2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

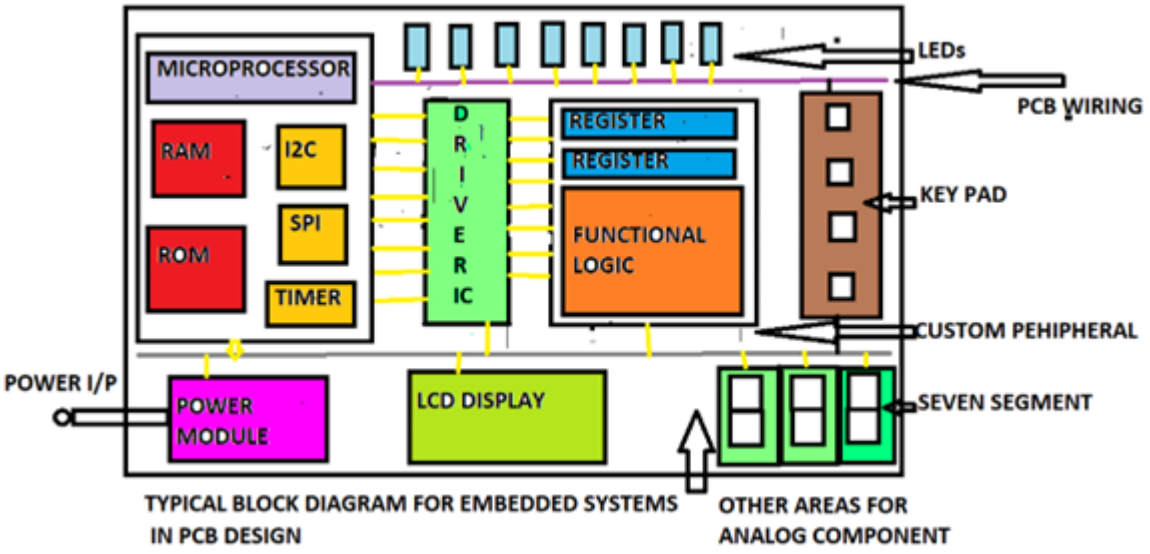
Hardware/Software co-design and co-verification of embedded systems

Mayank Nigam, Agnisys Technology Pvt. Ltd, India, mayank@agnisys.com

Nikita Gulliya, Agnisys Technology Pvt. Ltd, India, nikita@agnisys.com

***Abstract*—** *As hardware chip complexity continues to increase, these chips are frequently utilized as targets in Initiator-Target Architectures, where the targets are commonly referred to as custom peripheral IPs. The initiator can write to and read from these peripherals. While silicon companies provide functionality and register information in their datasheets, they often overlook validation from the programmer's perspective—the ones who develop software to control these IPs. In embedded systems, programmers write software into the processor's program memory to interface with external devices and manage peripheral hardware pins, tailored to real-world applications. Therefore, it is crucial for silicon vendors to adopt a comprehensive approach to design, verify, and validate these peripheral IPs. This paper presents methodologies for the co-design, verification, and validation of hardware and software. Entire process will be demonstrated through detailed examples.*

Keywords—RAL,NRE,VIVADO ,SDK,RTL,UDP



I. Introduction

A digital peripheral custom IP is typically divided into two main sections: the logic circuits, which are designed based on specific functionality, and the registers, which are used for configuring the IP, as well as sending and receiving data to and from the initiator and external devices. These registers also act as inputs for the internal functional circuits that send data to the output pins of the IP. Programmers use the status of these pins/ports to drive external devices in the embedded world.

Therefore, hardware/software co-design and co-verification are essential before complex IPs are released to the market. Addressing this need in a timely manner is crucial; otherwise, the non-recurring engineering (NRE) costs for complex peripherals could escalate, and the failure rate of embedded systems could increase, posing risks to industries such as automotive, avionics, medical, and communication.

A. Approach

Our approach allows programmers to write data to the IP's registers from the software side. These register values must match the expected values based on functionality at any given time after reset or default conditions. Correct register values ensure that IPs are functioning correctly. In embedded systems, these registers are used in application programs, so it is crucial to write to and read the correct values from the correct register address locations. We address this problem through the hardware-software interface by creating prototypes and verifying and validating them. We use HDL languages for RTL generation, UVM for verification, and Embedded C for validation, independent of any processor, to test simple and complex registers for functionality such as lock, shadow, aliasing, TMR, trigger buffer, RW pair, FIFO, counters, etc. These validated simple and complex peripheral registers can be used in UART, I2S, GPIO, EEPROM, dual port RAM, or any custom IPs for embedded applications.

B. Detailed Work

For the hardware/software co-design and co-verification approach, we need four crucial components:

1. Register specification in RDL format
2. RTL generation
3. Register map (header files)
4. UVM code for the verification environment
5. C-API and C-code for the validation environment

These components undergo a comprehensive process to ensure the IP functions correctly within the embedded domain.

