# IOBUS IP

# SPECIFICATION

REVISIONS

| Rev. | Date | Description |
|---|---|---|
| A | 2014/11/17 | Release |
| A.1 | 2014/11/26 | Minor changes |
| A.2 | 2014/12/04 | Minor changes |
| A.3 | 2015/02/17 | Minor changes for the IP on the GEB-VME Board |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

| | | | | | |
|---|---|---|---|---|---|
| Emission | 2015/02/17 | PRG | Vittorio Di Paolo | RSQ | |

# CONTENTS

# 1. INTRODUCTION

The **IOBUS** IP is a dynamically reconfigurable asynchronous bus master designed by GEB Enterprise. It can be used with either fixed latency or undetermined latency slaves. In this document, the IP will be described thoroughly; moreover, the example FPGA design, driver and software provided in conjunction with the IP, will be analyzed.

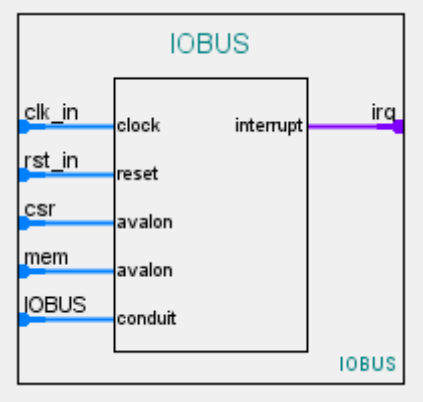
# 2. IP DESCRIPTION

The IOBUS IP is written in VHDL and has been deeply simulated within the Aldec Active-HDL environment. It is also compliant with the Altera Avalon Specification, so it can be easily added to any Altera Qsys design.

## 2.1. IOBUS IP interfaces

The IOBUS IP shows 5 interfaces, as can be seen in the following figure:

**Figure 1: IOBUS IP interfaces.**




These interfaces will be analyzed in detail in the following sections.

### *2.1.1. Clock and reset signals*

**Table 1: Clock and reset signals.**

| Name | Direction | Type | Description |
|---|---|---|---|
| clk | Input | std_logic | Input clock |
| reset_n | Input | std_logic | Active-low reset signal |

As in any synchronous digital design, every action is performed on the rising edge of the clock signal, but only if the reset signal is not asserted. In that case, every signal in the component is set to its reset value.

### *2.1.2. CSR interface*

The CSR interface is used for configuring and for getting the status of the component. Its interface is described in the table below:

**Table 2: CSR interface.**

| Name | Direction | Type | Description |
|---|---|---|---|
| avs_csr_read_n | Input | std_logic | Active-low read signal |
| avs_csr_readdata | Output | std_logic _vector(63 downto 0) | Read data bus |
| avs_csr_write_n | Input | std_logic | Active-low write signal |
| avs_csr_writedata | Input | std_logic _vector(63 downto 0) | Write data bus |

The CSR interface is an Altera Avalon MM Slave with fixed latency (1 cycle for reads and 0 cycles for writes). See the Altera Avalon Specification for further details.

### *2.1.3. MEM interface*

The MEM interface is an Altera Avalon MM Slave used for starting IOBUS cycles. Table 3 describes the MEM interface thoroughly:

**Table 3: MEM interface.**

| Name | Direction | Type | Description |
|---|---|---|---|
| avs_mem_address | Input | std_logic _vector(8 downto 0) | Address bus |
| avs_mem_read_n | Input | std_logic | Active-low read signal |
| avs_mem_readdata | Output | std_logic_vector(31 downto 0) | Read data bus |
| avs_mem_write_n | Input | std_logic | Active-low write signal |
| avs_mem_writedata | Input | std_logic _vector(31 downto 0) | Write data bus |
| avs_mem_be_n | Input | std_logic_vector(3 downto 0) | Byte-enable signal |
| avs_mem_waitrequest | Output | std_logic | Waitrequest signal |

The cycle is started when the Avalon Master asserts the read or the write signal and ends when the waitrequest signal is asserted for one clock period. See the Altera Avalon Specification for further details.

### *2.1.4. IOBUS interface*

The IOBUS is an asynchronous bus used for communication between a master and an unspecified number of slaves. The following table lists and describes the IOBUS interface:

**Table 4: IOBUS interface.**

| Name | Direction | Type | Description |
|---|---|---|---|
| D | Bidirectional | std_logic_vector(31 downto 0) | Bidirectional data |
| A | Output | std_logic_vector(7 downto 0) | 32-bit Register Address |
| CSL | Output | std_logic_vector(1 downto 0) | 2 active-low chipselects |
| RDL | Output | std_logic | Active-low read strobe |
| WRL | Output | std_logic | Active-low write strobe |
| WAIT | Input | std_logic | Used for undetermined latency slaves |
| INT | Input | std_logic | Interrupt request |

The IOBUS cycle is started when the address is put on the bus and one of the two chipselect signals is asserted. After a setup time, either the read or the write signal is asserted for a specified time, after which it is de-asserted. The chipselect signal is then de-asserted after a specified hold time. See section 2.2 for further details on the IOBUS protocol.

