

# Designing a Custom AXI-Stream Peripheral

## Final Project

---

LECTURE 10

IOULIIA SKLIAROVA

# AXI4-Stream

---

For high-speed streaming data in point-to-point communications.

AXI4-Stream removes the requirement for an address phase altogether and allows unlimited data burst size.

AXI4-Stream interfaces and transfers do not have address phases and are therefore **not considered to be memory-mapped**.










The AXI4-Stream protocol defines a single unidirectional channel for transmission of streaming data (with a handshaking data flow).

The AXI4-Stream channel models the write data channel of AXI4.

Use the AXI4-Stream protocol for applications that typically focus on a data-centric and data-flow paradigm where the concept of an address is not present or not required.

# AXI4-Stream Interface Signals

Signal	Source	Description
<b>ACLK</b>	Clock source	The global clock signal. All signals are sampled on the rising edge of <b>ACLK</b> .
<b>ARESETn</b>	Reset source	The global reset signal. <b>ARESETn</b> is active-LOW.
<b>TVALID</b>	Master	<b>TVALID</b> indicates that the master is driving a valid transfer. A transfer takes place when both <b>TVALID</b> and <b>TREADY</b> are asserted.
<b>TREADY</b>	Slave	<b>TREADY</b> indicates that the slave can accept a transfer in the current cycle.
<b>TDATA[(8n-1):0]</b>	Master	<b>TDATA</b> is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
<b>TSTRB[(n-1):0]</b>	Master	<b>TSTRB</b> is the byte qualifier that indicates whether the content of the associated byte of <b>TDATA</b> is processed as a data byte or a position byte.
<b>TKEEP[(n-1):0]</b>	Master	<b>TKEEP</b> is the byte qualifier that indicates whether the content of the associated byte of <b>TDATA</b> is processed as part of the data stream. Associated bytes that have the <b>TKEEP</b> byte qualifier deasserted are null bytes and can be removed from the data stream.
<b>TLAST</b>	Master	<b>TLAST</b> indicates the boundary of a packet.
<b>TID[(i-1):0]</b>	Master	<b>TID</b> is the data stream identifier that indicates different streams of data.
<b>TDEST[(d-1):0]</b>	Master	<b>TDEST</b> provides routing information for the data stream.
<b>TUSER[(u-1):0]</b>	Master	<b>TUSER</b> is user defined sideband information that can be transmitted alongside the data stream.

- Manuais e guias da Xilinx
-  [Vivado Design Suite User Guide](#)
  -  [MicroBlaze Processor Reference Guide](#)
  -  [AXI GPIO v2.0 - LogiCORE IP Product Guide](#)
  -  [AXI Timer v2.0 - LogiCORE IP Product Guide](#)
  -  [Fixed Interval Timer v2.0 - LogiCORE IP Product Guide](#)
  -  [AXI Interrupt Controller \(INTC\) - LogiCORE IP Product Guide](#)
  -  [AXI Reference Guide](#)
  -  [AXI4 Protocol Specification](#)
  -  [AXI4-Stream Protocol Specification](#)

# AXI4-Stream Handshake Process

---

The **TVALID** and **TREADY** handshake determines when information is passed across the interface.

A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface.

For a transfer to occur both the **TVALID** and **TREADY** signals must be asserted.

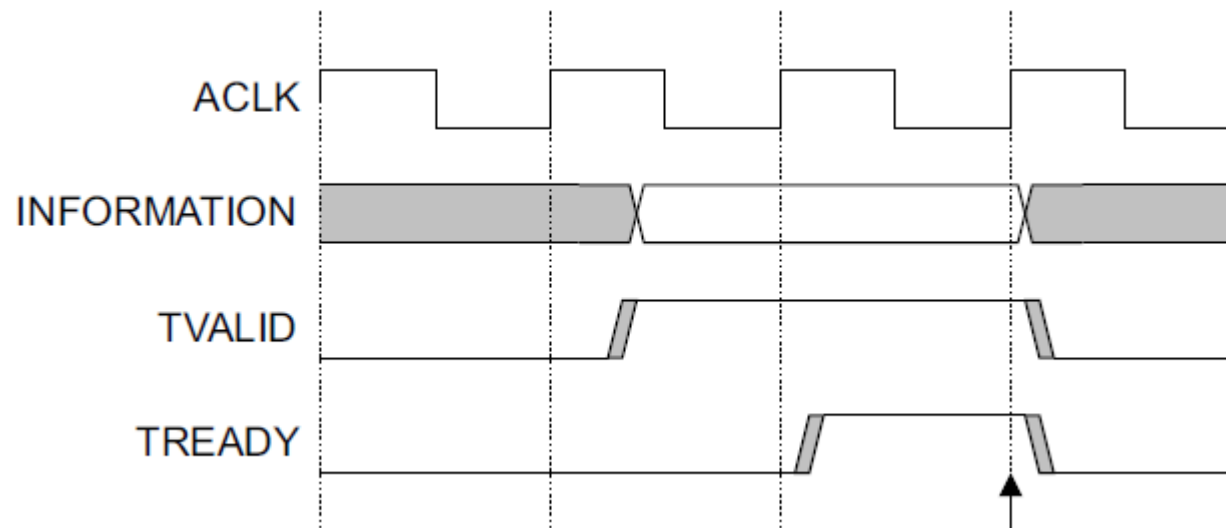
Either **TVALID** or **TREADY** can be asserted first or both can be asserted in the same **ACLK** cycle.

A master is not permitted to wait until **TREADY** is asserted before asserting **TVALID**. Once **TVALID** is asserted it must remain asserted until the handshake occurs.

A slave is permitted to wait for **TVALID** to be asserted before asserting the corresponding **TREADY**. If a slave asserts **TREADY**, it is permitted to deassert **TREADY** before **TVALID** is asserted.

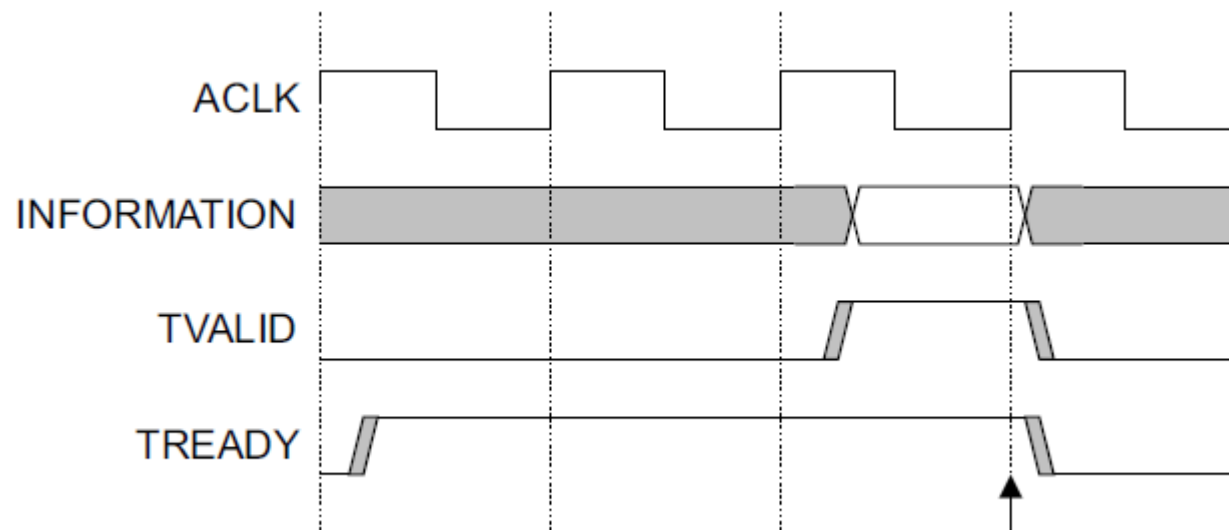
# TVALID Before TREADY Handshake

The master presents the data and control information and asserts the TVALID signal HIGH. Once the master has asserted **TVALID**, the data or control information from the master must remain unchanged until the slave drives the **TREADY** signal HIGH, indicating that it can accept the data and control information. In this case, transfer takes place once the slave asserts **TREADY** HIGH. The arrow shows when the transfer occurs.



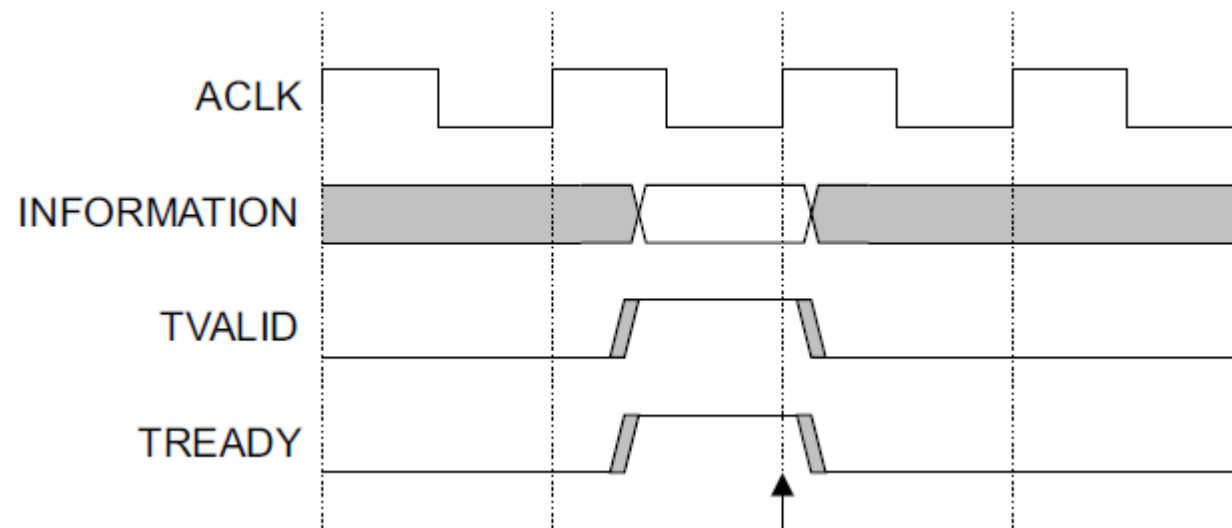
# TREADY Before TVALID Handshake

The slave drives TREADY HIGH before the data and control information is valid. This indicates that the destination can accept the data and control information in a single cycle of ACLK. In this case, transfer takes place once the master asserts TVALID HIGH. The arrow shows when the transfer occurs.



# TVALID With TREADY Handshake

The master asserts TVALID HIGH and the slave asserts TREADY HIGH in the same cycle of ACLK. In this case, transfer takes place in the same cycle as shown by the arrow.



# Examples

---

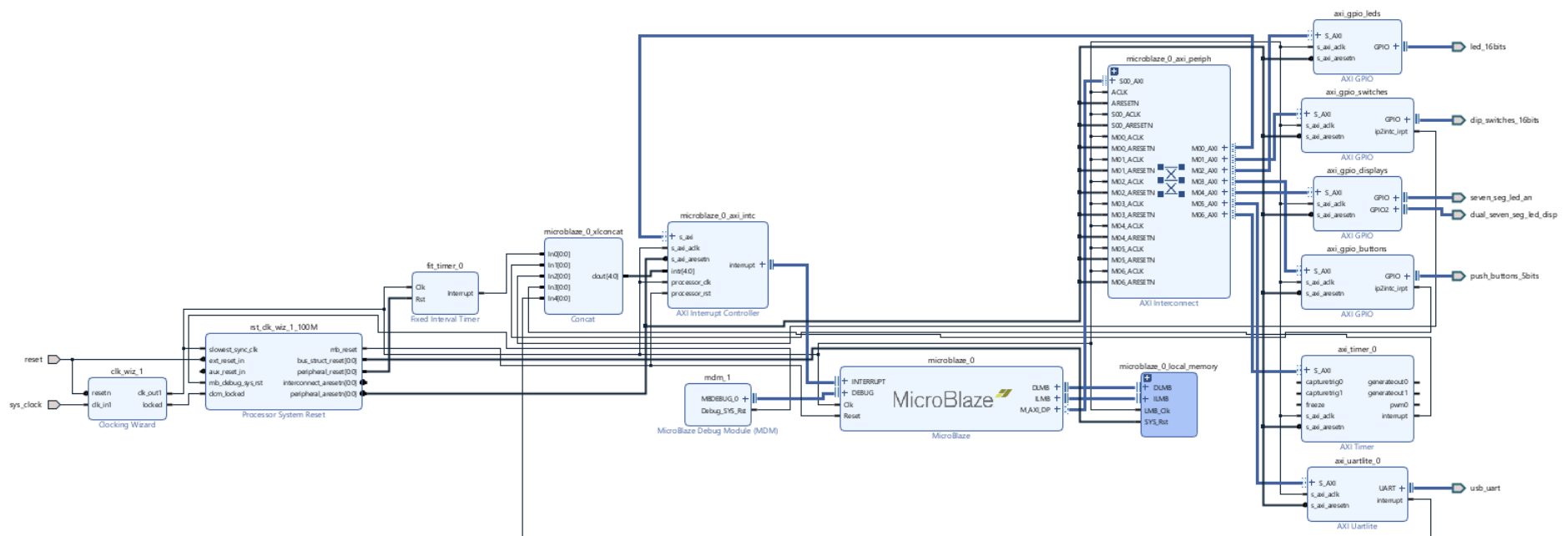
## Reverse endianness

- **Endianness** is the order of bytes in a word of digital data.
- Endianness is primarily expressed as **big-endian** or **little-endian**.
- A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest.
- A little-endian system, in contrast, stores the least-significant byte at the smallest address.
- 0xAB347801 => 0x017834AB

## Population count (Hamming weight)



# Example 1 – Starting Point



# Adding Stream Links to MicroBlaze

Re-customize IP

**MicroBlaze (11.0)**

Documentation IP Location Advanced

IP Symbol Resources

Frequency  
Area  
Performance

Resource Estim

Percent (%)

100.0  
90.0  
80.0  
70.0  
60.0  
50.0  
40.0  
30.0  
20.0

Component Name microblaze\_0

**AXI and ACE Interfaces**

Select Bus Interface AXI

☐ Enable Peripheral AXI Instruction Interface

☒ Enable Peripheral AXI Data Interface

**Stream Interfaces**

Number of Stream Links 1 [0 - 16]

**Other Interfaces**

☐ Enable Trace Bus Interface

< Back Next > Page 4 of 4

OK Cancel

# Create & Package New IP

Create and Package New IP

**Add Interfaces**  
Add AXI4 interfaces supported by your peripheral

☐ Enable Interrupt Support

**Interfaces**

- S00\_AXIS
- M00\_AXIS

ReverseEndiannessCop\_v1.0

Name: M00\_AXIS

Interface Type: Stream

Interface Mode: Master

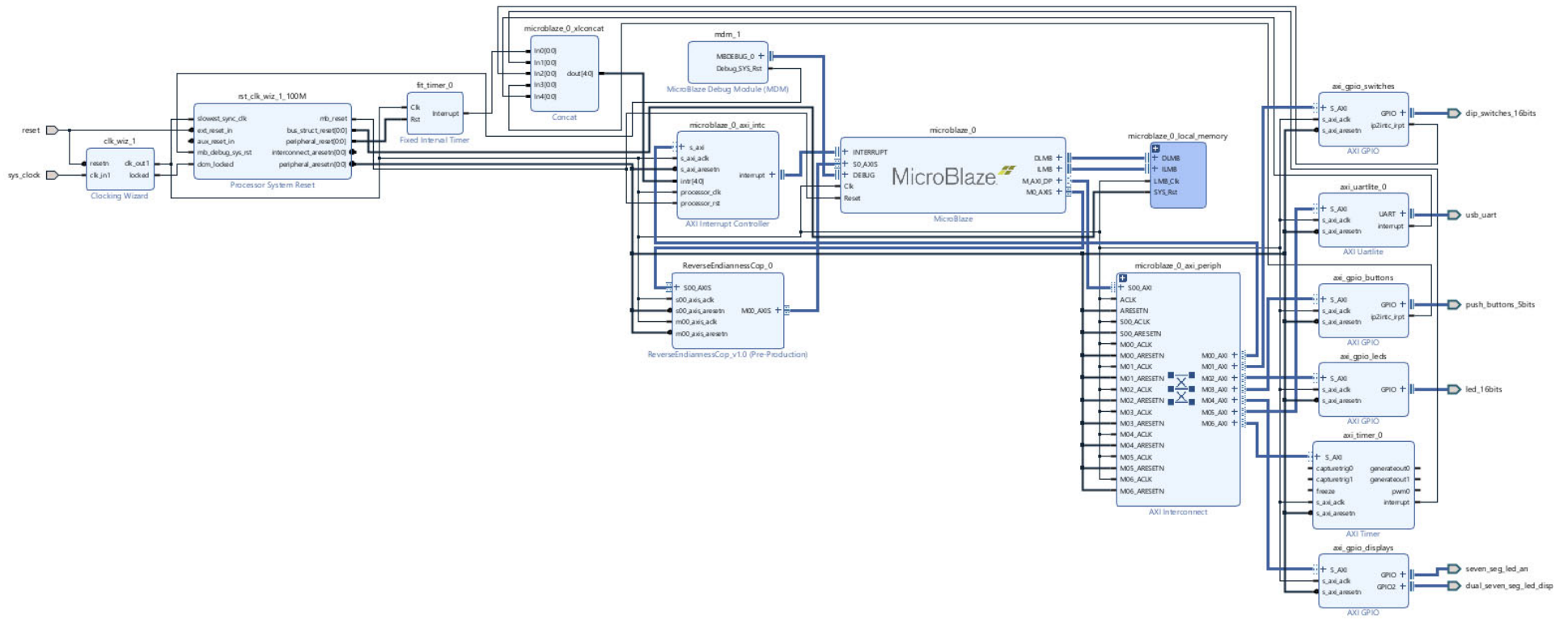
Data Width (Bits): 32

Memory Size (Bytes): 64

Number of Registers: 4 [4..512]

< Back Next > Finish Cancel

# Example 1 Block Design



# Example 1 – Reverse Endianness

---

Import Block Design

Configure the MicroBlaze to have one pair of stream links

Create and Package new IP (*ReverseEndiannessCop*)

Add IP

Edit in IP Packager (the code is given on eLearning)

Generate output products

Create HDL Wrapper

Generate Bitstream

Export Hardware

Launch Vitis

# Vitis

---

The C code is given on eLearning

There is no need to correct the IP makefiles

The code processes an array of N (N=4000) integers (32b = 4B)

The code uses the AXI timer to measure time

The code does not work. Why?

```
#define N 4000  
  
int srcData[N], dstData[N];
```

What is the size of data?

Where do these data reside?

# Vitis – Changing the Stack Size

The screenshot shows the Vitis IDE interface. On the left, the 'Explorer' pane displays the project structure. The file 'Iscrip.ld' is highlighted with a yellow circle. On the right, the 'Linker Script: Iscrip.ld' configuration page is shown. The 'Stack and Heap Sizes' section is also highlighted with a yellow circle, showing 'Stack Size' set to '0x400' and 'Heap Size' set to '0x800'. Below this, the 'Section to Memory Region Mapping' table is visible.

**Explorer**

- resources
- platform.spr
- platform.tcl
- ReverseEndiannessApp\_system [ ReverseEndianness ]
  - ReverseEndiannessApp [ standalone\_microblaze\_0 ]
    - Binaries
    - Includes
    - Debug
    - src
      - helloworld.c
      - platform\_config.h
      - platform.c
      - platform.h
      - ReverseEndianness.c
      - Iscrip.ld**
    - \_ide
    - ReverseEndiannessApp.prj
  - \_ide
  - Debug
  - ReverseEndiannessApp\_system.sprj

**Linker Script: Iscrip.ld**

A linker script is used to control where different sections of the program are loaded in memory. In this page, you can define new memory regions, and

**Available Memory Regions**

Name
microblaze_0_local_memory_ilmb_bram_if_cntlr_...

**Stack and Heap Sizes**

Stack Size: 0x400  
Heap Size: 0x800

**Section to Memory Region Mapping**

Section Name	Memory Region
.text	microblaze_0_local_memory_ilmb_bram_if_cntlr_...
.note.gnu.build-id	microblaze_0_local_memory_ilmb_bram_if_cntlr_...
.init	microblaze_0_local_memory_ilmb_bram_if_cntlr_...
.fini	microblaze_0_local_memory_ilmb_bram_if_cntlr_...
.stap	microblaze_0_local_memory_ilmb_bram_if_cntlr_...

# Examples (AXI-Stream Coprocessor)

---

## Reverse endianness

## Population count (Hamming weight)

- the number of non-zero entries ('1' bits) in a word of data.
- 0xAB347801 => 10101011\_00110100\_01111000\_00000001 => 13



# Example 2 – Starting Point

Continue to work on the same project (as in the previous class)

Change the number of stream links in the MicroBlaze to 2

The screenshot shows the 'Re-customize IP' dialog for 'MicroBlaze (11.0)'. The 'Resources' tab is active, displaying a bar chart of resource estimates (Frequency, Area, Performance) and configuration options for buses and stream interfaces. The 'Number of Stream Links' is set to 2, highlighted with a yellow circle.

**Resource Estimates**

Resource	Frequency (%)	Area (%)	Performance (%)
Frequency	95.0	15.0	10.0
Area	15.0	15.0	10.0
Performance	10.0	15.0	10.0

**Buses**

**Local Memory Bus Interfaces**

- ☒ Enable Local Memory Bus Instruction Interface
- ☒ Enable Local Memory Bus Data Interface

**AXI and ACE Interfaces**

Select Bus Interface: AXI

- ☐ Enable Peripheral AXI Instruction Interface
- ☒ Enable Peripheral AXI Data Interface

**Stream Interfaces**

Number of Stream Links: 2 [0 - 16]

**Other Interfaces**

< Back Next > Page 4 of 4

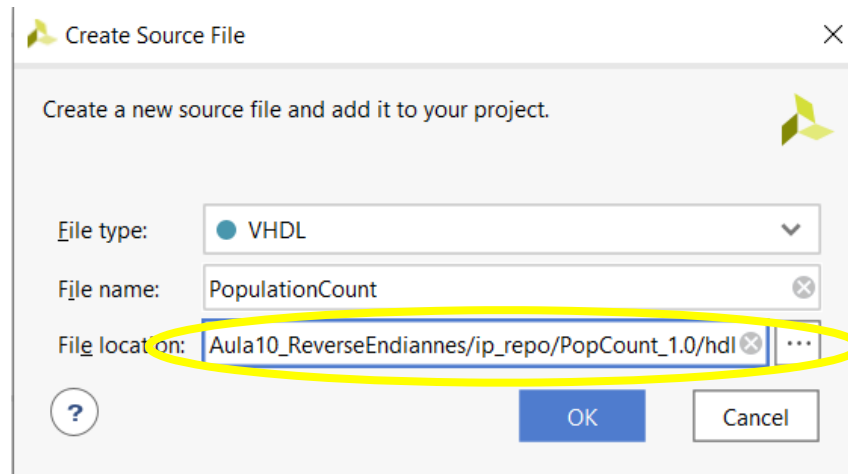
# Example 2 – Add New IP

Create and Package new IP - **PopCount**

Edit in IP packager

Change the three “default” files like in Example 1

Create a new source – PopulationCount.vhd (the code is given on eLearning)



Instantiate the PopulationCount module in the slave stream interface:

```
calc: PopulationCount
  generic map (N      => C_S_AXIS_TDATA_WIDTH)
  port map ( dataIn   => ... );
```

# For Loop VHDL Statement

---

A **for loop** statement is a sequential statement that can be used inside a process.

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

```
[loop_label:]  
for identifier in discrete_range loop  
  { sequential_statement }  
end loop [loop_label];
```

The **for loop** statement is used whenever an operation needs to be repeated.

The loop is **unrolled** statically – the number of loop iterations must be known at compile time.

**Loop unrolling** is a systematic method of achieving parallelism that can be automated.

This comes at a cost of a larger fabric footprint (more FPGA area).

# For Loop vs For Generate

---

The **for loops** are **sequential statements, containing sequential statements** (i.e. each iteration is sequenced to be executed after the previous one).

The **for-generate loops** are **concurrent** statements, **containing concurrent statements**, and this is how you can use it to make several instances of a component.

**For-generate loops** are used when specifying the exact hardware structure.

**For loops** are more suited for behavioral descriptions.

# Variables

---

For loops are often used with **variables**.

**Variables** are declared in the declaration part of processes:

```
variable_declaration  $\Leftarrow$   
variable identifier { , ... } : subtype_indication  
[:= expression] ;
```

The syntax of a **variable assignment** statement is given by the rule

```
variable_assignment_statement  $\Leftarrow$   
[label :] name := expression ;
```

A variable assignment **immediately overwrites the variable** with a new value (a signal assignment, on the other hand, schedules a new value to be applied to a signal at some later time).

# Population Count With a For Loop

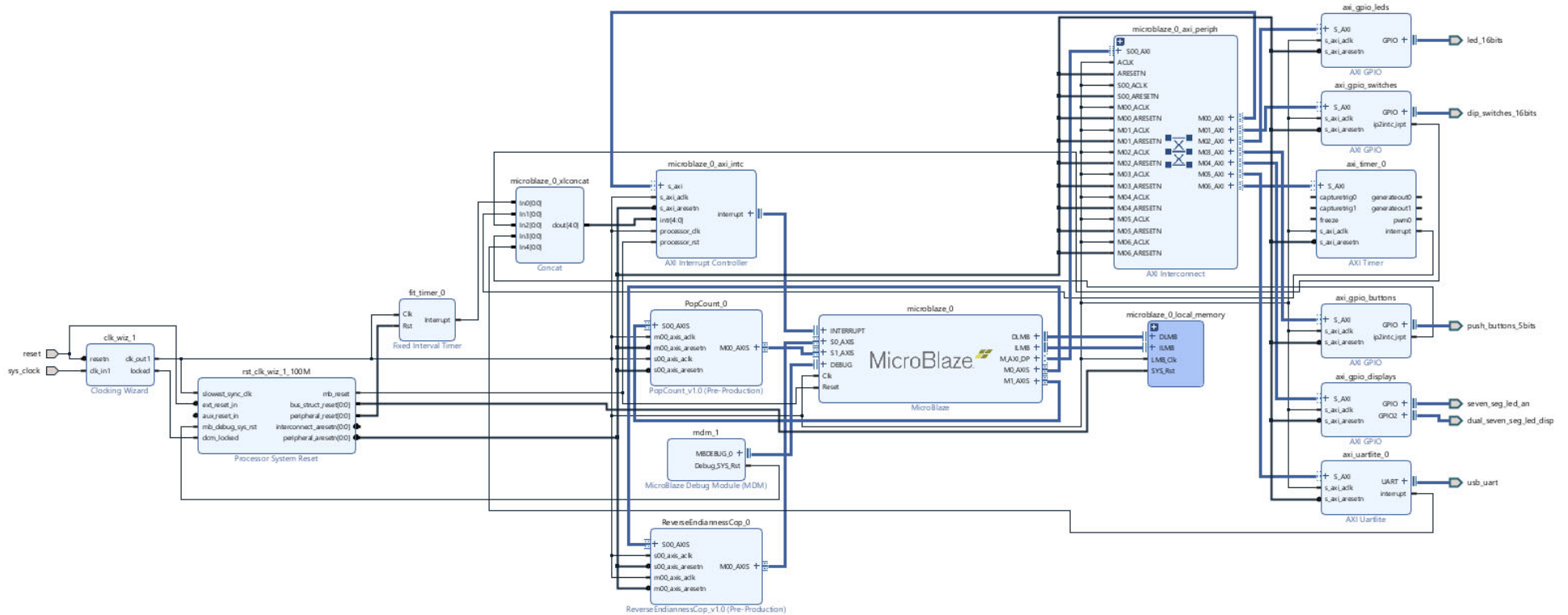
```
entity PopulationCount is
    generic(N      : positive := 4);
    port(dataIn   : in  std_logic_vector(N-1 downto 0);
          cntOut  : out std_logic_vector(N-1 downto 0));
end PopulationCount;

architecture Behavioral of PopulationCount is
    signal s_cnt : natural range 0 to N;
begin
    process(dataIn)
        variable v_cnt : natural range 0 to N;
    begin
        v_cnt := 0;
        for i in 0 to N-1 loop
            if dataIn(i) = '1' then
                v_cnt := v_cnt + 1;
            end if;
        end loop;
        s_cnt <= v_cnt;
    end process;

    cntOut <= std_logic_vector(to_unsigned(s_cnt, N));
end Behavioral;
```

A long sequence of 31 adders will be generated.

# Example 2 Block Design



# Example 2 – Further Steps

---

Generate output products

Create HDL Wrapper

Generate Bitstream

Export Hardware

Launch Vitis



# Vitis

---

Write the C code (on the basis of the ReverseEndianness example)

There is no need to correct the IP makefiles

Configure the right stack size

# Final Project

---

No formal guidelines will be given

Hardware + software

Hardware must include MB, memory, standard peripherals, and custom modules

Select an **operation** suitable for hardware (to increase the performance, to have a clearer implementation)

Compare software and hardware implementations of the selected operation

Demonstrate that you are familiar with the design flow:

- write VHDL code
- simulate with a testbench
- incorporate to block design
- do synthesis, implementation, report analysis
- determine critical path, optimize operating frequency
- write software

# Final Project - Operation

---

Select an **operation** suitable for hardware (to increase the performance, to have a clearer implementation)

- an instruction/function not supported by MB (popcnt, vector operations, complex bitwise/shift logic, specific peripheral, cryptography...)

Repeated operations are not allowed among students

Operations considered during classes are not allowed

Either bring your proposals to lab on June 8 or send them to me by e-mail

- title of the project
- brief description of the functionality (one phrase)
- brief description of the proposed architecture
- why to use the suggested custom hardware module?
- test procedure and user interaction

# Final Project – Proposal Example

---

Title of the project:

- Accelerating Population Count with a Hardware Co-Processor

Brief description of the functionality (one phrase):

- System with a hardware accelerator for calculating population count over a configurable-length array of 32-bit vectors

Brief description of the proposed architecture:

- The system will include a custom hardware module executing the population count operation over a 32-bit input. The module will interact with the MicroBlaze through AXI-Stream interface. The partial results will be accumulated in software. The performance of software and hardware implementations will be analyzed and compared.

Why to use the suggested custom hardware module?

- To reduce the processing time.

Test procedure and user interaction

- Input data will be randomly generated. Software will check the hardware results. Testbench for the accelerator. User interaction through UARTLite and serial terminal.

# Final Remarks

---

At the end of this lecture you should be able to:

- Design custom hardware modules interacting with the MicroBlaze through AXI-Stream interface
- Write C programs that make use of stream-connected custom hardware

## To do:

- Construct the considered hardware platforms
- Test the given applications in Vitis
- Complete lab. 8