UM–SJTU JOINT INSTITUTE

System-on-Chip Design (ECE4810J)

LABORATORY REPORT

Lab 1. Getting Started with Arty Z7 SoC development platform

Group 2

Name: Haochen Wu	ID: 518021910558
Name: Yihua Liu	ID: 518021910998
Name: Siyuan Zhang	ID: 518370910180

Date: September 30, 2021

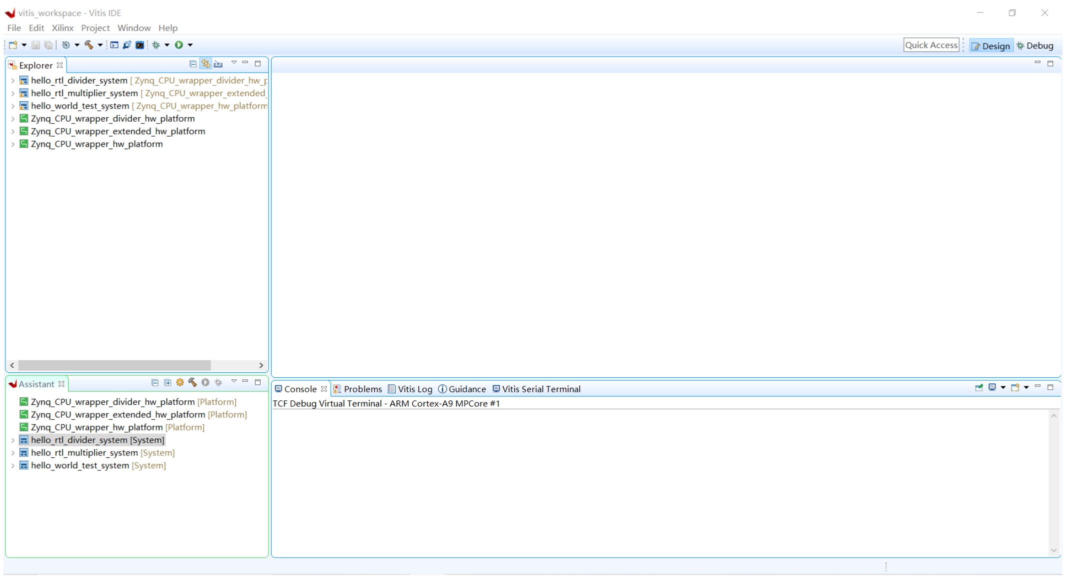
Contents

1 Introduction

In this project, we gain the basic knowledge of the Vivado and Vitis environment. We practice to build a basic Zynq system on the Arty Z7-20 SoC. And we learn to create custom IP blocks at RTL level using Verilog.

2 Part3: Building a basic ZYNQ system on the Arty Z-7 board

In this part, we add the Zynq PS into a Vivado design, so that this hardware design can be exported to Vitis as a platform where we can run software program. Here of some screen shots related to part 3 in lab manual.

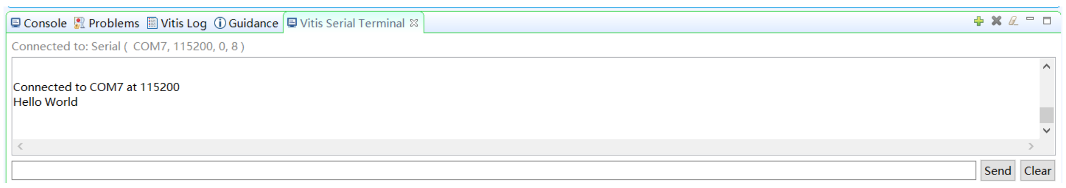


```
hello_rtl_divider_system [ Zynq_CPU_wrapper_divider_hw_p
hello_rtl_multiplier_system [ Zynq_CPU_wrapper_extended_
hello_world_test_system [ Zynq_CPU_wrapper_hw_platform
Zynq_CPU_wrapper_divider_hw_platform
Zynq_CPU_wrapper_extended_hw_platform
Zynq_CPU_wrapper_hw_platform
```

```
Zynq_CPU_wrapper_divider_hw_platform [Platform]
Zynq_CPU_wrapper_extended_hw_platform [Platform]
Zynq_CPU_wrapper_hw_platform [Platform]
hello_rtl_divider_system [System]
hello_rtl_multiplier_system [System]
hello_world_test_system [System]
```

