TTÜ1918

# Verilog / SystemVerilog

- **History & main concepts – structure, description styles, data types**

- **Procedural & assignment; if-then, case & loop statements**

- **Functional hierarchy – tasks & functions**

- **Time & events; parallelism; fork, join & disable statements**

- **Structural & behavioral descriptions**

- **Michael John Sebastian Smith, "Application-Specific Integrated Circuits." Addison-Wesley – http://www10.edacafe.com/book/ASIC/ASICs.php    [see ch. 11]**

- **Stuart Sutherland, Simon Davidmann, Peter Flake and Phil Moorby. "SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling." Springer.**

- **Ken Coffman, "Real world FPGA design with Verilog." Prentice Hall.**

- **Donald E. Thomas, Philip R. Moorby, "The Verilog® Hardware Description Language." Kluwer Academic Publishers.**

- **James M. Lee, "Verilog Quickstart: a practical guide to simulation and synthesis in Verilog." Kluwer Academic Publishers.**

- **http://systemverilog.in/**

- **http://www.project-veripage.com/**

- **http://www.angelfire.com/in/verilogfaq/**

# History

- **Invented as a <u>simulation</u> language**

- **1983/85 – Automated Integrated Design Systems (later as Gateway Design Automation)**

- **1989/90 – acquired by Cadence Design Systems**

- **1990/91 – opened to the public in 1990 - OVI (Open Verilog International) was born**

- **1992 – the first simulator by another company**

- **1993 – IEEE working group (under the Design Automation Sub-Committee) to produce the IEEE Verilog standard 1364**

- **May 1995 – IEEE Standard 1364-1995**

- **2001 – IEEE Standard 1364-2001 – revised version**

- **2005 – IEEE Standard 1364-2005 – clarifications; Verilog-AMS**

- **2005 – IEEE Standard 1364-2001 – SystemVerilog**

- **2009 – Verilog and SystemVerilog merged – IEEE Standard 1800-2009**

- **2013 – IEEE Standard 1800-2012 – SystemVerilog 2012**

- **development continues...**

# SystemVerilog extras

- **New data types - int, shortint, longint, byte, bit, logic, enum, typedef, struct**

- **C-like parameters for modules**

- **New processes - always_comb, always_latch, always_ff**

- **Packages, additional operators, assertions, interfaces**

# SystemVerilog presentations

- **Doulos Ltd – The Guide to SystemVerilog (...–2019)**
  - **https://www.doulos.com/knowhow/sysverilog/**

- **System Verilog Tutorial (2015)**
  - **http://www.slideshare.net/phagwarae2matrix/system-verilog-tutorial-vhdl**

- **SystemVerilog 3.1a LRM (2015)**
  - **http://www.slideshare.net/jainatush/system-verilog-44103589**

- **Manmohan Singh (2007)**
  - **http://www.powershow.com/view1/d3b73-ZDc1Z/
    An_Introduction_to_SystemVerilog_powerpoint_ppt_presentation**

- **Narges Baniasadi (2004)**
  - **http://www.powershow.com/view1/29796d-ZDc1Z/System_Verilog_powerpoint_ppt_presentation**

# Hello, world!

```
module world;

  initial
    begin
      $display ( "Hello, world!" );
    end

endmodule
```

- **ModelSim**

```
run -all
# Hello, world!
```

# Main concepts

- **Modules**

  - **modules**

  - **functions & tasks**

- **Case sensitive**

  - **lower case keywords**

  - **identifier - a sequence of letters, digits, dollar sign ($), and underscore (_)**
    ```
    identifier ::= simple_identifier | escaped_identifier
    simple_identifier ::= [a-zA-Z][a-zA-Z_0-9$]*
    escaped_identifier ::= \{any_ASCII_character_except_white_space} white_space
    ```

- **No delta-delay**
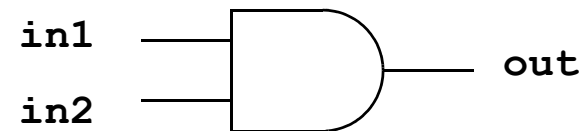
  - **non-deterministic parallelism**

# Module

in1
in2
out

**module** *name* **(** *input_output_list* **);**
   *module_body*
**endmodule**

Ports:

**wire** - by default (can be skipped)

**reg** ~ keeps content

```
// structural
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    wire in1, in2, out;
    and u1 (out, in1, in2);
endmodule
```

TTÜ1918

```
// behavioral
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    wire in1, in2;
    reg out;
    always @( in1 or in2 )
        out = in1 & in2;
endmodule
```

```
// data flow
module AND2 (in1, in2, out);
    input in1;
    input in2;
    output out;
    wire in1, in2, out;
    assign out = in1 & in2;
endmodule
```

TTÜ1918

```
module test_and2;
    reg i1, i2;    wire o;

    AND2 u2 (i1, i2, o);

    initial begin
        i1 = 0; i2 = 0;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
        i1 = 0; i2 = 1;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
        i1 = 1; i2 = 0;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
        i1 = 1; i2 = 1;
        #1 $display("i1 = %b, i2 = %b, o = %b", i1, i2, o);
    end
endmodule
```

- **always**
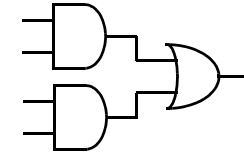- **initial**
- **begin ... end**

```
i1 = 0, i2 = 0, o = 0
i1 = 0, i2 = 1, o = 0
i1 = 1, i2 = 0, o = 0
i1 = 1, i2 = 1, o = 1
```

**Results...**

TTÜ1918

# Example AND-OR

```verilog
module and_or (in1, in2, in3, in4, out);
    input in1, in2, in3, in4;
    output out;
    wire tmp;
    and #10 u1 (tmp, in1, in2),
            u2 (undec, in3, in4);
    or #20 (out, tmp, undec);
endmodule


module and_or (in1, in2, in3, in4, out);
    input in1, in2, in3, in4;
    output out;
    wire tmp;
    assign #10 tmp = in1 & in2;
    wire #10 tmp1 = in3 & in4;
    assign #20 out = tmp | tmp1;
    // assign #30 out = (in1 & in2) | (in3 & in4);
endmodule
```