C. Description:

1. SystemRDL Format: Specifications of registers can be written in SystemRDL format. SystemRDL (Register Description Language) is a language specifically designed to describe the registers in digital systems, capturing the structure, behavior, and constraints of registers in a formalized way. This format facilitates automated generation of RTL, documentation, and software drivers, ensuring consistency and reducing manual errors.

```
addrmap block_name{
          reg {
            field {
                hw=rw;
                sw=rw;
            } fld[31:0];
            regwidth=32;
        } reg_name ;
    };
```

2. RTL Generation: The RTL (Register Transfer Level) or hardware design can be written in any HDL (Hardware Description Language), such as VHDL or Verilog. This step involves defining the detailed logic and structure of the digital circuits based on the specifications, ensuring that the design meets the required functionality and performance criteria.

```
----------part of code----------
always @(posedge clk)  begin
    if (!reset_l)
        begin
            reg_name_fld_q <= 32'bx;
        end
    else
        begin

        if (reg_name_fld_in_enb)      //fld : HW Write
            begin
                reg_name_fld_q <= reg_name_fld_in;
            end
        else
            begin
            if (reg_name_wr_valid)    //FLD : SW Write
                begin
                    reg_name_fld_q <= (wr_data [31 : 0]  & reg_enb  [31 : 0] ) |
(reg_name_fld_q & (~reg_enb  [31 : 0] ));
                end
            end
        end
    end //end always

----------part of code----------
```

3. Register Map: The register map is essential for the hardware-software interface. It defines the address locations of the registers, facilitating the software's ability to correctly access and manipulate the hardware registers. This map ensures seamless communication between the software and the hardware components.

```
----------part of code----------
#define block_name_s_ADDRESS 0x0
#define block_name_reg_name_ADDRESS 0x0
#define BLOCK_NAME_REG_NAME_FLD_OFFSET 0
#define BLOCK_NAME_REG_NAME_FLD_MASK 0xFFFFFFFF
```

```
#define BLOCK_NAME_REG_NAME_FLD_INV_MASK 0x0
#define BLOCK_NAME_REG_NAME_FLD_VALUE_MASK 0x7FFFFFFF
#define BLOCK_NAME_REG_NAME_FLD_INV_VALUE_MASK 0x80000000
#define BLOCK_NAME_REG_NAME_FLD_SIZE 32
#define BLOCK_NAME_REG_NAME_FLD_DEFAULT 0
#endif /* _BLOCK_NAME_REGS_H_ */
/* end */
-----------------part of code-----------------------------------------
```

4. UVM Code: UVM (Universal Verification Methodology) code can be written in an environment consisting of the Register Abstraction Layer (RAL) and test sequences. The RAL provides an abstract, high-level representation of the registers, facilitating the verification of register operations. Test sequences are then used to simulate various scenarios and validate that the registers and the overall system function correctly under different conditions.

```
`ifndef CLASS_block_name_reg_name
`define CLASS_block_name_reg_name
class block_name_reg_name extends uvm_reg;
    `uvm_object_utils(block_name_reg_name)

    rand uvm_reg_field fld;/**/

    // Function : new
    function new(string name = "block_name_reg_name");
        super.new(name, 32, build_coverage(UVM_NO_COVERAGE));
        add_coverage(build_coverage(UVM_NO_COVERAGE));
    endfunction

    // Function : build
    virtual function void build();
        this.fld = uvm_reg_field::type_id::create("fld");
        this.fld.configure(.parent(this), .size(32), .lsb_pos(0), .access("RW"),
.volatile(0), .reset(32'd0), .has_reset(1), .is_rand(1),
.individually_accessible(0));
    endfunction
endclass
`endif

.     .     .
.     .

task body;

        uvm_reg_data_t reg_name ;

        if(!$cast(rm, model)) begin
            `uvm_error("RegModel : block_name_block","cannot cast an object of type
uvm_reg_sequence to rm");
        end
```

