University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Fall 2019

HW5: Accelerator

Wednesday, September 25

Due: Friday, October 4, 5:00PM

In this assignment, we will accelerate an application by implementing functions on the programmable fabric. You can find the sources for this homework the course website.

This assignment is longer than the previous week and involves CAD tool tasks that take a long time. We strongly recommend that you *start early* on this assignment.

Note: Lectures for C and pragmas don't occur until the week the assignment is due. You aren't writing new code here, so we expect you can begin working through this and things will become clearer during the week.

Collaboration

In this assignment, you work with partners that we assigned. You can find the assignment on Canvas in the *Partners* map under the *Files* section. In the event that the partner assignment does not work out, contact the instructor or TA as soon as possible. Partners may share code and results and discuss analysis, but each writeup should be prepared independently. Outside the assigned groups, only sharing of tool knowledge is allowed. See the course policies on the course web page `http://www.seas.upenn.edu/~ese532` for full details of our policies for this course.

Hardware Acceleration

To implement a hardware function, it will ultimately be necessary to perform low-level placement and routing of the hardware onto the FPGA substrate. That is, SDSoC must decide which particular instance of each primitive is used (placement) or which wires to use for connections (routing). These tasks are typically much slower (20 minutes to an hour) than the compilation time for software (a few minutes). This means you will need to plan your time carefully for this lab and for subsequent labs. One way to optimize our development time is to be careful about when we invoke low-level placement and routing and when we can avoid it. This lab and next will show you a few techniques that allow you to reduce the number of times you need to invoke low-level placement and routing and introduce simulation and emulation you can use validate your design before invoking low-level placement and routing.

Ideally, you can tell SDSoC which function to implement in hardware, add a few pragmas to your code, and SDSoC does the rest. In practice, you will often discover that existing functions cannot be mapped directly on the hardware because they are not synthesizable or because their direct implementation is inefficient. In those cases, you will need to rewrite your code. You can iteratively improve your design in SDSoC, but our experience is that the errors and warnings in SDSoC are often unclear or hidden. Moreover, a debug cycle typically takes longer. SDSoC uses Vivado HLS under the covers. Ideally, SDSoC would hide the need to use Vivado HLS independently, but it also hides useful errors and warnings. Hence, we start the acceleration of a function in Vivado HLS, which has better debug facilities. Once your function has been verified and synthesized successfully in Vivado HLS, you can copy it to SDSoC and integrate it into the system. Vivado HLS is also based on Eclipse, so most of the GUI should be familiar.

Creating a new project in Vivado HLS is explained here. Make sure you enter the top-level function during the creation of the project (although you can also change it later). The top-level function is the function that will be called by the part of your application that runs in software. Vivado HLS needs it for synthesis. You can also indicate which files you want to create. It is wise to add a testbench file too, while you are creating the project.

Although you can and should also verify your system in SDSoC like before, you will generally need more time to build your project since it will also automatically invoke placement and routing; consequently, we have provided a testbench in Vivado HLS to debug the hardware. The requirements for testbenches are not any different from other software applications written in C. Similar to them, testbenches have a `main` function that is invoked. To the main function you can add any functionality needed to test your function. That includes calling the top function that you would like to test. When the testbench is satisfied that the function is correct, it should return 0. Otherwise, it should return another value.

You can run the testbench by selecting *Project → Run C Simulation* from the menu. A window should pop up. The default settings of the dialog should be fine. You can dismiss the dialog by pressing *OK*. You can see in the *Console* whether your test has passed. If your test fails, you can run the test in debug mode. This can be done by repeating the same procedure, except that you should check the box in front of *Launch Debugger* this time before you dismiss the dialog. This will take you to the *Debug* perspective, which should look familiar by now. You can go back to the original perspective by pressing the *Synthesis* button in the top, right corner. Note that in SDSoC you may be used to the fact that it builds your project automatically when you change your code and start or relaunch a debug session. For Vivado HLS, it is not the case. To rebuild the code, you should go back to Synthesis mode, and click *Run C Simulation* again to rebuild the code.

