# Lab Member

Table Number : <Insert your table number here>

Section: Select section

| Name | Student Number |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

# 

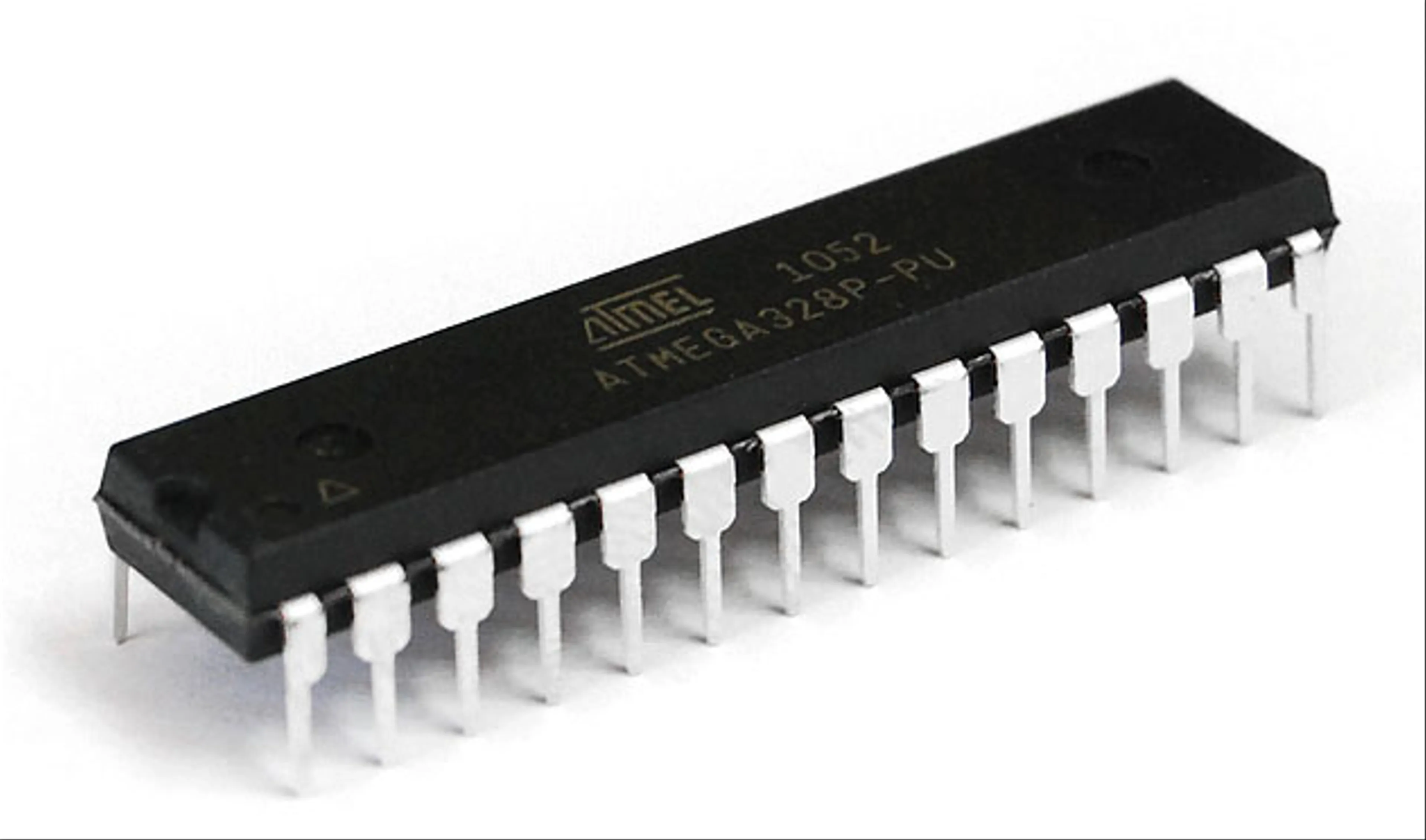
# 

# **Objectives**

1. To understand ROM and RAM.
2. To learn how to generate and instantiate ROM and RAM with values.
3. To get familiar with the IP Catalog.

# **R**ead-Only Memory (ROM)

Read-Only Memory (ROM) is non-volatile memory used to store fixed data, such as firmware, look-up tables, or initialization code, that remains intact even when the power is off. ROM is ideal for applications where data does not need to be modified frequently, ensuring reliability and consistency. It works by storing preloaded data in memory cells, which can be read but not altered during normal operation. In FPGA designs, ROM is efficiently implemented using embedded memory blocks, reducing resource usage compared to using registers or flip-flops.

