TSTE12 Design of Digital Systems Lecture 9

Kent Palmkvist

Agenda

- Practical issues
- Design process
  - FPGA vs ASIC
- Code style

TSTE12 Deadlines Y,D,ED

- Weekly meetings should have started
  - Internal weekly meeting with transcript sent to supervisor
- Project completion
  - Friday 20 October
  - Presentation
  - Project report

TSTE12 Deadlines MELE, erasmus

- Design sketch, project plan, time plan
  - What building blocks in the design (design sketch)
  - Who and when should these be implemented (project plan, time plan)
- Wednesday 20 September 21.00: Lab 2 soft deadline
  - Lab 2 results will be checked after project completed

Handin (homework), Individual!

- 1st handin deadline today Monday 18 September 23:30
- Use only plan text editor (emacs, vi, modelsim or similar) for code entry.
- Solve tasks INDIVIDUALLY
- Submit answers using Lisam assignment function
  - 4 different submissions for code, one for each code task
  - 1 submission for all theory question answers
- Use a special terminal window when working with handins

  module load TSTE12 ; TSTE12handin

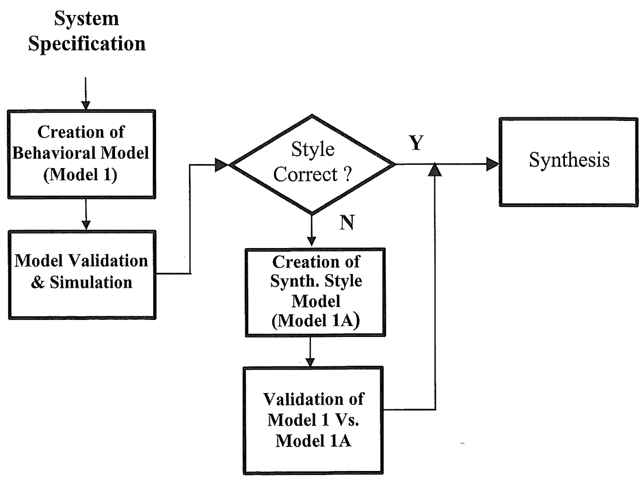
Design process

- Best would be to write a direct synthesizable model direct
  - Hard to do
- First create executable model
  - Validate system (check for correct behavior)
  - Use complex data types, real values
  - Not synthesizable, may use full power of the VHDL language

09/17/2023 22:14

Design process, cont.

- Often use an iterative design flow
- First model is a behavioral model
  - Check against customer requirements
  - Not interested of synthesis, use all available VHDL language constructs
  - Create a testbench



Design process, cont.

- Model 1A (after modification to match expected code style)
  - Synthesizable
  - Fixed point number systems
  - Limited memory size
- Difference in behavior
  - Noise like errors in signal processing systems
  - Timing differences
  - Need to know the effect of these errors on the overall behavior
  - Need to know what can be and not be done in the model, i.e., application area knowledge is needed, not only implementation in general (Karnough maps, VHDL etc.)

Application Specific vs Language

- Application specific
  - Use description formats common in the application domain
  - Models often simulated and/or translated to other computer languages
  - Example representations
    - Dataflow diagram, e.g., DSP
  - Tools
    - SPW, Simulink (Matlab), DSP station, DSP builder
  - Only suitable for the application domain
    - Demonstrate working algorithm in simulation
  - Often supports statistical calculations to evaluate performance reduction due to limited wordlength etc.
- Describe operations and how they communicate
  - Not every block corresponds to a hardware block, only describes a function

Language-Domain modeling

- Models described in a computer language instead of graphical entry
  - System-C, VHDL, Verilog, C++, Java
- Hierarchy important to reduce complexity of the description
- Application specific information must be added by the designer
  - No/little help with application specific functions
- Support any application domain

Comparison

- Application domain
  + Well defined, correct functionality. Fast and easy to verify functionality. No need to understand language details
  - Not very optimal/efficient if models not directly connected to the intended application area. Covers only a limited set of applications
- Language domain
  + Can be used for any application domain
  - Specific measures, tests or constructs common to a particual application domain require explicit adding to the system

