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

ESE532, Fall 2019 | HW6: Accelerator Interface | Wednesday, October 9

Due: Friday, October 18, 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 on the course website.

Be warned that this homework requires a number of full SDSoC builds, each of which can easily take 20–30 minutes, so begin on time and plan your schedule accordingly. Question 1 requires 5 builds and Question 3 requires 5 builds. Question 2 has no builds, so we suggest starting with it once you have completed Question 1b (you will need the full project information that comes from the question 1b build), such that you can work while other builds are running. The builds in Question 1 do not depend on each other, so they can be run concurrently.

The SDSoC Programmer's Guide is a generally useful resource.

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.

Homework Submission

1. **Accelerator interface** In this question, we will analyze various ways in which a processor core can communicate with an accelerator. We tell you some specific things to experiment with, but you should also read the Xilinx documentation on what these pragmas mean.

   Read pages 7–9 (Execution Model of an SDSoC Application), pages 14–15 (Allocating Memory), and pages 23–27 ("Hardware Function Argument Types" through the end of "Data Movers") of the SDSoC Programmer's Guide to get an overview of function calls and data movement to accelerators.

(a) Create an SDSoC project. Add the provided source files in the `MatMulLoop` directory of the archive to the project. Set the optimization level of the SDS++ compiler to `-O3`. Report the latency of the software version function `mmult_golden`. (1 line)

For Parts 1b–1e use the event tracing functionality of SDSoC to understand the performance difference between the implementations. We ask for the explanation in Part 1f.

To collect a trace, enable *Enable event tracing* in the project overview, rebuild your application, right-click on your project, and choose *Debug As*→*Trace Application (SDSoC Debugger)*. Include pictures of the trace in your report. Screenshots are fine for this question, but make sure to crop them to the area of interest.

(b) Measure the duration of `mmult_accel` when data is transferred to and from the accelerator using DMA, as follows: Set `mmult_accel` as hardware function. Insert a `data_mover` pragma immediately before `mmult_accel` in the code. You can check whether the pragma was successfully applied by looking at the *Data Motion Network Report* in `Assistant` view. Report the latency of `mmult_accel` in clock cycles. (1 line)

Hints:

- Note that the code uses `sds_alloc` instead of `malloc` as discussed in the SDSoC Programmer's Guide.
- Make sure you specific the data range with fixed constants (e.g. use the macro `MATRIX_WIDTH`) in the data copy pragma.

(c) Measure the duration of `mmult_accel` when the data transfer is variable length. For the extent of the data in the data copy pragma, use a C program variable rather than a constant.

i. Report latency (1 line)

ii. Note that the compiler chooses a different DMA mode than in the previous case. Identify the difference. (1 line)

iii. Where does the latency difference with Problem 1b show up?
(Hint: see the *Data Motion Network Report* in the `Assistant` view mentioned in Problem 1b.)

(d) Returning to fixed-length data transfer, measure the duration of `mmult_accel` when data is retrieved and stored in a shared region of DRAM (shared memory). Make a new project identical to the previous one, except for replacing the `data_mover` pragma with a `zero_copy` pragma that is applied to all parameters. (1 line)

(e) Add async/wait pragmas to overlap communication and computation for the data copy case and measure the duration of `mmult_accel` for varoius pipeline depths.

i. Read Pages 34–36 of the SDx Pragma Reference Guide.

ii. Move `mmult_accel` into hardware.

iii. Report the speedup as a function of pipeline depth from 1 to 5.
When you just change the pipeline depth, it should not require a full rebuild of the hardware.

(f) Explain the performance difference among Parts 1b–1e. Your explanation should cover both what is happening and why it is happening. (3-5 lines)
The *Data Motion Network Report* may be useful.

(g) Under which circumstances do you expect DMA to perform better than shared memory? Motivate your answer. (5 lines)

(h) Use the `SDS resource` pragma to run 2 `mmult_accel` instances in parallel. Include your modified code in the report. Report the latency and speedup.
Hints:

- Change the definition of CHUNKS from 1 to 8.
- You will need to use async/wait as in Part 1e.
- Read Pages 47–48 of the SDx Pragma Reference Guide.
- Also useful may be Pragma SDS resource.
- Make sure you specify data is physically contiguous and accessed sequentially.

2. **Analyze implementation**

In this question, we will investigate what the FPGA implementation of the matrix multiplication with DMA looks like using Vivado (not Vivado HLS). Vivado is part of the SDx installation.

