DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
2015/2016 Term 2

Tutorial 1.9: Interfacing NIOS to External IO Components using a Bridge

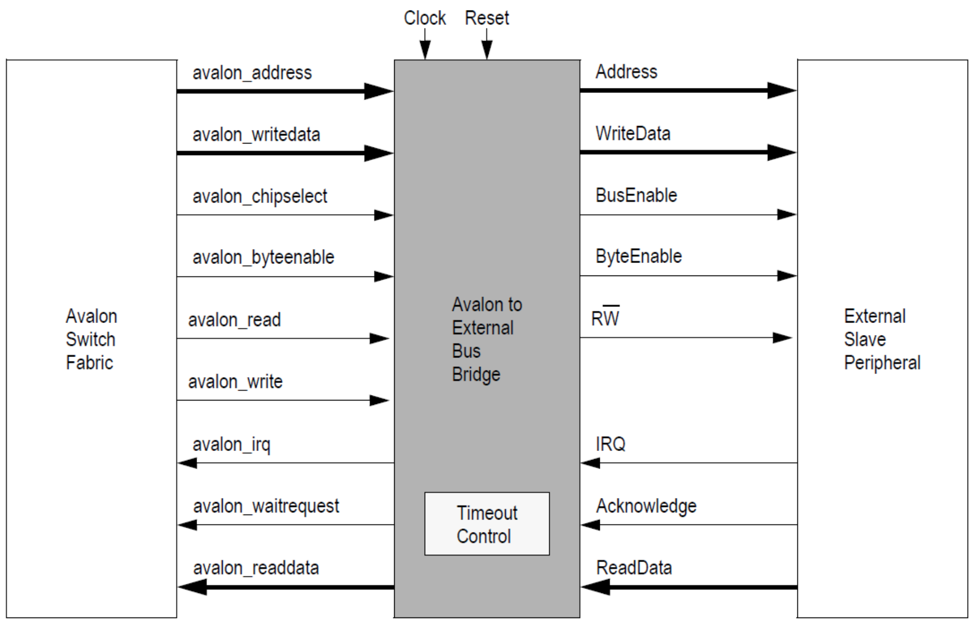
In tutorial 1.8 we saw how we could create custom hardware component and interface that to our NIOS II processor via the Avalon bus interface which is quite complex but suitable for a wide range of hardware. For simple circuits like the bit flipper with a computer interface, this approach is fine since the design and timing parameters are fixed and known at compile time. An alternative and simpler approach for more complex IO and major sub-system interfacing to NIOS 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 circuit 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 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 interfacing 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 design have a common computer interface and we would like to interface it to the NIOS process. This is where the Bridge circuit comes in. It acts as a "go between" or "middle man" translating microcomputer slave or IO peripheral interface signals, protocols and timing on one side into Avalon signals, protocols and timing on the other, i.e. a bridge between NIOS and external microcomputer IO devices
(via the Avalon bus).

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 NIOS via the bridge, when it is performed a data transfer (i.e. read or write operation) via a handshake signal rather than have that timing defined at compile time. This means the speed of NIOS data transfer can be changed to match that of the interface chip that it is communicating with