© Peeter Ellervee

TTÜ1918

```verilog
module and_or (in1, in2, in3, in4, out);
    input in1, in2, in3, in4;
    output out;
    reg out;
    always @(in1 or in2 or in3 or in4) begin
        if (in1 & in2)  out = #30 1;
        else            out = #30 (in3 & in4);
    end
endmodule


module test_and_or;
    reg r1, r2, r3, r4;
    wire o;
    and_or u2 (.in2(r2), .in1(r1), .in3(r3), .in4(r4), .out(o));
    initial begin : b1
        reg [4:0] i1234;
        for ( i1234=0; i1234<16; i1234=i1234+1 ) begin
            { r1, r2, r3, r4 } = i1234[3:0];
            #50 $display("r1r2r3r4=%b%b%b%b, o=%b",r1,r2,r3,r4,o);
        end
    end
endmodule
```

# Data types

- **Constants - decimal, hexadecimal, octal & binary**

  - **Format**  `<width>'<radix><value>`

    - **<width> - optional, in bits, decimal constant**

    - **<radix> - optional, base, can be one of** `b, B, d, D, o, O, h` **or** `H`

    - **<value> - a sequence of symbols depending on the radix:**
      **binary -** `0, 1, x, X, z & Z`
      **octal - also** `2, 3, 4, 5, 6 & 7`
      **hexadecimal - also** `8, 9, a, A, b, B, c, C, d, D, e, E, f & F`
      **decimal -** `0` **to** `9`**, but not** `x` **or** `z`

```
15        (decimal 15)
'h15      (decimal 21, hex 15)
5'b10011  (decimal 19, binary 10011)
12'h01F   (decimal 31, hex 01F)
'b01x     (no decimal value, binary 01x)
```

  - **String constants, e.g. "my-string"**

    - **are converted to their ASCII equivalent binary format, e.g.** `"ab" == 16'h5758`
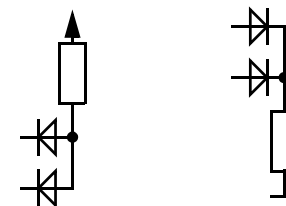
  - **Real constants - ordinary scientific notation**

    - **e.g.**  `22.73 , 12.8e12`

TTÜ1918

- **Physical data types**
  - **binary nets - `wire`, `wand`, `wor`, etc., and**
    - **continuously driven**
  - **registers - `reg`**
    - **"remembers" the last assignment**

  - **Registers can be assigned only inside behavioral instances**
  - **Nets are driven all the time and cannot be assigned in behavioral block**
  - **Register can be interpreted as a storage element (latch, flip-flop) but not necessarily**
  - **Nets & registers are interpreted as unsigned integers**

- **Abstract data types**
  - **`integer` - almost as a 32-bit `reg` but <u>signed</u>**
  - **`time` - 64-bit unsigned integer**
  - **`real` - floating point, platform depending**
  - **`event` - a special variable without value, used for synchronization**
  - **`parameter` - "named constant", set before simulation starts**

# Declarations

- **Width in bits - physical variables only**

- **Arrays - only types *integer*, *real* and *reg***

```verilog
integer i, j;
real f, d;
wire [7:0] bus;          // 1x8 bits
reg [0:15] word;         // 1x16 bits
reg arr[0:15];           // 16x1 bits
reg [7:0] mem[0:127];    // 128x8 bits
event trigger, clock_high;
time t_setup, t_hold;
parameter width=8;
parameter width2=width*2;
wire [width-1:0] ww;
// The following are illegal
wire w[0:15];            // No arrays
wire [3:0] a, [7:0] b;   // Only one width per decl.
```

# Operations

```
+  -  *  /  %      (arithmetic)
>  >=  <  <=       (relational)
!  &&  ||          (logical)
==  !=             (logical equality)
?:                 (conditional)
{}                 (concatenate)
===  !==           (case equality)
~  ^  ^~  &  |     (bit-wise)
<<  >>             (shift)
```

```
+  -  !  ~      (highest)
*  /  %
+  -           (binary op.)
<<  >>
<  <=  >  >=
=  ==  !=
===  !==
&  ~&
^  ^~
|  ~|
&&
||
?:             (lowest)
```

## Bit-wise as unary operations

```
^word === 1'bx
&word === 0
```

## Comparisons

```
'bx ==  'bx     ≡ x
'bx === 'bx     ≡ 1
```

## Concatenation

```
{2'b1x, 4'h7} === 6'b1x0111
{cout, sum} = in1 + in2 + cin;
{sreg, out} = {in, reg};
{3{2'b01}} === 6'b010101
```
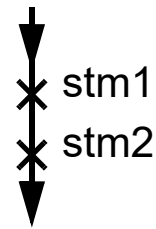
## Indexing

```
reg [15:0] array [0:10];
reg [15:0] temp;
    ...
temp = array[3];
... temp[7:5] ...
// array[3][7:5] is illegal
```
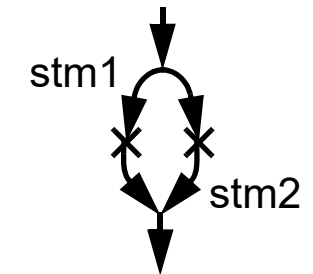
# Procedural and assignment statements

- **Procedural statements**

```
begin
    stm1;
    stm2;
end
```

stm1
stm2

```
fork
    stm1;
    stm2;
join
```

stm1
stm2

- **Assignments**

```
lhs-expression = expression;
lhs-expression = #delay expression;
lhs-expression = @event expression;
```

`Blocking`

```
lhs-expression <= expression;
lhs-expression <= #delay expression;
lhs-expression <= @event expression;
```

`Non-blocking`

# Conditional Statements

```
if ( bool-expr )
   statement
else
   statement
```

```
case ( expr )
expr [, expr]* : statement
default: statement
endcase
```

- **Case**

  - **bit by bit comparison (like ===)**

  - **casez - 'z' is interpreted as don't care**

  - **casex - 'z' & 'x' are interpreted as don't care**

# Loop statements

```
module for_loop;
   integer i;
   initial
     for (i=0;i<4;i=i+1) begin
       ...
     end
endmodule
```

```
module while_loop;
   integer i;
   initial begin
     i=0;
     while (i<4) begin
       ...
       i=i+1;
     end
   end
endmodule
```

```
module repeat_loop(clock);
   input clock;
   initial begin
     repeat (5)
       @(posedge clock);
     $stop;
   end
endmodule
```

```
module forever_loop(a,b,c);
   input a, b, c;
   initial forever begin
     @(a or b or c)
       if ( a+b == c ) $stop;
   end
endmodule
```

