# ORCA® Series 4 MPI/System Bus

March 2002 Technical Note TN1017

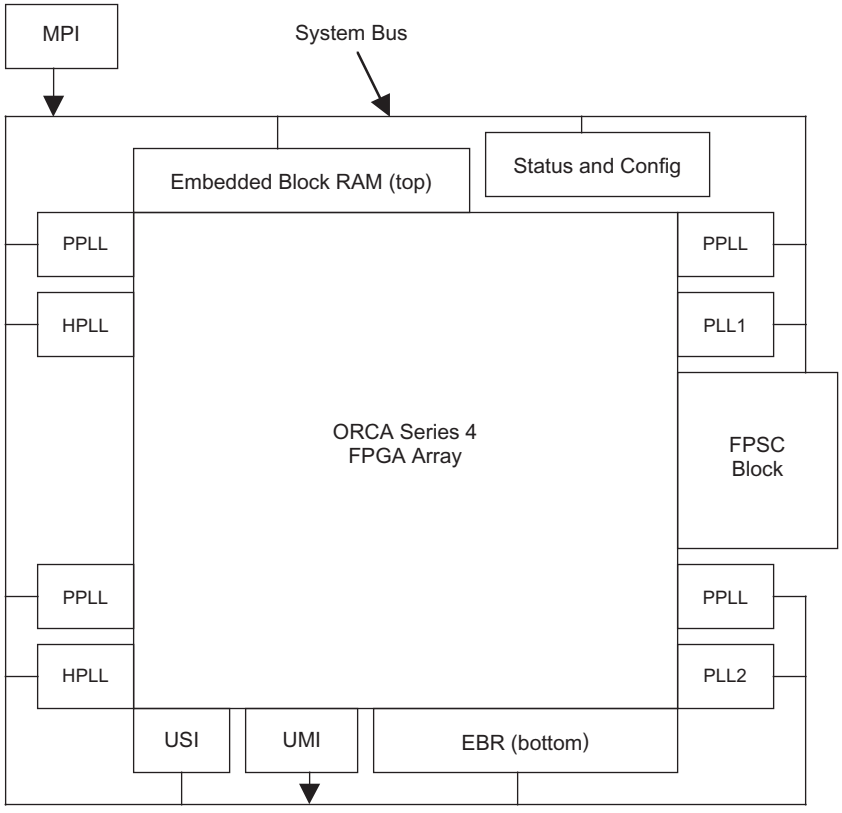
## Introduction

The Lattice Semiconductor ORCA Series 4 devices contain an embedded microprocessor interface (MPI) that can be used to interface any Series 4 field-programmable gate array (FPGA) or field-programmable system chip (FPSC) to any MPC860/MPC8260 PowerPC microprocessor or compatible interface.

The MPI is available prior to and optionally after configuration of the programmable logic in the FPGA/FPSC. The MPI can be used prior to device configuration to identify, test, initialize, and download configuration data into the device. After configuration of the programmable logic, the MPI can be used to read back the configuration and internal status data, write or read the contents of the embedded random access memory (RAM) blocks, control parameters in the phase-locked loops (PLL), access status and control registers for an embedded FPSC block (if present), and interact with the users design configured in programmable logic.

The MPI is one element on the embedded system bus illustrated in Figure 1. The system bus provides multi-master/multi-slave communication between the MPI and the status and configuration interface, the embedded RAM interfaces (RAMT, RAMB), the PLLs, the user logic interface (ULI), and one or more FPSC interface blocks as needed in each specific FPGA/FPSC device.

***Figure 1. ORCA Series 4 Embedded System Bus Elements***




This document describes the operation of each element on the embedded system bus and illustrates how to use these embedded features in conjunction with programmable logic in the device.

Terms used:

- **MSB** - Most Significant Byte
- **MSb** - Most Significant Bit
- **LSB** - Least Significant Byte
- **LSb** - Least Significant Bit
- **MPI** - Microprocessor Interface
- **UMI** - User Master Interface
- **USI** - User Slave Interface
- **EBR** - Embedded Block RAM
- **PLL** - Phase-Lock Loop

## System Bus

The embedded system bus on the ORCA Series 4 ties all of the programmable elements together in a bus framework. There are two types of interfaces on the system bus; master and slave. A master interface has the ability to perform actions on the bus such as writes and reads to and from a specific address. A slave interface responds to the actions of a master by accepting data and address on a write and providing data on read. The system bus has a memory map which describes each of the slave peripherals that is connected on the bus. Using the addresses listed in the memory map a master interface can access each of the slave peripherals on the system bus. Any and all peripherals on the system bus can be used at the same time. Table 1 lists all of the available user peripherals on the system bus.

***Table 1. System Bus User Peripherals***

| Peripheral | Interface Type |
|---|---|
| MPI | Master |
| User Master Interface | Master |
| User Slave Interface | Slave |
| PLLs | Slave |
| Embedded Block RAMs | Slave |
| FPSC | Master or Slave (Specific to Device) |

The system bus decodes 18 bits (256k addresses) in the device address space. The address space is divided into eight address ranges as shown in Table 1. The contents of some address ranges will vary slightly from device to device. For example, the number of block RAMs implemented in RAM banktop (RAMT) and RAM bankbottom (RAMB) will be different from one device to another based on the size of the device. Likewise, generic Series 4 FPGAs do not implement any registers in the FPSC address range, while each FPSC device implements various functions consistent with the requirements of a specific embedded core.

***Table 2. System Bus Address Ranges***

| Start Address | End Address | Size | Description |
|---|---|---|---|
| 0x00000 | 0x001FF | 1k | Status and control registers (STAT) |
| 0x00200 | 0x07FFF | 31k | Reserved |
| 0x08000 | 0x0FFFF | 32k | User logic slave interface (ULI) |
| 0x10000 | 0x17FFF | 32k | RAM banktop (up to 16 banks of data and parity) |
| 0x18000 | 0x1FFFF | 32k | Reserved for parity bit storage |
| 0x20000 | 0x27FFF | 32k | RAM bankbottom (up to 16 banks of data and parity) |
| 0x28000 | 0x2FFFF | 32k | Reserved for parity bit storage |
| 0x30000 | 0x3FFFF | 64k | FPSC slave interface (FPSC) |

## Multi Master

The system bus is a multiple master bus meaning that there can be more than one bus master present on the bus at the same time. A single bus arbiter controls the traffic on the bus by ensuring only one master has access to the bus at any time. This bus arbiter monitors a number of different requests to use the bus and decides which request is currently the highest priority. The configuration logic has the highest priority and overrides all normal user interfaces. By default, all master interfaces have equal priority when requesting the embedded system bus, and a fair round robin scheme is used to rotate arbitration priority. Optionally, you can specify a priority of low=1, medium=2, or high=3 for each master interface in SCUBA® (discussed later). As a result, if all three master interfaces are waiting for the system bus, an interface with higher priority will be granted the bus over one with a lower priority. If two requesting interfaces share the same priority, the round robin scheme will rotate arbitration priority.

## System Bus Clock (HCLK)

The system bus is a synchronous element that has an internal clock. This clock is sometimes referred to as the HCLK. Each of the peripherals on the system are also synchronous and have an interface clock. This peripheral clock is the clock that the FPGA designer sources to the peripheral to clock data in and out of the user interface and eventually onto the system bus. The system bus itself needs to be sourced by a clock (HCLK). This main system bus clock is the clock on which all of the traffic will run through the system bus. As traffic is passed to and from each peripheral a domain change will occur from the system bus clock domain to the peripheral domain. For burst and single transfers this domain change is taken care of inside the system bus.

The system bus clock can be driven from several sources; CCLK, JTAG TCK, MPI, FPSC, User, or Oscillator. Before the bitstream is loaded into the FPGA (pre-configuration) the system bus clock is selected via the mode pins. After the bitstream has been successfully loaded into the FPGA the system bus clock is selected by the bitstream programming in the FPGA design.

### CCLK

Before a bitstream is loaded and during device configuration and reconfiguration the system bus clock is defaulted to the configuration clock (CCLK). The CCLK is either an input or output of the FPGA depending on the configuration mode pins state. The CCLK is driven internally on the system bus from the configuration logic block. There will be no input or output pin associated on the system bus module for the CCLK. Using the CCLK is for pre-configuration only and can not be simulated.

### JTAG TCK

If the user interrupts the configuration cycle with a JTAG program the TCK clock will then drive the system bus clock and configuration logic. The JTAG TCK clock is also selected during a readback via the built-in JTAG test access port during device operation. The JTAG TCK clock is driven internally on the system bus from the configuration logic block. There will be no input or output pin associated on the system bus module for the JTAG TCK. Using the JTAG TCK is for pre-configuration and readback only and can not be simulated.

### MPI

If the user has selected to program the device via the MPI then the microprocessor clock will drive the system bus during configuration. In post configuration the MPI option is set via the bitstream and will select the clock on the MPI to drive the system bus. The MPI clock corresponds to the input mpi_clk on the system bus module. This is most often selected since the MPI is the master most often used to access the system bus and its peripherals. Using the MPI clock the system bus clock can be driven up to speeds of 66MHz.

### FPSC

The FPSC option allows the embedded ASIC block of the FPGA to drive the system bus clock. The FPSC Configuration Wizard will connect the FPSC to the system bus module in the FPSC template. The FPSC clock will be connected to the correct module pin on the system bus for this mode.

### User

The User clock allows the FPGA design to source a clock to drive the system bus clock. This user clock input port is provided on the usr_clk port of the system bus. The user clock can be used to create a synchronous interface

between the user interfaces (master and slave) and the system bus. To create a synchronous interface between the interfaces simply use the same clock source for the usr_clk and interface clocks um_clk and us_clk.

The user clock is driven from the FPGA design and thus can be driven by any signal in the user's design. If the user clock is derived in any way from the output of a PLL, changing the PLL control register will not be allowed. When changing the PLL control register the output clock from the PLL will stop for a period of time. If this clock stops the user clock will stop and thus the HCLK will stop. If the HCLK stops the system bus will lock. The PLL control register will not be able to be changed and the FPGA will lock. To prevent this condition do not drive this signal with a PLL output or do not change the PLL control register for this PLL.

**Oscillator**
The Oscillator option allows the internal configuration oscillator to drive the system bus clock. The internal oscillator drives the CCLK during master mode pre-configuration. This is a valid option to select for the system bus clock driver, but does not have a practical application after configuration.

## System Bus Interrupts

The system bus has the ability to generate and accept interrupt signals from several peripherals. These interrupts can then be used to alert either on chip modules or external modules. An internal interrupt can be generated from the FPGA design, the configuration block during device configuration, or from the embedded ASIC block of an FPSC. The interrupt cause register (0x00010) contains a bit for each source of an interrupt. When one of the peripherals sets an interrupt the particular bit in the cause register will bet set to a 1. To clear the interrupt cause bit a master interface will need to write a 1 to clear the bit.

Some peripherals on the system bus can pass interrupts found in the interrupt cause register through their interface. For a peripheral to pass interrupts through the interface the interrupt enable register must be properly provisioned. There is an interrupt enable register for each possible master interface; MPI (active low mpi_irq), User (active hi user_irq), and FPSC. When a bit is set to a 1 in the interrupt enable register (0x00011 - 0x00013), this corresponding interrupt will be passed through the peripheral. When the interrupt signal for the peripheral indicates an interrupt was created, a master should read the interrupt cause register to determine the source of the interrupt.

For example, to pass interrupts created by device configuration to the MPI the interrupt enable register (0x00013) must set the CFG_IRQ and CFG_ERR bits. When the configuration logic throws an interrupt CFG_IRQ and CFG_ERR in the interrupt cause register will go to a 1. These interrupts will then drive the mpi_irq pin low to indicate an interrupt to the microprocessor. The microprocessor should then read the interrupt cause register to determine that the interrupt came from the configuration block.

For more information on the interrupt cause register and interrupt enable register refer to the system bus memory map.

## Address and Data Bus Ordering

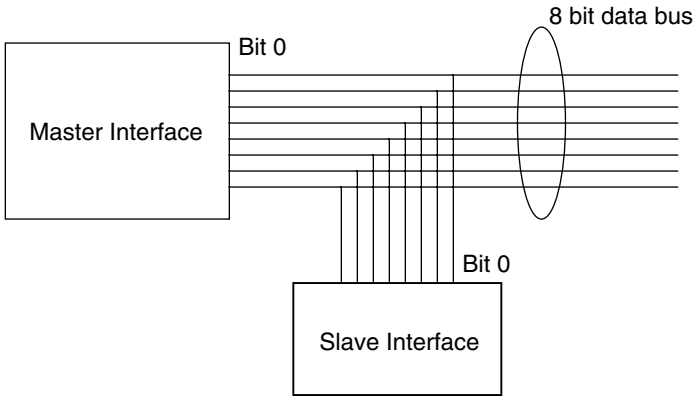
The system bus handles bus transfers of address and data at all of its interfaces. The orientation of the address bus and the data bus may be different depending on which peripheral is accessed. Internally on the bus the address bus is always oriented the same, where the data bus is dependent on the driving master/slave interface.

There are 18 address bits (17:0) on the system bus. These address lines can be driven by any of the master interfaces on the system bus. The address bus on a slave interface will always be provided to the slave with bit 0 as the LSb. For the master interfaces (MPI, User, and FPSC) the address bus is dependent on the master. The user interface will use the same orientation as the slaves, bit 17 as the MSb and bit 0 as the LSB. The FPSC interface will be dependent on the FPSC implementation. The MPI will match the PowerPC address bus orientation which is bit 0 MSb and bit 31 the LSb. More information on this address bus and its connections are found in the MPI section of this document.

The data bus on the system bus is 36 bits wide, 32 bits for data and 4 bits for optional parity. On the system bus the data bits are passed unchanged through the peripherals. So the orientation of the data on the system bus and is dependent on the data driving master/slave interface. This means that bit 0 on the master interface will be bit 0 on

the slave interface, bit 1 will be bit 1, etc. So if bit 0 is the LSb on the master interface then bit 0 will be the LSb on the slave interface as shown in Figure 2. Care must be taken by the user when accessing an address location to make sure data bits are used properly in terms of LSb and MSb.

***Figure 2. System Bus Data Bus Bit Mapping***




## Internal Data Bus

The internal data bus is labeled d(35:0). Bits d(31:0) are used to carry the data between peripherals. Bits d(35:32) are used to carry parity to the peripherals. Parity is not checked internally by any of the peripherals on the system bus. If parity is enabled (SCUBA), internal non-user defined registers will generate parity for read operations. User parity is only passed through the system bus and its interfaces. Parity is mapped to byte lanes as shown in Table 3. The orientation of the data bus d(31:0) depends on the master interface driving the data bus, but the parity bits are always stored on d(35:32).

***Table 3. Internal Data Bus Bit Mapping***

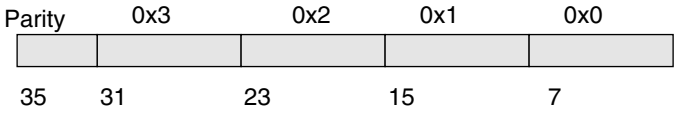
| Internal Data Bus | MPI | User Master Interface | Description |
|---|---|---|---|
| D(35) | Mpi_parity(4) | Um_w/rdata(35) | Parity bit for d(31:24) |
| D(34) | Mpi_parity(3) | Um_w/rdata(34) | Parity bit for d(23:16) |
| D(33) | Mpi_parity(2) | Um_w/rdata(33) | Parity bit for d(15:8) |
| D(32) | Mpi_parity(1) | Um_w/rdata(32) | Parity bit for d(7:0) |
| D(31:24) | Mpi_data(31:23) 32-bit LSB, bit 31 = LSb | Um_w/rdata(31:24) | Data byte 31:24 - LSB of the MPI in 32-bit mode. |
| D(23:16) | Mpi_data(23:16), bit 23 = LSb | Um_w/rdata(23:16) | Data byte 23:16 |
| D(15:8) | Mpi_data(15:8), 16-bit LSB, bit 15 = LSb | Um_w/rdata(15:8) | Data byte 15:8, LSB of the MPI in 16-bit mode |
| D(7:0) | Mpi_data(7:0), 8-bit LSb=7, 16-bit/32-bit MSB | Um_w/rdata(7:0) | Data byte 7:0, MSB of the MPI in 16/32-bit mode. In MPI 8-bit mode bit 0 is MSb and bit 7 is the LSb |

## 32-bit Data Bus

When a master interface is configured for 32-bit operation all 32 bits (d(31:0)) are used to carry the data. If the master interface is using parity, then the four parity bits will be carried on d(35:32) using the parity byte mapping previously described. Again, the orientation (MSb,LSb) of the data bus depends on which master/slave interface drives data onto the system bus. When using a 32-bit data bus the address resolution will be limited to 32-bit boundaries. Data will be carried on the bus such that the lowest address maps to byte lane d(31:24) and the highest address maps to byte d(7:0).

For example a 32-bit read at address 0x00000 will read addresses 0x00000 - 0x00003. Address 0x00000 data will be on d(7:0), address 0x00001 data will be on d(15:8), address 0x00002 data will be on d(23:16), and address 0x00003 data will be on d(31:24) as shown in Figure 3.

***Figure 3. 32-bit Data Bus***

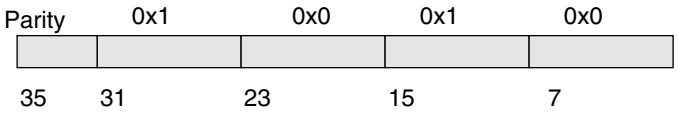



## 16-Bit Data Bus

For 16-bit operation of a master interface the data will be carried on d(15:0). The slave interface will only need to read data from d(15:0). The MSB and LSB of the data will be determined by how the data was driven onto the bus by the master/slave interface. When using a 16-bit data bus the address resolution will be limited to 16-bit boundaries. Data will be carried on the bus such that the lowest address maps to byte lane d(15:8) and the highest address maps to byte d(7:0). The same data will also be replicated on d(31:16) along with parity(3:2).

For example a 16-bit read at address 0x00000 will read addresses 0x00000 - 0x00001. Address 0x00000 data will be on d(7:0) and address 0x00001 data will be on d(15:8) as shown in Figure 4.

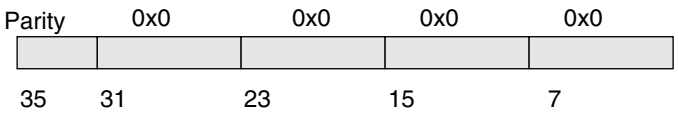
***Figure 4. 16-bit Data Bus***




## 8-Bit Data Bus

For 8-bit operation of a master interface the data will be carried on d(7:0). The slave interface will only need to read and write data on d(7:0). The same data will also be replicated on d(31:24), d(23:16), and d(15:8) along with parity(3:1) as shown in Figure 5. The MSb and LSb of the data will be determined by the master/slave interface.

***Figure 5. 8-Bit Data Bus***




## MPI Data/Address Ordering

When using the MPI, the external chip data bus is labeled mpi_d(0:31) for a 32 bit bus. For the PowerPC mpi_d(0) is the MSb while mpi_d(31) is the LSb. The internal system bus data is d(35:0). The data bus on the system is a one to one mapping of bits. So mpi_d(0) connects to internal data bus bit d(0) as the MSb of the PowerPC. The resulting slave interface will see d(0) as the MSb.

**Example: PowerPC writes an 8-bit value of 0x01 to a USI address.**

***Table 4. Example of MPI to USI***

| Data Bus | MPI | Internal | Us_rdata |
|---|---|---|---|
| Bit 0 | 0 (MSb) | 0 | 0 |
| Bit 1 | 0 | 0 | 0 |
| Bit 2 | 0 | 0 | 0 |
| Bit 3 | 0 | 0 | 0 |
| Bit 4 | 0 | 0 | 0 |
| Bit 5 | 0 | 0 | 0 |
| Bit 6 | 0 | 0 | 0 |
| Bit 7 | 1 | 1 | 1 |

For the PowerPC a value of 0x01 will have bit 7 '1' with 0:6 bits '0'. On the internal data bus the bits map directly and on the resulting USI the data bits map directly as shown in Table 4. If the designer wants a value of 0x01 to be pulled off the us_rdata bus the orientation will need to be the same as the MPI with bit 0 as the MSb.

Mydata <= us_rdata(0:7)

### User Master Data Ordering

The user master interface on the system bus is defined by the FPGA design. The data bus orientation designed for the user master interface should match the targets of the user master interface. If the user master interface is targeting specific status and control registers, the orientation of the status and control registers will dictate the MSb and LSb of the data bus.

### FPSC Master Interface Data Ordering

The FPSC master interface on the system bus is defined by the FPSC block. The FPSC block design will dictate the data bus orientation on the slave interfaces the master references. The FPSC master interface is very specific to the FPSC device being used. More information on specific interfaces are provided in the FPSC data sheets.

## System Bus Peripherals

The following section discusses each of the peripherals on the system bus in detail. Some of the peripherals on the system bus do not have FPGA design user ports and are fully contained inside the system bus. These peripherals' registers will be described in the memory map and do not have any user inputs or outputs on the system bus.

### PowerPC MPI

The microprocessor interface (MPI) acts as a bridge between an external PowerPC processor and the embedded system bus. Externally the MPI acts as a slave peripheral interface through which a PowerPC can access the embedded features of the device. Internally the MPI acts as a master peripheral interface on the system bus to initiate data transfers as directed by the external processor.

The PowerPC interface built into the Series 4 ORCA devices is based on the PowerPC Big Endian mode of operation. Series 4 ORCA devices will bolt up gluelessly to a PowerPC in this mode only.

### MPI Interface

Table 5 shows the MPI signals used by the PowerPC to perform transactions with the ORCA device. The MPI is a fixed block on the FPGA array and the interface signals are mapped to dedicated pins on the ORCA device. PowerPC pins to ORCA pins mapping can be found in the appendix of this application note.

