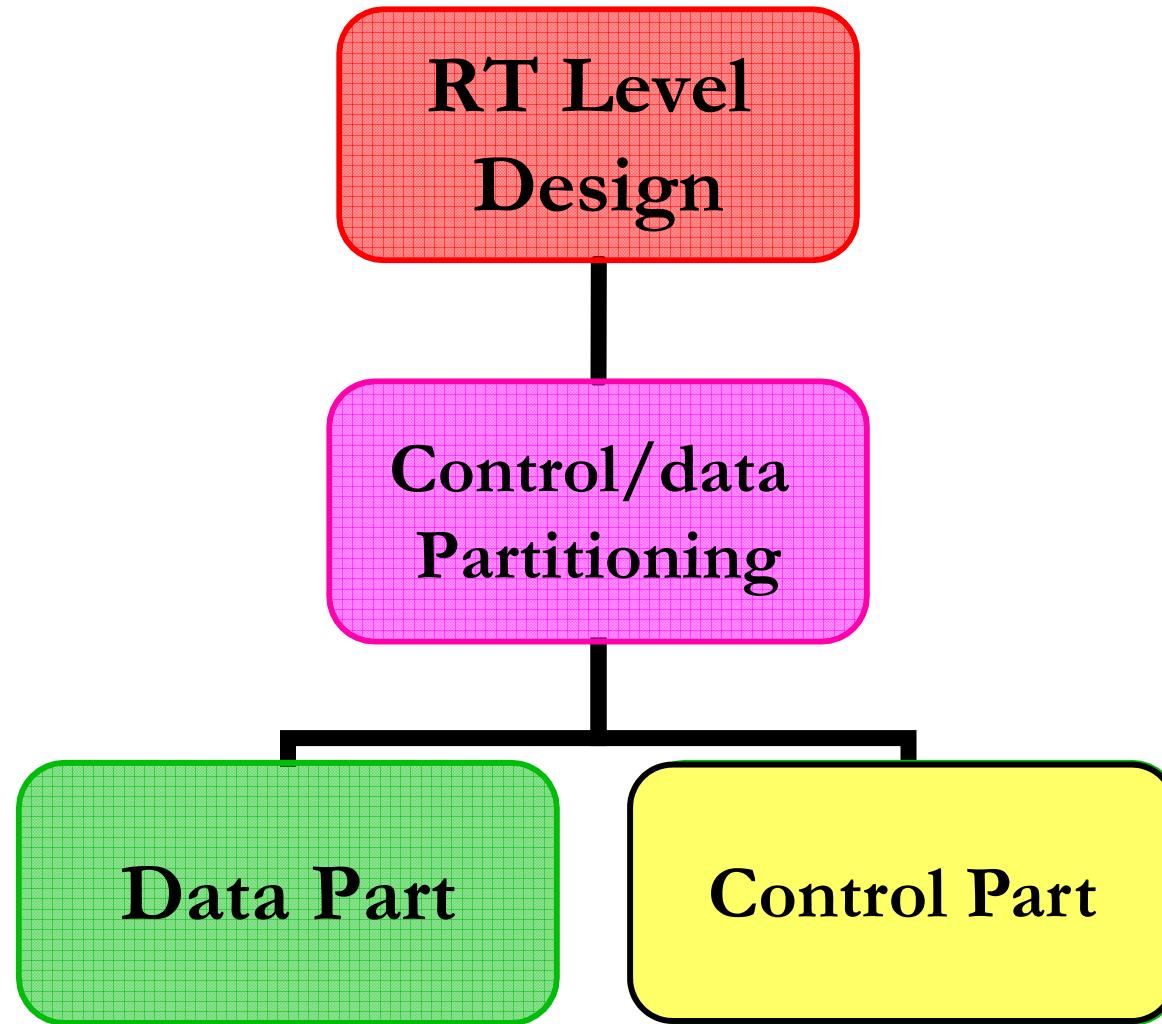
Control Signals:  
Inputs to data part, sent to the data components and busses

Control Signals for the busses:  
Select the sources and routing of data from one data component to another

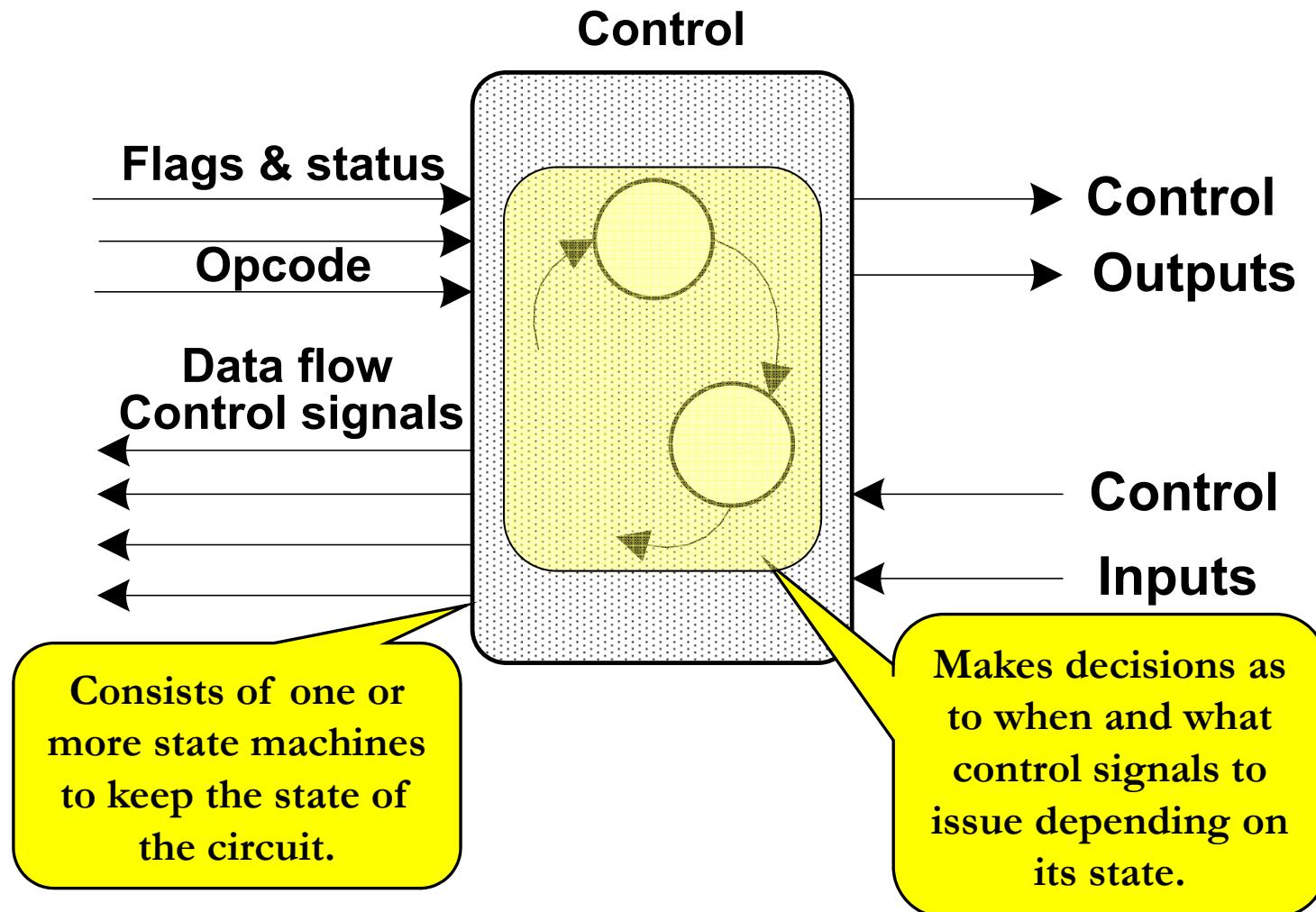
# Data Part

```
module DataComponent  
  (DataIn, DataOut, ControlSignals);  
  
  input  [7:0] DataIn;  
  output [7:0] DataOut;  
  input  ControlSignals;  
  // Depending on ControlSignals  
  // Operate on DataIn and  
  // Produce DataOut  
endmodule
```

**Data Component:**  
Shows how the component uses its input control signals to perform various operations on its data inputs



# Control Part



# Control Part

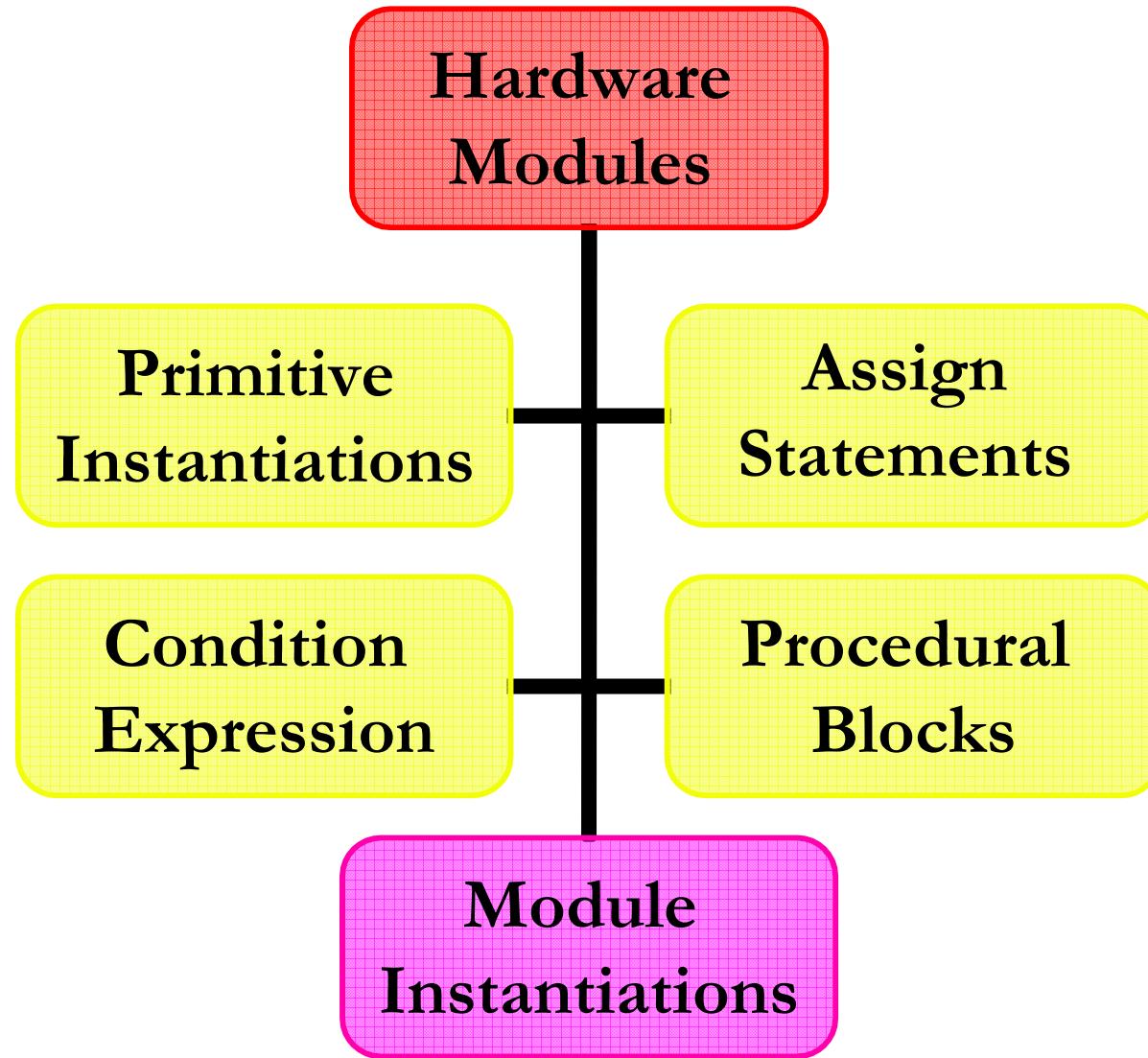
```
module ControlUnit
  (Flags, Opcodes, ExternalControls, ControlSignals);

  input Flags, ...;
  input Opcodes, ...;
  input ExternalControls, ...;
  output ControlSignals;
  // Based on inputs decide :
  // What control signals to issue,
  // and what next state to take
endmodule
```

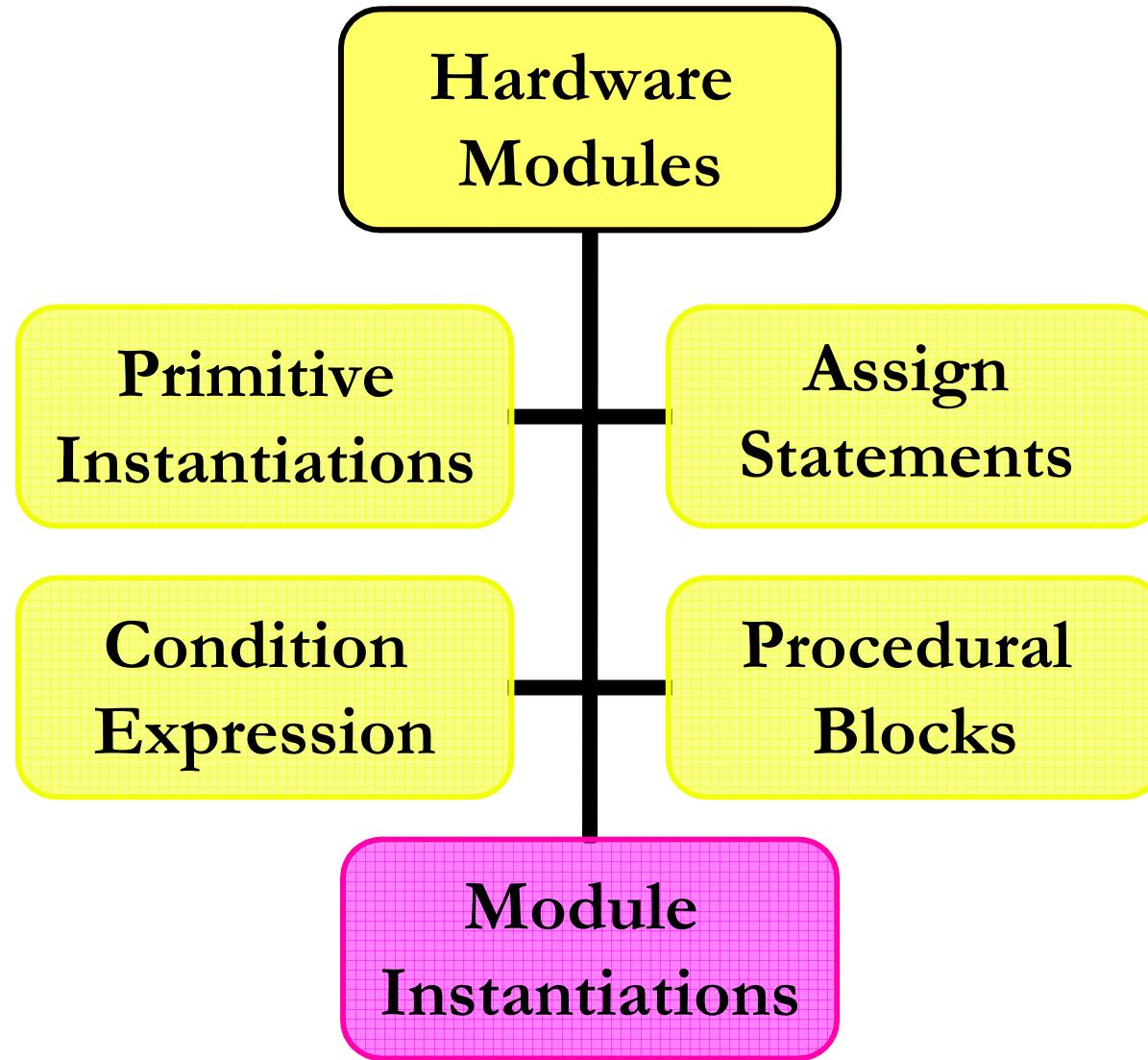
Takes control  
inputs from the  
Data Part

- Outline of a Controller

# Elements of Verilog



# Hardware Modules



# Hardware Modules

Keyword  
*module*

```
module module-name
List of ports;
Declarations
...
Functional specification of module
```

Variables, wires, and  
module parameters  
are declared.

*module* :  
The Main  
Component  
of Verilog

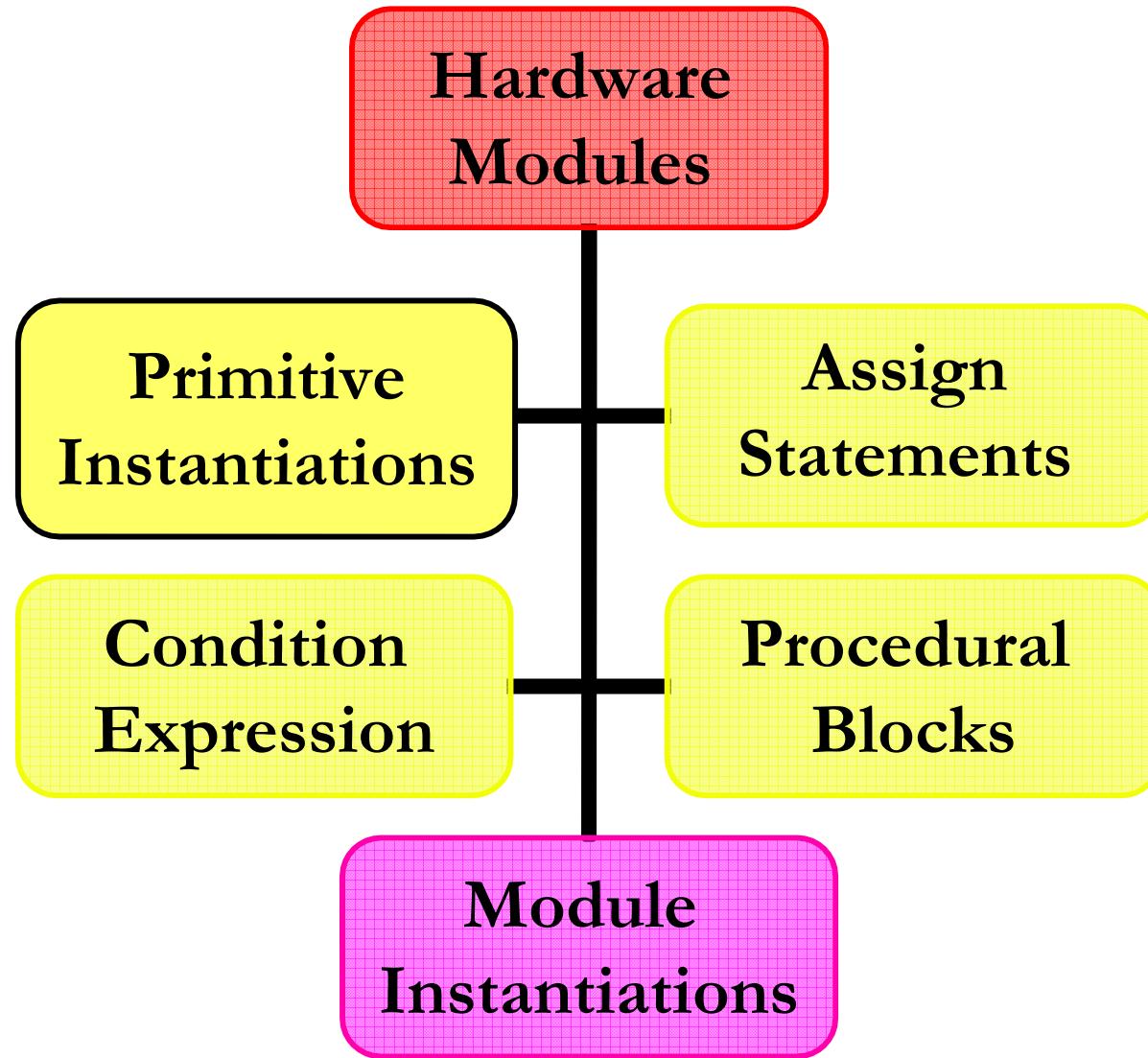
**endmodule**

Keyword  
*endmodule*

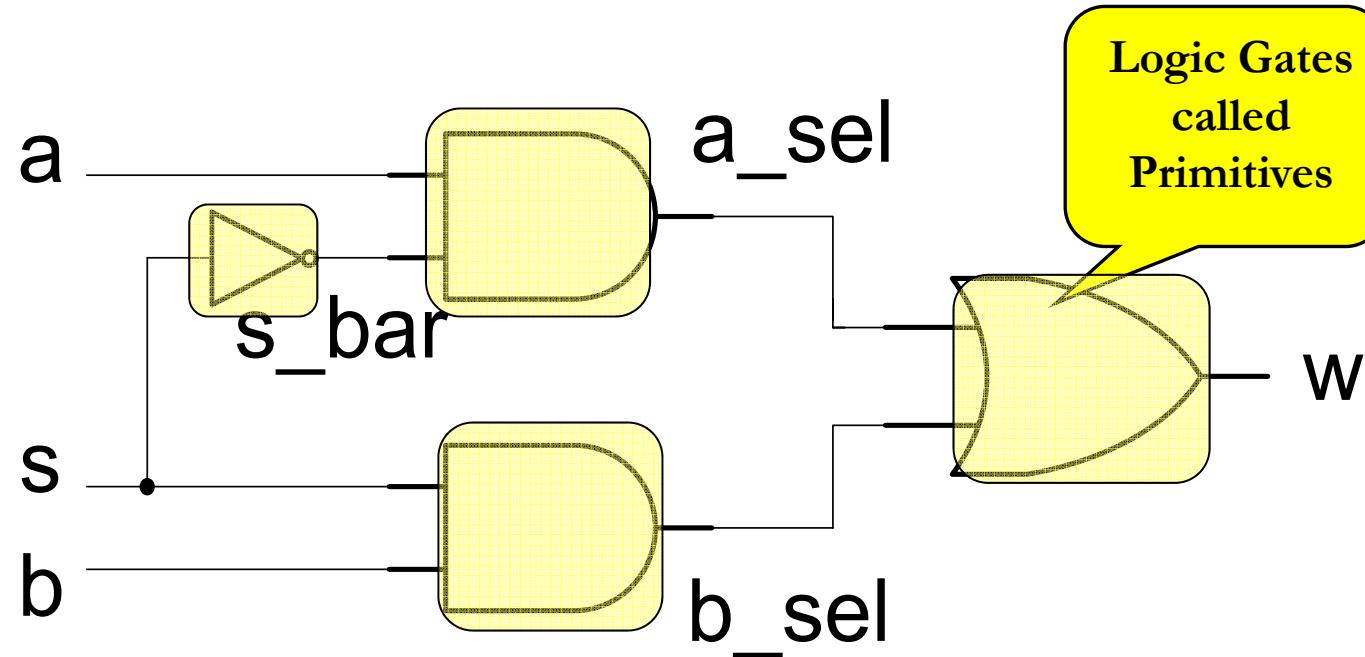
- Module Specifications

- There is more than one way to describe a Module in Verilog.
- May correspond to descriptions at various levels of abstraction or to various levels of detail of the functionality of a module.
- Descriptions of the same module need not behave in exactly the same way nor is it required that all descriptions describe a behavior correctly.
- We discuss basic constructs of Verilog language for a hardware module description.
- We show a small example and several alternative ways to describe it in Verilog.

# Primitive Instantiations

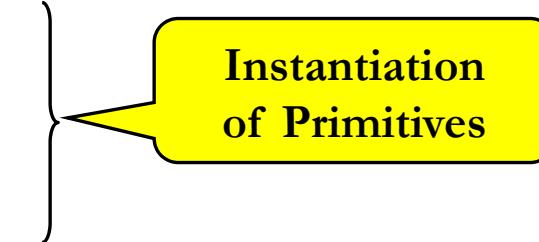


# Primitive Instantiations



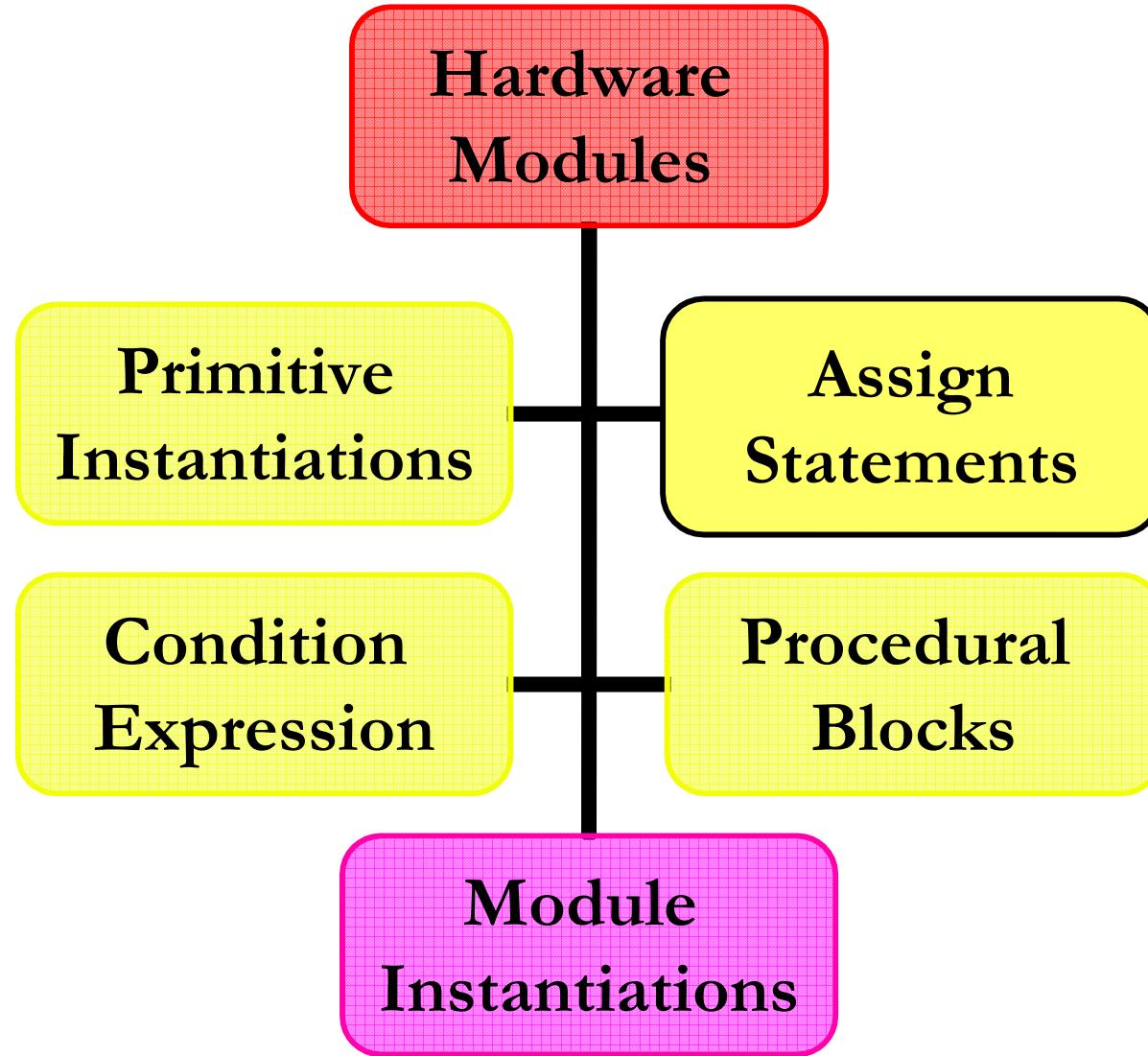
# Primitive Instantiations

```
module MultiplexerA (input a, b, s, output w);
    wire a_sel, b_sel, s_bar;
    not U1 (s_bar, s);
    and U2 (a_sel, a, s_bar);
    and U3 (b_sel, b, s);
    or  U4 (w, a_sel, b_sel);
endmodule
```



- Primitive Instantiations

# Assign Statements



# Assign Statements

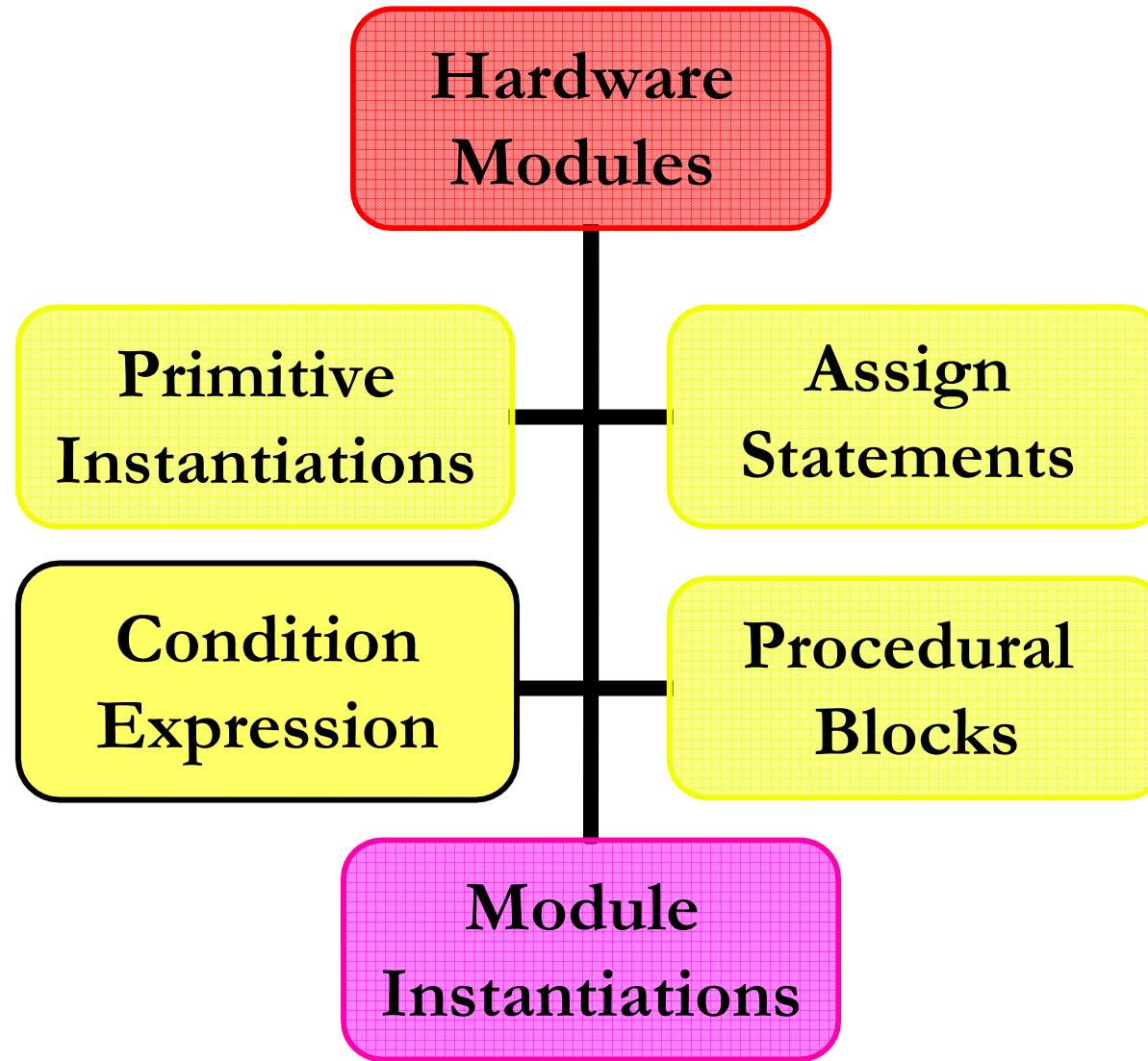
Continuously  
drives  $w$  with the  
right hand side  
expression

```
module MultiplexerB (input a, b, s, output w);
    assign w = (a & ~s) | (b & s);
endmodule
```

- Assign Statement and Boolean

Using Boolean  
expressions to  
describe the logic

# Condition Expression



# Condition Expression

```
module MultiplexerC (input a, b, s, output w);
    assign w = s ? b : a;
endmodule
```

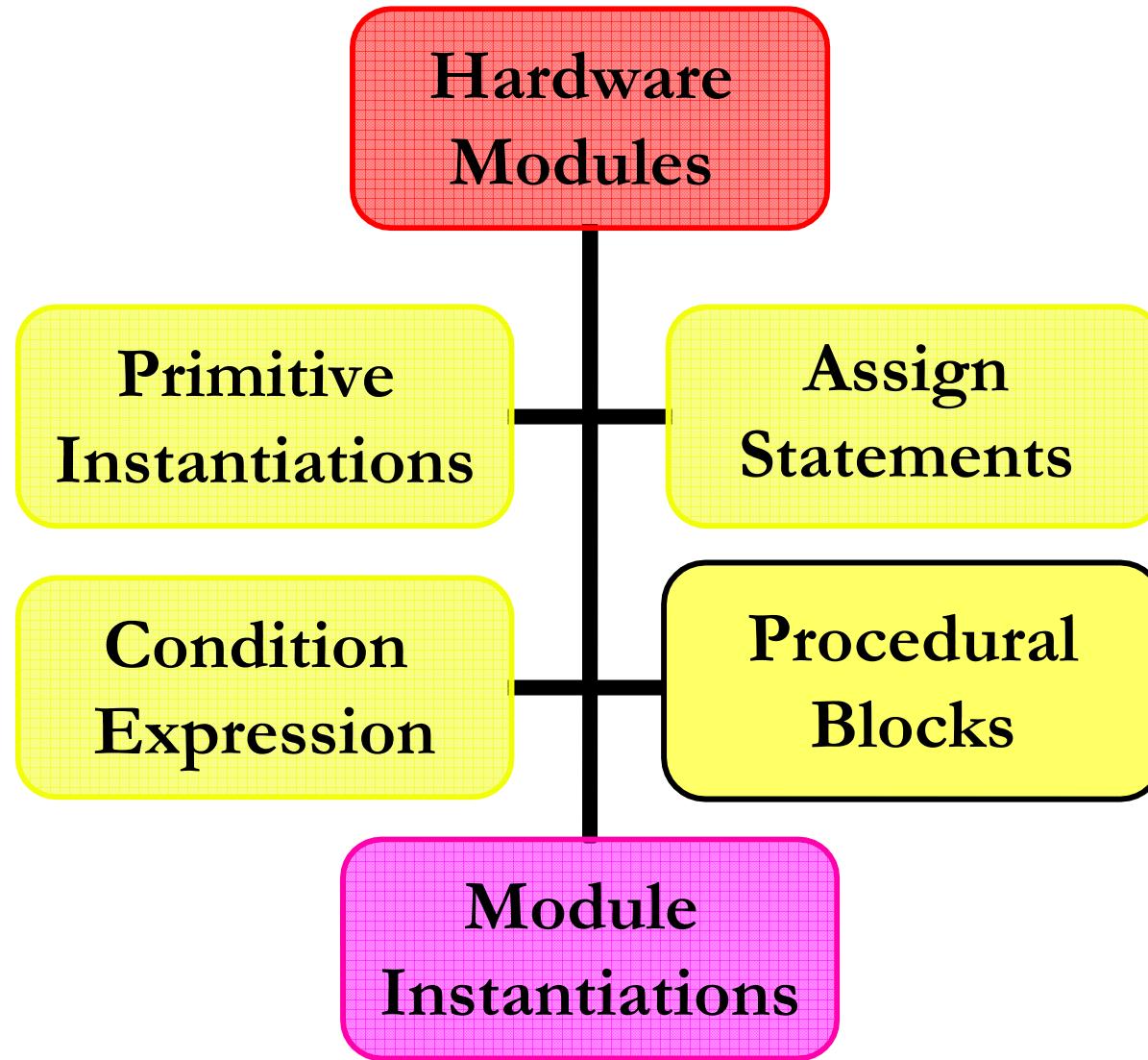
Can be used when the operation of a unit is too complex to be described by Boolean expressions

