Small Microprocessor (SM) Design

Using AXI4-Full of ARM Cortex-A9

List of Authors (Abd Al-Rahman Al-Nounou, Hitesh Sojitra)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI

e-mails: aalnounou@oakland.edu, hzsojitra@oakland.edu

***Abstract*—** Central Processing Unit (CPU) design is an area of engineering that focuses on the design of a computer's CPU. The CPU acts as the "brain" of the machine, controlling the operations carried out by the computer. In this work, we design an AXI4_Full interface for a small microprocessor (SM) peripheral. The custom peripheral is integrated into an embedded system in vivado. A software application is created in the SDK tool that can communicate with the peripheral. The output is mapped to external outputs on the development board LEDs.

I. INTRODUCTION

The basic task for a CPU is to execute the instructions contained in the programming code used to write software. Different CPU designs can be more or less efficient than one another. Some designs are better at addressing certain types of problems.

A small Microprocessor (SM) Design is a simple CPU that handles up to 4 bits of data and 3 bits of instruction to perform operations where the instructions are provided in one clock cycle. The data flow is wrapped with the AXI-4 Full Peripheral and allows PS+PL integration with C coding.

The CPU consists of a Datapath and Control Unit. Datapath includes the Register File (set of Registers) Which holds data and memory address values during the execution of the instruction and Arithmetic Logic Unit (ALU): A shared operation unit that performs arithmetic (e.g., addition, subtraction, division) and bitwise logic (e.g., AND, OR operations).

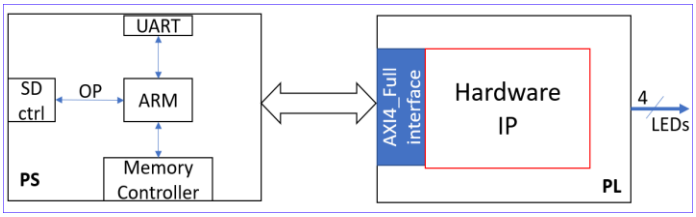
Control Unit (CU) controls operations performed on the Datapath and other components (e.g., Instruction memory (IM)). It interprets and executes the instructions. Instructions are read from a specific memory address in IM which is provided by the Program Counter (PC) component sequentially (one by one). The IM can store up to 16 8-bits instructions. To execute a particular instruction, CU asserts specific signals at certain times to control the registers, ALU, memories, and ancillary logic.

The Instruction Decoder (ID) reads the instructions and generates control signals to the Datapath and other components. It is usually implemented as a combinational circuit (single-cycle computers) or as a large Finite State Machine (FSM) with ancillary logic (multi-cycle computers).

II. OVERVIEW

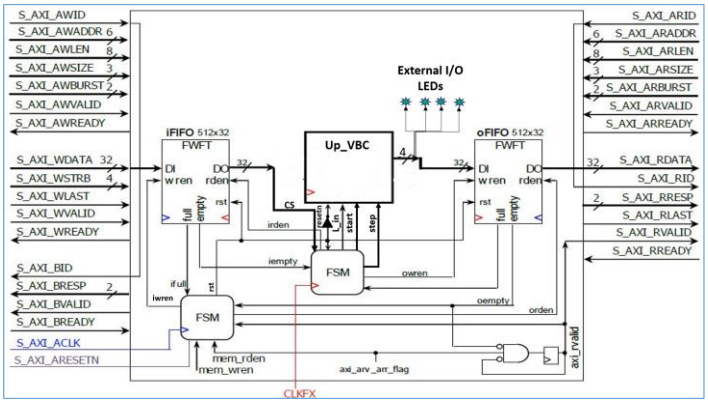
This is the Hardware Architecture with the proposed draft Block Diagram along with the I/O mechanism.

A. General Top-level Block Diagram



B. Peripheral and FSM

Below is the peripheral and FSM architecture that has been implemented with this project.



C. Software (PS) Component

The project implements the software application using the SDK tool by the C code to read and write the AXI full FIFO memory that can communicate with the peripheral. The start word or FSM triggered word will be written using CPU_mWriteMemory and the result will be obtained by the CPU_mReadMemory APIs from BSP.

III. METHODOLOGY

A. AXI4-Full Protocol and Interface introduction

The Advanced eXtensible Interface (AXI) is part of ARM AMBA, a family of microcontroller buses first introduced in 1996. The first version of AXI was first included in AMBA 3.0, released in 2003. AMBA 4.0, released in 2010, includes the second major version of AXI, AXI4.

