Spring 2021

California State University, Northridge

Department of Electrical & Computer Engineering

Lab 3
IP Creation and Integration

March 23, 2021

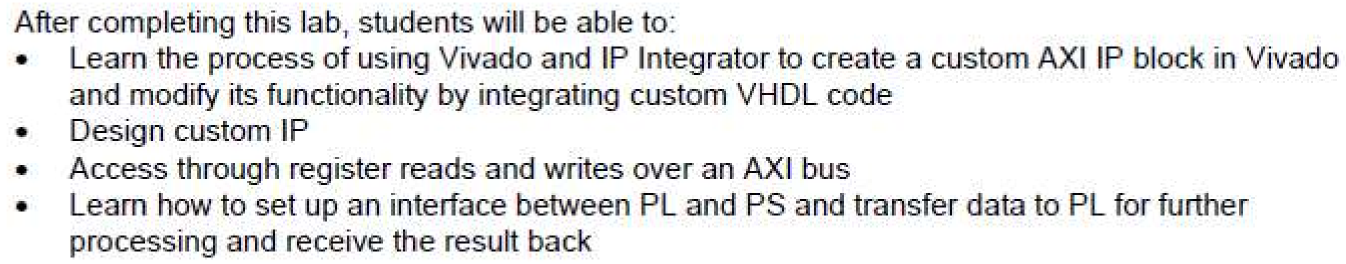
ECE 526L

Written By: Jose Luis Martinez

Introduction

In this lab we will be learning about IP creation and integration using the Vivado software. An IP stands for intellectual property in Vivado and it is used for creating design blocks for Soc's. We will be designing our own IPs in this lab and will also be integrating it into our block design. We will then use Vitis to implement into our program and do work for us. The specific IPs we will be creating are a multiplier and complex multiplier.

Objective



After completing this lab, students will be able to:

- Learn the process of using Vivado and IP Integrator to create a custom AXI IP block in Vivado and modify its functionality by integrating custom VHDL code
- Design custom IP
- Access through register reads and writes over an AXI bus
- Learn how to set up an interface between PL and PS and transfer data to PL for further processing and receive the result back

Methodology

I made both tasks into the same Vivado/Vitis project with separate IPs for each.

1. Create a Vivado project selecting the ZYBO board files as our target device.
2. Create a block design in the IP integrator in the project.
3. Add the ZYNQ processor to the block design and run auto configuration.
4. Create an IP for the multiplier.
    a. Go to tools and click Create and Package IP.
    b. Follow the steps in the lab manual for configuring the IP.
    c. Make a multiplier.vhd file that multiplies the inputs.
    d. Modify the IP to include the VHDL file and then repackage.
5. Create an IP for the multiplier.
    a. Go to tools and click Create and Package IP.
    b. Follow the steps in the lab manual for configuring the IP.
    c. Make a comp_mul.vhd file that performs complex multiplication.
    d. Modify the IP to include the VHDL file and then repackage.
6. Add each IP and run the auto config.
7. Validate the block design.
8. Generate output products.
9. Generate HDL wrapper.
10. Generate bitstream.
11. Export hardware with the bitstream file.
12. Create an Application Project and create a hardware platform from the xsa file in Vitis.
13. Modify the C file to test out both IP blocks.
14. Program the FPGA and verify the results in the terminal.