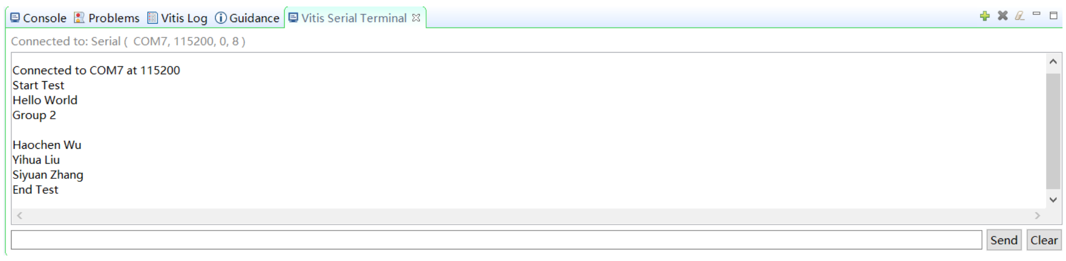
```
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1
```

Figure 1. Eclipse environment



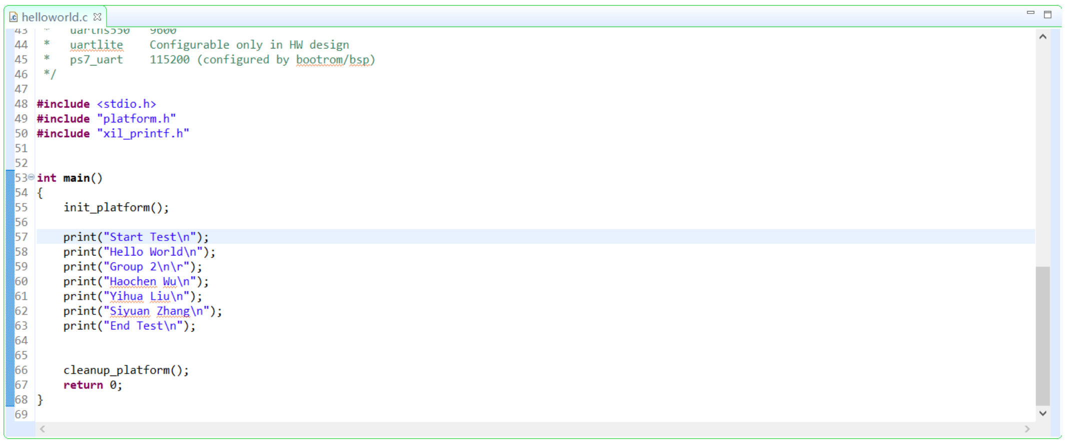
```
Connected to: Serial ( COM7, 115200, 0, 8 )

Connected to COM7 at 115200
Hello World
```

Figure 2. The Message Hello World in the SDK Terminal



```
Connected to: Serial ( COM7, 115200, 0, 8 )

Connected to COM7 at 115200
Start Test
Hello World
Group 2

Haochen Wu
Yihua Liu
Siyuan Zhang
End Test
```

Figure 3. The SDK Terminal Where It Outputs Group Name and Members' Names



```c
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();

    print("Start Test\n");
    print("Hello World\n");
    print("Group 2\n\r");
    print("Haochen Wu\n");
    print("Yihua Liu\n");
    print("Siyuan Zhang\n");
    print("End Test\n");


    cleanup_platform();
    return 0;
}
```

Figure 4. The Modified C Code

The screenshot of modified C code is shown above. The code file is submitted together with this report in the zip package.