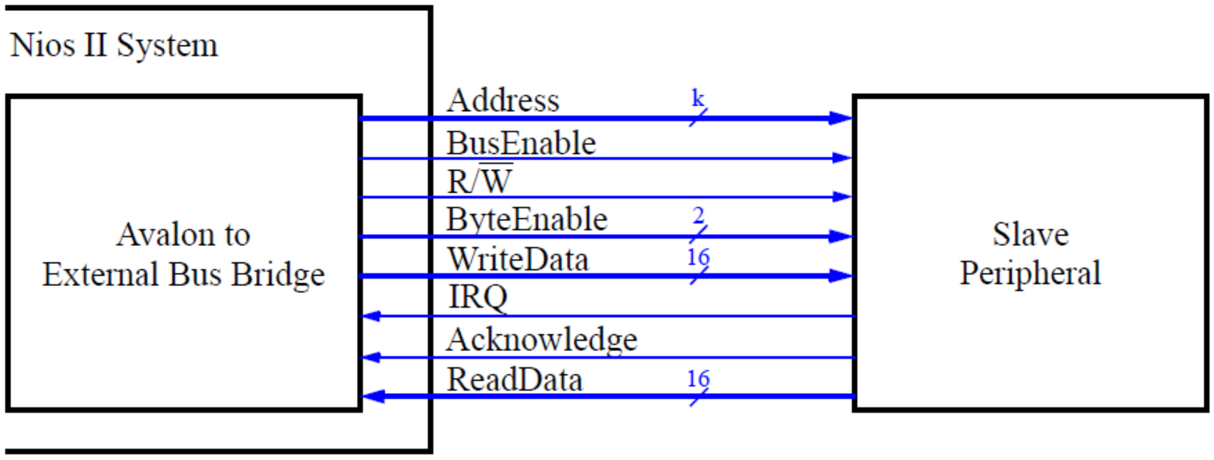


Bridge to External Peripheral chip Signals

- Address - *k* bits (up 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. The byte-enable signals can be used to transfer less than 4 bytes
- BusEnable - 1 bit. This signal is driven high when the bridge detects that the NIOS 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 NIOS via the data bus and bridge, a write means it should store the data that NIOS is presenting on the data bus (via the bridge)
- ByteEnable - 16, 8, 4, 2 or 1 bits. bit indicates 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 NIOS 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 **low**. Can be used by an External Slave device to interrupt the NIOS II processor via the bridge.

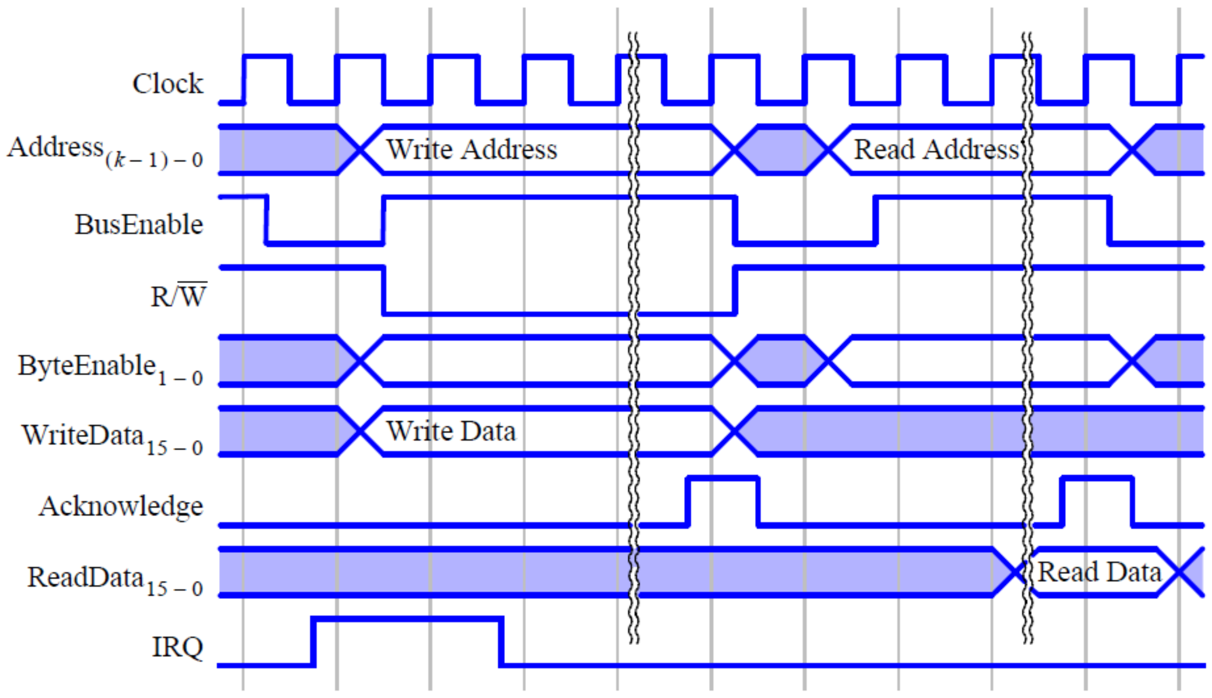
Interfacing an External Slave device/peripheral to an Avalon to External IO Bridge