Results/Code

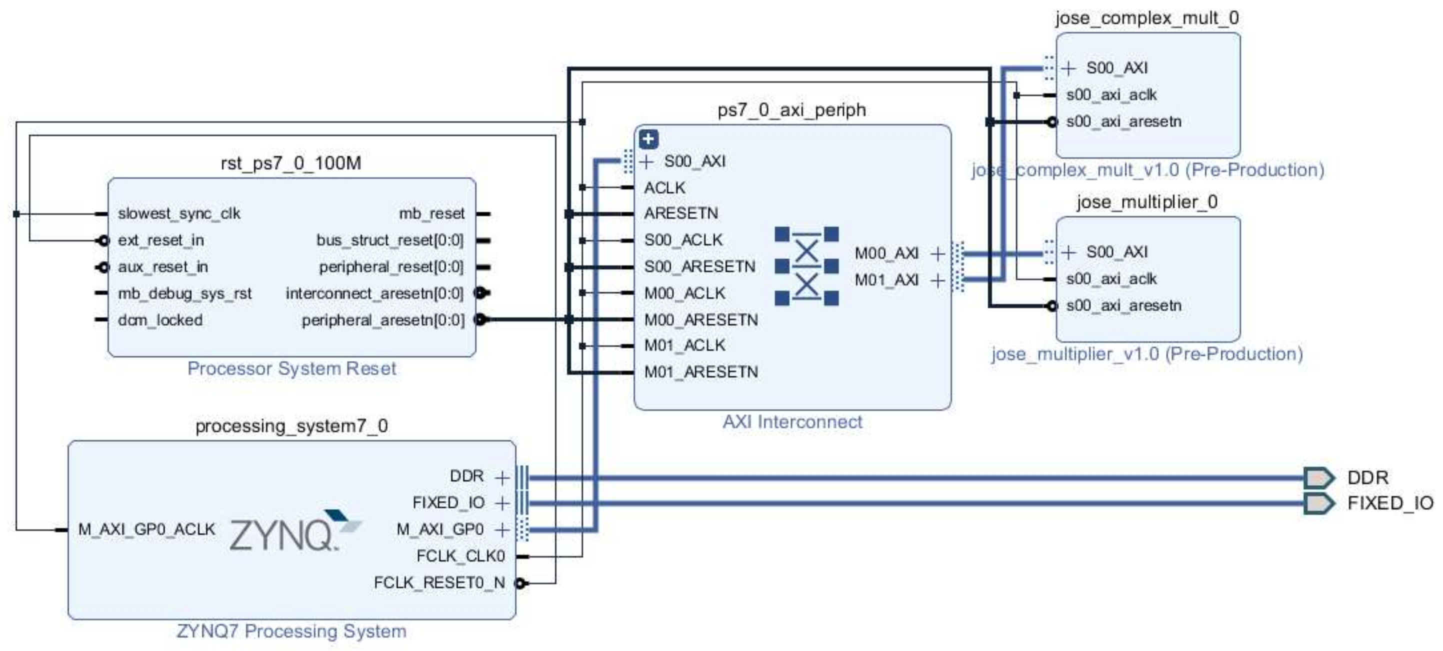
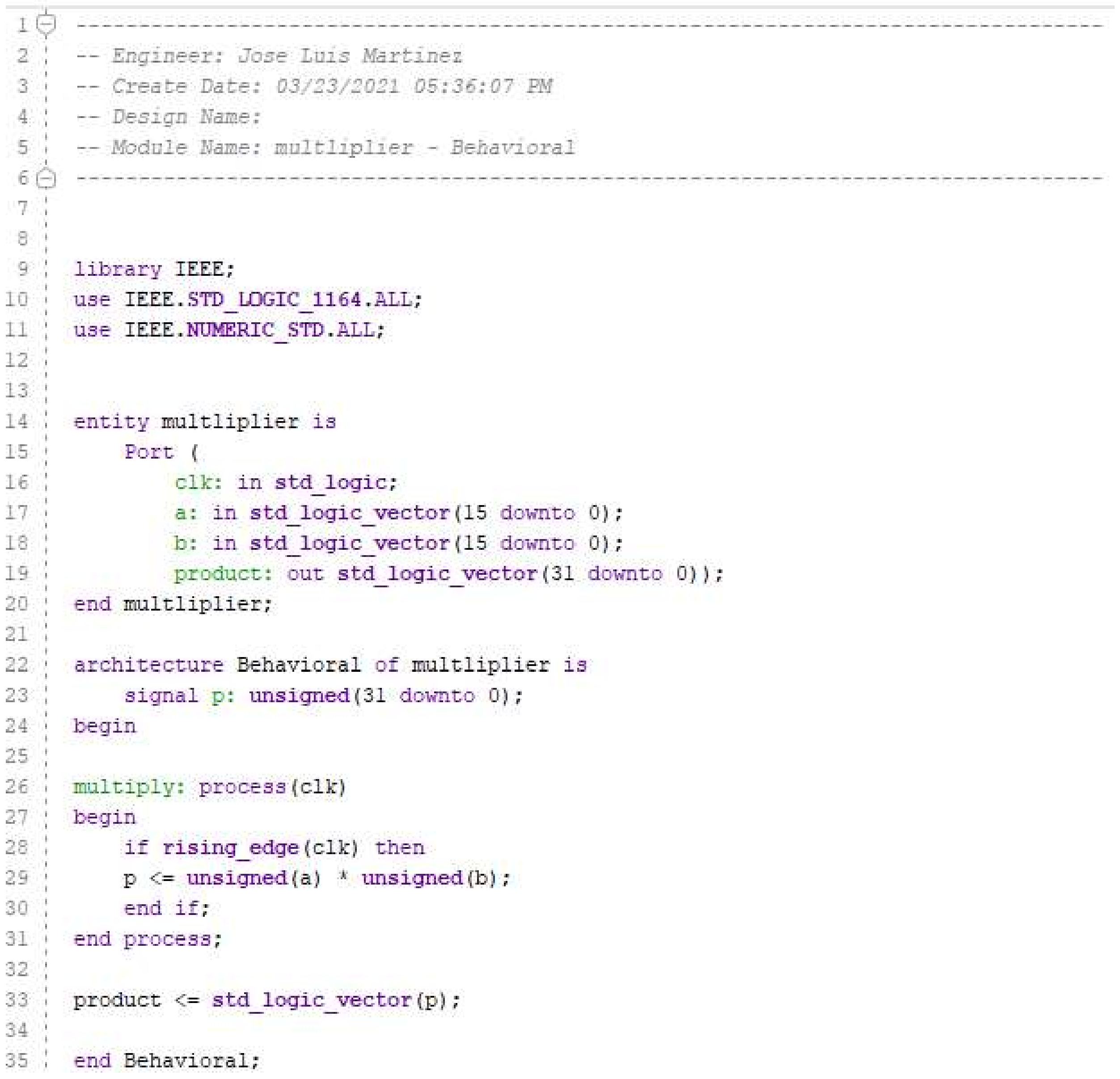
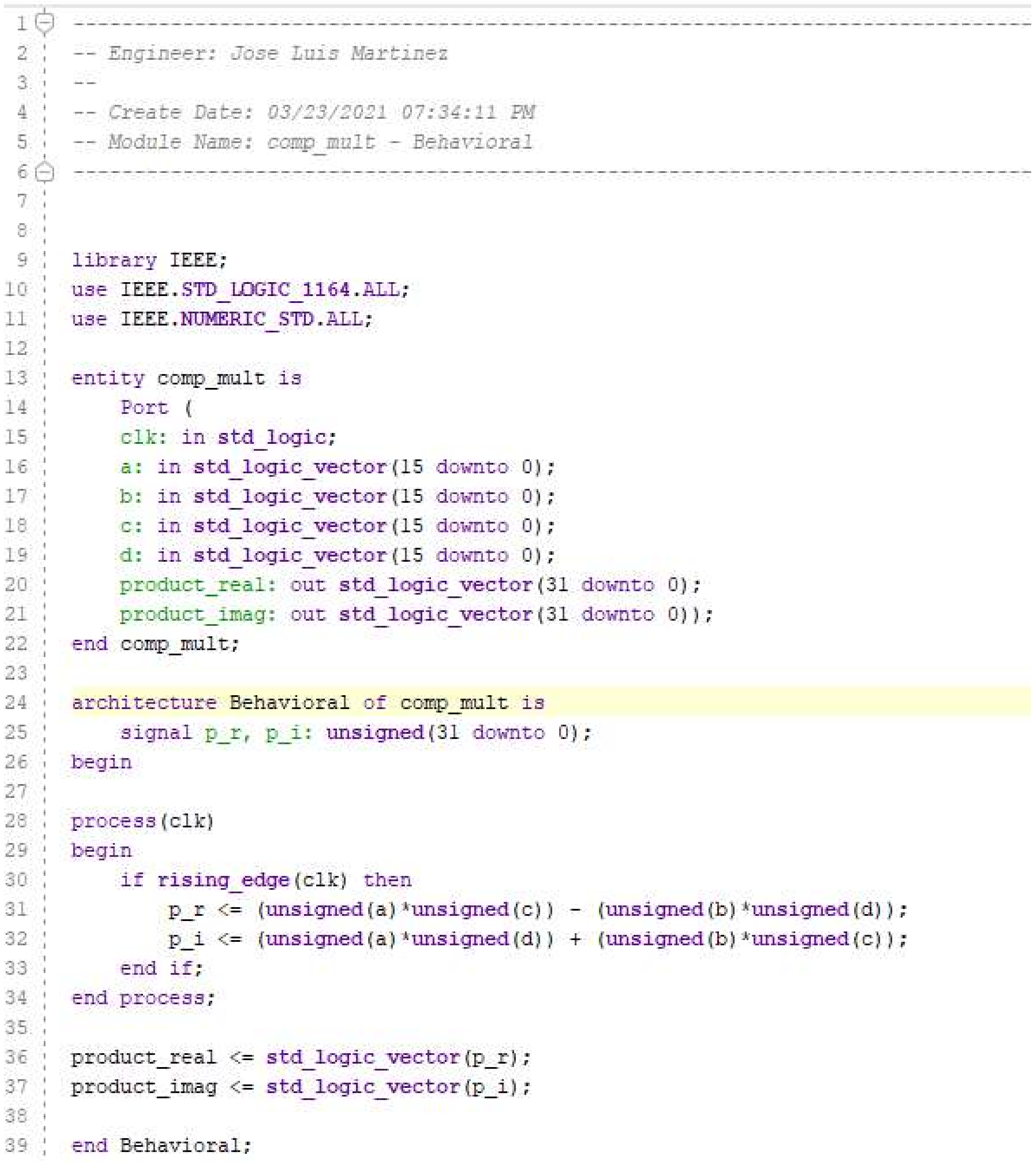


Fig. 1 Block Design with Multiplier and Complex Multiplier



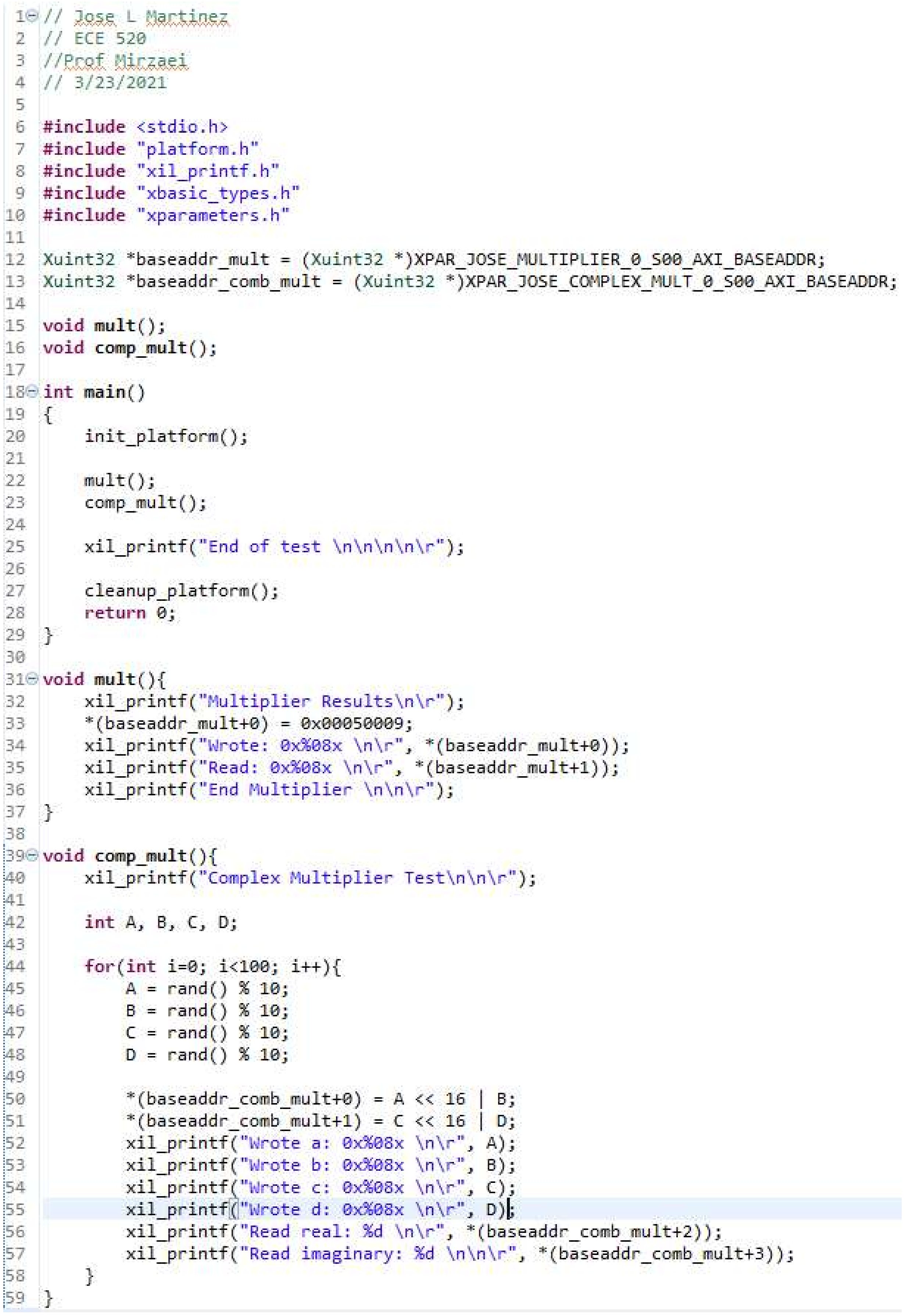
```vhdl
--------------------------------------------------------------------------------
-- Engineer: Jose Luis Martinez
-- Create Date: 03/23/2021 05:36:07 PM
-- Design Name:
-- Module Name: multliplier - Behavioral
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity multliplier is
    Port (
        clk: in std_logic;
        a: in std_logic_vector(15 downto 0);
        b: in std_logic_vector(15 downto 0);
        product: out std_logic_vector(31 downto 0));
end multliplier;

architecture Behavioral of multliplier is
    signal p: unsigned(31 downto 0);
begin

multiply: process(clk)
begin
    if rising_edge(clk) then
    p <= unsigned(a) * unsigned(b);
    end if;
end process;

product <= std_logic_vector(p);

end Behavioral;
```

Fig. 2 multiplier.vhd



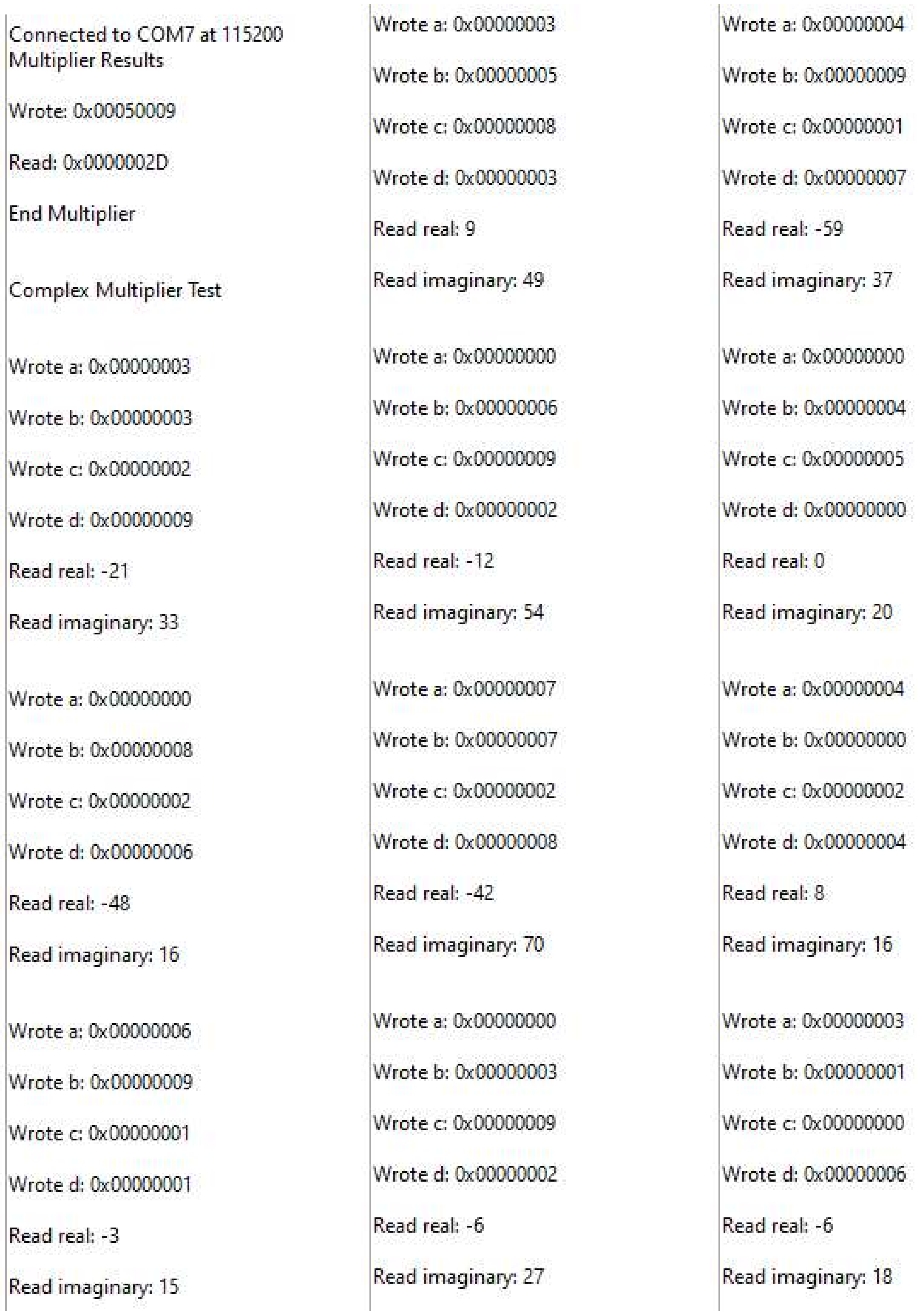
```vhdl
----------------------------------------------------------------------------------
-- Engineer: Jose Luis Martinez
--
-- Create Date: 03/23/2021 07:34:11 PM
-- Module Name: comp_mult - Behavioral
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity comp_mult is
    Port (
    clk: in std_logic;
    a: in std_logic_vector(15 downto 0);
    b: in std_logic_vector(15 downto 0);
    c: in std_logic_vector(15 downto 0);
    d: in std_logic_vector(15 downto 0);
    product_real: out std_logic_vector(31 downto 0);
    product_imag: out std_logic_vector(31 downto 0));
end comp_mult;

architecture Behavioral of comp_mult is
    signal p_r, p_i: unsigned(31 downto 0);
begin

process(clk)
begin
    if rising_edge(clk) then
        p_r <= (unsigned(a)*unsigned(c)) - (unsigned(b)*unsigned(d));
        p_i <= (unsigned(a)*unsigned(d)) + (unsigned(b)*unsigned(c));
    end if;
end process;

product_real <= std_logic_vector(p_r);
product_imag <= std_logic_vector(p_i);

end Behavioral;
```

Fig. 3 comp_mult.vhd



```c
// Jose L Martinez
// ECE 520
//Prof Mirzaei
// 3/23/2021

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

Xuint32 *baseaddr_mult = (Xuint32 *)XPAR_JOSE_MULTIPLIER_0_S00_AXI_BASEADDR;
Xuint32 *baseaddr_comb_mult = (Xuint32 *)XPAR_JOSE_COMPLEX_MULT_0_S00_AXI_BASEADDR;

void mult();
void comp_mult();

int main()
{
    init_platform();

    mult();
    comp_mult();

    xil_printf("End of test \n\n\n\n\r");

    cleanup_platform();
    return 0;
}

void mult(){
    xil_printf("Multiplier Results\n\r");
    *(baseaddr_mult+0) = 0x00050009;
    xil_printf("Wrote: 0x%08x \n\r", *(baseaddr_mult+0));
    xil_printf("Read: 0x%08x \n\r", *(baseaddr_mult+1));
    xil_printf("End Multiplier \n\n\r");
}

void comp_mult(){
    xil_printf("Complex Multiplier Test\n\n\r");

    int A, B, C, D;

    for(int i=0; i<100; i++){
        A = rand() % 10;
        B = rand() % 10;
        C = rand() % 10;
        D = rand() % 10;

        *(baseaddr_comb_mult+0) = A << 16 | B;
        *(baseaddr_comb_mult+1) = C << 16 | D;
        xil_printf("Wrote a: 0x%08x \n\r", A);
        xil_printf("Wrote b: 0x%08x \n\r", B);
        xil_printf("Wrote c: 0x%08x \n\r", C);
        xil_printf("Wrote d: 0x%08x \n\r", D);
        xil_printf("Read real: %d \n\r", *(baseaddr_comb_mult+2));
        xil_printf("Read imaginary: %d \n\n\r", *(baseaddr_comb_mult+3));
    }
}
```

Fig. 4 Vitis C code (main.h)


```
Connected to COM7 at 115200
Multiplier Results

Wrote: 0x00050009

Read: 0x0000002D

End Multiplier


Complex Multiplier Test


Wrote a: 0x00000003

Wrote b: 0x00000003

Wrote c: 0x00000002

Wrote d: 0x00000009

Read real: -21

Read imaginary: 33


Wrote a: 0x00000000

Wrote b: 0x00000008

Wrote c: 0x00000002

Wrote d: 0x00000006

Read real: -48

Read imaginary: 16


Wrote a: 0x00000006

Wrote b: 0x00000009

Wrote c: 0x00000001

Wrote d: 0x00000001

Read real: -3

Read imaginary: 15
```

```
Wrote a: 0x00000003

Wrote b: 0x00000005

Wrote c: 0x00000008

Wrote d: 0x00000003

Read real: 9

Read imaginary: 49


Wrote a: 0x00000000

Wrote b: 0x00000006

Wrote c: 0x00000009

Wrote d: 0x00000002

Read real: -12

Read imaginary: 54


Wrote a: 0x00000007

Wrote b: 0x00000007

Wrote c: 0x00000002

Wrote d: 0x00000008

Read real: -42

Read imaginary: 70


Wrote a: 0x00000000

Wrote b: 0x00000003

Wrote c: 0x00000009

Wrote d: 0x00000002

Read real: -6

Read imaginary: 27
```

```
Wrote a: 0x00000004

Wrote b: 0x00000009

Wrote c: 0x00000001

Wrote d: 0x00000007

Read real: -59

Read imaginary: 37


Wrote a: 0x00000000

Wrote b: 0x00000004

Wrote c: 0x00000005

Wrote d: 0x00000000

Read real: 0

Read imaginary: 20


Wrote a: 0x00000004

Wrote b: 0x00000000

Wrote c: 0x00000002

Wrote d: 0x00000004

Read real: 8

Read imaginary: 16


Wrote a: 0x00000003

Wrote b: 0x00000001

Wrote c: 0x00000000

Wrote d: 0x00000006

Read real: -6

Read imaginary: 18
```

Fig. 5 Serial Terminal Output

Analysis

I managed to combine both parts of the lab into one Block Design as shown in **Fig. 1**. I then created an Application Project in Vitis and tested both IP designs at once as shown in **Fig. 4.** We can then see the results of my code in **Fig. 5.**

For the multiplier I am using *baseaddr_mult to store the address of the first register from the multiplier IP block.

For the Multiplier I tested out the following:

*(baseaddr_mult+0) = 0x 00050009

Meaning: A=5 and B=9.

So our result should be 45. To confirm that our multiplier is working properly I printed the result by accessing the next register.

The result of the *(baseaddr_mult+1) is printed at a value of 0x0000002D, in decimal form that is 45 which confirms that our multiplier is working as intended.

For the complex multiplier I am using *baseaddr_comp_mult to store the address of the first register from the multiplier IP block. We can access the other 3 registers by altering the value of this address.

For the complex multiplier block I tested out 100 random sets of complex numbers. This was achieved by using the C function rand() and then shifting left by 16 bits.

A couple examples from **Fig. 5** are shown below:

A = 0x00000003, B = 0x00000003, C = 0x00000002, and D =0x0000000 9.

Meaning that our complex numbers should look like 3+j3 and 2+j9.

So then *(baseaddr_comp_mult+0) = 0x00030003 and *(baseaddr_comp_mult+0) = 0x00020009.

Our result should be -21+j39 and as seen in **Fig. 5** we got the correct answer.

A = 0x00000000, B = 0x00000008, C = 0x00000002, and D =0x0000000 6.

Meaning that our complex numbers should look like 0+j8 and 2+j6.

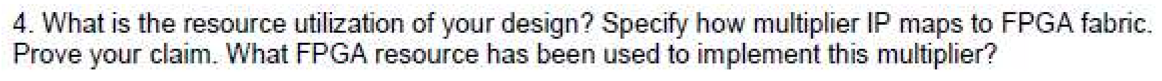
So then *(baseaddr_comp_mult+0) = 0x00000008 and *(baseaddr_comp_mult+0) = 0x00020006.

Our result should be -48+16j and as seen in **Fig. 5** we got the correct answer.

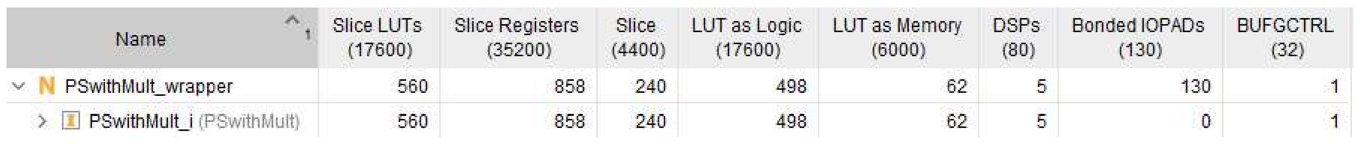
Conclusion

In conclusion, we learned how to create our own IP in vivado, modify our IP to include any vhdl files or registers we want, and how to access and implement our IP in the Vitis IDE. I started off by creating the vhdl fies that holds our logic for multiplication and complex multiplication. Then by adding them to their respective IP we added each IP to the block design inVivado. I then exported the hardware with the bitstream to Vitis in order to write software for the SOC. I managed to test the multiplier and complex multiplier and with the results shown in **Fig. 5** we are able to verify that with the results our IP designs are both working as expected.

Questions

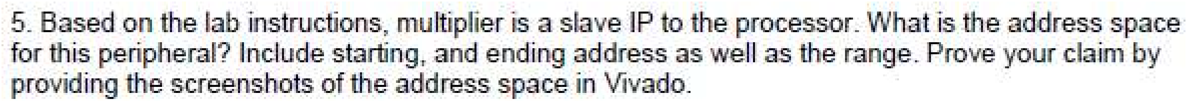


4. What is the resource utilization of your design? Specify how multiplier IP maps to FPGA fabric. Prove your claim. What FPGA resource has been used to implement this multiplier?

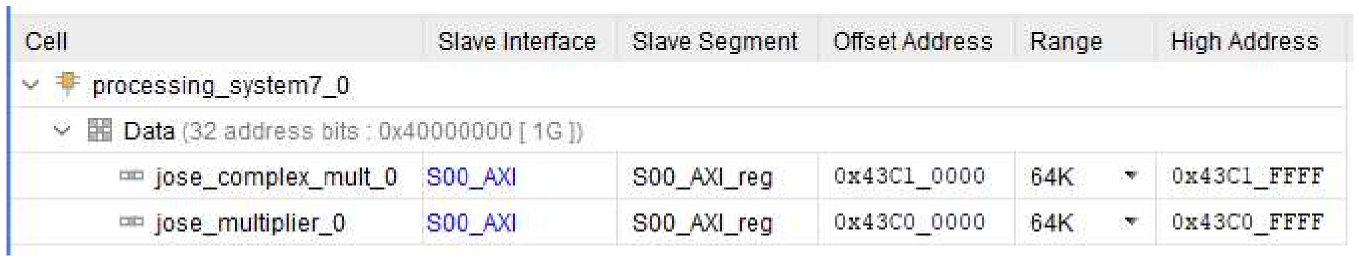


Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	LUT as Memory (6000)	DSPs (80)	Bonded IOPADs (130)	BUFGCTRL (32)
PSwithMult_wrapper	560	858	240	498	62	5	130	1
PSwithMult_i (PSwithMult)	560	858	240	498	62	5	0	1

Fig. 6 Multiplier Utilization Report

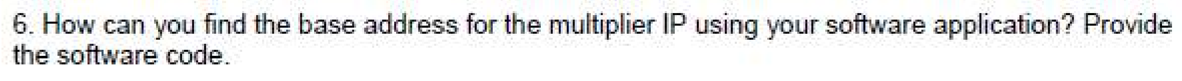


5. Based on the lab instructions, multiplier is a slave IP to the processor. What is the address space for this peripheral? Include starting, and ending address as well as the range. Prove your claim by providing the screenshots of the address space in Vivado.

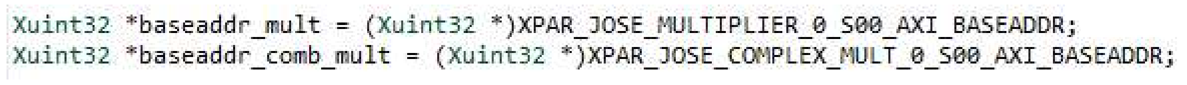


Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
jose_complex_mult_0	S00_AXI	S00_AXI_reg	0x43C1_0000	64K	0x43C1_FFFF
jose_multiplier_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF

Fig. 7 IP Slave Addresses



6. How can you find the base address for the multiplier IP using your software application? Provide the software code.



```
Xuint32 *baseaddr_mult = (Xuint32 *)XPAR_JOSE_MULTIPLIER_0_S00_AXI_BASEADDR;
Xuint32 *baseaddr_comb_mult = (Xuint32 *)XPAR_JOSE_COMPLEX_MULT_0_S00_AXI_BASEADDR;
```

Fig. 8 Variable to store the base address of the respective IP.

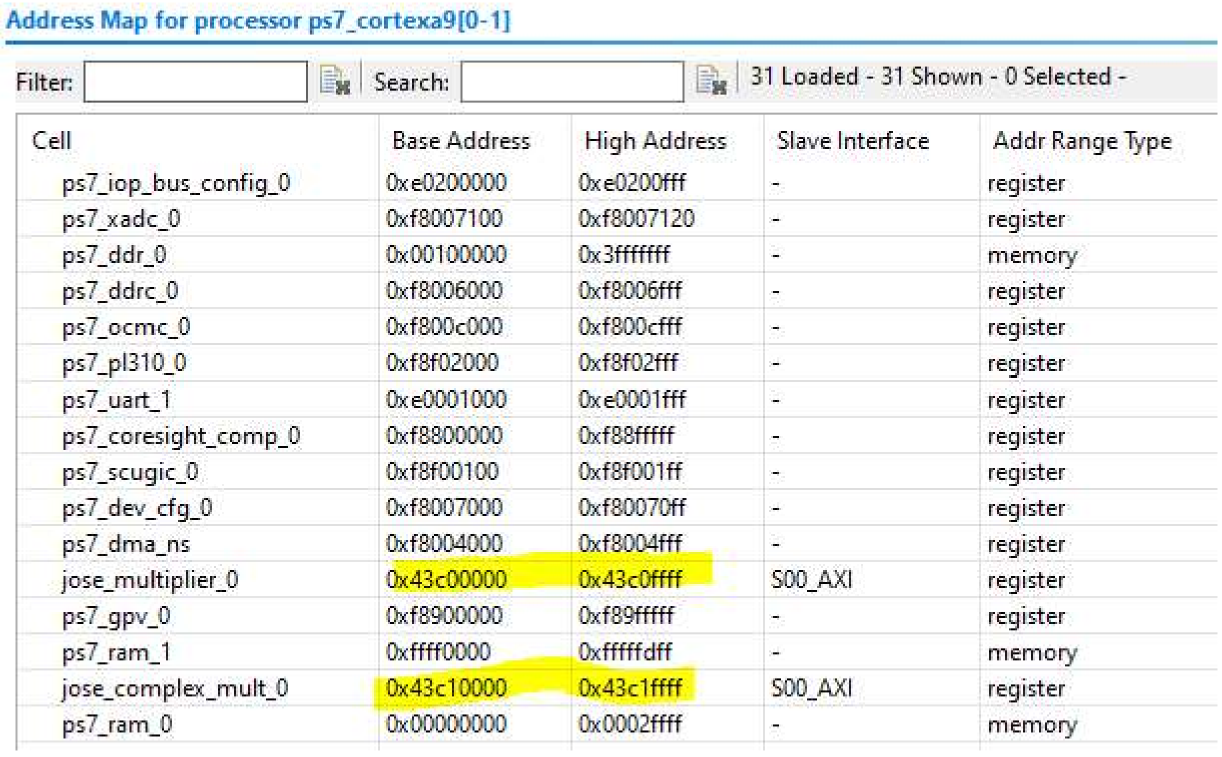
7. State how the design is partitioned to software and hardware. Consider the example that you implemented in this lab and explain the flow of the data completely from the time that you enter the inputs to when you observe the outputs. Explain exactly how the data is passed from software to hardware and how the results is returned.

In lab 3 for the hardware part we had to design our own IPs and add them to an IP Block Design. Then we exported our hardware and bitstream file so that we can use it in our software design. In Vitis IDE we use the base addresses of the IPs in order to access their registers. For the multiplier we use the first register to pass on the values we want to multiply and the second register to access the product. For the complex multiplier we used the first two registers to send a complex number to each and the last two will give us the product of their real and imaginary numbers.

8. In step 2 (Modify the software application), the following line specifies the base address of the multiplier IP:

```
Xuint32 *baseaddr_p = (Xuint32 *)XPAR_MY_MULTIPLIER_0_S00_AXI_BASEADDR;
```

How do we find this base address? What is the value of the parameter XPAR_MY_MULTIPLIER_0_S00_AXI_BASEADDR used as pointer to the data being passed to the multiplier?



Address Map for processor ps7_cortexa9[0-1]

Filter: Search: 31 Loaded - 31 Shown - 0 Selected -

Cell	Base Address	High Address	Slave Interface	Addr Range Type
ps7_iop_bus_config_0	0xe0200000	0xe0200fff	-	register
ps7_xadc_0	0xf8007100	0xf8007120	-	register
ps7_ddr_0	0x00100000	0x3fffffff	-	memory
ps7_ddrc_0	0xf8006000	0xf8006fff	-	register
ps7_ocmc_0	0xf800c000	0xf800cfff	-	register
ps7_pl310_0	0xf8f02000	0xf8f02fff	-	register
ps7_uart_1	0xe0001000	0xe0001fff	-	register
ps7_coresight_comp_0	0xf8800000	0xf88fffff	-	register
ps7_scugic_0	0xf8f00100	0xf8f001ff	-	register
ps7_dev_cfg_0	0xf8007000	0xf80070ff	-	register
ps7_dma_ns	0xf8004000	0xf8004fff	-	register
jose_multiplier_0	0x43c00000	0x43c0ffff	S00_AXI	register
ps7_gpv_0	0xf8900000	0xf89fffff	-	register
ps7_ram_1	0xffff0000	0xfffffdff	-	memory
jose_complex_mult_0	0x43c10000	0x43c1ffff	S00_AXI	register
ps7_ram_0	0x00000000	0x0002ffff	-	memory

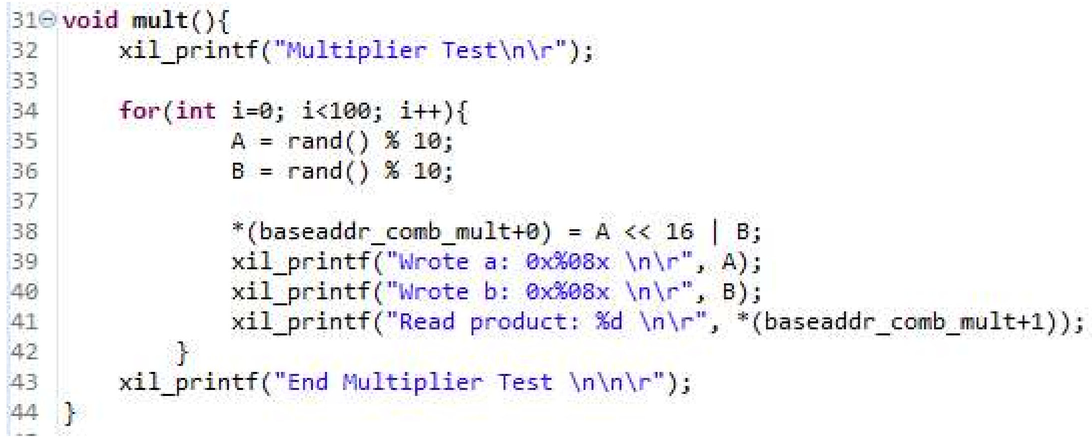
Fig. 9 IP Base Address

The value of the parameter is the address of our respective IP.

9. What is the physical address of the input and output data to/from the multiplier?

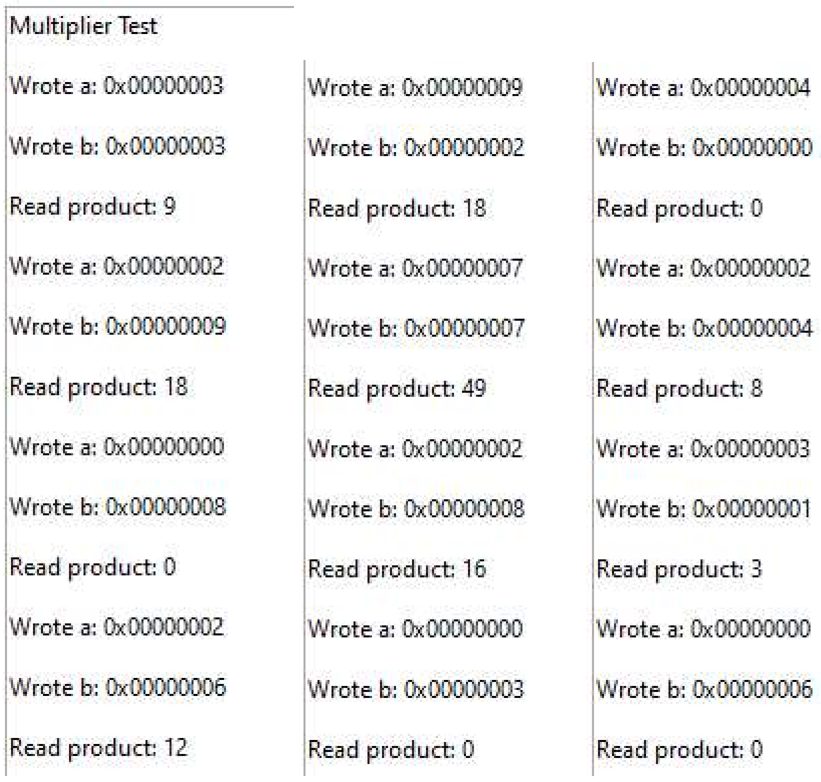
Multiplier IP: inputs = 0x48c00000 | output = 0x48c00001.

10. (Prelab) Use for loop and arrays, modify the software application to pass 100 pairs of numbers to the multiplier and calculate the outputs using the multiplier output. Show output data on serial terminal to your instructor as part of the lab function verification.



```
void mult(){
    xil_printf("Multiplier Test\n\r");

    for(int i=0; i<100; i++){
            A = rand() % 10;
            B = rand() % 10;

            *(baseaddr_comb_mult+0) = A << 16 | B;
            xil_printf("Wrote a: 0x%08x \n\r", A);
            xil_printf("Wrote b: 0x%08x \n\r", B);
            xil_printf("Read product: %d \n\r", *(baseaddr_comb_mult+1));
        }
    xil_printf("End Multiplier Test \n\n\r");
}
```

Fig. 10 For loop for testing the multiplier IP

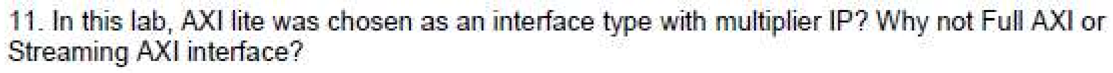


```
Multiplier Test

Wrote a: 0x00000003

Wrote b: 0x00000003

Read product: 9

Wrote a: 0x00000002

Wrote b: 0x00000009

Read product: 18

Wrote a: 0x00000000

Wrote b: 0x00000008

Read product: 0

Wrote a: 0x00000002

Wrote b: 0x00000006

Read product: 12
```

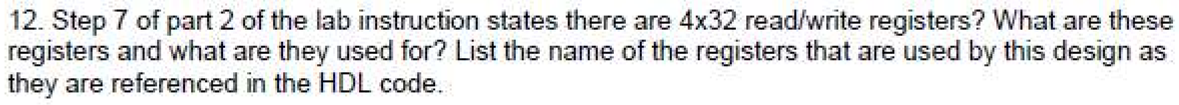
```
Wrote a: 0x00000009

Wrote b: 0x00000002

Read product: 18

Wrote a: 0x00000007

Wrote b: 0x00000007

Read product: 49

Wrote a: 0x00000002

Wrote b: 0x00000008

Read product: 16

Wrote a: 0x00000000

Wrote b: 0x00000003

Read product: 0
```

```
Wrote a: 0x00000004

Wrote b: 0x00000000

Read product: 0

Wrote a: 0x00000002

Wrote b: 0x00000004

Read product: 8

Wrote a: 0x00000003

Wrote b: 0x00000001

Read product: 3

Wrote a: 0x00000000

Wrote b: 0x00000006

Read product: 0
```

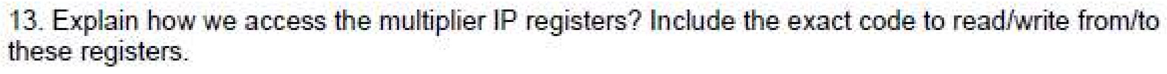
Fig. 11 Multiplier Loop Results


11. In this lab, AXI lite was chosen as an interface type with multiplier IP? Why not Full AXI or Streaming AXI interface?

AXI lite was used for the IPs in this lab because it is more appropriate for less intensive applications such as just passing one or two 32bit numbers to the registers and then reading the results from its other registers.


12. Step 7 of part 2 of the lab instruction states there are 4x32 read/write registers? What are these registers and what are they used for? List the name of the registers that are used by this design as they are referenced in the HDL code.

These registers are used for input and output of the IP and can be accessed by the processor and the modules within the IP. The register names are as follows: slv_reg0, slv_reg1, slv_reg2, and slv_reg3.


13. Explain how we access the multiplier IP registers? Include the exact code to read/write from/to these registers.

In order to access the IPs we need to use the base address of the IP and write/read to it.

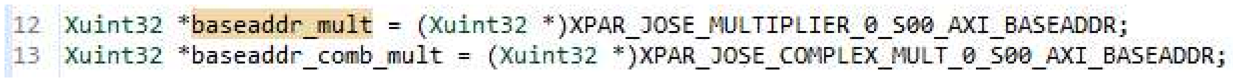

```
Xuint32 *baseaddr_mult = (Xuint32 *)XPAR_JOSE_MULTIPLIER_0_S00_AXI_BASEADDR;
Xuint32 *baseaddr_comb_mult = (Xuint32 *)XPAR_JOSE_COMPLEX_MULT_0_S00_AXI_BASEADDR;
```

Fig. 12 Base address of IPs

To read and write we can do the following:

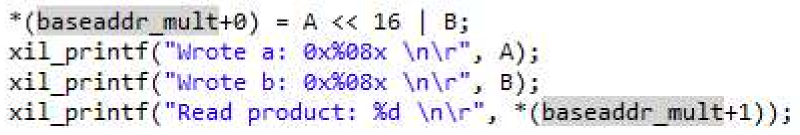
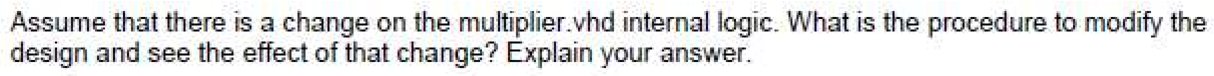

```
*(baseaddr_mult+0) = A << 16 | B;
xil_printf("Wrote a: 0x%08x \n\r", A);
xil_printf("Wrote b: 0x%08x \n\r", B);
xil_printf("Read product: %d \n\r", *(baseaddr_mult+1));
```

Fig. 13 Reading and writing to the IP base registers


Assume that there is a change on the multiplier.vhd internal logic. What is the procedure to modify the design and see the effect of that change? Explain your answer.

If you want to modify an IP design, in your block design right click the IP you wish to modify. Then select "edit in IP Packager" and a new Vivado window will appear with the HDL code for the IP. In there, you can edit your IP and change the behavior. Once you are done modifying the IP you can simply just review and repackage the IP and it will take you back to your main project. In the main Vivado window you just update the IP and you should be good to Generate Bitstream. Then in Vitis you just update the Hardware Specification and that should be all for editing the IP.