

Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
(csdsp@bristol.ac.uk)

January 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
 - ▶ the associated notes page may be pre-populated with extra, written explanation of material covered in lecture(s), plus
 - ▶ anything with a “grey’ed out” header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

► **Agenda:**

1. comments, questions, recap, then
2. a brief introduction to **Hardware Description Languages (HDLs)**.

Notes:

HDLs in concept (1)

Definition

In contrast to a conventional programming language which are (typically) used to describe software, a **Hardware Description Language (HDL)** is used to describe (or model) hardware (e.g., digital logic).

► (Selected) **examples:**

1. **Verilog**
2. **VHDL**
3. **MyHDL** ⊃ **Python**
4. **Chisel** ⊃ **Scala**

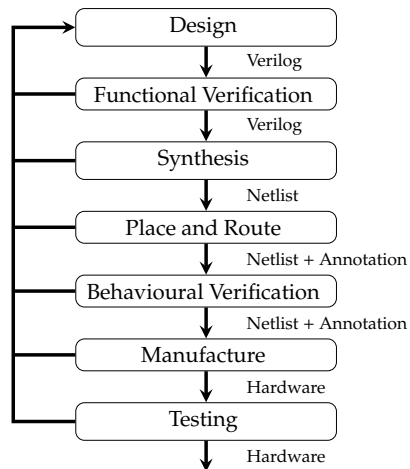
Notes:

HDLs in concept (2)

- ▶ **Question:** why bother?!
- ▶ **Answer:** *similar* reasoning to why you'd select C rather than assembly language, e.g.,
 - +ve: HDLs (and EDA tools more generally) allow design and manufacture of larger, more complex components: they
 - ▶ raise the level of abstraction (vs. direct use of transistors or logic gates), and so
 - ▶ support a focus on higher-level behaviour rather than low-level implementation.
 - +ve: Using EDA tools allows automation wrt.
 - ▶ simulation,
 - ▶ verification, and
 - ▶ translationof our now machine-readable HDL model.
 - ve: The potential trade-off is less control: by delegating work to EDA tools, we have less (direct) control over the result.

Notes:

HDLs in concept (3)



- ▶ Use of a HDL is typically an iterated process.
 - ▶ You can think of
 - synthesis \simeq compilation
 - place and route \simeq linking
- since
- ▶ the former translates from high- to low-level, in this case a HDL model to a gate-level netlist,
 - ▶ the latter works out how to use the standard cell library (e.g., the type and location of gates).
- ▶ Verification steps rely on simulation of the model at different levels of detail.

Notes:

- ▶ The main concepts in Verilog can be described by analogy

Definition (C)	Definition (Verilog)
<ol style="list-style-type: none"> 1. A program is described using static function definitions. 2. Each function has an interface (i.e., what it does and how it can be used) and a body (i.e., how it does it). 3. The functions reference each other via calls; a function call implies an active, <i>transient</i> use. 4. Values are stored in variables, on which computation is performed by functions. 	<ol style="list-style-type: none"> 1. A model is described using static module definitions. 2. Each module has an interface (i.e., what it does and how it can be used) and a body (i.e., how it does it). 3. The modules reference each other via instantiations; a module instantiation implies an active, <i>permanent</i> use. 4. Values are carried by nets, on which computation is performed by modules.

but **beware**:

- ▶ on one hand, the analogy is attractive if you have some C programming experience, *but*
- ▶ on the other hand, the analogy is unattractive (perhaps even *dangerous*) because it's *imperfect* in some ways.

Notes:

- It doesn't matter a lot, but it might be useful to note that we'll focus on the IEEE 1364-2001 version of Verilog.

HDLs in concept (5)

Verilog as a case-study

- ▶ Fundamentally, Verilog allows description of components (or **modules**) and the connections between them ...
- ▶ **Example**:

```
module fa( output wire co,
          output wire s,
          input wire ci,
          input wire x,
          input wire y );

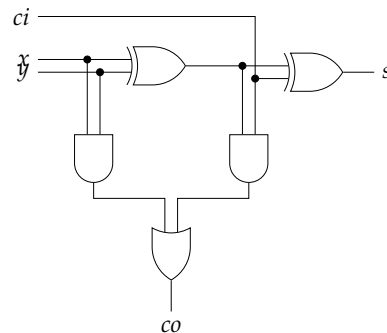
  wire w0, w1, w2;

  xor t0( w0, x, y );
  and t1( w1, x, y );

  xor t2( s, w0, ci );
  and t3( w2, w0, ci );

  or t4( co, w1, w2 );

endmodule
```



Notes:

- Fundamentally, Verilog allows description of components (or **modules**) and the connections between them ...
- **Example:**

```
module fa( output wire co,
           output wire s,
           input wire ci,
           input wire x,
           input wire y );

    wire w0, w1, w2;

    xor t0( w0, x, y );
    and t1( w1, x, y );

    xor t2( s, w0, ci );
    and t3( w2, w0, ci );

    or t4( co, w1, w2 );

endmodule
```



1. The component has
 - an **interface** describing how other components should interact with it (e.g., which inputs and outputs are available), and
 - an **implementation** describing what it does (e.g., how *s* is computed)and **instanciates** internal (sub-)components.
2. Connections are described using **wires**:
 - **internal** wires relate to the implementation, while
 - **external** wires relate to the interface.

Notes:

- ... it does so using
 1. high-level, behaviour-oriented style, or
 2. low-level, implementation-oriented style, or
 3. a *mix* of the two

for a given module:

Example (option #1: switch-level Verilog)

At the lowest-level, the model can be described using individual transistors. For example, the four transistor instances

```
pmos( t, VDD, b );
pmos( a, t, c );
nmos( a, VSS, c );
nmos( a, VSS, b );
```

replicate the previous circuit for a MOSFET-based NOR gate, meaning they continuously drive the wire *a* with the the result of evaluating $\neg(b \vee c)$.

Notes:

► ... it does so using

1. high-level, behaviour-oriented style, or
2. low-level, implementation-oriented style, or
3. a *mix* of the two

for a given module:

Example (option #2: gate-level Verilog)

Forces the model to be described at a low-level, using only primitive logic gates (e.g., AND, OR, NOT). For example, the gate instantiation

```
nor t( a, b, c );
```

continuously drives the wire a with the the result of evaluating $\neg(b \vee c)$.