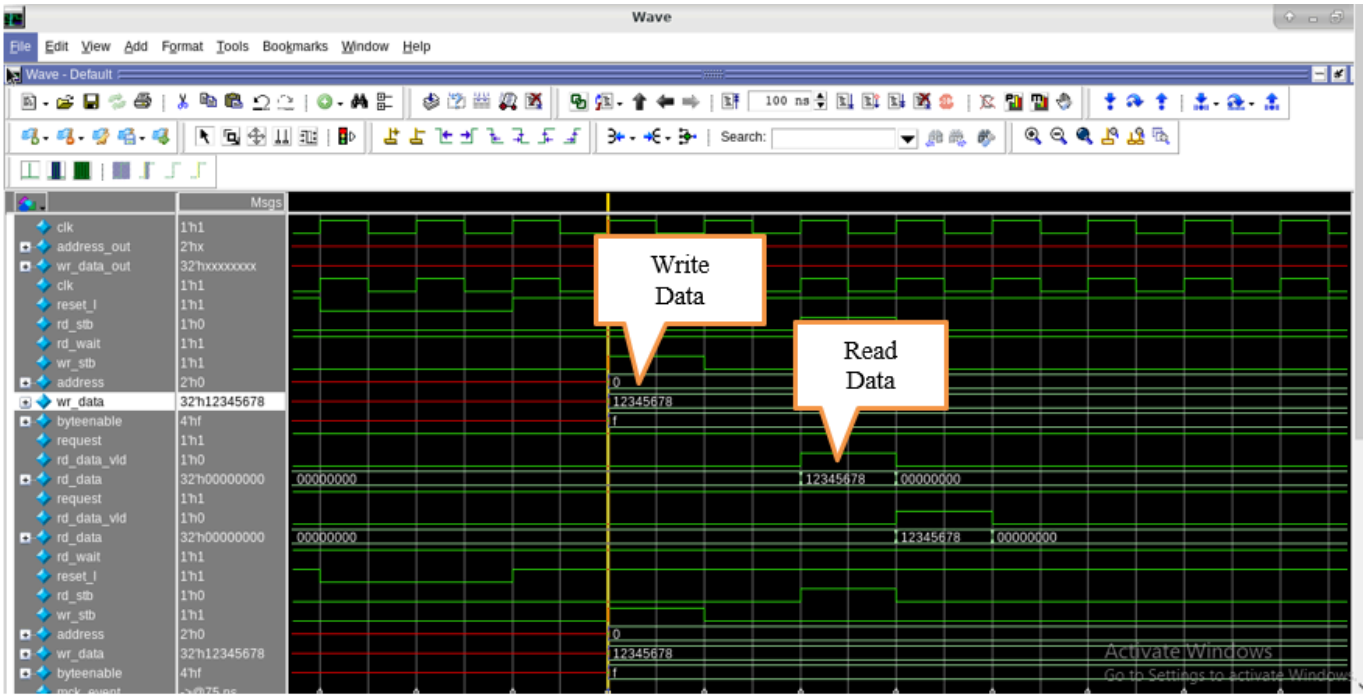
```
        if (rm == null)  begin
            `uvm_error("block_name_block", "No register model specified to run
sequence on, you should specify regmodel by using property 'uvm.regmodel' in the
sequence")
            return;
        end

        rm.reg_name.write(status, 'h012345678, .parent(this));
        rm.reg_name.read(status, reg_name, .parent(this));

        var1=reg_name;

    endtask: body
```

5. wr_data and rd_data Simulation: Verification of the sequences created above can be performed in Questa-Sim by simulating the write (wr_data) and read (rd_data) operations. This involves creating testbenches that generate various data patterns to be written to the registers and subsequently reading back the data to ensure correctness. The simulation checks the integrity and functionality of the registers, validating that data written to the registers can be accurately read back, and that all sequences and corner cases are handled properly.



6. C-API and C-Code for the Validation Environment: The C-APIs are macros designed to write to and read from the registers, while the C-code constitutes the firmware for custom IPs. This firmware is processor-independent, ensuring that it can be used across various embedded systems. These components facilitate the validation of the hardware by providing a straightforward interface for software to interact with the registers, confirming their correct operation in different scenarios.

C-API (Hardware-Software Interface)

```
----------------------part of code--------------
   #define IDS_MACROS_DEFINE_H_
   #define REG_READ(ADDR)
   #define REG_WRITE(ADDR,WDATA)
   #define FIELD_READ(ADDR,MASK,OFFSET)
   #define FIELD_WRITE(ADDR,WDATA,MASK,OFFSET)

-------------------------part of code-----------
```

C-Code

```
#include "write_read.h"
#include "block_name_seq_name_iss.h"
int block_name_seq_name(int baseAddress) {
int reg_name;
int var1 = 0 ;
REG_WRITE(BLOCK_NAME_REG_NAME_ADDRESS(baseAddress),0x12345678);
reg_name = REG_READ(BLOCK_NAME_REG_NAME_ADDRESS(baseAddress));
var1 = reg_name;
return 0;
}
```

