VHDL
Modeling Behavior from Simulation Perspective

EL 310

Erkay Savaş
Sabancı University

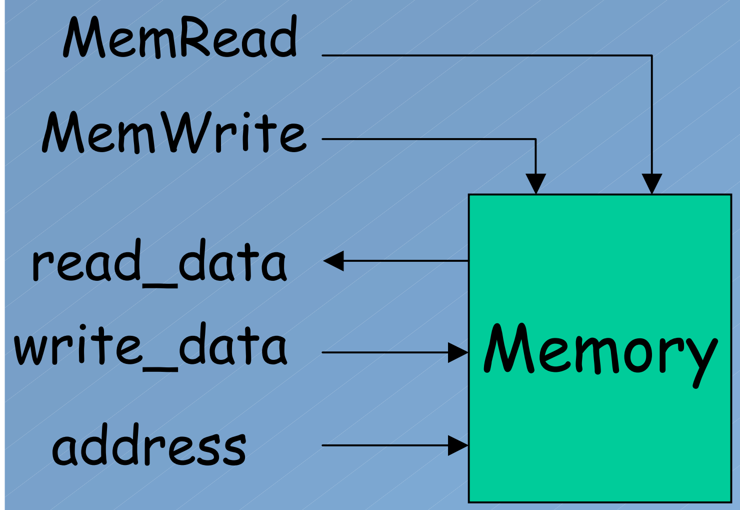
Motivation

- Previously
  - we modeled the digital components as delay elements and their internal behavior is describe using CSA statements
  - This approach works well when modeling digital systems at gate level.
- Large and complex systems cannot always be built at gate level.
  - The goal is to hide the unnecessary details while capturing and preserving the external behavior.
  - More abstraction
  - More powerful statements than CSA statements
- <u>Process</u> construct

Modeling Complex Behavior

- CSA statements are good for simple operations
  - logical operations, simple arithmetic operations etc.
- Most digital components feature more complex behaviors
  - It is not feasible or possible to capture the behavior of complex digital systems such as CPU, memory, communication protocols with CSA statements.
  - Event model can still be used
  - Large and complex systems utilize state information
  - They incorporate complex data structures

Memory as a Complex Digital Component



- Event model is still valid
  - events on input address, data or control lines produce events that can cause the memory model to be executed.
  - Memory access for read an write operations can be assigned for propagation delays.
  - How about internal behavior of memory

Memory

- Problems
  - How can we represent the memory words?
  - How can we address the correct word given the values of address lines?
  - How can memory write operations store values to be accessed by subsequent memory read operations?
- If VHDL had conventional <u>sequential</u> programming language construct these problems would be easily overcome.
  - Concurrent behavior is sometimes not appropriate.
  - Array construct, address as an index to this array
  - Depending on the value of control signals, we can decide whether the array element is to be read or written

The Process Statement

- Sequential behavior of digital systems is best modeled using the process statement
  - 8x32-bits memory

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all; -- we need this package for
                                 1164 related functions

entity memory is
port(address: in unsigned(31 downto 0); -- use unsigned for
                                          memory address
     write_data: in std_logic_vector(31 downto 0);
     Mem Read, MemWrite: in std_logic;
     read_data: out std_logic_vector(31 downto 0));
end entity memory;

architecture behavioral of memory is

..

end architecture behavioral;
```

The Process Statement for Memory

Declarative region of a process

```
...
architecture behavioral of memory is
type mem_array is array(0 to 7) of std_logic_vector(31 downto 0);
                     -- define a new type, for memory arrays
begin  -- begin for architecture
mem_process: process (address, write_data) is
variable data_mem: mem_array :=( -- declare a memory array
        X"0000000",             -- initialize data memory
        X"0000000",             -- X denotes a hexadecimal number
        X"0000000",
        X"0000000",
        X"0000000",
        X"0000000",
        X"0000000",
        X"0000000");
variable addr: integer;
begin   -- begin for process
...
end process mem_process;
end architecture behavioral;
```

The Process Statement for Memory

Computational part of a process

```
...
begin   -- begin for architecture

...
begin  -- begin for process
L1: addr := conv_integer(address(2 downto 0));
             -- this
            -- conversion function is in std_logic_arith

L2: if MemWrite = ‘1’ then          -- perform a read or write
L3:    data_mem(addr) := write_data;

L4: elsif MemRead = ‘1’ then
L5:      read_data <= data_mem(addr) after 10 ns;
end if;

end process mem_process;

end architecture behavioral;
```

Process Construct

```
label: process (sensitivity_list)
-- declarative region for variables and constants
begin -- begin for process
...
end process label;
```

- All statements in a process are executed sequentially in the order implied by the text
- Therefore, values assigned to variables (not signals though) in a statement are immediately visible to the subsequent statement
- Compare sequential nature of statements in a process to concurrent signal assignment statements.

Properties of Process Construct

- Control flow is strictly sequential
  - Altered by constructs such as **`if-then-else`** or **`loop`** statements
  - Process can be thought of a traditional sequential program.
  - However, a process can assign internally computed values to signals in the interface after a specified delay.
  - `read_data <= data_mem(addr)` **`after`** `10` **`ns;`**

    `data_mem(addr) := write_data;`
  - Externally, discrete event execution model is applicable to process.
  - Internally, many complex behavior are described.
  - *With respect to simulation time, a process executes in zero time.*

Sensitivity List

- `label: process (sensitivity_list)`
- Concurrent signal assignment statements are executed when there is an event on the signals in RHS of the statement
- Process statements are executed when there is an event on the signals in the sensitivity list.
- When an input that is not in the sensitivity list changes, process won't execute
- Once started, process executes to the end and may generate further events.
- We can think of a process as a big and complex, nevertheless, CSA statement that executes concurrently with other CSA statements.
- Or CSA can be seen a simple and special processes.

Miscellaneous

- new type definition:
  - **type** mem_array **is array**(0 **to** 7) **of** std_logic_vector(31 downto 0);
- Variable declaration of the new type
  - **variable** data_mem: mem_array := (...);
- new type description:
  - unsigned in std_logic_arith.
  - used for memory addresses.
- type conversion function
  - conv_integer() in std_logic_arith.
  - memory array is indexed by integers.

Behavioral Models

- VHDL models using process constructs are usually referred to as <u>behavioral models</u>.
- Like in a structural programming languages, we can write complex VHDL code incorporating several processes and CSA statements.
- As we can gain insight, we will know when and how to use processes in order to effectively describe digital systems for both simulation & synthesis
  - Behavioral models often make synthesis more challenging while it offers many advantages in simulation

If-Then-Else and If-Then-Elsif Statements

```
if MemWrite = ‘1’ then
       data_mem(addr) := write_data;
elsif MemRead = ‘1’ then
      read_data <= data_mem(addr) after 10 ns;
[else]
...
end if;
```

Case Statement

```
case expression is
   when choices => sequential-statements   -- branch #1
   when choices => sequential-statements   -- branch #2
   -- Can have any number of branches
   [when others => sequential-statements ] -- last branch
end case;
```

```
type week_day is (mon, tue, wed, thu, fri, sat, sun);
type dollars is range 0 to 10;
variable day: week_day;
variable pocket_money: dollars;

case day is
   when tue => pocket_money := 6;              -- branch #1
   when mon|wed => pocket_money := 2;          -- branch #2
   when fri to sun => pocket_money := 7;       -- branch #3
   when others => pocket_money := 0;           -- last branch
end case;
```

4-by-1 MUX with Case Statement

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX is
port(a,b,c,d: in std_logic;
     ctrl: in std_logic_vector(0 to 1);
     z : out std_logic);
end entity MUX;

architecture mux_behavior of mux is
  constant mux_delay:time:=10 ns;
begin
mux_process: process(a,b,c,d,ctrl) is
  variable tmp:std_logic;
begin

  case ctrl is
    when “00” => tmp := a;               -- branch #1
    when “01” => tmp := b;               -- branch #2
    when “10” => tmp := c;               -- branch #3
    when “11” => tmp := d;               -- branch #4
   when others => tmp := ‘X’;            -- last branch
  end case;

  z <= tmp after mux_delay;
end process mux_process;
end architecture mux_behavior;
```

A VHDL Model with Two Processes

```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port(x, y: in std_logic;
     sum, carry: out std_logic);
end entity half_adder;

architecture behavioral of half_adder is
begin
  sum_proc: process(x, y) is
  begin
    if (x = y) then
      sum <= ‘0’ after 5 ns;
    else
      sum <= (x or y) after 5 ns;
    end if;
  end process sum_proc;

  carry_proc: process(x, y) is
  begin
    ...
  end process carry_proc;

end architecture behavioral;
```

A VHDL Model with Two Processes

```
carry_proc: process(x, y) is
    begin
      case x is
        when ‘0’ => carry <= x after 5 ns;
        when ‘1’ => carry <= y after 5 ns;
        when others => carry <= ‘X’ after 5 ns;
      end case;
    end process carry_proc
end architecture behavioral;
```

- Each value of the case expression being tested can belong to only one branch of the statement.
- A branch can have more then one sequential statement.
- The branches of case statements must cover all possible values of expression being tested.
- port signals (in the interface) are visible within processes.
- We can have a mix of processes and CSA in a VHDL program

Process + CSA

- Memory with four 8-bit words.
  - One process combined with CSAs.
  - A clock is present

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity register_file is
port(address, write_data: in std_logic_vector(7 downto 0);
     Mem Read, MemWrite, clk, reset: in std_logic;
     read_data: out std_logic_vector (7 downto 0));
end entity register_file;

architecture behavioral of register_file is
  signal reg0, reg1, reg2, reg3:std_logic_vector(7 downto 0));
begin
...
end architecture behavioral;
```

A Register File Model 1

```
...
begin
mem_proc: process(clk) is
begin
if (rising_edge(clk)) then -- wait until next clock edge

  if reset=‘1’ then                -- initialize values on reset
    reg0 <= x”00”;                 -- memory locations are
    reg1 <= x”11”;                 -- initialized to random values
    reg2 <= x”22”;
    reg3 <= x”33”;


  elsif MemWrite = ‘1’ then    -- if not reset
    case address(1 downto 0) is
      when “00” => reg0 <= write_data;
      when “01” => reg1 <= write_data;
      when “10” => reg2 <= write_data;
      when “11” => reg3 <= write_data;
      when others => reg0 <= x”ff”;
    end case;
  endif;
endif;
end process mem_proc;
-- CSA statements

end architecture behavioral;
```

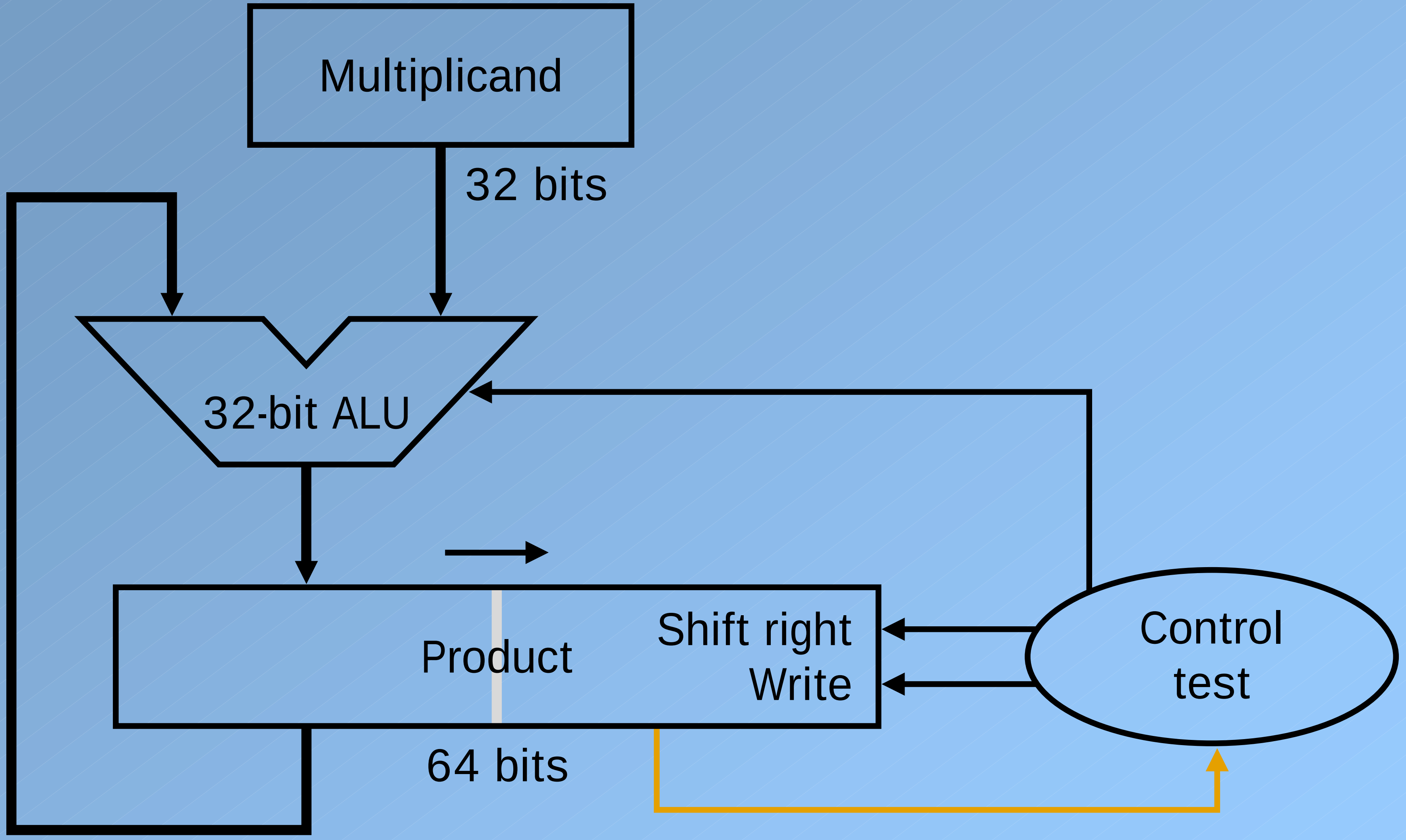
A Register File Model 2