Notes:

► ... it does so using

1. high-level, behaviour-oriented style, or
2. low-level, implementation-oriented style, or
3. a *mix* of the two

for a given module:

Example (option #3: Register Transfer Level (RTL) Verilog)

Uses a syntax similar to C, but focuses on describing the model in terms of the data-flow between components rather than high-level statements. For example, the continuous assignment

```
assign a = ~( b | c )
```

continuously drives the wire a with the the result of evaluating $\neg(b \vee c)$.

Notes:

► ... it does so using

1. high-level, behaviour-oriented style, or
2. low-level, implementation-oriented style, or
3. a *mix* of the two

for a given module:

Example (option #4: behavioural-level Verilog)

Allows a high-level, C-style description of the model using assignments, loops and conditional statements. For example, the procedural assignment

$$a = \sim(b \mid c)$$

sets the register a equal to the result of evaluating $\neg(b \vee c)$.

Notes:

Demo

Notes:

- ▶ A **wire** (resp. **wire vector**) is a connection, or **net**, used to communicate values; their definition demands

1. a type, and
2. an identifier.

- ▶ **Example(s)**:

- `wire w` \Rightarrow an internal 1-bit wire `w`
- `wire [3 : 0] x` \Rightarrow an internal 4-bit wire vector `x`
- `input wire [3 : 0] y` \Rightarrow an input 4-bit wire vector `y`
- `output wire [3 : 0] z` \Rightarrow an output 4-bit wire vector `y`

Notes:

- ▶ **Beware**: neither a wire nor wire vector can remember state (e.g., doesn't behave like a C variable); we need to *drive* a value on it.

Notes:

Example (C)

► The definition

`char u` \leadsto 8 *separate* 1-bit elements

but `u` is typically used as 1 *single* 8-bit object.

► The definition

`char v[32]` \leadsto 32 *separate* 8-bit elements

and `v` is typically used as 32 *separate* 8-bit elements.

Example (Verilog)

► The definition

`wire x` \leadsto 1 *single* 1-bit wire

and `u` is used as 1 *single* 1-bit object.

► The definition

`wire [3 : 0] y` \leadsto 4 *separate* 1-bit wires

st.

1. `y` can be used as 1 *single* 4-bit object, or
2. `y` can be used as 4 *separate* 1-bit wires.

Notes:

► Any given wire (resp. wire vector) can communicate values that

1. support the concept of 3-state logic, e.g.,

- `0` \Rightarrow 0 (i.e., logical **false**)
- `1` \Rightarrow 1 (i.e., logical **true**)
- `X` \Rightarrow unknown (i.e., neither 1 or 0)
- `Z` \Rightarrow high impedance (i.e., disconnected)

2. can be written in binary, decimal, or hexadecimal, e.g.,

- `2'b10` \Rightarrow a 2-bit binary literal, with value $10_{(2)}$, $2_{(10)}$, or $2_{(16)}$
- `8'd17` \Rightarrow a 8-bit decimal literal, with value $00010001_{(2)}$, $17_{(10)}$, or $11_{(16)}$
- `4'hF` \Rightarrow a 4-bit hexadecimal literal, with value $1111_{(2)}$, $15_{(10)}$, or $F_{(16)}$

3. can include 3-state values on a per-bit basis, e.g.,

- `1'bX` \Rightarrow a 1-bit binary literal, st. the bit is unknown
- `4'b10XZ` \Rightarrow a 4-bit binary literal, st. the bits are high impedance, unknown, 0, and 1

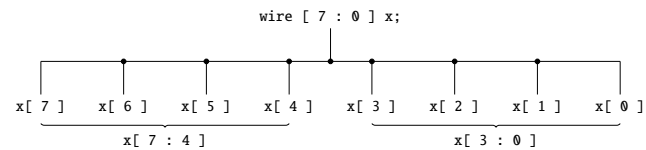
Notes:

- **Beware:** if you omit the size or base, a literal might not mean what you expect.

Notes:

Example (subscript operator)

Imagine $x = 8'b11110000$. As a result of



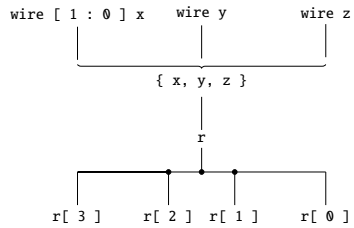
we have that

- $x[7]$, $x[6]$, $x[5]$ and $x[4]$ are all 1-bit wires with value $1'b1$,
- $x[3]$, $x[2]$, $x[1]$ and $x[0]$, are all 1-bit wires with value $1'b0$,
- $x[7 : 4]$ is a 4-bit wire vector with value $4'b1111$, and
- $x[3 : 0]$ is a 4-bit wire vector with value $4'b0000$.

Notes:

Example (concatenate operator)

Imagine $x = 2'b10$, $y = 1'b1$ and $z = 1'b0$. As a result of



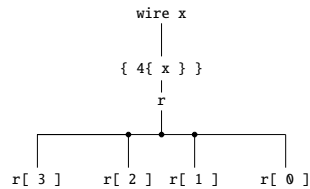
we have that

- ▶ $\{ x, y, z \}$ is a 4-bit wire vector with value $4'b1010$,
- ▶ $r[3]$ is a 1-bit wire with value $1'b1$ (matching $x[1]$),
- ▶ $r[2]$ is a 1-bit wire with value $1'b0$ (matching $x[0]$),
- ▶ $r[1]$ is a 1-bit wire with value $1'b1$ (matching y), and
- ▶ $r[0]$ is a 1-bit wire with value $1'b0$ (matching z).

Notes:

Example (replicate operator)

Imagine $x = 1'b1$. As a result of



we have that

- ▶ $\{ 4\{ x \} \}$ is a 4-bit wire vector with value $4'b1111$,
- ▶ $r[3]$ is a 1-bit wire with value $1'b1$,
- ▶ $r[2]$ is a 1-bit wire with value $1'b1$,
- ▶ $r[1]$ is a 1-bit wire with value $1'b1$, and
- ▶ $r[0]$ is a 1-bit wire with value $1'b1$.