### *2.1.5. IRQ interface*

The IRQ interface is an Altera Avalon Interrupt Sender interface made by a single signal named "irq". When the IOBUS INT signal is asserted and the interrupts are enabled in the CSR, the "irq" is asserted. See the Altera Avalon Specification for further details.

## 2.2. IOBUS functional description

As stated before, the IOBUS IP is composed of several interfaces, among which we can find the CSR interface. This interface is used to access the Control Status Register (called CSR from now on) for writing and reading. The following table shows the CSR fields along with their description:

**Table 5: Control Status Register.**

| Offset | Length | Name | Access | Description |
|---|---|---|---|---|
| 0 | 1 | IE | R/W | Interrupt Enable (write '1' to enable interrupts) |
| 1 | 1 | IP | R/W | Interrupt Polarity ('0' for active-low , '1' for active-high interrupts) |
| 2 | 1 | IRQ | RO | Interrupt Request ('1' when an interrupt is pending, '0' otherwise) |
| 3 | 1 | WE | R/W | Wait Enable (write '1' to enable the WAIT signal) |
| 4 | 1 | WP | R/W | Wait Polarity ('0' for active-low , '1' for active-high WAIT signal) |
| 5 | 3 | RSU | R/W | Read Setup Time |
| 8 | 3 | RHD | R/W | Read Hold Time |
| 11 | 3 | WSU | R/W | Write Setup Time |
| 14 | 3 | WHD | R/W | Write Hold Time |
| 17 | 5 | RPW | R/W | Read Pulse Width (not used if WE is asserted) |
| 22 | 5 | WPW | R/W | Write Pulse Width (not used if WE is asserted) |

The last 6 fields are used for the timing of the component, which is based on an internal clock (derived from the PCIe bus reference clock) whose period is 16 ns. In fact, the formula for calculating the actual time is `(VALUE+1)·CLK_PERIOD`. This means that, for each of them, the minimum time is equal to one clock period, that is 16 ns.

IMPORTANT: The CSR configuration is the first thing to do after each power-up and reset of the component.

In the same way, the CSR can be read back after having written it for checking if it was written correctly, or for checking if a new interrupt is pending.

After the CSR is properly configured, the user can start IOBUS cycles by means of the MEM interface. That is, for each MEM cycle, the data is properly translated and a new IOBUS cycle is initiated: when this ends, the waitrequest signal of the MEM interface is asserted for one clock cycle, and the MEM cycle ends.

IMPORTANT: the 9-bit MEM address is translated to the 8-bit IOBUS address, plus one of the two CSL signals is asserted.

For example, if the MEM address is "010101010", then the IOBUS address will be "10101010" and the CSL(0) signal will be asserted. Instead, if the MEM address is "101010101" , then the IOBUS address will be "01010101" and the CSL(1) signal will be asserted.

The following figures show some example configurations for the CSR and their resulting timings:

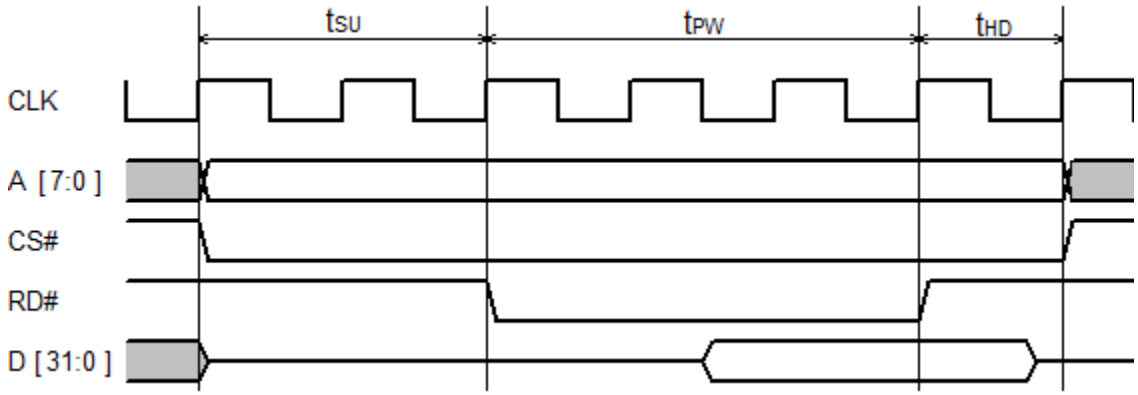



**Figure 2: Fixed latency read example.**

In this example, the following values were chosen:

- WE = '0'
- RSU = “001”
- RHD = “000”
- RPW = “0010”

So, the read setup time resulted to be 2, the pulse width 3 and the hold time 1 clock periods long.

