High Level Digital Design

Lab 5 Report: Implementation of a radix-4 Booth multiplier IP with an AXI4 interface

Miguel Sanchez-Arjona & Adriana Costafreda

November 24, 2023

0. Goals

The goal of this laboratory assignment is to design and implement of an AXI4 IP based on the 16-bit x 16-bit radix-4 Booth multiplier designed in laboratory 4. This IP will be managed by the HPS of the Cyclone V device by means of interrupts. The laboratory session will consist in creating a hardware interface which will be mapped to the hardware code described in laboratory 4 with System Verilog. Moreover, a software project will be described in C++ to create the application which will be capable of introducing the multiplier and multiplicand numbers through the terminal, providing the result in a controlled manner by means of the interrupts which were described in the Booth multiplier code.

1. Design of the IP and hardware platform

As it was introduced in Laboratory 2, we will connect the booth multiplier to the HPS to the AXI Lite Interface, using the code which has been provided to us. The interface will manage the communication with the HPS using the AXI Lite protocol. Figure 1 shows a simplified diagram of the interconnection between all the components in the design.

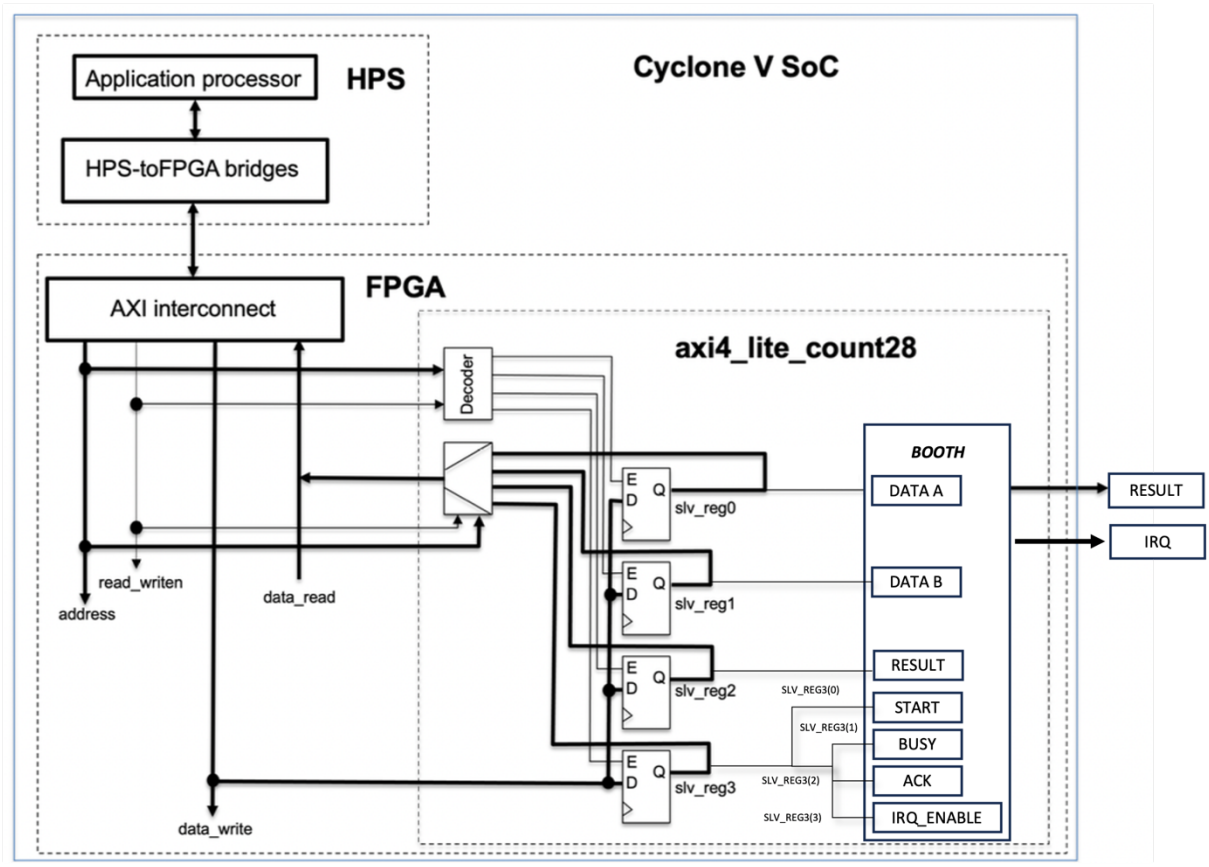


Figure 1. Organization of the components of the design.

The first step to design the hardware platform is to instantiate the booth multiplier developed in laboratory 4 in the AXI4 interface which is provided to us in the file *axi4_lite_booth.vhd*.

Thus, we need to modify the VHDL file to introduce the input and output signals of the booth multiplier. To do that we must know that:

- Input ***data_a*** (multiplicand) is stored in *slv_reg0*. This register will be written by HPS.
- Input ***data_b*** (multiplier) is stored in *slv_reg1.* The register is also written by HPS.
- Output ***result*** of the multiplier is stored in *slb_reg2* and it is read by the HPS. The result will be provided by the booth multiplier; thus, we will create a signal.
- Input ***start*** of the multiplier is stored in the first position (0) of the *slv_reg3.*
- Output ***busy*** of the multiplier is mapped to the second position (1) *slv_reg3*. The logic of the output busy will be read by the HPS, which is provided by the booth multiplier hardware system developed in laboratory 4.
- Input ***ack*** of the multiplier will be stored in the third position of the *slv_reg3.* This value will be written by the HPS.
- Regarding the input ***irq_enable,*** it will be stored in the fourth position of the *slv_reg3.* The value of the *irq_enable* will be written by the HPS.
- Lastly, the output ***irq*** defined in the booth multiplier will be propagated as an external output.