Notes:

- ▶ A **module definition** is roughly analogous to a C function definition; it demands

1. an identifier,
2. a port list, of internal input and output ports, and
3. a body

e.g.,

Listing (Verilog)	Listing (Verilog)
<pre>1 module mux2_1bit(output wire r, 2 input wire c, 3 input wire x, 4 input wire y); 5 6 ... 7 8 endmodule</pre>	<pre>1 module mux2_1bit(r, c, x, y); 2 3 output wire r; 4 input wire c; 5 input wire x; 6 input wire y; 7 8 ... 9 10 endmodule</pre>

noting the two forms are equivalent, where

1. the left-hand option specifies the port list “inline”, and
2. the right-hand option specifies the port list “inside”.

Notes:

- ▶ A **module instantiation** is roughly analogous to a C function call; it demands

1. an identifier,
2. an argument list, of external input and output wires, and
3. a type.

- ▶ Keep in mind that

module definition \Rightarrow passive (or static) description

module instance \Rightarrow active (or dynamic) component

so we get one component *per instantiation* of the (single) description ...

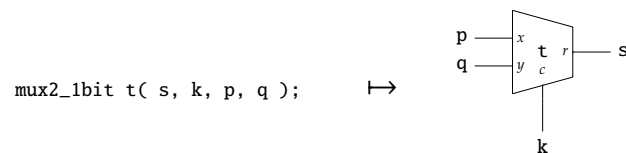
Notes:

- **Beware:** the analogy with C isn't ideal for (at least) reasons, i.e.,

Definition (C)	Definition (Verilog)
<ol style="list-style-type: none"> 1. Each “callee” function is represented by a single, <i>shared</i> instruction sequence; this can be used by many different “callers”. 2. The statements within a function are executed in <i>sequence</i>; it matters if one is before another. 	<ol style="list-style-type: none"> 1. Each module instantiation implies a <i>separate</i> instance of the circuit it is describing, i.e., separate physical hardware. 2. Each module instance is working in <i>parallel</i> with others; order does not matter (as much).

Notes:

- ... the idea is we create a module instance, and connect the internal ports of that instance to some external wires:



- In other words, we've
 1. created an instance of the `mux2_1bit` module and called the instance `t`, and
 2. connected the internal ports `r`, `c`, `x` and `y` to the external wires `s`, `k`, `p` and `q`
 meaning
 1. any input we drive onto `k`, `t` can “see” on `c`, and
 2. any output `t` drives onto `r`, we can “see” on `s`.

Notes:

- **Concept:** **gate-level** implementation describes module behaviour via
 1. primitive (or built-in) modules, and/or
 2. other user-defined modules.
- The primitive modules represent standard logic gates, e.g.,

buf t0(r, x);	\mapsto	$r = x$
not t1(r, x);	\mapsto	$r = \neg x$
nand t2(r, x, y);	\mapsto	$r = x \overline{\wedge} y$
nor t3(r, x, y);	\mapsto	$r = x \overline{\vee} y$
and t4(r, x, y);	\mapsto	$r = x \wedge y$
or t5(r, x, y);	\mapsto	$r = x \vee y$
xor t6(r, x, y);	\mapsto	$r = x \oplus y$

noting that versions with more inputs are automatically available, e.g.,

xor t8(r, w, x, y);	\mapsto	$r = w \oplus x \oplus y$
xor t9(r, w, x, y, z);	\mapsto	$r = w \oplus x \oplus y \oplus z$

plus ...

- ... **User-Defined Primitives (UDPs)** can further supplement those available: each UDP acts as a specification for a Boolean function of the form

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

via a *truth table* (vs. a combination of logic gates), e.g.,

Listing (Verilog)

```
1 primitive mux2_lbit( output r,
2                     input  c,
3                     input  x,
4                     input  y );
5
6     table
7         0 0 ? : 0;
8         0 1 ? : 1;
9         1 ? 0 : 0;
10        1 ? 1 : 1;
11    endtable
12 endprimitive
```

Truth table

MUX2			
c	x	y	r
0	0	?	0
0	1	?	1
1	?	0	0
1	?	1	1

which can then be used per a user-defined module.

Notes:

Notes:

- In terms the UDP, there can be as many inputs as you like, there can be one output (which is *first* in the port list); both inputs and outputs *must* be 1-bit wires (i.e., wire vectors are not allowed).
- The ? symbol denotes don't care, meaning the first line can be read as "when c and x are both 0 and y is any value, set r to 0".

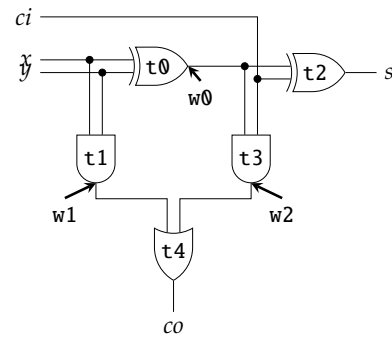
Listing (Verilog)

```

1 module fa( output wire co,
2           output wire s,
3           input wire ci,
4           input wire x,
5           input wire y );
6
7   wire w0, w1, w2;
8
9   xor t0( w0, x, y );
10  and t1( w1, x, y );
11
12  xor t2( s, w0, ci );
13  and t3( w2, w0, ci );
14
15  or t4( co, w1, w2 );
16
17 endmodule

```

Circuit (full-adder)



Notes:

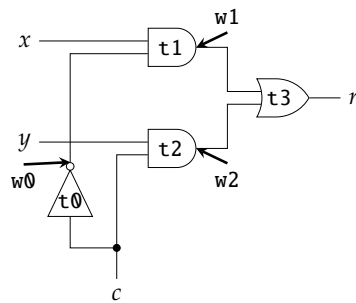
Listing (Verilog)

```

1 module mux2_1bit( output wire r,
2                  input wire c,
3                  input wire x,
4                  input wire y );
5
6   wire w0, w1, w2;
7
8   not t0( w0, c );
9
10  and t1( w1, x, w0 );
11  and t2( w2, y, c );
12
13  or t3( r, w1, w2 );
14
15 endmodule

```

Circuit (2-input, 1-bit multiplexer)



Notes:

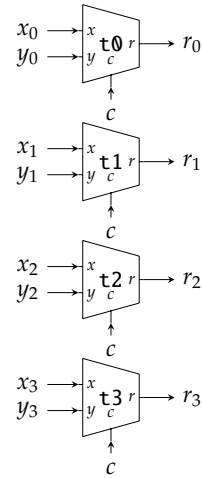
Listing (Verilog)

```

1 module mux2_4bit( output wire [ 3 : 0 ] r,
2                   input wire    c,
3                   input wire [ 3 : 0 ] x,
4                   input wire [ 3 : 0 ] y );
5
6   mux2_1bit t0( r[ 0 ], c, x[ 0 ], y[ 0 ] );
7   mux2_1bit t1( r[ 1 ], c, x[ 1 ], y[ 1 ] );
8   mux2_1bit t2( r[ 2 ], c, x[ 2 ], y[ 2 ] );
9   mux2_1bit t3( r[ 3 ], c, x[ 3 ], y[ 3 ] );
10
11 endmodule

```

Circuit (2-input, 4-bit multiplexer)



Notes:

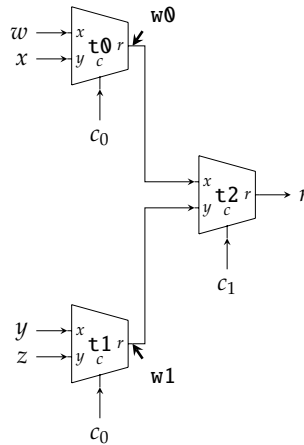
Listing (Verilog)

```

1 module mux4_1bit( output wire  r,
2                   input wire  c0,
3                   input wire  c1,
4                   input wire  w,
5                   input wire  x,
6                   input wire  y,
7                   input wire  z );
8
9   wire w0, w1;
10
11   mux2_1bit t0( w0, c0, w, x );
12   mux2_1bit t1( w1, c0, y, z );
13   mux2_1bit t2( r, c1, w0, w1 );
14
15 endmodule

```

Circuit (4-input, 1-bit multiplexer)



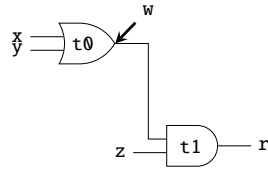
Notes:

► **Concept:** RTL implementation describes module behaviour via

1. a set of **continuous assignments**, plus
2. any additional gate-level description

e.g.,

```
assign r = ( x | y ) & z;  ⇨  or t0( w, x, y );  
                           and t1( r, w, z );  ⇨
```



► Note that

1. the LHS *must* be a wire or wire vector, whereas
2. the RHS can contain many C-style operators

- **arithmetic operators**, e.g., +, -, and *,
- **logical operators**, e.g., <<, >>, ~, &, |, and ^,
- **comparison operators**, e.g., ==, >, and <.

involving wires or wire vectors as operands

Notes:

Notes:

► **Beware:** it's tempting to think of this as analogous to a C assignment: this is dangerous, because the RTL version is *continuous*.

- **Example:** if we have wire [3 : 0] x, wire [3 : 0] y, and wire c, then
1. A **reduction** operator is a short-hand for combining the wires within a wire vector; for example, the expression
$$\wedge x$$
is the same as
$$((x[3] \wedge x[2]) \wedge x[1]) \wedge x[0]$$
and hence similar to **reduce** (or **foldr**) in Haskell.
 2. A **ternary** (or “choice”) operator acts in *exactly* the same way as a multiplexer; the expression
$$c \text{ ? } y : x$$
evaluates to
 - 2.1 x if c evaluates to 0, or
 - 2.2 y if c evaluates to 1.

Notes:

Listing (Verilog)

```
1 module fa( output wire co,
2           output wire s,
3           input wire ci,
4           input wire x,
5           input wire y );
6
7   wire w0, w1, w2;
8
9   xor t0( w0, x, y );
10  and t1( w1, x, y );
11
12  xor t2( s, w0, ci );
13  and t3( w2, w0, ci );
14
15  or t4( co, w1, w2 );
16
17 endmodule
```

Listing (Verilog)

```
1 module fa( output wire co,
2           output wire s,
3           input wire ci,
4           input wire x,
5           input wire y );
6
7   wire [ 1 : 0 ] t;
8
9   assign t = ci + x + y;
10
11   assign s = t[ 0 ];
12   assign co = t[ 1 ];
13
14 endmodule
```

Notes:

Listing (Verilog)

```
1 module fa( output wire co,
2           output wire s,
3           input wire ci,
4           input wire x,
5           input wire y );
6
7   wire w0, w1, w2;
8
9   xor t0( w0, x, y );
10  and t1( w1, x, y );
11
12  xor t2( s, w0, ci );
13  and t3( w2, w0, ci );
14
15  or t4( co, w1, w2 );
16
17 endmodule
```

Listing (Verilog)

```
1 module fa( output wire co,
2           output wire s,
3           input wire ci,
4           input wire x,
5           input wire y );
6
7   assign { co, s } = ci + x + y;
8
9 endmodule
```

Notes:

Listing (Verilog)

```
1 module mux2_1bit( output wire r,
2                  input wire c,
3                  input wire x,
4                  input wire y );
5
6   wire w0, w1, w2;
7
8   not t0( w0, c );
9
10  and t1( w1, x, w0 );
11  and t2( w2, y, c );
12
13  or t3( r, w1, w2 );
14
15 endmodule
```

Listing (Verilog)

```
1 module mux2_1bit( output wire r,
2                  input wire c,
3                  input wire x,
4                  input wire y );
5
6   assign r = c ? y : x;
7
8 endmodule
```

Notes:

Listing (Verilog)

```
1 module mux2_4bit( output wire [ 3 : 0 ] r,  
2                 input wire    c,  
3                 input wire [ 3 : 0 ] x,  
4                 input wire [ 3 : 0 ] y );  
5  
6   mux2_1bit t0( r[ 0 ], c, x[ 0 ], y[ 0 ] );  
7   mux2_1bit t1( r[ 1 ], c, x[ 1 ], y[ 1 ] );  
8   mux2_1bit t2( r[ 2 ], c, x[ 2 ], y[ 2 ] );  
9   mux2_1bit t3( r[ 3 ], c, x[ 3 ], y[ 3 ] );  
10  
11 endmodule
```

Listing (Verilog)

```
1 module mux2_4bit( output wire [ 3 : 0 ] r,  
2                 input wire    c,  
3                 input wire [ 3 : 0 ] x,  
4                 input wire [ 3 : 0 ] y );  
5  
6   assign r = c ? y : x;  
7  
8 endmodule
```

