DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
2019/2020 Term 2

Tutorial 1.9: Interfacing ARM/HPS to an External IO Component using an IO Bridge

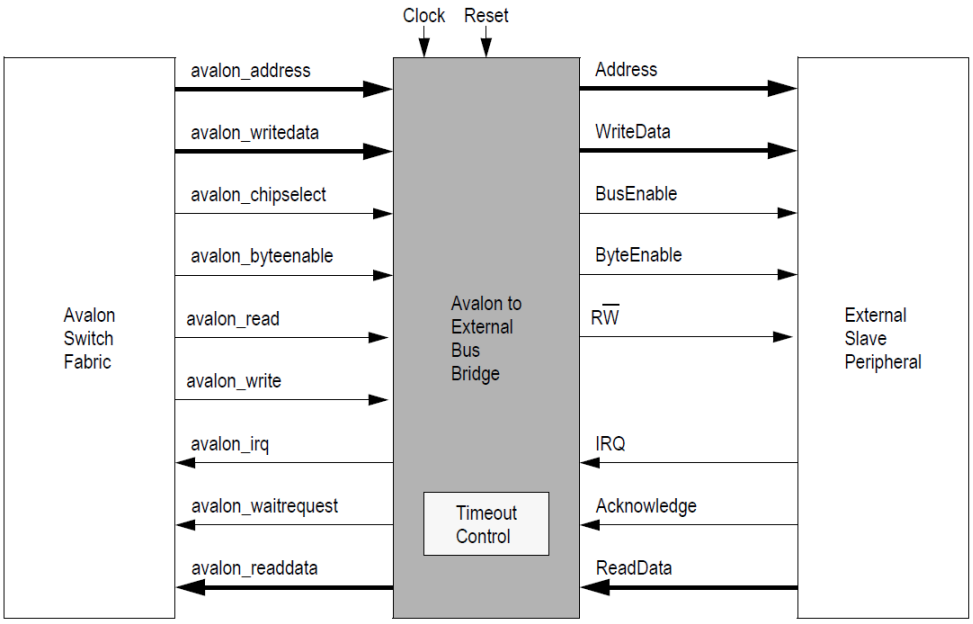
In **tutorial 1.8** we saw how we could create a custom hardware component and interface that to our ARM processor via the Altera "Avalon bus" interface which is suitable for a wide range of hardware components that our processor is expected to control by writing/reading data to memory mapped registers. An alternative approach for more interfacing a range of memory mapped components to the ARM memory space, e.g. IO devices and sub-system interfacing is to use the Avalon to External Bus Bridge.

(read ftp://ftp.altera.com/up/pub/University_Program_IP_Cores/Avalon_to_External_Bus_Bridge.pdf)

This "**bridge**" is itself a component that has been pre-designed with an Avalon interface on one side (just like the bit flipper) and a more general purpose microcomputer interface on the other. It is provided as part of the Altera University program files that you installed as part of Tutorial 1.0 and is very easy to use.

For example, suppose you had downloaded some intellectual property (IP) from say **OpenCores.org** (there's some great computer hardware designs that have been posted in VHDL and Verilog there that we will use in this course). Most of these designs are expected to interface to a microcomputer and act as memory mapped peripherals with registers that the processor can read/write to to control its operation and communication with the outside/real work. This is where a Bridge circuit comes in. It acts as a "***go between***" or "**middle man**" translating microcomputer slave or IO device interface signals, protocols and timing on one side into Avalon compatible signals, protocols and timing on the other, i.e. a bridge between ARM processor on one side and external microcomputer IO devices on the other.

One useful feature this bridge provides is to permit external interface chips to define their own timing via "**wait states**". In simple terms, this means that an interface chip can tell the ARM processor, via the bridge, when it is actually completed a data transfer (i.e. read or write operation) via a handshake signal, rather than have that timing dictated at compile time. This means the speed of ARM data transfers to a device can be changed to match that of the device it is communicating with.



Bridge to External Peripheral chip Signals

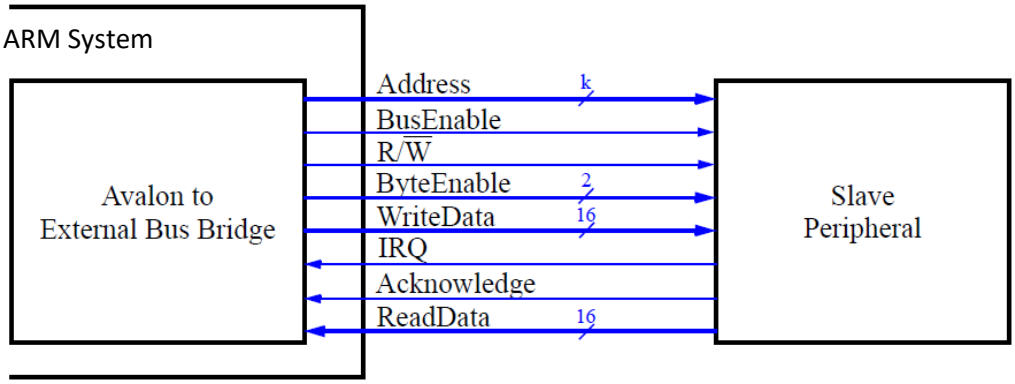
When you instantiate an instance of the Bridge inside QSys, it will generate a bunch of signals that need to be wired up to ARM/HPS fabric on one side (i.e. connect to CPU clock, reset, ARM address and data buses etc.) This is pretty straightforward and mirrors what we did in previous tutorials, e.g. adding an instance of the PIO to connect to Leds and switches (see tutorial 1.2).

Qsys also creates a bunch of signals for the bridge that need to be exported as **output** signals, so that we can connect to an IO device that we add later. These exported signals are described below

- Address - *k* bits (k = 1 to 32). The address of the data to be transferred. The address is aligned to the data size. For 32-bit data, the address bits *Address*1−0 are equal to 0. For 16 bit data transfers, Address 0 is equal to 0. For 8 bit data transfers all address lines are used to specify the address.
- BusEnable - 1 bit. This signal is driven high when the bridge detects that the ARM CPU is accessing the External Slave device. It Indicates that all other signals to the external slave are valid, and a data transfer should occur.
- RW - 1 bit. Indicates whether the data transfer is a Read (1) or a Write (0) operation. A read means the External slave devices supplies data back to ARM via the data bus and bridge, a write means it should store the data that ARM is presenting on the data bus (via the bridge)
- ByteEnable - 16, 8, 4, 2 or 1 bits. These indicate whether or not the corresponding byte should be read or written or the read or write data bus. These signals are active high. For example a bridge that has a 16 bit data bus, would have two Byte Enable signals to indicate which halves (bytes) of the 16 bit data bus are being used during the transfer. ByteEnable[0] when asserted would indicate data is being transferred over data bit D7-D0. Similarly ByteEnable[1] would indicate that data is being transferred via data bits D18-D8. A bridge with a 32 bit data bus would have 4 Byte Enable signals.
- WriteData - 128, 64, 32, 16 or 8 bits. The data to be written to the External Slave device device during a Write transfer.
- ReadData - 128, 64, 32, 16 or 8 bits. The data that is read from the External Slave device during a Read transfer.
- Acknowledge - 1 bit - active high. Used by the peripheral device to indicate that it has completed the data transfer. This signal must be driven during a data transfer or the ARM processor will either "wait" until one is supplied, or timeout leading to incorrectly transferred data. Delaying the acknowledge for a number of clock cycles (e.g. using a timer or shift register triggered when BusEnable is asserted) is a way that we can slow down data transfers to/from slower devices. For fast devices, we could simply connect Acknowledge to BusEnable so that there is no delay.
- IRQ—1 bit - Active **HIGH**. Can be used by an External Slave device to interrupt the ARM processor via the bridge.

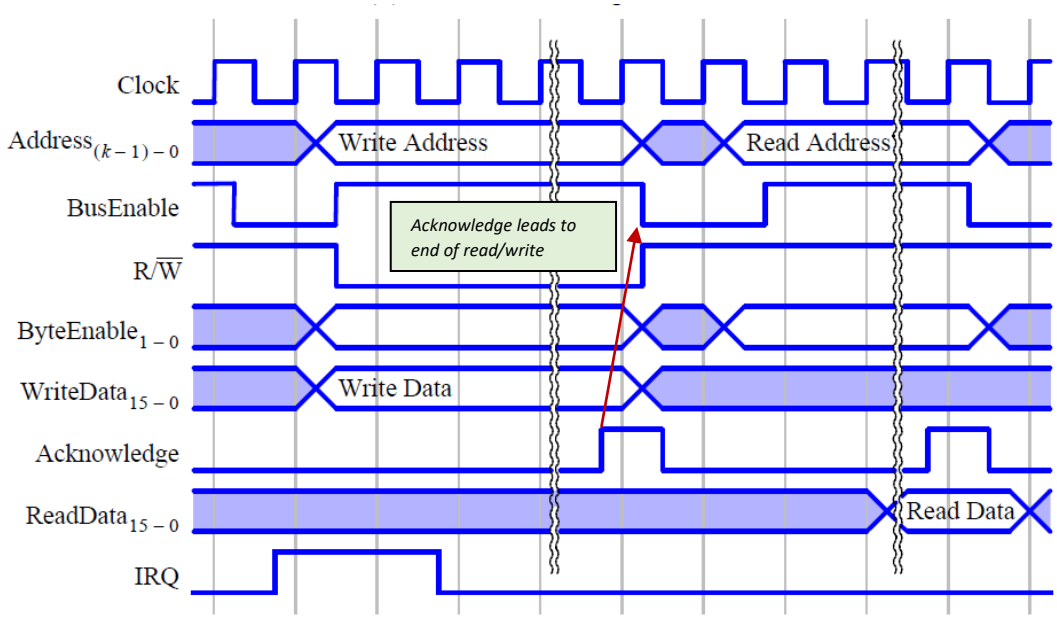
Interfacing an External Slave device/peripheral to an Avalon to External IO Bridge

The illustration below shows the signals **exported** by the bridge that can be used to control an external IO slave device or peripheral.



(*a*) External Bus Signals

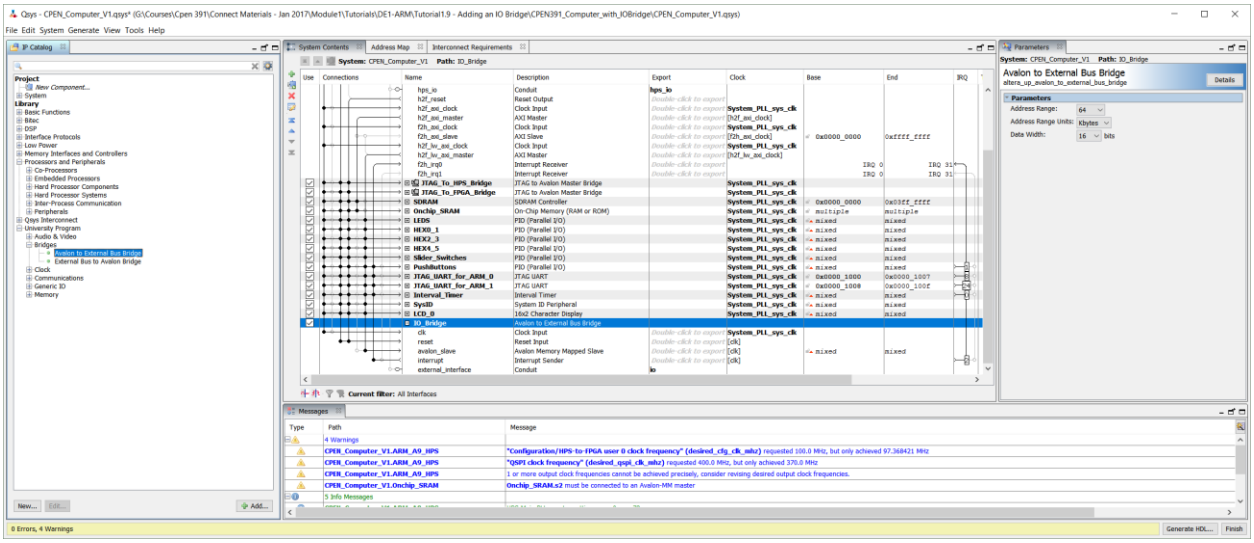
The illustration below shows the timing diagram for a bridge designed to utilise **16 bit** data transfers to a slave. Remember, the bridge asserts these signals when the CPU (ARM) accesses memory locations **reserved by or allocated to the bridge in QSys**. Notice how the address, data, R/W*, Byte enable signals are asserted followed immediately by the **BusEnable**. These signals remain asserted until an **Acknowledge** signal is generated from the Slave. During a write, the data is presented immediately to the Slave. During a Read, the slave presents its data some time later. Either way, the Slave ***must*** assert Acknowledge, which will then terminate the data transfer other the processor will hang forever when it access the bridge. ***IMPORTANT****: Wed could connect* ***Acknowledge*** *input to the* ***BusEnable*** *output so that an acknowledge is always generated when the bridge is accessed. This assumes of course that the IO device hanging off the end of the bridge is quick enough and does not need to "extend" the bus cycle.*



(b) External Bus Timing Diagram

Adding an External Bridge to your ARM based System

Open **Qsys** and add an **Avalon to External Bridge** component from the **University Program->Bridges** (*see below*) to your project (**Note** the image below does **not** show the **BitFlipper** design from tutorial 1.8)



Fill in the information in the dialog box related to Address Range, units and data width (see above top right). Make the connections as shown above – **Remember** to export the Bridge interface using the name "**io**" and also **remember** to add the **interrupt** from the bridge to the interrupt receiver **f2h_irq0** as this gives us the option of adding an interrupt from our slave devices back to the ARM processor via the bridge. Don't forget the set the IRQ level to the CPU (it doesn't matter which level, but in this case I have used IRQ level 2)

In the above example we defined a bridge interface with a **16 bit data bus** and an **address range** of **64kbytes**. Using this bridge we could hang a large number of 16 bit wide slave IO devices so that they all exist or respond within a 64k byte (or 32k word) space within the ARM processors 4GByte address range.

Setting the Base Address of the IO Bridge

Switch to the **Address map** tab and set the addresses for the new Bridge as shown below. Our ARM processor can now talk to devices hanging off the bridge by issuing read/write requests to addresses in the range **0xFF210000** – **0xFF21FFFF**. We'll see in later exercises how we can design a slave IO device to occupy and respond to a **subset** of those addresses, so multiple devices can hang off the bridge without conflict.

System Contents | Address Map | Interconnect Requirements

System: CPEN_Computer_V1 **Path:** IO_Bridge

	ARM_A9_HPS.h2f_lw_axi_master	JTAG_To_FPGA_Bridge.master	JTAG_To_HPS_Bridge.master
ARM_A9_HPS.f2h_axi_slave			0x0000 0000 – 0xffff ffff
HEX0_1.s1	0x0000 0030 – 0x0000 003f	0xff20 0030 – 0xff20 003f	
HEX2_3.s1	0x0000 0040 – 0x0000 004f	0xff20 0040 – 0xff20 004f	
HEX4_5.s1	0x0000 0050 – 0x0000 005f	0xff20 0050 – 0xff20 005f	
IO_Bridge.avalon_slave	0x0001 0000 – 0x0001 ffff	0xff21 0000 – 0xff21 ffff	
Interval_Timer.s1	0x0000 2000 – 0x0000 201f	0xff20 2000 – 0xff20 201f	
JTAG_UART_for_ARM_0.avalon_...	0x0000 1000 – 0x0000 1007		
JTAG_UART_for_ARM_1.avalon_...	0x0000 1008 – 0x0000 100f		
LEDS.s1	0x0000 0020 – 0x0000 002f	0xff20 0020 – 0xff20 002f	
Onchip_SRAM.s1		0x0800 0000 – 0x0800 0fff	
Onchip_SRAM.s2			
PushButtons.s1	0x0000 0010 – 0x0000 001f	0xff20 0010 – 0xff20 001f	
SDRAM.s1		0x0000 0000 – 0x03ff ffff	
Slider_Switches.s1	0x0000 0000 – 0x0000 000f	0xff20 0000 – 0xff20 000f	
SysID.control_slave	0x0000 2020 – 0x0000 2027	0xff20 2020 – 0xff20 2027	
LCD_0.avalon_lcd_slave	0x0000 2030 – 0x0000 2031	0xff20 2030 – 0xff20 2031	

Save the design and **generate** the new Verilog system by clicking on the **generate** button as before. You will now have to go and **add** the new signals to the ARM system (*see next section*) and finally recompile the design in Quartus.

Qsys also generated these signals in the **CPEN391_Computer component** file - see below.

```
////////////////////////////////////////////////////////////////////////////////////
// u0 is an instanace of the QSYS generated computer
// map its IO ports as described below
////////////////////////////////////////////////////////////////////////////////////

   CPEN391_Computer u0 (
     .hex0_1_export                   (Temp_hex0_1),                //              hex0_1.export
     .hex2_3_export                   (Temp_hex2_3),                //              hex2_3.export
     .hex4_5_export                   (Temp_hex4_5),                //              hex4_5.export
     .hps_io_hps_io_emac1_inst_TX_CLK (HPS_ENET_GTX_CLK),           //              hps_io.hps_io_emac1_inst_TX_CLK
     .hps_io_hps_io_emac1_inst_TXD0   (HPS_ENET_TX_DATA[0]),        //                    .hps_io_emac1_inst_TXD0
     .hps_io_hps_io_emac1_inst_TXD1   (HPS_ENET_TX_DATA[1]),        //                    .hps_io_emac1_inst_TXD1
     .hps_io_hps_io_emac1_inst_TXD2   (HPS_ENET_TX_DATA[2]),        //                    .hps_io_emac1_inst_TXD2
     .hps_io_hps_io_emac1_inst_TXD3   (HPS_ENET_TX_DATA[3]),        //                    .hps_io_emac1_inst_TXD3
     .hps_io_hps_io_emac1_inst_RXD0   (HPS_ENET_RX_DATA[0]),        //                    .hps_io_emac1_inst_RXD0
     .hps_io_hps_io_emac1_inst_MDIO   (HPS_ENET_MDIO),              //                    .hps_io_emac1_inst_MDIO
     .hps_io_hps_io_emac1_inst_MDC    (HPS_ENET_MDC),               //                    .hps_io_emac1_inst_MDC
     .hps_io_hps_io_emac1_inst_RX_CTL (HPS_ENET_RX_DV),             //                    .hps_io_emac1_inst_RX_CTL
     .hps_io_hps_io_emac1_inst_TX_CTL (HPS_ENET_TX_EN),             //                    .hps_io_emac1_inst_TX_CTL
     .hps_io_hps_io_emac1_inst_RX_CLK (HPS_ENET_RX_CLK),            //                    .hps_io_emac1_inst_RX_CLK
     .hps_io_hps_io_emac1_inst_RXD1   (HPS_ENET_RX_DATA[1]),        //                    .hps_io_emac1_inst_RXD1
     .hps_io_hps_io_emac1_inst_RXD2   (HPS_ENET_RX_DATA[2]),        //                    .hps_io_emac1_inst_RXD2
     .hps_io_hps_io_emac1_inst_RXD3   (HPS_ENET_RX_DATA[3]),        //                    .hps_io_emac1_inst_RXD3
     .hps_io_hps_io_qspi_inst_IO0     (HPS_FLASH_DATA[0]),          //                    .hps_io_qspi_inst_IO0
     .hps_io_hps_io_qspi_inst_IO1     (HPS_FLASH_DATA[1]),          //                    .hps_io_qspi_inst_IO1
     .hps_io_hps_io_qspi_inst_IO2     (HPS_FLASH_DATA[2]),          //                    .hps_io_qspi_inst_IO2
     .hps_io_hps_io_qspi_inst_IO3     (HPS_FLASH_DATA[3]),          //                    .hps_io_qspi_inst_IO3
     .hps_io_hps_io_qspi_inst_SS0     (HPS_FLASH_NCSO),             //                    .hps_io_qspi_inst_SS0
     .hps_io_hps_io_qspi_inst_CLK     (HPS_FLASH_DCLK),             //                    .hps_io_qspi_inst_CLK
     .hps_io_hps_io_sdio_inst_CMD     (HPS_SD_CMD),                 //                    .hps_io_sdio_inst_CMD
     .hps_io_hps_io_sdio_inst_D0      (HPS_SD_DATA[0]),             //                    .hps_io_sdio_inst_D0
     .hps_io_hps_io_sdio_inst_D1      (HPS_SD_DATA[1]),             //                    .hps_io_sdio_inst_D1
     .hps_io_hps_io_sdio_inst_CLK     (HPS_SD_CLK),                 //                    .hps_io_sdio_inst_CLK
     .hps_io_hps_io_sdio_inst_D2      (HPS_SD_DATA[2]),             //                    .hps_io_sdio_inst_D2
     .hps_io_hps_io_sdio_inst_D3      (HPS_SD_DATA[3]),             //                    .hps_io_sdio_inst_D3
     .hps_io_hps_io_usb1_inst_D0      (HPS_USB_DATA[0]),            //                    .hps_io_usb1_inst_D0
     .hps_io_hps_io_usb1_inst_D1      (HPS_USB_DATA[1]),            //                    .hps_io_usb1_inst_D1
     .hps_io_hps_io_usb1_inst_D2      (HPS_USB_DATA[2]),            //                    .hps_io_usb1_inst_D2
     .hps_io_hps_io_usb1_inst_D3      (HPS_USB_DATA[3]),            //                    .hps_io_usb1_inst_D3
     .hps_io_hps_io_usb1_inst_D4      (HPS_USB_DATA[4]),            //                    .hps_io_usb1_inst_D4
     .hps_io_hps_io_usb1_inst_D5      (HPS_USB_DATA[5]),            //                    .hps_io_usb1_inst_D5
     .hps_io_hps_io_usb1_inst_D6      (HPS_USB_DATA[6]),            //                    .hps_io_usb1_inst_D6
     .hps_io_hps_io_usb1_inst_D7      (HPS_USB_DATA[7]),            //                    .hps_io_usb1_inst_D7
     .hps_io_hps_io_usb1_inst_CLK     (HPS_USB_CLKOUT),             //                    .hps_io_usb1_inst_CLK
     .hps_io_hps_io_usb1_inst_STP     (HPS_USB_STP),                //                    .hps_io_usb1_inst_STP
     .hps_io_hps_io_usb1_inst_DIR     (HPS_USB_DIR),                //                    .hps_io_usb1_inst_DIR
     .hps_io_hps_io_usb1_inst_NXT     (HPS_USB_NXT),                //                    .hps_io_usb1_inst_NXT
     .hps_io_hps_io_spim1_inst_CLK    (HPS_SPIM_CLK),               //                    .hps_io_spim1_inst_CLK
     .hps_io_hps_io_spim1_inst_MOSI   (HPS_SPIM_MOSI),              //                    .hps_io_spim1_inst_MOSI
     .hps_io_hps_io_spim1_inst_MISO   (HPS_SPIM_MISO),              //                    .hps_io_spim1_inst_MISO
    .hps_io_hps_io_spim1_inst_SS0     (HPS_SPIM_SS),               //                    .hps_io_spim1_inst_SS0
    .hps_io_hps_io_uart0_inst_RX      (HPS_UART_RX),               //                    .hps_io_uart0_inst_RX
    .hps_io_hps_io_uart0_inst_TX      (HPS_UART_TX),               //                    .hps_io_uart0_inst_TX
    .hps_io_hps_io_i2c0_inst_SDA      (HPS_I2C1_SDAT),             //                    .hps_io_i2c0_inst_SDA
    .hps_io_hps_io_i2c0_inst_SCL      (HPS_I2C1_SCLK),             //                    .hps_io_i2c0_inst_SCL
    .hps_io_hps_io_i2c1_inst_SDA      (HPS_I2C2_SDAT),             //                    .hps_io_i2c1_inst_SDA
    .hps_io_hps_io_i2c1_inst_SCL      (HPS_I2C2_SCLK),             //                    .hps_io_i2c1_inst_SCL
    .hps_io_hps_io_gpio_inst_GPIO09   (HPS_CONV_USB_N),            //                    .hps_io_gpio_inst_GPIO09
    .hps_io_hps_io_gpio_inst_GPIO35   (HPS_ENET_INT_N),            //                    .hps_io_gpio_inst_GPIO35
    .hps_io_hps_io_gpio_inst_GPIO40   (HPS_GPIO[0]),               //                    .hps_io_gpio_inst_GPIO40
    .hps_io_hps_io_gpio_inst_GPIO41   (HPS_GPIO[1]),               //                    .hps_io_gpio_inst_GPIO41
    .hps_io_hps_io_gpio_inst_GPIO48   (HPS_I2C_CONTROL),           //                    .hps_io_gpio_inst_GPIO48
    .hps_io_hps_io_gpio_inst_GPIO53   (HPS_LED),                   //                    .hps_io_gpio_inst_GPIO53
    .hps_io_hps_io_gpio_inst_GPIO54   (HPS_KEY),                   //                    .hps_io_gpio_inst_GPIO54
    .hps_io_hps_io_gpio_inst_GPIO61   (HPS_GSENSOR_INT),           //                    .hps_io_gpio_inst_GPIO61
    // IO Bridge Connections to wires
    .io_acknowledge                   (IO_ACK_WIRE),
    .io_irq                           (IO_IRQ_WIRE),
    .io_address                       (IO_Address_WIRE),
    .io_bus_enable                    (IO_Bus_Enable_WIRE),
    .io_byte_enable                   (IO_Byte_Enable_WIRE),
    .io_rw                            (IO_RW_WIRE),
    .io_write_data                    (IO_Write_Data_WIRE),
    .io_read_data                     (IO_Read_Data_WIRE),
```

These signals are shown connected to wires, so we need to introduce some wires as shown below. You need to **Type these in** as **new wires** for the top level Verilog file shown above.

```
// TEMP Wires  TO CONNECT IO BRIDGE from QSYS generated IOBridge TO SUBSYSTEMS INCLUD

reg IO_ACK_WIRE;                          // reg because driven by always@ block
wire IO_IRQ_WIRE;
wire unsigned [15:0] IO_Address_WIRE;
wire IO_Bus_Enable_WIRE;
wire unsigned [1:0] IO_Byte_Enable_WIRE;
wire IO_RW_WIRE;
wire unsigned [15:0] IO_Write_Data_WIRE;
wire unsigned [15:0] IO_Read_Data_WIRE;
```

Then add this **always@** block near the bottom of the Verilog file to generate the **acknowledge** pulse for the Bridge whenevet the CPU attempts to access the bridge. This code, delays the acknowledge by 1 clock pulse

```
    // process to generate an acknowledge for the IO Bridge 1 clock cycle after bridge IO BUS ENABLE
    always@(posedge CLOCK_50)
    begin
       IO_ACK_WIRE <= IO_Bus_Enable_WIRE;
    end


endmodule
```

Recompile your system. In the exercises that come later, we will add serial ports and a graphics controller to this bridge. This bridge will enable the ARM processor to access the Graphics and Serial Ports via ‘C’ code and pointers to the correct addresses.