The Polynomial Evaluation Accelerator

Noah Olson, Erin Quartararo, Taruni Sanjay, Cole Schneider

Team 1

ENEE408C Section 0101

Shuvra Bhattacharyya

December 12, 2022

Contents

1 Team Contributions

1.1 Noah Olson

Early contributions to the project included the final breakdown of bits for the control input and how to handle N and S as one-dimensional arrays within Verilog. For the LIDE-C, Noah worked along with Erin to write the following: PEA actor header file (`lide_c_PEA.h`), constructor function, destructor function, enable function, and the struct for PEA actor within the source file (`lide_c_PEA.c`). Noah also handled the unit testing of EVB within the LIDE-C implementation.

Noah's contributions to the LIDE-V included finalizing the upper-level modules. These included: `enable_PEA.v`, and `invoke_PEA.v`. He also designed the full system test bench. After compilation, Noah handled the system testing of `enable_PEA.v`, `invoke_PEA.v`, `firing_state_FSM_2.v`, `mem_controller.v`, `get_command_FSM_3.v`, and `STP_FSM_3.v` modules. During the system-level testing, Noah made edits to the port lists of all the modules above and fixed Verilog run-time errors within these modules. Within `firing_state_FSM_2.v`, `mem_controller.v`, `get_command_FSM_3.v`, and `STP_FSM_3.v` Noah added and removed states and signals to achieve the desired functionality for the whole system. Noah also assisted in the system testing of both `EVP_FSM_3.v` and `EVB_FSM_3.v`. His final contribution was the implementation of a third design for the hardware.

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment /examination.

Noah Olson

1.2 Erin Quartararo

For the software implementation, Erin and Noah jointly wrote the header file (`lide_c_PEA.c`) and the invoke function. Erin wrote the enable function and applied error handling. Once this was finished, Erin set up the Cmake files. For testing the software implementation, Erin designed one test for STP, one test for EVP, and five tests for multi-instruction testing.

For the hardware implementation, Erin led the planning of the whole-system design, using mind maps and charts to plan module interactions. After this planning stage was finished, Erin wrote initial implementations for all instruction-related modules. These modules included `get_command_FSM_3.v`, the module used to split a command token into its proper segments, as well as `STP_FSM_3.v`, `EVP_FSM_3.v`, `EVB_FSM_3.v`, and `RST_FSM_3.v`. She also made revisions as necessary to `firing_state_FSM_2.v`, the module responsible for handling interactions between instruction FSMs (and RAM), and the invoke top-level module. When testing of individual modules began, Erin tested and debugged the EVP and EVB modules. Once full-system testing began, Erin designed a few tests for each instruction, debugging along the way. She wrote three multi-instruction, full-system tests and confirmed their proper operation.

Erin also contributed to writing and revising this report.

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

Erin Quartararo

1.3 Taruni Sanjay

For the software implementation, Taruni completed the unit testing for the EVP and RST instructions. Taruni ran the test commands to check equivalence for all the tests respectively and confirmed that these instructions work. Taruni wrote a total of 3 unit tests for the EVP instruction and 3 for the RST instruction within the LIDE-C implementation. For the hardware implementation, Taruni drew out the design hierarchy diagrams for the entire project. After this, Taruni and Cole drew the state transition diagrams during the planning stage for every instruction computed. During the implementation stage, Taruni assisted Cole with the `enable_PEA.v` and `invoke_PEA.v` modules. After this, Taruni wrote down the second-level finite-state machine module, `firing_state_FSM2.v` which acts as the binding element between the top and lowest level modules in the design. Taruni categorized each instruction into firing and wait states in `firing_state_FSM2.v`, implying functionality for each of the states. After this, Taruni wrote down the output signal assignments for every state mode by declaring different output registers and wires to pass the data between the upper and lower-level codes. Taruni also received some help from Erin, Noah, and Cole during the module instantiation phase in the firing state. For documentation, Taruni completed the documentation for the source codes of the hardware component of design 1.

Finally, Taruni led the implementation of multiple sections of the report and project planning.

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

Taruni Sanjay

1.4 Cole Schneider

Cole made active contributions during all phases of the project. These include planning, implementation, verification, and analysis of results.

During the initial planning phase, he proposed design decisions that stuck throughout all versions of the Polynomial Evaluation Accelerator (PEA)'s implementation, such as the contents of the command token.

During the implementation phase, Cole contributed to modules at the top-level, creating diagrams for the interface of the PEA and writing drafts of the `invoke_PEA.v`, and `enable_PEA.v` modules. However, Cole's main contributions were at the lower levels of the PEA. He designed and implemented the `mem_controller.v` module, as well as the interactions between lower-level finite state machines with multiple memory modules. One such contribution included the design of a RAM module with additional reset capabilities, `N_ram.v`. Due to the familiarity he gained during the planning and implementation phases, Cole was able to develop a second design for the PEA that tweaked the memory layout of the original PEA implementation.

During the verification phase, Cole designed multiple unit tests and system-level tests for both the software and hardware components. For the software component, he designed unit tests for the STP function. For the hardware component, he designed multiple unit and system-level tests, from which he discovered bugs and other unintended functionalities across several modules. He applied the appropriate fixes, and in some cases, added new elements to the design. These include the multiplexers found in the `firing_state_FSM2.v` module and a revamp of the state transitions and outputs for multiple finite state machines.

For the results and concluding phase of the project, Cole ran the synthesis and implementation processes on all three of the PEA designs. He used the Vivado design suite to generate the reports needed to analyze the designs and compare their performance.

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

Cole Schneider

2 Executive Summary

The Polynomial Evaluation Accelerator (PEA) is an application-specific hardware accelerator unit that is dedicated to polynomial evaluation. The PEA performs the following functions: storing a polynomial up to a degree of 10 (STP), evaluating a polynomial at a certain integer value (EVP), repeated polynomial evaluation (EVB), and resetting the module (RST). The PEA is designed to interface with four dual port first-out (FIFO) buffers, two at the input containing 16-bit data and command tokens, and two at the output expecting to receive 32-bit result and status tokens.

In this report, the PEA has been implemented as two separate components based on different development environments: software (C-based) and hardware (Verilog-based). The software component implements the Polynomial Evaluation Accelerator (PEA) as a LIDE-C actor, an element of a high-performance tool set for designing lightweight dataflow models. The hardware component implements the PEA in LIDE-V, a similar tool set for implementing dataflow models as an FPGA-based digital design.