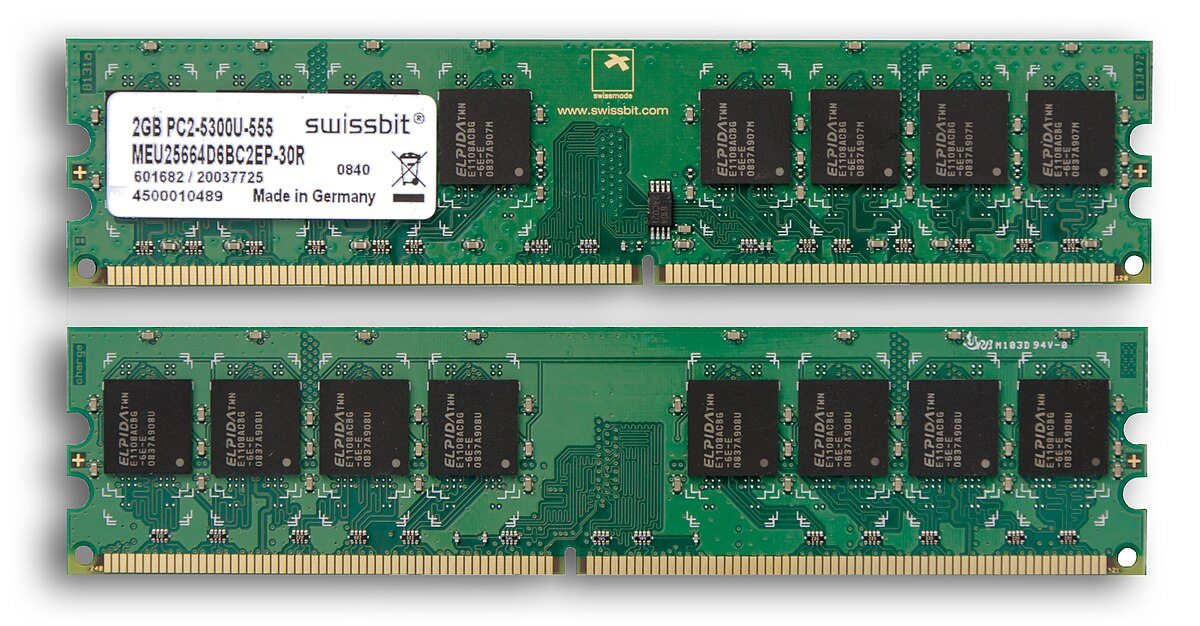


**Figure 1: Read-Only Memory (ROM)**

# 

# Random Access Memory (RAM)

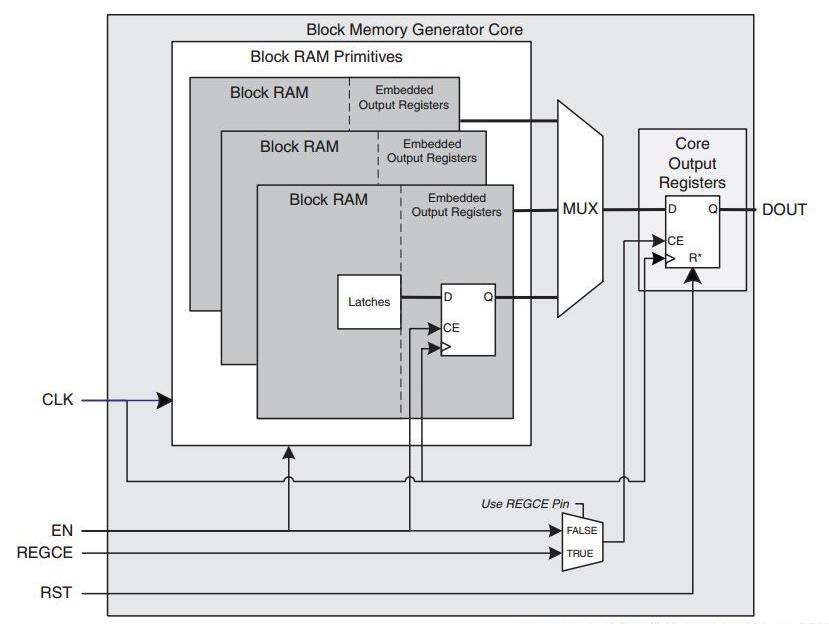
Random Access Memory (RAM) is a volatile memory used for temporary data storage, allowing both read and write operations. Like ROM, RAM could be implemented using registers, but this approach is resource-intensive and inefficient. To address this, FPGAs include dedicated memory blocks, such as BlockRAM in Xilinx or Memory Blocks in Altera, providing an efficient and cost-effective way to implement RAM within FPGA designs.



**Figure 2: Random Access Memory (RAM)**

# Block RAM

Block RAM (BRAM) is a dedicated memory resource in FPGAs, optimized for efficient memory implementation without using general-purpose logic like CLBs. BRAM sizes range from 10-20 Kbits, with FPGAs containing dozens to thousands of them. In Xilinx, BRAMs can be inferred automatically during synthesis if the HDL code follows specific patterns.