# Functional hierarchy

- **Tasks**

```
task tsk;
   input i1, i2;
   output o1, o2;
   $display("Task tsk, i1=%0b, i2=%0b",i1,i2);
   #1 o1 = i1 & i2;
   #1 o2 = i1 | i2;
endtask
```

- **Access:**   `tsk(a,b,c,d);`

- **A task may have timing control construct**
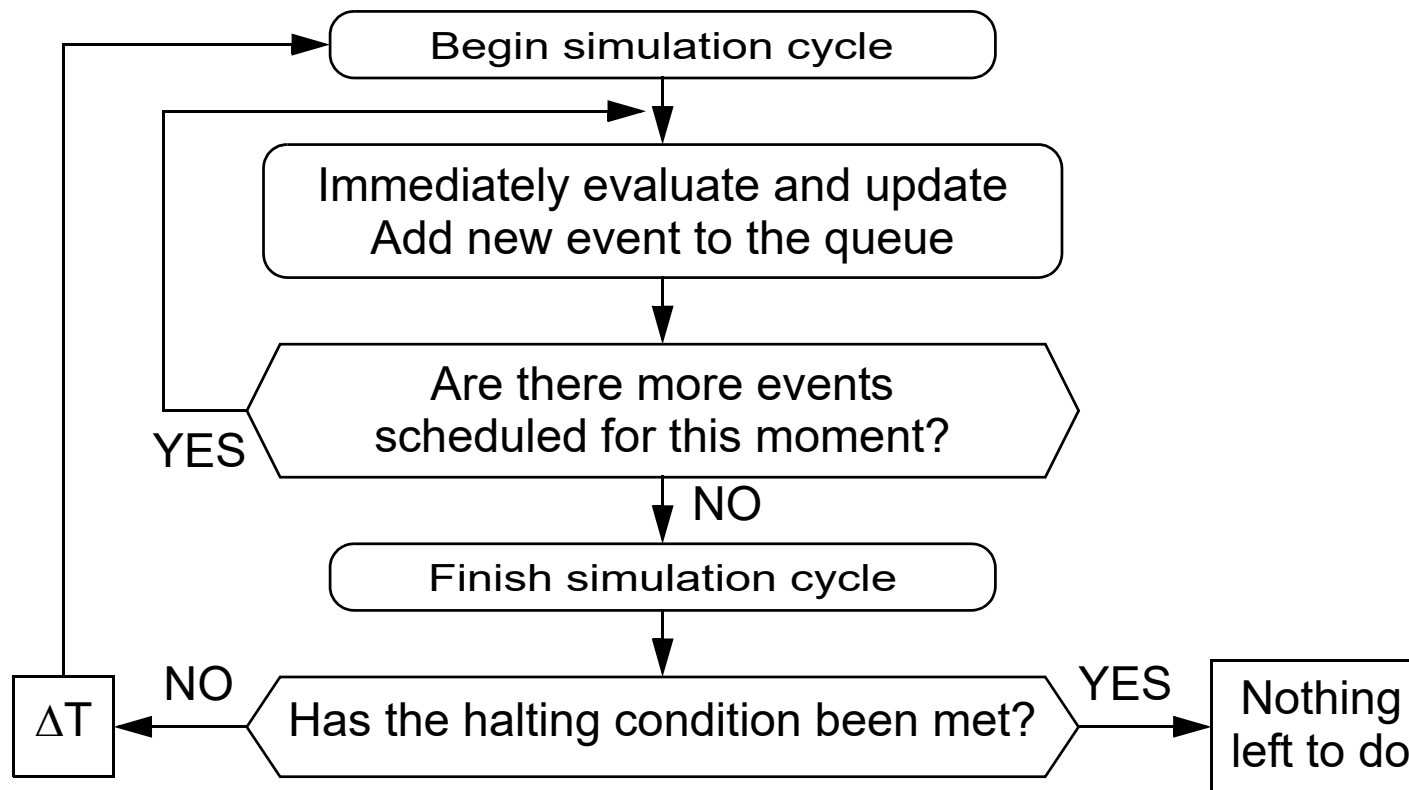
TTÜ1918

- **Functions**

```
function [7:0] func;
  input i1;
  integer i1;
  reg [7:0] rg;
  begin
    rg=i1+2;
    func=rg;
  end
endfunction
```

- **Access:   x = func(n);**

- **A function may not have timing control construct –
  executed in zero simulation time**
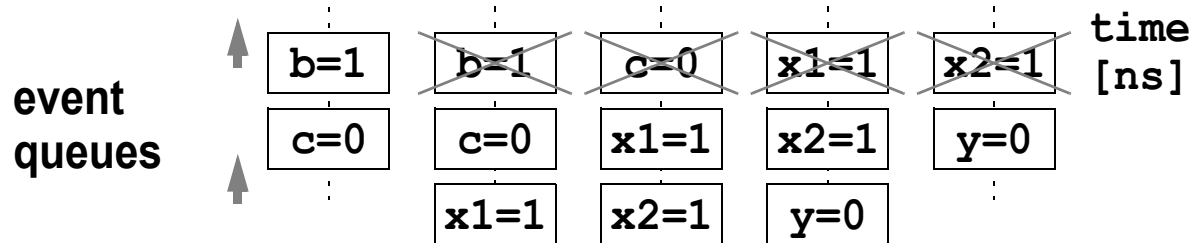
TTÜ1918