The development of the PEA was achieved by a team of four. Starting from initial discussions about the requirements, we worked collaboratively to plan, implement, verify, and analyze the requirements of the system to produce a high-quality and effective product that met the goals and standards of the team.

The software component was developed first in order to provide a high-level but technical overview of the functionality required for the hardware component. After successful verification of the software component, drafts of the interface, system-level test benches, and high-level internal finite state machines (FSMs) were created. Lower-level FSMs were then developed around the algorithms and descriptions of the software component and the interfaces of the high-level modules.

After the development phase, unit tests of the hardware component were created and analyzed using both LIDE-V and the Vivado design suite. As the design came to fruition, system-level tests were developed, standardized, and performed to ensure system quality. Initial experimental results of system-level tests showed the need to address concurrency and multi-driver output errors. Multiple designs were then explored to address alternate paths that arose during the previous project phases. These PEA designs were then synthesized and implemented using the Vivado design suite.

Afterwards, the performance of the various hardware modules was analyzed and compared using reports generated by the Vivado synthesis and implementation process. These reports showcased the critical importance of timing in digital system design - which can make or break the success of digital hardware.

3 Goals and Design Overview

The main goal of this project was to design, in both software and hardware, an effective and efficient polynomial evaluation accelerator. A secondary main goal was to implement multiple hardware designs and determine the best design. The first goal was accomplished. The second goal was accomplished in part. While we originally intended to implement several different hardware designs, only two came to fruition. A third design was unable to pass timing requirements and was thus deemed an ineffective design. It is still mentioned throughout this report.

The software and hardware implementations were designed as separate entities. See Figure **??** for a very basic overview of the system. The system is described in more detail below.

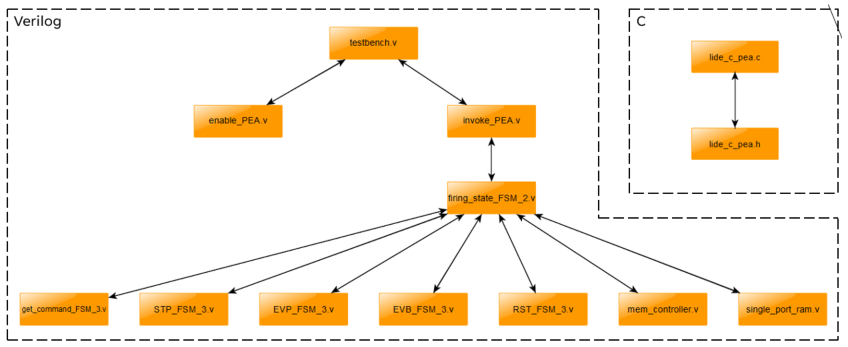


Figure 1: Basic layout of the organization of both the software and hardware implementations

The software implementation is intuitive. It is comprised of a header file (`lide_c_PEA.h`) and a C file (`lide_c_PEA.c`). Within the header file are definitions needed by the C file such as the actor structure definition. The C file contains the actual implementation, including four necessary functions:

- `lide_c_pea_new`, a constructor for a PEA actor
- `lide_c_pea_enable`, which determines, given the current state, if the actor is allowed to fire
- `lide_c_pea_invoke`, which contains the implementation for what happens when the actor fires given its current state
- `lide_c_pea_terminate`, a destructor for a PEA actor

The hardware implementation is more complex and involves many modules working together. The diagram in Figure 2 provides a high-level glimpse of what this design entails.

At the level closest to the user, which we refer to as the highest level, is the testbench. The testbench is modified for each test in the `tests/verilog` directory, but maintains across all tests the same goal of controlling the flow of the PEA and providing a way of interfacing with the FIFOs and the enable module.

When the writer of the testbench wishes to invoke an instruction, they must first check that the enable signal, which at that instance is relevant to the desired instruction, is high. When it is, the PEA may proceed through the invoke module. This enable and invoke functionality is the key to the hardware design.

One level lower than the testbench is the enable module, which works as described above. Also on this level is the invoke module, which controls interactions between the testbench (and thus the enable module) and the firing_state module. the firing_state module, one level lower than the invoke module, directly communicates with the lowest-level modules, which we also call the level-3, layer-3, or instruction-level modules. The firing_state module is responsible for allowing the level-3 FSMs to communicate with each other and for information to be synced between the level-3 FSMs and the higher-level testbench and enable module.

At the lowest level are the instruction modules and RAM modules. The instruction modules are responsible for performing specific tasks related to the instructions they are named for. The RAM modules store tokens from the FIFOs to allow easier, controlled access by the instruction modules.

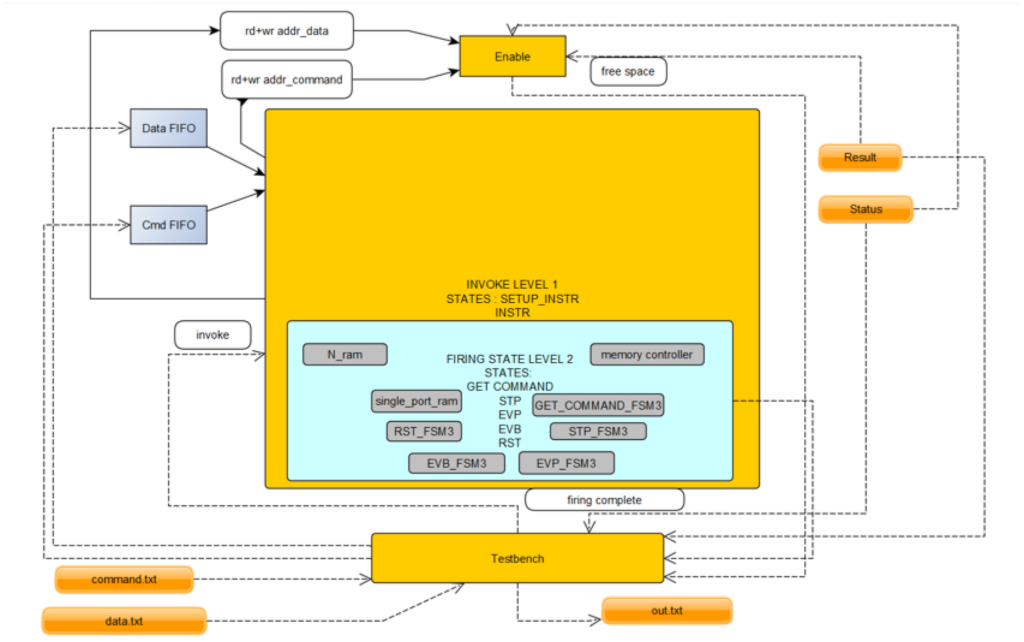


Figure 2: High-Level Design Hierarchy of the Polynomial Evaluation Accelerator

A major challenge of this project was figuring out how the modules should interact with each other. Among the group, we pondered over what signals needed to be passed between modules, what signals needed to be placed in the port list, and what signals were necessary for proper timing control. To ease this issue and speed up the planning phase, we made use of several ”mind maps” and charts, an example of which is shown in Figure 3.