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

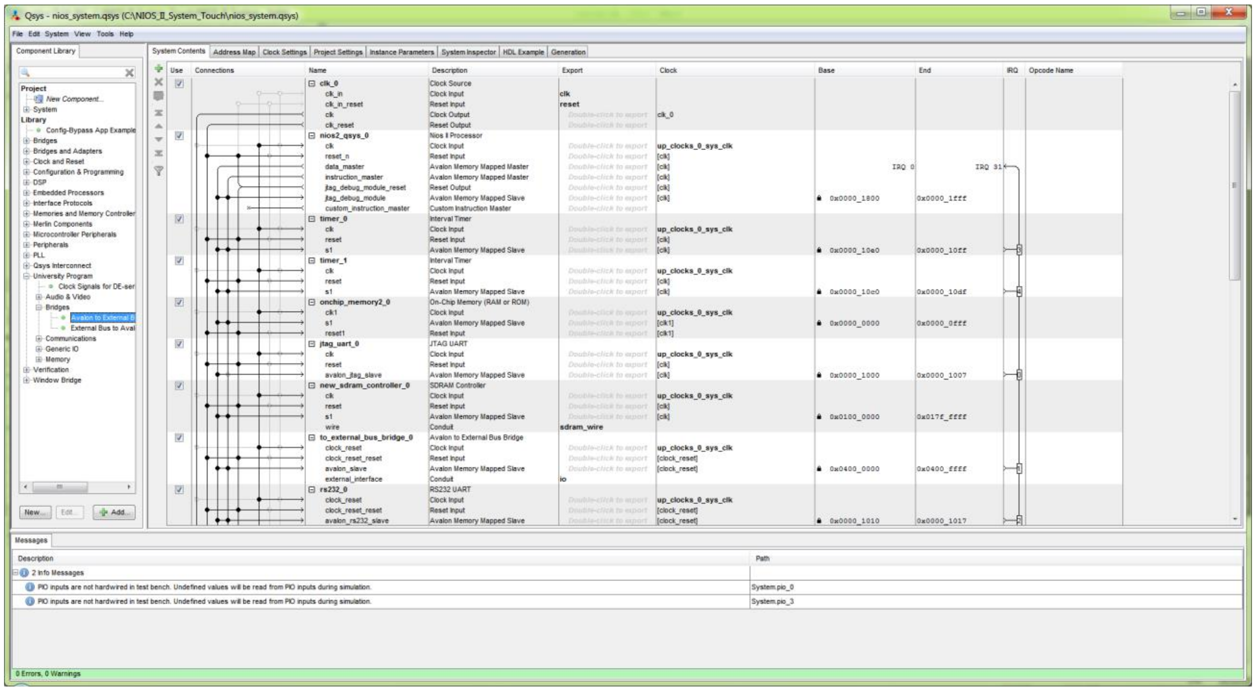
The illustration below shows the timing diagram for **16 bit** data transfers. Remember, the bridge asserts these signals when the CPU accesses memory locations reserved by the bridge. Notice how the address, data, R/W*, Byte enable signals are asserted followed immediately by the BusEnable. These signals remain asserted until an Acknowledge signals 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.