```
...
begin
mem_proc: process(clk) is
begin
...
end process mem_proc;

-- CSA statements
-- memory read is implemented with a conditional signal
-- assignment and concurrent to mem_proc process.
-- Read operations are not synchronous to rising clk edge.

read_data <= reg0 when address( 1 downto 0)=“00”
                  and MemRead = ‘1’ else
             reg1 when address( 1 downto 0)=“01”
                  and MemRead = ‘1’ else
             reg2 when address( 1 downto 0)=“10”
                  and MemRead = ‘1’ else
             reg3 when address( 1 downto 0)=“11”
                  and MemRead = ‘1’ else
             x”00”;

end architecture behavioral;
```

A Register File Model 3

- The function **rising_edge**(clk) is defined in the package std_logic_1164, and is true when the signal clk has just experienced a rising edge (i.e. a true 0-to-1 transition)
  - What if the address is out of range?
  - Is reset synchronous or asynchronous?
- Process for memory write is executing concurrently with a CSA performing memory read operation.
  - memory read operation could be included within the process. What if we did so?

Multiplier


Multiplicand
32 bits
32-bit ALU
Product
Shift right
Write
Control test
64 bits

32-bit Multiplier 1

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity mult32 is
port(multiplicand, multiplier: in std_logic_vector(31 downto 0);
     product: out std_logic_vector (63 downto 0));
end entity mult32;

architecture behavioral of mult32 is
  constant module_delay: Time:= 10 ns;
begin     -- for architecture

mult_process: process(multiplier, multiplicand) is

  variable prod_reg: std_logic_vector(63 downto 0):=X"00000000";
  variable multiplicand_reg: std_logic_vector(31 downto
                                                0):=X"00000000";

begin -- for process
...
end process mult_process;

end architecture behavioral;
```

32-bit Multiplier 2

```
...
mult_process: process(multiplier, multiplicand) is
...
begin – for process
multiplicand_reg := multiplicand;
prod_reg(63 downto 0):= X”00000000” & multiplier;

-- repeated shift-and-add loop
for index in 1 to 32 loop
  if prod_reg (0)=‘1’ then
    prod_reg(63 downto 32) :=
          prod_reg(63 downto 32) + multiplicand_reg(31 downto 0);
  end if;
  -- perform a right shift with zero fill
  prod_reg (63 downto 0) :=‘0’ & prod_reg(63 downto 1);
end loop;

-- write result to output port
product <= prod_reg after module_delay;


end process mult_process;

end architecture behavioral;
```

What is New?

- Loop statements
  - for loop:
    ```
    for index in 1 to 32 loop
      ...
    end loop;
    ```
  - while loop:
    ```
    while j < 32 loop
      ...
      j := j + 1;
    end loop;
    ```
- Concatenation operator &
  - Shift right by one bit using concatenation
  - `prod_reg(63 downto 0) :='0' & prod_reg(63 downto 1);`
  - There are other ways to implement shift operation
- `IEEE.std_logic_unsigned` package contains the definition for the “+” operator.

More on Loop

- In the for loop
  - Loop index variable is declared implicitly.
  - It is local to the loop. If a variable or signal is declared with the same name elsewhere in the process or architecture, it is treated as a distinct object. Within the loop, loop index is taken
  - loop index cannot be modified in the loop.
- While loop enables that the loop index be assigned new values within the body of the loop.
  - Thus, the loop can execute for data dependent number of times.

Process Behavior

- Upon initialization all processes are executed once.
- Thereafter dataflow determines process execution
  - e.g when events occur on the signals in sensitivity list.
  - or **`wait`** statement is used for process initiation.
  - sensitivity list may be imagined as inputs to a digital circuit
  - When an input changes, output of the circuit is re-evaluated for the new input independent of whether the output changes its value or not.

Signals vs. Variables 1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity sig_var is
port(x, y, z: in std_logic;
     res1, res2: out std_logic);
end entity sig_var;

architecture behavioral of sig_var is
  signal sig_s1, sig_s2: std_logic;
begin

proc1: process(x,y,z) is
  variable var_s1, var_s2:std_logic;
begin -- for process
  L1: var_s1 := x and y;
  L2: var_s2 := var_s1 xor z;
  L3: res1  <= var_s1 nand var_s2;
end process proc1;

proc2: process(x,y,z) is
...
end process;

end architecture behavioral;
```

- Two processes: proc1 and proc2

Signals vs. Variables 2

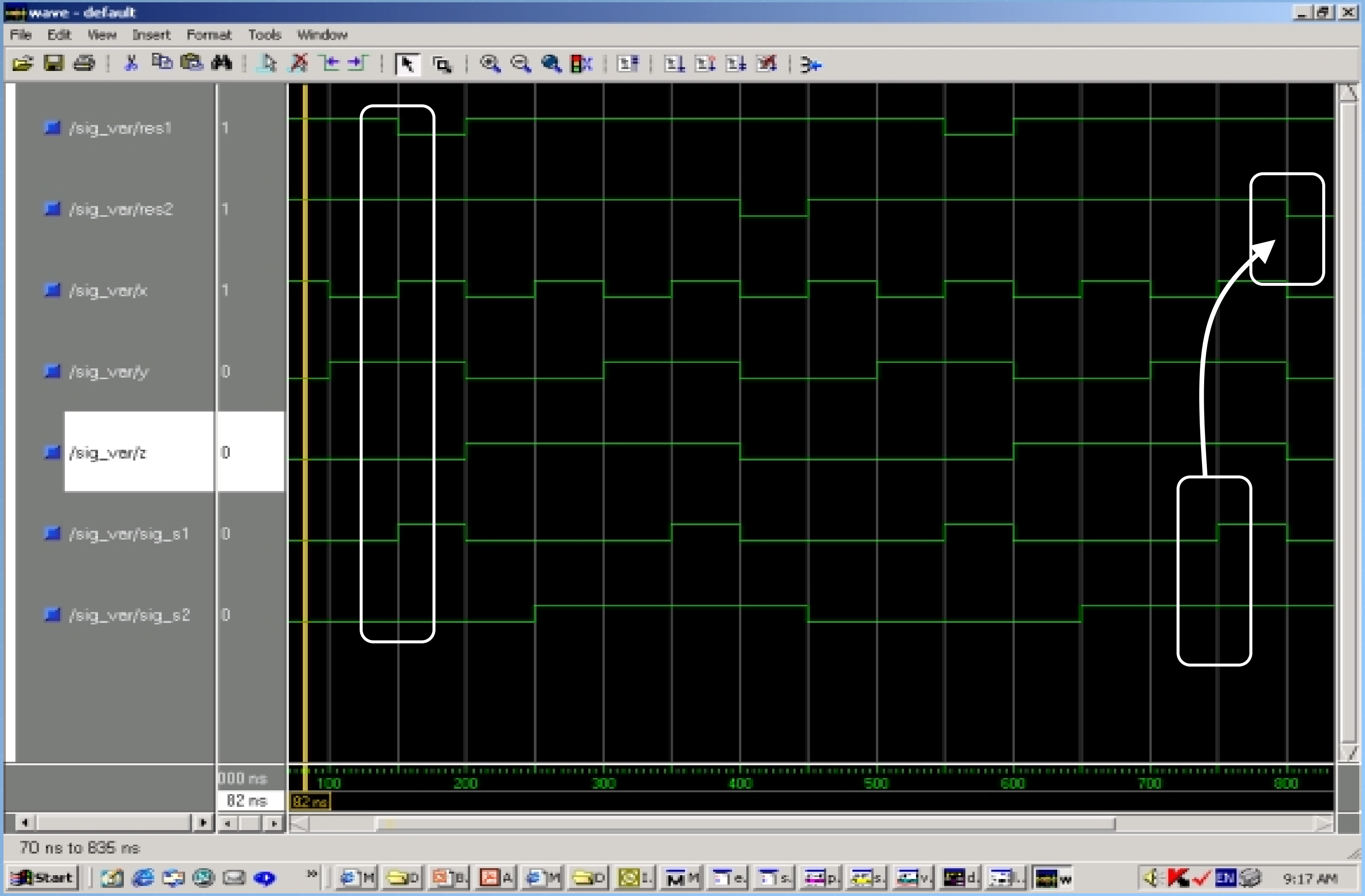
```
...
proc1: process(x,y,z) is
begin
  L1: var_s1 := x and y;
 L2: var_s2 := var_s1 xor z;
 L3: res1  <= var_s1 nand var_s2;
end process;

proc2: process(x,y,z) is
begin
  L1: sig_s1 <= x and y;
  L2: sig_s2 <= sig_s1 xor z;
  L3: res2  <= sig_s1 nand sig_s2;
end process;

end architecture behavioral;
```

- proc1 uses intermediate variables var_s1, var_s2
- proc2 uses intermediate signals sig_s1, sig_s2 (delta delay model)
- They demonstrate different execution behaviors

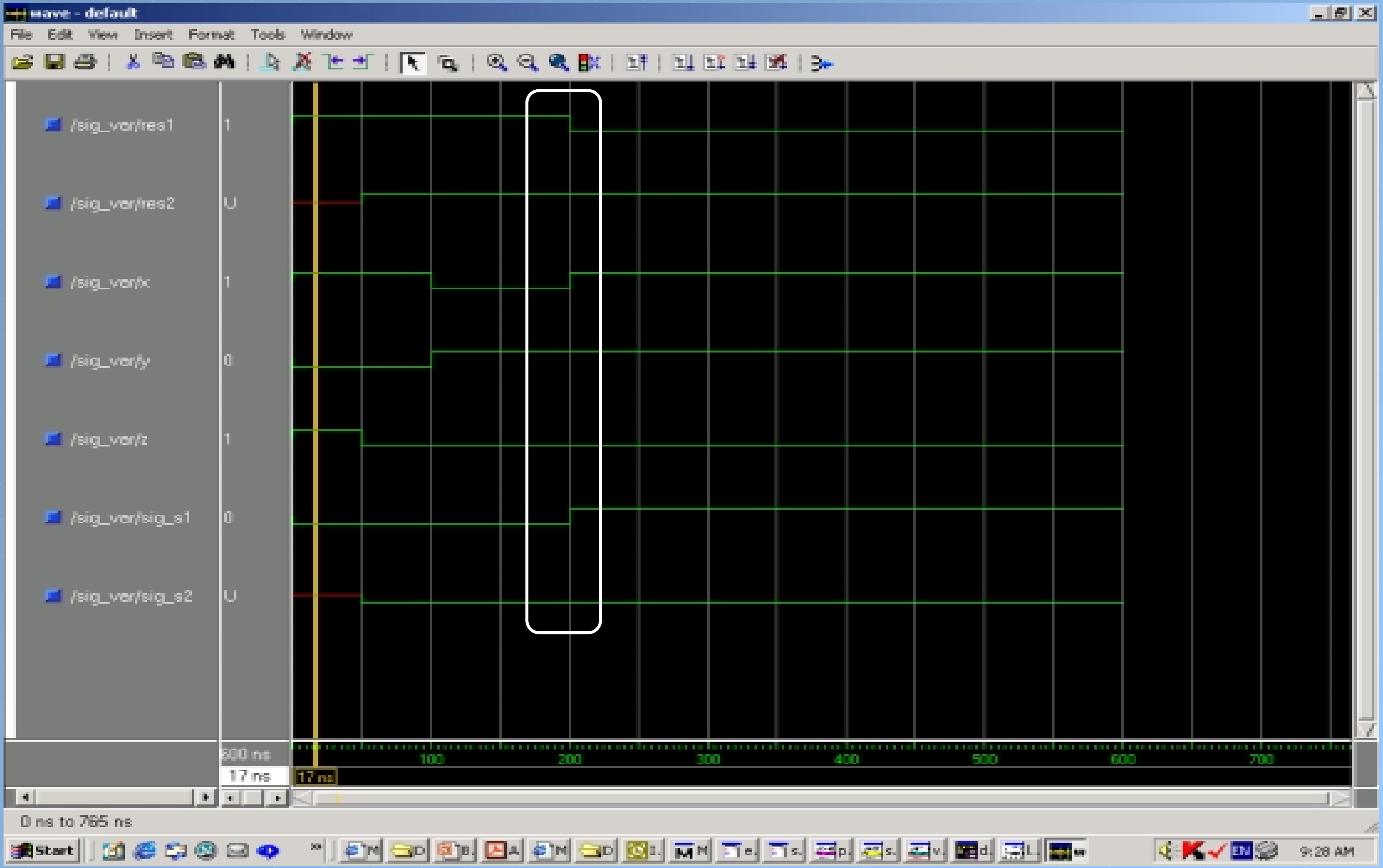
Signals vs. Variables 3

- proc1
  - the new value of var_s1 computed in L1 is used in L2 in the same execution
  - L3 is executed with the new values of var_s1 and var_s2
- proc2
  - L1 and L2 computes the new values of sig_s1 and sig_s2.
  - These signals will not acquire these new values until after delta delay elapses.
  - This means that the new values of sig_s1 and sig_s2 will not be used in this execution of the process.

Signals vs. Variables 4



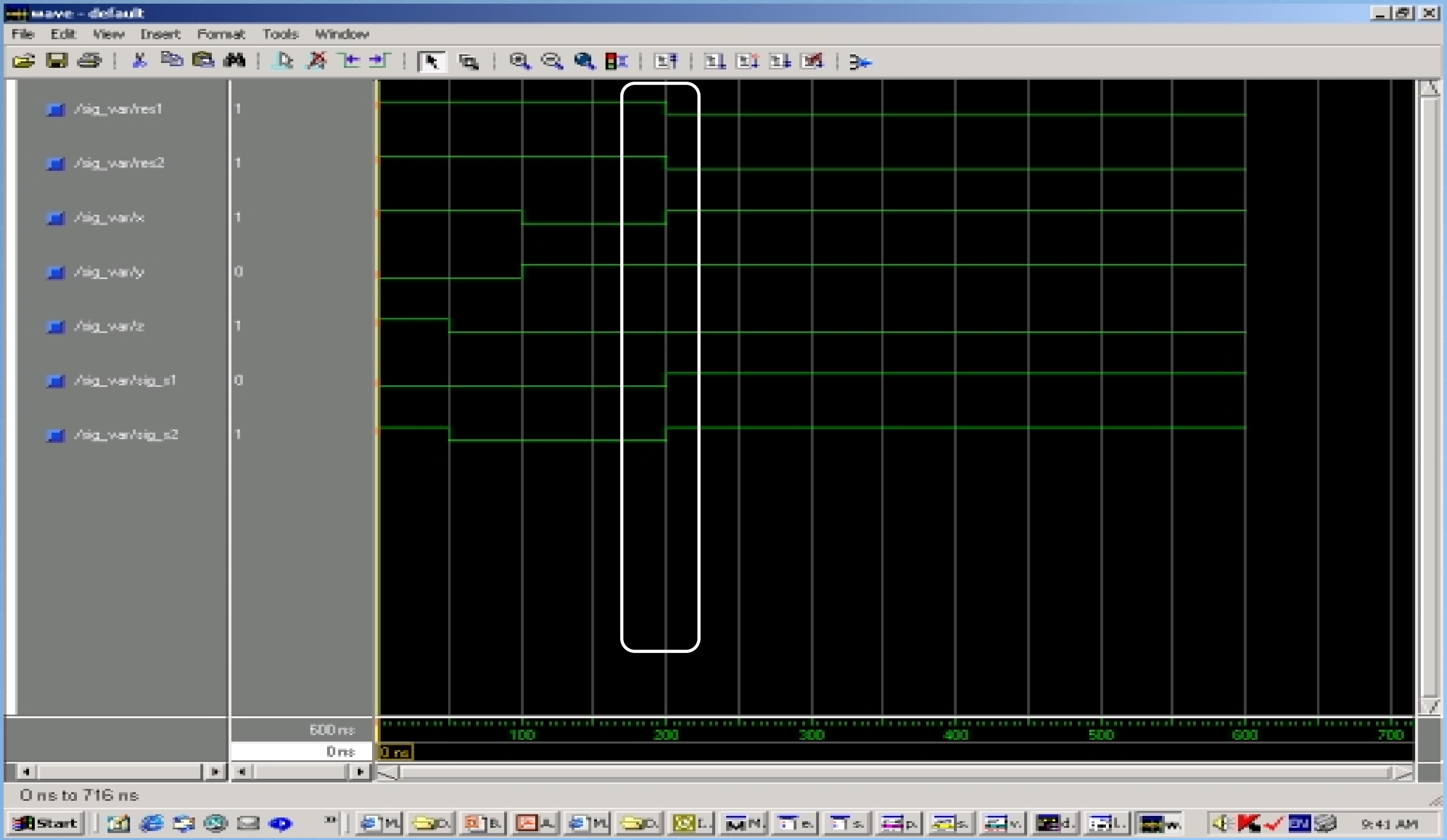
Sensitivity List Subtleties 1