Synthesis and simulation

- Synthesis style is tools dependent
  - Something working in one tool may not work in another tool!
  - Continuous development, new features added in each new release
  - A standard also exist specifying a common set of expected synthesis constructs
    - Lower limit of features, tools may support other/additional language features
- Wordlength and data types: Real -> Integer -> bitvectors
  - Real values must first be translated into integer computations
  - Integer computations must be translated into bitvectors of limited length

ASIC design flow (standard cell)

- Behavoural model development
- Behavoural model validation
  - testbench design
- Logic synthesis
- Post synthesis simulation
  - gate delay, no wire delay alternatively only a coarse wire delay estimation
- System partitioning
  - divide into chips or large blocks on chip
  - I/O is limiting chip size and data speed

ASIC design flow, cont.

- Floor planning
  - where to put modules/subsystems on chip
- Placement
  - detailed description on where each cell is placed on the chip
- Routing
  - connect cells with wires
  - Clock tree, power routing
- Circuit extraction
  - extract more detailed timing from circuit

ASIC design flow, cont.

- Post layout simulation
  - including wire capacitance, cross talk etc.
  - Verify function for all combinations of manufacturer and environment tolerances (fast, slow, typical transistor speed, high/low voltage, high/low temperature, etc.)
- Send masks to manufacturer
  - One or more masks for each type of layer on the chip (doping, metal, etc.)
  - Turn around time at least 4 weeks, probably 1-3 month
- Evaluate recieved circuit

FPGA design flow

- Behavoural model development
- Behavoural model validation (testbench)
- Logic synthesis
  - Slightly different goal structure (lookup tables and flipflops) for FPGA
- Mapping to CLBs
  - What logic and flipflop to combine into one unit
- Placement
  - Select one of a large set of
- Routing
  - Select wire segment in space between CLBs for connecting them together
- Circuit level extraction
- Post layout simulation
- Generation of a POF/SOF/BIT file

Design manager design flow (Xilinx)

- Translate: Convert to local database format. Some mapping into technology dependent mappings (e.g., memories).
- Map: Allocate CLB, IOB, etc.
- Place & route: Place and route, timing limitations may be included.
- Timing: Extract timing. Performed through static timing analysis (Sum contributing delays from flip-flop outputs to flip-flop inputs).
- Configure: Translate layout information into a POF/SOF (bit) file to program the FPGA. May be stored in ROM or load through a processor/PC.

Synthesis design flow Precision logic

- Analyse
  - Parse HDL
  - Find libraries and cells
  - Check dependencies
  - Resolve generics
- Elaborate
  - Translate into a generic RTL + black box operators
  - Create hierarchy, infer flipflops & latches, memory, operators, FSM
- Pre-optimization
  - Boundary optimization
    - propagating constants, remove unused outputs, shared input signals
  - Constant propagation
  - Resource sharing

Synthesis design flow Precision logic, cont.

- Operator implementation
  - Adders, counters etc.
- Hierarchy manipulations
  - Flatten
- Tristate handling
- DRC checking (Design Rule Checking)
  - Short circuits, multiple output driving one node etc.
- Technology mapping
- Register retiming

Control of the synthesis process

- Additional information required by synthesis
  - Pin assignment
  - Timing requirements
  - General placement information
  - Precompiled netlists
- VHDL attributes
  - No standard yet
- Synthesis tool control scripts
  - Tools dependent
  - Optimization, hierarchy

09/17/2023 22:14

Syntheis example

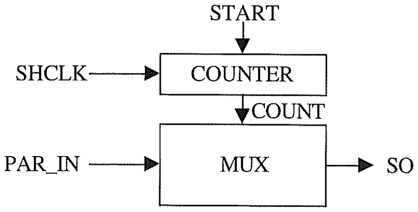
- Parallel to serial converter
- Shift out parallel input data from PAR_IN onto SO once START = '1'
- Lower abstraction level, bit datatypes

```
Library ieee;
Use ieee.std_logic_1164.all;

entity PAR_TO_SER is
Port(
  START,SHCLK: in STD_LOGIC;
  PAR_IN: in STD_LOGIC_VECTOR(7 downto 0);
  SO: out STD_LOGIC);
end PAR_TO_SER;
```


