Hardware Description Languages

Andre Mitterbacher
andre.mitterbacher@fhv.at

Fachhochschule Vorarlberg

v01 – 10.10.2016: Corrected typos.

Introduction to HDL

Complexity in digital designs

- Modern digital circuitry comprises millions of transistors.
- Modern FPGAs include numerous functionalities.
- It is not humanly possible to comprehend such complex systems in their entirety.
- We need to find methods of dealing with the complexity $\rightarrow$ abstraction.

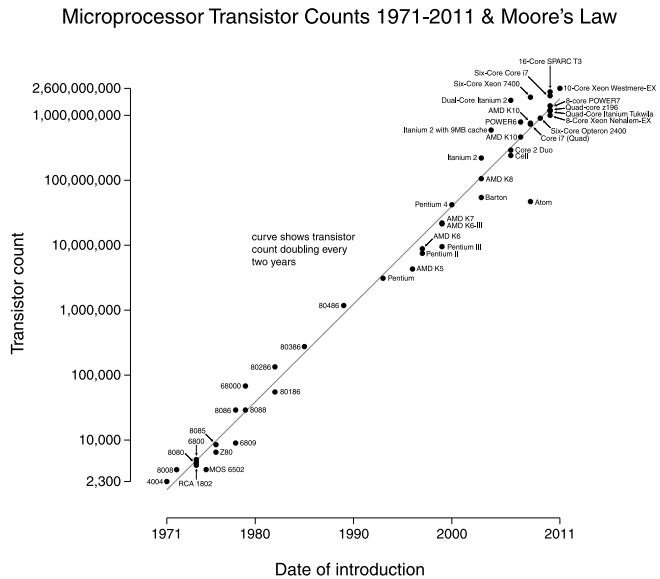


Figure: Moore's law: Over the history of computing hardware, the number of transistors in a dense integrated circuit doubles approximately every two years.

Source: By Wgsimon via Wikimedia Commons (link)

Abstraction layers

- Structural: Description in terms of interconnection of more primitive components. (FPGA: Connection of LE, memories and additional resources)
- Behavioral: Description by input/ output response. (i.e. description of the desired logic).
- Physical: Description by a physical layout. (FPGA: Place and Route)

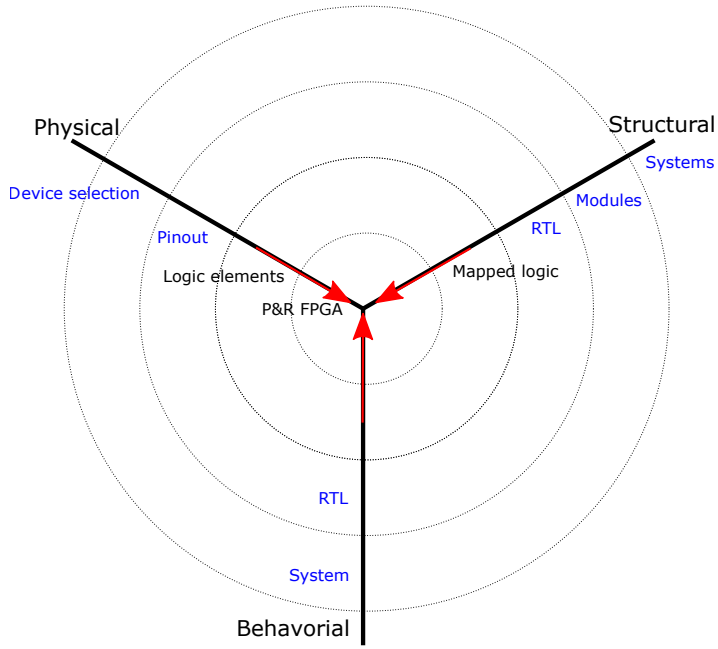


Figure: Abstraction layers applied to FPGA design. The blue design inputs are mostly done by humans. The rest is machine work.

Source: own work

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.

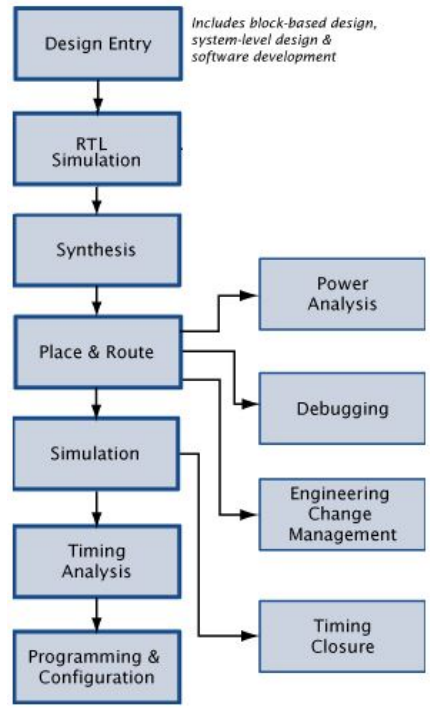


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.

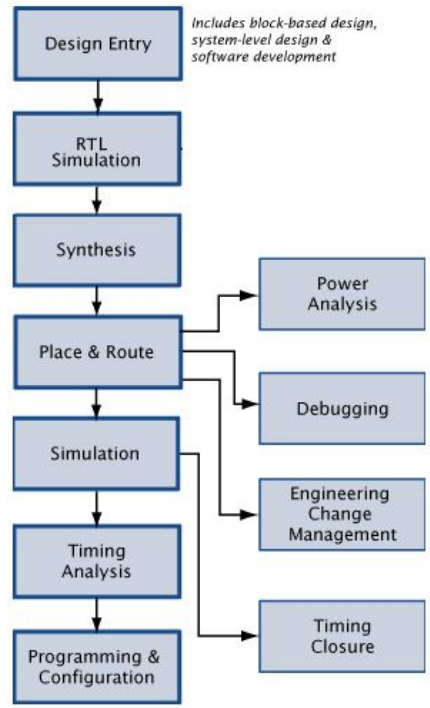


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.
- Analysis: Analyze the design entry for errors and convert it to RTL netlist.

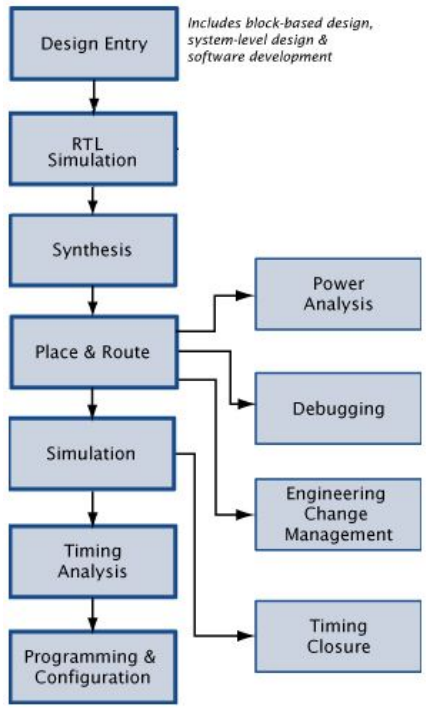


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.
- Analysis: Analyze the design entry for errors and convert it to RTL netlist.
- Synthesis: Map the RTL netlist to the available logic cells (fab design kit, PLD logic elements) → gate level netlist.

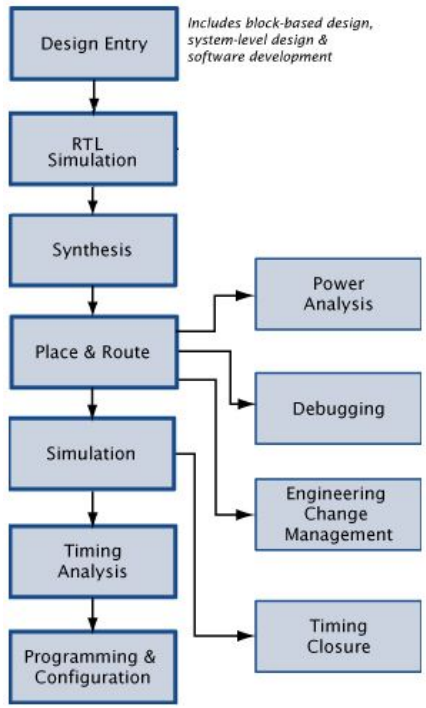


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.
- Analysis: Analyze the design entry for errors and convert it to RTL netlist.
- Synthesis: Map the RTL netlist to the available logic cells (fab design kit, PLD logic elements) $\rightarrow$ gate level netlist.
- Place and Route: Place the logic cells in the available digital Si area or in the PLD.

