ETIN25 - Digital IC Project 1

VT 1 2016

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., a vector 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 130 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 major milestones/deadlines for the project are

- Jan 29th: ASMD and datapath, Milestone
- Feb 8th: Synthesizable model
- Feb 15th: ASIC Synthesis and netlist simulation
- Feb 23th: Placement and Routing
- Feb 29th: Power Simulations and STA
- Mar 14th: Report

Contents

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 requires one multiplier unit, which accommodates several multipliers, 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 (soft) approved in time to be able to continue with the project part of the course. The final report has a hard deadline an will be posted in the course website.

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 (result), $X(n)$ is the input matrix of size (4x8) and A 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 provided in a ROM. The first element in P is computed as

$$p_{1,1} = x_{1,1}a_{1,1} + x_{1,2}a_{2,1} + x_{1,3}a_{3,1} + x_{1,4}a_{4,1} + x_{1,5}a_{5,1} + x_{1,6}a_{6,1}, + x_7(n)a_{7,1} + x_8(n)a_{8,1},$$

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 $(j = 1, 2, \ldots)$ in (3). Similarly, compute the other rows of matrix P, and store these rows in a RAM.

Post-Processing: (Mandatory for Grade- 5)

After computation of the matrix P, some algorithms require further post-processing to compute important statistics of a matrix. In this assignment three such functions need to be implemented as mentioned below.

Average of Diagonal Elements

Compute the average of the diagonal elements (*i.e.* matlab code : mean(trace(P)). This would basically require adding the diagonal elements and dividing as

$$avg = \frac{1}{M} \sum_{i=1}^{M} p_{i,i}. \tag{4}$$

Fortunately, value of $M = 4$ is our case, making the division trivial.

Maximum value

Another often used parameter is the element with maximum value of the matrix. This can be described as (matlab code : max(max(P))) :

$$max\_val = max(p_{1,1}, \ldots, p_{4,4}). \tag{5}$$

This operation can be performed by updating a register with the latest maximum value, which in hardware would require a comparator and DFF.

Sorting

This task requires additional handshake pins *sort_order*, *sort_start* and *sort_valid*. The sorting engine will act independently of the matrix multiplication. When *sort_start* is high, a state-machine should wait till the current matrix multiplication is done. Only after the matrix multiplication is completed the sorting process should begin. Note that during sorting the module is busy and should halt matrix multiplication and the corresponding master state machine, since it should not update the resultant matrix !!

If *sort_order* is high then an ascending sort needs to be performed else perform descending sort. The result of the sort needs to be streamed out element-wise. One can start with a simple bubble sort which is two nested loops, and then think of more advanced sorting methods.

3 Hardware Implementation

The arithmetic operation specified in (1) needs to be realized by a hardware accelerator. This is accomplished by either one or two multiplier units (MU)(up to the student to decide), which is 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). 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 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, the 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 depends on the number of vectors you want to save. The width of the RAM is 18 bit and thus it is possible to save one $p_{i,j}(n)$ on one address. The provided RAM has 256 memory locations which can be utilized. Since, one matrix P requires 16 address locations (4×4), upto 16 different resultant matrices can be saved in the RAM.

Note that the design has to be verified/simulated for more than 1 matrix continuously without asserting global reset.

4 Models and Verification

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

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. Note that the matrix is also written column wise, but for multiplication one might need to access its rows, which can be done using a simple addressing scheme in testbench or in design.

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

5 Compulsory tasks

The assignment is divided into four tasks. All tasks need to get approved before deadline in order to be able to continue with the project 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: Behavioural and Synthesizable model

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 architecture on register transfer level (RTL) and you need to investigate how your design may be optimized. The outcome of this study will be the initial architecture, supplemented with a discussion on throughput and the critical path. You need to implement a behavioural model that produces the reference data. This task can be accomplished with Matlab. 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 behavioural model. The required tool for this task is Modelsim.

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

- Implement a behavioural model (Matlab)
- Draw a block diagram showing your initial architecture.
- Sketch ASMD (according to Chapter 11 in "RTL Hardware Design Using VHDL", VLSI course book).
- Try to optimize the architecture and document possible improvements you have made.
- Calculate throughput and critical path.
- Implement the HW-accelerator in VHDL.
- Verify your design with the testbench and the reference data from the behavioural model.
- Document your design.

5.2 Task 2: ASIC synthesis

In task two you need to synthesize your design. You will use a standard-cell library in 130 nm CMOS. You need to synthesize the design for

A) maximum speed

B) minimum area

In summary, for task two 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.
  No timing violations are allowed!.
- Document the synthesis results.

5.3 Task 3: Physical Implementation

In task three you need to place and route your netlist. The tool you are going to use is 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 task two 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.

5.4 Task 4: Power Simulations (Mandatory for Grade- 5)

In task four 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 require 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.

5.5 Task 5: Report

The entire design process needs to be well documented in a report. Do not include any screen dumps from synthesis or layout tool. Filter the log information from the tool and put relevant information in tables. Generated scripts need to be included in the appendix. The script generated for PnR needs to be cleaned from repetitive instructions. All figures needs to be explained.