Accelerating Systems with Programmable Logic Components – 1DT109

2021 – HT1-HT2

Lab 2 – Designing an AXI Accelerator for Matrix Multiplication

Overview

In this lab you will be designing an AXI accelerator for matrix multiplication for the Xilinx Zynq device. As in the previous lab, we will only be using the **simulator**.

Tools

Xilinx Vivado HLx Edition 2018.3

Intended Learning outcomes

- Understanding the design of a custom IP (Intellectual Property).
- Understanding the basics of accelerator design.
- Understanding the AXI bus handshake logic.

Assessment

- Your design needs to produce the expected result using the provided test bench.
- You must demonstrate your solution online, and explain your design to the lab assistants in one of the office hours (every Friday 15:15 – 16:00) before the Lab2 deadline
    - Friday, October 8th, 15:15 – 16:00,

  In order to not get overloaded, we recommend that you start early, make good use the lecture recordings, and only leave the demonstration to the last lab session if possible (you can present at an earlier session as well).
- Both students must understand all parts of the solution by themselves (this will be checked), and both students must be present during the demonstration. If you cannot both be present during any of the lab sessions for demonstration and have a valid reason, get in touch with the teaching assistants to book another time slot.
- Before submitting your solution, carefully test that it is working correctly, format your code properly and add comments where necessary. Note that the results for Lab 2 are one of the factors that will determine your course grade.

1 What to Accelerate

We consider matrix multiplication for acceleration, a basic mathematical operation that is important in a wide range of applications and algorithms (e.g., in scientific computing or machine learning). Matrix multiplication is a good candidate for acceleration, since the matrices needed in practice are often huge, and processing them can be a bottleneck on a general purpose computing platform (e.g., PC or server).

Given the matrix multiplication as follows,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{41} & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

an unoptimized sequential implementation of matrix multiplication is:

```
1.  for (row = 0; row < dim; ++row)
2.    for (col = 0; col < dim; ++col) {
3.      matR[row][col] = 0;
4.      for (tmp = 0; tmp < dim; ++tmp)
5.        matR[row][col] = matA[row][tmp] * matB[tmp][col] + matR[row][col]
6.    }
```

2 System Overview

Below you can see the overall architecture for a very simple accelerator system.

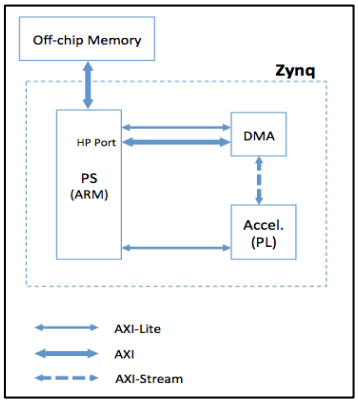


Figure 1 Zynq with accelerator

PS: Processing System (ARM Cores)

PL: Programmable Logic, i.e., the FPGA fabric

AXI-Lite: Simple AXI bus used by the processor to configure the IPs (memory-mapped)

AXI: High performance AXI bus (memory-mapped)

AXI-Stream: Streaming bus (address-less)

HP: High Performance port, used by the Direct Memory Access (DMA) controller to access the off-chip memory

In this lab, we will not be designing the whole system. Instead we will only deal with designing and testing the accelerator (shown as "Accel. (PL)" in Figure 1), with a focus on the AXI bus communication. We will provide a design skeleton and all you need to

do is to fulfil the TODO list in the skeleton code. However, it is very important to see the bigger picture (such as what components the accelerator has and their working detail), in order to understand the part that you will be working on. The following sections will guide you through some key components in the accelerator design.

3 Accelerator micro-architecture

The micro-architecture of the accelerator you will be working on (shown as "Accel. (PL)" in Figure 1) is given in Figure 2.