AXI4-full removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-full interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped.

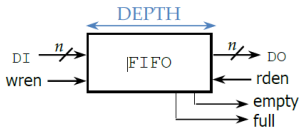
The AXI4-full protocol is used for applications that typically focus on a data-centric and data-flow paradigm where the concept of an address is not present or not required. Each AXI4-full acts as a single unidirectional channel with a handshaking data flow. At this lower level of operation (compared to the memory-mapped protocol types), the mechanism to move data between Ips is defined and efficient, but there is no unifying address context between Ips. The AXI4-full IP can be better optimized for performance in data flow applications, but also tends to be more specialized around a given application space.

AXI4-full is Hardware peripherals: 64-byte memory (or 16 32-bit word memory). Data Width: 32 bits.

B. AXI4-Full Protocol with Project Application

The AXI4-full protocol has been used with the project to communicate to the Internal FSM to start data and instruction processing. The AXI peripheral shall write the 32-bit data into Input FIFO when the wren signal will be asserted.

FIFO (FIRST-IN, FIRST-OUT) based AXI4 Full design:
The FIFO is the structure that is useful to implement queues and buffering large amounts of data.

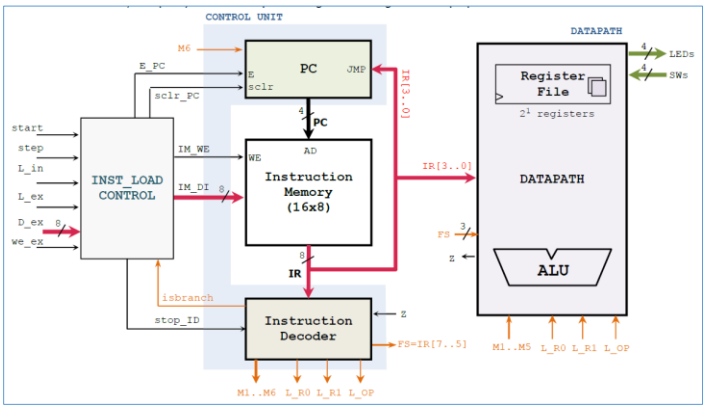


The Synchronous FIFO: data are written/read at the same clock rate while the Asynchronous FIFO: write and read clocks can be different. This is an ASIC design (not an RTL) that is useful to pass data between different clock domains.

The 32-bit data will be written by SDK software using the CPU_mWriteMemory function, the word to be written is constant data i.e. 0xAA55FFFF, this is called the processing start trigger word, once the internal FSM receives this word the conversion process starts.

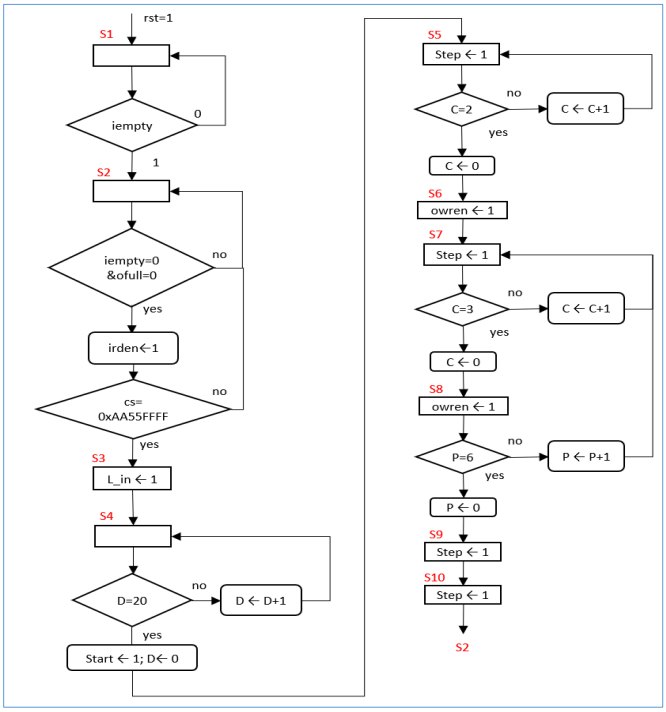
C. Microprocessor with instruction load control

This is a simple microprocessor where instructions are processed in one clock cycle.
- Only one instruction memory. No data memory. Data can be loaded onto the ALU on one clock cycle.
- Instruction Memory: Implemented as an array of registers. When reading, the output appears as soon as the address is ready.
- Instruction Decoder: the 'stop_ID' external signal makes sure that the ID outputs are '0' so that nothing gets updated.
- Note how this detailed figure fits into the Generic CPU model.