- Assign Statement and Condition Operator

Useful in describing a behavior in a very compact way

Very Effective in describing complex functionalities

# Procedural Blocks



# Procedural Blocks

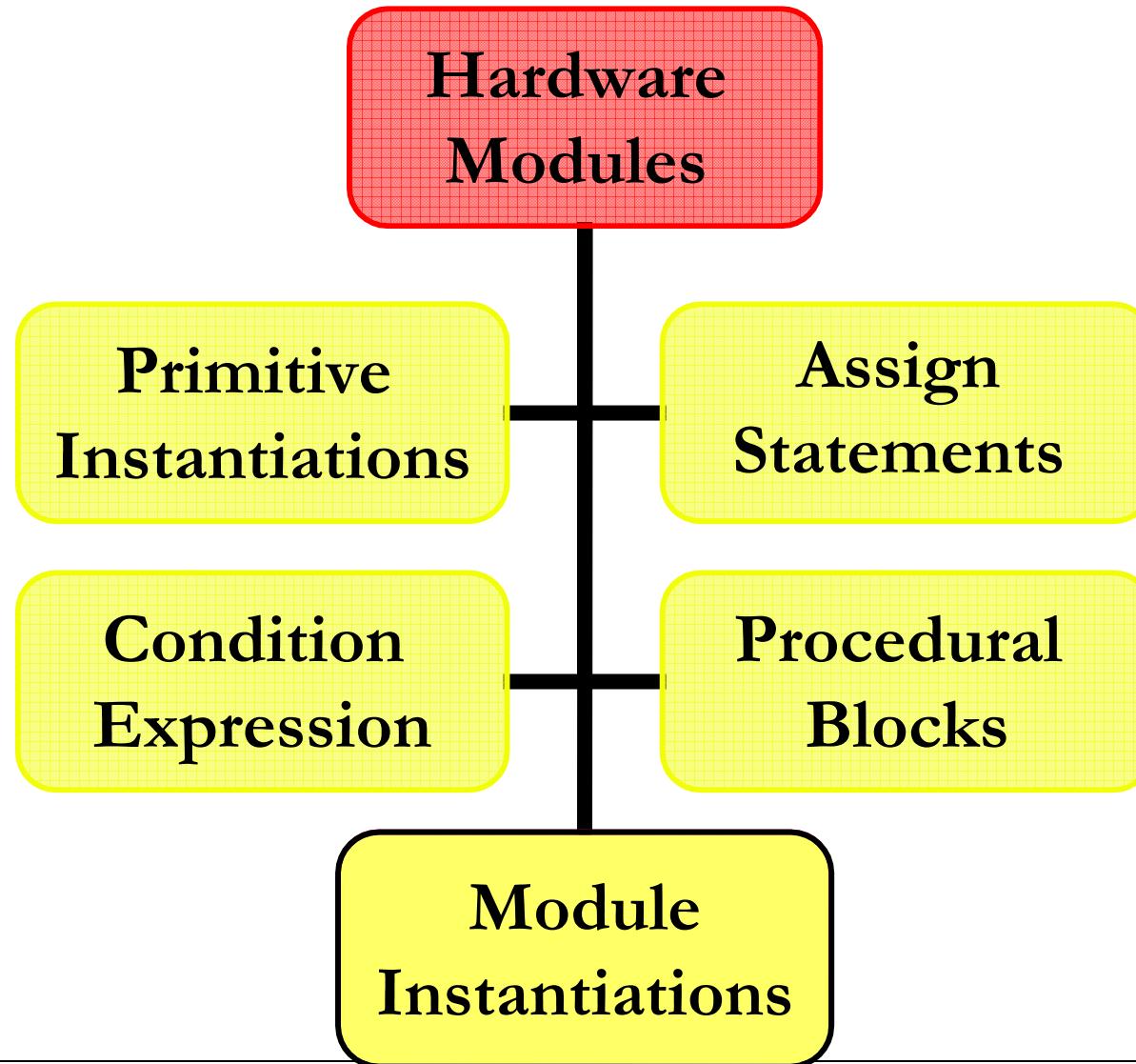
```
always  
statement  
  
Sensitivity list  
  
module MultiplexerD (input a, b, s, output w);  
    reg w;  
    always @(a, b, s) begin  
        if (s) w = b;  
        else w = a;  
    end  
endmodule
```

if-else  
statement

Can be used when the  
operation of a unit is  
too complex to be  
described by Boolean or  
conditional expressions

- Procedural Statement

# Module Instantiations



# Module Instantiations

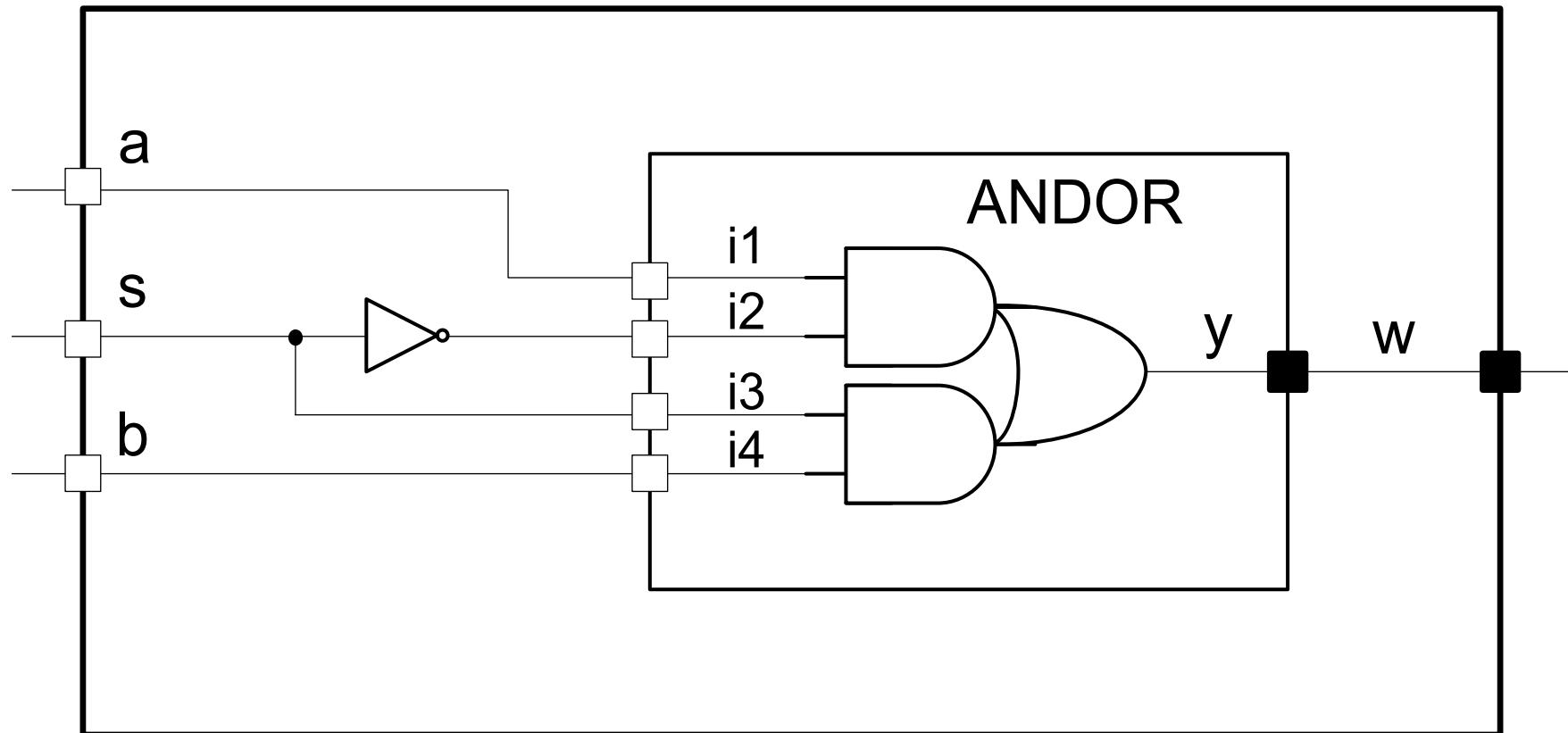
```
module ANDOR (input i1, i2, i3, i4, output y);
    assign y = (i1 & i2) | (i3 & i4);
endmodule
//
module MultiplexerE (input a, b, s, output w);
    wire s_bar;
    not U1 (s_bar, s);
    ANDOR U2 (a, s_bar, s, b, w);
endmodule
```

ANDOR  
module is  
defined

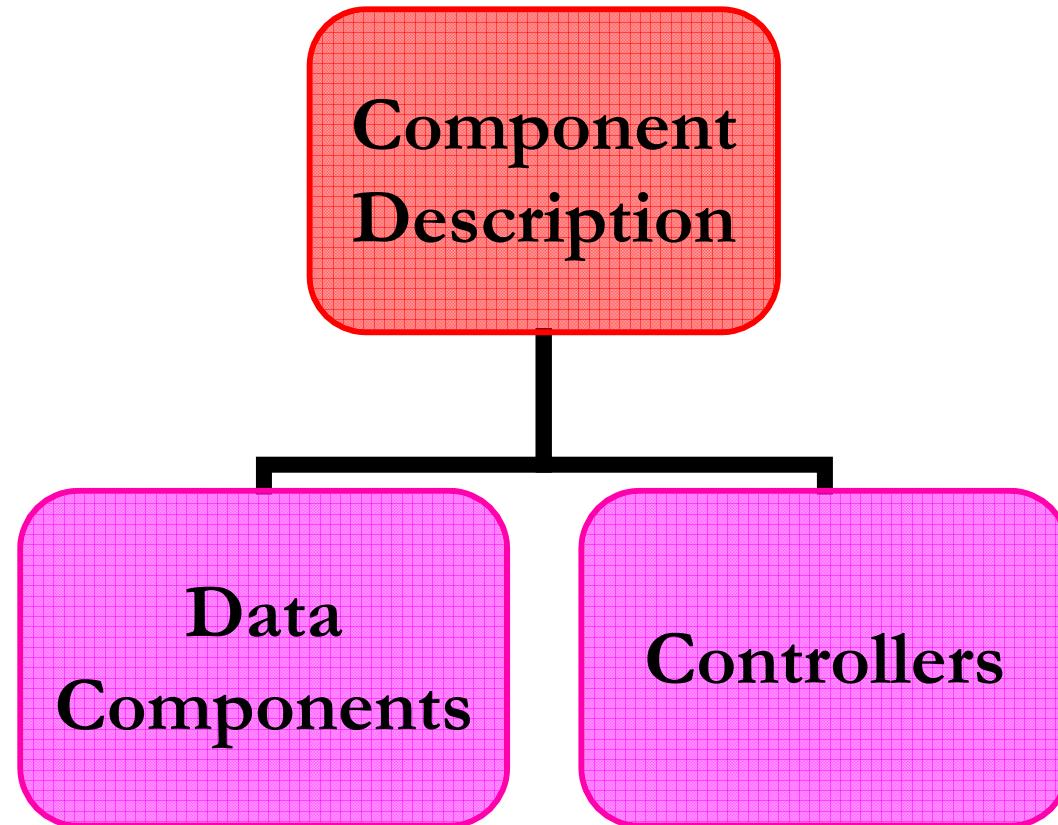
ANDOR  
module is  
instantiated

- Module Instantiation

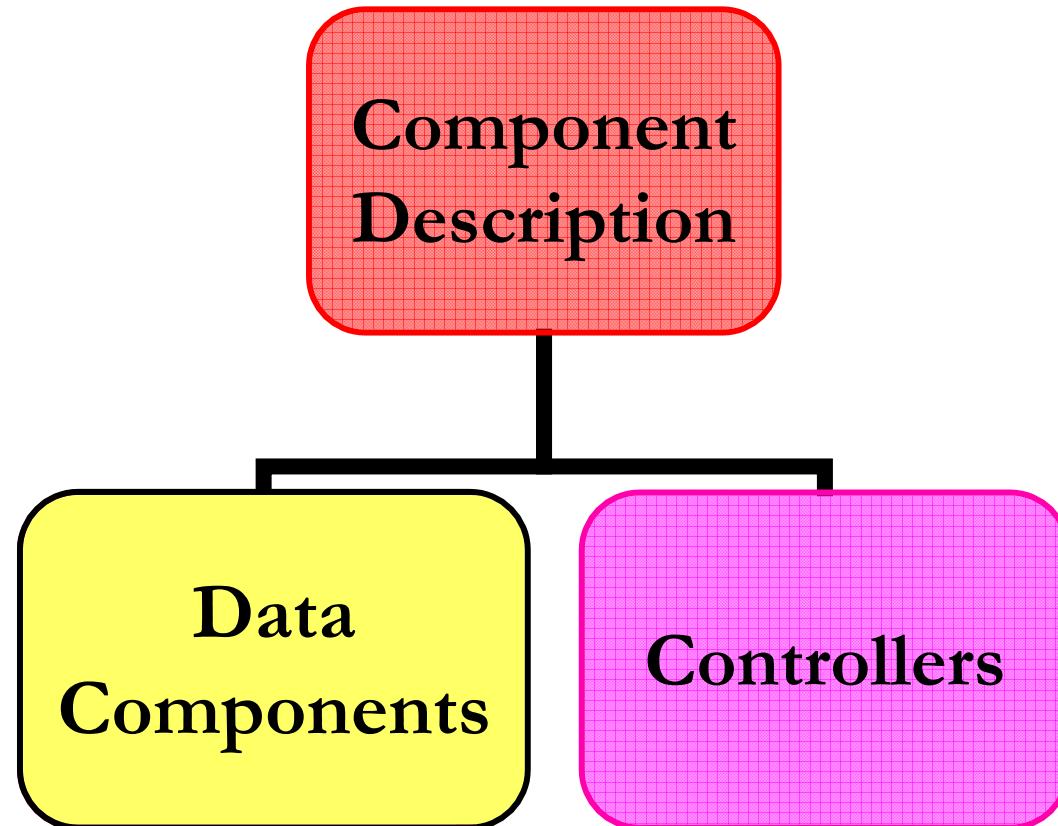
# Module Instantiations



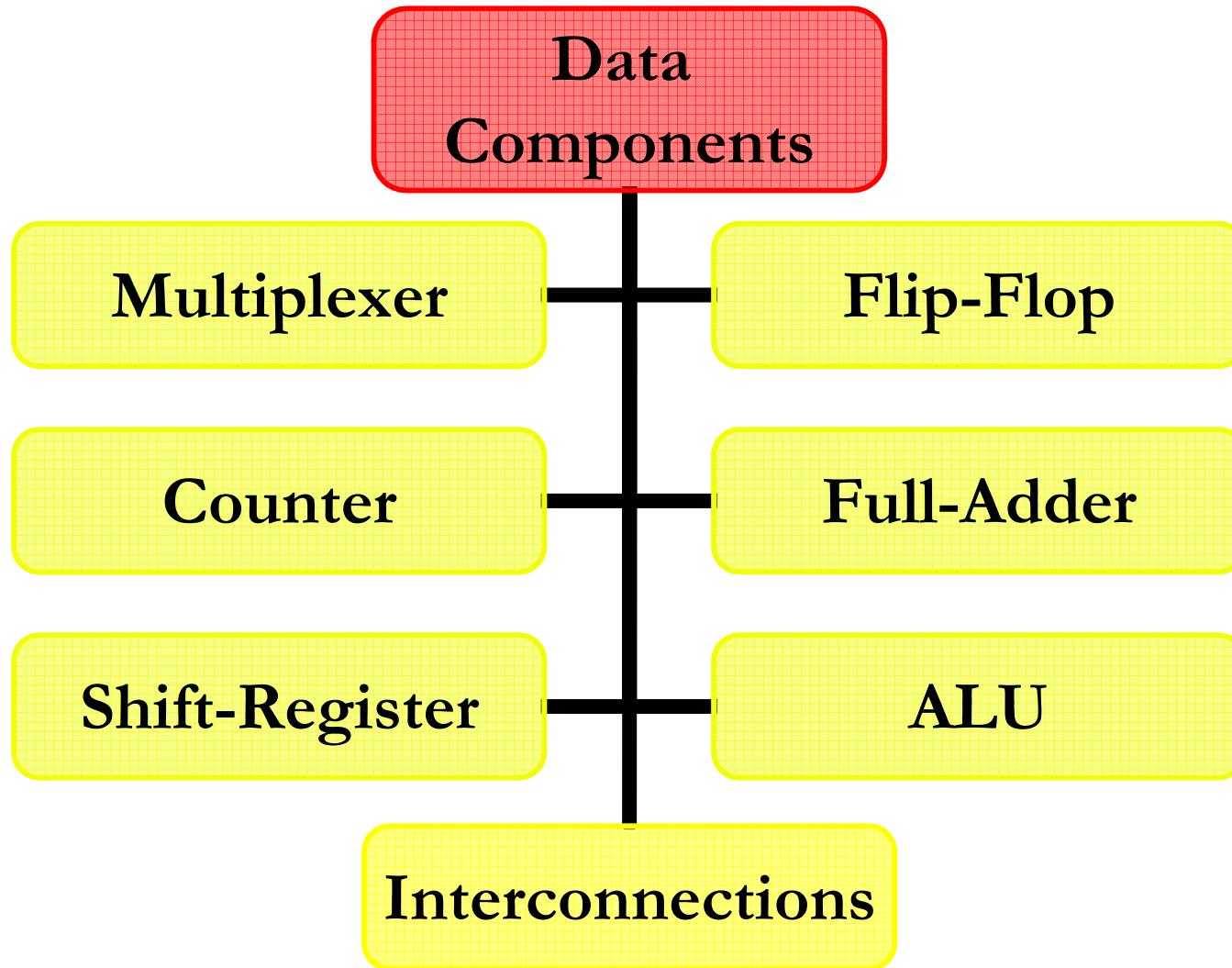
# Component Description in Verilog

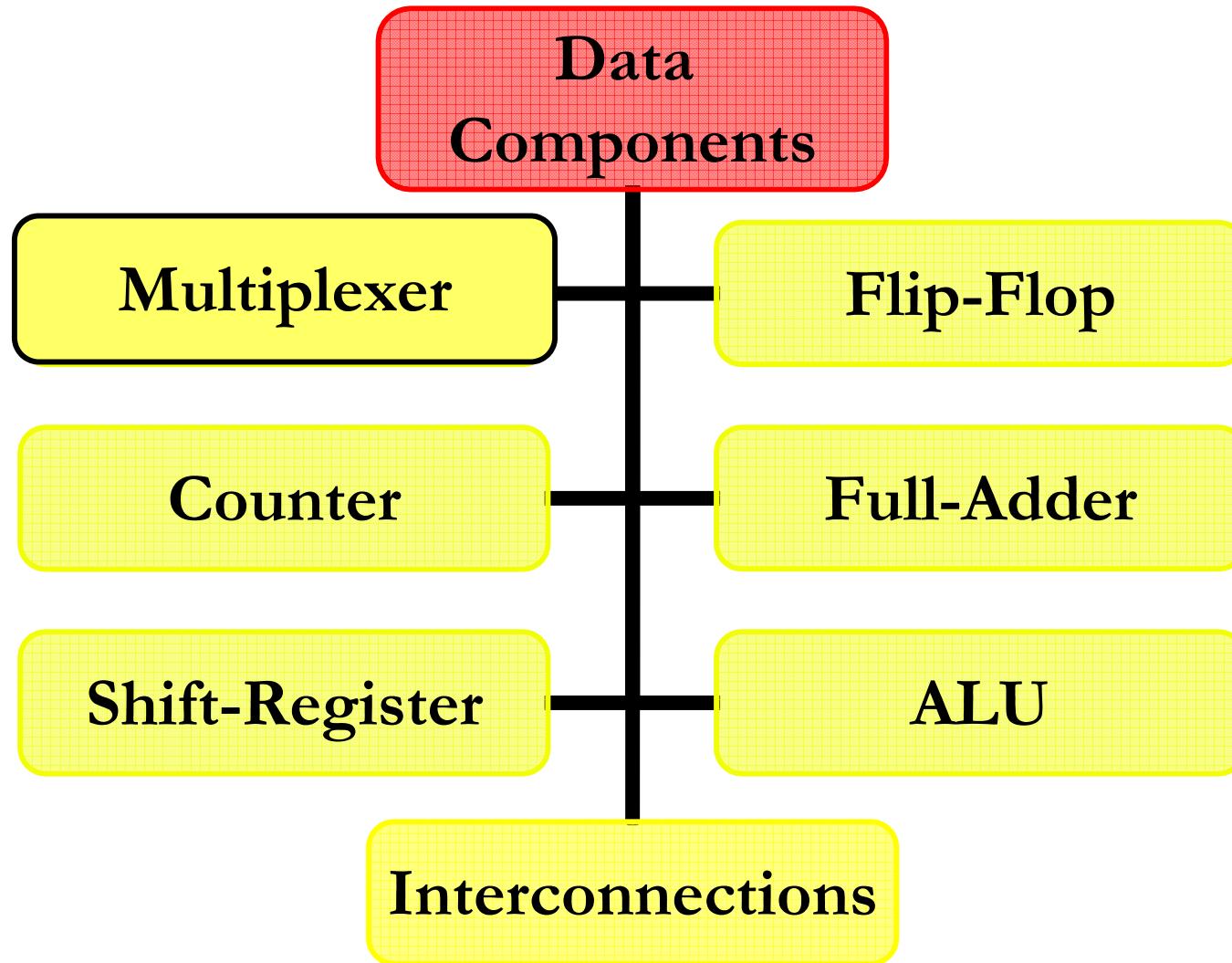


# Data Components



# Data Components





# Multiplexer

Defines a Time Unit of 1 ns  
and Time Precision of 100 ps.

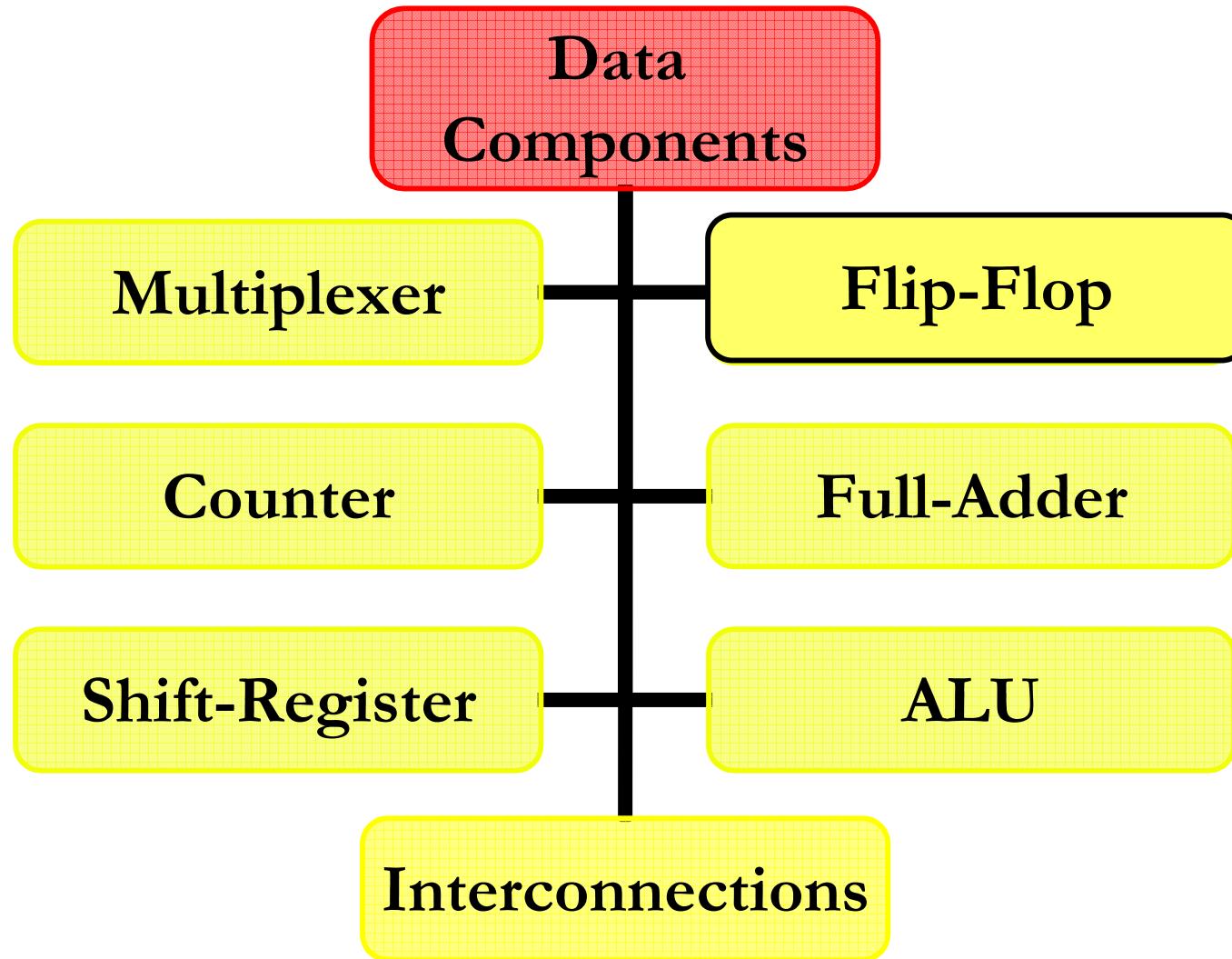
```
`timescale 1ns/100ps
```

```
module Mux8 (input sel, input [7:0] data1, data0,  
              output [7:0] bus1);  
    assign #6 bus1 = sel ? data1 : data0;  
endmodule
```

- Octal 2-to-1

A 6-ns Delay  
is specified for all  
values assigned to  
*bus1*

Selects its 8-bit  
*data0* or *data1* input  
depending on its  
*sel* input.



# Flip-Flop

```
'timescale 1ns/100ps

module Flop (reset, din, qout);
    input reset, din;
    output qout;
    reg qout;
    always @ (negedge clk) begin
        if (reset) qout <= #8 1'b0;
        else qout <= #8 din;
    end
endmodule
```

Synchronous  
*reset* input

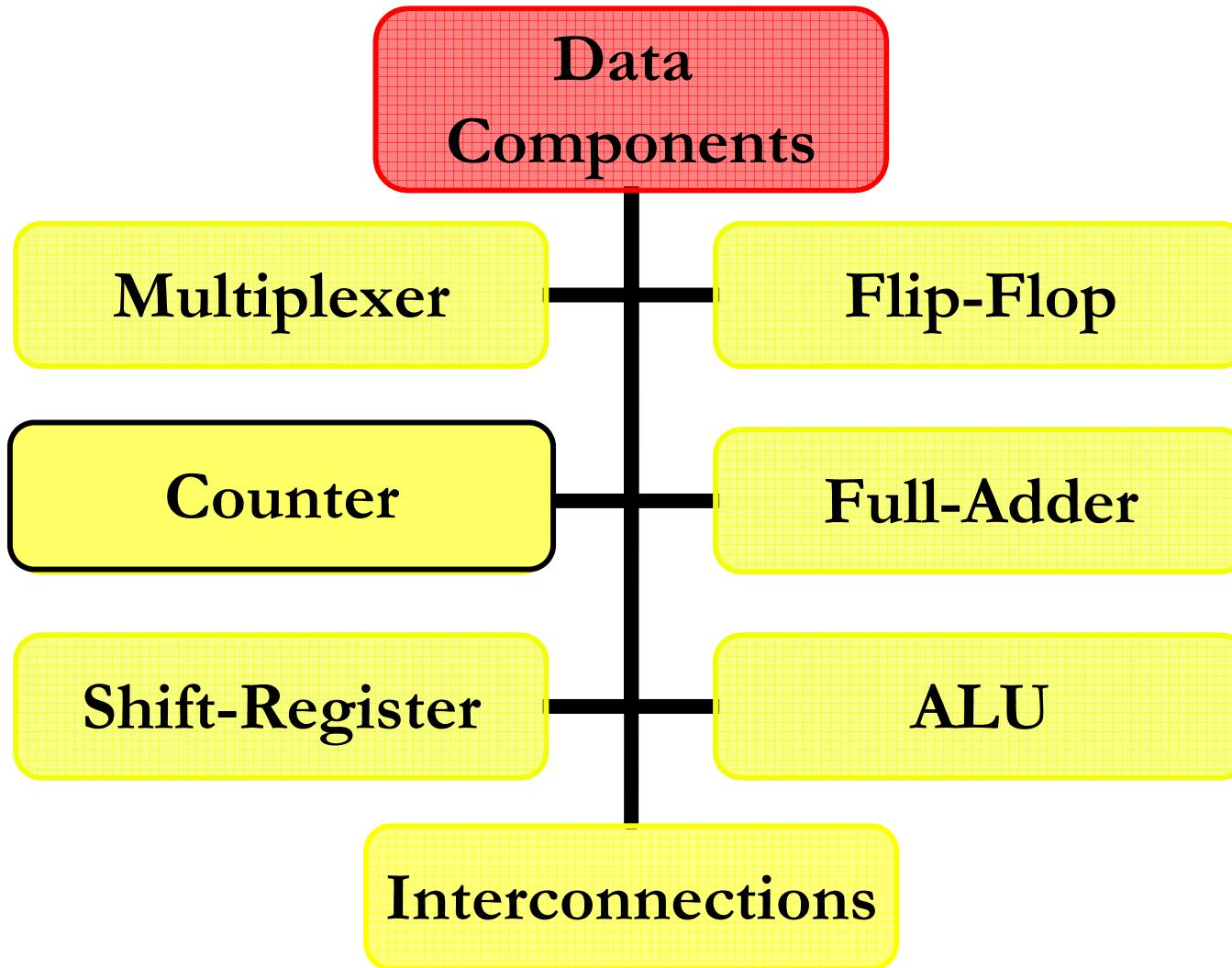
Flip-Flop  
triggers on the  
falling edge of  
*clk* Input

The Body of  
*always* statement is  
executed at the  
negative edge of  
the *clk* signal

An 8-ns  
Delay

A Non-blocking  
Assignment

- Flip-Flop Description



# Counter

A 4-bit modulo-16 Counter

4-bit Register

```
ale 1ns/100ps
Counter4 (input reset, clk,
           output [3:0] count);

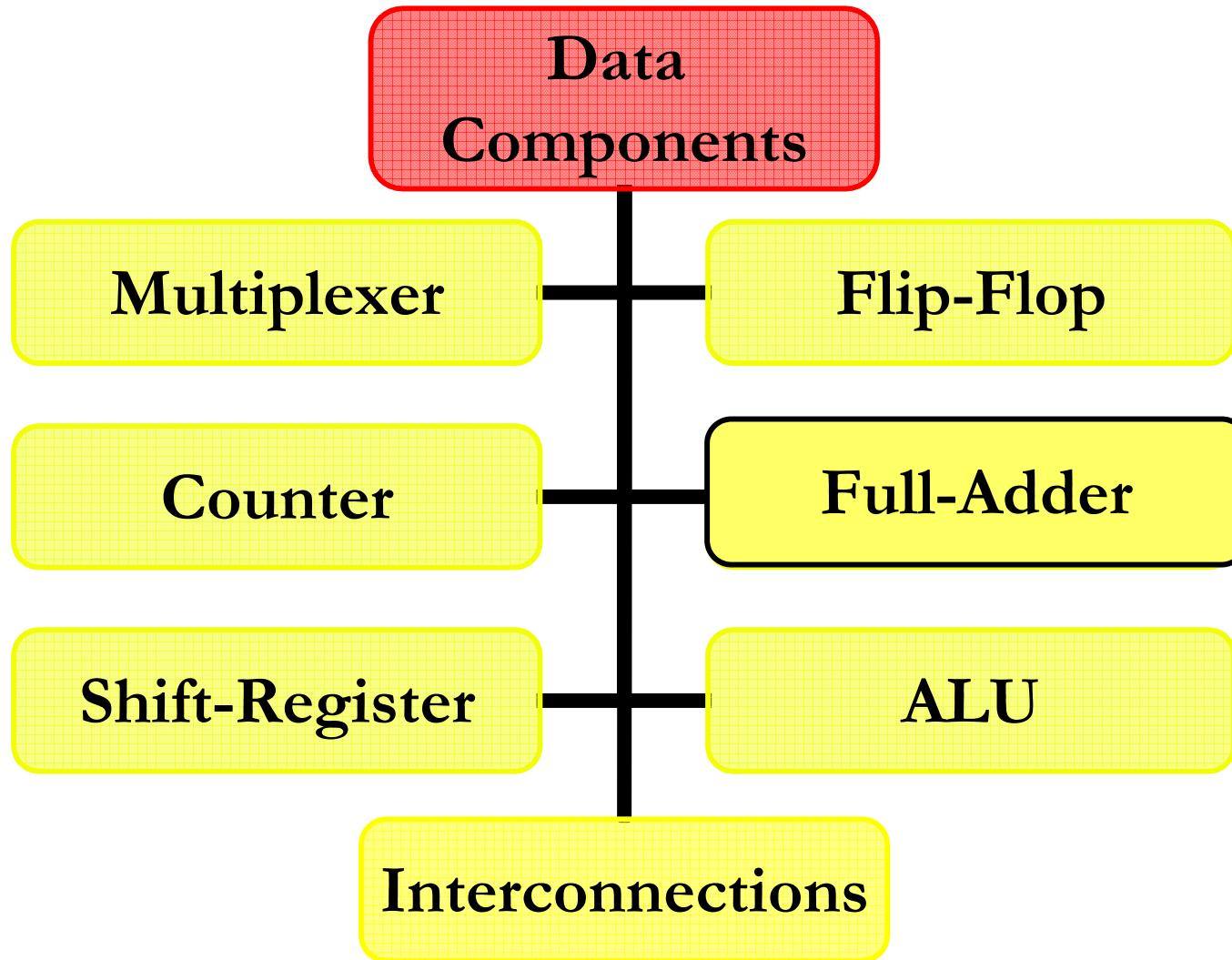
reg [3:0] count;
always @ (negedge clk) begin
    if (reset) count <= #3 4'b00_00;
    else count <= #5 count + 1;
end
endmodule
```