```
proc2: process(x,y,z)
```

Sensitivity List Subtleties 2

```
proc2: process(x,y,z, sig_s1, sig_s2)
```



Sensitivity List Subtleties 2

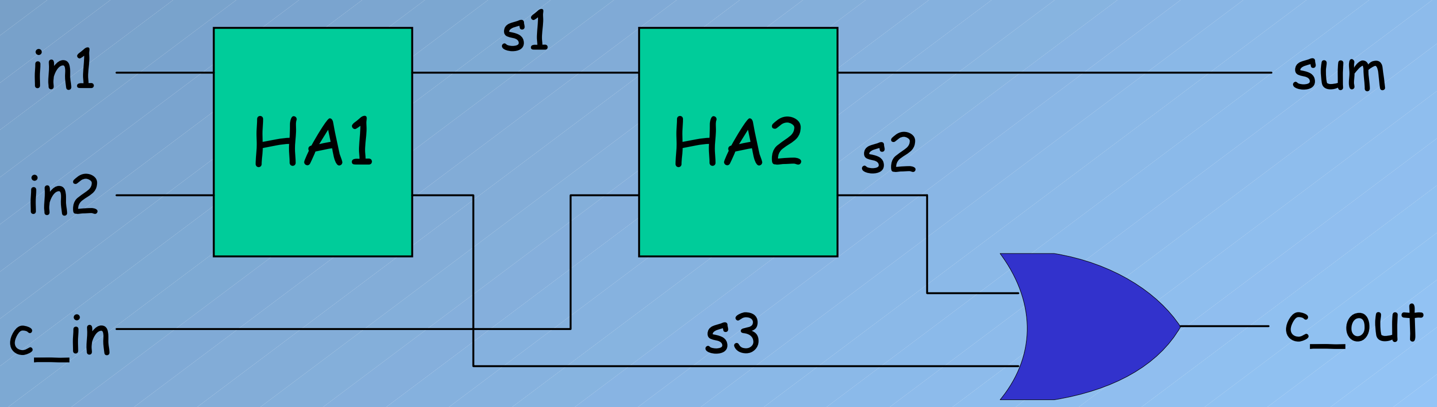
- 1st run of the process
  - sig_s1 : 0→1 (an event)
  - sig_s2 : 0→0
  - res2 : 1 → 1
- 2nd run of the process (after Δ delay)
  - sig_s1 : 1→1
  - sig_s2 : 0→1 (an event)
  - res2 1→1
- 3rd run of the process (after 2Δ delay)
  - sig_s1 : 1→1
  - sig_s2 : 1→1
  - res2 : 1→0

How Processes Communicate

- All of the ports of the entity and the signals declared within an architecture are visible to any process in the architecture.
  - A process can read and update these signals.
- Processes use these signals to communicate
  - For example process A writes signal which is in the sensitivity list of process B
  - the event on this signal will cause process B to execute

Communication Processes 1

- Full adder



- Full adder can be modeled as three communicating processes through internal signals: HA1, HA2, OR
    - they create events on signals s1, s2 and s3
    - s1 must be in the sensitivity list of HA2

Communication Processes 2

```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port(in1, in2, c_in: in std_logic;
     sum, c_out: out std_logic);
end entity full_adder;

architecture behavioral of full_adder is
  signal s1, s2, s3: std_logic;
constant delay: Time:=5ns;
begin
HA1: process(in1,in2) is
begin -- for process
  s1 <= (in1 xor in2) after delay;
  s3 <= (in1 and in2) after delay;
end process HA1;

HA2: process(s1,c_in) is
  sum <= (s1 xor c_in) after delay;
  s2 <= (s1 and c_in) after delay;
end process HA2;

OR: process(s2,s3) is
  c_out <= (s2 or s3) after delay;
end process OR;

end architecture behavioral;
```

The Wait Statement 1

- Event model
  - CSA: when an event occurs on signal in RHS, then CSA statement is executed
  - Process: when an event occurs on signal in the sensitivity list, then process is executed. Otherwise the process is suspended.
- Wait statement allows to model circuits
  - where output values are computed at specific points in time independent of event on the input
  - that respond to only certain events on the input signals (rising edge of the clock)
  - provides a more general way of specifying when a process is executed or suspended
  - We can suspend the process at multiple points not just beginnings

The Wait Statement 2

- **`wait for`** *time expression;*
  - suspend a process for a certain period of time and when this time specified by *time expression* elapses the process is executed.
  - **`wait for`** `20` **`ns`**`;`
- **`wait on`** *`signal;`*
  - suspend a process until an event occurs on any one of one or more signals
  - **`wait on`** `clk, reset, status;`
- **`wait until`** `<`*`condition`* `>`
  - suspends a process until `<`*`condition`* `>` becomes TRUE.
- wait and sensitivity list shouldn't be used together.

Wait Statement: Examples 1

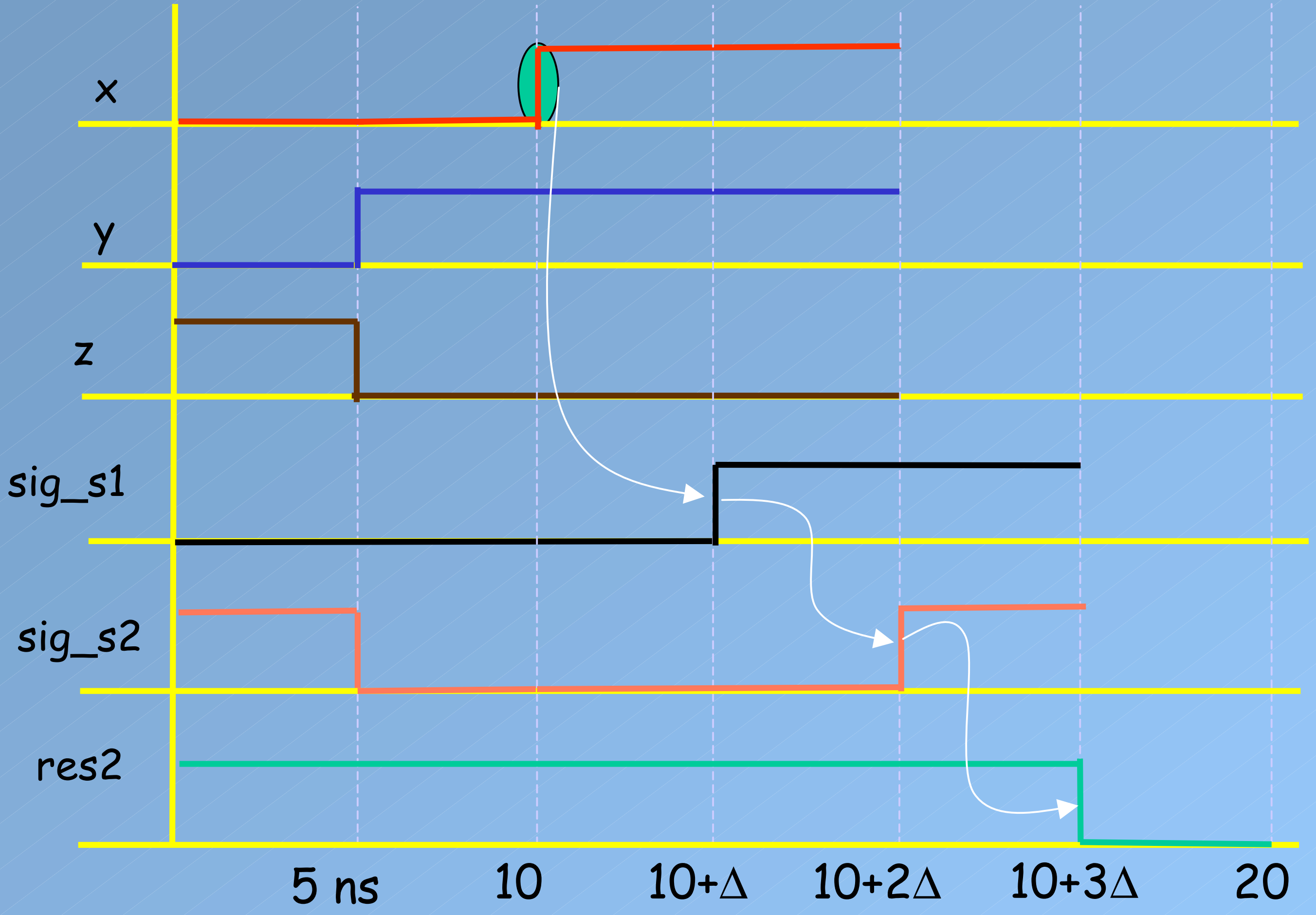
```
HA2: process(s1,c_in) is
  sum <= (s1 xor c_in) after delay;
  s2 <= (s1 and c_in) after delay;
end process HA2;
```

```
HA2: process
  sum <= (s1 xor c_in) after delay;
  s2 <= (s1 and c_in) after delay;
  wait on s1, c_in
end process HA2;
```

- **wait for 0 ns;   ?**

```
proc2: process
begin
  wait on x, y, z;
  L1: sig_s1 <= x and y;
  wait for 0 ns;
  L2: sig_s2 <= sig_s1 xor z;
  wait for 0 ns;
  L3: res2  <= sig_s1 nand sig_s2;
end process;
```

Wait Statement: Examples 2



Wait Statements: Examples 3

- wait on x, y, z;
- wait until A = B;
- wait for 10 ns;
- wait on CLOCK for 20 ns;
- wait until SUM > 100 for 50 ns;
- wait on CLOCK until SUM > 100;

Positive-Edge-Triggered D Flip-Flop

- D input is sampled at the rising edge of the clock and transferred to the output.
- Computation is done at a particular event
- **wait until**(clk’**event and** clk = ‘1’);

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
port(D, clk: in std_logic;
     Q, Qbar: out std_logic);
end entity dff;

architecture behavioral of dff is
begin
output: process is
begin -- for process
  wait until(clk’event and clk = ‘1’);
  Q <= D after 5 ns;
  Qbar <= not D after 5 ns;
end process output;

end architecture behavioral;
```

Positive-Edge-Triggered D Flip-Flop

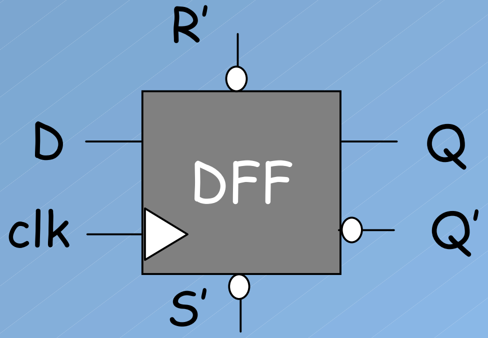
- `clk'`**`event`**
- event is said to be an <u>attribute</u> of signal clk and
- the predicate `clk'event` is TRUE when there is a transition from 0 to 1 in the clock in the most recent simulation cycle.
- Recall that an event is a transition on a signal while a transaction occurs on a signal when a new assignment has been made to the signal, but the value may not have changed.
- std_logic_1164 package provides useful functions such as `rising_edge(clk)` and `falling_edge(clk)`.
- These functions take a `std_logic` type signal and return a Boolean value.