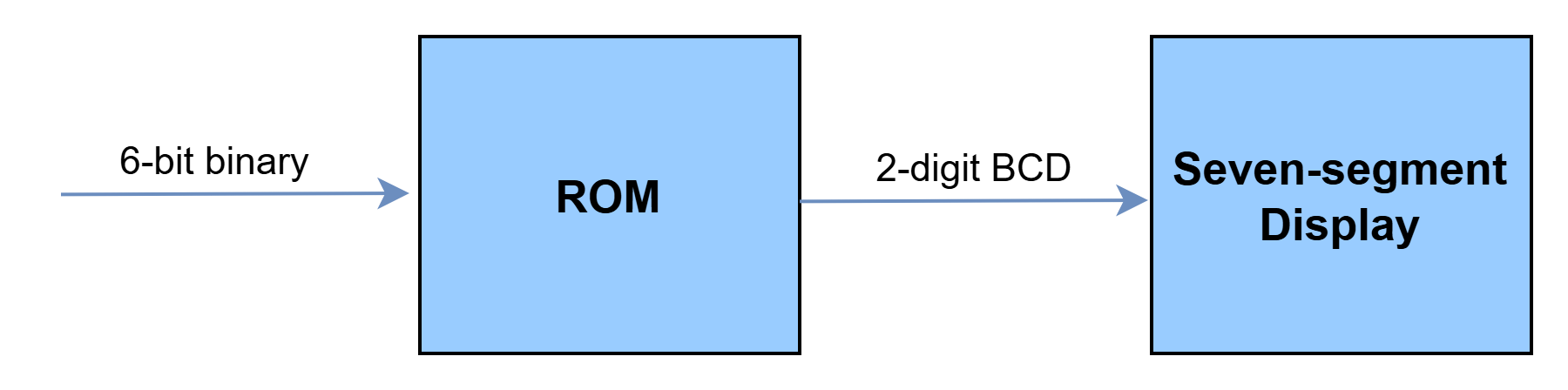
**Figure 3: Block RAM**

# IP Catalog

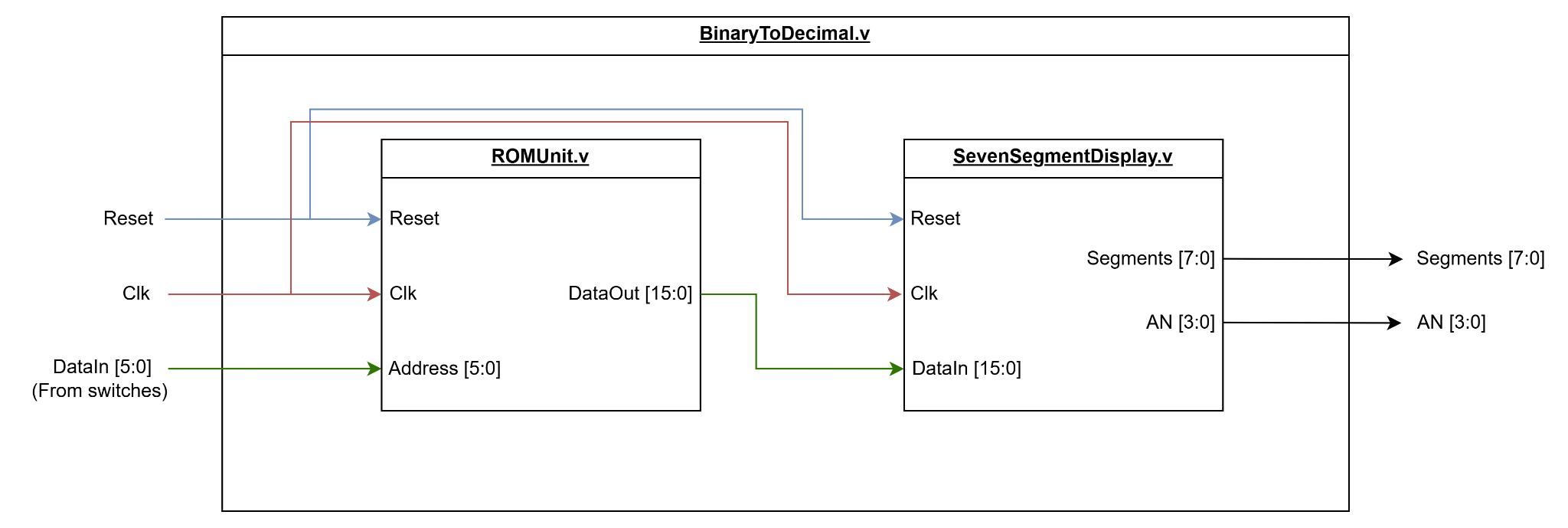
The Xilinx IP Catalog is a feature in Vivado that provides a library of pre-made building blocks, called IP cores, that you can use in your designs. These IP cores save time by letting you use tested and ready-to-use components instead of designing everything from scratch. It’s like a toolbox where you can pick and configure parts for your project, such as memory controllers, arithmetic blocks, or communication interfaces.