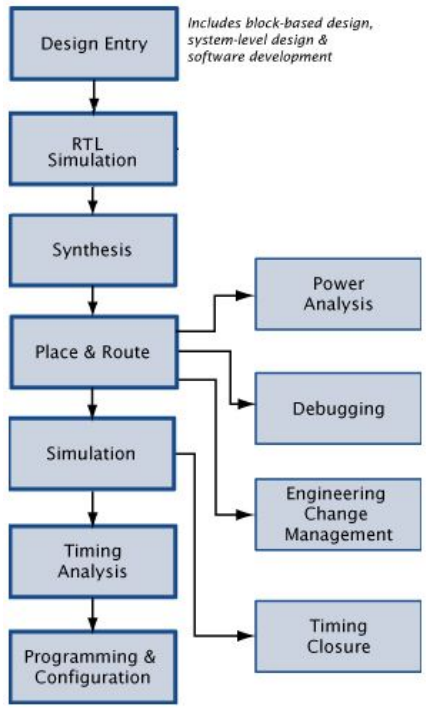


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.
- Analysis: Analyze the design entry for errors and convert it to RTL netlist.
- Synthesis: Map the RTL netlist to the available logic cells (fab design kit, PLD logic elements) $\rightarrow$ gate level netlist.
- Place and Route: Place the logic cells in the available digital Si area or in the PLD.
- Simulation of the gate level netlist together with estimated timing delays.

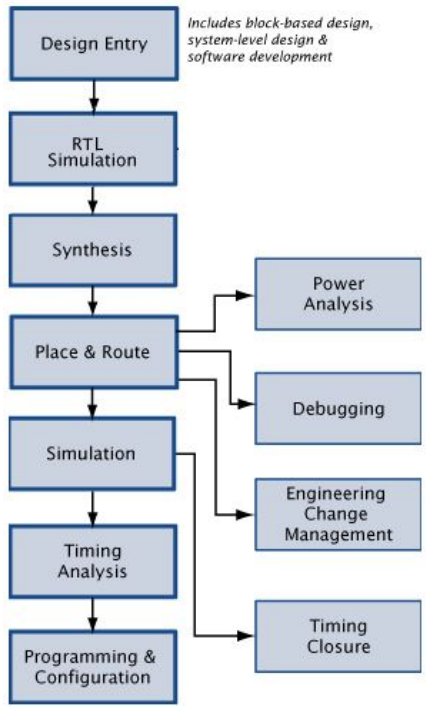


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.
- Analysis: Analyze the design entry for errors and convert it to RTL netlist.
- Synthesis: Map the RTL netlist to the available logic cells (fab design kit, PLD logic elements) $\rightarrow$ gate level netlist.
- Place and Route: Place the logic cells in the available digital Si area or in the PLD.
- Simulation of the gate level netlist together with estimated timing delays.
- Timing analysis use Static Timing Analysis to verify that setup and hold times are ok for all FFs.

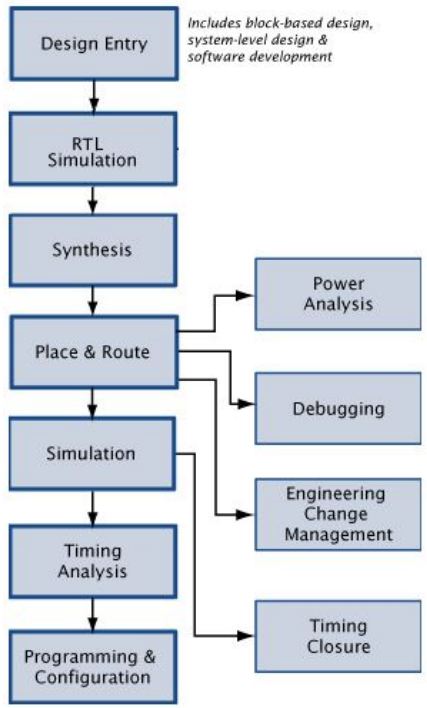


Figure: Digital design flow.
Source: Altera Quartus help.

Digital Design Toolchain

- Design Entry: Either schematic or HDL description of the intended logic.
- Simulation: Verify the correct behavior of the design.
- Analysis: Analyze the design entry for errors and convert it to RTL netlist.
- Synthesis: Map the RTL netlist to the available logic cells (fab design kit, PLD logic elements) → gate level netlist.
- Place and Route: Place the logic cells in the available digital Si area or in the PLD.
- Simulation of the gate level netlist together with estimated timing delays.
- Timing analysis use Static Timing Analysis to verify that setup and hold times are ok for all FFs.
- Programming of Chip manufacturing
  - IC design → Tape-out GDS2 files are sent to the fab. Mask and IC manufacturing starts.
  - PLD → programming files are loaded to the PLD.

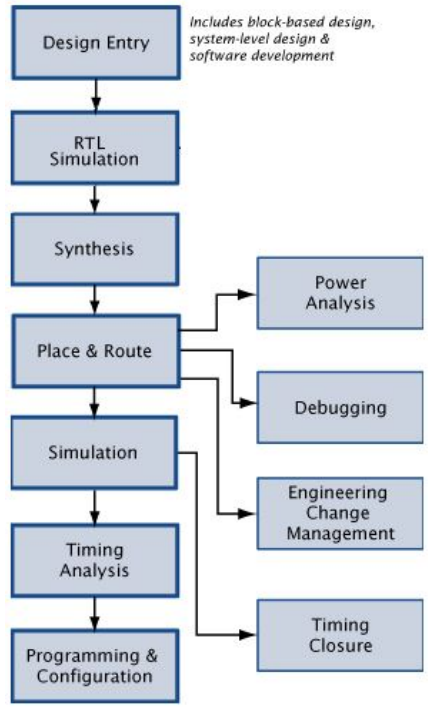


Figure: Digital design flow.
Source: Altera Quartus help.

Hardware Description Language

In order to facilitate

- the usage of abstraction layers
- the description of complex logic
- advanced test and verification methods
- revision control of the design

Hardware Description Languages were developed.

A model of the digital circuit is described – NOT programmed!

Differences to a programming language:

- Concurrent statements
- Time can be explicitly expressed

History

PLD development lead to the languages for the description of their behavior.
(1983: Data I/O introduced ABEL.)
In the late 1980's two HDL emerged: VHDL and Verilog.

VHDL

- Developed by the U.S Department of Defense to describe the behavior of ASICs.
- 1987 Submitted as IEEE standard 1076.
- Due to the Department of Defense requiring as much of the syntax as possible to be based on Ada.
- Strongly typed.

Verilog

- Created by Prabhu Goel and Phil Moorby during the winter of 1983/1984 for Gateway Design Automation.Cadence buys Gateway Design Automation in 1990.
- 1995 Cadence transferred Verilog into the public domain and submitted as IEEE standard 1364.
- 2002 SystemVerilog adds advanced verification $\rightarrow$ IEEE standard 1800.
- Verilog is based on C.
- Weakly typed.

Development

1. Description (design entry) of logic for specification and documentation.
2. Simulation / Verification of the described logic was so attractive that simulators were developed.
3. Synthesis of the logic to a definition of the physical implementation of the circuit.
4. Enhanced verification support (e.g. self-checking) was added to account for the growing complexity.