Hardware engineer view of the implementation

- Counter and multiplexer

```
Library ieee;
Use ieee.std_logic_1164.all;

entity PAR_TO_SER is
Port(
  START,SHCLK: in STD_LOGIC;
  PAR_IN: in STD_LOGIC_VECTOR(7 downto 0);
  SO: out STD_LOGIC);
end PAR_TO_SER;
```




```
architecture ALG1 of PAR_TO_SER is
 begin

P1:process(START,SHCLK)
   variable COUNT: INTEGER range 7 downto -1 := 0;
   variable DONE: BOOLEAN;
begin
   if  START = '1' then
     COUNT := 7;
     DONE := FALSE;
   elsif SHCLK'EVENT and SHCLK = '1'  then
     if DONE = FALSE then
      SO <= PAR_IN(COUNT);
      COUNT := COUNT - 1;
     end if;
     if COUNT < 0 then
      DONE := TRUE;
     else
      DONE  := FALSE;
     end if;
   end if;
end process;
end ALG1;
```

Programmer implementation

- Uses waveform assignment with delay information
- Same behavior, less obvious how to implement

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity PAR_TO_SER_SCHED is
generic(PERIOD: TIME);
Port(
  START: in STD_LOGIC;
  PAR_IN: in STD_LOGIC_VECTOR(7 downto 0);
  SO: out STD_LOGIC);
end PAR_TO_SER_SCHED;
```

```
architecture ALG2 of PAR_TO_SER_SCHED is
begin
P1:process(START)
   variable COUNT: INTEGER;
begin
   if  START = '1' then
     COUNT := 7;
     while COUNT >= 0 loop
     SO <= transport PAR_IN(COUNT)
                       after (7-COUNT)*PERIOD;
     COUNT := COUNT - 1;
     end loop;
    end if;
end process;
end ALG2;
```


Sensitivity list issues

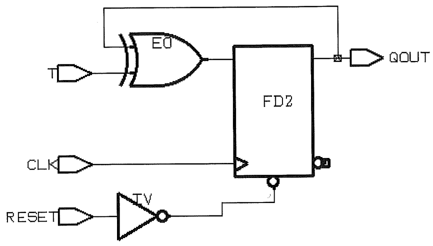
- Used in simulation to trigger processes
- In synthesis it only indicates inputs, often without affecting the synthesis
- Example:
  - Different simulation
  - Same synthesis result

```
architecture ALG of T_FF is
signal Q: STD_LOGIC;
begin

process(RESET,T,CLK)
 begin
   if (RESET = '1') then
      Q  <= '0';
   elsif (CLK'EVENT and CLK = '1') then
      if T = '1' then
      Q  <=  not Q ;
      end if;
   end if;
end process;

QOUT <= Q;
end ALG;
```

```
architecture ALG of T_FF2 is
signal Q: STD_LOGIC;
begin

process(RESET,T,CLK)
 begin
   if (RESET = '1') then
      Q  <= '0';
   elsif (CLK'EVENT and CLK = '1') then
      if T = '1' then
      Q  <=  not Q ;
      end if;
   end if;
   QOUT <= Q;
end process;

end ALG;
```

Example T-flipflop

- Different behavior in the two models
  - Output delayed in 2nd code due to missing Q in sensitivity list
- Synthesis can generate the same results
  - Flipflop with exor gate in feedback
- Delay
  - Can not use an assignment “after xx ns”, only wait for an event (on a clock)
  - Wait statements for fixed delay does not make sense



Data types

- Std_logic is prefered
  - Helps finding reset issues and similar
- Bit works, but the synthesized model will use std_logic
  - Testbenches require changes to support run of synthesis netlist

Clock detection

- CLK'EVENT AND CLK='1'
  - Do not use additional enable signals in the clock edge detection
- Exists also 'RISING_EDGE and 'FALLING_EDGE
  - Handles also L, H, and Z in the expected way (H->1 no edge, 0->H edge!)
- Synchronous/asynchronous reset/set

