TSTE12 Design of Digital Systems Lecture 4

Kent Palmkvist

LINKÖPING UNIVERSITY

Agenda

- Practical issues
- Short tool overview
- Introduction to VHDL, continued
  - Timing
  - Testbench

TSTE12 Deadlines Y,D,ED

- First meeting with supervisor should happen no later than today!
  - Determine project manager (contact person)
  - Questions (short meeting)
- Lab 1 deadline Wednesday 7 September at 21.00
- Tuesday 6 September: First version of requirement specification
  - We use LIPS ”light”, want to capture expected behavior of final result in requirement specification

TSTE12 Deadlines MELE, erasmus

- Group definition Wednesday 7 September (afternoon)
  - On web, include supervisor assignment
- Friday 9 September: First meeting with supervisor
  - Determine project manager (contact person)
  - Question (short meeting)
- Tuesday 13 September: First version of requirement specification

MUX lab access

- LiU-card should now give access to MUX1 lab
  - Email me if you can not get into the lab
- Lab available 5-23 every day
  - Make sure to verify in schedule server if lab is available outside course schedule
  - MUX1 mostly used only for TSTE12
  - MUX2 also sometimes available (used more by other courses)
- Remote login: use thinlinc

Project issues

- Expected project participation conduct
  - Do no be late to meetings
  - Inform the rest of the group if you have problem attending a meeting (in advance if possible)
  - Keep track of your project work, noting amount and type of task
- Documents should be discussed and approved by supervisor
- Possible to fail project even if design works
- Possible for individual to fail project even if rest of group get a pass!

Project hints

- Hints about Requirement specification
  - Possible subsystem: control, display, audio processing
  - Add plenty of features
  - Set priority (low, medium, high)
  - Avoid multiple requirements in one requirement statement
- Hints about design specification
  - Should indicate idea about general building blocks
    - Interfaces (signals/data to communicate)
    - Behavior

Design flow and tools

- Three types of examined activities in the course
  - Handin
  - Lab
  - Project
- For handins (start next week): use simple text editor + modelsim
  - Start the TSTE12handin shell
  - Write code, compile, simulate, finally upload code
- Chapter 2 tutorial notes shows how to use modelsim
  http://www.isy.liu.se/edu/kurs/TSTE12/kursmaterial/

HDL Designer tool

- Design entry tool, main entry tool to the project
  - Tutorial chapter 3 introduce this www.isy.liu.se/edu/kurs/TSTE12/kursmaterial
- Tools used to manage libraries, design, and other tools for use by larger designer groups
  - Graphic and text design entry
  - Tool startup configurations
  - Support many different languages and tools
  - Version control, team management....
- Highly configurable

HDL Designer tool, cont.

- Top level: The project (defined by xxx.hdp file)
  - Contains list of libraries, (1 or more)
- Each library contains design units
  - Described as components (green and blue boxes)
  - Each unit have different view
    - Graphic and/or textual
    - Various forms of architectures (text, block, FSM, …)
    - A default architecture view is indicated by a blue arrow
- Interfaces with simulation and synthesis tools

HDL Designer tool, cont.

- Green boxes (components)
  - Fixed interface (does not automatically update)
  - Possible to reuse in multiple designs
- Blue boxes (subsystems)
  - Updates interface when adding/removing inputs/outputs in block diagram (remember to save schematic to update VHDL)
- Tools can generate valid VHDL from graphical representation (schematics, state machines, etc.)
- State machine example in lab3 lab material

VHDL timing

- Two types of time in VHDL
  - Variables: no delay in update
  - Signals: standard time delay and/or delta delay
- Delta delay
  - Never adds up to a standard time unit
  - Default delay when assigning signals unless delay is specified
- Known as macro and micro timing

Timing implementation in simulation

- Simulator program flow
  1) If no entries in queue then stop, else increase time to next time entry in queue
  2) Start a new simulation cycle without advancing simulation time. Remove all entries scheduled for current simulation time, update all signals. Activate triggered processes
  3) Execute activated processes. Schedule new time queue entries.
  4) If there are new transactions on signals due to assignment with delta delay, then goto 2, otherwise goto 1
- Concurrent assignment can be seen as processes