7. Validation Code in Xilinx SDK Environment: The validation code is developed within the Xilinx Software Development Kit (SDK) environment. This involves writing firmware and test code to verify the functionality of the custom IPs. The Xilinx SDK provides tools and libraries that facilitate the development and debugging of software for Xilinx devices, ensuring that the custom peripherals operate correctly and efficiently within the target embedded system.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#define block_name_reg_name_ADDRESS  0x0
#define ADDR 0x43c00000
int main()
{
    init_platform();
   Xil_Out32(block_name_reg_name_ADDRESS+ADDR,0x12345678);
   u32 value1= Xil_In32(block_name_reg_name_ADDRESS+ADDR);
   cleanup_platform();
    return 0;
}
```

D. Examples

With the use of UDP (User Defined Property) in SystemRDL, custom properties can be added to the register definitions for special registers. This enables the design of specialized registers tailored to specific needs. Once defined, the code can be implemented and verified in Vivado. This approach allows for precise customization and thorough verification of IP components, ensuring they meet the desired specifications and perform reliably in the final application

SHADOW Register

Vivado HW Design:

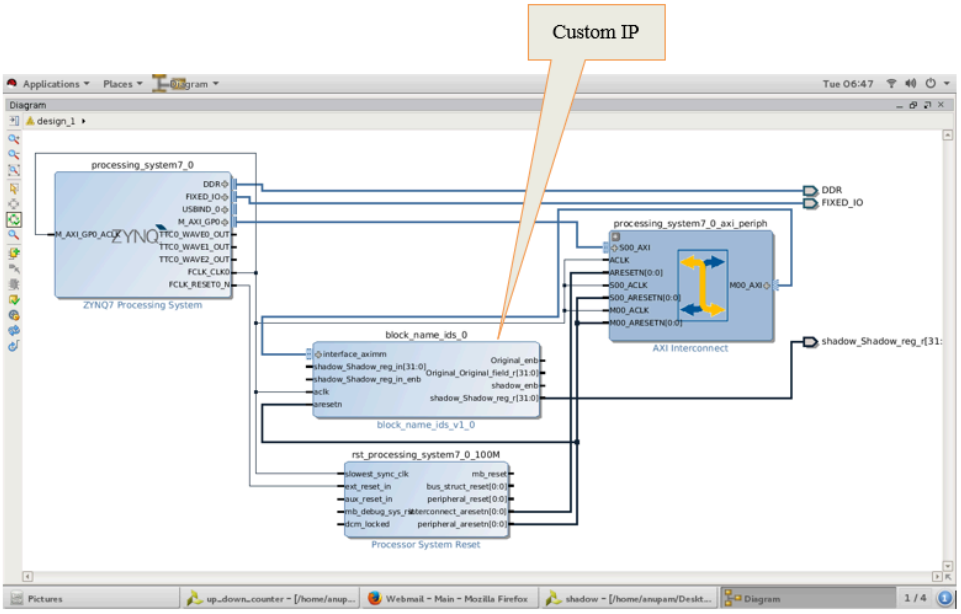


Figure 1: SHADOW REGISTER IP interconnected with Zynq processor with well-generated bitstream

```
-----------------part of implementation log----------------------

 Loading data files...
 Loading site data...
 Loading route data...
 Processing options...
 Creating bitmap...
 Creating bitstream...
 Writing bitstream ./design_1_wrapper.bit...
 INFO: [Vivado 12-1842] Bitgen Completed Successfully