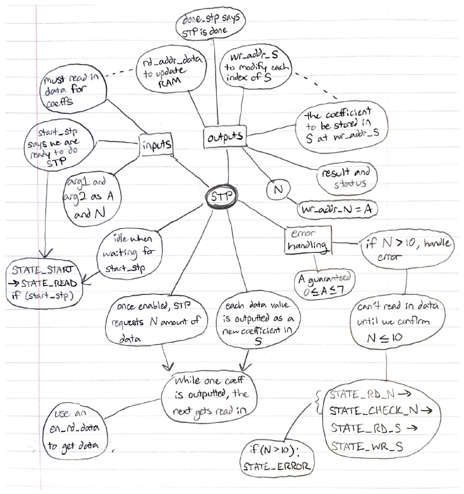

done_stp says STP is done
rd_addr_data to update RAM
wr_addr_S to modify each index of S
must read in data for coeffs
the coefficient to be stored in S at wr_addr_S
start_stp says we are ready to do STP
inputs
outputs
result and status
arg1 and arg2 as A and N
STP
N
wr_addr_N = A
idle when waiting for start_stp
error handling
if N > 10, handle error
STATE_START → STATE_READ if (start_stp)
A guaranteed 0≤A≤7
once enabled, STP requests N amount of data
each data value is outputted as a new coefficient in S
can't read in data until we confirm N ≤ 10
while one coeff is outputted, the next gets read in
STATE_RD_N → STATE_CHECK_N → STATE_RD_S → STATE_WR_S
use an en_rd_data to get data
if(N>10): STATE_ERROR

Figure 3: "Mind map" of the STP instruction module

Another major challenge faced involved concurrency. In the first draft of our hardware implementation, we saw strange and unexpected behavior from the modules all because we had several conflicts in assignments among all the modules. This particularly was a problem at the lowest level modules. This served as a key learning point for the whole group.

4 Realistic Constraints

As a group, our aim was to build a design with high performance and reduced latency. The main factors that acted as obstacles to our goals are described below.

The project requirements state specific bit widths for all FIFO input and output tokens and some consistent buffer depth for all FIFOs. The size of both input buffers are 16-bits wide and the output buffers are 32-bits wide. These external FIFOs are subject to setup and hold delays and timing considerations with respect to the clock. For example, the result and status outputs must be held constant by the PEA for at least one clock cycle to ensure they can be captured properly by the output FIFOs.

In the software implementation of our project, the enable and invoke functions were the two most important and crucial parts. The authenticity of the enable function would be to check and compare the size of the input buffers with their populations. The invoke function handles the computation of all the instructions required to be computed for the actor to be generated, fired, and terminated. Both these functions give the readers a clear picture of how the actor implementation is done, despite not directly performing the polynomial computations themselves. Therefore, the enable and invoke functions are important constraints to be considered.

Implementation of the PEA requires a quality programming language and hardware description language.

The software component of our project is written in `C` due to its high level of performance at low-level procedural programming. The hardware component of our project is fully implemented in the `Verilog HDL`. Verilog can easily simplify, simulate, debug and synthesize designs using its various automated tools, helping the team focus purely on the core obstacles of digital design.

At a bare minimum, the PEA must satisfy the desired functionality specified in the requirements and implemented in the software component, and pass timing constraints. A design that does not meet its specification is potentially life-threatening and breaks the sense of trust and reliability between engineer and consumer. Similarly, a design that does not meet timing requirements can not be turned into an end product.

The four instructions being on a parallel level within the FSM model present an implementation constraint on the design of the actor. Because the instruction is not known in advance, both the software and hardware elements must be able to dynamically respond to the input in at least four different ways. This means that the processing of information must occur at each step of the CFDF table and model, requiring a large set of interconnected digital states.

5 Engineering Standards

The software implementation is done using the LIDE-C package environment provided by DSPCAD. LIDE (DSPCAD Lightweight Dataflow Environment) is a flexible, lightweight design environment that allows designers to experiment with dataflow-based approaches for the design and implementation of digital signal processing (DSP) systems. LIDE contains libraries of dataflow graph elements (primitive actors, hierarchical actors, and edges) and utilities that assist designers in modeling, simulating, and implementing DSP systems using formal dataflow techniques. The hardware implementation is constructed in a LIDE-V Dataflow environment through which users can access 'makeme' and 'runme' files to compile their codes in a successful manner. The testing for both hardware and software are done using DICE packages which are user-friendly and fast. The ability to use the same environment for both software and hardware development is a huge advantage of LIDE. LIDE-V for Verilog has built-in interactive project development for Xilinx Vivado to simulate and synthesize designs along with the Tcl console.

C continues to be the realm of low-level language programming. The utilization of abstract data types and interfaces in the project can be well-understood in C. The actor implementation in C makes use of C structures, which make a design more readable. C is in close association with high-level, low-level, and middle-level languages too. This project required a large amount of code broken down into small fragments, which works well with C. For instance, the enable and invoke functions are two different major blocks coded in `lide_c_pea.c` which can be differentiated clearly. C also has fewer libraries in comparison with other high-level languages and is utilized in embedded programming applications too.

Verilog as a hardware description language is now a standard language for the design of digital subsystems, micro-controllers, and flip-flops. Verilog is used to simplify the design schematic and make it easier for the programmer to lay out the hierarchy of the design in code(low-level language). In our project, we have utilized top-down and bottom-up designs at behavioral, register transfer, and gate-level abstraction.

Xilinx Vivado is able to verify and simulate design integrity at different stages of the design process such as behavioral, post-synthesis functional, and timing simulation and Post-implementation functional and timing simulation. We use the Vivado tool suite to generate the schematic HDL by running a behavioral simulation, synthesis, and post-synthesis. Vivado supports both Windows and Linux operating systems with powerful debugging features that are aimed to address verification needs.