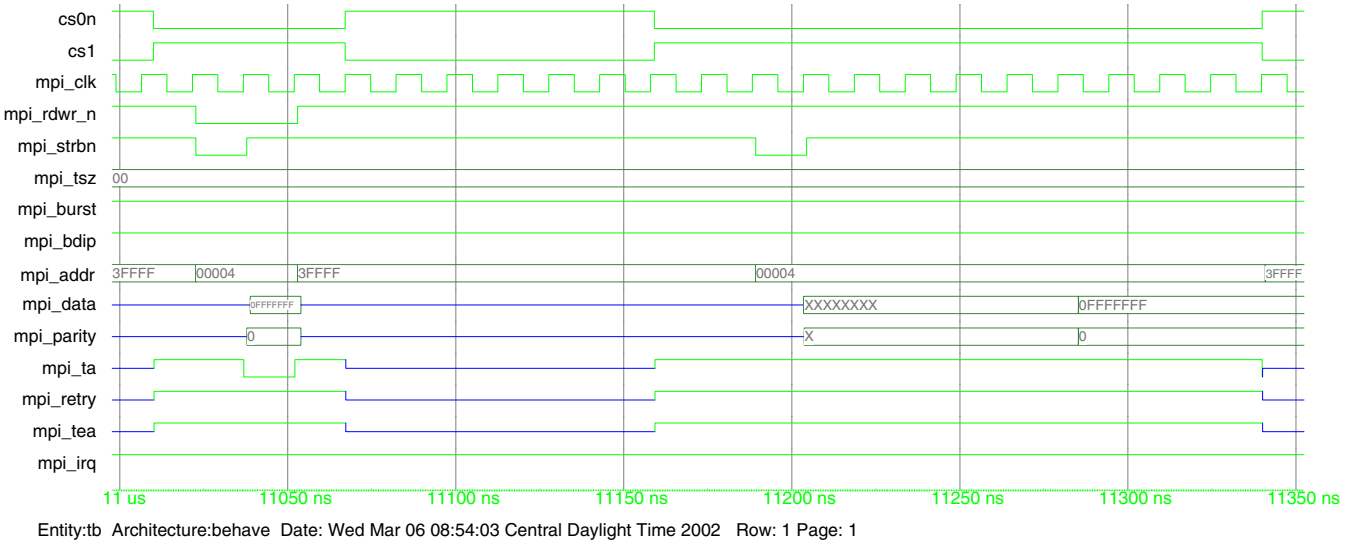
Externally the MPI implements a 36-bit PowerPC bus slave which internally drives the system bus as a master. data bus width is selectable among 8 bits, 16 bits, and 32 bits with parity of 1, 2, or 4 bits, respectively (one parity

bit for each active byte). Note that the MPI does not check or generate parity, but simply passes it from the PowerPC bus to the system bus and vice versa. It is left to the other peripherals on the system bus to check or generate correct parity.

***Table 5. MPI Signals to PowerPC Bus***

| Name | I/O | Description |
|---|---|---|
| MPI_CLK | I | This is the clock from the PowerPC (CLKOUT). This clock input will clock the MPI. This clock may optionally clock the main system bus clock if selected. The MPI clock can be driven up to 66Mhz operation. |
| MPI_TSZ(0:1) | I | Transfer size (00-double word, 10-word, 01-byte). The MPI_TSZ pins connect directly up to the PowerPC TSIZ1 and TSIZ0. These pins select the size of the PowerPC data transaction. This is the transfer size of the data transaction from the microprocessor's perspective. This is different from the MODE pin selected size discussed later. |
| MPI_RDWR_N | I | Transfer type (0-write, 1-read). This signal indicates to the MPI whether the transaction initiated by the microprocessor is a read or a write. If the transaction is a read data will be provided to the microprocessor from the address specified. If the transaction is a write data will be written to the address specified by the microprocessor inside the ORCA device. This signal connects to the PowerPC RD//WR signal. |
| MPI_BURST | I | Indicates that a burst transfer is in progress when low. This signal informs the MPI that the PowerPC is performing a burst transaction using the PowerPC burst pin. |
| MPI_BDIP | I | Burst Data In Progress. This signal from the PowerPC will go low on the first clock of data during a burst and go high on the last clock of data of the burst transfer. |
| MPI_STRBn | I | This active low signal indicates the start of a transactions or the strobe. This pin is connected to the TS pin of the PowerPC. |
| CS0n/CS1 | I | Chip selects for active high (CS1) and active low (CS0n). Both of these chip selects must be active for the ORCA device to be selected. Typically CS1 is connected to logic 1 and CS0n is connected to a PowerPC CS pin. |
| MPI_ADDR[14:31] | I | The PowerPC address bus is 32 bits wide. The Series 4 ORCA devices only support 18 bits of address space. The ORCA uses the least significant bits of the PowerPC address space using address bits 14:31. |
| MPI_DATA[0:31] | I/O | The PowerPC data bus can be up to 32 bits wide. Bit 0 is the MSb and bit 31 is the LSb. For multi-byte transfers, the most significant byte has the lowest address. The PowerPC data pins D[0:31] connect directly to the ORCA pins mpi_data(0:31). Data pins not used by virtue of selecting 8-bit or 16-bit data widths are available as general-purpose user I/O. |
| MPI_PARITY[0:3] | I/O | Parity can be up to 4 bits wide depending on the data bus size. Parity connects directly to the PowerPC DP[0:3] pins. Parity pins not used by virtue of selecting 8-bit or 16-bit data widths are available as general-purpose user I/O. |
| MPI_TA | O | This active low signal indicates the transfer acknowledge from the ORCA device. This pin is connected to the TA pin of the PowerPC. For a MPI write transaction the mpi_ta will come back on the next clock. See Consecutive Writes section. For a MPI read transaction the mpi_ta will come back after the target slave responds. The result is a difference in time it takes to complete a transaction depending on the slave that is accessed. Other dependencies are the clock rate of the system bus (HCLK), clock rate of the slave interface, and slave acknowledge protocol. |
| MPI_TEA | O | This active low signal indicates a transfer error acknowledge during the current transaction. More information on cause of this error can be found under MPI exceptions. |
| MPI_IRQ | O | Active-low interrupt request from the ORCA device. See system bus interrupts. |
| MPI_RETRY | O | Active-low request for processor to relinquish the bus and retry the cycle. Exception signal indicating the ORCA device is not ready to accept the requested transaction. More information on the cause of this error can be found under MPI exceptions. |
| MODE[3:0] | I | MPI data width (1010, 1011, 1110 => 8, 16, 32-bits, respectively). The MODE pins are used to select the type of bitstream configuration the ORCA device will utilize. More information on the MODE pins can be found under the MPI configuration section. |

***Figure 6. MPI Single Beat Data Transfer Timing***




## MPI Burst Transfers

The MPI will support burst transfers of 4 beats (32-bit width), 8 beats (16-bit width), or 16 beats (8-bit width), depending upon the selected data bus width. Burst transfers can be of any size that is compatible with the selected data bus width given the limitation that the MPI will handle 4, 8, or 16 beats as indicated.

The burst mechanism uses MPI_BURST to indicate that the transfer is a burst transfer and MPI_BDIP to indicate the duration of the burst. Figure 7 shows the signal timing for a 4-beat burst write. Figure 8 shows the signal timing for a 4-beat burst read.

***Figure 7. MPI Burst Write Transfer Timing***

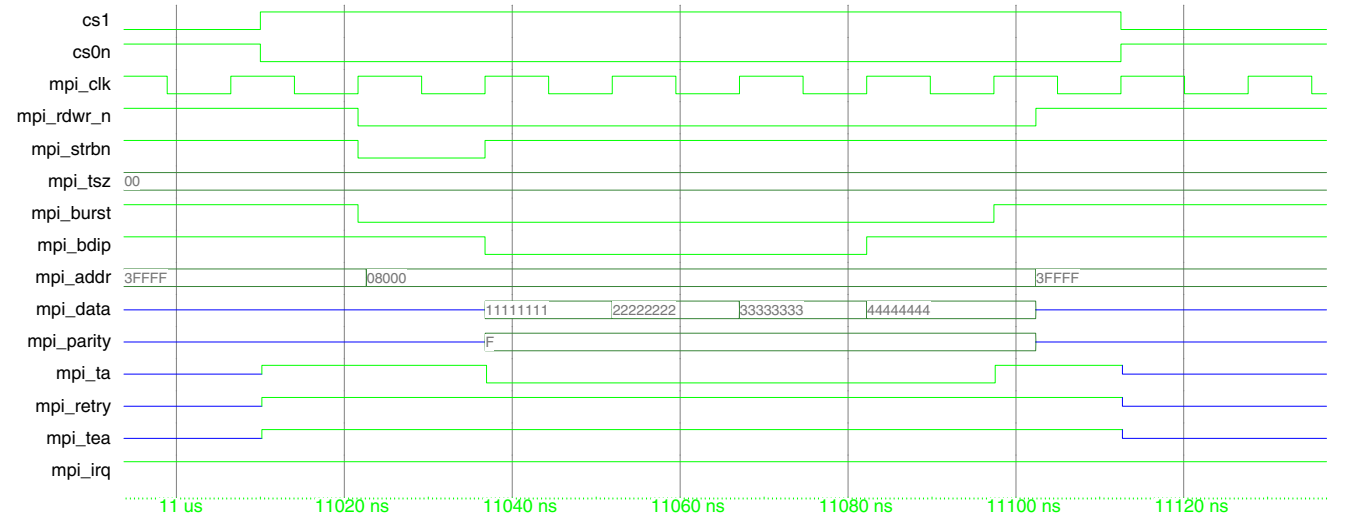
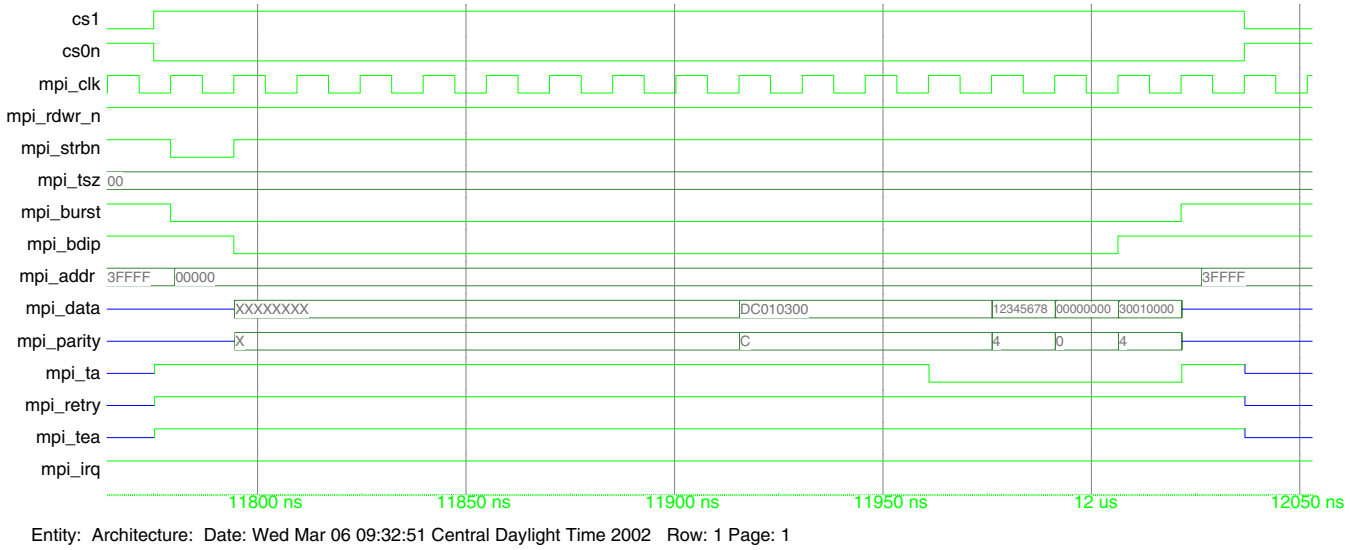


***Figure 8. MPI Burst Read Transfer Timing***




Along with the address and transfer control signals, the PowerPC asserts MPI_BURST during the address phase of the transfer. In the data phase, the microprocessor asserts MPI_BDIP until the next to the last word is received. The MPI continues to send/receive data until it detects MPI_BDIP deasserted at the rising edge of MPI_CLK while MPI_TA is asserted.

## Consecutive Writes with the MPI

The MPI uses a single write post buffer implementation. This means that on each MPI write, the MPI_TA comes back on the next clock cycle to terminate the PowerPC transaction. Internally on the MPI and system bus the data is not yet transmitted to the target. It will take several HCLK clock cycles before the target receives the data and terminates the transaction. Until this termination takes place any additional writes from the MPI will issue a retry (MPI_RETRY).

The number of PowerPC clock cycles (MPI_CLK) it takes until the next write can take place without a retry is variable. The variables include the source of the HCLK, the target being accessed, and the target's termination protocol and clock. The user should add appropriate delay based on their board and system behavior.

## MPI Exceptions

Three signals, MPI_TEA, MPI_RETRY, and MPI_TA are monitored during the termination phase of a transfer. A normal termination is indicated when MPI_TA is asserted and both MPI_TEA and MPI_RETRY are deasserted. If either MPI_TEA or MPI_RETRY are asserted, a MPI bus exception is indicated.

MPI_TEA is asserted for one MPI_CLK cycle to indicate either an internal system bus error, or a transfer with MPI_TSZ larger than the data port size, or physical data size selected by the MODE[3:0] inputs.

MPI_RETRY is asserted for one MPI_CLK cycle when the MPI is busy to request that the PowerPC relinquish the bus and reissue the current transfer. A retry is issued when the following occurs:

1. The MPI gets a read transaction while its write FIFOs are not empty.
2. The MPI gets a write transfer while its write FIFOs are full.
3. The MPI receives a retry indication from the embedded system bus during a read transfer.

For burst transfers, the MPI will issue retry before acknowledging the first data phase; if MPI_RETRY is asserted after the first data phase of a burst transfer, it should be treated as a transfer error (MPI_TEA).

## Enabling the MPI

To enable the MPI at power-up, prior to device configuration, the external MODE pins (Table 6) must be set to specify one of the three MPI configuration modes as specified in the ORCA Series 4 data sheet. If the MPI is not used all of the MPI pins (mpi_ta, mpi_data, etc.) are tristated during configuration.

***Table 6. ORCA Device Configuration Modes***

| M3 | M2 | M1 | M0 | Description |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Serial master (high-speed) |
| 0 | 1 | 0 | 0 | Parallel master (high-speed) |
| 0 | 1 | 0 | 1 | Asynchronous peripheral (high-speed) |
| 0 | 1 | 1 | 1 | Reserved |
| 1 | 0 | 0 | 0 | Serial master (low-speed) |
| 1 | 0 | 0 | 1 | Parallel slave |
| 1 | 0 | 1 | 0 | MPC860 8-bit |
| 1 | 0 | 1 | 1 | MPC860 16-bit |
| 1 | 1 | 0 | 0 | Parallel master (low-speed) |
| 1 | 1 | 0 | 1 | Asynchronous peripheral (low-speed) |
| 1 | 1 | 1 | 0 | MPC860 32-bit |
| 1 | 1 | 1 | 1 | Serial slave |

Data port width is determined at power-up by the value presented on the MODE pins during the low-to-high transition of the INIT signal. The MODE pins select the data size of the transaction including the parity bits. MPC860 8-bit mode will also use mpi_parity(0) along with the data. MPC860 16-bit will use mpi_parity(0:1) and MPC860 32-bit will use mpi_parity(0:4). Unused mpi_parity bits will be tristated during configuration. All of the other MPI signals connect directly to the PowerPC bus. Pad locations vary depending upon device type, size, and package.

The data port width selected by the MODE[3:0] pins is not related to the transfer size specified by MPI_TSZ[0:1]. The port width selected by the MODE[3:0] pins determines how many data pins are used by the MPI, while the transfer size is determined by the master interface on the PowerPC bus. The transfer size used by an external master must not exceed the selected data port width; otherwise, the higher-order data bits will be lost and a bus exception is issued by the MPI.

To enable the MPI for use after device configuration and during normal operation, the user must instantiate the system bus with an MPI peripheral in the FPGA design. The system bus element along with the MPI is created using SCUBA and is discussed later in this document.

A third option of enabling the MPI is also available that is good for debugging purposes. The MPI can be enabled during normal operation by setting the MPI_USR_ENABLE bit in the command register (0x08 bit 2). This is done either by writing to the control register prior to device configuration (if the MPI is enabled via the MODE pins). If the MPI_USR_ENABLE bit is not set, MPI signal pins revert to general purpose I/O pins once the configuration process is completed. Using this bit the user can utilize the MPI in the design without having to modify the HDL code to instantiate the system bus. This aids in debugging by now allowing visibility into the control and status registers.

If the MPI is not utilized at all in the FPGA design the dedicated MPI pins can be used as general I/O pins for the user's design. More information on MPI configuration of the ORCA Series 4 devices can be found in technical note number TN1013, *ORCA Series 4 FPGA Configuration.*

## User Master Interface (UMI)

The user logic master interface (UMI) allows the FPGA design to perform transactions on the system bus. Through the UMI the FPGA design has access to any and all of the slave peripherals on the system bus. Signals for the user logic master interface are listed in Table 7.

*Table 7. User Logic Master Interface Signals*

| Signal | Type | Description |
|---|---|---|
| um_clk | I | The main clock for the user master interface. This clock only clocks the interface registers. A domain change is made from the um_clk domain to the system bus clock domain. The user master interface and the system bus can be made synchronous by using the same clock for the um_clk and usr_clk and selecting User as the HCLK. A frequency preference on this signal will constrain all of the inputs and outputs of the UMI. |
| um_reset | I | This active-high reset resets all of the controls in the user master interface. This should be pulsed once before any transactions can occur on the UMI. This is typically connected to the same reset as the GSR and pulsed at power-up. |
| um_wdata[35:0] | I | 36-bit write data bus to system bus. This data bus is oriented with bits 35:32 used for parity while bits 31:0 are used for data. Parity is not calculated or checked by any of the peripherals on the system bus, it is only carried. The MSB and LSB must be selected based on the peripheral target and application. |
| um_rdata[35:0] | O | 36-bit read data bus to system bus. This data bus is oriented with bits 35:32 used for parity while bits 31:0 are used for data. Parity is not calculated or checked by any of the peripherals on the system bus, it is only carried. The MSB and LSB must be selected based on the peripheral target and application. |
| um_addr[17:0] | I | This 18 bit address bus is used to select the address where a user master transaction will target. Bit 17 of the um_addr bus is the MSB while bit 0 is the LSB. |
| um_read | I | Active-high read request indicates the current transaction is a read. Data will be expected to be found on the um_rdata bus. |
| um_write | I | Active-high write request indicates the current transaction is a write. Data will be expected to be driven onto um_wdata bus. |
| um_lock | I | This active-high signal will be used to request ownership of the system bus. |
| um_granted | O | If the system bus arbiter grants ownership to the UMI the um_granted signal will go high indicating the system bus is locked to the UMI. The um_granted signal is clocked off the HCLK, not the um_clk. This is used for debugging purposes only. The user should use um_ack as the signal to indicate the UMI is ready for transactions |
| um_burst | I | Indicates this operation is a burst transfer. |
| um_size[1:0] | I | Data width for transfer (10-double word, 01-word, 00-byte, 11-invalid). Selects the size of the data transfer being done on the UMI. For 32-bit transactions all bits 31:0 will carry valid data. For 16-bit transactions only bits 15:0 will carry valid data. For 8-bit transactions only bits 7:0 will carry valid data. It is recommend for smaller than 32-bit transactions the remaining unused data bits are driven to 0 for a write operation. |
| um_ready | I | Data strobe indicates address and data ready. Should be driven high when valid address is provided for read and valid address and data is present for write. |
| um_ack | O | Acknowledge from master interface to indicate that it is ready for another operation. |
| um_retry | O | Asserted for one um_clk cycle when the UMI is busy to request that the master relinquish the bus and reissue the current transfer. A retry is issued when the following occurs:<br>1.The UMI gets a read transaction while its write FIFOs are not empty.<br>2.The UMI gets a write transfer while its write FIFOs are full.<br>3.The UMI receives a retry indication from the embedded system bus. |
| um_err | O | Bus error response is asserted for one clock cycle when um_ready is asserted. Indicates an internal system bus error. |
| um_irq | I | User logic master interface interrupt request. Active-high signal to indicate an interrupt to the system bus. This signal will map to the interrupt cause register USER_MSTR bit. |

## Locking the UMI

The system bus is a multi master bus and the um_lock signal will request ownership of the bus. In a multi master system bus application the UMI must be locked to guarantee uninterrupted multiple transactions through the UMI. Ownership of the bus is granted based on the priority of the master interface. If two masters request the bus at the same time the interface with higher priority will obtain the bus. Once the UMI has locked the bus the um_granted signal along with um_ack will go high indicating the interface is ready for a transaction. Um_granted is for debug only and should not be used in the user's design.