D. FSM Control (without LED control)

Below is the internal FSM that controls the starting of the ALU operation based on the received trigger signal from iFIFO and also controls the oFIFO to write the processed data to use with the Application.

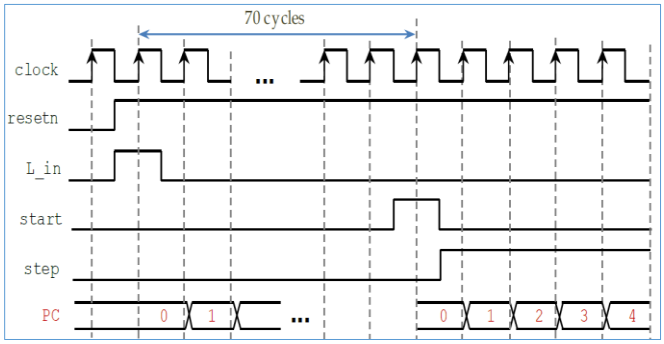


E. Program Execution

The following diagram shows the program execution states, here set the L_in = 1 for a clock cycle. Then wait for 70 cycles for the program to be written on the Instruction Memory.

Since the inputs L_ex, we_ex, and D_ex are not used with the application then they are set to 0's.

Now, the start is asserted to '1' for a clock cycle. The step should be asserted to '1' during the execution of the program (for as many cycles as needed).



F. Instruction Memory

The instruction decoder component is in charge of issuing control signals for the proper execution of instructions. The inputs to this circuit are the Instruction Register (IR) and the Z flag.
The outputs are all the control signals: M1, M2, M3, M4, M5, M6, L_R0, L_R1, and L_OP. The Function Select (FS) (the output to the ALU) is directly generated by IR[7..5] also, if stop_ID = 1, the following signals must be reset to '0': register enables in the Datapath (L_OP, L_R0, L_R1), and the PC control signal M6. This is useful to pause the execution of a program (PC and Datapath are not updated).

The following assembly program is for a counter from 2 to 13: 2,3,…, 13,2,3,…

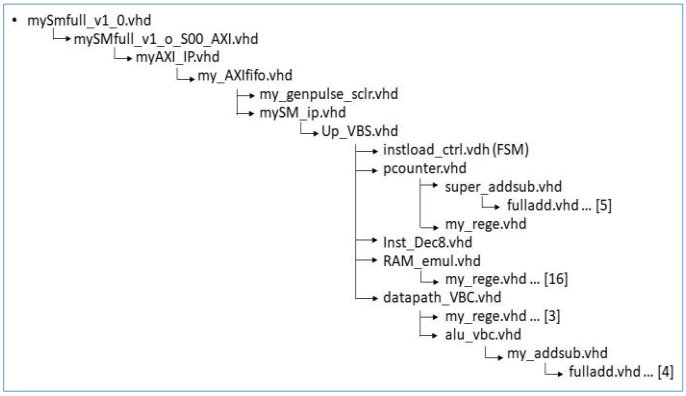
address	INSTRUCTION MEMORY
0000	00100010
0001	00110100
0010	11000000
0011	01100001
0100	01110001
0101	11110010
0110	00100001
0111	11100000
1000	00000000
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

```
* 2 to 13 ≡ 4 to 15


start: loadi R0,2          R0 ← 2
       loadi R1,4          R1 ← 4
loop:  out R0 → OUT: shows the count
       addi R0,1           R0 ← R0+1
       addi R1,1           R1 ← R1+1
       jnz R1, loop
       loadi R0,1          R0 ← 1
       jnz R0, start
```