Unexpected simulation results

- Time may stand still in simulation by continuous signal updates
  - Example: process triggered by a signal that it is updating
  - Combinatorial loops without macro delay in assignments

Simulation models

- Delta delay only
  - Functional verification of models
- Standard time unit delay only
  - Validate system timing
- Mixed
  - Delta delay where delay is not important
  - Standard time unit delay where delay is significant
  - Study system timing

Example of models

- A simple buffer examples
  - All buffer have different propagation delay
  - Difference in delta delays are difficult to see in waveform windows
  - Possible to create multiple delta delay

```
Entity BUFF is
  port (X: in BIT; Z out BIT);
end;

Architecture ONE of BUFF is
    signal Y: BIT;
begin
    process(X)
        variable Y : BIT;
    begin
        Z <= X;
    end process;
end ONE;
```

```
architecture TWO of BUF is
    signal Y: BIT;
begin
    process(X)
    begin
        Y <= X;
    end process
    Z <= Y;
end TWO;

architecture THREE of BUF is
    signal Y1,Y2: BIT;
begin
    Y1 <= X;
    Y3 <= Y2;
    Y2 <= Y1;
     Z <= Y3;
end THREE;
```

Example models, cont.

- Two almost identical buffers
  - Have very different simulation behavour
  - Both probably generate same hardware in synthesis
- Lacking entries in sensitivity list
  - Solution: Always add all input signals to the sensitivity list
- Drawback: unnecessary process triggering may give slower simulation

```
Architecture FIVE of BUFF is
   signal Y5: BIT;
begin
   process(X)
   begin
      Y5 <= X;
      Z <= Y5;
   end process;
end FIVE;

architecture FIVE_A of BUF is
   signal Y5: BIT;
begin
   process(X,Y5)
   begin
      Y5 <= X;
      Z <= Y5;
   end process
end FIVE_A;
```


Inertial and Transport delay

- Delay can be of two types (3 in VHDL93)
  - Inertial Z <= I after 10 ns;
    - If input change again before end of delay then do not update output
    - Filter out short glitches (RC delay)
  - Transport Z <= transport I after 10 ns;
    - “True” delay of signal (like transmission lines)
  - Reject (VHDL93) Q <= reject 4 ns inertial a after 10 ns;
    - Q_tmp <= A after 4 ns; Q <= Q_tmp after 6 ns;

Implementation of Inertial and Transport delay in simulator

- Important to understand why a signal change may not reach the assigned signal
- Transaction
  - Pair of value and time. What value when
- Waveform
  - A series of transactions (sorted by time value)
- Current value of driver
- Value of transaction whose time is not greater than current simulation time. Removed when simulation time is updated if next transaction time is reached

Waveform update algorithm

1. All old transactions with time at or after earliest new transaction are deleted. Add new transactions to the waveform

If inertial then

- 2. Mark all new transactions
- 3. Mark old transaction if it immediately precedes a marked transition and its value is the same as the marked transaction
- 4. Mark the current value transaction

5. All unmarked transactions are removed

Waveform update example

- Z <= I after 10 ns; (I is a 5 ns pulse starting at t=0)
- First change Z updated to '1' at t=0, (10,'1') transaction added
  - Both current and transaction marked and kept
- Second change, Z updated to '0' at t=5, (15,'0') transaction added
- If inertial: (10,'1') not marked, removed
- End result: the pulse on I is not visible on Z (filtered out)

Inertial delay side effects

- Process for generating reset signal Res
  - Only executed once at start
  - First assignment is eliminated by second assignment
- Use transport or combined assignment to get pulse

  Res <= transport '1' after 50 ns;

  Res <= transport '0' after 100 ns;
- Generate complete waveform instead

  Res <= '1' after 50 ns, '0' after 100 ns;

```
Process
 begin
  Res <= '1' after 50 ns;
  Res <= '0' after 100 ns;
  wait;
 end process;
```

Modeling of combinational and sequential logic

- Simple approach.
  - Process sensitivity list = circuit inputs
  - Compute new value using variables
  - Assign output signal with delay
  - Possible to synthesize (ignoring delay)
- Models uses generic in the port
  - Adds parameters to components without need of a signal
  - May have default values in entity declaration