# **Lab Exercise** 1

In this exercise, you will implement a BCD converter that reads inputs from 6 binary switches and uses read-only memory (ROM) for converting 6-bit binary to 2-digit BCD for displaying on a seven-segment display.



**Figure 4: Overview Idea of System**

****

**Figure 5: System Overview**

Figure 5 illustrates the system overview to be implemented in this lab exercise. The system consists of 3 modules, each with a specific role as described below:

1. **BinaryToDecimal**: This is the top-level module responsible for integrating and connecting ROM and SevenSegmentDisplay.
2. **ROMUnit**: The read-only memory which stores 2-digit BCD.
3. **SevenSegmentDisplay**: Displays the value on the 7-segment display. This module is already implemented.

## 

## **Part 1 :** ROM Module

Our read-only memory (ROM) stores 2-digit BCD for each 6-bit binary. Therefore, we must define our ROM as a lookup table in which the address represents 6-bit binary and the output data represents 2-digit BCD.

* The data represents 2-digit BCD, which each digit uses 4 bits. Therefore, the data must be 8 bits.
* For the address, the 6-bit binary number requires 26 = 64 addressable locations (0 to 63).

| **Address** | **Decimal Equivalent** | **BCD Output** |
| --- | --- | --- |
| 000000 | 0 | 0000 0000 |
| 000001 | 1 | 0000 0001 |
| 000010 | 2 | 0000 0010 |
| … | … | … |
| 111110 | 62 | 0110 0010 |
| 111111 | 63 | 0110 0011 |

For our **ROMUnit** module, the module has the following ports:

* **Inputs**:
  + Reset: Reset signal.
  + Clk: Clock signal for synchronization.
  + Address: 6-bit address for looking up in the lookup table.
* **Output**:
  + DataOut: 8-bit BCD output data.

#### 

#### Instruction

1. Create a new project and import all the necessary files from the following source: <https://github.com/2110363-HW-SYN-LAB/lab/tree/main/Lab4>
2. Create a ROM module for storing our BCD data. **Choose one of the following methods** to create ROM module:

**Method 1: Writing your own ROM module**

* Create your memory file (.mem file) which will contain data stored in each address. For example, line 1 of the file represents data stored at address 0.

Hint: You can read how a .mem file looks like in Lab 4 Guide at “How to initialize ROM value” section (path 1).

* Modify the provided ROMUnit module to initialize the memory and make it work as described above.

Hint: You can initialize memory by using $readmemb(...) or $readmemh(...) functions.

| // Initialize memory using binary data  $readmemb("<bin\_memory\_file\_name>", memory\_array)  // OR  // Initialize memory using hexadecimal data  $readmemh("<hex\_memory\_file\_name>", memory\_array) |
| --- |

You can read how to create and initialize memory in Lab 4 Guide at “How to initialize ROM value” section (path 1).

**Method 2: Generating ROM module using IP Catalog**

In case you are lazy at writing modules from scratch, you can generate the modules using IP Catalog. Generate the ROM module using IP Catalog by selecting “**Block Memory Generator**” in IP Catalog

* Make sure that you select the right memory type (Single Port ROM).
* Adjust **port width** (length of each data), **port height** (total number of addressable location), and **enabled port type** correctly in **Port A Options**.
* Adjust the reset signal behavior in **Port A Output Reset Options** in **Port A Options**.
* Initialize the memory in **Memory Initialization** tab in **Other Options**.

Hint: You can read how to generate ROM and initialize memory (.coe file) using IP Catalog in Lab 4 Guide at “How to initialize ROM value” section (path 2).

Submit the evidence of your ROM module.

* If you write your own ROM module (method 1), paste ROMUnit module code here.
* If you generate ROM module using IP Catalog (method 2), capture the screenshot of IP Symbol of your module in Block Memory Generator.

|  |
| --- |

In case you are writing your own ROM module (method 1), run the testbench to verify your ROMUnit module using either Xilinx or CocoTB and capture your testbench result and waveform. (You can skip this if you use IP Catalog)

[If you use CocoTB, don’t forget to copy memory file (.mem) and paste inside CocoTB directory]

|  |
| --- |

## Part 2 : Top-level Module and Hardware Programming

Now, we will integrate **ROMUnit** and **SevenSegmentDisplay** modules together such that the output of ROM can be sent to a seven-segment display to show in the Basys3 board.

For the **BinaryToDecimal** module (top-level module), The module has the following ports:

* **Inputs**:
  + Reset: Resets the system.
  + Clk: Clock signal for synchronization.
  + DataIn [5:0]: The 6-bit binary value which will be converted to 2-digit BCD.
* **Output**:
  + Segments [7:0]: Signal that is used to control cathode of the 7-segments display.
  + AN [3:0]: Signal that is used to control anode of the 7-segments display.