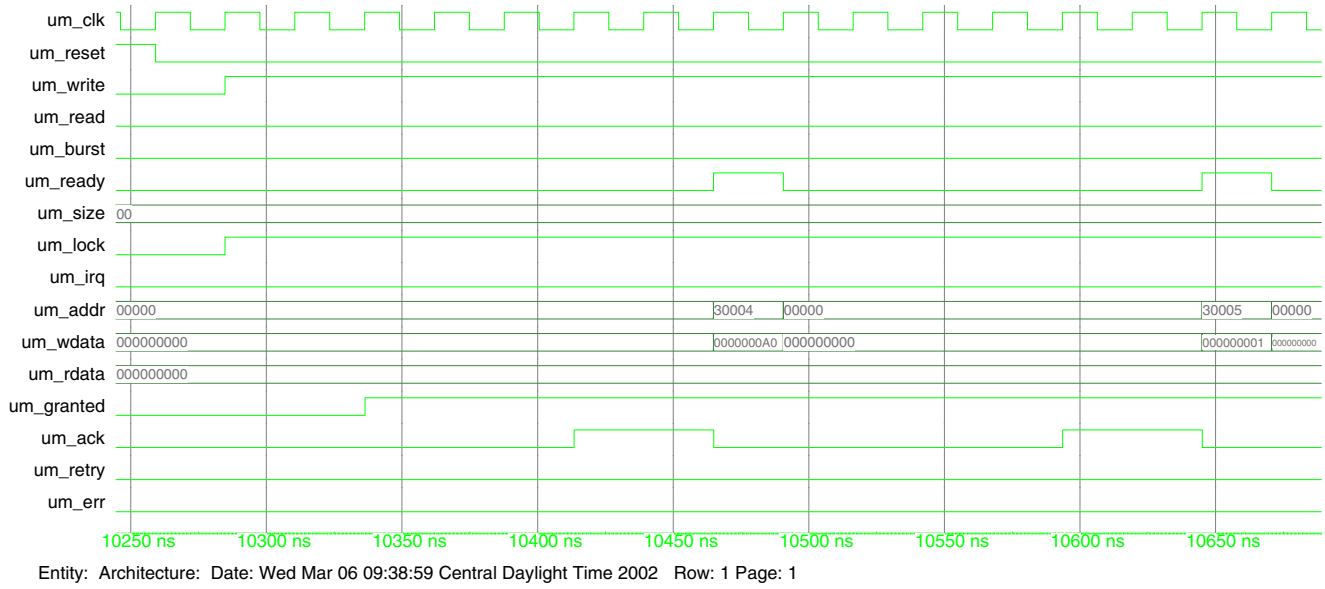
To release the lock on the system bus a um_ready pulse must be given after um_lock is driven low. The um_ready pulse will allow the system bus to sample um_lock and release the bus lock.

For a single UMI transaction um_lock is not required.

## UMI Single Access

A typical operation of a single access write transactions are as follows. For this description the um_lock signal will be used to lock the system bus and guarantee uninterrupted multiple transactions.

***Figure 9. Single Access Write from USER Master Interface***

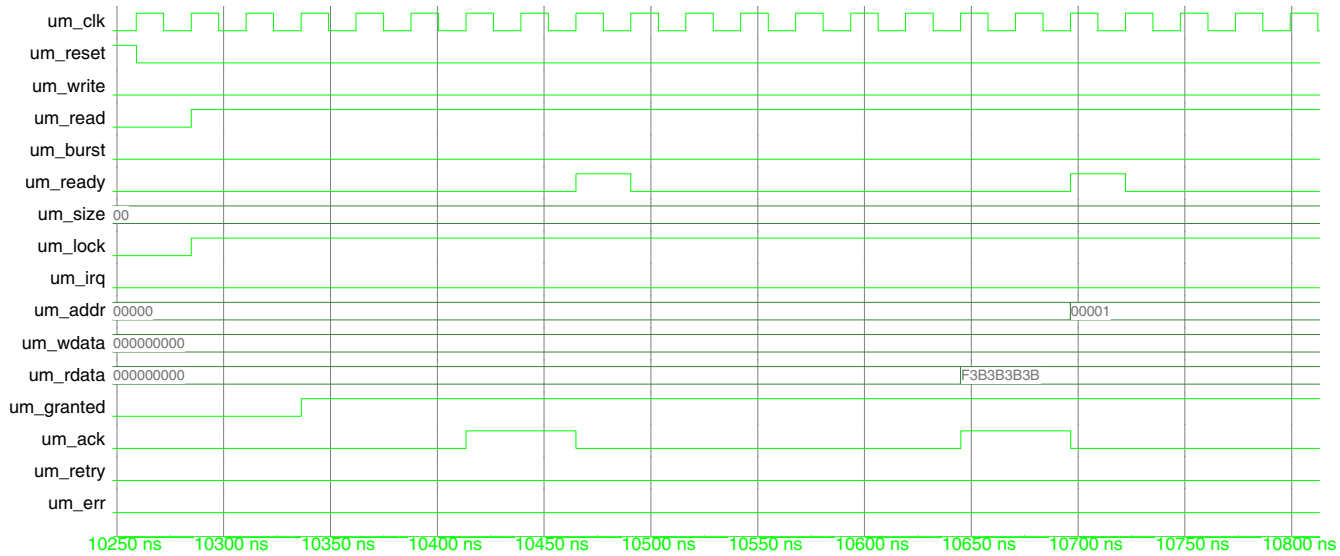



A single write access is initiated with the assertion of um_lock and um_write. This in turn requests ownership of the system bus. When granted ownership (um_granted), the UMI returns with um_ack. The user then asserts um_ready with valid data and address on um_addr and um_wdata. Um_ready is the gating signal for valid data and address and is to be asserted for every single access write transaction. When the write transaction is complete, um_ack is asserted by the UMI. Um_ack is a synchronous signal and stays asserted after the single write is completed.

Consecutive single writes may be performed with the assertion of um_ready along with new data and address. Um_write is to be asserted for the entire period of the transaction. System bus lock requests are not made for subsequent back-to-back write operations. Um_ack gets deasserted on the clock cycle after um_ready and gets asserted when the transaction is complete and ready for the next transaction.

A typical operation of a single access read transaction is as follows. Again the um_lock signal will be used to lock the system bus.

***Figure 10. Single Access Read from USER Master Interface***




A single access read is initiated with the assertion of um_lock and um_read. This in turn requests ownership of the system bus. When granted ownership (um_granted), the UMI returns with um_ack. The user then asserts um_ready with a valid read address on um_addr. Um_ready is the gating signal for valid address and is to be asserted for only one um_clk cycle for every single read transaction. When the read data is ready at the um_rdata ports, um_ack is asserted by the UMI. Um_ack is a synchronous signal and stays asserted after the single read is completed. Consecutive signal reads may be performed with the assertion of um_ready along with the new address. Um_read is to be asserted for the entire period of the transaction. System bus lock requests are not made for subsequent back-to-back write operations. On the next um_clk with um_ready high, um_ack is deasserted and is re-asserted when the transaction is complete and ready for the next transaction. Signal um_ready should only be high for 1 um_clk cycle.

## UMI Burst Access

Burst access is initiated by asserting the um_burst signal along with um_lock and um_write/um_read. The UMI handles bursts that are four beats deep. It employs a four beat deep FIFO that is 36 bits wide. It handles and generates burst writes and reads that are four deep regardless of the size of the bus chosen by um_size. All address and data mapping is retained across the master interface.

A typical burst write operation is as follows: a burst access write is initiated with the assertion of um_lock, um_burst and um_write. This in turn requests ownership of the system bus. When granted ownership (um_granted), the UMI returns with um_ack.

The user then asserts um_ready with valid data address on um_addr and um_wdata, respectively, for four consecutive cycles. There is no address FIFO in the UMI, so the address on um_addr qualified by the first um_ready phase is incremented internally for the next three data words on um_wdata. Internal incrementing of the address depends on the size specified by um_size. When the write transaction is complete, um_ack is asserted by the UMI. Signal um_ack is a synchronous signal and stays asserted after the burst write is completed. Consecutive burst writes may be performed with the assertion of um_ready along with new data and address. Signal um_write is to be asserted for the entire period of the transaction. System bus lock requests are not made for subsequent back-to-back write operations. On the next um_clk with um_ready high, um_ack is deasserted and is re-asserted when the transaction is complete and ready for the next transaction. Signal um_ready should only be high for 1 um_clk cycle.

A typical burst read operation is as follows: a burst access read is initiated with the assertion of um_lock, um_burst, and um_read. This in turn requests ownership of the system bus. When granted ownership (um_granted), the UMI returns with um_ack.

The user then asserts um_ready with valid address on um_addr. There is no address FIFO in the user master interface, so the address on um_addr is qualified when the um_ready phase is incremented internally for a total of four addresses. Internal incrementing of the address depends on the size specified by um_size. when the read transaction is complete, um_ack is asserted by the umi. um_ack is a synchronous signal and stays asserted for four clock cycles after the burst read is completed. in this case, the um_ack qualifies the availability of valid data on um_rdata. Consecutive burst reads may be performed with the assertion of um_ready along with a new address. Signal um_read is to be asserted for the entire period of the transaction. System bus lock requests are not made for subsequent back-to-back read operations. The valid data on um_rdata is available on the first four clocks of um_ack. Signal um_ack gets deasserted only on the clock cycle after um_ready. Signal um_ack get asserted again when a new transaction is complete and ready for the next transaction.

## User Slave Interface (USI)

The user logic slave interface includes the signals listed in Table 8.

***Table 8. User Logic Slave Interface Signals***

| Signal | Type | Description |
|---|---|---|
| us_clk | I | User slave interface clock. This clock will only clock the USI. A domain transfer will occur from the USI to the system bus (HCLK). The user slave interface and the system bus can be made synchronous by using the same clock for the us_clk and usr_clk and selecting User as the HCLK. A frequency preference on this signal will constrain all of the inputs and outputs of the USI. |
| us_reset | I | This active-high reset resets all of the controls in the user slave interface. This should be pulsed once before any transactions can occur to the USI. This is typically connected to the same reset as the GSR and pulsed at power-up. |
| us_wdata[35:0] | O | 36-bit write data from the system bus. This data bus is oriented with bits 35:32 used for parity while bits 31:0 are used for data. Parity is not calculated or checked by any of the peripherals on the system bus; it is only carried. The MSb and LSb will be based on the driving master. For 32-bit access bits (31:0) are used. For 16-bit access bits 15:0 are used. The same data on 15:0 will be present as well on 31:16 as well. For 8-bit access bits 7:0 are used and the same data will be present on d(31:24), d(23:16), and d(15:8) as well. |
| us_rdata[35:0] | I | 36-bit read data bus to system bus. This data bus is oriented with bits 35:32 used for parity while bits 31:0 are used for data. Parity is not calculated or checked by any of the peripherals on the system bus, it is only carried. The MSB and LSB must be selected based on the accepting master and application. For 32-bit access bits (31:0) are used. For smaller than 32-bit transfers the data must be replicated on all byte lanes. For 16-bit access bits 15:0 and 31:16 are used. For 8-bit access bits 7:0, 15:8, 23:16, and 31:24 are used with the same read data on all bytes. |
| us_addr[17:0] | O | This 18 bit address bus is provides the address where a slave transaction will operate. Bit 17 of the um_addr bus is the MSB while bit 0 is the LSB. The user only needs to decode the address bits that make up the application. Since the USI always resides at address offset 0x8000, bit 15 will always be 1 during a USI access. |
| us_wr | O | Indicates whether the current transaction is a read or a write. 1 indicates a write transaction and the us_wdata should be captured. 0 indicates a read transaction and data should be placed on us_rdata. |
| us_size[1:0] | O | Transfer size (10-double word, 10-word, 00-byte, 11-invalid). Indicates the size of the current transaction. For a 32-bit transaction all of the data bits 31:0 will be valid. For 16-bit transaction only bits 15:0 will be valid. For 8-bit data only bits 7:0 will be valid. For smaller than 32-bit transactions the remaining unused data bits should driven to 0 for a read operation. |
| us_burst | O | Transfer is a burst (high) or single beat (low). |
| us_err | I | Active-high error response from the user to the USI when data size is not consistent with us_size, or the address on the us_addr is out of range for the application. |
| us_ack | I | Active-high acknowledge for read operations. User drives us_ack high to terminate the transaction. Signal us_ack is passed through the system bus to eventually terminate the transaction from the driving master. |

*Table 8. User Logic Slave Interface Signals (Continued)*

| Signal | Type | Description |
|---|---|---|
| us_rdy | O | Active-high ready response from the USI for either read or write transactions. Signal us_rdy goes high when an address is ready on us_addr. |
| us_retry | I | Signal available for the user to drive when a transaction is not ready to be processed. This retry signal is then sent to the originating master interface. |
| us_irq | I | Active-high interrupt pin to indicate an interrupt to the system bus. This signal will map to the interrupt cause register USER_SLAVE bit (0x00010 bit 6). |

## User Slave Transfer Errors

The user logic slave interface responds with a transfer error to the system bus under the following circumstances. This is an internally generated error from the USI and is separate from the us_err signal. The resulting error to the master interface will be either a mpi_retry or um_retry for the MPI and UMI. For the FPSC this will be specific to the FPSC master interface and will be covered in the FPSC data sheet.

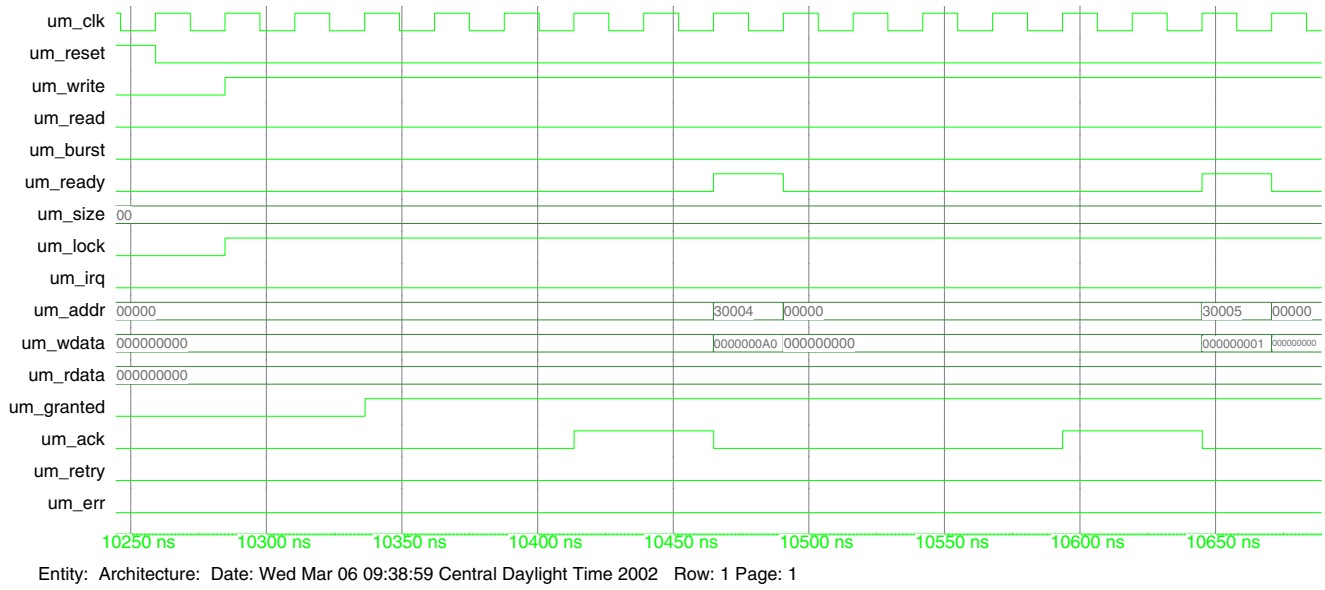
1. us_reset is high during the transaction
2. us_err is high during the transaction
3. The device is in bitstream configuration.
4. The address does not conform to the us_size specified. The setting for us_size will dictate the granularity of the address available from the USI. If us_size is selected for 8-bit mode, then all address can be selected. If us_size is selected for 16-bit (word) mode, then only addresses on word boundaries can be selected.

For example address 0x0, 0x2, 0x4, etc. are valid addresses. 0x1, 0x3, 0x5, etc. are not valid addresses for word accesses. The same holds true for 32-bit (double word) mode; 0x0, 0x4, 0x8, etc. are valid while 0x1, 0x2, 0x3, 0x5, 0x6, 0x7, 0x9, etc. are not valid.

## USI Single Access

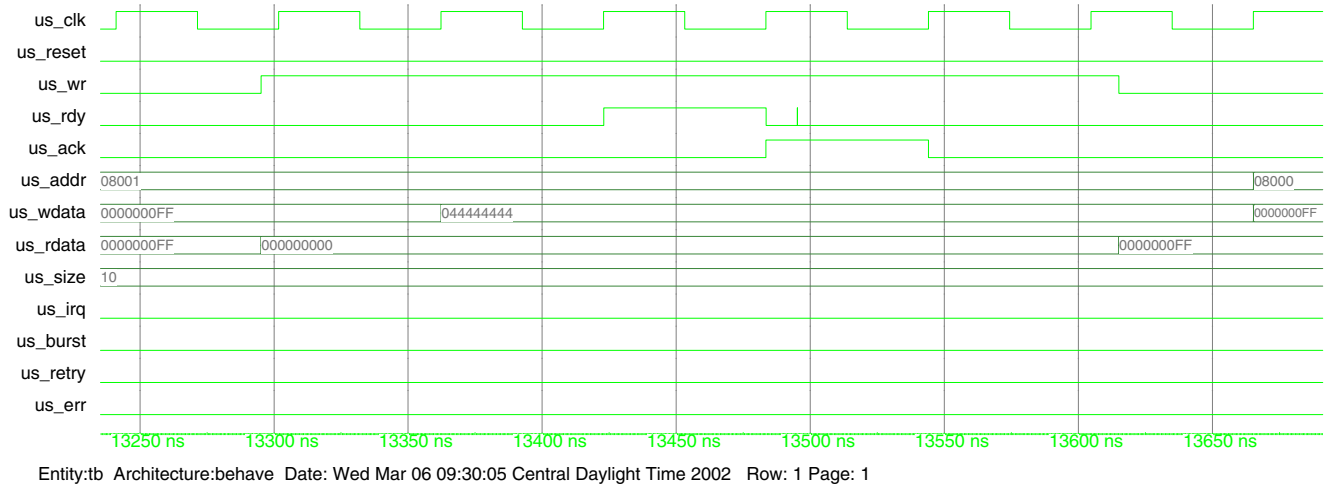
A typical operation of a single access write transaction is as follows. The us_wr signal goes high to indicate that a write is taking place. The synchronous signal us_rdy signal indicates the availability of valid address and data on the us_addr and us_wdata ports, respectively. The USI inserts additional wait states until the us_ack signal is asserted by the user. For write operations, if the data received by the slave interface can be accepted by the user slave in one cycle, it is acceptable to leave us_ack asserted continuously during us_wr (write operation). If the USI takes more than one us_clk cycle to terminate the transaction then us_ack should be driven high when the transaction is complete.

***Figure 11. Single Access Write at USER Slave Interface***




A typical operation of a single access read transaction is as follows. the us_wr signal is low to indicate that a read is taking place. The synchronous us_rdy signal indicates the availability of valid address on the us_addr. The user must then respond with read data on the us_rdata bus and assert us_ack. The USI inserts additional wait states until the us_ack signal is asserted by the user.

***Figure 12. Single Access Read at USI***




## USI Burst Access

A burst access to the USI is indicated by the us_burst signal. However, burst accesses are treated similar to single accesses. There is no FIFO in the USI and thus all burst accesses are handled as back-to-back signal accesses with wait states. It is possible for the slave to handle bursts of user design defined lengths. During a burst access the us_addr bus will increment on each us_clk during the burst transfer.

## FPSC Master/Slave Interface

The FPSC system bus interface can be either a master or a slave interface. The details and protocol of the FPSC interface are specific to the FPSC being used. The connections from the system bus to the FPSC block are main-

tained by the FPSC software and are fixed in the actual device. Therefore this interface is not covered in this document. The FPSC template from the FPSC Configuration Wizard makes all of the HDL connections for the user.

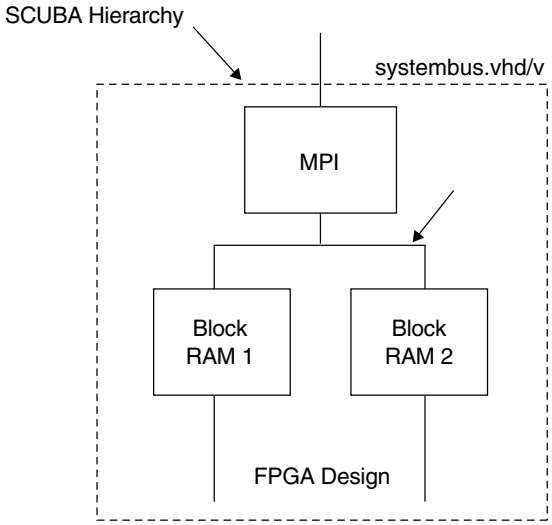
The FPSC memory map resides at address offset 0x30000. When referencing the FPSC data sheet, all addresses begin at base address 0x30000.

## System Bus Embedded Block RAMs

Embedded block RAMs (EBR) in an ORCA Series 4 device are quad-port 512-bit x 18-bit static random access memories (SRAM). The embedded memory blocks may be configured in either quad port (two user write ports and two user read ports), or system bus mode (one pair of read/write ports available to FPGA logic, one pair of read/write ports dedicated to embedded system bus access). Prior to device configuration, all of the embedded memory blocks operate in system bus mode to facilitate bitstream or microprocessor initialization of the memory contents. After configuration, embedded memory blocks are not accessible from the system bus unless the user specifically includes the system bus element and associated block memories in system bus mode in their FPGA design.

If an EBR is used on the system bus the EBR cannot be used in any other mode (FIFO, CAM, or Multiplier) in the FPGA design. Figure 13 shows how system bus EBRs are connected to the system bus and represented in SCUBA. One of the read/write ports of the quad port EBR is connected to the system bus and the other is available for the FPGA design. The read/write ports for the user will be found on the system bus element when creating the system bus and EBR using SCUBA. EBR memories that do not require a connection to the system bus are created separately from the system bus and have no connection in the HDL to the system bus.

***Figure 13. SCUBA HDL for System Bus EBRs***




## EBR Memory Mapping

The embedded system bus provides two memory spaces for accessing the contents of the embedded memory blocks. The first (0x10000 to 0x17FFF) provides access to embedded memory blocks along the top edge of the device. The second (0x20000 to 0x27FFF) provides access to embedded memory blocks along the bottom edge of the device. Adjacent memory blocks are paired to form a single 512-bit x 32-bit data structure that is accessed via the embedded system bus. Table 9 illustrates the address mapping of the first double word of each embedded memory block as well as device location, availability and SCUBA location designator.

