Designing a Custom AXI-Stream Peripheral

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.

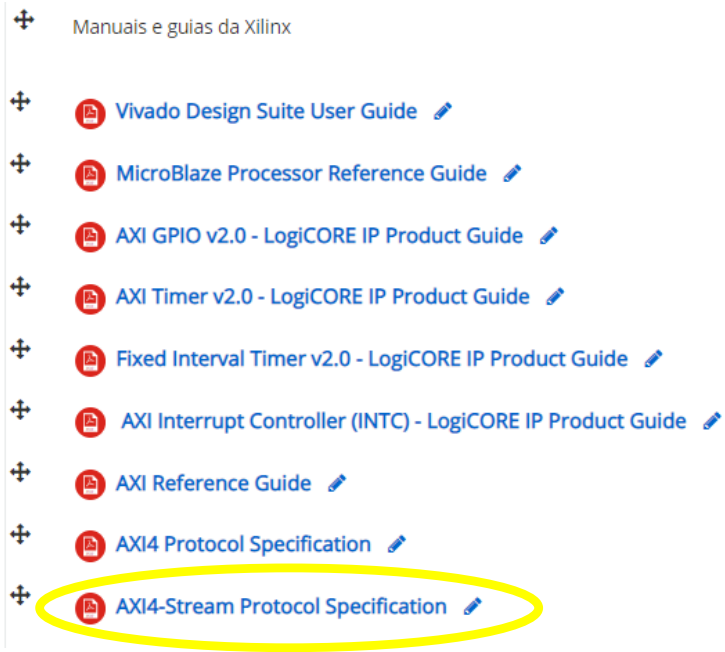
The AXI4-Stream protocol defines a single unidirectional channel for transmission of streaming data (with a handshaking data flow).

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



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.

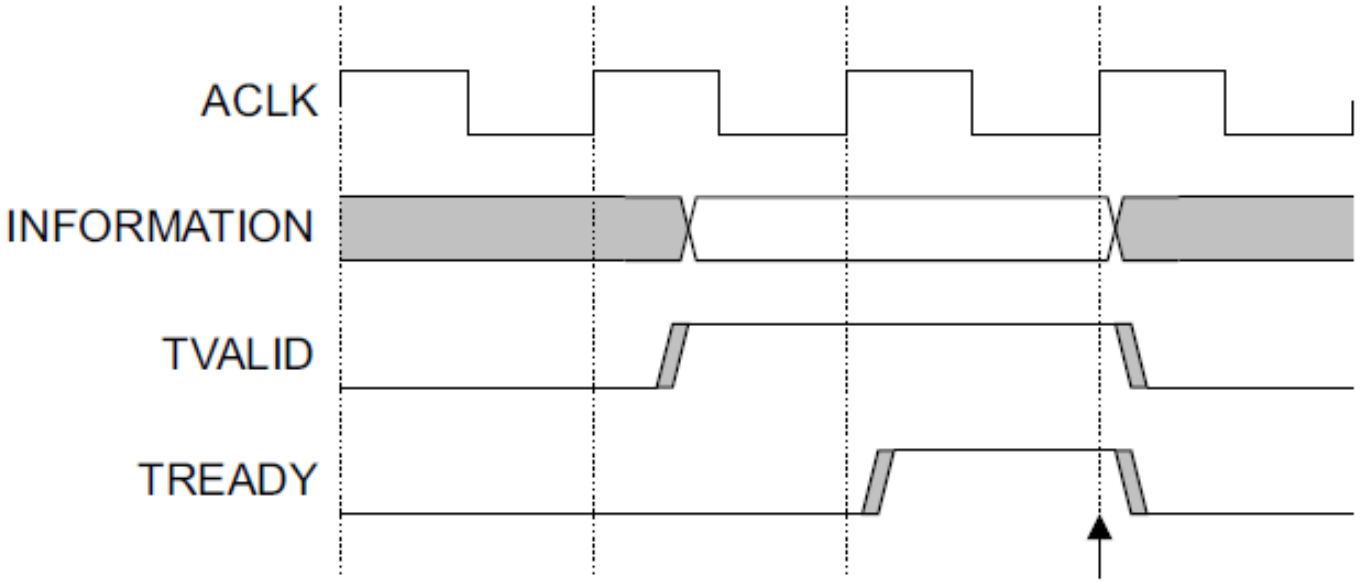
Either **TVALID** or **TREADY** can be asserted first or both can be asserted in the same **ACLK** cycle.

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.

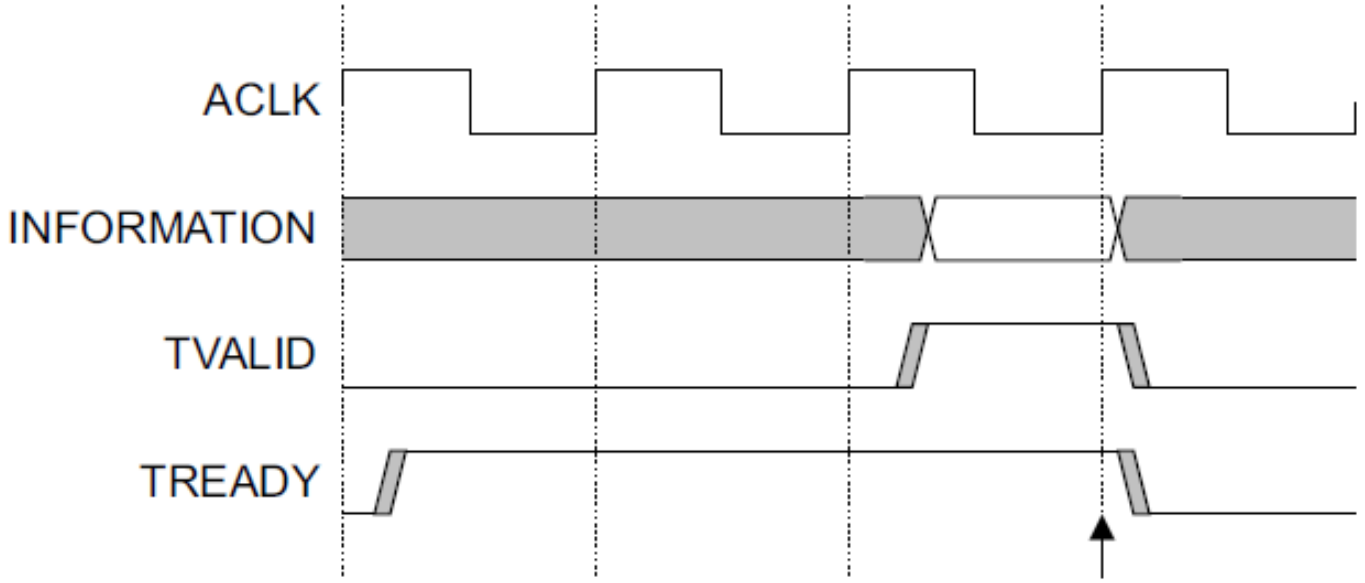
TVALID Before TREADY Handshake

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.



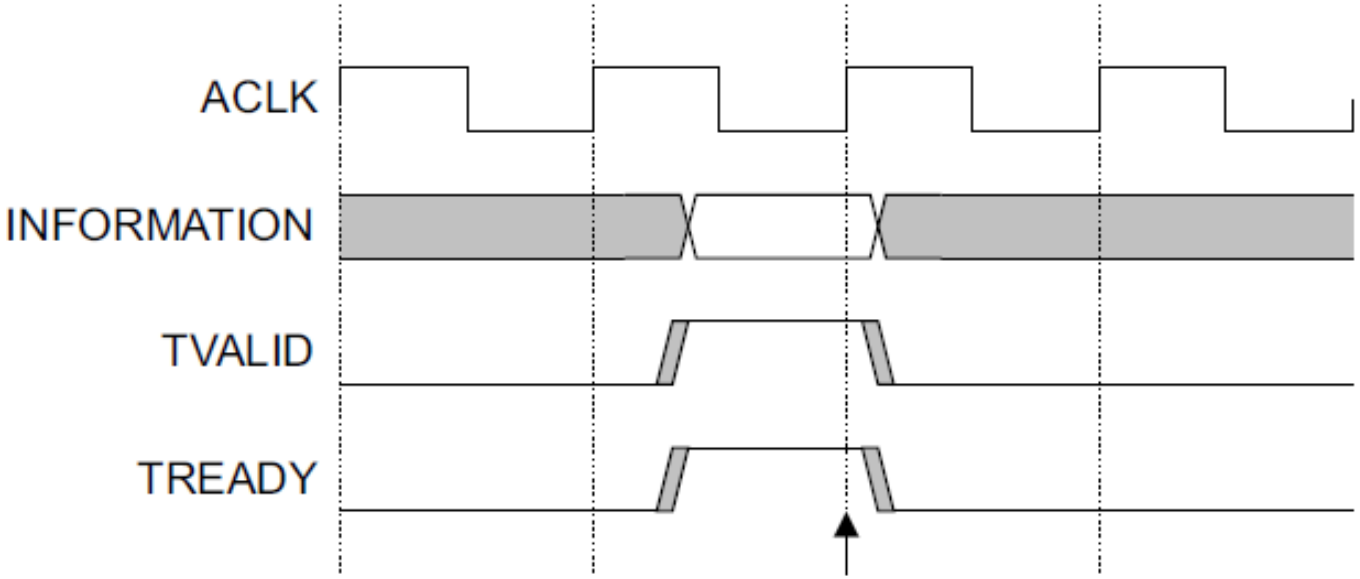
TREADY Before TVALID Handshake

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.



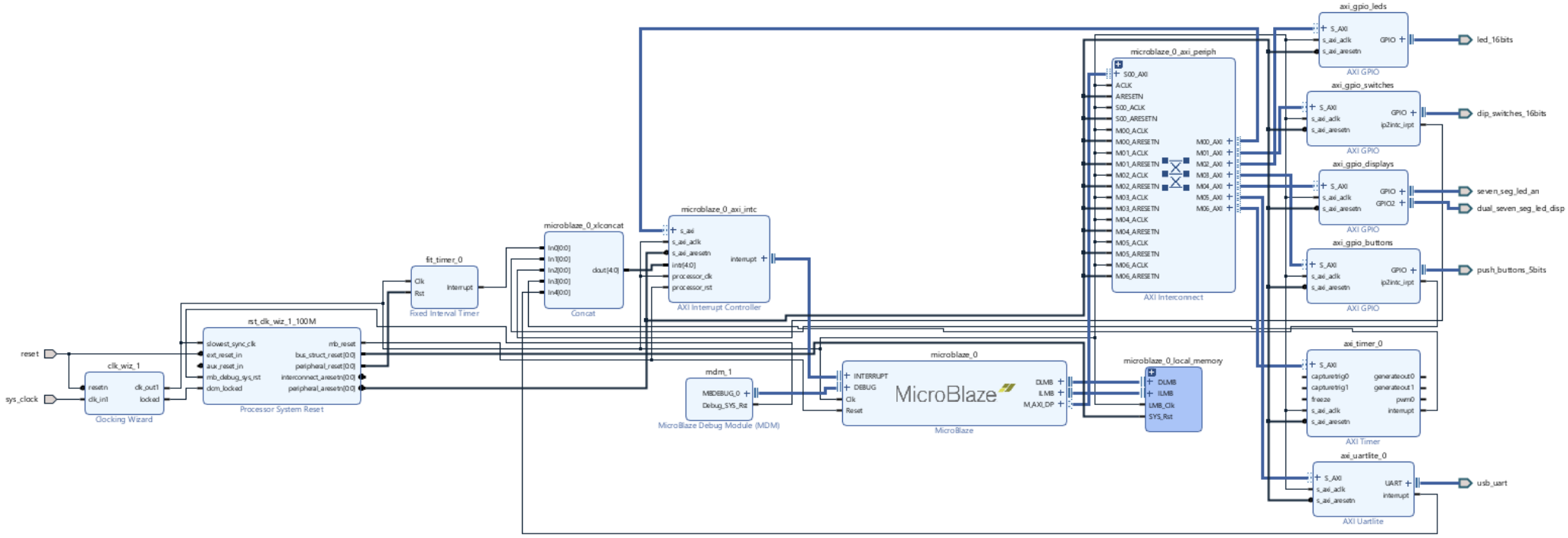
Examples

Reverse endianness

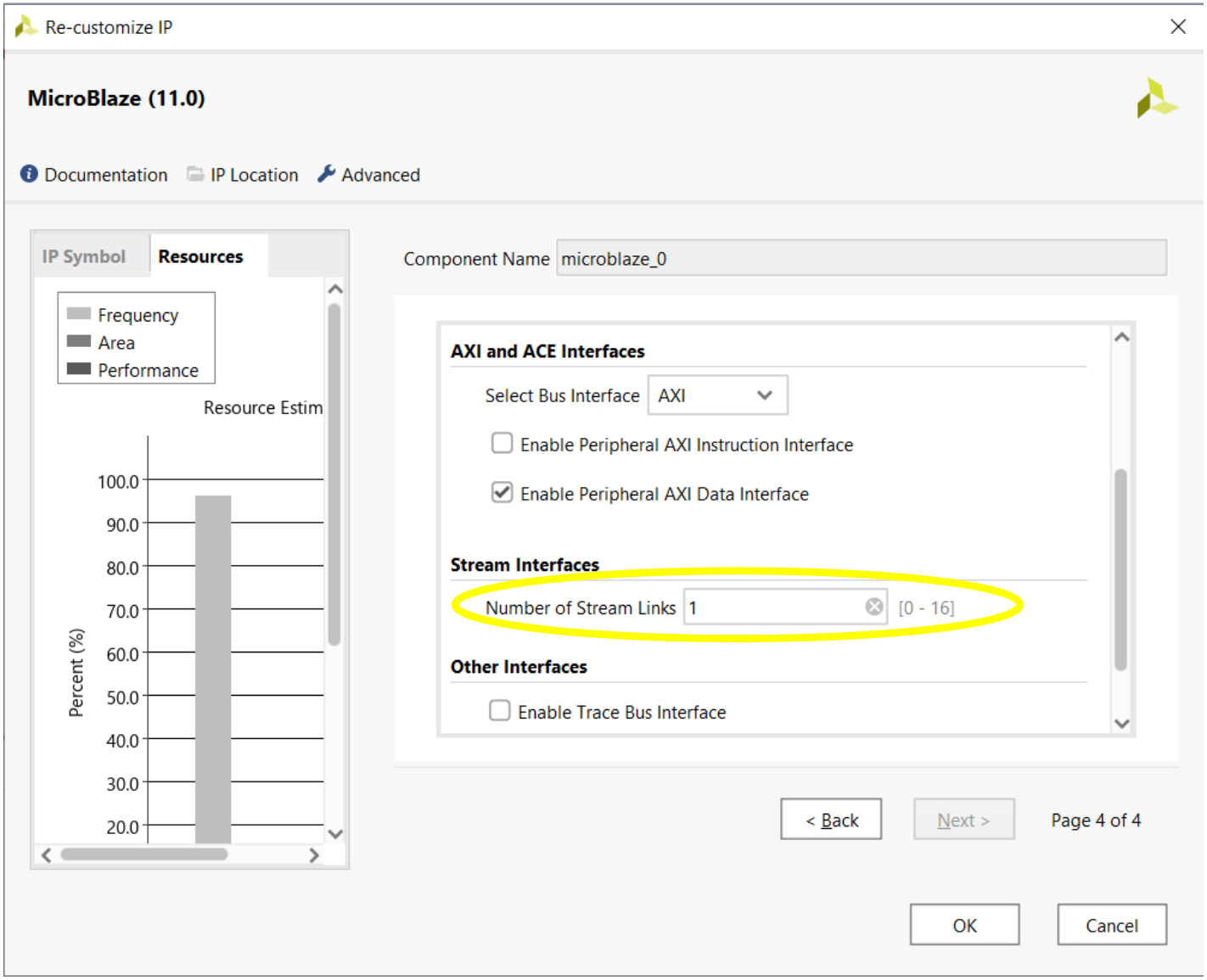
- **Endianness** is the order of bytes in a word of digital data.
- Endianness is primarily expressed as **big-endian** or **little-endian**.
- A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest.
- A little-endian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

Population count (Hamming weight)

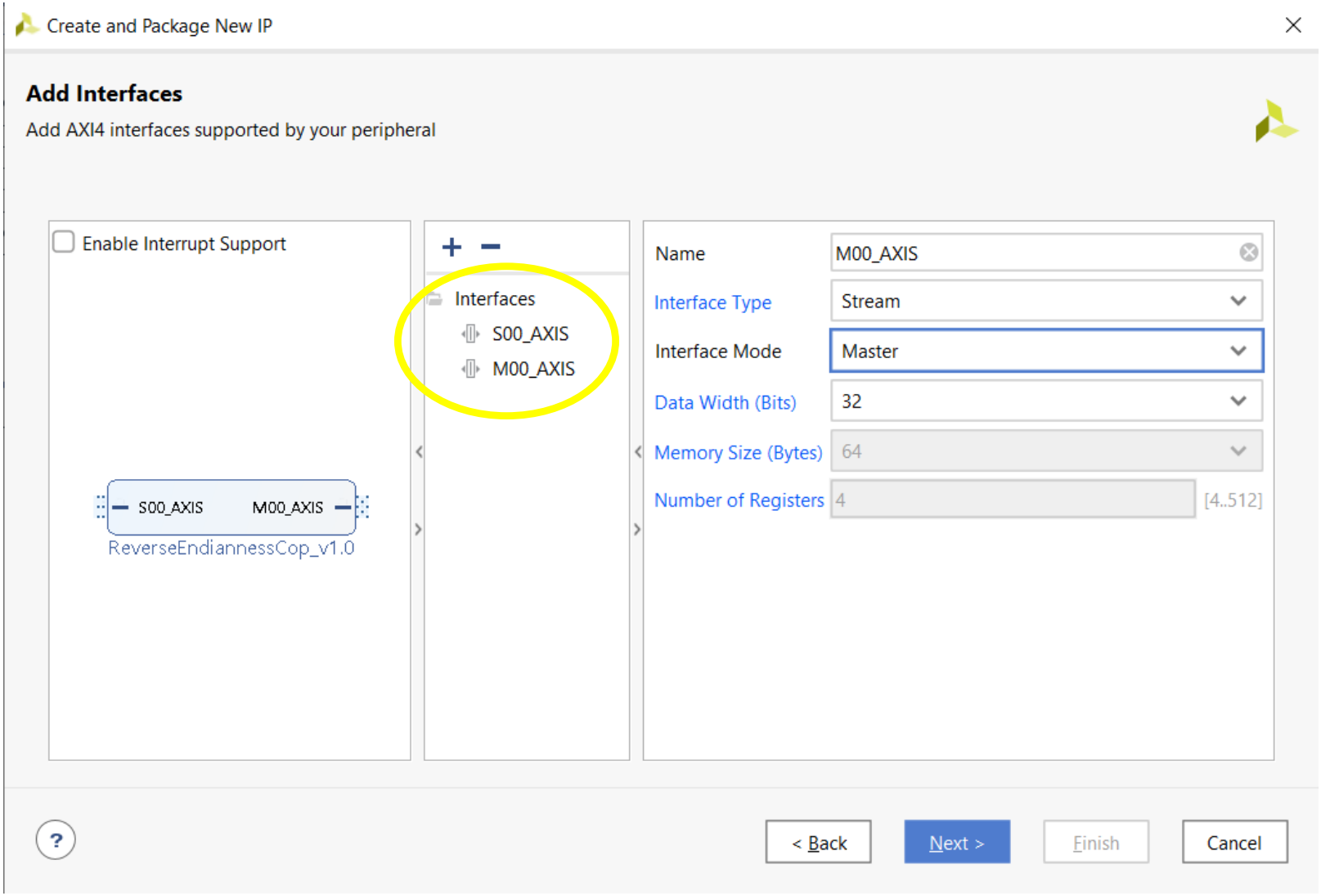
Example 1 – Starting Point



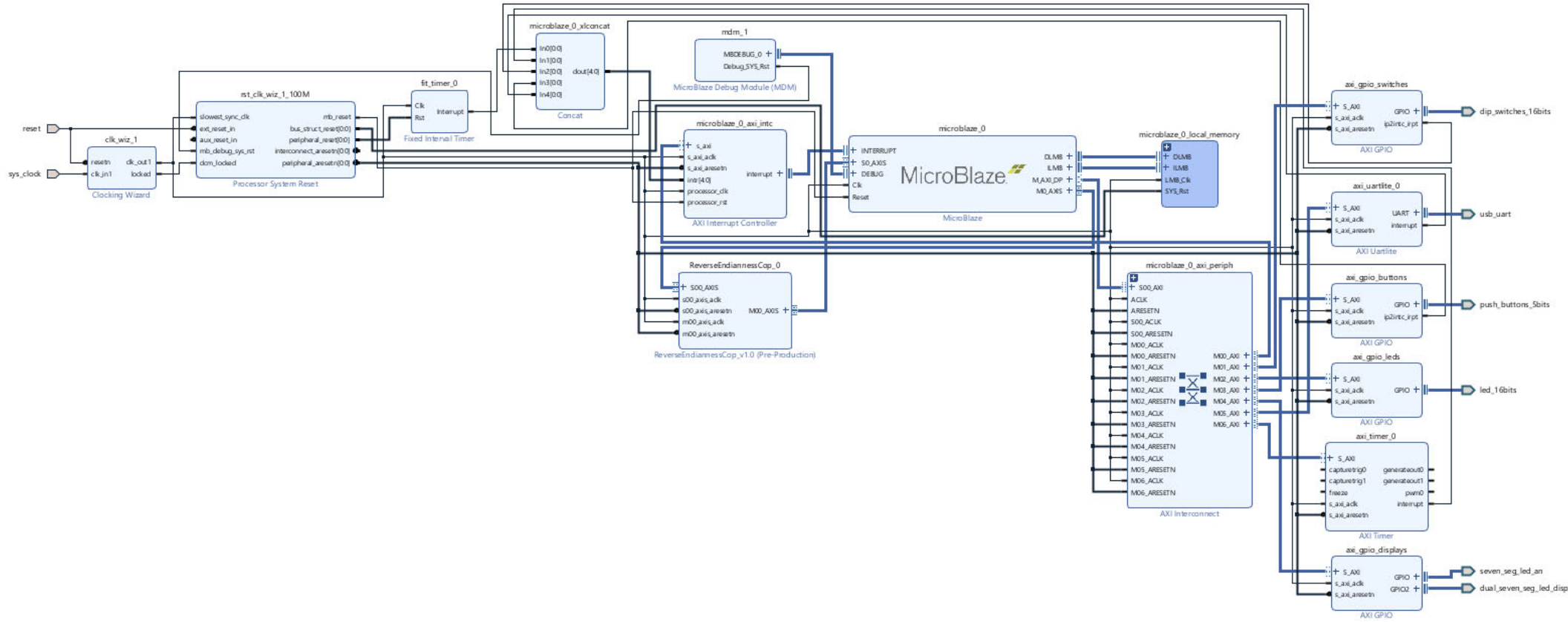
Adding Stream Links to MicroBlaze



Create & Package New IP



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

- `set_property CONFIG_VOLTAGE 3.3 [get_designs synth_1]`
- `set_property CFGBVS VCCO [get_designs synth_1]`

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

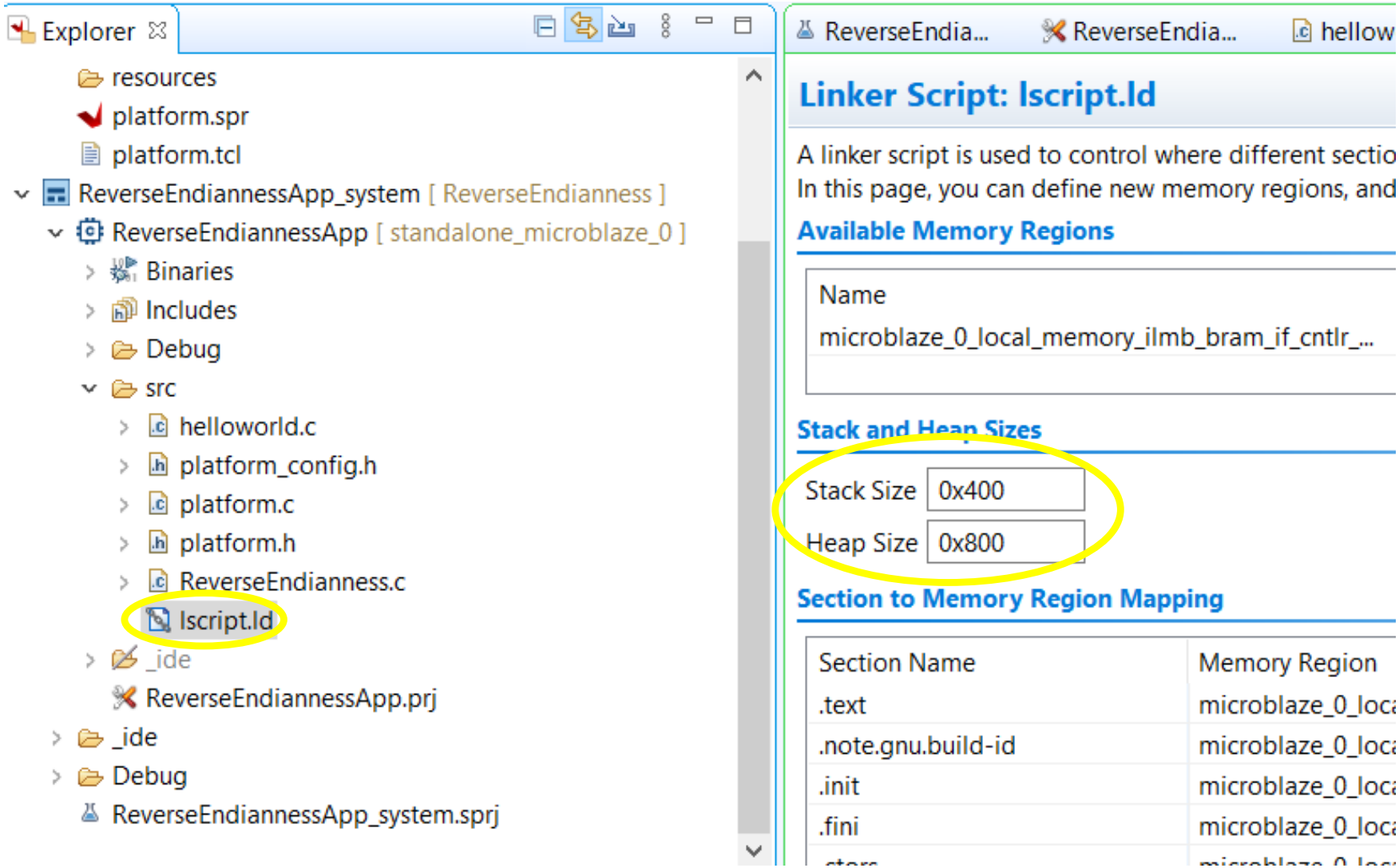
The code does not work. Why?

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

What is the size of data?

Where do these data reside?

Vitis – Changing the Stack Size



Examples (AXI-Stream Coprocessor)

Reverse endianness

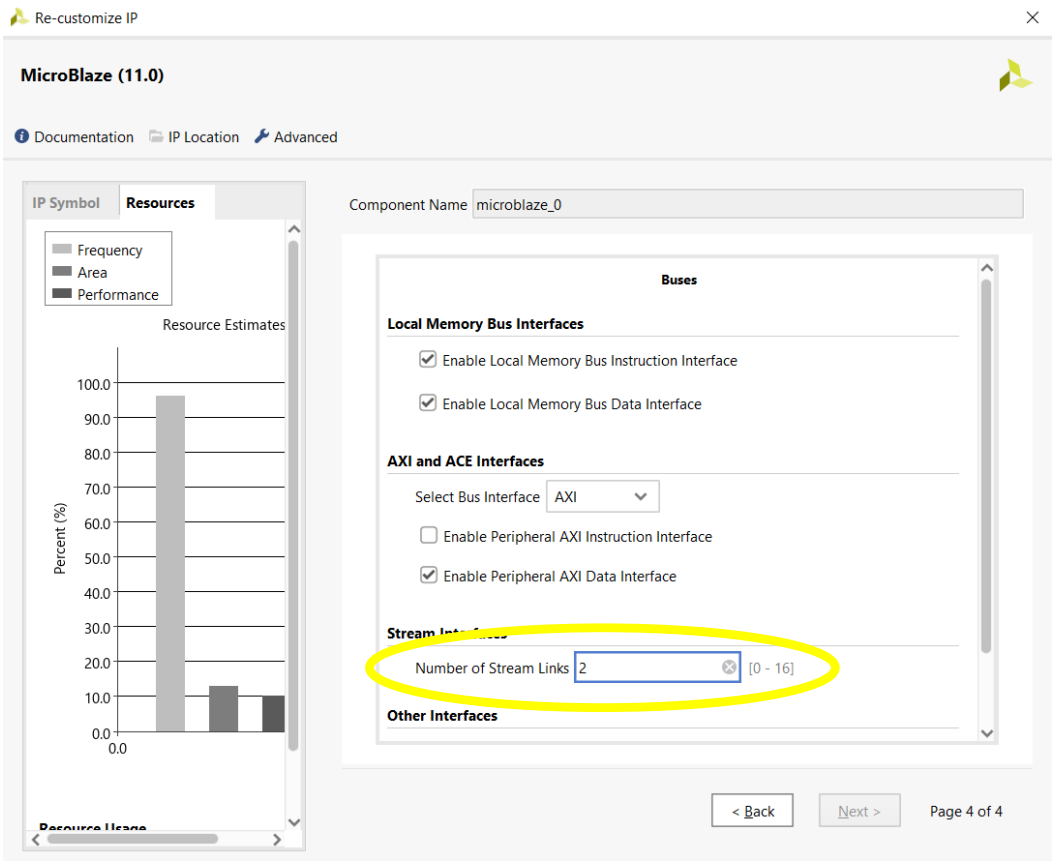
Population count (Hamming weight)

- 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

Change the number of stream links in the MicroBlaze to 2



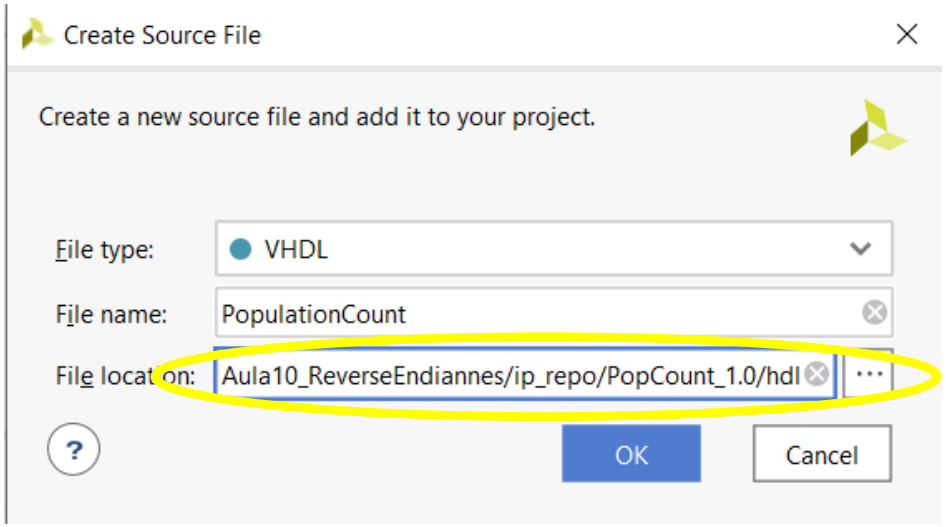
Example 2 – Add New IP

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).

Variables

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

Variables are declared in the declaration part of processes:

```
variable_declaration ⇐
variable identifier { , … } : subtype_indication
[:= expression] ;
```

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

```
variable_assignment_statement ⇐
[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.

For Loop vs For Generate

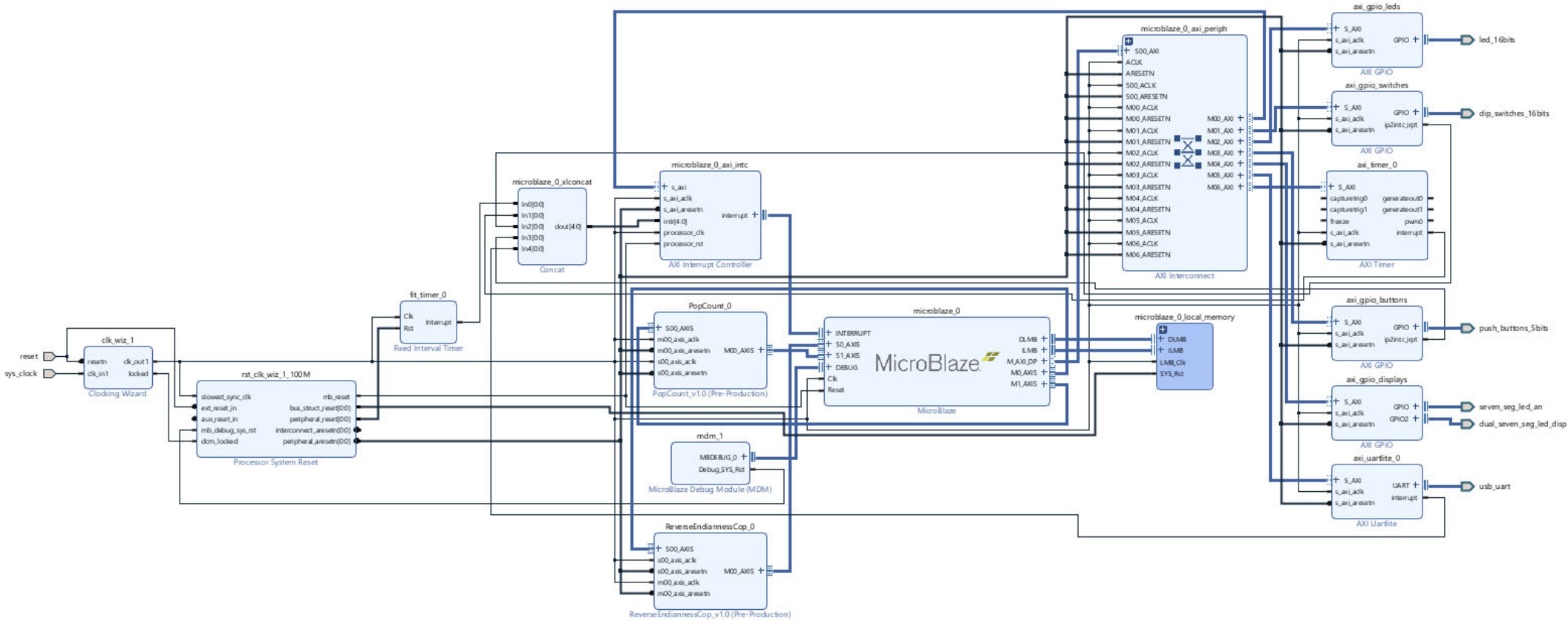
The **for loops** are **sequential statements**, **containing sequential statements** (i.e. each iteration is sequenced to be executed after the previous one).

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.

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 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 application in Vitis
- Complete lab. 8