

### 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     $\supset$     Python
4. Chisel     $\supset$     Scala
- ⋮

### 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     $\supset$     Python
4. Chisel     $\supset$     Scala
- ⋮

#### ► Agenda: Verilog, or, more specifically,

1. foundational concepts,
2. low-level modelling,
3. high-level modelling, and
4. development concepts, e.g., testing and test stimuli.

#### ► Caveat!

~ 2.5 hours  $\Rightarrow$  introductory coverage of *core* language features and workflow.

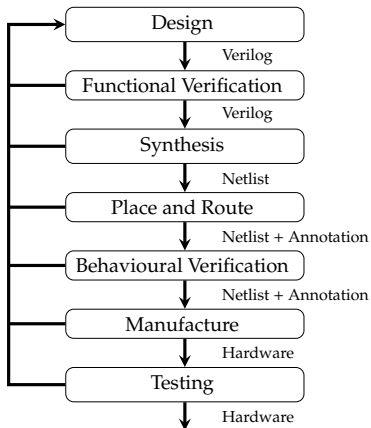
## Part 1: foundational concepts (1)

- ▶ **Question:** *why?!*
- ▶ **Answer:** HDLs (and EDA tools more generally) help to
  1. facilitate automation, e.g., with respect to
    - ▶ simulation,
    - ▶ verification, and
    - ▶ translationof what is a more clearly machine-readable design,
  2. address the challenge of scale, e.g., with respect to design size and complexity.

## Part 1: foundational concepts (2)

► **Question:** *how?!*

► **Answer:** as part of a broader development workflow, such as



► You can think of

synthesis  $\approx$  compilation  
place and route  $\approx$  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.

## Part 1: foundational concepts (3)

### ► Analogy:

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

*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 various ways.

## Part 1: foundational concepts (4)

### ► Example:

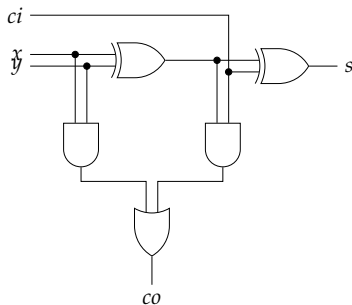
```
module fa( output wire co,
           output wire s,
           input wire ci,
           input wire x,
           input wire y );

  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
```



## Part 1: foundational concepts (5)

- **Example:** a model can be described in
  1. a high-level, behaviour-oriented style, or
  2. a low-level, implementation-oriented style, or
  3. a *hybrid* of the two

so, e.g.,

### 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  $\neg(b \vee c)$ .

## Part 1: foundational concepts (5)

- **Example:** a model can be described in
1. a high-level, behaviour-oriented style, or
  2. a low-level, implementation-oriented style, or
  3. a *hybrid* of the two

so, e.g.,

### 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  $\neg(b \vee c)$ .



## Part 1: foundational concepts (5)

- **Example:** a model can be described in
1. a high-level, behaviour-oriented style, or
  2. a low-level, implementation-oriented style, or
  3. a *hybrid* of the two

so, e.g.,

### 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  $\neg(b \vee c)$ .

## Part 1: foundational concepts (5)

- **Example:** a model can be described in
1. a high-level, behaviour-oriented style, or
  2. a low-level, implementation-oriented style, or
  3. a *hybrid* of the two

so, e.g.,

### 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  $\neg(b \vee c)$ .

## Part 2: low-level modelling (1)

### Wires and values

#### ► **Concept: wires** (resp. **wire vectors**)

► are a form of **net** used to *communicate* values,

► e.g.,

- `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`

## Part 2: low-level modelling (2)

### Wires and values

#### ► Concept: values

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; the bit is unknown
  - $4'b10XZ \Rightarrow$  a 4-bit binary literal; the bits are high impedance, unknown, 0, and 1