(b) External Bus Timing Diagram

Adding an External Bridge to your NIOS Design

Open Qsys and add an **Avalon to External Bridge** component from the **University Program->Bridges** (see below)

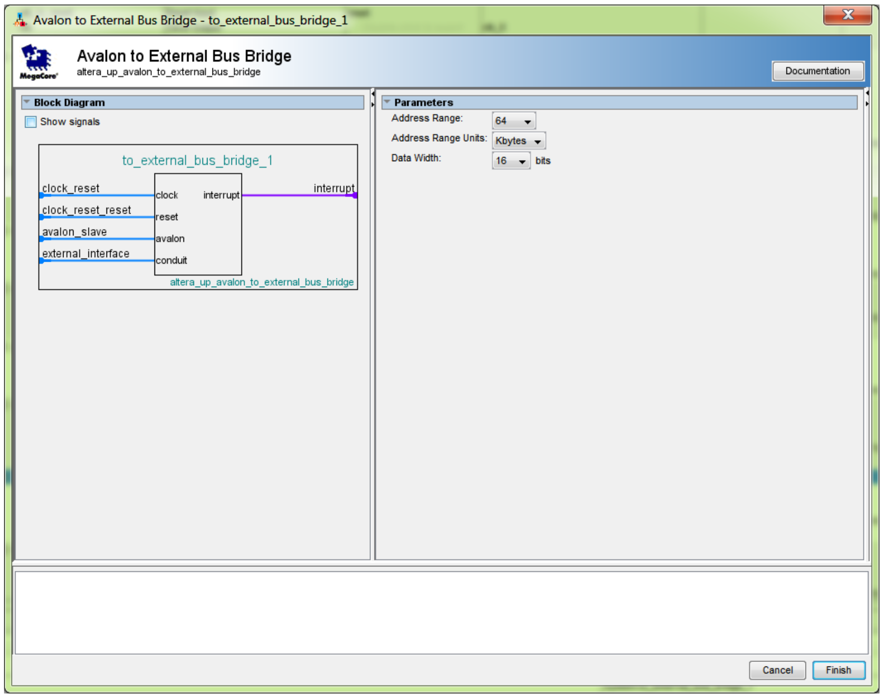


Name	Description	Export	Clock	Base	End	IRQ
clk_0	Clock Source					
clk_in	Clock Input	clk				
clk_in_reset	Reset Input	reset				
clk	Clock Output	Double-click to export	clk_0			
clk_reset	Reset Output	Double-click to export				
nios2_qsys_0	Nios II Processor					
clk	Clock Input	Double-click to export	up_clocks_0_sys_clk			
reset_n	Reset Input	Double-click to export	[clk]			
data_master	Avalon Memory Mapped Master	Double-click to export	[clk]	IRQ 0	IRQ 31	
instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
jtag_debug_module_reset	Reset Output	Double-click to export	[clk]			
jtag_debug_module	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_1800	0x0000_1fff	
custom_instruction_master	Custom Instruction Master	Double-click to export				
timer_0	Interval Timer					
clk	Clock Input	Double-click to export	up_clocks_0_sys_clk			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_10e0	0x0000_10ff	3
timer_1	Interval Timer					
clk	Clock Input	Double-click to export	up_clocks_0_sys_clk			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_10c0	0x0000_10df	4
onchip_memory2_0	On-Chip Memory (RAM or ROM)					
clk1	Clock Input	Double-click to export	up_clocks_0_sys_clk			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	0x0000_0000	0x0000_0fff	
reset1	Reset Input	Double-click to export	[clk1]			
jtag_uart_0	JTAG UART					
clk	Clock Input	Double-click to export	up_clocks_0_sys_clk			
reset	Reset Input	Double-click to export	[clk]			
avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_1000	0x0000_1007	0
new_sdram_controller_0	SDRAM Controller					
clk	Clock Input	Double-click to export	up_clocks_0_sys_clk			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0100_0000	0x017f_ffff	
wire	Conduit	sdram_wire				
to_external_bus_bridge_0	Avalon to External Bus Bridge					
clock_reset	Clock Input	Double-click to export	up_clocks_0_sys_clk			
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]			
avalon_slave	Avalon Memory Mapped Slave	Double-click to export	[clock_reset]	0x0400_0000	0x0400_ffff	1
external_interface	Conduit	io				
rs232_0	RS232 UART					
clock_reset	Clock Input	Double-click to export	up_clocks_0_sys_clk			
clock_reset_reset	Reset Input	Double-click to export	[clock_reset]			
avalon_rs232_slave	Avalon Memory Mapped Slave	Double-click to export	[clock_reset]	0x0000_1010	0x0000_1017	2