6 Alternative Designs and Design Choices

An alternative design to the original PEA uses a different layout for the RAM used to store the coefficient vectors, which we denote S RAM. The original PEA stores the coefficient vectors in a linear fashion, indexing each 16-bit word within a one-dimensional array of 88 words: with each word representing a single coefficient, there are 11 words per coefficient vector and a total of 8 coefficient vectors. This one-dimensional array method is shown below in Figure 4.

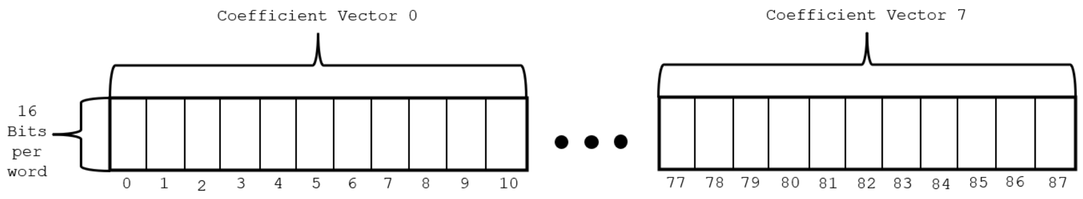


Figure 4: The S RAM layout of the original design, which stores all CV coefficients in a linear fashion

The alternative design stores the coefficients in a two-dimensional, 8x11 array of words. This method is shown below in Figure 5.

The motivation behind this design change is to reduce the combinational logic required to read and write from the S RAM module. In the original design, a 7-bit adder is used within the STP, EVP, and EVB modules to determine the read and write address inputs for the S RAM module. However, for the second design, this 7-bit adder is not required. Instead, two addresses – one for the coefficient vector (the row) and one for the specific coefficient (column) – are determined by the command token and a 4-bit adder, respectively. By reducing the complexity and the number of gates for the adder circuit, the resource utilization of the

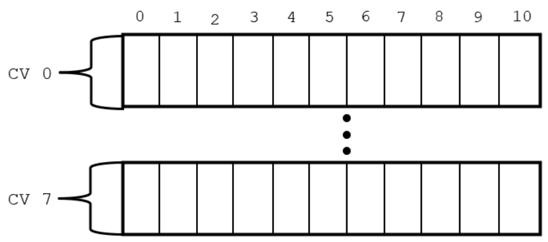
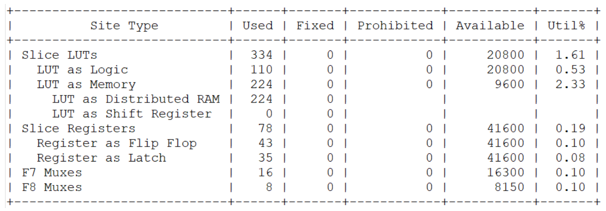
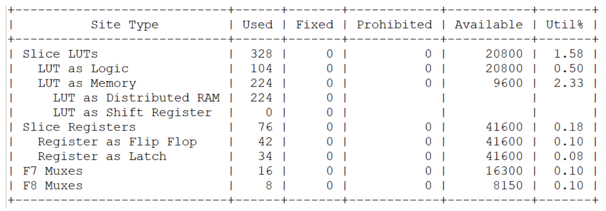


Figure 5: The S RAM layout of design 2, which stores all 88 CV coefficients in an 8x11 two-dimensional array

Artix-7 FPGA decreases. Shown below in Figure 6 are the utilization reports generated using the Vivado implementation process. Notice that the utilization percentage for design 1 is slightly larger than for design 2, indicating that the alternative design is ever so slightly more efficient.



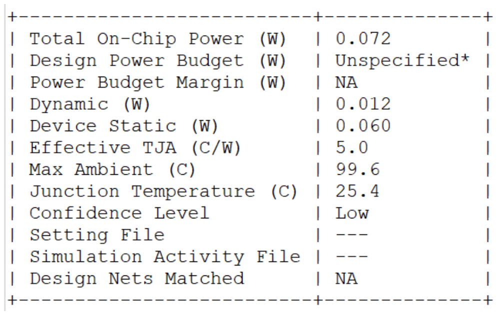
```
+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 |  334 |     0 |          0 |     20800 |  1.61 |
|   LUT as Logic             |  110 |     0 |          0 |     20800 |  0.53 |
|   LUT as Memory            |  224 |     0 |          0 |      9600 |  2.33 |
|     LUT as Distributed RAM |  224 |     0 |            |           |       |
|     LUT as Shift Register  |    0 |     0 |            |           |       |
| Slice Registers            |   78 |     0 |          0 |     41600 |  0.19 |
|   Register as Flip Flop    |   43 |     0 |          0 |     41600 |  0.10 |
|   Register as Latch        |   35 |     0 |          0 |     41600 |  0.08 |
| F7 Muxes                   |   16 |     0 |          0 |     16300 |  0.10 |
| F8 Muxes                   |    8 |     0 |          0 |      8150 |  0.10 |
+----------------------------+------+-------+------------+-----------+-------+
```



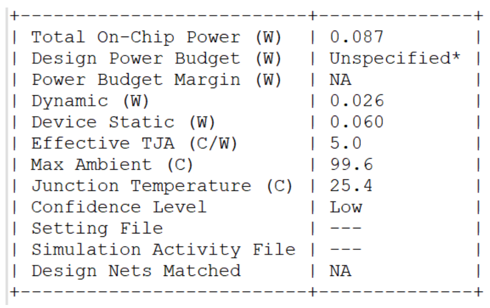
```
+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 |  328 |     0 |          0 |     20800 |  1.58 |
|   LUT as Logic             |  104 |     0 |          0 |     20800 |  0.50 |
|   LUT as Memory            |  224 |     0 |          0 |      9600 |  2.33 |
|     LUT as Distributed RAM |  224 |     0 |            |           |       |
|     LUT as Shift Register  |    0 |     0 |            |           |       |
| Slice Registers            |   76 |     0 |          0 |     41600 |  0.18 |
|   Register as Flip Flop    |   42 |     0 |          0 |     41600 |  0.10 |
|   Register as Latch        |   34 |     0 |          0 |     41600 |  0.08 |
| F7 Muxes                   |   16 |     0 |          0 |     16300 |  0.10 |
| F8 Muxes                   |    8 |     0 |          0 |      8150 |  0.10 |
+----------------------------+------+-------+------------+-----------+-------+
```

Figure 6: The utilization reports for design 1 (left) and design 2 (right), generated by the Vivado implementation process

The original design uses 110 Logic LUTs, 43 registers as flip flops, and 35 registers as latches. The alternate design uses 104 Logic LUTs, 42 registers as flip flops, and 34 registers as latches, which is a minuscule improvement. However, despite the reduction in LUTs and registers, the additional read and write address signals require an increase in power consumption. Shown below in Figure 7 are the routed power reports generated by the Vivado implementation process for each design. Notice that the total on-chip power consumption is lower for design 1 than for design 2.