(a) Report how many resources of each type (BlockRAM, DSP unit, flip-flop, and LUT) the implementation with DMA (Q 1b) consumes (you will need access to the build from Q 1b). You can find this information in the *Project Summary* of Vivado. Launch Vivado and open the project at the location `Debug/_sds/p0/vivado/prj.xpr` relative to the root directory of your SDSoC project with simple DMA interface to see this view. (4 lines)

(b) Report the expected power consumption of this design from the *Project Summary*. (1 line)

(c) Open the block design by selecting *Open Block Design* in the *Flow Navigator* on the left side of the main window. The block labeled `mmult_accel_1` is the accelerator. The logic that is not programmable (PS) is in the `ps7` block. The DMA controllers of the three accelerator ports are in `dm_0`, `dm_1`, and `dm_2`. The block named `mmult_accel_1_if` is a wrapper around the accelerator that contain, among other stuff, the BlockRAMs that data movers, such as simple DMA, write their data to. The blocks with label `AXI interconnect` are crossbars that may have buffers and converters if different types of buses are connected. Many data buses on the programmable fabric are either AXI4, AXI4-Lite, or AXI4 Stream buses. Which bus is used between each of the DMA controllers and the PS to transfer data (not control signals)? As the blocks and buses in Vivado are not very descriptive, you will probably have to dig a bit in the documentation of one of the blocks. To open the documentation for a particular block, you can double-click a block and press the *Documentation* button if the documentation was installed. Otherwise, you can generally find documentation on the internet by entering the name and version number of the block. (1 line)

(d) Open the *Address Editor* by choosing the corresponding tab above the block design. In which memory region is the accelerator wrapper (`mmult_accel_1_if`) mapped? This region is used for such communication as starting the accelerator and querying its status. Writes and reads by the ARM processor are to this region are sent over an AXI4-Lite bus to the accelerator wrapper, which handles them and controls the accelerator. (1 line)

(e) Open the timing report by returning to the *Project Summary* and pressing *Implemented Timing Report*. Click on the number next to `Worst Negative Slack`. Look at the *Path Properties*. Report in which of the hardware modules that we saw in the block design the path begins and ends. (1 line)

(f) Include a screenshot of the critical path in your writeup. Zoom in to make sure all elements of the path are clearly visible. Indicate the type of each element (e.g. LUT, flip-flop, carry chain) on the screenshot.

(g) Highlight the accelerators in green, the DMA controllers in red, and the interconnect (`M_AXI_HPM0_FPD`, `S_AXI_HP0_FPD`, `S_AXI_HP1_FPD`, and `AXI_HP2_FPD`) in yellow. You can do this by right- clicking the modules in the netlist view and selecting *Highlight Leaf Cells*. Include a screenshot of the entire device in your report.

3. **Streaming, serial, and parallel**

A brilliant engineer (but not in cryptography), inspired by the encryption example of HW1, came up with a novel way to encrypt messages. He believes that his algorithm will perform well on FPGAs because it has a lot of fine-grained parallelism. In this question, we will map his implementation on the hardware.

Read pages 16–18 (Sequential and Parallel Accelerator Execution) in SDSoC Programmer's Guide to get an overview.

(a) Create a new Vivado HLS project and add the sources of the `Encrypt` folder in the provided archive. Use a clock period of 7 ns. Map the `Encrypt_HW` function on the hardware. When you build the design, you will encounter a problem. Explain why this property of the code is problematic for hardware acceleration. (3 lines) Hint: You can set the data mover network clock and accelerator clocks independently.

(b) Solve the problem by changing the function declaration such that it explicitly deals with the worst case. Include the relevant code in your report.

(c) Add pragmas such that the accelerator can process at least one 32-bit word per cycle. Include the relevant sections of the code in your report.

(d) Create a new SDSoC project, and add the sources of your optimized application. Set the optimization level of the *SDSCC compiler* to `-O3`. Map `Encrypt_HW` to the hardware. Build the accelerator. Which limitation of the FPGA is at the root of the next problem that you encounter? (3 lines)

(e) Add a suitable pragma to inform SDx that we are accessing both the input and output buffers sequentially. Show the pragma that you added.
Note: Don't build and evaluate yet—that's step 3h.

(f) How does this pragma solve the problem? Consult the manuals as needed. (3 lines)

(g) Add an `SDS data copy` pragma to inform the SDSoC about the actual length of the data that is passed to and from the accelerator. This avoids the problem that more data is transferred than necessary. Show the pragma that you added.
Note: Build and evaluate is next step.

(h) Build the application and report the latency of the hardware implementation and the speedup with respect to the software implementation. (1 line)

(i) The engineer believes that he can obtain better encryption by putting two instances of the encryption module in series, using a different key for each of them. He predicts that the latency will be twice as long as for a single encryption module. Create a project with this new configuration and report the latency. (1 line)

(j) Explain why the speedup differs from what the engineer expected. You may need Vivado to investigate the issue. (3 lines)