Parallel Data Processing in Reconfigurable Systems

Lecture 1

2014/2015

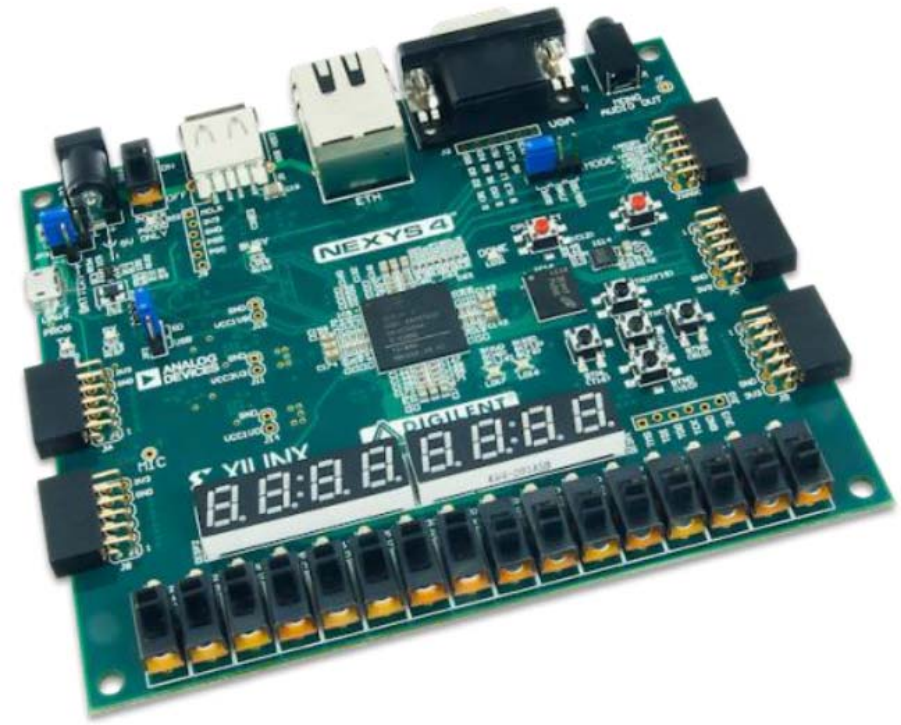
Introduction to Nexys-4 prototyping board and Vivado Design Suite

Iouliia Skliarova

Contents

- Brief overview of Nexys-4 prototyping board
- Introduction to Vivado Design Suite
  - understand the design flow
  - create and debug VHDL designs
  - configure the FPGA
- Development of test benches in VHDL and simulation in Vivado
  - perform simulation verification
- Specification of embedded and distributed memories
  - describe memories in VHDL
  - create and integrate IP cores into design flow using IP Catalog
  - initialize memories from files
- Useful modules for lab 1
  - clock divider
  - debouncer
  - 7-segment display control

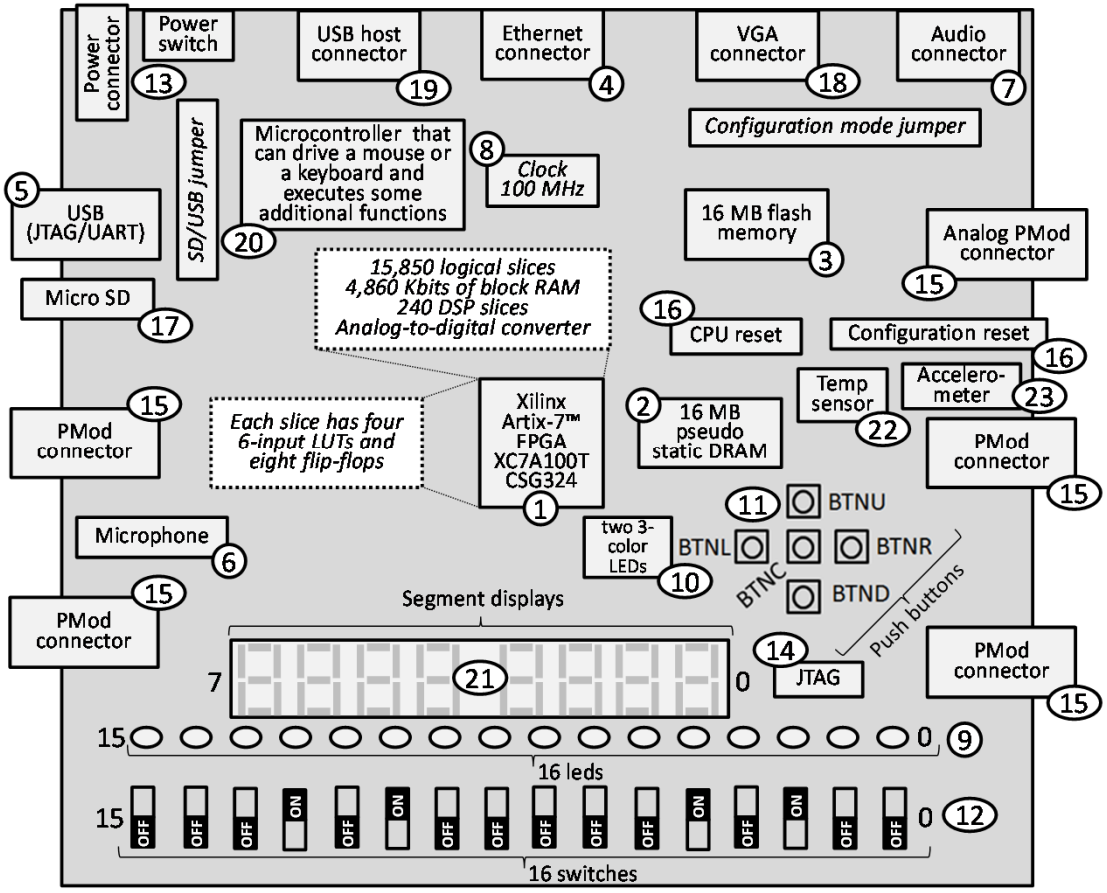
Nexys-4 board



- Manufactured by Digilent
- Contains an FPGA of Artix-7 family
- Configured directly from Vivado through USB JTAG/UART

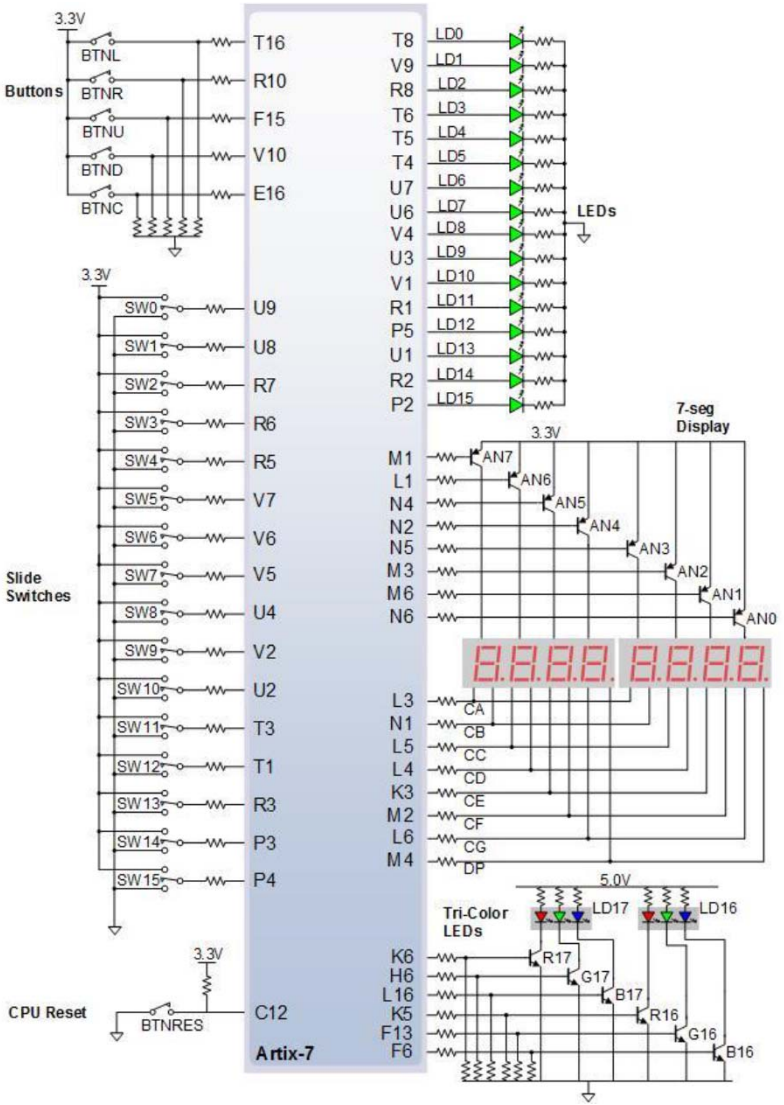
Picture from Nexys4™ FPGA Board Reference Manual

Nexys-4 components



1. Xilinx Artix-7™ FPGA xc7a100t-1csg324
2. 128Mb=16MB cellular RAM
3. 128 Mb=16MB SPI (quad-SPI) Flash
4. 10/100 Ethernet
5. USB-JTAG programming and USB-UART
6. Microphone
7. Audio connector
8. 100 MHz clock oscillator
9. 16 user LEDs
10. Two 3-color user LEDs
11. 5 user buttons
12. 16 slide switches
13. Power connector and power-on LED indicator
14. JTAG port
15. Pmod expansion connectors (2×6)
16. Two reset buttons
17. Micro SD card slot
18. VGA connector
19. USB host connector
20. Microcontroller
21. Eight 7-segment displays
22. Temperature sensor
23. Accelerometer

Basic I/O



Picture from Nexys4™ FPGA Board Reference Manual

- The five **pushbuttons** generate a low output when they are at rest, and a **high output** only **when** they are **pressed**.
- **Slide switches** generate constant high or low inputs depending on their position.
- The sixteen individual **LEDs** are anode-connected to the FPGA via resistors, so they will **turn** on when a logic **high voltage** is applied to their respective I/O pin.
- The anodes of the seven LEDs forming each 7-segment display digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate. The common anode signals are available as eight “digit enable” input signals to the 8-digit display. The cathodes of similar segments on all the displays are connected into seven circuit nodes. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted. To illuminate a segment, both the **anode** and the **cathode** should be driven **low**.

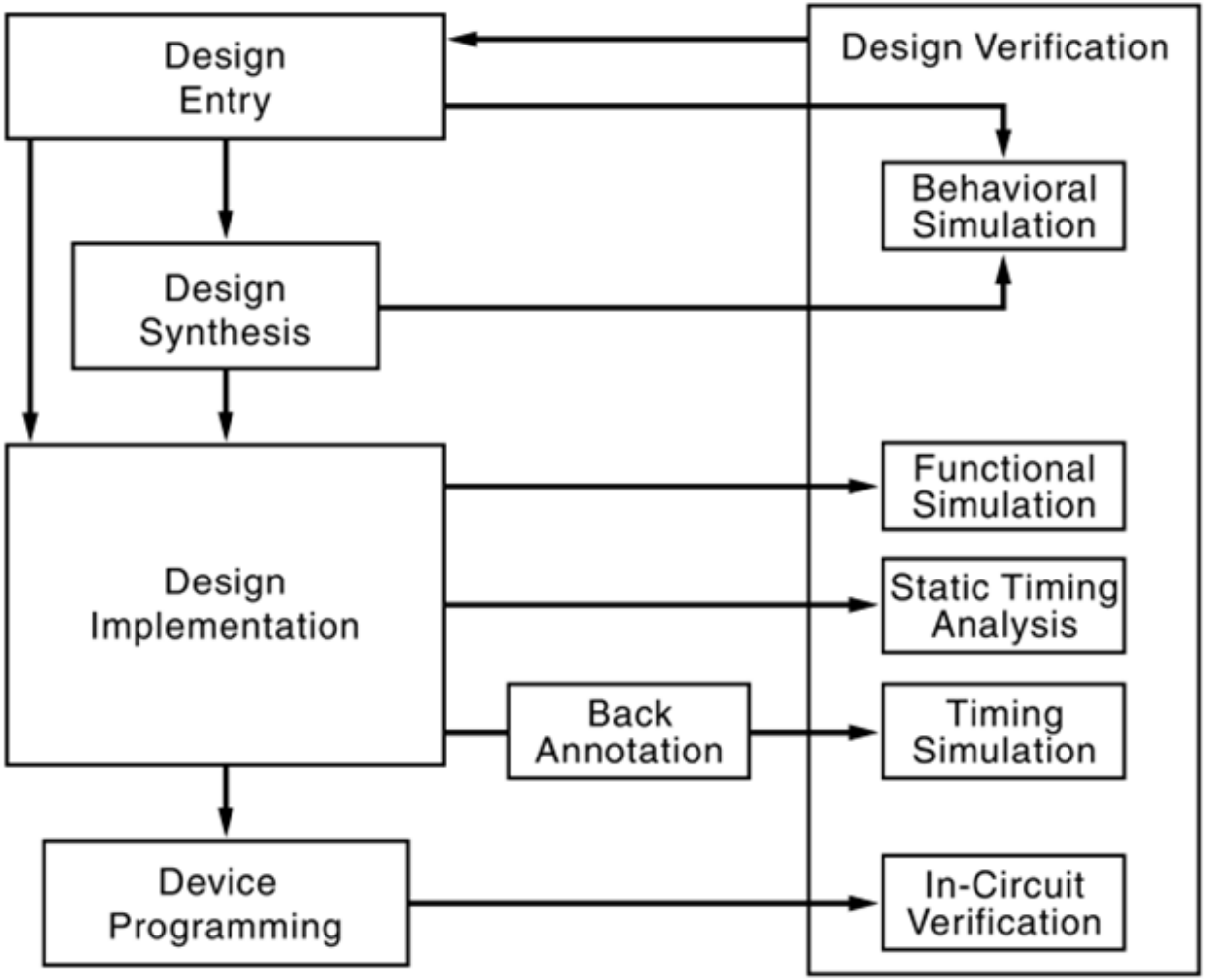
Xilinx Artix-7™ FPGA xc7a100t-1csg324

- Xilinx® 7 series FPGAs include three families:
  - The Artix®-7 family is optimized for lowest cost and power for the highest volume applications.
  - The Virtex®-7 family is optimized for highest system performance and capacity.
  - The Kintex®-7 family is an innovative class of FPGAs optimized for the best price-performance.
- Artix-7 xc7a100t feature summary:
  - 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops
  - 4,860 Kbits of fast block RAM (135 × 36 Kb blocks)
  - 6 clock management tiles, each with phase-locked loop (PLL)
  - 240 DSP slices (with 25 × 18 multiplier, 48-bit accumulator, and pre-adder)
  - On-chip analog-to-digital converter (XADC)
  - Configuration bitstream length of 30,606,304 bits (~3.6 MB)
- Nexys-3 FPGA (Xilinx Spartan®-6 XC6LX16-CS324):
  - 2,278 logic slices, each with four 6-input LUTs and 8 flip-flops
  - 576 Kbits of fast block RAM (32 × 18 Kb blocks)
  - 2 clock management tiles, each with phase-locked loop (PLL)
  - 32 DSP slices (with 18 × 18 multiplier, 48-bit accumulator, and pre-adder)
  - Configuration bitstream length of 3,731,264 bits (~3.6 Mb)

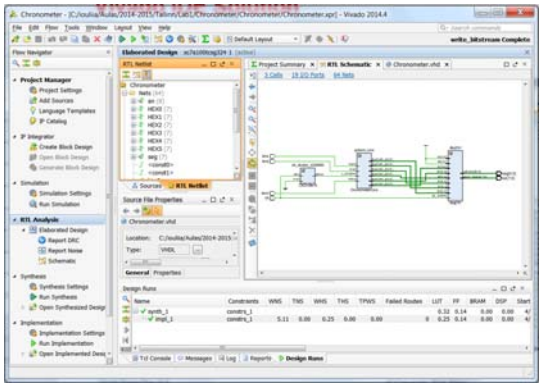
Vivado IDE solution

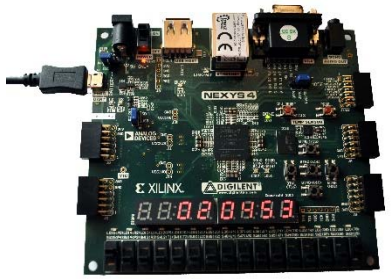
- 7-Series (or newer) devices
- Interactive design and analysis
  - Timing analysis, connectivity, resource utilization, timing constraint analysis and I/O assignment
- RTL development and analysis
  - Elaboration of HDL
  - Hierarchical exploration
  - Schematic generation
- XSIM simulator integration
- Full Tcl scripting support
- Interactive IP plug & play environment
- Synthesis and implementation in one package

FPGA design flow

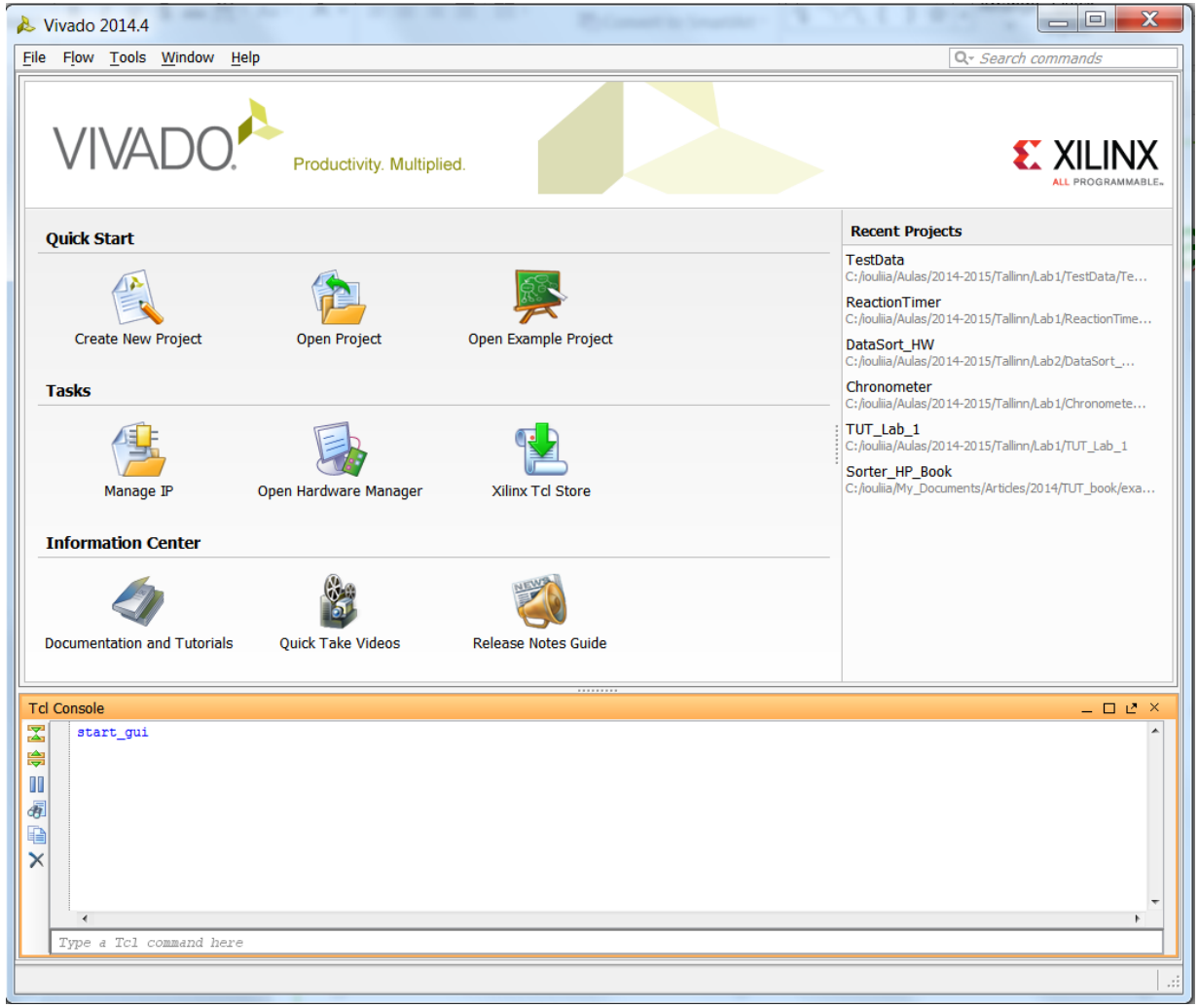


- Design entry based on:
  - VHDL
  - IP Catalog



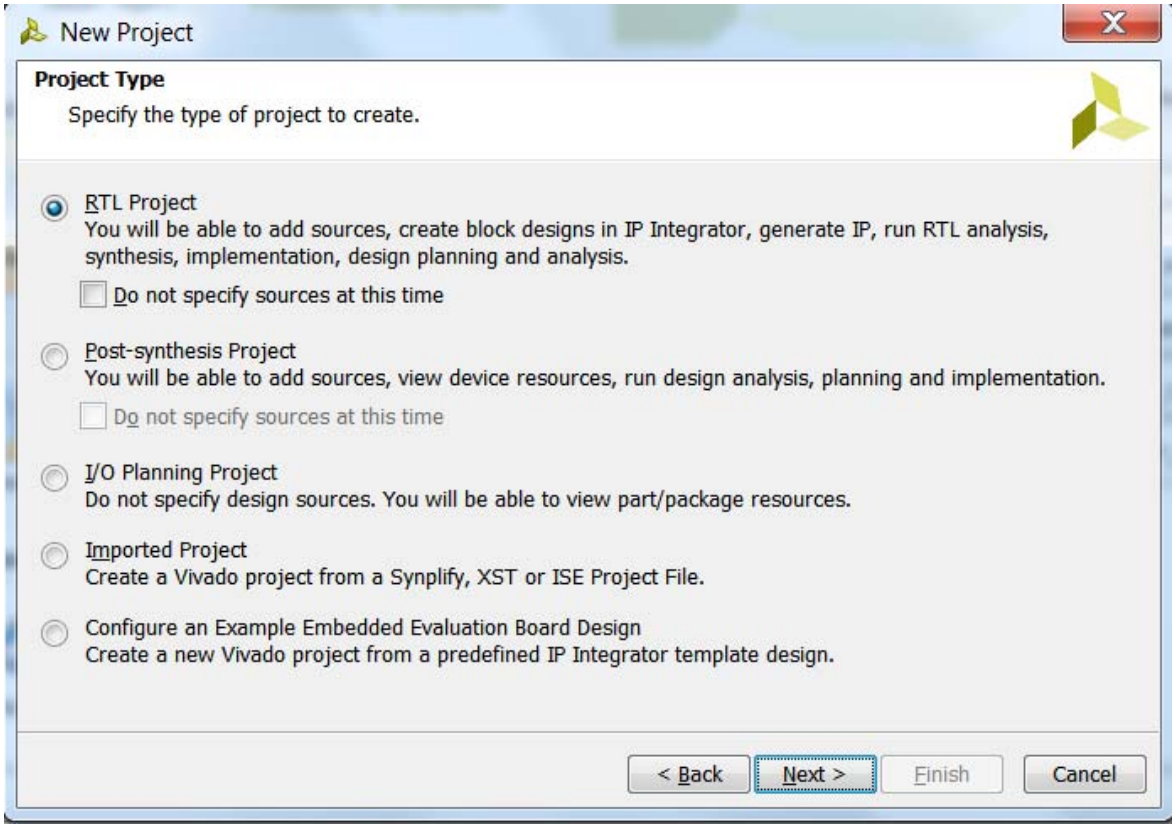


Getting started page



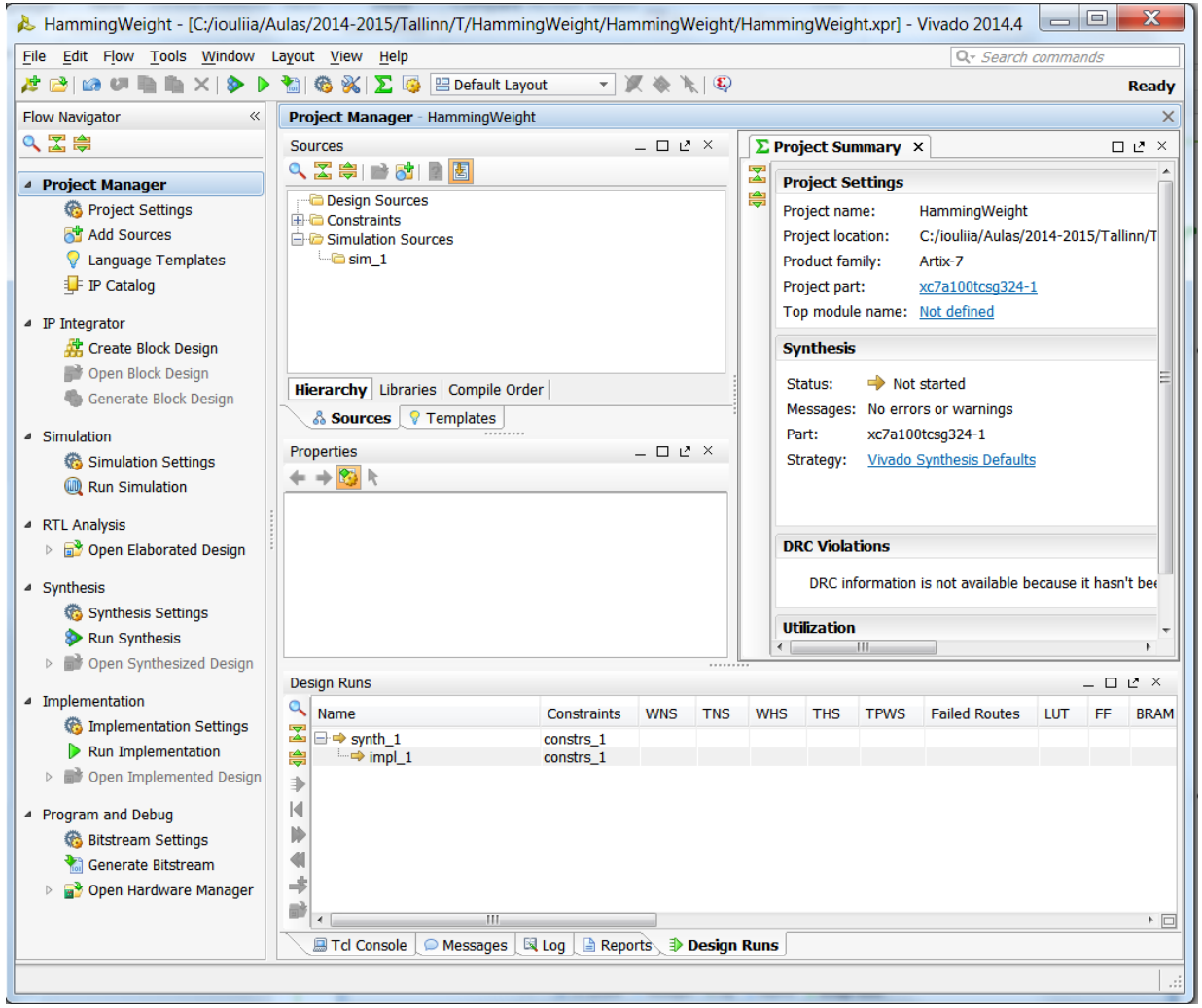
- Quick Start
  - Create new projects
  - Access to previous and example projects
- Tasks
  - IP management
  - Hardware Manager
  - Tcl store
- Information Center
  - Documents
  - Tutorials
  - Videos
- Tcl console
  - Command line access

Creating a new project



- Wizard guides through a creation of a new project
  - Define project name and location
  - Add source files
  - Specify target and simulator language
  - Add IPs
  - Add constraints
  - Select target device (**xc7a100tcsg324-1**)
- Different types of projects
  - RTL project
  - Import an existing project from ISE

Flow Navigator

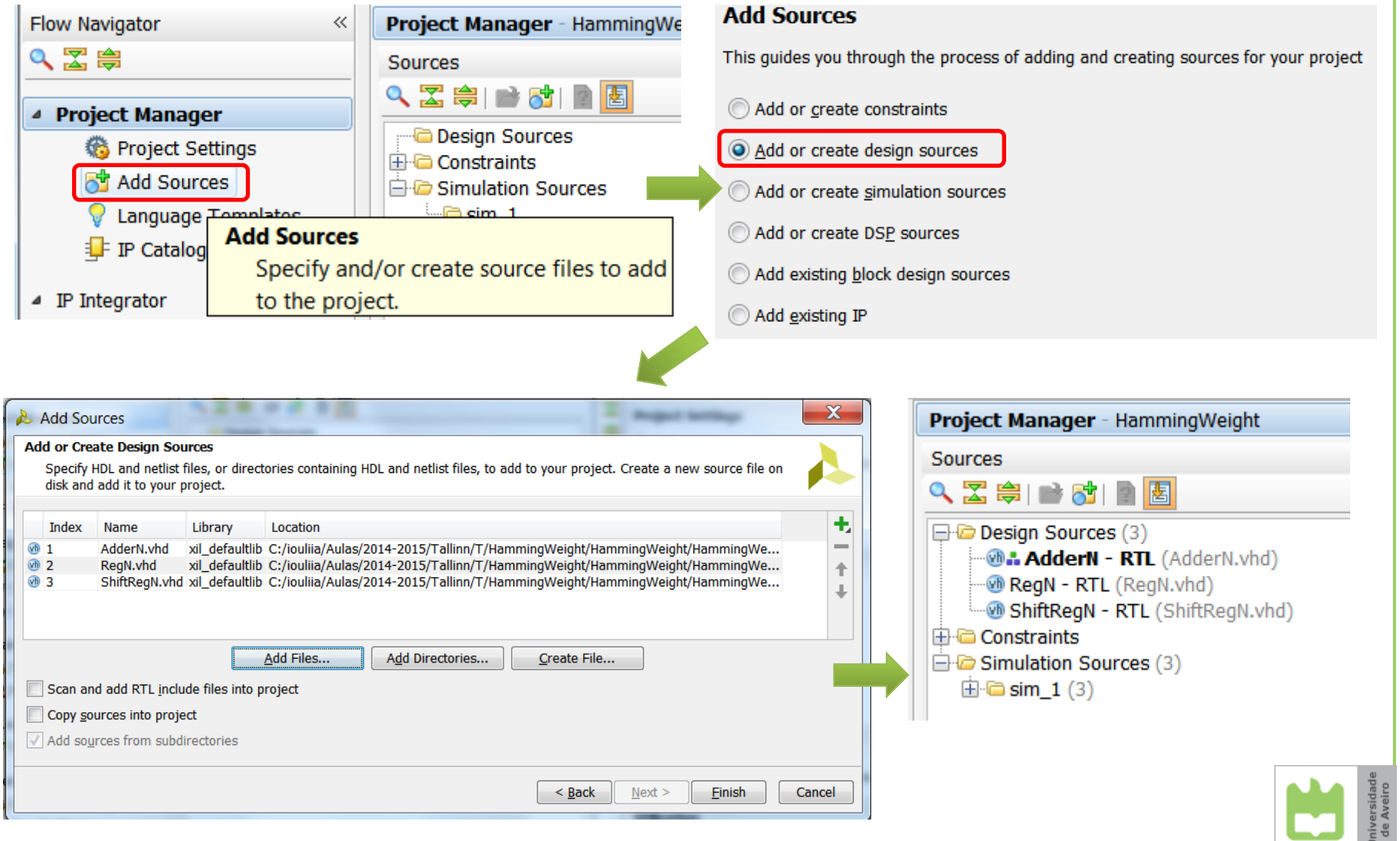


- Permits to manage project settings, add sources, customize IPs, launch simulation, execute RTL analysis, synthesis, implementation and device programming and debugging.
- Gives access to device utilization and timing summary

A complete example

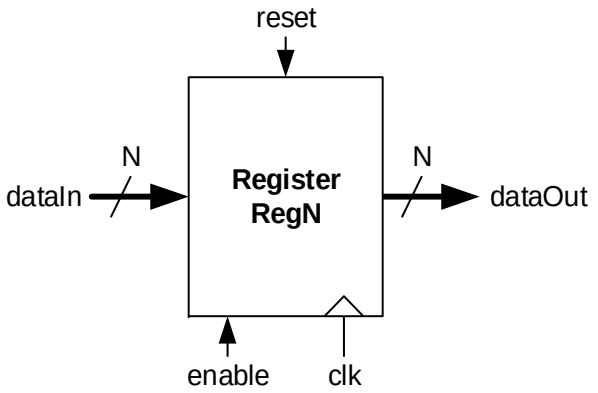
- Calculate the Hamming weight of a 16-bit vector using a sequential circuit
- The input vector is defined with Nexys-4 board switches
- The result should be displayed on LEDs
- Use the following components available:
  - Generic shift register (ShiftRegN.vhd)
  - Generic adder (AdderN.vhd)
  - Generic register (RegN.vhd)

Adding design source files



Generic register (RegN.vhd)

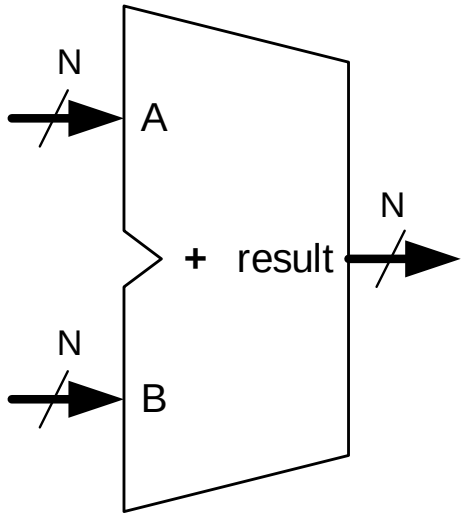
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity RegN is
  generic(N    : positive := 8);
  port(reset   : in  std_logic;
       clk     : in  std_logic;
       enable  : in  std_logic;
       dataIn  : in  std_logic_vector(N-1 downto 0);
       dataOut : out std_logic_vector(N-1 downto 0));
end RegN;

architecture RTL of RegN is
begin
  process(clk)
  begin
    if rising_edge(clk) then
         if reset = '1' then  dataOut <= (others => '0');
         elsif en = '1' then  dataOut <= dataIn;
         end if;
    end if;
  end process;
end RTL;
```



Is reset synchronous or asynchronous?

Generic adder (AdderN.vhd)

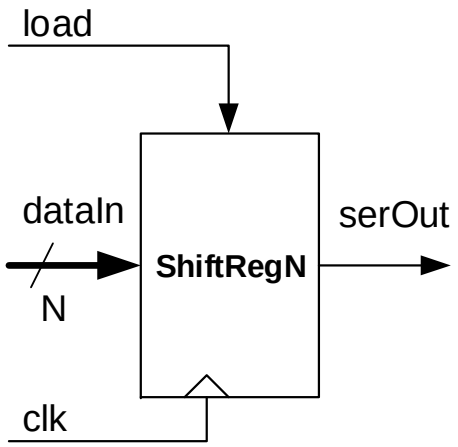
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity AdderN is
  generic (N : positive := 4);
  port(A, B      : in std_logic_vector(N-1 downto 0);
       result    : out std_logic_vector(N-1 downto 0));
end AdderN;

architecture RTL of AdderN is
begin
  result <= std_logic_vector(unsigned(A) + unsigned(B));
end RTL;
```



What is the difference between signed and unsigned additions?

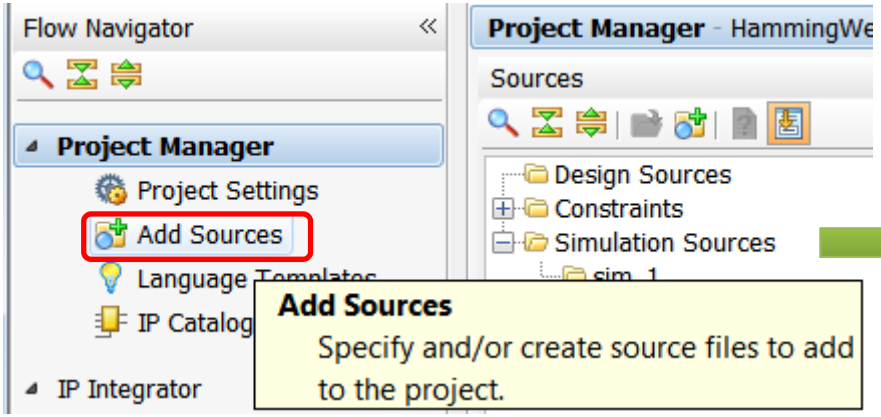
Generic shift register (ShiftRegN.vhd)

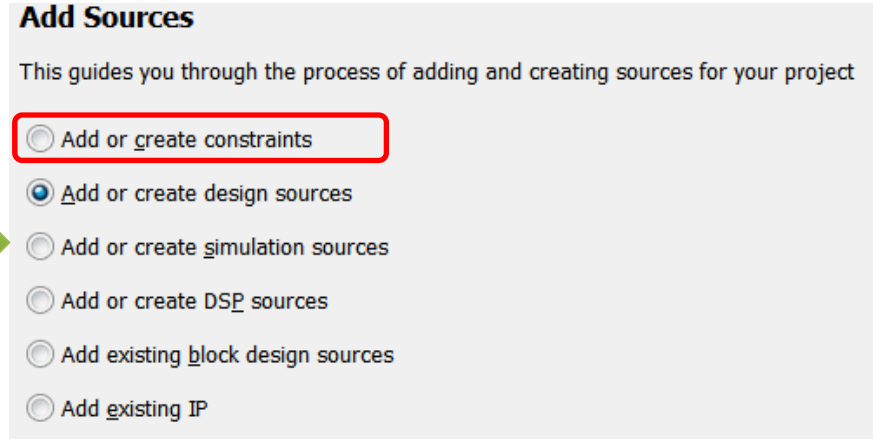
```
entity ShiftRegN is
  generic (N : positive := 4);
  port(clk        : in  std_logic;
       load       : in  std_logic;
       dataIn     : in  std_logic_vector(N-1 downto 0);
       serOut     : out std_logic);
end ShiftRegN;

architecture RTL of ShiftRegN is
  signal s_shiftReg : std_logic_vector(N-1 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if load = '1' then      s_shiftReg <= dataIn;
      else                    s_shiftReg <= '0' & s_shiftReg(N-1 downto 1);
      end if;
    end if;
  end process;

  serOut <= s_shiftReg(0);

end RTL;
```

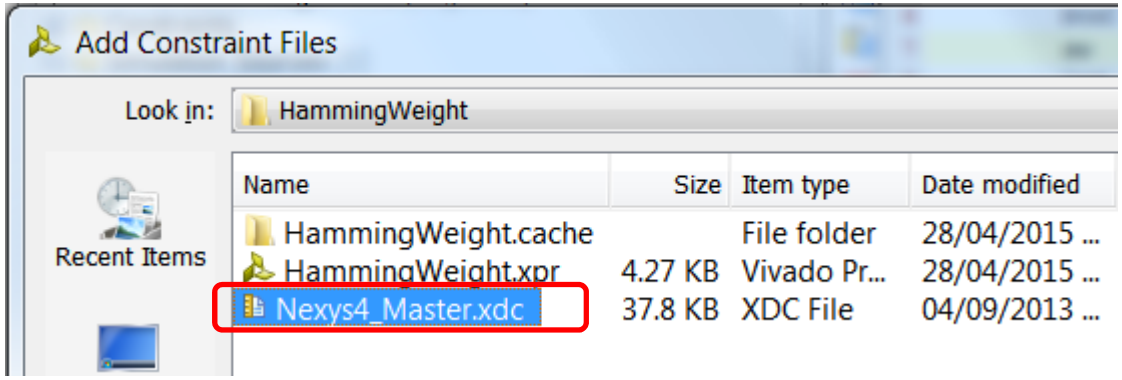


What is the direction of shift?

Specifying constraints

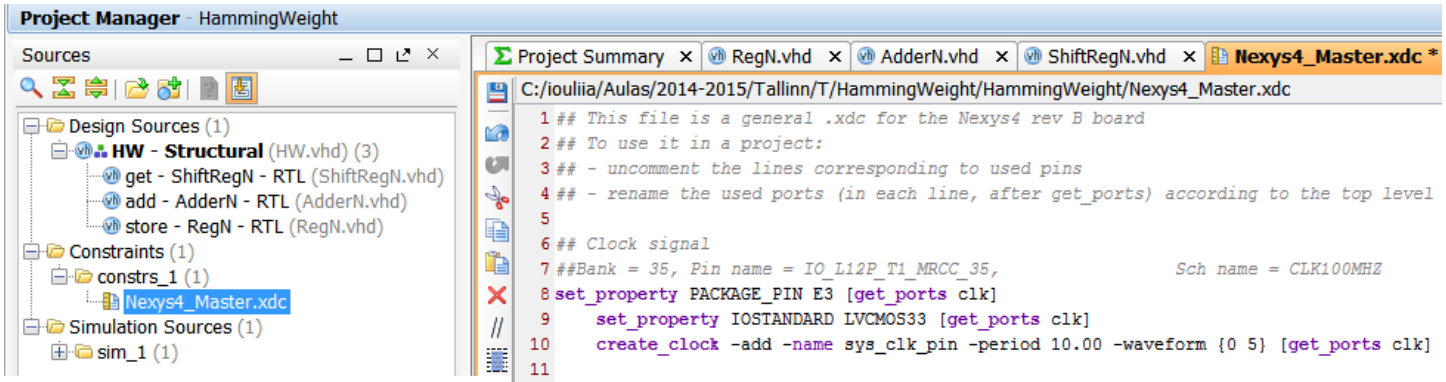




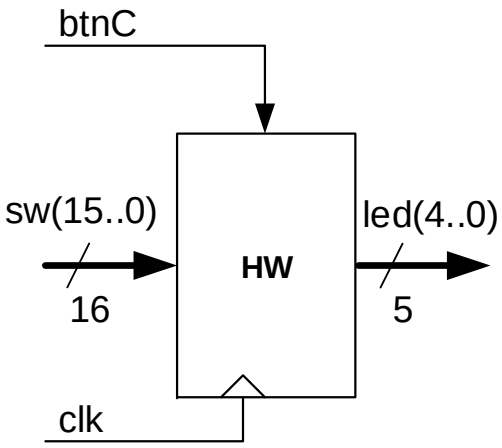


- XDC file
  - Pin locations
  - Clock properties
  - Case-sensitive

Nexys-4 XDC file

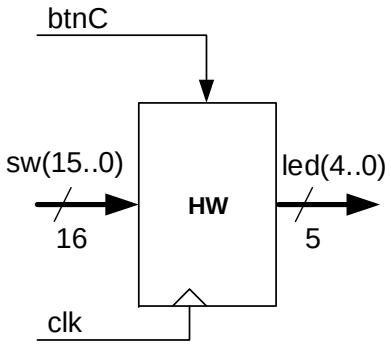



```
1 ## This file is a general .xdc for the Nexys4 rev B board
2 ## To use it in a project:
3 ## - uncomment the lines corresponding to used pins
4 ## - rename the used ports (in each line, after get_ports) according to the top level
5
6 ## Clock signal
7 ##Bank = 35, Pin name = IO_L12P_T1_MRCC_35,                Sch name = CLK100MHZ
8 set_property PACKAGE_PIN E3 [get_ports clk]
9     set_property IOSTANDARD LVCMOS33 [get_ports clk]
10    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]
11
12 ## Switches
13 ##Bank = 34, Pin name = IO_L21P_T3_DQS_34,                Sch name = SW0
14 set_property PACKAGE_PIN U9 [get_ports {sw[0]}]
15    set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
16 #Bank = 34, Pin name = IO_25_34,                          Sch name = SW1
17 set_property PACKAGE_PIN U8 [get_ports {sw[1]}]
18    set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
19 #Bank = 34, Pin name = IO_L23P_T3_34,                     Sch name = SW2
20 set_property PACKAGE_PIN R7 [get_ports {sw[2]}]
21    set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
22 #Bank = 34, Pin name = IO_L19P_T3_34,                     Sch name = SW3
23 set_property PACKAGE_PIN R6 [get_ports {sw[3]}]
24    set_property IOSTANDARD LVCMOS33 [get_ports {sw[3]}]
25 #Bank = 34, Pin name = IO_L19N_T3_VREF_34,                Sch name = SW4
26 set_property PACKAGE_PIN R5 [get_ports {sw[4]}]
27    set_property IOSTANDARD LVCMOS33 [get_ports {sw[4]}]
28 #Bank = 34, Pin name = IO_L20P_T3_34,                     Sch name = SW5
29 set_property PACKAGE_PIN V7 [get_ports {sw[5]}]
```




```
189 ##Buttons
190 ##Bank = 15, Pin name = IO_L3P_T0_DQS_AD1P_15,          Sch name = CPU_RESET
191 #set_property PACKAGE_PIN C12 [get_ports btnCpuReset]
192     #set_property IOSTANDARD LVCMOS33 [get_ports btnCpuReset]
193 #Bank = 15, Pin name = IO_L11N_T1_SRCC_15,               Sch name = BTNC
194 set_property PACKAGE_PIN E16 [get_ports btnC]
195     set_property IOSTANDARD LVCMOS33 [get_ports btnC]
196 ##Bank = 15, Pin name = IO_L14P_T2_SRCC_15,              Sch name = BTNU
197 #set_property PACKAGE_PIN F15 [get_ports btnU]
198     #set_property IOSTANDARD LVCMOS33 [get_ports btnU]
```

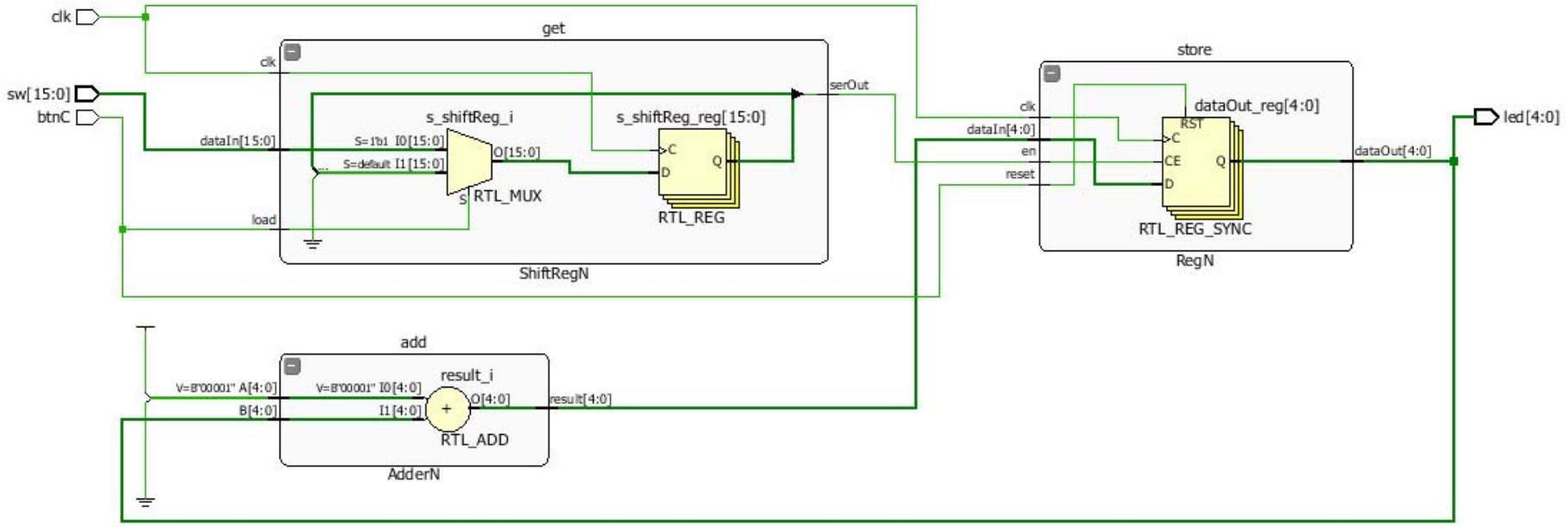
Interconnecting the modules (HW.vhd)



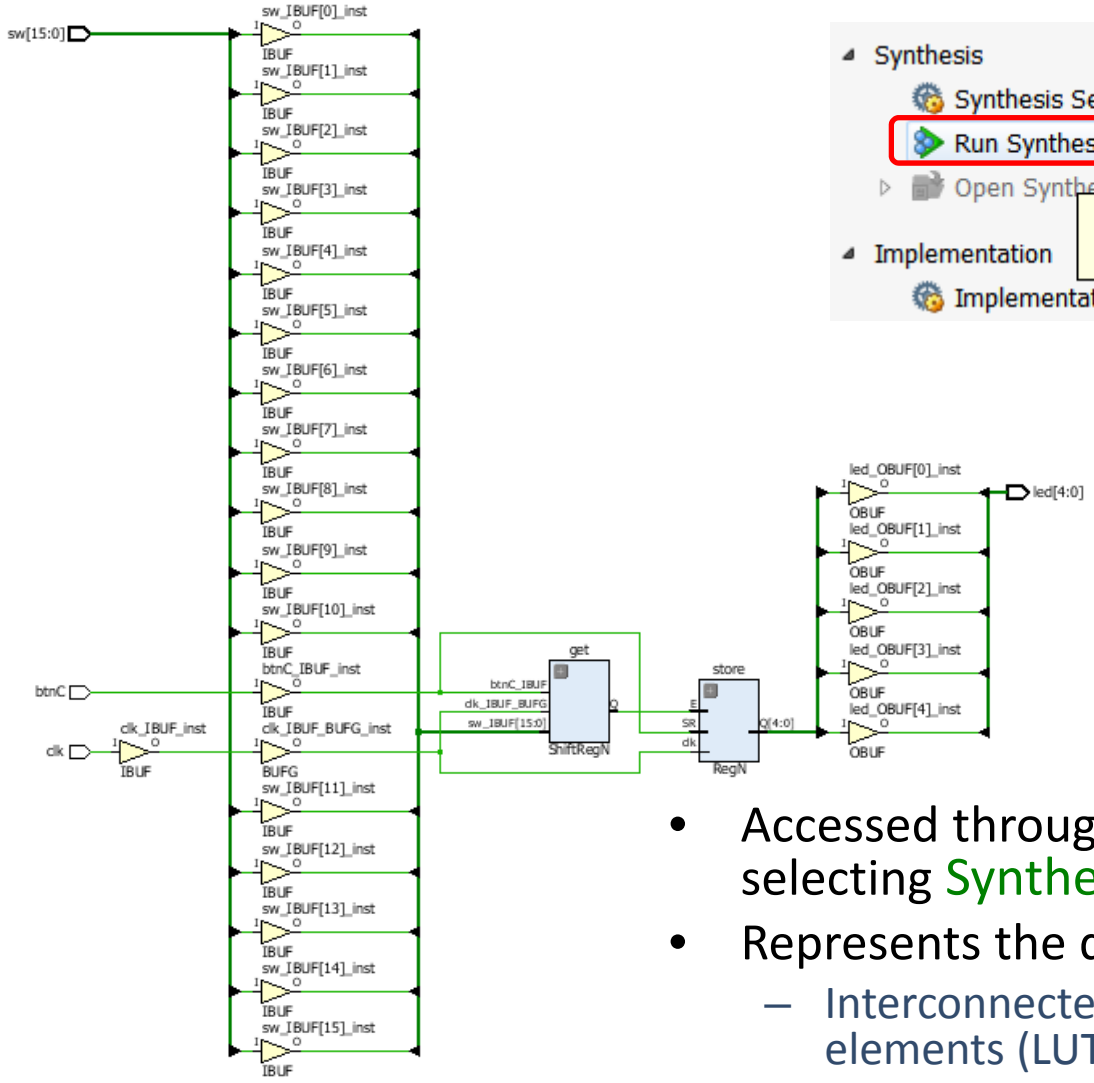

```
entity HW is
    port( clk, btnC: in  std_logic;
          sw       : in std_logic_vector(15 downto 0);
          led      : out std_logic_vector(4 downto 0));
end HW;

architecture Structural of HW is
    signal s_en              : std_logic;
    constant s_value         : std_logic_vector(4 downto 0) := "00001";
    signal s_acc, s_add      : std_logic_vector(4 downto 0);
begin
    get: entity work.ShiftRegN(RTL)
      generic map(N => 16)
      port map (clk => clk, load => btnC, dataIn => sw, serOut => s_en);

    add: entity work.AdderN(RTL)
      generic map (N => 5)
      port map (A => s_value, B => s_acc, result => s_add);

    store: entity work.RegN(RTL)
      generic map (N => 5)
      port map (clk => clk, reset => btnC, en => s_en, dataIn => s_add, dataOut =>
    s_acc);

    led <= s_acc;
end Structural;
```

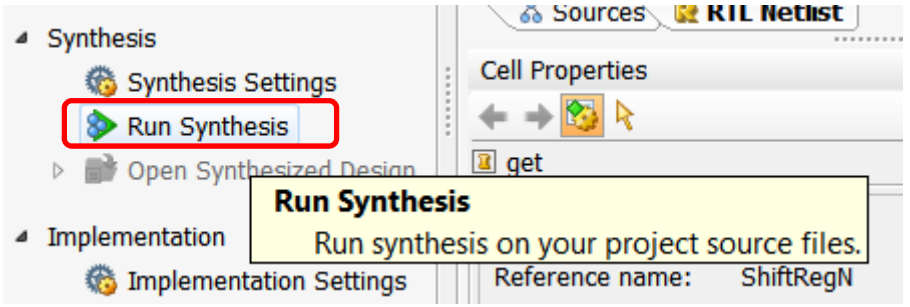
Elaborated design

- Accessed through the Flow Navigator by selecting Open Elaborated Design
- Represents the design before synthesis
  - Instances of modules
  - Generic representation of hardware components
  - Interconnections



Synthesis and the synthesized design

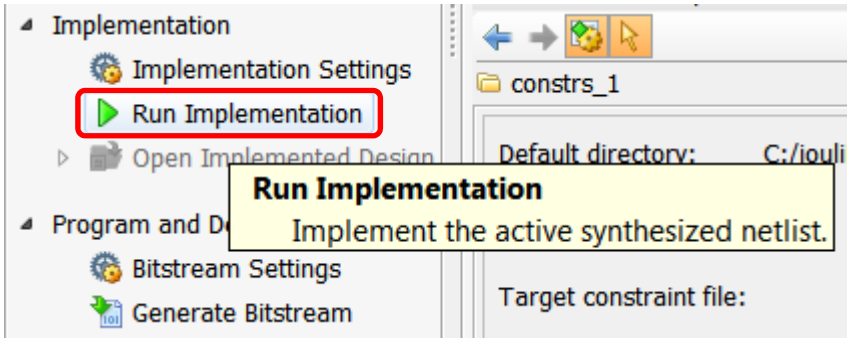


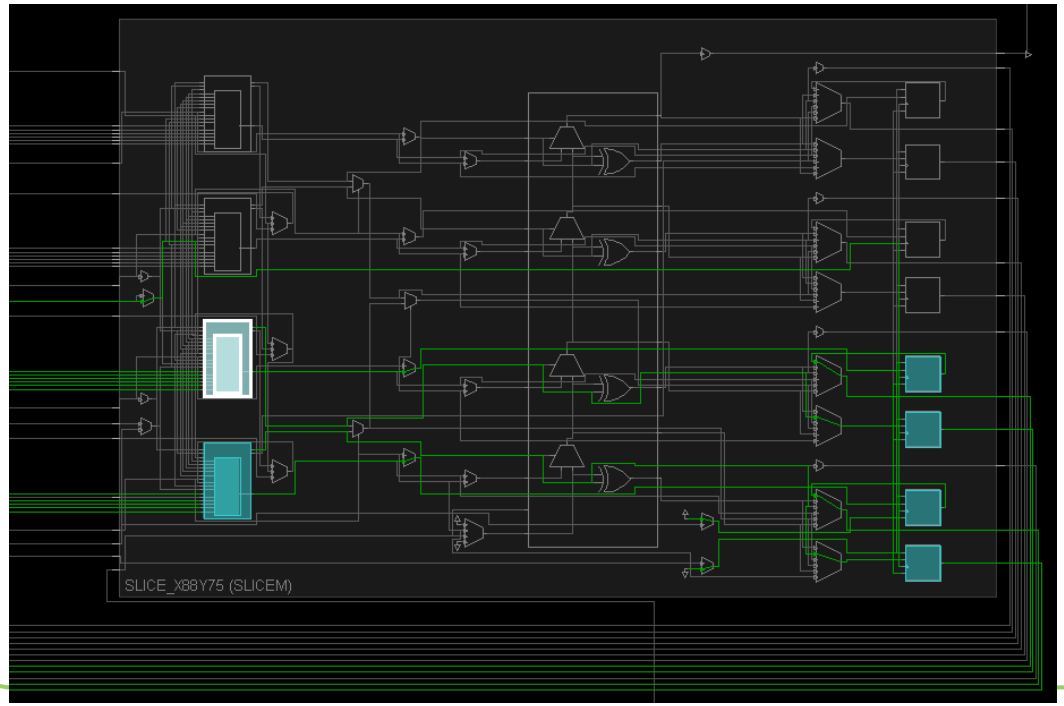


- Accessed through the Flow Navigator by selecting Synthesized Design -> Schematic
- Represents the design after synthesis
  - Interconnected netlist of modules and basic elements (LUTs, block RAMs, IBUFs, etc.)

Implementation and the implemented design

- Accessed after the implementation process by selecting Open Implemented Design
- Represents the design after the implementation
  - Structurally similar to the synthesized design
  - Cells have locations, and nets are mapped to specific routing resources





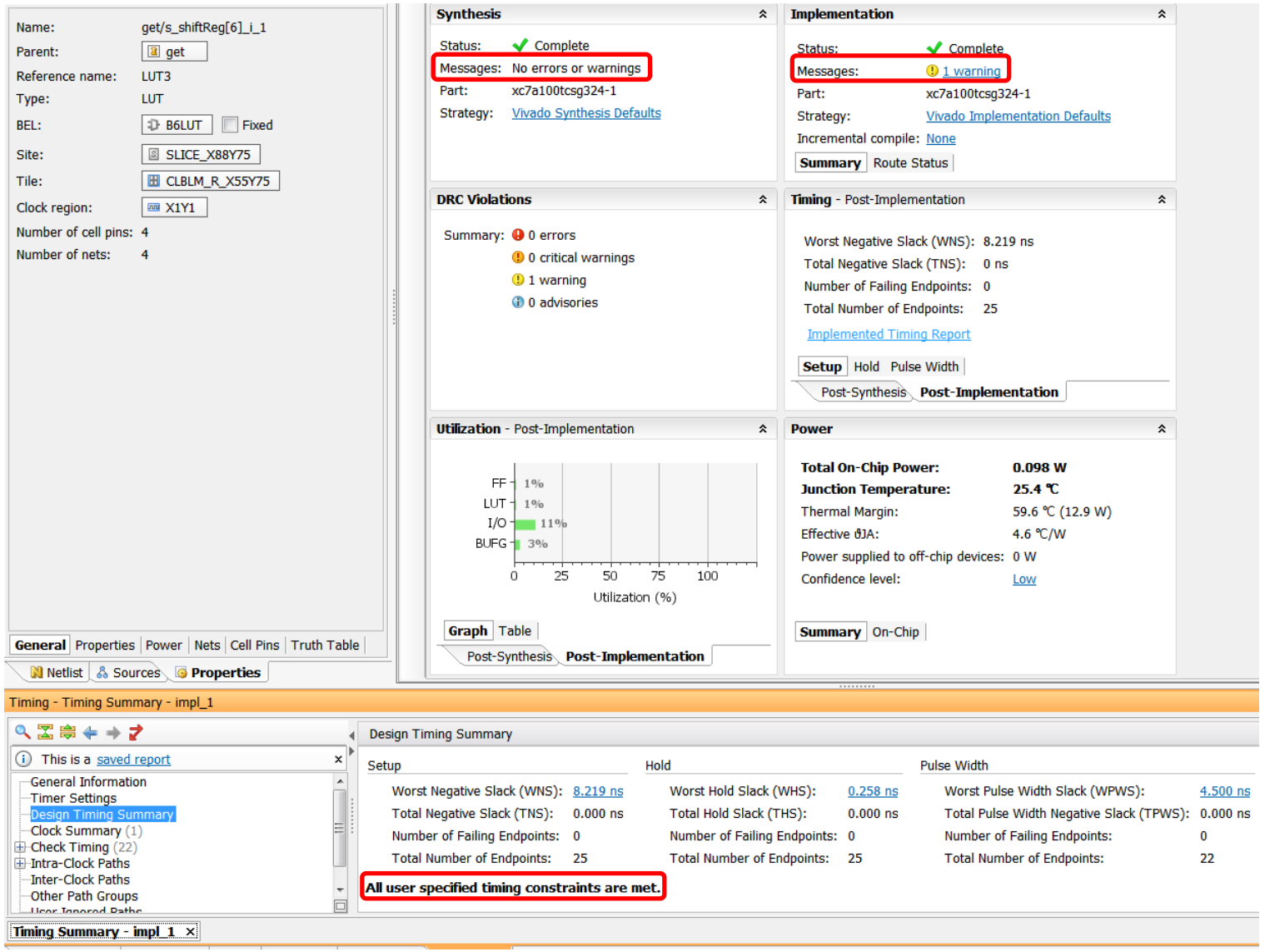
Utilization - Post-Implementation

Resource	Utilization	Available	Utilization...
FF	21	126800	0.02
LUT	11	63400	0.02
I/O	23	210	10.95
BUFG	1	32	3.12

Graph **Table**

Post-Synthesis **Post-Implementation**

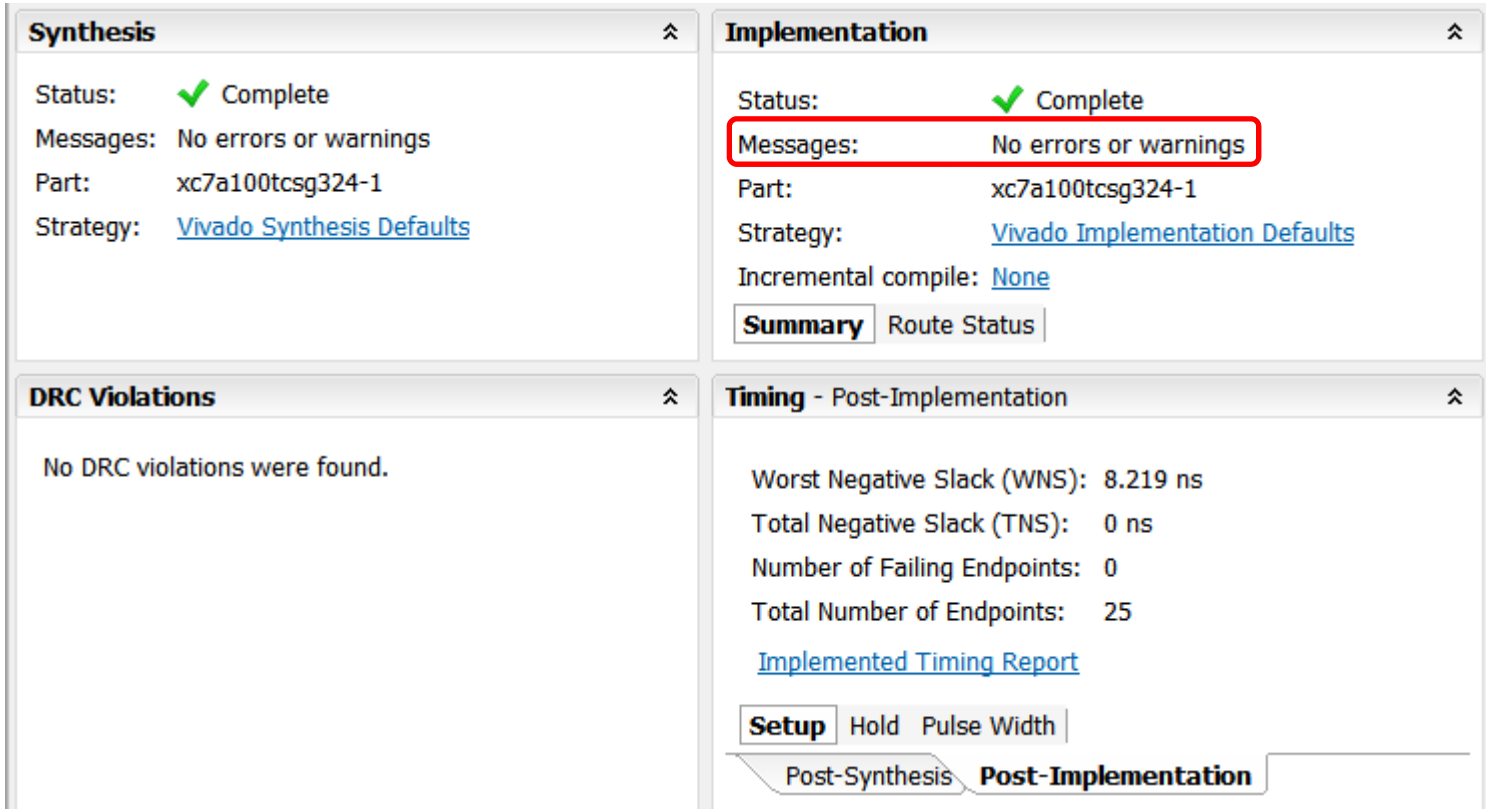
Reports


Name: get/s_shiftReg[6]_i_1
Parent: get
Reference name: LUT3
Type: LUT
BEL: B6LUT Fixed
Site: SLICE_X88Y75
Tile: CLBLM_R_X55Y75
Clock region: X1Y1
Number of cell pins: 4
Number of nets: 4
General Properties Power Nets Cell Pins Truth Table
Netlist Sources Properties
Synthesis
Status: Complete
Messages: No errors or warnings
Part: xc7a100tcsg324-1
Strategy: Vivado Synthesis Defaults
Implementation
Status: Complete
Messages: 1 warning
Part: xc7a100tcsg324-1
Strategy: Vivado Implementation Defaults
Incremental compile: None
Summary Route Status
DRC Violations
Summary: 0 errors
0 critical warnings
1 warning
0 advisories
Timing - Post-Implementation
Worst Negative Slack (WNS): 8.219 ns
Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 25
Implemented Timing Report
Setup Hold Pulse Width
Post-Synthesis Post-Implementation
Utilization - Post-Implementation
FF 1%
LUT 1%
I/O 11%
BUFG 3%
0 25 50 75 100
Utilization (%)
Graph Table
Post-Synthesis Post-Implementation
Power
Total On-Chip Power: 0.098 W
Junction Temperature: 25.4 °C
Thermal Margin: 59.6 °C (12.9 W)
Effective θJA: 4.6 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low
Summary On-Chip
Timing - Timing Summary - impl_1
This is a saved report
General Information
Timer Settings
Design Timing Summary
Clock Summary (1)
Check Timing (22)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
Design Timing Summary
Setup
Worst Negative Slack (WNS): 8.219 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 25
Hold
Worst Hold Slack (WHS): 0.258 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 25
Pulse Width
Worst Pulse Width Slack (WPWS): 4.500 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 22
All user specified timing constraints are met.
Timing Summary - impl_1

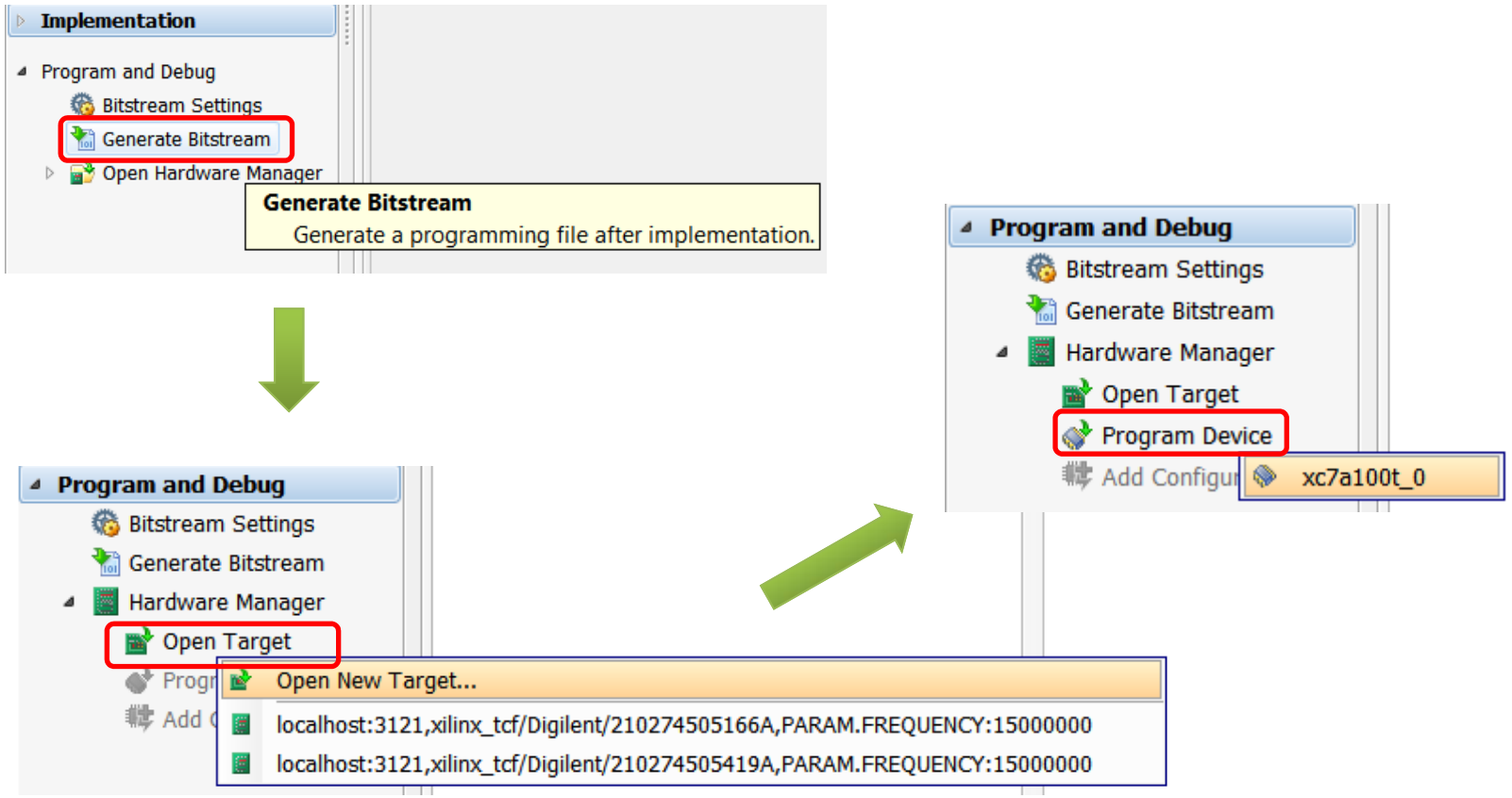
Errors and warnings

- Correct the XDC file

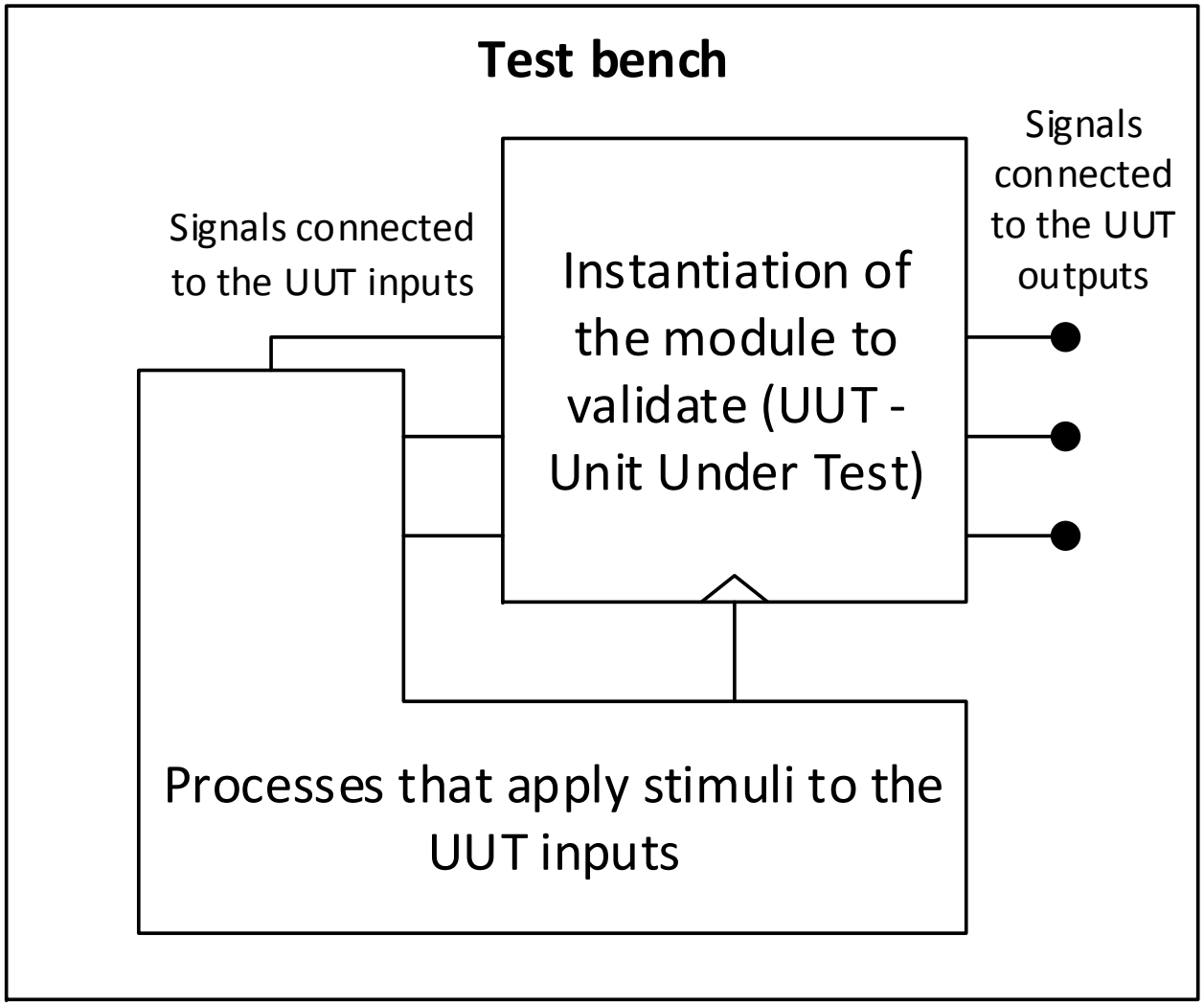
```
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```



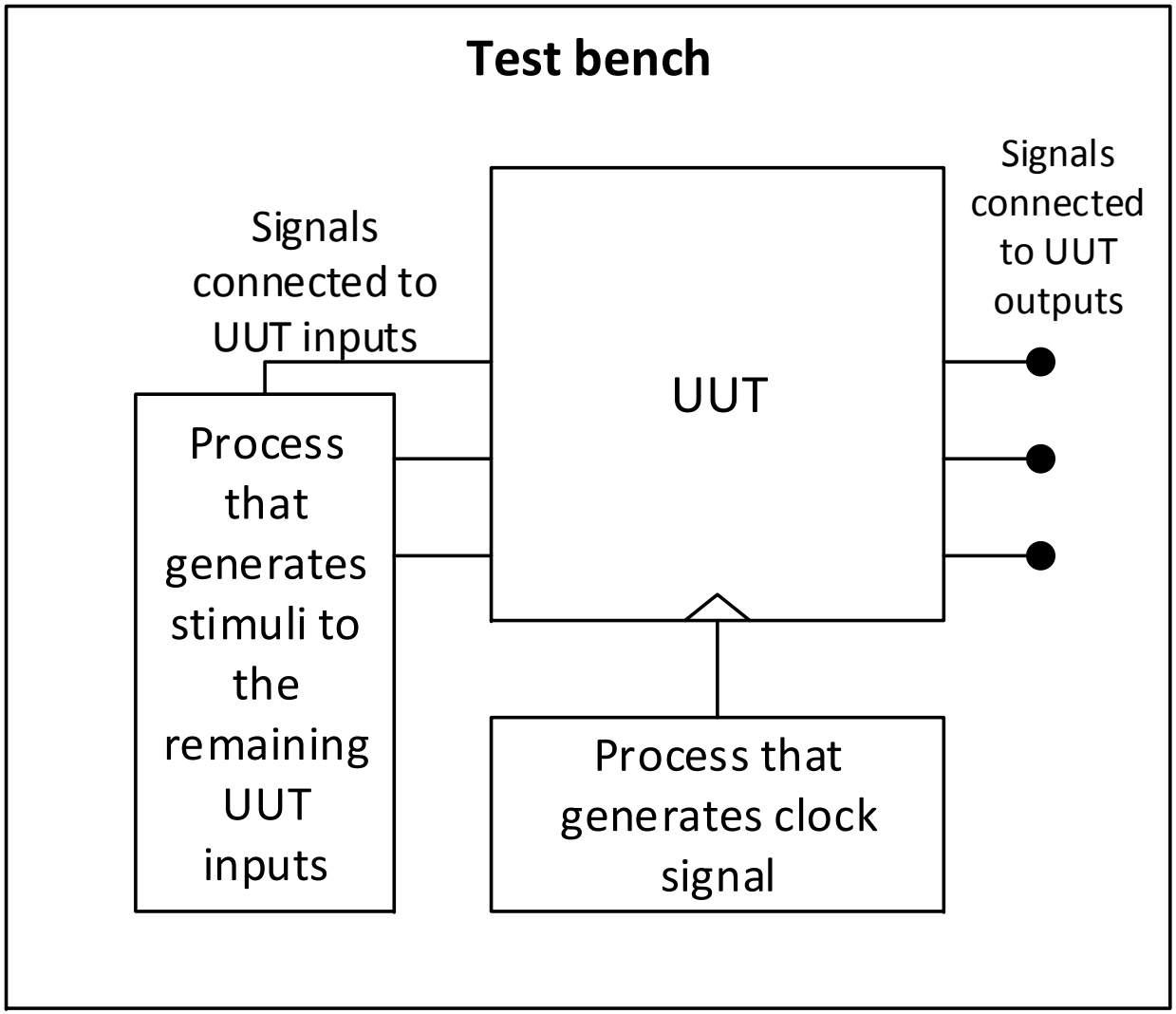
Bitstream generation and device programming



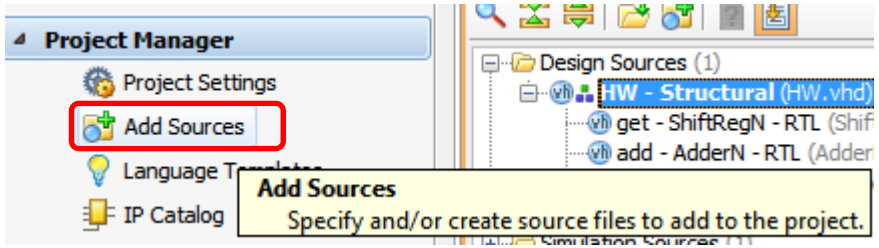
Design of test benches in VHDL

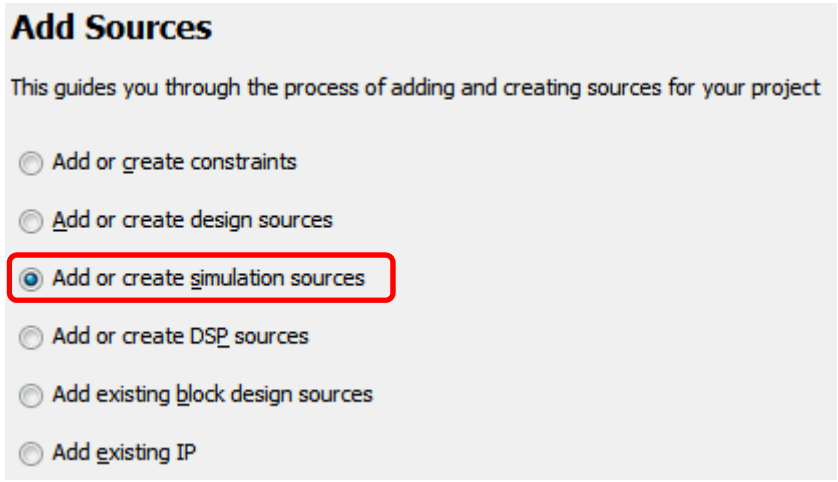


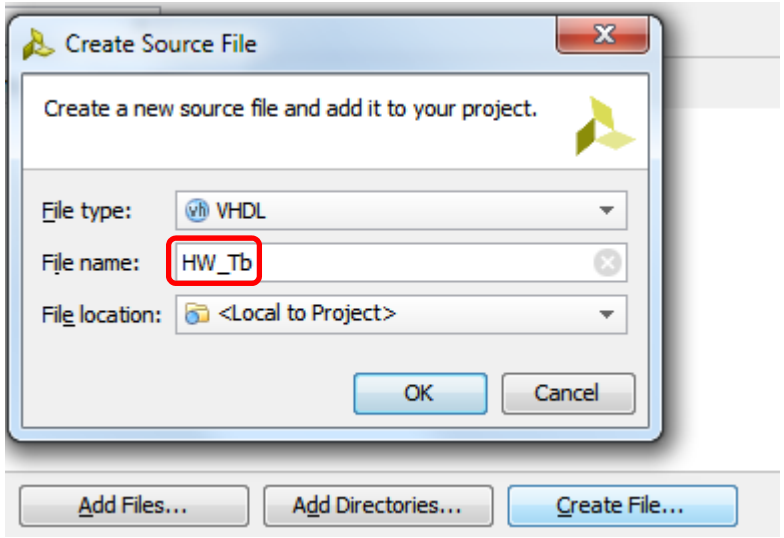
Test benches for sequential UUTs

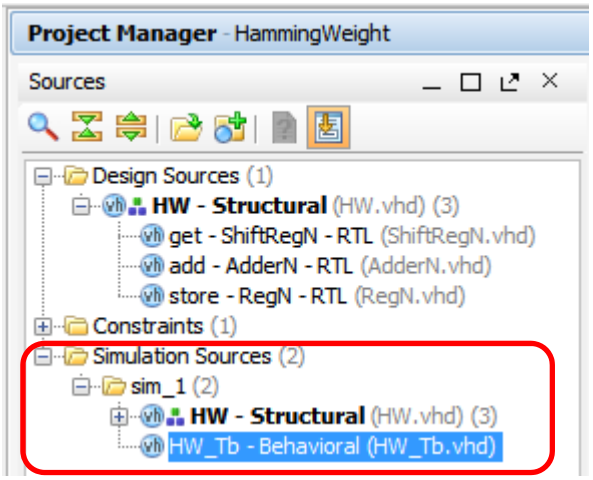


Adding test bench files



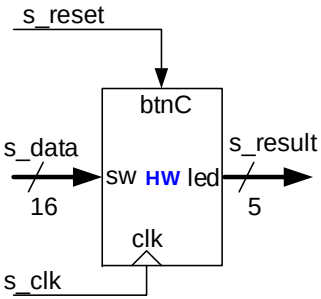






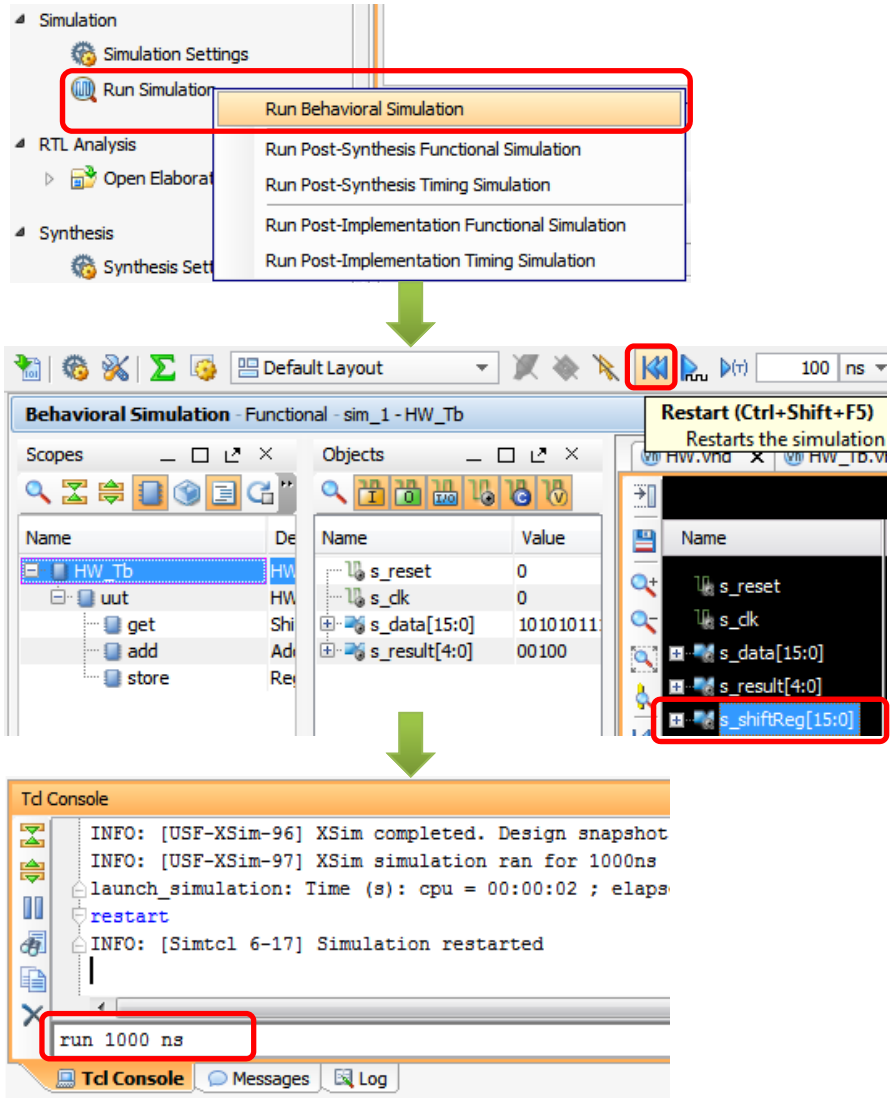
Test bench for HW counter

```
-- Entity with no ports
entity HW_Tb is
end HW_Tb;

architecture Stimulus of HW_Tb is
  -- Signals to connect to UUT inputs
  signal s_reset, s_clk : std_logic;
  signal s_data         : std_logic_vector(15 downto 0);
  -- Signals to connect to UUT outputs
  signal s_result : std_logic_vector(4 downto 0);
begin
  -- Unit Under Test (UUT) instantiation
  uut : entity work.HW(Structural)
    port map (clk   => s_clk,
              btnC  => s_reset,
              sw    => s_data,
              led   => s_result);
  -- Process clock
  clock_proc : process
  begin
    s_clk <= '0'; wait for 10 ns;
    s_clk <= '1'; wait for 10 ns;
  end process;
```



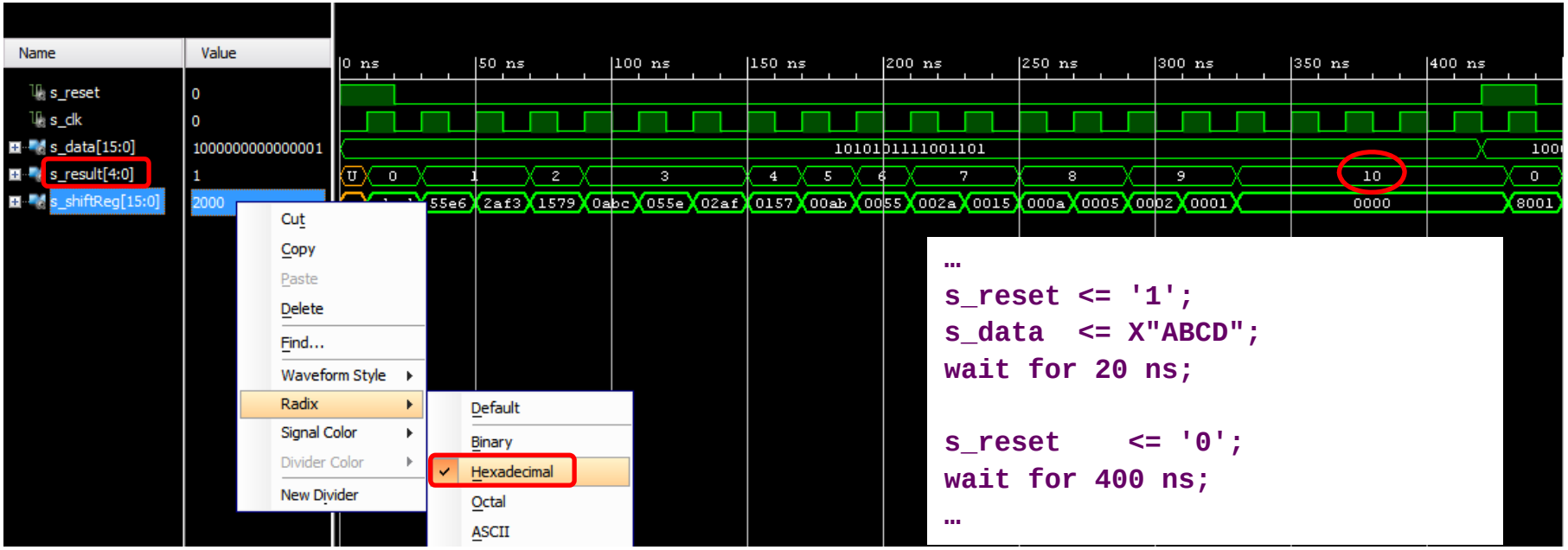

```
  --Process stim
  stim_proc : process
  begin
    s_reset <= '1';
    s_data  <= X"ABCD";
    wait for 20 ns;

    s_reset <= '0';
    wait for 400 ns;

    s_reset <= '1';
    s_data  <= X"8001";
    wait for 20 ns;

    s_reset <= '0';
    wait for 400 ns;
  end process;
end Stimulus;
```

Simulation in Vivado

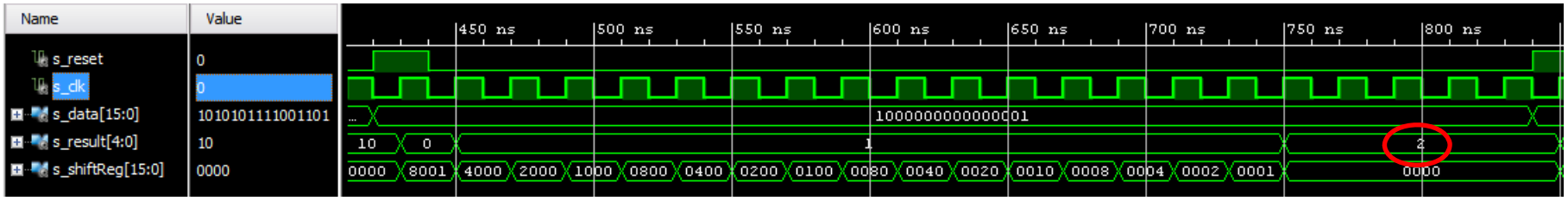


- Different simulation options are supported
- Possibility to add internal signals of UUT components
- Interaction with the simulator is done either through menus/buttons or via Tcl command line
- Standard waveform viewer options permit to change the scale, radix, etc.

Simulation results




```
…
s_reset <= '1';
s_data  <= X"ABCD";
wait for 20 ns;

s_reset    <= '0';
wait for 400 ns;
…
```




```
s_reset <= '1';
s_data  <= X"8001";
wait for 20 ns;
```

```
s_reset    <= '0';
wait for 400 ns;
```

Some problems with simulation

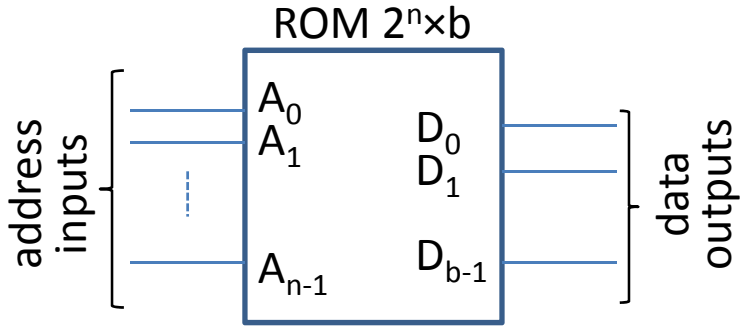
Intended functionality	Incorrect specification	Comment

D flip-flop	`process(clk)`
	`begin`
	`  if (clk = '1') then`
	`  dataOut <= dataIn;`
	`  end if;`
	`end process;`
		Simulates correctly, **but** malfunctions in hardware!!!

D flip-flop with asynchronous reset	`process(clk)`
	`begin`
	`  if (reset = '1') then`
	`  dataOut <= '0';`
	`  elsif rising_edge(clk) then`
	`  dataOut <= dataIn;`
	`  end if;`
	`end process;`
		Does not simulate correctly, **although** synthesizes and works correctly in hardware!!!

How to solve?

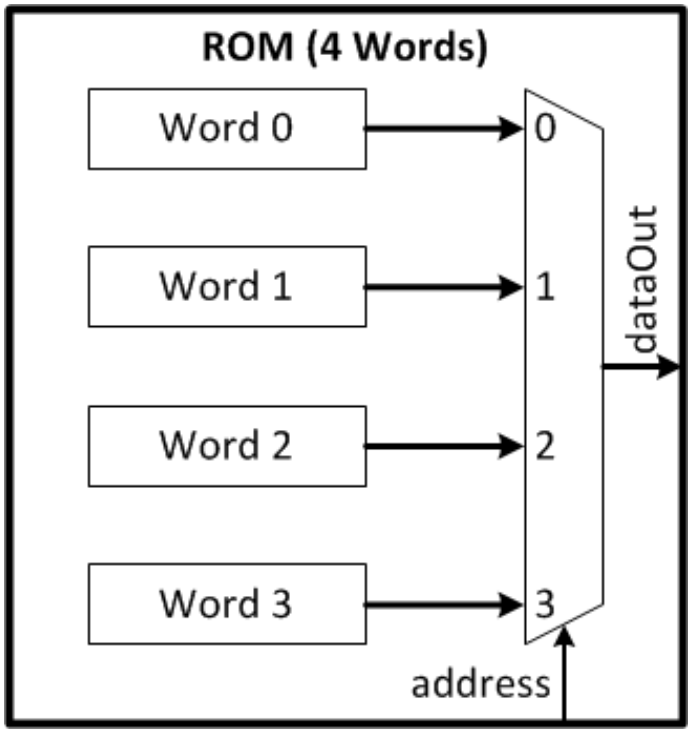
Types of Memories



- *Read Only Memory* (**ROM**)
  - Memory just for reading
  - Any memory position can be accessed
  - Useful for storing constants/non-changing data
    - Static tables
    - Messages, strings, etc.
  - The content is defined at the time of manufacture or device programming

Conceptual internal ROM structure

- Example of a ROM for storing 4 words
- The size of *dataOut* is the same as the size of every memory word
- In this case, is reading synchronous or asynchronous?
- What is the size of the input *address*?
  - Usually power of 2
- What is the size of the input *address* for a ROM storing 1M words?

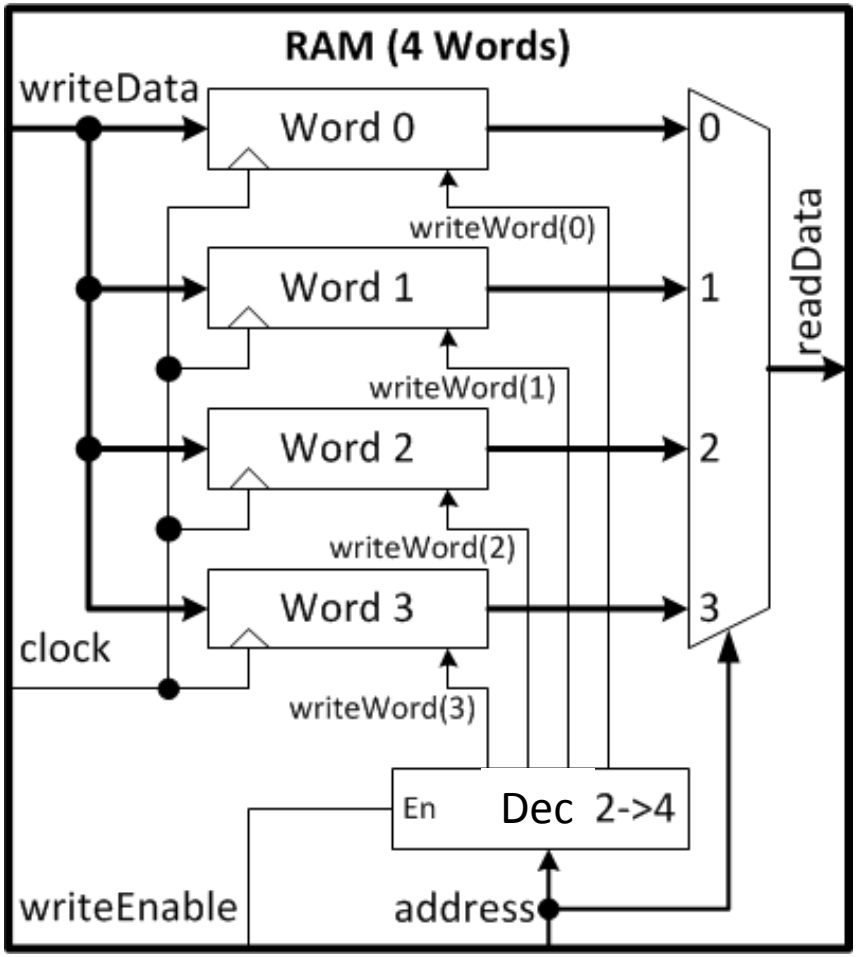


Types of Memories

- *Random Access Memory* (**RAM**)
  - Memory for reading and writing
  - Any memory position can be accessed
  - Useful for storing data that can be changed during operation of the system
    - The content is written during system operation
  - Can have
    - 1 port (only one read or write operation in a given time)
    - Multi-port (various read or write operations in a given time)

Conceptual internal RAM structure

- Example of a RAM for storing 4 words
- The size of data input *writeData* and data output *readData* is the same as the size of every memory word
- In this case
  - is reading synchronous or asynchronous?
  - is writing synchronous or asynchronous?



Modeling a ROM

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

entity ROM_8x4 is
   port(address : in  std_logic_vector(2 downto 0);
        dataOut : out std_logic_vector(3 downto 0));
end ROM_8x4;

architecture RTL of ROM_8x4 is
   subtype TDataWord is std_logic_vector(3 downto 0);
   type TROM is array (0 to 7) of TDataWord;
   constant c_memory: TROM := ("0000", "0001", "0010", "0100",
                               "1000", "1010", "1100", "1110");
begin
   dataOut <= c_memory(to_integer(unsigned(address)));
end RTL;
```

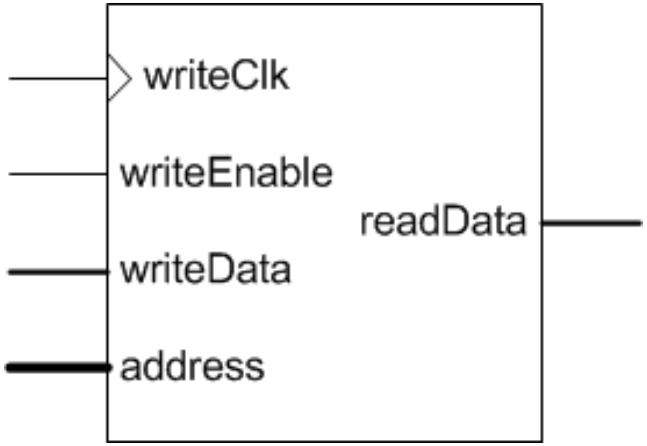
Subtype defines a restricted set of values of its base type

Constant allows to define constant values

to_integer function converts (un)signed values to integers.

Modeling a 32x8 bit RAM (1 port; **asynchronous read**) - entity

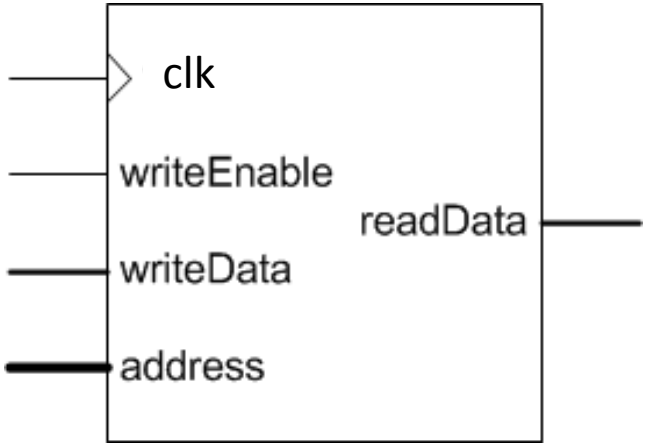
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity RAM_1_SW_AR is
   port(writeClk    : in  std_logic;
        writeEnable : in  std_logic;
        writeData   : in  std_logic_vector(7 downto 0);
        address     : in  std_logic_vector(4 downto 0);
        readData    : out std_logic_vector(7 downto 0));
end RAM_1_SW_AR;
```



Modeling a 32x8 bit RAM (1 port; **asynchronous read**) - architecture

```
architecture RTL of RAM_1_SW_AR is

   constant NUM_WORDS : integer := 32;
   subtype TDataWord is std_logic_vector(7 downto 0);
   type TMemory is array (0 to NUM_WORDS-1) of TDataWord;
   signal s_memory : TMemory;

begin
   process(writeClk)
   begin
      if (rising_edge(writeClk)) then
         if (writeEnable = '1') then
            s_memory(to_integer(unsigned(address))) <= writeData;
         end if;
      end if;
   end process;

   readData <= s_memory(to_integer(unsigned(address)));
end RTL;
```

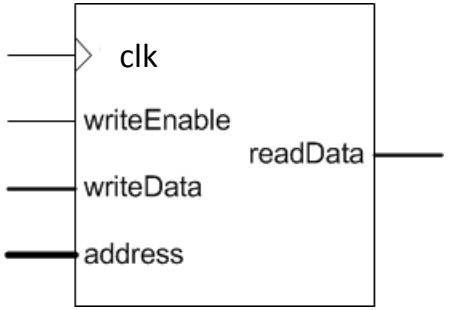
Modeling a parameterizable RAM (1 port; **synchronous read**) - entity




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

entity RAM_2_SW_AR is
  generic(addrBusSize : integer := 4;
          dataBusSize : integer := 8);
  port(clk           : in  std_logic;
       writeEnable   : in  std_logic;
       address       : in  std_logic_vector((addrBusSize - 1) downto 0);
       writeData     : in  std_logic_vector((dataBusSize - 1) downto 0);
       readData      : out std_logic_vector((dataBusSize - 1) downto 0));
end RAM_2_SW_AR;
```

Modeling a parameterizable RAM (1 port; **synchronous read**) – arch.

```
architecture RTL of RAM_2_SW_AR is
   constant NUM_WORDS : integer := (2 ** addrBusSize);
   subtype TDataWord is std_logic_vector((dataBusSize - 1) downto 0);
   type TMemory is array (0 to NUM_WORDS-1) of TDataWord;
   signal s_memory : TMemory;

begin
   process(clk)
   begin
      if (rising_edge(clk)) then
         if (writeEnable = '1') then
            s_memory(to_integer(unsigned(address)) <= writeData;
         end if;
      end if;
   end process;

   read_proc : process(clk) --read-first mode
   begin
      if (rising_edge(clk)) then
         readData <= s_memory(to_integer(unsigned(address)));
      end if;
   end process;
end RTL;
```



read-first: old content is read before new content is loaded

How to instantiate a 64Kx32 RAM?

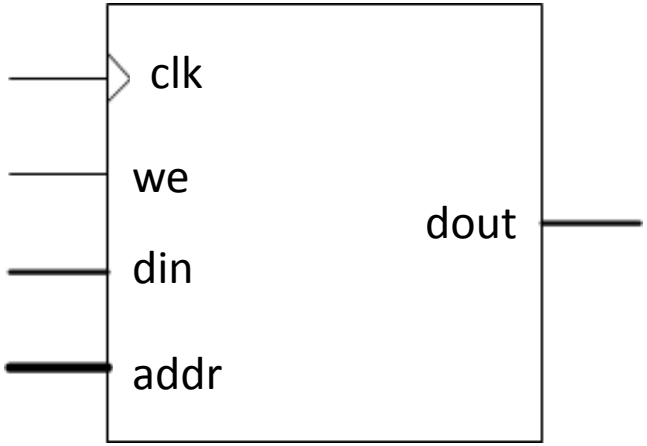
Initialization of memories from files

- ROM/RAM contents can be initialized during synthesis from a text file.
- Example of a test file “data.dat”:

```
00000001
00000010
00000011
00000100
00000101
00000110
00000111
00001000
00001001
00001010
00001011
00001100
00001101
00001110
00001111
11111000
```

Initializing a 16x8 bit RAM (1 port; **synchronous read**) - entity

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use std.textio.all;
use IEEE.NUMERIC_STD.ALL;

entity BRAM_ini is
    port(clk    : in std_logic;
         we     : in std_logic;
         addr   : in std_logic_vector(3 downto 0);
         din    : in std_logic_vector(7 downto 0);
         dout   : out std_logic_vector(7 downto 0));
end BRAM_ini;
```



The package textio declares the file type **text**, representing files of strings

Initializing a 16x8 bit RAM (1 port; **synchronous read**) - function

```
architecture Behavioral of BRAM_ini is
    constant NUM_WORDS : integer := 16;
    subtype TDataWord is bit_vector(7 downto 0);
    type TMemory is array (0 to NUM_WORDS-1) of TDataWord;

  impure function InitRamFromFile (RamFileName : in string)
                        return TMemory is
        FILE RamFile : text is in RamFileName;
        variable RamFileLine : line;
        variable RAM : TMemory;
        begin
            for I in TMemory'range loop
                readline (RamFile, RamFileLine);
                read (RamFileLine, RAM(I));
            end loop;
        return RAM;
    end function;

    signal RAM : TMemory := InitRamFromFile("Data.dat");
```

A function is impure if it refers to any variables or signals declared by its parents (any process, sub-program or architecture body in which the function declaration is nested)

Working with files

- readline operation reads a complete line of text from an input file (*RamFile*).

```
readline (RamFile, RamFileLine);
```

- It creates a string object *(RamFileLine)* in the host computer's memory and returns a pointer to the string.
- We then use various versions of the read operation to extract values of different types from the string:

```
read (RamFileLine, RAM(I));
```

- Each version of read has at least two parameters: a pointer to the line of text from which to read *(RamFileLine)* and a variable in which to store the value (*RAM(I)*).
- For bit-vector values, the literal in the line should be a binary string without quotation marks or a base specifier (that is, just a string of '0' or '1' characters).

Initializing a 16x8 bit RAM (1 port; **synchronous read**) – architecture

```
begin

process (clk) --write-first mode
    begin
    if clk'event and clk = '1' then
        if we = '1' then
            RAM(to_integer(unsigned(addr))) <= to_bitvector(din);
            dout <= din;
        else
            dout <= to_stdlogicvector(RAM(to_integer(unsigned(addr))));
        end if;
    end if;

end process;

end Behavioral;
```

write-first: new content is immediately made available for reading (a.k.a. read-through)

A function to_bitvector converts a standard logic vector to a bit vector

A function to_stdlogicvector converts a bit vector to a standard logic vector

Synthesis of memories

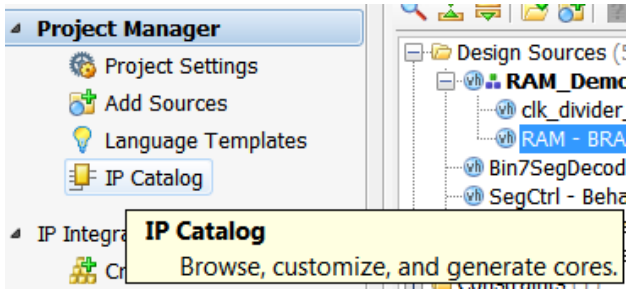
- Memory can be synthesized to either Distributed ROM/RAM (constructed from LUTs) or Block RAM (check the post-implementation resource utilization report).
- By default, the tool selects which RAM to infer, based upon heuristics that give the best results for most designs.
- Data is written synchronously into the RAM for both types.
- Reading is done synchronously for Block RAM and asynchronously for Distributed RAM.
- ram_style/rom_style attribute instructs the Vivado synthesis tool on how to infer memory:

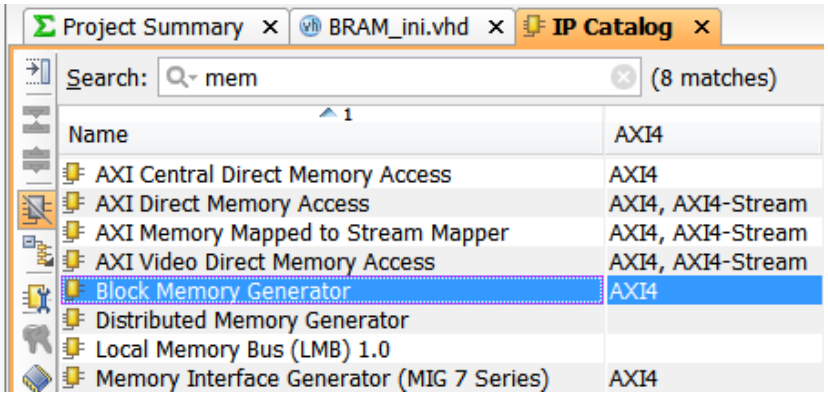
```
signal RAM : TMemory := InitRamFromFile("Data.dat");
attribute ram_style : string;
attribute ram_style of RAM : signal is "block";
attribute ram_style of RAM : signal is "distributed";
```

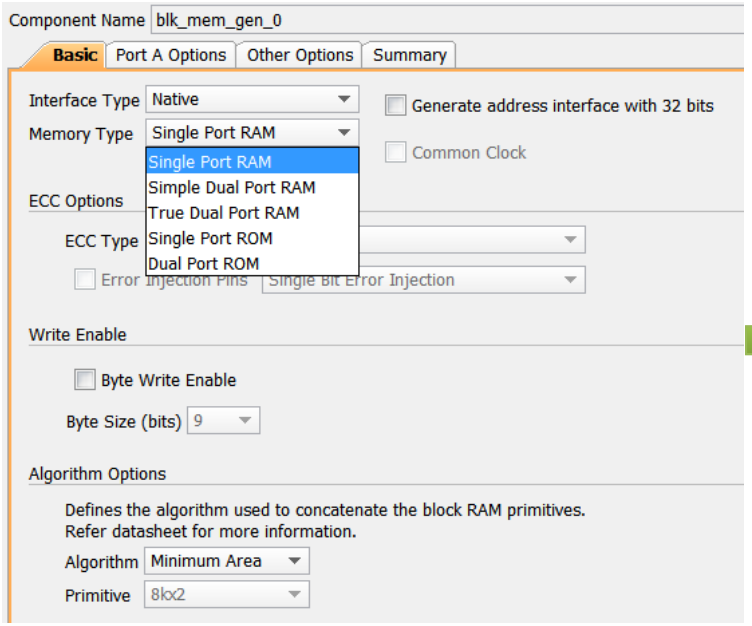
FPGA memory resources

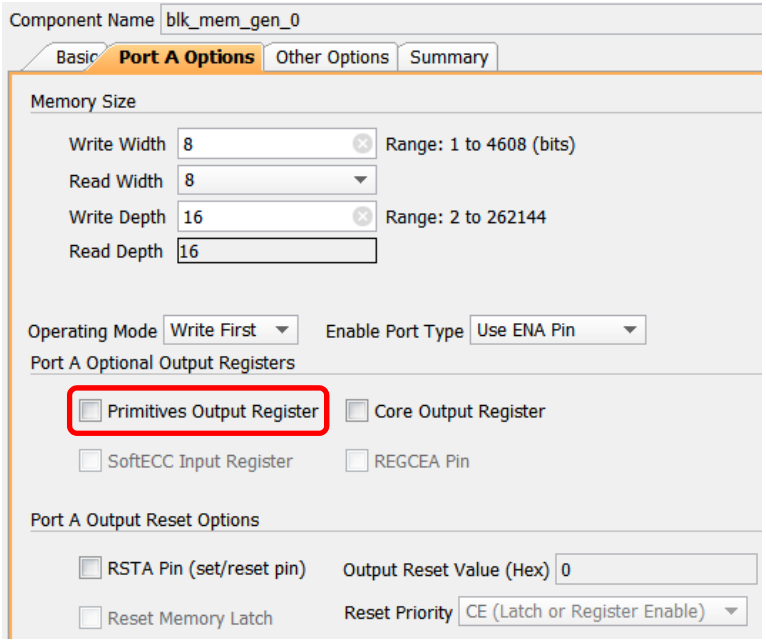
- The Artix-7™ FPGA xc7a100t-1csg324 contains 1,188 Kb of Distributed RAM and 4,860 Kb of Block RAM.
- Some of the key features of the Block RAM include:
  - Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks. There are 135 36 Kb blocks.
  - Each block RAM has two completely independent ports that share nothing but the stored data (see Vivado Synthesis Guide for modeling dual-port RAMs).
  - Each memory access, read or write, is controlled by the clock.
  - Each port can be configured as 32K × 1, 16K × 2, 8K × 4, 4K × 9 (or 8), 2K × 18 (or 16), 1K × 36 (or 32), or 512 × 72 (or 64).
  - The two ports can have different aspect ratios without any constraints.
  - During a write operation, the data output can reflect either the previously stored data (read-first), the newly written data (write-first), or can remain unchanged (no-change).

Using IP Catalog

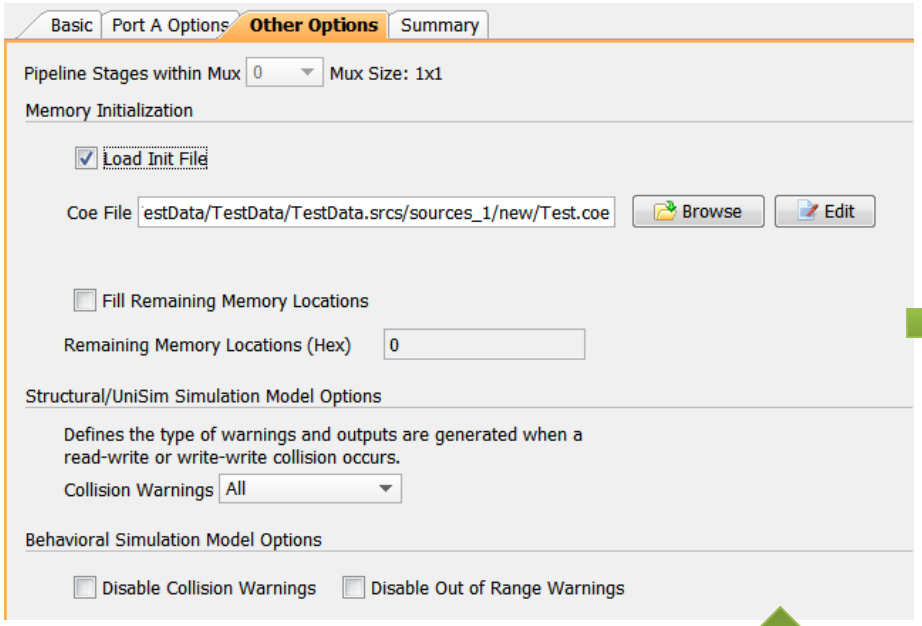


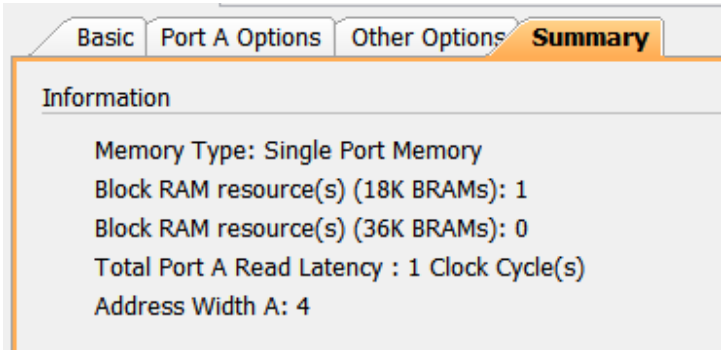






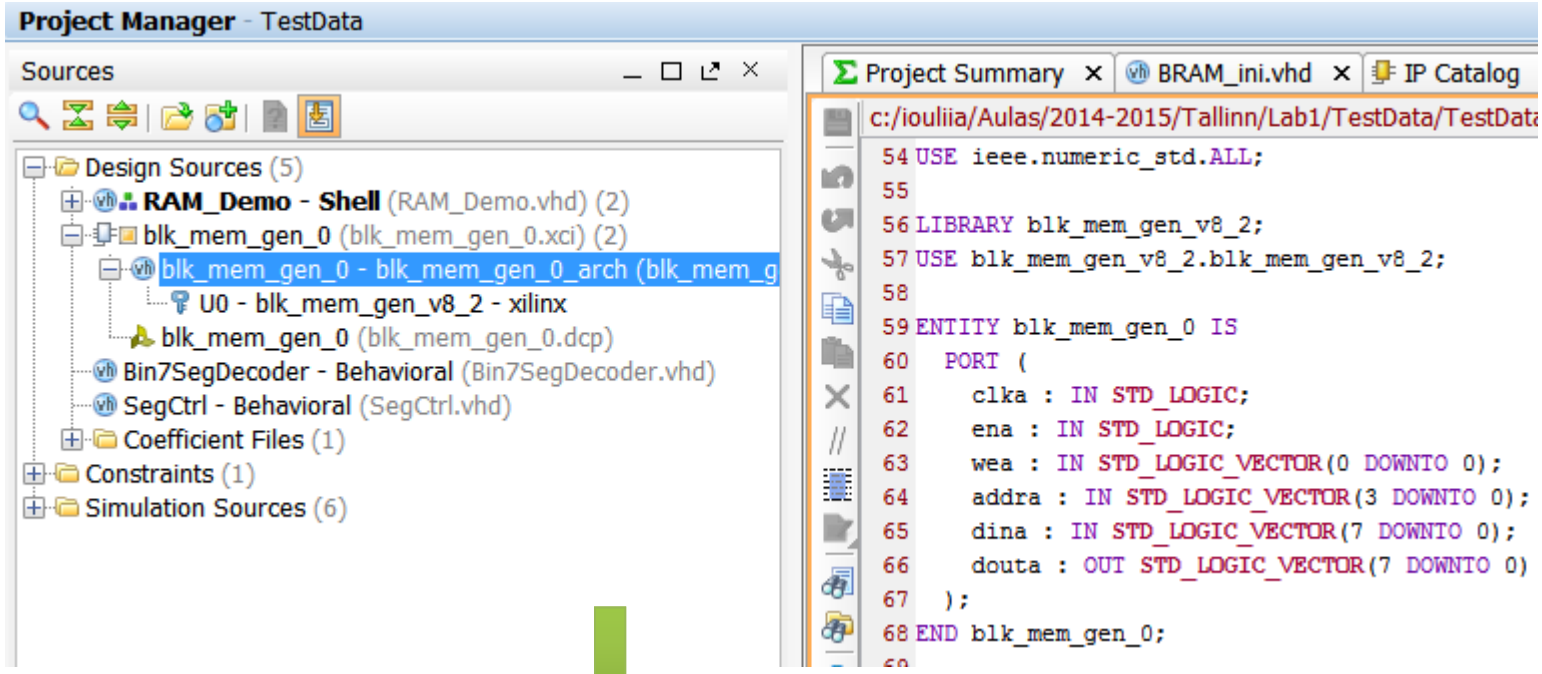
Initialization from files






```
memory_initialization_radix = 16;
memory_initialization_vector =  AB, BA, CD, DE, 99, 81, 71, 63,
                                A7, A6, 10, EC, 33, 71, 22, F4
```

Instantiation of modules from the IP Catalog




```
COMPONENT blk_mem_gen_0
 PORT (
      clka : IN STD_LOGIC;
      ena : IN STD_LOGIC;
      wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
      addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    END COMPONENT;
```

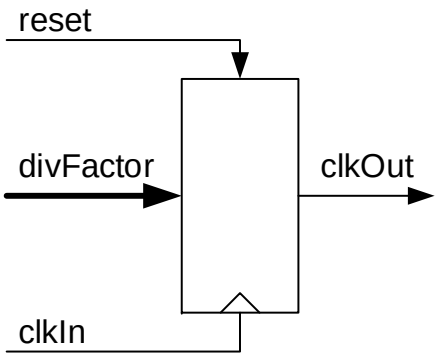
```
BRAM: blk_mem_gen_0
   port map(clka => s_clk1Hz,
            ena    => std_logic'('1'),
            wea    => s_wea,
            addra  => sw(11 downto 8),
            dina   => sw(7 downto 0),
            douta  => led);
```

Useful modules for Lab 1

- Parameterizable clock divider
- Parameterizable debouncer
- 7-segment display control for Nexys-4

Clock divider

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

entity ClkDividerN is
   generic(divFactor : positive);
   port(reset           : in  std_logic;
        clkIn           : in  std_logic;
        clkOut          : out std_logic);
end ClkDividerN;

architecture Behavioral of ClkDividerN is
   signal s_divCounter : natural;
begin
   process(reset, clkIn)
   begin
      if (reset = '1') then
            clkOut       <= '0';
            s_divCounter <= 0;
      elsif (rising_edge(clkIn)) then
            if (s_divCounter = divFactor - 1) then
               clkOut       <= '0';
               s_divCounter <= 0;
            else
               if (s_divCounter = (divFactor / 2 - 1)) then
                  clkOut    <= '1';
               end if;
               s_divCounter <= s_divCounter + 1;
            end if;
      end if;
   end process;
end Behavioral;
```



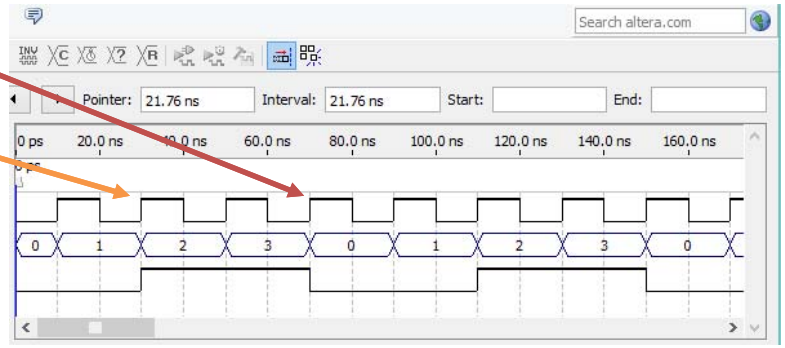
- The frequency division of a clock signal by integer factors, which are powers of 2, can be done by a binary counter
- The frequency division by arbitrary integer factors requires more elaborate hardware
- Example of a statically configurable frequency divider (*divFactor* is fixed at compile time upon instantiation with generic map)

Simulation of the clock divider

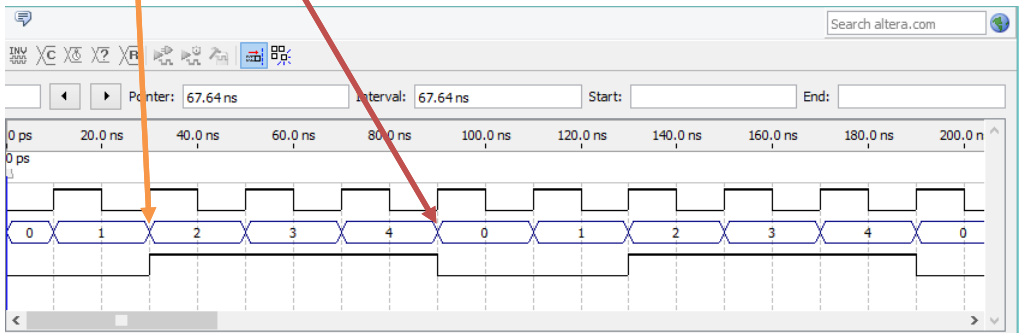
Let f_{clkIn} = 50 MHzc

```
elsif (rising_edge(clkIn)) then
   if (s_divCounter = divFactor - 1) then
      clkOut       <= '0';
      s_divCounter <= 0;
   else
      if (s_divCounter = (divFactor / 2 - 1)) then
         clkOut    <= '1';
      end if;
      s_divCounter <= s_divCounter + 1;
   end if;
end if;
```

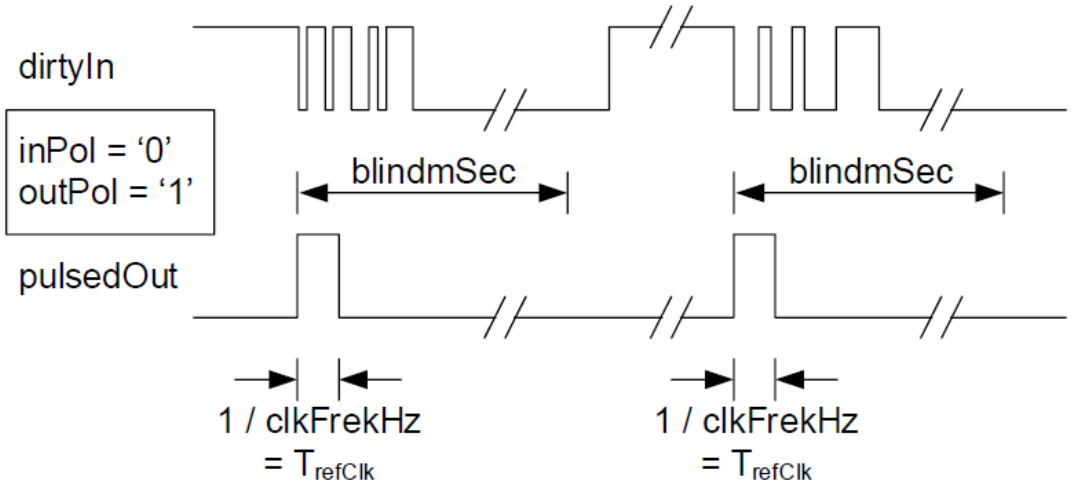
divFactor=4
f_{clkOut} ?
Duty cycle?



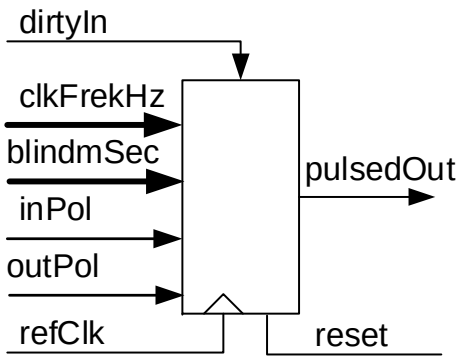
divFactor=5
f_{clkOut} ?
Duty cycle?



Parameterizable debouncer




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

entity DebounceUnit is
   generic(clkFrekHz   : positive;
           blindmSec   : positive;
           inPol       : std_logic;
           outPol      : std_logic);
   port( reset     : in  std_logic;
         refClk    : in  std_logic;
         dirtyIn   : in  std_logic;
         pulsedOut : out std_logic);
end DebounceUnit;
```

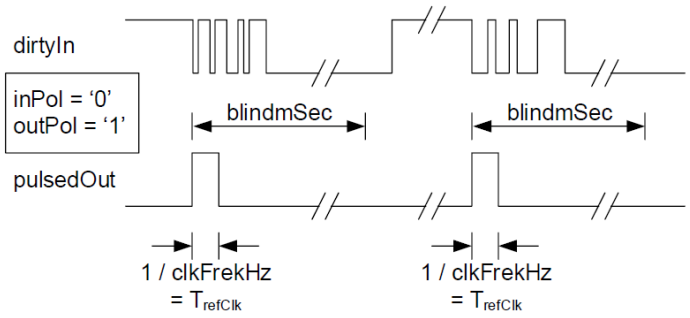


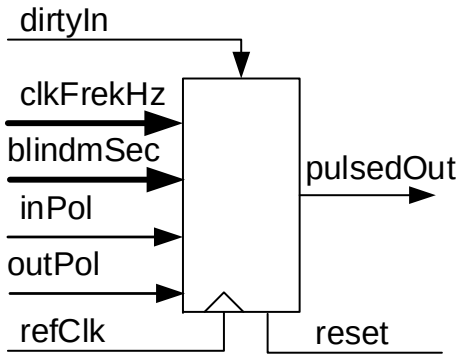
- Generic parameters:
  - clkFrekHz - frequency of the reference clock signal (in kHz)
  - blindmSec – blind time during which the circuit ignores input variations <1..100 ms>
  - inPol - input polarity ('0' for active low input; '1' for active high input)
  - outPol - output polarity ('0' for active low output; '1' for active high output)

Parameterizable debouncer VHDL code

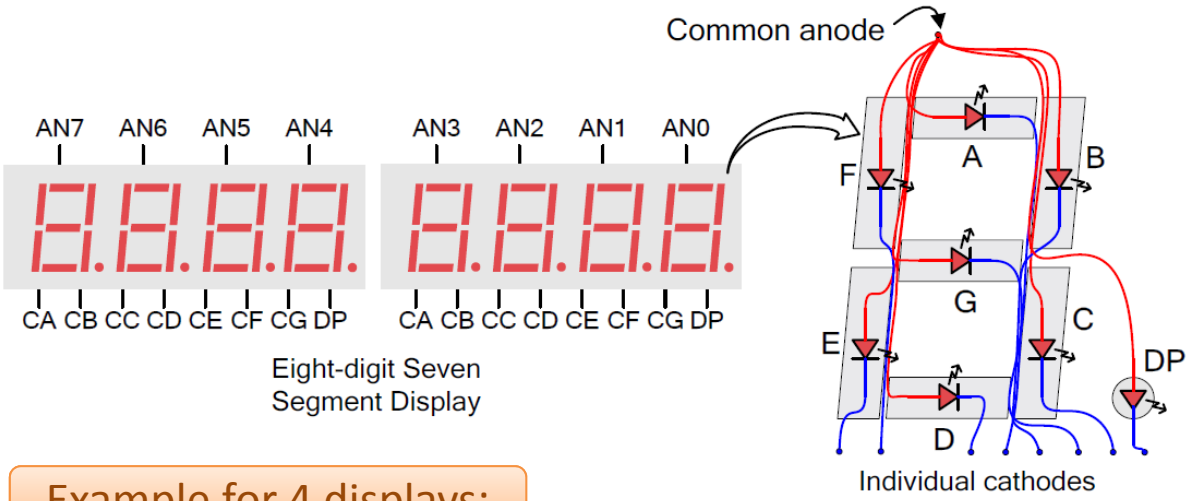
```
architecture Behavioral of DebounceUnit is
   signal s_dirtyIn, s_resetPulse, s_pulsedOut : std_logic;
   signal s_debounceCnt : natural;
begin
   sync_proc : process(refClk)
   begin
      if (rising_edge(refClk)) then
            s_dirtyIn <= dirtyIn;
      end if;
   end process;

   out_proc : process(reset, s_resetPulse, s_dirtyIn)
   begin
      if ((reset = '1') or (s_resetPulse = '1')) then
            s_pulsedOut <= not outPol;
      elsif ((s_dirtyIn'event) and s_dirtyIn = inPol) then
            s_pulsedOut <= outPol;
      end if;
   end process;

   pulsedOut <= s_pulsedOut;
```

```
   timer_proc : process(reset, refClk)
   begin
      if (reset = '1') then
            s_debounceCnt <= 0;
            s_resetPulse  <= '0';
      elsif (rising_edge(refClk)) then
            if (s_debounceCnt /= 0) then
               s_debounceCnt <= s_debounceCnt - 1;
               s_resetPulse  <= '1';
            elsif (s_pulsedOut = outPol) then
               s_debounceCnt <= blindmSec * clkFrekHz;
               s_resetPulse  <= '1';
            else
               s_resetPulse  <= '0';
            end if;
      end if;
   end process;
end Behavioral;
```

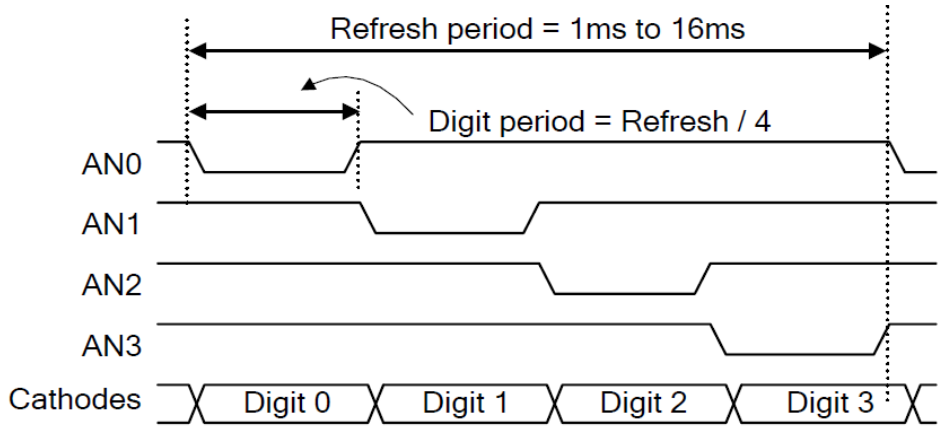




7-segment display control for Nexys-4



Example for 4 displays:

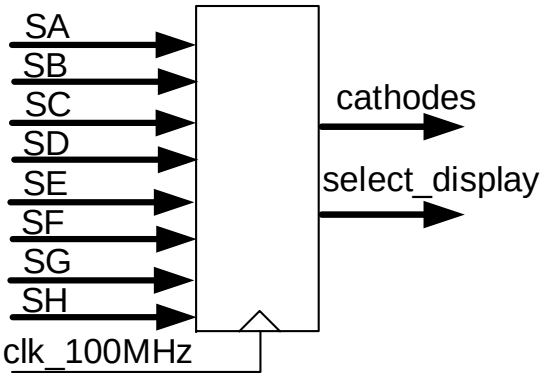


- The anodes of the seven LEDs forming each 7-segment display digit are tied together into one "common anode" circuit node.
- The LED cathodes remain separate. The cathodes of similar segments on all the displays are connected into seven circuit nodes.
- The common anode signals are available as eight "digit enable" input signals to the 8-digit display.
- The cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.
- To illuminate a segment, both the **anode** and the **cathode** are driven **low**.

Pictures from Nexys4™ FPGA Board Reference Manual

VHDL code of 7-segment display control for Nexys-4

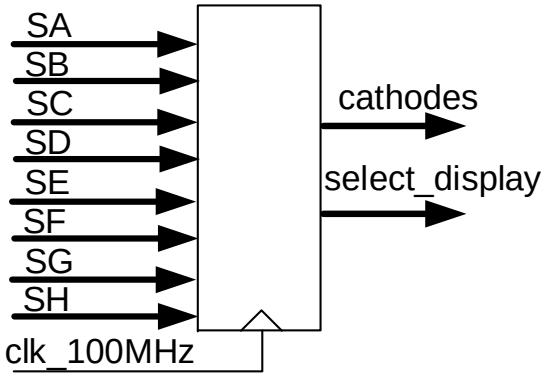
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SegCtrl is
   port ( clk_100MHz        : in  std_logic;
          SA, SB, SC, SD,
          SE, SF, SG, SH    : in  std_logic_vector (6 downto 0);
          cathodes          : out  std_logic_vector (6 downto 0);
          select_display    : out  std_logic_vector (7 downto 0));
end SegCtrl;

architecture Behavioral of SegCtrl is
   signal div      : unsigned(16 downto 0);
   signal clk_seg  : std_logic;
   signal state    : unsigned (2 downto 0);
begin

-- clock divider
divider: process(clk_100MHz)
begin
   if rising_edge(clk_100MHz) then
      div <= div + 1;
      clk_seg <= div(div'left);
   end if;
end process divider;
```



- Displays must be updated with a refresh frequency 60 Hz ... 1 KHz => clock period t
- We generate 8 times higher frequency 480 Hz ... 8 KHz and activate each display for 1/8 of t
- Division of 100 MHz by 2^17 gives ~763 Hz

VHDL code of 7-segment display control for Nexys-4

```
-- FSM that updates info on 7-segment displays in 8 cycles:
state_update: process (clk_seg)
begin
   if rising_edge(clk_seg) then
      state <= state + 1;
   end if;
end process state_update;

display_update: process (state, SA, SB, SC, SD, SE, SF, SG, SH)
begin
   case state is
      when "000" => cathodes <= SA; select_display <= "01111111";
      when "001" => cathodes <= SB; select_display <= "10111111";
      when "010" => cathodes <= SC; select_display <= "11011111";
      when "011" => cathodes <= SD; select_display <= "11101111";
      when "100" => cathodes <= SE; select_display <= "11110111";
      when "101" => cathodes <= SF; select_display <= "11111011";
      when "110" => cathodes <= SG; select_display <= "11111101";
      when "111" => cathodes <= SH; select_display <= "11111110";
      when others => cathodes <= (others => '1');
                     select_display <= (others => '1');
   end case;
end process display_update;

end Behavioral;
```



Summary

- After completing this class and lab 1 you should be able to:
  - create and implement simple VHDL designs in Vivado Design Suite
  - configure the FPGA and interact with Nexys-4 prototyping board
  - develop VHDL test benches and simulate your designs in the integrated Vivado simulator
  - use appropriate coding styles to specify and initialize memories in VHDL
  - create and instantiate modules from the IP Catalog
  - employ in your designs modules such as:
    - parameterizable frequency divider
    - parameterizable debouncer
    - 7-segment display controller
- … lab 1 is available at
  - http://sweet.ua.pt/iouliia/Courses/PDP_TUT/index.html