ECE 527 MP1

Released: Thursday 8/28/2025

Due: Sunday 9/14/2025 at 11:59 PM Central Time

1 Introduction

This MP can be done (and we recommend doing it) with a partner. This MP is an introductory assignment to familiarize you with Xilinx Vivado-based system design. This is a 4-part MP. In this MP, you will:

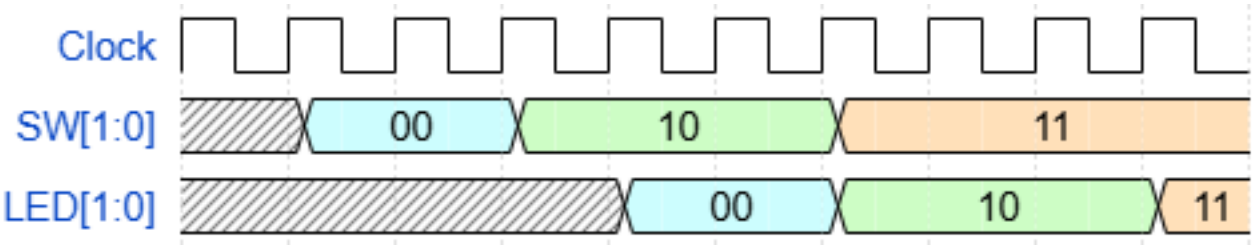
- Get accustomed to FPGA design on Zynq-based systems
- Learn the basics of the Vivado tools
- Get accustomed to SoC design and embedded system design
- Learn the basics of the AXI protocol
- Understand how DMA works

2 The Assignment

2.1 Part A (10%)

Implement a system that reads the states of the GPIO slide switches on the board and displays them on the onboard tri-color LEDs. The Pynq-Z2 board has two slide switches and two tri-color RGB LEDs (specifically LD4 and LD5 on the board). **Your hardware design should be written in Verilog. Also, you must edit the constraints file. There is no software in this part.** *Hint: Consider using a shift register.*

When a switch flips state, there should be a delay (lag) of 3 positive clock edges between when the signal change reaches your module and when the output signal of your module changes. See the timing diagram below.



Additional Constraints:

- Use a 125MHz clock (reference clock on the PL side)
- All outputs must be buffered (the outputs from your module should be tied to registers)
- Your LED cells should be driven with 3.3V logic (check your constraints file)
- When an LED is on, it should have a pure blue colored output.
- Your logic must work for both of the switches and LEDs (1:1 mapping between switches and LEDs)
- Write a testbench that exhaustively tests all combinations of LED switch states.

2.1.1 Demoing Part A

You must show functionality works (matches our specifications) on the PYNQ board and in your testbench.

- Testbench (5%)
- On-Board (5%)

2.2 Part B (15%)

Implement a system that reads the GPIO switch states and displays them on the four regular LEDs onboard. Since there are two slide switches and four regular LEDs (LD0 to LD3), we define the following mappings between the input switches and the state represented by LEDs (as usual, 0 indicates off and 1 indicates on)... Your hardware design should be written in Verilog. You must also edit the constraints file. There is no software in this part.

Switches [1:0] $\rightarrow$ State (represented by output LEDs[3:0])

- 00 $\rightarrow$ 0001
- 01 $\rightarrow$ 0011
- 10 $\rightarrow$ 0111
- 11 $\rightarrow$ 1111

Furthermore, the system will operate in four modes, triggered by the onboard push buttons. Each push button changes the mode.