The port mappings from the **BinaryToDecimal** module (top-level module) to Basys3 board are as follows:

| **BinaryToDecimal ports** | **Basys3 board** |
| --- | --- |
| Reset | BTNC |
| Clk | Basys3 100Mhz Clk |
| DataIn [5:0] | {SW5, SW4, SW3, SW2, SW1, SW0} |
| AN [3:0] | {AN3, AN2, AN1, AN0} |
| Segments [7:0] | {CA, CB, CC, CD, CE, CF, CG, DP} |

#### 

#### Instruction

1. According to Figure 4 and 5, integrate both **ROMUnit** and **SevenSegmentDisplay** modules together in the **BinaryToDecimal** module (top-level module).

Insert your modified BinaryToDecimal module here.

|  |
| --- |

1. From the table above, complete this mapping table below to map your system's inputs and outputs onto the Basys3 pins.

Hint #1: Look up the [Basys3 reference manual](https://digilent.com/reference/_media/reference/programmable-logic/basys-3/basys3_rm.pdf) on page 7 and 15.

Hint #2: Many pin mappings are similar to Lab 1-3 exercises.

Complete the mapping table.

| **Top-level module input/output** | **Basys3 Pins** |
| --- | --- |
| Reset |  |
| Clk |  |
| DataIn[0] |  |
| DataIn[1] |  |
| DataIn[2] |  |
| DataIn[3] |  |
| DataIn[4] |  |
| DataIn[5] |  |
| AN[0] |  |
| AN[1] |  |
| AN[2] |  |
| AN[3] |  |
| Segments[0] |  |
| Segments[1] |  |
| Segments[2] |  |
| Segments[3] |  |
| Segments[4] |  |
| Segments[5] |  |
| Segments[6] |  |
| Segments[7] |  |

1. Create the constraint file.

Submit your constraint file here.

|  |
| --- |

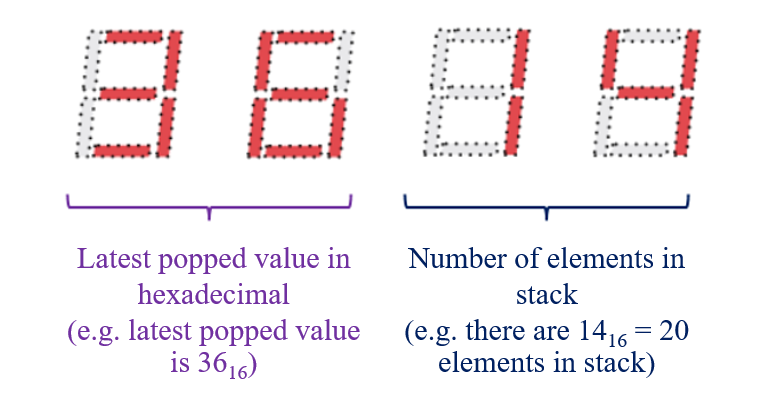
1. Run the synthesis.
2. Run the implementation, generate the bitstream, and program the **Basys3** device.
3. [Checkpoint] Call TA to inspect your work.

# 

# Lab Exercise 2

In this exercise, we will design and build a circuit to work as a **Stack** (LIFO). The user can use two push buttons in order to PUSH or POP. We will use 8 switches on the board as a value. When the user hits a PUSH button, it will store the value from the switches to the stack. When the user hits the POP switch, it will display the popped value in the 2 hexadecimal digits on the left of a seven-segment display. The other 2 right digits in the seven-segment display are used to display the number of elements currently in the stack.

* The stack can keep up to 256 elements.
* If the stack is empty, hitting POP should not do anything.
* If the stack is full, hitting PUSH should not do anything.
* Pressing the RESET button will reset the stack and system.