-----------------------------------------------------------------
```

SDK Sequences

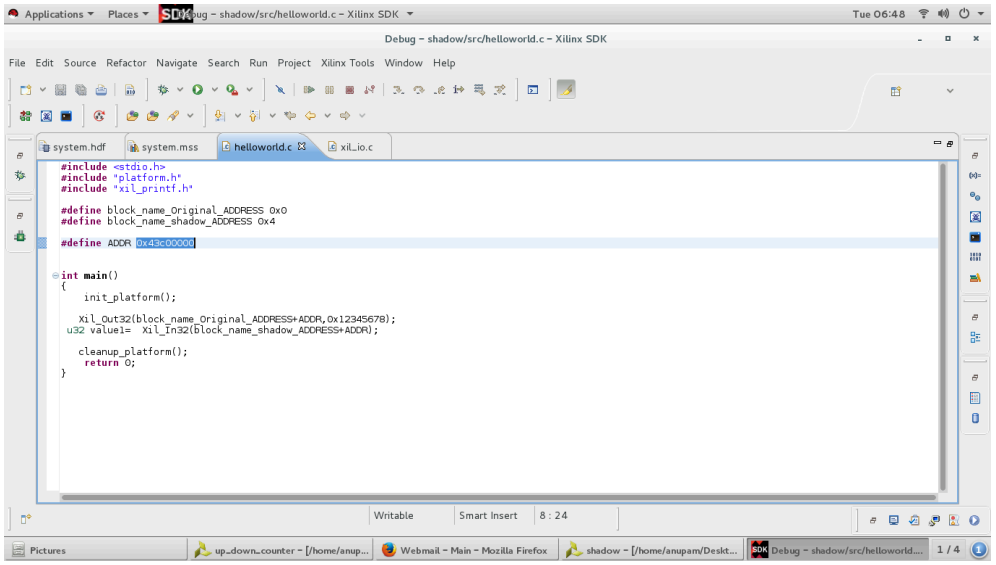


Figure 2:C-Program on SDK to check the functionality of SHADOW Register.

LOCK Register

VIVADO HW Design

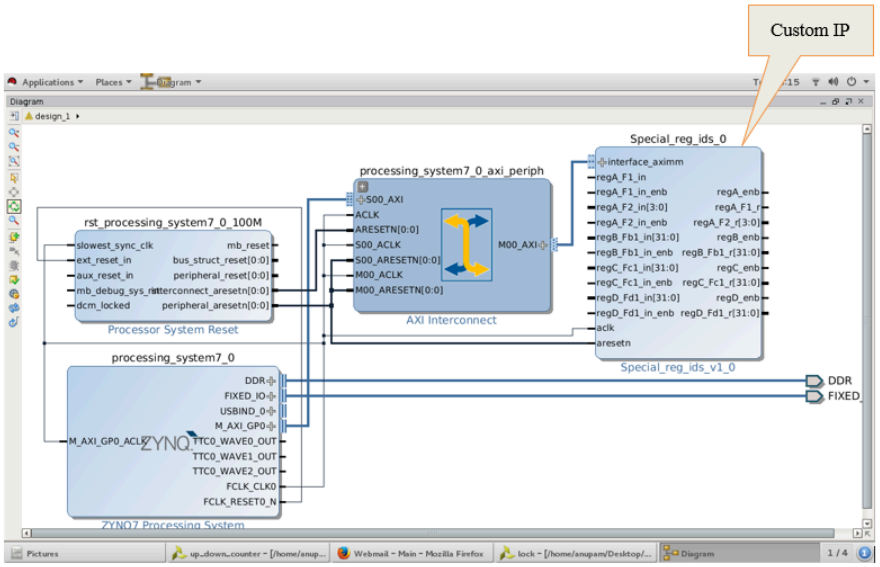


Figure 3: LOCK Register IP interconnected with Zynq processor with well-generated bitstream

Software SDK for C headers

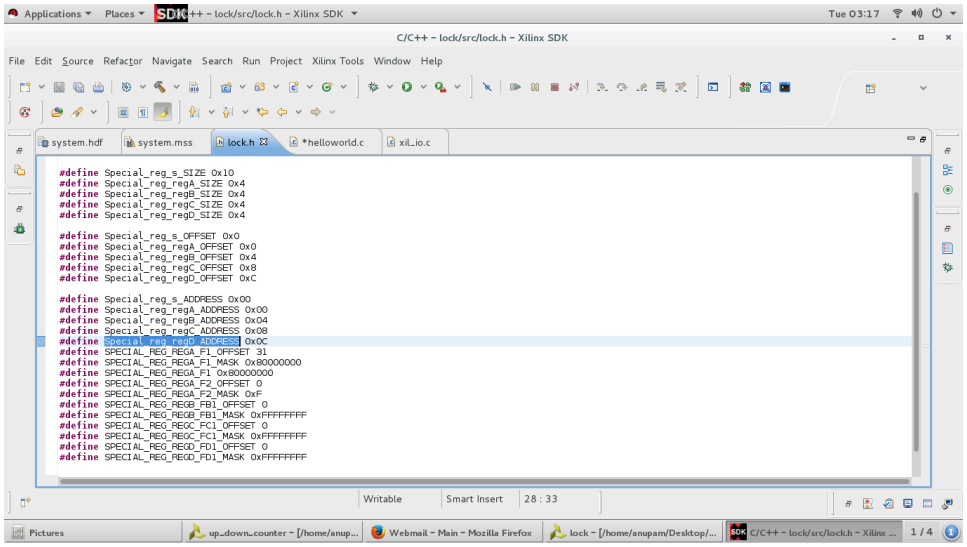


Figure 4: C-Headers for Validation of LOCK Register

Software Sequences in SDK

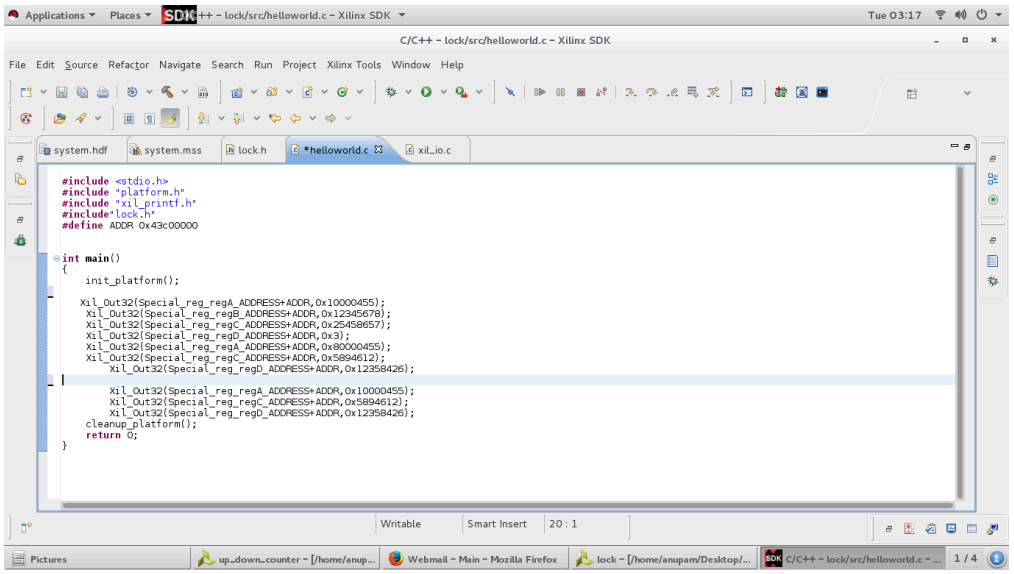


Figure 5: C-Program on SDK to check the functionality of LOCK Register.

Trigger buffer register

VIVADO HW Design:

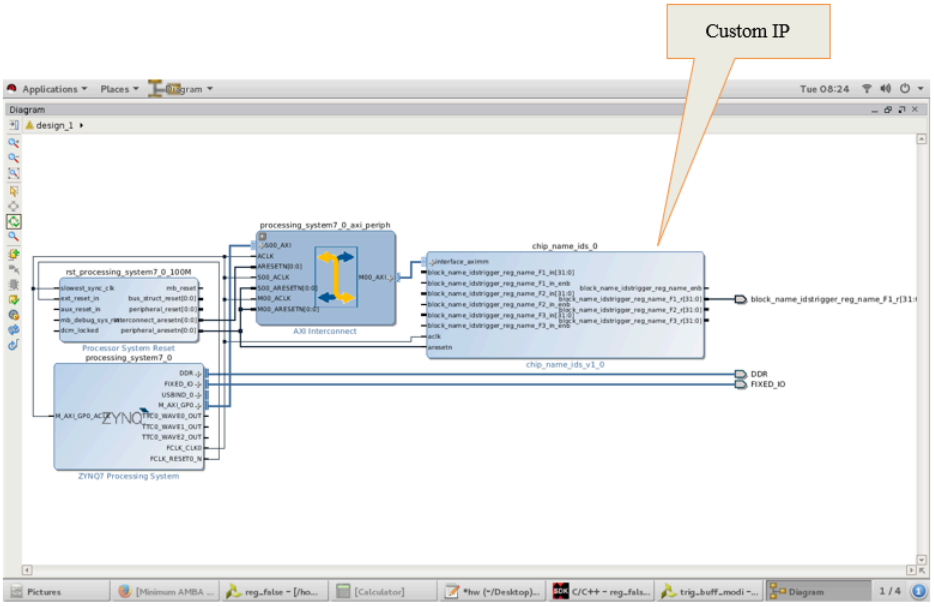


Figure 6: TRIGGER BUFFER REGISTER IP interconnected with Zynq processor with well-generated bitstream

SDK Sequences:

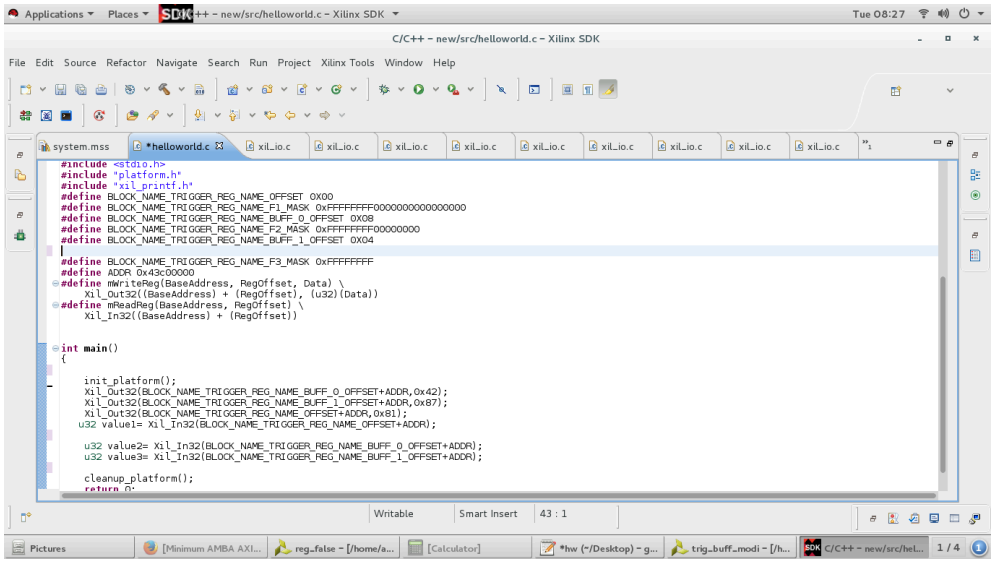


Figure 7:C-Program on SDK to check the functionality of TRIGGER BUFFER Register.

Counters:

VIVADO Design

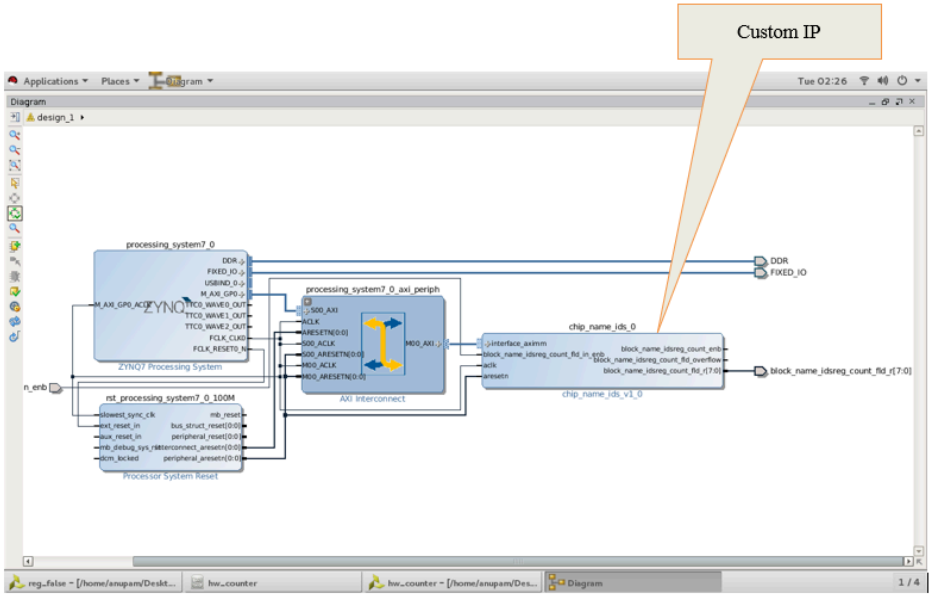


Figure 8: COUNTER IP interconnected with Zynq processor with well-generated bitstream

Software Embedded C code in Xilinx SDK