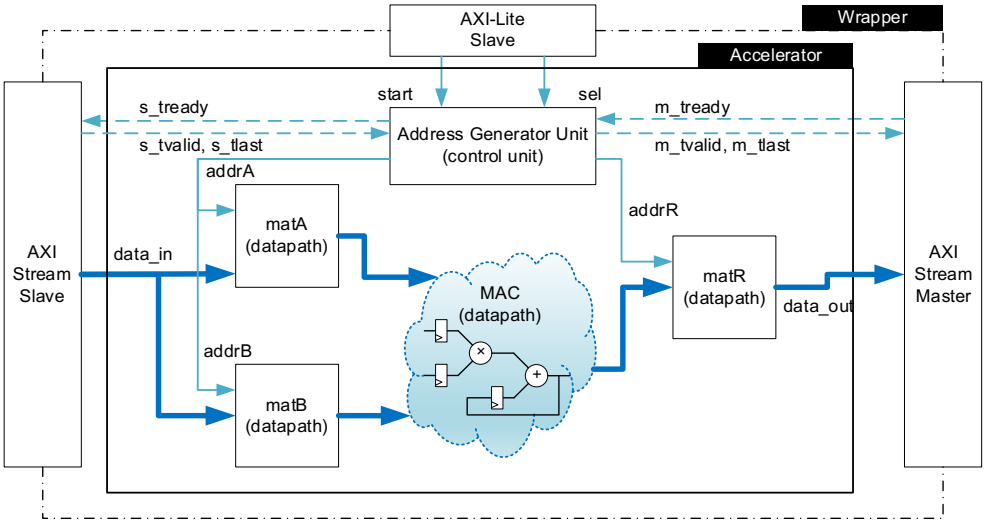


Figure 2 Accelerator micro-architecture

- **sel:** used by the PS to select matA or matB before loading the matrix to the accelerator (memory-mapped via the axi-lite config bus)
  - "0" for selecting matA
  - "1" for selecting matB
- **start:** used by the PS to start the accelerator after loading the matrices (memory-mapped via the axi-lite config bus)
- **AGU:** Address Generator Unit, used to generate addresses to:
  - Fill-in the input matrices matA and matB via axi-stream bus interface
  - Read operands from matA and matB during MAC
  - Write the result to matR
  - Sending the result to PS via axi-stream bus interface
- **matX:** input and output matrices implemented as dual-port block RAMs
- **MAC:** used to implement the datapath for the multiplication and accumulation

The provided skeleton file “mat_mul.v” implements this architecture, however, leaving the FSM in the “AGU” empty, for which you need to fill in. As you can see from the figure, the “AGU” FSM should implement the logic for reading from and writing to the AXI Stream bus (bus handshaking), as well as the controlling signals (e.g., addresses) for controlling datapath components (matA, matB, matR, MAC) to launch the actual operations for matrix multiplication.

The AGU FSM can have the following states:

- S_IDLE: The idle state. All controlling signals reset to default.
- S_LOAD_A: Load matrix A from the AXI stream bus. This state should generate controlling signals for the AXI stream slave interface in Figure 2 as well as addresses for matA during matrix loading.
    - Why should we generate address on the slave-side when loading matries using AXI-stream interface?
- S_LOAD_B: Load matrix B from the AXI stream bus. Again, this state should generate controlling signals for the AXI stream slave interface in Figure 2 as well as address for matB during matrix loading.
    - The AXI-stream is a streaming bus. In order to fully use that property, what kind of addresses should you generate in state S_LOAD_A and S_LOAD_B?
- S_CALCULATE: Launch matrix calculation. Read all operands loaded in matA and matB and write results to matR.
    - What kind of addresses should you generate in S_CALCULATE?
- S_OUTPUT: Output matR to the AXI stream master interface in Figure 2 after the whole matrix multiplication is done.
    - Why should we output matR only after the whole matrix multiplication is done? Why should not we output partial results in matR on the fly?

4 AXI-Stream State Machine

We will be using the following handshaking protocol to receive data from and send data to the AXI Stream bus, which should be implemented by the FSM inside the AGU.

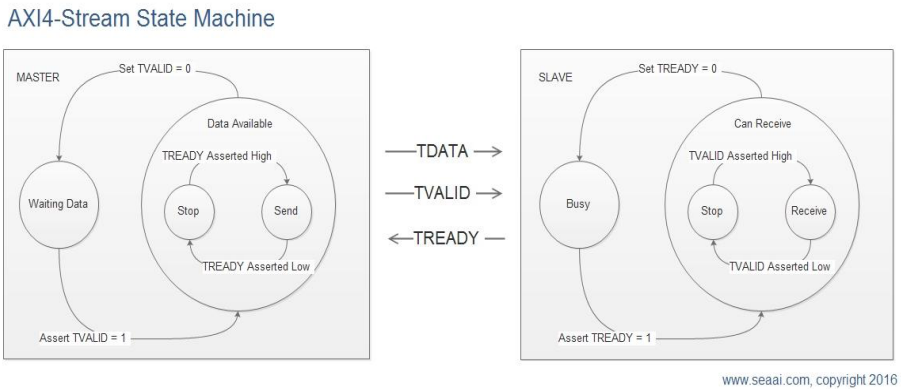


Figure 3 AXI stream state machine

You can read more on the AXI stream interface on Page 45 of the Xilinx guide "ug761_axi_reference_guide.pdf".

5 Address Generation for MAC

The AGU generates the addresses (the indices shown below) for reading from the input matrices and writing to the output matrix.

```
1.  for (row = 0; row < dim; ++row)
2.    for (col = 0; col < dim; ++col)  {
3.      matR[row][col] = 0;
4.      for (tmp = 0; tmp < dim; ++tmp)
5.        matR[row][col] = matA[row][tmp] * matB[tmp][col] + matR[row][col]
6.    }
```

N.B. In hardware, multi-dimentional matrices are always stored as 1-D arrays in memory. We thus must generate the addresses to index the 2-D matrices using 1-D addresses during matrix multiplication (i.e., from *matA[row][tmp]* to *matA[addr]*). One way to achieve this is to design an address generation logic as shown in Figure 4. Assume that matA and matB are both dim*dim in size with iterators "row", "col", and "temp" ranging in [0, dim-1]. The address generation logic for matA is given below.