3 Part4: Creating and using custom IP blocks in Verilog

In this part, we firstly add a multiplier as a IP block into the system. Secondly, we create our own divider, which computes the qoutient and remainder of two 16-bit numbers, add this new IP block, and connect it to Zynq PS via AXI interface. Here of some screen shots related to part 4 in lab manual.

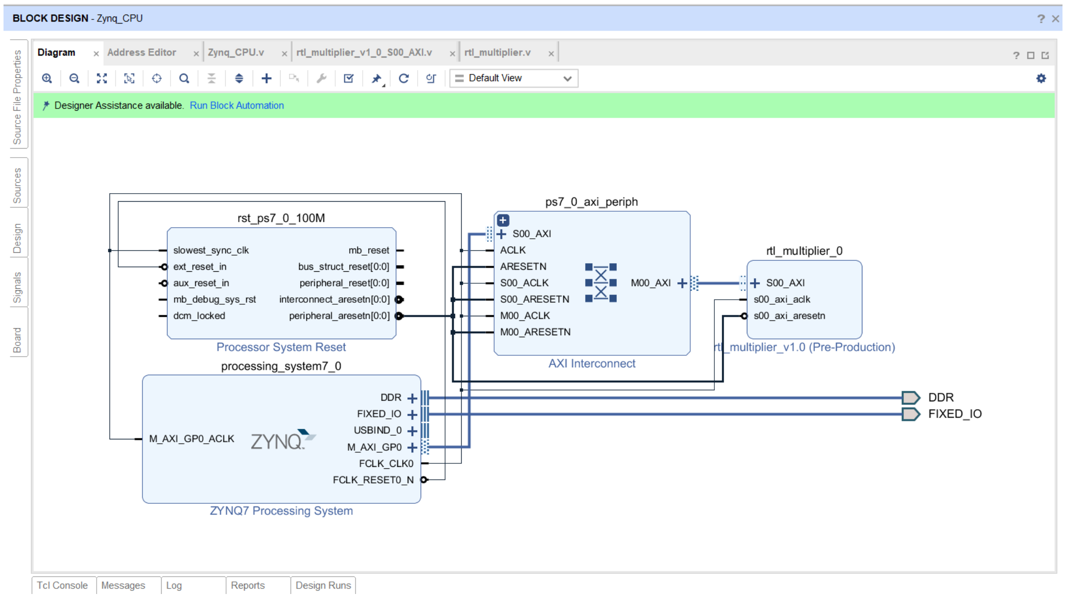
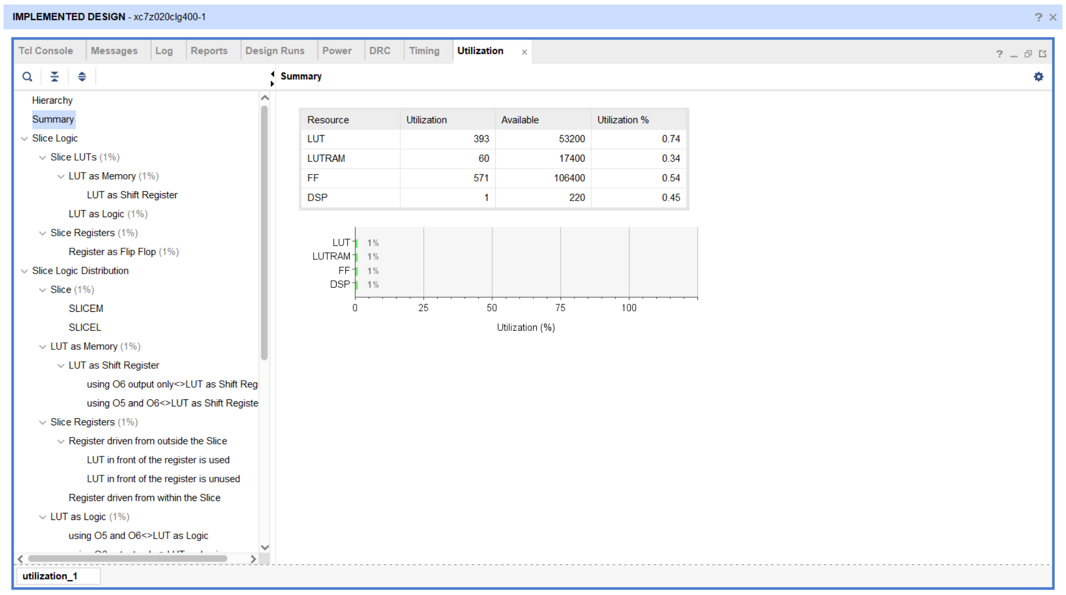


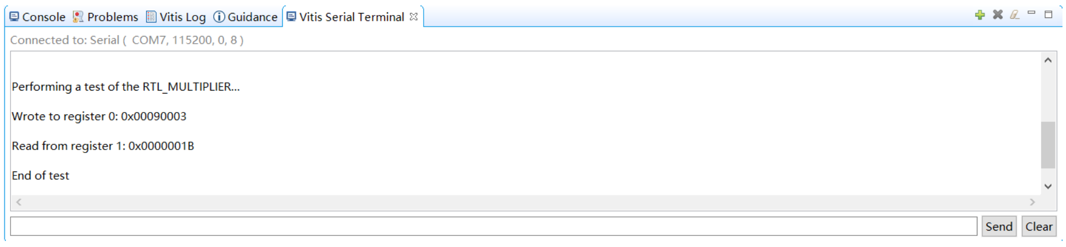
Figure 5. Screenshot of the Multiplier



Resource	Utilization	Available	Utilization %
LUT	393	53200	0.74
LUTRAM	60	17400	0.34
FF	571	106400	0.54
DSP	1	220	0.45

Figure 6. Resource Utilization Report of Multiplier

The resource utilization report is shown in picture above. Specifically, the multiplier uses 49 LUTs, 138 FFs and 1 DSP. Generally, this design is reasonable with compact logic (small number of LUTs); but the number of FF is a bit large, which may be used to satisfy timing issues.



```
Connected to: Serial ( COM7, 115200, 0, 8 )

Performing a test of the RTL_MULTIPLIER...

Wrote to register 0: 0x00090003

Read from register 1: 0x0000001B

End of test
```

Figure 7. Screenshot of your SDK Terminal for the multiplier

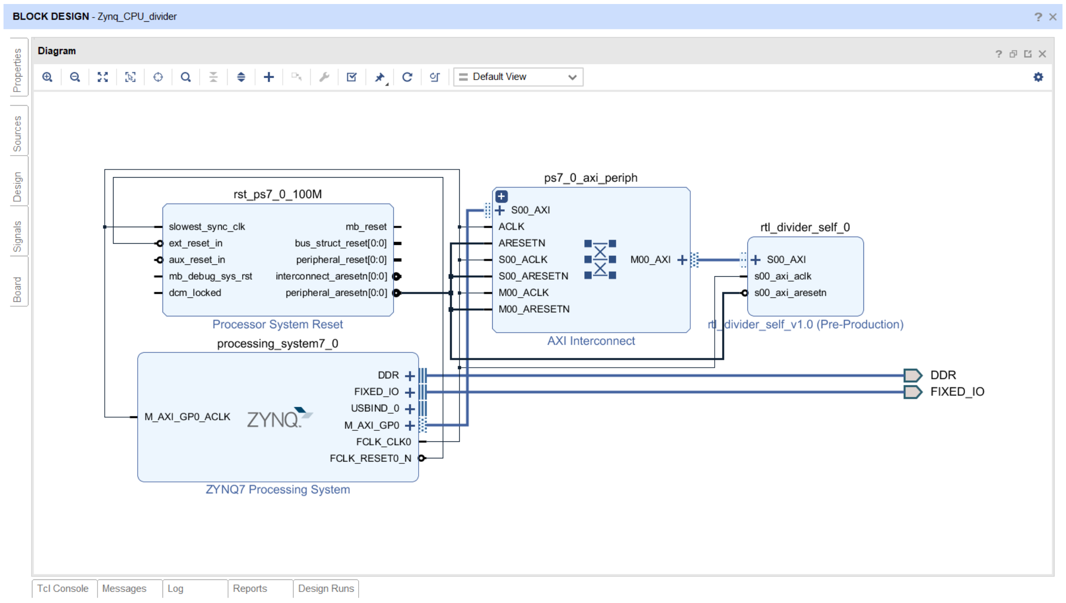
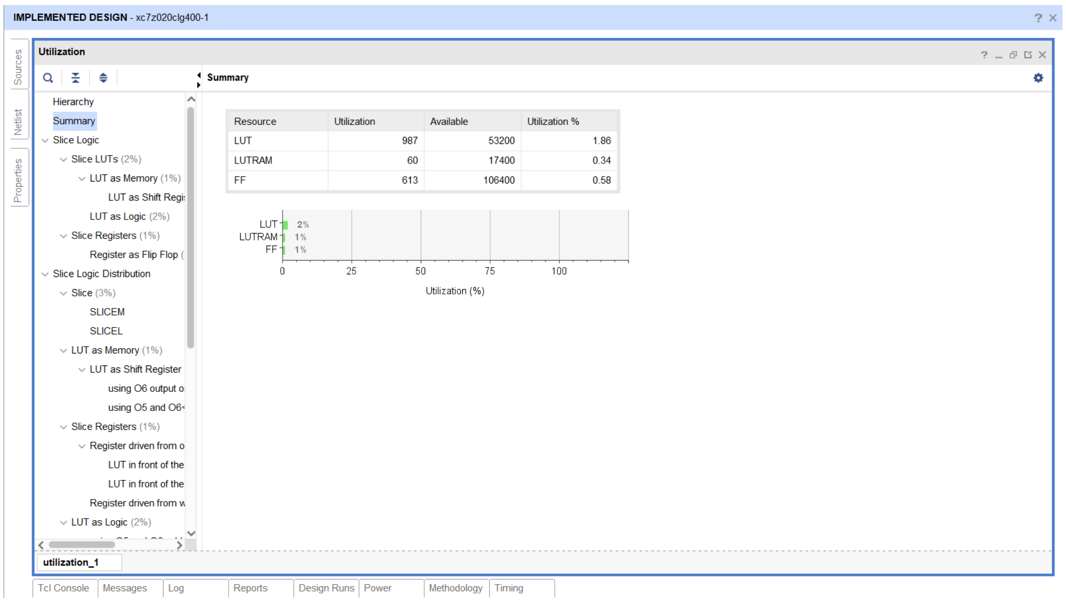


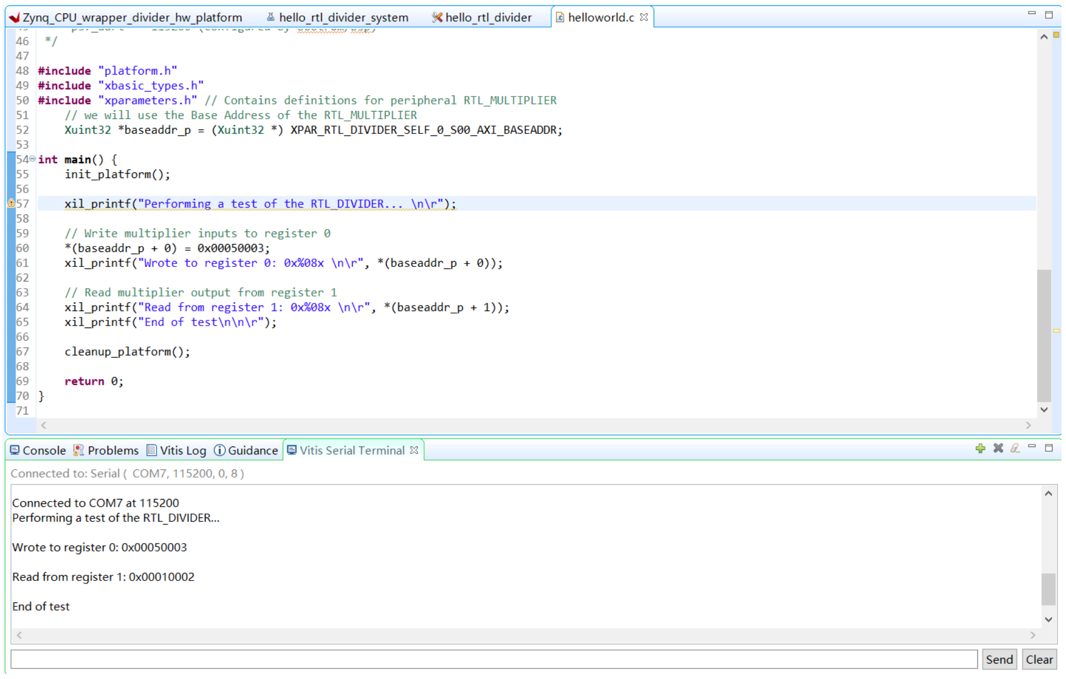
Figure 8. Screenshot of the Divider



Resource	Utilization	Available	Utilization %
LUT	987	53200	1.86
LUTRAM	60	17400	0.34
FF	613	106400	0.58

Figure 9. Resource Utilization Report of the Divider

The resource utilization report of our divider is shown in the picture above. Specifically, the divider uses 403 LUTs, and 32 FFs. The number of LUTs seems too large, which indicates the logic might be simplied. The number of flip-flops is reasonable.

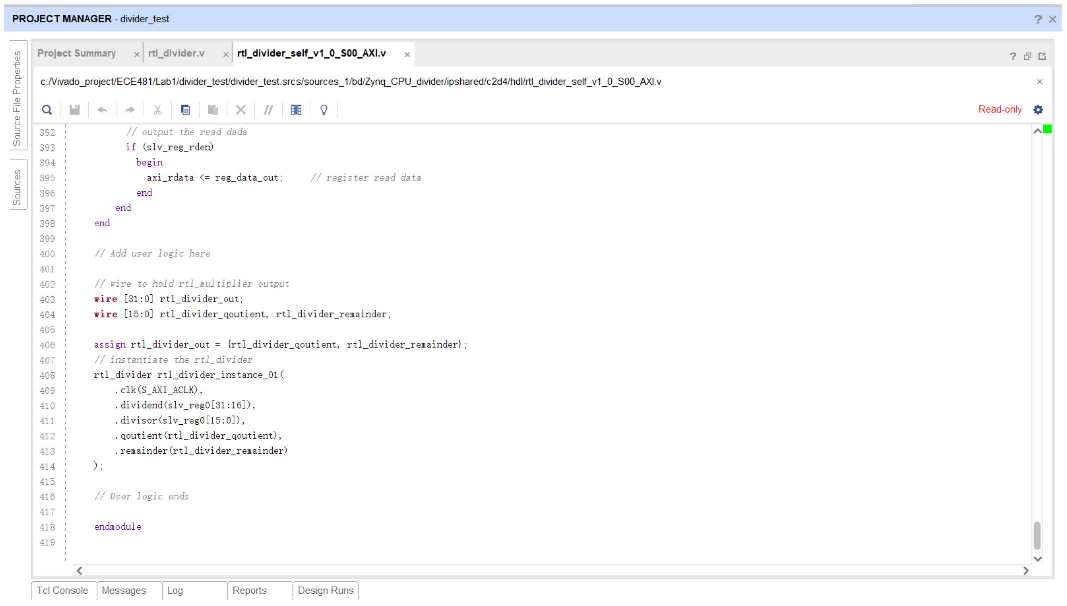


```c
 */

#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h" // Contains definitions for peripheral RTL_MULTIPLIER
    // we will use the Base Address of the RTL_MULTIPLIER
    Xuint32 *baseaddr_p = (Xuint32 *) XPAR_RTL_DIVIDER_SELF_0_S00_AXI_BASEADDR;

int main() {
    init_platform();

    xil_printf("Performing a test of the RTL_DIVIDER... \n\r");

    // Write multiplier inputs to register 0
    *(baseaddr_p + 0) = 0x00050003;
    xil_printf("Wrote to register 0: 0x%08x \n\r", *(baseaddr_p + 0));

    // Read multiplier output from register 1
    xil_printf("Read from register 1: 0x%08x \n\r", *(baseaddr_p + 1));
    xil_printf("End of test\n\n\r");

    cleanup_platform();

    return 0;
}
```

```
Connected to: Serial ( COM7, 115200, 0, 8 )

Connected to COM7 at 115200
Performing a test of the RTL_DIVIDER...

Wrote to register 0: 0x00050003

Read from register 1: 0x00010002

End of test
```