Therefore, we must modify the AXI4 interface following the previous points. In the code below, it can be easily seen highlighted the modifications that were done to properly instantiate the hardware platform.

The first modification has been to include a signal in the AXI4 lite booth entity which accounts for the *irq* output. This output will be the interrupt signal to manage the IP. Secondly, we instantiate the booth component module in the hardware platform. It includes the inputs and outputs of the component and additionally, we add signals for the *result* and *busy* outputs. Then, in the process we map the inputs and outputs to its corresponding memory address. It consists of mapping the signals to the corresponding *slv_reg* and positions as it is detailed. Lastly, we instantiate the functional part of the IP mapping the booth multiplier module to the AXI4 interface. The inputs and outputs of the system are properly mapped to the corresponding memory positions.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity axi4_lite_booth is
    generic (

        -- Width of S_AXI data bus
        C_S_AXI_DATA_WIDTH  : integer   := 32;
        -- Width of S_AXI address bus
        C_S_AXI_ADDR_WIDTH  : integer   := 4
    );
    port (

        -- Section where external ports of the IP are declared
```

```
        -- IRQ Output signal
        irq         : out std_logic;


        -- Global Clock Signal
        clk         : in std_logic;
        -- Global Reset Signal. This Signal is Active LOW
        reset_n     : in std_logic;
        -- Write address (issued by master, acceped by Slave)
        axs_s0_AXI_AWADDR   : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Write channel Protection type. This signal indicates the
            -- privilege and security level of the transaction, and whether
            -- the transaction is a data access or an instruction access.
        axs_s0_AXI_AWPROT   : in std_logic_vector(2 downto 0);
        -- Write address valid. This signal indicates that the master signaling
            -- valid write address and control information.
        axs_s0_AXI_AWVALID  : in std_logic;
        -- Write address ready. This signal indicates that the slave is ready
            -- to accept an address and associated control signals.
        axs_s0_AXI_AWREADY  : out std_logic;
        -- Write data (issued by master, acceped by Slave)
        axs_s0_AXI_WDATA    : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Write strobes. This signal indicates which byte lanes hold
            -- valid data. There is one write strobe bit for each eight
            -- bits of the write data bus.
        axs_s0_AXI_WSTRB    : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto
0);
        -- Write valid. This signal indicates that valid write
            -- data and strobes are available.
        axs_s0_AXI_WVALID   : in std_logic;
        -- Write ready. This signal indicates that the slave
            -- can accept the write data.
        axs_s0_AXI_WREADY   : out std_logic;
        -- Write response. This signal indicates the status
            -- of the write transaction.
        axs_s0_AXI_BRESP    : out std_logic_vector(1 downto 0);
        -- Write response valid. This signal indicates that the channel
            -- is signaling a valid write response.
        axs_s0_AXI_BVALID   : out std_logic;
        -- Response ready. This signal indicates that the master
            -- can accept a write response.
        axs_s0_AXI_BREADY   : in std_logic;
        -- Read address (issued by master, acceped by Slave)
        axs_s0_AXI_ARADDR   : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Protection type. This signal indicates the privilege
            -- and security level of the transaction, and whether the
            -- transaction is a data access or an instruction access.
        axs_s0_AXI_ARPROT   : in std_logic_vector(2 downto 0);
        -- Read address valid. This signal indicates that the channel
            -- is signaling valid read address and control information.
        axs_s0_AXI_ARVALID  : in std_logic;
```

```
        -- Read address ready. This signal indicates that the slave is
            -- ready to accept an address and associated control signals.
        axs_s0_AXI_ARREADY  : out std_logic;
        -- Read data (issued by slave)
        axs_s0_AXI_RDATA    : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Read response. This signal indicates the status of the
            -- read transfer.
        axs_s0_AXI_RRESP    : out std_logic_vector(1 downto 0);
        -- Read valid. This signal indicates that the channel is
            -- signaling the required read data.
        axs_s0_AXI_RVALID   : out std_logic;
        -- Read ready. This signal indicates that the master can
            -- accept the read data and response information.
        axs_s0_AXI_RREADY   : in std_logic
    );
end axi4_lite_booth;

architecture arch_imp of axi4_lite_booth is

    -- AXI4LITE signals
    signal axi_awaddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_awready  : std_logic;
    signal axi_wready   : std_logic;
    signal axi_bresp    : std_logic_vector(1 downto 0);
    signal axi_bvalid   : std_logic;
    signal axi_araddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_arready  : std_logic;
    signal axi_rdata    : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal axi_rresp    : std_logic_vector(1 downto 0);
    signal axi_rvalid   : std_logic;