```
IF asyncexpression THEN
  -- async reset & init
elsif clockdetection
  -- sync expressions
end if;
```


Gated clocks

- Generally not a good idea
  - Glitch in control signal may produce glitch on clock!
  - Wrong timing on control signal may give errornous trigger
  - Clock buffers may introduce large delays
    - Less time left for the calculation of the control signal value
- Must not combine clock edge detection with logic

  ~~if clk'event and clk='1' and enable = '1' then~~

```
if clk'event and clk = '1' then
  if enable = '1' then
```

- Some hardware supports gated clocks
  - Special forms of flipflops

Reset of internal states

- What to do if no asynchronous reset?
  - Initial data must be clocked in using a control signal
- Code example without reset
  - Works in simulation due to initialisation of TEQDET
- Simulation of synthesis error due to initialisation to 'U'

```
entity EQDET is
Port(
  I,CLK: in STD_LOGIC;
  TEQDET: inout STD_LOGIC :='0');
end EQDET;

architecture ALG of EQDET is
  begin
  process
    variable  EQ,IBK1,IBK2: STD_LOGIC;
    begin
    wait until (CLK'EVENT and CLK = '1');
      if(IBK1 =IBK2) and (IBK2 = I) then
        EQ := '1';
       else
        EQ := '0';
      end if;
      TEQDET <= (EQ xor TEQDET);
      IBK2 := IBK1;
      IBK1 := I;
  end process;
end ALG;
```


Using explicit reset

- Asynchronous reset
- Possible to use synchronous reset instead

```
entity EQDET is
Port(
  RESET,I,CLK: in STD_LOGIC;
  TEQDET: inout STD_LOGIC);
end EQDET;
```

```
architecture ALG of EQDET is
  begin
  process(RESET,CLK)
    variable  EQ,IBK1,IBK2: STD_LOGIC;
    begin
      if (RESET = '1') then
        IBK1 :=  '0';
        IBK2 :=  '0';
        TEQDET <= '0';
      elsif (CLK'EVENT and CLK = '1') then
        if (IBK1 = I) and (IBK1 = IBK2) then
          EQ := '1';
        else
          EQ := '0';
        end if;
        TEQDET <= (EQ xor TEQDET);
        IBK2 := IBK1;
        IBK1 := I;
      end if;
  end process;
end ALG;
```

Simulation and Synthesis results

- Order of IBK1 and IBK2 updates are important if variables are used
- Update order not important if signals are used
  - EQ still a variable!
- Both descriptions give same synthesis result

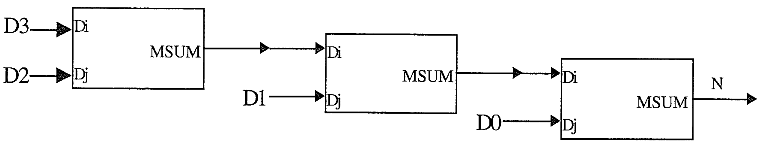
```
architecture ALG of EQDET is
  signal IBK1,IBK2: STD_LOGIC;
  begin
  process(RESET,CLK)
    variable  EQ: STD_LOGIC;
    begin
       if (RESET = '1') then
        IBK1 <=  '0';
        IBK2  <=  '0';
          TEQDET <= '0';
       elsif (CLK'EVENT and CLK = '1') then
        if (IBK1 = I) and (IBK1 = IBK2) then
         EQ := '1';
        else
         EQ := '0';
        end if;
        TEQDET <= (EQ xor TEQDET);
        IBK1 <= I;
        IBK2  <= IBK1;
       end if;
  end process;
end ALG;
```


Arithmetic operations

- Add, sub supported
  - Translates into full adder before simplified
  - Operands are not extended
- Multiplication
  - Translated into combinational expressions
  - Multiple possible structures: Wallace, Carry Save array.
  - Constant values usually produces add and shift implementations (simplified multiplications)
- Division usually not supported

Hierarchical arithmetic: BCD to binary conversion

- Want to implement a 4 digit BCD to binary converter
  - describe decimal number using 4 bits for each digit
- Use Horners rule: $d_3x10^3 + d_2x10^2+d_1x10+d_0=(d_3x10+d_2)x10+d_1)x10+d_0$, i.e., by arbitrary length converter can be built by repeated multiplication by 10 and addition
- Implement the multiply add



Multiply and add operators

- Use unsigned datatype

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity MULT10 is
port(DATA_IN: in STD_LOGIC_VECTOR(3 downto 0);
     PRODUCT: out STD_LOGIC_VECTOR(7 downto 0));
end MULT10;

architecture ALG of MULT10 is
  begin
  process(DATA_IN)
    variable PROD_US: UNSIGNED(7 downto 0);
  begin
    PROD_US :=
          UNSIGNED(DATA_IN)*10;
    PRODUCT <= STD_LOGIC_VECTOR(PROD_US);
  end process;
end ALG;
```