Notes:

Listing (Verilog)

```
1 module mux4_1bit( output wire  r,  
2                 input wire c0,  
3                 input wire c1,  
4                 input wire w,  
5                 input wire x,  
6                 input wire y,  
7                 input wire z );  
8  
9   wire w0, w1, w2, w3, w4, w5;  
10  not t0( w0, c0 );  
11  not t1( w1, c1 );  
12  
13  and t2( w2, w0, w1, w );  
14  and t3( w3, c0, w1, x );  
15  and t4( w4, w0, c1, y );  
16  and t5( w5, c0, c1, z );  
17  
18  or  t6(  r, w2, w3, w4, w5 );  
19  
20  
21 endmodule
```

Listing (Verilog)

```
1 module mux4_1bit( output wire  r,  
2                 input wire c0,  
3                 input wire c1,  
4                 input wire w,  
5                 input wire x,  
6                 input wire y,  
7                 input wire z );  
8  
9   assign r = c1 ? ( c0 ? z : y ) :  
10              ( c0 ? x : w );  
11  
12 endmodule
```

Notes:

- ▶ A **register** (resp. **register vector**) is a net that retains *state*; their definition demands
 1. a type, and
 2. an identifier.

- ▶ **Example(s):**

- `reg w` \Rightarrow an 1-bit register `w`
- `reg [3 : 0] x` \Rightarrow an 4-bit register vector `x`

where you could think of either as (more) analogous to a C variable, and implemented via a latch or flip-flop.

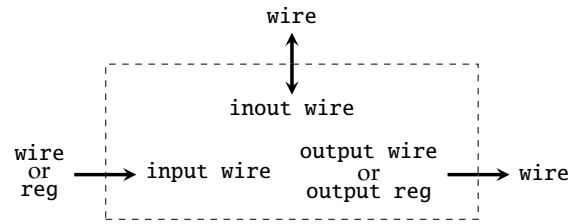
Notes:

- ▶ **Beware:**
 - ▶ a register can act as an input (i.e., be read) like a wire, *but*
 - ▶ it can't act as an output (i.e., be written to, or driven), in quite the same way.

Notes:

- Verilog enforces some **module interfacing rules**, namely

Definition



which are (somewhat) intuitive:

- externally an input port can either be a *wire or reg* *but* internally we have to be pessimistic and assume it is a *wire*,
- internally an output port can either be a *reg or wire* *but* externally we have to be pessimistic and assume it is a *wire*,
- violating the rules is analogous to a C type error.

- Concept:** **behavioural-level** implementation describes module behaviour by

- (at least partly) using behavioural **processes**, each of which
 - can be triggered to perform computation, and
 - operates in parallel with the others (much like a module instance does),
 then
- describing each process using some **blocks** of behavioural **statements**,
- each of which is “executed” when the containing process is triggered.

Notes:

- There is an extra port type in this diagram: an *inout* port is basically to allow input *and* output, i.e., bi-directional rather than uni-directional communication.

Notes:

- There is an implicit dependence on time and therefore state: if statements are executed as steps in sequence, we need to “remember” values between steps.
- A statement (or block) *cannot* exist outside a process, and a process *cannot* exist outside a module; this would be analogous to writing an if statement outside any function in C for example.
- We *can* still mix styles, so it is okay to describe the behaviour of a module partly in RTL and partly as a behavioural process for example.
- Each behavioural process (or block) is composed *only* of behavioural statements: you can’t place other things in them, with examples including module instantiations (in an attempt to “call” a module like C function).

- ▶ There are two types of process (the `id` is optional)

Listing (Verilog)
<pre>1 always begin:id 2 ... 3 end</pre>

Listing (Verilog)
<pre>1 initial begin:id 2 ... 3 end</pre>

where you can think of

- ▶ an `always` process as begin executed in a *loop* as long as the module is powered, *and*
- ▶ an `initial` process as being executed only *once* when the module is first powered up.

Notes:

- ▶ **Beware:** this style of process (where this is no information about timing) is bad practise ...

Notes:

► ... we should *trigger* processes via a **sensitivity list**:

- @(x) ⇒ triggers when x changes
- @(posedge x) ⇒ triggers when x changes from 0 to 1 (a positive edge)
- @(negedge x) ⇒ triggers when x changes from 1 to 0 (a negative edge)

► **Example(s)**: using a trigger we could write

Listing (Verilog)	Listing (Verilog)	Listing (Verilog)
<pre>1 always @ (x) begin 2 ... 3 end</pre>	<pre>1 always @ (posedge x) begin 2 ... 3 end</pre>	<pre>1 always @ (negedge x) begin 2 ... 3 end</pre>

and have

1. the process content executed when x changes from or to anything,
2. the process content executed when x changes from 0 to 1, or
3. the process content executed when x changes from 1 to 0.

Notes:

HDLs in detail (27)

Behavioural-level Verilog: blocks and statements

► The simplest statement is a **procedural assignment**:

Listing (Verilog)
<pre>1 module foo(input wire clk); 2 3 reg x, y; 4 5 always @ (posedge clk) begin 6 x = 1'b0; 7 y = 1'b1; 8 end 9 10 endmodule</pre>

Notes:

- ▶ **Beware:** this is different from a continuous assignment, because
 1. a continuous assignments *must* use a wire on the LHS, whereas a procedural assignments *must* use a register; we assign (or latch) *not* drive the value, and
 2. the LHS is assigned to whatever the RHS evaluates to when the statement executes, *not* whenever the RHS changes.

Notes:

- ▶ Standard procedural assignments can be augmented by
 - ▶ introducing one of two forms of **delay**:
 1. A **regular** delay, e.g.,
$$\#10 \ x = 0;$$

means that relative to the previous statement, this one will execute 10 time units later.
 2. An **intra-assignment** delay, e.g.,
$$x = \#10 \ 0;$$

means that the RHS is evaluated straight away, but only assigned to the LHS after 10 time units.
 - ▶ using the **non-blocking** (vs. **blocking**) type:
 1. If we write
$$x = 0; \ y = 1;$$

the assignment to y is blocked until the assignment to x is executed.
 2. If we write
$$x <= 0; \ y <= 1;$$

the assignments to x and y can happen at the same time.

