ETIN35 - Digital IC-Project 1

VT-1 2024

v.1.0

Abstract

This document describes the assignment that needs to be completed to become familiar with the tools, which are required for an ASIC specific design flow. You will design a hardware accelerator that realizes a matrix operation, i.e., an input matrix is multiplied with a coefficient matrix. To reduce area cost, generic multipliers need to be re-used, achieved by a time-multiplexed architecture. The design needs to be evaluated with synthesis constraints on speed and area, using a 65 nm standard cell library. The matrix coefficients are available in a ROM, and the product needs to be stored in a RAM. The design needs to be verified with back-annotated timing information, and afterwards, physically placed and routed. Finally, a report needs to be submitted to get approved on the assignment.

The suggested timeplan for the projects:

- Jan 22^{nd}: Algorithm, ASMD (control part), block-diagrams, and interface definitions approved by another group (Deadline)
- Feb 1^{st}: RTL coding, RTL simulation
- Feb 8^{th}: ASIC Synthesis and netlist simulation (post-synthesis)
- Feb 22^{nd}: Physical placement and signal routing
- Feb 28^{th}: Design get approved by TAs (Deadline)

1 Introduction

In this project you will design a hardware accelerator which computes the product of a matrix multiplication. The matrix multiplier may be realized with various speed and area constraints. An efficient matrix multiplier is a trade-off between area and speed, i.e., the number of required clock cycles to compute the product vector. For instance, an area efficient matrix multiplier can have only one multiplier unit, to produce one product per clock cycle. For low throughput applications, such an architecture may be fast enough, however, in a system with a higher demand on throughput, a parallelized architecture is required.

The multipliers need to be implemented in a unit that performs a matrix multiplication. The architecture of this unit is optimized by resource sharing, i.e., a controller switches data in a time-multiplexed fashion. The multiplier coefficients are stored in a ROM (two 7-bit coefficients per address), and the result of the matrix multiplication needs to be stored in a RAM. A *ready* signal indicates when the computation is completed and the next input vector may be processed. All tasks need to get approved in time to be able to continue with the project part of the course.

2 Matrix Multiplication

A matrix multiplication is a computation expensive operation and thus often subject to hardware acceleration. The product of a matrix multiplication is specified as

$$P(n) = X(n)A, \tag{1}$$

where $P(n)$ is the product matrix, $X(n)$ is the input matrix of size 4x8 and A is the coefficient matrix of size 8x4. Matrix A has fixed coefficients and is specified as

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} \\ a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} \\ a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} \\ a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} \end{bmatrix}, \tag{2}$$

where the coefficients are of type *unsigned* with a wordlength of 7 bits. The coefficients of the matrix are stored in a ROM. The first element in P is computed as

$$p_{1,1} = x_{1,1}a_{1,1} + x_{1,2}a_{2,1} + x_{1,3}a_{3,1} + x_{1,4}a_{4,1} + x_{1,5}a_{5,1} + x_{1,6}a_{6,1}, + x_{1,7}(n)a_{7,1} + x_{1,8}(n)a_{8,1},$$

or in a generalized equation

$$p_{i,j} = \sum_{r=1}^{N} x_{i,r}a_{r,j}. \tag{3}$$

The first row of matrix P is computed by setting $i = 1$ and processing all the columns ($j = 1, 2, ...$) in (3). Similarly, compute the other rows of matrix P, and store these rows in a RAM.

3 Hardware Implementation

The arithmetic operation specified in (1) needs to be realized by a hardware accelerator. This is accomplished by the use of multiplier units (MU), which are governed by a controller. The MU accommodates ONE generic multiplier, i.e., to perform the multiplication with a column serially, and ONE adder that sum up the products, according to (3). You have the freedom to add registers. Two configurations are defined based on the number of MU to implement in the design, and each group will be assigned which configuration to use:

- Config. 1: two multiplier units (2 MU)
- Config. 2: four multiplier units (4 MU)

Before you trigger the hardware accelerator you need to store 32 input samples (4x8) in an *input register*, and then you raise the trigger signal *start*. The design inputs one matrix element per clock cycle, using an 8-bit input signal. The controller calculates the address of the ROM coefficients and switches the values from the ROM, as well as the input sample from the *input register* to the MU. Two matrix coefficients are read in one clock cycle from the ROM. Store the column in the RAM as soon as the value for **p** is computed. With the next clock cycle, you continue with the next column by taking the next coefficients and input samples. Thus, a design with one MU needs to be used 128 times for a complete matrix multiplication. Thereafter, the controller will indicate *finish* for one clock cycle, and the accelerator remains in idle mode until the next *start* signal. The input matrix X and the matrix coefficients have a wordlength of 8 and 7 bits, respectively. Both numbers are of type *unsigned*. The following signals need to be used:

- dataROM: provides the matrix coefficients
- address: address of the value to be written/read
- dataRAM: data that will be stored in the RAM
- web: enable signal for the RAM

3.1 RAM

The size of the RAM (160 Words $\times$ 32bits) determines the number of data elements which could be stored. The provided RAM has a number of memory locations which can be utilized, see the "ASIC design environment" in canvas for the specific RAM which are being used in this project. The number of $p_{i,j}(n)$ to be stored on one memory address is dependent on the number of bits defined for the output and the width of the RAM address. Note that the design has to be verified/simulated for more than 1 matrix continuously without asserting global reset.