Why Use `rising_edge()`?

- Recall that a signal of type std_logic may have nine different values.
- Therefore, the predicate `(clk'`**`event and`**` clk = '1')` cannot really capture a 0 to 1 transition
  - The transition may be X to 1.
  - Therefore, use `rising_edge()` and `falling_edge()` for detecting a true rising edge and falling edge of a signal, respectively.
- <u>Next issue</u>: how do we initialize a flip-flop?
  - when a system is powered, flip-flops are set to a known initial state
  - However, what we want is to have some control over the initial state

D Flip-Flops with Asynchronous Inputs

- Asynchronous inputs such as `Clear` or `Set` and `Preset` and `Reset` are used to initialize flip-flops to known states.
  - These signals do not work with the clock, on the contrary they override its effect.



S'	R'	clk	D	Q	Q'
0	1	X	X	1	0
1	0	X	X	0	1
1	1	R	1	1	0
1	1	R	0	0	1
0	0	X	X	?	?

Both S and R are active low

D Flip-Flops with Asynchronous Inputs

```
library IEEE;
use IEEE.std_logic_1164.all;

entity asynch_dff is
port(D, Clk, R, S: in std_logic;
     Q, Qbar: out std_logic);
end entity asynch_dff;

architecture behavioral of asynch_dff is
begin
output: process(R, S, Clk) is
begin -- for process
  if (R = ‘0’) then
    Q <= ‘0’ after 5 ns;
    Qbar <= ‘1’ after 5 ns;
  elsif (S = ‘0’) then
    Q <= ‘1’ after 5 ns;
    Qbar <= ‘0’ after 5 ns;
  elsif (rising_edge(Clk)) then
    Q <= D after 5 ns;
    Qbar <= not D after 5 ns;
  endif;
end process output;

end architecture behavioral;
```

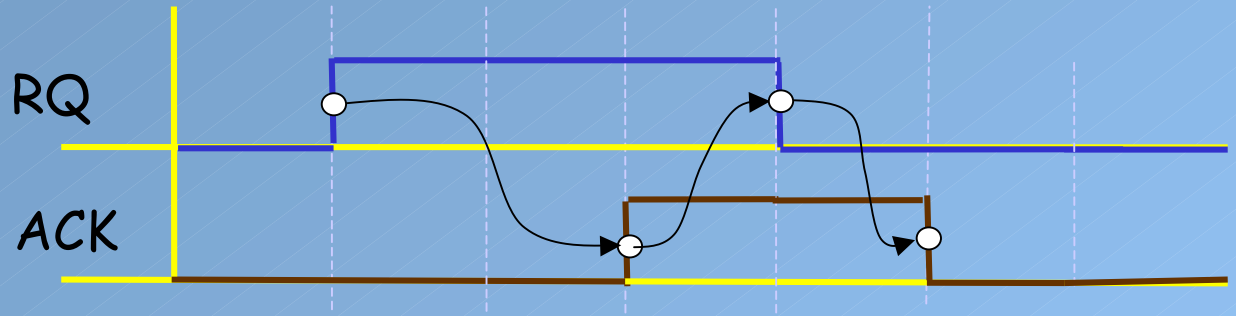
- During synchronous operation both R and S must be held to 1.

Registers and Counters

```
library IEEE;
use IEEE.std_logic_1164.all;

entity reg4 is
port(D: in std_logic_vector(3 downto 0);
     Cl, enable, Clk: in std_logic;
     Q: out std_logic_vector(3 downto 0));
end entity reg4;

architecture behavioral of reg4 is
begin
  reg_process: process(Cl, Clk) is
  begin -- for process
    if (Cl = ‘1’) then
      Q <= “0000” after 5 ns;
    elsif (rising_edge(Clk)) then
      if(enable = ‘1’) then
        Q <= D after 5 ns;
      endif;
    endif;
  end process reg_process;

end architecture behavioral;
```

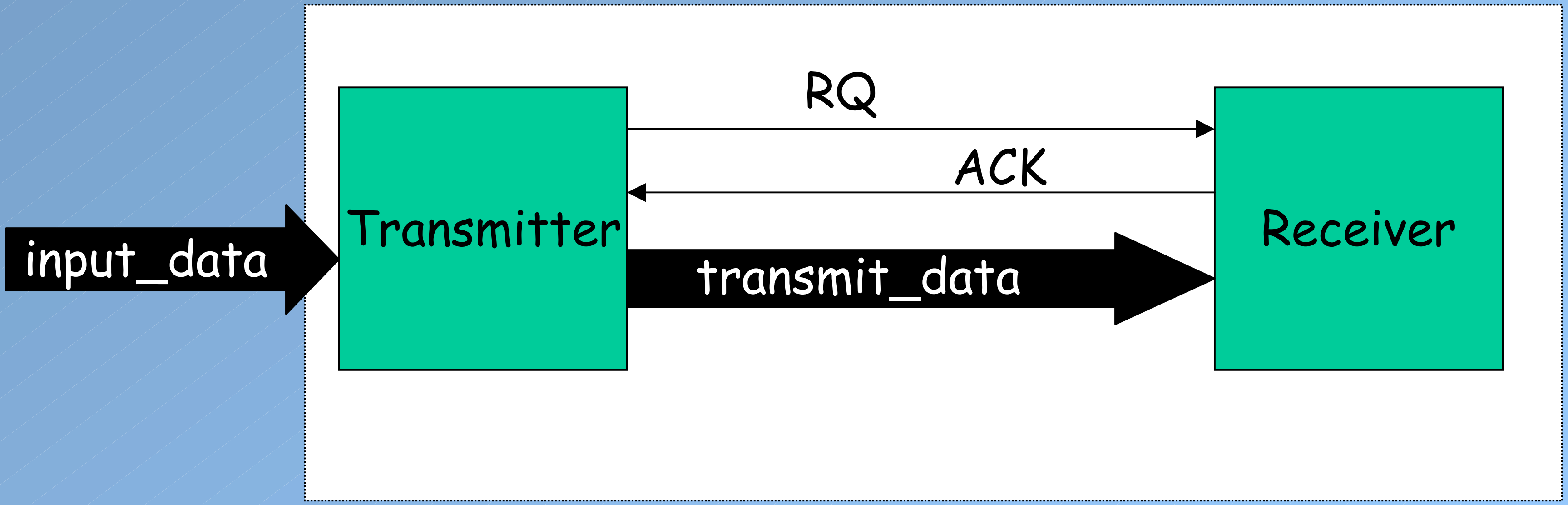
- Constructing a 4-bit register from D flip-flops

How to Model Asynchronous Communication



- 4 –phase protocol for asynchronous data transfer
  - can be modeled two processes communicating via signals.
  - The model uses wait statement that can suspend the process at multiple times.
  - Not possible with sensitivity list.

How to Model Asynchronous Communication



Asynchronous Communication 1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity handshake is
port(input_data: in std_logic_vector(31 downto 0));
end entity handshake;

architecture behavioral of handshake is
  signal transmit_data: std_logic_vector(31 downto 0);
  signal RQ, ACK: std_logic;
begin
  transmitter: process is
  begin -- for process
    wait until input_data’event; -- wait until input data is
                                 -- available
    transmit_data <= input_data; -- provide the data as producer
    RQ = ‘1’;
    wait until ACK = ‘1’;
    RQ = ‘0’;
    wait until ACK = ‘0’;
  end process transmitter;

  receiver: process is
  begin -- for process
  ...
  end process receiver;
end architecture behavioral;
```

Asynchronous Communication 2

```
library IEEE;
use IEEE.std_logic_1164.all;

entity handshake is
port(input_data: in std_logic_vector(31 downto 0));
end entity handshake;

architecture behavioral of handshake is
  signal transmit_data: std_logic_vector(31 downto 0);
  signal RQ, ACK: std_logic;
begin
...
  transmitter: process is
  begin -- for process
  ...
  end process transmitter;

  receiver : process is
    variable receive_data: std_logic_vector(31 downto 0);
  begin -- for process
    wait until RQ = ‘1’;
    receive_data:= transmit_data; -- read the data as consumer
    ACK = ‘1’;
    wait until RQ = ‘0’;
    ACK = ‘0’;
  end process receiver;
end architecture behavioral;
```

Attributes 1

- We have seen that an attribute is used to determine various type of information about a signal
  - For example, `clk'`**`event`** returns TRUE if there has been a change in the value of the signal
  - **event** is said to be an attribute of a signal
- Attributes can also be used to gather information about the other type of VHDL constructs such as entities and arrays.
  - Some attributes are predefined by the language
  - Programmer can also create attributes

Attributes 2

- An attribute can be a value, function, type, range, signal, or constant that can be associated with certain names within a VHDL description.

Type of Attributes

- Function Attributes
  - invokes a function which returns a value
- Value Attributes
  - returns a constant value
- Signal Attributes
  - returns a signal
- Type Attributes
  - returns a type
- Range Attributes
  - returns a range

Function Attributes

- These attributes represent functions that are called to obtain a value.

Function Attribute	Function
`signal_name'`**`event`**	returns a Boolean value signifying a change on this signal
`signal_name'`**`active`**	returns a Boolean value signifying an assignment made to this signal. This assignment may not be a new value
`signal_name'`**`last_event`**	returns the time since the last event on this signal
`signal_name'`**`last_active`**	returns the time since the signal was last active
`signal_name'`**`last_value`**	returns the previous value of this signal
`my_array'`**`length`**	returns the length of the array my_array

Function Attributes: Examples

```
signal CLOCK: std_logic;
constant setup_time: Time:= 5 ns;
signal A:std_logic;
signal COUNT: integer;

if CLOCK = ‘1’ and CLOCK’event then
if A’last_event < setup_time then
if COUNT=20 and COUNT’last_value=10 then
```

- `s1’driving`: returns FALSE if, the driver for the signal s1 is disconnected; otherwise it returns TRUE.
- `s1’driving_value`: returns the current value of the driver for s1. It is illegal to access this attribute when `s1’driving` is FALSE.

Function Attributes: Examples

```
process
begin
   x <= ‘Z’, ‘1’ after 5ns, ‘1’ after 5ns, null after 15ns,
‘U’ after 25ns;
   ... x’driving ...                -- use of driving attribute
   ... x’driving_value ...          -- use of driving_value
                                    -- attribute

wait;
end process;
```

x	'U' @ 25 ns	**null** @ 15 ns	'1' @ 5 ns	'Z' @ Δ
x’driving	TRUE	FALSE	TRUE	TRUE
x’driving_value	'U'	illegal access	'1'	'Z'

Value Attributes 1

- They return a constant value
- For example
  - `type mem_array is array(0 to 7) of std_logic_vector(31 downto 0);`
  - `mem_array’left = 0`
  - `mem_array’ascending = TRUE`
  - `mem_array’lenght = 8`
- Another example
  - `type statetype is (state0, state1, state2, state3);`
  - `statetype’left = state0`
  - `statetype’right = state3`
  - Do not have to remember the every state we just assign `state’left` to the state on reset

Value Attributes 2

Value Attribute	Value
`scalar_name'`**`left`**	returns the leftmost value of `scalar_name` in its defined range
`scalar_name'`**`right`**	returns the rightmost value of `scalar_name` in its defined range
`scalar_name'`**`high`**	returns the highest value of `scalar_name` in its defined range
`scalar_name'`**`low`**	returns the lowest value of `scalar_name` in its defined range
`scalar_name'`**`ascending`**	returns true if `scalar_name` has an ascending range of values (VHDL'93)
`array_name'`**`length`**	returns the number of elements
`array_name'`**`left`**	the left bound of `array_name`
`array_name'`**`right`**	the right bound of `array_name`

Value Attributes: Examples