Once you are satisfied with your code, you can run *Solution → Run C Synthesis → Active Solution* from the menu to synthesize your design. You can also verify the synthesized version of your accelerator in your testbench. If you choose to do so, Vivado HLS will run your accelerator in a simulator, so this method is called C/RTL Cosimulation. The employed cycle-level simulation is much slower than realtime execution, so this method may not be practical for every testbench. It avoids needing to run low level-placement and routing and will give you more visibility into the behavior of your design. Anyway, you can start it by choosing *Solution → Run C/RTL Cosimulation* from the menu.

The hardware implementation that Vivado HLS selects can be controlled by including pragmas such as `#pragma HLS inline` in your code. The different pragmas that you can use in your functions are listed in the sections for the associated TCL commands in the Vivado HLS manual. If you need information about the `inline` pragma, you can look up the `set_directive_inline` command for example. While you could also use TCL commands, we do not recommend that because they cannot be imported into SDSoC as easily as the pragmas in a C file.

When you have obtained a satisfying hardware description in Vivado HLS, you can import the same source into a new `SDSoC project`. Chapter 2 of the SDSoC tutorial explains how you can create a project and select the function that must be accelerated.

The builds in Question 4 will take 15–40 minutes. You may want to plan your work to start the first two builds there while you are working through Questions 1–3. The final step in Question 3 will also be a long build.

Homework Submission

1. **Initial implementation**

    (a) Report the latency of the matrix multiplier in `Multiply_SW` on the ARM core without hardware acceleration. This is our baseline. Make sure you set the optimization level of the SDS++ compiler to `-O3`. (1 line)

    (b) Simulate the matrix multiplier in Vivado HLS. Start with launching Vivado HLS. Afterwards, create a new project and add `MatrixMultiplication.cpp` as source file and `Testbench.c` as testbench. Specify `Multiply_HW` as top function. Select the xczu3eg-sbva484-1-e in the device selection. Use a 5 ns clock, launch a C simulation, and verify that the test passes in the console. Include the console output in your report.

    (c) Look at the testbench. How does the testbench verify that the code is correct? (3 lines)
    (We provide you a testbench here. As you develop your own components for the project, you will need to develop your own testbenches. Our testbenches can serve as an example and template for you.)

    (d) Synthesize the matrix multiplier in Vivado HLS. Analyze the *Synthesis Report*. What is the expected latency of the hardware accelerator? (1 line)

    (e) How many resources of each type (BlockRAM, DSP unit, flip-flop, and LUT) does the implementation consume? (4 lines)

    (f) Analyze how the computations are scheduled in time. You can see this information in the *Performance* view of the *Analysis* perspective. How many cycles does a multiplication take? (1 line)

    (g) Make a schematic drawing of the hardware implementation consisting of the data path and state machine similar to Figure 1-2 of the Vivado HLS manual. You can ignore the addressing and loop hardware (such as `phi_mux` and `icmp`) in your data path.

(h) Explain why the performance of this accelerator is worse than the software implementation. (3 lines)

2. **Loop unrolling**

   (a) Go back to the *Synthesis* perspective, and unroll the loop with label `Main_loop_k` 2 times using an `unroll` pragma (See p. 36 of the SDSoC Programmers Guide for example of unroll pragma). Synthesize the code and look again at the schedule. Explain how the schedule for the unrolled loop is able to reduce the latency of the entire loop evaluation (all iterations) compared to the original (non-unrolled) loop. (3-4 lines).
   Hint: What characteristic of the original code prevented this optimization? and why is the unrolled loop able to exploit more parallelism?

   (b) We could also have unrolled the loop manually. What would the equivalent C code look like?