## Part 2: low-level modelling (3)

### Wires and values

#### ► Analogy:

#### C

- The definition

`char u`  $\rightsquigarrow$  8 *separate* 1-bit elements

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

- The definition

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

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

#### Verilog

- The definition

`wire x`  $\rightsquigarrow$  1 *single* 1-bit wire

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

- The definition

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

such that

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

*but, beware:*

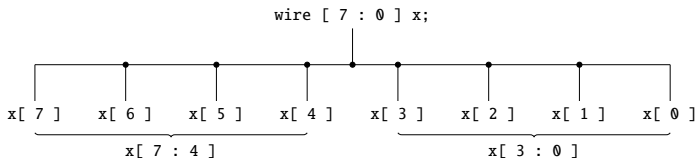
- a wire (resp. wire vector) cannot retain state (e.g., doesn't behave like a C variable),
- we need to *drive* a value on it.

## Part 2: low-level modelling (4)

### Wires and values

#### ► Concept: subscript operator.

- consider a case where  $x = 8'b11110000$ , and



#### ► we have that

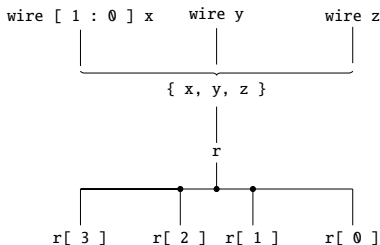
- $x[7]$ ,  $x[6]$ ,  $x[5]$  and  $x[4]$  are all 1-bit wires with value  $1'b1$ ,
- $x[3]$ ,  $x[2]$ ,  $x[1]$  and  $x[0]$ , are all 1-bit wires with value  $1'b0$ ,
- $x[7:4]$  is a 4-bit wire vector with value  $4'b1111$ , and
- $x[3:0]$  is a 4-bit wire vector with value  $4'b0000$ .

## Part 2: low-level modelling (5)

### Wires and values

#### ► **Concept: concatenate operator.**

- consider a case where  $x = 2'b10$ ,  $y = 1'b1$ , and  $z = 1'b0$ , and



#### ► we have that

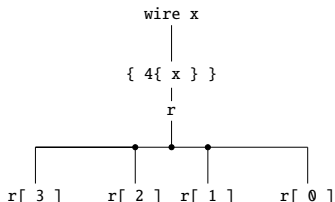
- { x, y, z } is a 4-bit wire vector with value  $4'b1010$ ,
- $r[ 3 ]$  is a 1-bit wire with value  $1'b1$  (matching  $x[ 1 ]$ ),
- $r[ 2 ]$  is a 1-bit wire with value  $1'b0$  (matching  $x[ 0 ]$ ),
- $r[ 1 ]$  is a 1-bit wire with value  $1'b1$  (matching  $y$ ), and
- $r[ 0 ]$  is a 1-bit wire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

## Part 2: low-level modelling (6)

### Wires and values

#### ► **Concept:** replicate operator.

- consider a case where  $x = 1'b1$ , and



#### ► 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  $1'b1$ .



## Part 2: low-level modelling (7)

### Modules

#### ► Concept: module **definition**

- are a passive (or static) *description* of a component,
- e.g.,

#### Listing (Verilog)

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

#### Listing (Verilog)

```
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
```

noting the two forms are equivalent.

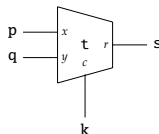
## Part 2: low-level modelling (8)

### Modules

#### ► **Concept:** module **instantiation**

- are an active (or dynamic) *use* of a component,
- e.g.,

```
mux2_1bit t( s, k, p, q );
```



where we've

- created an instance of the `mux2_1bit` module identified by `t`, and
- connected the internal ports `r`, `c`, `x` and `y` to the external wires `s`, `k`, `p` and `q`

## Part 2: low-level modelling (9)

### Modules

#### ► Analogy:

#### C

- A caller function invokes (or calls) a callee function.
- 1 *shared* copy of a callee function is used by  $n$  invocations.
- Each invocation executes in *sequence*, and *discontinuously*.