Notes:

- ▶ The next simplest are **conditional statements**:

Listing (Verilog)	Listing (Verilog)
<pre> 1 module bar(input wire clk); 2 3 reg x, y; 4 5 always @ (posedge clk) begin 6 if(x == 1'b0) begin 7 y = 1'b1; 8 end else begin 9 y = 1'b0; 10 end 11 end 12 13 endmodule </pre>	<pre> 1 module baz(input wire clk); 2 3 reg x, y; 4 5 always @ (posedge clk) begin 6 case(x) 7 1'b0 : y = 1'b1; 8 1'b1 : y = 1'b0; 9 default : y = 1'b0; 10 endcase 11 end 12 13 endmodule </pre>

- ▶ Note that:
 - ▶ It starts to be attractive to leave out the **begin** and **end** keywords for single line blocks; this is equivalent to the same rule with “curly braces” in C.
 - ▶ We need to take care with unknown or high impedance values; if **x** doesn’t equal 0 or 1 you may get unexpected behaviour.

- ▶ Verilog supports two distinct approaches to intra-module modularity, namely

Definition (function)	Definition (task)
<ul style="list-style-type: none">▶ defined using the function and endfunction keywords,▶ can only invoke other functions,▶ can be invoked as part of an expression,▶ must have at least one input and no (explicit) outputs because there is always one implicit output, and▶ cannot contain delays.	<ul style="list-style-type: none">▶ are defined using the task and endtask keywords,▶ can invoke other tasks and other functions,▶ cannot be invoked as part of an expression: invocation is more like a statement,▶ any number of inputs and outputs, and▶ can contain fairly arbitrary statements including delays.

where both

- ▶ are local to (i.e., can only be used by) the module they are defined in,
- ▶ can access all wires and registers defined within said module and can define their own wires and registers.

Notes:

Notes:

Listing (Verilog)

```
1 function parity;
2   input [ 7 : 0 ] x;
3
4   begin
5     parity = x[ 0 ] ^ x[ 1 ] ^ x[ 2 ] ^ x[ 3 ] ^
6             x[ 4 ] ^ x[ 5 ] ^ x[ 6 ] ^ x[ 7 ];
7   end
8 endfunction
```

Listing (Verilog)

```
1 function [ 31 : 0 ] byteswap;
2   input [ 31 : 0 ] x;
3
4   begin
5     byteswap = { x[ 7 : 0 ], x[ 15 : 8 ],
6                x[ 23 : 16 ], x[ 31 : 24 ] };
7   end
8 endfunction
```

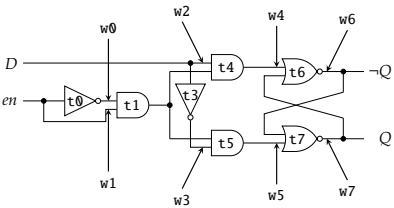
Notes:

Listing (Verilog)

```
1 task parity_check;
2   output
3   input [ 7 : 0 ] x;
4   input y;
5
6   begin
7     if ( parity( x ) == y ) begin
8       r = 1;
9     end else begin
10      r = 0;
11    end
12  end
13 endtask
```

Notes:

Circuit

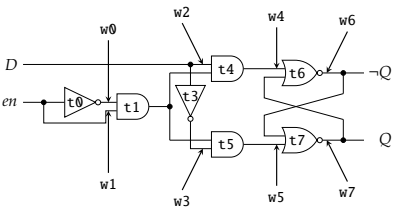


Listing (Verilog)

```
1 module dff( input  wire  en,
2
3             input  wire  D,
4             output wire  Q );
5
6   wire w0, w1, w2, w3, w4, w5, w6, w7;
7
8   not t0( w0,      en );
9   and t1( w1, w0, en );
10
11  buf t2( w2, D      );
12  not t3( w3, D      );
13
14  and t4( w4, w2, w1 );
15  and t5( w5, w3, w1 );
16
17  nor t6( w6, w4, w7 );
18  nor t7( w7, w5, w6 );
19
20  buf t8( Q, w7      );
21
22 endmodule
```

Notes:

Circuit



Listing (Verilog)

```
1 module dff( input  wire  en,
2
3             input  wire  D,
4             output wire  Q );
5
6   reg t;
7
8   assign Q = t;
9
10  always @ ( posedge en ) begin
11    t = D;
12  end
13
14 endmodule
```

Notes:

Circuit

```
graph LR
    x --> mul["x"]
    y --> mul
    mul --> add["+"]
    z --> add
    add --> reg["Flip-flop based register"]
    reg --> r
```

Listing (Verilog)

```
1 module mac_8bit( output wire [ 7 : 0 ] r,
2                 input wire [ 7 : 0 ] x,
3                 input wire [ 7 : 0 ] y,
4                 input wire [ 7 : 0 ] z );
5
6     assign r = ( x * y ) + z;
7
8 endmodule
```

Notes:

Circuit

```
graph LR
    x --> mul["0-th stage"]
    y --> mul
    mul --> reg1["1-st register(s)"]
    reg1 --> add["+"]
    z --> reg1
    add --> reg2["1-st register(s)"]
    reg2 --> reg3["2-nd register(s)"]
    reg3 --> r
```

Listing (Verilog)

```
1 module mac_8bit( input wire adv,
2
3                 output wire [ 7 : 0 ] r,
4                 input wire [ 7 : 0 ] x,
5                 input wire [ 7 : 0 ] y,
6                 input wire [ 7 : 0 ] z );
7
8     reg [ 7 : 0 ] r1a, r2a;
9     reg [ 7 : 0 ] r1b, r2b;
10
11     wire [ 7 : 0 ] x0 = x;
12     wire [ 7 : 0 ] y0 = y;
13     wire [ 7 : 0 ] z0 = z;
14
15     wire [ 7 : 0 ] t1 = x0 * y0;
16     wire [ 7 : 0 ] t2 = r1a + r1b;
17
18     assign r = r2a;
19
20     always @ ( posedge adv ) begin
21         r2a = t2;
22         r1a = t1; r1b = z0;
23     end
24
25 endmodule
```

Notes:

Algorithm

```

graph LR
    start((start)) --> S_even((S_even))
    S_even -- "X_i = 0" --> S_even
    S_even -- "X_i = 1" --> S_odd((S_odd))
    S_odd -- "X_i = 1" --> S_even
    S_odd -- "X_i = 0" --> S_odd
    
```

Listing (Verilog)

```

1 module fsm( input  wire clk,
2
3             output wire  r,
4             input  wire  X );
5
6   reg Q;
7
8   assign r = ( Q == 1'b0 );
9
10  initial begin
11    Q = 1'b0;
12  end
13
14  always @ ( posedge clk ) begin
15    case( { Q, X } )
16      { 1'b0, 1'b0 } : Q = 1'b1;
17      { 1'b0, 1'b1 } : Q = 1'b0;
18      { 1'b1, 1'b0 } : Q = 1'b0;
19      { 1'b1, 1'b1 } : Q = 1'b1;
20    endcase
21  end
22
23 endmodule

```

Notes:

- It’s now reasonable to start demanding more from the EDA tools:
 - We can use a **pre-processor** to
 - define symbolic names for literals, e.g.,


```

`define TRUE 1

then
`use those symbolic names e.g.,

assign r = x ^ `TRUE;

```

Notes:

- ▶ It's now reasonable to start demanding more from the EDA tools:

2. We can use **named ports** to avoid misconnections, e.g.,

```
fa t( .co(a), .s(b), .ci(c), .x(d), .y(e) );
```

is the same as

```
fa t( .co(a), .s(b), .ci(c), .y(e), .x(d) );
```

Notes:

- ▶ It's now reasonable to start demanding more from the EDA tools:

3. We can **parametrise** modules:

- ▶ their interface and behaviour is specified by a *single* fragment of source code,
- ▶ each instance can be altered to suit the context it is used in.

Notes:

- It's now reasonable to start demanding more from the EDA tools:

4. We can **generate** regular fragments of source code at compile-time (cf. meta-programming, vs. "copy and paste").

Notes:

- It's now reasonable to start demanding more from the EDA tools:

1. We can use a **pre-processor** to
 - define symbolic names for literals, e.g.,

```
`define TRUE 1
```

```
then
```

- use those symbolic names e.g.,

```
assign r = x ^ `TRUE;
```

2. We can use **named ports** to avoid misconnections, e.g.,

```
fa t( .co(a), .s(b), .ci(c), .x(d), .y(e) );
```

is the same as

```
fa t( .co(a), .s(b), .ci(c), .y(e), .x(d) );
```

3. We can **parametrise** modules:
 - their interface and behaviour is specified by a *single* fragment of source code,
 - each instance can be altered to suit the context it is used in.
4. We can **generate** regular fragments of source code at compile-time (cf. meta-programming, vs. "copy and paste").

Notes:

Listing (Verilog)

```
1 `define N 8
2
3 module mux2_nbit( output wire [ `N - 1 : 0 ] r,
4                  input wire      c,
5                  input wire [ `N - 1 : 0 ] x,
6                  input wire [ `N - 1 : 0 ] y );
7
8     assign r = c ? y : x;
9
10 endmodule
```

Notes:

Listing (Verilog)

```
1 module mux2_1bit( output wire r,
2                  input wire c,
3                  input wire x,
4                  input wire y );
5
6 `ifdef GATES
7     wire w0, w1, w2;
8
9     not t0( w0, c );
10
11     and t1( w1, x, w0 );
12     and t2( w2, y, c );
13
14     or t3( r, w1, w2 );
15 `else
16     assign r = c ? y : x;
17 `endif
18
19 endmodule
```

Notes:

Listing (Verilog)

```
1 module mux2_nbit( r, c, x, y );
2
3   parameter N = 1;
4
5   output wire [ N - 1 : 0 ] r;
6   input wire      c;
7   input wire [ N - 1 : 0 ] x;
8   input wire [ N - 1 : 0 ] y;
9
10  assign r = c ? y : x;
11
12 endmodule
```

Notes:

Listing (Verilog)

```
1 module mux2_4bit( output wire [ 3 : 0 ] r,
2                   input wire      c,
3                   input wire [ 3 : 0 ] x,
4                   input wire [ 3 : 0 ] y );
5
6   mux2_nbit t( r, c, x, y );
7
8   defparam t.N = 4;
9
10 endmodule
11
12 module mux2_8bit( output wire [ 7 : 0 ] r,
13                  input wire      c,
14                  input wire [ 7 : 0 ] x,
15                  input wire [ 7 : 0 ] y );
16
17   mux2_nbit t( r, c, x, y );
18
19   defparam t.N = 8;
20
21 endmodule
```

Notes:

HDLs in detail (41)

Development best-practice: (more) effective modelling

Listing (Verilog)

```
1 module mux2_4bit( output wire [ 3 : 0 ] r,  
2                   input wire    c,  
3                   input wire [ 3 : 0 ] x,  
4                   input wire [ 3 : 0 ] y );  
5  
6   mux2_1bit t0( r[ 0 ], c, x[ 0 ], y[ 0 ] );  
7   mux2_1bit t1( r[ 1 ], c, x[ 1 ], y[ 1 ] );  
8   mux2_1bit t2( r[ 2 ], c, x[ 2 ], y[ 2 ] );  
9   mux2_1bit t3( r[ 3 ], c, x[ 3 ], y[ 3 ] );  
10  
11 endmodule
```

Notes:

HDLs in detail (42)

Development best-practice: (more) effective modelling

Listing (Verilog)

```
1 module mux2_4bit( output wire [ 3 : 0 ] r,  
2                   input wire    c,  
3                   input wire [ 3 : 0 ] x,  
4                   input wire [ 3 : 0 ] y );  
5   genvar i;  
6   generate  
7     for( i = 0; i < 4; i = i + 1 ) begin: id  
8       mux2_1bit t( r[ i ], c, x[ i ], y[ i ] );  
9     end  
10  endgenerate  
11  
12  
13 endmodule
```

Notes:

- ▶ Sometimes it's useful for the model and simulator to interact; this is achieved using **system tasks** and **system functions** ...
- ▶ ... they look like normal tasks and functions, but are “executed” by the simulator rather than user-defined Verilog; probably the most useful are:
 - `$random` ⇒ generates random value(s)
 - `$display` ⇒ displays value(s) synchronously
 - `$monitor` ⇒ displays value(s) asynchronously
 - `$stop` ⇒ halt current simulation
 - `$finish` ⇒ terminate current simulation