3.2 ROM

In order to avoid over-complication of the design, instead of using a memory IP, the ROM should be defined as hard-coded memory elements (i.e., **registers**) in the design. The ROM coefficients are kept constant to have consistency among groups. Also, the ROM elements are stored column wise with two elements in one memory location.

4 Models and Verification

In the canvas, the files required for memory model and verification are provided. The memory model for the RAM is available in *mem* folder, along with the corresponding wrapper files in *mem_wrapper* folder.

A Matlab script (*generate_stimuli.m*) generates random inputs (X) which are written in the file *input_stimuli.txt*. For simulations, the student should read this stimuli file in testbench and send the data to the design.

For verification, the output results from the Matlab script *generate_stimuli.m*, which is written in *output_results.txt*, should match with the hardware implementation results.

5 Compulsory tasks

The assignment includes different tasks as part of the project plan. All tasks need to get completed before deadline in order to be able to continue with the next part of the course. All tasks need to be presented as print-outs, i.e., you need to do the drawings using a CAD tool (Visio or inkscape).

5.1 Task 1: HW-accelerator design

Before you start implementing your design in hardware you need to "design" on paper. A thorough preparation with paper and pencil will enhance understanding of the topic and ease the implementation. You need to analyze the requirements of the HW-accelerator. Thereafter, you need to specify an ASMD for the control part and you need to define the interface of your design. The outcome of this study will be the initial architecture. You need to use a behavioral model that produces the reference data for verifying your results. This task can be accomplished with Matlab, simply by adding some lines of code for comparing the output of the HW-accelerator and the expected output.

In summary, for Task 1 you are required to complete:

- Use the provided behavioural model (Matlab) to generate and verify data.
- Draw a block diagram showing your initial architecture.
- Sketch ASMD (according to Chapter 11 in "RTL Hardware Design Using VHDL", VLSI course book). Get it approved by another team.
- Describe interconnection signals between different function blocks of your design.
- Estimate throughput and critical path.
- Complete the first three sections of the report before starting the implementation.

5.2 Task 2: RTL

Thereafter, you need to realize the HW-accelerator by writing VHDL code. Functionality needs to get approved by comparing output of the RTL model with the data obtained from the behavioral model. The required tool for this task is Modelsim.

- Implement the HW-accelerator in VHDL.
- Verify your design with the testbench and the reference data from the behavioral model.

5.3 Task 3: ASIC Synthesis

In task three you need to synthesize your design. The tool you are going to use is Genus form Cadence. You will use a standard-cell library in 65 nm CMOS. You need to synthesize the design for maximum speed and minimum area.

In summary, for this task you are required to complete:

- Synthesize your RTL model, and **scrutinize the synthesis report for inferred latches!**
- Use different area and timing constraints for synthesis.
- Run a netlist simulation with back-annotated timing information.
- Document the synthesis results.

5.4 Task 4: Physical Implementation

In task four you need to place and route your netlist. The tool you are going to use is Innovus (SoCEncounter). You need to do floorplaning, power planning, cell placement, clock tree synthesis, and final power and signal routing. This needs to be done as demonstrated in the tutorial on PnR. During the course of placement and routing you need to generate a script that will do the entire flow automatically. Finally, you need to extract timing information and you need to run a post-layout simulation with back-annotated timing information in Modelsim.

In summary, for this task you are required to complete:

- Floorplaning, power planning, cell placement, clock tree synthesis, and final power and signal routing.
- Place the memories on the upper boundary to the left and right of the core.
- Two sets of supply pads for the core need to be placed in the middle of top and bottom pad row.
- Do not use more than one set of stripes.
- Distance between core and core-ring needs to be 10 μm.
- Generate a tcl script that does placement and routing.
- Post-layout simulation with back-annotated timing information.
- Run a netlist simulation with back-annotated timing information.
- Document the routing results.

5.5 Task 5: Power Simulations (Suggested)

For this task you need to perform power simulations. Basically, use the toggle information extracted from Modelsim (format vcd) and compute the dynamic power. The tool used for this is Prime-time, and apart from toggle information it requires the netlist and timing information.

- Load netlist and back annotate timing information (sdf) into prime time.
- Simulate the netlist in Modelsim and extract toggle information (vcd file).
- Compute the dynamic and leakage power.