    -- Example-specific design signals
    -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
    -- ADDR_LSB is used for addressing 32/64 bit registers/memories
    -- ADDR_LSB = 2 for 32 bits (n downto 2)
    -- ADDR_LSB = 3 for 64 bits (n downto 3)
    constant ADDR_LSB  : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
    constant OPT_MEM_ADDR_BITS : integer := 1;
    ------------------------------------------------
    ---- Signals for user logic register space example
    --------------------------------------------------
    ---- Number of Slave Registers 4
    signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg_rden : std_logic;
    signal slv_reg_wren : std_logic;
    signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal byte_index   : integer;
    signal aw_en    : std_logic;
```

```
    ------------------------------------------------
    ---- User declarations (components and signals)
    ---- to be included here
    ------------------------------------------------

    component booth is
    port(
        clk: in std_logic;       -- clock input, active by rising edge
        resetn: in std_logic;    -- synchronous reset input, active low
        start: in std_logic;     -- start signal that activates the multiplication
        ack: in std_logic;       -- input used to deassert the IRQ and busy outputs
        irq_enable: in std_logic;  -- the signal enables, at high level, the
-- activation of the irq output
        data_a: in std_logic_vector(15 downto 0);  -- 16-bit multiplicand
        data_b: in std_logic_vector(15 downto 0);  -- 16-bit multiplier
        busy: out std_logic;     -- output that indicates that a multiplication ---
-- process is in progress
        irq: out std_logic;      -- IRQ signal activated when the multiplication
-- has completed
        result: out std_logic_vector(31 downto 0) -- result of the multiplication
    );
    end component;

    signal result: std_logic_vector(31 downto 0);  -- result signal declaration
    signal busy: std_logic;                        -- busy signal declared

begin
    -- I/O Connections assignments

    axs_s0_AXI_AWREADY  <= axi_awready;
    axs_s0_AXI_WREADY   <= axi_wready;
    axs_s0_AXI_BRESP    <= axi_bresp;
    axs_s0_AXI_BVALID   <= axi_bvalid;
    axs_s0_AXI_ARREADY  <= axi_arready;
    axs_s0_AXI_RDATA    <= axi_rdata;
    axs_s0_AXI_RRESP    <= axi_rresp;
    axs_s0_AXI_RVALID   <= axi_rvalid;
    -- Implement axi_awready generation
    -- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
    -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
    -- de-asserted when reset is low.

    process (clk)
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          axi_awready <= '0';
          aw_en <= '1';
        else
```

```
          if (axi_awready = '0' and axs_s0_AXI_AWVALID = '1' and axs_s0_AXI_WVALID
= '1' and aw_en = '1') then
            -- slave is ready to accept write address when
            -- there is a valid write address and write data
            -- on the write address and data bus. This design
            -- expects no outstanding transactions.
            axi_awready <= '1';
            elsif (axs_s0_AXI_BREADY = '1' and axi_bvalid = '1') then
                aw_en <= '1';
                axi_awready <= '0';
          else
            axi_awready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_awaddr latching
    -- This process is used to latch the address when both
    -- S_AXI_AWVALID and S_AXI_WVALID are valid.

    process (clk)
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          axi_awaddr <= (others => '0');
        else
          if (axi_awready = '0' and axs_s0_AXI_AWVALID = '1' and axs_s0_AXI_WVALID
= '1' and aw_en = '1') then
            -- Write Address latching
            axi_awaddr <= axs_s0_AXI_AWADDR;
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_wready generation
    -- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
    -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
    -- de-asserted when reset is low.

    process (clk)
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          axi_wready <= '0';
        else
          if (axi_wready = '0' and axs_s0_AXI_WVALID = '1' and axs_s0_AXI_AWVALID =
'1' and aw_en = '1') then
              -- slave is ready to accept write data when
              -- there is a valid write address and write data
```

```
              -- on the write address and data bus. This design
              -- expects no outstanding transactions.
              axi_wready <= '1';
          else
            axi_wready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement memory mapped register select and write logic generation
    -- The write data is accepted and written to memory mapped registers when
    -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
strobes are used to
    -- select byte enables of slave registers while writing.
    -- These registers are cleared when reset (active low) is applied.
    -- Slave register write enable is asserted when valid address and data are
available
    -- and the slave is ready to accept the write address and write data.
    slv_reg_wren <= axi_wready and axs_s0_AXI_WVALID and axi_awready and
axs_s0_AXI_AWVALID ;

    process (clk)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          slv_reg0 <= (others => '0');
          slv_reg1 <= (others => '0');
          slv_reg2 <= (others => '0');
          slv_reg3 <= (others => '0');
        else
          loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
          if (slv_reg_wren = '1') then
            case loc_addr is
              when b"00" =>
                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                  if ( axs_s0_AXI_WSTRB(byte_index) = '1' ) then
                    -- Respective byte enables are asserted as per write strobes
                    -- slave registor 0
                    slv_reg0(byte_index*8+7 downto byte_index*8) <=
axs_s0_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                  end if;
                end loop;
              when b"01" =>
                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                  if ( axs_s0_AXI_WSTRB(byte_index) = '1' ) then
                    -- Respective byte enables are asserted as per write strobes
                    -- slave registor 1
                    slv_reg1(byte_index*8+7 downto byte_index*8) <=
axs_s0_AXI_WDATA(byte_index*8+7 downto byte_index*8);
```