Figure 10. Screenshot of SDK Terminal for the Divider

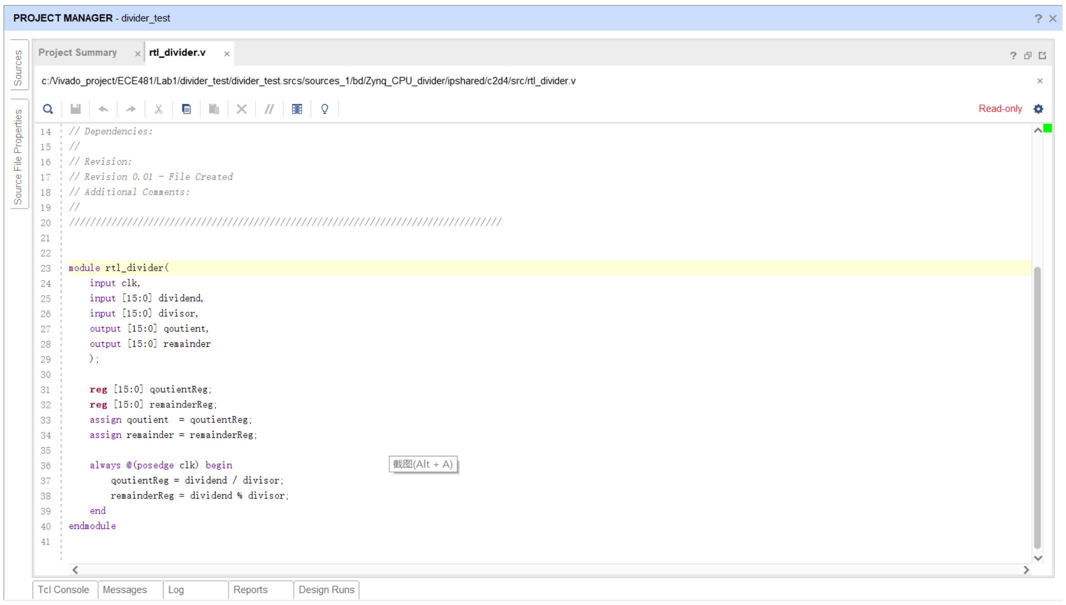
For this divider, we also input two 16-bit numbers, whose concatenation corresponding to register 0. The output is the quotient and the remainder, both are also 16-bit, since if we assume both input numbers as integers, the remainder will be smaller than the divisor and the qoutient will be smaller than the dividend.



c:/Vivado_project/ECE481/Lab1/divider_test/divider_test.srcs/sources_1/bd/Zynq_CPU_divider/ipshared/c2d4/hdl/rtl_divider_self_v1_0_S00_AXI.v

```verilog
            // output the read dada
            if (slv_reg_rden)
              begin
                axi_rdata <= reg_data_out;     // register read data
              end
          end
      end

      // Add user logic here

      // wire to hold rtl_multiplier output
      wire [31:0] rtl_divider_out;
      wire [15:0] rtl_divider_qoutient, rtl_divider_remainder;

      assign rtl_divider_out = {rtl_divider_qoutient, rtl_divider_remainder};
      // instantiate the rtl_divider
      rtl_divider rtl_divider_instance_01(
          .clk(S_AXI_ACLK),
          .dividend(slv_reg0[31:16]),
          .divisor(slv_reg0[15:0]),
          .qoutient(rtl_divider_qoutient),
          .remainder(rtl_divider_remainder)
      );

      // User logic ends

      endmodule
```

Figure 11. Connection between the divider and AXI registers


PROJECT MANAGER - divider_test
Project Summary
rtl_divider.v
c:/Vivado_project/ECE481/Lab1/divider_test/divider_test.srcs/sources_1/bd/Zynq_CPU_divider/ipshared/c2d4/src/rtl_divider.v
Read-only
Sources
Source File Properties
截图(Alt + A)
Tcl Console
Messages
Log
Reports
Design Runs

```verilog
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////


module rtl_divider(
    input clk,
    input [15:0] dividend,
    input [15:0] divisor,
    output [15:0] qoutient,
    output [15:0] remainder
    );

    reg [15:0] qoutientReg;
    reg [15:0] remainderReg;
    assign qoutient  = qoutientReg;
    assign remainder = remainderReg;

    always @(posedge clk) begin
        qoutientReg = dividend / divisor;
        remainderReg = dividend % divisor;
    end
endmodule

```

Figure 12. The Divider Source Code (Verilog)

The screenshot of the divider module is shown above. The code file is submitted together with this report in the zip package.

4 Answer the Questions

1.What is a bare metal test?

"bare metal" means that the computer purely combined with hardware without operating system. A bare metal test is the test that tests the system or desired software feature directly without involving of operating system.

2.How does the Zynq PS communicate with the IP blocks we created in this lab?

The Zynq PS mainly communicate with the IP blocks, which we created in PL (programmable logic) part via AXI interface. For instance, the multiplier/divider IP block we created is connected to AXI-Lite interface as a slave. The input number is from the *slv_reg0*, and the module output is connected to *slv_reg1*.

3.Briefly read out the AXI, and summarize the main features.

AXI means Advanced eXtensible Interface. It is a interconnection structure designed by ARM. It can support wide bandwidth and small latency design. Because it is a one-way channel, the number of gates is reduced and thus latency can be controlled. In our design, AXI interconnect is used to connect rtl_multiplier_v1.0 (Pre-Production), ZYNQ7 Processing System, and Processor System Reset. AXI standard is used to specify connections between the PS (Processing System) and PL (Programmable Logic). AXI Interconnects manages and directs traffic between attached AXI interfaces, and AXI Interfaces passes data, addresses, and hand-shaking signals between master and slave clients within the system [2]. The AXI protocol in particular exhibits the following key features [3]:

- address/control phases are separate from data phases
- byte strobes enable unaligned data transfers
- burst-based transactions possible with only start address issued
- read and write data channels are separate allowing low-cost Direct Memory Access (DMA)
- multiple outstanding addresses can be issued
- transactions can be completed out-of-order
- register stages are easily added for timing closure

4.Overall, what did you learn from this lab? Any difficulties?

As a short conclusion, we learned how to build a Vivado project including Zynq PS system and export the hardware part as a platform in Vitis. Also, we learned how to use Vitis IDE for design on a specific hardware, and how to use serial communication with FPGA board on Vitis. More importantly, we learned how to design and package our own IP blocks, how to connect it to AXI interface, and how to test the new IP block on FPGA board

One major difficuly we meet in this lab is how to run C program using Vitis IDE on FPGA board, since the operations on Vitis is different from SDK on the lab instruction, we cannot follow the instructions directly. After taking a long time searching online, finally we found out the correct steps to build a project on Vitis and run it on the hardware.

5 Reference

[1] ECE4810J Lab#1 instructions.

[2] The Zynq Book: Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC.

[3] ARM, “Introduction — Channel Definition” in *AMBA AXI Protocol Specification*, v1.0, June 2003.