# Time and events

## Zero-delay simulation model

```
        ┌──────────────────────────────┐
        │     Begin simulation cycle    │
        └──────────────────────────────┘
                      │
        ┌──────────────────────────────┐
        │  Immediately evaluate and update │
        │    Add new event to the queue   │
        └──────────────────────────────┘
                      │
         Are there more events
  YES    scheduled for this moment?
                      │ NO
        ┌──────────────────────────────┐
        │     Finish simulation cycle    │
        └──────────────────────────────┘
                      │
   NO                                    YES    ┌──────────┐
   ΔT   ◄── Has the halting condition been met? ──►│ Nothing  │
                                                  │ left to do│
                                                  └──────────┘
```
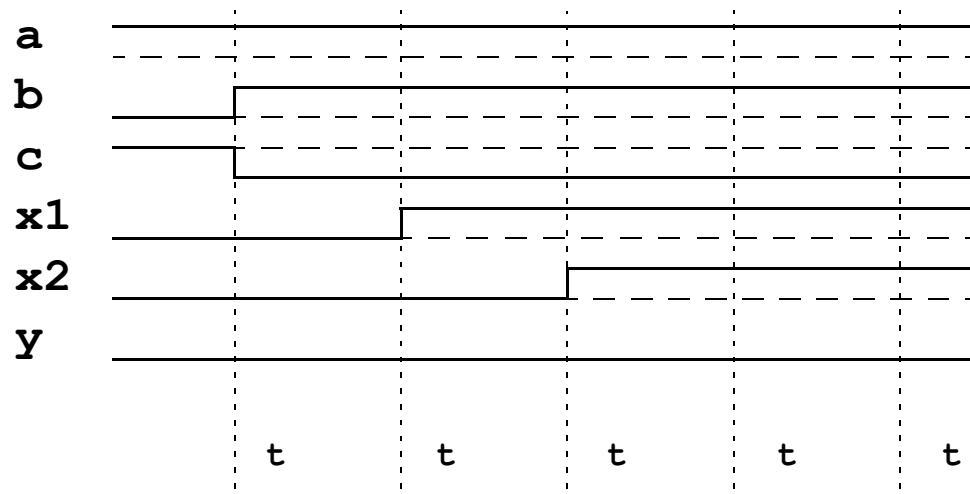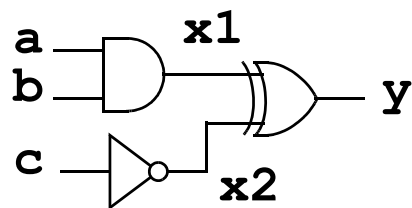
# Zero-delay simulation model (example #1)

```
assign x1 = a & b;
assign x2 = ! c;
assign y  = x1 ^ x2;
```
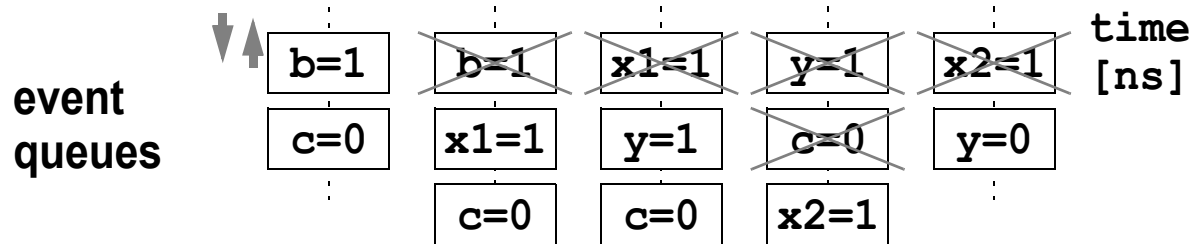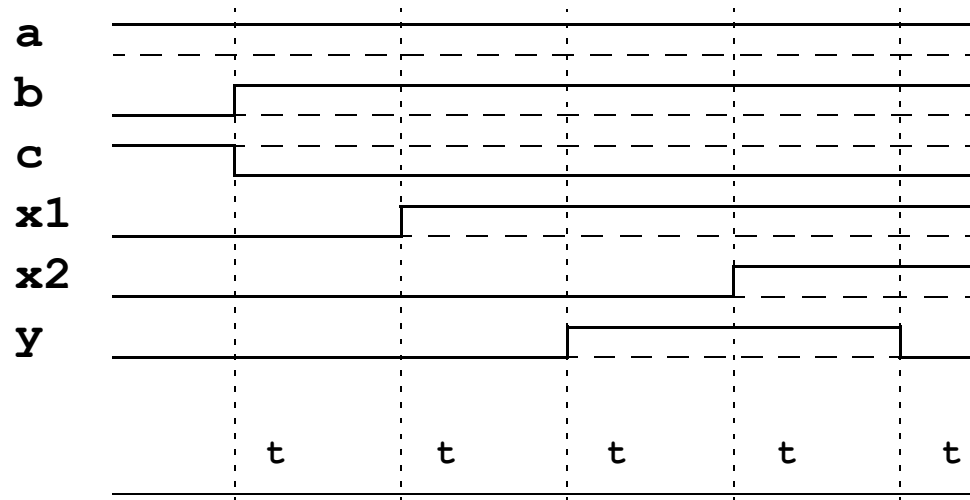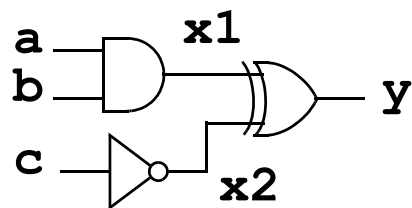
# Zero-delay simulation model (example #2)

```
assign x1 = a & b;
assign x2 = ! c;
assign y  = x1 ^ x2;
```

# Non-deterministic behavior

```
module stupidVerilogTricks (f,a,b);
   input   a, b;
   output f;
   reg     f, q;

   initial    f = 0;

   always @(posedge a)    #10 q = b;

   not ( qBar, q );

   always @q    f = qBar;

endmodule
```

```
q=0
f=qBar=b=1
a=0


a=1
#10  q=1     [b==1]


f==?

1) qBar=0    [q==1]
   f=0

2) f=1       [qBar==1]
   qBar=0    [q==1]
```

TTÜ1918

# Timing control

- **Suspending execution for a fixed time period**

- **Suspending execution until an event occurs**

- **Suspending execution until an expression comes true**
  - **level sensitive event control**

```
#expression
@event-expression
wait ( expression )
```

```
variable [ or variable ... ]
posedge one-bit-variable
negedge one-bit-variable
event-variable
```

```
@(posedge clk) #5 out=in;
```

# Event Control

```
module event_control;
   event e1, e2;
   initial @e1 begin
     $display("I am in the middle.");
     ->e2;
   end
   initial @e2
     $display("I am the last one...");
   initial begin
     $display("I am the first!");
     ->e1;
   end
endmodule
```

```
I am the first!
I am in the middle.
I am the last one...
```

