Streaming Histogram

Hudson River Trading

Take-home Assignment

Overview

In this assignment you will design and implement a module that counts the number of occurrences of each value of the valid data words within a multi-word streaming data bus. The resulting counts are read out one at a time via the querying interface.

The goal of the assignment is to design a block that will work in real hardware, not just in simulation. As you approach the solution think how the design will be synthesized and what the resulting circuit will look like when mapped to the FPGA primitives. In order to confirm that the design works in hardware the included makefile will run synthesis and re-rerun the simulation using the gate-level netlist instead of the verilog design file. Please make sure the included makefile successfully completes all the steps with the default parameter values. We will rerun the make flow with your submission and the designs that fail any of the steps will not be accepted.

Administering a unix/linux system and installing the required packages is a part of the assignment as well. We are looking for results-oriented candidates that are able to tackle real-world issues independently.

Additionally you should be able to understand the tool limitations and be able to work around them. Different tools may support different subsets of Verilog and SystemVerilog features. Therefore the design should be written such that it works for all the tools, in this case both Icarus simulator and Yosys synthesis.

Files

When you unpack the zip-file, you should find the files listed below in a folder called `assignment/`.

File Name	Description
StreamingHistogram.pdf	This document.
StreamingHistogram.v	Assignment template. This is the file you need to modify.
StreamingHistogram_tb.v	Verilog testbench with random stimulus, model, and checker.
Makefile	Simulation build and run rules for Icarus Verilog.

Requirements

- The submission should contain syntatically valid and *synthesizable* Verilog-2005 implementation.
- The design should be correct when presented with a valid stimulus pattern.
- The design should efficiently infer and consume resources in a modern FPGA.
- The design should be scalable. Incrementally increasing the values of the parameters should not drastically affect the performance or the size of the design.
- The design should be reasonably portable between different FPGA vendors and should not instantiate vendor technology specific primitive cells.

Interface Definitions

Streaming Interface

The streaming interface transfers data over a multi-word data bus. An asserted `stream_valid` signal indicates that the data transfer is taking place in the current clock cycle and validates all the data words within `stream_data` signal. The `stream_valid` signal can be asserted and negated arbitrarily.

Querying Interface

The querying interface consists of a `query_valid` request validation signal, `query_word` request value signal, and `query_count` response signal. When `query_valid` signal is asserted the `query_word` contains the word value which identifies the counter being queried. 3 cycles following the assertion of the `query_valid` signal, the `query_count` signal contains the number of times `query_word` value was observed on any of the valid data words within `stream_data` signal.

Hint

There is no timing relationship between the streaming and querying interfaces. For ease of implementation the streaming interface is guaranteed to have been idle with negated `stream_valid` for at least 10 cycles prior to `query_valid` assertion, and will remain idle for the duration of querying interface activity.

Module Parameters

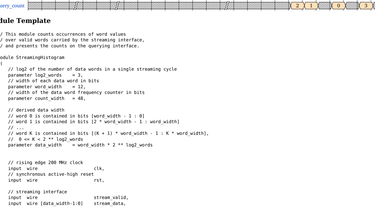
Name	Values	Description
log2_words	2-4	log2 of the number of data words in a single streaming cycle.
word_width	8-12	width of each data word in bits.
count_width	32-48	width of the data word frequency counter in bits.
data_width	32-192	derived stream data width equal to `word_width * 2 ** log2_words`. Word K is contained in `stream_data` bits `(K + 1) * word_width - 1 : K * word_width`, for 0 <= K < 2 ** log2_words.

Module Signals

Name	Direction	Width	Description
clk	input	1	Rising-edge 200 MHz clock.
rst	input	1	The module is supplied with an active-high synchronous reset. The `rst` input will be asserted for a minimum of 10 clock cycles and will be followed by a minimum of 100,000 clock cycles of quiescent and inactive stimulus.
stream_valid	input	1	Stream interface valid. When asserted indicated the transfer is taking place and validates `stream_data` signal.
stream_data	input	data_width	Stream interface data words. Word K is contained in `stream_data[(K + 1) * word_width - 1 : K * word_width]`, for 0 <= K < 2 ** log2_words.
query_valid	input	1	Querying interface request valid. Validates `query_word` and requests the count of the number of times `query_word` was observed on the streaming interface to be driven on `query_count` in 3 cycles.
query_word	input	word_width	Querying interface request word value.
query_count	output	count_width	Querying interface response count. Valid 3 cycles after `query_valid` assertion. Contains the number of times the corresponding value of `query_word` was observed on the streaming interface among valid data words withing `stream_data` signal.

Example Waveform

The figure below shows an example of module operation with a 2-word data bus (log2_words = 1). In order to make things clearer, the `stream_data[0]` waveform in the diagram represents the lower half of the `stream_data` signal, and `stream_data[1]` represents the upper half. After completion of the streaming stimulus, the querying interface sampled the counts for several word values. The values of 10, 11 and 13 were observed 2, 1 and 3 times respectively, and the value of 12 was never observed.



Module Template

```
// This module counts occurrences of word values
// over valid words carried by the streaming interface,
// and presents the counts on the querying interface.

module StreamingHistogram
#(
    // log2 of the number of data words in a single streaming cycle
    parameter log2_words   = 3,
    // width of each data word in bits
    parameter word_width   = 10,
    // width of the data word frequency counter in bits
    parameter count_width  = 48,

    // derived data width
    // word K is contained in bits [word_width - 1 : 0]
    // word 1 is contained in bits [2 * word_width - 1 : 1 * word_width]
    // ...
    // word K is contained in bits [(K + 1) * word_width - 1 : K * word_width],
    //  0 <= K < 2 ** log2_words
    parameter data_width   = word_width * 2 ** log2_words
)
(
    // rising edge 200 MHz clock
    input  wire                   clk,
    // synchronous active-high reset
    input  wire                   rst,

    // streaming interface
    input  wire                   stream_valid,
    input  wire [data_width-1:0]  stream_data,

    // counter querying interface
    input  wire                   query_valid,
    input  wire [word_width-1:0]  query_word,
    output wire [count_width-1:0] query_count
);

    // your implementation here ...

endmodule
```

Testing the design

Installation

The included flow uses make, git, Icarus Verilog, Yosys, and GTKWave to compile, simulate, synthesize, and debug your design. For example in Ubuntu 21.04 you can get the necessary tools by running the command:

```
sudo apt install make git iverilog yosys gtkwave
```

and following the prompts.

Icarus simulator
https://steveicarus.github.io/iverilog/

Yosys Open Synthesis Suite
https://yosyshq.net/yosys/

GTKWave VCD viewer
http://gtkwave.sourceforge.net/

If you cannot install these tools and have no other tools at your disposal, please email `fpgadesigntakehome@hudson-trading.com`.

Compilation, Simulation and Debug

Once the installation is completed, you can compile and run the simulation by invoking `make sim`. This command will compile the design and stop if there are any syntax errors. If the compilation is successful, the simulation will run saving the log into `sim.log` and the waves into `sim.vcd` files. Run `gtkwave sim.vcd` to visually inspect the waves. Feel free to read and modify the testbench while you are debugging but revert any changes to make sure you design also runs with default settings.

Synthesis

Once the tests are passing, it is time to synthesize the design and investigate resource utilization. Invoke `yosys` and read in your design file:

```
yosys> read_verilog StreamingHistogram.v
```

This step should take no more than a minute. If the tool seems unresponsive, decrease the parameter values, try again, and investigate potential scaling issues in your solution.

If the design is read successfully, it is time to map to Xilinx FPGA primitives:

```
yosys> synth_xilinx
```

Similarly, if the synthesis step is taking more then a few minutes it means there is a scaling problem with the design. If successful, you will see statistics about resource utilization (the actual numbers are omitted and the output is trimmed below).

```
2.22. Printing statistics.

=== StreamingHistogram ===

   Number of wires:                 ???
   Number of wire bits:             ???
   Number of public wires:          ???
   Number of public wire bits:      ???
   Number of memories:              ???
   Number of memory bits:           ???
   Number of processes:             ???
   Number of cells:                 ???
     FDRE                           ???
     LUT1                           ???
     LUT2                           ???
     ...                            ...
```

Do these utilization numbers seem correct and efficient? Look especially carefully at stateful cells such as FDRE flip-flops. How are you storing the counts? Is there a better approach? Consider the types and number of available resources on a KU5P device:
https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf

The above steps can also be performed via invoking `make synth` target.

`make synth` step can be run to quickly repeat these manual synthesis steps. The command includes a one-minute timeout to break out of a optimization loop of an inefficient implementation that renders the synthesis tools unresponsive. The command also writes out the gate-level netlist.

Gate-level simulation

The output of the synthesis step is a gate-level netlist. In order to make sure that the design was synthesized correctly, we need to rerun the exact same stimulus we applied to the behavioral HDL and observe that the design still operates correctly. This step may expose issues with initialization, conflicting or missing assignments leading to logic being optimized out, etc. Run `make gatesim` step to invoke gate-level simulation. The first time you run this step, the makefile will checkout Xilinx Unisims libraries from GitHub, which contain simulation models for the gate-level cells present in the netlist.

Gate-level netlist may contain many internal signals, and dumping these internal signals can create huge VCD files. Therefore the `gatesim` target only dumps the top-level testbench signals by passing +VCDLEVEL=1 plusarg option to the gatesim binary. You can increase the value to 2 or more if you need to observe internal netlist signals.