22T2 COMP4601 Chapter 5 Lab/Week 8 Hand-in exercises 30/06/2022
(Notes and questions on PP4FPGAs – Chapter 5 exercises)

In this lab you will study and accelerate three versions of a 1024-point FFT. First you will optimize typical FFT software code for an FPGA implementation. Next, you'll compare its performance with that of a dataflow design. And finally, a streaming version will be implemented. The goal is to achieve a 100X improvement in the task interval compared with the unoptimized hardware implementation of the software code.

For this lab, you are encouraged to complete Chapter 7 of the 2020.1 Vivado HLS Tutorial and to carry out exercises relating to Ch. 5 of the text. Your answers to indicated exercises should be submitted electronically by **5:00 pm Monday 18 July**. This hand-in is worth 10% of your mark in the course.

Completing the exercises below should take 3 – 4 hours after you have completed Chs 2 – 7 of the Vivado HLS Tutorial. The files needed for this lab are contained in the `ch5.zip` zipfile attached to the Week 6 & 7 lab section of the course website.

Since the designs you will be working on in this lab are larger than those used in previous labs, you will need to target a larger Ultrascale+ device. If you did not install the Zynq Ultrascale + MPSoC devices at step B.3 when you setup your lab environment in Week 1, then follow the instructions for downloading and installing the Zynq Ultrascale+ MPSoC devices provided in the *Vivado device installation guide* attached to Week 6 & 7 lab section of the course website before you commence the exercises below.

FFT implementation exercises

Code for the lab is contained in three file sets: `fft_sw`, `fft_stages` and `fft_stages_loop`, corresponding to software, dataflow and streaming implementations of the FFT. Each file set comprises a Tcl command script to create a project for the implementation, C++ code describing the FFT algorithm, and a C header file.

In the following exercises, unless instructed to do otherwise, place all directives in the directives script for the current solution.

Implementing the software code in hardware

solution1

> Run `vivado_hls -f fft_sw.tcl` from the Vivado HLS command prompt to create your first implementation. Open the synthesized baseline design using the command `vivado_hls -p fft_sw.proj`
>
> Describe the performance and resource utilization of this baseline before adding any directives. Note that the latency is not reported because the compiler was unable to determine how many iterations of the `dft_loop` and `butterfly_loop` need to

be executed. Refer to the grey text box at the top of page 115 in the text for an explanation of this phenomenon.

The first exercise is to obtain a reasonably precise estimate of the task latency.

solution2

Create a new solution from ***solution1***.

Apply the **LOOP_TRIPCOUNT** directive to the `dft_loop` and `butterfly_loop` to obtain a reasonably precise estimate of the task latency. You may find it useful to try using a median rather than mean iteration tripcount for each loop. It is desirable to reduce the difference between min & max latency to below 1.5%.

What do you estimate the latency to be? How did you obtain your estimate?
Has the resource utilization changed as a consequence of adding the directives?
If so, by how much?

Next, we will try reducing the latency of the `bit_reverse` function.

solution3

Create a new solution from ***solution2***.

Compare the performance and utilization of alternatively (1) unrolling the `reverse_bits_loop`, (2) pipelining the `reverse_bits_loop`, or (3) pipelining the function `reverse_bits`. The synthesis results for each solution can be found by navigating to *solution3* → *syn* → *report* → *bit_reverse_synth.rpt* from the Explorer pane after you have provided the necessary directives and synthesized the design.

Can the performance/utilization of function `bit_reverse` be improved beyond that achieved by unrolling `reverse_bits_loop`?

solution4

Create a new solution from ***solution3***, which has unrolled the `reverse_bits_loop`.

Pipeline the `dft_loop` and run synthesis.

You will notice an II violation that prevents an II=1 from being achieved. Explain how this violation comes about. Why is the loop-carried dependency a false (invalid) dependency?

Next, you'll try to eliminate the false dependency.

solution5

Create a new solution from ***solution4***.

Refer to pages 137 – 138 of the 2020.1 Vivado Design Suite User Guide on High-Level Synthesis (UG902) linked to the General Guides section of the Labs page of the course website for an explanation on how to use the **DEPENDENCE** directive to remove false loop-carried dependencies so as to improve loop pipelining.

What is the minimum II and iteration latency you are able to achieve for the `dft_loop`? How did you achieve this result?

Is it worthwhile optimizing the outer loops of function `fft`? What constraints are there on improving the performance any further?

solution6

Create a new solution from ***solution5***.

Edit `fft_sw.h` to change the data type of `DTYPE` from `float` to `ap_fixed<22,11>`. Does this improve performance/resource utilization any further?

Please include a copy of your `directives.tcl` file as well as the following sections of your synthesis report for ***solution6*** in your report:

Performance Estimates – Timing and Latency (Summary & Details of Instances & Loops);
Utilization Estimates – Summary

Implementing the dataflow code in hardware

solution1

Run `vivado_hls -f fft_stages.tcl` from the Vivado HLS command prompt to create the project and *solution1*. Open the synthesized baseline dataflow design using the command `vivado_hls -p fft_stages.proj`

Comment on the performance and resource utilization of this baseline design before adding any directives.

solution2

Create a new solution from ***solution1***.

Apply the directives you used in ***fft_sw/solution6*** and compare the performance and resource utilization of *fft_stages/solution2* with *fft_sw/solution6*. Comment on any problematic aspects of the *fft_stages/solution2* results.

Please include a copy of your `directives.tcl` file as well as the following sections of your synthesis report for ***fft_stages/solution2*** in your report:

Performance Estimates – Timing and Latency (Summary & Details of Instances & Loops);
Utilization Estimates – Summary

Implementing the streaming code in hardware

solution1

Run `vivado_hls -f fft_stages_loop.tcl` from the Vivado HLS command prompt to create the project and first HLS solution. Open the synthesized baseline streaming design using the command `vivado_hls -p fft_stages_loop.proj`

Comment on and compare the performance and resource utilization of this baseline design with that of *fft_stages/solution2*. Explain the improvement in performance. What is the task latency and the task interval reported by the tools?

solution2

Create a new solution from ***solution1***.

The goal of this exercise is to improve the task interval and latency by pipelining the `bit_reverse_loop`.

If you were able to improve the performance, what measures did you take? What impact did your changes have on resource utilization?

Overall, what improvement did you measure in task latency and interval between *fft_sw/solution2* and *fft_stages_loop/solution2*?

How would you assess the performance of the *fft_stages_loop/solution2* implementation, were it implemented as predicted on the specified device, relative to what you think could be achieved with current general-purpose processors? What factors are you ignoring in your assessment?

Please include a copy of your `directives.tcl` file as well as the following sections of your synthesis report for ***fft_stages_loop/solution2*** in your report:

Performance Estimates – Timing and Latency (Summary & Details of Instances & Loops);
Utilization Estimates – Summary