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity SIMP_ADD is
port(A,B: in STD_LOGIC_VECTOR(3 downto 0);
     CIN: in STD_LOGIC;
     C: out STD_LOGIC_VECTOR(3 downto 0);
     CAR_OUT: out STD_LOGIC);
end SIMP_ADD;
architecture ALG of SIMP_ADD is
  begin
  P1:process(A,B,CIN)
    variable  PADDED_CIN: STD_LOGIC_VECTOR(3 downto 0);
    variable A_UNSIGNED: UNSIGNED(3 downto 0);
    variable C_UNSIGNED: UNSIGNED(4 downto 0);
  begin
     A_UNSIGNED := UNSIGNED(A);
     PADDED_CIN  :="000"&CIN;
     C_UNSIGNED  := (A_UNSIGNED(3) & A_UNSIGNED,5) +
              UNSIGNED(B) +  UNSIGNED(PADDED_CIN);
     C  <= STD_LOGIC_VECTOR(C_UNSIGNED(3 downto 0));
     CAR_OUT  <= C_UNSIGNED(4);
   end process;
end ALG;
```

Combined add and mult

- Varying word length

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity MADD is
generic(IN_WIDTH: NATURAL := 4);
port(DI: in STD_LOGIC_VECTOR(IN_WIDTH-1 downto 0);
     DJ: in STD_LOGIC_VECTOR(3 downto 0);
     MSUM: out STD_LOGIC_VECTOR(IN_WIDTH+3 downto 0));
end MADD;

architecture ALG of MADD is
  begin
  P1: process(DI,DJ)
    variable MSUM_US: UNSIGNED(IN_WIDTH+3 downto 0);
    variable PROD:UNSIGNED(2*IN_WIDTH-1 downto 0);
  begin
    PROD :=  UNSIGNED(DI)*to_unsigned(10,IN_WIDTH);
    MSUM_US := PROD(IN_WIDTH+3 downto 0)+ UNSIGNED(DJ);
    MSUM <= STD_LOGIC_VECTOR(MSUM_US);
  end process;
end ALG;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity BCDCONV is
port(D0,D1,D2,D3: in STD_LOGIC_VECTOR(3 downto 0);
     BIN_OUT: out STD_LOGIC_VECTOR(15 downto 0));
end BCDCONV;

architecture STRUCTURAL of BCDCONV is
component MADD
generic(IN_WIDTH: NATURAL := 4);
port(DI: in STD_LOGIC_VECTOR(IN_WIDTH-1 downto 0);
     DJ: in STD_LOGIC_VECTOR(3 downto 0);
     MSUM: out STD_LOGIC_VECTOR(IN_WIDTH+3 downto 0));
end component;
signal MSUM2: STD_LOGIC_VECTOR(7 downto 0);
signal MSUM1:  STD_LOGIC_VECTOR(11 downto 0);
begin
C1: MADD
  generic map(4)
  port map(D3,D2,MSUM2);
C2: MADD
  generic map(8)
  port map(MSUM2,D1,MSUM1);
C3: MADD
  generic map(12)
  port map(MSUM1,D0,BIN_OUT);
end STRUCTURAL;
```