**Figure 6: Example of Seven-segment Display in System**

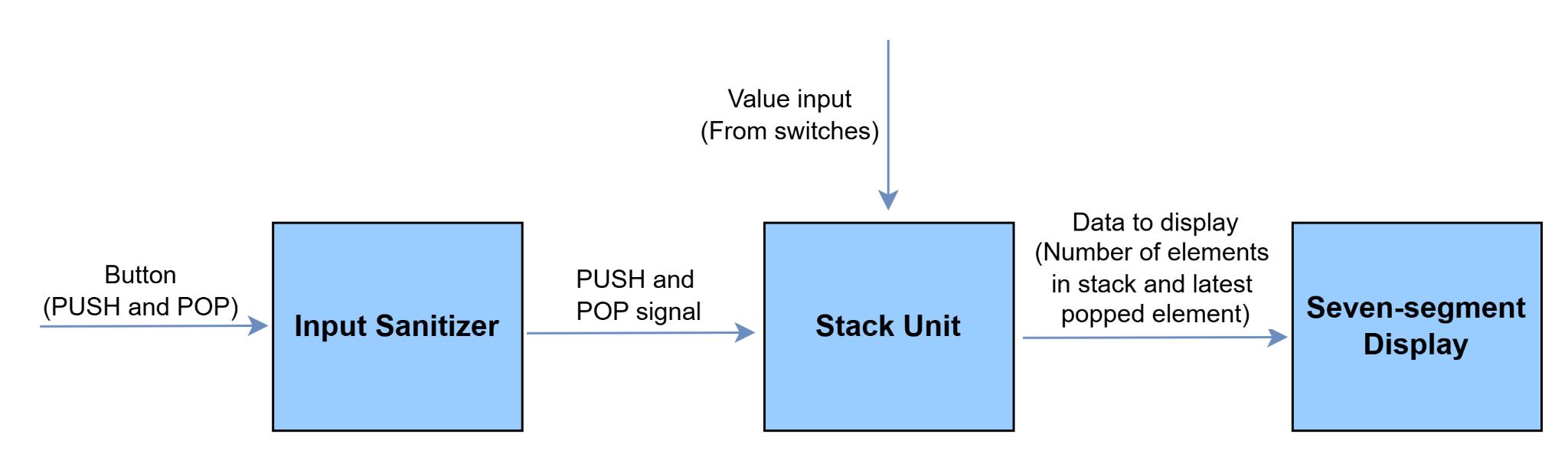
Figure 7 shows the example of pushing and popping elements into the stack. Note that the 2 left digits of the seven-segment display will be changed only when the stack is popped.

## 

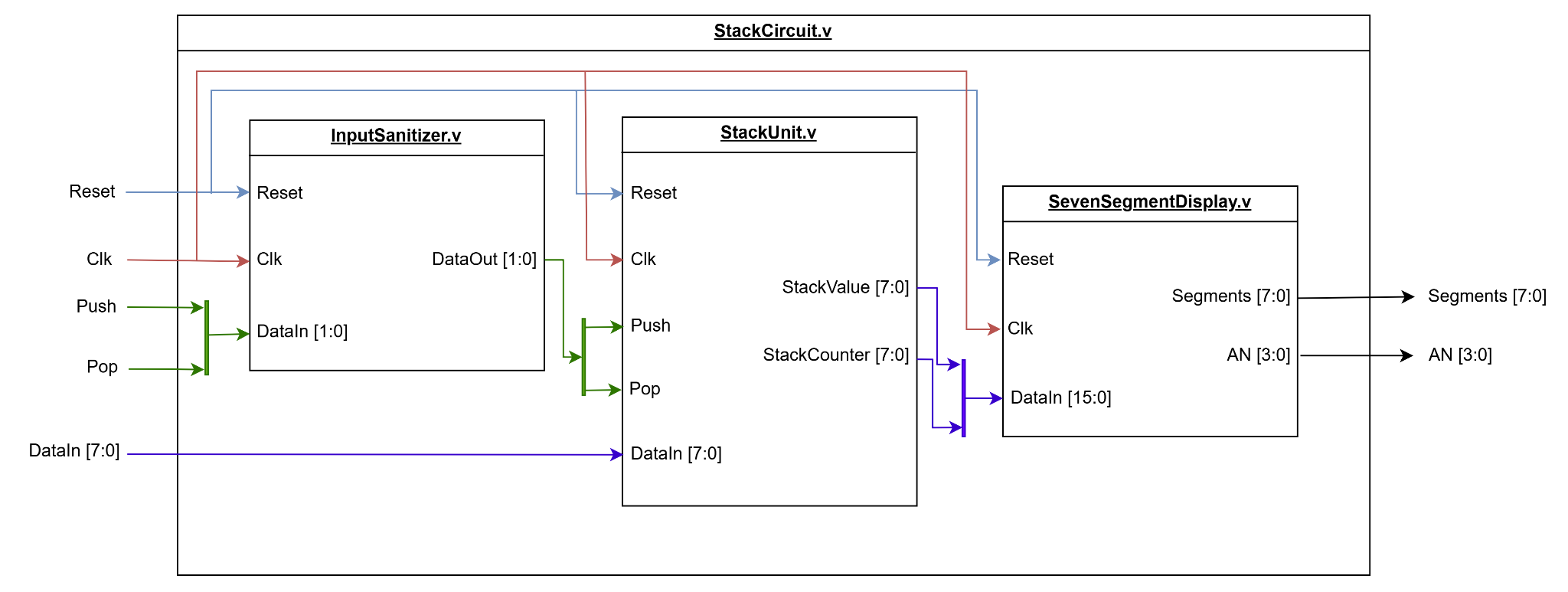
## 

**Figure 7: Example of Pushing and Popping Elements**

For our design, which is shown in Figure 8, we will sanitize the signals from buttons (PUSH and POP signal) then feed these signals with inputs from switches to a stack unit, which will handle operations in stack. In addition, the data from the stack unit will be fed to a seven-segment display to show the number of elements and the latest popped value of our stack.