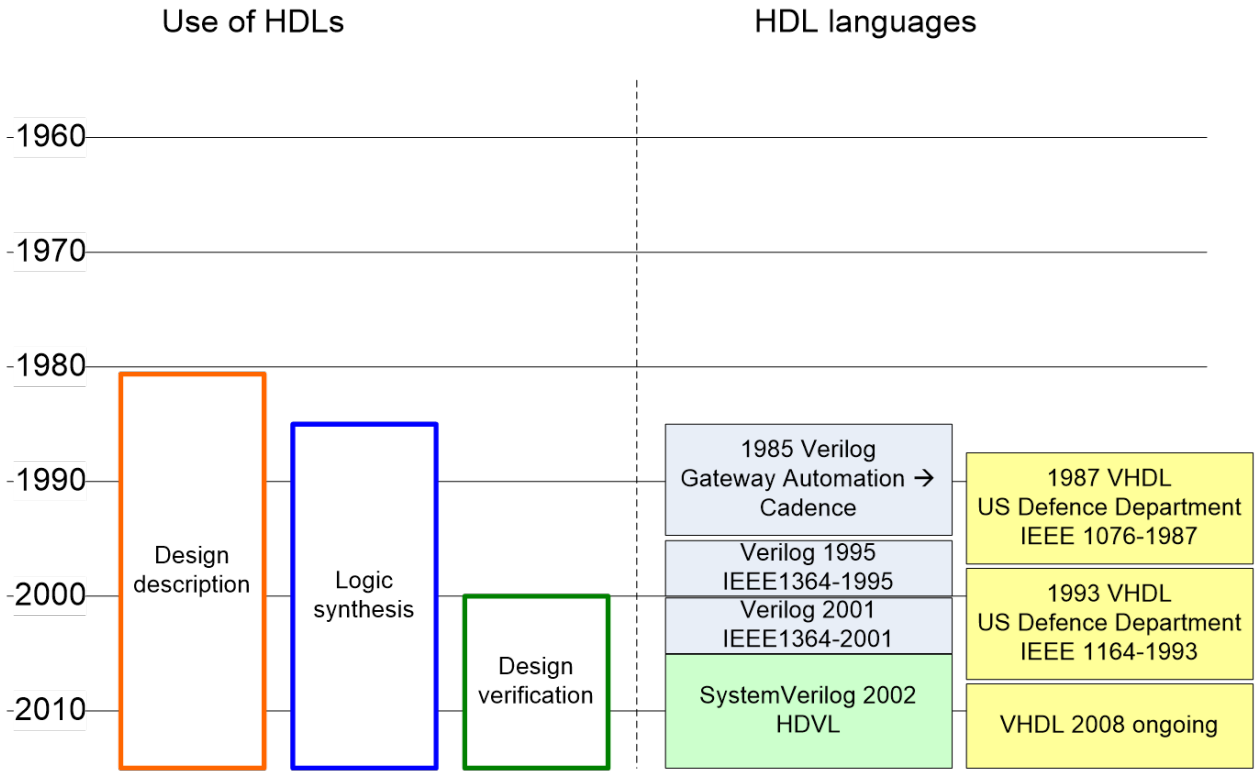


Figure: Development of HDLs

HDL for Digital Design Entry

HDLs are used to do the design entry on RTL level.

- RTL → Register Transfer Level (i.e. it is described how register are linked to other registers and IOs with combinatorial logic)
- Therefore a HDL needs to offer possibilities to describe:
  - Inputs and Outputs of a module (ports)
  - Hierarchy of modules (connection of submodules)
  - Connection of signals (similar to a wire or a PCB track)
  - Combinatorial logic (gates but also behavioral)
  - Sequential logic → FFs (FlipFlops)

HDL for Digital Verification

It is essential to have the possibility to simulate the behavior of a digital design!

- Full custom IC design have huge NRE (non-recurring engineering) cost ($\approx$ Mio USD). $\rightarrow$ Tape-out is only signed off when verification coverage is close to 100%.
- FPGA designs offer only limited debug possibilities.

$\rightarrow$ Before a design is implemented (FPGA or IC) it needs to be verified thoroughly!

HDL for Digital Verification

It is essential to have the possibility to simulate the behavior of a digital design!

- Full custom IC design have huge NRE (non-recurring engineering) cost ($\approx$ Mio USD). $\rightarrow$ Tape-out is only signed off when verification coverage is close to 100%.
- FPGA designs offer only limited debug possibilities.

$\rightarrow$ Before a design is implemented (FPGA or IC) it needs to be verified thoroughly!

Therefore a HDL needs to offer the possibility to

- Drive signals
- Check signals (e.g. compare against expected values)
- Output and log signals
- Advance the time

The code that is generated to verify a digital design is typically called a test bench.

- In test benches code elements can be used that are not synthesizable.

Verification vs. Validation

- Validation: The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with verification."
- Verification: The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation."

Source: IEEE. "IEEE Guide–Adoption of the Project Management Institute (PMI) Standard A Guide to the Project Management Body of Knowledge (PMBOK Guide)–Fourth Edition". p. 452. doi:10.1109/IEEESTD.2011.6086685. Retrieved 7 December 2012.

HDL for Digital Synthesis

Synthesis means "conversion of the higher level abstract description of the design to the actual components on the gate and flip-flop level" [RJ08].

- This is done by the synthesis tool.
- Only the synthesizable part of HDL expressions can be processed (`$fprintf("Hello World")` is perfectly fine Verilog code but cannot be synthesized).
- To ensure that the algorithm understands the described logic, design templates shall be used (There is exactly one way how a FF needs to be coded).

HDL for Digital Synthesis

Synthesis means "conversion of the higher level abstract description of the design to the actual components on the gate and flip-flop level" [RJ08].

- This is done by the synthesis tool.
- Only the synthesizable part of HDL expressions can be processed (`$fprintf("Hello World")` is perfectly fine Verilog code but cannot be synthesized).
- To ensure that the algorithm understands the described logic, design templates shall be used (There is exactly one way how a FF needs to be coded).

There are different synthesis tools for FPGAs, IC design.

- FPGAs for IC prototyping.
- Sim and synthesis tools need to behave the same!

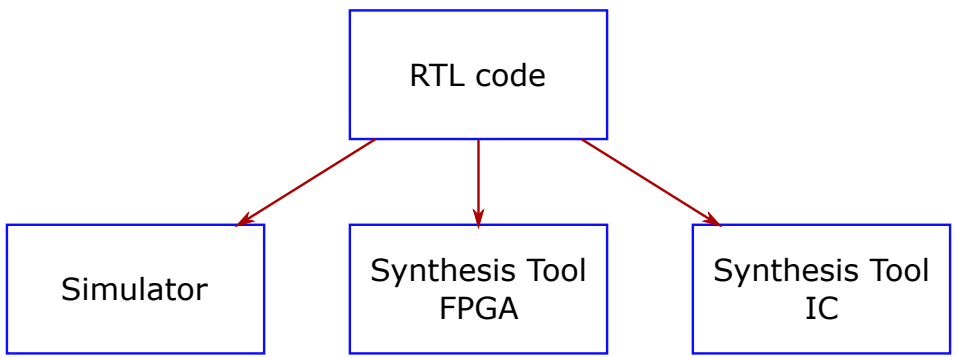


Figure: RTL code is used by more than one tool.

Design flow with HDLs

1. Start (w/o HDL) to structure the problem
   - Toplevel / Division into modules / Interfaces between modules
   - Block diagram
   - State charts

Design flow with HDLs

1. Start (w/o HDL) to structure the problem
   - Toplevel / Division into modules / Interfaces between modules
   - Block diagram
   - State charts
2. Design entry for modules
   - It's not programming → Think in digital logic!
   - Choose your favorite editor.
   - Use Version Control (e.g. SVN `https://tortoisesvn.net/index.de.html`).

Design flow with HDLs

1. Start (w/o HDL) to structure the problem
   - Toplevel / Division into modules / Interfaces between modules
   - Block diagram
   - State charts
2. Design entry for modules
   - It's not programming → Think in digital logic!
   - Choose your favorite editor.
   - Use Version Control (e.g. SVN `https://tortoisesvn.net/index.de.html`).
3. Verification
   - Verify that the design of your submodule/module fulfills the specification.
   - When designing think also about how to verify the module.