Counters are used in data part for registering data, accessing memory or queues and register stacks

Constant Definition

When *count* reaches 1111, the next count taken is 10000

- Counter Verilog Code



# Full-Adder

A combinational circuit

All Changes  
Occur after 5 ns

Full-Adders are used  
in data part for  
building  
Carry-Chain adders

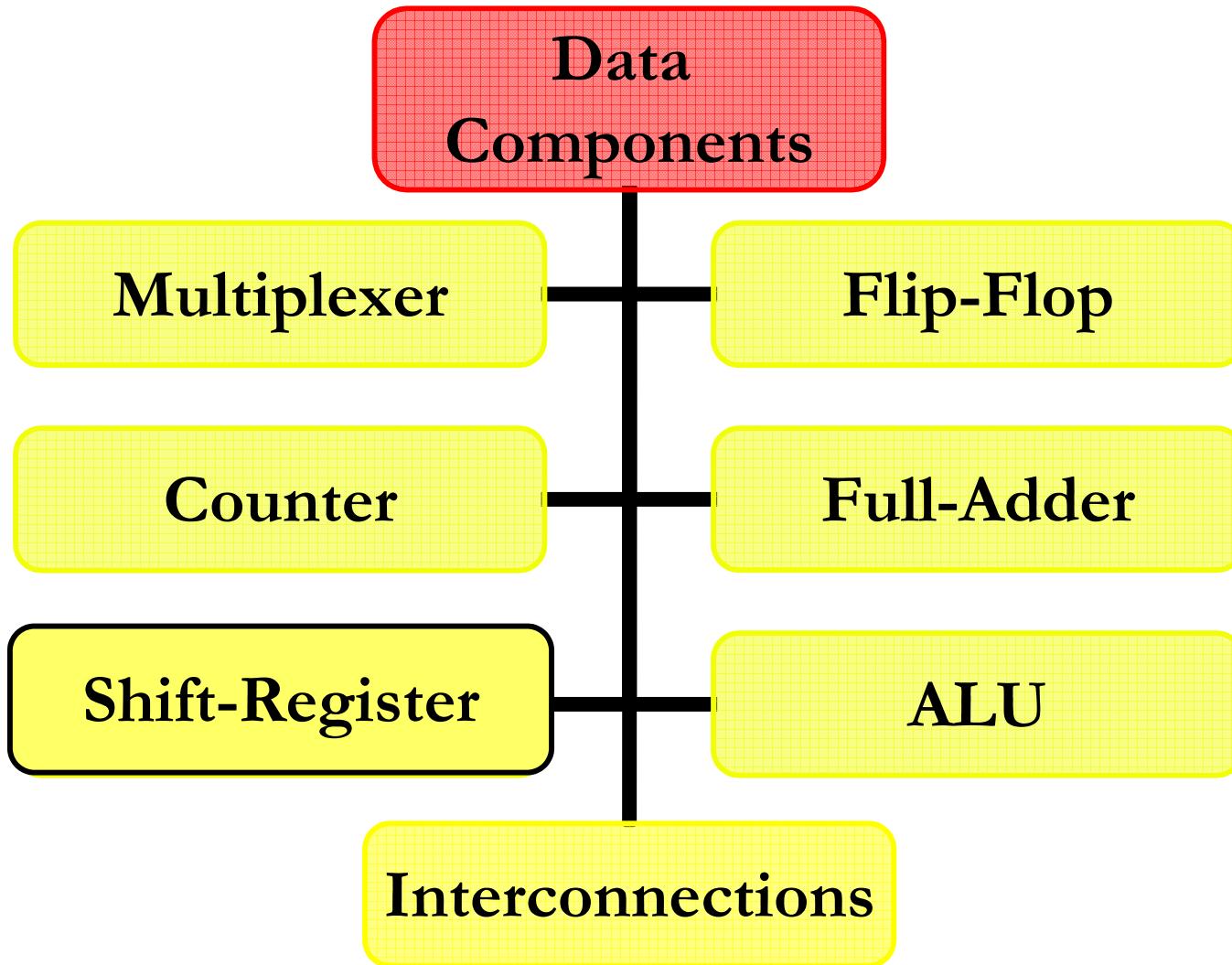
```
`timescale 1ns/100ps

module fulladder (input a, b, cin, output sum, cout);
    assign #5 sum = a ^ b ^ cin;
    assign #3 cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

One delay for  
every output:  
 $t_{PLH}$  and  $t_{PHL}$

All Changes  
Occur after 3 ns

# Shift-Register



# An 8-bit Universal Shift Register

2 Mode inputs  
 $m[1:0]$  form a 2-bit number

```
module ShiftRegister8
  (input sl, sr, clk,
   input [1:0] m, output [7:0] ParOut);
  always @ (negedge clk) begin
    case (m)
      0: ParOut <= ParOut;
      1: ParOut <= {sl, ParOut [7:1]};
      2: ParOut <= {ParOut [6:0], sr};
      3: ParOut <= ParIn;
    default: ParOut <= 8'b0;
    endcase
  end
endmodule
```

Case Statement  
With 4 case-alternatives  
and default Value

In,  
arOut);

$m=0$ : Does Nothing

$m=1,2$ : Shifts Right  
and Left

$m=3$ : Loads its Parallel  
input into the register

# Shift-Register (Continued)

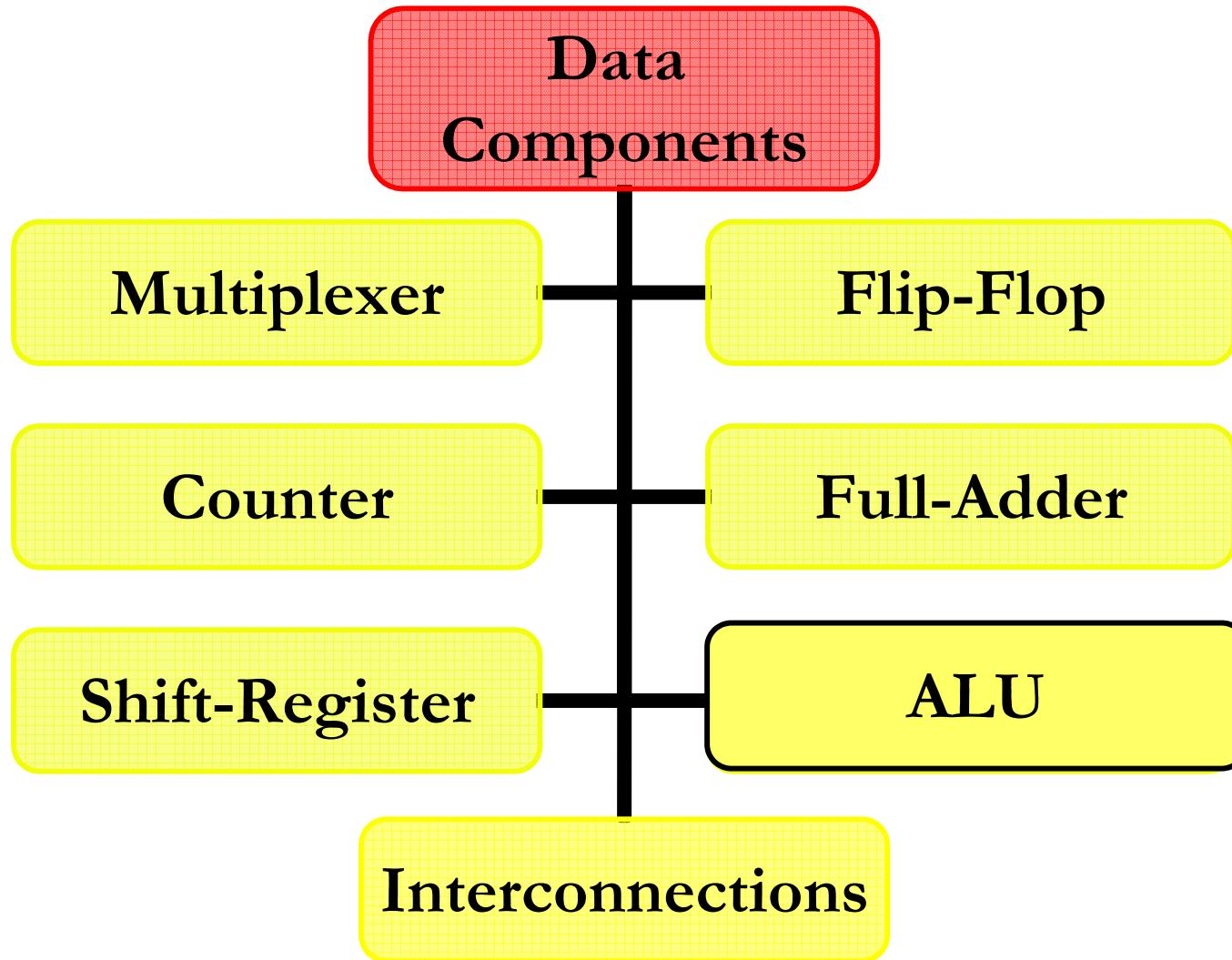
```
`timescale 1ns/100ps

module ShiftRegister8
  (input sl, sr, clk, input [7:0] ParIn,
   input [1:0] m, output reg [7:0] ParOut);

  always @ (negedge clk) begin
    case (m)
      0: ParOut <= ParOut;
      1: ParOut <= {sl, ParOut [7:1]};
      2: ParOut <= {ParOut [6:0], sr};
      3: ParOut <= ParIn;
      default: ParOut <= 8'bX;
    endcase
  end
endmodule
```

Shift Right:  
The *SL* input is concatenated to the left of *ParOut*

Shifting the *ParOut* to the left



```
`timescale 1ns/100ps

module ALU8 (input [7:0] left, right,
               input [1:0] mode,  

               output reg [7:0] ALUout);
    always @ (left, right, mode) begin
        case (mode)
            0: ALUout = left + right;
            1: ALUout = left - right;
            2: ALUout = left & right;
            3: ALUout = left | right;
        default: ALUout = 8'bX;
    endcase
end
endmodule
```

2-bit *mode* Input  
to select one of its  
4 functions

Add  
Subtract  
AND  
OR

- An 8-bit ALU

# ALU (Continued)

```
'timescale 1ns/100ps

module ALU8 (input [7:0] left, right,
               input [1:0] mode,
               output reg [7:0] ALUout);
    always @ (left, right, mode) begin
        case (mode)
            0: ALUout = left + right;
            1: ALUout = left - right;
            2: ALUout = left & right;
            3: ALUout = left | right;
        default: ALUout = 8'bX;
    endcase
end
endmodule
```

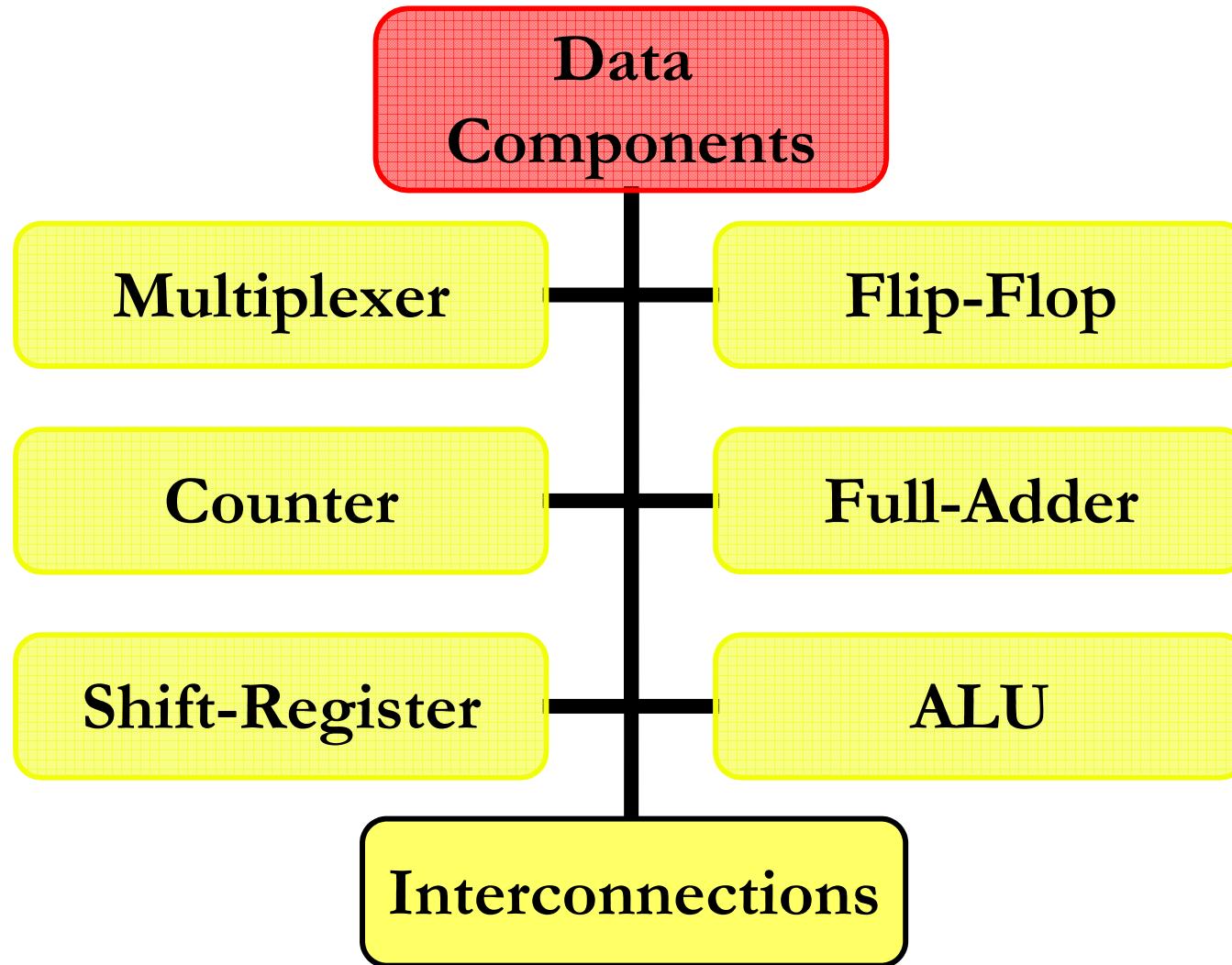
The Declaration of *ALUout* both as *output* and *reg*:  
Because of assigning it within a Procedural Block

Blocking Assignments

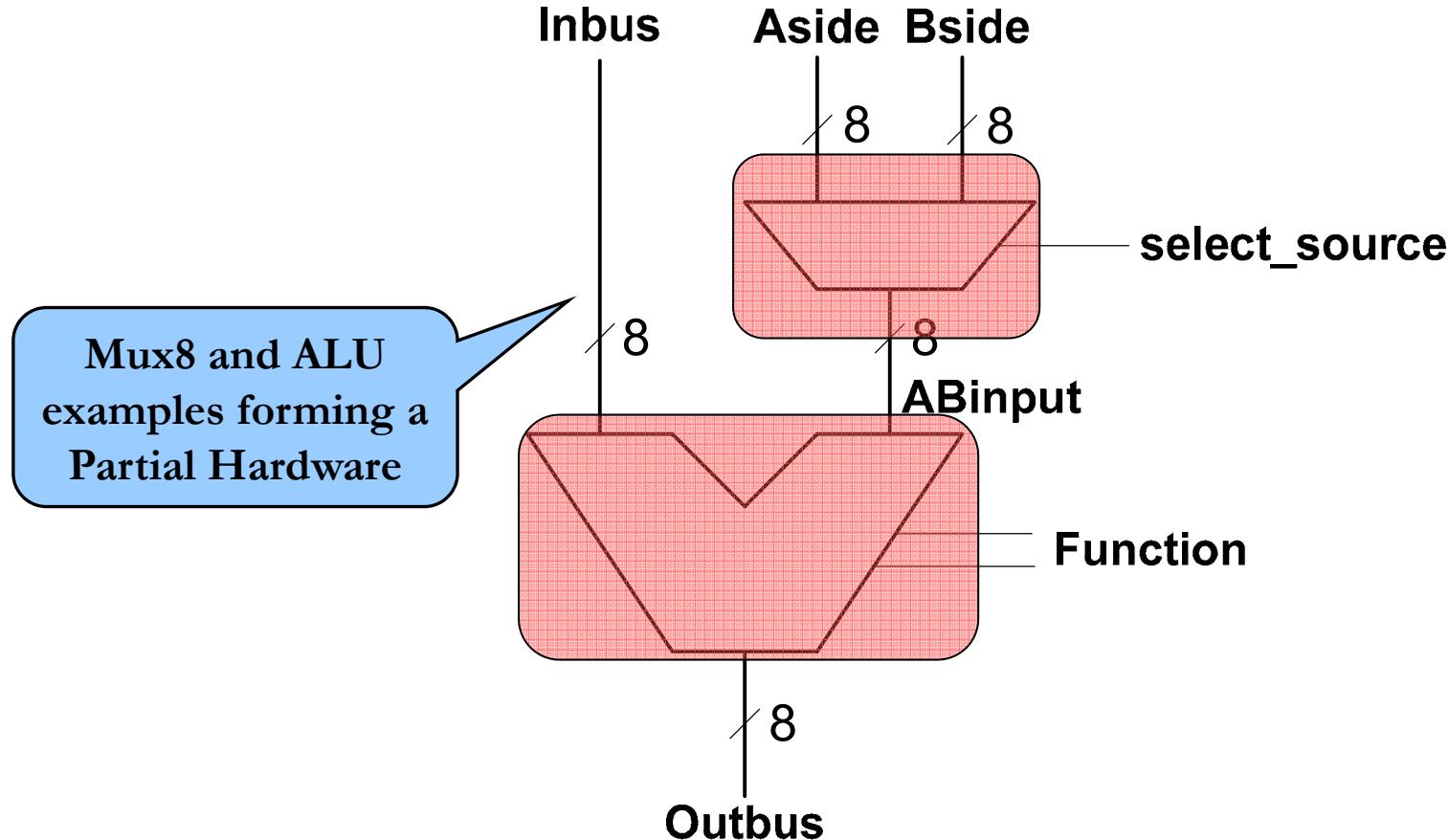
*default* alternative puts all *Xs* on *ALUOut* if *mode* contains anything but *1s* and *0s*

- An 8-bit ALU

# Interconnections



# Interconnections



- Partial Hardware Using *MUX8* and *ALU*

# Interconnections

Instantiation of  
*ALU8* and *MUX8*

A Set of parenthesis  
enclose port  
connections to the  
instantiated modules

```
ALU8 U1 ( .left(Inbus), .right(ABinput),  
           .mode(function), .ALUout(Outbus) );  
Mux8 U2 ( .sel(select_source), .data1(Aside),  
           .data0(Bside), .bus1 (ABinput));
```

*u1* and *u2*:  
Instance Names

# Interconnections

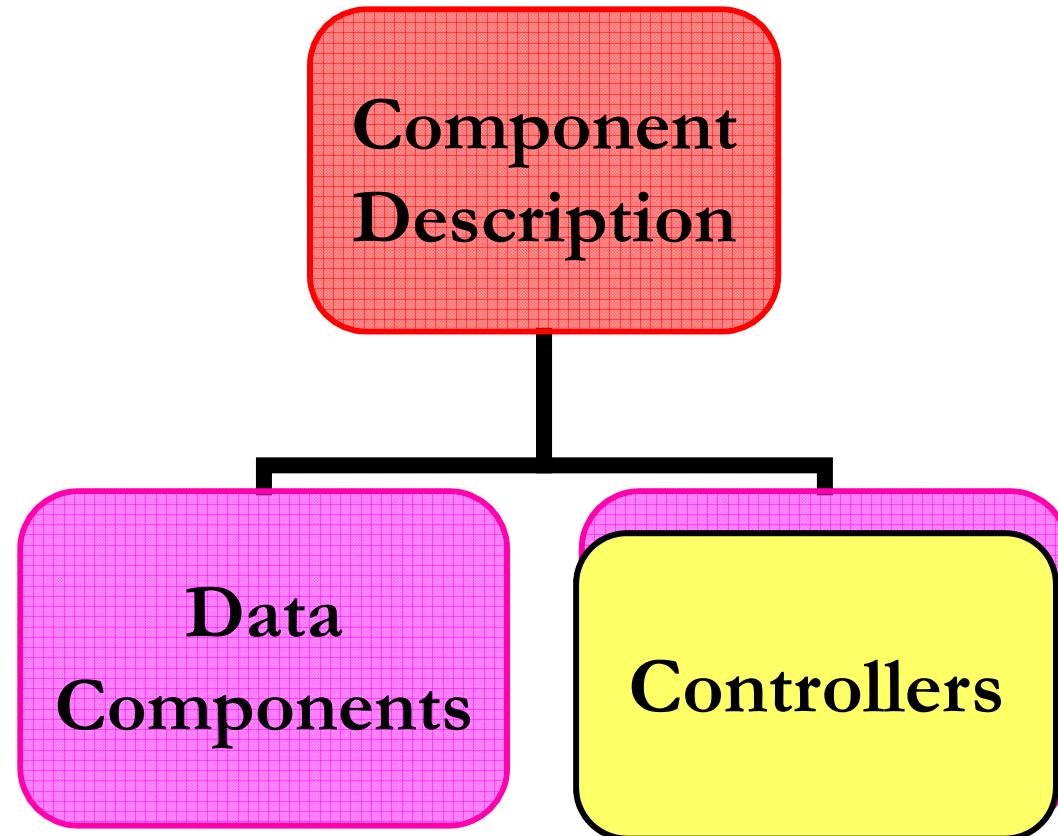
An Alternative format  
of port connection

The actual ports  
of the instantiated  
components  
are excluded

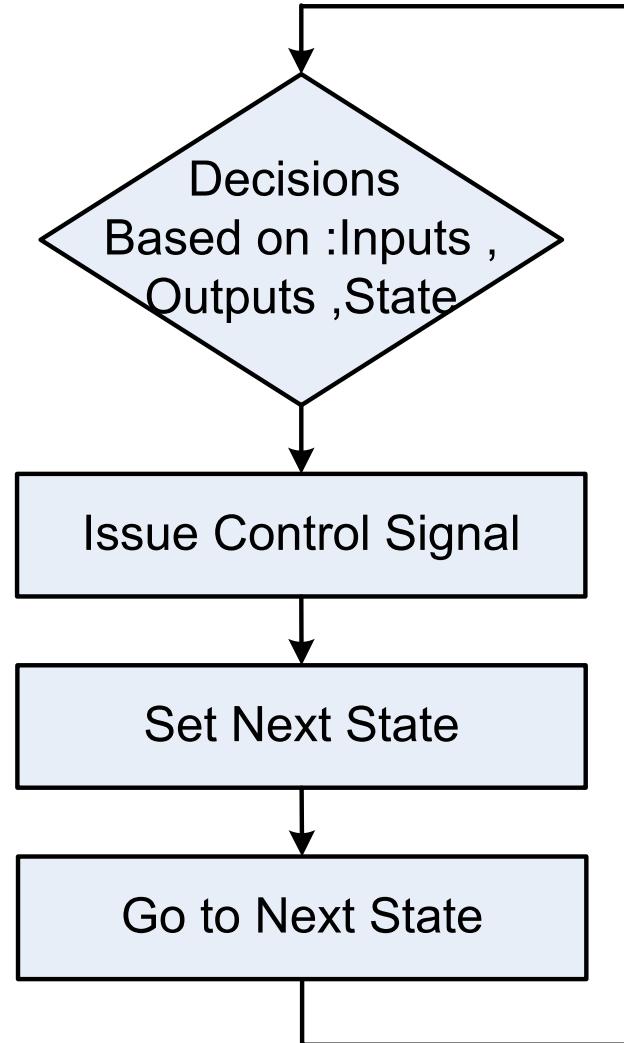
```
ALU8 U1 ( Inbus, ABinput, function, Outbus );  
Mux8 U2 ( select_source, Aside, Bsides, ABinput );
```

- Ordered Port Connection

The list of local signals  
in the same order as  
their connecting ports



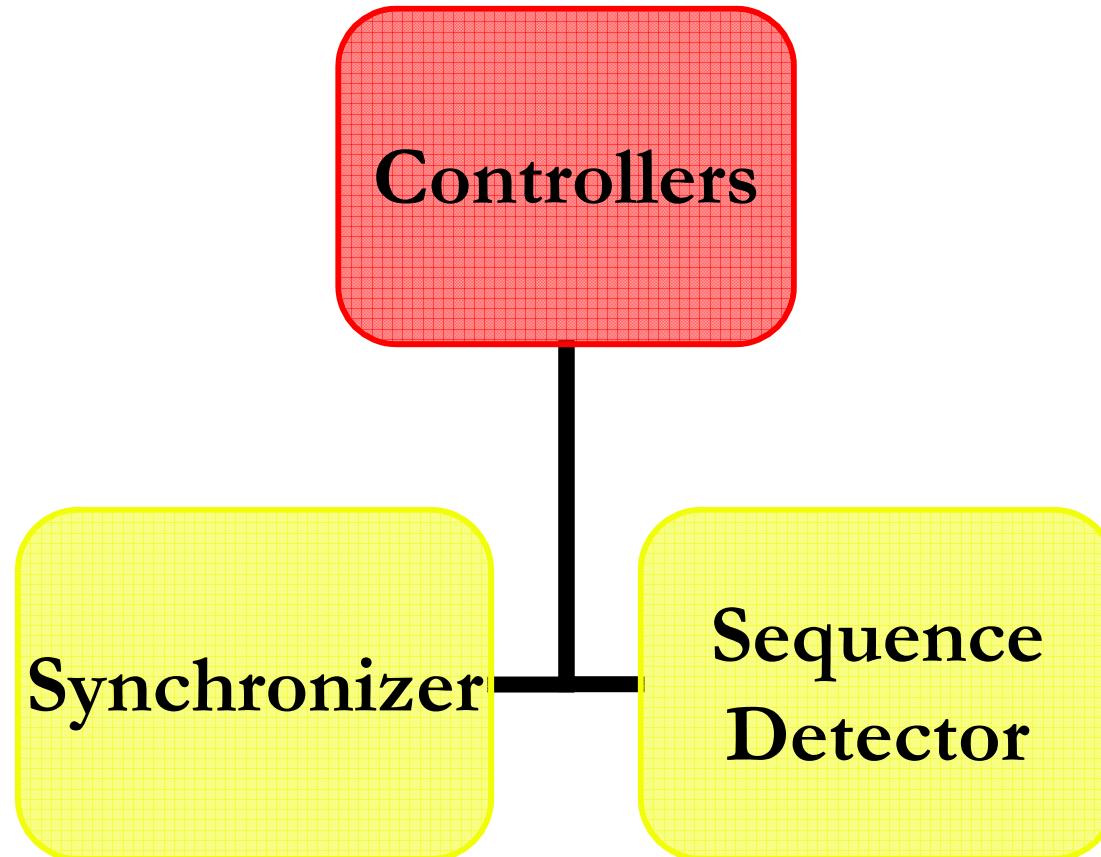
# Controllers



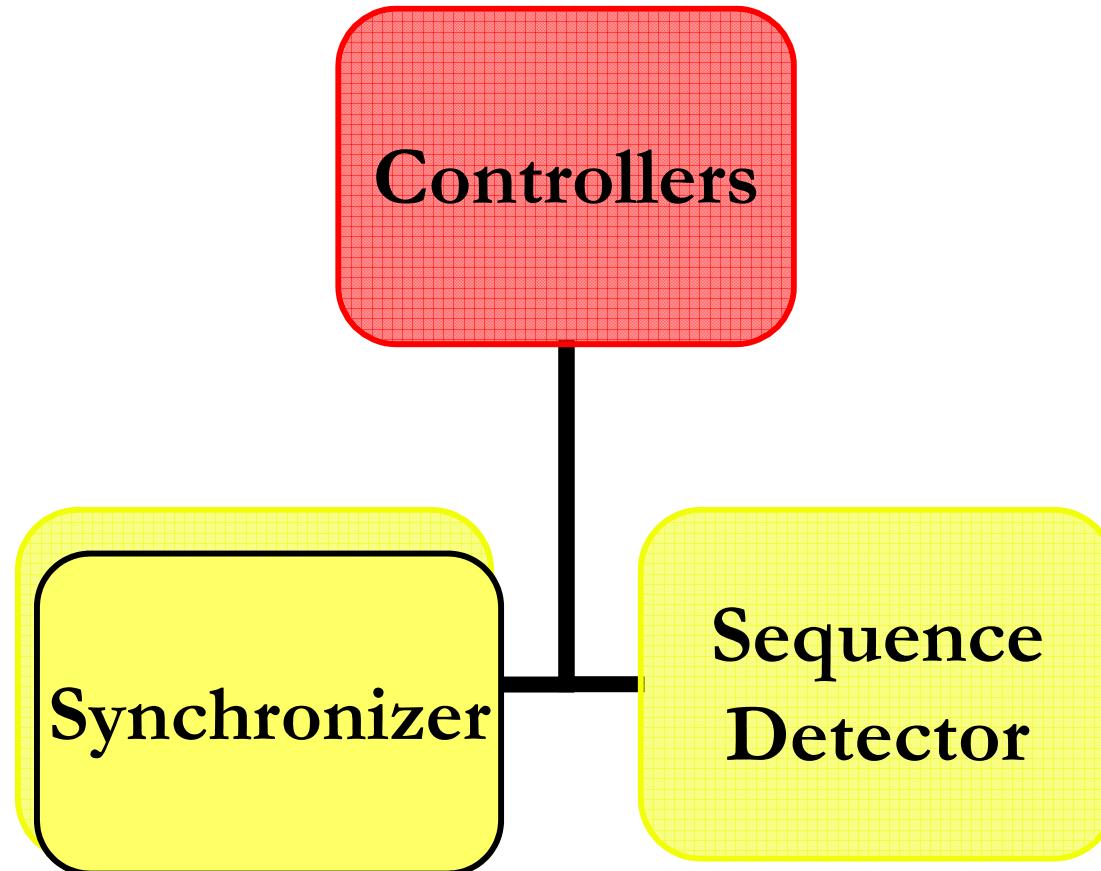
- Controller Outline

# Controllers

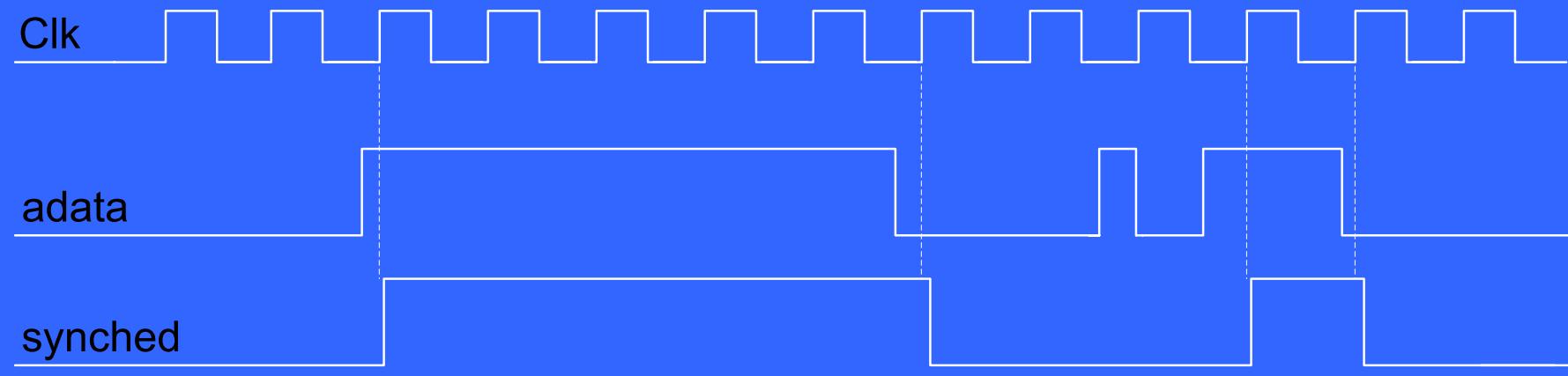
- **Controller:**
  - Is wired into data part to control its flow of data.
  - The inputs to it controller determine its next states and outputs.
  - Monitors its inputs and makes decisions as to when and what output signals to assert.
  - Keeps the history of circuit data by switching to appropriate states.
- **Two examples to illustrate the features of Verilog for describing state machines:**
  - Synchronizer
  - Sequence Detector



# Synchronizer



# Synchronizer



- **Synchronizing *adata***

# Synchronizer

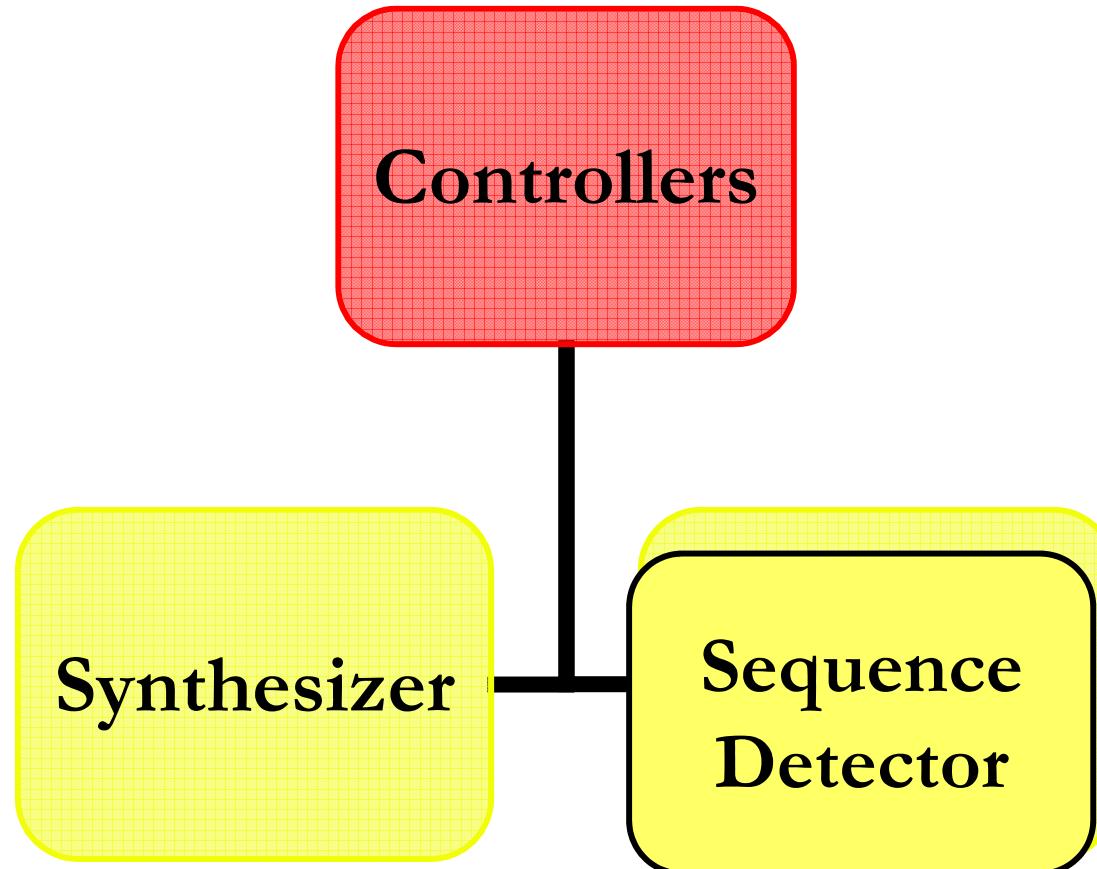
```
`timescale 1ns/100ps