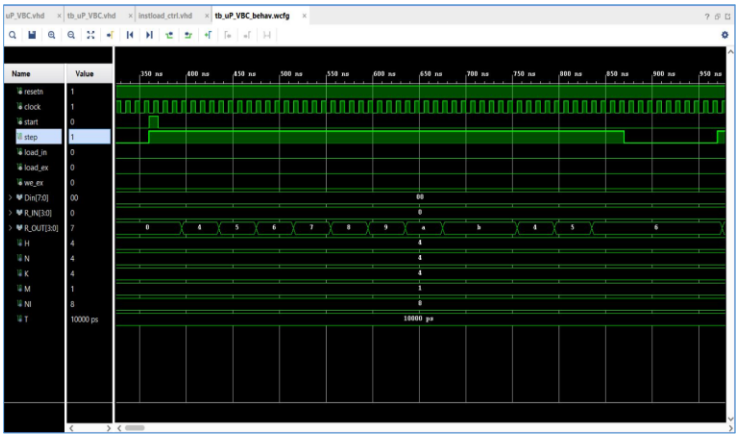
G. Design Hierarchy

The project software has the following VHDL files that are implemented and designed the architecture.



H. Operation: Internal VBC Component Execution

The following scope image shows the simulation result of the base circuit (Up_VBS.vhd), it shows the correct output as per the project architecture.



IV. SOFTWARE ARCHITECTURE AND DESIGN

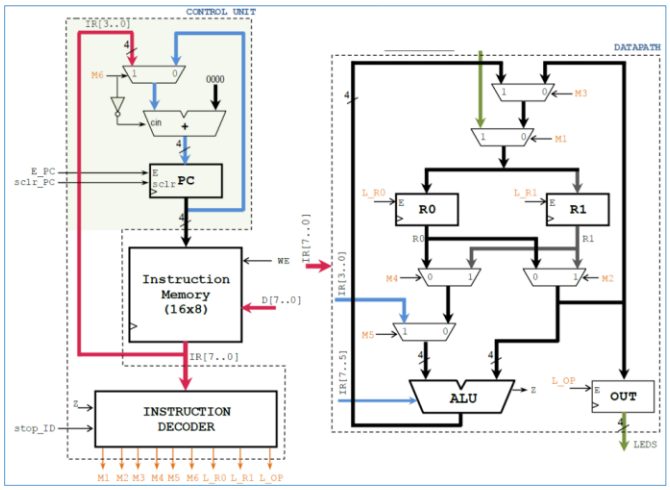
The software has been designed and implemented with Vivado 2019.1. Initially design the basic computer which can decode the instruction as per described in the Instruction memory section.

The Very Basic Computer (VBC) design is provided to start with. The scope of this project is to implement AXI4-Full interface around the VBC components. AXI interface is used to communicate with FSM that triggers the ALU operation within the VBC component.

The output of the VBC component is provided to the output of the FIFO for the AXI4-Full's interface. The output is visible in the SDK terminal via serial communication and it is connected to external I/Os (LEDs) on the Zybo board.

A. *VBC component*

VBC component: This is a simple microprocessor where instructions are processed in one clock cycle. Only one instruction memory. No data memory. Data can be loaded into the ALU on one clock cycle.



B. *Instruction Set*

Instructions are the collection of bits that instruct the compute to perform a specific operation. Each instruction specifies i) an operation the system is to perform, ii) the registers or memory words where the operands are to be found and the result is to be placed, and/or iii) which instruction to execute next.

Instructions are usually stored in memory (RAM or ROM). To execute the instructions sequentially, the address in memory of the instruction is needed to be executed. The address comes from the Program Counter (PC).

Executing an instruction means activating the necessary sequence of microoperations in the Datapath (e.g.: add, subtract, load, shift) and elsewhere required to perform the operation specified by the instruction.

Instruction Set is the collection of instructions for a computer. Instruction Set Architecture (ISA) is a thorough description of the instruction set. Simple ISAs have three major components: storage resources (IM, DM, Register File), instruction formats, and instruction specifications.

Program is the List of instructions that specifies the operations, the operands, and the sequence in which processing is to occur. It is where the user specifies the operations to be performed and their sequence.

The data processing performed by a computer can be altered by specifying a new program with different instructions or by specifying the same instructions with different data. Instruction and Data can be stored in the same memory, in different memories, or they might appear to come from different memories.

The Control Unit reads an instruction from memory, decodes and executes the instruction by issuing a sequence of one or more microoperations (in single-cycle CPUs, only microoperationis are performed per instruction).

V. PROJECT SETUP

A new Vivado project is created. The corresponding Zybo board and other basic configurations are selected.