Design flow with HDLs

1. Start (w/o HDL) to structure the problem
   - Toplevel / Division into modules / Interfaces between modules
   - Block diagram
   - State charts
2. Design entry for modules
   - It's not programming → Think in digital logic!
   - Choose your favorite editor.
   - Use Version Control (e.g. SVN `https://tortoisesvn.net/index.de.html`).
3. Verification
   - Verify that the design of your submodule/module fulfills the specification.
   - When designing think also about how to verify the module.
4. Synthesis
   - After verification the design can be deployed onto the target hardware with the Synthesis Tool.
   - A post-simulation simulation can be done to proof that the synthesized netlist complies with the spec.
   - Check the design on the target hardware.

Programmable logic devices (PLD)

Programmable logic devices (PLD)

A brief review from BuS2 – let's use the decoder for a seven segment display here:

S	bcd[3]	bcd[2]	bcd[1]	bcd[0]	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

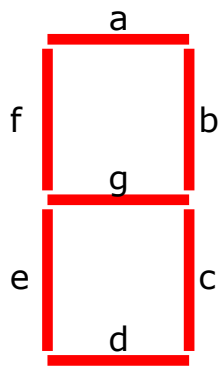


Figure: Seven segment display.

- How can this functionality be implemented?

ASIC

An ASIC is an Application Specific Integrated Circuit.

- It is custom designed and manufactured for a specific use.
- Examples: Audio processor for a hearing aid, a control IC that controls the power electronics in your smartphone charger, ...
- Pro: Best performance, exclusive use, IP protection
- Con: Highest NRE[1] cost, highest risk

[1] non recurring engineering

ASSP

An ASSP is an Application Specific Standard Product.

- ICs that are dedicated to a specific application market and sold to more than one user (and hence, standard) in contrast to ASICs.
- Examples: 74xx family ICs, power electronic control ICs, 555, Micro-controllers ...
- Pro: Cheap off the shelf solution. Specifically designed for a certain application. Risk depends on the IC.
- Con: Easy to copy. Constrained to the functionality of the IC. Small room for differentiation.

PLD

A PLD is a Programmable Logic Device.

- ICs that contain logic building blocks which are connected after wafer fabrication to offer the desired function.
- Examples: FPGA, CPLD, PAL, PLA
- Pro: Short development cycle, Low NRE cost, bug fixes at lower cost.
- Con: Power consumption, higher per unit cost, usually no analog peripherals on board

PLD

A PLD is a Programmable Logic Device.

- ICs that contain logic building blocks which are connected after wafer fabrication to offer the desired function.
- Examples: FPGA, CPLD, PAL, PLA
- Pro: Short development cycle, Low NRE cost, bug fixes at lower cost.
- Con: Power consumption, higher per unit cost, usually no analog peripherals on board
- PLD types and building blocks were already covered in BuS2.
- We will review all that when working with the devices.

Verilog – SystemVerilog

Basic elements

A brief introduction into SystemVerilog showing the main building blocks of a design. Refer to the example design for more information.

- Comments:
  - // starts a line comment
  - /* start a block comment
  - */ ends a block comment
- Case sensitive
- White spaces and tabs are not critical.

Listing 1: SV comments and names

```
1 /* This is a block comment */
2
3 // A line comment
4
5     small != SmaLL;
6
7
8         y = x      + 1;
9         y         =x+1;
```

Numbers

- Integers: The notation contains value, width und number format.
- Floating point: Decimal numbers with floating point . and *e* for powers of ten.

Listing 2: SV numbers

```
// integers
123                 // decimal 123, unknown bit width
8'ha5               // hexadecimal a5, 8 bits wide
'o125               // octal 125, unknown bit width
-4'd6               // two's complement of 6, 4bits wide, thus 4'b1010
8'b0000_1010        // binary grouped using '_'

//floating point
1.2                 // decimal 1.2
3.4e2               // 340
3.4E3               // also 340
```

Strings

- Strings are mainly used in simulation.
- Data type string

Listing 3: SV strings

```
1 // strings
2 string s1;                          // strings are initially ""
3 string s2 = "I am a string";        // initial values can be used
4
5 s1 = "That's life";                 // assign a value to a string
6 s2 = {s2,s1};                       // concatenate strings
7 $display(s1.len);                   // output string properties
```

Data types

Data types can be split into

- Synthesizable boolean and integer types used to describe RTL logic of digital systems (logic, integer)
- Support data types for test benches (float, double, string, time)

Digital signals can have four different values:

- 0 (logic low)
- 1 (logic high)
- Z (high-ohmic, not driven)
- X (unknown)

Note: The high-ohmic 'Z' value cannot be used inside an FPGA. Output pins can use tristate drivers to act like 'Z' towards other ICs. There are also two-valued (0,1) signals available for test benches to improve simulation time.

- Signals can be grouped to vectors (buses).
- Integers and logic signals can be assigned to each other w/o problems (SV is weakly typed).
- Signals can be concatenated using a,b notation.

Assignments

The description of signal routings can be done using assignments.

- Connection of submodules.
- Connection of other signals.
- Concatenation of signals.
- 'Simple' logic operations.

Listing 4: Assignments

```
1  // Assignments
2  logic [2:0] x;
3  logic [2:0] y;
4  logic [2:0] z;
5  logic [5:0] wide;
6  logic [1:0] lsbs;
7
8  assign y = x & y;              // bit wise AND
9  assign wide = {y,x}            // 6bit vector containing yyy_xxx
10 assign lsbs = {x[0],y[0]};     // Merge the LSBs of y and x
11                                // into a new vector 'small'
```

Logic / Integer / Assignments

Listing 5: Logic and integer data types

```
1  // Create an 'adder' of two 8bit signals
2  logic [7:0] x1;                    // 8bit vector
3  logic [7:0] x2;                    // 8bit vector
4  logic [7:0] y,z;                   // Two 8bit vectors
5  logic co;                          // 1bit carry out
6  integer result;                    // integer data type is 32bit wide
7
8  assign {co,y} = x1 + x2;           // Add two 8bit numbers and
9                                     // assign the result to the 8bit
10 assign result = {co,y};            // Assign the logic vector to an
11                                    // integer data type --> no problem!
12 // overflow
13 assign z = x1+x2;                  // Syntactically ok
14                                    // but overflow is possible :-(
```

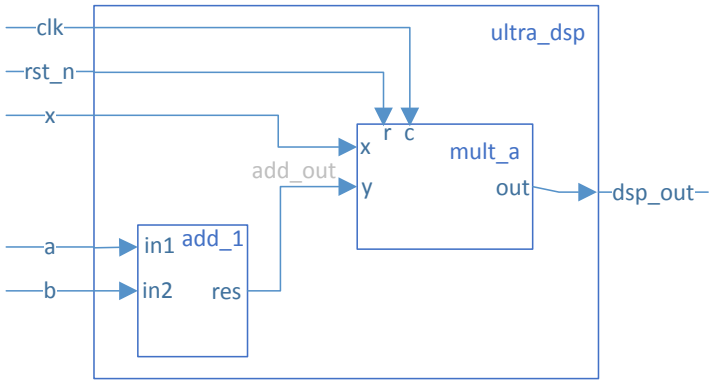
Note: Logic vectors of different sizes can be assigned to each other (weakly typed).

- Left hand side is smaller than the right hand side → data is truncated.
- Left hand side is wider than the right hand side → MSBs are filled with zeros (unsigned) or the MSB of the left hand side (signed).

Here the designer needs to be extremely careful to avoid overflows!

Design units

- Encapsulate parts of the code.
- Form larger parts
- The Toplevel unit interfaces to the outside world.
- IOs are defined using input and output.
- Interconnects are done using logic data types.



Listing 6: SV module declaration

```
1  // Toplevel module ultra_dsp
2  module ultra_dsp(
3     input                    clk,
4     input                    rst_n,
5     input [7:0]              x,
6     input [7:0]              a,
7     input [7:0]              b,
8     output logic [15:0]      dsp_out
9  );
10
11 logic [8:0] add_out;       //create internal signal add_out, 9bit wide
12
13 add_1 u_add_1(             //create a instance of add_1 called u_add_1
14         .in1    (a)           //connect input in1 to a
15         .in2    (b)           //connect input in2 to b
16         .res    (add_out)     //connect output res to add_out
17 );
18
19 mult_a u_mult_a(          //create an instance of mult_a called u_mult_a
20         .c      (clk),        //
21         .r      (rst_n),      //
22         .x,                   //if the name is the same automatic connection can
23         .y      (add_out),    //
24         .out    (dsp_out)     //
25 );
26
27 endmodule;
```

Arithmetic operators

Table: Arithmetic operators

Operator	Description
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder of an integer division

Synthesis of arithmetic operators

- The addition and subtraction are synthesizable.
- The multiplication is synthesizable for FPGAs and will be mapped to on-board multipliers.
- The division is not in general not synthesizable.

Comparison

Table: Comparison

Operator	Description
$a < b$	a smaller than b
$a > b$	a larger than b
$a <= b$	a smaller or equal b
$a >= b$	a larger or equal b
$a == b$	a has the same value as b
$a! = b$	a has a different value to b
$a === b$	a has the same value as b including X, Z
$a! == b$	a has a different value to b including X, Z

Logic vs. case equality

- The logic equalities $(==, ! =)$ is
  - 0 if the comparison is false
  - 1 if the comparison is true
  - X if at least 1 bit of the operands a,b is X or Z.
- The case equalities $(===, ! ==)$ are always either 0 or 1.

Logic operators

Table: Logic operators

Operator	Description
&&	logic and
||	logic or
!	Negation

Table: Bitwise logic operators

Operator	Description
&	Bitwise and
|	Bitwise or
~	Bitwise inversion
^	Bitwise xor
^ ~	Bitwise xnor

Reduction operators

Table: Reduction operators

Operator	Description
$\&x$	and operation an all bits in x
$\|x$	or operation on all bits in x
$\wedge x$	xor operation on all bits in x
$\sim \&x$	and all bits in x, then negate
$\sim \|x$	or all bits in x, then negate
$\sim \wedge x$	xor all bits in x, then negate

Listing 7: Reduction operators

```
1 logic [7:0] data;
2 logic all_one;
3 logic all_zero;
4
5 assign all_one = &data;
6 assign all_zero = ~|data;
```

Shift, Concatenate

Table: Shift and concatenation

Operator	Description
$a << b$	shift a by b bits to the left
$a >> b$	shift a by b bits to the right, shift in zeros $\rightarrow$ only correct for unsigned
$a <<< b$	shift a by b bits to the left
$a >>> b$	shift a by b bits to the right, shift in the MSB for a correct sign
a, b, c	Concatenate vectors a, b and c

Higher abstraction levels

- The operators shown so far allow the description of low level logic.
- To describe logic on a higher level of abstraction and to create test benches to check the correct behavior of a design more complex statements are required

Higher abstraction levels

- The operators shown so far allow the description of low level logic.
- To describe logic on a higher level of abstraction and to create test benches to check the correct behavior of a design more complex statements are required

initial

- All initial processes are started in parallel when simulation starts.
- All statements inside the initial are executed one after each other.
- # statements are used to wait for a specified time.
- Simulation only – not synthesizable.

Higher abstraction levels

- The operators shown so far allow the description of low level logic.
- To describe logic on a higher level of abstraction and to create test benches to check the correct behavior of a design more complex statements are required

initial

- All initial processes are started in parallel when simulation starts.
- All statements inside the initial are executed one after each other.
- # statements are used to wait for a specified time.
- Simulation only – not synthesizable.

always

- always statements are executed if one of the signals in the sensitivity list changes.
- Used to describe sequential or combinatorial logic:
  - always_comb → combinatorial logic (e.g. decoders, muxers)
  - always_ff → sequential logic (e.g. flipflop)

Combinatorial logic – always_comb

- Use the always_comb process to describe complex combinatorial logic.
- Control structures like if,case,for or while can be used.
- Output signals need to be defined for all conditions!

Listing 8: Combinatorial logic – always_comb

```
1  always_comb begin : majority_voter
2      y = 0;                          // default value
3      if (x == 3'b011) begin           // override default for 011
4          y = 1;
5      end
6      else if (x == 3'b101) begin  // override default for 101
7          y = 1;
8      end
9      else if (x == 3'b110) begin  // override default for 110
10         y = 1;
11     end
12     else if (x == 3'b111) begin  // override default for 111
13         y = 1;
14     end
15 end : majority_voter
```

Sequential logic – always_ff

- Use the always_ff process to describe sequential logic (FFs, counters)
- For good synthesis results defined templates shall be used.
- Clock and reset conditions are specified with @
- Only changes of the output signal needs to specified (else it stays at the stored value).

Listing 9: D-FF

```
1 always_ff @ (negedge rst_n or posedge clk10m) begin : d_ff_1
2     if (!rst_n) begin          // async reset (active low)
3         q <= 1'b0;             // reset value
4     end
5     else begin                 // synchronous behavior
6         q <= d;                // at rising edge of the clock, q follows d
7     end
8 end : d_ff_1
```

4bit counter – always_ff

Example: 4 bit up counter with enable and synchronous data load

- Async active low reset to all zeros.
- Counts up on the rising edge of the clock if enabled.
- data_in can be loaded using the load signal.

Listing 10: 4 bit up counter

```
1  always_ff @ (negedge rst_n or posedge clk10m) begin : 4bit_up_cnt
2      if (!rst_n) begin       // async reset (active low)
3          cnt <= '0;          // '0 is all zeros
4      end
5      else if (load) begin
6          cnt <= data_in;
7      end
8      else if (en) begin
9          cnt <= cnt + 4'd1;
10     end
11     // no else is required here -- in this case the cnt stays where it is
12 end : 4bit_up_cnt
```

Test pattern – initial

- In a test bench stimuli are sent the DUT.
- The timing of the test stimuli can be controlled using an initial process.
- ‘timescale defines the fundamental time step of the simulator
- # wait for the specified number of time steps or a time unit.
- @ waits for an event

Listing 11: Test stimuli for the 4bit counter

```
initial begin : test_stim
    rst_n = 1'b0;                   // assert rst_n
    en = 1'b0;                      // initial condition of en
    load = 1'b0;                    // initial condition of load
    data = 4'b0000;                 // initial condition of data

    # 100ns;                        // wait for 100ns
    rst_n = 1'b1;                   // --- release rst_n ---
    @ (negedge clk10m)              // wait for 2 negative edges of clk10m
    @ (negedge clk10m)
    en = 1'b1;                      // --- enable the counter ---
    @ (negedge clk10m)              // wait for 2 negative edges of clk10m
    @ (negedge clk10m)
    data = 4'b1111;                 // set data to all ones
    #1us;
    @ (negedge clk10m)              // wait for a negative edges of clk10m
    load = 1'b1;                    // --- assert load ---
    @ (negedge clk10m)              // wait for a negative edges of clk10m
    load = 1'b0;                    // --- clear load ---
```

Signal assignments

Without explanation three different ways of signal assignments were used so far:

- continuous assignment: assign $y = x$;
- blocking assignment: $y = x$;
- non-blocking assignment: $y <= x$;

assign $y = x$;

- Models a permanent connection (wire).
- To be used outside processes.
- Signal routing and simple logic.

$y = x$;

- For comb logic.
- Inside always_comb or initial processes.
- Executed one line after another (like software).

$y <= x$;

- For sequential logic.
- Inside always_ff processes.
- All $<=$ in a process are executed at the same time.

If statement

- C-style
- Curly brackets ({}) are replaced by begin end statements.
- Introduces a priority!

Listing 12: if statement

```
1  always_comb begin : if_example
2      if (sel == 2'b00) begin          // prio 1
3          y = x1;
4      end
5      else if (sel == 2'b01) begin      // prio 2
6          y = x2;
7      end
8      else if (sel == 2'b10) begin      // prio 3
9          y = x3;
10     end
11     else begin                        // prio 4
12         y = x4;
13     end
14 end : if_example
```

If statement – synthesized

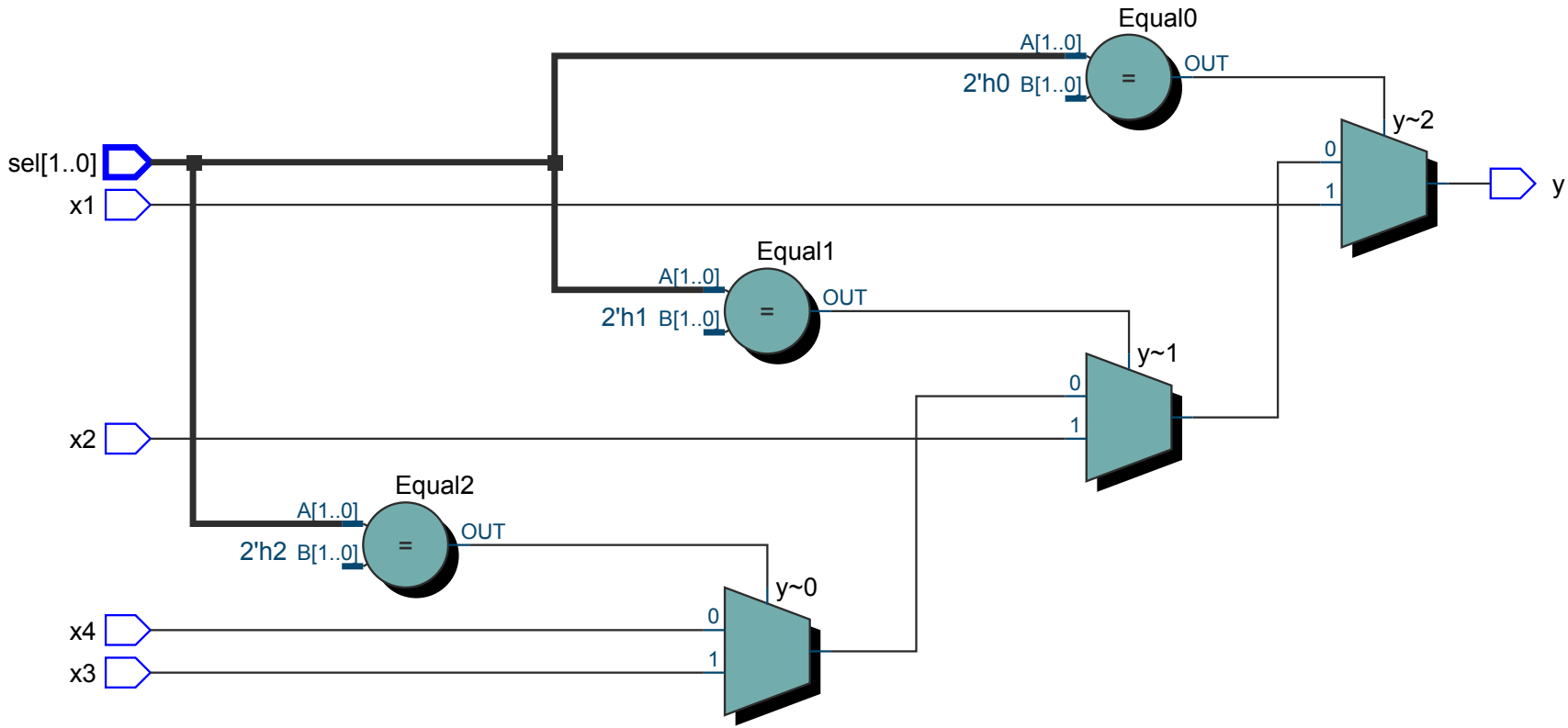


Figure: Synthesized if statement from listing 12

Source: own work created with Altera Quartus II.

Case statement

- Similiar to if, else but without priority.
- casez and casex can also use wildcards.

Listing 13: case statement

```
1  always_comb begin : case_example
2      case (sel)
3          2'b00 : begin
4              y = x1;
5          end
6          2'b01 : begin
7              y = x2;
8          end
9          2'b10 : begin
10             y = x3;
11         end
12         default : begin
13             y = x4;
14         end
15     endcase
16 end : case_example
```

Case statement – synthesized

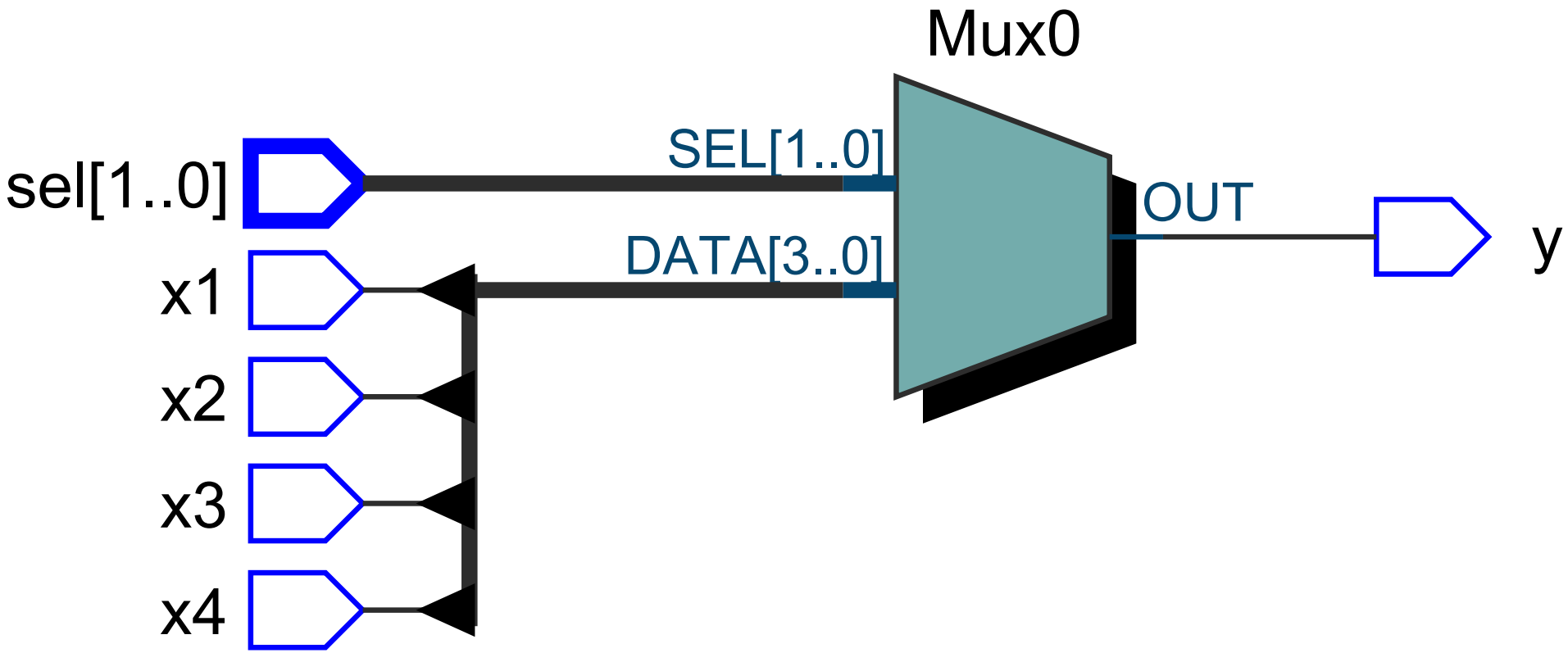


Figure: Synthesized if statement from listing 13

Source: own work created with Altera Quartus II.

For loop

- C-style
- Used primarily in test benches to generate stimuli
- In special applications also for synthesis.
- Special derivatives: repeat(n), forever

Listing 14: for loop to test the muxer

```
1 initial begin : for_example
2     sel = 2'b00;                  // initial value for sel.
3     #100ns;                       // wait for 100ns
4
5     for (int i = 0; i <= 3; i++) begin
6         sel = i;                  // assign the loop counter to sel
7         #500ns;                   // wait for 500ns;
8     end
9 end : for_example
```

While loop

- C-style
- Used primarily in test benches to generate stimuli

Listing 15: while loop

```
1  initial begin : while_example
2      sel = 2'b00;                    // initial value for sel.
3      #100ns;                         // wait for 100ns
4      integer i = 0;
5      while (i<4) begin
6          sel = i;                    // assign i to sel
7          i++;                        // increment i
8          #500ns;                     // wait for 500ns;
9      end
10 end : while_example
```

VHDL – Very High Speed Integrated Circuit Hardware Description Language

Basic elements

A brief introduction into VHDL showing the main building blocks of a design. Refer to the example design for more information.

- Comments:
  - – starts a line comment, end at the EOL
  - No block comment available
- Case insensitive
- Names need to start with a letter
- White spaces and tabs are not critical.

Listing 16: VHDL comment and names

```
1   -- This is a VHDL comment
2
3   small = SmaLL;  -- case insensitive -> but do not exploit that feature
4
5       y <= x      + 1;
6       y          <=x+1;
```

Data Types

- VHDL is a strongly typed language.
- There is no implicit type conversion like in C or SystemVerilog!

The following basic data types are available:

- boolean: (true, false)
- bit: (0,1)
- integer: range $-(2^{31}-1) : +(2^{31}-1)$
- real: single precision floating point
- character: any character. enclosed with ".
- time: integer with unit (fs, ps, ns, us, ms, sec, min of hr)

Listing 17: VHDL basic data types

```
1     signal X0, X1, X2, X3: bit;  -- four signals X0 to X3 of type bit
2     signal on_off, res: boolean;      -- two boolean signal
```

VHDL assignements

Once signals are declared a value can be assigned to them.

- Concurrent assignment → it's rather a wire than a variable!
- Similar to connecting a PCB track to a source!
- The <= operator is used to assign a value to a signal.

Listing 18: VHDL assignments

```
1     X0 <= '0';                    -- drive the signal X0 low
2     on_off <= true;               -- on_off is true
3
4     X1 <= (X0 or X2) and X3;      -- logic gates
5     res <= (X1=X2);       -- compare X1 to X2 and assign the result to res
```

VHDL basic operators

- Logic operators
  - not $\rightarrow$ Negation
  - and, or
  - nand, nor
  - xor, xnor
- Arithmetic and comparison
  - $+$, $-$ $\rightarrow$ Addition and subtraction are synthesizable
  - $*$, $/$ $\rightarrow$ Multiplication supported by most FPGA synthesis tools, Division not supported.
  - $=$, $/=$ $\rightarrow$ equal and unequal (synthesizable)
  - $<$, $<=$ $\rightarrow$ less than and less or equal (synthesizable)
  - $>$, $>=$ $\rightarrow$ greater and greater or equal (synthesizable)

Vectors

Signals of the same type can be bundled into vectors.

- Allows easier handling of multiple signals.
- Index is an integer.

Listing 19: VHDL vectors

```
signal x: bit_vector(0 to 7);     -- eight signals of type bit: x(0), x(1), x(2)
signal y: bit_vector(0 to 7);
signal z: bit_vector(0 to 7);

signal r: bit_vector(3 downto 0);  -- four signals of type bit: r(0), r(1), r(2)
signal s: bit_vector(3 downto 0);
signal t: bit_vector(3 downto 0);
```

Listing 20: VHDL vectors assignment

```
    x <= ('1','0','0','1','1','1','0','0');      -- assign 0x9A to x
    y <= x"9A";        -- same thing but in a more compact form
    z <= b"1001_1100"; -- underline can be used to structure

    r <= b"0011";    -- assign d3 to r
    s <= '1' & r(2 downto 0);    -- assign d11 to s
    t(1 downto 0) <= b"11";      -- assignment of parts of a vector is possible
    t(3 downto 2) <= b"00";
```

VHDL libraries

VHDL libraries are used to extend the functionality by defining types, functions and overloaded operators.
Two important libraries that are often used are

- IEEE.std_logic_1164
- IEEE.numeric_std

Listing 21: VHDL libraries

```
1 library ieee;
2 use ieee.std_logic_1164.all;     -- standard unresolved logic UX01ZWLH-
3 use ieee.numeric_std.all;        -- for the signed, unsigned types and arithmeti
```

Std_logic, std_logic_vector, numeric arithmetic

The IEEE 1164 standard describes a 9-valued logic system that can be used to model typical CMOS logic designs and allows for easier simulation.

- Std_logic and std_logic_vector shall be used to describe digital logic.

Table: Possible values of std_logic.

Character	**Value**
U	uninitialized, unknown logic value
X	strong drive, conflict 0 vs. 1
0	strong drive, logic zero
1	strong drive, logic one
Z	high impedance
W	weak drive, conflict L vs. H
L	weak drive (open source), logic zero
H	weak drive (open drain), logic one
-	don't care

STD LOGIC data types

For logic description the following possible values of std_logic are important:

- 0: Active drive to low.
- 1: Active drive to high.
- Z: high ohmic. Can be used to describe tristate buffers (if the target hardware supports it).
- -: don't care: Allows for logic optimization.

Numeric_std

The package numeric_std is used to perform integer arithmetic using vectors.

- Two numeric types are defined: unsigned (represents UNSIGNED numbers in vector form) and signed (represents a SIGNED number in vector form).
- The base element type is type STD_LOGIC.
- Vectors can be casted from std_logic_vector to unsigned/signed and vice versa.

The types defined in numeric_std shall be used to describe synthesizable integer arithmetic.

For an example see listing 22.

Listing 22: VHDL integer arithmetic

```
1  entity test_data_types is
2          port(
3                  x1, x2:                 in  std_logic_vector(7 downto 0);
4                  sum1:                   out std_logic_vector(8 downto 0)
5          );
6  end test_data_types;
7
8  architecture behavioral of test_data_types is
9          signal x1_u, x2_u       :       unsigned (7 downto 0);
10         signal sum1_u           :   unsigned (8 downto 0);
11
12         begin
13
14         -- Adding std_logic_vectors does not work!
15         -- sum1 <= x1 + x2;
16
17         -- So we need to convert x1 and x2 into unsigned
18         x1_u <= unsigned(x1);   -- type cast
19         x2_u <= unsigned(x2);
20
21         sum1_u <= ('0' & x1_u) + ('0' & x2_u); -- numeric vectors need to be sa
22
23         sum1 <= std_logic_vector(sum1_u);
24 end behavioral;
```

Modul declaration

Entities are VHDL blocks with a defined interface and internal logic. The logic is defined by at least one architecture.

- The module that connects to the pins of the IC is the toplevel.
- Modules can be instantiated as submodules. Thereby a hierarchical description of the whole IC is possible.

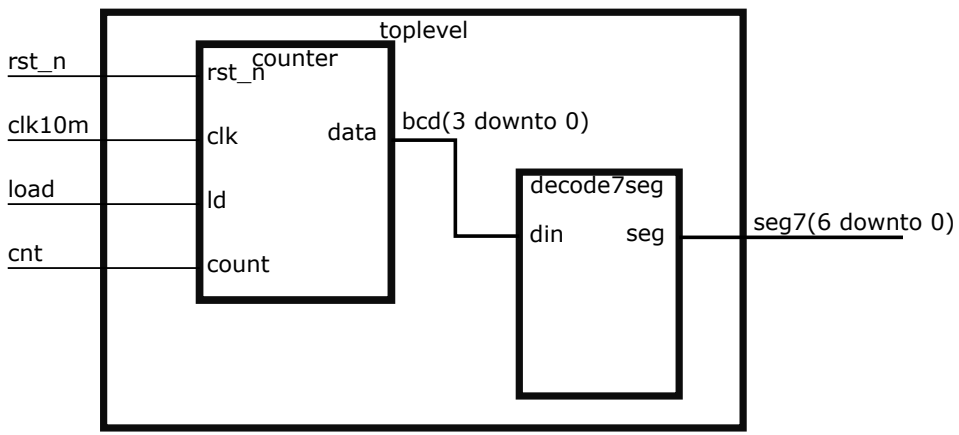


Figure: Toplevel containing two submodules(counter and decode7seg).

Entity with sub-entities

Listing 23: VHDL entities and sub-entities

```
1  entity toplevel is
2          port(
3                  rst_n    :        in       std_logic;
4                  clk10m   :        in       std_logic;
5                  load     :        in       std_logic;
6                  cnt              :        in       std_logic;
7                  seg7     :        out      std_logic_vector (6 downto 0)
8          );
9  end toplevel;
10
11 architecture hierarchical of toplevel is
12         -- declare internal signals
13         signal bcd       :        std_logic_vector (3 downto 0);
14 begin
15         -- instantiate sub-entities
16         u1_counter       :        entity counter(behavorial)
17         port map(
18                 -- sub_name     =>       toplevel_name
19                 rst_n          =>       rst_n,
20                 clk                     =>       clk10m,
21                 ld                      =>       load,
22                 count          =>       cnt,
23                 data           =>       bcd
24         );
25
26         u1_decode7seg    :        entity decode7seg(behavorial)
27         port map(
```

VHDL Sequential environments

All VHDL signal assignments so far were concurrent.
To allow for a behavioral (more abstract) description there exists the process environment.
In a process:

- All statements are treated sequentially.
- The sensitivity list needs to contain all signals that can trigger a process.
- When the last line of a process is reached it immediately starts at the beginning.
- Inside a process if/else, case, for,... structures can be used to describe the behavior of the logic.

VHDL Sequential environments

All VHDL signal assignments so far were concurrent.
To allow for a behavioral (more abstract) description there exists the process environment.
In a process:

- All statements are treated sequentially.
- The sensitivity list needs to contain all signals that can trigger a process.
- When the last line of a process is reached it immediately starts at the beginning.
- Inside a process if/else, case, for,... structures can be used to describe the behavior of the logic.

Processes can be used to describe combinatorial and sequential logic.

- If the logic shall be combinatorial $\rightarrow$ make sure that the output signal is assigned for all cases (e.g. all branches of an IF/ELSIF/ELSE statement).

VHDL Sequential environments

All VHDL signal assignments so far were concurrent.
To allow for a behavioral (more abstract) description there exists the process environment.
In a process:

- All statements are treated sequentially.
- The sensitivity list needs to contain all signals that can trigger a process.
- When the last line of a process is reached it immediately starts at the beginning.
- Inside a process if/else, case, for,... structures can be used to describe the behavior of the logic.

Processes can be used to describe combinatorial and sequential logic.

- If the logic shall be combinatorial $\rightarrow$ make sure that the output signal is assigned for all cases (e.g. all branches of an IF/ELSIF/ELSE statement).

Processes are furthermore used to drive the test stimuli in a test bench in a certain sequence.

VHDL process – update of signals

To understand sequential logic it is important when signals are updated:

- In a process all signals are updated at the next clock cycle.

For example a 4-bit shift register:

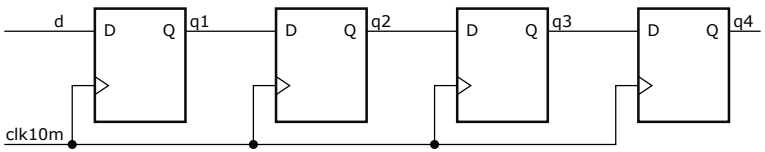


Figure: 4bit shift register block diagram.

Listing 24: VHDL 4 bit shift register

```
1
2     process (clk10m)
3     begin
4         if (clk10m = '1') and clk10m'event then
5             q1 <= d;
6             q2 <= q1;
7             q3 <= q2;
8             q4 <= q3;
9         end if;
10
11    end process;
```

- q1 is updated to the value of d just before the clock event.

VHDL process – update of variables

To allow for values that are immediately updates variables can be used.

- In a process all variables are updated immediately.

For example a muxer:

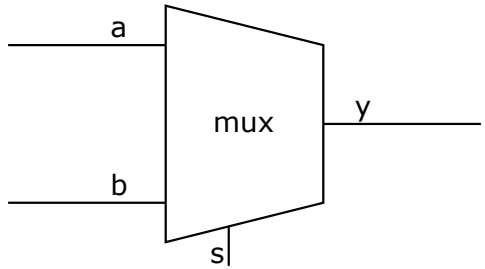


Figure: 2 bit multiplexer.

Listing 25: VHDL 2 bit muxer

```
1     process (s)
2         variable y_1 : std_logic;    -- declaration of y as local variable
3     begin
4         y_1 := a;                    -- default assignement (makes sure that the
5         if (s = '1') then            -- override the previous assignment
6             y_1 := b;
7         end if;
8
9         y <= y_1;                    -- assign y_1 to the output signal y
10
11    end process;
```

Concurrent assignments

VHDL offers behavorial descriptions also in concurrent statements:

- with / select
- when / else

Listing 26: VHDL 2 bit muxer concurrent

```
1
2   -- no need for a process (it's concurrent)
3
4   signal y,z: std_logic
5
6   with s select                -- use s as selector
7       y <= a when '0',         -- assign a if s = '0'
8       y <= b when '1',         -- assign b if s = '1'
9       y <= 'x' when others;    -- default assignment (s could be L, H, Z,...)
10
11  z <= a when (s='0') else b; -- alternative description of a muxer
```

Sequential assignments in a process

In a process many constructs known from software can be used to describe logic:

- if/elsif/else
- case/is
- loops (for, while)

The same multiplexer as above now in a process environment:

Listing 27: VHDL 2 bit muxer sequential

```
1   -- concurrent with process and if/else
2       process (s,a,b)
3       begin
4               if (s='0') then
5                       y3 <= a;
6               else
7                       y3 <= b;
8               end if;
9       end process;
10
11      --concurrent with process and case/is
12      process (s,a,b)
13      begin
14              case (s) is
15                      when '0'          => y4 <= a;
16                      when '1'          => y4 <= b;
17                      when others => y4 <= 'U'; -- is it needed?
18              end case;
19      end process;
```

VHDL Test benches

Test benches are used to stimulate a DUT (device under test) and to check that the outputs behave correctly (verification).

- Stimuli are driven concurrently (e.g. clock) or from processes (e.g. sequential test patterns).
- The wait statement is used to advance the the simulation time.
  - wait until condition; → wait until clk = '1'; waits for rising edge
  - wait for time; → wait for 100 ns;
  - wait; → suspends a process indefinitely

VHDL Test benches (2)

Listing 28: VHDL TB with time control

```
1  -- Clock generator can be done in a concurrent statement with delay
2          clk50m <= not clk50m after 10 ns when (run_sim /= '0') else '0';
3
4          -- test pattern to stimulate the inputs
5          test_stimuli : process is
6          begin
7                  action  <= "Init     ";
8                  rst_n   <= '0';
9                  load    <= '0';
10                 preload <= x"00";
11                 wait for 99 ns;
12
13                 action  <= "POR      ";
14                 rst_n   <= '1';
15
16                 -- Let the counter count
17                 wait for 10 us;
18
19                 -- Load with 0xFF
20                 action  <= "load FF  ";
21                 preload <= x"FF";
22                 wait until clk50m'event and clk50m='1';
23                 load    <= '1';
24                 wait until clk50m'event and clk50m='1';
25                 load    <= '0';
26                 wait for 10 us;
27
```


ROTH, Charles H. ; JOHN, Lizy K.:
Digital systems design using VHDL.
2. ed.
Toronto (u.a.) : Thomson, 2008. –
ISBN 978–0–534–38462–3