*Table 9. Memory Block Address Mapping*

| Address | Bits | RAM Block | Device Location | SCUBA Location Name | Device |
|---|---|---|---|---|---|
| 0x10001:0x10000 | [0:15] | BLOCK0 [0:15] | Top Left | 1024_0 | 4e2,4e4,4e6 |
| 0x10003:0x10002 | [16:31] | BLOCK1 [0:15] | Top | 1024_0 | 4e2,4e4,4e6 |
| 0x10801:0x10800 | [0:15] | BLOCK2 [0:15] | Top | 1024_1 | 4e2,4e4,4e6 |
| 0x10803:0x10802 | [16:31] | BLOCK3 [0:15] | Top | 1024_1 | 4e2,4e4,4e6 |
| 0x11001:0x11000 | [0:15] | BLOCK4 [0:15] | Top | 1024_2 | 4e4,4e6 |
| 0x11003:0x11002 | [16:31] | BLOCK5 [0:15] | Top | 1024_2 | 4e4,4e6 |
| 0x11801:0x11800 | [0:15] | BLOCK6 [0:15] | Top | 1024_3 | 4e6 |
| 0x11803:0x11802 | [16:31] | BLOCK7 [0:15] | Top Right | 1024_3 | 4e6 |
| 0x20001:0x20000 | [0:15] | BLOCK32 [0:15] | Bottom Left | 1024_16 | 4e2,4e4,4e6 |
| 0x20003:0x20002 | [16:31] | BLOCK33 [0:15] | Bottom | 1024_16 | 4e2,4e4,4e6 |
| 0x20801:0x20800 | [0:15] | BLOCK34 [0:15] | Bottom | 1024_17 | 4e2,4e4,4e6 |
| 0x20803:0x20802 | [16:31] | BLOCK35 [0:15] | Bottom | 1024_17 | 4e2,4e4,4e6 |
| 0x21001:0x21000 | [0:15] | BLOCK36 [0:15] | Bottom | 1024_18 | 4e4,4e6 |
| 0x21003:0x21002 | [16:31] | BLOCK37 [0:15] | Bottom | 1024_18 | 4e4,4e6 |
| 0x21801:0x21800 | [0:15] | BLOCK38 [0:15] | Bottom | 1024_19 | 4e6 |
| 0x21803:0x21802 | [16:31] | BLOCK39 [0:15] | Bottom Right | 1024_19 | 4e6 |

If enabled, parity for each byte is provided via the data bus parity bits. Parity is stored in the most significant bit of the byte mapped 32k addresses above the related data byte. This space is reserved in the memory map shown in Table 9. If parity is enabled, the parity bits are recombined with the data to form 36-bit data on the embedded system bus, and 18-bit data on the user ports of the embedded memory block.

## EBR User Ports

When using an EBR in the system bus one read/write port is connected to the system bus while the other port is available to the user FPGA design. Table 10 shows the user ports and their description. These ports will exist on the system bus element if a system bus block RAM is utilized.

*Table 10. Block RAM User Logic Interface Signals*

| Signal | Type | Description |
|---|---|---|
| Waddr[N:0] | I | Write port address (N: 8 = 512, 9 = 1024). |
| Din[17:0] | I | Write port data. Parity is carried on 17:16, data on 15:0. Parity is mapped that bit 17 is used for 15:8 and bit 16 is used for 7:0. |
| BW[1:0] | I | Active-high, byte lane write enables.<br>00 - Both byte lanes are disabled<br>01 - Byte lane with bits 16,7:0 is enabled<br>10 - Byte lane with bits 17,15:8 is enabled<br>11 - Both byte lanes are enabled for 18-bit write |
| WREN | I | Active-high, write enable. |
| WCLK | I | Write port clock input. |
| RADDR[N:0] | I | Read port address (N: 8 = 512, 9 = 1024). |
| RDEN | I | Active-high, read enable. |
| RCLK | I | Read port clock input. |
| DOUT[17:0] | O | Read port data. Parity is carried on 17:16, data on 15:0. Parity is mapped that bit 17 is used for 15:8 and bit 16 is used for 7:0. |

### Creating the System Bus in HDL

The system bus is a very large element that is contained in the ORCA Foundry VHDL and Verilog libraries for Series 4 ORCA devices. In order to facilitate the use of this large block, SCUBA is used to create a custom system bus for a user's FPGA design and application. The SCUBA Wizard will guide the user through the different interfaces of the system bus and allow them to tailor the system bus for the specific application. SCUBA will create an HDL file output that includes a module with only the system bus peripheral interfaces specified in SCUBA. This new module instantiates the library element for the system bus and disables all of the unused interfaces.

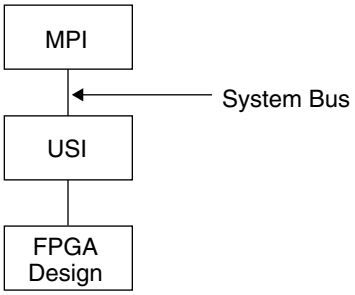
SCUBA is used to create the system bus and all connected EBRs. System Bus EBRs are accessed as 512-bit x 36-bit on the system bus. Designers have the choice of accessing the EBR as a single 1024-bit x 18-bit or two 512-bit x 18-bit interfaces. A checkbox for arbiter mode is available in SCUBA to use the arbiter to perform arbitration between two EBRs. SCUBA also allows the designer to choose whether to use EBR input and output registers. These registers pipeline the address and/or data across the FPGA boundary in and out of the EBR. A memory file is also available to initialize the EBR to any value. This memory file is used to initialize the EBR to values other than 0x0. Be default the EBR will be initialized to 0x0 at power-up. However, when utilizing the EBR memories using the MPI to perform bitstream configuration, the MPI will be required to initialize the EBR.

## Applications of the System Bus

### MPI and User Slave

A common application using the system bus involves the use of the MPI and user slave interface to create soft control and status registers inside of the FPGA design. Using the system bus with a MPI and a user slave interface can be accomplished with an 8, 16, and 32-bit data bus and using up to all 18 bits of address in the FPGA design. As an example application we will use an 8-bit data bus using two address bits to create four registers in the FPGA. Two registers will be used as R/W control registers and two registers will be used as read only status registers. We will also use an interrupt generated from the FPGA design to signal the processor via the MPI. Figure 14 shows a block level diagram of the FPGA design.

***Figure 14. MPI and User Slave Application Diagram***




First the designer needs to create the system bus module in HDL. This is done using the SCUBA Wizard.

1. Open the SCUBA Wizard and select the or4e00 architecture.
2. On the next page select System Bus from the module pull down list and specify the HDL source, output directory, and HDL destination (synthesis or simulation).
3. The next page is specific to the MPI. Enable the MPI and select an 8-bit data bus with parity.
4. On the next page select the checkbox to generate an FPGA slave interface.
5. Skip over the block RAM page since system bus block RAMs are not used in this application.
6. The final page is for system bus general attributes. Select the system bus clock to be the MPI. This will allow the MPI_CLK to drive the system bus clock (HCLK). Since this is a single master design the multi master options of priority are not applicable.

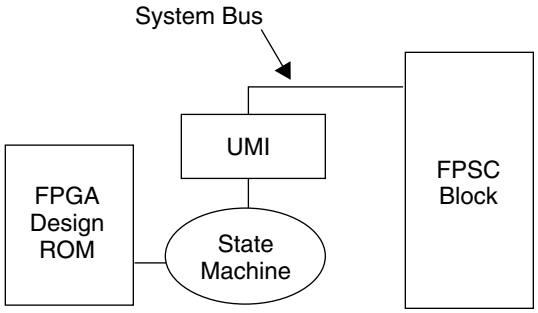
Once the system bus is created from SCUBA it is now time to connect the element up to the FPGA design. The MPI ports on the system bus will all be routed to the dedicated pins on the ORCA device. These pins do not have to be located to pads since the software knows exactly where they are to be placed. All of the pins on the user slave interface are internal. The user slave interface can be connected as follows:

- The us_clk drives the user slave interface. For this simple example the us_clk can be driven as well by the MPI_CLK.
- The us_reset can be tied to an active high reset in the design. Typically this is connected to the same reset as the GSR. At power-up, the GSR is pulsed to reset all of the FFs in the FPGA. This is an active low signal so the us_reset would need to be inverted before connecting to the user slave interface.
- This application will not support burst mode so us_burst will be left unconnected.
- The size of the data transaction will be fixed at eight bits so we can leave us_size open. If the design was to support multiple data sizes then this port would need to be utilized in this application.
- This application uses four soft registers so we only need to use two bits of the address bus. Using the least significant bits of the address bus us_addr(1:0) and leaving the rest unconnected we can use addresses 0x8000, 0x8001, 0x8002, and 0x8003. Internally we will only be decoding two address bits so off of the us_addr(1:0) we will see 0x00, 0x01, 0x02, and 0x03. This is valid since a us_rdy will only go high when the user slave interface is selected with base address 0x8000 from the master. Internally we simply leave us_addr(17:2) unconnected.
- We will assume that our FPGA soft registers can handle a byte of data and 2-bit address in one clock cycle. This means that as us_rdy goes high from the USI the data and address can be latched into the registers and out of the registers in one clock cycle. Since this is the case we can have us_rdy be terminated with a us_ack on the next us_clk clock cycle. Us_retry can also be connected to logic 0 since all transactions will terminate immediately.
- Us_err will not be used for this simple design.
- The us_wr can be used as an active high write enable to the control registers.
- Us_rdata should be a tristated bus connected to the outputs of all of the soft registers. The address decode for each register should deactivate the tristate control when the register is selected. Since we are using only 8-bit registers the same data must be replicated on all byte lanes of us_rdata. We are using the MPI driven by the PowerPC so the MSB would be us_rdata(0).
- Us_wdata should be connected to both of the R/W control registers. The us_wr along with the address decode should select when a register accepts data on a write from us_wdata. Since we are using only 8-bit registers only us_wdata(7:0) should be used. Parity for this byte can be found on us_wdata(32). We are using the MPI driven by the PowerPC so the MSB would be us_wdata(0).
- Us_irq is used for an interrupt request from the user slave. All of the bits in the status registers could be OR'd together. When a single bit goes high in one of these status registers, the us_irq signal would indicate to the interrupt cause register that it had an interrupt. The MPI interrupt enable register would need to be enabled to accept interrupts from the USER_SLAVE (0x00010 bit 6). An interrupt from the us_irq indicating that a status bit is set would then be seen on the MPI_IRQ pin to inform the processor.

## User Master and FPSC

This sample application for the user master interface uses an FPGA ROM to provision FPSC registers. In this example we will use an ort8850 and use the user master interface to provision the ort8850 core registers. The ort8850 has approximately a dozen registers that control the SERDES channels and major modes of the device. Users typically have to write these registers once after power-up using an external processor through the MPI. To simplify the board design and to not require a processor the registers in the ort8850 can be programmed directly from the FPGA. Using the user master interface and a ROM inside the FPGA, a state machine can be designed to run through the ROM table and write data values to addresses inside the FPSC to set up the SERDES and control registers. Figure 15 shows a block-level diagram of the FPGA design.

***Figure 15. User Master and FPSC Application Diagram***




First, the designer needs to create the system bus module in HDL. This is done using the SCUBA Wizard.

1. Open SCUBA and select the or4e00 architecture.
2. On the next page select System Bus from the module pull down list and specify the HDL source, output directory, and HDL destination (synthesis or simulation).
3. The next page is specific to the MPI. We will not be using an MPI in this application.
4. On the next page select the checkbox to generate an FPGA master interface and FPSC interface.
5. Skip over the block RAM page since system bus block RAMs are not used in this application.
6. The final page is for system bus general attributes. Select the system bus clock to be the User.

Once the system bus is created from SCUBA it is now time to connect the element up to the FPGA design. The FPSC connection to the system bus is fixed in the silicon device. In the HDL the connections from the FPSC to the system bus are provided in the FPSC HDL template generated by the FPSC Configuration Wizard. These connections should always be used. The user master interface will require user design to implement our desired ROM function.

First the ROM should be created using SCUBA. The ROM will contain the address and data of each of the writes to the ort8850 core. All of the addresses we will access inside the ort8850 be at address offset 0x30000 through 0x300FF. To limit the size of the ROM we only need to store 8 bits of the address (0x00 to 0xFF). We also need to store 8 bits of data since the ort8850 has 8-bit registers inside the core. So the ROM will need to be 16 bits wide, 8 bits for the address and 8 bits for the data. The depth of the ROM will be determined by how many writes we need to perform. For our application, we will use 10 writes (1 per channel (8) and 2 mode writes) which will require 4 address lines. So the final ROM size will be a 4x16 and will fit into 4 PFUs (32x4 ROM max in each PFU). SCUBA provides an input for a memory initialization file. This file should contain the address/data pairs for each of the 10 writes.

Now that we have the required elements of the design it is time to create the state machine to perform the writes and connect this up to the user master interface. The state machine will simply start by obtaining the lock from the system bus, issue transactions using um_ready and um_ack incrementing through the ROM on each write, and finally releasing the lock on the system bus. Below are the connections and descriptions of each port:

- The um_clk and the user_clk signals should be driven by the same clock. We have selected the USER clock to drive the system bus clock and this will be expected on the user_clk port. The driver for this clock is not important and could come from off chip.
- We will not be doing a burst transaction so um_burst can be connected to logic 0.
- The size of the data transaction will be fixed at 8-bits so we can connect um_size to “00”. The ort8850 only contains 8-bit registers.
- This state machine will only perform writes through the UMI so the um_rdata bus can be left unconnected and um_read connected to logic 0.

- On the 16-bit data bus from the ROM we have address and data. Address is on data(15:8) and data is on data(7:0). On the um_wdata bus we need to place the data on bits 7:0. Um_wdata is a 36-bit bus so the remaining bits of 35:8 can be connected to logic 0. The ort8850 does not check parity on the 8-bit register transactions.
- The address from the ROM is on data(15:8). The um_addr needs to be prefixed with 0x300 along with the ROM data. (um_addr <= “1100000000” & data(15 downto 8))
- The um_write can be connected to logic 1 since we will only be doing writes.
- The first state in the state machine should be to drive um_lock high and wait for um_ack. When um_ack goes high the UMI is ready for transactions.
- The um_granted signal is only for debug and should not be connected.
- If um_retry from the UMI goes high the state machine should repeat the current transaction.
- If um_err from the UMI goes high there is a serious internal problem.
- The state machine should drive um_ready and wait for um_acks for each transaction.
- There are no interrupts in this design so the um_irq signal is connected to logic 0.

## Series 4 FPGA System Bus Memory Map

Table 11 describes the memory map for the internal registers on the system bus. Memory maps for FPSC devices are found in the device specific data sheets.

This memory map is structured with bit 0 as the MSb to match the PowerPC bus orientation. All power-up default values are 0x0 unless otherwise specified.

***Table 11. Memory Map***

| Absolute Address | Type | Bits | | Description |
|---|---|---|---|---|
| 0x00000 - 0x00003 | R | 0:31 | | 32-Bit Manufacturer and Device ID Code: The manufacturer identification code register contains a unique code for each FPGA and FPSC device in the ORCA Series 4 family. The code is comprised of a leading logic 1 [0 = MSB], the manufacturer code [1:11], family code[12:17], device code[18:27], and version code[28:31].<br><br>The manufacturer code is: [1:11] = 10111000000. The family code for series-4 is: [12:17] = 000100. The device code and version code will vary for each unique FPGA/FPSC in the Series 4 family.<br><br>OR4E02 = DC 01 01 C0<br>OR4E04 = DC 01 02 80<br>OR4E06 = DC 01 03 00<br><br>Device ID codes for devices not listed can be found in the device specific data sheet.<br><br>The system bus simulation model will always show a OR4E06 device ID. The device ID is part of the device silicon, the simulation model was built from a OR4E06. |
| 0x00004 - 0x00007 | R/W | 0:31 | | 32-bit scratchpad register. Free register used for debugging purposes. |
| 0x00008 | R/W | 0:1 | RDBK_SIZE | These two bits specify the number of valid bytes in the read back data register (0x00018) during a readback operation.<br><br>00 - no bytes<br>01 - 2 bytes<br>10 - 1 byte<br>11 - 3 bytes |
| | R/W | 2 | MPI_USR_ENABLE | Active-high, enables the MPI interface to the user. Used to keep the MPI available during a reconfiguration process. During reconfiguration the mode pins are not sampled to check for MPC mode. Set this bit before reconfiguration to keep MPI available. |
| | R/W | 3 | REPEAT_RDBK | Active-high, inhibits autoincrement of the readback address (0x00014) when the readback data register (0x00018) is read. |
| | R/W | 4 | SYS_DAISY | Enables bitstream daisy chaining when configuring more than 1 device via MPI. |
| | R/W | 5 | Reserved | Do not write a ‘1’ to this location |
| | R/W | 6 | Reserved | Do not write a ‘1’ to this location |
| | R/W | 7 | Reserved | Do not write a ‘1’ to this location |
| 0x00009 | R/W | 0 | LOCK_FPSC | Indicates that the internal system bus is currently locked by the FPSC interface. Used for multicycle operations that must retain bus ownership. Only the FPSC can write this bit. |
| | R/W | 1 | LOCK_USER | Indicates that the internal system bus is currently locked by the user logic interface. Used for multicycle operations that must retain bus ownership. Only the User Master can write this bit. |
| | R/W | 2 | LOCK_MPI | Active-high, locks the internal system bus for use by the MPI. Used for multicycle operations that must retain bus ownership. Only the MPI can write this bit. |
| | R/W | 3 | PGM_FPSC | Indicates PGRM bit set by FPSC logic. |
| | R/W | 4 | PGRM_USER | Indicates PGRM bit set by user logic. |
| | R/W | 5 | PGRM_MPI | Active-high, forces reconfiguration of the FPGA logic using the mode specified on the MODE pins during initial power-up. |
| | R/W | 6 | SYS_RD_CFG | Active-high, initializes the readback logic. |
| | R/W | 7 | SYS_GSR | Active-high, asserts the global set/reset. |
| 0x0000A - 0x0000B | — | — | — | Unused |

*Table 11. Memory Map (Continued)*

| Absolute Address | Type | Bits | | Description |
|---|---|---|---|---|
| 0x0000C | R | 0:1 | ERR_FLAG | In the event of an error during device configuration these bits will indicate the nature of the error.<br>00 - no error<br>01 - checksum error indicates one or more corrupt bits in bitstream<br>10 - device ID error indicates bitstream does not match target<br>11 - framing/alignment error. data bits may be reversed. |
| | R | 2 | INIT | Reflects the state of the INIT I/O pad. |
| | R | 3 | DONE | Reflects the state of the DONE I/O pad. |
| | — | 4 | — | Unused |
| | R | 5:6 | CFG_ERR | If an internal system bus error occurs during configuration, the error is captured in these two bits. The address of the first error is captured in the bus error address register (0x00024).<br>00 - no errors<br>01 - invalid response code<br>10 - one error occurred<br>11 - multiple errors occurred |
| | R | 7 | RDBK_AOR | Active-high readback address out of range error alarm. |
| 0x0000D | R | 0:1 | CFG_SIZE | Indicates the number of active bytes in the configuration data register during configuration writes.<br>00 - no bytes<br>01 - 2 bytes<br>10 - 1 byte<br>11 - 3 bytes |
| | R | 2 | RAM_BIT_ERR | Active-high indicates an error occurred while trying to initialize the contents of a block RAM. |
| | R | 3 | FPSC_BIT_ERR | Active-high indicates an error occurred while trying to initialize configuration bits in the FPSC. |
| | — | 4:5 | — | Unused |
| | R | 6 | RDBK_RDY | Active-high indicates that data is pending in the readback data register (0x00018). |
| | R | 7 | CFG_ACK | Active-high indicates that the configuration logic is ready for data to be written into the configuration data register (0x0001C). |
| 0x0000E - 0x0000F | | | | Unused |
| 0x00010 | Interrupt Cause Register | | | |
| | R | 0 | FPSC_MSTR | Active-high, interrupt request from the FPSC master interface. Write 1 to clear this bit. |
| | R | 1 | FPSC_SLAVE | Active-high, interrupt request from the FPSC slave interface. Write 1 to clear this bit. |
| | R | 2 | MPI_IRQ | Active-high, interrupt request from the MPI. Write 1 to clear this bit. |
| | R | 3 | CFG_ERR | Active-high, indicates that the ERR_FLAG bits in the status register were changed during device configuration. |
| | R | 4 | CFG_IRQ | Active-high, interrupt request from the configuration logic requesting another word/byte of data. Write 1 to clear this bit. |
| | R | 5 | USER_MSTR | Active-high, interrupt request from the User Master interface (um_irq). Write 1 to clear this bit. |
| | R | 6 | USER_SLAVE | Active-high, interrupt request from the User Slave interface (us_irq). Write 1 to clear this bit |
| | R | 7 | USER_IRQ | Active-high, interrupt request from the usr_irq_general signal. Write 1 to clear this bit. |

*Table 11. Memory Map (Continued)*

| Absolute Address | Type | Bits | | Description |
|---|---|---|---|---|
| 0x00011 | R/W | 0 | EN_IRQ_FPSC | Logic 1 enables the FPSC master interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 1 | | Logic 1 enables the FPSC slave interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 2 | | Logic 1 enables the MPI interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 3 | | Logic 1 enables the CFG_ERR interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 4 | | Logic 1 enables the CFG_IRQ master interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 5 | | Logic 1 enables the USER_MSTR interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 6 | | Logic 1 enables the USER_SLAVE interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| | R/W | 7 | | Logic 1 enables the USER_IRQ interrupt bit from address 0x10 to generate an interrupt to the FPSC. |
| 0x00012 | R/W | 0 | EN_IRQ_USER | Logic 1 enables the FPSC master interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 1 | | Logic 1 enables the FPSC slave interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 2 | | Logic 1 enables the MPI interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 3 | | Logic 1 enables the CFG_ERR interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 4 | | Logic 1 enables the CFG_IRQ master interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 5 | | Logic 1 enables the USER_MSTR interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 6 | | Logic 1 enables the USER_SLAVE interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| | R/W | 7 | | Logic 1 enables the USER_IRQ interrupt bit from address 0x10 to generate an interrupt to the user_irq pin. |
| 0x00013 | R/W | 0 | EN_IRQ_MPI | Logic 1 enables the FPSC master interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 1 | | Logic 1 enables the FPSC slave interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 2 | | Logic 1 enables the MPI interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 3 | | Logic 1 enables the CFG_ERR interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 4 | | Logic 1 enables the CFG_IRQ master interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 5 | | Logic 1 enables the USER_MSTR interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 6 | | Logic 1 enables the USER_SLAVE interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| | R/W | 7 | | Logic 1 enables the USER_IRQ interrupt bit from address 0x10 to generate an interrupt to the mpi_irq pin. |
| 0x00014 - 0x00017 | R/W | 18:31 | | Configuration memory readback address register (14 bits) |
| 0x00018 - 0x0001B | R/W | 0:31 | | Configuration memory readback data register |
| 0x0001C - 0x0001F | R/W | 0:31 | | Configuration data register |