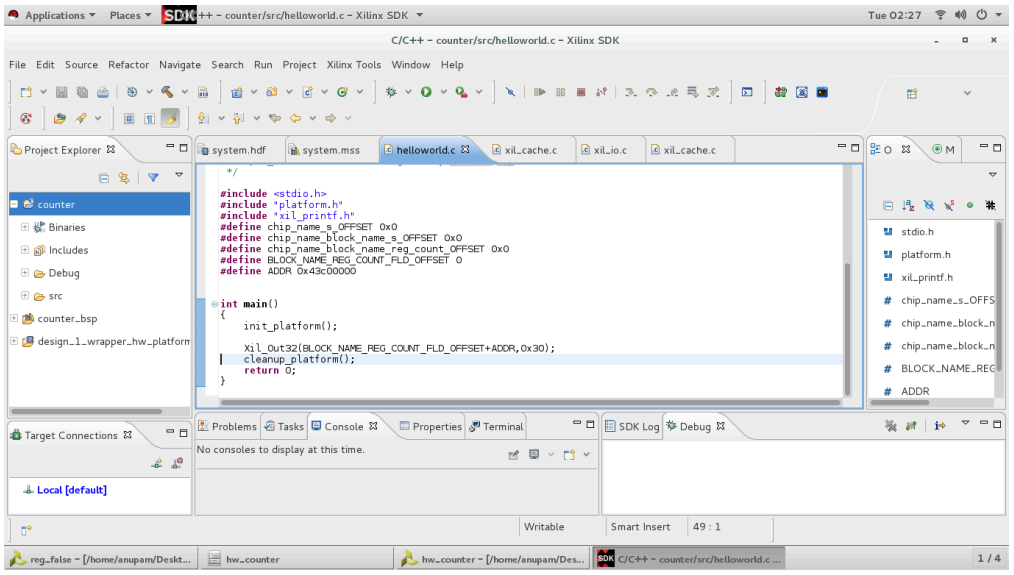
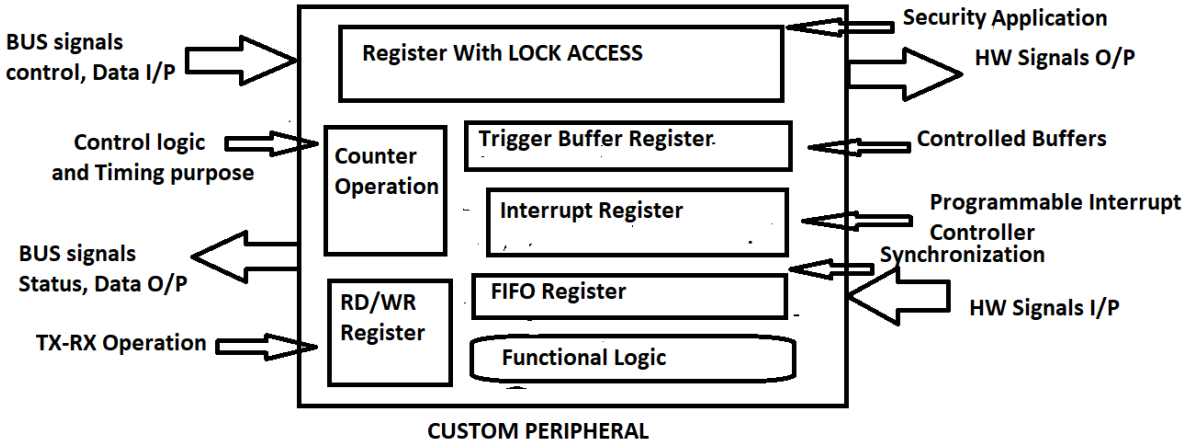


Figure 9:C-Program on SDK to check the functionality of COUNTER.

By following this procedure, various types of registers can be validated, including Interrupt Registers, FIFO Registers, and Read/Write Registers. This ensures that all specialized registers function correctly and meet the necessary specifications. Utilizing SystemRDL with UDP for design and Vivado for verification streamlines the process, providing a robust framework for the development and validation of custom IPs.

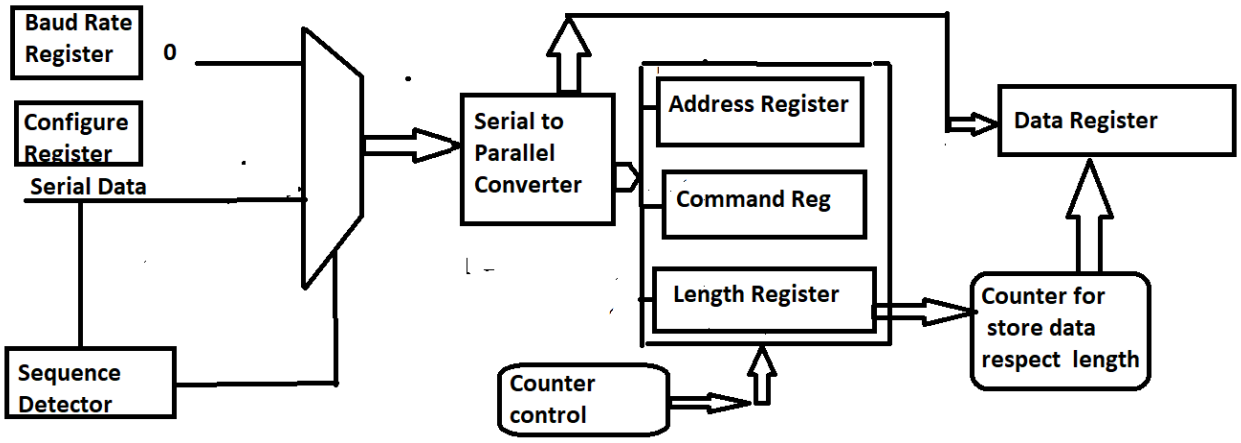
II. APPLICATION

A. Example - 1



If the registers used in custom peripherals are already validated, they can be directly connected with the functional logic and interconnected according to the application requirements. This simultaneous design and verification/validation process minimizes the time to market and ensures that these registers are reusable across different custom IPs without needing re-verification. Only the functional blocks within the IP need to be tested. The validated registers are tested in every sequence, considering all aspects and corner cases.

B. Example - 2