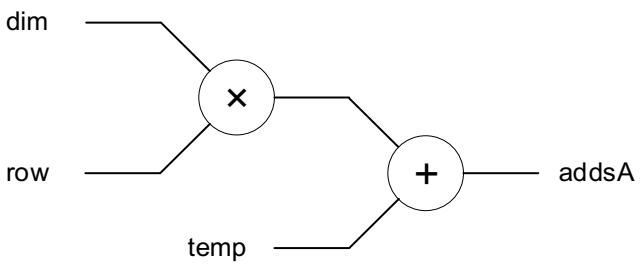


Figure 4 Address converting logic for matA

Observe the inputs and output of the logic and design the address generation logic for matB and matR in the code skeleton.

6 Multiplier and Accumulator (MAC) Unit

In the lab we use the Multiplier and Accumulator (MAC) unit to implement Line 5 in the matrix multiplication algorithm, which is one of the datapath components of the accelerator. A MAC has the following hardware structure. Understand its behaviour and implement it in the skeleton code. Think about why we need 3 registers in the MAC unit.

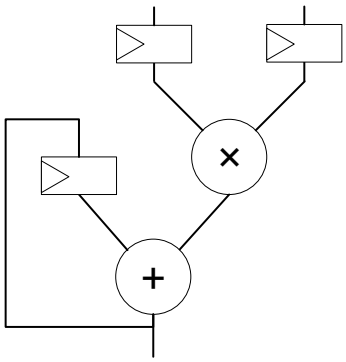


Figure 5 MAC unit structure

7 Your Task

You are required to design the accelerator based on the afore-mentioned components. Moreover, you should make sure that all the components are correctly verified and synchronized. This is achieved by designing synthesizable logic and ensuring correct timing between all the components using the testbench we ship.

You can open the design files in Vivado HLx and complete the design according to the instructions in this manual (you will need to complete all the TODO parts in the

skeleton files). We suggest not to change anything in the parts not annotated with a “TODO” comment, as only filling in these parts should be enough.

Finally, you need to benchmark your design. A necessary but not sufficient condition to pass this lab is that the provided test bench should produce the output "HW/SW result match!" when simulated to completion. It might not be sufficient, as there might be some edge cases which produce this output even though the design is not complete. It is your responsibility to check that the output is correct (e.g. through the simulator signals or writing the output to a file) even if you receive this message. To make debugging easier, you can reduce the matrix sizes by modifying the MATRIX_DIMENSION_LOG_2 value. The sizes need to be changed both in the test bench and in the accelerator code.

You are welcome to design your own test bench as well. Regardless, in addition to your solution, if asked you should be able to explain to the TAs:

- the test bench file that you are using (either your own or the provided one),
- how to interpret the results.

8 Hand-in

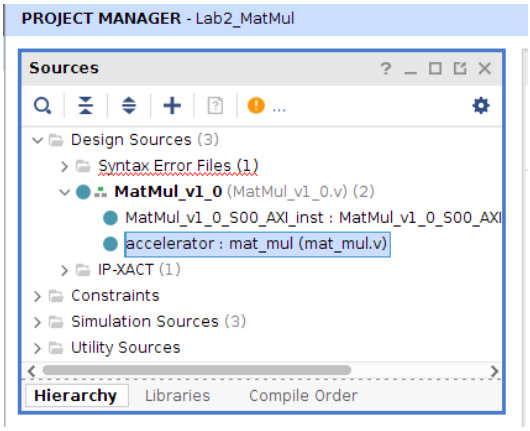
You must submit only the Verilog code for your solution, unarchived, on the Studium module “submission” (until midnight of **midnight, October 8th**). This is the file where you have filled in all the “TODO” parts located under lab2/src/hdl/mat_mul.v.

9 Opening the Skeleton Files in Vivado

After downloading the zip file and extracting it in a folder, open the project for the accelerator by opening in Vivado the following file:

lab2_matmul/Lab2_MatMul/Lab2_MatMul.xpr.

When the project is opened, double-click on **mat_mul.v** in the Sources pane, and complete the TODO parts (you can of course find and edit the same file under lab2_matmul/src/hdl using your preferred editor as well).



After completing your modifications, you should test your design by simulating it. The test bench is provided in the file mat_mul_tb.v.

N.B. If launching the behavioural simulation fails, make sure that the library "xil_defaultlib" is selected in the "Source File Properties" pane for all source files (see the figure on the next page).