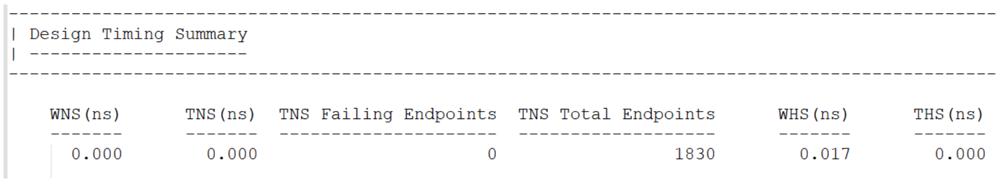
```
+--------------------------+--------------+
| Total On-Chip Power (W)  | 0.072        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 0.012        |
| Device Static (W)        | 0.060        |
| Effective TJA (C/W)      | 5.0          |
| Max Ambient (C)          | 99.6         |
| Junction Temperature (C) | 25.4         |
| Confidence Level         | Low          |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
```



```
+--------------------------+--------------+
| Total On-Chip Power (W)  | 0.087        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 0.026        |
| Device Static (W)        | 0.060        |
| Effective TJA (C/W)      | 5.0          |
| Max Ambient (C)          | 99.6         |
| Junction Temperature (C) | 25.4         |
| Confidence Level         | Low          |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
```

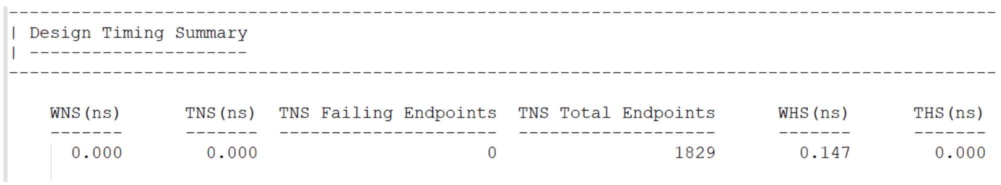
Figure 7: The routed power reports for design 1 (left) and design 2 (right)

The original design utilizes a total on-chip power of 7.2mW, while the alternate design utilizes 8.7mW, which is approximately a 20% increase. This increase is likely due to the increased number of signals at the interfaces of third-level FSM modules like STP, EVP, EVB, and S RAM. The large increase in power consumption does not outweigh the slight improvement in resource utilization for the second design. As such, the alternative design requires a greater number of resources overall compared to the original design. The performance of both designs is the same, despite the alternative design requiring additional resources. The performance can be compared using the timing reports shown in Figure 8. The worst negative slack (WNS) and total negative slack are 0 ns for both designs when a 20 ns clock period is used.



```
------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
    -------      -------  ---------------------  -------------------      -------      -------
      0.000        0.000                      0                 1830        0.017        0.000
```



```
------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
    -------      -------  ---------------------  -------------------      -------      -------
      0.000        0.000                      0                 1829        0.147        0.000
```

Figure 8: The routed timing reports for design 1 (top) and design 2 (bottom)

The WNS values for both designs are 0 ns. This means that the maximum possible clock frequency is $\frac{1}{20}$ ns = 50 MHz for both designs. Due to the similar data flow of both designs, the number of clock cycles required to provide output to the external buffers is the same. Given that the maximum possible speed of computation and the required number of cycles are the same for each design, both designs perform equally.

Given that designs 1 and 2 perform equally well and that the second design is less efficient with resource usage, the original design is better. This is reflected in the Pareto diagram in Figure 9, in which the original design lies on a greater utility curve than the alternative design. Design 1 lies on a greater utility curve despite it having equal performance with design 2.

The change in S RAM layout is not the only additional design. A third design, which focuses on reducing the number of states in the third-level FSMs, has also been explored.

During the verification phase of the project, additional states such as `STATE_IDLE` were added to modules at the third level. Further analysis of the behavioral description of these modules indicated that certain states could be combined or removed entirely to achieve the same behavioral function. As such, `STATE_IDLE` was removed in the STP and EVP modules, and the `GET_NEXT_COEFF` and texttttCOMPUTE_EXPONENT states were combined in the EVP FSM.

However, this design cannot be evaluated since it does not meet timing requirements. Based on the

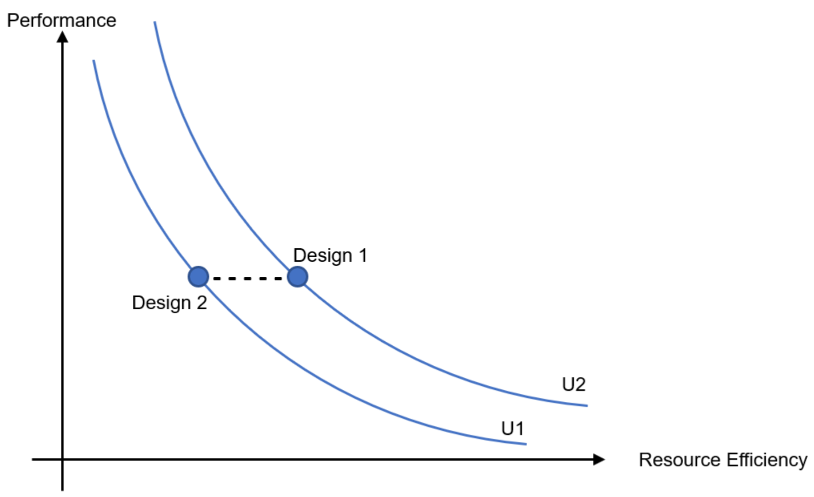
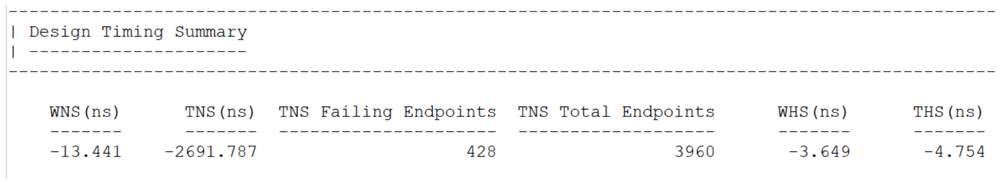