The code for ALU, IM, DM, and Instruction Load Control blocks is already designed. We instantiate these components and set up the corresponding generic parameters.

The FSM and the AXI full peripheral around the circuit are designed to communicate outside the world, the VHDL code is written for the given circuit. Then, the circuit is synthesized to clear syntax errors.

The VHDL testbench is written to simulate the circuit. The simulation providing input and timing parameters is run as is shown belwo.

Another Vivado project is created to develop IP for the VBC component. The default language is made sure to be VHDL so that the system wrapper and template files are created in VHDL.

A. *Create Block Design in Vivado*

A Block Design is created and instantiated the Zynq PS and the project IP (mySMfull) AXI4_Full peripheral. The Block Automation and Connection Automation are run.

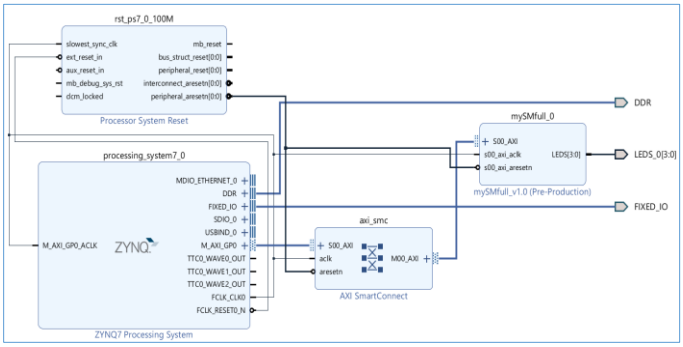
There is a .xdc file required to be added to create external I/Os for LED outputs and make them external ports.

Then, the VHDL wrapper is created (Sources Window → right-click on the top-level system design → Create HDL Wrapper)

Synthesize, implement, and generate the bitstream.

Below is the IP design circuit implemented using the Vivado 2019.1

External output LEDs are connected to LED_0[0, 3].

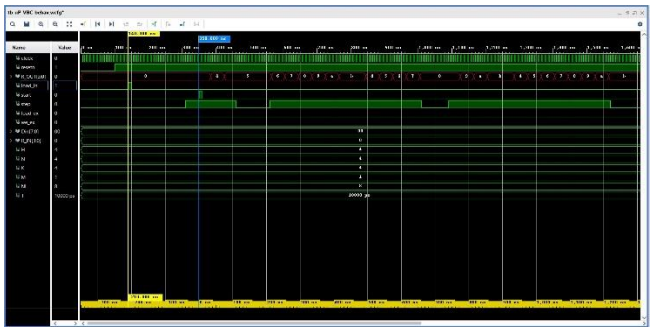


VI. TESTBENCH SETUP

For this project, AXI4_Full testbench have been used to test the project design and the architecture. AXI4_full peripheral uses the FIFO concept in which the input FIFO will read the input data from the user input or the SDK.

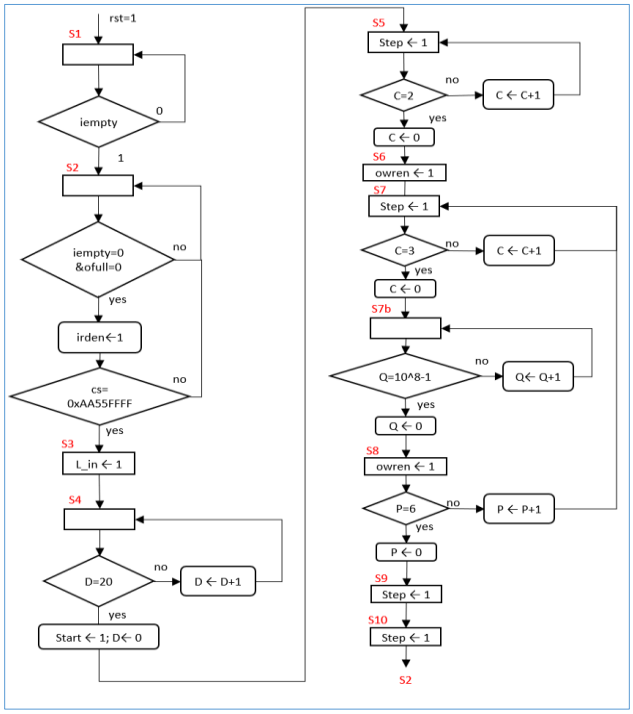
The input data 0xAA55FFFF is provided to the FSM. The FSM triggers the CPU operation with ALU and starts counting provided counter sequences as is shown in the below simulation scope.

The counter counts value from 0x04 to 0x0B and repeats the same sequence if the state machine is triggered again.



For this project, the LEDs are used as external output interfaces, the counter which is generated by the CPU is displayed on the LEDs of the Zybo board.

To make LEDs visible on the board, the original state machine is modified to add approximately a 0.8-second delay after each output so that human eyes can see the LEDs blinking.

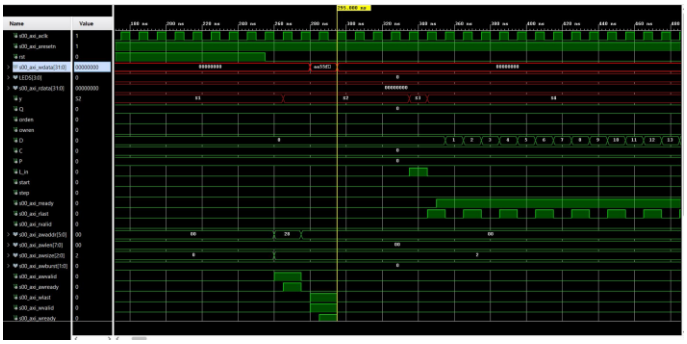


VII. RESULT

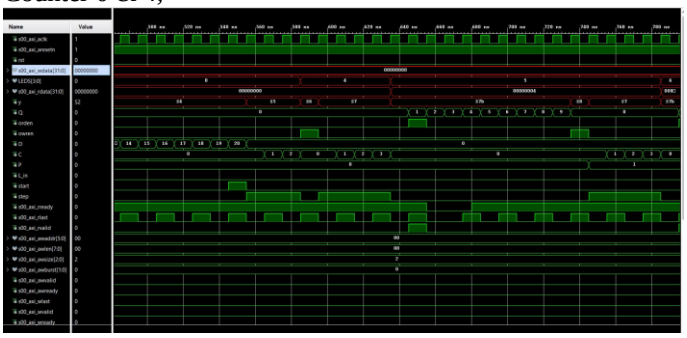
The final state machine is used to simulate the AXI interface with LED outputs enabled.

Here are the figures that represent the full counter sequence after providing the Input (0xAA55FFFF) from the test bench to the end of the counter.

Trigger point and counter start counting.

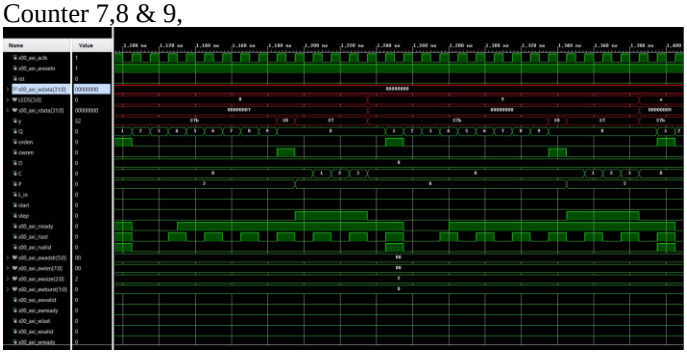


Counter 0 & 4,

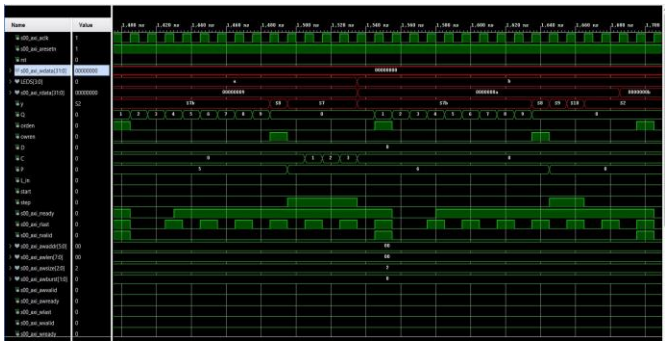


Counter 5 & 6,

Counter 7,8 & 9,



Counter 9, A & B.



A. Project Experiment with SDK and External LEDs

The SDK project is created to write the C code that is used to provide input trigger to internal FSM. The CPU_mWriteMemory function is used to write data into the AXI bus and the result is obtained by the CPU_mReadMemory APIs from BSP.

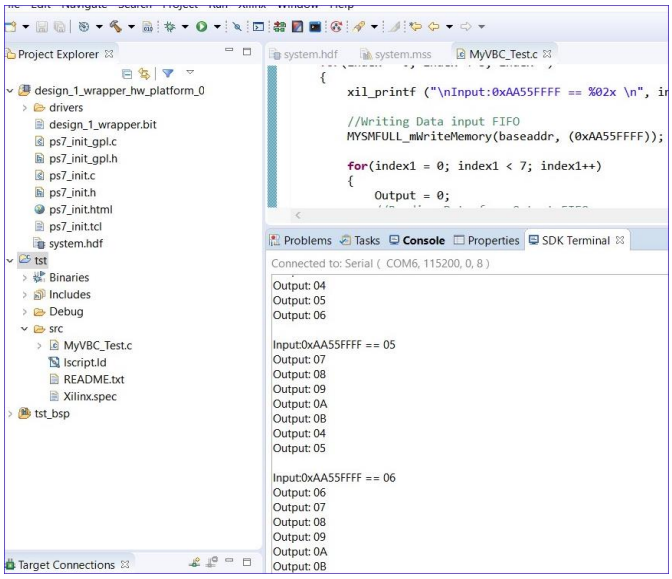
Once the program and SDK are compiled, the ZYBO Board is connected to the USB port of the computer and downloaded the bitstream on the PL: Xilinx Tools → Program FPGA

Then, the SDK Terminal is connected to the proper COM port. The project created is selected from the SDK. Right-click and select Run As → Launch on Hardware (GDB).

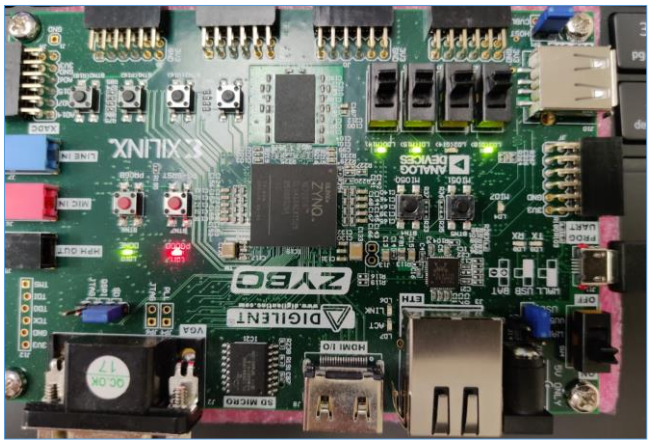
After that the Zybo board is ready to test and verify the project implementation.

Below are the output from the SDK and the LEDs from the board which are running in parallel.

The complete project demo shows the actual counter incrementing and blinking LEDs according to the counter value displayed in SDK Terminal.



Programmed Zybo Board with LED outputs,



We are able to accomplish the planned architecture and design for this project. The project outcome matches the required performance of the initially planned and expected result.

VIII. CONCLUSIONS

A Small microprocessor is designed to implement an AXI4_Full peripheral. A specific signal word is used to trigger the state machine to start fetching the instructions from the instruction memory to provide a one-by-one counting sequence from 0x04 to 0x0B. The same count can be repeated if the state machine is triggered again. External outputs on the Zybo board are used to provide the result.

All over, this project is great learning from the class materials and also provides the platform to grow our skills in related filed within industries. Many industries are using these same tools and methodologies to develop actual industrial products hence we believe that this project work is very useful to learn about real-world technology.

A. Feature work and expansion

This design could be expanded into a more complex processor with a stack to allow for less reading and writing between the software and hardware. Also, The instruction set could be expanded from a 3-bit to a 4-bit or higher set depending on the required project complexity.
The software can be used with more than two processors. In addition, we can add the reconfigurable partition to the ALU to allow multiple configuration switches at run time.

IX. REFERENCES

[1] Dr. Daniel Llamocca "Reconfigurable Computing Research Laboratory (RECRLab), Electrical and Computer Engineering Department, Oakland University"

[2] Dr. Daniel Llamocca, ECE5736: Reconfigurable Computing "http://www.secs.oakland.edu/~llamocca/Summer2021_ece5736.html