Notes:

Listing (Verilog)

```

1 module fa_test();
2
3   wire t_co,      t_s;
4   reg  t_ci; t_x, t_y;
5
6   fa( t_co( t_co ), .s( t_s ), .ci( t_ci ), .x( t_x ), .y( t_y ) );
7
8   initial begin
9     #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b0;
10    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
11    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b1;
12    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
13    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b0;
14    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
15    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b1;
16    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
17
18    #10 $finish;
19  end
20
21 endmodule

```

Notes:

Listing (Verilog)

```
1 module fa_test();
2
3   wire t_co,      t_s;
4   reg  t_ci; t_x, t_y;
5
6   fa t( .co( t_co ), .s( t_s ), .ci( t_ci ), .x( t_x ), .y( t_y ) );
7
8   initial begin
9       $monitor( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
10
11       $monitoron;
12
13       #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b0;
14       #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b1;
15       #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b0;
16       #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b1;
17
18       #10 $monitoroff;
19       $finish;
20   end
21
22 endmodule
```

Notes:

Listing (Verilog)

```
1 module fa_test();
2
3   wire t_co,      t_s;
4   reg  t_ci; t_x, t_y;
5
6   fa t( .co( t_co ), .s( t_s ), .ci( t_ci ), .x( t_x ), .y( t_y ) );
7
8   wire o_co,      o_s;
9
10  assign { o_co, o_s } = t_ci + t_x + t_y;
11
12  wire f = ( o_co == t_co ) &
13           ( o_s == t_s );
14
15  initial begin
16      $monitor( "f=%s ci=%b x=%b y=%b", f ? "pass" : "fail", t_ci, t_x, t_y );
17
18      $monitoron;
19
20      #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b0;
21      #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b1;
22      #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b0;
23      #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b1;
24
25      #10 $monitoroff;
26      $finish;
27  end
28
29 endmodule
```

Notes:

Conclusions

- ▶ Take away points:

1. In essence, a HDL model is just machine-readable short-hand for a design you could develop and reason about on paper.
2. It's important to remember that, despite appearances,

HDL modelling \neq software development,

i.e., you *still* have to understand the fundamentals and “think in hardware”.

3. Even within this unit, HDLs offer various useful properties, e.g.,

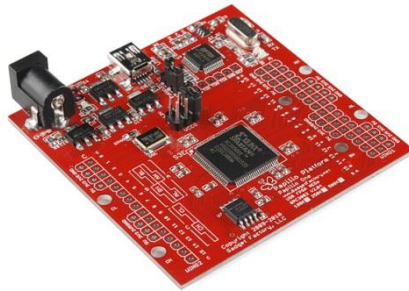
- ▶ adopt a more accurate experimental approach to design,
- ▶ deal with designs of a larger scale,
- ▶ interface with other concepts (e.g., verification),
- ▶ ...

so some up-front, invested effort *could* pay off ...

Notes:

Conclusions

- ▶ **Example:** Field Programmable Gate Arrays (FPGAs).



- ▶ basic idea is that the hardware fabric is reconfigurable, so, in a sense,

hardware (e.g., ASIC) hybrid (e.g., FPGA) software (e.g., micro-processor)

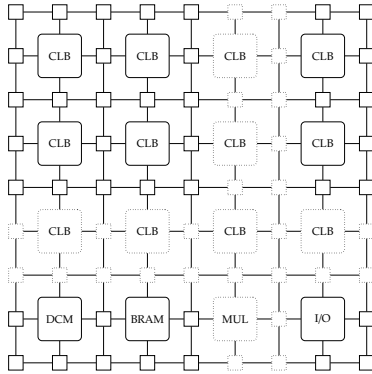
- ▶ and therefore offers a trade-off:

efficiency \simeq hardware
flexibility \simeq software

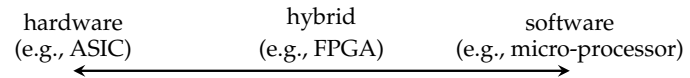
Notes:

Conclusions

► Example: Field Programmable Gate Arrays (FPGAs).



- basic idea is that the hardware fabric is reconfigurable, so, in a sense,



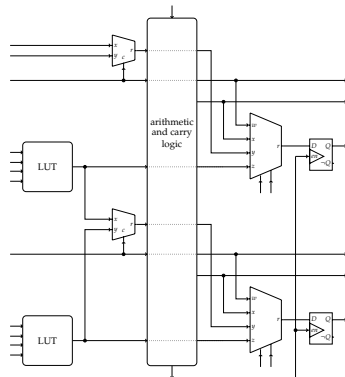
- and therefore offers a trade-off:

efficiency \approx hardware
flexibility \approx software

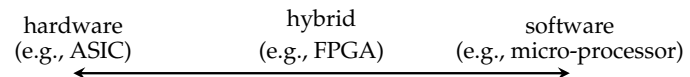
Notes:

Conclusions

► Example: Field Programmable Gate Arrays (FPGAs).



- basic idea is that the hardware fabric is reconfigurable, so, in a sense,



- and therefore offers a trade-off:

efficiency \approx hardware
flexibility \approx software

Notes:

Additional Reading

- ▶ *Wikipedia: Hardware Description Language (HDL)*. . URL: http://en.wikipedia.org/wiki/Hardware_description_language.
- ▶ *Wikipedia: Verilog*. URL: <http://en.wikipedia.org/wiki/Verilog>.
- ▶ S. Palnitkar. *Verilog HDL: A Guide in Digital Design and Synthesis*. 2nd ed. Prentice-Hall, 2003.
- ▶ D. Page. “Chapter 3: Hardware design using Verilog”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009.

Notes:

References

- [1] *Wikipedia: Hardware Description Language (HDL)*. URL: http://en.wikipedia.org/wiki/Hardware_description_language (see p. 157).
- [2] *Wikipedia: Verilog*. URL: <http://en.wikipedia.org/wiki/Verilog> (see p. 157).
- [3] D. Page. “Chapter 3: Hardware design using Verilog”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 157).
- [4] S. Palnitkar. *Verilog HDL: A Guide in Digital Design and Synthesis*. 2nd ed. Prentice-Hall, 2003 (see p. 157).

Notes: