# Parallel Data Processing in Reconfigurable Systems

## Lecture 2

2014/2015

Parallel data sort

High-level synthesis

Iouliia Skliarova

# Contents

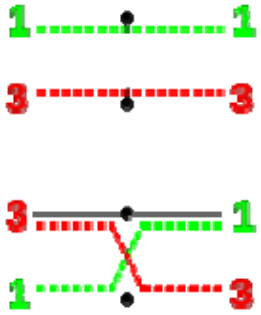
- Data sort in hardware
  - sorting networks
  - specification in VHDL
- Introduction to high-level synthesis (HLS)
  - Vivado HLS
  - HLS coding styles
  - synthesis directives
  - high-level verification
  - RTL cosimulation

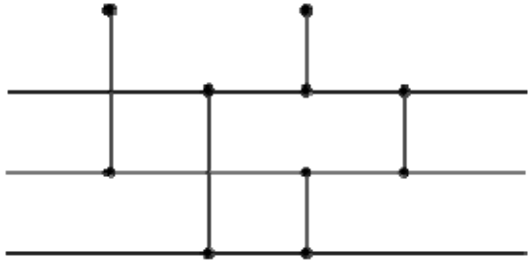
# Data sort

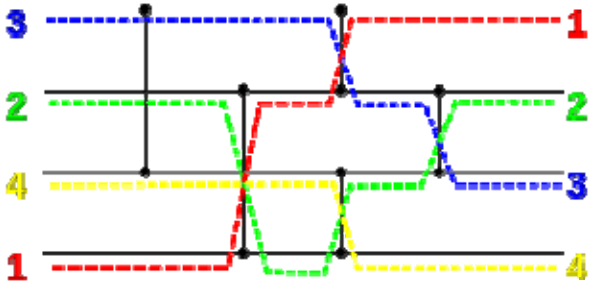
- A sorting algorithm is an algorithm that puts elements of a set in a certain order
- Efficient sorting is important for optimizing the use of other algorithms which require input data to be sorted
- Popular sorting algorithms:
  - insertion sort
  - merge sort
  - quicksort
  - different kinds of bubble sort (shell sort, comb sort)
  - different kinds of distribution sort (radix sort)
- Sorting algorithms are often classified by
  - computational complexity
  - memory and other resource usage
  - possibility to parallelize
  - adaptability (whether or not the presortedness of the input affects the running time)

# Sorting networks

- A sorting network is an algorithm that sorts a fixed number N of values using a fixed sequence of comparisons
- A sorting network consists of two types of items: comparators and wires. Each comparator connects two wires and swaps the input values if and only if the top wire's value is greater than the bottom wire's value
- Sorting networks are a very good choice for implementation in hardware because various comparisons can be executed in parallel
- Sorting networks are used to construct sorting algorithms to run on GPUs and FPGAs



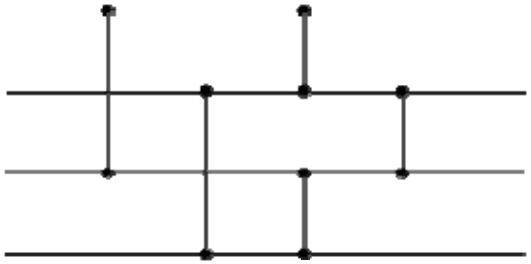






Pictures from http://en.wikipedia.org/wiki/Sorting_network

# Sorting network parameters

- Sorting networks are characterized by
  - the number of comparators C(N)
  - depth D(N) – the number of time steps required to execute it (assuming that all independent comparisons run in parallel)



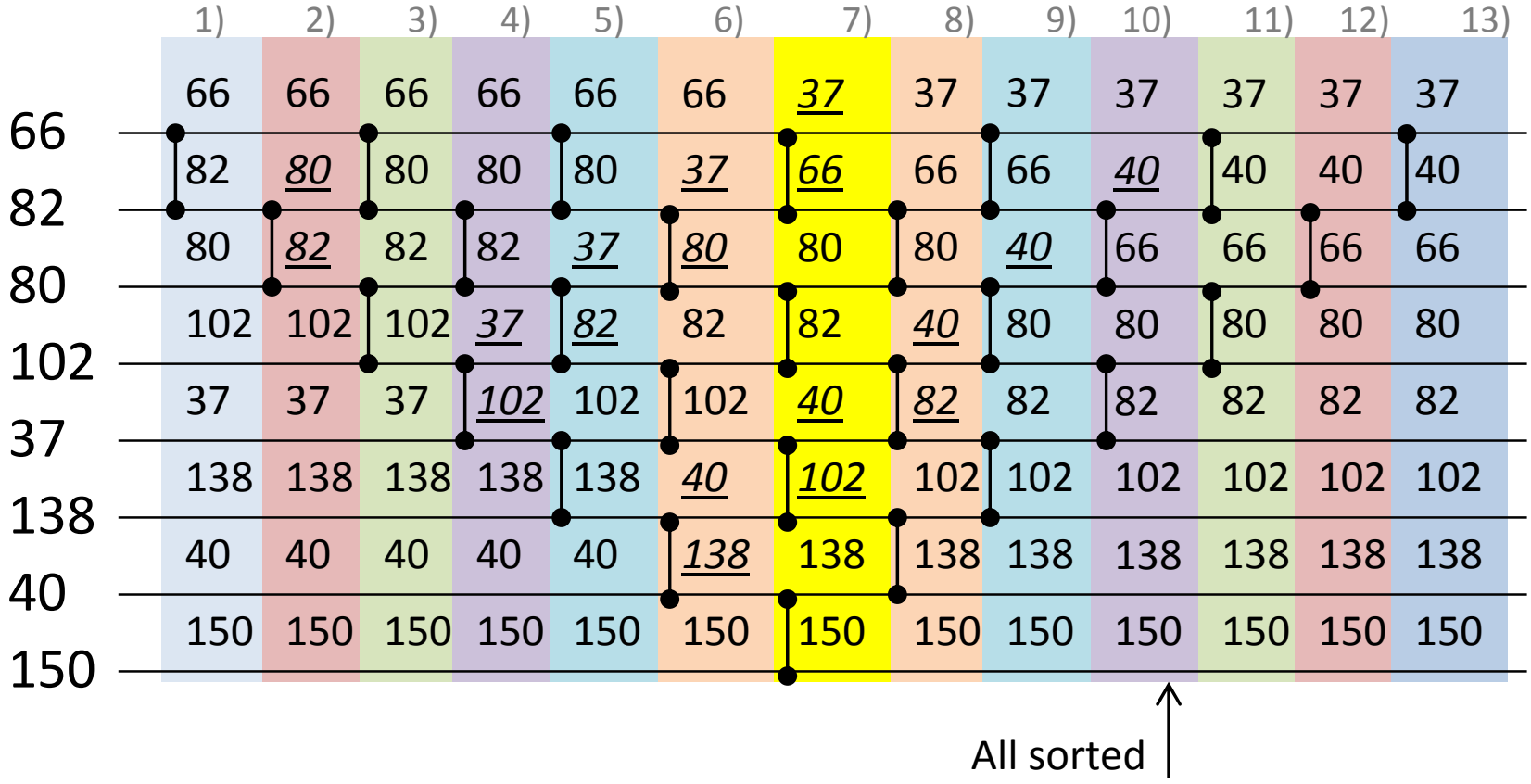
What is the depth of this network?

# Sorting network types

- Bubble/insertion
- Even-odd transition
- Even-odd merge
- Bitonic merge

# Bubble/insertion network

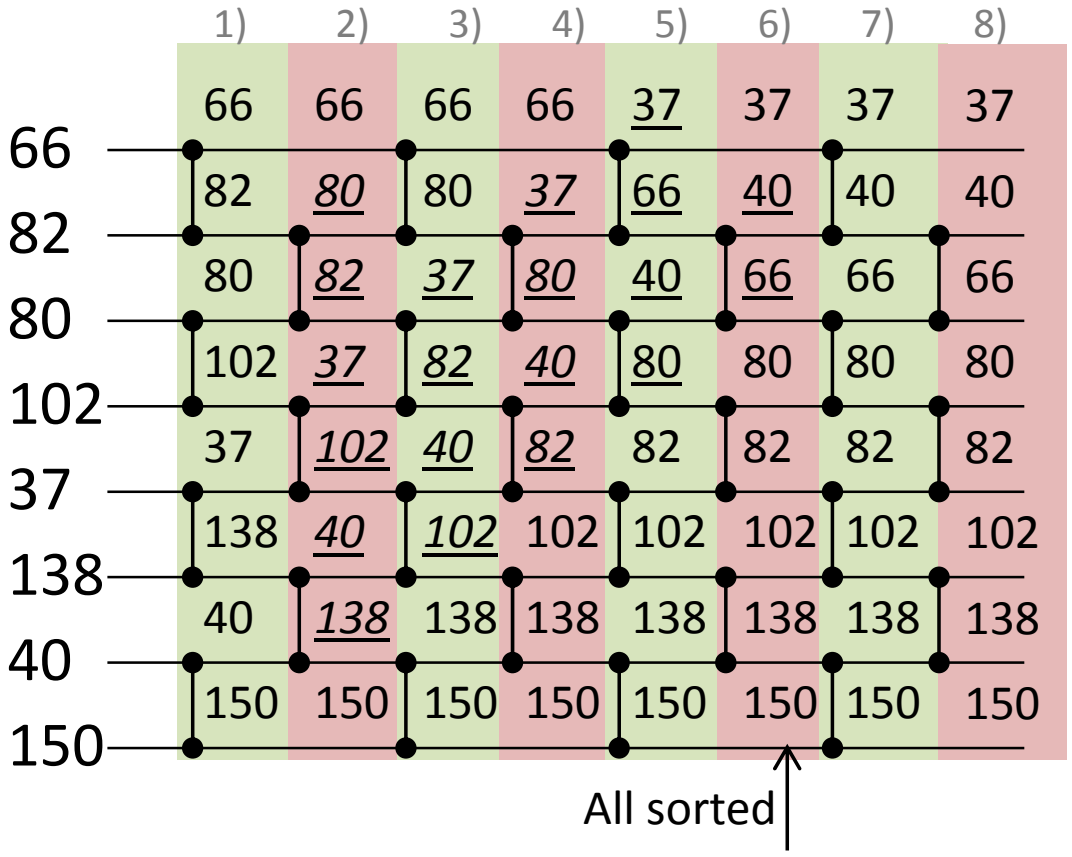
$D(N) = 2 \times N - 3$ $C(N) = N \times (N-1)/2$




N=8: D(N)=13 C(N)=28

N=1,024: D(N)=2,045 C(N)=523,776

# Even-odd transition network




$D(N) = N$

$C(N) = N \times (N-1)/2$

N=8:

D(N)=8

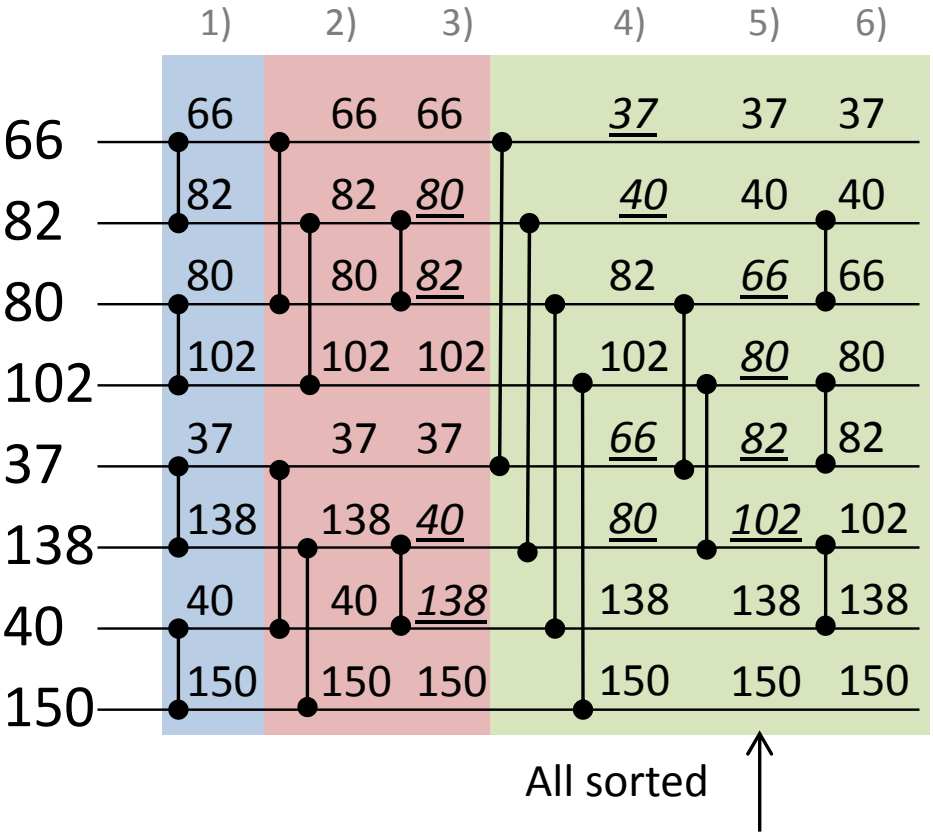
C(N)=28

N=1,024

D(N)=1,024

C(N)=523,776

# Even-odd merge network




$$D(N = 2^p) = p \times (p + 1)/2$$

$$C(N = 2^p) = (p^2 - p + 4) \times 2^{p-2} - 1$$

N=8:

D(N)=6

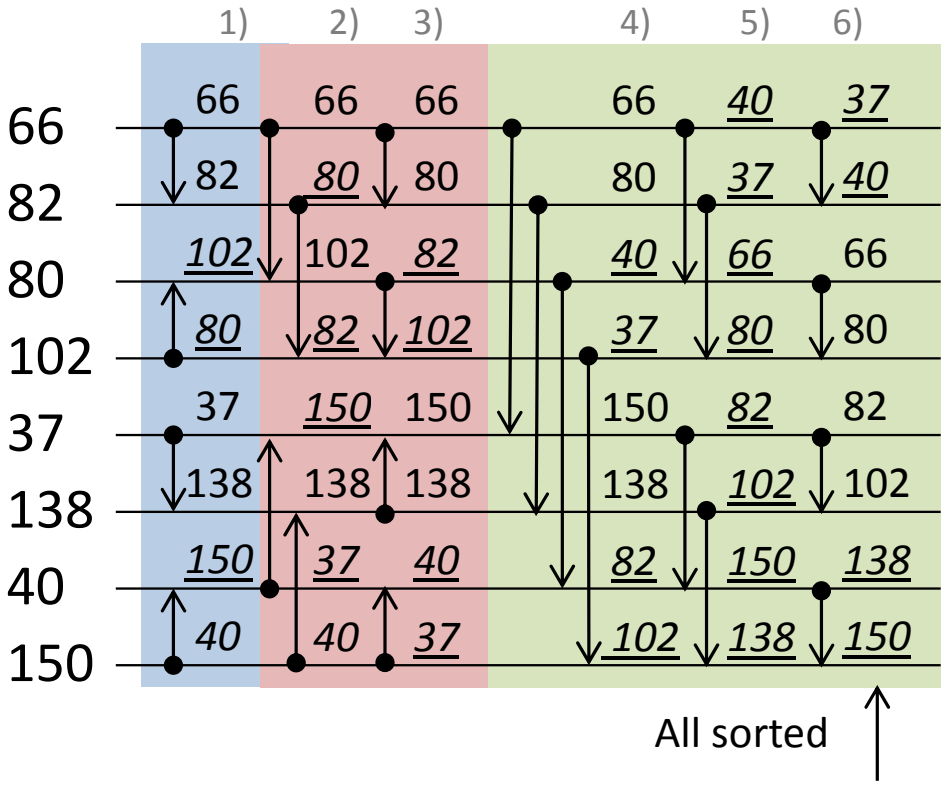
C(N)=19

N=1,024

D(N)=55

C(N)=24,063

# Bitonic merge network




$$D(N = 2^p) = p \times (p + 1)/2$$

$$C(N = 2^p) = (p^2 + p) \times 2^{p-2}$$

N=8:

D(N)=6

C(N)=24

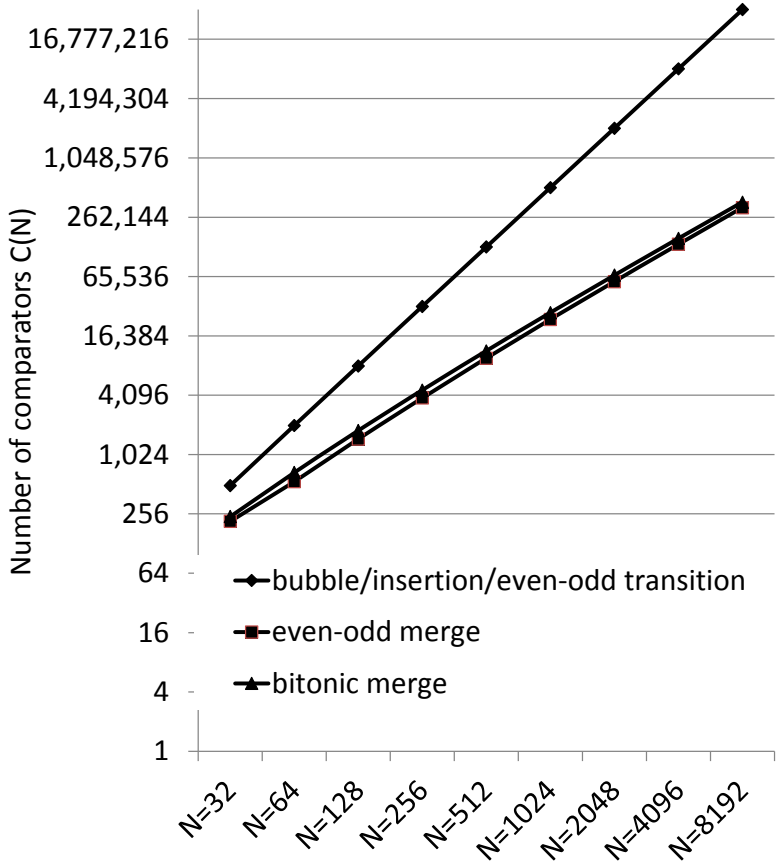
N=1,024

D(N)=55

C(N)=28,160

# Summary of sorting networks

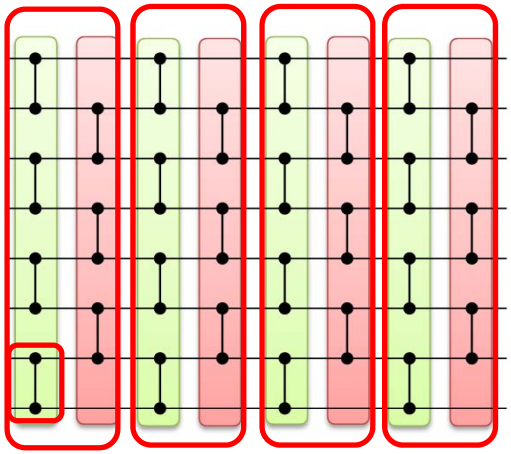
| Bubble/insertion | Even-odd transition | Even-odd merge | Bitonic merge |
|---|---|---|---|
| $C(N)=N\times(N-1)/2$ | $C(N)=N\times(N-1)/2$ | $C(N=2^p)=(p^2-p+4)\times2^{p-2}-1$ | $C(N=2^p)=(p^2+p)\times2^{p-2}$ |
| $D(N)=2\times N-3$ | $D(N) = N$ | $D(N=2^p)=p\times(p+1)/2$ | $D(N=2^p)=p\times(p+1)/2$ |




- The sorting networks are easily scalable
- Even-odd transition networks are the most regular
- Even-odd merge and bitonic merge networks are very fast:
  - to sort 134 million data items (~$2^{27}$) just 378 steps are required ☺
- Are the required hardware resources prohibitive?
  - 23,689,428,991 comparators are needed for the even-odd merge network ☹

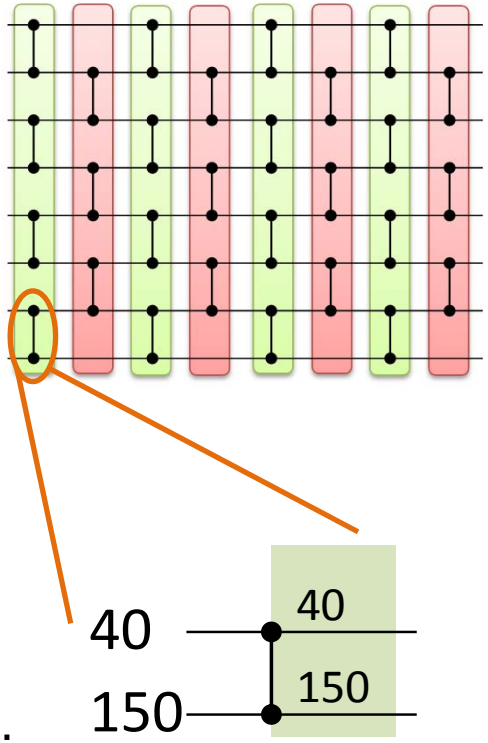
# Implementation in FPGA

- Even-odd transition networks have the most regular structure
- VHDL specification for sorting N M-bit data items can be based on
  - 2-input, 2-output M-bit comparator
  - a pair of comparator lines
    - the even line has N/2 comparators
    - the odd line has N/2-1 comparators
  - the sorter containing N/2 pairs of comparator lines
- The sorter can be implemented as a regular iterative composition of subsystems



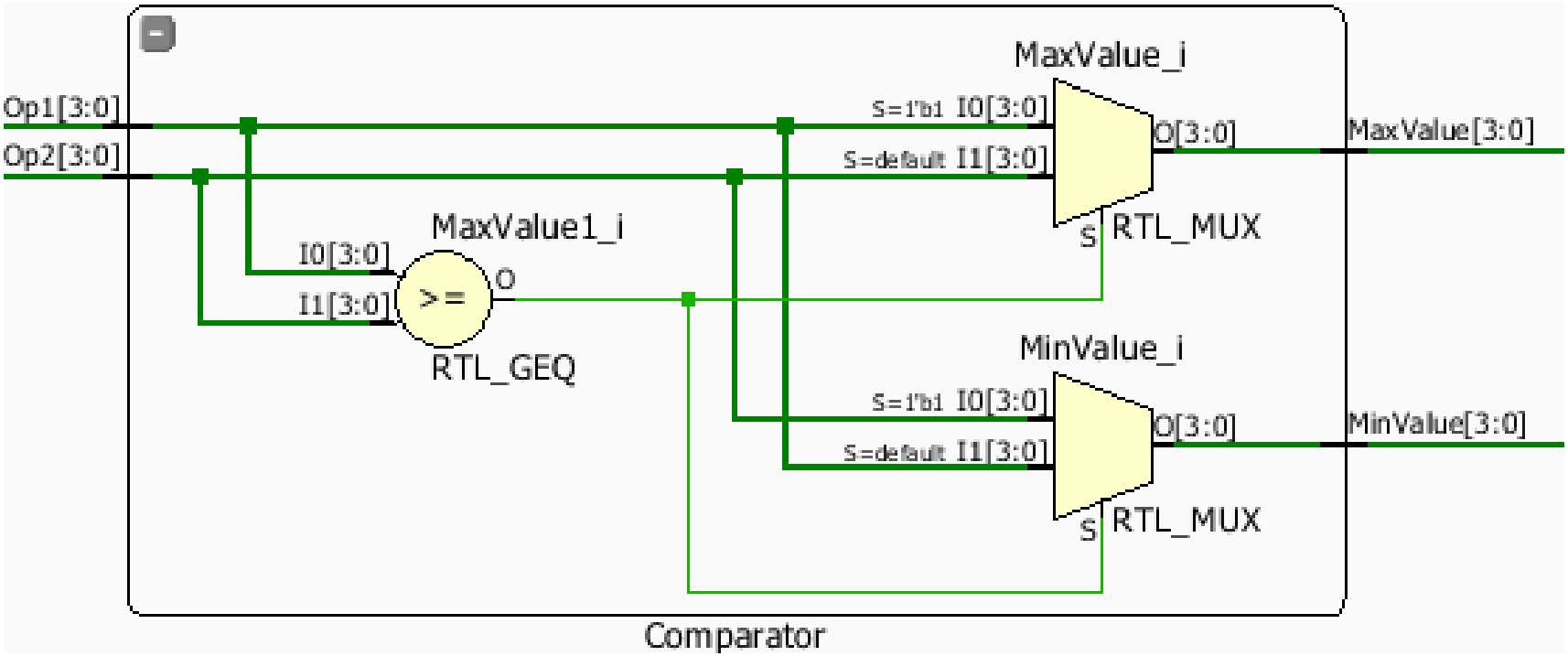
# VHDL specification of even-odd transition network - comparator

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

entity Comparator is
    generic (M     : positive);
    port(Op1, Op2  : in std_logic_vector(M - 1 downto 0);
         MaxValue  : out std_logic_vector(M - 1 downto 0);
         MinValue  : out std_logic_vector(M - 1 downto 0));
end Comparator;

architecture Behavioral of Comparator is
begin

    process(Op1,Op2)
    begin
       if Op1 >= Op2 then   MaxValue <= Op1; MinValue <= Op2;
       else                 MaxValue <= Op2; MinValue <= Op1;
       end if;
    end process;

end Behavioral;
```



# Elaborated design - comparator

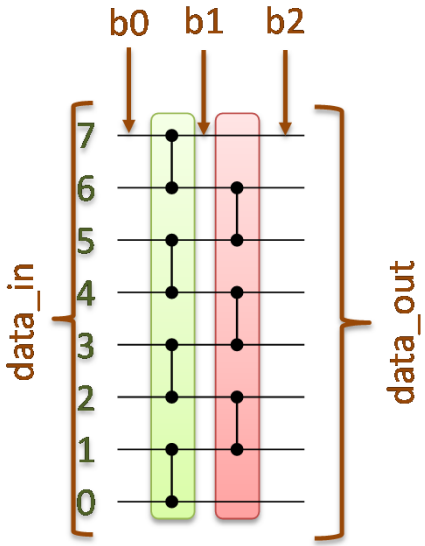
- For M = 4 the elaborated design is:

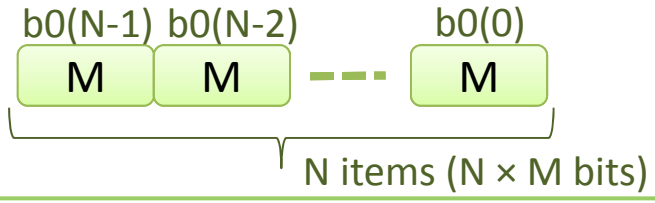


# VHDL specification of even-odd transition network – even and odd lines

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

entity Two_lines_sorter is
    generic( M       : positive;
             p       : positive);
    port ( data_in   : in std_logic_vector(M * 2 ** p - 1 downto 0);
           data_out  : out std_logic_vector(M * 2 ** p - 1 downto 0));
end Two_lines_sorter;

architecture Behavioral of Two_lines_sorter is
    constant N  : positive := 2 ** p;
    type BETWEEN_LEVELS is array (0 to N - 1) of
                           std_logic_vector(M - 1 downto 0);
    signal b0, b1, b2 : BETWEEN_LEVELS;
begin
    process (data_in)
    begin
    for i in 0 to N - 1 loop
        b0(i) <= data_in(M * (i + 1) - 1 downto M * i);
    end loop;
    end process;

-- contunues
```

A for loop includes a specification of how many times the body of the loop is to be executed

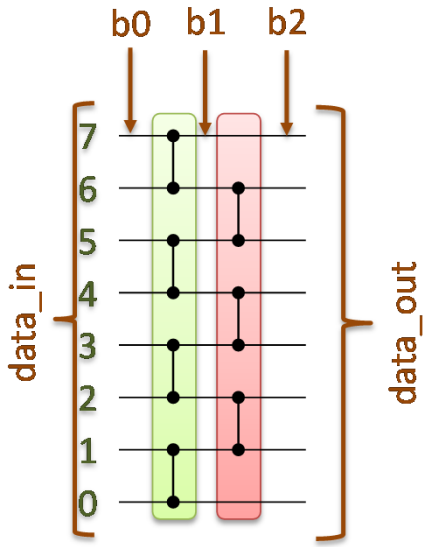


# VHDL specification of even-odd transition network – even and odd lines

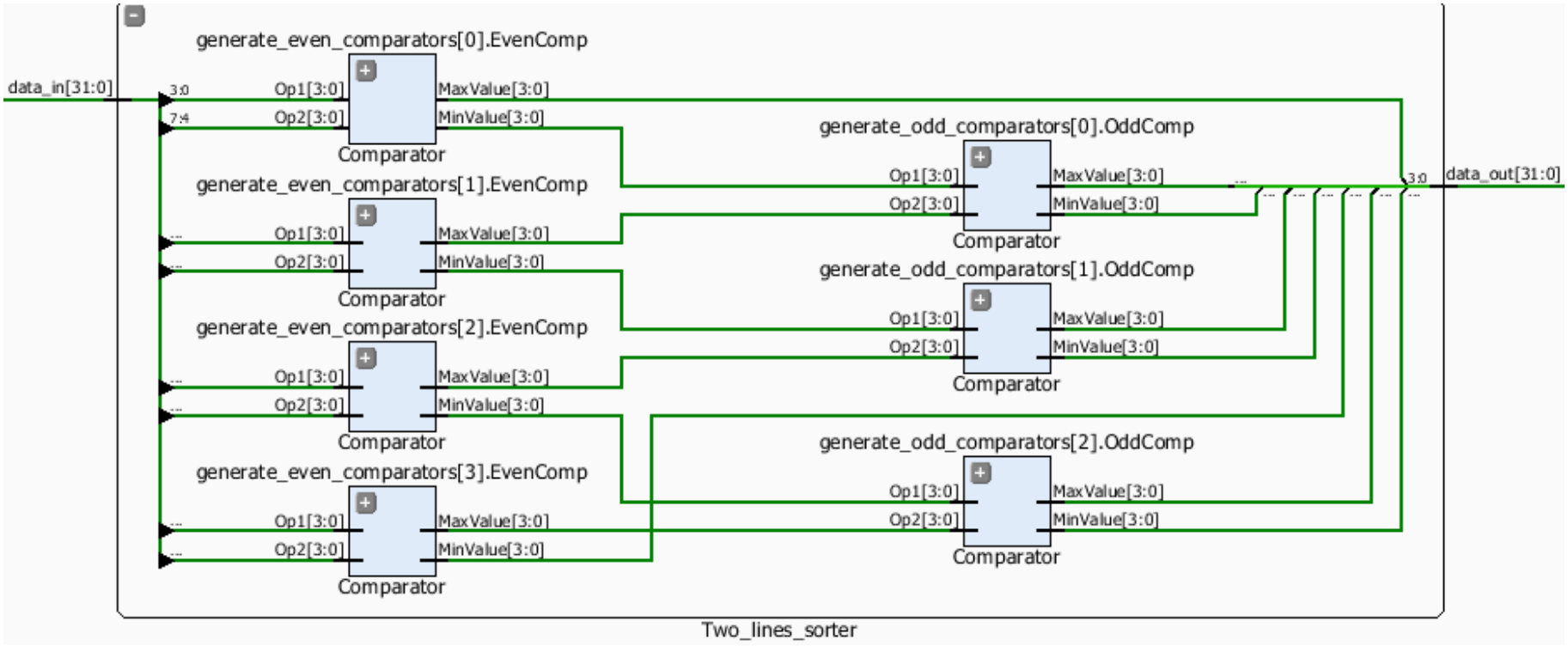
A generate statement is used for replicating subsystems

```
generate_even_comparators:
    for i in 0 to N / 2 - 1 generate
    EvenComp:  entity work.Comparator(Behavioral)
          generic map (M)
          port map(b0(i*2), b0(i*2+1), MaxValue => b1(i*2), MinValue => b1(i*2+1));
end generate generate_even_comparators;

generate_odd_comparators:
    for i in 0 to N / 2 - 2 generate
    OddComp:  entity work.Comparator(Behavioral)
          generic map (M)
          port map(b1(2*i+1), b1(2*i+2), MaxValue => b2(i*2+1), MinValue => b2(i*2+2));
    end generate generate_odd_comparators;

    b2(0)   <= b1(0);          b2(N-1) <= b1(N-1);

    process (b2)
    begin
    for i in 0 to N - 1 loop
        data_out(M * (i + 1) - 1 downto M * i) <= b2(i);
    end loop;
    end process;

end Behavioral;
```



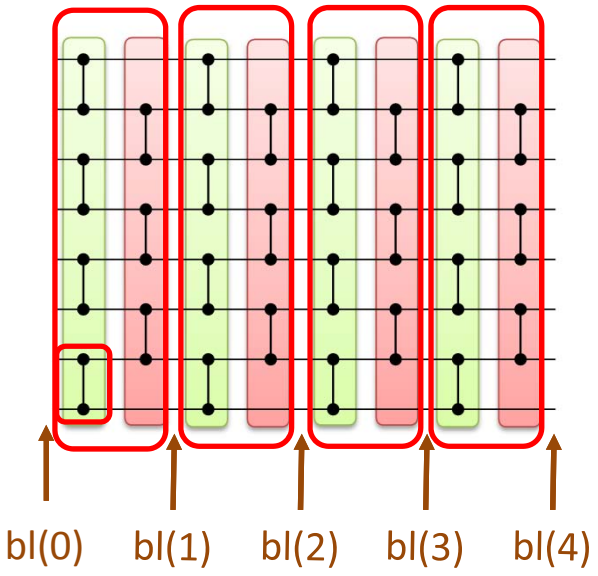
# Elaborated design – a pair of comparator lines

- For N = 8 and M = 4 the elaborated design is:



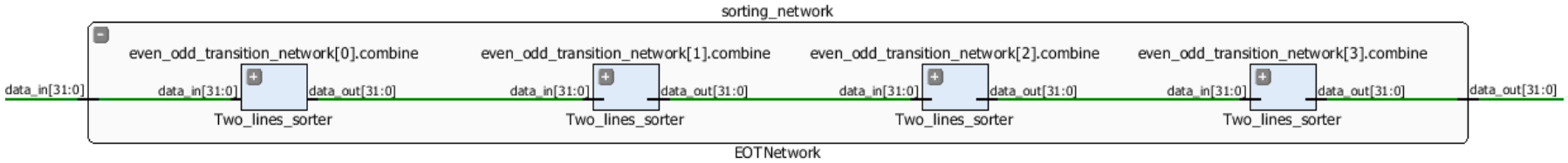
# VHDL specification of even-odd transition network – the complete sorter

```
entity EOTNetwork is
    generic(M       : positive;
            p       : positive );
    port   (data_in : in std_logic_vector(M * 2 ** p - 1 downto 0);
            data_out: out std_logic_vector(M * 2 ** p - 1 downto 0));
end EOTNetwork;

architecture Behavioral of EOTNetwork is
    constant N : positive := 2 ** p;
    type BETWEEN_LEVELS is array (0 to N/2) of std_logic_vector(M * 2**p - 1 downto 0);
    signal bl : BETWEEN_LEVELS;
begin

    bl(0) <= data_in;

even_odd_transition_network:
    for i in 0 to N / 2 - 1 generate
        combine: entity
                 work.Two_lines_sorter(Behavioral)
            generic map (M, p)
            port map    (bl(i), bl(i+1));
end generate even_odd_transition_network;

    data_out <= bl(N / 2);

end Behavioral;
```



# Elaborated design – the complete sorter

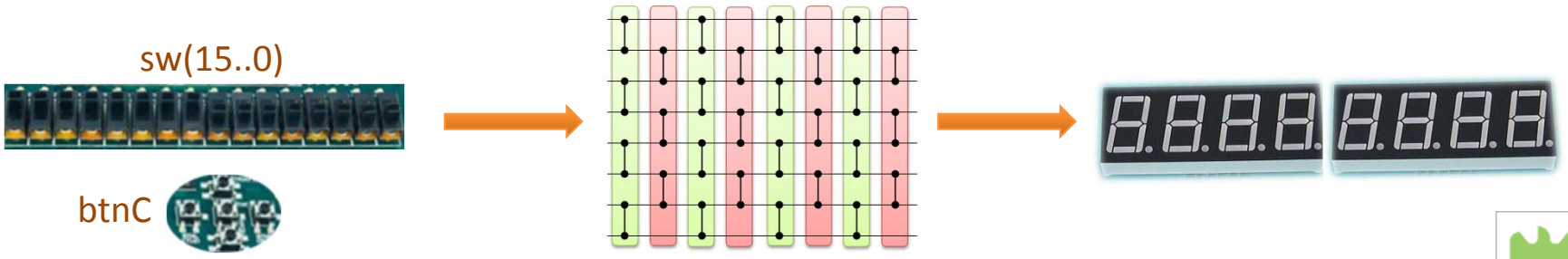
- For N = 8 and M = 4 the elaborated design is:



# Test of the even-odd transition network

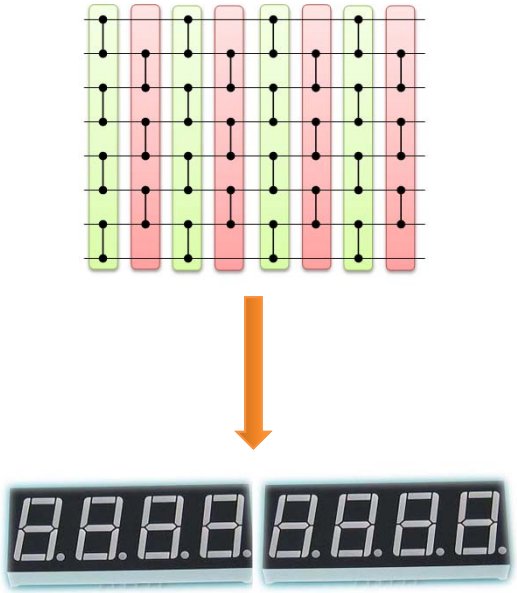
```
entity Top_EOTN is
   port (clk              : in  std_logic;
         btnC             : in  std_logic;
         sw               : in std_logic_vector(15 downto 0);
         seg              : out std_logic_vector(6 downto 0);
         an               : out std_logic_vector(7 downto 0));
end Top_EOTN;

architecture Shell of Top_EOTN is
    signal s_test_data, s_result : std_logic_vector(31 downto 0);
    type SEGMENTS is array (0 to 7) of std_logic_vector(6 downto 0);
    signal HEX : SEGMENTS;
begin

    s_test_data <= sw(15 downto 0) & X"FEDC" when btnC = '0' else
                   X"1234" & sw (15 downto 0);

-- contunues
```

N = 8 (p = 3)
M = 4



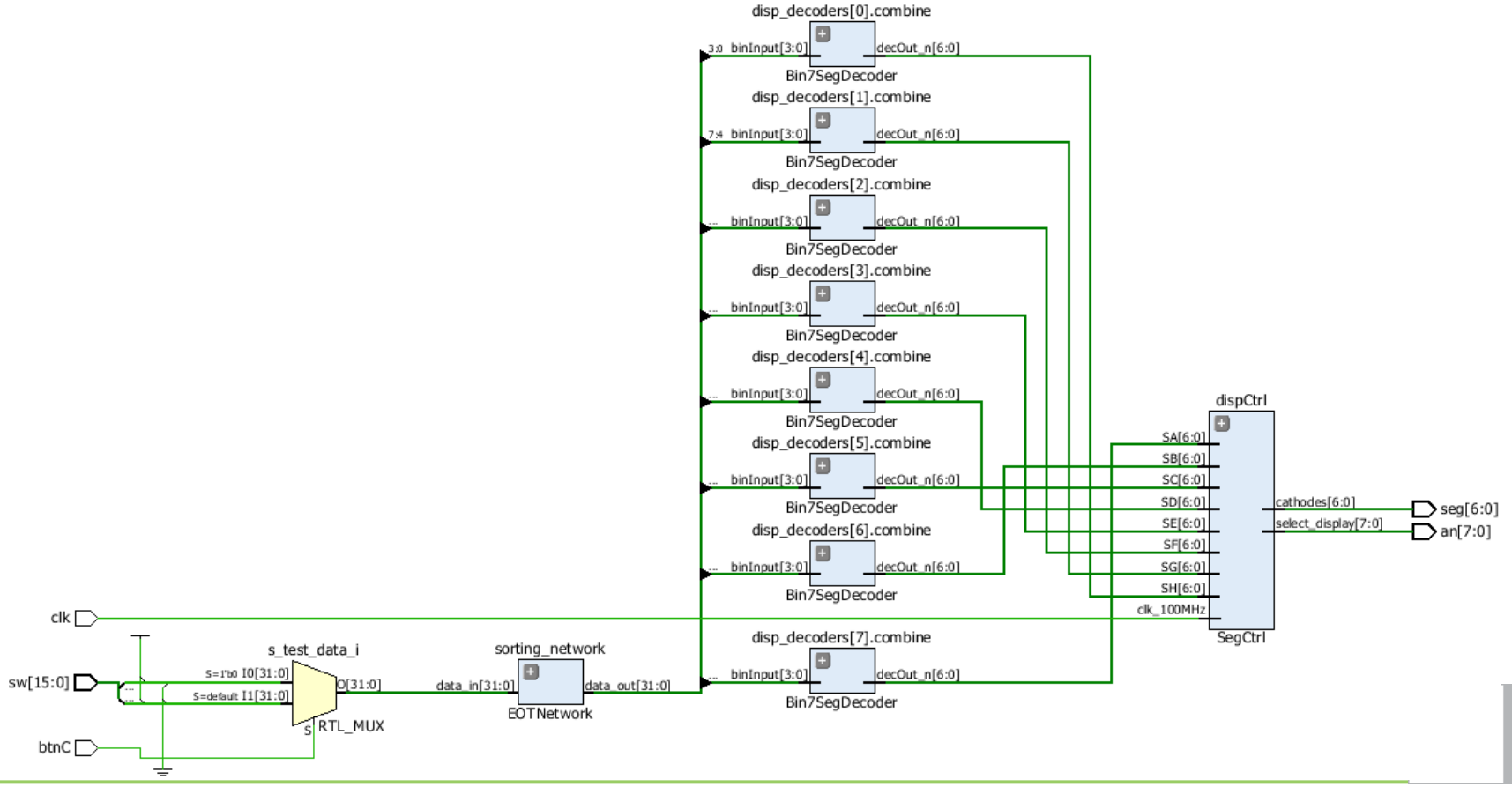
# Test of the even-odd transition network

```
sorting_network: entity work.EOTNetwork(Behavioral)
    generic map (M => 4, p => 3)
    port map   (data_in => s_test_data,
                data_out => s_result);

dispCtrl: entity WORK.SegCtrl(Behavioral)
    port map(clk_100MHz     => clk,
             SA             => HEX(7),
             SB             => HEX(6),
             SC             => HEX(5),
             SD             => HEX(4),
             SE             => HEX(3),
             SF             => HEX(2),
             SG             => HEX(1),
             SH             => HEX(0),
             cathodes       => seg,
             select_display =>  an);

disp_decoders: for i in 0 to 7 generate
        combine: entity work.Bin7SegDecoder(Behavioral)
            port map(binInput   => s_result(i * 4 + 3 downto i * 4),
                     decOut_n   => HEX(i));
end generate disp_decoders;

end Shell;
```



# Elaborated design – demo circuit

- For N = 8 and M = 4 the elaborated design is:



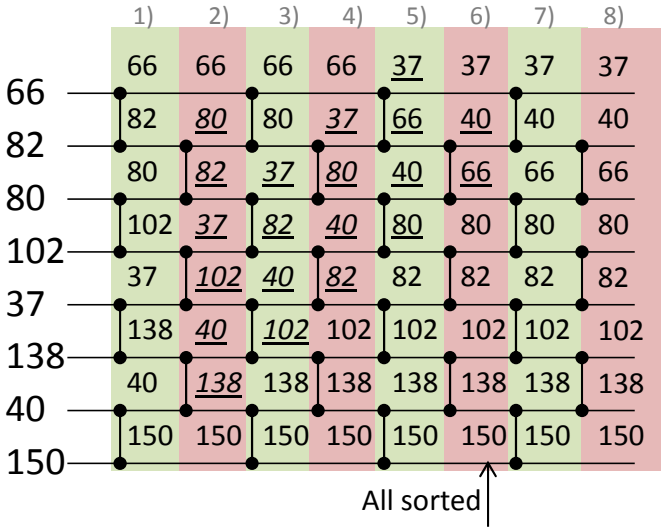
# Hardware resources

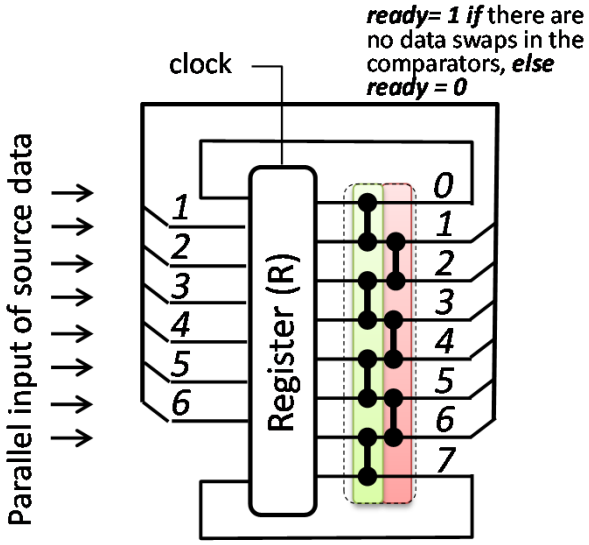
| | | |
|---|---|---|
| **Xilinx Spartan-6 xc6slx45** | $S_{FPGA}$ | 6,822 |
| | $S_{comp}$ | 21 |
| | $K_{comp}^{max}$ | 324 |
| **Xilinx Virtex-5 xc5vlx110t** | $S_{FPGA}$ | 17,280 |
| | $S_{comp}$ | 43 |
| | $K_{comp}^{max}$ | 401 |
| **Xilinx Virtex-5 xc5vfx130t** | $S_{FPGA}$ | 20,480 |
| | $S_{comp}$ | 55 |
| | $K_{comp}^{max}$ | 384 |
| **Xilinx Virtex-6 xc6vlx240t** | $S_{FPGA}$ | 37,680 |
| | $S_{comp}$ | 27 |
| | $K_{comp}^{max}$ | 1,395 |
| **Xilinx Virtex-6 xc6vlx760** | $S_{FPGA}$ | 118,560 |
| | $S_{comp}$ | 21 |
| | $K_{comp}^{max}$ | 5,645 |
| **Xilinx APSoC Zynq XC7Z020** | $S_{FPGA}$ | 13,300 |
| | $S_{comp}$ | 17 |
| | $K_{comp}^{max}$ | 782 |
| **Xilinx Virtex-7 xc7vx1140t** | $S_{FPGA}$ | 178,000 |
| | $S_{comp}$ | 25 |
| | $K_{comp}^{max}$ | **7,120** |
| **Altera Cyclon-IVe EP4CE115** | $LE_{FPGA}$ | 114,480 |
| | $LE_{comp}$ | 96 |
| | $K_{comp}^{max}$ | 1,192 |

- $S_{FPGA}/LE_{FPGA}$ - number of FPGA slices/logic elements
- $S_{comp}/LE_{comp}$ - number of FPGA slices/logic elements needed for one comparator (for M=32-bit data)
- $K_{comp}^{max}$ - maximum number of comparators that can be accommodated in one FPGA/APSoC
- To implement combinationally the *even-odd merge* network for N=1,024, M=32, 24,063 comparators are required

# Iterative sorting networks

- An iterative even-odd transition network iteratively reuses the same N-1 comparators through a feedback register
- The resulting circuit is very regular and easily scalable
- Since the traditional even-odd transition networks are hardwired, the latency is fixed: N × t (t is a delay of any vertical line of comparators)
- The latency of the iterative even-odd transition network depends on the presortedness of input data and may be smaller
- The required hardware resources are reduces drastically (from C(N) = N × (N-1)/2 to C(N) = N-1) permitting data sorters to be constructed for significantly bigger values of N

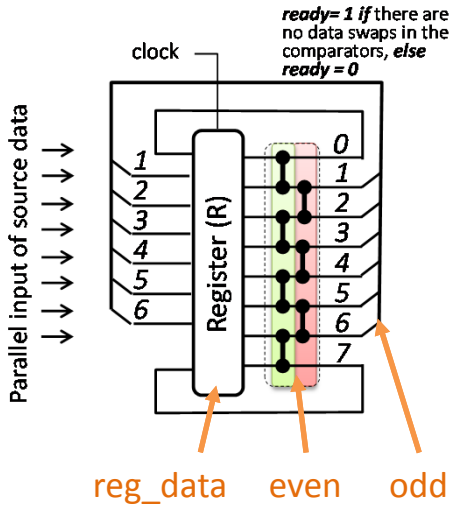


# VHDL specification of the iterative even-odd transition network

```
entity EvenOddIterSorterRTL is
    generic (M : positive;  -- M is the size of a data item
             N : positive); -- N is the number of data items
    port   (clk         : in std_logic;
            reset       : in std_logic;
            ready       : out std_logic;
            input_data  : in std_logic_vector(N * M - 1 downto 0);
            sorted_data : out std_logic_vector(N * M - 1 downto 0));
end EvenOddIterSorterRTL;

architecture Behavioral of EvenOddIterSorterRTL is
    type DATA_SET is array (N-1 downto 0) of std_logic_vector(M-1 downto 0);
    signal reg_data, even, odd : DATA_SET;
begin

data_register: process(clk)
begin
    if rising_edge(clk) then
        ready <= '0';
        if reset = '1' then
            for i in N downto 1 loop
                reg_data(i-1) <= input_data(i*M-1 downto (i-1)*M);
            end loop;
        else
            reg_data <= odd;
            if reg_data = odd then
                ready <= '1';
            end if;
        end if;
    end if;
end process data_register;
```

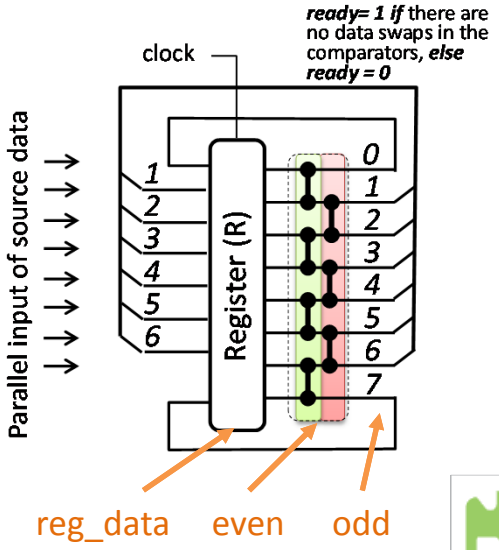
Is it safe to make such a test?




-- contunues

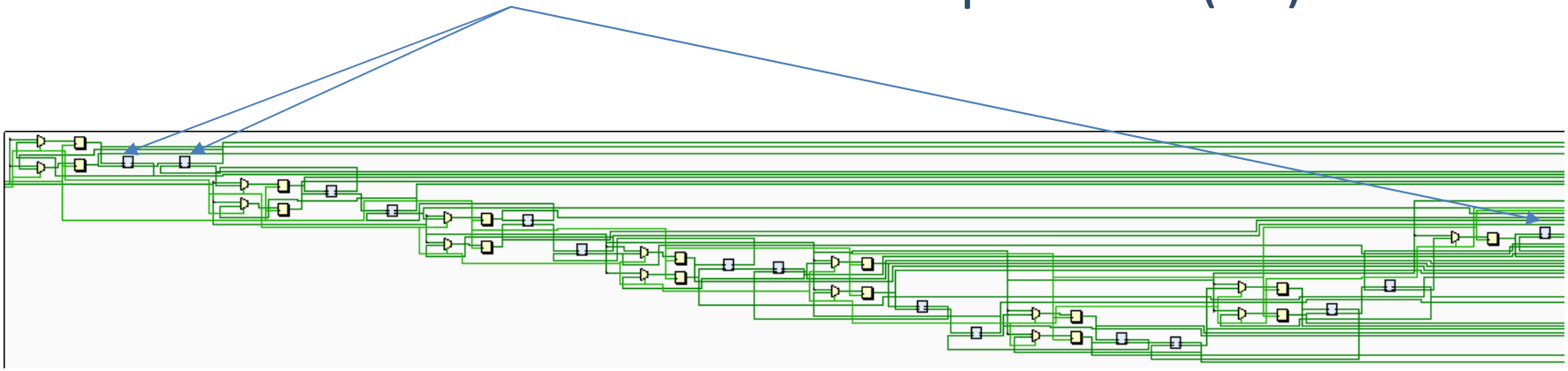
# VHDL specification of the iterative even-odd transition network

```
generate_even_comparators:  for i in N/2-1 downto 0 generate
    EvenComp: entity work.Comparator
        generic map (M => M)
        port map(reg_data(i*2), reg_data(i*2+1), even(i*2), even(i*2+1));
end generate generate_even_comparators;

generate_odd_comparators: for i in N/2-2 downto 0 generate
    OddComp: entity work.Comparator(Behavioral)
        generic map (M => M)
        port map(even(2*i+1), even(2*i+2), odd(i*2+1), odd(i*2+2));
end generate generate_odd_comparators;

    odd(0) <= even(0);
    odd(N-1) <= even(N-1);

output_data: for i in N downto 1 generate
    sorted_data(i*M-1 downto (i-1)*M) <= reg_data(i-1);
end generate output_data;

end Behavioral;
```



# Elaborated design – the complete iterative even-odd transition network

- For N = 16 and M = 8 the elaborated design is:
  - the blue modules are the comparators (15)

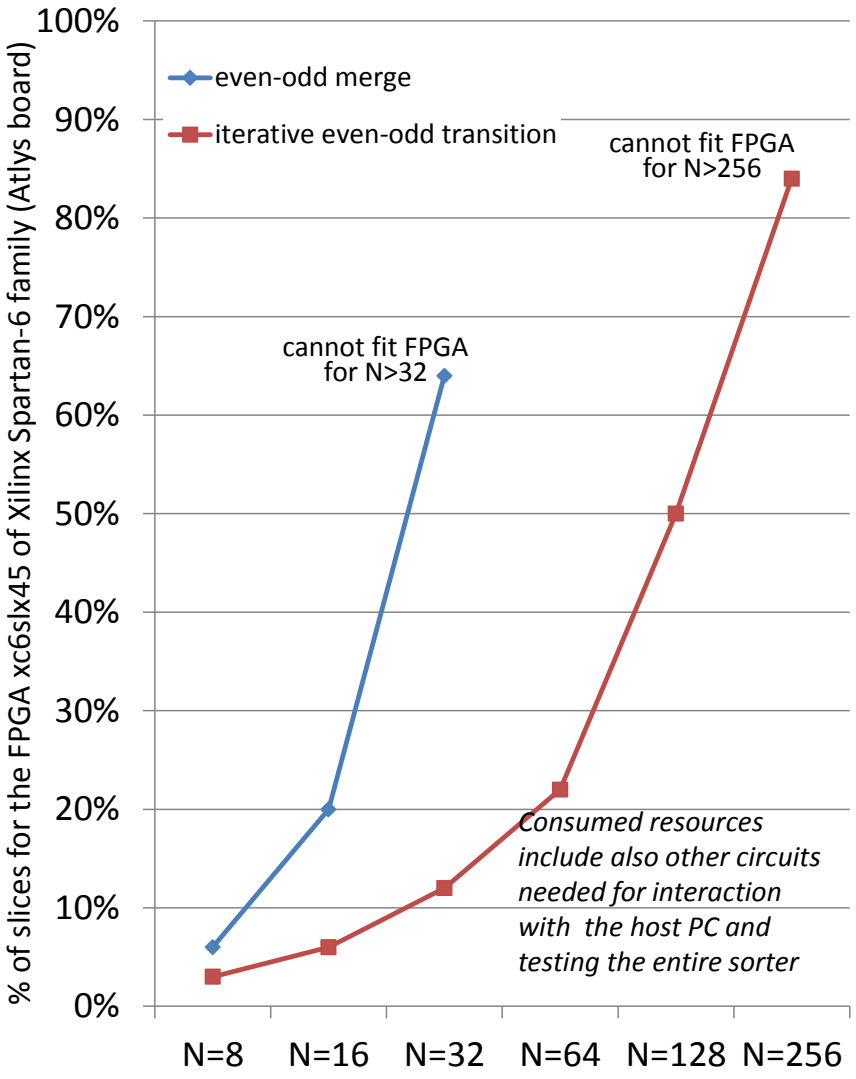


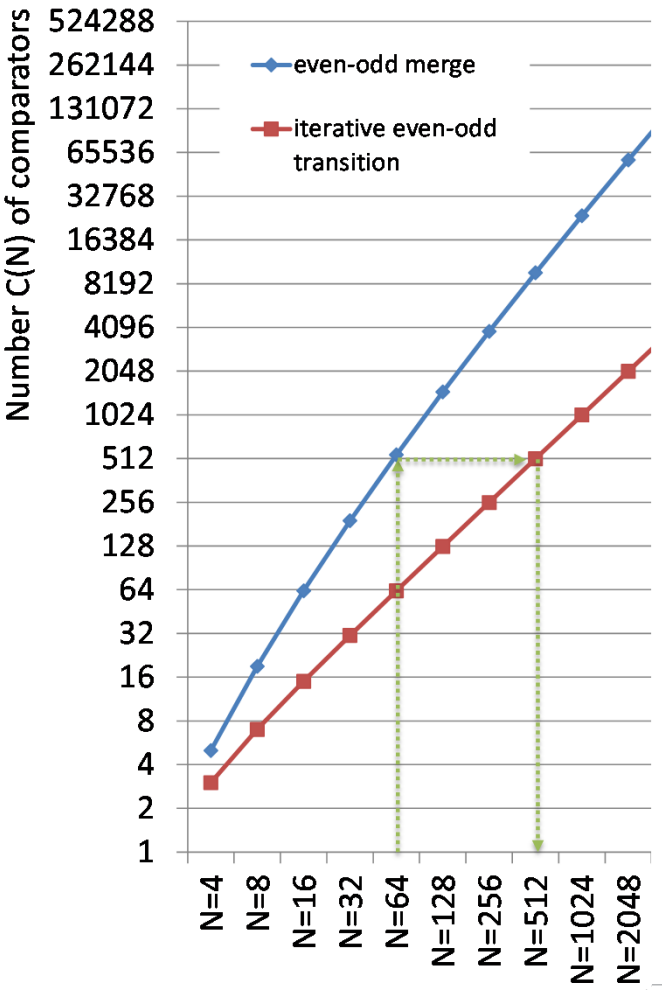
# Hardware resources

- Xilinx ISE 14.4, Atlys prototyping board, XC6SLX45 FPGA of Xilinx Spartan-6 family
- N=8, M=32
- Source data were taken from a host PC through USB and the results were sent back to the PC
- Resources include also circuits for interactions with the host computer

| | $S_{FPGA}$ | $F_{max}$ (MHz) | clock period (ns) |
|---|---|---|---|
| **Even-odd merge** | 474 (6%) | 21 | 47 |
| **Bitonic merge** | 584 (8%) | 21 | 46 |
| **proposed iterative even-odd** | 279 (4%) | 122 | 8 |

# Hardware resources



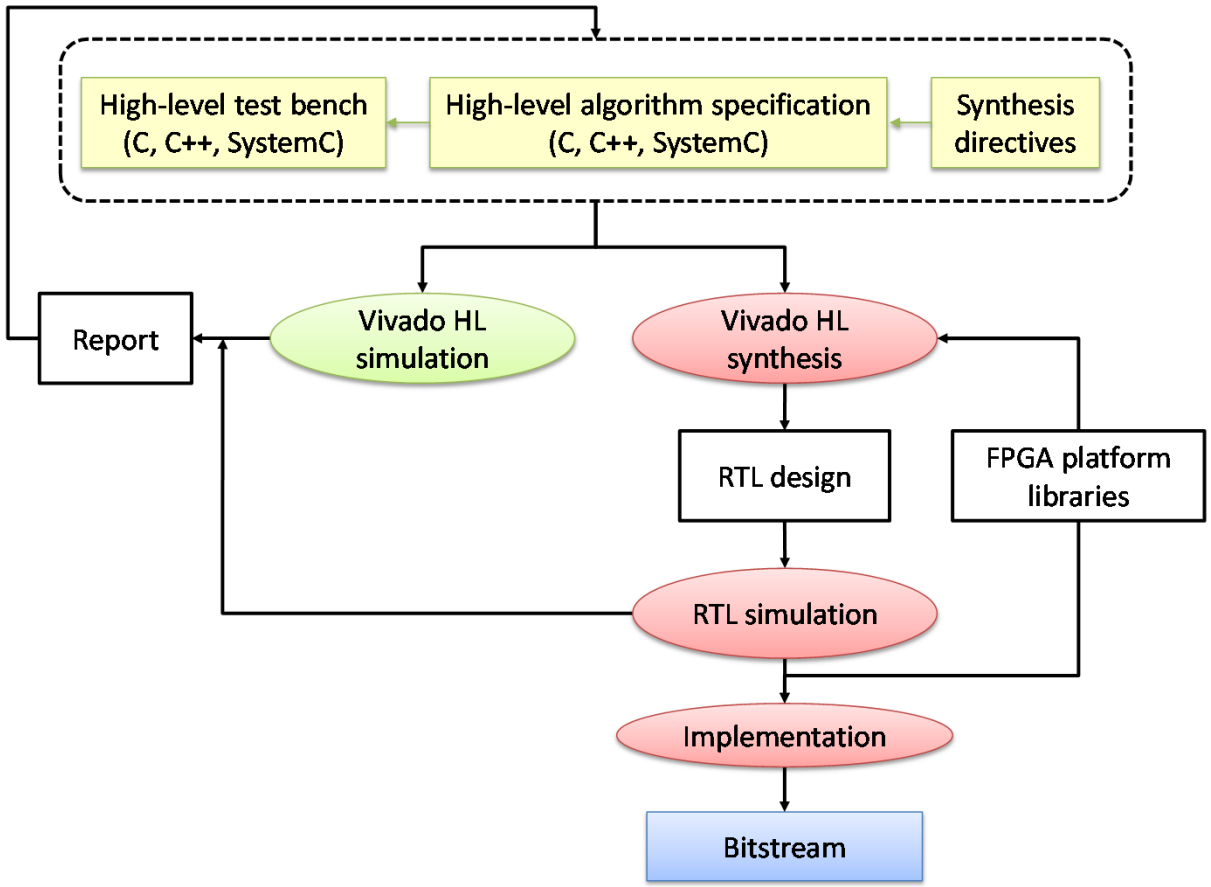
# High-level synthesis

- High-level synthesis (HLS) is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior
- Commercial tools allowing digital circuits to be synthesized from high-level specification languages, such as HardwareC, Handel-C and SystemC, appeared on the market in the mid-1990s - early-2000s
- Early versions of HLS tools leaded to performance degradation and not very efficient resource usage
- The recent HLS tools focus on using as many standard C/C++ constructs as possible to capture the design intent
- HLS languages have many advantages such as ease to learn, ease to change and maintenance, and a short development time
- HLS languages may become the predominant hardware specification methodology

# Why HLS?

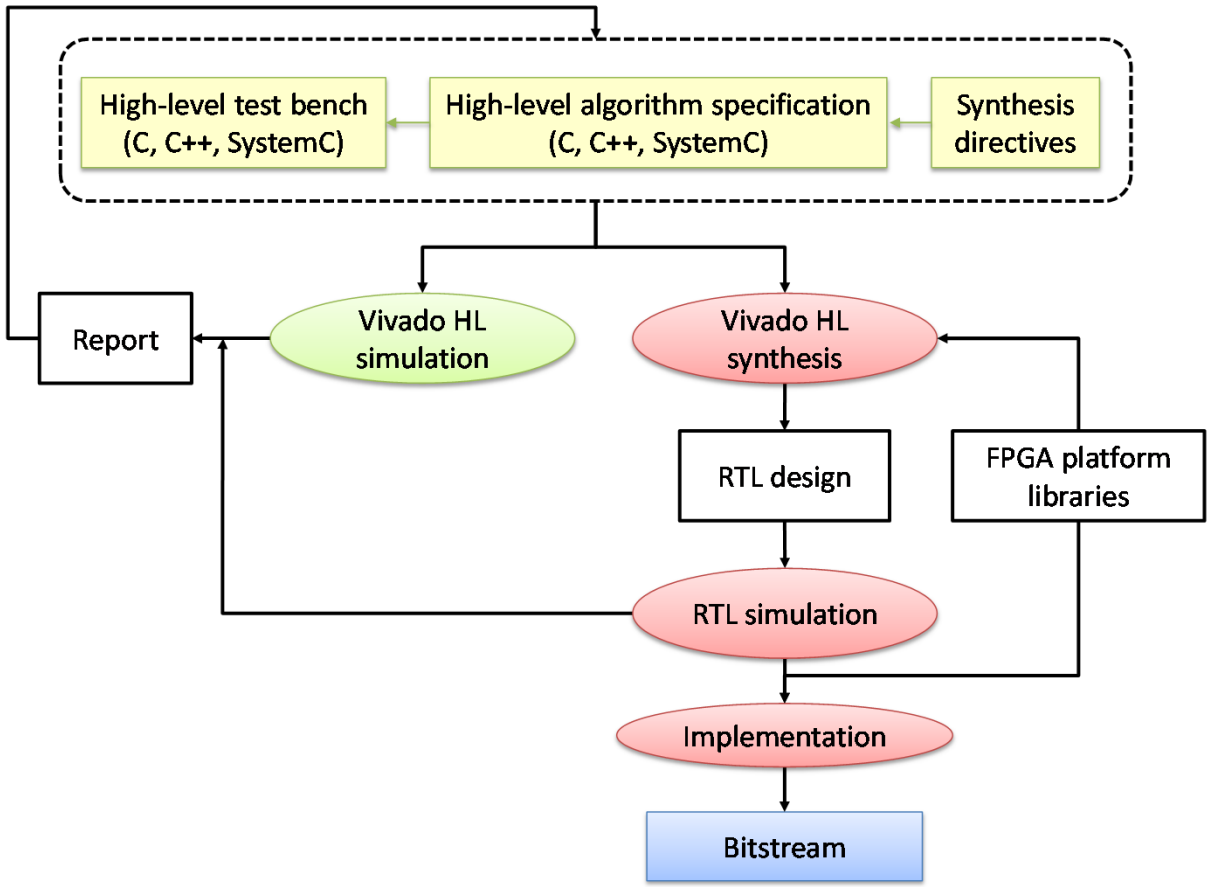
- The growing design complexity pushes to raise the level of abstraction beyond RTL. HLS specifications allow code density to be reduced significantly compared to HDLs, resulting in increased design productivity.
- The design functionality may be specified for both embedded software and reconfigurable hardware. Therefore different software/hardware partitioning boundaries may easily be experimented and the respective area/performance trade-offs straightforwardly explored.
- HLS permits to leverage the experience of engineers comfortable with C/C++ languages for the purposes of digital design.
- The available system-level verification methods permit the RTL output to be simulated with the same test bench that was used to check the high-level code. Moreover, the verification and debug may be performed at earlier stages. This significantly reduces the verification effort because the simulation at higher levels runs much faster than at RTL level and the design errors are much easier to locate and fix at higher abstraction levels.
- As a result, a growing number of designs are produced using HLS tools in various application domains.

# HLS design flow – entry and simulation




- The top-level of every C/C++ program is the main() function and any function below the level of main() can be synthesized
- The function to be synthesized is called the *top-level function*
- The main() function is used for verification and is therefore referred to as *test bench*
- The test bench permits the behavior of the top-level function to be validated at both high and register transfer levels

# HLS design flow – synthesis

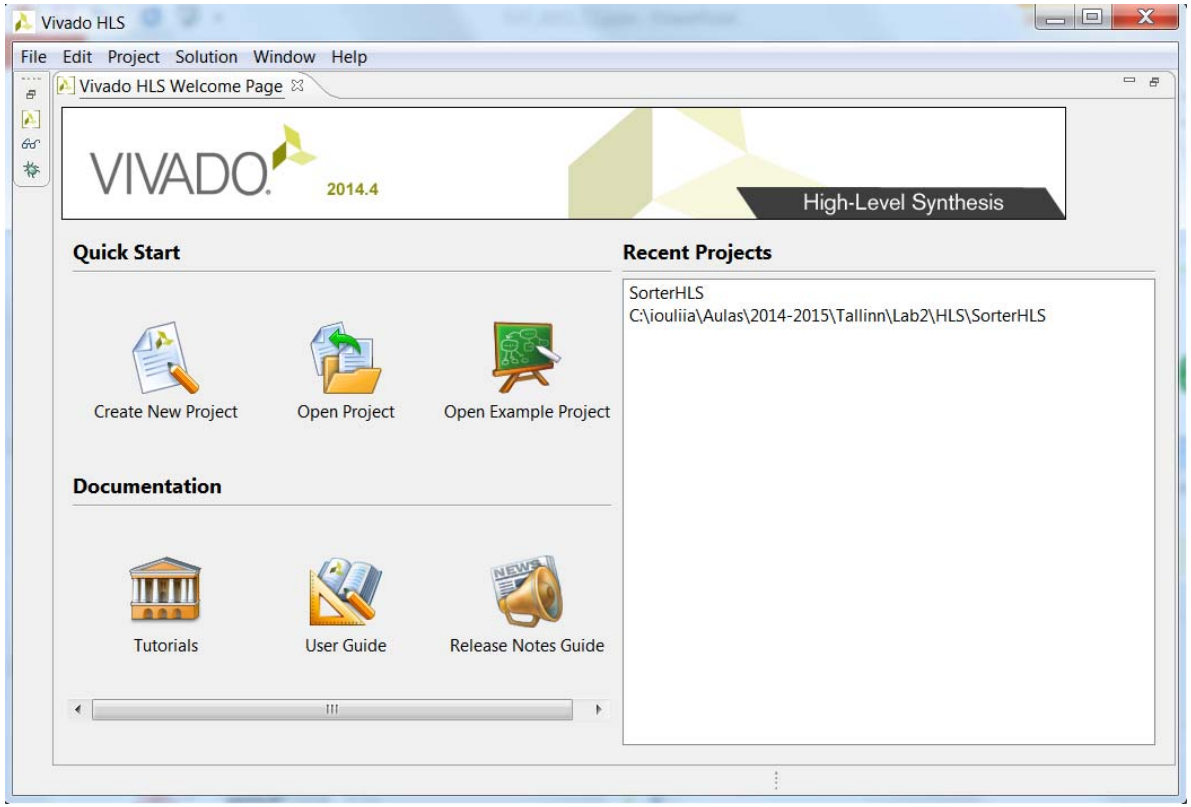



- HLS performs two types of synthesis on the design:
  - algorithmic synthesis (converts functional statements into RTL statements)
  - interface synthesis (transforms function arguments and return values into RTL ports)
- Synthesis directives permit the designer to improve the results of synthesis by driving the optimization engine towards the desired performance goals and RTL architecture
- The generated RTL might be verified using the original high-level test bench
- The final RTL output can also be packaged as IP

# Vivado HLS

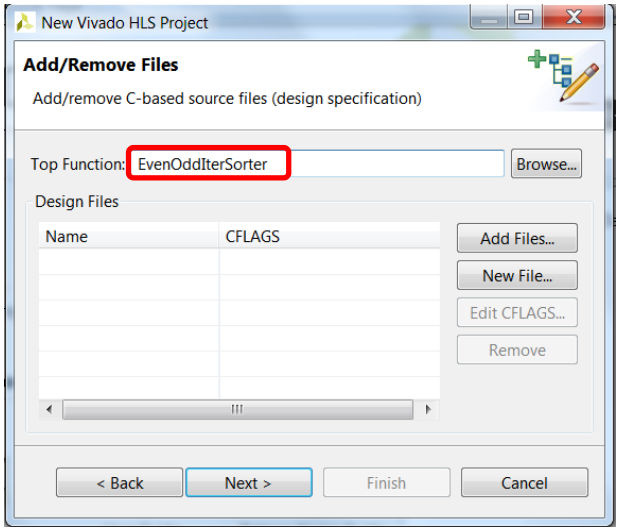
- Create a new synthesis project
- Validate the function of the C design
- Synthesize the C design to an RTL implementation
- Validate the RTL design
- Apply synthesis directives
- Incorporate the synthesized RTL into Vivado projects

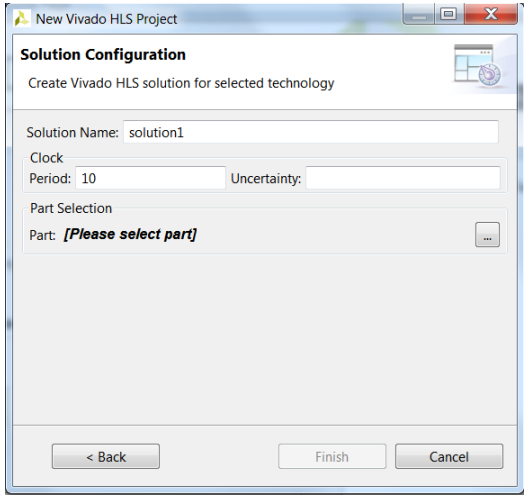
# Welcome page

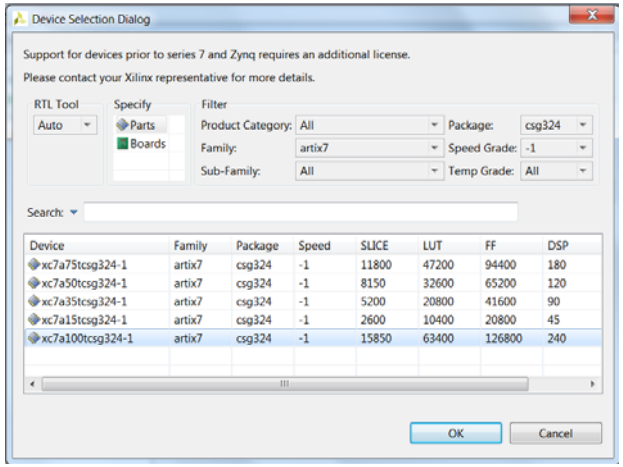


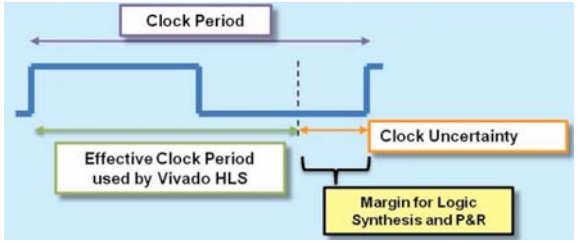

- Quick Start
  - create new projects
  - access to previous and example projects
- Documentation
  - tutorials
  - user guides

# Creating a new project



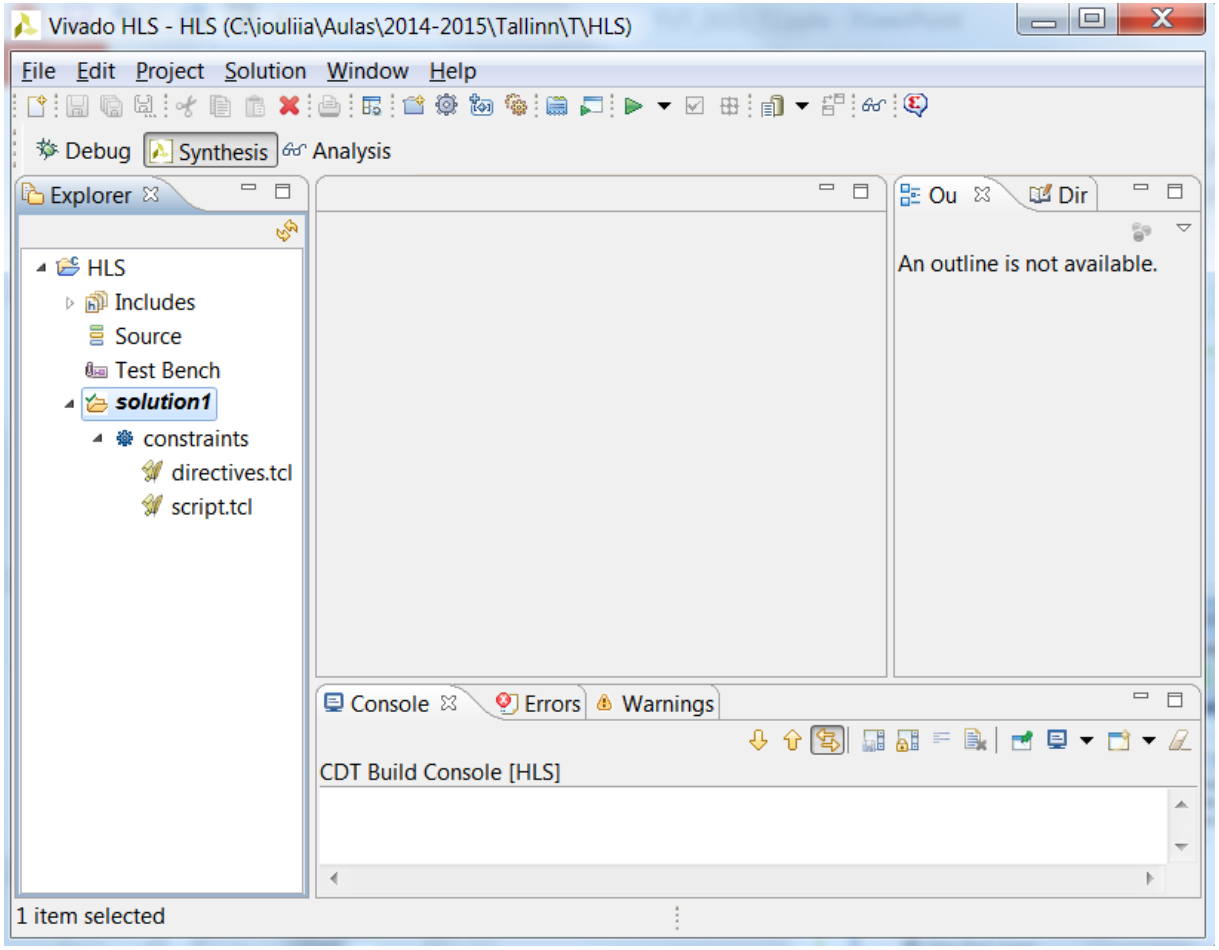




Wizard guides through a creation of a new project

- define project name and location
- specify the top-level function
- add source and test bench files
- add clock constraints
- select target device (**xc7a100tcsg324-1**)

Picture from http://www.xilinx.com/training/vivado/getting-started-with-vivado-high-level-synthesis.htm

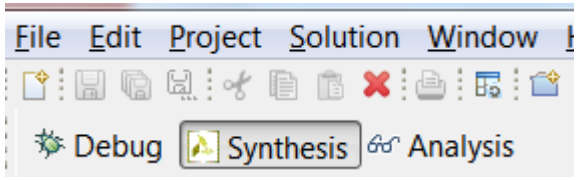
# Vivado HLS GUI



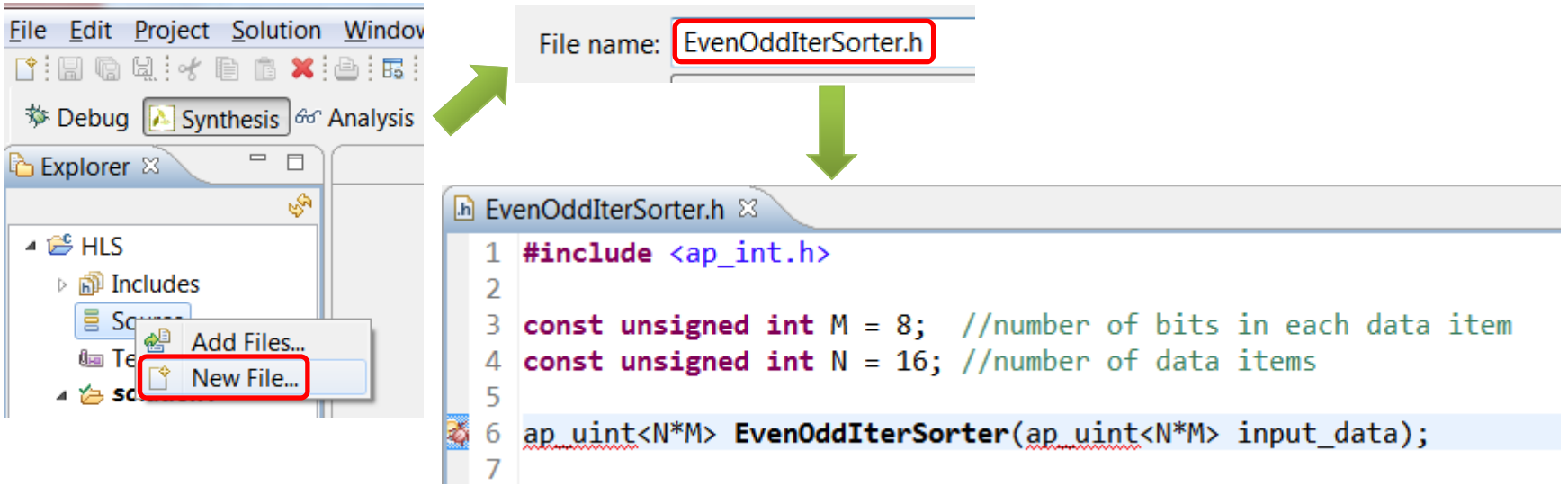

- Explorer pane
  - shows the project hierarchy where project files are organized in categories
    - source files
    - test benches
    - solutions
- Information pane
  - shows the contents of files opened from the *Explorer pane* and the results of synthesis
- Auxiliary pane
  - includes content-sensitive information depending on the file currently active in the *Information pane*
- Console pane
  - shows messages produced by Vivado HLS organized in different categories (*Console*, *Errors* and *Warnings*)

# Project perspectives

- The project might be seen and analyzed in three perspectives:
  - synthesis perspective allows to specify and synthesize designs, run simulations and package the IP
  - debug perspective permits the design to be debugged by step running through the code and analyzing the results of operations
  - analysis perspective is only active after synthesis completes and helps to analyze the synthesis report results, in particular the resource usage and the performance



# Adding design source files




# Language support

- Vivado HLS supports the following standards for C compilation/simulation:
  - ANSI-C (GCC 4.6)
  - C++ (G++ 4.6)
  - SystemC (IEEE 1666-2006 -Version 2.2-)
- Synthesis support is provided for a wide range of C, C++ and SystemC language constructs and all native data types for each language, including float and double types
- There are however some constructs which cannot be synthesized:
  - dynamic memory allocation
  - all data to and from the FPGA must be read from the input ports or written to output ports; as such, OS operations such as files accesses and OS queries cannot be supported
  - the C constructs must be of a fixed or bounded size
  - recursive functions
  - Standard Template Library functions

# Arbitrary precision data types

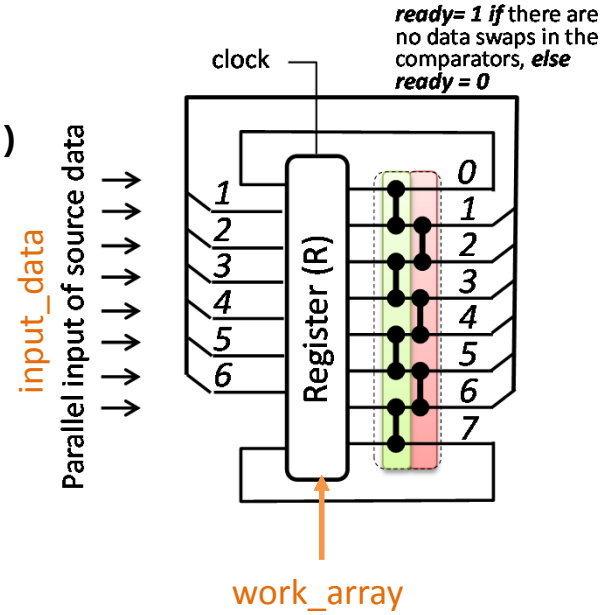
- C/C++ built-in data types are supported by Vivado HLS, but these data types are on 8-bit boundaries (8, 16, 32, 64 bits)
- Since we need to process M-bit unsigned integers, we have to indicate this requirement explicitly so that M-bit components will be synthesized
- ap_int.h defines C++ arbitrary precision types ap_uint<W> (for unsigned integers) and ap_int<W> (for signed integers); for C code include ap_cin.h
- the maximum width allowed for ap_uint<W> type is 1,024 bits
- this default may be overridden by defining the macro AP_INT_MAX_W with a positive integer value ≤ 32,768 before inclusion of the ap_int.h header file: `#define AP_UINT_MAX_W 16384`

EvenOddIterSorter.h

```
1  #include <ap_int.h>
2
3  const unsigned int M = 8;  //number of bits in each data item
4  const unsigned int N = 16; //number of data items
5
6  ap_uint<N*M> EvenOddIterSorter(ap_uint<N*M> input_data);
7
```

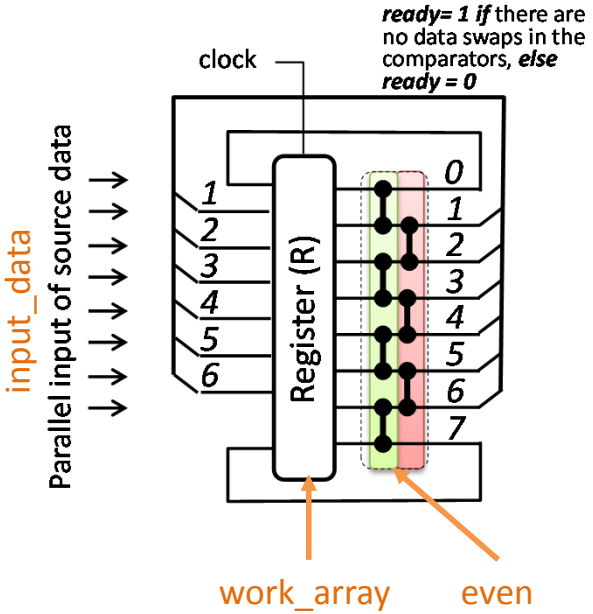
# C++ specification – prepare the data

```
#include "EvenOddIterSorter.h"

ap_uint<N*M> EvenOddIterSorter (ap_uint<N*M> input_data)
{
   ap_uint<N*M> sorted_data;
   bool sorting_completed;

   ap_uint<M> work_array[N];
   ap_uint<M> even[N];

   //1. Fill in the work array
   ap_uint<M> mask = ~0;
   init_loop: for (unsigned i = N; i > 0; i--)
   {
         work_array[i-1] = input_data & mask; // extract M LSBs
         input_data >>= M;                    // shift right M bits
   }

   // continues
```



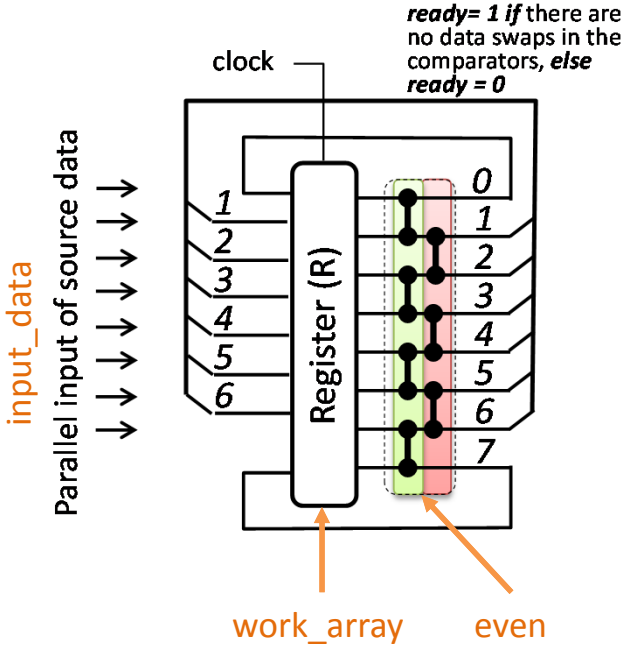
# C++ specification - sorting

```
//2. Sort the data
sorting_completed = false;
sort_loop: while (!sorting_completed)
{  // even line of comparators
   sorting_completed = true;
   sort_even: for (unsigned j = 0; j < (N / 2); j++)
         if (work_array[2 * j] > work_array[2 * j + 1])
         {  sorting_completed = false;
            even[2 * j]     = work_array[2 * j + 1];
            even[2 * j + 1] = work_array[2 * j];
         }
         else
         {  even[2 * j]     = work_array[2 * j];
            even[2 * j + 1] = work_array[2 * j + 1];
         }
   // odd line of comparators
   sort_odd: for (unsigned j = 0; j < (N / 2 - 1); j++)
         if (even[2 * j + 1] > even[2 * j + 2])
         {  sorting_completed = false;
            work_array[2 * j + 1] = even[2 * j + 2];
            work_array[2 * j + 2] = even[2 * j + 1];
         }
         else
         {  work_array[2 * j + 1] = even[2 * j + 1];
            work_array[2 * j + 2] = even[2 * j + 2];
         }
   work_array[0] = even[0];
   work_array[N-1] = even[N-1];
}
// continues
```



# C++ specification – write the result

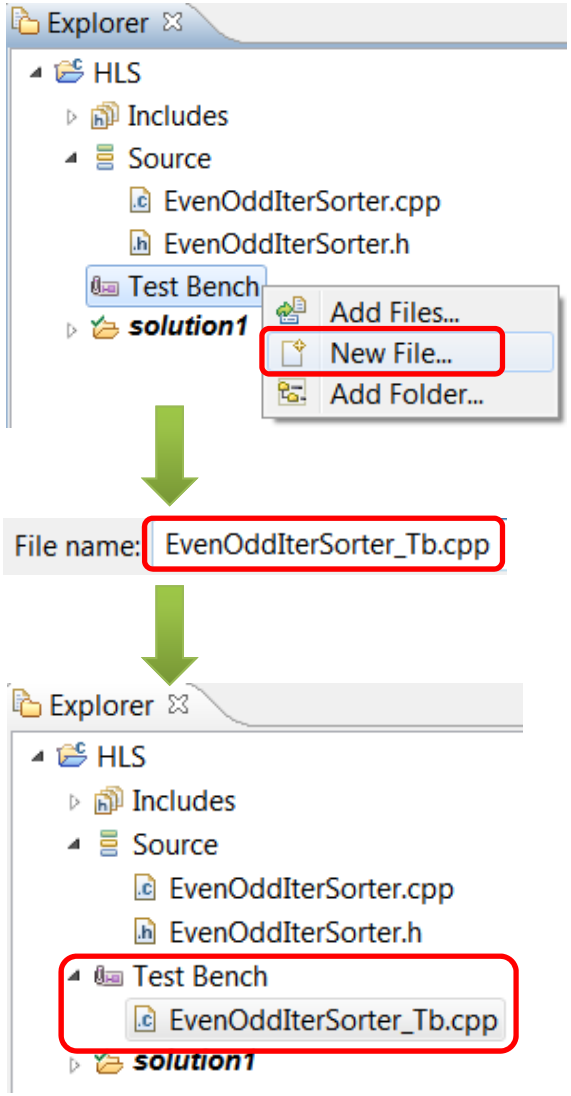
```
    //3. Write the result
    write_res_loop: for (unsigned i = 0; i < N; i++)
    {
          sorted_data <<= M;              // shift left M bits
          sorted_data |= work_array[i]; // write M LSBs
    }

    return sorted_data;
}
```



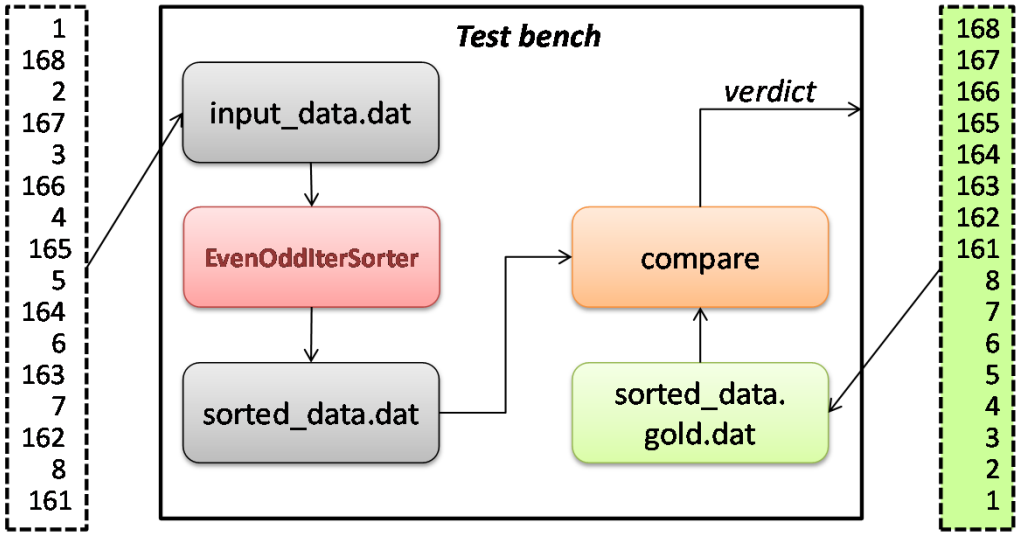
# Creating test bench

- Testing is done with the *main()* function
- The test bench has to respect the following rules:
  - the test bench has to call the function to be synthesized (EvenOddIterSorter)
  - the test bench has to save the output from the function and to compare the results with the known expected results
  - if the produced results do match the expected results a message is issued confirming that the function has passed the test and the return value of the test bench main() function is set to 0
  - otherwise, if the output is different from the expected results, a message indicates this, and the return value of main() is set to any value different from 0
  - the test bench must generate the same input stimuli every time it is executed to be used successfully for RTL verification (*i.e.* test values cannot be generated randomly)

# Adding the test bench

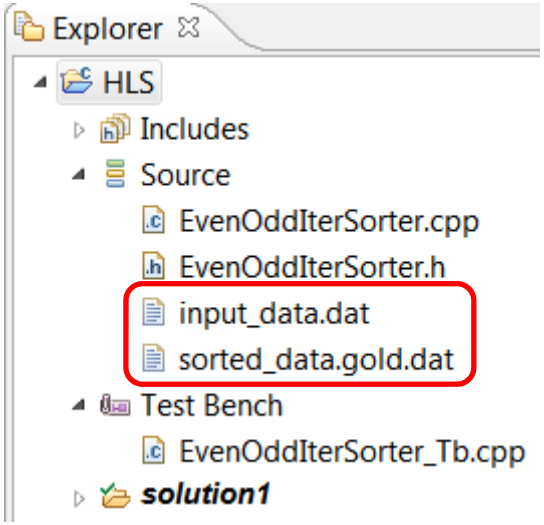



- read data to sort from a text file input_data.dat
- call the function EvenOddIterSorter
- store the result in the file sorted_data.dat
- compare the results produced with the golden results kept in the file sorted_data.gold.dat and issue the verdict



# Coding the test bench in C++

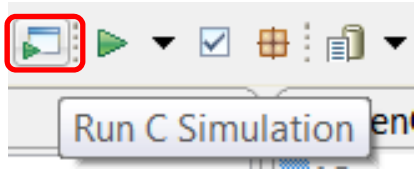
```
#include <iostream>
#include <fstream>
#include "EvenOddIterSorter.h"

//error codes
const int SORT_ERROR = 200;
const int OK = 0;
using namespace std;

int main ()
{
   ap_uint<N*M> input_data, sorted_data;
   ap_uint<M> item;

   ifstream unsorted_data_stream("input_data.dat"); // open for reading
   ofstream sorted_data_stream("sorted_data.dat");  // open for writing

   //get the input data
   cout << "Reading input data" << endl;
   for (unsigned i = 0; i < N; i++)
   {
         unsorted_data_stream >> item;
         cout << item << endl;
         input_data <<= M;    // shift left M bits
         input_data |= item; // write M LSBs
   }
   // continues
```

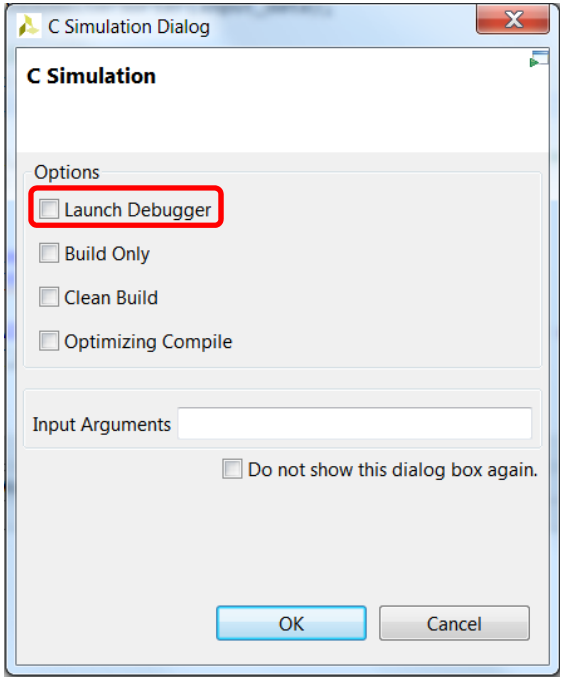


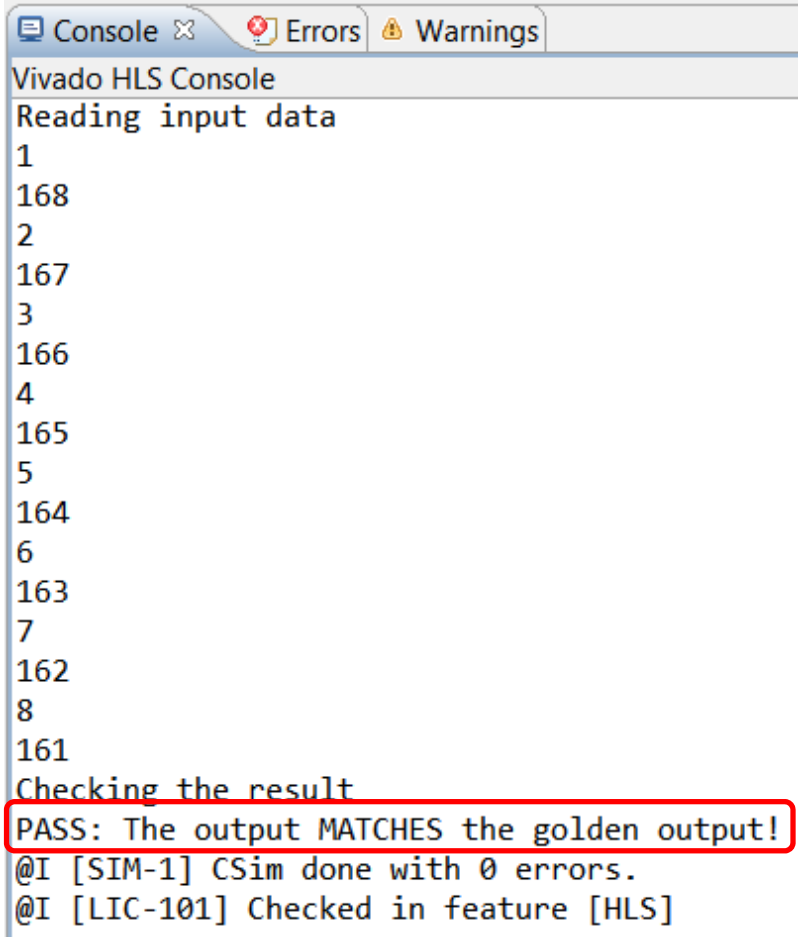
# Coding the test bench in C++

```
    //perform sorting
    sorted_data = EvenOddIterSorter(input_data);

    //save the result
    ap_uint<M> mask = ~0;
    for (unsigned i = 0; i < N; i++)
    {
          sorted_data_stream << (sorted_data & mask) << endl; // extract M LSBs
          sorted_data >>= M; // shift right M bits
    }

    cout << "Checking the result" << endl;
    if (system("diff -w sorted_data.dat sorted_data.gold.dat"))
    {
          cout << "FAIL: Output DOES NOT match the golden output." << endl;
          return SORT_ERROR;
    }
    else
    {
          cout << "PASS: The output MATCHES the golden output!" << endl;
          return OK;
    }
}
```

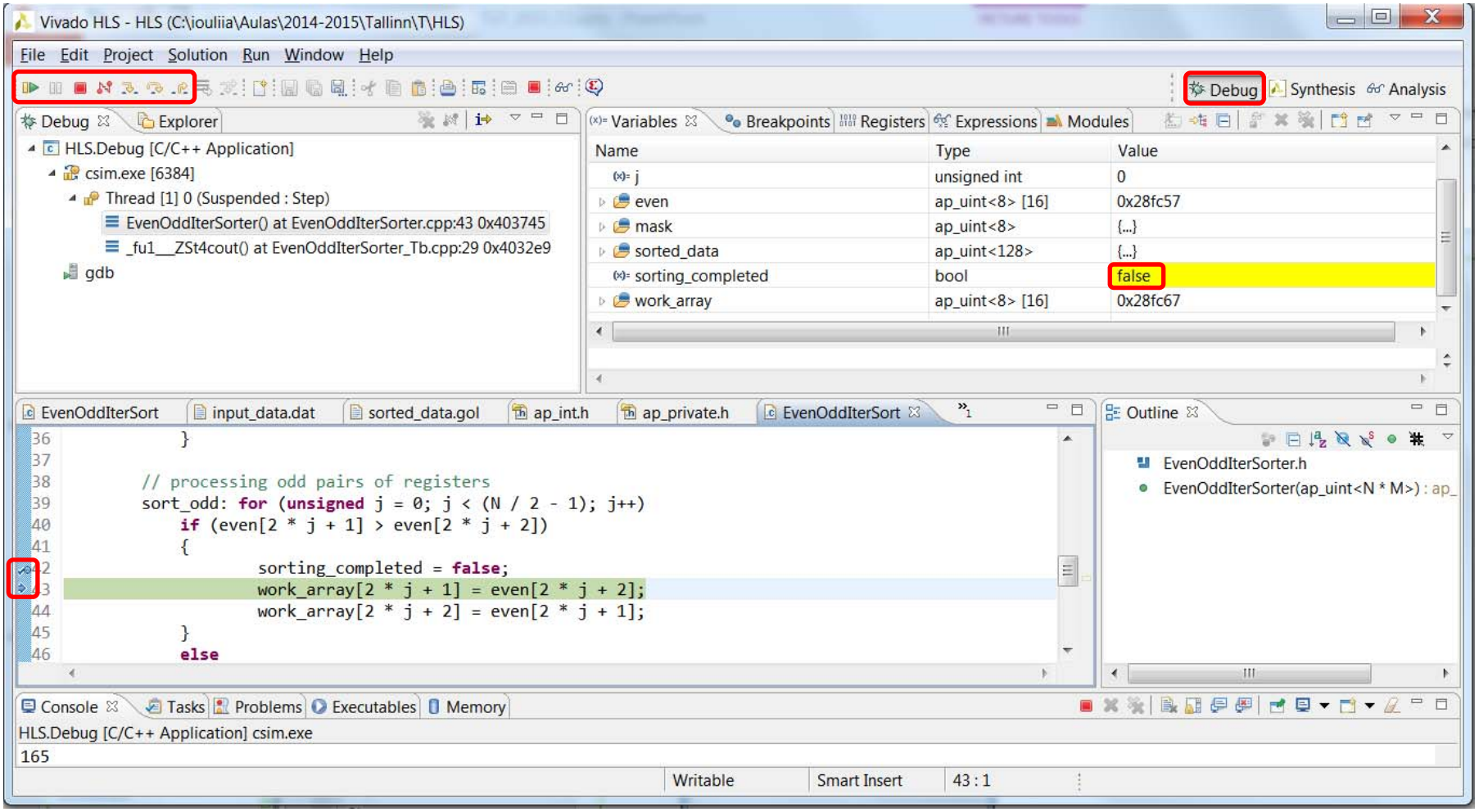
# C simulation



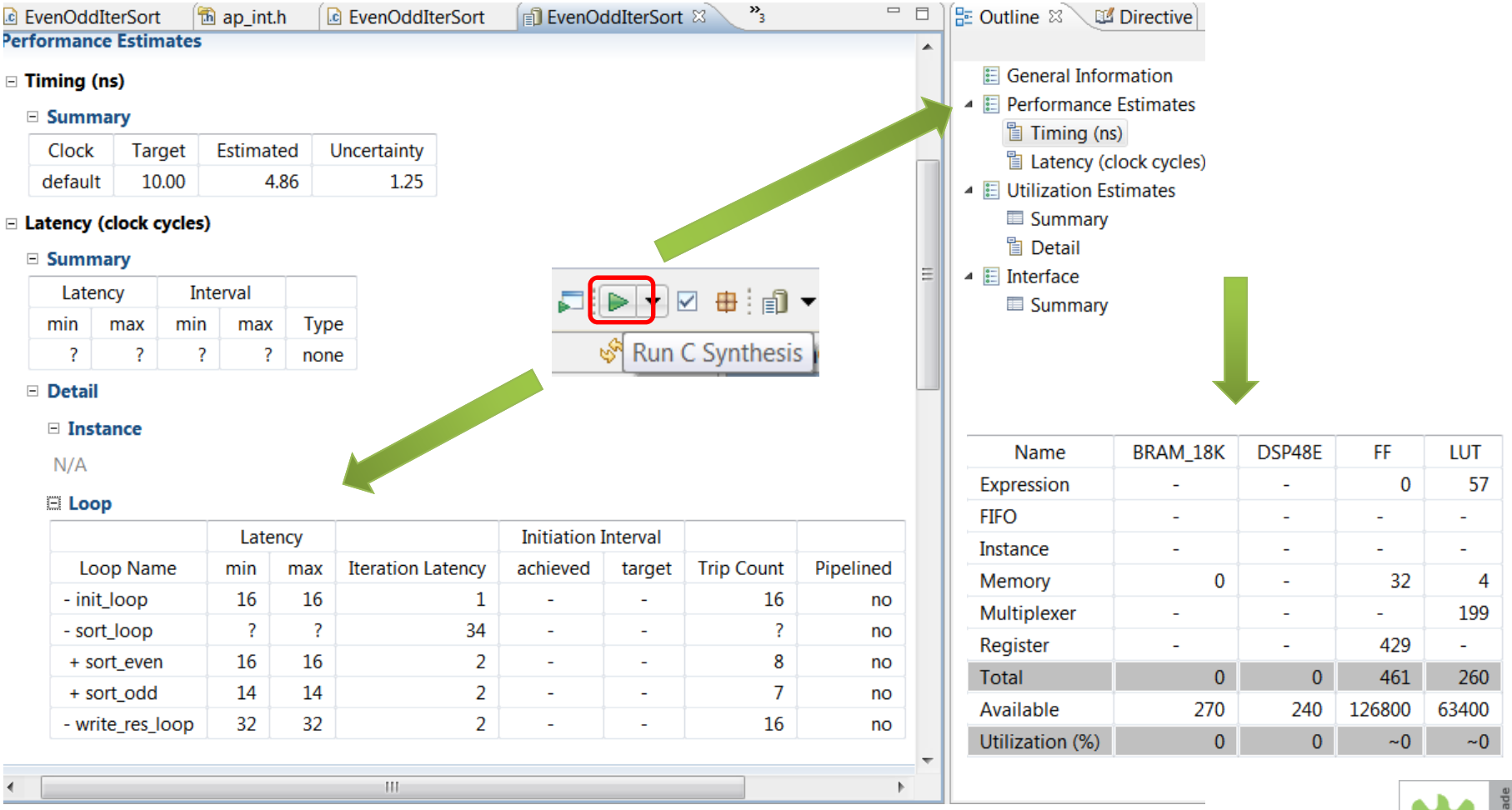



# Debugging




- Design errors are much easier to locate and fix at higher abstraction levels
- Arbitrary precision types cannot be debugged for ANSI C designs (C++ OK)

# C synthesis


EvenOddIterSort
ap_int.h
EvenOddIterSort
EvenOddIterSort
Performance Estimates
Timing (ns)
Summary
Clock | Target | Estimated | Uncertainty
default | 10.00 | 4.86 | 1.25
Latency (clock cycles)
Summary
Latency | Interval
min | max | min | max | Type
? | ? | ? | ? | none
Detail
Instance
N/A
Loop
Latency | Initiation Interval
Loop Name | min | max | Iteration Latency | achieved | target | Trip Count | Pipelined
- init_loop | 16 | 16 | 1 | - | - | 16 | no
- sort_loop | ? | ? | 34 | - | - | ? | no
+ sort_even | 16 | 16 | 2 | - | - | 8 | no
+ sort_odd | 14 | 14 | 2 | - | - | 7 | no
- write_res_loop | 32 | 32 | 2 | - | - | 16 | no
Run C Synthesis
Outline
Directive
General Information
Performance Estimates
Timing (ns)
Latency (clock cycles)
Utilization Estimates
Summary
Detail
Interface
Summary
Name | BRAM_18K | DSP48E | FF | LUT
Expression | - | - | 0 | 57
FIFO | - | - | - | -
Instance | - | - | - | -
Memory | 0 | - | 32 | 4
Multiplexer | - | - | - | 199
Register | - | - | 429 | -
Total | 0 | 0 | 461 | 260
Available | 270 | 240 | 126800 | 63400
Utilization (%) | 0 | 0 | ~0 | ~0

# Interface synthesis

- Vivado synthesizes
  - input pass-by-value arguments and pointers as simple wire ports with no associated handshaking signal (ap_none protocol)
  - output values and pointers with an associated output valid signal to indicate when the output is valid (ap_ctrl_hs protocol)
  - other types of interfaces can be synthesized such as two-way handshakes, RAM access ports and FIFO ports
- The default function-level handshaking I/O protocol is ap_ctrl_hs
  - input signal ap_start controls the block execution and must be asserted for the design to start its operation
  - output signal ap_done indicates that the function has finished and, if there is a return value, this value is valid and may be read
  - output signal ap_idle, when asserted, indicates that the function is not operating (idle)
  - output signal ap_ready indicates when the design is ready to accept new inputs (this signal is not asserted if ap_start is low and the design completed all the operations)

ap_ctrl_hs

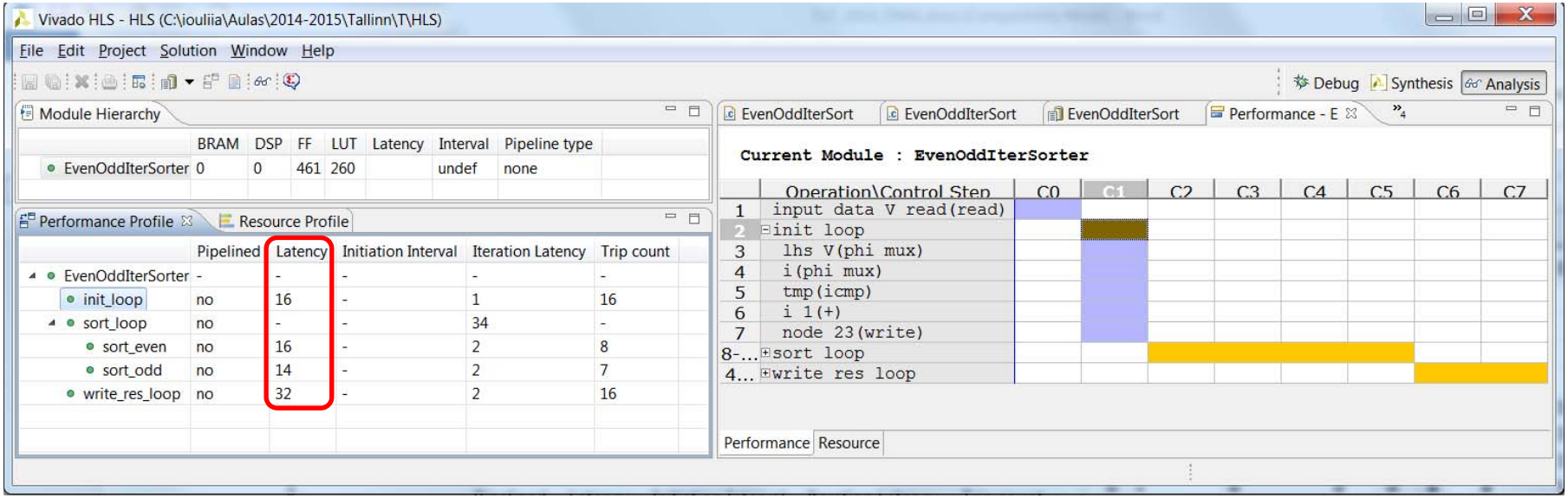
ap_none

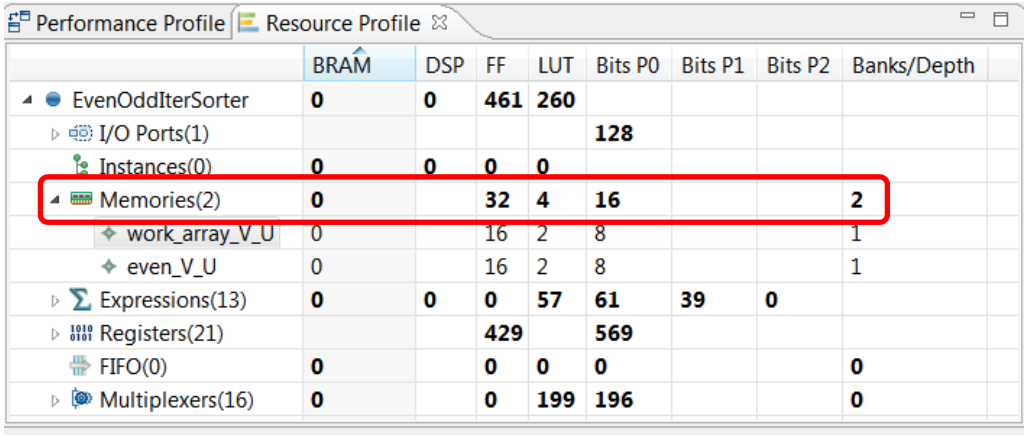
**Interface**

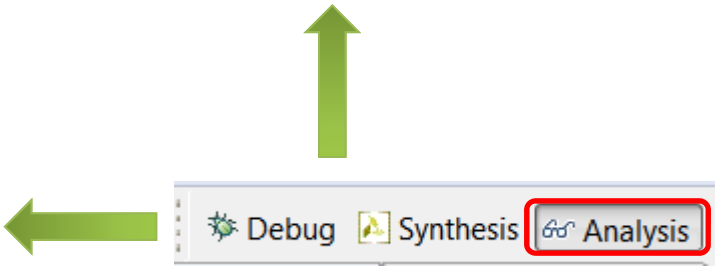
**Summary**

| RTL Ports | Dir | Bits | Protocol | Source Object | C Type |
|---|---|---|---|---|---|
| ap_clk | in | 1 | ap_ctrl_hs | EvenOddIterSorter | return value |
| ap_rst | in | 1 | ap_ctrl_hs | EvenOddIterSorter | return value |
| ap_start | in | 1 | ap_ctrl_hs | EvenOddIterSorter | return value |
| ap_done | out | 1 | ap_ctrl_hs | EvenOddIterSorter | return value |
| ap_idle | out | 1 | ap_ctrl_hs | EvenOddIterSorter | return value |
| ap_ready | out | 1 | ap_ctrl_hs | EvenOddIterSorter | return value |
| ap_return | out | 128 | ap_ctrl_hs | EvenOddIterSorter | return value |
| input_data_V | in | 128 | ap_none | input_data_V | scalar |

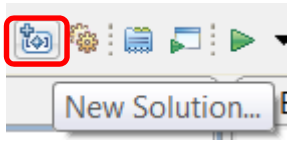
# Analysis perspective

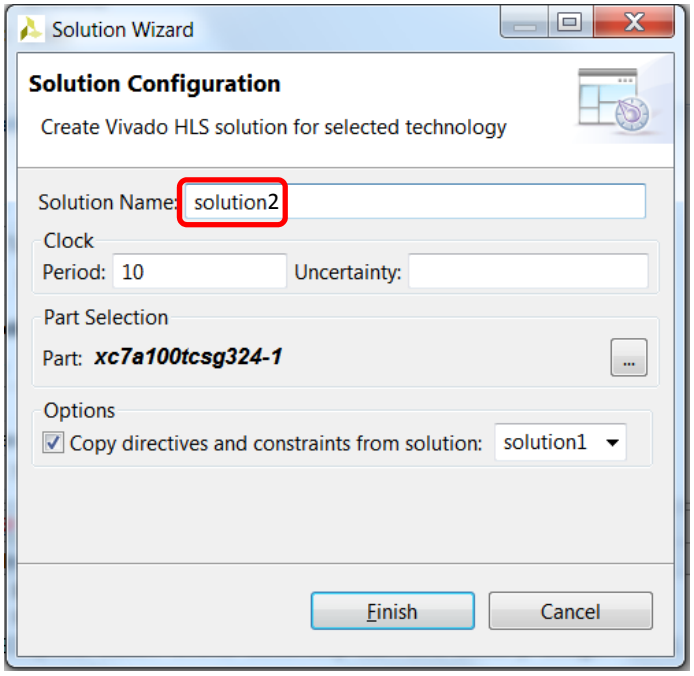


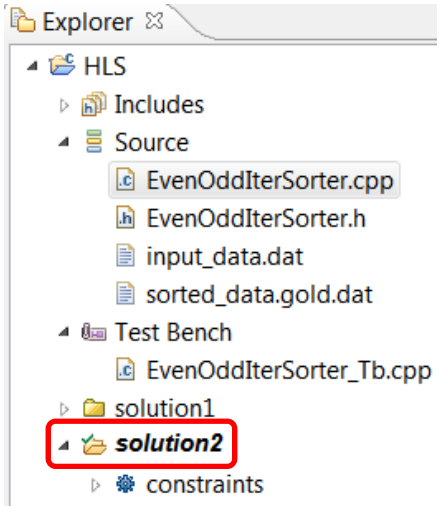



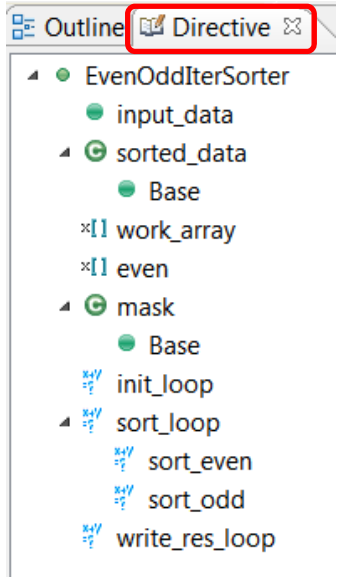



# Optimizing the design



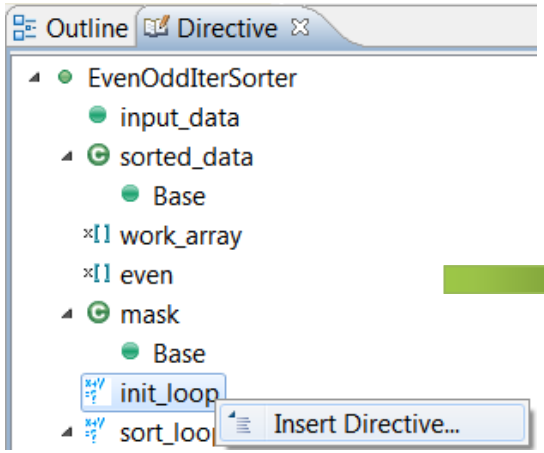



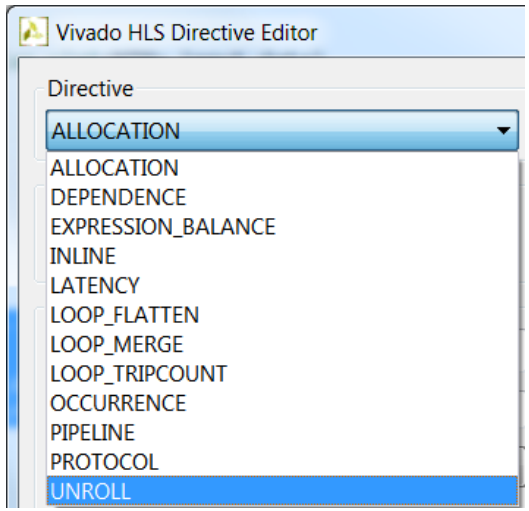
# Synthesis directives – loop unroll

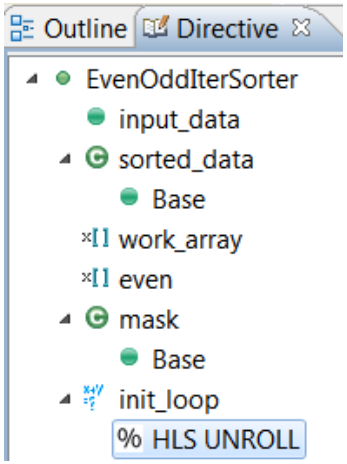
- By default, all loops are kept ***rolled*** in HLS, *i.e.* all operations in the loop are implemented using the same hardware resources and each iteration is performed in a separate clock cycle (while the intermediate results are stored in registers)
- HLS provides the ability to unroll *for* loops, either partially or fully
  - when a loops is *fully unrolled* all the loop operations are performed in a single clock cycle by replicating the required for each iteration hardware resources
  - there is also a possibility to unroll a loop partially with a specific factor

**Loop**

| | Latency | |
|---|---|---|
| Loop Name | min | max |
| - init_loop | 16 | 16 |
| - sort_loop | ? | ? |
| + sort_even | 16 | 16 |
| + sort_odd | 14 | 14 |
| - write_res_loop | 32 | 32 |



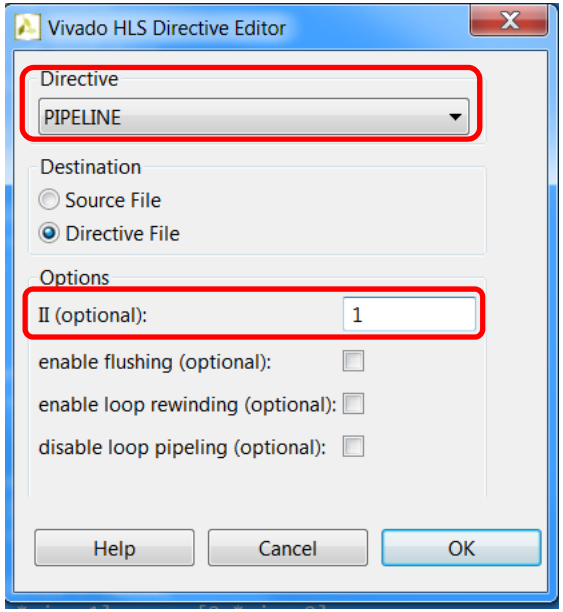



# Synthesis directives - pipelining

- The latency is the number of cycles required to produce the output
- The initiation interval (II) is the throughput, *i.e.* the number of cycles between new input reads

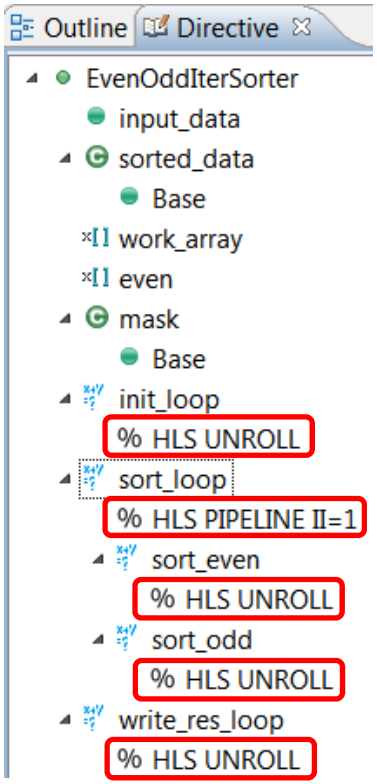
| | Latency | | | Initiation Interval | | | |
|---|---|---|---|---|---|---|---|
| Loop Name | min | max | Iteration Latency | achieved | target | Trip Count | Pipelined |
| - Loop 1 | ? | ? | 2 | - | - | ? | no |




**Utilization Estimates**

**Summary**

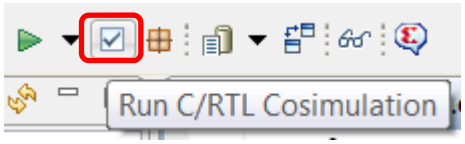
| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 342 |
| FIFO | - | - | - | - |
| Instance | - | - | - | - |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 129 |
| Register | - | - | 259 | - |
| Total | 0 | 0 | 259 | 471 |
| Available | 270 | 240 | 126800 | 63400 |
| Utilization (%) | 0 | 0 | ~0 | ~0 |

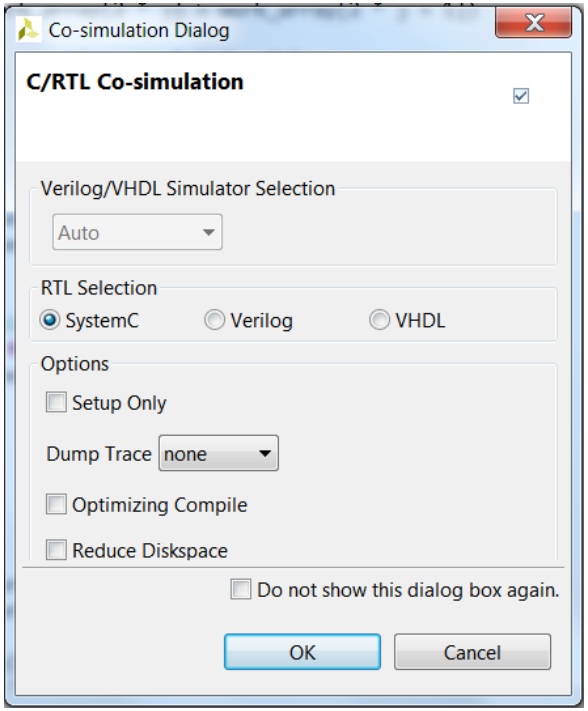



**Loop**

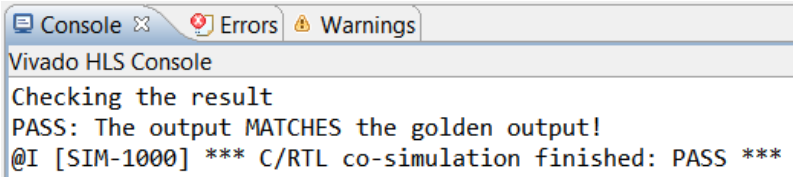
| | Latency | | | Initiation Interval | | | |
|---|---|---|---|---|---|---|---|
| Loop Name | min | max | Iteration Latency | achieved | target | Trip Count | Pipelined |
| - sort_loop | ? | ? | 1 | 1 | 1 | ? | yes |

# C/RTL cosimulation

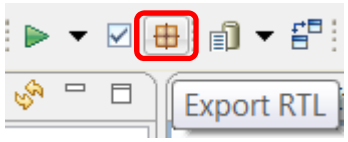


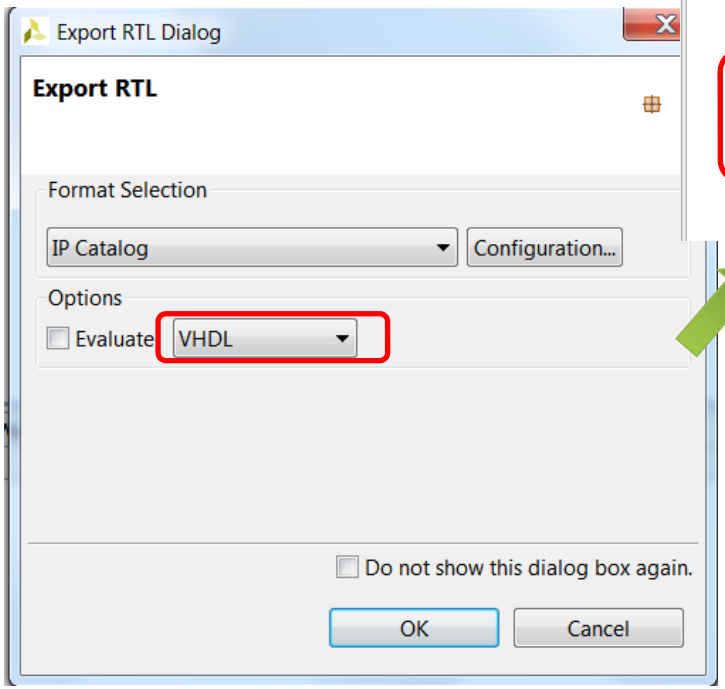



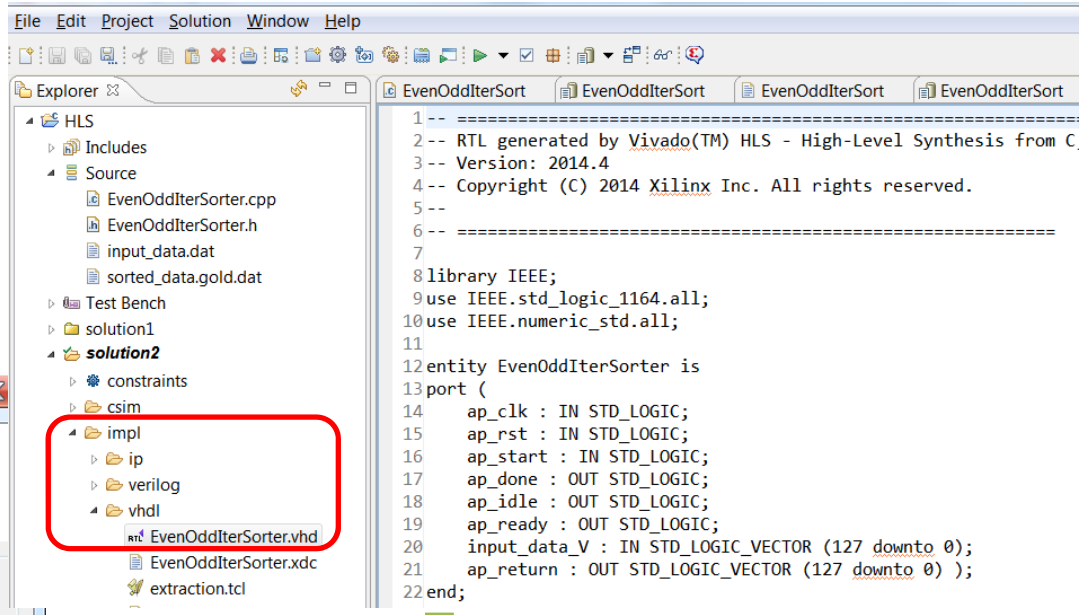

- By default, the cosimulation is performed using the generated SystemC RTL which uses the built-in C compiler
- It is also possible to perform the verification in VHDL and Verilog
- Cosimulation involves the following steps:
  - the C test bench is executed to generate input stimuli for the RTL design
  - an RTL test bench is created containing the input stimuli from the C test bench and the RTL simulation is performed with the selected simulation tool
  - the output from the RTL simulation is fed back to the C test bench to check the results and a message is issued informing whether the design passed the test



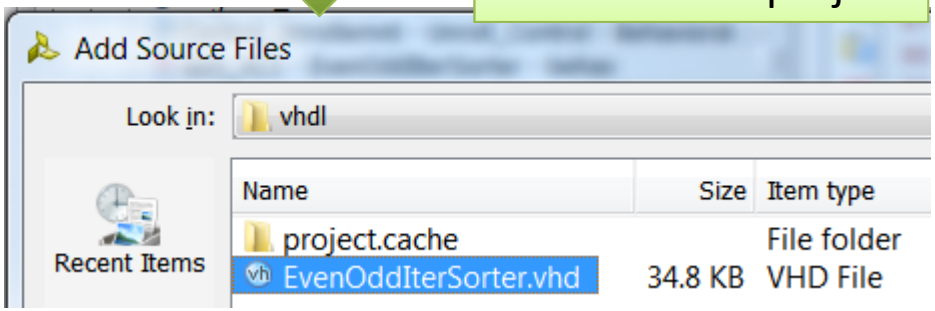
# Exporting RTL



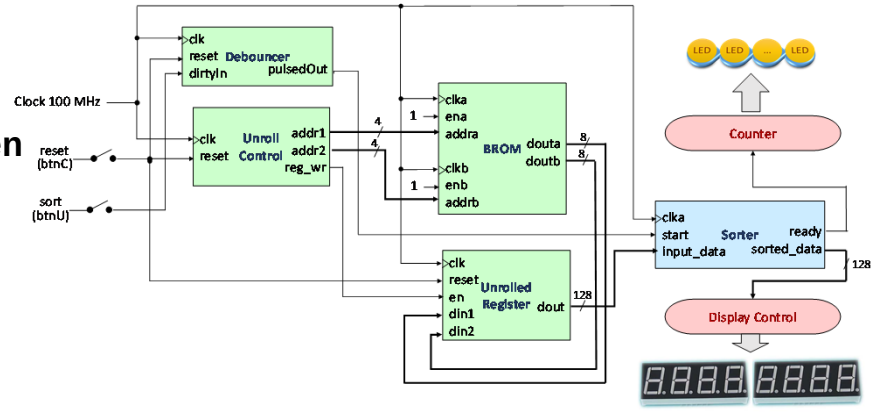




In Vivado RTL project



# Using RTL

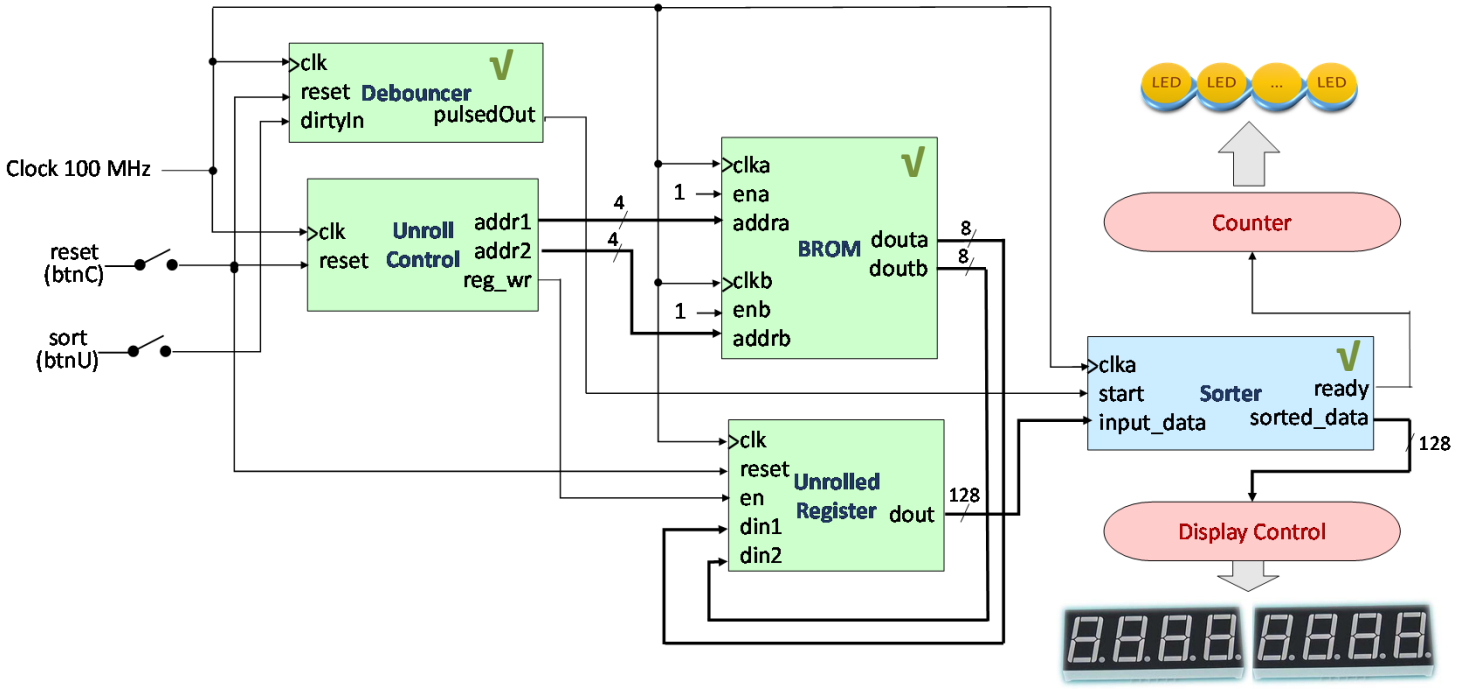
```
control_HLS: process(clk)
begin
    if rising_edge(clk) then
        if s_reset = '1' or s_done = '1' then
            s_start <= '0';
        elsif s_sort_debounced = '1' then
            s_start <= '1';
        end if;
    end if;
end process;
```




```
sort_HLS: entity work.EvenOddIterSorter(behav)
    port map (ap_clk        => clk,              -- board's clock
              ap_rst        => s_reset,          -- btnC
              ap_start      => s_start,          -- to start sorting
              ap_done       => s_done,           -- sorting finished
              ap_idle       => open,
              ap_ready      => open,
              input_data_V  => s_unsorted_data,
              ap_return     => s_sorted_data);

count_cycles: entity work.CountUpN(Behavioral)   -- binary counter
    generic map (N => 16)
    port map (reset     => s_sort_debounced,
              clk       => clk,
              clkEnable => s_start,
              count     => led);
```

# Useful modules for Lab 2

- Unroll control
- Unrolled register



# Unroll control

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity Unroll_Control is
    port (clk           : in std_logic;
          reset         : in std_logic;
          addr1, addr2  : out std_logic_vector(3 downto 0);
          reg_wr        : out std_logic);
end Unroll_Control;

architecture Behavioral of Unroll_Control is
   type TState is (INIT, READ, WRITE, FINISH);
   signal s_currentState, s_nextState : TState;
   signal s_addr : unsigned (2 downto 0);
   signal s_reset, s_inc : std_logic;
begin

sync_proc : process(clk)
begin
   if (rising_edge(clk)) then
      if (reset = '1') then  s_currentState <= INIT;
      else                   s_currentState <= s_nextState;
      end if;
   end if;
end process;

    addr1 <= std_logic_vector(s_addr) & '0';
    addr2 <= std_logic_vector(s_addr) & '1';

inc_address: process(clk)
begin
    if rising_edge(clk) then
        if s_reset = '1'  then s_addr <= (others => '0');
        elsif s_inc = '1' then s_addr <= s_addr + 1;
        end if;
    end if;
end process;
```

```
comb_proc : process(s_currentState, s_addr)
begin
   s_nextState <= s_currentState;

   case (s_currentState) is
      when INIT =>
             reg_wr  <= '0';
             s_reset <= '1';
             s_inc <= '0';
             s_nextState <= READ;
      when READ =>
             reg_wr  <= '0';
             s_nextState <= WRITE;
             s_reset <= '0';
             s_inc <= '0';
      when WRITE =>
             reg_wr  <= '1';
             s_reset <= '0';
             s_inc <= '1';
             if s_addr /= "111" then
                s_nextState <= READ;
             else
                s_nextState <= FINISH;
             end if;
      when FINISH =>
             reg_wr <= '0';
             s_reset <= '0';
              s_inc <= '0';
      when others =>
             reg_wr  <= '0';
             s_nextState <= INIT;
             s_reset <= '0';
             s_inc <= '0';
   end case;
end process;

end Behavioral;
```

# Unrolled register

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

entity ShiftRegN is
    generic (L : positive;           -- total number of bits N * M
              M : positive);
    port (  clk         : in std_logic;
            reset       : in std_logic;
            en          : in std_logic;
            din1, din2  : in std_logic_vector(M-1 downto 0);
            dout        : out std_logic_vector(L-1 downto 0));
end ShiftRegN;

architecture Behavioral of ShiftRegN is
    signal s_data : std_logic_vector(L-1 downto 0);
begin

process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            s_data <= (others => '0');
        elsif en = '1' then
            s_data <= s_data(L-M*2-1 downto 0) & din1 & din2;
        end if;
    end if;
end process;

          dout <= s_data;

end Behavioral;
```

# Summary

- After completing this class and lab 2 you should be able to:
  - identify different types of sorting networks
  - analyze complexity and efficiency of sorting networks
  - specify parameterizable sorting networks in VHDL
  - understand the HLS design flow
  - create HLS projects in Vivado
  - validate and debug your C design
  - synthesize the C design to an RTL implementation and apply synthesis directives
  - use the results of HLS in other Vivado RTL projects
- … lab 2 is available at
  - http://sweet.ua.pt/iouliia/Courses/PDP_TUT/index.html