Interfacing Custom IP Cores to HPS/FPGA SOC Platform

COE838: Systems-on-Chip Design
Lab 4

1. Objectives

This lab provides students with a brief tutorial on how to interface IP cores to HPS/FPGA systems. Therefore the knowledge obtained in Lab 3 is a pre-requisite for this lab. Students will learn how to integrate custom IP cores into their design, and create Avalon Memory Map (MM) slaves to interface these cores with HPS/FPGA based SoCs. Students will then apply this knowledge by interfacing IP(s) into a SoC design of their choice.

2. Hardware Design

2.1 Project Setup

Create a Quartus II project called *custom_ip* in your COE838 lab folder, setting up the necessary parameters for a HPS/FPGA system with the Cyclone-V 5CSEMA5F31C6 device. Create a new VHDL file called *custom_ip.vhdl* and set it as your top-level entity. Next we will add an Altera-based IP core to the project (formerly known as MegaCore). For the sake of this lab, we will include a customized multiplier IP (which will be integrated to a HPS/FPGA SoC and interfaced with Avalon MM slaves).

1. To include the multiplier IP core, in Quartus go to "Tools" - "IP Catalog". A window will appear on the right.
2. Expand the field "Library" - "Basic Functions" - "Arithmetic". Double click "LPM_MULT". A window will appear.
3. This window requires you to name your IP core and select the directory to create/save the IP. Name the IP "mult" and ensure that the project directory listed is correct. Under "IP variation file type" select VHDL. Press OK.
4. Another window will appear (may take some time) presenting the MegaWizard Plug-in Manager. Using this wizard, we will specify the parameters of our custom multiplier IP.
    a. [Parameter Settings > General] Page
        i. Under "Multiplier Configuration" ensure "multiply 'dataa' by 'datab' input" is selected.
        ii. Select the datawidth of dataa to be 16 bits.
        iii. Select the datawidth of datab to be 16 bits.
        iv. Ensure that the result datawidth is 32 bits.
        v. Press Next

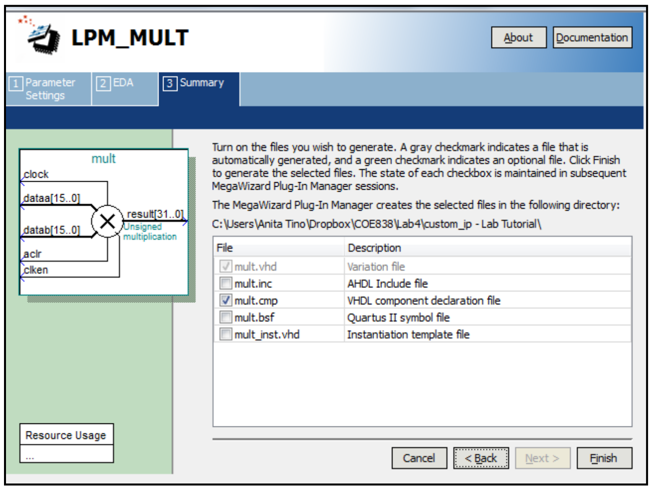
    b. [Parameter Settings > General2] Page
        i. Ensure the three defaulted bullet selections read "No", "Unsigned", and "Use the Default implementation" respectively.

    ii. Press Next.

c. [Parameter Settings > Pipelining] Page
    i. Under the "Pipelining" subsection, select the option "Yes, I want an output latency of **3** clock cycles", writing the number 3 in the blank field.
    ii. Enable "Create an 'aclr' synchronous port".
    iii. Enable "Create a 'clken' clock enable clock".
    iv. Leave optimization as "Default".
    v. Press Next.

d. [EDA] Page
    i. Select Next

e. [Summary] Page
    i. Ensure "mult.vhd" is enabled by default.
    ii. Ensure mult.cmp is also enabled.
    iii. The rest of the options should be disabled. The LPM_MULT component block diagram should resemble that of Fig. 1.
    iv. Press "Finish".



Turn on the files you wish to generate. A gray checkmark indicates a file that is automatically generated, and a green checkmark indicates an optional file. Click Finish to generate the selected files. The state of each checkbox is maintained in subsequent MegaWizard Plug-In Manager sessions.

The MegaWizard Plug-In Manager creates the selected files in the following directory:

C:\Users\Anita Tino\Dropbox\COE838\Lab4\custom_ip - Lab Tutorial\

File	Description
✓ mult.vhd	Variation file
☐ mult.inc	AHDL Include file
✓ mult.cmp	VHDL component declaration file
☐ mult.bsf	Quartus II symbol file
☐ mult_inst.vhd	Instantiation template file

Fig. 1: LPM_MULT Summary

5. The multiplier IP will now be generated in Quartus according to the specifications provided. Include this design in your Quartus project by selecting "Project" - "Add/Remove Files in Project..." and select mult.qip, press Add..., and press OK. You may find the multiplier declaration to instantiate the component in your VHDL design in *mult.vhd*.

2.2 IP Wrapper

The multiplier IP we created contains two inputs, a clock source, an asynchronous reset, and an enable signal. Thus when we create our HPS .c program we will have no issue writing our values and setting up the IP for execution. However, how will we be able to determine when the multiplier has completed computing by simply using our HPS software?

We specified in the MegaCore Wizard that the multiplier should take 3 clock-cycles to execute. Therefore we must create a type of IP "wrapper" which will input all the signals to the multiplier when enabled, and count three clock cycles before outputting a "done" flag to signal to the HPS that the multiplier's product is ready to be read.

The wrapper for the mult IP is located in the course folder at */coe838/labs/lab4/rtl/mult_unit.vhd*. This VHDL file includes the multiplier IP component we just generated called "mult" and port maps it, while creating a wrapper with the functionality described above, i.e. the done signal is pipelined 3 times before outputting the *mult_done* signal. Ensure that your mult IP is identical to the component instantiated in this VHDL file, else make the necessary corrections.

2.3 Avalon MM Slave Interface Design

In the previous lab, we used Altera's built-in QSys compatible *pio* and *SEG7_IF* components which had their own custom Avalon MM slave design embedded into their component. However because we created our own IP from the MegaWizard and would like to include it in our SoC design, we must create our own Avalon MM slave design and incorporate it into our SoC using QSys. This MM slave design allows our IP to communicate on the FPGA fabric and be controlled through the HPS C code we will develop.

As seen in lab 3, memory mapping allows our software to communicate with our hardware components (through the Avalon/AXI bus and lwh2f bridge). However when our IP contains multiple ports, we need to specify these addresses. For example, writing to address 0x0001 + *virtual_base_address* allows us to write data to our mult's dataa port (using the HPS software), writing to address 0x0002 *virtual_base_address* writes to mult's datab port, reading from 0x0000 + *virtual_base_address* reads the product output once complete etc. In lab 3, our pio and SEG7_IF only had one input port to which we wrote data, and was interpretted by the IP to turn on the appropriate LEDs etc. Now our component possess multiple ports and we must be able to handle this communication using bus interfaces.

To make our slave designs concise and simple, we will use the easiest bus interface - the Avalon MM interface. The Avalon MM employs a master-slave protocol with the CPU acting as a master, and the FPGA peripherals acting as slaves. We will create two Avalon MM slaves for the multiplier SoC and classify them into the categories control and data- called *mult_control* and *mult_data* respectively. These VHDL files can be found in the *coe838/labs/lab4/rtl* folder (*mult_control.vhdl* and *mult_data.vhdl*). Copy them to your project directory. Open these files to view the VHDL.

Table I: Common Avalon Bus Signals

Name	Width	Direction	Comments
avs_address	<= 64 bits	Input	Address of slave being accessed
avs_read	1 bit	Input	Read operation requested
avs_write	1 bit	Input	Write operation requested
avs_readdata	8, 16, 32 or 64 bits	Output	Data read from slave
avs_writedata	8, 16, 32 or 64 bits	Input	Data to be written to slave

The contents in the Avalon MM slave VHDL files look like any standard VHDL file, however they contain keywords to properly identified Avalon bus signals in QSys. In particular, the avs_* prefix signifies to QSys that this port communicates with Avalon bus, where other signals usually imply exported conduits, clocks, reset etc. Table I provides common signal names and characteristics used by MM slaves for the Avalon bus.

Let us take *mult_control.vhdl* as an example to obtain an understanding for Avalon MM bus interface design. In terms of control in the multiplier IP, we will need a reset, enable, and done signal for handling control from the HPS software. Therefore our C code will send out the reset and start signals to the multiplier, and read the done signal to determine completion. Looking at *mult_control.vhdl*, we include the mult_* ports so that we may export them as conduits in QSys, and then port map them directly from the HPS to the multiplier to provide this control. The avs_* ports are used to read the control addresses and data from the FPGA → Avalon bus → HPS, and the reverse for writes. Therefore as seen in this VHDL code, if our C application were to *write* to mult_control_base + 0x0000, we would be writing to the IP's start signal, whereas mult_control_base + 0x0001 would write to the reset signal etc. Similarly if we would like to read the status of the multiplier's done signal, we would read from the mult_control_base + 0x0010 address. The other signals are used for debugging and ensuring that the signals are the value we expect (such as reset is unasserted to calculate etc).

The same concept is applied to the *mult_data* bus for reading and writing data. Notice that all data based ports are 32 bits so that we may easily access the bus and conduits with the alt_read_word() and alt_write_word() APIs in the HPS while following the Avalon bus conditions specified in Table I. A graphical representation of the SoC and Avalon MM slaves discussed here is presented in Fig. 2.

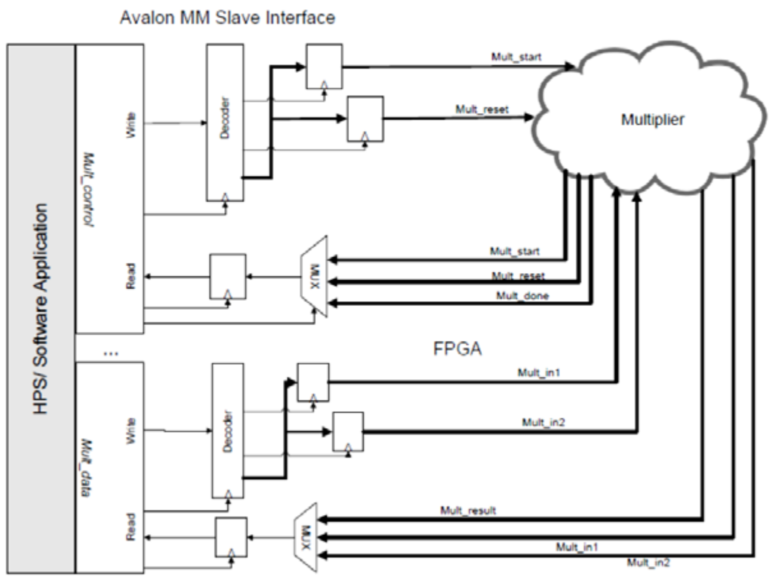


Fig. 2: Multiplier SoC Avalon MM Slave Interface

2.4 Creating the SoC Hardware

2.4.1 QSys Component Creation and SoC Design

Next, we will create a SoC with the Avalon MM slave interfaces using QSys. Copy the soc_system.qsys file from the */coe838/labs/lab3* directory that we used in the previous lab to your project directory. Open QSys and the soc_system.qsys file. We will now learn how to create new QSys components for the Avalon MM slaves we designed and include them in our SoC design.

1. In QSys' left window pane, double-click *New Component...*. This will open a component editor window.

2. [Component Type] Page
    a. In the Name field, highlight new_component and overwrite it with *mult_control.*
    b. Also call the Display name field *mult_control*
3. [Files] Page
    a. Under **Synthesis Files** select the "+" button. A browse window will open. Select *mult_control.vhdl* and press Open.
    b. Click the "Analyze Synthesis Files" button. A status window will open signifying that your VHDL file is being synthesized and analyzed as a component. Once complete, a message will appear stating that QSys has finished analyzing the files. Press Close. You will likely see errors pop up in the Message tab. Ignore these for now.
4. Skip to [Signals] Page
    a. The avs_* keyword signal names should be correctly recognized by QSys and interfaced as "s0" type i.e. an Avalon slave. Clk and reset should also be classified as clock and reset respectively. However the mult_* signals will likely be misinterpreted. We must correct these signals and place them as conduit interfaces.
    b. At mult_start, press its interface name field (i.e. likely called *avalon_slave_0*) . This will open a drop down menu. Select *new Conduit...*. This will rename the interface to *conduit_end.* Under signal type, type in *m_start* (can be given any unique name besides Avalon keyword names).
    c. For mult_reset, open the interface drop down menu and select *conduit_end* so that all the conduits are classified under the same name. Rename it's signal type to a name of your choice i.e. *m_reset*
    d. Repeat step c. for the mult_done signal.
5. [Interface] Page
    a. Click "Remove Interfaces With No Signals". This will remove several messages from the QSys Message window. We will resolve the remaining errors using this page.
    b. Every "Associated Clock" and "Associated Reset" field for all interfaces listed will need to be specified with a clock and reset respectively. Therefore for all the interfaces listed on this page, use the drop down menu to select the clock and/or reset signal listed if "None" is specified as its type. All messages will disappear after this step.
    c. Next, we would like to name our conduits something more meaningful other than *conduit_end*. Therefore scroll to the "conduit_end (Conduit)" interface and type *mult_control* in the Name field. Press Enter to set the change in the Block Diagram displayed.
    d. Double check that all parameters have been specified as described in the steps above. Once you are confident with these parameters, press "Finish".
    e. A window will show prompting you to save the changes to a .tcl file. Press "Yes, Save".

6. You will be brought back to the main QSys page, with a new component listed in the IP catalog under the **Project** heading called *mult_control*. Highlight *mult_control* and press the "+Add..." button. Press Finish to successfully add it to your SoC.

7. Connect the *mult*_control_0.clock port to clk_0.clk in your *Connections*.

8. Connect mult_control_0.reset port to clk_0.reset.

9. Connect mult_control_0.s0 to hps_0.h2f_lw_axi_master.

10. Export the mult_control_0.mult_control conduit. There should now be no errors or warnings in the QSys Message window.
11. Repeat steps 1 - 5 to create a new component for the mult_data Avalon MM interface.

12. Repeat steps 6 - 10 for the mult_data component using the port names you specified during component creation.

13. Resolve any base address overlaps (refer to Lab 3).

14. Copy the HDL example that QSys generated for you and place it in your project's top-level entity file.

15. In QSys, *Generate HDL* to create the VHDL for your project's SoC system.

16. Once successfully generated, use the NIOS II shell to generate.h files for your C application.

17. Exit QSys by selecting Finish. Include the soc_system.qip file in your Quartus project.

2.4.2 Finalizing the Quartus Project

Finally, we need to integrate the SoC design with our multiplier using your *custom_ip* top-level entity file. Use the sample top-level file provided to you in lab 3 (*HEX_LED_FPGA.vhd)* with the sample HDL you copied in QSys to code your top-level VHDL. Omit any reference to HEX and LEDR in the *HEX_LED_FPGA* ENTITY, and rename the ENTITY to *custom_ip*. Port map the start, reset and done conduits from the soc_system to its respective ports in the multiplier. Similarly, port map the in1, in2, and result conduits from the soc_system to the multiplier.

You will find a top-level VHDL sample to reference in the Appendix of this lab if you are having difficulties. Note that the port names may vary depending on what you named your Avalon interface conduits in the mult_control and mult_data slaves (in QSys).

Once you have completed your top-level VHDL design, run the two .tcl scripts for mapping the pin assignments. Compile your design, correct any errors and successfully generate a .sof file to program the FPGA.

3. Software Design

A sample .c application can be found in *coe838/labs/lab4/software*. Copy the main.c file to your project's software directory which contains the *.h files you generated using the NIOS II shell. Open the file to view its contents.

The general template for coding the application is the same as Lab 3. The only difference is that now you have more addresses to work with, and must apply an offset to your API virtual base address depending on what signals/ports you are trying to access when reading (alt_read_word) or writing (alt_write_word) to mult_data and mult_control. You must use the Avalon MM slave VHDL files to obtain the offset that should be applied to the virtual base address.

Note: you may use this main.c file for compiling and generating a binary for the next step. However ensure that the hps_0.h file's component names and base addresses correctly reflect your QSys system components. If they do not match, make the changes needed to the .c file accordingly.

Create a project in DS-5 and compile the .c application to generate a binary. Copy the binary to a USB.

4. HPS/FPGA Execution

Power on the DE1-SoC board. Upload the .sof file to the FPGA using the Quartus II Programmer. Insert the USB to the HPS, mount it using the Linux command, copy the binary to your home directory and execute it. Your terminal output should resemble that of Fig. 3.

```
---------------- Iteration 27 ------------------
Reset done. Deasserting signal
Start successful
waiting for done
conversion done
0x0000001b * 0x0000001c = 0x000002f4. [Expected] 0x000002f4
[SUCCESSFUL]
------------------------------------------------
---------------- Iteration 28 ------------------
Reset done. Deasserting signal
Start successful
waiting for done
conversion done
0x0000001c * 0x0000001d = 0x0000032c. [Expected] 0x0000032c
[SUCCESSFUL]
------------------------------------------------
---------------- Iteration 29 ------------------
Reset done. Deasserting signal
Start successful
waiting for done
conversion done
0x0000001d * 0x0000001e = 0x00000366. [Expected] 0x00000366
[SUCCESSFUL]
------------------------------------------------
---------------- Iteration 30 ------------------
Reset done. Deasserting signal
Start successful
waiting for done
conversion done
0x0000001e * 0x0000001f = 0x000003a2. [Expected] 0x000003a2
[SUCCESSFUL]
------------------------------------------------
[TEST PASSED] 30/30
```

Fig. 3: Sample output from Multiplier SoC

5. What To Hand In

Using the interfacing methodology introduced in this lab, students are to create a HPS/FPGA SoC integrating an IP core of their choice. This core may be selected from the IP Catalog (MegaCore), designed as a custom IP, or retrieved from other IP sources (OpenCores, etc). Marks will be awarded based on creativity and complexity of the design.

This lab is due in week 8/9. When submitting your lab, please include:

- All VHDL files used in your design including the top-level file and all Avalon MM Slave(s) VHDL for interfacing your hardware design. Include any IP VHDL that you believe is relevant to the design (i.e. wrapper, declaration etc).
- A snapshot of your QSys system
- All .c files used for your HPS application
- The hps_0.h file generated by the NIOS II Shell.
- Output of your SoC execution in the HPS terminal.

The lab must be handed in with the University cover page, dated and signed. Your lab instructor will also ask question and you need to demo your work during submission. Be prepared.

Appendix

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

ENTITY custom_ip IS
        PORT( CLOCK_50, HPS_DDR3_RZQ,HPS_ENET_RX_CLK, HPS_ENET_RX_DV : IN STD_LOGIC;
              HPS_DDR3_ADDR                       : OUT STD_LOGIC_VECTOR(14 DOWNTO 0);
              HPS_DDR3_BA                         : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
              HPS_DDR3_CS_N                               : OUT STD_LOGIC;
              HPS_DDR3_CK_P, HPS_DDR3_CK_N, HPS_DDR3_CKE    : OUT STD_LOGIC;
              HPS_USB_DIR, HPS_USB_NXT, HPS_USB_CLKOUT              : IN STD_LOGIC;
              HPS_ENET_RX_DATA                    : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
              HPS_SD_DATA, HPS_DDR3_DQS_N         : INOUT STD_LOGIC_VECTOR(3 DOWNTO 0);
              HPS_DDR3_DQS_P                      : INOUT STD_LOGIC_VECTOR(3 DOWNTO 0);
              HPS_ENET_MDIO                               : INOUT STD_LOGIC;
              HPS_USB_DATA                                : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
              HPS_DDR3_DQ                         : INOUT STD_LOGIC_VECTOR(31 DOWNTO 0);
              HPS_SD_CMD                          : INOUT STD_LOGIC;
              HPS_ENET_TX_DATA, HPS_DDR3_DM       : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
              HPS_DDR3_ODT, HPS_DDR3_RAS_N, HPS_DDR3_RESET_N        : OUT STD_LOGIC;
              HPS_DDR3_CAS_N, HPS_DDR3_WE_N                                 : OUT STD_LOGIC;
              HPS_ENET_MDC, HPS_ENET_TX_EN                                  : OUT STD_LOGIC;
              HPS_USB_STP, HPS_SD_CLK, HPS_ENET_GTX_CLK                  : OUT STD_LOGIC);
END ENTITY custom_ip;


ARCHITECTURE Behaviour of custom_ip IS
    component soc_system is
      port(clk_clk                          : in    std_logic                     := 'X';
           hps_0_h2f_reset_reset_n           : out   std_logic;
           hps_io_hps_io_emac1_inst_TX_CLK    : out   std_logic;
            hps_io_hps_io_emac1_inst_TXD0      : out   std_logic;
            hps_io_hps_io_emac1_inst_TXD1      : out   std_logic;
            hps_io_hps_io_emac1_inst_TXD2      : out   std_logic;
            hps_io_hps_io_emac1_inst_TXD3      : out   std_logic;
            hps_io_hps_io_emac1_inst_RXD0      : in    std_logic                     := 'X';
            hps_io_hps_io_emac1_inst_MDIO      : inout std_logic                     := 'X';
            hps_io_hps_io_emac1_inst_MDC       : out   std_logic;
            hps_io_hps_io_emac1_inst_RX_CTL    : in    std_logic                     := 'X';
            hps_io_hps_io_emac1_inst_TX_CTL    : out   std_logic;
            hps_io_hps_io_emac1_inst_RX_CLK    : in    std_logic                     := 'X';
            hps_io_hps_io_emac1_inst_RXD1      : in    std_logic                     := 'X';
            hps_io_hps_io_emac1_inst_RXD2      : in    std_logic                     := 'X';
            hps_io_hps_io_emac1_inst_RXD3      : in    std_logic                     := 'X';
            hps_io_hps_io_sdio_inst_CMD        : inout std_logic                     := 'X';
            hps_io_hps_io_sdio_inst_D0         : inout std_logic                     := 'X';
            hps_io_hps_io_sdio_inst_D1         : inout std_logic                     := 'X';
            hps_io_hps_io_sdio_inst_CLK        : out   std_logic;
            hps_io_hps_io_sdio_inst_D2         : inout std_logic                     := 'X';
            hps_io_hps_io_sdio_inst_D3         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D0         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D1         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D2         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D3         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D4         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D5         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D6         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_D7         : inout std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_CLK        : in    std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_STP        : out   std_logic;
            hps_io_hps_io_usb1_inst_DIR        : in    std_logic                     := 'X';
            hps_io_hps_io_usb1_inst_NXT        : in    std_logic                     := 'X';
            memory_mem_a                       : out   std_logic_vector(14 downto 0);
            memory_mem_ba                      : out   std_logic_vector(2 downto 0);
            memory_mem_ck                      : out   std_logic;
            memory_mem_ck_n                    : out   std_logic;
            memory_mem_cke                     : out   std_logic;
            memory_mem_cs_n                    : out   std_logic;
            memory_mem_ras_n                   : out   std_logic;
            memory_mem_cas_n                   : out   std_logic;
            memory_mem_we_n                    : out   std_logic;
            memory_mem_reset_n                 : out   std_logic;
            memory_mem_dq                      : inout std_logic_vector(31 downto 0) := (others => 'X');
            memory_mem_dqs                     : inout std_logic_vector(3 downto 0)  := (others => 'X');
           memory_mem_dqs_n                   : inout std_logic_vector(3 downto 0)  := (others => 'X');
            memory_mem_odt                     : out   std_logic;
            memory_mem_dm                      : out   std_logic_vector(3 downto 0);
           memory_oct_rzqin                   : in    std_logic                     := 'X';
           reset_reset_n                       : in    std_logic                     := 'X';
           mult_control_0_mult_control_m_start : out   std_logic_vector(31 downto 0);
           mult_control_0_mult_control_m_reset : out   std_logic_vector(31 downto 0);
           mult_control_0_mult_control_m_done  : in    std_logic_vector(31 downto 0) := (others => 'X');
           mult_data_0_mult_data_m_in1         : out   std_logic_vector(31 downto 0);
           mult_data_0_mult_data_m_in2         : out   std_logic_vector(31 downto 0);
           mult_data_0_mult_data_m_result      : in    std_logic_vector(31 downto 0) := (others => 'X')
);
end COMPONENT soc_system;


COMPONENT mult_unit
        PORT( clk, reset, enable                   : IN STD_LOGIC;
             mult_a, mult_b                        : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
             mult_done                             : OUT STD_LOGIC;
             mult_result                           : OUT STD_LOGIC_VECTOR(31 DOWNTO 0):= (others => '0'));
END COMPONENT;


SIGNAL reset_reset_n, done : STD_LOGIC ;
SIGNAL mult_input_reset : STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL mult_output_result, mult_input_start : STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL in1, in2 : STD_LOGIC_VECTOR(31 DOWNTO 0);


BEGIN
    u0 : component soc_system
        port map (
            clk_clk                            => CLOCK_50,
            reset_reset_n                      => '1',
            memory_mem_a                       => HPS_DDR3_ADDR,
            memory_mem_ba                      => HPS_DDR3_BA,
            memory_mem_ck                      => HPS_DDR3_CK_P,
            memory_mem_ck_n                    => HPS_DDR3_CK_N,
            memory_mem_cke                     => HPS_DDR3_CKE,
            memory_mem_cs_n                    => HPS_DDR3_CS_N,
            memory_mem_ras_n                   => HPS_DDR3_RAS_N,
            memory_mem_cas_n                   => HPS_DDR3_CAS_N,
            memory_mem_we_n                    => HPS_DDR3_WE_N,
            memory_mem_reset_n                 => HPS_DDR3_RESET_N,
            memory_mem_dq                      => HPS_DDR3_DQ,
            memory_mem_dqs                     => HPS_DDR3_DQS_P,
            memory_mem_dqs_n                   => HPS_DDR3_DQS_N,
            memory_mem_odt                     => HPS_DDR3_ODT,
            memory_mem_dm                      => HPS_DDR3_DM,
            memory_oct_rzqin                   => HPS_DDR3_RZQ,
            hps_io_hps_io_emac1_inst_TX_CLK    => HPS_ENET_GTX_CLK,
            hps_io_hps_io_emac1_inst_TXD0      => HPS_ENET_TX_DATA(0),
            hps_io_hps_io_emac1_inst_TXD1      => HPS_ENET_TX_DATA(1),
            hps_io_hps_io_emac1_inst_TXD2      => HPS_ENET_TX_DATA(2),
            hps_io_hps_io_emac1_inst_TXD3      => HPS_ENET_TX_DATA(3),
            hps_io_hps_io_emac1_inst_RXD0      => HPS_ENET_RX_DATA(0),
            hps_io_hps_io_emac1_inst_MDIO      => HPS_ENET_MDIO,
            hps_io_hps_io_emac1_inst_MDC       => HPS_ENET_MDC,
            hps_io_hps_io_emac1_inst_RX_CTL    => HPS_ENET_RX_DV,
            hps_io_hps_io_emac1_inst_TX_CTL    => HPS_ENET_TX_EN,
            hps_io_hps_io_emac1_inst_RX_CLK    => HPS_ENET_RX_CLK,
            hps_io_hps_io_emac1_inst_RXD1      => HPS_ENET_RX_DATA(1),
            hps_io_hps_io_emac1_inst_RXD2      => HPS_ENET_RX_DATA(2),
            hps_io_hps_io_emac1_inst_RXD3      => HPS_ENET_RX_DATA(3),
            hps_io_hps_io_sdio_inst_CMD        => HPS_SD_CMD,
            hps_io_hps_io_sdio_inst_D0         => HPS_SD_DATA(0),
            hps_io_hps_io_sdio_inst_D1         => HPS_SD_DATA(1),
            hps_io_hps_io_sdio_inst_CLK        => HPS_SD_CLK,
            hps_io_hps_io_sdio_inst_D2         => HPS_SD_DATA(2),
            hps_io_hps_io_sdio_inst_D3         => HPS_SD_DATA(3),
            hps_io_hps_io_usb1_inst_D0         => HPS_USB_DATA(0),
            hps_io_hps_io_usb1_inst_D1         => HPS_USB_DATA(1),
            hps_io_hps_io_usb1_inst_D2         => HPS_USB_DATA(2),
            hps_io_hps_io_usb1_inst_D3         => HPS_USB_DATA(3),
            hps_io_hps_io_usb1_inst_D4         => HPS_USB_DATA(4),
            hps_io_hps_io_usb1_inst_D5         => HPS_USB_DATA(5),
            hps_io_hps_io_usb1_inst_D6         => HPS_USB_DATA(6),
            hps_io_hps_io_usb1_inst_D7         => HPS_USB_DATA(7),
            hps_io_hps_io_usb1_inst_CLK        => HPS_USB_CLKOUT,
            hps_io_hps_io_usb1_inst_STP        => HPS_USB_STP,
            hps_io_hps_io_usb1_inst_DIR        => HPS_USB_DIR,
            hps_io_hps_io_usb1_inst_NXT        => HPS_USB_NXT,
           hps_0_h2f_reset_reset_n            => reset_reset_n,
            mult_data_0_mult_data_m_result     => mult_output_result,
           mult_control_0_mult_control_m_done => "0000000000000000000000000000000" & done,
            mult_data_0_mult_data_m_in1        => in1,
            mult_data_0_mult_data_m_in2       => in2,
            mult_control_0_mult_control_m_start => mult_input_start,
            mult_control_0_mult_control_m_reset => mult_input_reset
        );

                 m0 : mult_unit
                 PORT MAP( clk => CLOCK_50, reset => mult_input_reset(0), enable => mult_input_start(0),
                           mult_a => in1(15 DOWNTO 0), mult_b => in2(15 DOWNTO 0),
                           mult_done => done, mult_result => mult_output_result);

END Behaviour;
```