```
type ALLOWED_VALUE is range 31 downto 0;
type WEEK_DAY is (sun, mon, tue, wed, thu, fri, sat);
subtype WORK_DAY is WEEK_DAY range fri downto mon;
```

- ALLOWED_VALUE’high = ALLOWED_VALUE’left = 31
- ALLOWED_VALUE’right = ALLOWED_VALUE’low = 0

- WEEK_DAY’left = WEEK_DAY’low = sun
- WEEK_DAY’right = WEEK_DAY’high = sat

- WORK_DAY’right = WORK_DAY’low = mon
- WORK_DAY’left = WORK_DAY’high = fri

- ALLOWED_VALUE’ascending is FALSE
- WEEK_DAY’ascending is TRUE
- WORK_DAY’ascending is FALSE

Signal Attributes

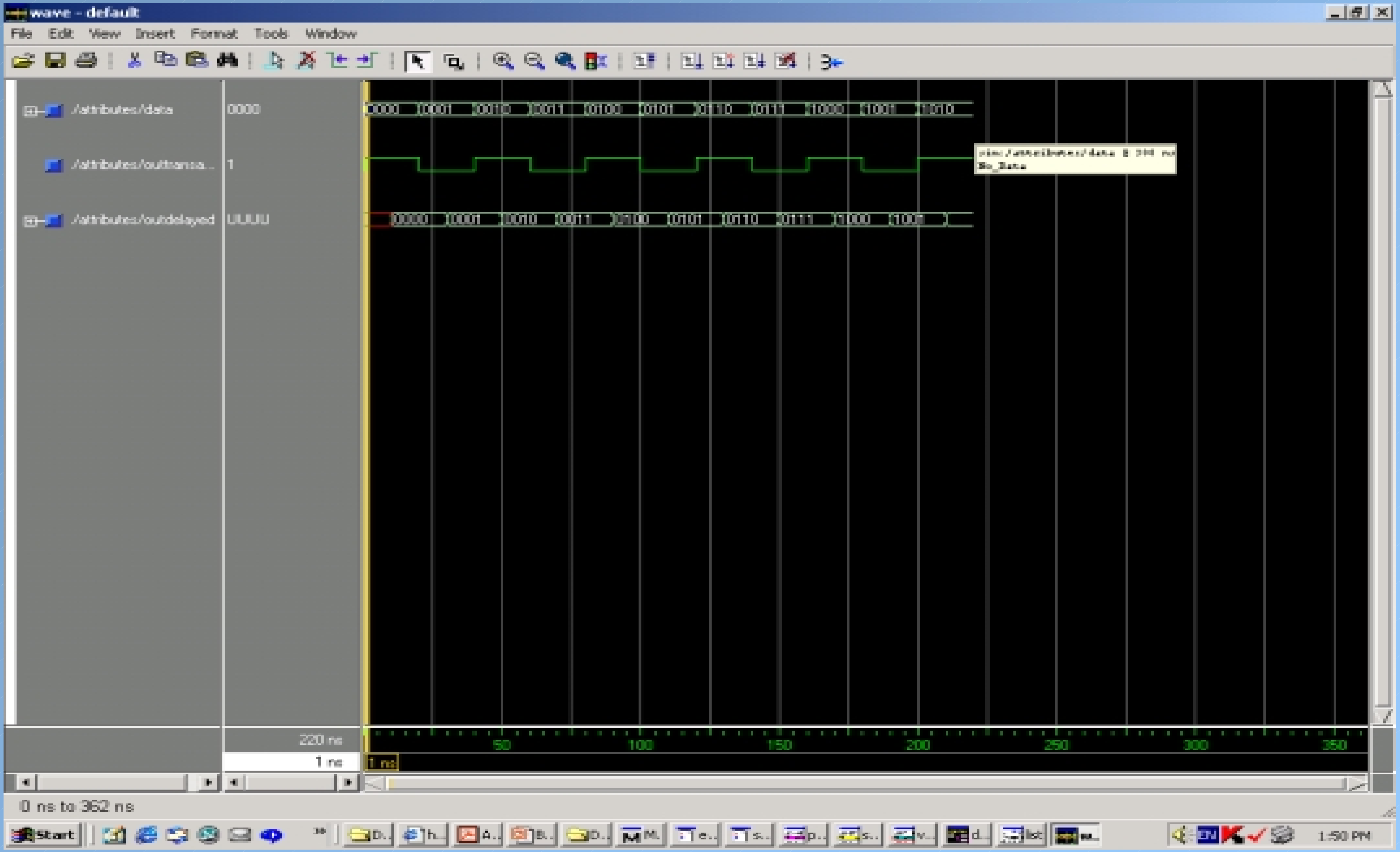
- Signal attributes create new signals from signals explicitly declared in VHDL models.
  - these types of signals are called implicit signals.
  - For example, `signal_name'`**`delayed`**`(T)` create a new signal of the same type, which is delayed by T.

Signal Attribute	Signal
`signal_name'`**`delayed(T)`**	signal delayed by T units of time
`signal_name'`**`transaction`**	returns signal of type bit whose value toggles when `signal_name` is active
`signal_name'`**`quiet(T)`**	true when `signal_name` **`has been quiet for T units of time`**
`signal_name'`**`stable(T)`**	true when event has not occurred on `signal_name` for T units of time

Signal Attributes: Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity attributes is
port(data:in std_logic_vector(3 downto 0));
end entity attributes;

architecture behavioral of attributes is
signal outtransaction:bit;
signal outdelayed:std_logic_vector(3 downto 0);
begin
outdelayed <= data'delayed(10 ns);
outtransaction <= data'transaction;
end architecture behavioral;
```

Signal Attributes: Example 2



When to Use Implicit Signals?

- If we are interested in a change in the value of the signal `data` for example, we can wait for events on the `data`'**`transaction`** implicit signal.
- Delayed signal is used to check for the relationship between the current value and older value of a signal

-
  ```
  wait on ReceiveData'transaction
  if ReceiveData'delayed(5 ns)- ReceiveData > 5 then
      ...
  ```

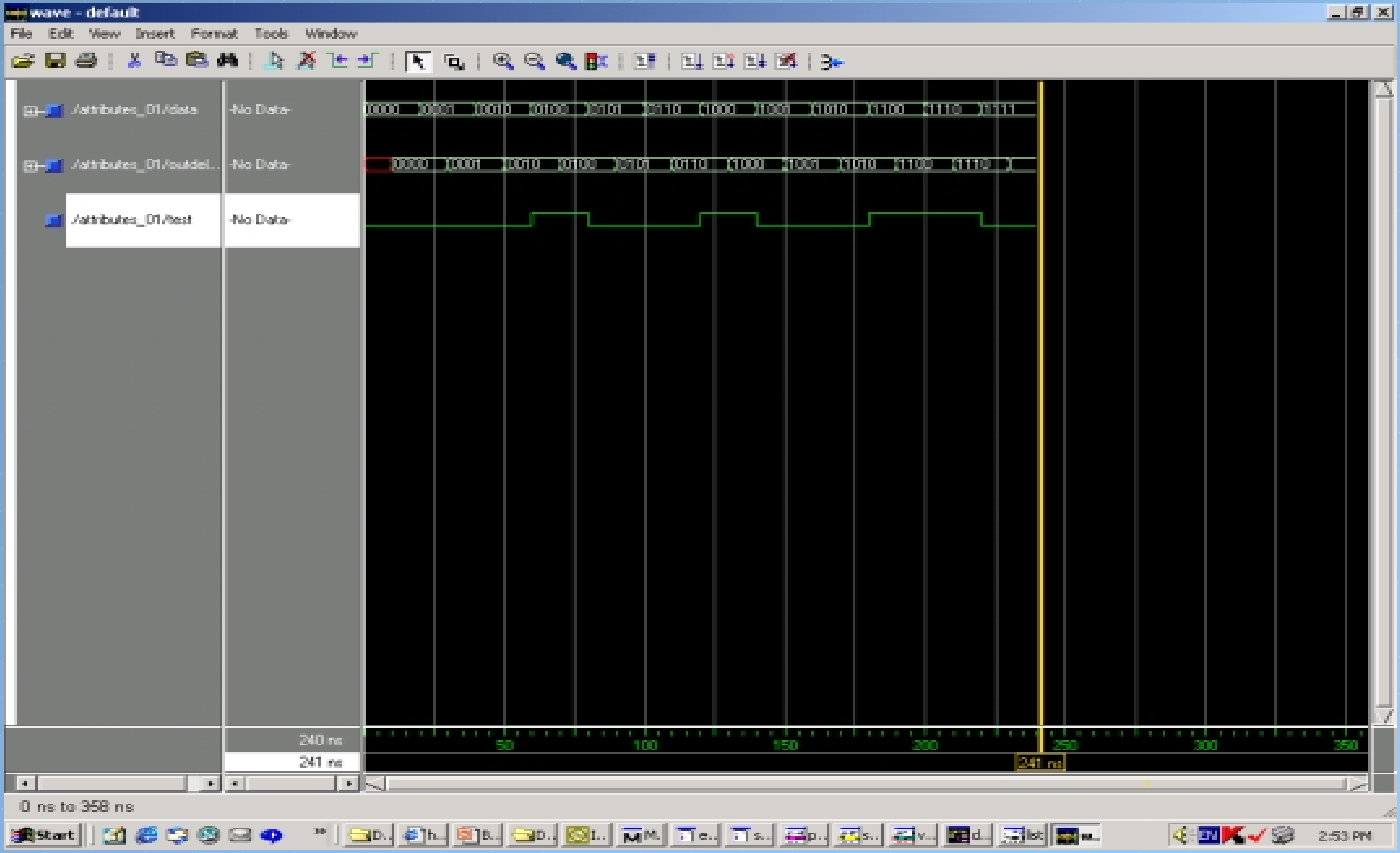
Implicit Signals: Example 1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity attributes_01 is
port(data:in std_logic_vector(3 downto 0));
end entity attributes_01;

architecture behavioral of attributes_01 is
  function to_integer(signal arg:std_logic_vector(3 downto 0)) return
  integer is
  begin
    case arg is
        when "0000" => return 0;
        when "0001" => return 1;
        when "0010" => return 2;
        when "0011" => return 3;
        when "0100" => return 4;
        when "0101" => return 5;
        when "0110" => return 6;
        when "0111" => return 7;
        when "1000" => return 8;
        when "1001" => return 9;
        when "1010" => return 10;
        when "1011" => return 11;
        when "1100" => return 12;
        when "1101" => return 13;
        when "1110" => return 14;
        when "1111" => return 15;
        when others => return 0;
    end case;
  end function to_integer;
```

Implicit Signals: Example 2

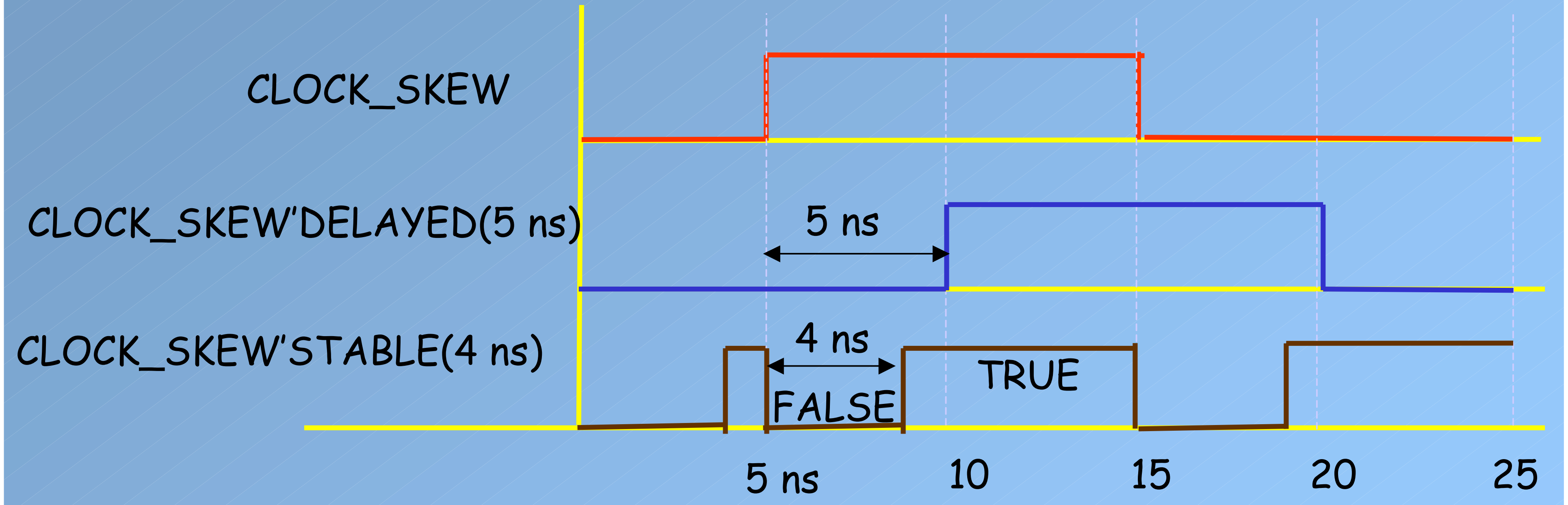
```
signal outdelayed:std_logic_vector(3 downto 0):="0000";
signal test: std_logic:='0';
begin
attributes: process(data)
begin
if(to_integer(data)-to_integer(data'delayed(10 ns))  > 1) then
   test <= '1';
else
   test <= '0';
end if;
end process attributes;
outdelayed <= data'delayed(10 ns);
end architecture behavioral;
```

Implicit Signals: Example 3



Signal Attributes

- `signal CLOCK_SKEW;`



Signal Attributes for Setup Constraint

- 'STABLE attribute can be used for catch a setup violation.
  - It is required to report a violation if signal DATA has not been stable for the specified SETUP time before the falling edge of the signal CLK occurs.
  - Example:

```
process
   constant SETUP: TIME:=1.2 ns;
begin
   wait until CLK = ‘1’ and CLK’event;
   assert DATA’STABLE(SETUP)
      report “setup violation”
      severity WARNING;
end process;
```

A D Flip-Flop with Setup & Holdup Violations Check

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
port(d, clk:in std_logic; q, qbar:out std_logic);
end entity dff;

architecture check_times of dff is
  constant hold_time: Time := 5 ns;
  constant setup_time: Time := 3 ns;
begin
  process(d, clk)
  begin
    -- check for hold time
    if d'event then
       assert NOW = 0 ns or clk'last_event >= hold_time
         report "hold time too short!"
         severity FAILURE;
    end if;

...
end architecture check_times;
```

NOW is a predefined function that returns the current simulation time.

A D Flip-Flop with Setup & Holdup Violations Check

```
...
    -- check for setup time
    if clk = 1 and clk’event then
       assert NOW = 0 ns or d’last_event >= setup_time
         report “setup time too short!”
         severity FAILURE;
    end if;

    -- behavior of D FF
    if clk = 1 and clk’event then
      q <= d;
      qbar <= not d;
    end if;
  end process;
end architecture check_times;
```

Signal Attributes vs. Function Attributes

- Function attributes return value
- Signal attributes creates a new signal
  - Signal attributes can be used wherever a signal is expected; for example in the sensitivity list of a process
  - Example:

```
PSTABLE: process(A, B, CLK'STABLE)
begin
...
end process;
```

  - It is illegal to use CLK'EVENT, for example, in the sensitivity list.

Range Attribute

- Returns a range
- Useful especially when writing loops
  - Consider a loop that scans all of the elements in an array `value_array()`;
  - The index range is returned by `value_array’`**`range`**`.`
  - We may not know the size of the array after all.
  - Example:

```
for i in value_array’range loop
   ...
   my_var := value_array(i);
   ...
end loop;
```

Range Attribute

- Examples:
  - `variable WBUS: std_logic_vector(7 `**`downto`**` 0);`
  - `WBUS’range returns the range “7 `**`downto`**` 0”`
  - `WBUS’reverse_range returns the range “0 `**`to`**` 7”`

Type Attribute

- If T is any type or subtype, T'BASE, which is the only type attribute, returns base type of T
  - This attribute cannot be used in expressions since it returns a type.
  - It can be used in conjunction with other attributes.
  - For example
    ```
    type ALU_OPS is (ADD,SUB,MULT,DIV,AND,NAND,OR,NOR);
    subtype ARITH_OPS is ALU_OPS range ADD to DIV;
    ```
  - then
    `ARITH_OPS'BASE` is `ALU_OPS`.
  - `ARITH_OPS'BASE'LEFT` returns `ADD`.

User-Defined Attributes 1

- Constants of any type (except access or file types)
  - They are declared using attribute declarations.
  - **attribute** *attribute-name*: *value-type*;
  - Example:

```
architecture beh of user_defined is
  attribute pin_no: natural;
  attribute technology: string;
...
```

- They are yet not associated with an entity.
- Entity, in this context, could be and entity, a signal, an architecture, a variable, type, subtype, package, procedure, or function.

User-Defined Attributes 2

To associate

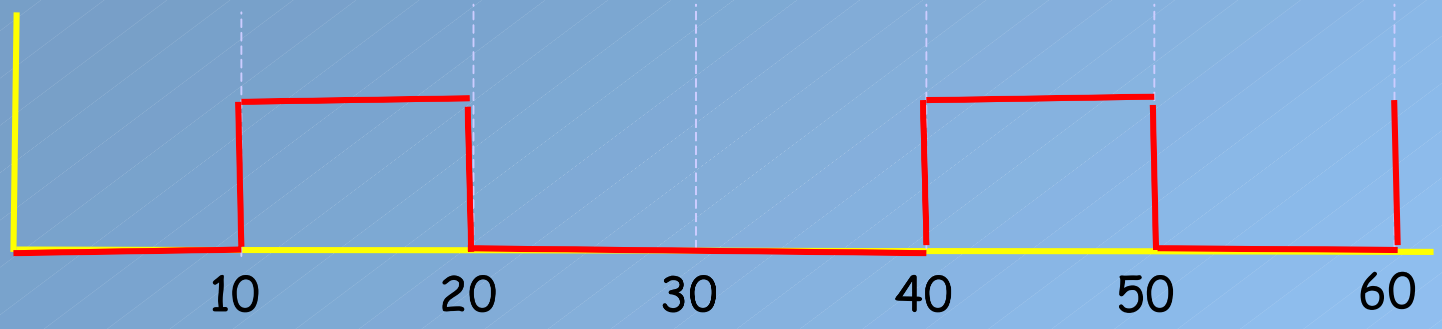
- **attribute** *attribute-name* **of** item-names: name_class **is** expression;

```
architecture beh of user_defined is
...
attribute pin_no of Q: signal is 42;
attribute technology of all: component is “CMOS”;
```

Generating Clocks and Periodic Waveforms

```
library IEEE;
use IEEE.std_logic_1164.all;
entity periodic is
port(Z: out std_logic);
end entity periodic;
architecture behavioral of periodic is
begin
process is
begin
Z <= ’0’, ‘1’ after 10 ns, ‘0’ after 20 ns, ‘1’ after
40 ns;
wait for 50 ns;
end process;
end architecture behavioral;
```

Generating Clocks and Periodic Waveforms



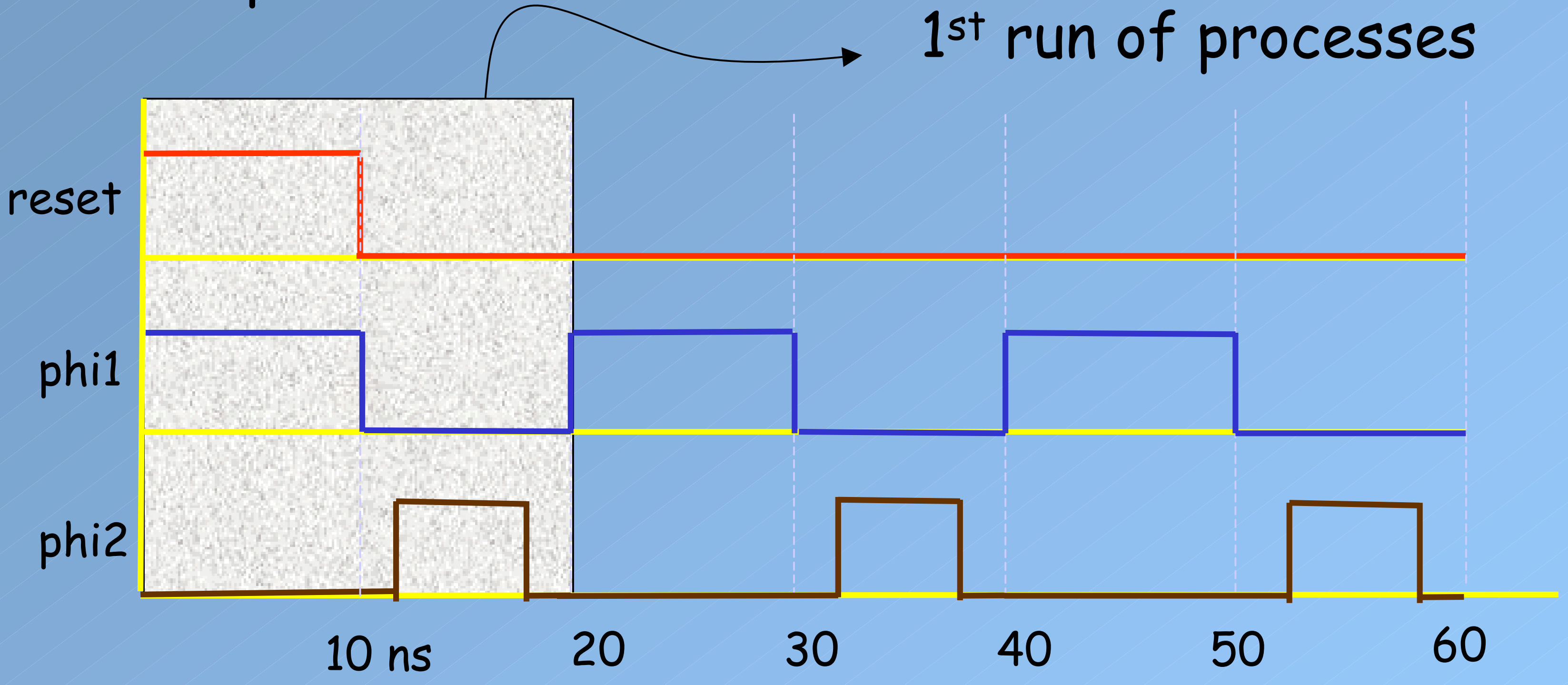
- This process is executed repeatedly with 50 ns period
  - recall upon initialization, all processes are executed.
  - therefore, every process is executed at least once.
  - wait for 50 ns cause the process to be re-executed.

Generating a Two-Phase Clock

```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phases is
port(phi1, phi2: out std_logic);
end entity two_phases;
architecture behavioral of two_phases is
begin
  reset_process: reset <= ‘1’, ‘0’ after 10 ns;
  clock_process: process is
  begin
    phi1 <= ‘1’, ‘0’ after 10 ns;
    phi2 <= ‘0’, ‘1’ after 12 ns, ‘0’ after 18 ns;
    wait for 20 ns;
  end process clock_process;
end architecture behavioral;
```

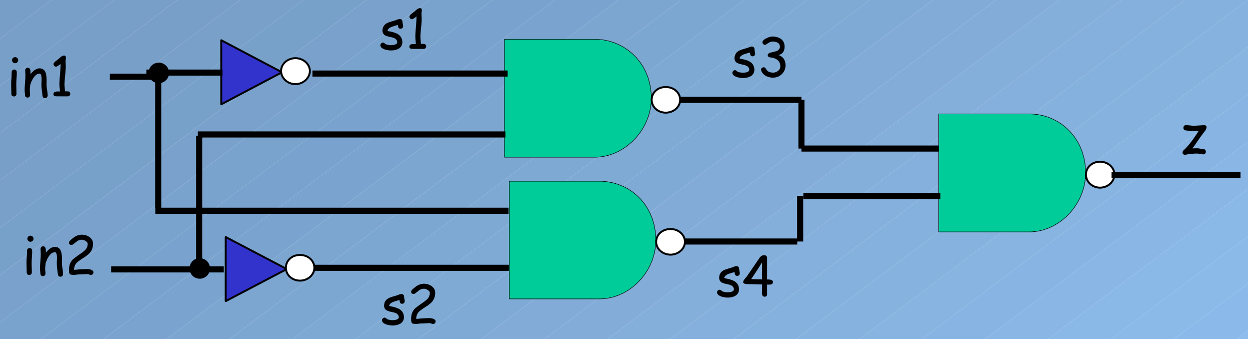
Generating a Two-Phase Clock

- The pulses in two clock signals are adjusted not to overlap



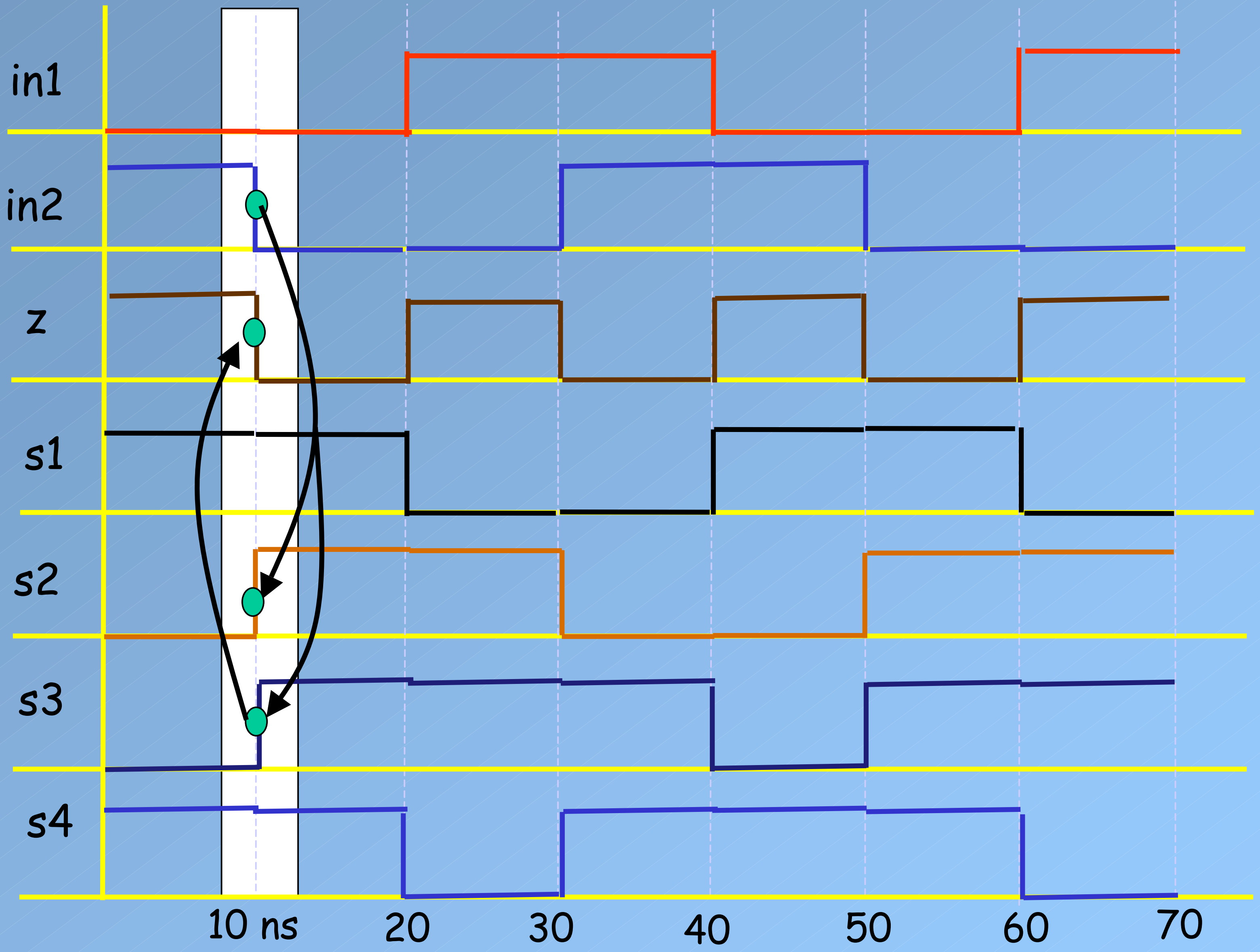
- Note that reset process is executed once.

Using Signals in a Process



```
library IEEE;
use IEEE.std_logic_1164.all;

entity combinational is
port (in1, in2: in std_logic;
      z: out std_logic);
end entity combinational;

architecture behavior of combinational is
signal s1, s2, s3, s4: std_logic := ‘0’;
begin
  s1 <= not in1;
  s2 <= not in2;
  s3 <= not (s1 and in2);
  s4 <= not (s2 and in1);
  z <= not (s3 and s4);
end architecture behavior;
```

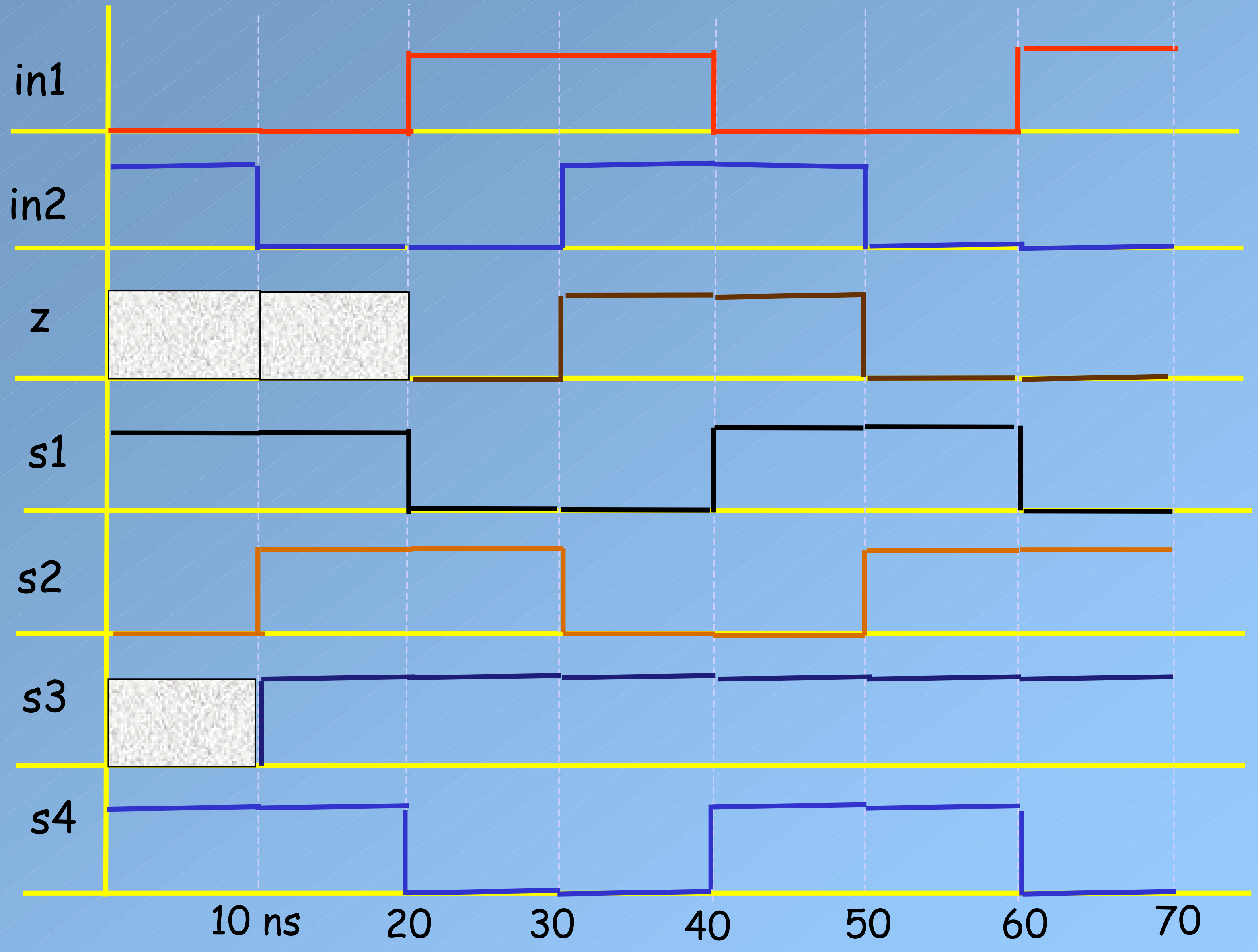
Delta Delays: Example



Using Signals in a Process 2

```
library IEEE;
use IEEE.std_logic_1164.all;

entity combinational is
port (in1, in2: in std_logic;
      z: out std_logic);
end entity combinational;

architecture behavior of combinational is
signal s1, s2, s3, s4: std_logic := ‘0’;
begin
delta_process: process(in1, in2) is
begin
  s1 <= not in1;
  s2 <= not in2;
  s3 <= not (s1 and in2);
  s4 <= not (s2 and in1);
  z  <= not (s3 and s4);
end process delta_process;
end architecture behavior;
```

Using Signals in a Process 3



Using Signals in a Process 4

	0 ns	10	20	30	40	50	60	70
in1	0	0	1	1	0	0	1	1
in2	1	0	0	1	1	0	0	0
s1	1	1	0	0	1	1	0	0
s2	0	1	1	0	0	1	1	1
s3	U	1	1	1	1	1	1	1
s4	1	1	0	0	1	1	0	0
z	U	U	0	1	1	0	0	0

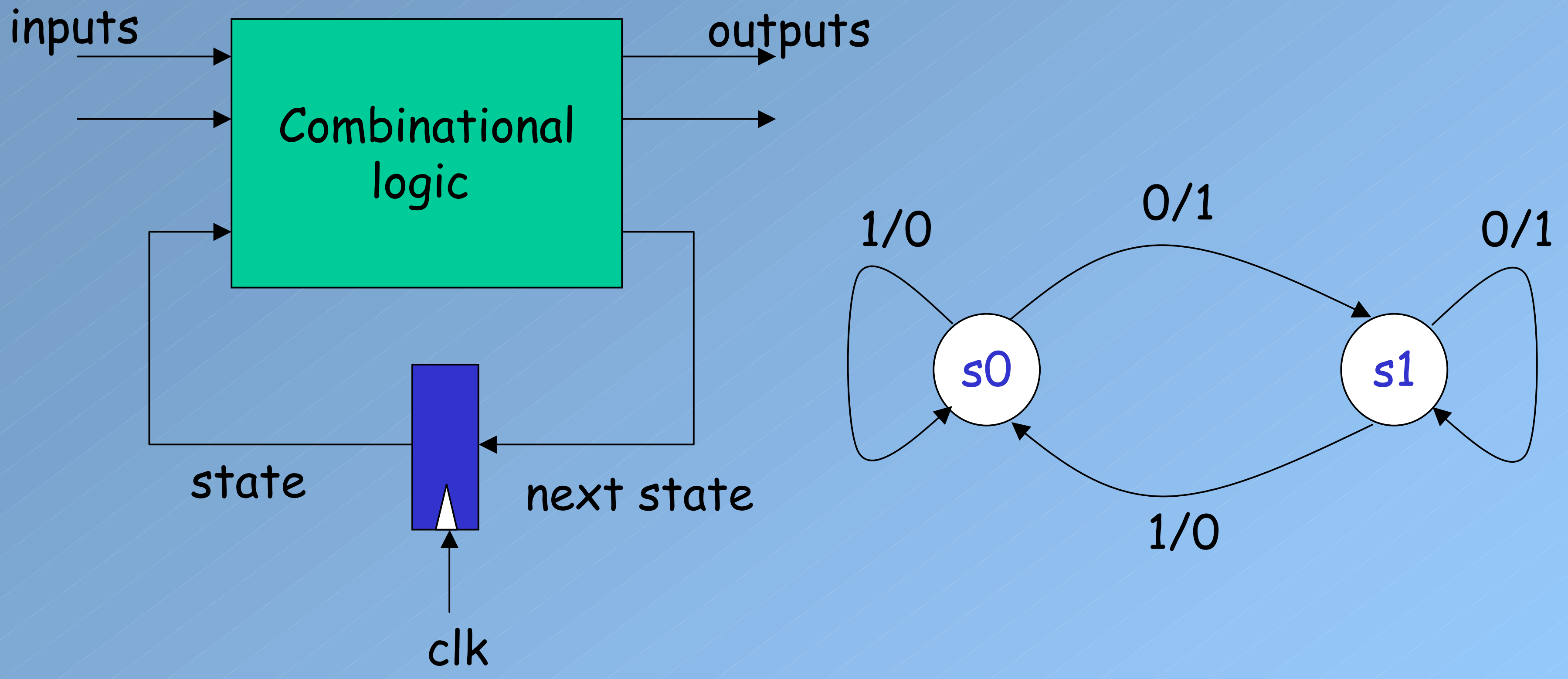
```
s1 <= not in1;
s2 <= not in2;
s3<=not(s1 and in2);
s4<=not(s2 and in1);
z<=not(s3 and s4);
```

- When we made the process sensitive to all signals then we would obtain the identical trace.

State Machines

- Combinational circuits are sensitive to the inputs
  - activated when an event occurs on an input signal
- Sequential circuits retain information in internal devices such as flip-flops and latches.
  - the value stored in these devices are referred to as the <u>state</u> of the circuit.
  - There is a finite number of states.
  - the output values are computed as functions of both internal state and input signals.
  - The internal state is updated at discrete points in time determined by a periodic signal such as clock.

Finite State Machine (FSM)



VHDL Model: Two communicating processes

1. Process implementing combinational component. Sensitive to events on input signals and changes in the state.
2. Process implementing the sequential component. Sensitive to active edge of the clock

VHDL Model for FSM 1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity state_machine is
  port (reset, clk, x: in std_logic;
        z: out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
  type state_type is (state0, state1);
  signal state, next_state: state_type := state0;
begin
...
end architecture behavioral;
```

Combinational Part

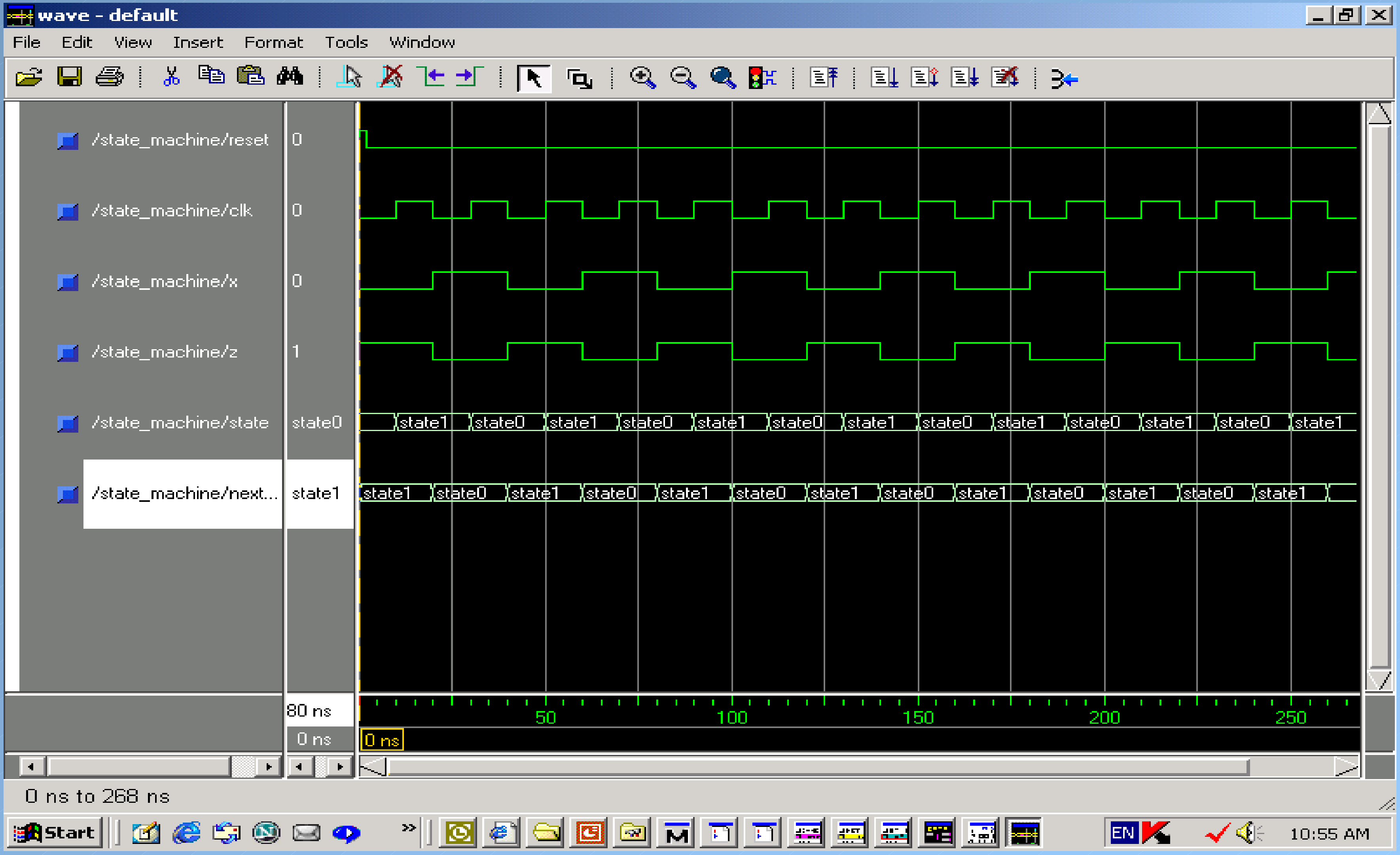
```
comb_process: process(state, x) is
begin
  case state is
    when state0 =>
       if x = ‘0’ then next_state <= state1; z <= ‘1’;
       else next_state <= state0; z <= ‘0’;
       end if;
    when state1 =>
       if x = ‘1’ then next_state <= state0; z <= ‘0’;
       else next_state <= state1; z <= ‘1’;
       end if;
  end case;
end process comb_process;
```

Sequential Part