Hierarchical circuit synthesis

- Ungrouping
  - remove artificial boarders between blocks
  - Allows optimize common subcalculation
  - Improves synthesis results
  - Example BCD: 342 -> 309 cells and 30.34 ->30.11 ns delay.
- Uniquify
  - Create different instances different implementations by repeating netlists
  - Allows different optimization of different parts

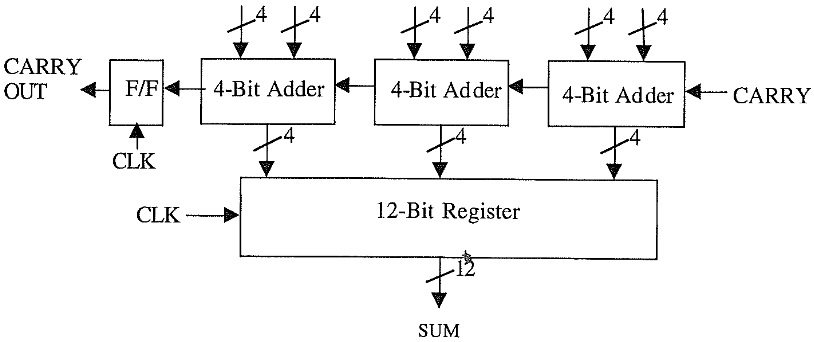
09/17/2023 22:14

Hierarchical Approach

- Bottom up
  - uniquify
  - Build each sub block, then combine
  - Requires good estimate of timing requirement
- Top down
  - Synthesize all to get initial requirements
  - Resynthesize parts not meeting requirements
- Golden instance
  - Synthesize one block, reuse

Example: 12 bit adder register

- Design based on the 4-bit adder
- Different requirement on sum and carry speed



Example: 12 bit adder register, cont.

- Top-down
  - Area 255, 8.84 ns
  - Difficult to know which part require more propagation time
- Bottom-up
  - Area 277, 8.38 ns
  - Some circuit overdesigned, hard to know before full circuit
- Golden instance
  - Area 254, 11.19 ns
  - One size does not fit all...

Inferred latches and don’t cares

- Synthesis may find that latches are needed
- Example: incomplete if

```
PROCESS(a,b,c,d)
  BEGIN
    IF (a = '1') THEN
      out_sig <= x;
    ELSIF (b = '1') THEN
      out_sig <= y;
    ENDIF;
  END PROCESS;
```

- out_sig not defined if a and b = 0! Require latch!

