HDL Simulation & Verilog Modeling

Stefano Spellini

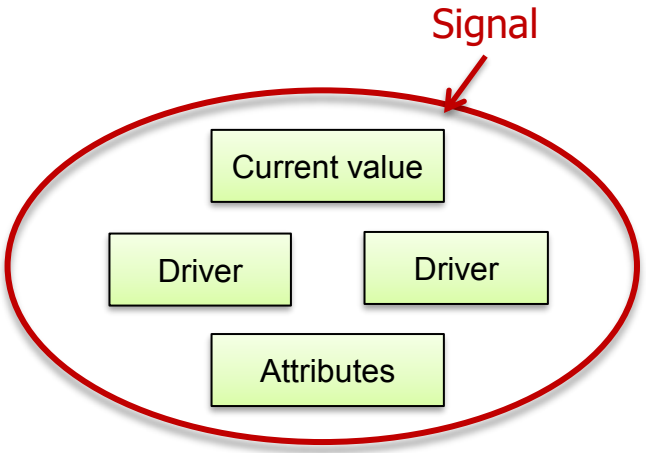
HDL SIMULATION

Signals

- Signals
  - Used to connect submodules in a design
  - Interfacing two or more entities
    - They provide the channels of communication
  - Declaration
    - Explicitly declare the signal using a signal declaration
    - Declare the signal as a port
  - Assignments to signals
    - Sequential or concurrent statements
      - E.g., test_signal <= '1' AFTER 25 ns, '0' AFTER 30 ns;
- Drivers → they handle the signals in the system
  - For each signal, the simulator creates a driver which is a source for the value of a signal
    - Each time the signal assignment statement executes, the value of the waveform element is appended to the driver when the time you designate arrives
  - Item that is read by the system to determine the new value for the signal

Signals (cont.d)

- Drivers
  - A signal may have more than one driver ⇒ tipically its only has ONE driver
    - Drivers collectively determine the value of the signal
    - Specify what happens to the signal by specifying a resolution
      - Resolution function → decides which value to assign at the signal
- Attribute information
  - Determine information about a signal
    - Last time an event took place
    - Last value of the signal
  - Some signal attributes are actually another signal, while others return a value



Signal assignments

- Sequential signal assignment
  - Schedules a waveform element to be assigned to a signal's driver after a time you specify
  - If you do not specify a time, a value of zero nanoseconds is used as the delay
    - Updated at the beginning of the next simulator iteration
  - If you have several assignments, the signal assignments are evaluated sequentially
- Current value for data_out is a '0'
- Current evaluation has scheduled data_out to be updated to a '1'
- The evaluation of the result signal uses the value for data_out within the current iteration
  - In this example is a '0'

```
data_out <= t AND d;
--Current value a '0', new value a '1'

result <= data_out AND carry_bit;
```

Delays

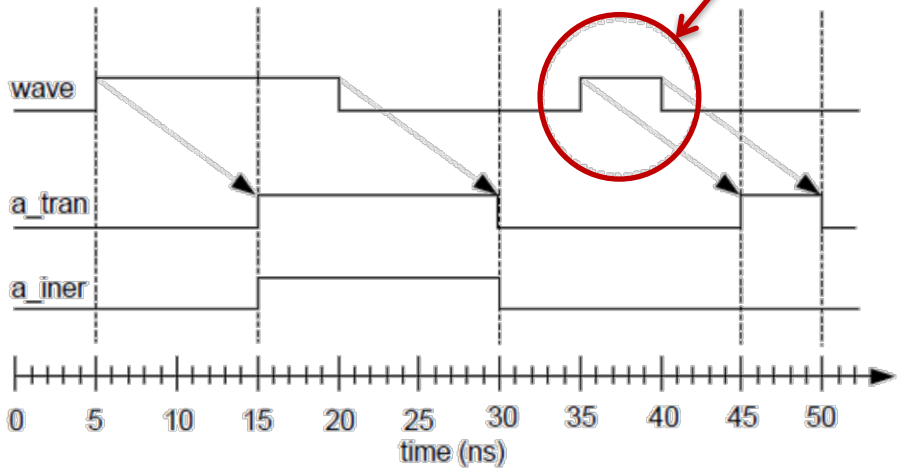
- Transport delay
  - ‘transport’ keyword
  - Any pulse is transmitted to the signal name you specify, with no regard to how short the pulse width or duration
    - Model pure delay, without any consideration of the real, physical behaviour, at high levels of abstraction
    - All input changes pass to the output after the delay time you specify

- Inertial delay
  - Default delay
  - Pulses with a width shorter than the delay time you specify in the waveform element are not transmitted to the signal name you specify
    - Model real, physical behaviour where a rise or fall time of the output is equivalent to the delay time
    - The output begins to react to the input change. However, the new value is not recognized until after the delay time.

Delays (cont.d)

Propagates as transport
Not propagated as inertial
Shorter than 10ns

```
a_tran <= TRANSPORT wave AFTER 10 ns;
a_iner <= wave AFTER 10 ns;
```



Delays (cont.d)

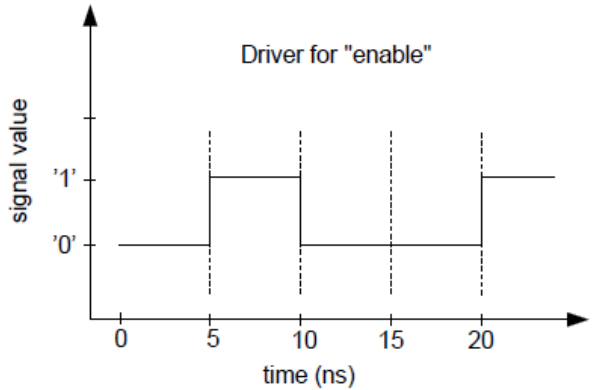
- Delta delay
  - Infinitesimal value of time that is greater than zero
    - One simulation iteration
  - Necessary when you make a signal assignment with no delay expression

- Simulation of delta delays
  - At the beginning of the iteration, any pending events for the given time mature
  - When the event maturity triggers a process that is sensitive to the signal change, the process is evaluated
  - When a signal assignment that has a 0 ns delay is found, the iteration increments and the preceding steps repeat
  - The simulation then advances by one simulator time step.

```
pure_delay <= a + b;

pure_delay <= a + b AFTER 0 ns;
```

Signal drivers

- Driver
  - Container for the projected output waveform that the system creates
  - The value of a signal is related to the current values of its drivers
- Transaction
  - Signal value and time for the transaction to occur



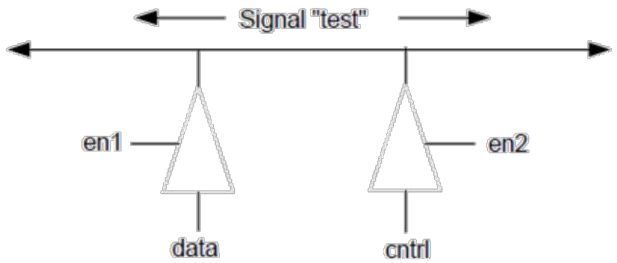
Value	Time
'0'	0ns
'1'	5ns
'0'	10ns
'1'	20ns

```
enable <= '0','1' AFTER 5
ns,
          '0' AFTER 10ns,
          '1' AFTER 20 ns;
```

Resolution function

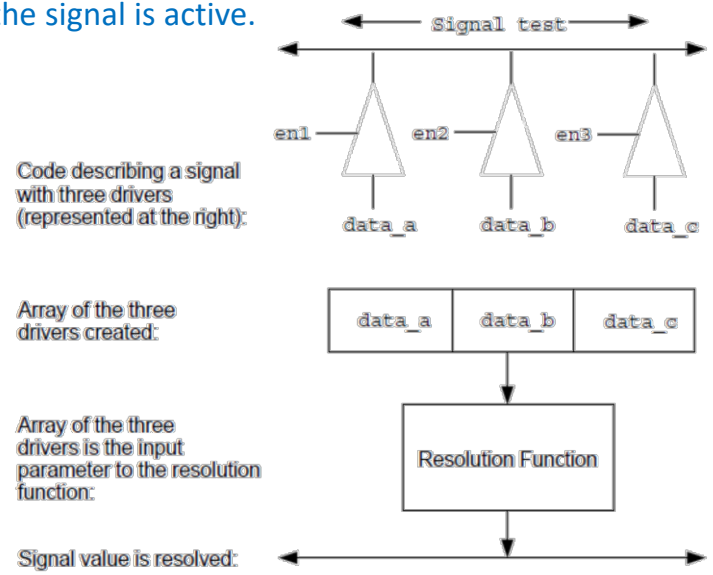
- Buses are used to connect a number of output drivers to a common signal
  - Tri-state drivers
    - At most one driver may be active at a time, and it determines the signal value

```
SIGNAL test : wired_or bit BUS;  -- signal declaration

--  partial code descriptions of circuit with signal "test"

test <= GUARDED data AFTER
        2 ns WHEN en1 = '1'
      ELSE
        test AFTER 3 ns;

test <= GUARDED cntrl AFTER
        2 ns WHEN en2 = '1'
      ELSE
        test AFTER 3 ns;
```



Resolution function (cont.d)

- Resolution function
  - Takes the values of all the drivers contributing to a signal, and combines them to determine the final signal value
  - Subprogram that defines what single value the signal should have when there are multiple drivers for that signal
    - Array that contains all the driver values for the signal
    - Called every time the signal is active.



VERILOG MODELING

What is Verilog?

- IEEE industry standard HDL – used to describe a digital system
- Used in both **hardware simulation & synthesis**

- Introduced in 1984 by Gateway Design Automation
- Latest stable release: IEEE 1364-2005 / 9 November 2005
- In 2009, the Verilog standard (IEEE 1364-2005) was merged into the SystemVerilog standard
  - Verilog is officially part of the SystemVerilog language

Module and Port Declaration

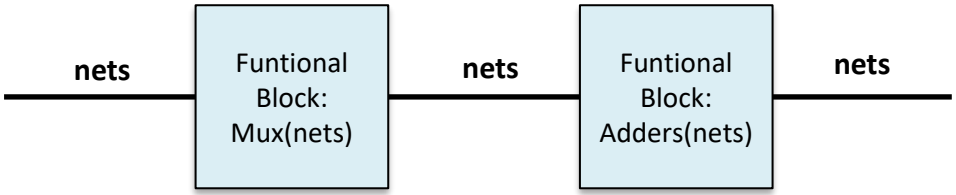
- **Module**: basic modeling structure
- Case-sensitive

- Typical HDL Ports
  - In
  - Out
  - Inout
- **Module and port declarations can be combined or separated**

```
module pippo(
    input clk,
    input[7:0] ina, inb,
    output[15:0] out
);
…
endmodule
```

Data Types

- **Net data type**
  - Represents physical interconnect between structures (activity flows)



- **Variable data type**
  - Represents element to store data temporarily

Net Data Type

Type	Definition
wire	Represents a node or connection
tri	Represents a tri-state node
supply0 **supply1**	Constant logic 0 Constant logic 1

- Bus Declarations:
  - <data_type> [MSB : LSB] <signal name>;
  - <data_type> [LSB : MSB] <signal name>;
- Examples:
  - `wire [7 : 0] out;`
  - `tri enable;`

Variable Data Types

- Allowed types:
  - **reg** (signed): Signed/unsigned variable of any bit size
  - **integer**: signed 32-bit variable
  - **real, time, realtime:** ***no synthesis support***
- Can be assigned only within a procedure, task or function
- Bus Declarations:
  - `reg [MSB:LSB] <signal_name>;`
  - `reg [LSB:MSB] <signal_name>;`
- Examples:
  - `reg [7:0] out;`
  - `integer count;`

Root - VHDL vs Verilog

VHDL

```
entity root is
   port (clk, rst   : in  bit;
   din      : in  UNSIGNED (SIZE-1 DOWNTO 0);
   din_rdy  : in  bit;
   dout     : out UNSIGNED (SIZE-1 DOWNTO 0);
   dout_rdy : out  bit
) ;

end root;
architecture root of root is
  subtype status_t is integer range 0 to 5;
  subtype internal_t is UNSIGNED (SIZE-1 DOWNTO 0);
  signal STATUS: status_t;
  signal NEXT_STATUS: status_t;
  signal Root: internal_t;
  signal Number: internal_t;
  signal Counter: internal_t;
  signal Rem_s: internal_t;
```

Verilog

```
module root(
    input clk,
    input rst,
    input din_rdy,
    input din,
    output dout,
    output dout_rdy
);

wire clk,rst,din_rdy,dout_rdy;
reg done;
wire[31:0] din,dout;
reg[31:0] Root,Counter,Rem_s,Number;
reg[2:0]  STATUS,NEXT_STATUS,prova;
```

Parameter

- Value assigned to a symbolic name (**constant**)
- Must resolve to a constant at compile time
- Can be overwritten at compile time
- **`localparam`:** same as **`parameter`** but cannot be overwritten

```
parameter size=8;
localparam outsize=16;

reg[size-1:0] data;
reg[outsize-1:0] out;

//Old style example
```

```
module pippo
  #(parameter size = 8)
  (…);

//New style example
```

Verilog Timescale

- Simulation depends on how time is defined
- The `timescale compiler directive specifies
  - **Time unit** -> measurement of delays and simulation time
  - **Precision** -> specifies how delay values are rounded before simulation

```
`timescale <time_unit>/time_precision>

// Example
`timescale 10ns/1ns

reg a;
...
#1 a=0; //After 1 time_unit (10ns) assign 0 to a
#5 a=1; //After 5 time_unit (50ns) assign 1 to a
```

Assigning Values - Numbers

- **Sized** or **unsized:** <size>'<base_format><number>
  - **Sized:** `3'b010` = 3-bit wide binary number
    - The prefix (3) indicates the size of number
  - **Unisized:** `123` = 32-bit wide decimal number by default
    - No specified <base_format> defaults to **decimal**
    - No specified <size> defaults to **32-bit** wide number
- **Negative numbers**, specified by putting a minus sign before the <size>
  - **Legal:** `-8'd3`
  - **Illegal:** `4'd-2`

Continuous Assignment Statements

- Model the **behavior** of **Combinational Logic** by using expressions and operators
- Left-hand side (**LHS**) must be a net data type
- Right-hand side (**RHS**) can be net, register or a function call
- **Executed continuously**
  - When one of RHS operands changes, expression is evaluated and LHS updated
- **Delay** values can be assigned to model gate delays

```
wire[15:0] adder_out;
assign adder_out = mult_out + out;

assign #5 adder_out = mult_out + out;
```

Procedural Assignment Blocks

- **`initial`**
  - Used to initialize behavioral statements for simulation
- **`always`**
  - Used to describe the circuit functionality using behavioral statements
- Each **`always`** and **`initial`** block represents a separate process
- Processes run in parallel and start at simulation time 0
- Statements inside a process execute sequentially
- **`always`** and **`initial`** blocks cannot be nested

Initial Block

- Consists of Behavioral statements
  - Starts at time 0
  - Executes only once during simulation
  - Does not execute again
- **`initial`** block executes only once, but the duration may be infinite
- Example uses
  - Initialization
  - Monitoring
  - Any functionality that needs to be turned on just once

Always Block

- Consists of Behavioral statements
  - Executes concurrently starting at time 0
  - Continuously in a looping fashion

- Example uses
  - Modeling a digital circuit
  - Any process or functionality that needs to be executed continuously

Always Block - Example

```
module clk_gen
   #(parameter period = 50)
(
   output reg clk
);

   initial clk = 1'b0;

   always
      #(period/2) clk = ~clk;

   initial #100 $finish;

endmodule
```

ogni 25ns scambia il clk da 0 a 1 e viceversa

⇒ La simulazione finisce a 100 ns

Time	Statement(s) Executed
0	clk = 1'b0;
25	clk = 1'b1;
50	clk = 1'b0;
75	clk = 1'b1;
100	$finish;

Root – VHDL vs Verilog

VHDL

```
process(STATUS, din_rdy, din, Counter)
begin
    case STATUS is
      when Reset_ST =>
        NEXT_STATUS<=ST_0;
          when ST_0 =>
            if din_rdy = '1' then
              NEXT_STATUS<=ST_1;
            else
              NEXT_STATUS<=ST_0;
            end if;
          when ST_1   =>
            NEXT_STATUS<=ST_2;
          when ST_2   =>
            if Counter<16 then
              NEXT_STATUS<=ST_3;
               else
                 NEXT_STATUS<=ST_4;
               end if;

…
```

Verilog

```
always@(STATUS,din_rdy,din,Counter)
begin
    case (STATUS)
        Reset_ST: begin
            NEXT_STATUS <= ST_0;
        end
        ST_0: begin
            if (din_rdy == 1'b1)
                NEXT_STATUS <= ST_1;
            else
                NEXT_STATUS <= ST_0;
        end
        ST_1: begin
            NEXT_STATUS <= ST_2;
        end
        ST_2: begin
            if (Counter < 32'd16)
                NEXT_STATUS <= ST_3;
            else
                NEXT_STATUS <= ST_4;
        end
```

Types of Procedural Assignments

- Blocking assignment (=):
  - Executed in the order they are specified in a sequential block
  - **Combinational logic**
- Nonblocking Assignment (<=)
  - Allows scheduling of assignment without blocking execution of the following statements
  - **Sequential logic**

- Reside inside of procedural blocks
- Update values of **`reg`**, **`integer`**, **`real`**, **`time`** or **`realtime`** variables

Blocking vs. Nonblocking Assignments

Blocking (=)

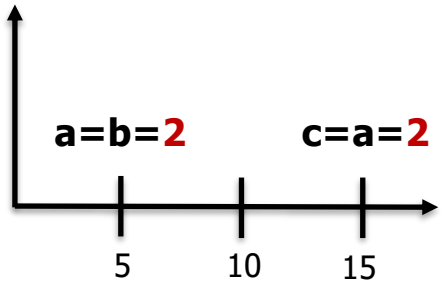
```
initial
   begin
      a = #5 b;
      c = #10 a;
   end
```

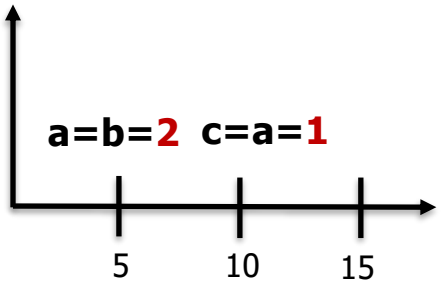
Nonblocking (<=)

```
initial
   begin
      a <= #5 b;
      c <= #10 a;
   end
```

The evaluation of the expressions is done at the same time!

Assuming initially a=1 and b=2





Behavioral Statements

- Must be inside a procedural block (initial or always)

- Behavioral statements:
  - **If-else** statement
    - Conditions are evaluated in order from top to bottom
    - Prioritization
  - **Case** statement
    - Conditions are evaluated at once
    - No Prioritization
  - **Loop** statements

Verilog Functions and Tasks

- Functions and Tasks are subprograms
- Consist of behavioral statements (like procedural block)
- Defined within a module
- Function
  - Return a value based on its inputs
  - Produces combinational logic
  - Used in expressions

```
assign add_out = add(ina, inb);
```

- Tasks
  - Like procedures in other languages
  - Can be combinational or registered
  - Tasks are invoked as statement

```
sim_out(nxt, first, sel, filter);
```

Functions vs. Tasks

Functions	Tasks

• **Always execute in zero time**
  ○ **Cannot pause their execution**
  ○ **Cannot contain any delay, event or timing control statement**
• Must have at least one input argument
  ○ Inputs may not be affected by function
• Arguments may not be outputs and inouts
• Always return a single value
• May call another function but not a task
	• **May execute in non-zero simulation time**
	  ○ **May contain delay, event, or timing control statements**
	• May have zero or more inputs, outputs or inout arguments
	• Modify zero or more values
	• May call functions or other tasks

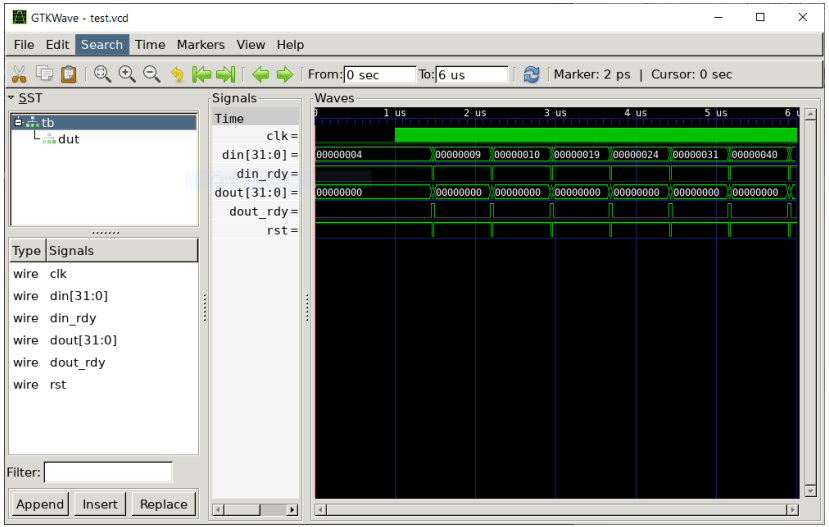
Simulation Into Action

Compile and Simulate the Design

- Download and extract **04_sources.tar.gz** from the moodle
  - **$** tar xzfv 04_sources.tar.gz
  - $ cd 04_sources && ls -> **HDL_simulation** <- Verilog_modeling
- Compile and simulate a design in Vivado
  - Same steps of last lesson
  - **HDL_simulation designs are in VHDL, so set the program accordingly**
- Save and export the .vcd file
  - In the TCL console type
    - open_vcd <namefile.vcd>
    - log_vcd [get_object /<toplevel_testbench/uut/*>]
      - E.g., log_vcd * (to log all signals)
    - run *ns
    - close_vcd
- The namefile.vcd will be in the Vivado project directory
  - <project_folder>/<project_name>.sim/sim_*n*/behav/xsim/namefile.vcd

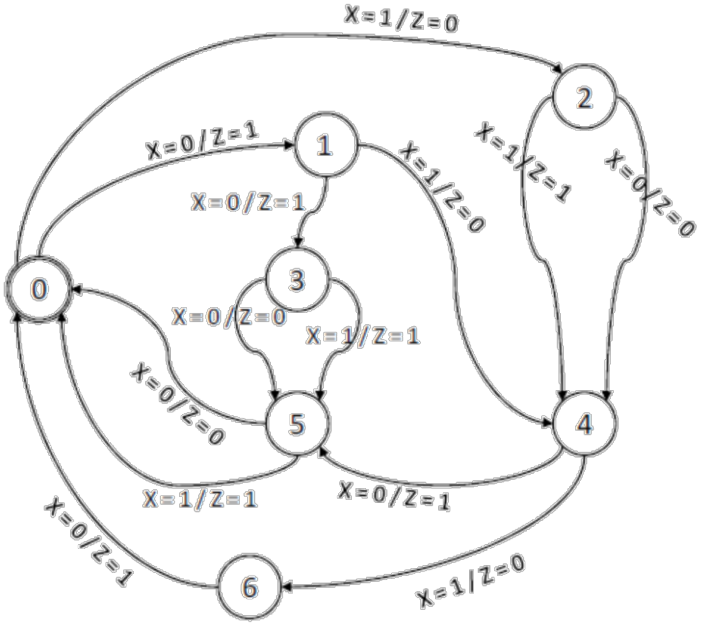
GTKWave

- Launch GTKWave and inspect the waveform
  - $ gtkwave namefile.vcd
    - You can open multiple tabs or windows to simplify the comparison between a tracefile and another



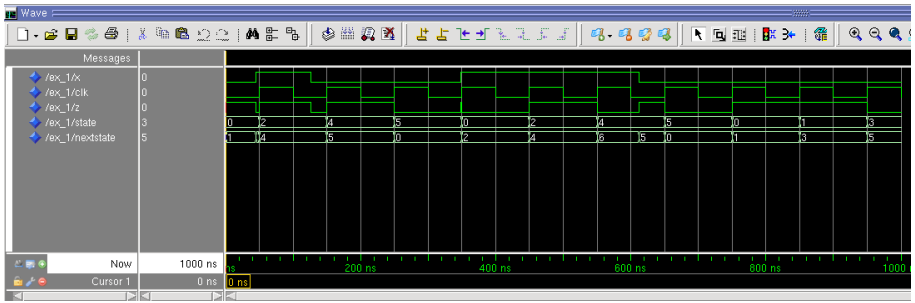
Exercise 1

- **Inside the HDL_simulation folder**
- ex_1/ex_1.vhd
  - Simulate the design under different conditions

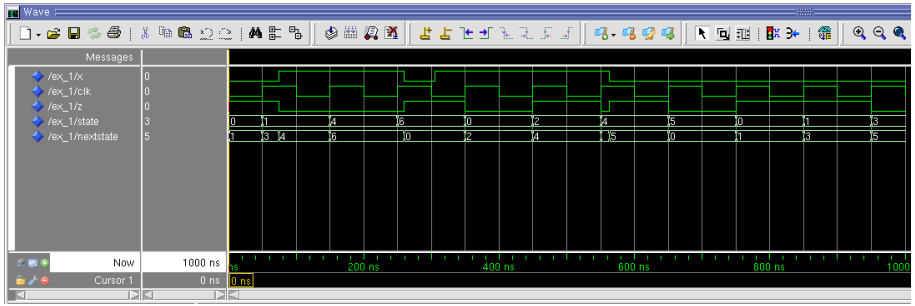


Exercise 1 (cont.d)

- x changes on the rising edge of the clock (stimuli_ex1_ex2/stimuli_1.do);

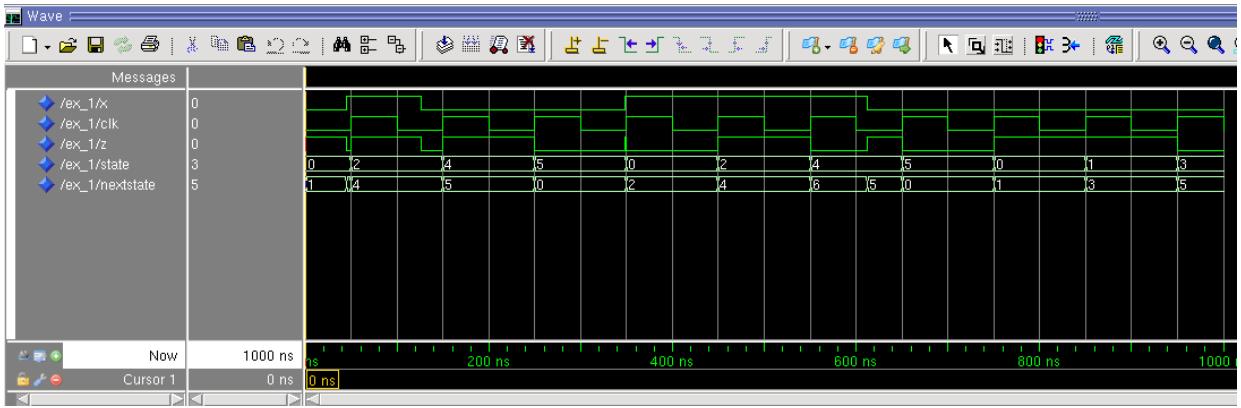


- x changes after the rising edge of the clock (stimuli_ex1_ex2/stimuli_2.do);



Exercise 1 (cont.d)

- x changes before the rising edge of the clock (stimuli_ex1_ex2/stimuli_3.do).



- Compare the waveforms and explain the different behaviours

Exercise 2

- The ex_2/ex_2.vhd
  - Same finite state machine as the previous example with a different coding style
    - ex_1 uses two processes
    - ex_2 uses one process

- Compare the two designs – do you get the same results? Consider the following cases:
  - x changes on the rising edge of the clock
  - x changes after the rising edge of the clock
  - x changes before the rising edge of the clock

Exercise 3

- Consider the following alternatives to the ex_2 finite state machine
  - Simulate all the finite state machines obtained by substituting the original code with one of the proposed alternatives
  - Compare the behaviour with the original finite state machine and explain the differences
- For example, what happens in case a) if x changes while the finite state machine is waiting 5 ns?

Original code:

```
if (clk'event and clk='1') then
   state <= nextstate;
   wait for 0 ns;
end if;
```

Version a)

```
if (clk'event and clk='1') then
   state <= nextstate after 5 ns;
   wait for 5 ns;
end if;
```

Version b)

```
if (clk'event and clk='1') then
   state <= nextstate;
   wait for 5 ns;
end if;
```

Version c)

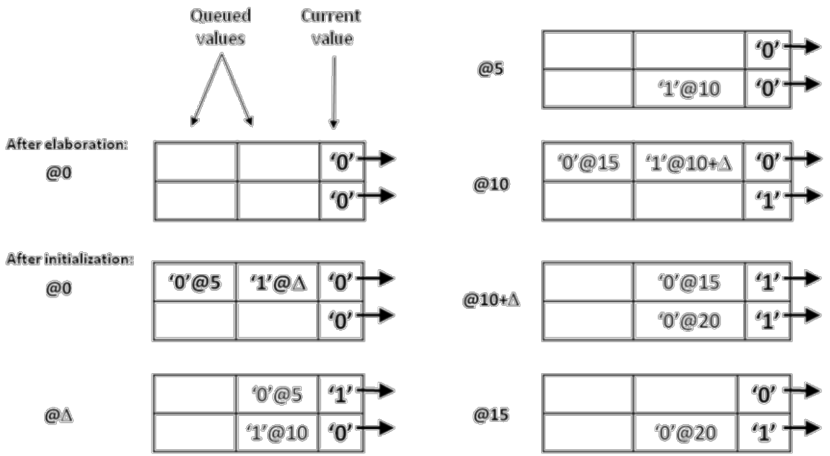
```
if (clk'event and clk='1') then
   state <= nextstate after 5 ns;
   wait for 0 ns;
end if;
```

Version d)

```
if (clk'event and clk='1') then
   state <= nextstate after 5 ns;
end if;
```

Exercise 4

- example_1/example_1.vhd
  - Compile and load the example_1 design:




```
entity example_1 is
end example_1;

architecture behav of example_1 is
signal A,B: bit;
begin

-- P1
P1: process(B)
begin
  A <= '1';
  A <= transport '0' after 5 ns;
end process P1;

-- P2
P2: process(A)
begin
  if A = '1' then B <= not B after 10 ns;
  end if;
end process P2;
end behav;
```

Exercise 5

- ex_3/ex_3.vhdl
  - Assume that signal D assumes value 1 at time 5 ns
  - Trace the simulation flow by drawing the signal drivers for all the design signals
  - Continue for at least 20 ns, or until you get a cyclic behaviour or none of the signal changes

```
C <= A;
A <= B or D;

P1: process (A)
  begin
    B <= A;
end process P1;

P2: process
begin
    wait until A = '1';
    wait for 0 ns;
    E <= B;
    D <= '0';
    F <= E;
end process P2;
```

Exercise 6

- Build the simulation flow by drawing the signal driver for all the signals
  - Signals A, B, C and D have value 0 at time 10 ns
  - If E starts with value 0 and changes to 1 at time 10 ns, what is the design evolution?

```
p1: process
begin
   wait on E;
   A <= 1 after 5 ns;
   B <= A +1;
   C <= B after 10 ns;
   wait for 0 ns;
   D <= B after 3 ns;
   A <= A + 5 after 15 ns;
   B <= B + 7;
end process p1;
```

VERILOG MODELING IN ACTION

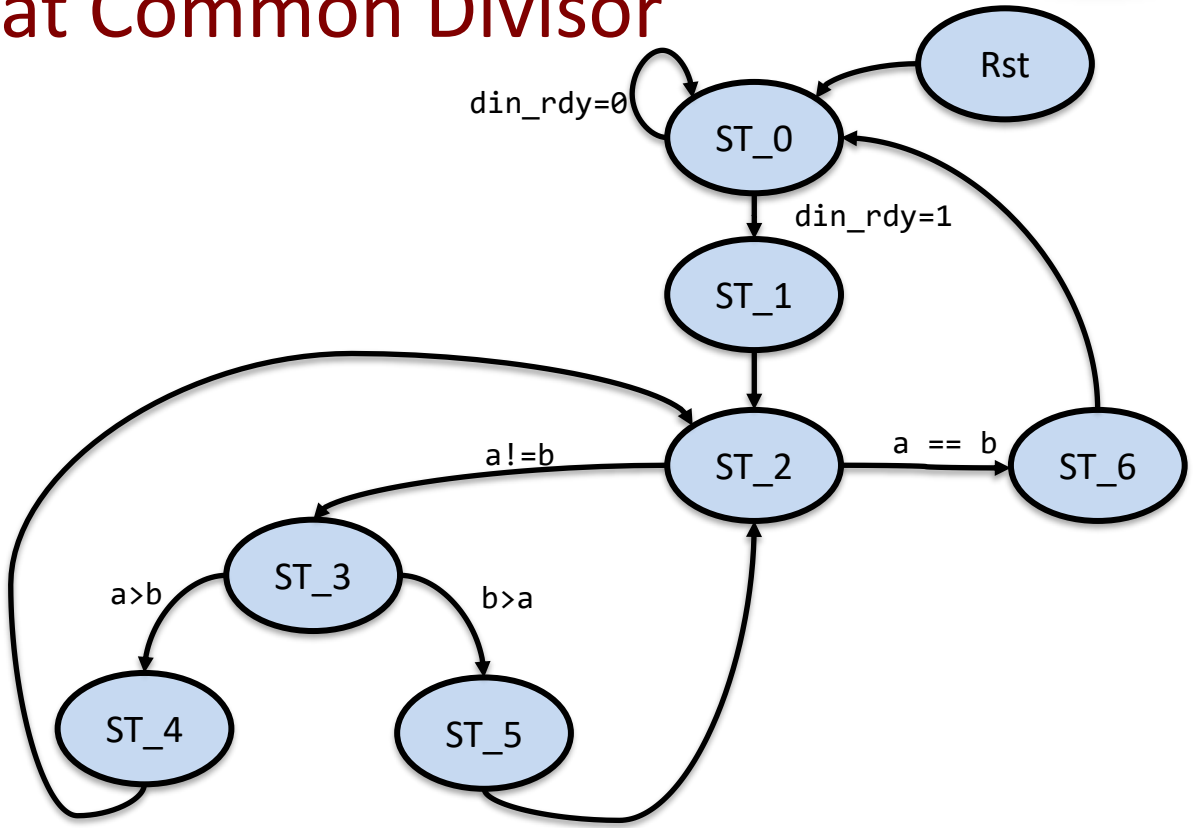
Verilog Modeling

- Switch to the Verilog_modeling folder
- Play with the (familiar) designs
- To compile and simulate a design
  - Always same steps
  - **Remember to change the design language to Verilog**

Great Common Divisor

```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b;
        else
            b := b − a;
    return a;
```

- **Complete the implementation of the GCD design**, inside the sources provided for this lecture



Lecture Assignment

- Write your Fixed-point Multiplier in Verilog
  - To stimulate the design, write a stimuli.tcl script
- Constraints:
  - The used data-types have to be HW datatypes for synthesis
- For the report
  - Justify the design choices you have made
  - Comment the produced simulation waveforms