module Synchronizer (input clk, adata,
                      output reg synched);
    always @ (posedge clk)
        if (adata == 0) synched <= 0;
        else synched <= 1;
endmodule
```

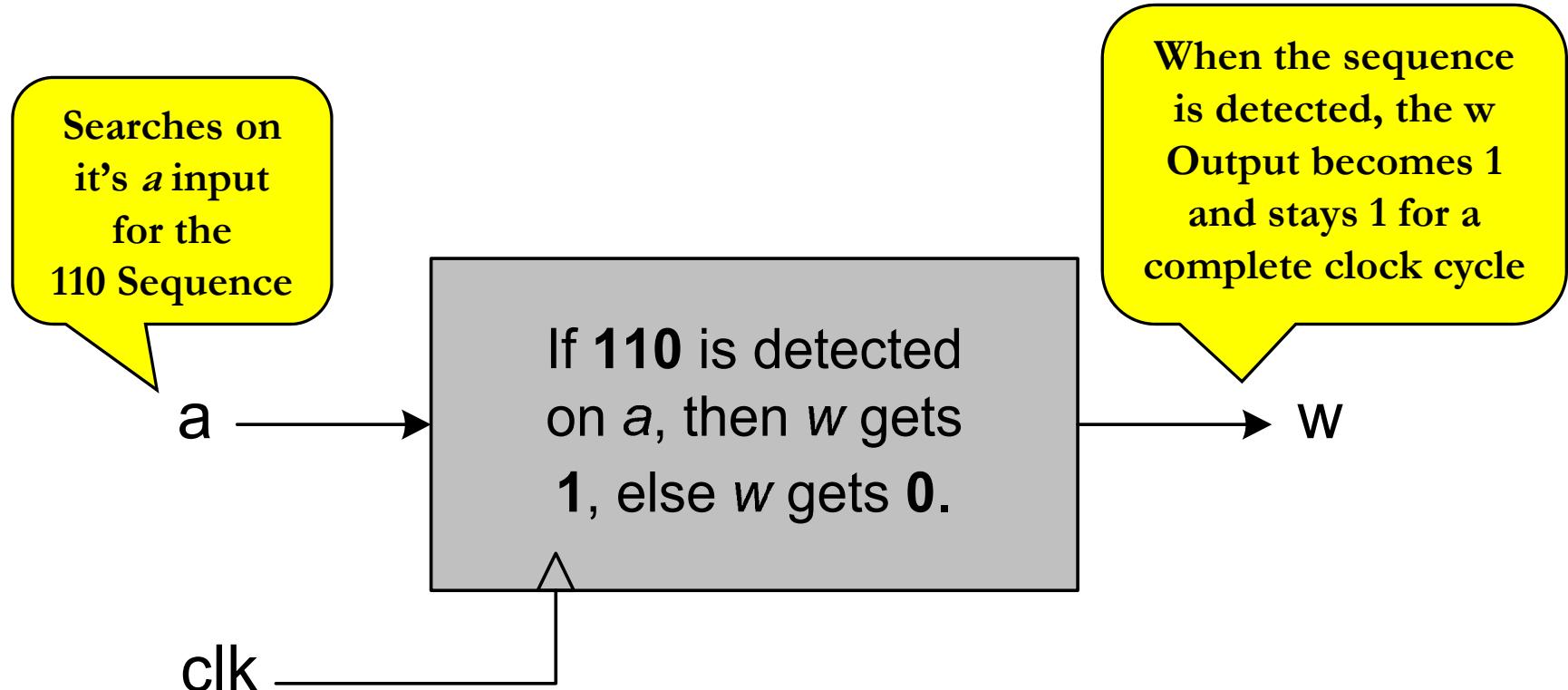
- A Simple Synchronization Circuit

If a 1 is Detected on *adata* on the rising edge of clock, *synched* becomes 1 and remains 1 for at least one clock period

# Sequence Detector



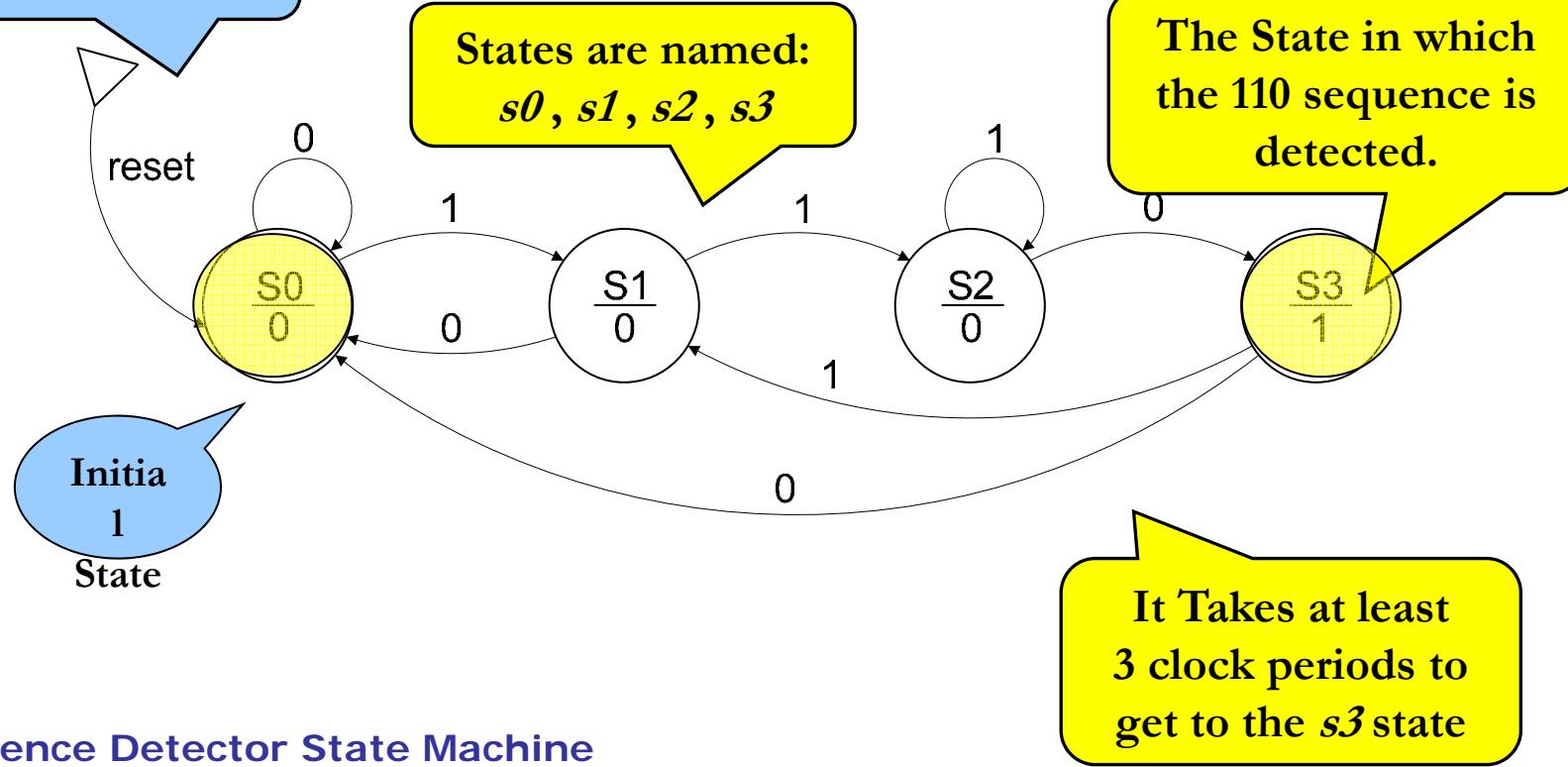
# Sequence Detector



- State Machine Description

# Sequence Detector

A Moore Machine  
Sequence Detector



# Sequence Detector

```
module Detector110 (input a, clk, reset, output w);
    parameter [1:0] s0=2'b00, s1=2'b01, s2=2'b10, s3=2'b11;
    reg [1:0] current;

    always @(posedge clk) begin
        if (reset) current = s0;
        else
            case (current)
                s0: if (a) current <= s1; else current <= s0;
                s1: if (a) current <= s2; else current <= s0;
                s2: if (a) current <= s2; else current <= s3;
                s3: if (a) current <= s1; else current <= s0;
            endcase
    end

    assign w = (current == s3) ? 1 : 0;
endmodule
```

- Verilog Code for 110 Detector

# Sequence Detector

Behavioral  
Description of the  
State Machine

```
module Detector110 (input a, clk, reset, output w);  
  
parameter [1:0] s0=2'b00, s1=2'b01, s2=2'b10,  
           s3=2'b11;  
  
reg [1:0] current; // A 2-bit Register  
  
always @ (posedge clk) begin  
    if (reset) current = s0;  
    else  
        .....  
        .....
```

Parameter declaration  
defines constants  
*s0, s1, s2, s3*

- Verilog Code for 110 Detector

# Sequence Detector

```
.....  
.....  
always @(posedge clk) begin  
    if (reset) current = s0;  
    else  
        case (current)  
            s0: if (a) current <= s1; else current <= s0;  
            s1: if (a) current <= s2; else current <= s0;  
            s2: if (a) current <= s2; else current <= s3;  
            s3: if (a) current <= s1; else current <= s0;  
        endcase  
    end
```

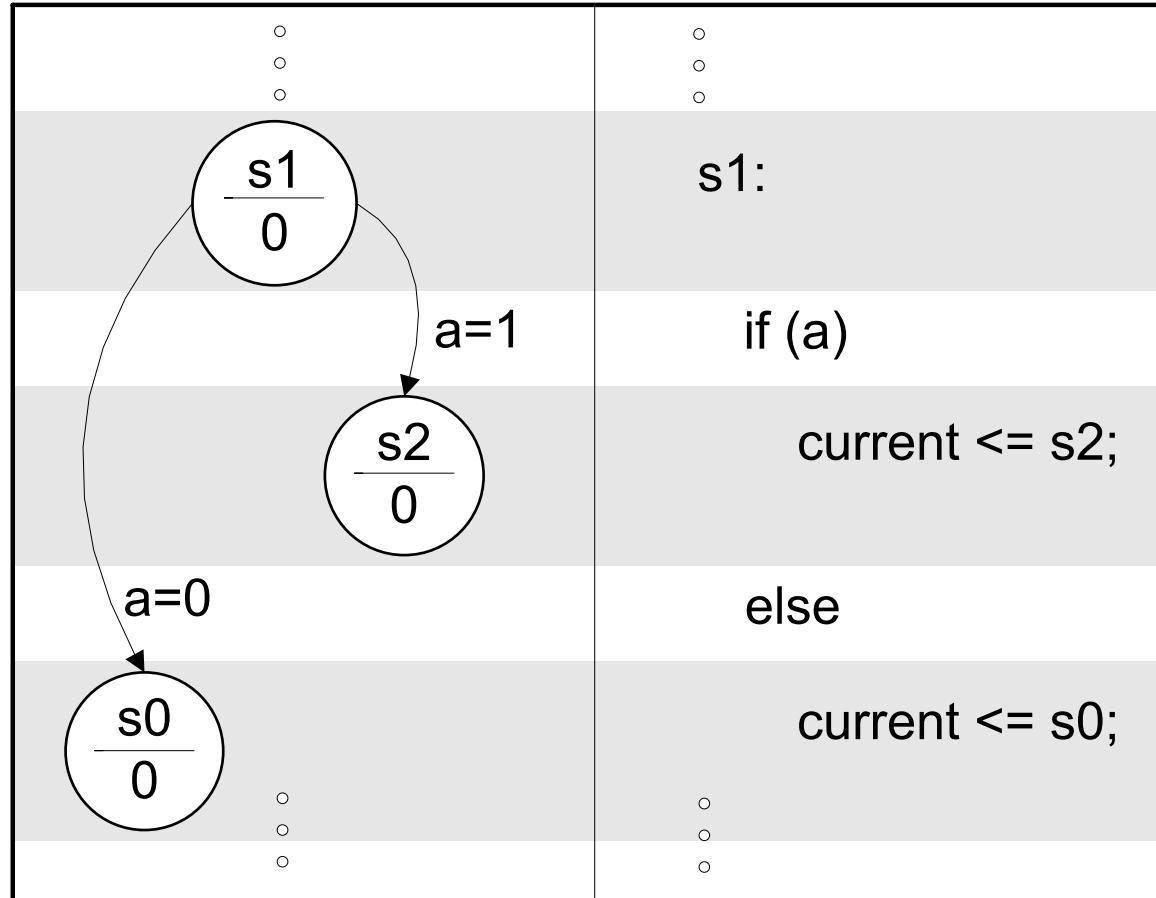
*if-else* statement  
checks for *reset*

At the  
Absence of  
a 1 on *reset*

The 4 Case-alternatives  
each correspond to a  
state of state machine

- Verilog Code for 110 Detector

# Sequence Detector



- State Transitions on Corresponding Verilog Code

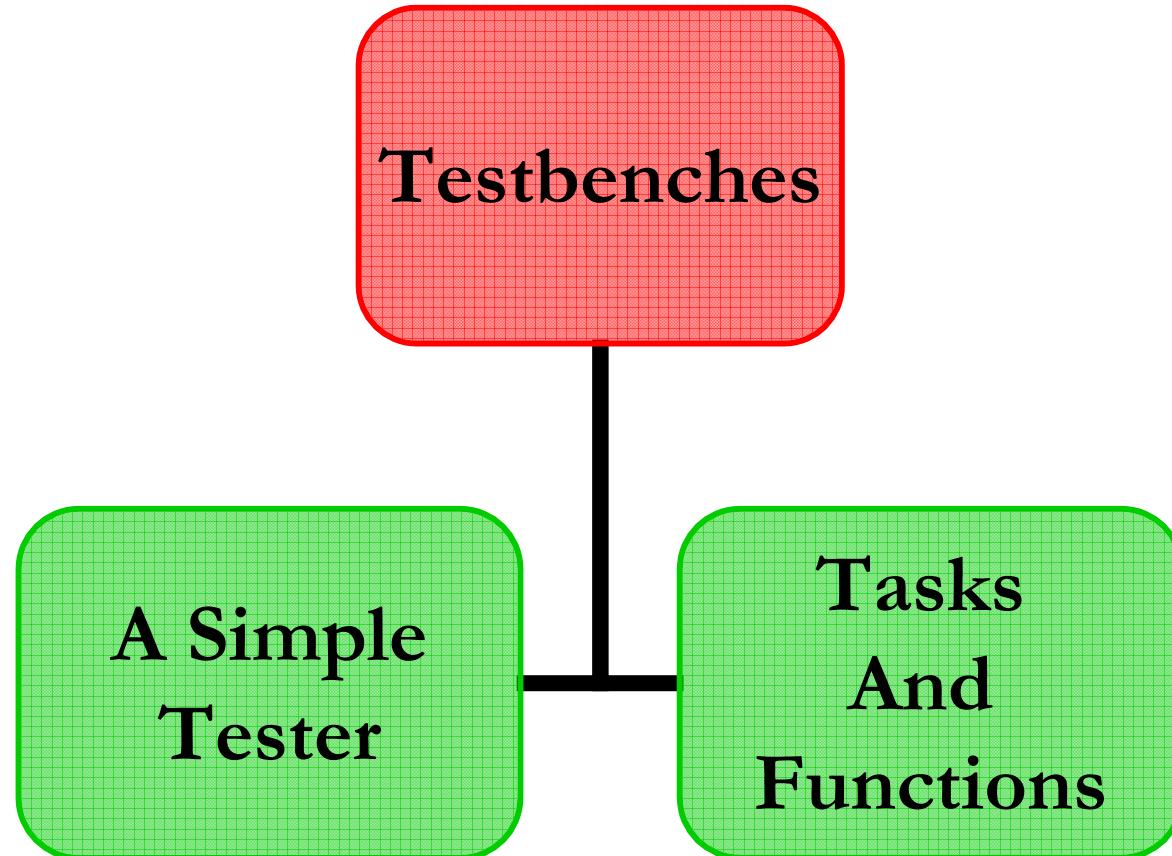
# Sequence Detector

```
end  
.....  
.....  
assign w = (current == s3) ? 1 : 0;  
endmodule
```

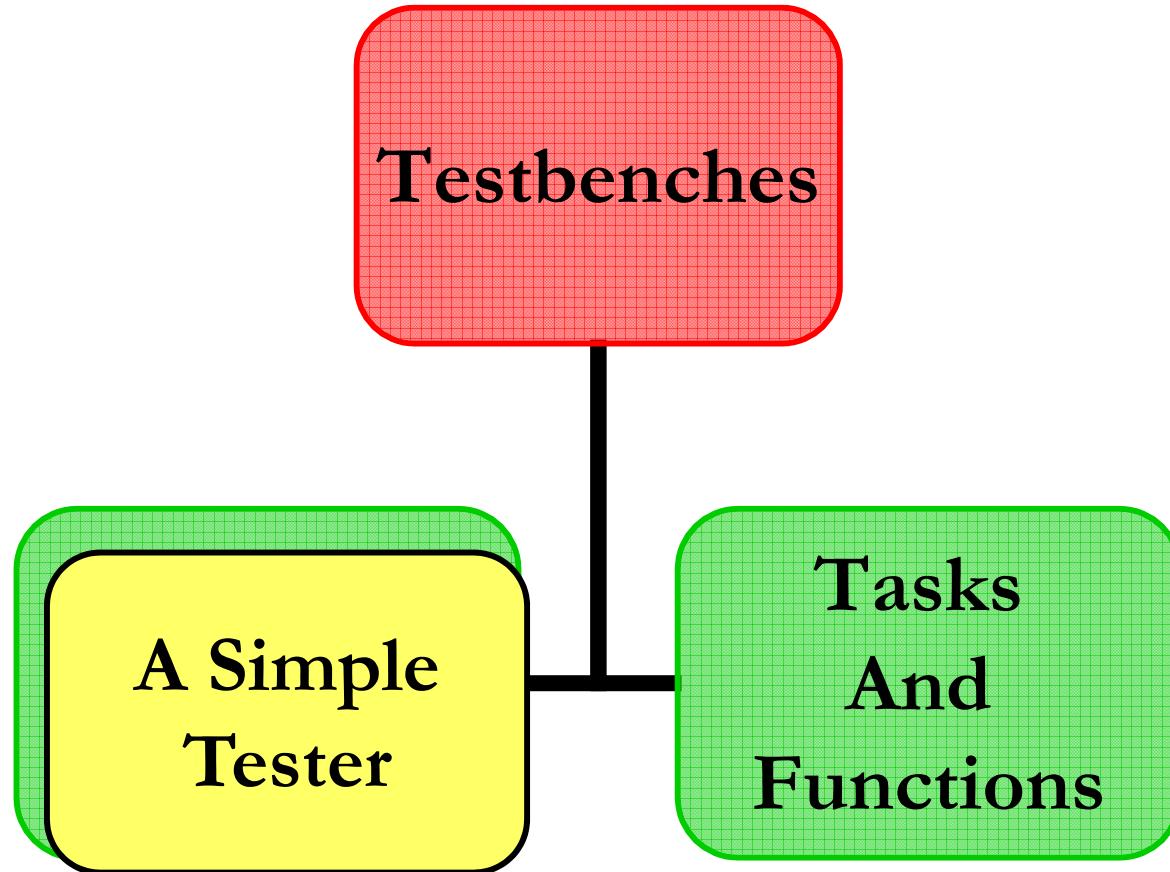
Outside of the  
*always* Block:  
A combinational  
circuit

Assigns a 1 to w  
output when  
Machine Reaches to  
*s3* State

- Verilog Code for 110 Detector



# A Simple Tester



# A Simple Tester

```
`timescale 1ns/100ps

module Detector110Tester;
    reg aa, clock, rst;
    wire ww;
    Detector110 UUT (aa, clock, rst, ww);
    initial begin
        aa = 0; clock = 0; rst = 1;
    end
    initial repeat (44) #7 clock = ~clock;
    initial repeat (15) #23 aa = ~aa;
    initial begin
        #31 rst = 1;
        #23 rst = 0;
    end
    always @(ww) if (ww == 1)
        $display ("A 1 was detected on w at time = %t",
        $time);
endmodule
```

- Testbench for *Detector110*

# A Simple Tester

Begins with  
the *module*  
keyword

```
timescale 1ns/100ps
```

Unlike other  
descriptions  
doesn't have input  
or output ports

```
module Detector110Tester;  
    reg aa, clock;  
    wire ww;  
    Detector110 UUT (aa, clock, rst, ww);  
    .....  
    .....
```

Outputs are  
declared as *wire*

Inputs are Declared as *reg*

The Instantiation of  
*Detector110* Module

- Testbench for *Detector110*

*initial* statement drives test values into the variables connected to the inputs.

```
...  

initial begin
    aa = 0; clock = 0; rst = 1;
end
initial repeat (44) #7 clock = ~clock;
initial repeat (15) #23 aa = ~aa;
initial begin
    #31 rst = 1;
    #23 rst = 0;
end
```

An *initial* statement: A sequential statement that runs once and stops when it reaches its last statement

All Initial Blocks Start at Time 0 and Run Concurrently

- Testbench for *Detector110*

# A Simple Tester

```
.....  
.....  
initial begin  
    aa = 0; clock = 0; rst = 1;  
end  
initial repeat (44) #7 clock = ~clock;  
initial repeat (15) #23 aa = ~aa;  
initial begin  
    #31 rst = 1;  
    #23 rst = 0;  
end
```

- Testbench for

Waits 31 ns before assigning 1 to *rst*

For Initial  
the Input S

Repeats 44 times of  
complementing the  
*clock* input with 7ns  
delay, generates a  
periodic signal  
on *clock*

Signal *aa* is also  
assigned a periodic  
signal, with a  
different frequency

# A Simple Tester

```
always Block  
Wakes up when  
ww Changes
```

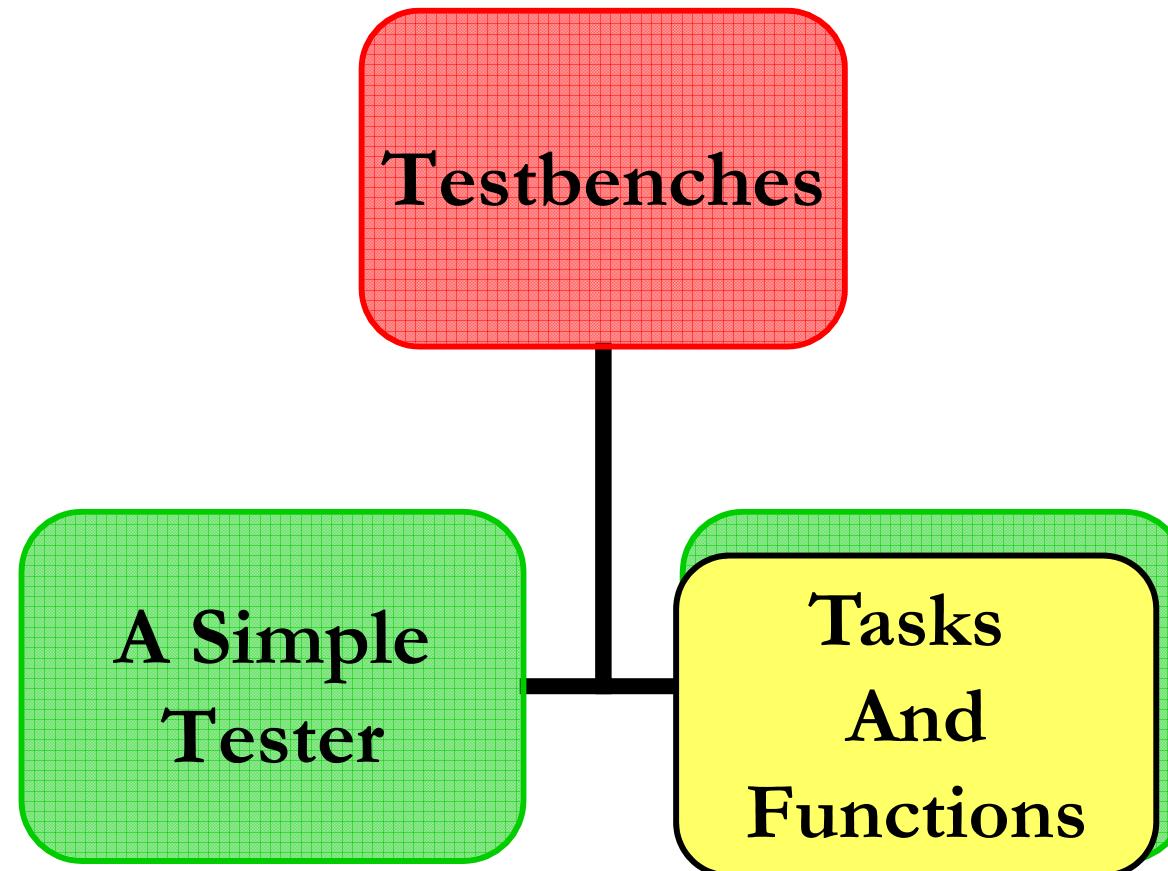
```
Reports the  
Times at which  
the ww Variable  
becomes 1
```

```
....  
....  
always @(ww) if (ww == 1)  
$display ("A 1 was detected on w at time = %t",  
$time);  
endmodule
```

- Testbench for *Detecting 110*

A Verilog  
System Task

This Note Will Appear  
in the Simulation  
Environment's  
Window: "Console" or  
"Transcript"



- **Verilog Tasks and Functions:**
  - System tasks for Input, Output, Display, and Timing Checks
  - User defined tasks and functions
- **Tasks:**
  - Can represent a sub
  - Begins with a **task** keyword
  - Its body can only consist of sequential statements like **if-else** and **case**
- **Functions:**
  - Can be used for corresponding to hardware entities
  - May be used for writing structured codes
  - Applications: Representation of Boolean functions, data and code conversion, and input and output formatting

# Synthesizable Verilog

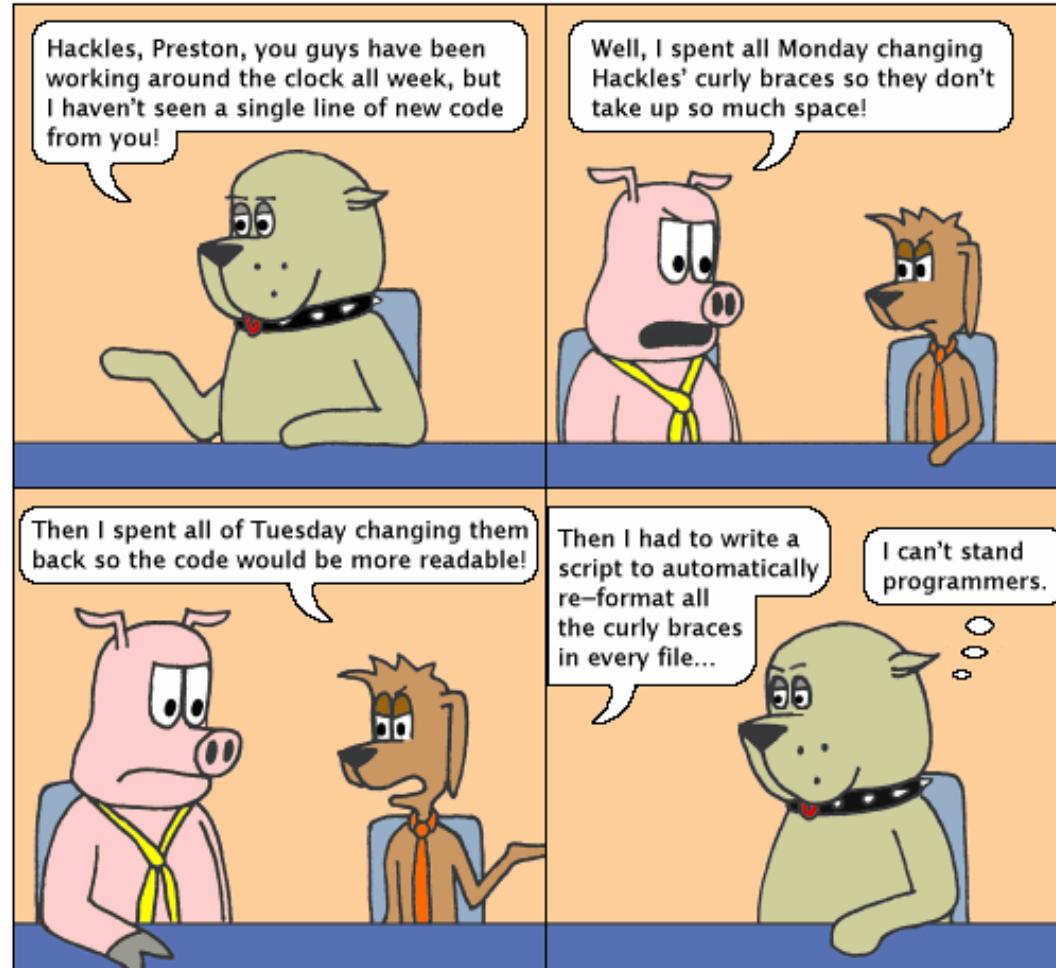
- Most Verilog language constructs are synthesizable (i.e. can be implemented as Hardware gates). This may even be dependent on the Synthesis tool used (e.g. Synopsys DC, or FPGAExpress)
- In general, the following constructs are not supported by a synthesis tool:
  - Wait construct
  - Repeat, fork, join
  - Data types: time, real, realtime
  - User defined primitives
  - Initial (a one-time sequential active flow)
  - Delay operator (#)
  - Switch level primitives: \*mos where \* is n, p, c, rn, rp, rc; pullup, pulldown;  
\*tran+ where \* is (null), r and + (null), if0, if1 with both \* and + not (null)
  - 7-signal strength (or higher) logic values
  - Tri-State Net definitions (such as triand, trior, tri0, tri1, trireg; but wand, wor, supply0, supply1 are!)
  - Some operators (/, %, ===, !==)

# Coding Style

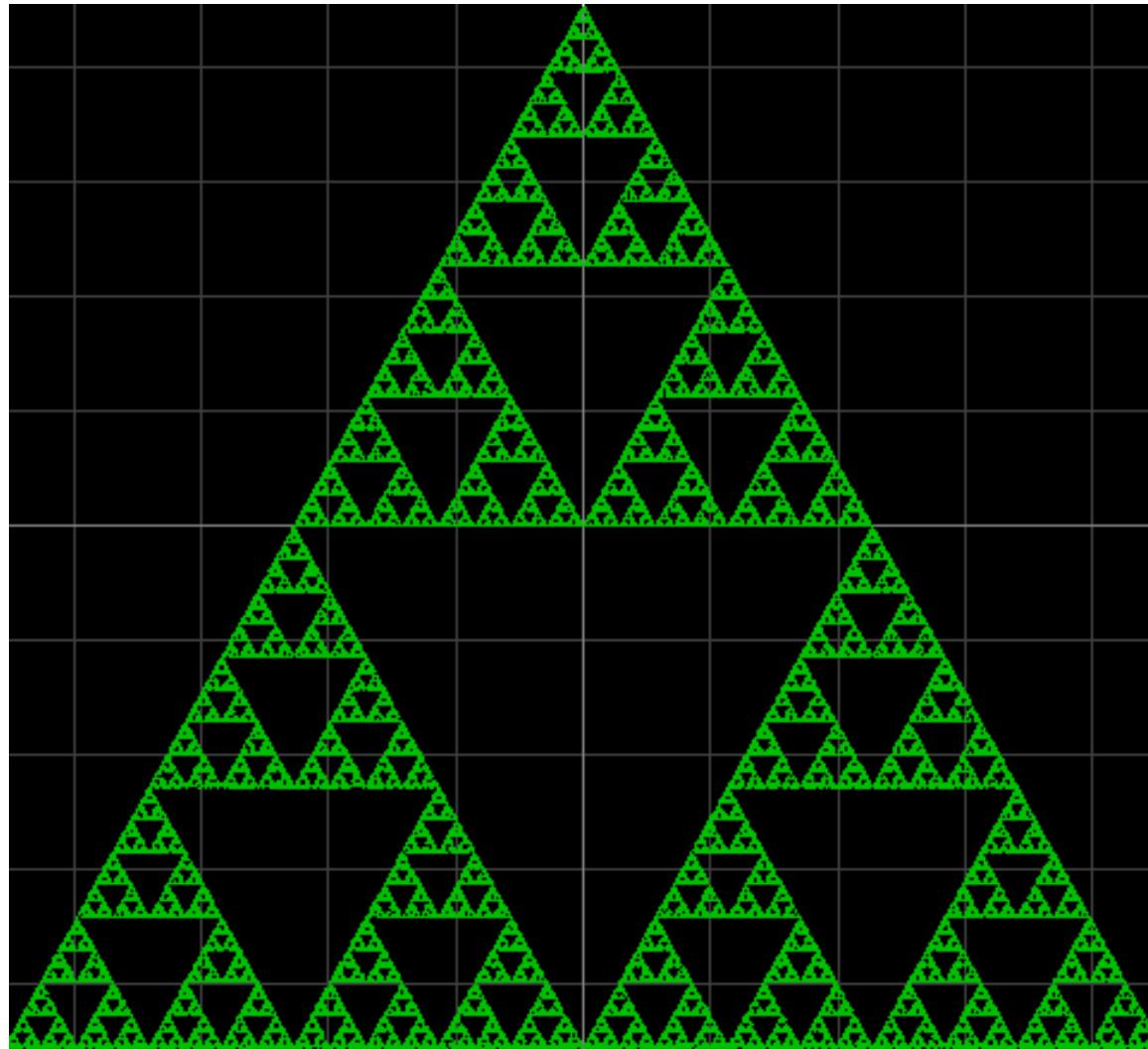


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Coding Style



# Coding Style



# Coding Style

- Uniform Coding
- Clock and Reset
- Coding for Synthesis
- Synthesis Partitioning

# Uniform Coding

- **Naming Convention:**

Develop and document the naming style to be used for the entire design.

1. Use lowercase letters for all signal names, variable names, port names, compound names and instance names.

- `data_in` Signal
- `ram_adder` Compound
- `u_ram_adder_12` Instance.

2. Use an underscore ("\_") to separate parts of a name. But don't use the underscore as a first or last character.

- `data_out`

# Uniform Coding

3. Meaningful user defined names improve readability.
  - Do not use **dr** for a data requested.
  - Instead use **data\_requested**.
4. Choose to use either a noun-verb or verb-noun convention but not both.
  - **data\_requested** or **requested\_data**

# Uniform Coding

5. Use the name **clk** for the clock signal. If there is more than one clock in the design, use **clk** as the prefix for all clock signals.
  - clk1
  - clk2
  - clk\_10mhz
6. Use the same name for all clock signals that are driven from the same source
7. For active low signals, use the suffix "\_n".
  - **rst\_n**

# Uniform Coding

8. When describing multi-bit buses, use a consistent order of bits.  
The preference may be for VHDL ( $y$  downto  $x$ ) and Verilog ( $y:x$ )  
with  $x < y$ .

The recommendation is to establish a standard, and thus achieve some consistency across multiple design and design teams.

Verilog

$(y:x)$  with  $x \leq y$

VHDL

$(y \text{ downto } x)$

# Uniform Coding

9. Instead of using hard-coded numeric values use constants, parameters for Verilog, and generics for VHDL

Verilog

```
parameter CHIP_ID = 8'b11110000;  
. . .  
if (decoded_addr == CHIP_ID)
```

VHDL

```
constant CHIP_ID :std_logic_vector (7 downto 0) :=  
'11110000';  
. . .  
if (decoded_addr == CHIP_ID)
```

# Uniform Coding

10. If the algorithm modeled by a Verilog module or VHDL entity can be generalized for arbitrary input and output bandwidths, then the ports may be declared with index ranges that are bounded by user defined parameter for Verilog or generic parameters for VHDL .

## Verilog

```
module interface (clk, data_in, data_out);
parameter WIDTH = 5;
input clk;
input [WIDTH -1 :0] data_in;
output [WIDTH -1 :0] data_out;
endmodule
```

## VHDL

```
entity interface is
generic (WIDTH : integer);
port (clk : in std_logic;
data_in : in std_logic_vector(WIDTH - 1 downto 0);
data_out : out std_logic_vector(WIDTH - 1 downto 0));
end interface;
```

# Uniform Coding

11. Ports should be listed in a specific order, functionally grouped, and declared one port per line.
  1. Inputs
    - Clocks (functionally grouped)
    - Resets (functionally grouped)
    - Enables (functionally grouped)
    - Other control signals (functionally grouped)
    - Data and address lines (functionally grouped)
  2. Output
    - Clocks (functionally grouped)
    - Resets (functionally grouped)
    - Enables (functionally grouped)
    - Other control signals (functionally grouped)
    - Data and address lines (functionally grouped)
  3. Bi-directional
    - IO ports (functionally grouped)

# Uniform Coding

12. Always use explicit mapping for ports, and generics, using named association rather than positional association.

Verilog

```
.ram_addr (ram_addr),  
.data_in (data_in),
```

VHDL

```
ram_addr => ram_addr;  
data_in => data_in,
```

# Uniform Coding

- **File Headers :**

A descriptive header should be written for every source file and script. Having a uniform header provides information about the file, and generates profession looking code. This is especially important if the HDL code is going to be release to a customer. The header should include the following.

1. Company's copyright notification Filename
2. Author
3. Date
4. Version number
5. Description of function
6. Modification history

# Uniform Coding

- **File Comments:**

Without the use of comments to describe the functionality it is sometimes impossible to figure out the contents of a program or module. The correct use of comments can help remove this problem and is useful for your own purposes.

1. Place comments logically near the code that they describe. Comments should be brief, concise, and explanatory.

2. Use comments appropriately to explain all process, functions, and declarations of types and subtypes.

```
--Create subtype INTEGER_256 for built-in error  
--checking of legal values
```

```
subtype INTEGER_256 is type integer range 0 to 255;
```

3. Use comments to explain ports, signals, and variables, or groups of signals or variables.

# Uniform Coding

- **Line Length:** Have you ever looked at someone's code and had to keep scrolling to see the entire line? How annoying is that? Oh, lets not forget about those who like to have multiple commands on one line. These two habits make the code harder to read and maintain. Guidelines need to be established to prevent this from happening.

1. Use a separate line for each HDL statement.
2. For HDL code, use carriage returns to divide lines that exceed 80 characters and indent the next line to show that it is a continuation of the previous line.
3. Add blank lines when needed to improve readability. For example between input and output ports.

1. The easier you can make your clocking scheme the happier you are going to be.
2. A complicated clocking scheme makes everything more difficult: test insertion, writing constraints, static timing analysis, and simulations just to name a few.

- **Mixed Clock Edges:**

1. If at all possible avoid using both positive and negative clock edges.
2. Using both edges just makes life harder: design, functional simulations, writing timing constraints, DFT, and synthesis.
3. If this can't be avoided, here are a couple of things to keep in mind.
  - Allow enough time in the schedule to thoroughly examine the timing using mixed clock edges.
  - For synthesis and timing analysis use the worst -case duty cycle.
  - Document the duty cycle.
  - Consider putting negative edge and positive edge flops into separate modules.

- **Clock buffers:**

1. There is no need to worry about adding clock buffers during the initial design phase and synthesis.
2. Adding clock buffers just adds fat to the code.
3. Besides, clock buffers are normally taken care of during physical design and synthesis treats clocks as ideal nets.
4. Clocks are treated as ideal nets. Buffers are not needed.

- **Gated clocks:**

1. Be careful using gated clocks.
2. Improper timing of a gated clock can generate a glitch causing a flop to clock in the wrong data.
3. In addition, testability can be limited because the logic clocked by the gated clock can't be made part of the scan chain.

- **Internally generated clocks:**

Internally generated clocks can cause testability issues to rise.

The reason is because the logic driven by the internally generated clock can't be made part of the scan chain.

Writing timing constraints for generated clocks becomes more difficult as well.

However in almost every design an internally generated clock is needed.

1. Add in test circuitry to bypass the internally generated clock. For example, if you have a divide-by-two, add in a mux to select a primary input clock over the internally generated one for test. The mux select line should be test\_mode signal coming from a primary input.
2. Fix test violations for internally generated clocks (without proprietary tools like autofix).

# Clock and Resets

- **Internally generated resets :**

Just like internally generated clocks, internally generated resets can cause test problems as well. But, I don't think I've seen a design that didn't require an internally generated reset.

1. Add in test circuitry to bypass the internally generated reset.
2. Fix test violations for internally generated resets (without proprietary tools like autofocus).
3. If possible, have all registers in the macro be reset at the same time. By using this approach it makes analysis and design simpler and easier.
4. If a conditional reset is required, create a separate signal for the reset and isolate the reset logic into a separate module. This approach produces more readable code and improves synthesis results. Each always block should have only one clock and reset.

## When writing HDL code keep in mind the hardware intent.

- When you start writing HDL think about the hardware you wish to produce.
- Draw a picture and sketch out a timing diagram so you'll know exactly what the hardware should look like when the HDL is synthesized.

- **Latches:**

1. If possible, avoid using latches in your design. Using latches can be more difficult to design correctly and to verify.
2. If latches are used, partition the logic in a separate module.
3. You can avoid inferred latches by using any of the following coding techniques.
  - Assign default values at the beginning of a process
  - Assign outputs for all input conditions
  - Use else (instead of elseif) for the final priority branch

- **Latches:**

In Verilog, latches are synthesized for a variable when all the following statements are true.

1. Assignment to the variable occurs in at least one but not all of the branches of a Verilog control statement.
2. Assignment to the variable does not occur on a clock's edge.

# Coding for Synthesis

- **Latches:**

Verilog - latch inferred example

```
always @ (enable or data)
begin
    if (enable)
        begin
            Q= data;
        end
    end
```

# Coding for Synthesis

- **Latches:**

Verilog - latch avoidance example

```
always @ (enable or ina or inb)
begin
    if (enable)
        begin
            data_out = ina;
        end
    else
        begin
            data_out = inb;
        end
    end
```

# Coding for Synthesis

- **Latches:**

Verilog - latch inferred example

```
input [3:0] data_in;
always @ (data_in)
begin
    case (data_in)
        0 : out1 = 1'b1;
        1,3 : out2 = 1'b1;
        2,4,5,6,7 : out3 = 1'b1;
        default : out = 1'b1;
    endcase
end
```

# Coding for Synthesis

- **Latches:**

Verilog - latch avoidance example

```
input [3:0] data_in;
always @ (data_in)
begin
    out1 = 1'b0;
    out2 = 1'b0;
    out3 = 1'b0;
    out4 = 1'b0;
    case (data_in)
        0 : out1 = 1'b1;
        1,3 : out2 = 1'b1;
        2,4,5,6,7 : out3 = 1'b1;
        default : out = 1'b1;
    endcase
end
```

# Coding for Synthesis

- **Flip-Flops:**

For Verilog , flip-flops are inferred when edges occur in an event list of posedge clock or negedge clock.

For Verilog, non-blocking assignments should be used to model synchronous circuits.

Verilog - flip flop inferred example

```
always @ (posedge clock)
begin
    data_out <= data_in;
end
```

- **Synchronous Reset:**

1. Is easy to synthesize.
2. Requires a free-running clock for reset to occur.

Verilog- Flip Flop with synchronous Reset

```
always @ (posedge clock)
begin
    if (reset)
        data_out <= 1'b0;
    else
        data_out <= data_in;
end
```

- **Asynchronous Reset:**

1. Does not require a free-running clock for a reset to occur
2. An asynchronous reset is harder to implement because it is a special signal like a clock. Usually, a tree of buffers is inserted at place and route.
3. Must be synchronously de-asserted in order to ensure that all flops exit the reset condition on the same clock. Otherwise, state machines can reset into invalid states.
4. For both VHDL and Verilog, the asynchronous signal must be in the process and always sensitivity list.

# Coding for Synthesis

- **Asynchronous Reset:**

Verilog- Flip Flop with asynchronous Reset

```
always @ (posedge clock or negedge reset_n)
begin
    if (!reset)
        data_out <= 1'b0;
    else
        data_out <= data_in;
end
```

- **Combinatorial Logic:**

For both Verilog and VHDL,

1. envision the combinational circuit that will be synthesized.
2. avoid combinational feedback that is the looping of combinational processes.
3. when modeling purely combinational logic, ensure signals are assigned in every branch of conditional signal assignments.
4. ensure the sensitivity list of process statements in VHDL and the event list of always statements in Verilog are complete.

# Coding for Synthesis

- **Combinatorial Logic:**

5. For VHDL, do not include the after clause in a signal assignment. This clutters the code and makes it harder to read.

VHDL - after clause used

```
C <= a and b after 10ns;
```

VHDL - after clause removed

```
C <= a and b;
```

# Coding for Synthesis

- **Combinatorial Logic:**

6. For Verilog, do not include delays in assignment statements.

Verilog - Delay used

```
assign #10 c = a & b;
```

Verilog - Delay not included

```
assign c = a & b;
```

7. For Verilog, the always statement is supported by synthesis. The initial statement is not.

# Coding for Synthesis

## ▪ Sensitivity Lists:

1. Specify complete sensitivity lists in each of your process VHDL statements and Verilog always blocks. If you don't use a complete sensitivity list, the behavior of the pre-synthesis design may differ from the post-synthesis netlist.
2. For sequential blocks, the sensitivity list must include the clock signal that is read by the block. If the sequential block uses a reset signal, include the reset signal in the sensitivity list.

Verilog example

```
always @ (posedge clk)
begin
    q <= d;
end
```

VHDL example

```
seq_example: process (clk)
begin
    if (clk'event and clk = '1') then
        a <= d;
    end if ;
```

# Coding for Synthesis

## ▪ Sensitivity Lists:

3. For combinational blocks, the sensitivity list must include every signal that is read by the process.

Verilog example

```
always @ (a or b)
begin
  if (b == 0)
    sum = a + 1;
  else
    sum = a - 1;
end
```

VHDL example

```
comb_ex: process (a, b)
begin
  if (b = '0') then
    sum <= a + 1;
  else
    sum <= a - 1;
  end if;
end process;
```

# Coding for Synthesis

- **Global Variables:**

1. Using non-local variables in tasks or functions may cause a simulation and synthesis mismatch.

In the following example simulation does not re-evaluate func when the value of input c changes:

```
module mod(a, b, c, d);
    input a, b, c;
    output [2:0] d;
    function [2:0] func;
        input in1, in2;
        begin
            func = {in1, in2, c};
        end
    endfunction
    assign d = func(a, b);
endmodule
```

2. Modify the Verilog RTL source to pass all needed variables as inputs to the task or function.

- **Block vs. Non-blocking:**

Verilog only.

1. Blocking assignments execute in sequential order, non-blocking assignments executed concurrently.
2. When writing synthesizable code use non-blocking assignments in sequential blocks (-> always @ (posedge clock) blocks).
3. Use blocking assignments in pure combinational blocks. Otherwise, the simulation behavior of the HDL and gate\_level designs may differ.

- **Signal vs. variable:**

VHDL only.

- During simulations, signal assignments are scheduled for execution in the next simulation cycle.
- Variable assignments take effect immediately, and they take place in the order in which they appear in code.
- When writing synthesizable code, use signals instead of variables to ensure that the simulation behavior of the pre-synthesis design matches post-synthesis netlist.

- **Case vs. if-then-else:**

For VHDL and Verilog

- a case statement infers a single-level multiplexer.
- if-then-else statement infers a priority-encoded, cascaded combination of multiplexers.
- A mux is a faster circuit. If the priority-encoding structure is not required use the case statement rather than if-then-else statement.

# Coding for Synthesis

- **Case vs. if-then-else:**

For VHDL and Verilog

for combinational logic from a case statement, ensure that

- Default outputs are assigned immediately before the case statement  
**OR**
- 2. The outputs are always assigned regardless of which branch is taken through the case statement. This will avoid latches being inferred.
- 3. The default in Verilog case branch is essential to ensure all branch values are covered and avoid inferring latches.
- 4. The others in VHDL default case branch are optional to ensure all branch values are covered.

# Partitioning for synthesis

A Design may have a top-level block showing all the I/O and inside the top-level block smaller blocks.

These smaller blocks may represent design functions for example an interface to a processor, a clock generation block, or an encoding block.

However, for synthesis, you might need to combine or break up these smaller blocks.

This combining or breaking up of blocks is what I'm calling partitioning for synthesis.

Correct design partitioning can enhance the synthesis results, reduce compile time, and simplify the constraint and script files.

# Partitioning for synthesis

- **Design Reuse:**

If you know that some part of your design may be reused, then consider the following

- Thoroughly define and document the design interface.
- Standardize interfaces whenever possible.
- Parameterize the HDL code.
- Group related combinational logic and its destination register together.
- Eliminate glue logic.

# Partitioning for synthesis

- **Design Reuse:**

If you know that some part of your design may be reused, then consider the following

6. For each block of a hierarchical design, register all output signals from the block.
7. Keep critical path logic in a separate module from non-critical path logic.
8. Place sharable resources in the same block. Design Compiler can share these resources but only if the resources belong to the same VHDL process or Verilog always block.
9. Isolate special functions (such as I/O pads, clock generation circuitry, boundary scan logic, and asynchronous logic) from the core logic.
10. For re-usability reasons it is desired to keep the RTL code clean from technology dependent entities.

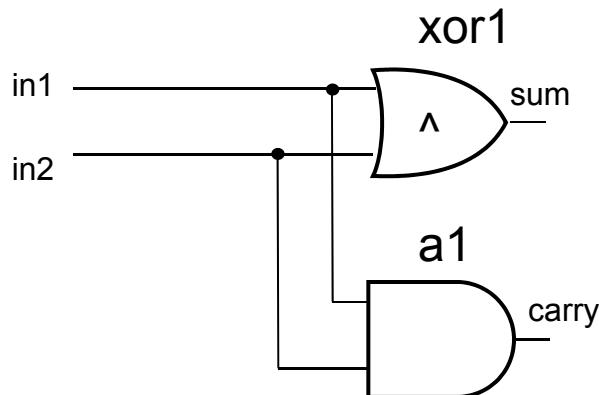
# Quick introduction to VHDL (1993 standard)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Overall idea of VHDL

- VHDL is a strongly typed hardware description language.
- VHDL has (among other ideas) the idea to distinguish between *interface* and *function*
- *Interface* should describe how this component/model is used (e.g. connected, parameterized, type of ports, ...)
- *Function* should define what the component/model is doing.
- Therefore, in contrast to Verilog, VHDL distinguished between *entity* and *architecture* of a component/model



Case insensitive!

```
entity half_Adder is
port ( in1, in2 : in bit;
       sum, carry :out bit);
end entity half_adder;
```

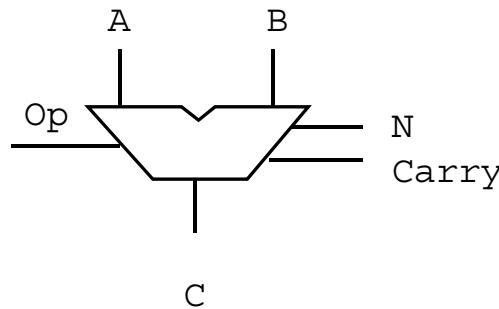
# Entity and Port declaration

- The interface definition of a VHDL model consists mainly of the declaration of ports:
  - Name
  - Type (e.g. bit, real, ...)
  - Mode (in, out, inout)
- VHDL (1993) supports four major objects:
  - Variables
  - Constants
  - Signals
  - Files

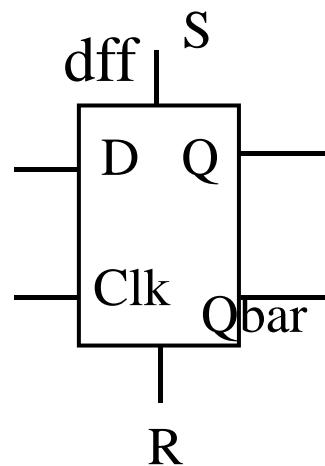
# VHDL Objects

- *Variables* and *Constants* are very much straightforward – like you would expect them (note that Variables are similar to variables in a standard programming language (Java, C/C++)). Variables are like abstract data containers, with no meaning of simulation time.
- *Signals* associate with each variable value also the meaning of time – this is what we need to model digital hardware behavior (very similar to what reg is in Verilog)
- In VHDL, every signal needs at least one driver driving this signal

# Examples for VHDL entities



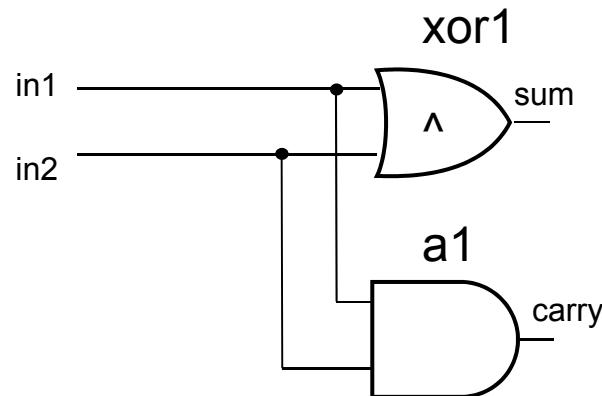
```
entity ALU16 is
port( A, B: in bit_vector (15 downto 0);
      C : out bit_vector (15 downto 0);
      Op: in bit_vector (5 downto 0);
      N, Carry: out bit);
end entity ALU16;
```



```
entity dff is
port( D, Q, Clk, R, S: in bit;
      Q, Qbar : out bit);
end entity dff;
```

# The VHDL architecture

- The VHDL architecture contains the functional description of the model



```
entity half_adder is
port ( in1, in2 : in bit;
       sum, carry :out bit);
end entity half_adder;

architecture behavioral of half_adder is
begin
sum <= (in1 xor in2) after 5 ns;
carry <= (in1 and in2) after 5 ns;
end architecture behavior;
```

You can specify more than one architecture of your entity and easily exchange between them

Signal assignment

Delay

# Predefined enumeration types

- The predefined enumeration types in the package *Standard* include

```
type Boolean is (false, true);

type Bit is ("0", "1");

type Character is (nul, soh, stx, ets, cot,
enq, ack, bel, 0 , 1 , 2 , ...);

Type integer is range -2147483647 to
2147483647;

Type real is range -1.0E308 to 1.0E308;

type Severity_level is (note, warning,
error, failure);

type File_open_kind is (read_mode,
write_mode, append_mode);

type file_open_status is (open_ok,
status_error, name_error, mode_error);
```

# Predefined enumeration types

- (cont)

```
type time is range -2147483647 to 2147483647
    units
        fs;
        ps = 1000fs;
        ns = 1000ps;
        us = 1000ns;
        ms = 1000us;
        sec = 1000ms;
        min = 60sec;
        hr = 60min;
    end units;

subtype delay_length is time range 0 to time high;

subtype natural is integer range 0 to integer high;

type bit_vector is array (natural range <>) of bit;

...
```

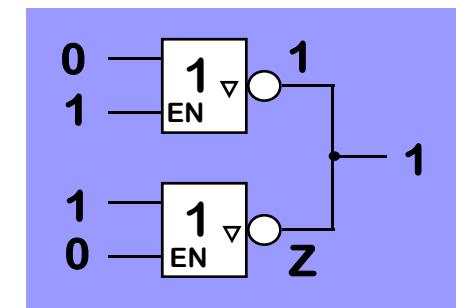
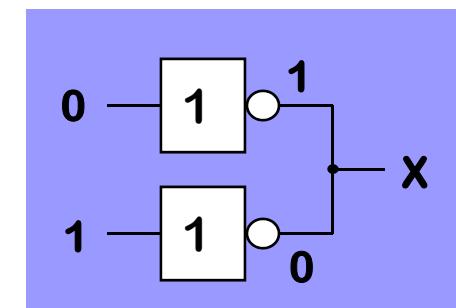
# More predefined packages

- There are plenty of available predefined packages, such as
  - STANDARD (containing elementary data types)
  - TEXTIO (containing useful datatypes and functions/procedures for reading/writing text from/to files)
  - STD\_LOGIC\_TEXTIO (improved version by Synopsys overwriting standard read/write)
  - STD\_LOGIC\_1164 (containing useful multi-value logic types and procedures/functions)
  - NUMERIC\_STD (containing useful information on numerical data types and functions)
  - STD\_LOGIC\_UNSIGNED / STD\_LOGIC\_SIGNED (containing special data types and functions/procedures for signed/unsigned data)
  - STD\_LOGIC\_ARITH
  - STD\_LOGIC\_MISC

# Signal types

- Signals defined as of type „bit“ are simple, but sometimes not sufficient to model real hardware.
- Bit can only be 0 and 1 – what about if you have more than one driver to a net? What about tristate, or unknown?
- VHDL provides two mechanisms for this:
  - There are so-called standard packages which contain a more general signal with more than only 0/1 values (std\_logic, std\_ulogic)
  - There are so-called resolution functions associated with this.

- Logic Systems
  - Signal values representing (logic) level and strength
  - Resolving multiple drivers via so-called resolution functions (e.g. '0' and '1' at the same node result in an 'X')
- 2-valued logic system (e.g. VHDL: Type bit)
  - '0' ("low", e.g.  $V < 2.5 \text{ V}$ ) and '1' ("high",  $V > 2.5 \text{ V}$ )
- 3-valued logic system
  - To describe circuit problems (signal conflicts)
  - '0' ("low"), '1' ("high")
  - 'X' ("unknown", may be '0' or '1')
- 4-valued logic system
  - To describe bus structures
  - '0', '1', 'X' (see above)
  - 'Z' ("high impedance")

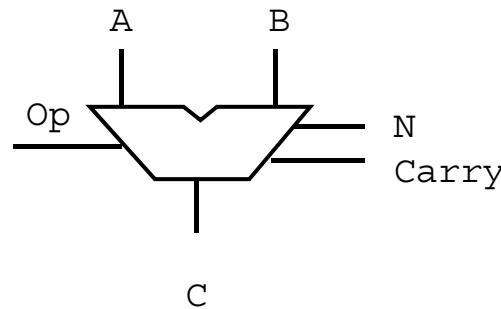


# Logic Systems

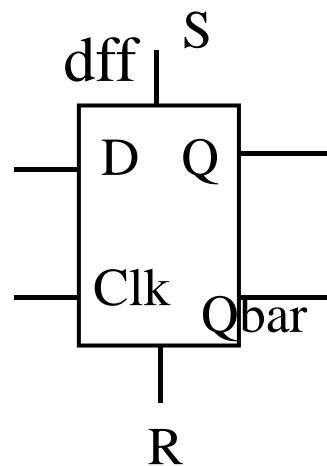
- 9-valued logic systems (VHDL: Type std\_logic\_1164)
  - 'U' ("uninitialized")
  - 'X' ("forcing unknown")
  - '0' ("forcing low"), '1' (forcing high")
  - 'Z' ("high impedance")
  - 'W' ("weak unknown")
  - 'L' ("weak low"), 'H' ("weak high")
  - '-' ("don't care")

```
CONSTANT resolution_table : stdlogic_table := (
-- -----
-- | U   X   0   1   Z   W   L   H   -
-- |
-- -----
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X
( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X', 'X' ), -- | 0
( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X', 'X' ), -- | 1
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X', 'X' ), -- | Z
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X', 'X' ), -- | W
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X', 'X' ), -- | L
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X', 'X' ), -- | H
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | -
);
```

# Examples for VHDL entities (improved)



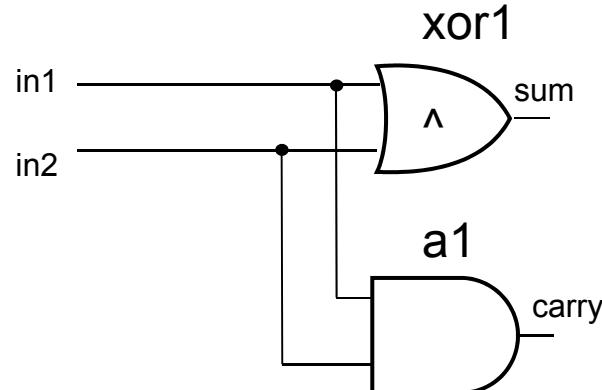
```
entity ALU16 is
port(  A, B: in std_ulogic_vector (15 downto 0);
       C : out std_ulogic_vector (15 downto 0);
       Op: in std_ulogic_vector (5 downto 0);
       N, Carry: out std_logic);
end entity ALU16;
```



```
entity dff is
port(  D, Q, Clk, R, S: in std_ulogic;
       Q, Qbar : out std_ulogic);
end entity dff;
```

# The VHDL architecture

- Before using the IEEE 1164 value system you need to include the library (named IEEE) and use the package definition for this kind of signal type



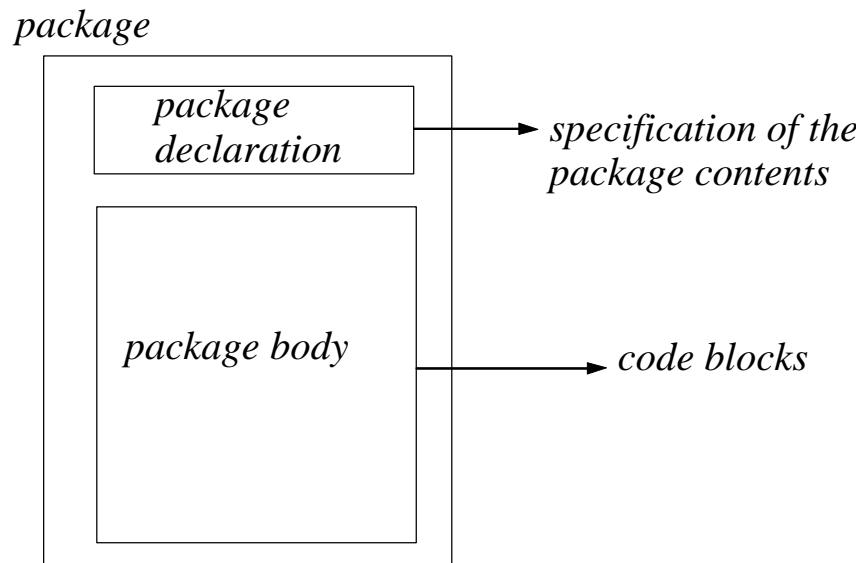
```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port ( in1, in2 : in std_ulogic;
       sum, carry :out std_ulogic);
end entity half_adder;

architecture behavioral of half_adder is
begin
sum <= (in1 xor in2) after 5 ns;
carry <= (in1 and in2) after 5 ns;
end architecture behavioral;
```

# Libraries and Packages

- Libraries are logical units that are mapped to physical directories
- Packages are repositories for type definitions, procedures, and functions
- User defined vs. system packages

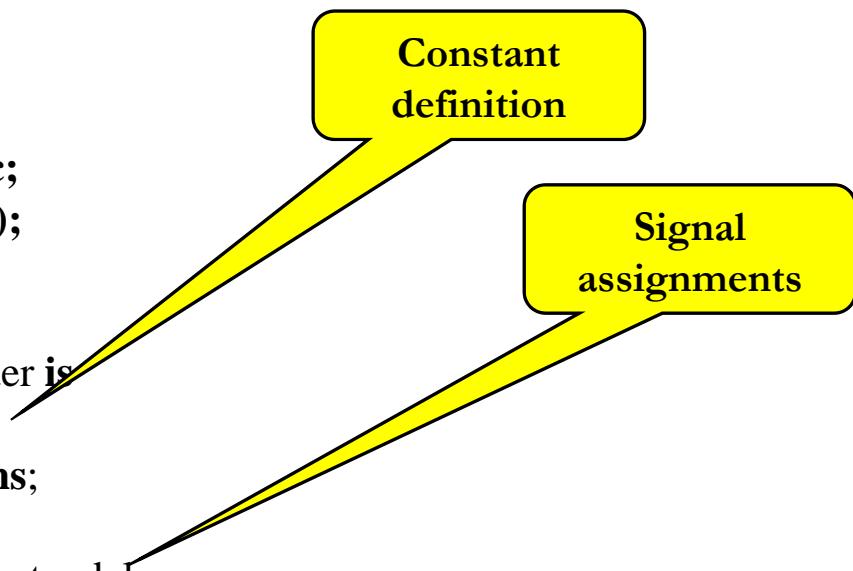


# Assignment examples

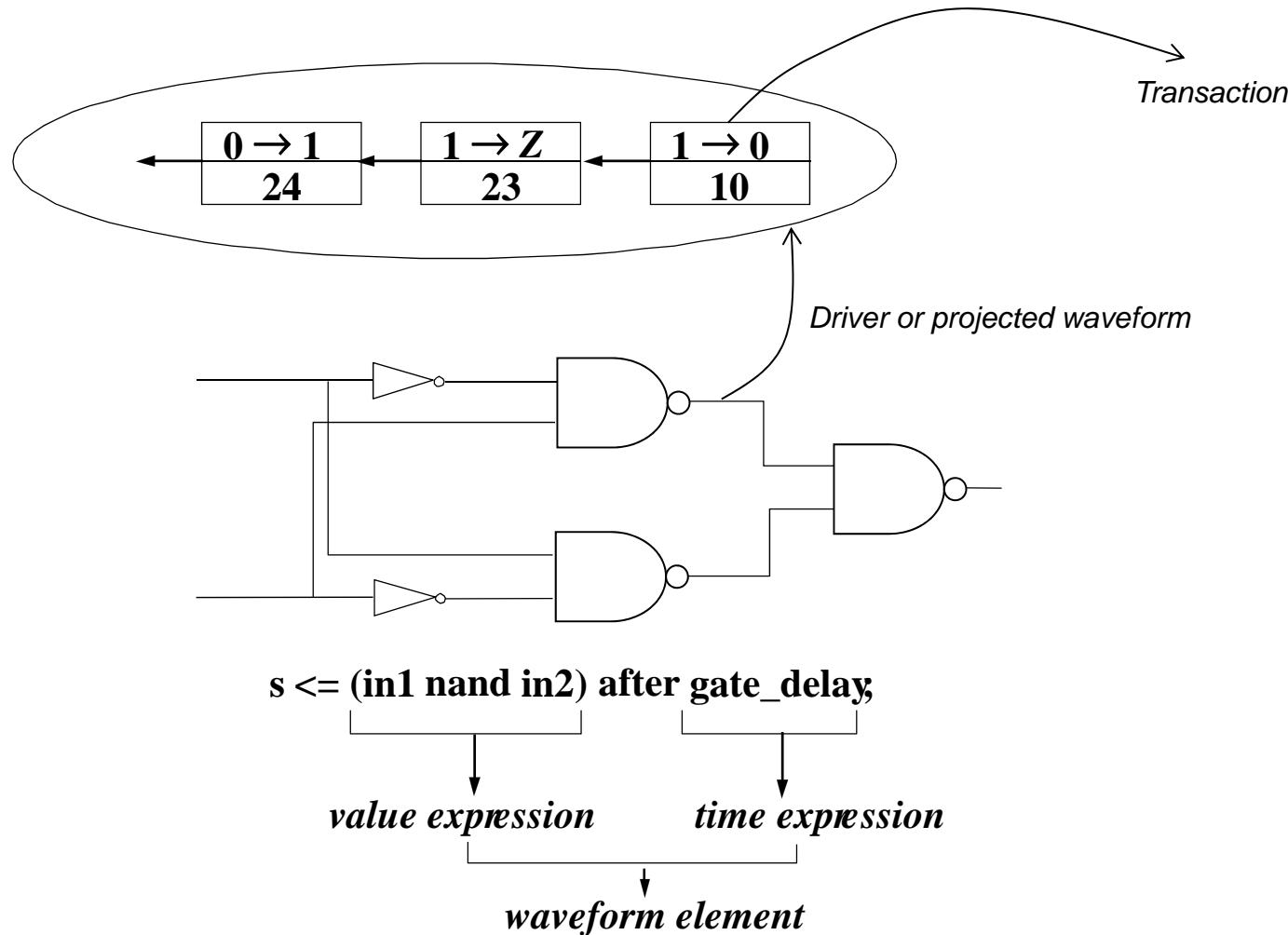
- Below you will find the example of a full adder in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
port (in1, in2, c_in: in std_ulogic;
      sum, c_out: out std_ulogic);
end entity full_adder;

architecture dataflow of full_adder is
signal s1, s2, s3 : std_ulogic;
constant gate_delay: Time:= 5 ns;
begin
  L1: s1 <= (in1 xor in2) after gate_delay;
  L2: s2 <= (c_in and s1) after gate_delay;
  L3: s3 <= (in1 and in2) after gate_delay;
  L4: sum <= (s1 xor c_in) after gate_delay;
  L5: c_out <= (s2 or s3) after gate_delay;
end architecture dataflow;
```



# Signal implementation

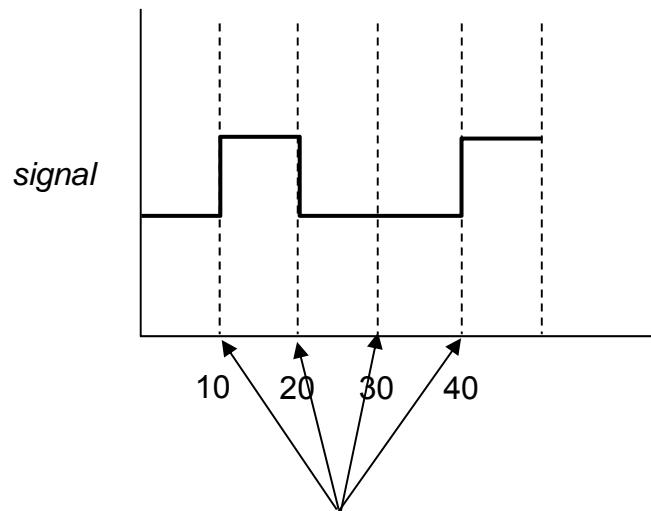


# More on signal assignments

- In the absence of initialization, default values are determined by signal type
- Waveform elements describe time-value pairs
- *Transactions* are internal representations of signal value assignments
  - Events correspond to new signal values
  - A transaction may lead to the same signal value

# More on signal assignments

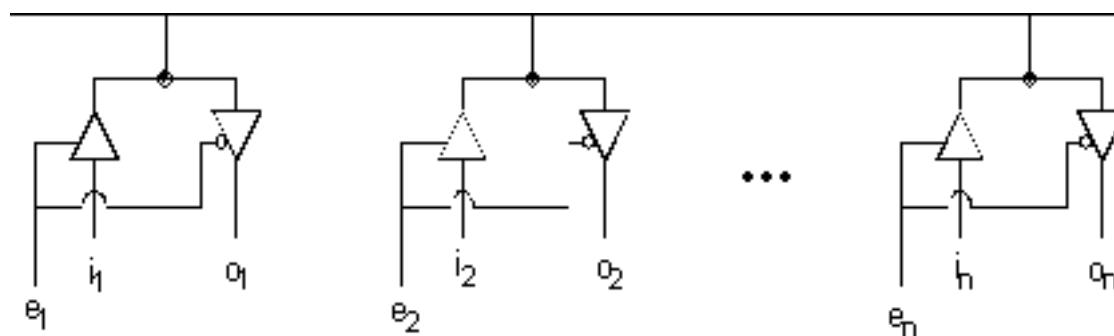
- Multiple waveform elements can be specified in a single signal assignment statement
- Describe the signal transitions at future point in time
  - Each transition is specified as a waveform element



**signal <= '0', '1' after 10 ns, '0' after 20 ns, '1' after 40 ns;**

# Resolution function

- At any point in time what is the value of the bus signal?
- We need to “resolve” the value
  - Take the value at the head of all drivers
  - Select one of the values according to a resolution function
- Predefined IEEE 1164 resolved types are std\_logic and std\_logic\_vector



# Conditional signal assignments

- The first true evaluated branch determines the signal assignment

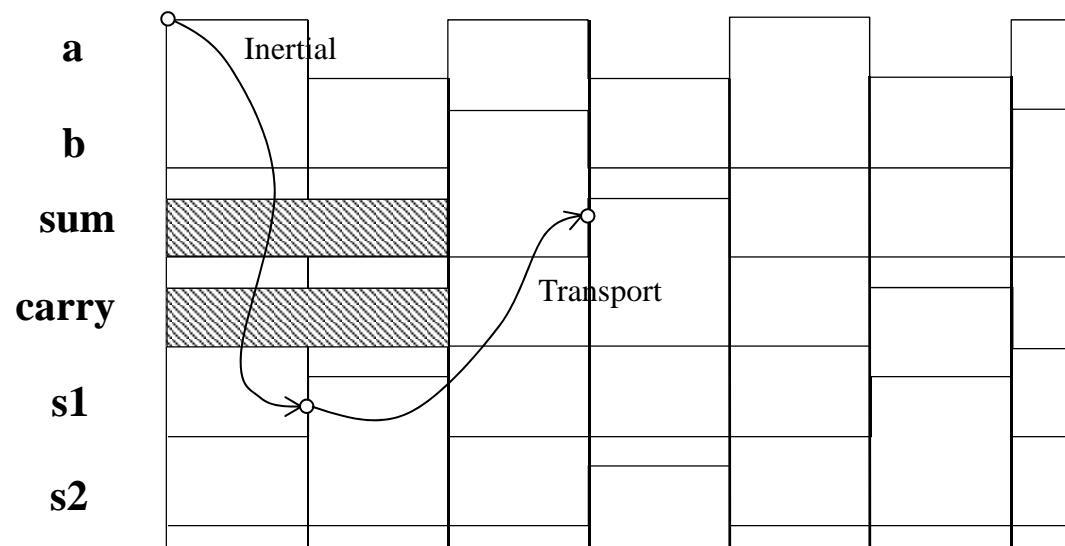
```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity mux4 is  
    port ( In0, In1, In2, In3 : in std_logic_vector (7 downto 0);  
           Sel: in std_logic_vector(1 downto 0);  
           Z : out std_logic_vector (7 downto 0));  
end entity mux4;  
  
architecture behavioral of mux4 is  
begin  
    Z <= In0 after 5 ns when Sel = "00" else  
          In1 after 5 ns when Sel = "01" else  
          In2 after 5 ns when Sel = "10" else  
          In3 after 5 ns when Sel = "11" else  
          "00000000" after 5 ns;  
end architecture behavioral;
```

# Delay models in VHDL

- Inertial delay
  - Default delay model
  - Suitable for modeling delays through devices such as gates (similar to # in Verilog)
- Transport Delay
  - Model delays through devices with very small inertia, e.g., wires
  - All input events are propagated to output signals (also somehow similar to modeling wire delays in Verilog)
- Delta delay
  - What about models where no propagation delays are specified?
  - Infinitesimally small delay is automatically inserted by the simulator to preserve correct ordering of events (similar to the  $a = \#0 b$ ; assignment in Verilog)

# Delay models in VHDL

```
architecture transport_delay of half_adder is
    signal s1, s2: std_logic:= '0';
begin
    s1 <= (a xor b) after 2 ns;
    s2 <= (a and b) after 2 ns;
    sum <= transport s1 after 4 ns;
    carry <= transport s2 after 4 ns;
end architecture transport_delay;
```

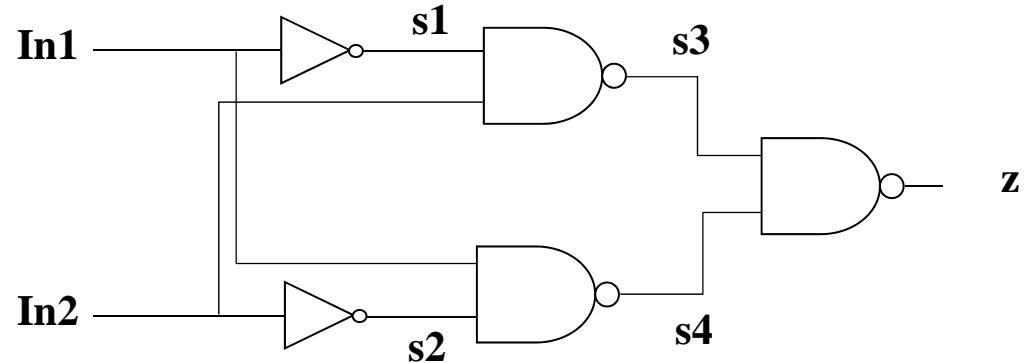


# Delta Delay

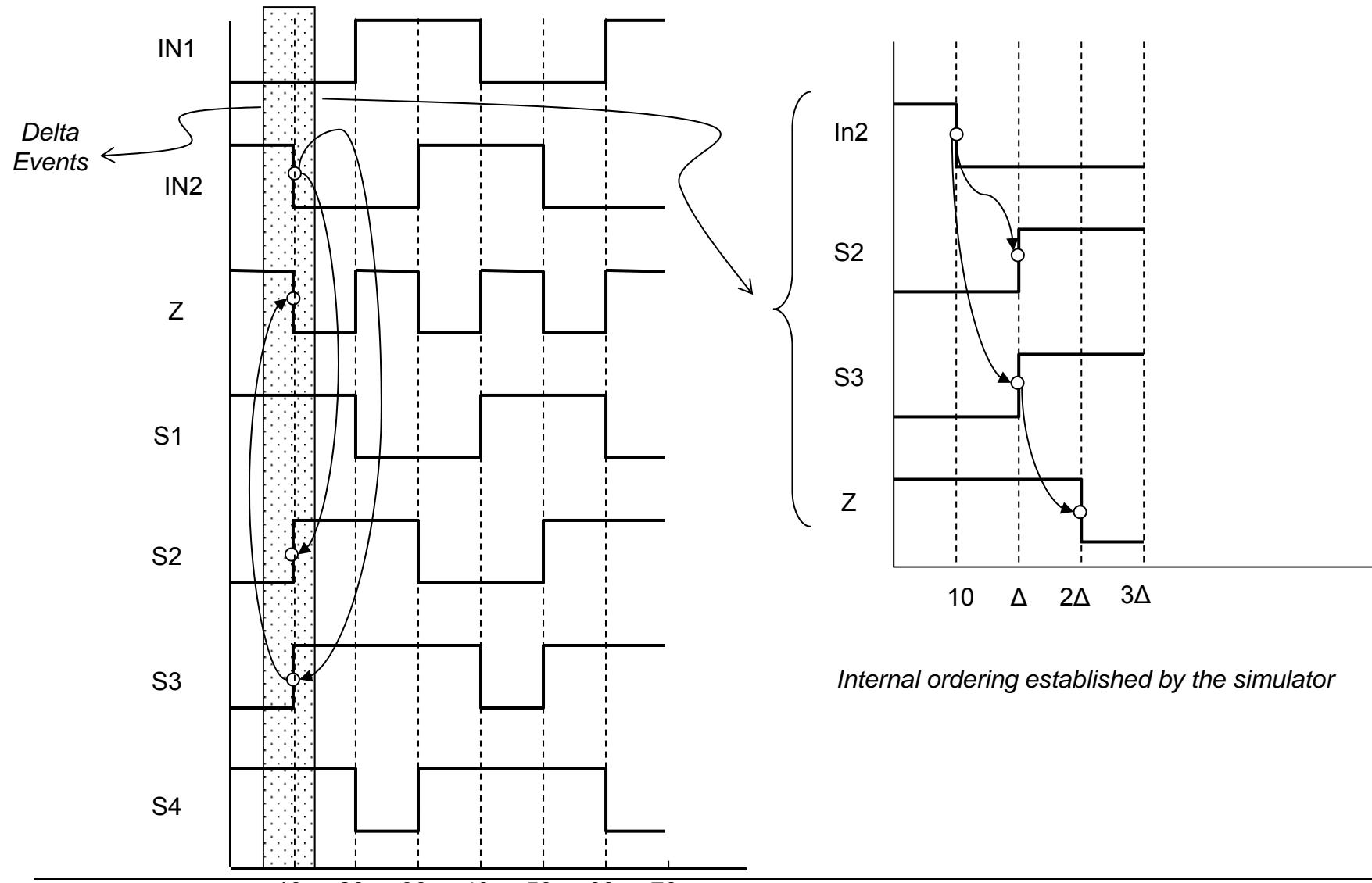
```
library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity combinational is  
port (In1, In2: in std_logic;  
z : out std_logic);  
end entity combinational;
```

```
architecture behavior of combinational  
signal s1, s2, s3, s4: std_logic:= '0';  
begin  
s1 <= not In1;  
s2 <= not In2;  
s3 <= not (s1 and In2);  
s4 <= not (s2 and In1);  
z <= not (s3 and s4);  
end architecture behavior;
```



# Delta Delay



# Delay models in VHDL

- Delay models
  - Inertial
    - For devices with inertia such as gates
    - VHDL 1993 supports pulse rejection widths
  - Transport
    - Ensures propagation of all events
    - Typically used to model elements such as wires
  - Delta
    - Automatically inserted to ensure functional correctness of code blocks that do not specify timing
    - Enforces the data dependencies specified in the code

# Process definition

- A process statement defines an independent sequential process representing the behavior of some portion of the design
- A process allows for synchronization with other concurrent statements through signals

# Process model in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4 is
port (In0, In1, In2, In3: in std_logic_vector (7 downto
0);
      Sel: in std_logic_vector(1 downto 0);
      Z : out std_logic_vector (7 downto 0));
end entity mux4;

architecture behavioral-3 of mux4 is

process (Sel, In0, In1, In2, In3) is
variable Zout: std_logic;
begin
    if (Sel = "00") then Zout := In0;
    elsif (Sel = "01") then Zout := In1;
    elsif (Sel = "10") then Zout := In2;
    else Zout:= In3;
    end if;
    Z <= Zout;
end process;
```

# Process model in VHDL

- Statements in a process are executed sequentially
- A process body is structured much like conventional C function
  - Declaration and use of variables
  - *if-then*, *if-then-else*, *case*, *for* and *while* constructs
  - A process can contain signal assignment statements
- A process executes concurrently with other concurrent signal assignment statements
- A process takes 0 seconds of simulated time to execute and may schedule events in the future
- We can think of a process as a complex signal assignment statement!

# Process model in VHDL (cont)

- The process sensitivity list must only contain elements from which reading is allowed, and which might affect the output
- You may either put the signal names from which reading is allowed in the sensitivity list, or in a wait on sig\_name1, sig\_name2; statement.

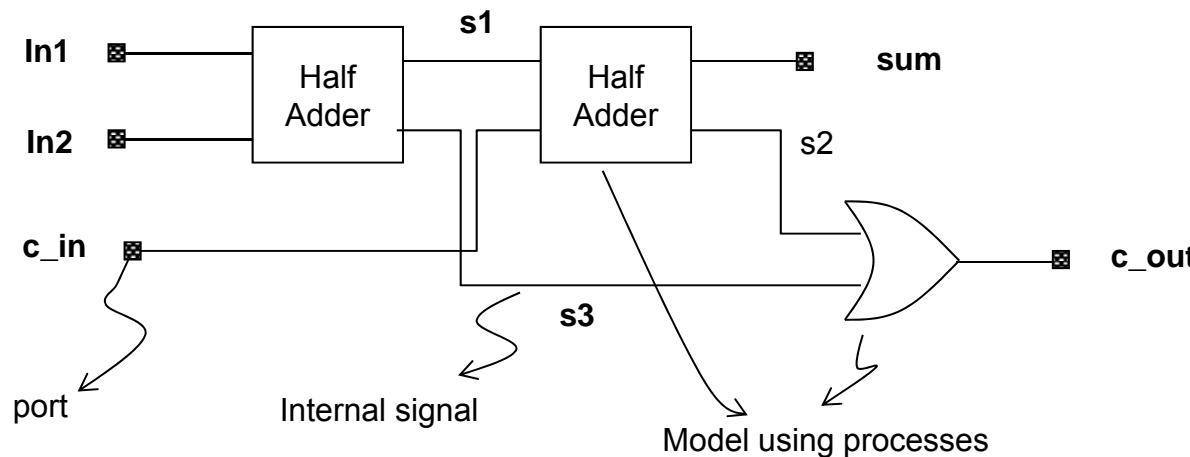
```
architecture RTL of and2 is
begin-- all Read Signals in Sensitivity List
process (a, b)
begin
  C <= a AND b;
end process;
end RTL;
```

```
architecture RTL2 of and2 is
begin -- implicit wait on all signals on
-- right hand side of the concurrent
-- signal assignment
  C <= a AND b;
end RTL2;
```

```
architecture RTL1 of and2 is
begin
  process
    begin
      if A = '1' then
        C <= B;
      else
        C <= '0';
      end if;
      wait on A, B;
    end process;
  end RTL1;
```

# Process model in VHDL

- Have a look at our fulladder below:
- Each of the components of the full adder can be modeled using a process
- Processes execute concurrently
  - In this sense they behave exactly like concurrent signal assignment statements
- Processes communicate via signals



# VHDL model of fulladder

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

entity full_adder is
port (In1, c_in, In2: in std_logic;
      sum, c_out: out std_logic);
end entity full_adder;

architecture behavioral of full_adder is
signal s1, s2, s3: std_logic;
constant delay:Time:= 5 ns;
begin

HA1: process (In1, In2) is
begin
s1 <= (In1 xor In2) after delay;
s3 <= (In1 and In2) after delay;
end process HA1;

HA2: process(s1,c_in) is
begin
sum <= (s1 xor c_in) after delay;
s2 <= (s1 and c_in) after delay;
end process HA2;

OR1: process (s2, s3) -- process
describing the two-input OR gate
begin
c_out <= (s2 or s3) after delay;
end process OR1;

end architecture behavioral;
```

# VHDL model of halfadder

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

entity half_adder is
port (a, b : in std_logic;
      sum, carry : out std_logic);
end entity half_adder;

architecture behavior of half_adder is
begin

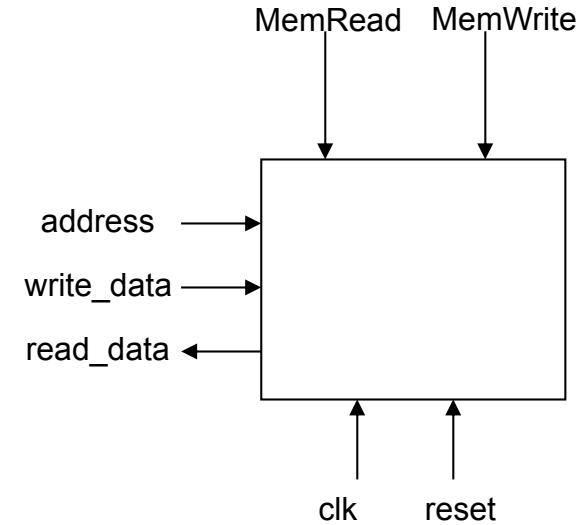
sum_proc: process(a,b) is
begin
  if (a = b) then
    sum <= '0' after 5 ns;
  else
    sum <= (a or b) after 5 ns;
  end if;
end process;

carry_proc: process (a,b) is
begin
  case a is
    when '0' =>
      carry <= a after 5 ns;
    when '1' =>
      carry <= b after 5 ns;
    when others =>
      carry <= 'X' after 5 ns;
  end case;
end process carry_proc;

end architecture behavior;
```

# Processes and concurrent signal assignments (CSA)

```
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_arith.all;  
  
entity memory is  
port (address, write_data: in std_logic_vector (7 downto 0);  
      MemWrite, MemRead, clk, reset: in std_logic;  
      read_data: out std_logic_vector (7 downto 0));  
end entity memory;  
  
architecture behavioral of memory is  
signal dmem0,dmem1,dmem2,dmem3: std_logic_vector (7 downto 0);  
begin  
mem_proc: process (clk) is  
  -- process body  
end process mem_proc;  
  -- read operation CSA  
end architecture behavioral;
```



# Processes and concurrent signal assignments (CSA)

```
mem_proc: process (clk) is
begin
  if (rising_edge(clk)) then -- wait until next clock edge
    if reset = '1' then -- initialize values on reset
      dmem0 <= x"00"; -- memory locations are initialized to
      dmem1 <= x"11";-- some random values
      dmem2 <= x"22";
      dmem3 <= x"33";
    elsif MemWrite = '1' then -- if not reset then check for memory write
      case address (1 downto 0) is
        when "00" => dmem0 <= write_data;
        when "01" => dmem1 <= write_data;
        when "10" => dmem2 <= write_data;
        when "11" => dmem3 <= write_data;
        when others => dmem0 <= x"ff";
      end case;
    end if;
  end if;
end process mem_proc;
```

# Processes and concurrent signal assignments (CSA)

- *memory read is implemented with a conditional signal assignment*

```
read_data <= dmem0 when address (1 downto 0) = "00" and MemRead = '1' else
    dmem1 when address (1 downto 0) = "01" and MemRead = '1' else
    dmem2 when address (1 downto 0) = "10" and MemRead = '1' else
    dmem3 when address (1 downto 0) = "11" and MemRead = '1' else
        x"00";
```

# Processes and concurrent signal assignments (CSA)

- A process can be viewed as single concurrent signal assignment statement
- The external behavior is the same as a CSA
- Processes describe more complex event generation behavior
- Processes execute concurrently in simulated time with other CSAs

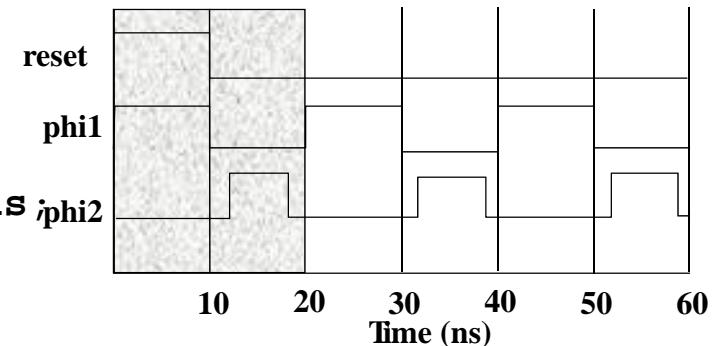
# VHDL multiplier

```
architecture behavioral of mult32 is
  constant module_delay: Time:= 10 ns;
  begin
    mult_process: process(multiplicand,multiplier) is
      variable product_register : std_logic_vector (63 downto 0) := X"0000000000000000";
      variable multiplicand_register : std_logic_vector (31 downto 0) := X"00000000";
    begin
      multiplicand_register := multiplicand;
      product_register(63 downto 0) := X"00000000" & multiplier;
      for index in 1 to 32 loop
        if product_register(0) = '1' then
          product_register(63 downto 32) := product_register (63 downto 32) +
            multiplicand_register(31 downto 0);
        end if;
        -- perform a right shift with zero fill
        product_register (63 downto 0) := '0' & product_register (63 downto 1);
      end loop;
      -- write result to output port
      product <= product_register after module_delay;
    end process mult_process;
```

# The VHDL wait statement

```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out std_logic);
end entity two_phase;

architecture behavioral of two_phase is
begin
rproc: reset <= '1', '0' after 10 ns;
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns;
after 18 ns;
    wait for 20 ns;
end process clock_process;
end architecture behavioral;
```



Wait for time suspends a process until the elapsed time in the time expression – similar to # inside a Verilog process

# Asynchronous DFF in VHDL

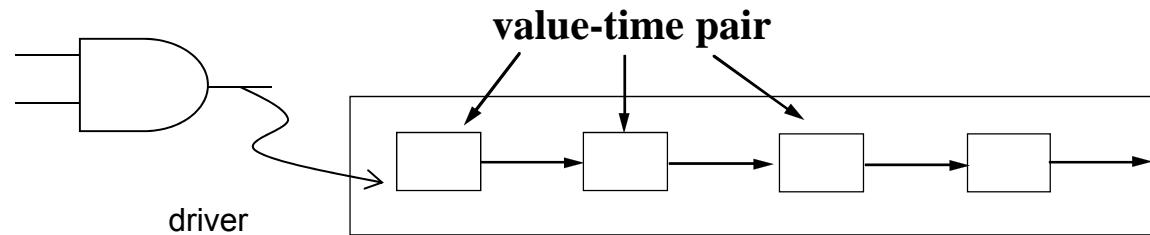
```
library IEEE;                               architecture behavioral of asynch_dff is
use IEEE.std_logic_1164.all;    begin
   output: process (R, S, Clk) is
entity asynch_dff is
  port (R, S, D, Clk: in
        std_logic;
        Q, Qbar: out std_logic);
end entity asynch_dff;
   begin
   if (R = '0') then
   Q <= '0' after 5 ns;
   Qbar <= '1' after 5 ns;
   elsif S = '0' then
   Q <= '1' after 5 ns;
   Qbar <= '0' after 5 ns;
   elsif (rising_edge(Clk)) then
   Q<= D after 5 ns;
   Qbar <= (not D) after 5 ns;
   end if;
   end process output;
end architecture behavioral;
```

# Some thoughts on wait and PSL

- A process can have multiple wait statements
- A process cannot have both a wait statement and a sensitivity list (it should have one or the other, not both)
- wait statements provide explicit control over suspension and restart of processes
  - Representation of both synchronous and asynchronous events in a digital systems

# Attributes in VHDL

- Data can be obtained about VHDL objects such as types, arrays and signals.  
**object' attribute**
- Example: consider the implementation of a signal



- What types of information about this signal are useful?
  - Occurrence of an event
  - Elapsed time since last event
  - Previous value, i.e., prior to the last event

# Value Attributes in VHDL

- Return a constant value
  - **type statetype is** (state0, state1, state2 state3);
    - **state\_type'left** = state0
    - **state\_type'right** = state3

| Value attribute          | Value                                                          |
|--------------------------|----------------------------------------------------------------|
| <b>type_name'left</b>    | returns the left most value of type_name in its defined range  |
| <b>type_name'right</b>   | returns the right most value of type_name in its defined range |
| <b>type_name'high</b>    | returns the highest value of type_name in its range            |
| <b>type_name'low</b>     | returns the lowest value of type_name in its range             |
| <b>array_name'length</b> | returns the number of elements in the array array_name         |

# Example

```
clk_process: process
begin
  wait until (clk'event and clk = '1');
  if reset = '1' then
    state <= statetype'left;
  else state <= next_state;
  end if;
end process clk_process;
```

- The signal state is an enumerated type
  - **type** statetype **is** (state0, state1, state3, state4);
  - **signal** state:statetype:= statetype'left;

# Function attributes

- Use of attributes invokes a function call which returns a value
  - if (Clk'**event** and Clk = '1')
- *function call*
- Examples: function signal attributes

| Function attribute              | Function                                                                                                     |
|---------------------------------|--------------------------------------------------------------------------------------------------------------|
| signal_name' <b>event</b>       | Return a Boolean value signifying a change in value on this signal                                           |
| signal_name' <b>active</b>      | Return a Boolean value signifying an assignment made to this signal. This assignment may not be a new value. |
| signal_name' <b>last_event</b>  | Return the time since the last event on this signal                                                          |
| signal_name' <b>last_active</b> | Return the time since the signal was last active                                                             |
| signal_name' <b>last_value</b>  | Return the previous value of this signal                                                                     |

# Function attributes

| Function attribute       | Function                                   |
|--------------------------|--------------------------------------------|
| array_name' <b>left</b>  | returns the left bound of the index range  |
| array_name' <b>right</b> | returns the right bound of the index range |
| array_name' <b>high</b>  | returns the upper bound of the index range |
| array_name' <b>low</b>   | returns the lower bound of the index range |

```
type mem_array is array(0 to 7) of bit_vector(31 downto 0)
    mem_array'left = 0
    mem_array'right = 7
    mem_array'length = 8 (value kind attribute)
```

# Variable attributes

- Convenient use of variable attributes in a loop:

```
for i in value_array`range loop
...
...
my_var := value_array(i);
...
end loop
```

# Signal attributes

- Creates a new “implicit” signal

| Signal attribute                     | Implicit Signal                                                                  |
|--------------------------------------|----------------------------------------------------------------------------------|
| <code>signal_name'delayed(T)</code>  | Signal delayed by T units of time                                                |
| <code>signal_name'transaction</code> | Signal whose value toggles when <code>signal_name</code> is active               |
| <code>signal_name'quiet(T)</code>    | True when <code>signal_name</code> has been quiet for T units of time            |
| <code>signal_name'stable(T)</code>   | True when event has not occurred on <code>signal_name</code> for T units of time |

# State machines in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
entity state_machine is
    port(reset, clk, x : in std_logic;
          z : out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
type statetype is (state0, state1);
signal state, next_state : statetype := state0;
begin
    comb_process: process (state, x) is
        begin
            --- process description here
    end process comb_process;
    clk_process: process is
        begin
            -- process description here
    end process clk_process;
end architectural behavioral;
```

# State machines in VHDL

```
comb_process: process (state, x) is
begin
  case state is -- depending upon the current state
    when state0 => -- set output signals and next state
      if x = '0' then
        next_state <= state1;
        z <= '1';
      else next_state <= state0;
        z <= '0';
      end if;
    when state1 =>
      if x = '1' then
        next_state <= state0;
        z <= '0';
      else next_state <= state1;
        z <= '1';
      end if;
    end case;
  end process comb_process;
```

# State machines in VHDL

```
clk_process: process is
begin
  wait until (clk'event and clk = '1'); -- wait until the rising edge
  if reset = '1' then -- check for reset and initialize state
    state <= statetype'left;
  else state <= next_state;
  end if;
end process clk_process;
end behavioral;
```

# Function calls in VHDL

```
architecture behavioral of dff is
    function rising_edge (signal clock : std_logic) return boolean is
        variable edge : boolean:= FALSE;
        begin
            edge := (clock = '1' and clock'event);
            return (edge);
        end rising_edge;

        begin
        output: process
        begin
            wait until (rising_edge(Clk));
            Q <= D after 5 ns;
            Qbar <= not D after 5 ns;
        end process output;
    end architecture behavioral;
```

# Another example

```
function to_bitvector (svalue : std_logic_vector) return bit_vector is
    variable outvalue : bit_vector (svalue'length-1 downto 0);
begin
    for i in svalue'range loop -- scan all elements of the array
        case svalue (i) is
            when '0' => outvalue (i) := '0';
            when '1' => outvalue (i) := '1';
            when others => outvalue (i) := '0';
        end case;
    end loop;
    R eturn outvalue;
end to_bitvector
```

- A common use for function is type conversion, as shown above

# Procedures

- A procedure is the second type of subprogram allowed in VHDL
- It performs operations on all visible parameters and objects
- It may include formal parameters that define the objects used in the execution of a procedure
- A procedure may contain wait statements
  
- Two type of procedures:
  - Sequential procedures are called from within processes
  - Concurrent procedures instantiated as concurrent statements

# Procedures

```
procedure read_v1d (variable f: in text; v :out std_logic_vector)
--declarative region: declare variables local to the procedure
--
begin
-- body
--
end read_v1d;
```

- Concurrent and sequential procedure calls
- Parameters may be of mode **in** (read only) and **out** (write only)
- Default class of input parameters is constant
- Default class of output parameters is variable
- Variables declared within procedure are initialized on each call

# Visibility of Procedures

**architecture behavioral** of cpu is

--

-- *declarative region*

-- *procedures can be placed in their entirety here*

--

**begin**

process\_a: **process**

-- *declarative region of a process*

-- *procedures can be placed here*

**begin**

--

-- *process body*

--

**end** process\_a;

process\_b: **process**

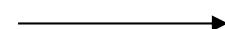
--*declarative regions*

**begin**

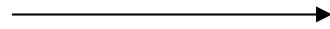
-- *process body*

**end** process\_b;

**end architecture behavioral;**



*visible to all  
processes*

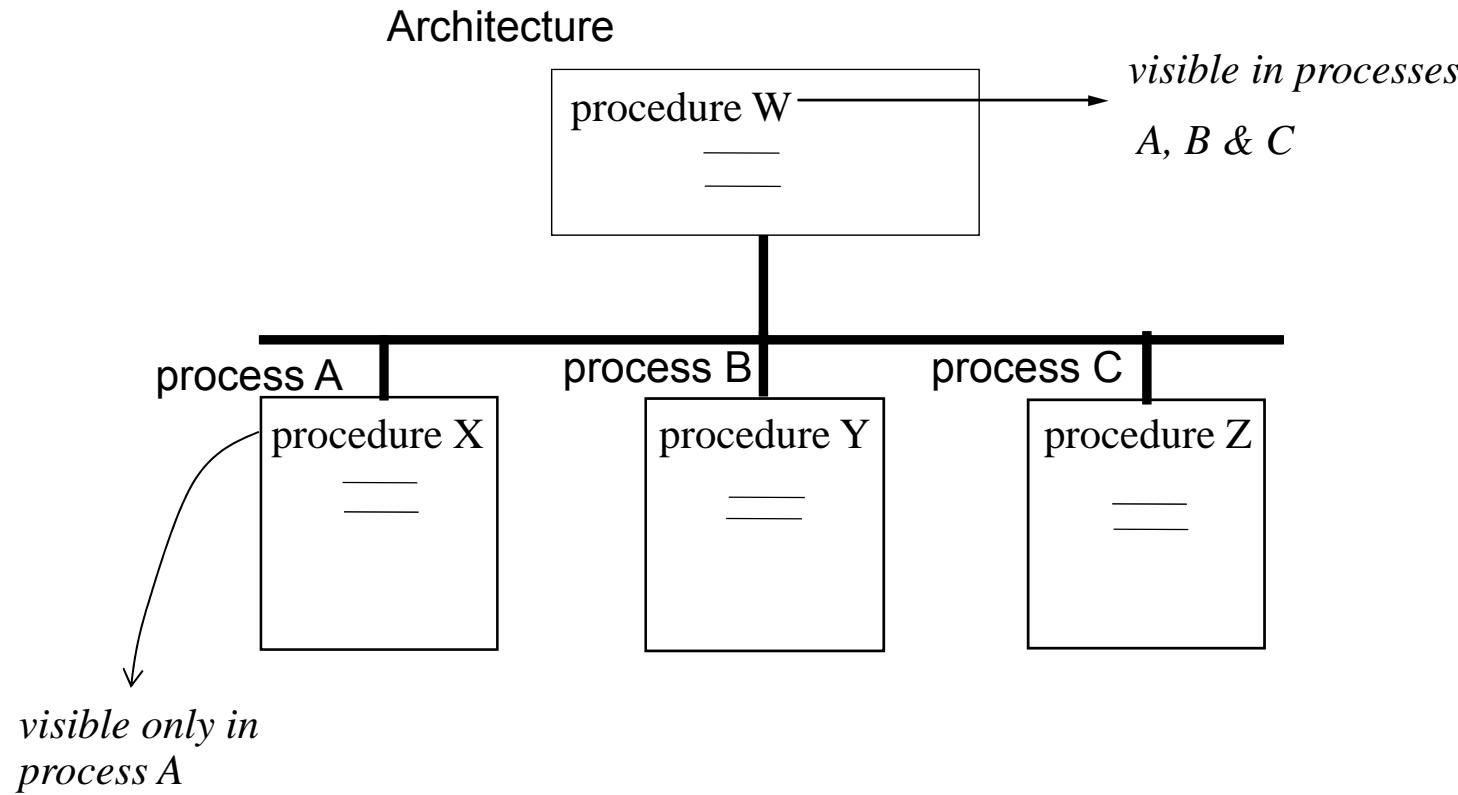


*visible only within  
process\_a*



*visible only within  
process\_b*

# Visibility of Procedures



# Packages

- Package Declaration
  - Declaration of the functions, procedures, and types that are available in the package
  - Serves as a package interface
  - Only declared contents are visible for external use
- You must specify what you want to use from a package using the **use** clause
- Package body
  - Implementation of the functions and procedures declared in the package header
  - Instantiation of constants provided in the package header

# Packages

```
package std_logic_1164 is
type std_ulogic is ('U', --Uninitialized
'X', -- Forcing Unknown
'0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance
'W', -- Weak Unknown
'L', -- Weak 0
'H', -- Weak 1
'-' -- Don't care
);
type std_ulogic_vector is array (natural range <>) of std_ulogic;
function resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
function "and" (l, r : std_logic_vector) return std_logic_vector;
--..<rest of the package definition>
end package std_logic_1164;
```

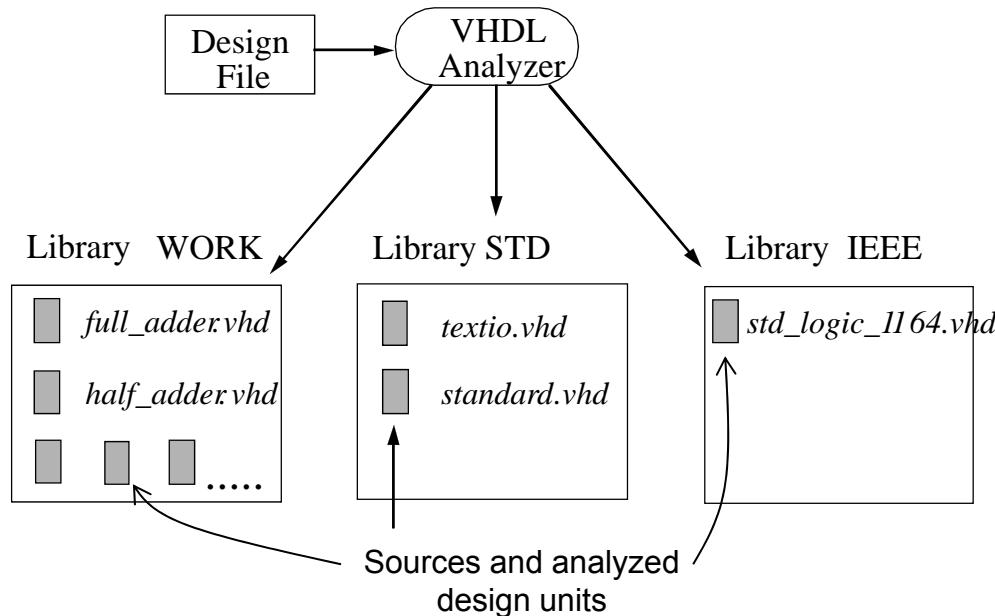
# Package body

```
package body my_package is
-- 
-- type definitions, functions, and procedures
--
end my_package;
```

- Packages are typically compiled into libraries
- New types must have associated definitions for operations such as logical operations (e.g., and, or) and arithmetic operations (e.g., +, \*)

# Libraries

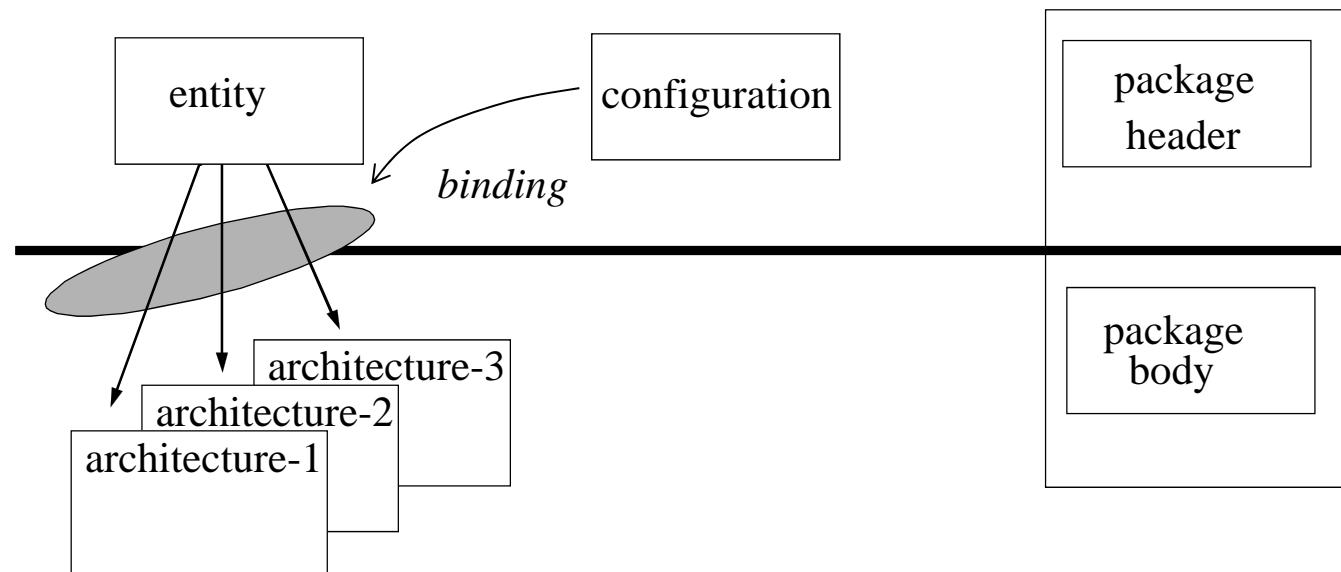
- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared



# Design Units

- Distinguish the primary and secondary design units
- Compilation order

*Primary Design Units*



*Secondary Design Units*

# Final words on VHDL vs Verilog

- The library & package concept make VHDL more suitable for complex system modeling – you can define data types on your own, and put them together with functions in a package
- VHDL is much richer in built-in functions than Verilog
- VHDL is missing completely the possibility to model switch-level hardware (e.g. on transistor level)
- Gatelevel primitives are very limited in VHDL, but you could easily build your own gatelevel set
- In practice the most annoying behavior of VHDL is that you must write your own conversion function if you want to assign data to another data\_type

# Feedback: yes, please ...

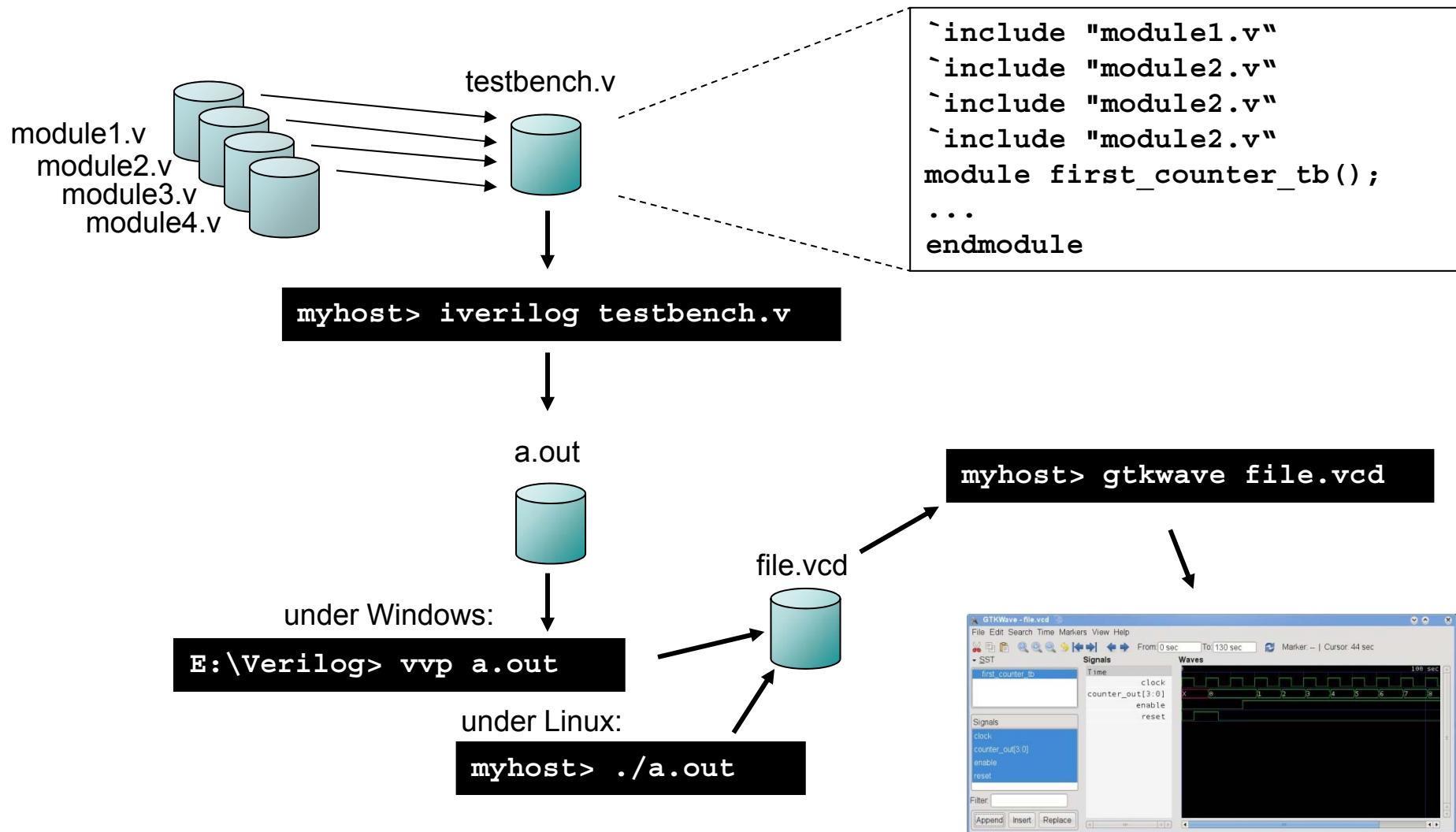


# Appendix 1: Verilog Simulation Tools (Icarus)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Icarus Verilog Simulation Flow



# Icarus Verilog Simulator Download

Main Web Site:

- <http://iverilog.icarus.com/>

Icarus Packages:

- Microsoft Windows:

<http://bleyer.org/icarus/>

- 1) `iverilog-0.9.6_setup.exe` [5.33MB]
- 2) GTKWave for Windows
- 3) optional: IVI, a graphical frontend for Icarus  
(based on Eclipse)
- 4) optional: Verilog syntax highlighting for UltraEdit

# Icarus Verilog Simulator Download

## Icarus Packages:

- Linux (SuSE): Installation Option 1

[http://www.geda.seul.org/wiki/geda:suse\\_rpm\\_installation](http://www.geda.seul.org/wiki/geda:suse_rpm_installation)

(read this page)

- Start Yast and go to "Change Installation Source"
- Add a **HTTP** source having the following properties:

Servername: `software.opensuse.org`

Directory:    `/download/repositories/science/openSUSE_11.1/`

(Choose the directory for your distribution)

- Start "Install Software" in Yast, goto "Search" and look for gEDA. All packages appear.
- Install the packages „verilog“ and „gtkwave“ as usual.

# Icarus Verilog Simulator Download

Icarus Packages:

- Linux (SuSE): Installation Option 2

[http://iverilog.wikia.com/wiki/Installation\\_Guide](http://iverilog.wikia.com/wiki/Installation_Guide)

SuSE: <http://software.opensuse.org/search>

- 1) enter `verilog` and your SuSE Version -> download .rpm-File
- 2) enter `gtkwave` and your SuSE Version -> download .rpm-File
- 3) Start YAST software installation tool and install gtk 2.0
- 4) For both downloaded rpm files do: `rpm -i <name>.rpm`

# Icarus Verilog Simulator Download

## Icarus Packages:

- Linux (Ubuntu): Installation (z.B. Ubuntu 8.04 „hardy”)  
<http://ns2.canonical.com/de/hardy/electronics/>  
[http://iverilog.wikia.com/wiki/Installation\\_Guide](http://iverilog.wikia.com/wiki/Installation_Guide)

- Add the Universe repository in `/etc/apt/sources.list` using your favourite text editor (It would already be there but would have been commented). In Hardy (8.04), the line would be something like

```
deb http://archive.ubuntu.com/ubuntu/ hardy universe
deb-src http://archive.ubuntu.com/ubuntu/ hardy universe
```

- Run `sudo apt-get update` from terminal.
- Run `sudo apt-get install verilog`

# Running your first Example

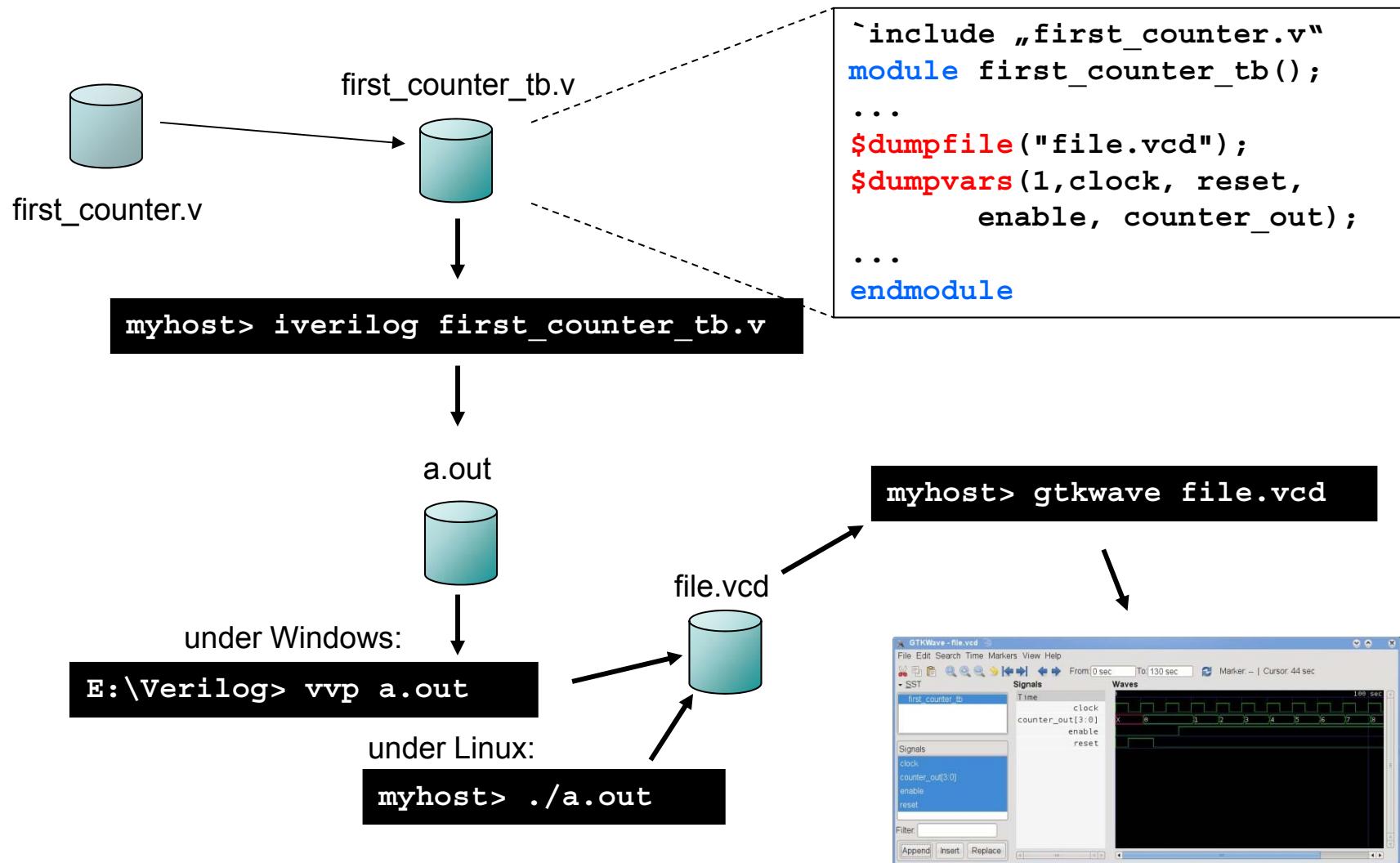
Example from: <http://www.asic-world.com/verilog/first1.html>

- 4-bit synchronous up counter
- synchronous reset (active high)
- enable (active high).

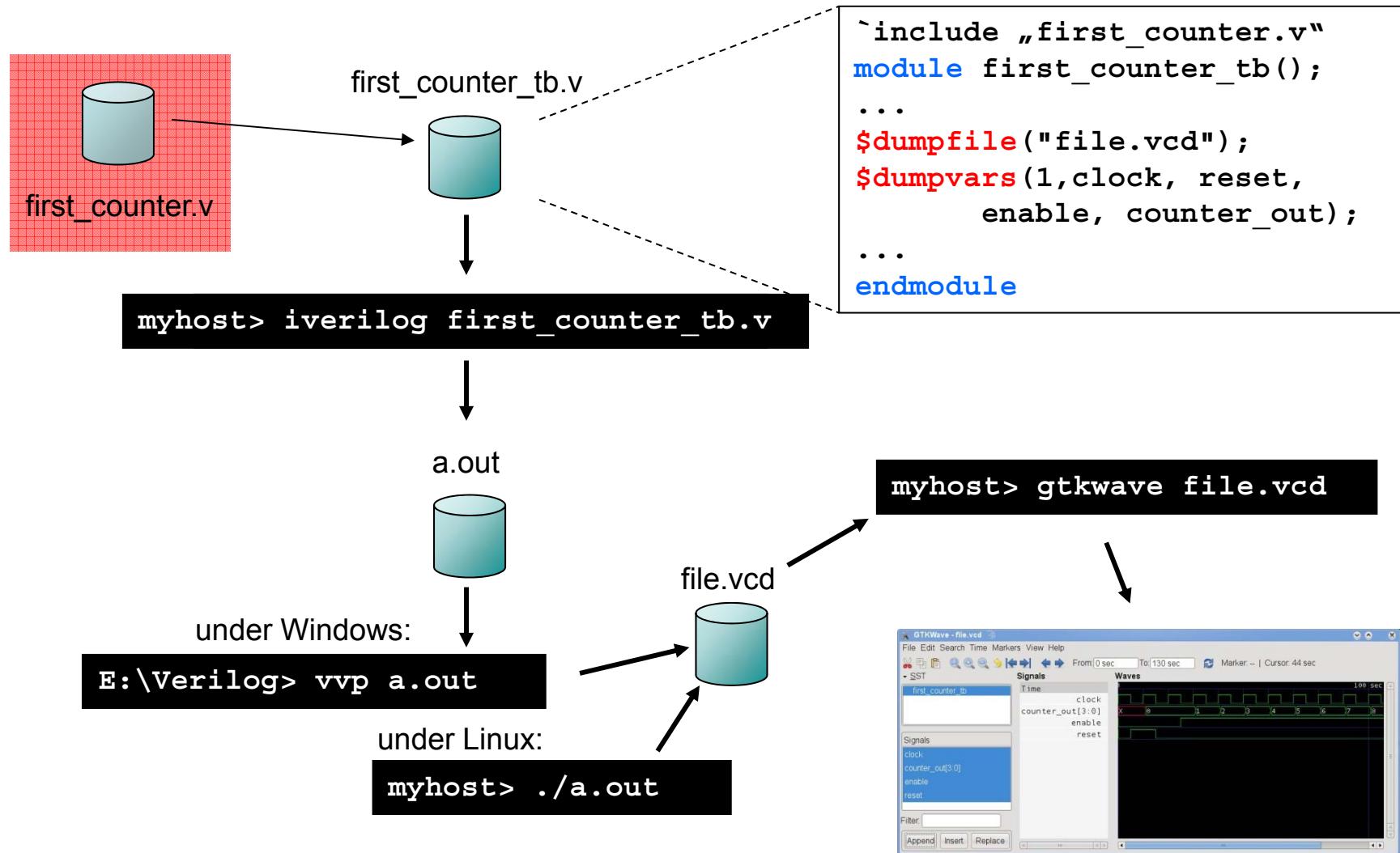
Design File: first\_counter.v

Testbench : first\_counter\_tb.v

# Icarus Verilog Simulation Flow



# Icarus Verilog Simulation Flow



# Running your first Example

```
//-----
// Design Name : first_counter
// File Name : first_counter.v
// Function : This is a 4 bit up-counter with
// Synchronous active high reset and
// with active high enable signal
//-----
module first_counter (
    clock , // Clock input of the design
    reset , // active high, synchronous Reset input
    enable , // Active high enable signal for counter
    counter_out // 4 bit vector output of the counter
); // End of port list

//### (1) Port Descriptions ###

//### (2) Code ###

endmodule // End of Module counter
```

first\_counter.v

# Running your first Example

first\_counter.v

```
//### (1) Port Descriptions ###

//-----Input Ports-----

```

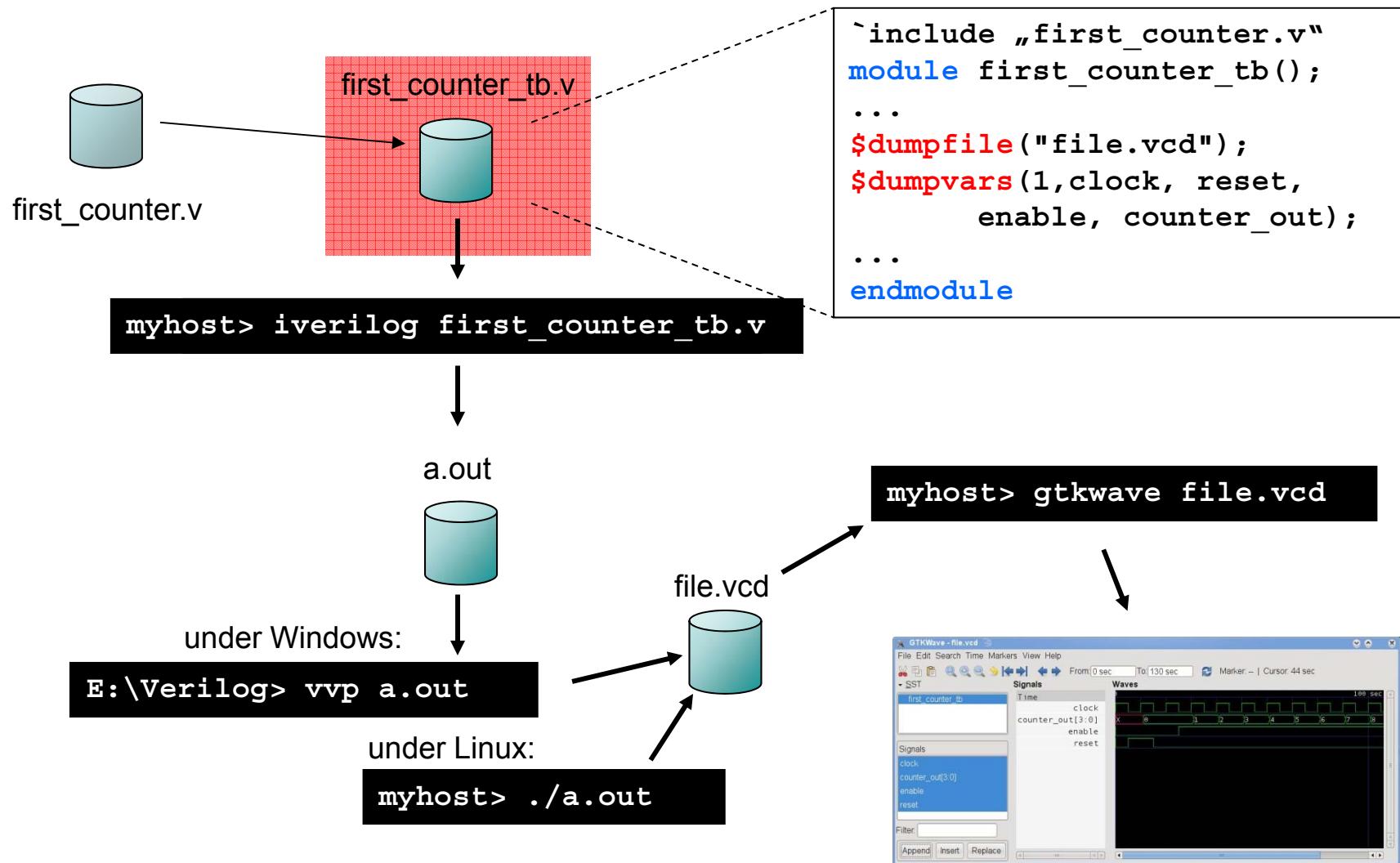
# Running your first Example

first\_counter.v

```
//### (2) Code ###

//-----Code Starts Here-----
// Since this counter is a positive edge triggered one,
// We trigger the below block with respect to positive
// edge of the clock.
always @ (posedge clock)
begin : COUNTER // Block Name
    // At every rising edge of clock we check if reset is active
    // If active, we load the counter output with 4'b0000
    if (reset == 1'b1) begin
        counter_out <= #1 4'b0000;
    end
    // If enable is active, then we increment the counter
    else if (enable == 1'b1) begin
        counter_out <= #1 counter_out + 1;
    end
end // End of Block COUNTER
```

# Running your first Example



# Running your first Example

first\_counter\_tb.v

```
// This is the TESTBENCH

// Include design file
`include "first_counter.v"
module first_counter_tb();
// Declare inputs as regs and outputs as wires
reg clock, reset, enable;
wire [3:0] counter_out;

// ### (1) Initialize Variables ###

// ### (2) Clock Generator and Device Connection ###

endmodule
```

# Running your first Example

```
// ### (1) Initialize Variables ###  
  
// Initialize all variables  
initial begin  
    // define file for simulation values dump, needed for GTKWave  
    $dumpfile("file.vcd");  
    // define signals to be recorded (in dumpfile) on actual design level  
    $dumpvars(1,clock, reset, enable, counter_out);  
  
    $display ("time\t clk reset enable counter");  
    $monitor ("%g\t %b %b %b",  
             $time, clock, reset, enable, counter_out);  
    clock = 1;           // initial value of clock  
    reset = 0;           // initial value of reset  
    enable = 0;          // initial value of enable  
    #5 reset = 1;        // Assert the reset  
    #10 reset = 0;       // De-assert the reset  
    #10 enable = 1;      // Assert enable  
    #100 enable = 0;     // De-assert enable  
    #5 $finish;          // Terminate simulation  
end
```

first\_counter\_tb.v

# Running your first Example

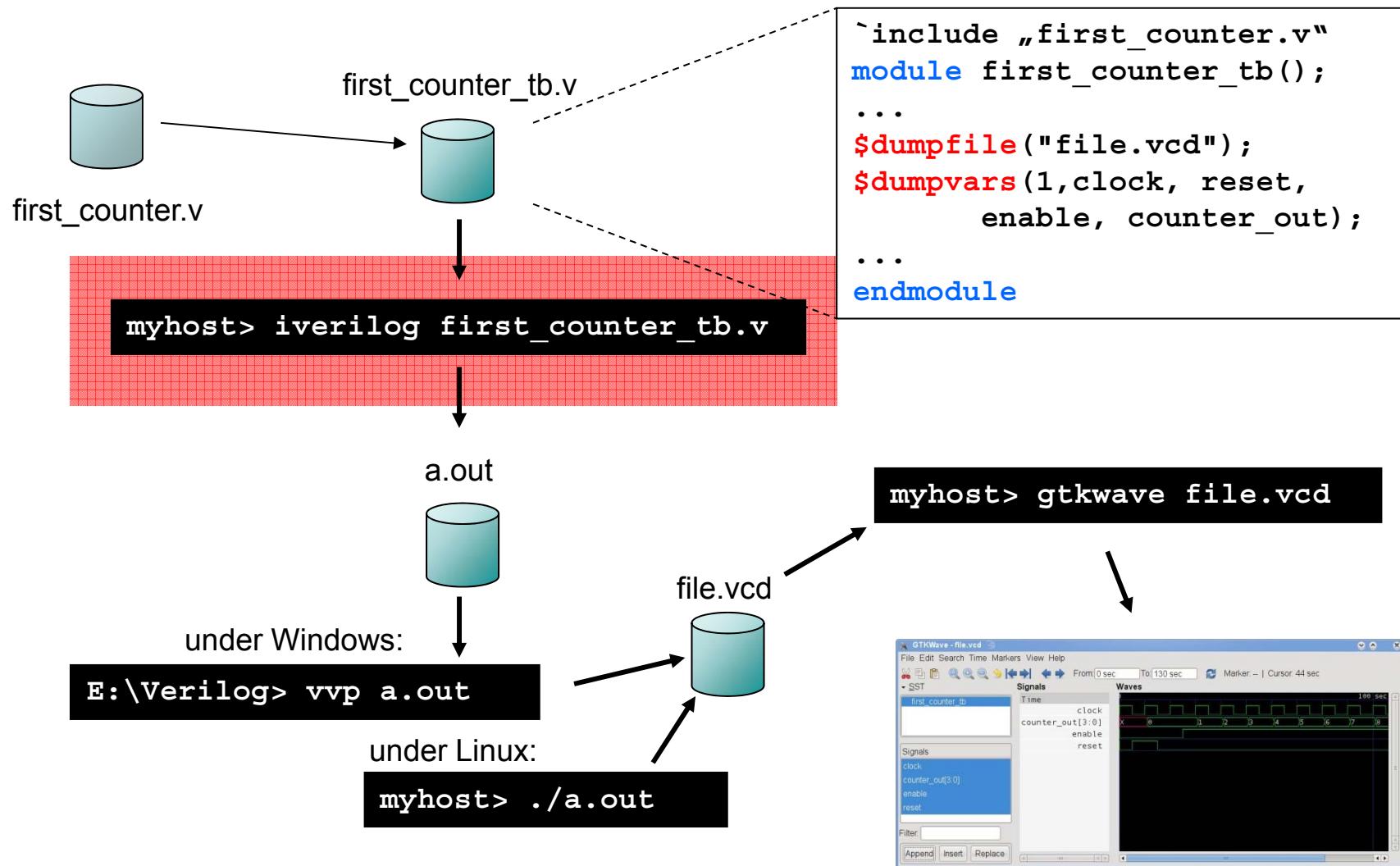
```
// ### (2) Clock Generator and Device Connection ###
```

first\_counter\_tb.v

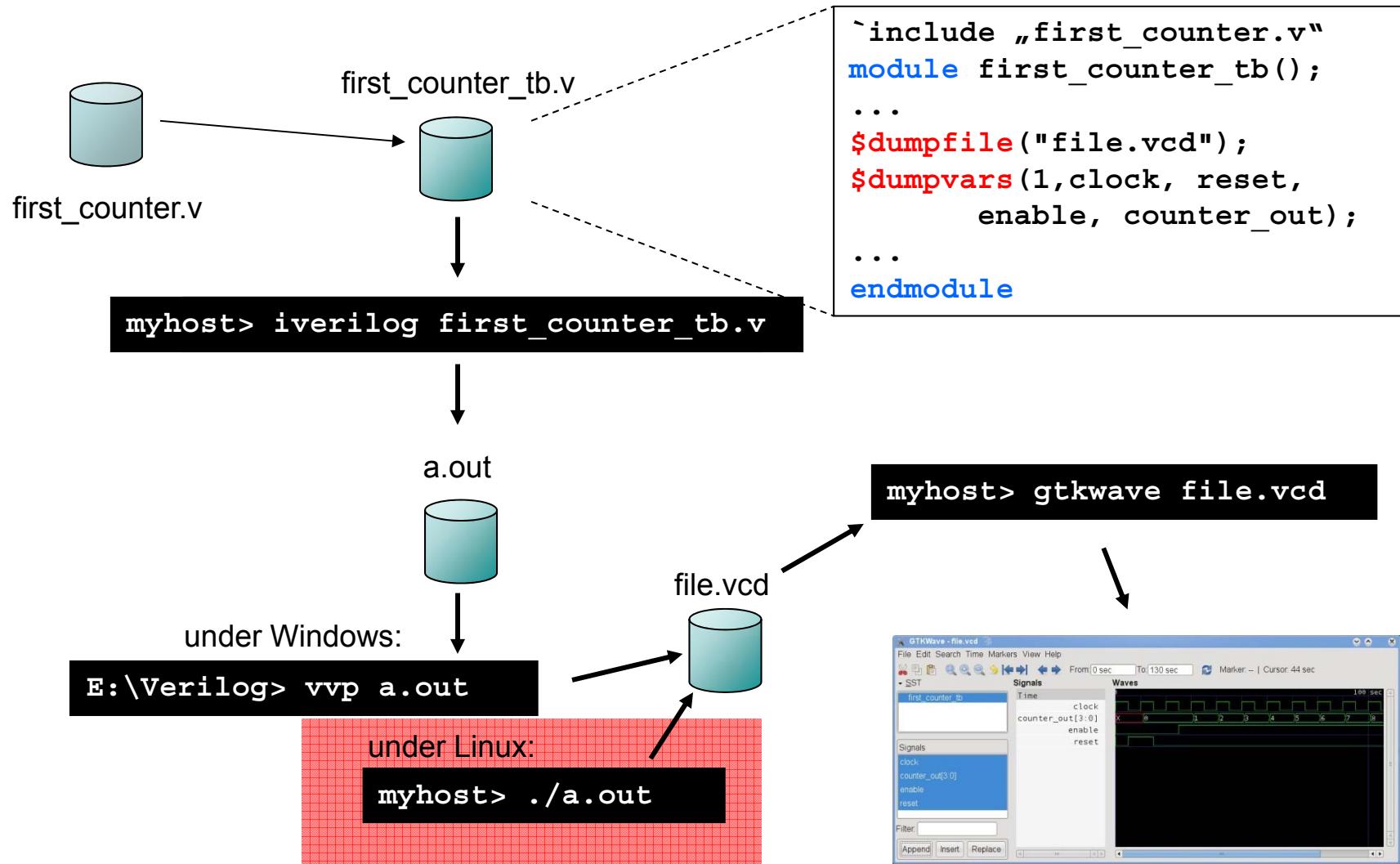
```
// Clock generator
always begin
#5 clock = ~clock; // Toggle clock every 5 ticks
end

// Connect DUT to test bench
first_counter U_counter (
clock,
reset,
enable,
counter_out
);
```