Figure 9: The pareto diagram with the utilities of designs 1 and 2 plotted against performance and resource efficiency

timing report shown in Figure 10, the worst negative slack value is negative, at -13.441 ns. For the design to be valid, this value must be non-negative, and thus design 3 is invalid.



```
------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)
    -------      -------  ---------------------  -------------------      -------      -------
    -13.441    -2691.787                    428                 3960       -3.649       -4.754
```

Figure 10: The timing report for the third PEA design

Throughout the planning, implementation, and verification phases of this project, many design decisions and paths were considered. The alternative designs, 2 and 3, were based on the many ideas and observations made during the design process. Unfortunately, design 2 is less efficient compared to the original design, and the third design has the potential to outperform its counterpart but did not come to fruition before the deadline of this project.

7 Technical Analysis for System and Subsystems

7.1 Top-Level Modules

Our design as shown in Figure 2, shows an overview of the design hierarchy that we derived as a group over the past few months. The input buffers, data and command, send their respective data into the finite state machines. The enable module reads in the read address for data and command as two signals each and checks if the population of the command and data FIFOs is greater than the value of the second argument to evaluate the functionality of EVB or not. Each finite-state machine at the lowest level is an independent finite-state machine that performs its relevant computation. The enable module was initially implemented in a way very similar to the prior assignments completed during the class. After a detailed discussion and a better understanding of this project, we came to the conclusion that it's best to check for the FIFO buffer's depth and the address commands it reads. The `SETUP_INSTR` state mode enables a high operation if the difference between the read address command from the write address command is greater than one. Moving on to the next state modes, the STP and EVB check if the difference is greater than the second argument (value of b). The EVP module checks if the difference is greater than one. After implementing these novel measures, the enable module was able to take in the command address and data address tokens in order to prepare them for firing later on.

The top module with the invoke function indicates the completion of one full operation cycle. The state modes comprise the Idle, Firing start, and Firing wait stages, wherein the very first always block checks for the reset instruction and transitions to the Idle state, and the second always block performs the state transition as the invoke signal is enabled. It first transitions to the firing state wherein tokens are getting fired, and once the output signal is enabled, the firing transitions to a wait state. The input signal assignment is enabled to a high operation, too. This top module is easy to understand and conveniently shows a clear difference between the functionality of the first and second levels of the finite state machine.

The firing state module acts as a binding hierarchy between the top module and the lowest level (instruction) modules. This design first checks the functionality of the reset instruction and begins the transition with the 'Start' state mode. The second segment comprises two modes – `SETUP_INSTR` and INSTR. The main difference between this firing state and others (assignments) is the declaration of the "get command" state mode – `GET_COMMAND_START`, `GET_COMMAND_WAIT`, and `GET_COMMAND_FINISH`. "Get command" is not a definite instruction and is utilized for pre-empting tokens to start the computation. The output signal assignment shows the signal which comes in from the parent finite state machine module.

7.2 Low-Level Modules

The third-level FSMs include instruction modules, a "get command" module, and RAM modules. Normal operation involves close interaction and passing of signals with the `firing_state_FSM_2` module. These signals include read and write addresses for the different RAMs, enable signals to allow reading and writing to and from RAM, and enable and done signals for each instruction module. Result and status are outputted by each instruction module to be passed up to the output FIFOs. Depending on each module's implementation, other signals may be passed as well, but the ones listed are the common ones.

The `get_command_FSM_3` module takes a command token, intentionally read in during one of its states, as input. In a subsequent state, the module splits the command token into an instruction, a first argument, and a second argument. These three command components are the outputs of this module. The outputted instruction is used by `firing_state_FSM_2` to determine what instruction will be processed next, and that instruction's enable signal is set appropriately. Only one instruction can be enabled at a time. The STP, EVP, and EVB modules take the first and second arguments as inputs. Each of them performs their respective operations and outputs a result and status appropriately.

Each instruction modules features three always blocks. The first always block depends upon the positive edge of the clock and the negative edge of the reset signal; on a reset, it resets the state to `STATE_IDLE` or `STATE_START` and resets counters, temporary registers, and anything else that requires clearing; on a positive clock edge, it pushes all temporary ”next” values into their respective ”final” values, which could be either registers local to the module or outputs. The second always block determines the next state, sometimes depending upon when certain conditions are met. For example, EVP's next state is set to `STATE_END` when counter tracking the index of S we are currently processing reaches N, the size of the coefficient vector. The third always blocks changes the values of registers and outputs depending upon what current state the module is operating in. In data-retrieval-related states or data-sending-related states, this may involve setting read enable or write enable signals high and changing read or write addresses. In computation-related states, this may involve performing a computation and updating registers or outputs as a result.

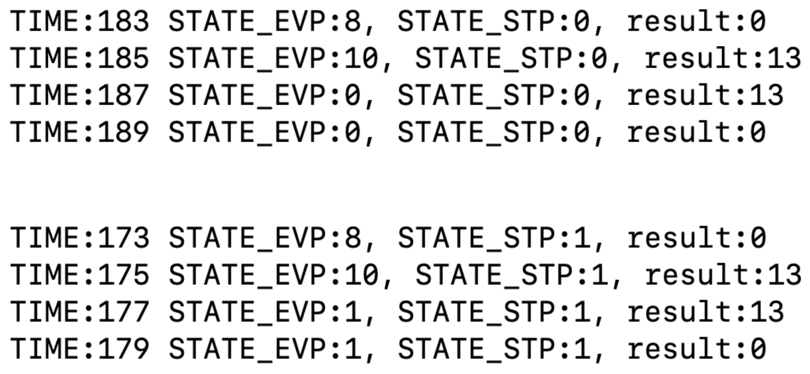
The design utilizes 4 separate RAM modules. They are used to store the set of coefficient vectors, S, N (the degree of each polynomial in S), data tokens, and command tokens. The memory controller unit senses whenever there are idle data or command tokens in the respective input FIFOs, and loads them into the data and command RAM modules. The N RAM and S RAM modules are used when calls are made to STP, EVP, or EVB. For STP, The selected CV has its degree taken from the Get Command module and written into the specified N RAM. Data tokens are then taken from the data ram and written into the S RAM modules. EVP and EVB interact with the RAM modules in a similar manner, reading values from the N RAM at the address specified in the command token, and from the S RAM in an incremental fashion, starting at the first coefficient of the specified CV. After STP, EVP, and EVB are done, write enable signals are driven high, along with result and status tokens.

8 Design Validation for System and Subsystems

The testbench inputs used to verify the design cover a large breadth of cases at the system level, ensuring a high level of confidence in the correct performance of the whole system. Various cases of each input were used, such as STP commands with large and small amounts of data inputs. Multi-instruction tests were also used to verify system performance, addressing behavior in unexpected corner cases.

These system-level testbenches were built from existing unit tests for the subsystem level. Every module on the third FSM layer, including the memory controller, STP, EVP, EVB, and RST was given its own thorough unit testing to ensure that functionality was correct. An incremental integration and test allowed for corner cases to be covered at every level.

From these testbenches, performance of design variations could be evaluated based on the amount of time required to write result and status tokens to the output FIFOs when given the same test inputs. Figure 11 shows that the time benchmark differs between different designs. Notice that the time required for the result to become a value of 13 is greater for design 1 than for design 3. This indicates that design 3 performed better in the behavioral simulation.



```
TIME:183 STATE_EVP:8, STATE_STP:0, result:0
TIME:185 STATE_EVP:10, STATE_STP:0, result:13
TIME:187 STATE_EVP:0, STATE_STP:0, result:13
TIME:189 STATE_EVP:0, STATE_STP:0, result:0


TIME:173 STATE_EVP:8, STATE_STP:1, result:0
TIME:175 STATE_EVP:10, STATE_STP:1, result:13
TIME:177 STATE_EVP:1, STATE_STP:1, result:13
TIME:179 STATE_EVP:1, STATE_STP:1, result:0
```

Figure 11: The EVP testbench output in BASH for design 1 (top) and design 3 (bottom) for an EVP unit test

Using a combination of thorough unit testing, system testing, and time benchmarking, the performance of the PEA can be evaluated for each design. Evaluation by simulation time allows for each design to be compared performance-wise in a consistent manner.

9 Test Plan

For testing, we used the unit testing tools ’dxtest’, ’dxmktest’, and ’dxitsout’ that come with the DICE testing suite. The organization of both LIDE-C and LIDE-V testing directories followed the DICE testing conventions with a separate testing sub-tree for each main testing area. Within each testing, sub-tree were individual testing sub-directories (ITS) for each test. Our approach to the testing for both LIDE-C and LIDE-V implementations of the PEA was the Incremental Development Workflow (IDW). This involved making incremental changes to the code and then testing it to ensure the desired outcome was reached.

9.1 LIDE-C Testing

The testing for each command (STP, EVP, EVB, RST) and multiple instructions were divided among the group members. Each member was tasked with generating an ITS testing suite that would validate that the command was working for various inputs. After the code was compiled, we created `lide_c_pea_driver.c`. This driver utilized the CFDF canonical scheduler `lide_c_util_simple_scheduler`. Along with the lide_c_util_guarded_execut to continue invoking the PEA actor when there were multiple instructions. For each test, the results and statuses of the respective input command were directed into a file out.txt within each ITS. This output was then validated and passed.

9.2 LIDE-V Testing

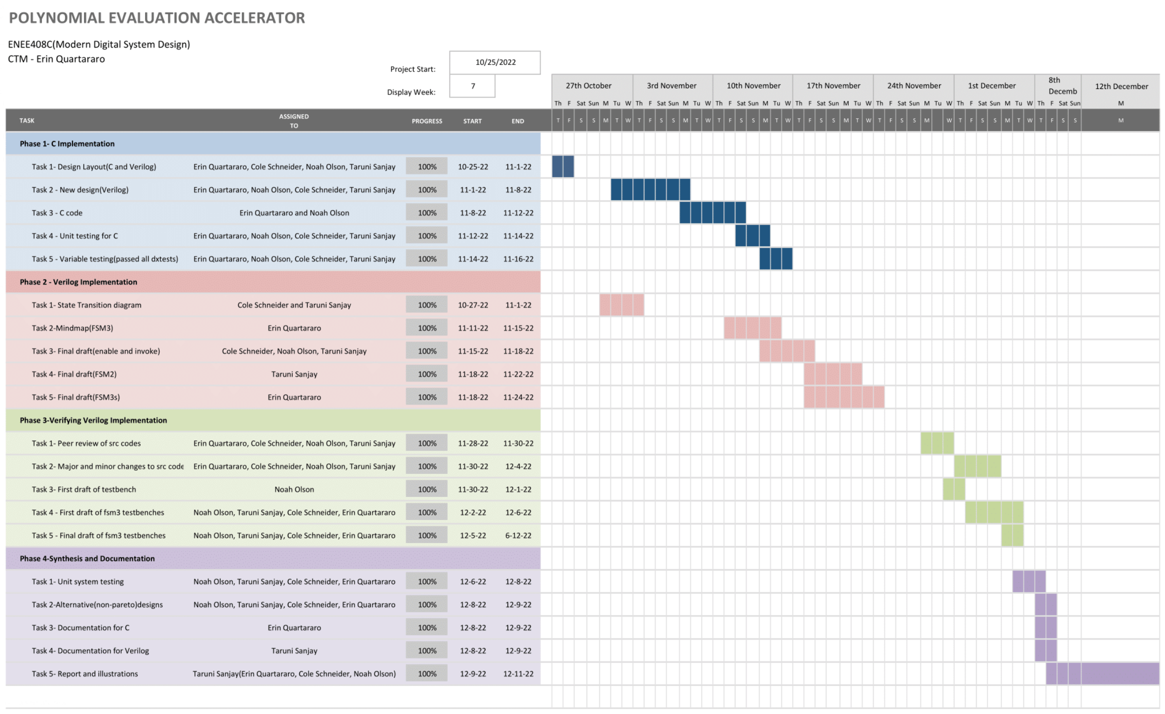
For testing the hardware implementation, we took advantage of Verilog's hierarchical design structure and tested different modules separately before testing the entire system together. Using this method allowed us to verify that the individual modules were generating the correct results and statuses for EVP, and EVB. While also ensuring that STP was writing to the CV's RAM properly. To build up to the whole system level testing, each module was incrementally instantiated after the previous modules' signals and states were behaving as expected. The order of incorporating the modules in the whole system was as follows: `enable_PEA.v`, `invoke_PEA.v`, `firing_state_FSM_2.v`, `mem_controller.v`, `single_port_ram.v`, `get_command_FSM_3.v`, `STP_FSM_3.v`, `EVP_FSM_3.v`, `EVB_FSM_3.v`, and `RST_FSM3.v`. During the whole system testing, '`$monitor`' statements and Vivado timing diagrams were used to create state transition diagrams, watch read/write addresses and view various enable signals timing.

10 Project Planning and Management

The project planning and scheduling are represented chronologically with the help of a Gantt chart. This Gantt chart, shown in Figure 12, provides a systematic way of depicting the allotment of work among group members and deadlines for every task.

11 Conclusion

Our project comprises of two fully-working hardware designs and one non-working "concept" design, out of which we determined design 1 is the pareto design. We have taken important benchmark considerations with regard to performance and latency measurements. The pareto design is the reference on which the two non-Pareto designs have been built. The software implementation of our project is the common reference



POLYNOMIAL EVALUATION ACCELERATOR

ENEE408C(Modern Digital System Design)
CTM - Erin Quartararo

Project Start: 10/25/2022
Display Week: 7

TASK	ASSIGNED TO	PROGRESS	START	END
Phase 1- C Implementation				
Task 1- Design Layout(C and Verilog)	Erin Quartararo, Cole Schneider, Noah Olson, Taruni Sanjay	100%	10-25-22	11-1-22
Task 2 - New design(Verilog)	Erin Quartararo, Noah Olson, Cole Schneider, Taruni Sanjay	100%	11-1-22	11-8-22
Task 3 - C code	Erin Quartararo and Noah Olson	100%	11-8-22	11-12-22
Task 4 - Unit testing for C	Erin Quartararo, Noah Olson, Cole Schneider, Taruni Sanjay	100%	11-12-22	11-14-22
Task 5 - Variable testing(passed all dxtests)	Erin Quartararo, Noah Olson, Cole Schneider, Taruni Sanjay	100%	11-14-22	11-16-22
Phase 2 - Verilog Implementation				
Task 1- State Transition diagram	Cole Schneider and Taruni Sanjay	100%	10-27-22	11-1-22
Task 2-Mindmap(FSM3)	Erin Quartararo	100%	11-11-22	11-15-22
Task 3- Final draft(enable and invoke)	Cole Schneider, Noah Olson, Taruni Sanjay	100%	11-15-22	11-18-22
Task 4- Final draft(FSM2)	Taruni Sanjay	100%	11-18-22	11-22-22
Task 5- Final draft(FSM3s)	Erin Quartararo	100%	11-18-22	11-24-22
Phase 3-Verifying Verilog Implementation				
Task 1- Peer review of src codes	Erin Quartararo, Cole Schneider, Noah Olson, Taruni Sanjay	100%	11-28-22	11-30-22
Task 2- Major and minor changes to src code	Erin Quartararo, Cole Schneider, Noah Olson, Taruni Sanjay	100%	11-30-22	12-4-22
Task 3- First draft of testbench	Noah Olson	100%	11-30-22	12-1-22
Task 4 - First draft of fsm3 testbenches	Noah Olson, Taruni Sanjay, Cole Schneider, Erin Quartararo	100%	12-2-22	12-6-22
Task 5 - Final draft of fsm3 testbenches	Noah Olson, Taruni Sanjay, Cole Schneider, Erin Quartararo	100%	12-5-22	6-12-22
Phase 4-Synthesis and Documentation				
Task 1- Unit system testing	Noah Olson, Taruni Sanjay, Cole Schneider, Erin Quartararo	100%	12-6-22	12-8-22
Task 2-Alternative(non-pareto)designs	Noah Olson, Taruni Sanjay, Cole Schneider, Erin Quartararo	100%	12-8-22	12-9-22
Task 3- Documentation for C	Erin Quartararo	100%	12-8-22	12-9-22
Task 4- Documentation for Verilog	Taruni Sanjay	100%	12-8-22	12-9-22
Task 5- Report and illustrations	Taruni Sanjay(Erin Quartararo, Cole Schneider, Noah Olson)	100%	12-9-22	12-11-22

Figure 12: Project Planning and scheduling

point with regard to the functionality of our project. We have used multiple references from the software component of our project in order to build designs and create alternative designs. We aimed to create something which could eventually scale up in the FPGA system design and subsystem design domain. While we did not come close to achieving this, evidenced by the ultimate performance of our pareto design, we strive to continue developing the technical skills and soft skills we made use of during the course of this subject and domain.

12 References

[1] 'ELSS: A Logic Synthesis Tool for FPGAs', R.P Ranauro and M.M. Ligthart, proceedings of the 4th annual IEEE Asic conference and Exhibit, 1991, pp.13.2.1-13.2.4.

[2] 'Technology Mapping in MIS', E. Detjens, G. Gannot et..al., proceedings of the ICCAD, 1987, pp.1 16-119.

[3] 'Module Generation for VHDL synthesis', R.W. Dekker and M.M. Ligthart, proceedings of the VIUF spring conference, 1993.

U. Afzaal, J. -A. Lee, "Low-cost Hardware Redundancy for Fault-mitigation in Power-constrained IoT Systems," 2020 International Conference on Information and Communication Technology Convergence (ICTC), 2020, pp. 60-62, doi: 10.1109/ICTC49870.2020.9289420.P. S. Abril, R. Plant, The patent holder's dilemma: Buy, sell, or troll? Communications of the ACM 50 (2007) 36-44. doi:10.1145/1188913.1188915.

[2] P. Mallavarapu, H. N. Upadhyay, G. Rajkumar and V. Elamaran, "Fault-tolerant digital filters on FPGA using hardware redundancy techniques," 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA), 2017, pp. 256-259, doi: 10.1109/ICECA.2017.8212811.

[3] Eckhardt, Dave E., Larry D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors." IEEE Transactions on software engineering 12 (1985) 1511-1517.

[4] Igor V. Kovalev, Mikhail V. Saramud, Vasiliy V. Losev, Information and Software Technology 120 (2020). doi:10.1016/j.infsof.2019.106245