   (c) Inspect the resource usage over time in the *Resource* view of the *Analysis* perspective. Of the computational resources (`fmul` and `fadd`) which one(s) are shared by multiple operations? (1 line)

   (d) Unroll the loop with label `Main_loop_k` completely, and synthesize the design again. You will likely notice that the estimated clock period in the *Synthesis Report* is shown in red.[1] What does this mean? (3 lines)

   (e) Change the clock period to 20 ns, and synthesize it again. What is the expected latency of the new accelerator? (1 line)

   (f) How many resources of each type (BlockRAM, DSP unit, flip-flop, and LUT) does this implementation consume? (4 lines)

   (g) You may have noticed that all floating-point additions are scheduled in series. What does this imply about floating-point additions? (2 lines)

   (h) We want to multiply two streams of matrices with each other. We can fill the FPGA with copies of one of the accelerators from question 1d or 2d. Which accelerator would you choose for the highest throughput?

[1]Due to variation among Vivado HLS versions, sometimes it works and nothing is flagged. The intent of this question is to illustrate things you may encounter and (with the following questions) show you how to address them. If it's not flagged in red, just report the estimated clock period.

3. **Pipelining**

   (a) Remove the unroll pragma, and pipeline the `Main_loop_j` loop with the minimal initiation interval (II) of 1 using the `pipeline` pragma. Restore the clock period to 5 ns. Synthesize the design again. Report the initiation interval that the design achieved. (You may find the timing is still not met. It does not matter, we will fix it later.) (1 line)

   (b) Draw a schematic for the data path of `Main_loop_j` and show how it is connected to the memories. You can find the variables that are mapped onto memories in the *Synthesis Report*.

   (c) Assuming a continuous flow of input data, how many data words does the pipelined loop need per clock cycle from `Buffer_1`? (1 line)

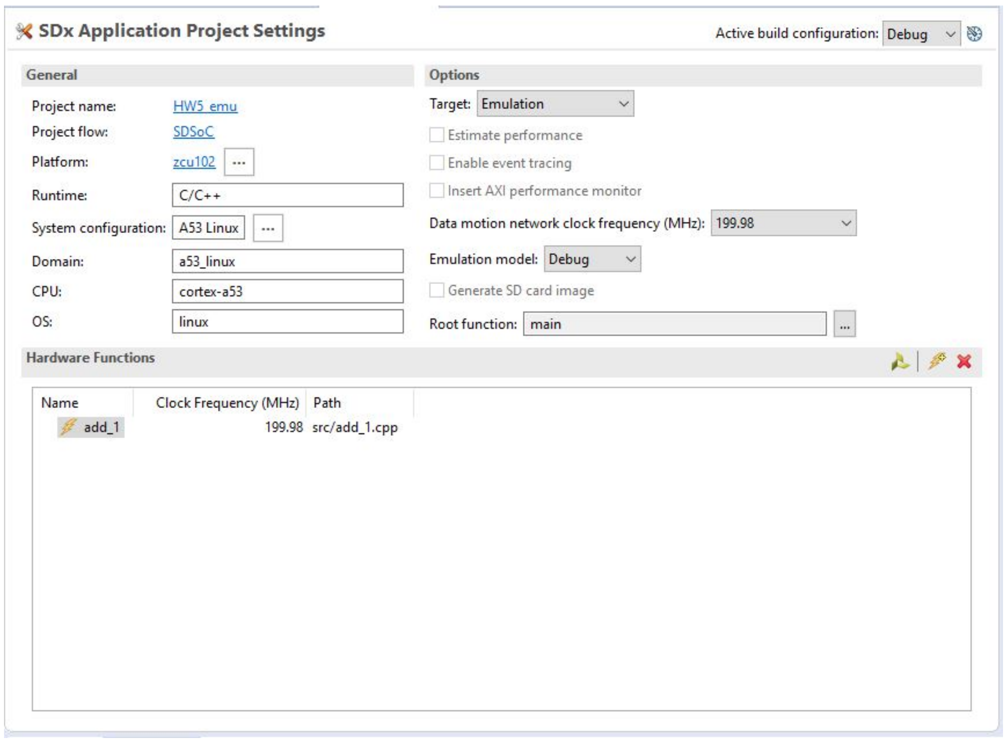
   (d) Considering what you found in the two previous questions, why does the tool not achieve an initiation interval of 1? (3 lines)

   (e) We can partition `Buffer_1` and `Buffer_2` to achieve a better performance. Illustrate the best way to divide each of the arrays with a picture that shows how the elements of these arrays are accessed by one iteration of the pipelined loop.

   (f) Partition the buffers according to your description in the previous question with the `array_partition` pragma. (See "Array Configuration" Section on p. 30 of the SDSoC Programmers Guide for examples of array partitioning pragma). Synthesize the design and report the expected latency. (1 line)

   (g) How many resources of each type (BlockRAM, DSP unit, flip-flop, and LUT) does this implementation consume? (4 lines)

   (h) Pipeline the `Init_loop_j` loop also with an II of 1. Save your design and quit Vivado HLS. Launch SDSoC and create a new *SDSoC project*. Import the sources that you optimized in Vivado HLS. Add `Multiply_HW` as hardware function in the project overview. Build the design and run it on the Ultra96. This build will take 30–40 minutes. To make best use of the high speed fabric in Ultra96, we developed our own SDSoC platform for our course. You can download here, and install the platform as we mentioned in HW1. Make sure you set the optimization level of the SDS++ compiler to `-O3` and set *Data motion network clock frequency* and *function clock frequency* to 200MHz. Note that this process is much slower than a synthesis in Vivado HLS because Vivado only translates the design to a lower-level hardware description language, but it does not perform low-level placement and routing. What is the speedup that the accelerated design achieves? (2 lines)



SDx Application Project Settings

Active build configuration: Debug

General

Project name:	HW5_emu
Project flow:	SDSoC
Platform:	zcu102
Runtime:	C/C++
System configuration:	A53 Linux
Domain:	a53_linux
CPU:	cortex-a53
OS:	linux

Options

Target: Emulation

☐ Estimate performance

☐ Enable event tracing

☐ Insert AXI performance monitor

Data motion network clock frequency (MHz): 199.98

Emulation model: Debug

☐ Generate SD card image

Root function: main

Hardware Functions

Name	Clock Frequency (MHz)	Path
add_1	199.98	src/add_1.cpp

Figure 1: Emulation Configuration for ZCU102

4. **SDSoC Emulation**

   System emulation provides the same level of accuracy as the final implementation without the need to compile the system into bitstream. It is heavily used in hardware and software co-debugging. You can read SDSoC Environment Debugging Guide for details.

   (a) Create an SDx Ultra96 project based on the source code here. Move function *add_1* into hardware. This build will take 30–40 minutes. Can you run the project successfully?

   (b) As Ultra96 emulation is not supported by SDx, we will use ZCU102 to do the emulation. Create another project with the same code as the previous question (Q 4a). Choose *ZCU102* as the platform. Choose *A53 Linux* as the System Configuration. Configure the project as shown in Figure 1.

   (c) Building the project. Launch the emulation by choosing *Xilinx*→ *Start/Stop Emulator*. Choose the right project name and click *Start*. This build will take 10–15 minutes, but may take longer the first time it is run. When the *VIVADO* software starts, choose the signals you are interested in. Click *run* in *VIVADO*. It will take a while for *SDx* to set up the environment. When you see 'Starting tcf-agent: OK' in *SDx* Emulation Console, right click the project in *Project Explorer*. Choose *Debug As* → *Launch on Emulator*. Can you figure out why the program cannot run to the end?
   Hint: Pay attention to DMA signal *m_axis_mm2s_tlast*.

(d) Go back to the Ultra96 project. Report how you fix the bug to run the project without hanging. This build will also take 30–40 minutes.

5. **Reflection**

   (a) Problems 1–3 in this assignment took you through a specific optimization sequence for this task. Describe the optmization sequence in terms of identification and reduction of bottlenecks. (4 lines)

   (b) Make an area-time plot for the three designs with a curve for DSPs and BlockRAMs.
   Hint: Run time on X-axis, DSPs and BlockRAMs as separate curves on the Y-axis.