#### Verilog

- A instanciator module instantiates a instantiatee module.
- $n$  *separate* copies of a instantiatee module are produced by  $n$  instantiations.
- Each instance operates in *parallel*, and *continuously*.

### ► Concept: gate-level module implementation

- describes module behaviour via
  1. primitive (or built-in) modules, and/or
  2. other user-defined modules,
- e.g.,

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

noting that multi-input variants such as

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

are automatically available.

### ► Concept: User-Defined Primitives (UDPs)

- describe module behaviour via a truth table,
- doing so assumes it models a Boolean function of the form

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

- e.g.,

#### Listing (Verilog)

```
1 primitive mux2_1bit( output r,  
2                     input c,  
3                     input x,  
4                     input y );  
5     table  
6         0 0 ? : 0;  
7         0 1 ? : 1;  
8         1 ? 0 : 0;  
9         1 ? 1 : 1;  
10    endtable  
11  
12 endprimitive
```

#### Truth table

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.

## Part 2: low-level modelling (12)

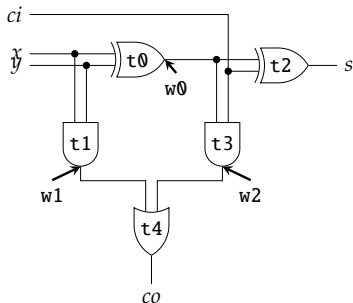
Module implementation using gate-level Verilog

### ► Example:

#### 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)



## Part 2: low-level modelling (13)

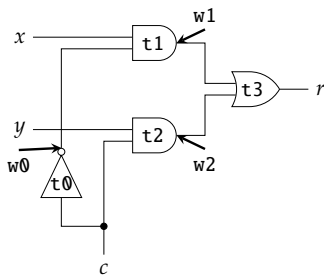
Module implementation using gate-level Verilog

### ► Example:

#### 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)



## Part 2: low-level modelling (14)

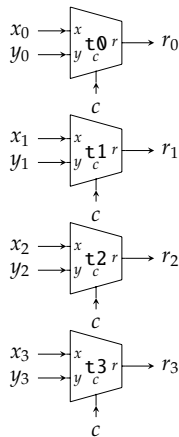
Module implementation using gate-level Verilog

### ► Example:

#### 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)





## Part 2: low-level modelling (15)

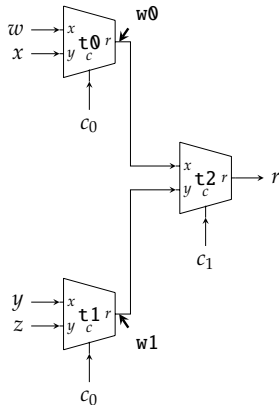
Module implementation using gate-level Verilog

### ► Example:

#### Listing (Verilog)

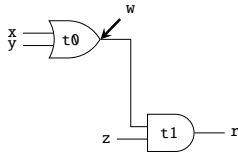
```
1 module mux4_1bit( output wire r,  
2                   input wire c0,  
3                   input wire c1,  
4                   input wire w,  
5                   input wire x,  
6                   input wire y,  
7                   input wire z );  
8  
9   wire w0, w1;  
10  
11  mux2_1bit 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)



► **Concept: Register Transfer Level (RTL) module implementation**

- ▶ describes module behaviour via
  1. a set of **continuous assignments**, plus
  2. any additional gate-level description
- ▶ e.g.,

$$\text{assign } r = (x \mid y) \ \& \ z; \quad \mapsto \quad \begin{array}{l} \text{or } t0( w, x, y ); \\ \text{and } t1( r, w, z ); \end{array} \quad \mapsto$$


- ▶ the LHS *must* be a wire or wire vector, whereas
- ▶ 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.

*but, beware:*

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

### ► **Concept: reduction operator.**

- consider a case where wire [ 3 : 0 ] x, wire [ 3 : 0 ] y, and wire c,
- we have that

$$\wedge x \mapsto ( ( x[3] \wedge x[2] ) \wedge x[1] ) \wedge x[0]$$

so is analagous to reduce (or foldr) in Haskell.

► **Concept: ternary operator.**

- consider a case where wire [ 3 : 0 ] x, wire [ 3 : 0 ] y, and wire c,
- we have that

$$c ? y : x \mapsto \begin{cases} x & \text{if } c = 0 \\ y & \text{if } c = 1 \end{cases}$$

so is analagous to a 2-input multiplexer.

## Part 2: low-level modelling (19)

Module implementation using RTL-level Verilog

### ► Example:

#### 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
```

## Part 2: low-level modelling (19)

Module implementation using RTL-level Verilog

### ► Example:

#### 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
```

## Part 2: low-level modelling (20)

Module implementation using RTL-level Verilog

### ► Example:

#### 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
```

## Part 2: low-level modelling (21)

Module implementation using RTL-level Verilog

### ► Example:

#### 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
```



### ► Example:

#### 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
11  not t0( w0, c0 );
12  not t1( w1, c1 );
13
14  and t2( w2, w0, w1, w );
15  and t3( w3, c0, w1, x );
16  and t4( w4, w0, c1, y );
17  and t5( w5, c0, c1, z );
18
19  or t6( r, w2, w3, w4, w5 );
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
```

## Part 3: high-level modelling (1)

### Registers

#### ► **Concept: registers** (resp. **register vectors**)

► are a form of **net** used to *store* values (i.e., retain state),

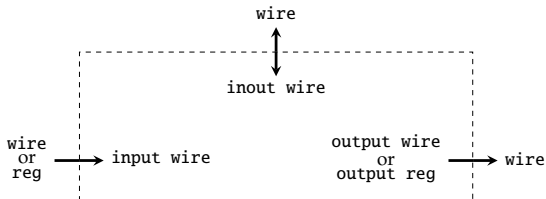
► e.g.,

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

*but, beware:* registers *feel* analogous to C-style variables, but care is required re. use.

#### ► Concept: module interfacing rules

##### Definition



which are *somewhat* intuitive when read as

input port :  $\begin{cases} \text{externally} & \text{can be a wire or reg} \\ \text{internally} & \text{must be a wire} \end{cases}$

output port :  $\begin{cases} \text{internally} & \text{can be a wire or reg} \\ \text{externally} & \text{must be a wire} \end{cases}$

i.e., we must be pessimistic when crossing the module boundary.

## Part 3: high-level modelling (3)

Module implementation using behavioural-level Verilog

### ► Concept: behavioural-level module implementation

- describes module behaviour via
  - (at least partly) using **processes**,
  - each process is formed from **blocks** of **statements**,
  - each process is “executed” in parallel with the others once **triggered**.
- e.g.,

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

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

shows the two process types

- **initial**    ⇒    triggered only *once* (when the module is first powered-on)
- **always**    ⇒    triggered in a *loop* (as long as the module is powered-on)

noting the right-hand case is problematic as is ...

## Part 3: high-level modelling (3)

Module implementation using behavioural-level Verilog

### ► Concept: behavioural-level module implementation

- describes module behaviour via
  - (at least partly) using **processes**,
  - each process is formed from **blocks** of **statements**,
  - each process is “executed” in parallel with the others once **triggered**.
- e.g.,

#### Listing (Verilog)

```
1 always @ ( x ) begin
2   ...
3 end
```

#### Listing (Verilog)

```
1 always @ ( posedge x ) begin
2   ...
3 end
```

#### Listing (Verilog)

```
1 always @ ( negedge x ) begin
2   ...
3 end
```

shows processes that are triggered 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)

- **Concept:** procedural assignment, e.g.,

### Listing (Verilog)

```
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
```

which differ from a continuous assignment: they

1. must use a register as the LHS (versus a wire), and
2. the LHS is assigned to whatever the RHS evaluates to when the statement executes (versus whenever the RHS changes).

- **Concept:** procedural assignment, e.g.,

### Listing (Verilog)

```
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
```

which can introduce modelled **delay**:

- a **regular** delay, e.g.,

`#10 x = 0;`

means that, relative to the previous statement, this one will execute 10 time units later, whereas

- 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.