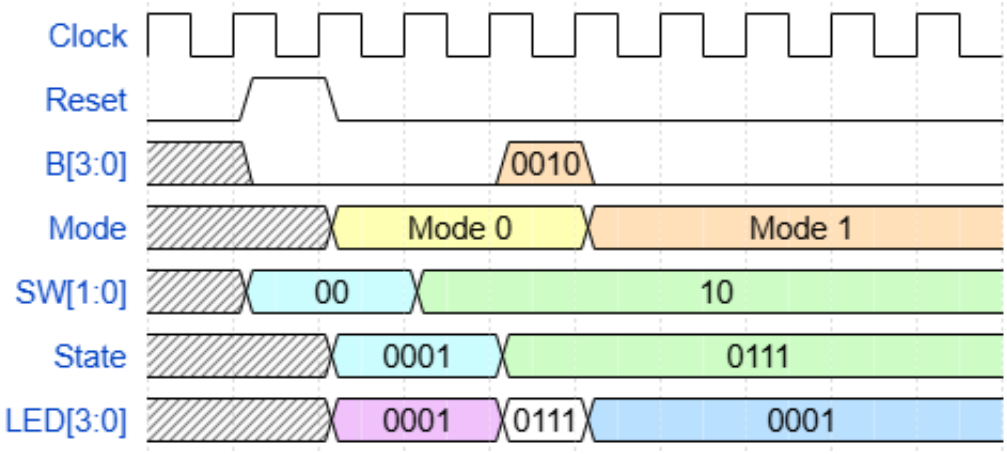
- Mode 0: Display the state of the switches.
- Mode 1: Display the state of the switches, *logically* shifted to the right by 2.
- Mode 2: Display the state of the switches, *circularly* shifted to the left by 3.
- Mode 3: Display the state of the switches, *bitwise* inverted.

This logic must be implemented as a state machine. You should not have to hold down the button to change the mode. After pressing a button, the mode should persist until a new button is pressed.

Use the following:

- A 125MHz clock (reference clock on PL side).
- All outputs must be buffered.
- You should generate a synchronous system reset. On its positive edge, initialize the system to Mode 0.
- Drive LED cells with 3.3V
- Push button 0 sets the system to Mode 0, push button 1 to Mode 1, and so on.
- Any changes to the inputs to the module (switches and push buttons) must be processed in a single cycle. In other words, if the input signals change on cycle 0, the buffer register tied to your module outputs should have their new value on cycle 1.

The waveform below may help in understanding the above requirements.



2.2.1 Demoing Part B

You must show functionality works (matches our specifications) on the PYNQ board and in your testbench.

- Testbench (10%)
- On-Board (5%)

2.3 Part C (20%)

Implement a Zynq-based SoC system. Now, we will control the LEDs and read the state of the switches in the software. We will automatically rotate between system Modes instead of reading from the push buttons.

You are expected to:

1. Connect the LEDs and the Switches via AXI
2. Write software code to:
   - **a** Read the switch positions via AXI
   - **b** Write to the LEDs via AXI
   - **c** Write your code in a "super-loop"
     - **i** Cycle through the 4 modes described in Part B, with a 1-second latency between each mode.
     - **ii** **Do not read or use the push buttons – the system should automatically cycle between the modes**
     - **iii** It is okay if flipping a switch does not cause a visual change until the next mode update.

Requirements:

- Unlike the previous parts, you will need to write software. Your software should be written in C.
- Unlike the previous parts, you will use the IP integrator for your hardware on the FPGA. See the tutorial for the basics.
- Ensure your constraints file is set up properly.

2.3.1 Demoing Part C

- (20%) On-Board

2.4 Part D (25%)

This part builds on Part C. We will go further into how information can be moved around the SoC. You will learn Direct Memory Access (DMA) and two Advanced eXtensible Interface (AXI4) protocols: AXI4-Stream (AXIS) and AXI4-Lite (AXIL).

You are an engineer working for an archaeological company named "*Pastword Recovery.*" Your company hunts for ancient artifacts of antiquity. Your boss is a huge fan of Julius Caesar and has recently uncovered a secret cache of historical documents. Unfortunately, these never-before-seen writings appear to make no sense!

Here's a sentence from a letter you found:

> "ovd vmalu kv fvb aopur hivba aol yvthu ltwpyl"

This stumps you for a while, so you consult a huge history nerd friend of yours named Scott. He explains to you that Julius Caesar invented a simple cipher that is now called the Caesar Cipher. It is very simple and easily broken by modern computers.

Here is how it works. Start by defining your alphabet. Here, our alphabet will be the set of all lowercase letters in the English language.

> "abcdefghijklmnopqrstuvwxyz"

Next, assign each letter a numerical index indicating its position in the alphabet. This is done in order, so 0→a, 1→b, ... 25→z.