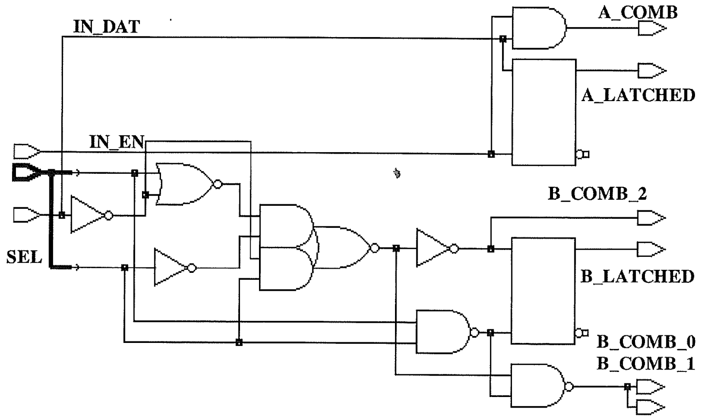
09/17/2023 22:14

Latch and undefined examples (SEL=11 not expected)

```
entity INFERRED is
port(IN_DAT,IN_EN: in STD_LOGIC; SEL: in STD_LOGIC_VECTOR(1 downto 0);
   A_LATCHED,A_COMB,B_LATCHED,B_COMB_0,B_COMB_1,B_COMB_2: out STD_LOGIC);
--pragma dc_script_begin
--set_flatten true
--pragma dc_script_end
end INFERRED;
architecture ALG of INFERRED is
begin
P_A_LATCHED: process(IN_DAT,IN_EN)
 begin
  if IN_EN = '1' then
    A_LATCHED <= IN_DAT;
  end if;
end process;
P_A_COMB: process(IN_DAT,IN_EN)
 begin
  if IN_EN = '1' then
    A_COMB <= IN_DAT;
  else
    A_COMB <= '0';
  end if;
end process;

P_B_LATCHED: process(IN_DAT,SEL)
begin
  case (SEL) is
    when "00" => B_LATCHED <= IN_DAT;
    when "01" => B_LATCHED <= not
IN_DAT;
    when  "10" => B_LATCHED <= '0';
    when  "11" =>  null;
    when others => null;
  end case;
end process;
P_B_COMB_0: process(IN_DAT,SEL)
begin
  case (SEL) is
    when "00" => B_COMB_0 <= IN_DAT;
    when "01" => B_COMB_0 <= not IN_DAT;
    when  "10" => B_COMB_0 <= '0';
    when  "11" => B_COMB_0 <= '1';
    when others => null;
  end case;
end process;

 P_B_COMB_1: process(IN_DAT,SEL)
  begin
  B_COMB_1 <= '1';
  case (SEL) is
    when "00" => B_COMB_1 <= IN_DAT;
    when "01" => B_COMB_1 <= not IN_DAT;
    when  "10" => B_COMB_1 <= '0';
    when  "11" => null;
    when others => null;
  end case;
end process;
P_B_COMB_2: process(IN_DAT,SEL)
  begin
   case (SEL) is
    when "00" => B_COMB_2 <= IN_DAT;
    when "01" => B_COMB_2 <= not IN_DAT;
    when "10" => B_COMB_2 <= '0';
    when "11" => B_COMB_2 <= '-';
    when others => null;
  end case;
 end process;
end ALG;
```


Synthesis results

- Synthesis sometimes generate latches



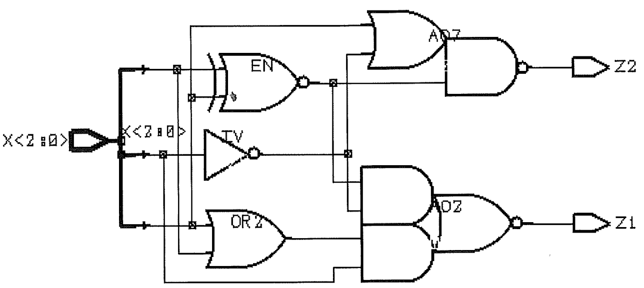
09/17/2023 22:14

Latch problem examples

- Latches can be fixed by
  - Add an assignment in all choices of a case
  - Add a default assignment before case
  - Use don't care symbol '-' to indicate non-important value
- Using a fixed value may use a non-efficient one
  - Use don’t care instead
  - Better let the tool know about unknown
  - Help reduce area and speed up synthesis

ROM-structure with don’t care

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity FUNCS is
port(X: in STD_LOGIC_VECTOR(2 downto 0); Z1,Z2: out STD_LOGIC);
end FUNCS;