*Table 11. Memory Map (Continued)*

| Absolute Address | Type | Bits | | Description |
|---|---|---|---|---|
| 0x00020 - 0x00023 | | 0:31 | | Reserved |
| 0x00024 - 0x00027 | R | 0:31 | | Bus error address register contains the address of the first configuration error. Indicated by the CFG_ERR bits of register 0x0000C. |
| 0x00028 - 0x0002B | R | 0:31 | | Interrupt vector #1 predefined by configuration bitstream |
| 0x0002C - 0x00029 | R | 0:31 | | Interrupt vector #2 predefined by configuration bitstream |
| 0x00030 - 0x00033 | R | 0:31 | | Interrupt vector #3 predefined by configuration bitstream |
| 0x00034 - 0x00037 | R | 0:31 | | Interrupt vector #4 predefined by configuration bitstream |
| 0x00038 - 0x0003B | R | 0:31 | | Interrupt vector #5 predefined by configuration bitstream |
| 0x0003C - 0x00039 | R | 0:31 | | Interrupt vector #6 predefined by configuration bitstream |
| 0x00040 - 0x00043 | R/W | 0:31 | | Top Left PPLL control register. For details, see the PLL_CONTROL register description. |
| 0x00044 - 0x00047 | R/W | 0:31 | | Top Left HPLL control register. For details, see the PLL_CONTROL register description. |
| 0x00048 - 0x0004B | R/W | 0:31 | | Top Right PPLL control register. For details, see the PLL_CONTROL register description. |
| 0x0004C - 0x0004E | R/W | 0:31 | | PLL1. This register is controlled by design characteristics. Do not change this value. |
| 0x00050 - 0x00053 | R/W | 0:31 | | Bottom Left PPLL control register. For details, see the PLL_CONTROL register description. |
| 0x00054 - 0x00057 | R/W | 0:31 | | Bottom Left HPLL control register. For details, see the PLL_CONTROL register description. |
| 0x00058 - 0x0005B | R/W | 0:31 | | Bottom Right PPLL control register. For details, see the PLL_CONTROL register description. |
| 0x0005C - 0x0005E | R/W | 0:31 | | PLL2. This register is controlled by design characteristics. Do not change this value. |
| | R/W | | PLL_CONTROL | This section describes each bit in the PLL control register. For smaller than 32-bit writes, the software must write to all bits of the register starting with bits 0:7. The PLL control register will be initialized by the PLL programming contained in the configuration bitstream. The PLL_CONTROL register is used to modify the PLL at run time. Changes to any of the DIVs will cause the PLL to lose lock and reacquire. |
| | R/W | 0 | PLL_CONTROL | Controlled by design routing. Do not change this value. |
| | R/W | 1 | PLL_CONTROL | Active-low PLL reset. This reset will reset the entire register to 0x00000000. |
| | R/W | 2 | PLL_CONTROL | Active-high PLL enable. |
| | R/W | 3:5 | PLL_CONTROL | PLL Div0 divide ratio = N+1<br>000 - Div0 = 1<br>001 - Div0 = 5<br>010 - Div0 = 3<br>011 - Div0 = 7<br>100 - Div0 = 2<br>101 - Div0 = 6<br>110 - Div0 = 4<br>111 - Div0 = 8 |

***Table 11. Memory Map (Continued)***

| Absolute Address | Type | Bits | | Description |
|---|---|---|---|---|
| | R/W | 6:8 | PLL_CONTROL | PLL Div1 divide ratio = N+1<br>000 - Div1 = 1<br>001 - Div1 = 5<br>010 - Div1 = 3<br>011 - Div1 = 7<br>100 - Div1 = 2<br>101 - Div1 = 6<br>110 - Div1 = 4<br>111 - Div1 = 8 |
| | R/W | 9:11 | PLL_CONTROL | PLL Div2 divide ratio = N+1<br>000 - Div2 = 1<br>001 - Div2 = 5<br>010 - Div2 = 3<br>011 - Div2 = 7<br>100 - Div2 = 2<br>101 - Div2 = 6<br>110 - Div2 = 4<br>111 - Div2 = 8 |
| | R/W | 12:14 | PLL_CONTROL | PLL Div3 divide ratio = N+1<br>000 - Div3 = 1<br>001 - Div3 = 5<br>010 - Div3 = 3<br>011 - Div3 = 7<br>100 - Div3 = 2<br>101 - Div3 = 6<br>110 - Div3 = 4<br>111 - Div3 = 8 |
| | R/W | 15:16 | PLL_CONTROL | Controlled by design routing. Do not change this value. |
| | R/W | 17:18 | PLL_CONTROL | Controlled by design routing. Do not change this value |
| | R/W | 19:20 | PLL_CONTROL | PLL output mode for nclk. Refer to technical note number TN1014, *ORCA Series 4 FPGA PLL Elements* for more information on each of these modes.<br>00 - Bypass<br>01 - Phase Shift<br>10 - Duty Cycle<br>11 - Delay |
| | R/W | 21:22 | PLL_CONTROL | PLL output mode for mclk. Refer to technical note number TN1014, *ORCA Series 4 FPGA PLL Elements* for more information on each of these modes.<br>00 - Bypass<br>01 - Phase Shift<br>10 - Duty Cycle<br>11 - Delay |
| | R/W | 23:25 | PLL_CONTROL | VCOTAP Setting bits. The VCOTAP controls different attributes in the different modes of the PLL. Refer to technical note number TN1014, *ORCA Series 4 FPGA PLL Elements* for more information on the values of VCOTAP.<br>000 - VCOTAP=0<br>001 - VCOTAP=4<br>010 - VCOTAP=2<br>011 - VCOTAP=6<br>100 - VCOTAP=1<br>101 - VCOTAP=5<br>110 - VCOTAP=3<br>111 - VCOTAP=7 |
| | R/W | 26:31 | PLL_CONTROL | Unused |

# Appendix A

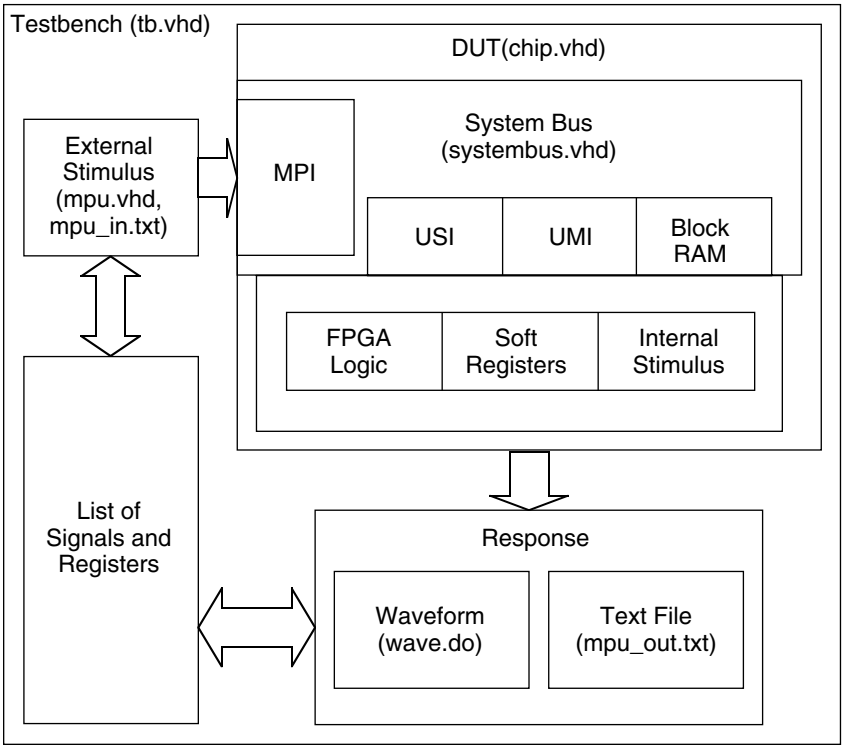
***Table 12. PowerPC and ORCA Pin Connections***

| PPC Address | ORCA Address Pin | PPC Data Pin | ORCA Data Pin |
|---|---|---|---|
| 31 | 17 | 0 | 0 |
| 30 | 16 | 1 | 1 |
| 29 | 15 | 2 | 2 |
| 28 | 14 | 3 | 3 |
| 27 | 13 | 4 | 4 |
| 26 | 12 | 5 | 5 |
| 25 | 11 | 6 | 6 |
| 24 | 10 | 7 | 7 |
| 23 | 9 | 8 | 8 |
| 22 | 8 | 9 | 9 |
| 21 | 7 | 10 | 10 |
| 20 | 6 | 11 | 11 |
| 19 | 5 | 12 | 12 |
| 18 | 4 | 13 | 13 |
| 17 | 3 | 14 | 14 |
| 16 | 2 | 15 | 15 |
| 15 | 1 | 16 | 16 |
| 14 | 0 | 17 | 17 |
| 13 | NC | 18 | 18 |
| 12 | NC | 19 | 19 |
| 11 | NC | 20 | 20 |
| 10 | NC | 21 | 21 |
| 9 | NC | 22 | 22 |
| 8 | NC | 23 | 23 |
| 7 | NC | 24 | 24 |
| 6 | NC | 25 | 25 |
| 5 | NC | 26 | 26 |
| 4 | NC | 27 | 27 |
| 3 | NC | 28 | 28 |
| 2 | NC | 29 | 29 |
| 1 | NC | 30 | 30 |
| 0 | NC | 31 | 31 |

# Appendix B

This appendix discusses the simulation of a Series 4 ORCA FPGA design with a system bus. The system bus and other modules required (e.g., synchronous ROM, etc.) are generated using SCUBA (Synthesis Compiler for User Programmable Arrays), and ModelSim is the simulation tool used. The system bus smartmodel is required to perform the simulation, which is provided with the ORCA Foundry software. The block diagram of a typical system bus simulation testbench is shown in Figure 16. This figure shows the components that can be used within a system bus in a Series 4 design. Section 1 of this document explains the details of each block Section 2 provides three simulation examples to help the user understand the simulation procedure of the system bus. All files used in these examples are available to users, enabling them to step through the simulation guide smoothly.

***Figure 16. Testbench Block Diagram***




## Stimulus

This block provides the stimulus to the inputs of the Device Under Test (DUT). Behavioral models in this block (such as “mpu.vhd” used in Examples 1 and 3) produce the following signals to drive the stimulus:

## Resets

### System Bus Resets

The system bus resets can be tied to the global reset of the design for convenience. There are two resets that are used independently for the User Master Interface (UMI) and the User Slave Interface (USI).

- **Us_reset:** User Slave Interface reset is active high and has to be provided at the beginning of the simulation to be able to use USI.
- **Um_reset:** User Master Interface reset is active high and has to be provided at the beginning of the simulation to be able to use UMI.

### FPGA Resets

The FPGA can have multiple resets for the different design blocks implemented. However, it is good practice to have just one reset for the whole design so that it uses the asynchronous Global Set Reset (GSR).

## Clocks

The following clocks need to be addressed in system bus designs:

- **MPI_CLK:** This is the Microprocessor Interface (MPI) clock that has to be provided if MPI is used in the design.
- **UM_CLK:** The UMI needs this clock for its operation.
- **US_CLK:** The USI needs this clock for its operation.
- **USR_CLK:** This is one of the clocks that can be used to drive the system bus HCLK. The other options are MPI_CLK and the internal oscillator clock. SCUBA selects one of these clocks during the generation of a system bus.

## Test Vectors

The test vectors can be generated using a mathematical algorithm or can be read by a behavioral model from a text file (such as mpu_in.txt used in Examples 1 and 3).

## Signals

The list of signals and constants represents the wires of the top-level testbench file that connects the components of the testbench together.

## Device Under Test (DUT)

The DUT is a Series 4 FPGA design with the system bus.The FPGA logic may contain the user-defined logic to perform various logical functions. It may also generate data for the testbench. Unlike the external stimulus, which is a component of the top-level testbench file, this lies inside the FPGA and may be termed as internal stimulus. The advantage of having internal stimulus is that this code may be synthesized and tested in hardware in real time. The system bus components in the FPGA that are typically used with system bus are:

- Microprocessor Interface
- User Master Interface
- User Slave Interface
- Embedded Block RAM

It is an option in SCUBA to generate all or any of the above-mentioned components when the system bus module is generated.

## Response

The response of the testbench can be stored in a number of ways

- Output Vectors
- Waveform: The waveform tool also uses the output vectors to generate the waveform.
- Text File: A text file can be generated during the simulation, reflecting the main events of the simulation. These main events are user defined.

## Simulation Examples

Three simulation examples are discussed here. The objective of these examples is to perform three different simulations using ORCA series 4 FPGA designs incorporating a system bus, and then to step through the MPI, UMI and USI usage within these designs. The following pages will step through the functional simulation of the DUT for the following operations:

- **Example 1 (Testbench Files: "mpius.zip")**
  - Use the MPI to interface with the FPGA soft registers using USI.

  - Show that the data written to FPGA soft registers can be used to perform a logical function: All the bits written to 0x08000 and 0x08001 are logically OR'd and the result is used to drive the system bus input, "us_irq".

- **Example 2 (Testbench Files: "umfpsc.zip")**
  - Use the UMI to read from a synchronous ROM and write to the ORT8850 FPSC Core at the base address of 0x30000. The ORT8850 Core has been left out of this example so that users can utilize this example without the FPSC design kit and core model.
- **Example 3 (Testbench Files: "mpiusbr.zip")**
  - Use the MPI to write to a system bus Block RAM.
  - Use the MPI to write "10101010" to the FPGA using USI to register location 0x08001 after it writes to the block RAM.
  - Use the FPGA Logic to detect the "10101010" from the register 0x08001 and start reading all the data from the System bus Block RAM.

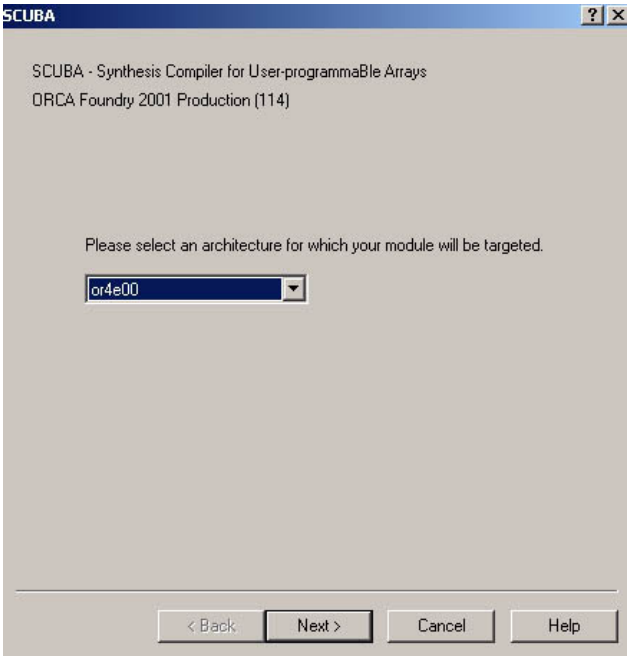
Since these examples need to generate a Series 4 FPGA design with a System bus with MPI, UMI, USI and Block RAM, the procedure of instantiation of all of the macros is also discussed. The topics are summarized below.

- Using SCUBA to generate system bus with MPI, UMI and USI.
- Using SCUBA to generate system bus with Block RAMs.
- Generate synchronous ROM for UMI using SCUBA.
- Complete the top-level design file by inserting user logic for UMI and USI and user ports.
- Integrate the respective components of three different testbenches.

## Generating the System Bus Using SCUBA

Figures 17-22 show the steps required to create the system bus with MPI, UMI and USI.

***Figure 17.***

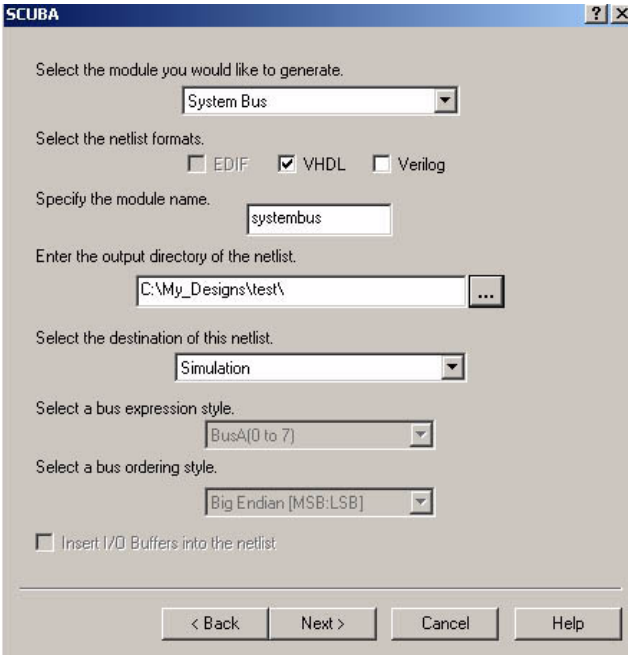



SCUBA Page 1

Screen shot (Figure 17)

Please select an architecture for which your module will be targeted: or4e00

***Figure 18.***




SCUBA next page

Screen (Figure 18)

Select the module you would like to generate: System Bus

Select the netlist formats: VHDL

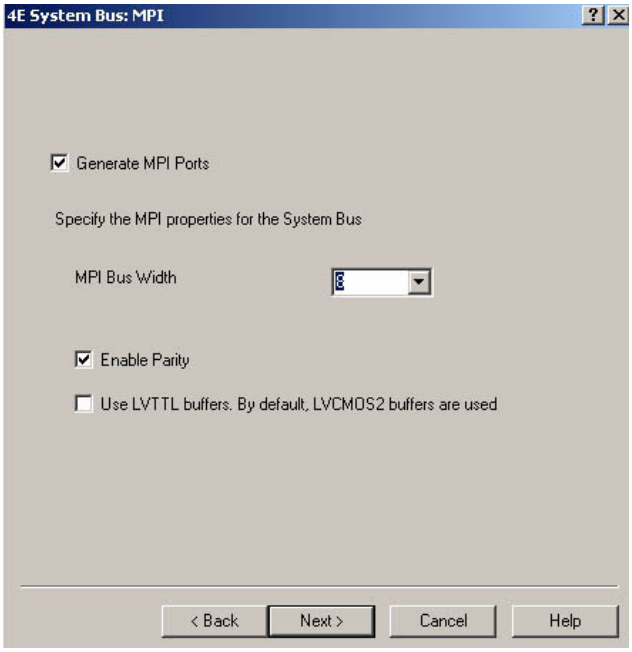
Specify the module name: systembus

Enter the output directory of the netlist: c:\my_designs\test\ (This should be the path of the directory where the user wants to store the files generated by SCUBA)

Select the destination of the netlist: simulation

CLICK NEXT

***Figure 19.***




SCUBA next page

Screen shot (Figure 19)

Generate MPI Ports: YES

MPI Bus Width: 8

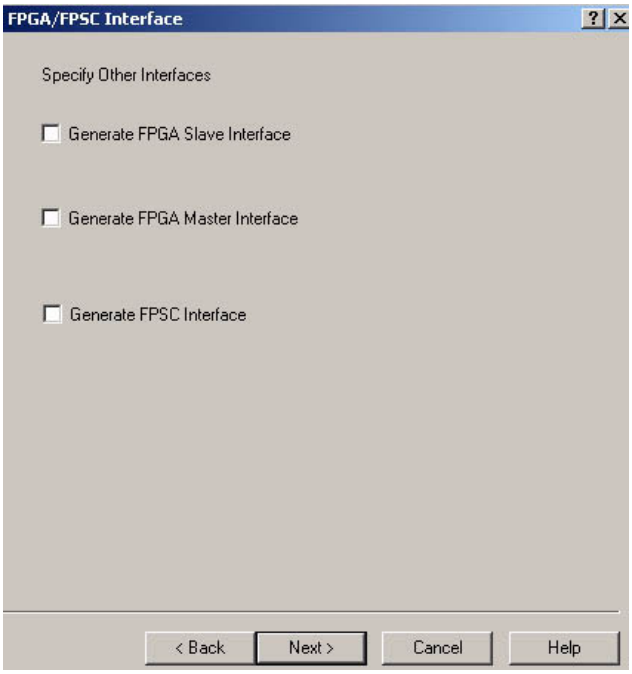
Enable Parity: YES

Use LVTTL buffers: NO

(The MPI is required for Examples 1 and 3 only. Skip this SCUBA page for Example 2)

CLICK NEXT

***Figure 20.***




SCUBA next page

Screen shot (Figure 20)

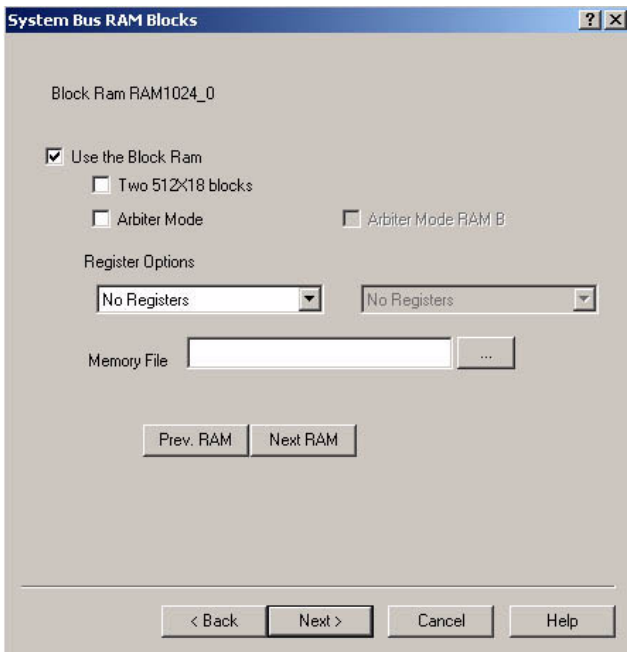
Generate FPGA Slave Interface: YES (FPGA slave interface is only used in Examples 1 and 3. This is not checked for Example 2)

Generate FPGA Master Interface: YES (This is only checked for Example 2)

Generate FPSC Interface: NO

CLICK NEXT

***Figure 21.***




SCUBA next page

Screen shot (Figure 21)

(Skip this SCUBA page for Examples 1 and 2 because Block RAM is only used in Example 3)

Use the Block RAM: YES

ÿ Two 512X18 blocks: NO (This generates a 1024 X 9 RAM module)

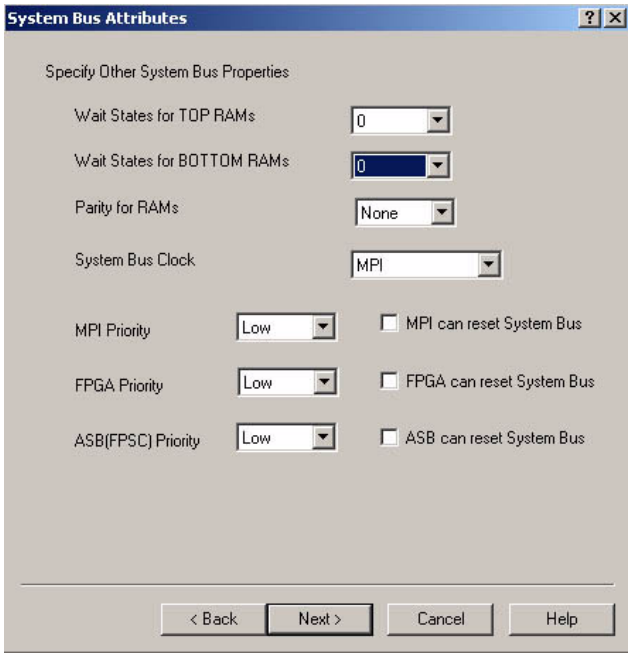
Arbiter Mode: NO

Register Options: No-registers

Memory File: <BLANK>

CLICK NEXT

*Figure 22.*




SCUBA next page

Screen shot (Figure 22)

Leave everything at defaults.

CLICK NEXT

SCUBA next page

CLICK ON FINISH

## Ports of the Microprocessor Interface for Example 1 and 3

The MPI generated with the system bus has the ports shown in Figure 23 and all MPI ports start with a “mpi” prefix, except for cs0n and cs1 ports. The MPI I/Os in the “systembus.vhd” module are port mapped to the I/Os of DUT-“chip.vhd”. The DUT’s MPI ports are port-mapped to the I/Os of the microprocessor emulator “mpu.vhd” file in the testbench file “tb.vhd”. The emulator acts as the master of the system bus whereas every other module acts as a slave of the microprocessor. The microprocessor emulator reads the memory map from a text file to read/write the address locations in one of the slaves’ registers.

***Figure 23. 8-bit MPI Ports in the “systembus.vhd” Module***

```
mpi_clk: in  std_logic;
mpi_rdwr_n: in  std_logic;
mpi_strbn: in  std_logic;
mpi_tsz: in  std_logic_vector(0 to 1);
mpi_burst: in  std_logic;
mpi_bdip: in  std_logic;
cs0n: in  std_logic;
cs1: in  std_logic;
mpi_addr: in  std_logic_vector(14 to 31);
mpi_data: inout  std_logic_vector(0 to 7);
mpi_parity: inout  std_logic;
mpi_ta: out  std_logic;
mpi_retry: out  std_logic;
mpi_tea: out  std_logic;
mpi_irq: out  std_logic;
```

## Ports of the User Master Interface for Example 2

The UMI generated with the system bus has the ports shown in Figure 24. All UMI ports start with a “um” prefix. The UMI I/Os in the “systembus.vhd” module are port mapped to the internal signals of the DUT. The “chip.vhd” module. These internal signals are used by the FPGA logic, created by the user in the “chip.vhd” module to access the system bus. That means that the stimulus is provided from within the DUT.

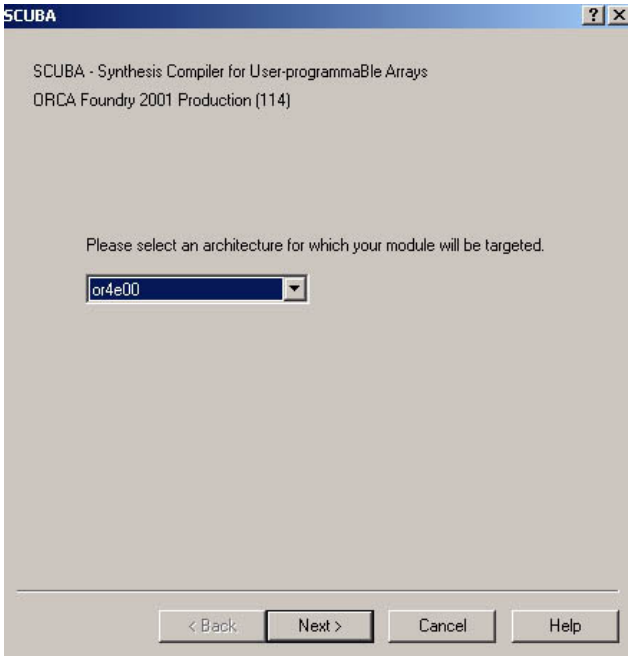
***Figure 24. UMI Ports in the "systembus.vhd" Module***

```
um_addr: in  std_logic_vector(17 downto 0);
um_wdata: in  std_logic_vector(35 downto 0);
um_clk: in  std_logic;
um_reset: in  std_logic;
um_write: in  std_logic;
um_read: in  std_logic;
um_burst: in  std_logic;
um_ready: in  std_logic;
um_size: in  std_logic_vector(1 downto 0);
um_lock: in  std_logic;
um_irq: in  std_logic;
um_rdata: out  std_logic_vector(35 downto 0);
um_granted: out  std_logic;
um_ack: out  std_logic;
um_retry: out  std_logic;
um_err: out  std_logic;
```

## Generating the Synchronous ROM using SCUBA for Example 2

This subsection shows all the steps (Figures 25-28) required to generate a synchronous ROM that is required by the UMI in Example 2.In this example, the UMI reads from a synchronous ROM instantiated inside the FPGA and writes to eleven different addresses with base address of 0x30000. This base address corresponds to an embedded ASIC core inside the ORT8850 FPSC device.

***Figure 25.***



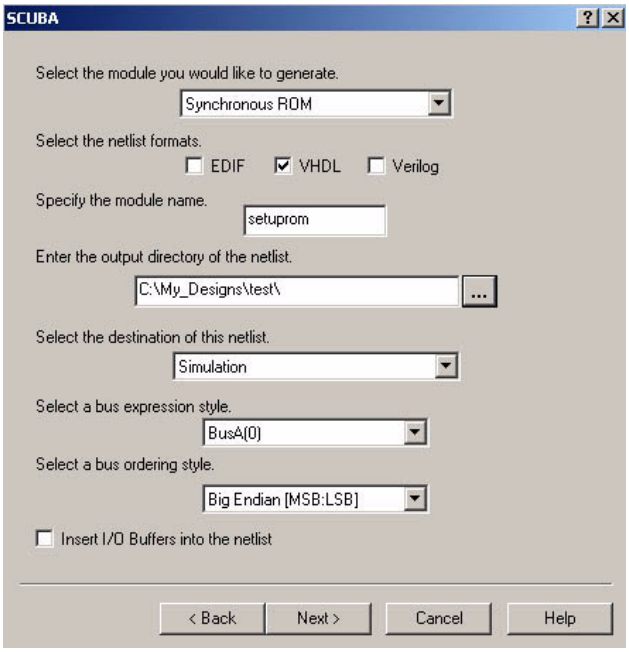

SCUBA page 1

Screen shot (Figure 25)

Please select an architecture for which your module will be targeted: or4e00

CLICK NEXT

***Figure 26.***




SCUBA next page

Screen shot (Figure 26)

Select the module you would like to generate: Synchronous ROM

Select the netlist formats: VHDL

Specify the module name: setuprom

Enter the output directory of the netlist: c:\my_designs\test\

Select the destination of the netlist: Simulation

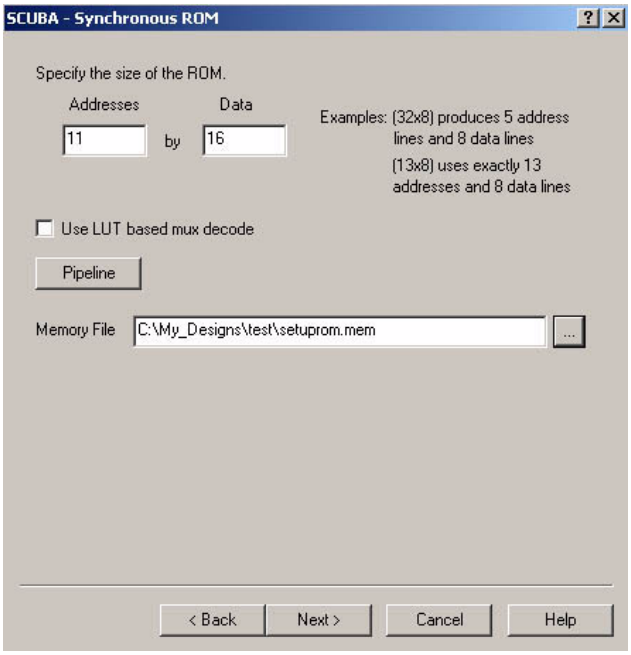
Select a bus expression style: BusA(0) (Default)

Select a bus ordering style: Big Endian [MSB:LSB] (Default)

Insert I/O Bufers into the netlist: NO

CLICK NEXT

***Figure 27.***




SCUBA next page

Screen shot (Figure 27)

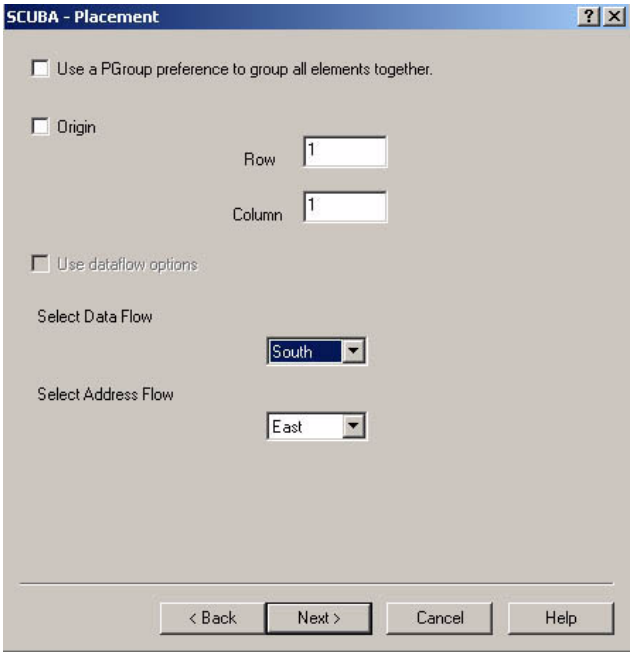
Specify the size of the ROM: 11 by 16

Use LUT based mux decode: NO

Memory File: C:\my_designs\test\setuprom.mem

CLICK NEXT

***Figure 28.***




SCUBA next page

Screen shot (Figure 28)

Leave everything at defaults

CLICK NEXT

SCUBA next page

CLICK FINISH

SCUBA requires a memory file to generate a synchronous ROM in order to store the data to be read by the UMI. The format of the memory file (setuprom.mem) is shown in Figure 29. There are 11 items in the memory file read by the UMI state machine in Example 2. Each item has an index (0 to a) and it is followed by 16 bits of data that is stored in the ROM. The first eight bits are used by the state machine as the address (the base address 0x30000 is attached to it by the state machine logic) and the next eight bits are used as data to be written to that address.

***Figure 29. "setuprom.mem" file Needed by SCUBA to Generate "setuprom.vhd"***

```
0: 04a0
1: 0501
2: e000
3: 20e0
4: 38e0
5: 50e0
6: 68e0
7: 80e0
8: 98e0
9: b0e0
a: c8e0
```

## Ports of the User Slave Interface for Example 1 and 3

The USI generated with the system bus has the ports shown in Figure 30. All USI ports start with a "us" prefix. The USI ports of the "systembus.vhd" module are mapped to the FPGA signals of the top-level "chip.vhd" file.

***Figure 30. USI Ports in the "systembus.vhd" Module***

```
us_rdata: in  std_logic_vector(35 downto 0);
us_clk: in  std_logic;
us_reset: in  std_logic;
us_ack: in  std_logic;
us_retry: in  std_logic;
us_err: in  std_logic;
us_irq: in  std_logic;
us_wdata: out  std_logic_vector(35 downto 0);
us_addr: out  std_logic_vector(17 downto 0);
us_burst: out  std_logic;
us_rdy: out  std_logic;
us_wr: out  std_logic;
us_size: out  std_logic_vector(1 downto 0);
```

**Ports of the System bus Block RAM**

The Block RAM generated with the system bus has the ports shown in Figure 31. All the ports end with a "top_0" suffix for a 1024 X 9 RAM. The other 1024 X 9 RAMs that can be generated will have "top_1", "top_2", "top_3", "bot_0", "bot_1", "bot_2", or "bot_3" suffix. The Block RAM ports of the "systembus.vhd" module are mapped to the FPGA signals of the top-level "chip.vhd" module.

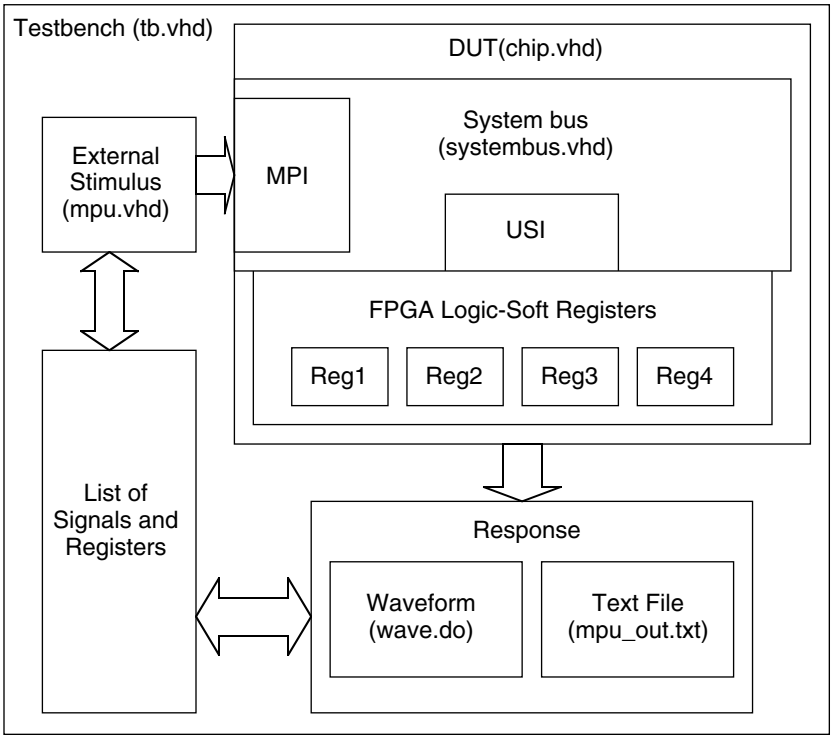
***Figure 31. Block RAM Ports of the System Bus***

```
waddr_top_0: in  std_logic_vector(9 downto 0);
raddr_top_0: in  std_logic_vector(9 downto 0);
din_top_0: in  std_logic_vector(17 downto 0);
bw_top_0: in  std_logic_vector(1 downto 0);
wclk_top_0: in  std_logic;
rclk_top_0: in  std_logic;
wren_top_0: in  std_logic;
rden_top_0: in  std_logic;
dout_top_0: out  std_logic_vector(17 downto 0);
```

## Testbench Simulation for Example 1

The block diagram of the testbench for Example 1 is shown in Figure 32. The components of the testbench and DUT are discussed in the following subsections.

***Figure 32. Testbench Block Diagram for Example 1***




Components of the testbench:

- DUT: the file “chip.vhd” defines The DUT and the all its ports are shown in Figure 10.
- The microprocessor emulator (provides stimulus). The file “mpu.vhd” defines the emulator.

Components of the DUT:

- System bus with MPI and USI. The system bus module is defined in the “systembus.vhd” module.
- Softreg_input: This is a user-defined component, which provides data to the USI when a read transaction is initiated by the MPI. It is defined in “registers.vhd”.
- Softreg_outptut: This is a user-defined component, which stores data to a register when MPI performs a write transaction. It is defined in “registers.vhd”.

***Figure 33. Ports of the Entity CHIP for Example 1***

```
entity CHIP is
port(
        reset: in std_logic;

MPI_CLK, MPI_RDWR_N, MPI_STRBN, MPI_BURST, MPI_BDIP,
CS0N, CS1, USR_CLK, USR_IRQ_GENERAL : IN STD_LOGIC;
MPI_ADDR : IN STD_LOGIC_VECTOR (14 TO 31);
MPI_TSZ : IN STD_LOGIC_VECTOR (0 TO 1);
MPI_TA, MPI_RETRY, MPI_TEA, MPI_IRQ, USR_IRQ : OUT STD_LOGIC;
MPI_PARITY : INOUT STD_LOGIC;
MPI_DATA : INOUT STD_LOGIC_VECTOR (0 TO 7))
end CHIP;
```

## The Microprocessor Emulator

The microprocessor emulator reads from the “mpu_in.txt” file shown in Figure 34 and performs the read/write transactions based upon the command line. It also outputs “mpu_out.txt” file, which contains a report of all the events of “mpu_in.txt” file.

***Figure 34. Contents of the “mpu_in.txt” file for Example 1***

```
MPU_IN.TXT file: Data bus is 8 bit wide
comment     "************************************************ "
simtime     12 us
comment     "Read and Write some registers "
write_byte  08000 00
simtime     250 ns
write_byte  08001 00
simtime     250 ns
write_byte  08000 34
simtime     250 ns
write_byte  08001 43
simtime     250 ns
read_byte   08000 34
simtime     250 ns
read_byte   08001 43
simtime     250 ns
read_byte   08002 34
simtime     250 ns
read_byte   08003 43
comment     " *************"
```

- **Comment:** The emulator file (mpu.vhd) treats this line segment as a comment. If a new line is required to extend the comment then it should also start with the “comment” instruction.
- **Read_byte:** This function is implemented in the emulator file to read the memory location. The various memory locations are explained in detail in the MPI and System bus application note. In this example the 0x08000 base address that corresponds to the User Slave Interface is used.
- **Write_byte:** This function writes a byte to the memory location. It requires two parameters; the memory location and the data byte to write to that memory location. This example uses 8-bit data transactions, although 16-bit and 32-bit data transactions are also possible. For example, write_byte 08000 34 will write to the memory location 0x08000 with “00110100”. The location 0x08001 is assigned a value of “01000011”.
- **Simtime:** This command waits for a number of units of time that the user provides as the parameter. This example uses 250ns wait time between instructions.

## Simulation Process for Example 1 Using ModelSim

The “vsim.do” is a macro file (Figure 35) that contains all of the commands required, to compile the testbench and run the simulation for 20μs. It also calls the “wave.do” file that contains all the signals to be viewed in the waveform window.

***Figure 35. Simulation Macro file "vsim.do" for Example 1***

```
vlib work
vcom systembus.vhd
vcom registers.vhd
vcom chip.vhd
vcom mpu.vhd
vcom tb.vhd

vsim tb
do wave.do
run 20 us
```

**Sequence of simulation events for Example 1**

The events that take place from the start of the simulation are discussed in this section:

- **At time=0ns:**
  - Simulation begins
  - The 66MHz clock is provided to the MPI
  - System bus' pre-configuration state begins
- **At time=10ns:**
  - Reset is asserted
  - System bus is still in its pre-configuration state.
- **At time=10µs:**
  - Reset is released
  - System bus is still in its pre-configuration state.
- **At time=11µs:**
  - System bus configuration is complete
- **At time=12µs:**
  - The microprocessor transactions with the system bus begin
  - The MPI write transaction: address= 0x08000, data=00: When the first write transaction takes place the MPI acknowledges with active low mpi_ta with mpi_rdwr_n='0' (indicating write command), valid address on mpi_addr=0x8000 and valid data on mpi_data="00" for single clock cycle (Figure 37 cursor position: 12029100ps). Consequently the data is available at the USI, with active high us_rdy signal asserted and the soft register at the us_addr=0x08000 stores the data by providing the us_ack='1' for 1 clock cycle (Figure 37 cursor position: 12210957ps). Similarly the second write transaction takes place at us_addr=0x08001. The internal logic "ORs" the bits of two bytes written to 0x08000 and 0x08001 and outputs the result to the us_irq input of the system bus.
  - The MPI read transactions. The MPI provides the "mpi_addr"=0x08003 from where data is to be read with mpi_rd_wr_n='1' (indicating read command, Figure 38 cursor position: 14907358ps). The "us_rdy" signal goes high and the User Slave Logic in the soft register at us_addr=0x08003 provides data with us_ack= '1' for just one clock cycle and when MPI receives this data it responds with an active low mpi_ta signal (Figure 38 cursor position: 15190539ps). After each MPI transaction the emulator records the time of transaction in a text file called "mpu_out.txt" the contents of which are shown in Figure 36.

*Figure 36. The Contents of the "mpu_out.txt" File for Example 1*

```
"************************************************ "
 Advance Simulation Time, From Time = 2 ns                    To Time = 12002 ns
"Read and Write some registers "
 Byte Write Cycle : Adrs = 08000; Data = 00
 Advance Simulation Time, From Time = 12052.825 ns           To Time = 12302.825
ns
 Byte Write Cycle : Adrs = 08001; Data = 00
 Advance Simulation Time, From Time = 12355.825 ns           To Time = 12605.825
ns
 Byte Write Cycle : Adrs = 08000; Data = 34
 Advance Simulation Time, From Time = 12658.825 ns           To Time = 12908.825
ns
 Byte Write Cycle : Adrs = 08001; Data = 43
 Advance Simulation Time, From Time = 12961.825 ns           To Time = 13211.825
ns
 Byte Read Cycle  : Adrs = 08000; Data = 34
 Advance Simulation Time, From Time = 13522.375 ns           To Time = 13772.375
ns
 Byte Read Cycle  : Adrs = 08001; Data = 43
 Advance Simulation Time, From Time = 14082.925 ns           To Time = 14332.925
ns
 Byte Read Cycle  : Adrs = 08002; Data = 34
 Advance Simulation Time, From Time = 14643.475 ns           To Time = 14893.475
ns
 Byte Read Cycle  : Adrs = 08003; Data = 43
" *************"
** SIM_IN PROCESSING COMPLETE ** : TIME = 15204.025 ns
```

*Figure 37. Functional Simulation Waveform for the MPI Write Transaction, at Address 0x08000, in Example 1*

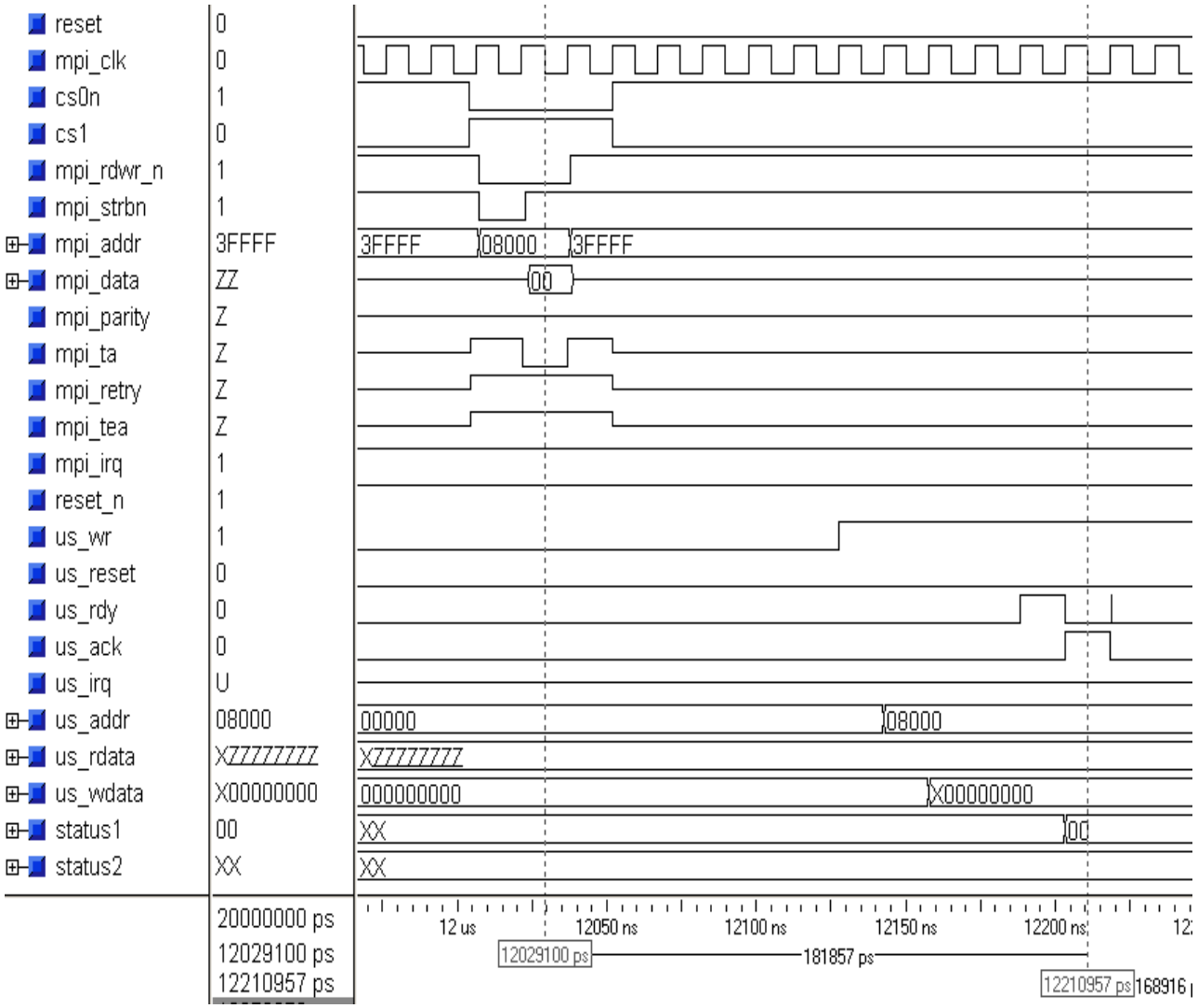
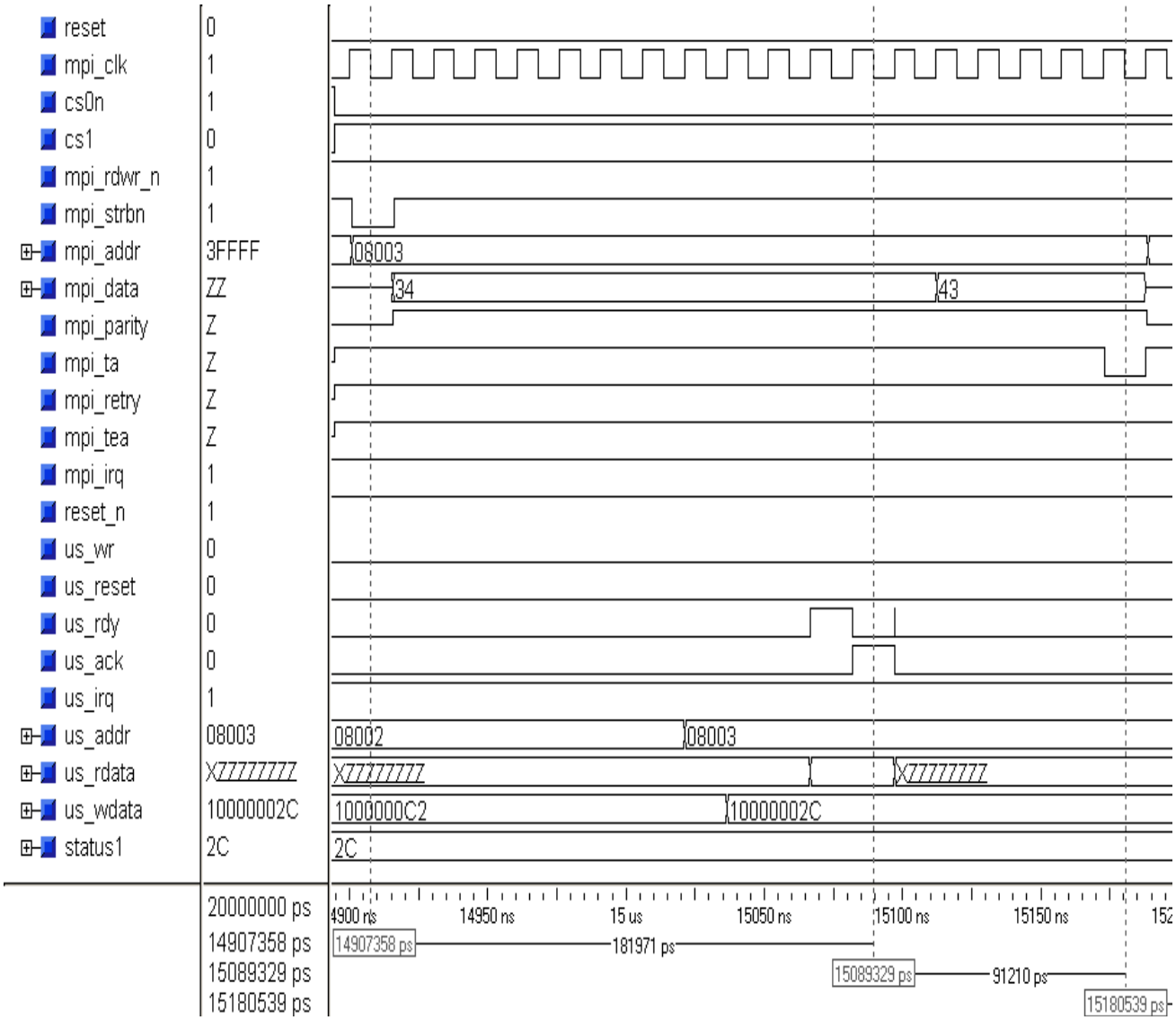


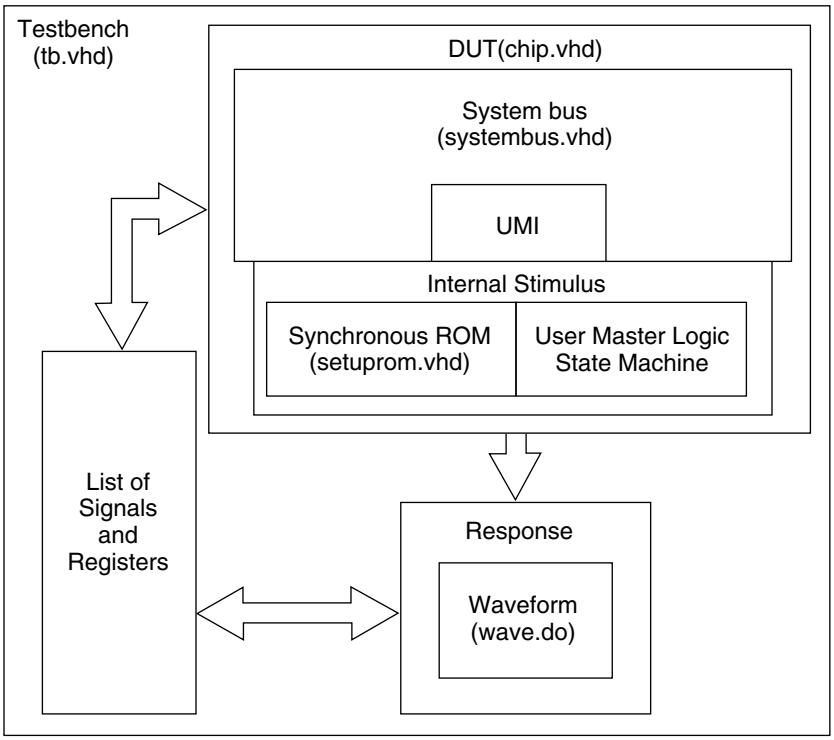
*Figure 38. Functional Simulation Waveform for the MPI Read Transaction, at Address 0x08003, in Example 1*



## Testbench Simulation for Example 2

The block diagram of the testbench for Example 2 is shown in Figure 39. The components of the testbench and DUT are discussed in the following subsections:

***Figure 39. Testbench Block Diagram for Example 2***




**Components of the Testbench**

- DUT: “chip.vhd”.

**Components of the DUT**

The DUT contains the clock and the reset as shown in Figure 40. This design uses the UMI to perform write transactions to the ORT8850 embedded ASIC block. To simplify this HDL code example and to provide a code example that is portable the ORT 8850 block has been removed from this example. The only peripheral on the system bus in this HDL code is the UMI. In the full code the FPSC interface would also be present on the system bus. For this example to be simulated, the user needs the ORT 8850 Design Kit, as well as the ORT 8850_core simulation model. The ORT 8850_core has been left out of this example so that users can utilize it without the Design Kit and core model. The components of the DUT are given below:

- System bus with UMI: “systembus.vhd”.
- Synchronous ROM generated from “setuprom.mem” file: “setuprom.vhd”.
- State machine for the UMI write protocol: The state machine has three states
  - Getbus: This state acquires the system bus for UMI by asserting the signals um_write=’1’ and um_lock=’1’. This prompts the system bus to return um_ack=’1’ signal to the UMI. The state-machine detects this signal and passes the control to the next state, which is startw.
  - Startw: This state provides the data and address stored in the synchronous ROM to the data bus um_wdata and the address bus um_addr respectively, with an active high um_ready=’1’ for 1 clock cycle only. Subsequently, in the next clock cycle the control of the state-machine is passed on to the donew state.

– Donew: This state finishes the single UMI write transaction by dissasserting the um_ready='0' signal and passing the control back to the getbus state.

Each UMI write cycle is repeated for the number of items stored in the ROM, and after all the address positions are written the lock for the system bus is relinquished by another write transaction with a low um_lock='0' signal. This step is very important if there are multiple masters performing Time-Division-Accesses on the system bus.

***Figure 40. Ports of "chip.vhd" (DUT) for Example 2***

```
entity chip is
  port(
    reset : in std_logic;
    clk: in  std_logic
);
end chip;
```

## Simulation process for Example 2 Using ModelSim

The "vsim.do" (macro file) for this example is shown in Figure 41. The simulation is run for 15µs. It also calls the "wave.do" file that contains all the signals to be viewed in the waveform window.

***Figure 41. Modelsim Simulation Macro File "vsim.do" for Example 2***

```
vlib work
vcom systembus.vhd
vcom setuprom_sim.vhd
vcom chip.vhd
vcom tb.vhd
vsim tb
do wave.do
run 15 us
```

## Sequence of Simulation Events for Example 2

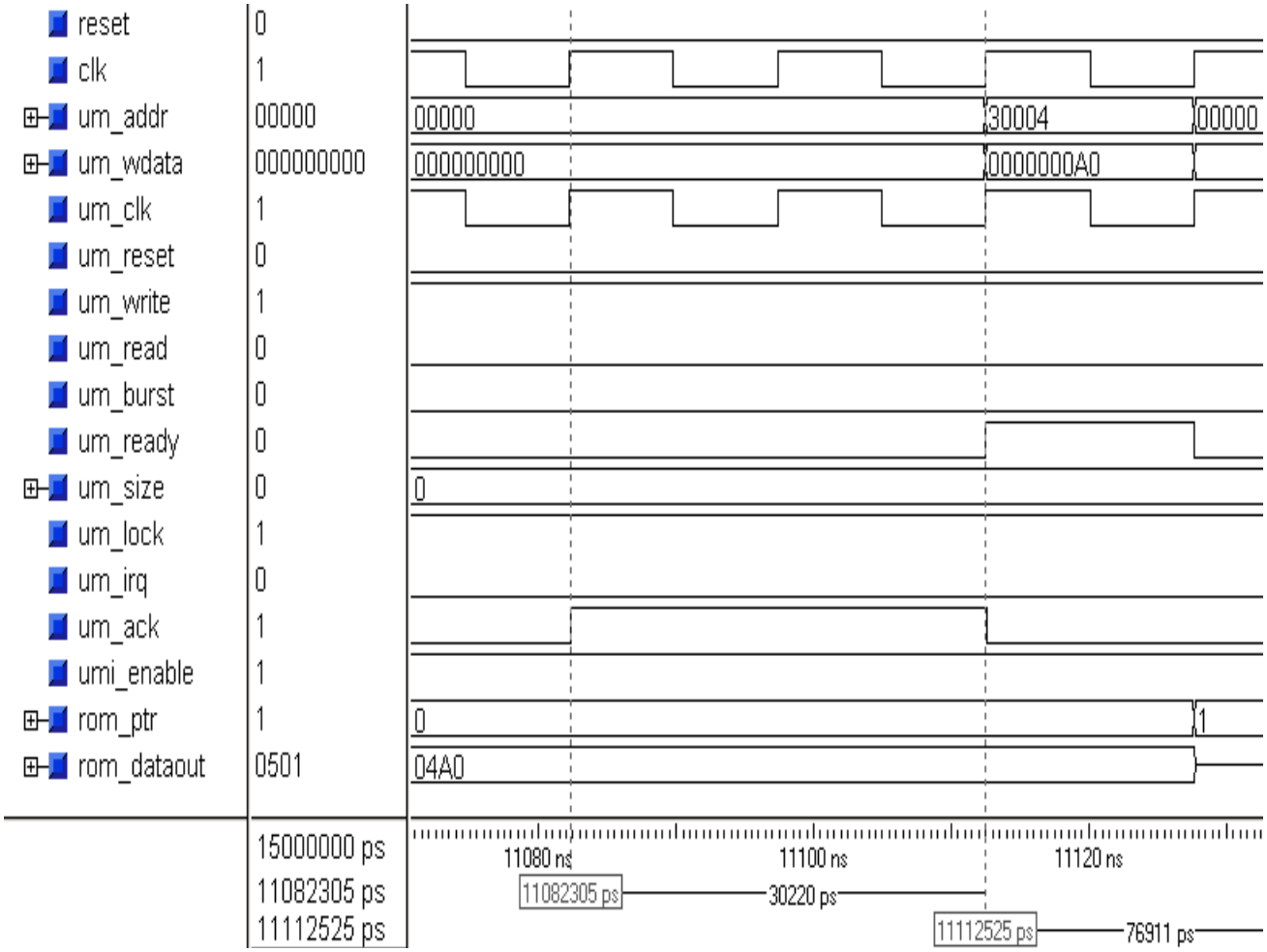
The following events take place from the start of the simulation to the first write transaction:

- **At time = 0ns:**
  - Simulation begins
  - The clock of 66MHz is provided to the DUT clocks
  - System bus' pre-configuration state begins
- **At time =1 0ns:**
  - Reset is asserted
  - System bus is still in its pre-configuration state.
- **At time = 11µs:**
  - System bus configuration is complete
  - Reset is released
- **At time = 11.006µs:**
  - UMI signals um_write='1' and um_lock='1' are asserted to acquire the system bus for UMI transactions.
- **At time = 11.082µs** (Figure 42 cursor position: 11082305ps):
  - The signal um_ack='1' is provided by the UMI to the User Master Logic implemented as a state machine. This signal signifies that the system bus is ready for the UMI to perform a write transaction.
- **At time = 11.11µs:**
  - The first UMI write transaction takes place as soon as an active high um_ready='1' is provided by the User Master Logic to the UMI for just one clock cycle, accompanied with valid data and address on

um_wdata=”A0” and um_waddr=0x30004 respectively (Figure 42 cursor position: 11112525ps). The um_ready=’1’ signal should be only provided after the um_ack=’1’ is detected.

– Similarly the subsequent write transactions take place. During all the write transactions the um_lock and um_write signals are always asserted.
– The last write transaction takes place with um_lock=’0’ to relinquish the control of the system bus so that other masters can perform transactions with the system bus.

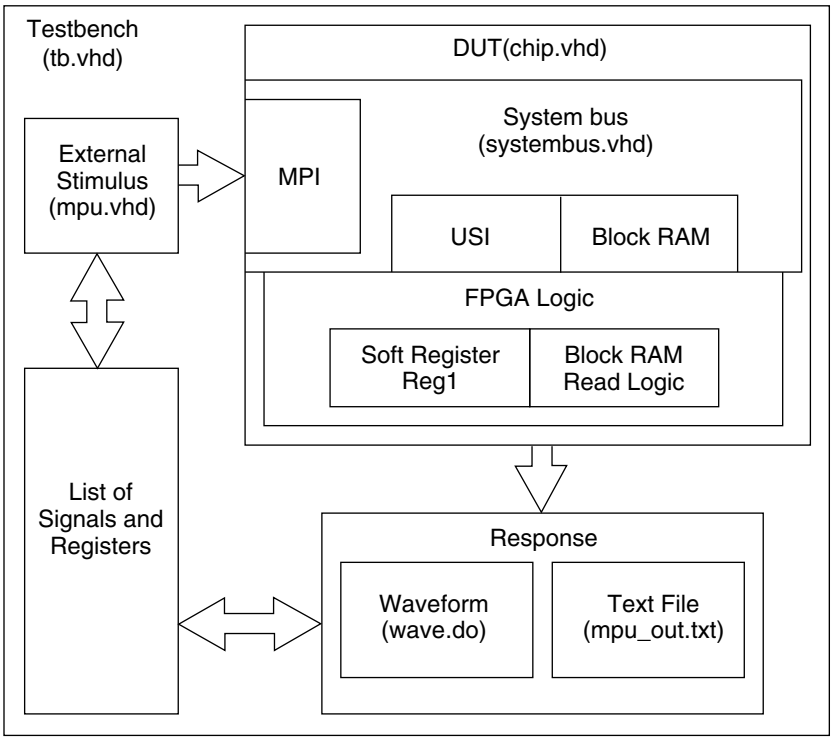
***Figure 42. Functional Simulation Waveform for the UMI Write Transaction at Address 0x30005 with Data A0, in Example 2***



## Testbench Simulation for Example 3

The block diagram of the testbench for Example 3 is shown in Figure 43. The components of the testbench and DUT are discussed in the following subsections:

***Figure 43. Testbench Block Diagram for Example 3***




Components of the testbench:

- DUT: The ports of the “chip.vhd” file are shown in Figure 44.
- The microprocessor emulator, “mpu.vhd”, (provides stimulus). Similar to Example 1, the emulator reads from the “mpu_in.txt” file (Figure 45) for the memory locations to read/write data.

Components of the DUT:

- System bus with MPI, USI and System bus Block RAM: “systembus.vhd”.
- Softreg_input: This is a user-defined component, which provides data to the USI when a read transaction is initiated by the MPI: “registers.vhd”.
- Softreg_outptut: This is a user-defined component, which stores data in a register when MPI performs a write transaction: “registers.vhd”.