To encrypt your data, add the cipher shift amount to the index of each letter in our original string. If you go past the bound (25), rotate back to the beginning. To decrypt, you simply subtract.

For example, to apply a Caesar Cipher with a shift of 3 to the letter "a", we would first figure out the index of the letter "a" (0), add 3 to that (0 + 3 = 3) and then decode that back to our alphabet to get the letter "d".

For a larger example, assume we want to use the Caesar Cipher on the following message:

> "how often do you think about the roman empire"

If you perform the cipher with a shift value of 7, you will get the seemingly nonsense message from earlier in the problem description!

2.4.1 AXI Protocols

Advanced eXtensible Interface (AXI) is a popular protocol used in multiple domains, including AMD FPGAs. There are multiple versions to the specification, but we will use AXI4 in this class.

We'll start be defining AXI4 interfaces. This knowledge is generally applicable to all AXI4 variants. AXI4 interfaces are either master **(M)** or slave **(S)** interfaces (some newer document versions might label these as manager and subordinate). **M** interfaces initiate transactions (reads/writes) and **S** interfaces respond to them.

AXI4-Stream (AXIS):
Here, we will describe the basics of AXIS, but there are variants of and nuances that we will not discuss. You can find the full documentation on this page (remember we are using AXI4).

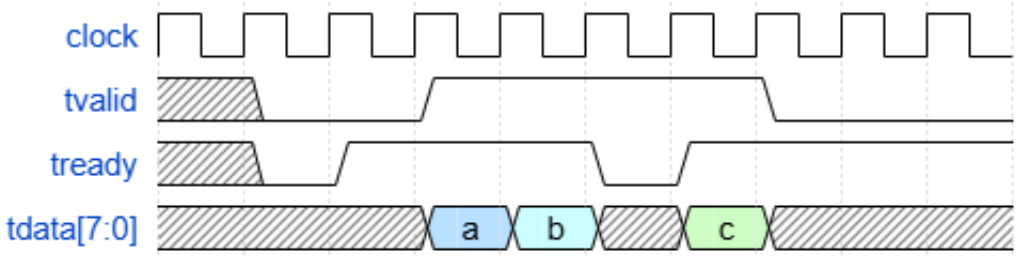
AXIS is the simplest AXI protocol. It supports only writes. The **M** interface writes to the **S** interface.

AXIS transfers take place over 1 or more **beats**. A **beat** is simply a cycle in which data transfer occurs. A data transfer occurs whenever *tvalid* and *tready* are both 1. *tvalid* is driven by the **M** interface and *tready* is driven by the **S** interface.

tdata is a bus of flexible bitwidth that actually carries the data to be transferred and is driven by the **M** interface.

The above comprises the bare minimum signals for AXIS, but some IPs use additional signals.

See the diagram below for an example. It shows 3 valid AXIS writes.



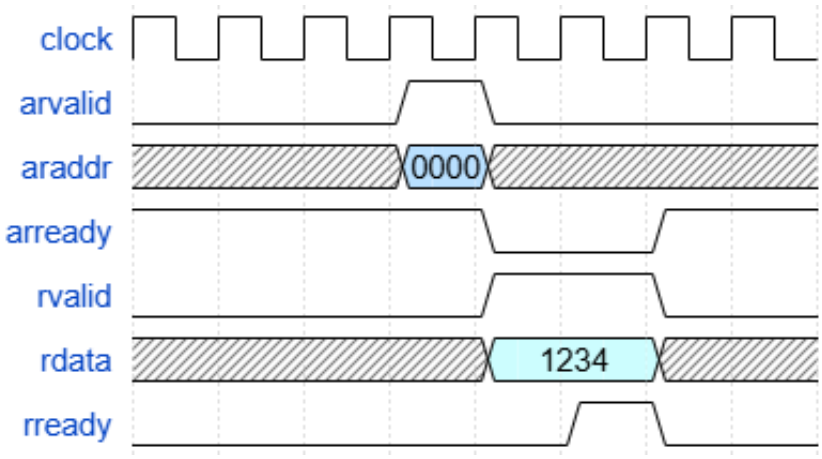
Other signals also exist in AXIS, if you encounter them, you may need to read up on how they work.

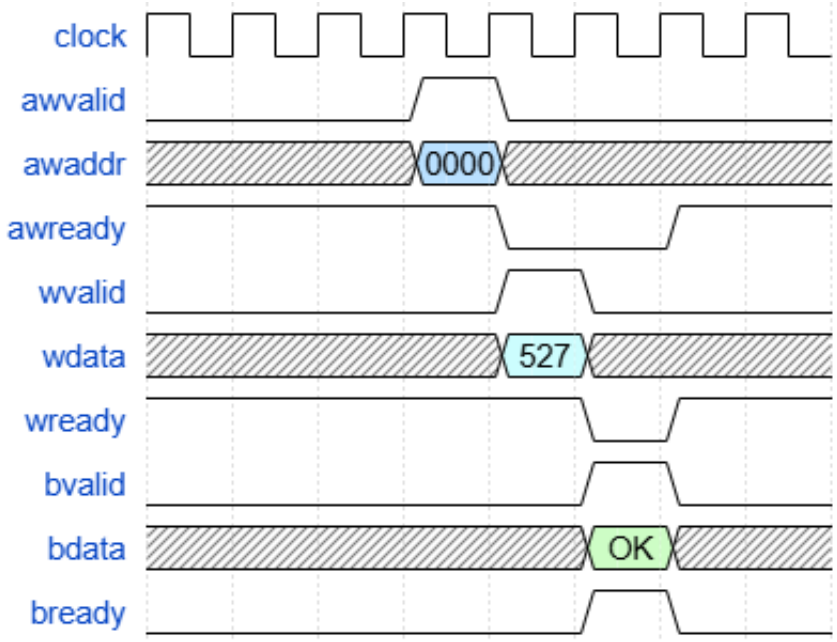
AXI4-Lite

AXIL is a little more complicated than AXIS. AXIL supports both reads and writes and is comprised of 5 **independent** channels. Each channel has unique signals (including valid/ready). Multiple channels can assert valid/ready simultaneously. AW and W can be asserted in either order (but a write will only be able to complete once AW and W have both been set by **M** and accepted by **S**). AW, W, and AR are completely independent (can be asserted in any order).

1. **Address Write (AW)**: The **M** interface specifies the address to write to and sets valid high. The **S** interface drives a ready signal when it is able to accept the address.
2. **Write Data (W):** The **M** interface writes the data and asserts valid. The **S** interface asserts ready when it can accept the data.
3. **Write Response (B):** Once the write has finished, the **S** interface sets BRESP (usually to OKAY [2'b00]) and asserts valid. the **M** interface asserts ready when it has accepted the response.
4. **Address Read (AR):** The **M** interface specifies a target ready address and asserts valid. The **S** interface asserts ready when it has accepted the address.
5. **Address Data (A):** The **S** interfaces drives the data bus and a valid signal. The **M** interface raises ready when it has accepted the data.

Example waveforms for reads and writes are shown below. The read grabs data from address 0. The **S** interface responds and the value is the number 1234. The write operation targets address 0 with the value 527.





2.4.2 Direct Memory Access

We will now explain Direct Memory Access (DMA). DMA is a critical feature in a huge number of devices and systems (and you will use it in this part of the MP!) so we will briefly explain how it works. Throughout this section we will ignore virtual memory. We won't need to worry about it in this class, but it can complicate DMA significantly.

The primary objective of DMA is data movement between devices and/or parts of a SoC. PCIe Devices such as GPUs, NICs, etc. all typically come equipped with DMA engines capable of high bandwidth data movement.

The DMA engine is controlled via *descriptors*. Descriptors describe how data is to be moved. See this link for information on how AXI DMA descriptors are formatted. Multiple descriptors are sometimes needed to describe a data transfer. Descriptors are handled sequentially.

The most important fields for us are the buffer address (where the data is located for reads, where it should be placed for writes), length (how much data to move), and next descriptor fields (a pointer to the next descriptor to process). You may need to modify other fields to get things working properly.

The AXI DMA engine supports both moving data from RAM to the PL (reads) and from the PL to RAM (writes). Separate descriptors are used for reads/writes.

During the read flow, the PS first sets up the read descriptor(s) to be used. It then tells the DMA engine where to find the first one. The first descriptor is grabbed from host memory by the DMA engine. It describes where and how much data to transfer from RAM to the PL. Using the descriptor, data is grabbed from RAM and shoved out of the AXI DMA **M** streaming interface. In addition, the descriptor includes a next descriptor pointer. This is used to fetch the next descriptor from memory. The process is then repeated.

Writes work similarly, but the direction is reversed. The PS still sets up the descriptors, but this time they describe where to place data into RAM. When data comes in over the AXI DMA slave interface, the DMA engine will write that data to RAM locations described by the descriptors.

2.4.3 The Task

You are tasked by your boss to build a Caesar Cipher hardware accelerator.

Your accelerator will have 3 AXI4 interfaces: one AXIS **S** interface for taking in a sequence of characters, one AXIS **M** interface for outputting the encrypted/decrypted data, and one AXIL **S** interface to enable the host to control the shift size. Your accelerator must support both encryption and decryption. A full summary is below.

(You might find this ASCII tale helpful ASCII table.)

Requirements Summary:

- Your accelerator must have 3 interfaces: AXIS **S**, AXIS **M**, AXIL **S**

  AXIS S: 32-bit wide bus that receives chunks of 4 characters that need to be shifted.

  AXIS M: 32-bit wide bus that writes all 4 shifted characters.

  AXIL S: 32-bit wide bus that can be used to read/write a register that controls the amount each character is shifted by. This should be interpreted as a signed value by your code to enable both encryption and decryption.

- A DMA engine is used to move data to/from host RAM. It is controlled by the CPU on the SoC PS.
- Your PS code should:

  i Take in two inputs over UART. The first is a 32-bit signed number that sets the shift amount. The second is a character array of at most 256 characters to be Caesar Cipher encrypted/decrypted.

  ii Run the accelerator on that string. You will need to set up the DMA engine. You need to create the DMA descriptors for moving the character array to the PS RAM and the descriptors for moving the shifted data back to the PS RAM.

  iii Write the shifted string back to the terminal over UART.

  iv The above should run in a loop so multiple strings can be quickly tested.

General guidelines to accomplish this task are described below. Note that we are not as explicit on how to accomplish every single step. This is by design. We intend for you to use the documentation to learn how to finish this MP. That is a critical skill for any SoC/hardware/embedded engineer.

1. Write the RTL for your accelerator and get it into your block design (See here).
2. Instantiate and configure the ZYNQ PS block
   - Modify the PS block to enable the AXI High-Performance Ports
   - Enable interrupts from the PL to PS
3. Instantiate the AXI DMA engine (See here).
   - Disable the control/status ports
   - Enable scatter/gather mode
4. Run the block design auto-connect
5. Add your accelerator to the block design.
6. Connect the accelerator to the block design and wire it into the DMA engine.
7. Export the design to the SDK.
8. Use the DMA examples in the SDK to learn how to use the DMA loop and modify it to work with your accelerator.
9. Use the UART examples to learn how to take in input and write output to your computer's terminal.

2.4.4 Demoing Part D

Your TA will give you a 5 different strings and shift amounts to test. You will receive 5% for each correctly encrypted/decrypted string.

3 General Demo Requirements

You will need to demo your project. Demos open the day the MP is released and end 7 days **after the MP is due**. You must come to one of the TA office/lab hours to demo. If in person, you will need your board and your designs. If online, you will need a camera so we can see your board and your design working on your board.

You can demo as many times as you like before the assignment is due! However, if you demo after the submission is due (during the 7-day window), you must download and run your design as it was when submitted to Canvas. Thus, you only have one attempt and it must be on your Canvas-submitted code. Therefore, it is to your **significant advantage** to demo **before** the due date.

4 The Report (30%)

You will need to write a report for this MP. In the report, you must include:

1. Full Name(s) and NetID(s) of team members
2. For each part (A, B, C, and D) you must include (5% for each part):

   a A functional block diagram that should explain the functionality and timing of the system. (2.5%)

   i You need not represent every gate; that would be a circuit diagram. A block representing a large register, a bundle of combinational logic, or even a module will suffice. (Use unique symbols for flops, combinational logic, and module blocks).

   ii Pseudo-code/flow charts can further explain the software if you feel that words are insufficient.

   iii All arrows should be labeled with the information they carry and the protocol (if any) used to carry that information.

   iv DO NOT SCREENSHOT VIVADO. Your must draw your diagrams by hand or using software tools. Draw.io, Powerpoint, etc. are acceptable tools.

   b A list of the entities/modules in your design, including concise explanations of what each does and some features. (1%)

   c A description of your design process - explain how you met the required specifications and include any calculations. (0.5%)

   d The post-implementation results for resource utilization, power, and timing. Put these into a well-formatted table. (0.5%)

   e What you learned, what you thought was tricky, and what you thought was easy. (0.5%)

   f Any extra feedback or notes (optional)

   In addition, for part D, answer the following questions (5%):

   a Answer the following about the protocols we used in Part D.

   i Why did our accelerator use two different protocols? What about the protocols makes them well-suited to our use-case? (1%)

   ii Look up the full AXI4-MM protocol. What features does it possess? Why did we use AXIL instead of full AXI4 for the control interface in part D? (1%)

   iii What part of the block diagram did use full AXI4? (1%)

   b You are a hardware engineer designing an SoC. You are creating an IP for calculating vector dot products.

   – Here, we are defining the dot product as $c = \sum \boldsymbol{a}_i \boldsymbol{b}_i$ where a, b are vectors, i is an index into that vector, and c is a scalar. In other words, calculate a running sum of the elementwise product of two vectors.

   – Each element in a and b are 32-bits. c is a 32-bit integer.

   – The accelerator should have an AXIS input. Data should be streamed in the pattern a[0], b[0], a[1],b[1], ...

   – The accelerator continuously updates an AXIL accessible register with the new value of the dot product. Every time an individual elementwise multiplication and accumulate operation is performed, update that AXIL accessible register.

   – This AXIL accessible register can also be overwritten by the host processor with a value.

   – A DMA engine is used to move data from CPU RAM to the FPGA.

   Do the following:

   i By hand or with a drawing software tool, draw (do not implement) a functional block diagram for the system. Include the internals of the accelerator. Assume there are no other accelerators that need to be included. (1%)

   ii Write psuedo-code that explains how the processor would use this accelerator. (1%)

5 Submission

5.1 Grading Criteria

- Functionality (70%) – Correctness

  b Part A (10%) - graded via Demo

  c Part B (15%) - graded via Demo

  d Part C (20%) - graded via Demo

  e Part D (25%) - graded via Demo

- Report (30%)

  a Report Quality (5%) – Grammar, spelling, formatting, etc.

  b Part A Report Requirements (5%)

  c Part B Report Requirements (5%)

  d Part C Report Requirements (5%)

  d Part D Report Requirements (10%)

- We will dock you a **minimum** of 10% if you do not follow file submission formatting instructions (see below).

5.2 Uploading your Project

You will submit to Canvas. Only one partner in the group needs to submit. You must submit the following for this project:

- A consolidated report document with write-ups for all the requirements described in the Report Section (the document should be a PDF).
- A zip file for each part of the MP (4 total zip files)

To export a project from Vivado:

1. File → Project → Archive Project
2. Under Archive options make sure both are selected:

   Include Configuration Settings

   Include Run Results

We should be able to extract your project, open it, put the bitstream onto the FPGA, and see it work. We should also be able to recompile your project if we so choose.

Your entire submission will be in a single zip file submitted with the file named:*lastname1_lastname2_mp1.zip*
For example, assume Scott Smith and Hanchen Ye were in a group. Their first submission would be *smith_ye_mp1.zip*.

The file structure within that zip file should be:

smith_ye_mp1.zip

- smith_ye_mp1_report.pdf
- mp1a.zip
- mp1b.zip
- mp1c.zip
- mp1d.zip