Description	Path
2 Info Messages	
PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.	System.pio_0
PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.	System.pio_3

0 Errors, 0 Warnings

This dialog box opens allowing you to define the size of the address space for the computer interface that you are going to hang off the bridge and width of the data bus. It also gives us the option of adding an interrupt from our interface chip back to the NIOS processor

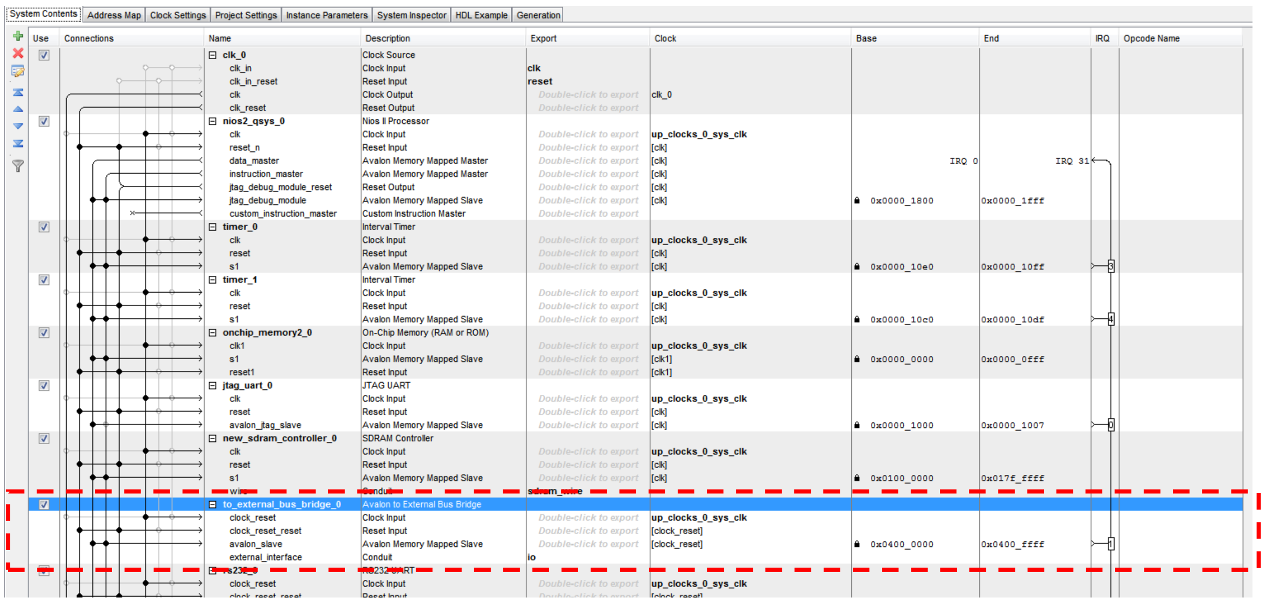


Address Range: 64
Address Range Units: Kbytes
Data Width: 16 bits

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

Setting the Base Address of the IO Interface

Locate the new bridge in the Qsys window (*see below*) make the connections as before to the *clock*, *reset* and *Avalon interface*, set the base address of the 64k IO interface. In this case it starts at hex **04000000** and ends at hex **0400FFFF**. The address range must not conflict with any other components in the system or Qsys will flag it with an error. You can also define the interrupt level used by the IO interface back to NIOS. In this case interrupt level 1



System Contents | Address Map | Clock Settings | Project Settings | Instance Parameters | System Inspector | HDL Example | Generation

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Opcode Name
✓		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**					
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to export*	clk_0				
		clk_reset	Reset Output	*Double-click to export*					
✓		**nios2_qsys_0**	Nios II Processor						
		clk	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		reset_n	Reset Input	*Double-click to export*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]	IRQ 0	IRQ 31		
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		jtag_debug_module_reset	Reset Output	*Double-click to export*	[clk]				
		jtag_debug_module	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1800	0x0000_1fff		
		custom_instruction_master	Custom Instruction Master	*Double-click to export*					
✓		**timer_0**	Interval Timer						
		clk	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_10e0	0x0000_10ff	3	
✓		**timer_1**	Interval Timer						
		clk	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_10c0	0x0000_10df	4	
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)						
		clk1	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	0x0000_0fff		
		reset1	Reset Input	*Double-click to export*	[clk1]				
✓		**jtag_uart_0**	JTAG UART						
		clk	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		reset	Reset Input	*Double-click to export*	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	0x0000_1007	0	
✓		**new_sdram_controller_0**	SDRAM Controller						
		clk	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0100_0000	0x017f_ffff		
		wire	Conduit	**sdram_wire**					
✓		**to_external_bus_bridge_0**	Avalon to External Bus Bridge						
		clock_reset	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		clock_reset_reset	Reset Input	*Double-click to export*	[clock_reset]				
		avalon_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock_reset]	0x0400_0000	0x0400_ffff	1	
		external_interface	Conduit	**io**					
✓		**rs232_0**	RS232 UART						
		clock_reset	Clock Input	*Double-click to export*	**up_clocks_0_sys_clk**				
		clock_reset_reset	Reset Input	*Double-click to export*	[clock_reset]				

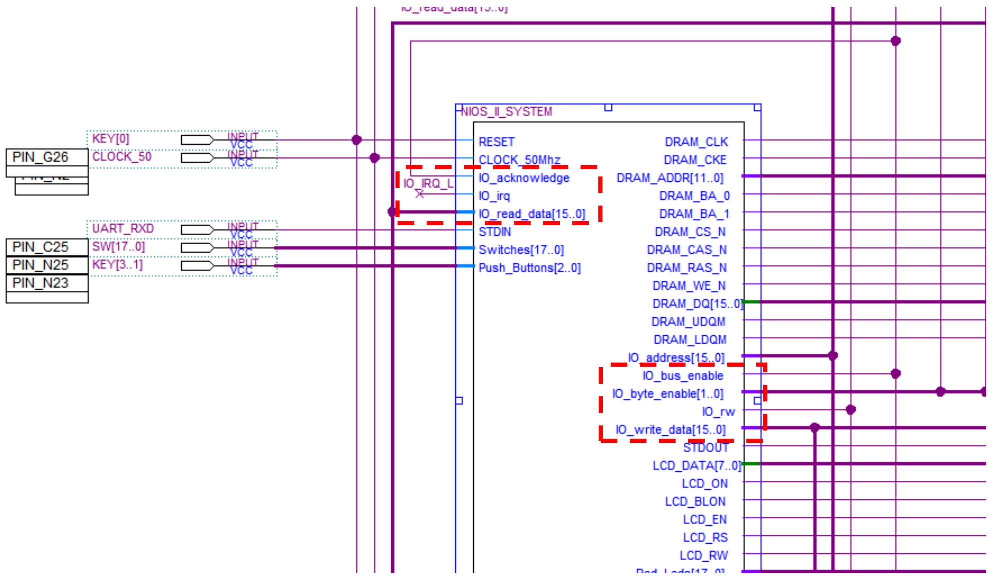
Generate the new system by clicking on the **Generation tab** and then **generate** button as before. You will now have to go and recompile your NIOS computer design in Quartus and finally add the new signals to the NIOS symbol and update as we did in tutorial 1.3. Here's the VHDL with the Bridge signals added (labelled IO). Double click your NIOS II System symbol on your top level schematic diagram

```
component nios_system is
        port (
                -- start of bridge signals
                io_acknowledge          : in    std_logic               := 'X';         -- acknowledge
                io_irq                  : in    std_logic               := 'X';         -- irq
                io_address              : out   std_logic_vector(15 downto 0);          -- address
                io_bus_enable           : out   std_logic;                              -- bus_enable
                io_byte_enable          : out   std_logic_vector(1 downto 0);           -- byte_enables
                io_rw                   : out   std_logic;                              -- rw
                io_write_data           : out   std_logic_vector(15 downto 0);          -- write_data
                io_read_data            : in    std_logic_vector(15 downto 0) := (others => 'X');
        -- end of bridge signals
        );
end component nios_system;
```

```
        -- Instantiate the Nios II system entity generated by the Qsys tool.
        NiosII: nios_system
        PORT MAP (
                --  start of port map for Bridge

                io_acknowledge          => IO_acknowledge,
                io_irq                  => IO_irq,
                io_address              => IO_address,
                io_bus_enable           => IO_bus_enable,
                io_byte_enable          => IO_byte_enable,
                io_rw                   => IO_rw,
                io_write_data           => IO_write_data,
                io_read_data            => IO_read_data,
        );
END Structure;
```

Recompile and update the symbol. Here is the completed NIOS II system symbol that I created (yes it has some other stuff too, but you can see the IO signals that we can use to connect to IO devices later.) The IO_IRQ_L signal shown below on the LHS will connect to an interrupt output on the IO device/peripheral we hang off the bridge, either that or we must tie it to logic '1' to prevent false interrupt requests being generated.



Accessing the interface in C

You will recall from tutorial 1.8 that it is problematic to use a pointer to access IO device data such as switch inputs and in this case, Slave devices connected to a bridge when the CPU employs a **data cache** since there is no guarantee that the CPU will perform genuine read accesses to the device, it may instead use the data from it's cache which could be "stale" and not reflect the actual data coming in from the real world. The solution in tutorial 1.8 was to use functions/macros such IOWR_8DIRECT() or IORD_8DIRECT() to access 8 bit interface.

However we *can* use a pointer if we employ a little *trick*. If you are using NIOS II/f with a data cache, then any access to memory where the most significant address bit is set to '1' (i.e. address bit 31 in an address bus from a0-a31) will cause NIOS to bypass the cache and perform a true access. That is any address such as **0x8XXX_XXXX** will cause an access to location **0x0XXX_XXXX**. In other words, a31 is only used as a means to *use* or *bypass* the data cache within our code. The value presented to the bridge and hence the Slave device will always have its a31 line set to 0. We can make use of this here.

```
#include <stdio.h>
#include "io.h"

// define a pointer to 16 bit data at address hex 0400_0000
// but set most significant bit of address to '1' i.e. hex 8400_0000

#define IO_BRIDGE  (volatile unsigned short int *)0x84000000

int main()
{
      unsigned short int data;

// write 16 bits of data to an IO device via the bridge at address 0400_0000
      *IO_BRIDGE = 0x0055 ;

// write 16 bits of data to an IO device via the bridge at address 0400_0002
      *(IO_BRIDGE+1) = 0xaa55 ;

// write 16 bits of data to an IO device via the bridge at address 0400_0004
      *(IO_BRIDGE+2) = 0xff00;

// read 16 bits of data to an IO device via the bridge at address 0400_0000
      data = *IO_BRIDGE;

// read 16 bits of data to an IO device via the bridge at address 0400_0002
      data = *(IO_BRIDGE+1);

// read 16 bits of data to an IO device via the bridge at address 0400_0004
      data = *(IO_BRIDGE+2);

      return 0 ;
}
```

Try this approach with the bit_flipper program in tutorial 1.8, e.g. define this new pointer to 32 bit data located at address 0x80002010

```
#define bit_flipper_base ((volatile unsigned int *)(0x80002010))
```

Now instead of using something like this:-

```
IOWR_32DIRECT(bit_flipper_base,0,v);   // write the value to the component
v = IORD_32DIRECT(bit_flipper_base,4)  // read a value from the component
```

Try this instead

```
*(bit_flipper_base + 0) = v ;
v = *(bit_flipper_base + 4) ;
```

As you have probably guessed, the `IORD_32DIRECT` and `IOWR_32DIRECT` macros are just small bits of code that create a pointer with bit a31 set to 1 (within the CPU) by taking your programs address and ORing it with 0x8000000 and then adds the offset to the result as we did above.