**Figure 8: Overview Idea of System**



**Figure 9: System Overview**

Figure 9 illustrates the system overview in this lab exercise. The system consists of 4 main modules, each with a specific role as described below:

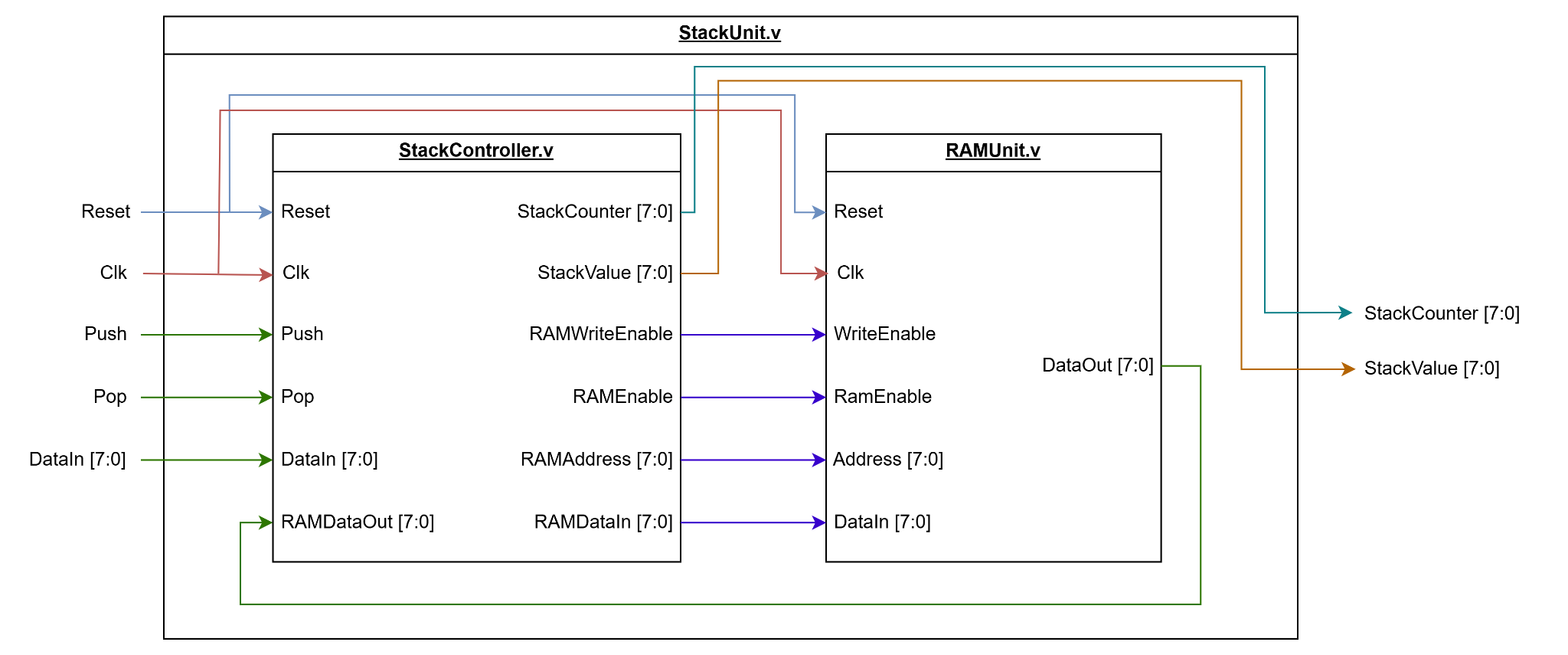
1. **StackCircuit**: This is the top-level module responsible for integrating and connecting input sanitizer, stack unit, and seven-segment display.
2. **InputSanitizer**: Module which eliminates noise and generates a clean single pulse.
3. **StackUnit**: Module which performs operations and stores elements of stack.
4. **SevenSegmentDisplay**: Displays the value on the 7-segment display.

## Part 1 : Stack Module

First, we will implement the stack module which will perform operations and store elements of stack.

* The module must have something which can store elements for our stack, which we will use random-access memory (RAM) to be the data storage.
* When PUSH signal are triggered, element must be added into RAM.
* When POP signal are triggered, last element must be removed from RAM.
* When RESET signal are triggered, all data in RAM must be cleared and everything must be reset.

In the design (Figure 10), **StackController** module has responsibility for controlling all logic about stack, and **ROMUnit** has responsibility for element storage for our stack.



**Figure 10: Design of StackUnit**

### 

### Part 1.1 : RAM Module

We must implement a RAM module to store our elements in the stack. Our requirements are:

* The RAM module shall store 256 elements for the stack.
* Each element can be represented as 8-bit value or 2-hexadecimal digits.

#### Instruction

1. Create a RAM module to be our data storage for the stack. **Choose one of the following methods** to create RAM module:

**Method 1: Writing your own RAM module**

Modify the provided RAMUnit module to initialize the memory and make it work as described above.

Hint: You can read how RAM actually works in the Lab 4 Guide at the “RAM behavior” section.

**Method 2: Generating RAM module using IP Catalog**

In case you are lazy at writing RAM from scratch, you can generate the modules using IP Catalog. Generate the RAM module using IP Catalog by selecting “**Block Memory Generator**” in IP Catalog

* Make sure that you select the right memory type (Single Port RAM).
* Adjust **port write/read width** (length of each data), **port write/read height** (total number of addressable location), and **enabled port type** correctly in **Port A Options**.
* Select **Primitives Output Register**.
* Ensure that **RSTA Pin** is enabled, **Output Reset Value** is 0, and **Reset Priority** is CE (Latch or Register Enable).

Hint: You can read how to generate RAM using IP Catalog in Lab 4 Guide at “Random-Access Memory” section.

Submit the evidence of your RAM module.

* If you write your own RAM module (method 1), paste RAMUnit module code here.
* If you generate RAM module using IP Catalog (method 2), capture the screenshot of IP Symbol of your module in Block Memory Generator.

|  |
| --- |

In case you are writing your own RAM module (method 1), run the testbench to verify your RAMUnit module using either Xilinx or CocoTB and capture your testbench result and waveform. (You can skip this if you use IP Catalog)

|  |
| --- |

### 

### 

### Part 1.2 : StackController

It’s time to implement the heart of our system, the StackController module. This module will perform every logic about stack such as pushing and popping elements in stack. Remind that our requirements are:

* When PUSH signal are triggered, element must be added into RAM.
* When POP signal are triggered, last element must be removed from RAM.
* When RESET signal are triggered, all data in RAM must be cleared and everything must be reset.
* The number of elements in the stack and the latest popped element must be returned, so that we can feed these values to a seven-segment display.

#### Instruction

1. From the requirements above, discuss in your group about the idea for implementing this **StackController** module (e.g. how can we retrieve the top element of stack, how to push and pop element, …).

Hint: If you are not familiar with stack or you are stuck on something here, you can find some ideas by watching [this video](https://www.youtube.com/watch?v=2_ngOO8j2Fo&list=PLW3DcQsnGanPGhY2Y0A9hc45KnfS55RZI&index=50) or reading Lab 4 Guide at “Stack from RAM” section.

Write down your idea for implementing StackController briefly.

|  |
| --- |

1. From your idea, implement this module by modifying the **StackController** module.

Insert your modified StackController module here.

|  |
| --- |

1. [Optional] Create a testbench (StackControllerTB in cocotb/StackController/TB.py) for the StackController Module. Your testbench must perform all possible operations of the stack. You can use either Xilinx or CocoTB.

Insert your testbench here.

|  |
| --- |

1. [Optional] Run your testbench to verify your module.

Insert your testbench result here.

|  |
| --- |

Insert your testbench waveform here.

|  |
| --- |

## Part 2 : Top-level Module and Hardware Programming

You are almost done. Last thing we will do is to integrate all modules together in the top-level module.

For the **StackCircuit** module (top-level module), The module has the following ports:

* **Inputs**:
  + Reset: Resets the system.
  + Clk: Clock signal for synchronization.
  + Push: Signal for pushing input into stack.
  + Pop: Signal for popping the top elements of the stack.
  + DataIn [7:0]: The 8-bit binary value which will be input of the stack when pushing.
* **Output**:
  + Segments [7:0]: Signal that is used to control cathode of the 7-segments display.
  + AN [3:0]: Signal that is used to control anode of the 7-segments display.

The port mappings from the **StackCircuit** module (top-level module) to Basys3 board are as follows:

| **StackCircuit ports** | **Basys3 board** |
| --- | --- |
| Reset | BTNC |
| Clk | Basys3 100Mhz Clk |
| Push | BTNR |
| Pop | BTNL |
| DataIn [7:0] | {SW7, SW6, SW5, SW4, SW3, SW2, SW1, SW0} |
| AN [3:0] | {AN3, AN2, AN1, AN0} |
| Segments [7:0] | {CA, CB, CC, CD, CE, CF, CG, DP} |

#### 

#### Instruction

1. In the **StackUnit**, wire up **RAM** and **StackController** together according to Figure 10. In the other word, complete **StackUnit** module.

Insert your modified StackUnit module here.

|  |
| --- |

1. According to Figure 9, integrate **InputSanitizer**, **StackUnit**, and **SevenSegmentDisplay** modules together in the **StackCircuit** module (top-level module).

Insert your modified StackCircuit module here.

|  |
| --- |

1. Map your system's inputs and outputs onto the Basys3 pins, then create a constraint file.

Hint: Look up the [Basys3 reference manual](https://digilent.com/reference/_media/reference/programmable-logic/basys-3/basys3_rm.pdf) on page 7 and 15.

Submit your constraint file here.

|  |
| --- |

1. Run the synthesis.
2. Run the implementation, generate the bitstream, and program the **Basys3** device.
3. [Checkpoint] Call TA to inspect your work.