A Simple Block Diagram for Reciever which takes data and store data according to length of data

In this application, a sample receiver block diagram is presented. The receiver processes data packets composed of start-address-command-data-stop sequences. Each packet transmission ends with a stop condition, and a new packet transmission begins with a start condition. A sequence detector identifies the start conditions and enables data passage through a multiplexer. A counter control mechanism ensures that parts of the packet are stored in 8-bit segments within specific registers. Each 8-bit segment serves as input to another counter, dictating the length of data to be stored in the data register. For instance, if the length data of the packet (8-bit) is all ones, 256 bits of data will be stored in the data register.

The counter functions as a down counter, loaded with the length value, and allows data storage until it reaches zero. The programmer sets the baud rate and configures the registers accordingly.

In this example, registers and counters are crucial components of the block diagram. If these components have already been verified and validated, and are reused in this context, no further verification is necessary.

III. Results

The approach of hardware/software co-design and co-verification significantly improves efficiency, reducing cycle time by up to 30%. This method is not only faster but also more cost-effective and compact compared to traditional methodologies. The simultaneous generation of results during testing in both the verification and validation environments for hardware and software further enhances productivity and reliability.

IV. Conclusion

This solution provides a generalized approach for any custom peripheral IP, effectively addressing the challenges in the embedded real world. By integrating hardware and software co-design and co-verification, the approach delivers a transformative impact on the development and deployment of embedded systems, setting a new standard for efficiency and effectiveness in the industry.

V. References:

1) https://en.wikipedia.org/wiki/Embedded_system
2) https://docs.amd.com/r/en-US/ug910-vivado-getting-started/Using-the-Vivado-IDE
3) SystemRDL documentation: https://www.accellera.org/downloads/standards/systemrdl
4) https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf