AXI Timeout Bridge Example Design for Stratix 10 GX Development Kit

Author: Juliusz Frackowiak,
EU MAG AE,
juliusz.frackowiak@intel.com

Table of Contents

1. General description.

This document describes how to operate within AXI Timeout Bridge Example Design for Stratix 10 GX Development Kit.
AXI Timeout Bridge IP allows to perform debug process on interconnection bus, like AXI-4 or Avalon MM. Although the IP was primarily designed for AXI-4 bus, if you connect the IP to AV MM bus the Interconnect inserted by Quartus tool will seamlessly connect the IP to the AV MM bus.

The IP consists of the following interfaces:

- *clock* and *reset* – the regular interfaces
- *slave-master* – which is the actual interface being protected and monitored by the IP
- *csr* – interface for Nios master to read monitoring data after the failure happens or to clear the interrupt
- *interrupt* – interrupt signal that indicates the failure happens

Here is a brief description of the AXI Timeout Bridge IP from our document:

The AXI Timeout Bridge Intel® FPGA IP allows your system to recover when it freezes, and facilitates debugging. You can place an AXI Timeout Bridge between a single manager and a single subordinate if you know that the subordinate may time out and cause your system to freeze. If a subordinate does not accept a command or respond to a command it accepted, its manager can wait indefinitely.
For a domain with multiple managers and subordinates, placement of an AXI Timeout Bridge in your design may be beneficial in the following scenarios:

- To recover from a freeze, place the bridge near the subordinate. If the manager attempts to communicate with a subordinate that freezes, the AXI Timeout Bridge frees the manager by generating error responses. The manager is then able to communicate with another subordinate.
- When debugging your system, place the AXI Timeout Bridge near the manager. This placement enables you to identify the origin of the burst, and to obtain the full address from the manager. Additionally, placing an AXI Timeout Bridge near the manager enables you to identify the target subordinate for the burst.

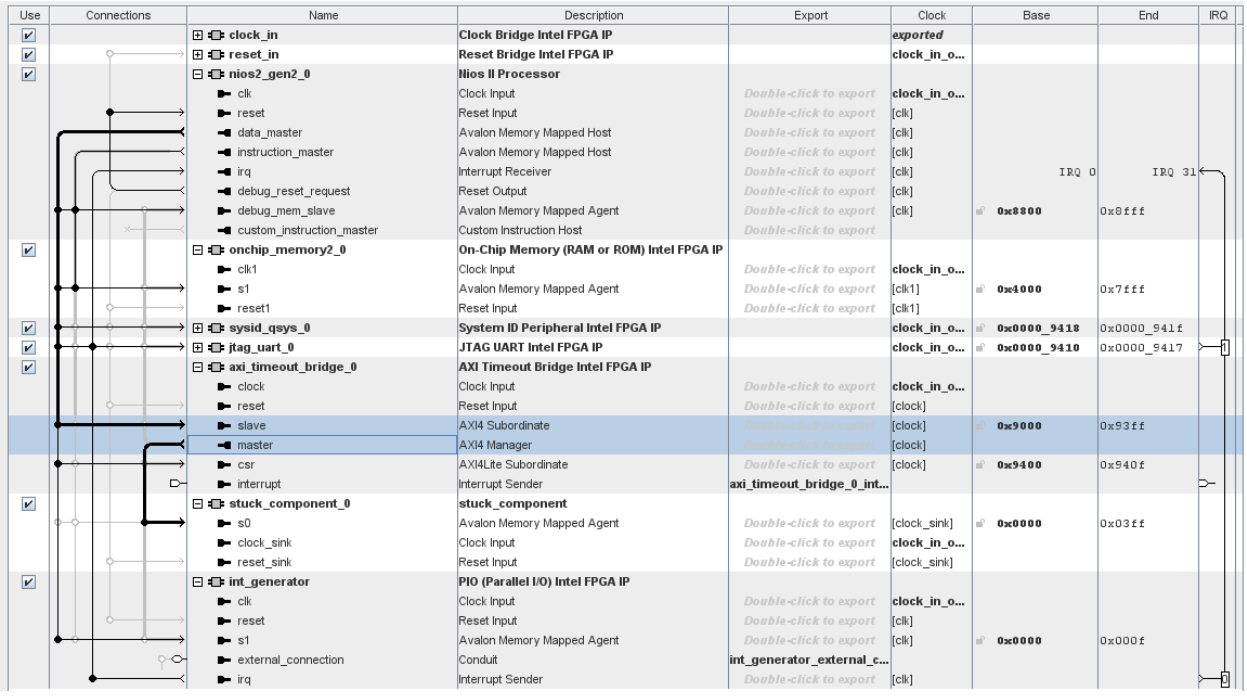
  Note: If you place the bridge at the subordinate's side and you have multiple subordinates connected to the same manager, you do not get the full address.

2. AXI Timeout Bridge Project description.

The main part of Quartus design is Platform Designer system named *qsys_top*. In particular it consists of the following components:

- *nios2_gen2_0* – Nios II processor that executes its program, initiate the flow on AVMM bus through AXI Timeout Bridge and monitors the state of the AXI Timeout Bridge
- *onchip_memory2_0* – Onchip RAM for program for Nios processor
- *axi_timeout_bridge_0* – AXI Timeout Bridge IP component
- *stuck_component_0* – dummy component that creates malfunction on the bus
- *int_generator* – Parallel Input/Output IP component that service interrupt to the Nios processor; this component is used because IRQ of the AXI Timeout Bridge IP is not working correctly at this point of time, it will be fixed in the future Quartus release

Below is the picture of the PD system where you can see AVMM interface from Nios to the AXI Timeout Bridge (nios2_gen2_0.data_master – axi_timeout_bridge_0.slave connection) and then AVMM interface from AXI Timeout Bridge to the Stuck Component (axi_timeout_bridge_0.master – stuck_component_0.s0 connection).



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✔		clock_in	Clock Bridge Intel FPGA IP		exported			
✔		reset_in	Reset Bridge Intel FPGA IP		clock_in_o...			
✔		nios2_gen2_0	Nios II Processor					
		clk	Clock Input	Double-click to export	clock_in_o...			
		reset	Reset Input	Double-click to export	[clk]			
		data_master	Avalon Memory Mapped Host	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Host	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31	
		debug_reset_request	Reset Output	Double-click to export	[clk]			
		debug_mem_slave	Avalon Memory Mapped Agent	Double-click to export	[clk]	0x8800	0x8fff	
		custom_instruction_master	Custom Instruction Host	Double-click to export				
✔		onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel FPGA IP					
		clk1	Clock Input	Double-click to export	clock_in_o...			
		s1	Avalon Memory Mapped Agent	Double-click to export	[clk1]	0x4000	0x7fff	
		reset1	Reset Input	Double-click to export	[clk1]			
✔		sysid_qsys_0	System ID Peripheral Intel FPGA IP		clock_in_o...	0x0000_9418	0x0000_941f	
✔		jtag_uart_0	JTAG UART Intel FPGA IP		clock_in_o...	0x0000_9410	0x0000_9417	1
✔		axi_timeout_bridge_0	AXI Timeout Bridge Intel FPGA IP					
		clock	Clock Input	Double-click to export	clock_in_o...			
		reset	Reset Input	Double-click to export	[clock]			
		slave	AXI4 Subordinate	Double-click to export	[clock]	0x9000	0x93ff	
		master	AXI4 Manager	Double-click to export	[clock]			
		csr	AXI4Lite Subordinate	Double-click to export	[clock]	0x9400	0x940f	
		interrupt	Interrupt Sender	axi_timeout_bridge_0_int...				
✔		stuck_component_0	stuck_component					
		s0	Avalon Memory Mapped Agent	Double-click to export	[clock_sink]	0x0000	0x03ff	
		clock_sink	Clock Input	Double-click to export	clock_in_o...			
		reset_sink	Reset Input	Double-click to export	[clock_sink]			
✔		int_generator	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clock_in_o...			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Agent	Double-click to export	[clk]	0x0000	0x000f	
		external_connection	Conduit	int_generator_external_c...				
		irq	Interrupt Sender	Double-click to export	[clk]			0

3. Test program performed by Nios processor.

The cycles of operation of the test program can be divided into the following steps:

1. In the *main* routine of the Nios application program has the part responsible for interrupt configuration:

```
int main()
{

    alt_putstr("Hello from Nios II!\n");

    register_interrupt();

    timed_out_operation = IORD_32DIRECT(AXI_TIMEOUT_BRIDGE_0_BASE, AXI_TOB_OPERATION_ADR);
```

   The *register_interrupt* function sets up the interrupt from the *int_generator* (PIO IP) component:

```
static void register_interrupt()
{
    // Enable pio interrupt.
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(INT_GENERATOR_BASE, 0x01);
    // Register handler.
    alt_ic_isr_register(INT_GENERATOR_IRQ_INTERRUPT_CONTROLLER_ID, INT_GENERATOR_IRQ, axitimeoutbridge_isr, NULL, 0);
    alt_ic_irq_enable(INT_GENERATOR_IRQ_INTERRUPT_CONTROLLER_ID, INT_GENERATOR_IRQ);
}
```

2. After configuring interrupt, the main routine clears the potential interrupt within AXI Timeout Bridge component:

```
    alt_putstr("main Clearing Timeout Bridge IRQ\n");
    IOWR_32DIRECT(AXI_TIMEOUT_BRIDGE_0_BASE, AXI_TOB_SLAVEISRESET, 0x01);
```

3. After that the main routine performs a few write/read operations to/from the *stuck_component*, like the one below:

```
    alt_putstr("main Writing 0x5678 to address 4 of test component\n");
    IOWR_32DIRECT(STUCK_COMPONENT_0_BASE, 4, 0x5678);
    alt_putstr("main Reading from address 4 of test component\n");
    reg_temp = IORD_32DIRECT(STUCK_COMPONENT_0_BASE, 4);
    alt_printf("main Readback of address 4 = %x\n", reg_temp);
    alt_putstr("\n");
```

4. At every write or read operation there is a malfunction on the bus (*stuck_component* is not responding properly on write or read requests) and the AXI Timeout Bridge interrupt is activated.

5. This interrupt signal is routed to the *int_generator* component by exporting interrupt signal to the top-level module. Here is the connection on the *top_timeoutbridge.v* module:

```
top_timeoutbridge.v

wire qsys_top0_irq_bridge;
    qsys_top qsys_top0 (
        .axi_timeout_bridge_0_interrupt_irq       (qsys_top0_irq_bridge),        //  output,
        .int_generator_external_connection_export (qsys_top0_irq_bridge), //   input,  widt
        .clk_clk       (iopll0_outclk0),      //   input,  width = 1,   clk.clk
        .reset_reset  (!iopll0_locked)  //   input,  width = 1, reset.reset
    );
```

6. The interrupt signal of the *int_generator* component is connected to the Nios component in the regular way. The presence of interrupt signal executes processing of the *axitimeoutbridge_isr* function, which was configured at step 1.
   In particular this function clears the interrupt within *int_generator* component (PIO IP):

```
    // Clear interrupt in PIO component
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(INT_GENERATOR_BASE, 0x01);
```

   reads *timed_out_operation** and *timed_out_address** values and prints it out (* you can see more info on these values here):

```
    // Read AXI_TIMEOUT_BRIDGE data
    timed_out_operation = IORD_32DIRECT(AXI_TIMEOUT_BRIDGE_0_BASE, AXI_TOB_OPERATION_ADR);
    timed_out_address = IORD_32DIRECT(AXI_TIMEOUT_BRIDGE_0_BASE, AXI_TOB_ADDRESS_ADR);

    printf_8hex_dec("isr timed_out_operation = %x\n", timed_out_operation);
    printf_8hex_dec("isr timed_out_address = 0x%x\n", timed_out_address);
```

   and clears the interrupt of AXI Timeout Bridge component:

```
     alt_putstr("isr Clearing Timeout Bridge IRQ\n");
     IOWR_32DIRECT(AXI_TIMEOUT_BRIDGE_0_BASE, AXI_TOB_SLAVEISRESET, 0x01);
```

7. After that program goes back to the main routine and performs next write/read operations.