***Figure 44. Ports of the Entity CHIP for Example 3***

```
entity CHIP is
port(
        reset: in std_logic;
MPI_CLK, MPI_RDWR_N, MPI_STRBN, MPI_BURST, MPI_BDIP,
CS0N, CS1, USR_CLK, USR_IRQ_GENERAL : IN STD_LOGIC;
MPI_ADDR : IN STD_LOGIC_VECTOR (14 TO 31);
MPI_TSZ : IN STD_LOGIC_VECTOR (0 TO 1);
MPI_TA, MPI_RETRY, MPI_TEA, MPI_IRQ, USR_IRQ : OUT STD_LOGIC;
MPI_PARITY : INOUT STD_LOGIC;
MPI_DATA : INOUT STD_LOGIC_VECTOR (0 TO 7));
end CHIP;
```

***Figure 45. Contents of "mpu_in.txt" file for Example 3***

```
comment     "************************************************* "
simtime     12 us
comment     "Read and Write some registers "
read_byte   00000
simime      250 ns
write_byte  10000 A0
simtime     250 ns
write_byte  10001 A1
simtime     250 ns
write_byte  10004 A4
simtime     250 ns
write_byte  10005 A5
simtime     250 ns
write_byte  10008 A8
simtime     250 ns
write_byte  10009 A9
simtime     250 ns
write_byte  1000c Ac
simtime     250 ns
write_byte  1000d Ad
simtime     250 ns
write_byte  08001 AA
simtime     1 us
comment     " *************"
```

**Simulation process for Example 3 using Modelsim**

The "vsim.do" is a macro file (Figure 46) that contains all of the commands to compile the testbench and run the simulation for 20μs. It also calls the "wave.do" file that contains all of the signals to be viewed in the waveform window.

***Figure 46. Simulation Macro file "vsim.do" for Example 3***

```
vlib work
vcom systembus.vhd
vcom registers.vhd
vcom chip.vhd
vcom mpu.vhd
vcom tb.vhd

vsim tb
do wave.do
run 20 us
```

**Sequence of simulation events for Example 3**

The events that take place from the start of the simulation are discussed in this section.

- **At time = 0ns:**
  - Simulation begins
  - The 66MHz clock is provided to the MPI
  - System bus' pre-configuration state begins
- **At time=10ns:**
  - Reset is asserted
  - System bus is still in its pre-configuration state.
- **At time=10µs:**
  - Reset is released
  - System bus is still in its pre-configuration state.
- **At time=11µs:**
  - System bus configuration is complete
- **At time>=12µs:**
  - The microprocessor transactions with the system bus begin
  - First MPI transaction is a read transaction: address=0x00000 the first eight bits of the Device ID. An active high mpi_rdwr_n='1' (indicating read command) and an active low mpi_strbn='0' with the mpi_addr=0x00000 for just 1 clock cycle prompt the system bus to return the data byte at the given address (Figure 48 cursor position=12013950ps). The system bus returns with valid data "DC" and outputs it on mpi_data bus with an active low mpi_ta='1' for 1 clock cycle (Figure 48 cursor position=12210900ps).
  - First MPI write transaction: address=0x10000, data=A0: When the first write transaction takes place the MPI acknowledges with active low mpi_ta='0' signal (Figure 49 cursor position= 1227296ps) and the data gets stored in the Block RAM at address "0" and it is available on the br_dout (Figure 49 cursor position= 1237750ps) signal which is mapped to the dout_top_0 output of the Block RAM inside the DUT. It should be noticed that mpi_data gets reversed when written to the Block RAM.
  - Next MPI write transactions: address= 0x10001, data=A1: Similarly the second write transaction takes place at mpi_addr=0x10001 which is System bus Block RAM address "1". It should be noted that next write transaction does not take place at mpi_addr=10002 because it does not correspond to address "2" of the System bus Block RAM that is being used in this example. It corresponds to the other System bus Block RAM which was not instantiated in this example (The ports of the other system bus block RAM would end with suffix top_1). Therefore every two alternate MPI addresses with base 10000 correspond to every consecutive addresses on one of the System bus Block RAMs with suffix top_0.
  - Last MPI write transaction: address=0x08001, data=AA: This transaction is done so that the FPGA logic can detect that, required number of MPI writes have been completed and the FPGA logic can start reading the data from the Block RAM. This write transaction is shown in Figure 50 and the data byte "AA" is stored in the register status2 as "55" (cursor position=14862150ps).

– FPGA read transactions: The read transactions begin when active high br_rden='1' signal is provided with the br_raddr bus. In this example this is taken care of by the FPGA logic by keeping the br_rden signal high for eight clock cycles and the data on the br_dout bus is available for the FPGA to be sampled for further processing (Figure 51).
– The emulator records the MPI transactions in a text file called "mpu_out.txt", the contents of which are shown in Figure 47.

***Figure 47. The contents of "mpu_out.txt" File***

```
"************************************************ "
 Advance Simulation Time, From Time = 2 ns              To Time = 12002 ns
"Read and Write some registers "
 Byte Read Cycle  : Adrs = 00000; Data = DC
Byte Write Cycle : Adrs = 10000; Data = A0
 Advance Simulation Time, From Time = 12295.225 ns      To Time = 12545.225 ns
 Byte Write Cycle : Adrs = 10001; Data = A1
 Advance Simulation Time, From Time = 12598.225 ns      To Time = 12848.225 ns
 Byte Write Cycle : Adrs = 10004; Data = A4
 Advance Simulation Time, From Time = 12901.225 ns      To Time = 13151.225 ns
 Byte Write Cycle : Adrs = 10005; Data = A5
 Advance Simulation Time, From Time = 13204.225 ns      To Time = 13454.225 ns
 Byte Write Cycle : Adrs = 10008; Data = A8
 Advance Simulation Time, From Time = 13507.225 ns      To Time = 13757.225 ns
 Byte Write Cycle : Adrs = 10009; Data = A9
 Advance Simulation Time, From Time = 13810.225 ns      To Time = 14060.225 ns
 Byte Write Cycle : Adrs = 1000C; Data = AC
 Advance Simulation Time, From Time = 14113.225 ns      To Time = 14363.225 ns
 Byte Write Cycle : Adrs = 1000D; Data = AD
 Advance Simulation Time, From Time = 14416.225 ns      To Time = 14666.225 ns
 Byte Write Cycle : Adrs = 08001; Data = AA
 Advance Simulation Time, From Time = 14719.225 ns      To Time = 15719.225 ns
" *************"
***** SIM_IN PROCESSING COMPLETE *****
SIM_IN PROCESSING COMPLETE : TIME = 15720.225 ns
```

***Figure 48. Functional Simulation Waveform for the MPI Read Transaction at Address 0x00000, for Series 4 Device ID, in Example 3***

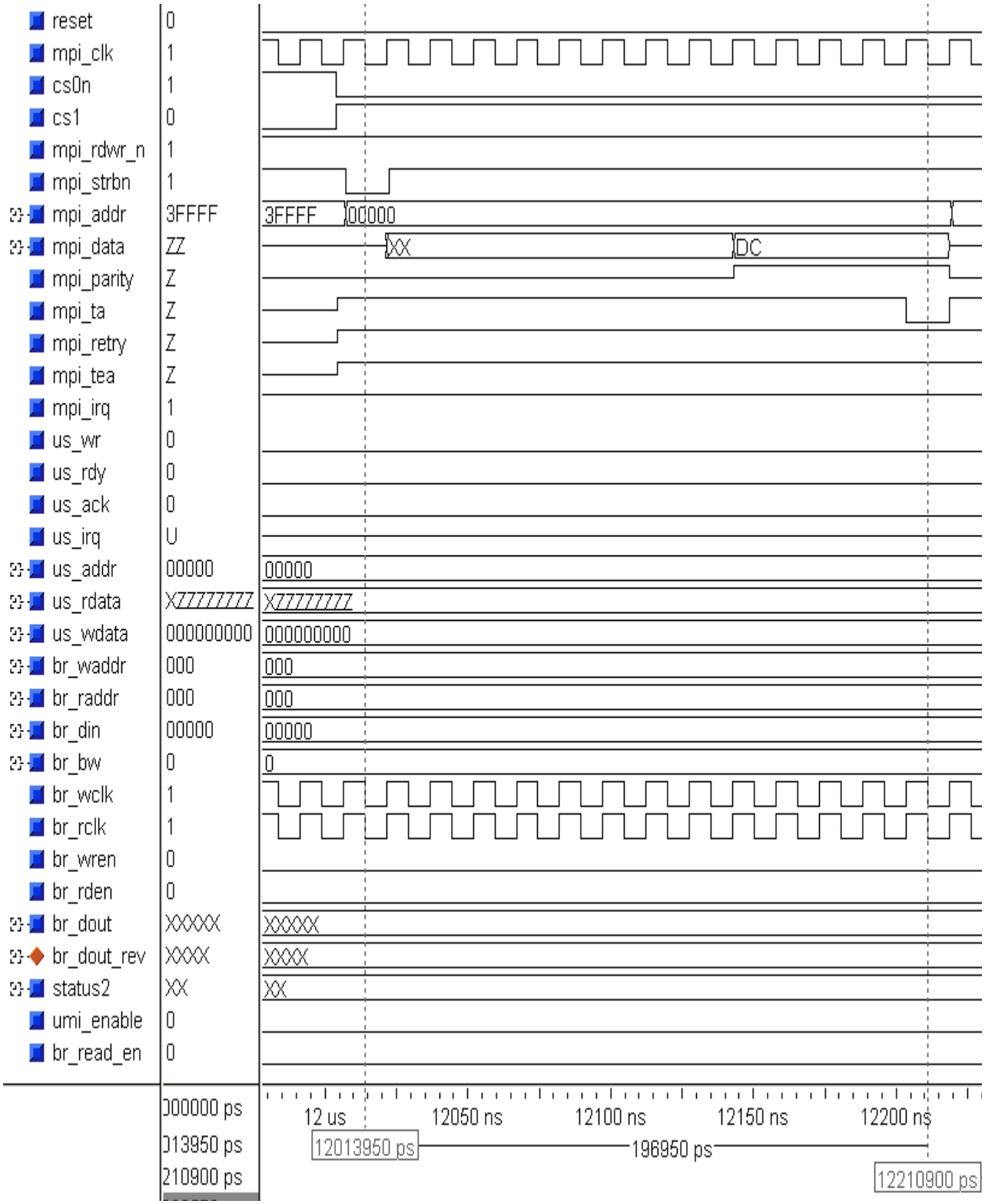


***Figure 49. Functional Simulation Waveform for the MPI Write Transaction at Address 0x10000 with Data A0, in Example 3***

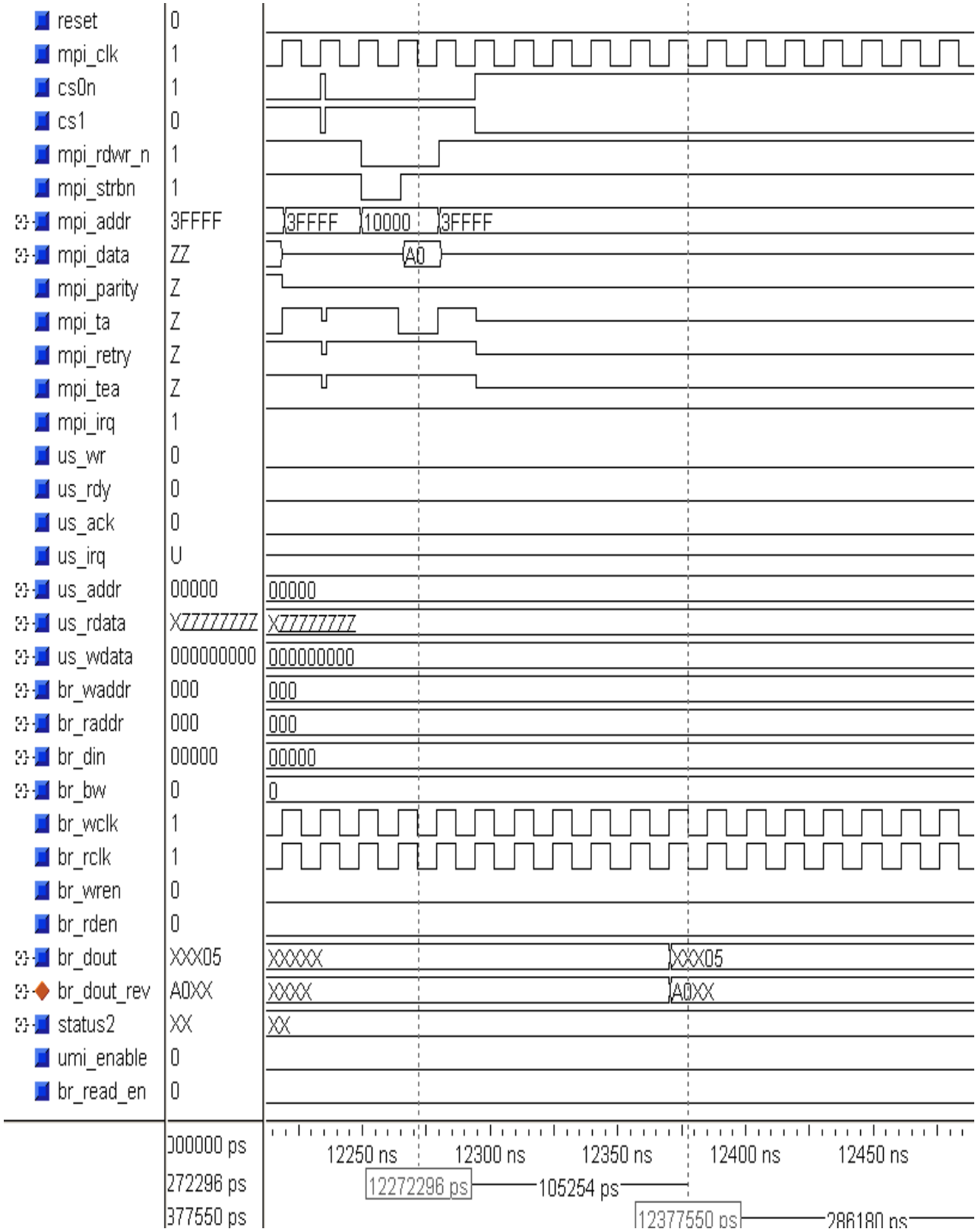


***Figure 50. Functional Simulation Waveform for the MPI Write Transaction at Address 0x08001 with Data AA, in Example 3***

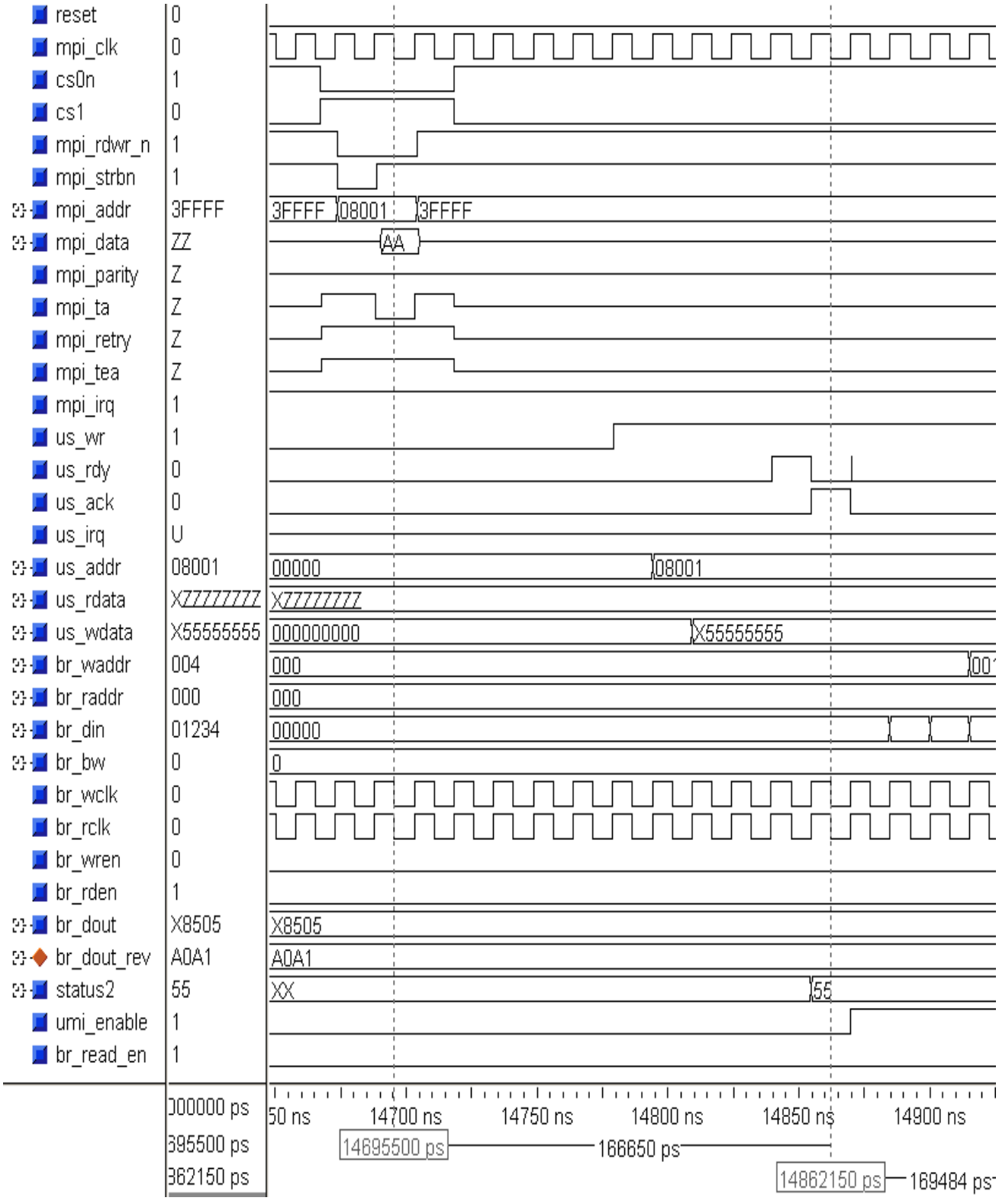
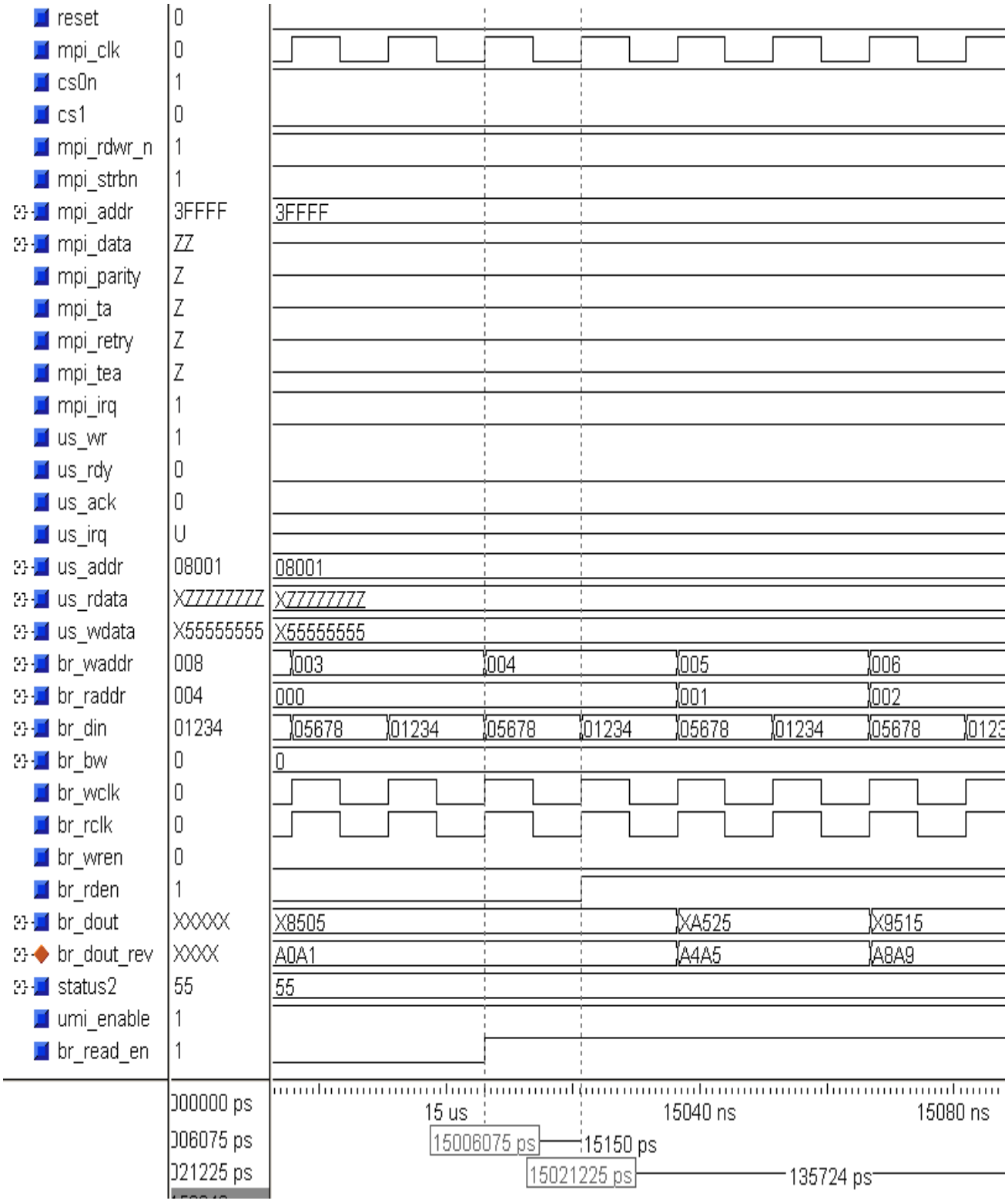


***Figure 51. Functional Simulation Waveform for Read Transactions of the System Bus Block RAM with the FPGA Logic, in Example 3***



## Technical Support Assistance

Hotline: 1-800-LATTICE (Domestic)
1-408-826-6002 (International)
e-mail: techsupport@latticesemi.com