Combinational logic examples

- Gates
  - Generic states delay
  - May have default delay defined

```
entity NAND2 is
  generic(DEL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end NAND2;

architecture DF of NAND2 is
begin
  O <= I1 nand I2 after DEL;
end DF;
```

```
architecture STRUCTURAL of ONES_COUNT is

component XOR_GATE
port (X,Y : in bit; O : out bit);
end component;

component NAND_GATE
Generic (DEL: TIME := 3 ns);
port (X,Y : in bit; O : out bit);
end component

signal I1, I2, I3 : bit;

begin

U1 : XOR_GATE port map(A(0),A(1),I1);
U2 : XOR_GATE port map(I1,A(2),C(0));
U3 : NAND_GATE generic map(5 ns)
             port map(A(0),A(1),I2);
U4 : NAND_GATE port map(A(2),I1,I3);
U5 : NAND_GATE port map(I2,I3,C(1));

end STRUCTURAL;
```

Combinational logic examples, cont.

- Two-to-4 decoder
  - Set one of the four outputs to ’1’ based on the I input value

```
entity TWO_TO_4_DEC is
  generic(DEL: TIME);
  port(I: in  BIT_VECTOR(1 downto 0);
       O: out BIT_VECTOR(3 downto 0));
end TWO_TO_4_DEC;

architecture ALG of TWO_TO_4_DEC is
begin
  process(I)
  begin
    case I is
      when "00" => O<= "0001" after DEL;
      when "01" => O<= "0010" after DEL;
      when "10" => O<= "0100" after DEL;
      when "11" => O<= "1000" after DEL;
    end case;
  end process;
end ALG;
```


Sequential logic process template

- Must check both event and level to detect clock edge
  - Alternative functions available in the std_l,ogic libraries
    - rising_edge, falling_edge
- Do NOT do the following:
- This is acting as a flip-flop based design, but is synthesized to a latch based one!

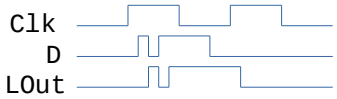
```
process(clk, ...)
begin
  if <async expressions> then
async behavior
  elsif clk'event and clk='1' then
sync behavior
  endif
end process;
```

```
process(clk)
begin
  if clk='1' then
sync behavior
  endif
end process;
```

```
process(clk)
begin
  if clk='1'
then
    Q <= D;
  endif
end process;
```

Sequential logic, cont.

- Latch
  - Latches missing the edge detection
  - Bad design style
  - Synthesis result not working
- Flipflop would only copy D when a positive edge on Clk




```
entity LATCH is
  generic(LATCH_DEL:TIME);
  port(D: in BIT_VECTOR(7 downto 0);
       CLK: in BIT;
       LOUT: out BIT_VECTOR(7 downto 0));
end LATCH;

architecture DFLOW of LATCH is
begin
  LATCH: process(clk,D)
  begin
    If (clk='1') then
      LOUT <= D after LATCH_DEL;
    end if;
  end process;
end DFLOW;
```


Sequential logic

- JK flipflop with asynchronous set/reset
  - Edge trigged using 'event
  - Asynchronous update
    - Higher priority than clocked circuit function
  - Synchronous update
    - Note use of elsif (must be used)
    - Edge trigged using 'event

```
entity JKFF is
  generic(SRDEL,CLKDEL: TIME);
  port(S,R,J,K,CLK: in BIT;
       Q,QN: inout BIT);
end JKFF;
```

```
architecture ALG of JKFF is
begin
  process(CLK,S,R)
  begin
    if S = '1' and R = '0' then
      Q <= '1' after SRDEL;
      QN <= '0' after SRDEL;
    elsif S = '0' and R = '1' then
      Q  <= '0' after SRDEL;
      QN <= '1' after SRDEL;
    elsif CLK'EVENT and CLK = '1' and
          S='0' and R='0' then
      if J = '1' and K = '0' then
        Q <= '1' after CLKDEL;
        QN <= '0' after CLKDEL;
      elsif J = '0' and K ='1'  then
        Q <= '0' after CLKDEL;
        QN <= '1' after CLKDEL;
      elsif J= '1' and K= '1'  then
        Q <= not Q after CLKDEL;
        QN <= not QN after CLKDEL;
      end if;
    end if;
  end process;
end ALG;
```

Sequential logic, cont.