```
                  end if;
                end loop;
              when b"10" =>
                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                  if ( axs_s0_AXI_WSTRB(byte_index) = '1' ) then
                    -- Respective byte enables are asserted as per write strobes
                    -- slave registor 2
                    slv_reg2(byte_index*8+7 downto byte_index*8) <=
axs_s0_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                  end if;
                end loop;
              when b"11" =>
                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                  if ( axs_s0_AXI_WSTRB(byte_index) = '1' ) then
                    -- Respective byte enables are asserted as per write strobes
                    -- slave registor 3
                    slv_reg3(byte_index*8+7 downto byte_index*8) <=
axs_s0_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                  end if;
                end loop;
              when others =>
                slv_reg0 <= slv_reg0;
                slv_reg1 <= slv_reg1;
                slv_reg2 <= slv_reg2;
                slv_reg3 <= slv_reg3;
            end case;
          end if;
        end if;
      end if;
    end process;

    -- Implement write response logic generation
    -- The write response and response valid signals are asserted by the slave
    -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
    -- This marks the acceptance of address and indicates the status of
    -- write transaction.

    process (clk)
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          axi_bvalid  <= '0';
          axi_bresp   <= "00"; --need to work more on the responses
        else
          if (axi_awready = '1' and axs_s0_AXI_AWVALID = '1' and axi_wready = '1'
and axs_s0_AXI_WVALID = '1' and axi_bvalid = '0'  ) then
            axi_bvalid <= '1';
            axi_bresp  <= "00";
          elsif (axs_s0_AXI_BREADY = '1' and axi_bvalid = '1') then   --check if
bready is asserted while bvalid is high)
```

```
            axi_bvalid <= '0';                                  -- (there is a
possibility that bready is always asserted high)
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_arready generation
    -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
    -- S_AXI_ARVALID is asserted. axi_awready is
    -- de-asserted when reset (active low) is asserted.
    -- The read address is also latched when S_AXI_ARVALID is
    -- asserted. axi_araddr is reset to zero on reset assertion.

    process (clk)
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          axi_arready <= '0';
          axi_araddr  <= (others => '1');
        else
          if (axi_arready = '0' and axs_s0_AXI_ARVALID = '1') then
            -- indicates that the slave has acceped the valid read address
            axi_arready <= '1';
            -- Read Address latching
            axi_araddr  <= axs_s0_AXI_ARADDR;
          else
            axi_arready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_arvalid generation
    -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
    -- S_AXI_ARVALID and axi_arready are asserted. The slave registers
    -- data are available on the axi_rdata bus at this instance. The
    -- assertion of axi_rvalid marks the validity of read data on the
    -- bus and axi_rresp indicates the status of read transaction.axi_rvalid
    -- is deasserted on reset (active low). axi_rresp and axi_rdata are
    -- cleared to zero on reset (active low).
    process (clk)
    begin
      if rising_edge(clk) then
        if reset_n = '0' then
          axi_rvalid <= '0';
          axi_rresp  <= "00";
        else
          if (axi_arready = '1' and axs_s0_AXI_ARVALID = '1' and axi_rvalid = '0')
then
            -- Valid read data is available at the read data bus
```

```
            axi_rvalid <= '1';
            axi_rresp  <= "00"; -- 'OKAY' response
          elsif (axi_rvalid = '1' and axs_s0_AXI_RREADY = '1') then
            -- Read data is accepted by the master
            axi_rvalid <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement memory mapped register select and read logic generation
    -- Slave register read enable is asserted when valid address is available
    -- and the slave is ready to accept the read address.
    slv_reg_rden <= axi_arready and axs_s0_AXI_ARVALID and (not axi_rvalid) ;
```

```
    process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, reset_n,
slv_reg_rden, result, busy)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
        -- Address decoding for reading registers
        loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
        case loc_addr is
          when b"00" =>
            reg_data_out <= slv_reg0;
          when b"01" =>
            reg_data_out <= slv_reg1;
          when b"10" =>
            reg_data_out <= result; -- assign result signal to the corresponding
register
          when b"11" =>
            -- assign busy signal to the correct register position
            reg_data_out <= X"0000000" & slv_reg3(3) & slv_reg3(2) & busy &
slv_reg3(0);
          when others =>
            reg_data_out  <= (others => '0');
        end case;
    end process;
```

```
    -- Output register or memory read data
    process( clk ) is
    begin
      if (rising_edge (clk)) then
        if ( reset_n = '0' ) then
          axi_rdata  <= (others => '0');
        else
          if (slv_reg_rden = '1') then
            -- When there is a valid read address (S_AXI_ARVALID) with
            -- acceptance of read address by the slave (axi_arready),
            -- output the read dada
            -- Read address mux
              axi_rdata <= reg_data_out;     -- register read data
```

```
          end if;
        end if;
      end if;
    end process;


    -- Add user logic here

    -- Instantiation of the instance which constitutes the functional part of the IP. We map inputs and outputs of the booth multiplier to the AXI4 interface. Inputs start, ack, data_a and data_b are mapped to their corresponding slave register position, the rest of ports of the booth are connected to global inputs and outputs.

    booth_instance: booth
    port map(
            clk     => clk,
            resetn      => reset_n,
            busy        => busy,
            irq     => irq,
            start       => slv_reg3(0),
            ack     => slv_reg3(2),
            irq_enable  => slv_reg3(3),
            data_a      => slv_reg0(15 downto 0),
            data_b      => slv_reg1(15 downto 0),
            result      => result
        );

    -- User logic ends

end arch_imp;
```

Once the AXI4 interface has been changed, a new IP is created using the Quartus platform designer interface. It is important to include the IP corresponding to the Booth multiplier and connecting it properly to the system. When creating this IP, we have set the interface for the output signal *irq* as Interrupt Sender.

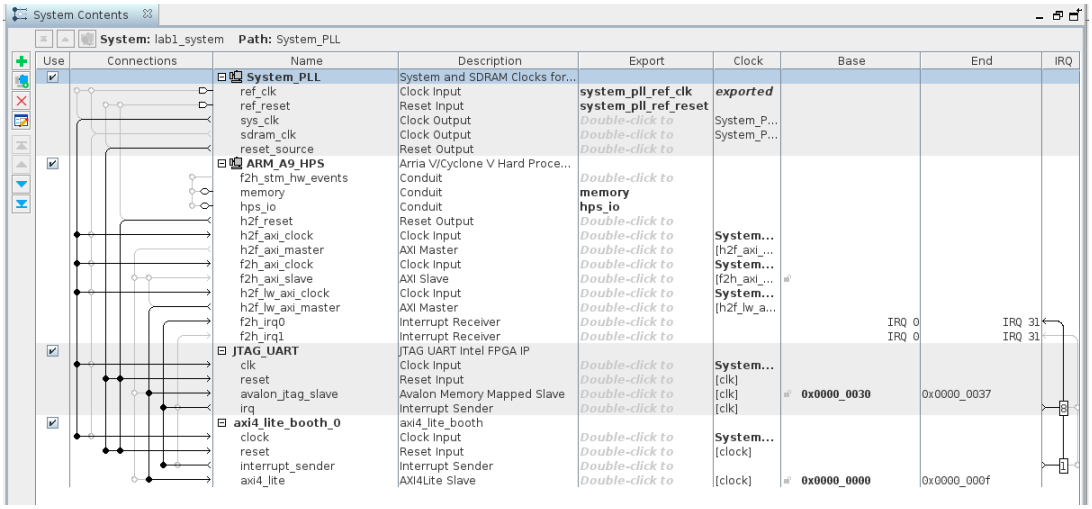


Figure 2. Hardware platform IP.

For the hardware design, two files have been included regarding the resource utilization and the timing constraints of the design.

1.1. Resource Utilization Report

```
Analysis & Synthesis Status : Successful - Wed Nov 15 18 : 09 : 56 2023
Quartus Prime Version : 21.1.0 Build 842 10/21/2021 SJ Standard Edition
Revision Name : lab1
Top - level ENTITY Name : lab1
Family : Cyclone V
Logic utilization (IN ALMs) : N/A
Total registers : 942
Total pins : 88
Total virtual pins : 0
Total BLOCK memory bits : 1, 024
Total DSP Blocks : 0
Total HSSI RX PCSs : 0
Total HSSI PMA RX Deserializers : 0
Total HSSI TX PCSs : 0
Total HSSI PMA TX Serializers : 0
Total PLLs : 1
Total DLLs : 1
```

1.2. Timing Report

```
------------------------------------------------------------
Timing Analyzer Summary
------------------------------------------------------------

TYPE : Slow 1100mV 85C Model Setup 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 1.730
TNS : 0.000

TYPE : Slow 1100mV 85C Model Setup 'altera_reserved_tck'
Slack : 10.807
TNS : 0.000

TYPE : Slow 1100mV 85C Model Hold 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.143
TNS : 0.000

TYPE : Slow 1100mV 85C Model Hold 'altera_reserved_tck'
Slack : 0.234
TNS : 0.000
```

```
TYPE : Slow 1100mV 85C Model Recovery 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 3.375
TNS : 0.000

TYPE : Slow 1100mV 85C Model Recovery 'altera_reserved_tck'
Slack : 17.618
TNS : 0.000

TYPE : Slow 1100mV 85C Model Removal 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.470
TNS : 0.000

TYPE : Slow 1100mV 85C Model Removal 'altera_reserved_tck'
Slack : 1.093
TNS : 0.000

TYPE : Slow 1100mV 85C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | pll_write_clk_dq_write_clk'
Slack : 0.608
TNS : 0.000

TYPE : Slow 1100mV 85C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.623
TNS : 0.000

TYPE : Slow 1100mV 85C Model Minimum Pulse Width 'altera_reserved_tck'
Slack : 19.183
TNS : 0.000

TYPE : Slow 1100mV 0C Model Setup 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 1.727
TNS : 0.000

TYPE : Slow 1100mV 0C Model Setup 'altera_reserved_tck'
Slack : 10.883
TNS : 0.000

TYPE : Slow 1100mV 0C Model Hold 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.162
TNS : 0.000

TYPE : Slow 1100mV 0C Model Hold 'altera_reserved_tck'
Slack : 0.232
TNS : 0.000
```

```
TYPE : Slow 1100mV 0C Model Recovery 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 3.468
TNS : 0.000

TYPE : Slow 1100mV 0C Model Recovery 'altera_reserved_tck'
Slack : 17.803
TNS : 0.000

TYPE : Slow 1100mV 0C Model Removal 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.460
TNS : 0.000

TYPE : Slow 1100mV 0C Model Removal 'altera_reserved_tck'
Slack : 1.023
TNS : 0.000

TYPE : Slow 1100mV 0C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | pll_write_clk_dq_write_clk'
Slack : 0.613
TNS : 0.000

TYPE : Slow 1100mV 0C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.630
TNS : 0.000

TYPE : Slow 1100mV 0C Model Minimum Pulse Width 'altera_reserved_tck'
Slack : 19.181
TNS : 0.000

TYPE : Fast 1100mV 85C Model Setup 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 2.110
TNS : 0.000

TYPE : Fast 1100mV 85C Model Setup 'altera_reserved_tck'
Slack : 12.888
TNS : 0.000

TYPE : Fast 1100mV 85C Model Hold 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.084
TNS : 0.000

TYPE : Fast 1100mV 85C Model Hold 'altera_reserved_tck'
Slack : 0.151
```

```
TNS : 0.000

TYPE : Fast 1100mV 85C Model Recovery 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 3.727
TNS : 0.000

TYPE : Fast 1100mV 85C Model Recovery 'altera_reserved_tck'
Slack : 18.748
TNS : 0.000

TYPE : Fast 1100mV 85C Model Removal 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.497
TNS : 0.000

TYPE : Fast 1100mV 85C Model Removal 'altera_reserved_tck'
Slack : 0.547
TNS : 0.000

TYPE : Fast 1100mV 85C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | pll_write_clk_dq_write_clk'
Slack : 0.883
TNS : 0.000

TYPE : Fast 1100mV 85C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.891
TNS : 0.000

TYPE : Fast 1100mV 85C Model Minimum Pulse Width 'altera_reserved_tck'
Slack : 19.159
TNS : 0.000

TYPE : Fast 1100mV 0C Model Setup 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 2.110
TNS : 0.000

TYPE : Fast 1100mV 0C Model Setup 'altera_reserved_tck'
Slack : 13.011
TNS : 0.000

TYPE : Fast 1100mV 0C Model Hold 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.077
TNS : 0.000

TYPE : Fast 1100mV 0C Model Hold 'altera_reserved_tck'
```

```
Slack : 0.132
TNS : 0.000

TYPE : Fast 1100mV 0C Model Recovery 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 3.825
TNS : 0.000

TYPE : Fast 1100mV 0C Model Recovery 'altera_reserved_tck'
Slack : 18.923
TNS : 0.000

TYPE : Fast 1100mV 0C Model Removal 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.473
TNS : 0.000

TYPE : Fast 1100mV 0C Model Removal 'altera_reserved_tck'
Slack : 0.474
TNS : 0.000

TYPE : Fast 1100mV 0C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | pll_write_clk_dq_write_clk'
Slack : 0.887
TNS : 0.000

TYPE : Fast 1100mV 0C Model Minimum Pulse Width 'lab1_system : u0 |
lab1_system_ARM_A9_HPS : arm_a9_hps | lab1_system_ARM_A9_HPS_hps_io :
hps_io | lab1_system_ARM_A9_HPS_hps_io_border : border | hps_sdram :
hps_sdram_inst | hps_sdram_pll : pll | afi_clk_write_clk'
Slack : 0.894
TNS : 0.000

TYPE : Fast 1100mV 0C Model Minimum Pulse Width 'altera_reserved_tck'
Slack : 19.163
TNS : 0.000

------------------------------------------------------------
Info : see the "DDR REPORT" FOR DDR timing results
------------------------------------------------------------
------------------------------------------------------------
```

2. Development of the software application

The sequence of actions to be performed by the HPS are:

1. Enable the interrupts generated by the multiplier by writing a value 1 in bit 3 of *slv_reg3* which corresponds to setting high the input *irq_enable.*

2. Verify that there is not a pending multiplication by checking bit 1 of *slv_reg3.* The system will check if the output *busy* is high. By checking the *busy* signal, we will ensure there is no multiplication process in progress.
3. Once we have verified if the busy signal is 0, we will write the multiplicand and multiplier in the corresponding registers *slv_reg0* and *slv_reg1.*
4. To initiate the multiplication process, we will write a value 1 on bit 0 of *slv_reg3.* This corresponds to the start input signal set in the booth multiplier.
5. We will use a flag called *operationComplete* to wait for the multiplication to complete. This flag will be set by the interrupt routine which is generated by the multiplier. Once the interrupt is detected which is the signal *irq,* the flag will be reset. Then, the *ack* signal will need to be activated and deactivated to reset the *irq* signal and the *busy* signal.
6. Lastly, after printing the result of the multiplication, we set to 0 the *start* signals and we reset all the registers.

The first code which can be seen below is the *main.c* file which contains the general description of the design.

```
#include "../Project_Headers/address_map_arm.h"
#include <stdio.h>
#include "../Project_Headers/address_registers.h"

/* operationComplete is written by an interrupt service routines; we have to
 * declare these as volatile to avoid the compiler caching their values in
 * registers */

volatile int operationComplete = 0;     // interrupt enable signal

/* ************************************************
*****************************
 * This program demonstrates use of interrupts with C code.

 * Initially the value of the switches is represented on the LEDs

 * Each time any of the Pushbuttons is pressed the value of variable key_pressed
 * is changed. When the value of this variabl is 0 the value of the switches
 * is represented on the LEDs. When its value is 1 the value of the switches is
 * represented on the LEDs, but LED9 is ON independently of the value estabiliched
 * in the switches
*********************************************************************************/

int main(void)
{
    /* Different pointers are initialized to the base address of the IP */

    volatile int *slv_reg0 = (int *)REG0_BASE_ADDRESS;
    volatile int *slv_reg1 = (int *)REG1_BASE_ADDRESS;
    volatile int *slv_reg2 = (int *)REG2_BASE_ADDRESS;
    volatile int *slv_reg3 = (int *)REG3_BASE_ADDRESS;
```

```
        set_A9_IRQ_stack();           // initialize the stack pointer for IRQ mode
        config_GIC();                 // configure the general interrupt controller
        enable_A9_interrupts();       // enable interrupts

        /*If the optional part of laboratory 4 is implemented, enable theinterrupts
        generated by the multiplier by writing a value 1 in bit 3 of slv_reg3.*/
        *slv_reg3 = *(slv_reg3) | (1 << 3);     // set irq_enable

        printf("\nProgram starts...\n");

    while (1)
    {
        // verify there is not a pending multiplication by checking the interrupt
// enable signal
        if (operationComplete==1){

            operationComplete=0;
            printf("Result: %i \n", *slv_reg2);        // print the result of the
// multiplication

            *slv_reg3 = *(slv_reg3) & ~(1 << 0);        // set start to 0
            *slv_reg2 = *slv_reg2 & ~(0xFFFFFFFF);      // set result slv_reg2 to 0
            *slv_reg1 = *slv_reg1 & ~(0xFFFFFFFF);      // set result slv_reg1 to 0
            *slv_reg0 = *slv_reg0 & ~(0xFFFFFFFF);      // set result slv_reg0 to 0

        }

/* Verify that there is not a pending multiplication by checking bit 1 of slv_reg3
(busy output of the multiplier) */

        if(((*slv_reg3 >> 1) & 1) == 0) {              // check if busy output is 0

            /* Write the multiplicand on slv_reg0 */
            printf("\nEnter first operand: ");
            scanf("%d", slv_reg0);

            /* Write the multiplier on slv_reg1 */
            printf("Enter second operand: ");
            scanf("%d", slv_reg1);

            /* Start the multiplication by writing a value 1 on bit 0 of slv_reg3
            (start input of the multiplier) */
            *slv_reg3 = *(slv_reg3) | 1;  // set start to 1

            // create a small delay
            for (int i = 0; i < 100; i++) {
            }
        }
    }
}
```

- *Exceptions.c* file

In this code, we will call the function booth_ISR which will allow to enter in the vector table in the interrupt system of the HPS.

```
#include "../Project_Headers/address_map_arm.h"
#include "../Project_Headers/defines.h"
#include "../Project_Headers/interrupt_ID.h"
#include "../Project_Headers/booth_ISR.h"
#include <stdio.h>

/* This file:
 * 1. defines exception vectors for the A9 processor
 * 2. provides code that sets the IRQ mode stack, and that dis/enables
 * interrupts
 * 3. provides code that initializes the generic interrupt controller
*/

// Define the IRQ exception handler
void __attribute__((interrupt)) __cs3_isr_irq(void)
{
    // Read the ICCIAR from the processor interface
    int address = MPCORE_GIC_CPUIF + ICCIAR;
    int int_ID  = *((int *)address);

    printf("We do not have hardware problems to generate the interrupt");
    if (int_ID == KEYS_IRQ) // check if interrupt is from the KEYs
        booth_ISR();
    else
        while (1)
            ; // if unexpected, then stay here

    // Write to the End of Interrupt Register (ICCEOIR)
    address           = MPCORE_GIC_CPUIF + ICCEOIR;
    *((int *)address) = int_ID;

    return;
}

// Define the remaining exception handlers
void __attribute__((interrupt)) __cs3_reset(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_undef(void)
{
    while (1)
        ;
}
```

```
void __attribute__((interrupt)) __cs3_isr_swi(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_pabort(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_dabort(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_fiq(void)
{
    while (1)
        ;
}

/* Initialize the banked stack pointer register for IRQ mode */
void set_A9_IRQ_stack(void)
{
    int stack, mode;
    stack = A9_ONCHIP_END - 7; // top of A9 onchip memory, aligned to 8 bytes
    /* change processor to IRQ mode with interrupts disabled */
    mode = INT_DISABLE | IRQ_MODE;
    asm("msr cpsr, %[ps]" : : [ps] "r"(mode));
    /* set banked stack pointer */
    asm("mov sp, %[ps]" : : [ps] "r"(stack));

    /* go back to SVC mode before executing subroutine return! */
    mode = INT_DISABLE | SVC_MODE;
    asm("msr cpsr, %[ps]" : : [ps] "r"(mode));
}

/* Turn on interrupts in the ARM processor */
void enable_A9_interrupts(void)
{
    int status = SVC_MODE | INT_ENABLE;
    asm("msr cpsr, %[ps]" : : [ps] "r"(status));
}

/* Configure the Generic Interrupt Controller (GIC) */
void config_GIC(void)
{
```

```
    int address; // used to calculate register addresses

    /* configure the KEYs interrupts */

    *((int *)0xFFFED848) = 0x00000100; // Assigns the interrupt channel 73 to CPU0
    *((int *)0xFFFED108) = 0x00000F00; // Enables forwarding of the interrupts
                                       // generated by the KEYs to the CPU interface

    // Set Interrupt Priority Mask Register (ICCPMR). Enable interrupts of all
    // priorities
    address           = MPCORE_GIC_CPUIF + ICCPMR;
    *((int *)address) = 0xFFFF;

    // Set CPU Interface Control Register (ICCICR). Enable signaling of
    // interrupts
    address           = MPCORE_GIC_CPUIF + ICCICR;
    *((int *)address) = ENABLE;

    // Configure the Distributor Control Register (ICDDCR) to send pending
    // interrupts to CPUs
    address           = MPCORE_GIC_DIST + ICDDCR;
    *((int *)address) = ENABLE;
}
```

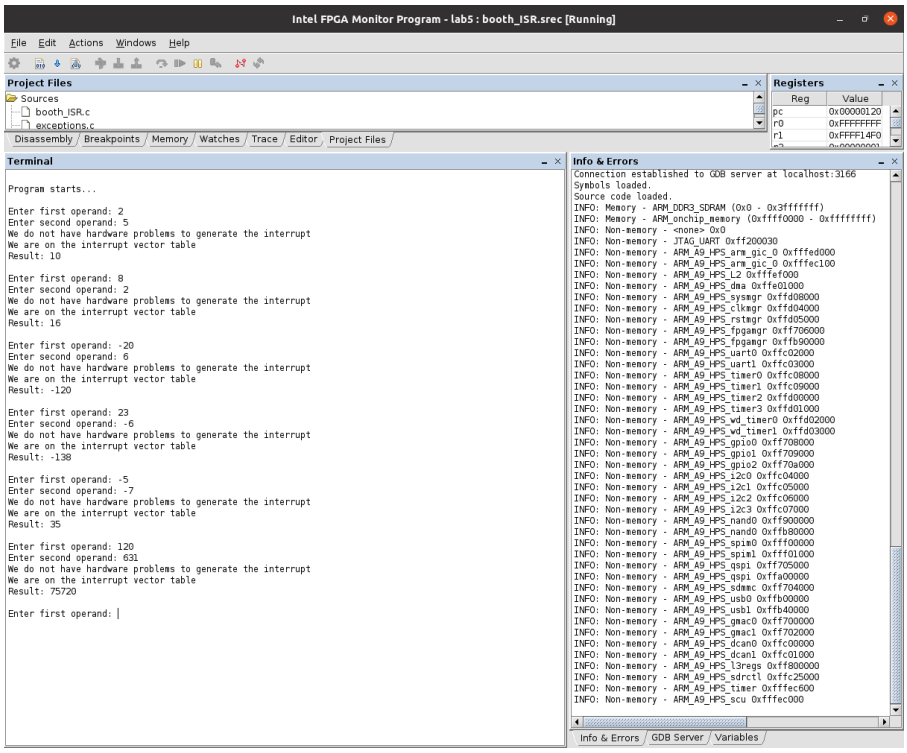
The last C program corresponds to the function *booth_ISR()*. In this function we clear the interrupt by writing a value 1 and then a value 0 on bit 2 of *slv_reg3* (ack input of the multiplier). Then, the flag *operationComplete* is set to let the application program know that an interrupt coming from the multiplier has been received.

```
#include <stdio.h>
#include "../Project_Headers/address_registers.h"
#include "../Project_Headers/address_map_arm.h"
#include "../Project_Headers/exceptions.h"

extern volatile int operationComplete;  // declaration of the flag
 /* Pointer slv_reg3 is initialized to the base address of the IP */
volatile int *slv_reg3 = (int *)0xFF20000C;

/*******************************************************************************
 * Booth - Interrupt Service Routine

 *
 * This routine clears the interrupt and set the operationComplete flag to 1
 *
 * This function should first clear the interrupt by writing a value 1 and then
 * a value 0 on bit 2 of slv_reg3 (ack input of the multiplier). Then, a global
flag should be
```

```
 * set in order to let the application program know that an interrupt coming from
the
 * multiplier has been received.
 *
*******************************************************************************
*****/

void booth_ISR(void)
{
    printf("\nWe are on the interrupt vector table\n");

    /* The ack signal should be activated and then deactivated to reset the irq
signal */
    *slv_reg3 = *(slv_reg3) | (1 << 2);     //  set ack
    *slv_reg3 = *(slv_reg3) & ~(1 << 2);    // clear ack

    /* Set operationComplete flag to 1 */
    operationComplete = 1;
    return;
}
```

3. Verification of the design

In the figure below, we see the verification of the design. We observe how we can enter operands through the terminal, and we obtain the result. Moreover, we have set ***printf*** commands before calling the booth interrupt function and once we have entered in the interrupt vector table. By doing this, we ensure that our design is properly controlled with interrupts.



4. References

[1] Laboratory 5: Implementation of radix-4 Booth multiplier IP with an AXI4 interface, *Universitat Politecnica de Catalunya, High-Level Digital Design*, 2023.

[2] Laboratory 2: Development of custom peripherals and interrupt management, *Universitat Politecnica de Catalunya, High-Level Digital Design*, 2023.