```
clk_process: process is
begin
  wait until (rising_Edge(clk));
        if reset = ‘1’ then
                state <= state_type’left;
        else
                state <= next_state;
        end if;
end process clk_process;
end architecture behavioral;
```

- state and next_state are used by two process to communicate values.
- Enumerated type is used.
- Attributes of enumerated type `statetype’left` is used to initialize state machine.

Simulation

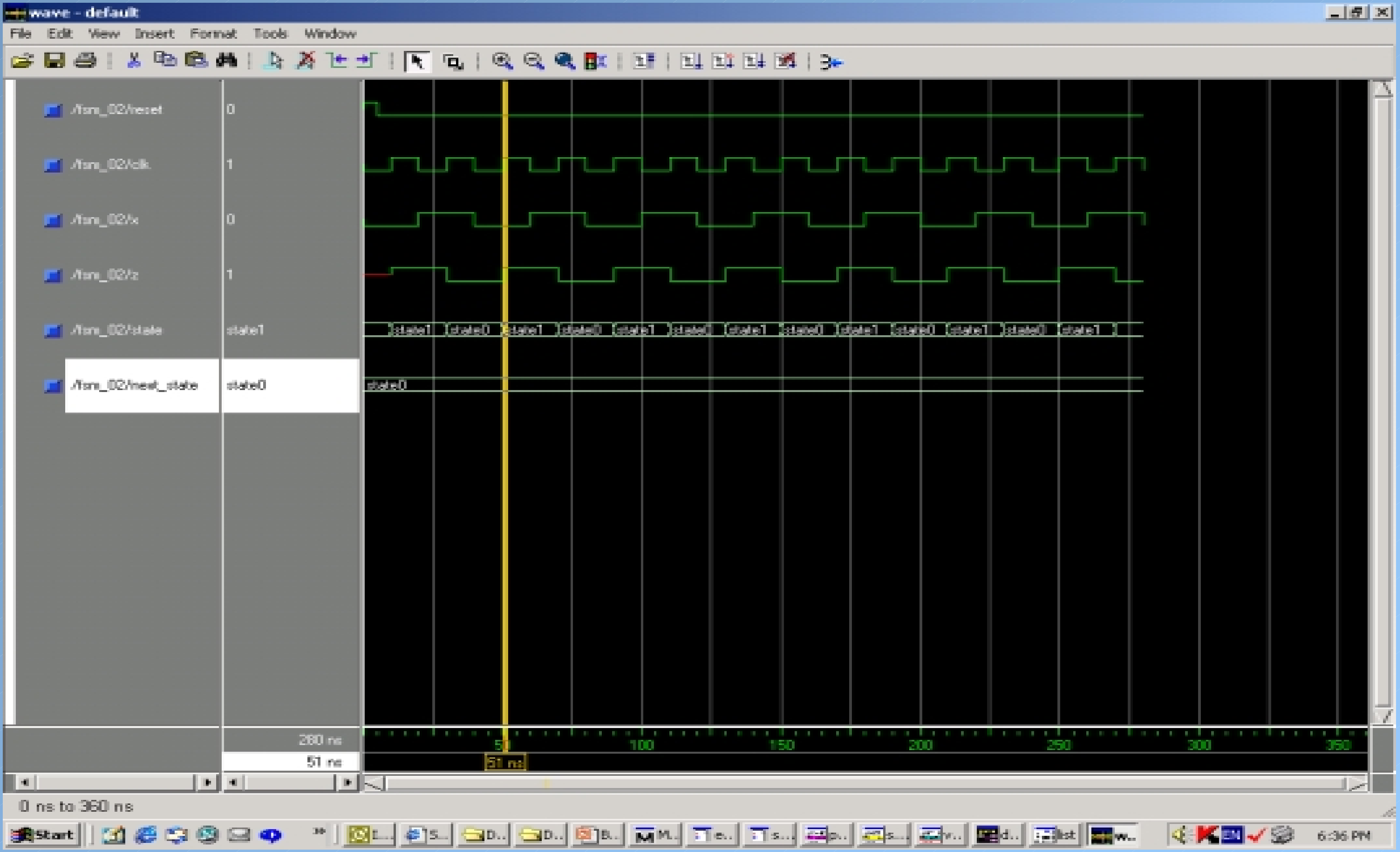


wave - default
File Edit View Insert Format Tools Window
/state_machine/reset 0
/state_machine/clk 0
/state_machine/x 0
/state_machine/z 1
/state_machine/state state0
/state_machine/next... state1
state1 state0 state1 state0 state1 state0 state1 state0 state1 state0 state1 state0 state1
state1 state0 state1 state0 state1 state0 state1 state0 state1 state0 state1 state0 state1
80 ns
0 ns
50 100 150 200 250
0 ns
0 ns to 268 ns
Start
EN
10:55 AM

Alternative Model: with One Process

```
process(clk, state, x) is
begin
  if (rising_edge(clk)) then
    if reset = '1' then
      state <= state_type'left;
      z <= '0';
    else
      case state is
        when state0 =>
         if x = '0' then state <= state1; z<='1';
         else state <= state0; z<='0';
         end if;
        when state1 =>
         if x = '1' then state <= state0; z<='0';
         else state <= state1; z<='1';
         end if;
      end case;
    end if;
  end if;
end process;
```

Simulation



```
/fsm_02/reset        0
/fsm_02/clk          1
/fsm_02/x            0
/fsm_02/z            1
/fsm_02/state        state1
/fsm_02/next_state   state0
```

Output signal z changes with clock.

Alternative Model: with Three Process

```
...
signal state, next_state: statetype := state0;
begin
output_process: process(state, x) is
begin
  case state is
    when state0 =>
       if x = ‘1’ then z <= ‘0’;
       else z <= ‘1’;
       end if;
    when state1 =>
       if x = ‘1’ then z <= ‘0’;
       else z <= ‘1’;
       end if;
  end case;
end process output_process;
```

Output Process

Alternative Model: with Three Process

```
next_state_process: process(state, x) is
begin
  case state is
    when state0 =>
       if x = '1' then next_state <= state0;
       else next_state <= state1;
       end if;
    when state1 =>
       if x = '1' then next_state <= state0;
       else next_state <= state1;
       end if;
  end case;
end process next_state_process;
```

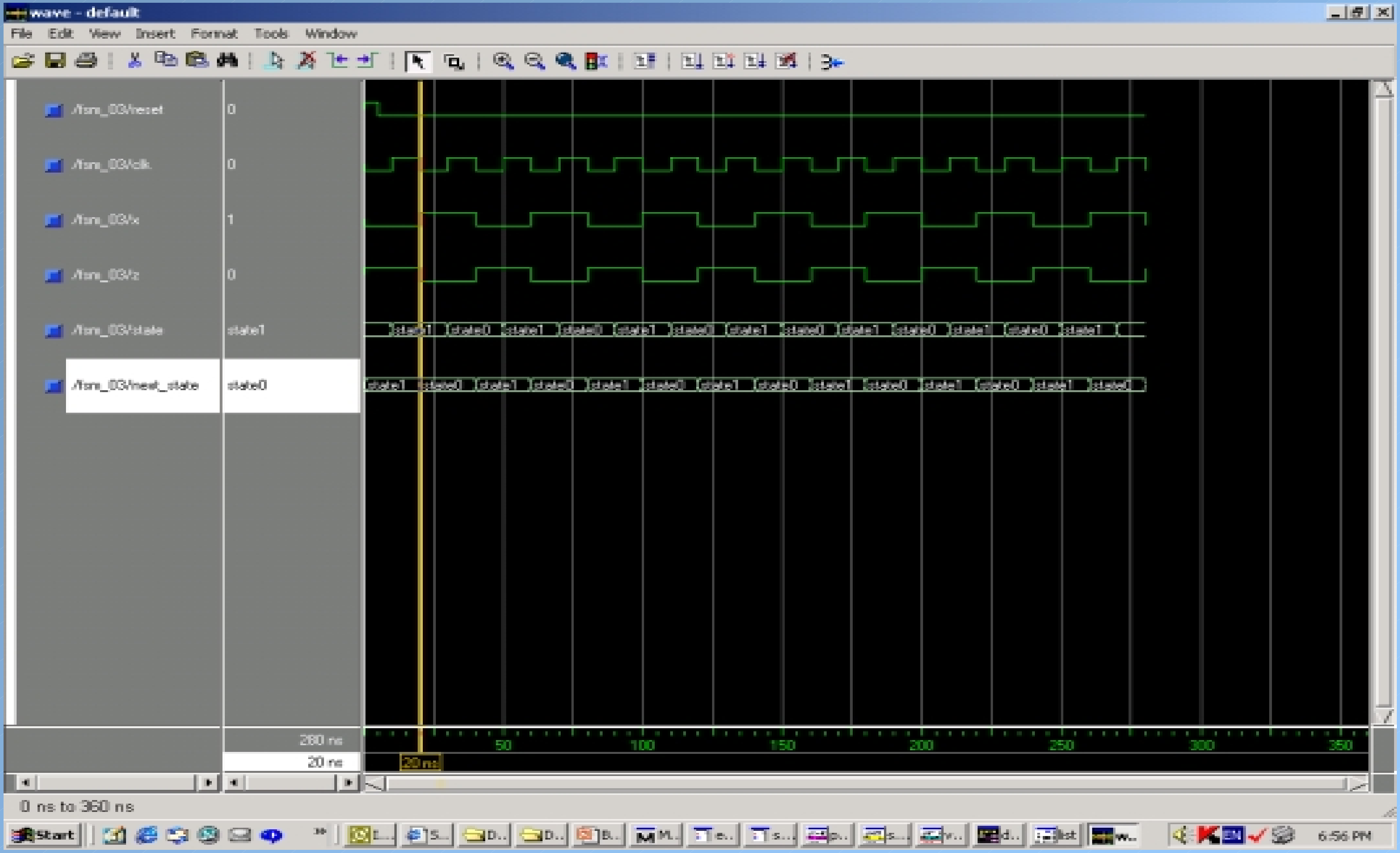
Next State Process

Alternative Model: with Three Process

```
clk_process: process is
begin
  wait until(rising_edge(clk));
  if reset = ‘1’ then state <= statetype’left;
  else state <= next_state;
  end if;
end process clk_process;
end architecture behavioral;
```

Clock Process

Simulation



```
/fsm_03/reset        0
/fsm_03/clk          0
/fsm_03/x            1
/fsm_03/z            0
/fsm_03/state        state1
/fsm_03/next_state   state0

                 280 ns
                  20 ns
0 ns to 360 ns
```

Constructing VHDL Models

```
library library-name-1, library-name-2;
use library-name-1.package-name.all;
use library-name-2.package-name.all;
entity entity_name is
port(input signals: in type;
     output signals: out type);
end entity entity-name;
architecture arch_name of entity_name is
  -- declare internal signals, you may have multiple signals
  -- of different types
signal internal signals: type := initialization;
begin
  -- first process
  -- second process
  -- simple, conditional, or selected CSA
  -- other processes or CSAs
end architecture arch_name;
```

Constructing VHDL Models

```
architecture arch_name of entity_name is
begin
  label-1: process(--sensitivity list--) is
  --declare variables to be used in the process
    variable variable names : type:=initialization;
  begin
  -- process body
  end process label-1;
  label-2: process is
  --declare variables to be used in the process
    variable variable names : type:=initialization;
  begin
    wait until (--predicate--);
    -- sequential statements
    wait until (--predicate--);
    -- sequential statements
  end process label-2;
internal-signal or ports<=simple,conditional,or selected CSA
-- other processes or CSAs
end architecture arch_name;
```

CSA and Process

- They can be in the same architecture body
  - as many CSA or process as we want
- A process and a CSA are concurrently executed with respect to each other.
- We use process when the computation of the signal values are too complex for CSA statements.
  - variables declared in a process' declaration field is only visible within this process.

Common Syntax Errors

1. elsif (not elseif)
2. end if (not endif)
3. no semicolon (;) after then or if-then-elsif construct
4. 10 ns (10ns is incorrect)
5. underscore is OK in label names (not hyphen)
6. Some simulators cannot do the following assignment

```
signal X: std_logic_vector(7 downto 0):="00000000";
```

we may have to use `to_stdlogicvector("00000000")` first to make this assignment work.

Common Run-Time Errors

1. Make sure if you need a signal or variable
   - Signals are updated with the evaluated value in the next execution cycle.
   - Variables are assigned immediately.
2. If multiple drivers exist for a signal, the type of the signal must resolved (i.e. a resolution function must be defined and associated).
3. A process should have either a sensitivity list or a wait statement; not both.
4. All processes are executed once when the simulation is started.

Common Run-Time Errors

5. Be careful of the assigned values of signals

```
wait until rising_edge(clk);
sig_a <= sig_x and sig_y;
sig_b <= sig_a;
```

Think of `sig_a` and `sig_b` are being stored in flip-flops.

Summary

- Processes
- Sequential statements
  - if-then-else, case, loop
- wait statement of sensitivity list
- attributes
- communicating processes
- modeling state machines
- using both CSA statements and processes within the same architecture description