- Register with alternative design
  - Use a guarded statement
  - Use ’STABLE instead of ’EVENT

```
entity REG is
  generic(DEL: TIME);
  port(RESET,LOAD,CLK: in BIT;
       DATA_IN: in BIT_VECTOR(3 downto 0);
       Q: inout BIT_VECTOR(3 downto 0));
end REG;

architecture DF of REG is
begin
  REG: block(not CLK'STABLE and CLK ='1')
  begin
    Q <= guarded "0000" after DEL when RESET ='1' else
         DATA_IN after DEL when LOAD ='1' else
         Q;
  end block REG;
end DF;
```


Output feedback problems

- Entity output can NOT be read in the architecture
- Three solutions
  - Use INOUT
    - Does not match OUT, enables output values to influence internal signal values
  - Use BUFFER
    - Does not match OUT, complicates building testbenches etc.
  - Use OUT with a temporary signal
    - use temporary signal everywhere needed (read and assign), assign entity out signal at the end of the architecture

Sequential logic, oscillator

- Run signal indicate when to start generating clock pulses.
- Feedback example
  - Need extra variable to guarantee complete clock cycles
  - Simulation use only, will not synthesize

```
entity CLOCK_GENERATOR
  generic(PER: TIME);
  port(RUN: in BIT;
       CLK: out BIT);
end CLOCK_GENERATOR;
```

```
architecture ALG of CLOCK_GENERATOR is
  signal CLOCK: BIT;
begin
  process (RUN,CLOCK)
    variable CLKE: BIT := '0';
  begin
    if RUN='1' and not RUN'STABLE then
      CLKE := '1';
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
    if RUN='0' and not RUN'STABLE then
      CLKE := '0';
    end if;
    if CLOCK='1' and not CLOCK'STABLE
                 and CLKE = '1'then
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
    CLK <= CLOCK;
  end process;
end ALG;
```


Sequential logic, oscillator

- Wait statement based
  - Can not have both wait and sensitivity list in process

```
entity COSC is
  generic(HI_TIME,LO_TIME: TIME);
  port(RUN: in BIT; CLOCK: out BIT := '0');
end COSC;

architecture ALG of COSC is
begin
  process
  begin
    wait until RUN ='1';
    while RUN = '1'  loop
      CLOCK <= '1';
      wait for HI_TIME;
      CLOCK <= '0';
      wait for LO_TIME;
    end loop;
  end process;
end ALG;
```

Numeric calculations

- Bit-vectors (and std_logic_vectors) does not directly correspond to a value
  - ”1011” could mean 11 in decimal (unsigned), or -5 in decimal (2’s complement)
- Datatypes are included in supporting packages to enable arithmetic on bit-vectors
  - ieee.numeric_bit.all
  - ieee.numeric_std.all
- Must use defined types signed or unsigned to allow calculations
  - Same definitions as bit_vector and std_logic_vector
  - Can copy values between types due to same element type

Numeric calculations example

- Counter incrementing 3-bit count value each clock cycle
  - Asynchronous reset

```
library ieee;
use ieee.numeric_bit.all;

entity INL3_KB is
 port (
   C : in bit;
   R : in bit;
   Q : out bit_vector(1 to 3));
 end entity;
```

```
architecture KB of INL3_KB is
 begin

  process(C,R)
   variable count : unsigned(1 to 3);
  begin
   if R = '1' then
     count := (others => '0');
   elsif C'event and (C='1') then
     count := count + 1;
   end if;
   Q <= bit_vector(count);
  end process;

end architecture;
```

Numeric calculations details

- Addition does not increment wordlengths
  - May get overflow
  - Must signextend to detect carry

```
  101        0101
 +011       +0011
 ------     --------
  000        1000
```

- Adding different length vectors will sign extend the shortest one
  - May still get overflow
- Multiplication always generates an output number of bits equal to the total number of input bits
  - Multiplying a 3-bit input with a 4-bit input generates a 7-bit output result