# Timing control inside assignments

```
state = #clk_period next_state;
```
≡
```
temp = next_state;
#clk_period state = temp;
```

```
state = @my_event next_state;
```
≡
```
temp = next_state;
@my_event state = temp;
```

```
always @(s1) #1 wb1  =     s1;
always @(s1)    wb1d = #1 s1;
always @(s1) #3 wb3  =     s1;
always @(s1)    wb3d = #3 s1;
```
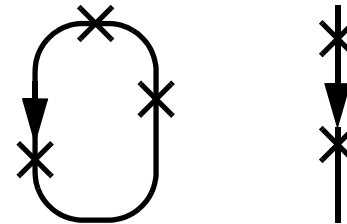
```
always @(s1) #1 wn1  <=     s1;
always @(s1)    wn1d <= #1 s1;
always @(s1) #3 wn3  <=     s1;
always @(s1)    wn3d <= #3 s1;
```
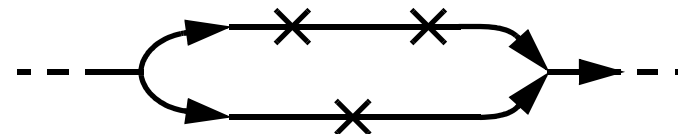
# Parallelism

- **Structural parallelism**

  - **modules**

  - **continuous assignments**
    **(data-flow style)**

  - **behavioral instances**
    **(always & initial blocks)**

- **Behavioral parallelism**

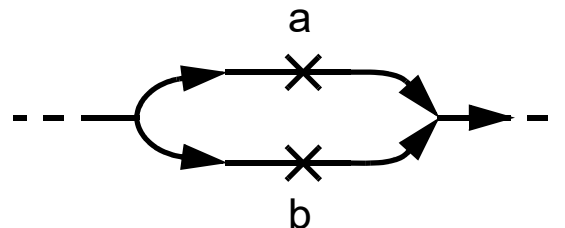  - **fork & join**

  - **disable**

TTÜ1918

# fork & join

```
module fork_join;
   event a, b;
   initial begin
   // ...
   fork
      @a ;
      @b ;
   join
   // ...
   end
endmodule
```
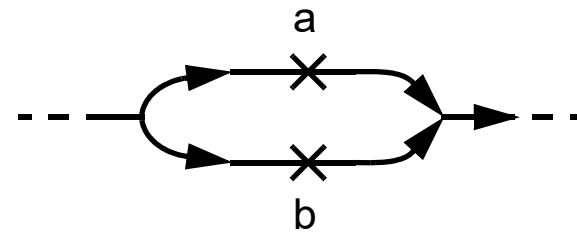
- **continues when both events, *a* and *b*, occur**

# fork & join + disable

```
module fork_join;
  event a, b; // Block name!
  initial begin
    // ...
    fork : block1
      @a disable block1;
      @b disable block1;
    join
    // ...
  end
endmodule
```



- **continues when either *a* or *b* occurs**

# disable

```
begin : break
  for (i=0;i<1000;i=i+1) begin : continue
    if (a[i]==0) disable continue;  // i.e. continue
    if (b[i]==a[i]) disable break;  // i.e. break
    $display("a[",i,"]=",a[i]);
  end
end
```

- **disable <block_name>**
  - **removes the rest of events associated with the block**
  - **named blocks and tasks only**

- **named blocks**
  - **local variables allowed**

TTÜ1918

# Structural and behavioral descriptions

- **Structural – created from lower level modules**

- **Data-flow – combinational logic – keyword `assign`**

- **Behavioral – algorithms etc. – keywords `initial` & `always`**

```
initial begin
  forever
  @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)  zero = 1;
    else           zero = 0;
  end
end
```

$\equiv$

```
always
  @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)  zero = 1;
    else           zero = 0;
  end
```
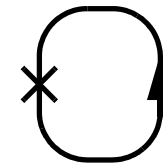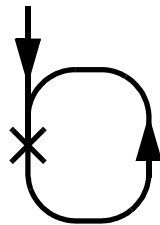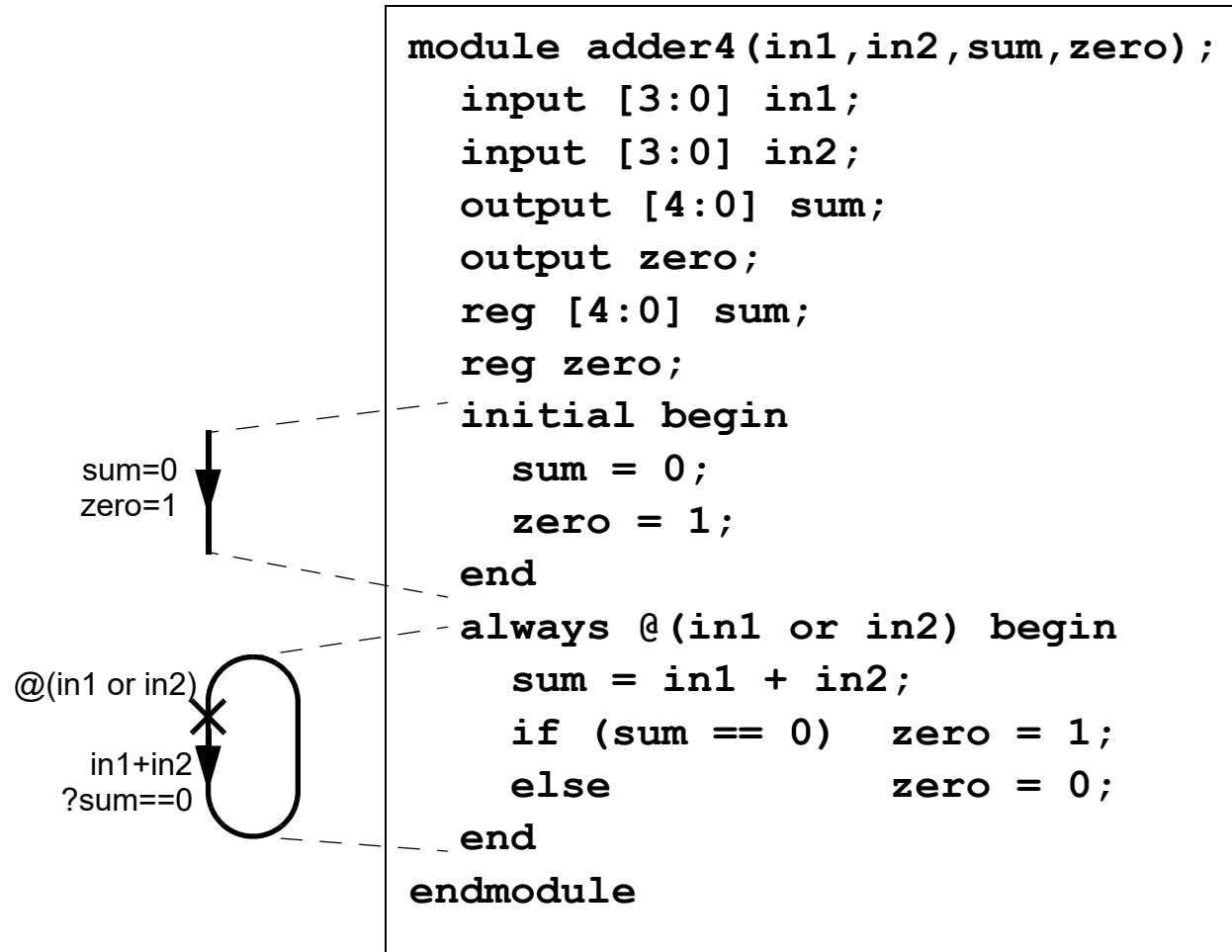
# Behavioral (cont.)

```
module adder4(in1,in2,sum,zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum;
  output zero;
  reg [4:0] sum;
  reg zero;
  initial begin
    sum = 0;
    zero = 1;
  end
  always @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)  zero = 1;
    else           zero = 0;
  end
endmodule
```

sum=0
zero=1

@(in1 or in2)

in1+in2
?sum==0

TTÜ1918

# **Behavioral** (cont.)

@(sum)

?sum==0

sum=0

@(in1 or in2)

in1+in2

```verilog
module adder4(in1,in2,sum,zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] sum;
  output zero;
  reg [4:0] sum;
  assign zero = (sum==0) ? 1 : 0;
  initial sum = 0;
  always @(in1 or in2)
    sum = in1 + in2;
endmodule
```

# Structural
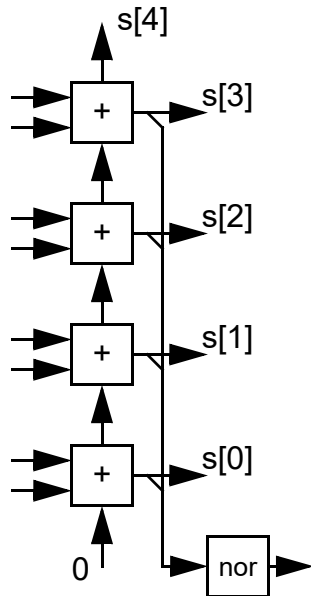
```
module adder4 (in1, in2, s, zero);
   input [3:0] in1;
   input [3:0] in2;
   output [4:0] s;
   output zero;
   fulladd u1 (in1[0],in2[0], 0, s[0],c0);
   fulladd u2 (in1[1],in2[1],c0, s[1],c1);
   fulladd u3 (in1[2],in2[2],c1, s[2],c2);
   fulladd u4 (in1[3],in2[3],c2, s[3],s[4]);
   nor u5 (zero,s[0],s[1],s[2],s[3],s[4]);
endmodule
```
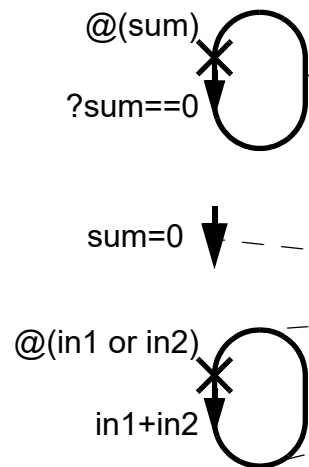
```
module fulladd (in1, in2, cin, sum, cout);
   input in1, in2, cin;
   output sum, cout;
   assign { cout, sum } = in1 + in2 + cin;
endmodule
```

TTÜ1918

# Combined – mixed mode

s[4]

s[3]

s[2]

s[1]
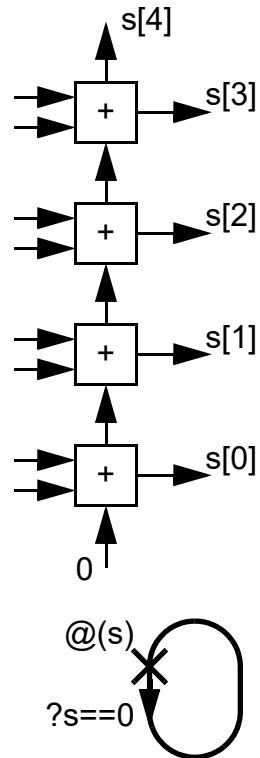
s[0]

0

@(s)

?s==0

```
module adder4 (in1, in2, s, zero);
  input [3:0] in1;
  input [3:0] in2;
  output [4:0] s;
  output zero;  reg zero;
  fulladd u1 (in1[0],in2[0], 0, s[0],c0);
  fulladd u2 (in1[1],in2[1],c0, s[1],c1);
  fulladd u3 (in1[2],in2[2],c1, s[2],c2);
  fulladd u4 (in1[3],in2[3],c2, s[3],s[4]);
  always @(s)
    if (s == 0)  zero = 1;
    else         zero = 0;
endmodule
```

TTÜ1918

# Advanced topics – parameterized modules

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

   assign #(delay)  xout = xin1 ^ xin2;
endmodule
```

```
// 8 bits, delay 10
xorx #(8) (vout,vin1,
    {b0,b1,b2,b3,b4,b5,b6,b7});
```

```
// 4 bits, delay 20
xorx #(4,20) (vout,vin1,
    {b0,b1,b2,b3});
```

# Advanced topics – compiler control

- **'define <macro_label> <replacement>**

- **'ifdef <macro_label>**
   **// code...**
  **'endif**

- **'include** *"verilog-file"*

```
'include "design.def"
 ...
'ifdef DEBUG_MODE     /* Debugging ...  */
  initial #1 begin
    $display("\n Time:  Address   Data");
  end
  always @(clk) begin
    $display("%t:  %h  %h",
       $time,address,data);
  end
'endif
```

TTÜ1918

# Advanced topics – compiler control (cont.)

```
parameter   WORD_SIZE = 32;
`define     WORD    [WORD_SIZE-1:0]
  // ...


reg `WORD address, data;
  // ...
```

```
// <time_unit>/<time_precision>
`timescale 1 ns / 1 ns
```

```
`timescale 10 ns / 0.1 ns
// ...
#7.748; // delay 77.5 ns
```

- **ModelSim SE/PE/XE User's Manual**
- **IEEE Std 1364-1995 compiler directives**

# Advanced topics – memory images

- **$readmemb**
- **$readmemh**

```
    ...
reg [DSIZE-1:0] MEM [0:MAXWORDS-1];
    ...
    $readmemh("PROG.FILE",MEM);
    ...
```

```
@0000      // Hexadecimal address
// Code      _IC_._ _RD.__ __.S1_ _.S2__ .___IMM_____
00000000  // 0000.0 000.00 00.000 0.0000 .0000.0000.0000  add.f   -,0,0
00000000  // 0000.0 000.00 00.000 0.0000 .0000.0000.0000  add.c.f -,0,0
2040007f  // 0010.0 000.01 00.000 0.0000 .0000.0111.1111  add.t   R1,127,127
20820ffd  // 0010.0 000.10 00.001 0.0000 .1111.1111.1101  add.t   R2,R1,-3
60c40000  // 0110.0 000.11 00.010 0.0000 .0000.0000.0000  add.c.t R3,R2,0
20043000  // 0010.0 000.00 00.010 0.0011 .0000.0000.0000  add.t   -,R2,R3
```

TTÜ1918

# Advanced topics – user primitives

```
primitive MUX_4_2 (Y,D0,D1,D2,D3,S1,S2);
  input D0,D1,D2,D3,S1,S2;
  output Y;
  table // D0 D1 D2 D3 S1 S2 : Y
    0 ? ? ? 0 0 : 0 ;
    1 ? ? ? 0 0 : 1 ;
    ? 0 ? ? 0 1 : 0 ;
    ? 1 ? ? 0 1 : 1 ;
    // ...
    ? ? ? 0 1 1 : 0 ;
    ? ? ? 1 1 1 : 1 ;
  endtable
endprimitive
```

# Advanced topics – user primitives (cont.)

- **one bit wide ports**

- `wire` - **combinational**

- `reg` - **sequential**

```
0      logic 0
1      logic 1
x      unknown
?      either 0, 1 or x   (input ports only)
b      either 0 or 1      (input ports only)
-      no change (outputs of sequential primitives)
(xy)   value change x,y=0,1,x,? or b
*      any value change (same as (??))
r      rising edge on input (01)
f      falling edge on input (10)
p      positive edge ((01),(0x) or (x1))
n      negative edge ((10),(1x) or (x0))
```

TTÜ1918

# Advanced topics – more about assignments

- **Behavioral assignments**
  - **assign <assignment>**
    - **reg type only**
  - **deassign <lvalue>**
    - **undoes behavioral assignment**
  - **force <assignment>**
    - **reg & net types**
    - **stronger than assign**
  - **release <lvalue>**
    - **reg & net types**
    - **undoes force statement**

```
<continous_assignment> ::=
    assign [<drive_strength>] [<delay2>] <list_of_net_assignments>;
```

# Advanced topics – more about nets

```
<net_declaration> ::=
    <net_type> [scalared|vectored] [<strength>]
        [<range>] [<delay>] <variable_list>;

<net_type> ::=  wire | tri | wand | wor | triand | trior |
                tri0 | tri1 | supply0 | supply1 | trireg
```

- **wire, tri - no logic function (only difference is in the name)**
- **wand, wor, triand, trior - wired logic (wand==triand, wor==trior)**
- **tri0, tri1 - connections with resistive pull**
- **supply0, supply1 - connections to a power supply**
- **trireg - charge storage on a net**

```
scalared - single bits are accessible (default)
vectored - single bits are not accessible

<range> ::= [ <msb>:<lsb> ]
```

# Advanced topics – more about nets – delays

```
<delay> ::=  #<delay_value> | #(<delay_value>) | <delay2> | <delay3>


<delay2> ::=  #(<delay_value>,<delay_value>)
<delay3> ::=  #(<delay_value>,<delay_value>,<delay_value>)


<delay_value> ::=  <unsigned_number> | <parameter_identifier> |
                   <constant_mintypmax_expression>


<constant_mintypmax_expression> ::=
    <constant_expression>:<constant_expression>:<constant_expression>
```

- **Delays**
  ```
  <delay>
  <rise_delay> <fall_delay>
  <rise_delay> <fall_delay> <turnoff_delay>
  ```

TTÜ1918

# Advanced topics – more about nets – strength

```
<strength> ::=  <charge_strength> | <drive_strength>

<charge_strength> ::= (small) | (medium) | (large)

<drive_strength> ::=  (<zero_strength>,<one_strength>) |
                      (<one_strength>,<zero_strength>)

<zero_strength> ::=  supply0 | strong0 | pull0 | weak0 | highz0

<one_strength>  ::=  supply1 | strong1 | pull1 | weak1 | highz1
```

TTÜ1918

# Advanced topics – more about gates

```
<gate_instantiation> ::=
    <gate_type> [<drive_strength>] [<delay>] [<label>] (<terminals>);
<gate_type> ::=  and | nand | or | nor | xor | xnor |
                 buf | not | bufif0 | bufif1 | notif0 | notif1 |
                 nmos | pmos | rnmos | rpmos |
                 tran | rtran | tranif0 | tranif1 | rtranif0 | rtranif1 |
                 cmos | rcmos | pullup | pulldown
<drive_strength> ::=  (<zero_strength>,<one_strength>) |
                      <one_strength>,<zero_strength>)
```

- **and, nand, or, nor, xor, xnor - simple logic gates (output, input1, input2[,...])**
- **buf, not - simple buffers (output, input)**
- **bufif0, bufif1, notif0, notif1 - three-state drivers (output, data-input, control-input)**
- **nmos, pmos, rnmos, rpmos - transistors (output, data-input, control-input)**
- **tran, rtran - true bidirectional transmission gates (inout1, inout2)**
- **tranif0, tranif1, rtranif0, rtranif1 - true bidirectional transmission gates (io1, io2, control-input)**
- **cmos, rcmos - transmission gates (data-output, data-input, n-channel-control, p-channel-control)**
- **pullup, pulldown - drive strengths (logic-1/logic-0) (output)**
- **r<type> - relatively higher impedance when conducting**

TTÜ1918

# Verilog 2001

*new*

- **ANSI-style ports**

```
module AND2 (i1,i2,z);
  input i1, i2;
  output z;
  assign z = i1 & i2;
endmodule
```

```
module AND2 (
  input i1, i2,
  output z );
  assign z = i1 & i2;
endmodule
```

- **Modified declarations**

```
output x;  reg x;
```

```
always @(a or b)
```

```
output reg x;
```

```
always @(a, b)
always @*
```

- **Multi-dimensional arrays**

```
// 4x2 array of 8-bit words
reg [7:0] arr2 [1:0][3:0];
```

- **Extended macros**

```
`define TEXT(a,b) $display("b a");
`TEXT(WORLD,HELLO)
```

- **Generate statement – if / case**

TTÜ1918

# SystemVerilog – new data types

- **2-state (0,1)**
  - **shortint, int, longint, byte + signed/unsigned – "int unsigned my_data;"**
  - **bit – user-defined vectors**

- **4-state (0,1,X,Z)**
  - **logic – user-defined vectors + signed/unsigned**

- **Real types – "real", "shortreal", "realtime"**

- **Complex data types**
  - **"typedef wire [63:0] my_bus;   my_bus data_bus = 64'b0;"**
  - **"typedef enum (RED,GREEN,BLUE) RGB;   RGB TLC_out = RED;"**

- **Structures & objects – struct / union / class**

```
typedef union packed {
  jump_instr jmp;
  oper_instr opc;
} cpu_instr;
```

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | cnd |  | addr |  |  |  |  |
| 0 | oper |  |  | r1 |  | r2 |  |

```
typedef struct packed {
  bit jmp;
  bit [1:0] cnd;
  bit [4:0] addr;
} jump_instr;
```

```
typedef struct packed {
  bit op0;
  bit [2:0] oper;
  bit [1:0] r1, r2;
} oper_instr;
```

```
class obj;
int a;
task set (int b); a = b; endtask
function int get; return a; endfunction
endclass

obj o1; o1 = new; // obj o1 = new;

initial begin
o1.set(10);
$display("o1.a = %d", o1.get());
end
```

# SystemVerilog – enhanced behavior

- **New operations**

  - **arithmetic – ++, +=, --, -=, ...**

  - **shifts – logical: << / >> ; arithmetic: <<< / >>>**

- **Enhanced loops – "do ... while" loops; break & continue commands**

- **New procedural statements – always_comb, always_latch, always_ff**

```
always @(a or b)    x <= a & b;

always @(posedge clk or negedge res)
  if (res==0) q <= 0;
  else        q <= d;
```

```
always_comb    x <= a & b;

always_ff @(posedge clk, negedge res)
  if (res==0) q <= 0;
  else        q <= d;
```

- **Interfaces**

```
module processor (address, data, clk, res);
  output [3:0] address;  reg [3:0] address;
  input  [3:0] data;     input clk, res;
  ...
endmodule

module memory (address, data);
  output [3:0] address;
  output [3:0] data;  reg [3:0] data;
  ...
endmodule
```

```
interface cpu_bus;
  logic [3:0] address, data;
endinterface : cpu_bus

module processor ( cpu_bus buscpu;
                   input bit clk, res );
  ...
  buscpu.address <= ...
endmodule

module memory ( cpu_bus busmem ); ...
```

# SystemVerilog – assertions

- **Two kinds – immediate (assert) & concurrent (assert property)**
  - **cover property & assume property == assert property**
- **Levels – $fatal, $error, $warning, $info ($display is also allowed)**
- **Immediate assertions**
  - **assert ( data_is == data_expected )   [ $display("OK") ] [ else $warning("Wrong?") ] ;**
  - **"begin ... end" statement blocks are allowed**
- **Concurrent assertions**
  - **assert property ( ! ( read_bus && write_bus ) );**   – error when both are true
  - **assert property ( @(posedge clk) rdy |-> ##[1:2] ack );**   – after "rdy" (at clock tick), "ack" is expected to be true at the next or at the next after (or both)
- **Implications – expected sequences of signals**
  - **overlapped: "s1 |-> s2" – if sequence "s1" matches, "s2" must also match**
  - **non-overlapped: "s1 |=> s2" – "s2" is evaluated on the next clock tick**
- **Complex properties – combining sequences and properties**

```
sequence Ready        property HndShk;
   rdy                   @(posedge clk)
endsequence                Ready |-> Ack;
sequence Ack          endproperty
   ##[1:2] ack
endsequence           assert property (HndShk);
```

```
"x ##1 y [*3] ##1 z" ==
   "x ##1 y ##1 y ##1 y ##1 z"

"s1 and s2" - both sequences succeeded
"s1 or s2" - one sequence succeeded
...
```

TTÜ1918

# SystemVerilog vs. VHDL ?

## or VHDL vs. SystemVerilog ?

| Feature | Verilog | SystemVerilog | VHDL |
|---|---|---|---|
| Type declaration | weak | weak (improved) | strong |
| User defined types | -   (macros) | ++ | ++ |
| User defined operators | -- | - (methods) | ++ |
| Archives / libraries | -   (simulator) | library, include, config | library+use, configuration |
| Reusability | -   (include) | +   (include, config) | ++   (package) |
| Pre-compilation | +   (limited macros) | ++   (macros) | -   (alias) |
| Flexibility of constructions | @ operation level (gates) | @ data & operation levels | @ data level   (attributes) |
| Usability & Synthesizability | RTL & lower level(s) | RTL & high level & verification | RTL & higher levels |
| Standardization | +   (simulator, now IEEE) | ++   (IEEE) | ++   (IEEE) |
| Programming language | C   (K&R) | C   (ANSI) | Ada   (OO Pascal) |