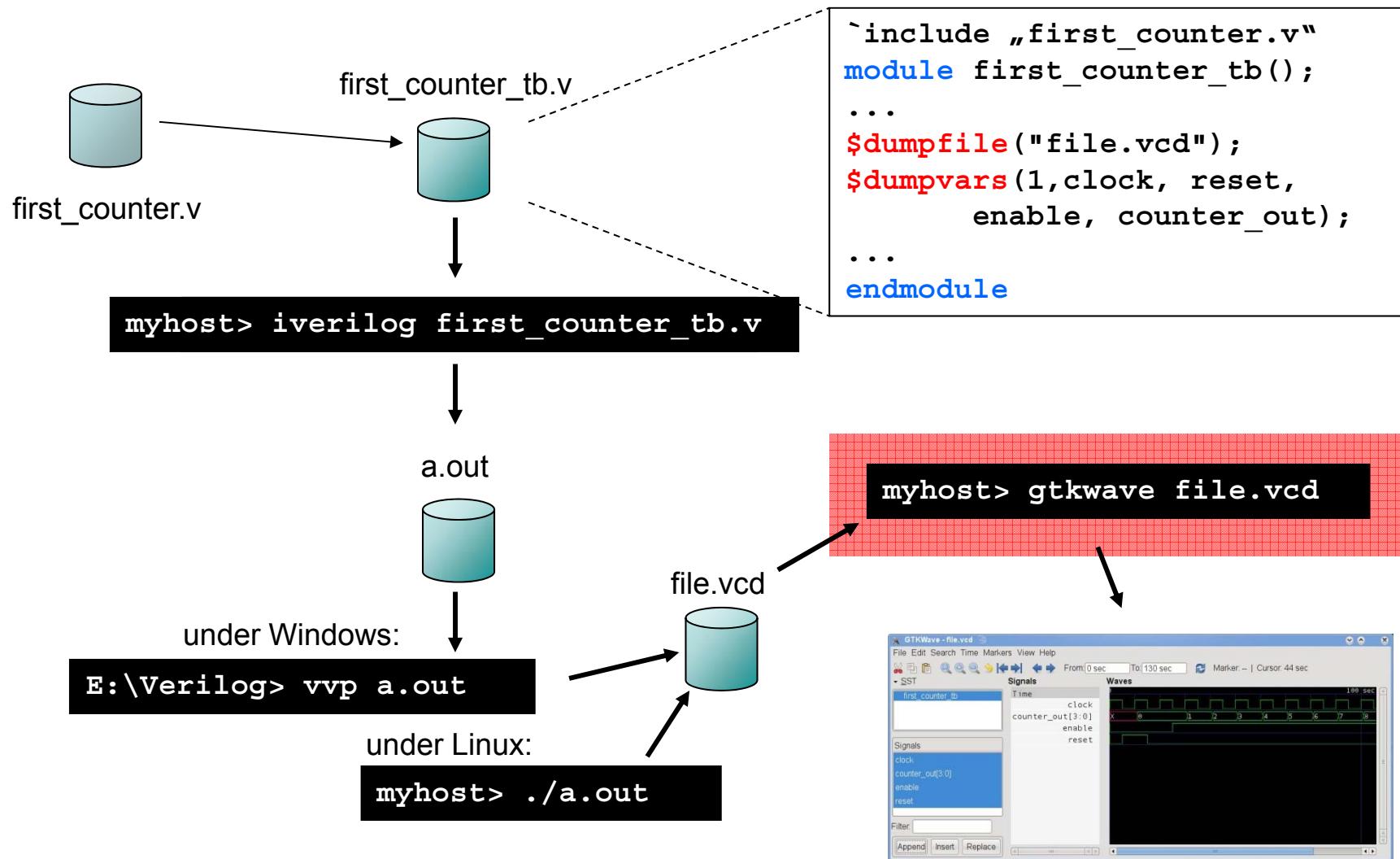
# Running your first Example



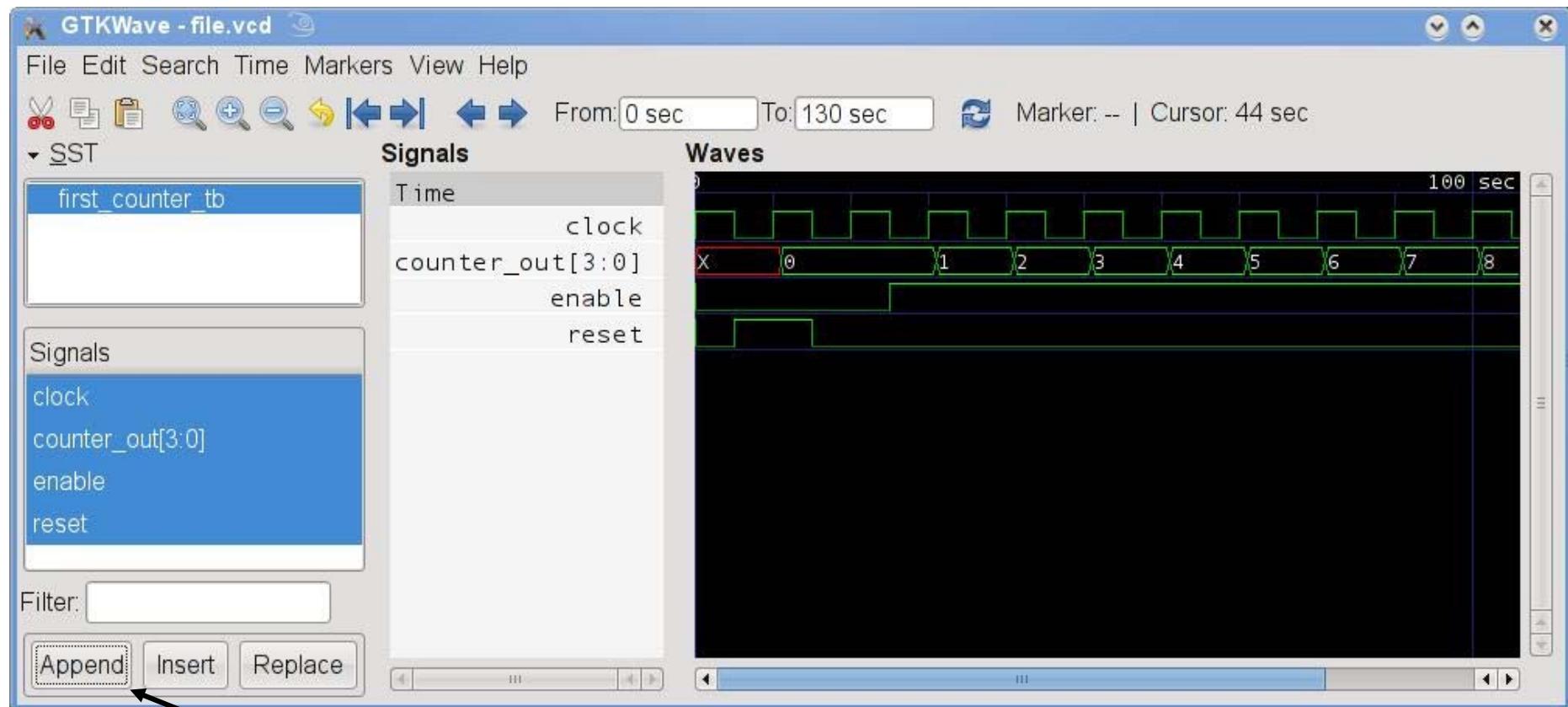
# Running your first Example



# Running your first Example



# Running your first Example

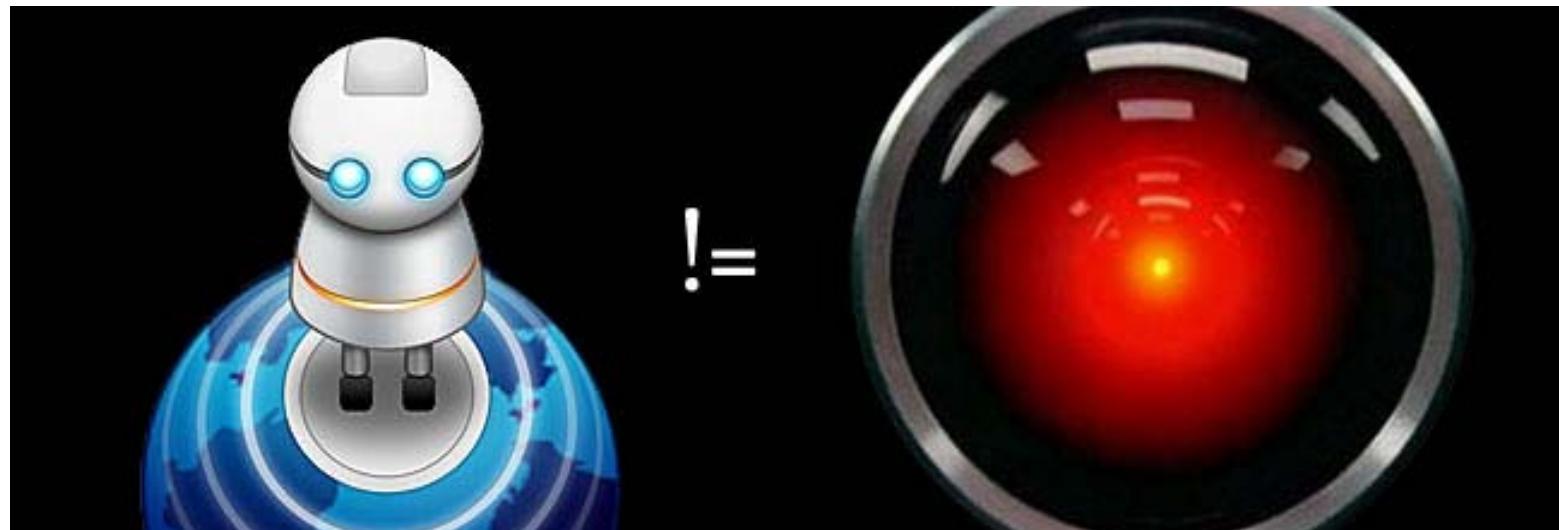


All signals have to be **Appended** in order to see the waveform

# Appendix 2: PLI



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



- Verilog PLI?WHAT
- Verilog PLI?WHY
- Verilog PLI?HOW
- Verilog PLI?Four Steps
- Verilog PLI?Evolution
- Verilog PLI?References

# Verilog PLI ?WHAT

Programming Language Interface (PLI) is a way to provide Application Program Interface (API) to Verilog HDL. .

Essentially it is a mechanism to invoke a C function from a Verilog code.

The construct which invokes a PLI routine in a Verilog code is usually called a system call.

The common system calls built-in to most Verilog simulators are \$display, \$monitor, \$finish etc.

# Verilog PLI ?WHY

PLI allows the user to create custom system calls, something that Verilog syntax does not allow us to do. Some of these are :

- Power Analysis
- Code coverage tools.
- Modifying the Verilog simulation data structure - more accurate delays.
- Custom output displays.
- Co-simulation.
- Design debug utilities.
- Simulation analysis.
- C-model interface to accelerate simulation.
- Testbench modeling.

- Write the functions in C/C++ code
- Compile them to generate shared libs (\*.DLL in Windows and \*.so in UNIX). Simulator like VCS allows static linking.
- Use these Functions in Verilog code
- Based on simulator, pass the C/C++ function details to simulator during compile process of Verilog Code (See user guide to understand how this is linked).
- Once linked just run the simulator like any other Verilog simulation.

## Problem description

Create a system call `$print_reg` which takes a register as parameter and prints its value. (This is a simplified version of `$display`). As an example, a verilog program like this:

```
module my_module;
    reg [3:0] r1;
    initial begin
        r1 = 4'ha;
        #100 $print_reg(r1);
        #100 r1 = 4'h3;
        #100 $print_reg(r1);
        #100 $finish;
    endmodule
```

Should produce the values 10 and 3 at times 100 and 300.

# Verilog PLI? Four Steps

*Step 1: Including the header files*

*Step 2 : Function prototype and Variable declaration*

*Step 3 : The essential data structure*

*Step 4: tf routines*

## *Step 1: Including the header files*

The file veriuser.c containing the main PLIC program must have  
in Verilog XL

```
#include <veriuser.h>  
  
#include <vxl_veriuser.h>  
in VCS  
  
#include <vcsuser.h>
```

the program must be compiled with *-include\_path\_name* option.

```
#include "include_path_name/veriuser.h"  
  
#include "include_path_name/vxl_veriuser.h"
```

# Verilog PLI? Four Steps

## *Step 2 : Function prototype and Variable declaration*

This part of the program contains all local variables and the functions that it invokes as part of the system call. In the present case, as explained in the next step, it will be as shown below.

```
int my_calltf(), my_checktf();
```

However, if the functions are in separate files, they should be declared as external functions.

```
extern int my_calltf(), my_checktf();
```

A typical PLI code, like any other C program, may need few other housekeeping variables.

## *Step 3 : The essential data structure*

There are a number of data structures that must be defined in a PLI program.

The exact number and syntax of these data structures vary from simulator to simulator.

Verilog-XL from Cadence Design Systems, for example, requires four such data structures and functions for any PLI routine to work

VCS, from Chronologic needs none, but needs a separate input file.

## ***Step 3 : The essential data structure***

For **Verilog-XL**

These data structures, in the order they should appear, are given below.

### ***1. veriuser\_version\_str***

This character pointer can be used to indicate a version of the compiled Verilog for your reference. It can be also set to NULL string. The string appears on standard out as well as goes to the Verilog log file when you run your compiled Verilog. A typical example of using it is shown below.

```
char *veriuser_version_str ="\n\
=====\\n\
VERILOG PLI FIRST_ATTEMPT \\n\
($print_reg) \\n\
=====\\n" ;
```

## *Step 3 : The essential data structure*

### 2. *endofcompile\_routines*

This data structure, as the name suggests, can be used for declaring functions that will be invoked at the end of the simulation. The present example does not need it. Nonetheless there has to be a default definition as shown below.

```
int (*endofcompile_routines[]) () = {0};
```

## *Step 3 : The essential data structure*

### 3. *err\_intercept*

This function, which returns either true or false (i.e. a boolean type), can be used to enhance the error detection mechanism of the code and can be ignored for small applications. A typical example of this routine would be as shown below.

```
bool err_intercept(level, facility,code)
int level; char * facility; char *code;
{ return (true); }
```

## ***Step 3 : The essential data structure***

### **4. veriusertfs**

The main interaction between the C-code that one writes and the Verilog simulator is done through a table.

The Verilog simulator looks at this table and figures out the properties that the system call corresponding to this PLI routine would be associated with.

This array, called veriusertfs in Verilog XL, may not be called by any other name as the simulator will not recognize that.

Each element of veriusertfs has a distinct function and is very important to achieve the overall objective of writing the PLI routine.

The number of rows in veriusertfs is same as that of the number of user-defined system calls, plus one for the last entry, which is a mandatory {0} .

## *Step 3 : The essential data structure*

### 4. veriusertfs

In the present case, the veriusertfs array should look like:

```
s_tfcell veriusertfs[] =  
{  
    /*** Template for an entry :  
    {usertask|userfunction,  
     data,  
     checktf(),  
     sizetf(),  
     calltf(),  
     misctf(),  
     "$tfname"  
    },  
    ***/  
    {usertask, 0, my_checktf, 0, my_calltf, 0, "$print_reg"},  
    {0} /* Final entry must be zero */  
}
```

## ***Step 4: tf routines***

The ***checktf*** routine

Usually it is a good practice to check two things in a PLI routine

- whether the total number of parameters are same as what is expected and
- whether each parameter is of the required type.

For example, in our case, we expect only one parameter to be passed and it must be a register type. This is done in the following function `my_checktf()`.

## ***Step 4: tf routines***

```
int my_checktf() {
    if (tf_nump() != 1) {
        tf_error("$print_reg:");
        tf_error("Usage:$print_reg(register_name);");
    }
    if (tf_typep(1) != tf_readwrite) {
        tf_error("$print_reg:");
        tf_error ("The argument must be a register type\n");
    }
}
```

The functions starting with `tf_` are the library functions and commonly known as utility routines.

- First the tf\_(task/function) PLI API was developed to allow adding user coded Verilog system tasks and functions written in C.
- The tf\_ interface was limited because design information could only be passed to and from C code by system task arguments.
- Second, in the early 1990s the acc\_ (access) PLI API was defined to remedy problems with the tf\_ interface.
- The acc\_ PLI interface was designed to allow accessing Verilog source information from within PLI API. Designs could be scanned using acc\_, **acc\_get\_next** source object grouping routines.
- The new PLI 2.0 vpi\_ routines API has been defined to simplify and improve Verilog PLI
- The API is called vpi\_ because routine and constant names start with "vpi" prefix to distinguish names from tf\_ and acc\_ names and to allow mixing of various PLI interfaces.
- The PLI 2.0 vpi\_ API replacement simplifies and improves tf\_ and acc\_ features

# Verilog PLI? Reference

- **Principles of Verilog PLI** by Swapnajit Mittra published from Kluwer Academic Publishers .
- *Verilog PLI Reference Manual*

# Appendix 3: Exercises and Processor Specification



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Processor: I/O

## ▪ Generic Parameters

`data_width` 8 bit or generic  $\geq 8$  bit.

→ indicates the size of a data register

`address_width` 12 bit or generic  $\geq 12$  bit

→  $2^{\text{address\_width}}$  addressable memory cells

## ▪ Ports

`clock` input (hardware is triggered on the rising clock edge).

`reset` input (active low, asynchronous).

`data_bus` connecting memory, I/O-devices and processor

→ Option 1: tristate bus

size: multiple of `data_width` (e.g.  $3 \times \text{data\_width}$ )

→ Option 2: multiplexer-based bus

size: multiple of `data_width` (e.g.  $3 \times \text{data\_width}$ )

for each direction (in and out)

`address_bus` from processor to memory, size: `address_width`

`p_select`: 1-Bit control signal (0: access memory; 1: access I/O devices)

`we_mem`: 1-Bit control signal to indicate write-access to the memory  
(1: write; 0: read).

`we_io`: 1-Bit control signal for i/o access: (1: write; 0: read)

`en_io`: 1-Bit control signal indicating i/o access (read or write)

`io_ready`: 1-Bit control signal (active high) to indicate if the addressed i/o is ready  
to receive/send data

# Processor: Architecture, Registers, Reset

## ▪ General Architecture

von Neumann architecture (program and data in the same memory)

## ▪ Registers

4 data registers ([data\\_reg\\_1](#), [data\\_reg\\_2](#), [data\\_reg\\_3](#), [data\\_reg\\_4](#))

width: `data_width`; initial values: 0; flexible use; registers which are currently not used as target for a operation maintain their values

program counter ([pc](#)): width: `address_width`; initial value: 0; after an instruction has been fetched from the memory the pc is pointing to the next instruction to be fetched; if the pc reaches its maximum value  $2^{\text{address\_width}} - 1$ , its next value will be 0

4 Flags ([Zero](#), [Carry](#), [Negative](#) and [Overflow](#))

## ▪ Reset

Active low

pc and data registers are initialized

Internal flags are initialized

# Processor: Instructions

## ▪ Instructions:

All bits of an instruction following the OpCode (size 6 bit) and the necessary parameters to the end of the instruction are „don't care“

Miscellaneous Instructions (2): NOP, STOP.

Arithmetic Instructions (4): ADD, ADDC, SUB, SUBC.

Logical Instructions (7): NOT, AND, OR, XOR, REA, REO, REX.

Shift / Rotate Instructions (7): SLL, SRL, SRA, ROL, ROLC, ROR, RORC.

Memory Access Instructions (5): LDC, LDD, LDR, STD, STR.

I/O Instructions (2): IN, OUT.

Jump Instructions (9): JMP, JZ, JC, JN, JO, JNZ, JNC, JNN, JNO..

## ▪ Flags:

In order to store particular results of some operations, there are 4 internal Flags:  
Zero, Carry, Negative and Overflow.

These flags are bits that can contain the values „0“ (flag cleared) or „1“ (flag set).

The flags are only affected when a data word resulting from an operation is written into a register ([flags are affected by the following instructions](#): ADD, ADDC, SUB, SUBC, NOT, AND, OR, XOR, REA, REO, REX, SLL, SRL, SRA, ROL, ROLC, ROR, RORC, LDC, LDD, LDR, IN).

The processor is able to work with [unsigned](#) and [signed](#) numbers. The [carry flag](#) is always [computed interpreting the operands as unsigned](#), while the negative and [overflow flags](#) are always [computed interpreting the operands as signed](#).

- **Zero Flag:**

- When executing one of the flag affecting operations, the result is evaluated; if the value is equal to 0, the zero flag is set, otherwise it is cleared.
- The initial value of the zero flag is 0.

- **Carry Flag**

- The carry flag is cleared by the following instructions: NOT, AND, OR, XOR, REA, REO, REX, ROL, ROR.
- When executing the ADD or ADDC instruction, the carry output of an unsigned addition of the two source operands is assigned to the carry flag.
- When executing the SUB or SUBC instruction, the carry output of an unsigned addition of the first source operand and the two.s complement of the second source operand is assigned to the carry flag.
- When executing the SLL or ROLC instruction, the most significant bit of the source operand is assigned to the carry flag.
- When executing the SRL, SRA or RORC instruction, the least significant bit of the source operand is assigned to the carry flag.
- LDC, LDD, LDR and IN don.t affect the carry flag.
- The initial value of the carry flag is 0.

- **Negative Flag:**

When executing the ADD, ADDC, SUB or SUBC instruction, the source operands are sign-extended by one bit; the additional bit of the result is assigned to the negative flag.

When executing one of the remaining flag affecting instructions, the most significant bit of the result is assigned to the negative flag.

The initial value of the negative flag is 0.

- **Overflow Flag**

- When executing the ADD, ADDC, SUB or SUBC instruction, the source operands are sign-extended by one bit; if the additional bit of the result differs from the (original) most significant bit of the result, the overflow flag is set, otherwise it is cleared.
- When executing the SLL instruction, the overflow flag is set if the most significant bit of the source operand differs from the most significant bit of the result, otherwise it is cleared.
- When executing one of the remaining flag affecting instructions, the overflow flag is cleared.
- The initial value of the overflow flag is 0.

# Processor: Instruction Set (I)

| Instruction Class | Spec. No. | Syntax     | Action                                                         | Instr.Code |
|-------------------|-----------|------------|----------------------------------------------------------------|------------|
| Miscellaneous     | 3.3.1.1   | NOP        | No operation                                                   | 0          |
|                   | 3.3.1.2   | STOP       | Stop processor                                                 | 1          |
| Arithmetic        | 3.3.1.3   | ADD X Y Z  | $X := Y + Z$                                                   | 2          |
|                   | 3.3.1.4   | ADDC X Y Z | $X := Y + Z + \text{Carry}$                                    | 3          |
|                   | 3.3.1.5   | SUB X Y Z  | $X := Y - Z$                                                   | 4          |
|                   | 3.3.1.6   | SUBC X Y Z | $X := Y - Z - \text{Carry}$                                    | 5          |
|                   | 3.3.1.7   | NOT X Y    | $X := \text{NOT } Y$                                           | 6          |
|                   | 3.3.1.8   | AND X Y Z  | $X := Y \text{ AND } Z$                                        | 7          |
| Logical           | 3.3.1.9   | OR X Y Z   | $X := Y \text{ OR } Z$                                         | 8          |
|                   | 3.3.1.10  | XOR X Y Z  | $X := Y \text{ XOR } Z$                                        | 9          |
|                   | 3.3.1.11  | REA *) X Y | $\text{LSB}(X) := \text{reduced\_and}(Y)$                      | 10         |
|                   | 3.3.1.12  | REO *) X Y | $\text{LSB}(X) := \text{reduced\_or}(Y)$                       | 11         |
|                   | 3.3.1.13  | REX *) X Y | $\text{LSB}(X) := \text{reduced\_xor}(Y)$                      | 12         |
|                   | 3.3.1.14  | SLL X Y    | $\text{Carry} \& X := Y \& '0'$                                | 13         |
|                   | 3.3.1.15  | SRL X Y    | $X \& \text{Carry} := '0' \& Y$                                | 14         |
| Shift / Rotate    | 3.3.1.16  | SRA X Y    | $X \& \text{Carry} := \text{MSB}(Y) \& Y$                      | 15         |
|                   | 3.3.1.17  | ROL X Y    | $X := \text{rotate\_left}(Y)$                                  | 16         |
|                   | 3.3.1.18  | ROLC X Y   | $\text{Carry} \& X := \text{rotate\_left}(\text{Carry} \& Y)$  | 17         |
|                   | 3.3.1.19  | ROR X Y    | $X := \text{rotate\_right}(Y)$                                 | 18         |
|                   | 3.3.1.20  | RORC X Y   | $\text{Carry} \& X := \text{rotate\_right}(\text{Carry} \& Y)$ | 19         |

\*) For REA, REO and REX operations the result is stored in the least significant bit while all other bits of the destination register are cleared.

# Processor: Instruction Set (II)

|               |          |     |     |   |                                  |    |
|---------------|----------|-----|-----|---|----------------------------------|----|
| Memory Access | 3.3.1.21 | LDC | X   | P | X := P                           | 32 |
|               | 3.3.1.22 | LDL | X   | P | X := mem(P)                      | 33 |
|               | 3.3.1.23 | LDR | X Y |   | X := mem(Y)                      | 34 |
|               | 3.3.1.24 | STD | X   | P | mem(P) := X                      | 35 |
|               | 3.3.1.25 | STR | X Y |   | mem(Y) := X                      | 36 |
| I/O           | 3.3.1.26 | IN  | X   |   | X := data_from_input_device      | 37 |
|               | 3.3.1.27 | OUT | X   |   | data_to_output_device := X       | 38 |
| Jump          | 3.3.1.28 | JMP |     | P | Unconditional Jump to P          | 48 |
|               | 3.3.1.29 | JZ  |     | P | Jump to P if zero flag = '1'     | 49 |
|               | 3.3.1.30 | JC  |     | P | Jump to P if carry flag = '1'    | 50 |
|               | 3.3.1.31 | JN  |     | P | Jump to P if negative flag = '1' | 51 |
|               | 3.3.1.32 | JO  |     | P | Jump to P if overflow flag = '1' | 52 |
|               | 3.3.1.33 | JNZ |     | P | Jump to P if zero flag = '0'     | 53 |
|               | 3.3.1.34 | JNC |     | P | Jump to P if carry flag = '0'    | 54 |
|               | 3.3.1.35 | JNN |     | P | Jump to P if negative flag = '0' | 55 |
|               | 3.3.1.36 | JNO |     | P | Jump to P if overflow flag = '0' | 56 |

When an **unknown OpCode** is decoded, it is **treated as a NOP instruction** and the user is informed of the illegal OpCode by an assertion.

When a **STOP instruction** is decoded the processor stops fetching new instructions and reaches an explicit deadlock state. In this state there are no changes to any registers (data registers, flags, PC) and no communication to memory or I/O devices.

- **I/O Instructions:**

IN instruction: input-device sends the content of its data register to the processor if the device is ready; if the device is not ready, the processor is blocked until the input-device is ready.

OUT instruction: output-device receives a data word from the processor and stores it in its data register if the device is ready; if the device is not ready, the processor is blocked until the output-device is ready.

- **Flags:**

In order to store particular results of some operations, there are 4 internal Flags:  
[Zero](#), [Carry](#), [Negative](#) and [Overflow](#).

These flags are bits that can contain the values „0“ (flag cleared) or „1“ (flag set).  
The flags are only affected when a data word resulting from an operation is written into a register ([flags are affected by the following instructions](#): ADD, ADDC, SUB, SUBC, NOT, AND, OR, XOR, REA, REO, REX, SLL, SRL, SRA, ROL, ROLC, ROR, RORC, LDC, LDD, LDR, IN).

The processor is able to work with [unsigned](#) and [signed](#) numbers. The [carry flag](#) is always [computed interpreting the operands as unsigned](#), while the negative and [overflow flags](#) are always [computed interpreting the operands as signed](#).

# Processor: I/O Devices

## ▪ I/O Instructions:

All I/O-devices contain two registers, one register for storing a data word (size `data_width`), and a one-bit register storing the information if the data register is empty or not.

The external interface of the I/O-devices is composed of the following signals:

- `Clock` input (I/O-devices are triggered on the positive clock edge).
- `Reset` input (asynchronous, active low).
- Control signal `io_ready` to indicate if the I/O-device is ready to send/receive data to/from the processor (active high).
- Control signal `en_io` to indicate access to the I/O-device (active high).
- Control signal `we_io` to distinguish between read- (if 0) and write-access (if 1) to the I/O-device.
- Data bus `data_processor2output` from the processor (only output device, width: `data_width`).
- Data bus `data_output2ext` to the outer environment (only output device; size according to 1.1.1).
- `out_data_ready` signal to the outside (only output device).
- `out_data_request` signal from the outside (only output device).
- Data bus `data_input2processor` to the processor (only input device, size: `data_width`).
- Data bus `data_ext2input` from the outside (only input device; size: `data_width`).
- `in_data_ready` signal from the outside (only input device).
- `in_data_request` signal to the outside (only input device).

# Processor: I/O Devices

## Input device

- The device includes a register for storing a data word from the outside bus to be sent to the processor. If the internal register contains data, the `io_ready` signal of this device has to be set (since the device is now ready to send this data to the processor); otherwise it has to be cleared.
- If the `en_io` input is set, the `we_io` input is set to 0 and the internal register contains data, this data has to be made visible at the port `data_input2processor`; in all other cases `data_input2processor` is set to 0.
- If the internal register is empty, the `in_data_request` signal of this device has to be set; otherwise it has to be cleared.
- If the `in_data_ready` input is set from outside and the internal register is empty, the data at the port `data_ext2input` has to be stored in the internal register; in all other cases (except reset) the content of the internal register must not change.

## Output device

- The device includes a register for storing a data word from the processor to be sent to the outside bus.
- If the internal register is empty, the `io_ready` signal of this device has to be set (since the device is now ready to receive data from the processor); otherwise it has to be cleared.
- If the `en_io` input is set, the `we_io` input is set to 1 and the internal register is empty, the data at the port `data_processor2output` has to be stored in the internal register; in all other cases the content of the internal register must not change (except reset).
- If the `out_data_request` input is set and the internal register contains data, this data has to be made visible at the port `data_output2ext` and `out_data_ready` has to be set; in all other cases `data_output2ext` has to be set to 0x0 and `out_data_ready` has to be cleared.

# Exercises:

- **Step 1:**

Installation of Icarus and running example given in installation instructions

- **Step 2:**

Describe a behavioral XOR function and structurally build an 8-bit adder based on this basic XOR function

- **Step 3:**

Model a simple ALU and a Memory. Design a controller which is doing the following:

1. File -> Memory
2. Add all Memory Data
3. Result -> File

# Exercises:

- **Step 4:**

Model a complex ALU, which can operate all arithmetic operations

- **Step 5:**

Model a processor based on the ALU with the following reduced  
Instruction set: Misc, Arithmetic, Memory access

- **Step 6:**

Enhance your Processor with Jump Instructions

# Appendix 4: BNF of Verilog & Quick Reference Card



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# BNF & Quick Verilog Card



- Available as html at <http://www.ies.tu-darmstadt.de> -> Teaching -> Lecture Material (protected area)