- **Concept:** procedural assignment, e.g.,

### Listing (Verilog)

```
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
```

which come in **blocking** or **non-blocking** variants:

- if we write

$$x = 0; y = 1;$$

then the assignment to y is blocked until the assignment to x is executed, whereas

- if we write

$$x <= 0; y <= 1;$$

then the assignments to x and y are executed in parallel.



## Part 3: high-level modelling (5)

Module implementation using behavioural-level Verilog

- **Concept:** conditional statements, e.g.,

### Listing (Verilog)

```
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
```

### Listing (Verilog)

```
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
```

noting 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.

## Part 3: high-level modelling (6)

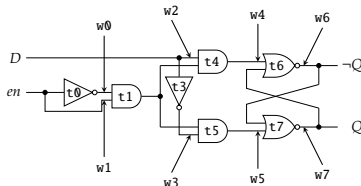
Module implementation using behavioural-level Verilog

### ► Example:

#### 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
```

#### Circuit



## Part 3: high-level modelling (6)

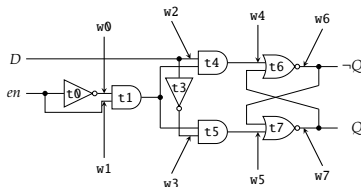
Module implementation using behavioural-level Verilog

### ► Example:

#### 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
```

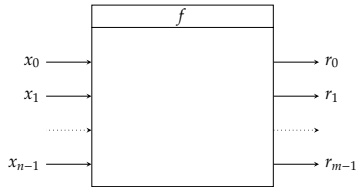
#### Circuit



## Part 4: development concepts (1)

### Testing

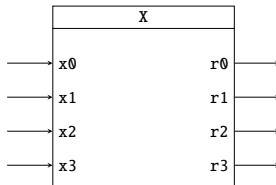
- **Concept:** test stimulus (or test harness).



## Part 4: development concepts (1)

### Testing

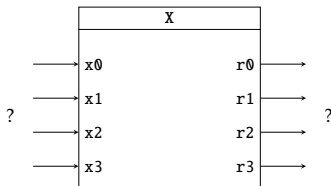
- **Concept:** test stimulus (or test harness).



## Part 4: development concepts (1)

### Testing

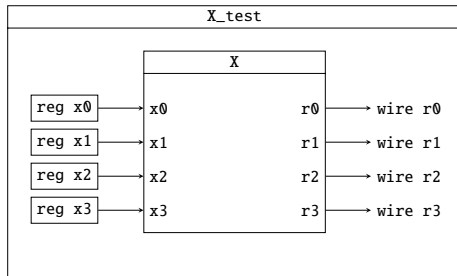
- **Concept:** test stimulus (or test harness).



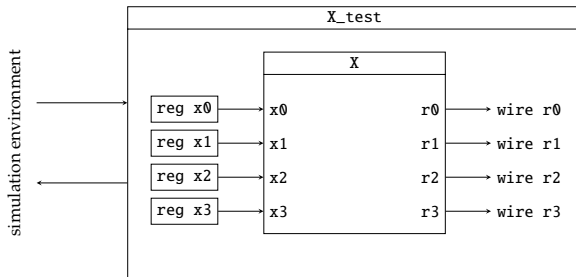
## Part 4: development concepts (1)

### Testing

- **Concept:** test stimulus (or test harness).



#### ► Concept: test stimulus (or test harness)



noting that **X\_test**

- is termed a (or the) **top-level module** in the sense it has no inputs or outputs,
- can interact with the simulation environment is via **system tasks** and **system functions**, e.g.,
  - **\$random** ⇒ generates random value(s)
  - **\$display** ⇒ displays value(s) synchronously
  - **\$monitor** ⇒ displays value(s) asynchronously
  - **\$stop** ⇒ halt current simulation
  - **\$finish** ⇒ terminate current simulation
- will apply some form of **test strategy** to the instance of **X**.



#### ► Example:

#### 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
```

## Part 4: development concepts (3)

### Testing

#### ► Example:

#### 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
```

## Part 4: development concepts (4)

“Quality-of-life” features

- **Concept:** a “better” model  $\leadsto$  greater **generalisation, maintainability**, etc.

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 (cf. meta-programming, versus “copy and paste”).

## Part 4: development concepts (5)

“Quality-of-life” features: pre-processor

### ► Example:

#### 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
```

## Part 4: development concepts (6)

“Quality-of-life” features: pre-processor

### ► Example:

#### 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
```

## Part 4: development concepts (7)

“Quality-of-life” features: parameterisation

### ► Example:

#### 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
```

#### 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
```

## Part 4: development concepts (8)

“Quality-of-life” features: generation

### ► Example:

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

## ► 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.,

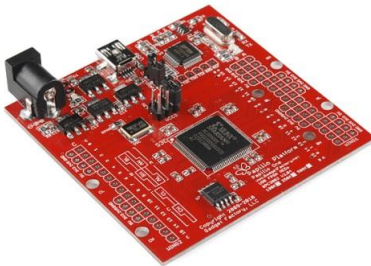
- adopt a more accurate experimental approach to design,
- deal with designs of a larger scale,
- interface with other concepts (e.g., verification),
- ...

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



## Conclusions

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



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

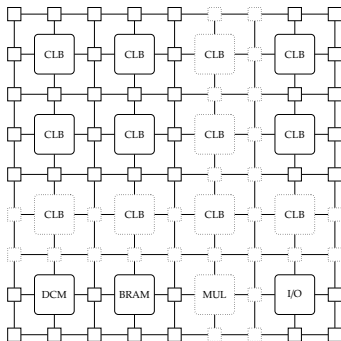
hardware (e.g., ASIC)      hybrid (e.g., FPGA)      software (e.g., micro-processor)

- ▶ and therefore offers a trade-off:

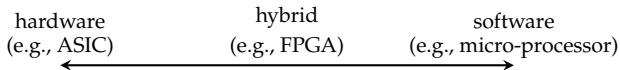
efficiency  $\simeq$  hardware  
flexibility  $\simeq$  software

## Conclusions

### ► Example: Field Programmable Gate Arrays (FPGAs).



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

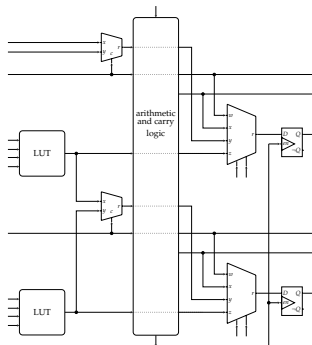


- and therefore offers a trade-off:

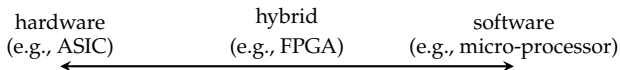
efficiency  $\approx$  hardware  
flexibility  $\approx$  software

## 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

## Additional Reading

- ▶ *Wikipedia: Hardware Description Language (HDL)*. URL: [https://en.wikipedia.org/wiki/Hardware\\_description\\_language](https://en.wikipedia.org/wiki/Hardware_description_language).
- ▶ *Wikipedia: Verilog*. URL: <https://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, 2009.

# References

- [1] *Wikipedia: Hardware Description Language (HDL)*. URL: [https://en.wikipedia.org/wiki/Hardware\\_description\\_language](https://en.wikipedia.org/wiki/Hardware_description_language) (see p. 60).
- [2] *Wikipedia: Verilog*. URL: <https://en.wikipedia.org/wiki/Verilog> (see p. 60).
- [3] D. Page. “Chapter 3: Hardware design using Verilog”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 60).
- [4] S. Palnitkar. *Verilog HDL: A Guide in Digital Design and Synthesis*. 2nd ed. Prentice Hall, 2003 (see p. 60).