architecture ROM of FUNCS is
type ROM_1D is array(0 to 7) of STD_LOGIC;
begin
FULLY_SPECIFIED: process(X)
  constant ROM1: ROM_1D:= "01101000";
  begin
    Z1 <=ROM1(CONV_INTEGER(X));
end process;
PARTIALLY_SPECIFIED: process(X)
  constant ROM2: ROM_1D:= "01101--0";
  begin
    Z2 <=ROM2(CONV_INTEGER(X));
end process;
end ROM;
```

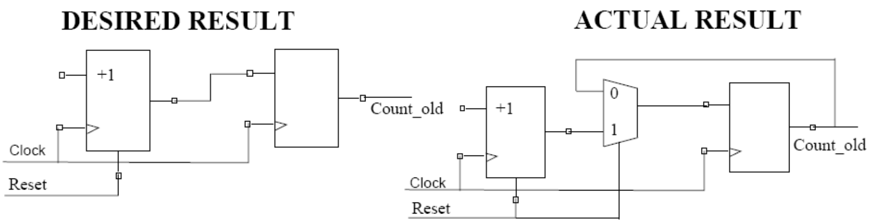


Reset problem

- Counter with delay that should set count_old to zero while being reset?

Count_old not reset!

```
PROCESS (clk, reset);
BEGIN
  if (reset = '0') then
    count <= 0;
  elsif rising_edge(clk) then
    count_old <= count;
    count <= count + 1;
  end if;
end process;
```

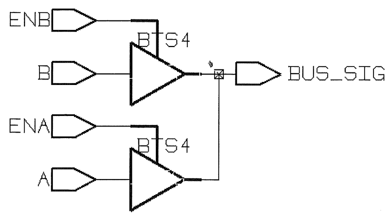


Tristate gates

- Some technologies does not support tristate internally in the design
- Floating wires may produce high power consumption due to short circuit current in inputs
- Possible to change a tristate version into a multiplexer based version (done automatically by some tools)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity TRISTATE is
port(A,B,ENA,ENB: in STD_LOGIC;
      BUS_SIG: out STD_LOGIC);
end TRISTATE;

architecture  ALG of  TRISTATE is
  begin
PROCA: process(A,ENA)
  begin
  if (ENA = '1') then
     BUS_SIG <= A;
  else
     BUS_SIG <= 'Z';
  end if;
end process;
```

```
PROCB: process(B,ENB)
  begin
  if (ENB = '1') then
     BUS_SIG <= B;
  else
     BUS_SIG <= 'Z';
  end if;
end process;
end  ALG;
```



Clock buffers and other aspects

- Attributes used to indicate clock signals
  - Information used to select special layout methods or hardware resources to reduce clock skew
  - Automatically detected in general
- High fanout signals
  - Buffer cells will be added
- Logic duplications
  - Allow larger fan-out without adding separate buffers
- Retiming/pipelining
  - Switch order between calculation and storage
- Multipliers/DSP blocks

Resource sharing

- Chose one of two sums. May add both or chose inputs first
  - Mux+add => 51 area, 8.47 delay
  - Add+mux => 73 area, 7.09 delay
- Flattening and structure. (logic level, not hierarchy)
- Logic can be flattened to e.g., two levels instead of three. Different results of area and logic

How is timing requirements defined?

- Often derived from a symbolic clock
- Signals are defined from edges of the clock
  - Fix setup and hold time. Include clock skew
- Usually defined as maximum delay
  - Expensive to guarantee minimum delay
  - Delay pin to flipflop, flipflop to pin
  - Time from flipflop to flipflop
- Possible to specify multi cycle delay
- False paths

Results

- Time reports
  - Generated by analysis of netlist/layout
  - Critical path reports
- Area reports
- Resource reports
  - Routing, flipflops, LUT, multipliers etc.
- VHDL simulation models
  - Post synthesis, post layout
- Layout possible to modify (edit at bit level)

Synthesis operation

- Synthesis is based on different types of pattern matching
  - Support most constructs
  - Behavour may still be different
  - Often adds complicated patterns that are then simplified
- Example: D flip flop with Qinvers output, but without Q in the sensitivity list. Generally generates a single flipflop, but timing of Qinvers differs between simulation of VHDL and synthesized design.

Recommended patterns

- Style guide exists (patterns)
  - Specific to the synthesis tools
- Specify patterns that are allowed and recommended
  - Important to produce efficient implementations
  - Example units: counters, memories, tristate buffers
- These manuals are available online

LINKÖPING UNIVERSITY

LINKÖPING
UNIVERSITY