Avoid old packages

- Before the introduction of numeric_std and numeric_bit there where other libraries
  - std_logic_unsigned, std_logic_signed
  - std_logic_arith
- Do NOT use these, they are obsolete
  - Made it difficult/impossible to mix signed and unsigned

Including integers

- Integers can be used for synthesis
  - If synthesis tool cannot figure out the limits, the result is 32-bit arithmetic
  - Subtypes (limiting range) help to reduce hardware and catch unexpected use
- Integers will be implemented as bitvectors
  - Either unsigned or signed (2's complement)
  - Translation between integer and bitvectors exist
    x_signed := to_signed(y_int,x_signed'size);
  - Translation other way around (unsigned to integer value)
    y_int = to_integer(x_signed);

Another aspect of signal assignment

- One signal can be assigned from different parts of the code
  - Support multiple entities driving the same wire
  - Example: Databus in a computer connecting multiple memories and CPU
- Modelling must be strict and clear
  - Same result independant of simulator tool
  - Should not be able to detect the order the processes where calculated
- Not all data types support multiple sources for the value

Multiple assignment on one signal

- Each process containing a signal assignment will have a driver in the simulator generating a contribution to the final signal value
  - Concurrent signal assignments will have one driver each
  - Processes only have one driver for each signal (even with multiple assignment)
  - The signal update seen before is done individually on each driver
  - One driver does not know anything about other drivers
- When the value of a signal is fetched, the contributions from the different drivers current values are collected.
  - The resulting signal value depends on the definition of how to combine the values from the different drivers, using a resolution function

Example of data types supporting multiple drivers

- Signals driven by multiple drivers must be resolved
  - Use a special function that resolves multiple drivers
- Resolution function
  - Example: Wired-OR
    - `signal X1 : WIRED_OR Bit;`
    - `subtype STD_LOGIC is RESOLVED STD_ULOGIC;`
    - `signal Y2 : STD_LOGIC;`
  - RESOLVED is the resolution function name
    - Called every time the value of the signal is calculated
    - Gets all driver values as input

Multivalued logic

- Not enough with 0 and 1 to model “real” logic
- Example: Bus
  - Requires bus release
  - Signal assignment driver can not drop its value
  - Use a value to indicate not driven, and indicate non-driven signals (Z)
  - Need to indicate conflicting driver (X)

Multivalued logic in VHDL

- Alternative to data type BIT

```
Type MVL4 is (’X’, ’0’, ’1’, ’Z’);
Type MVL4_VECTOR is array(NATURAL
  range <>) of MVL4;
```

- X leftmost to make it the initial value unless explicitly initialized in the code

Multidriver signals

- Requires a resolution signal
- Different combinations possible
  - X always overrides others
  - 0 and 1 at the same time gives X
  - Z and Z gives Z

Resolution function definition

```
Subtype DotX is wiredX MVL4;
```

- WiredX is the name of the resolution function

```
Function WiredX (V:MVL4_VECTOR)
  return MVL4;
```

- Where V is a vector containing all values of all drivers of a signal

Resolution function implementation

- Implement as a loop and lookup table

```
Function wiredX (V: MVL4_VECTOR) return MVL4 is
   Variable result: MVL4:= 'Z';
Begin
  For i in V'RANGE loop  -- range not known in advance
    Result = table_WIREDX(result,V(i));
    Exit when result = 'X';
  End loop;
  Return result;
End wiredX;
```


Resolution function impl., cont.

- Check of X in loop is not necessary, but speed up simulation
- Table should then look like:

```
Type MVL4_TABLE is array (MVL4, MVL4) of MVL4;
Constant table_WIREDX : MVL4_TABLE :=
--
--         X    0    1    Z
--
        (('X', 'X', 'X', 'X'),  --    X
         ('X', '0', 'X', '0'),  --    0
         ('X', 'X', '1', '1'),  --    1
         ('X', '0', '1', 'Z')); --    Z
```

Resolution function impl. Cont.

- Table lookup may be used for most functions
  - Not possible to know the order of the value in V, may therefore require a more complex algorithm