Firstly, the address is put on the A bus and the proper CSL bit is asserted. Then, after the setup time, the read strobe is asserted. At this point, after a time which depends on the IOBUS slave, the data is put on the D bus. Its value is then captured when the read strobe is de-asserted, which happens when the read pulse time is over. The de-assertion of the read strobe causes the IOBUS slave to leave the D bus by putting it in high-impedance. Finally, after a read hold time, the chip-select signal is de-asserted and the address bus is released. The data read is then put on the MEM bus and the waitrequest signal is asserted, so that the master is notified that the cycle is over.

The same happens for a write cycle:

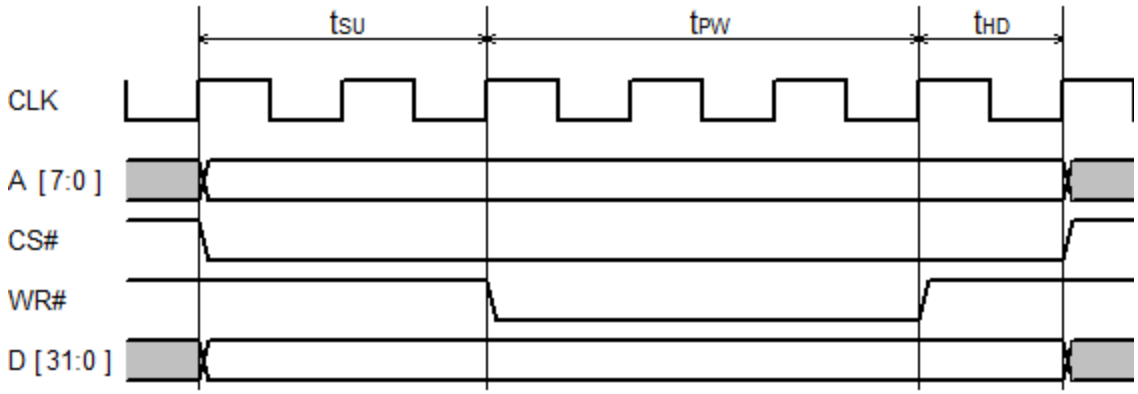



**Figure 3: Fixed latency write example.**

In this case, we have the following values:

- WE = '0'
- WSU = “001”
- WHD = “000”
- WPW = “0010”

So, the read setup time resulted to be 2, the pulse width 3 and the hold time 1 clock periods long.

The only difference is that the data to write is put on the D bus, and the write strobe is asserted instead of the read strobe.

When the WE field is written to '1', that means that we want the IOBUS slave to tell the master when the cycle is over, by means of the WAIT signal:

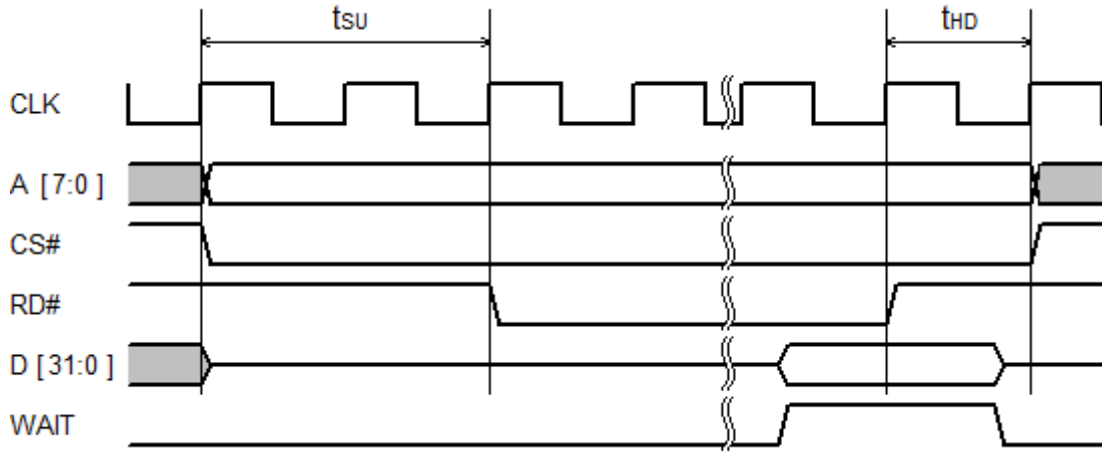



**Figure 4: Undefined pulse width read example.**

In this case, we have the following CSR configuration:

- WE = '0'
- WP = '1'
- RSU = “001”
- RHD = “000”

After the read strobe is asserted, the master waits until the WAIT signal is asserted by the slave (that is, its value is equal to the WP field value): when this happens, it de-asserts the read strobe on the next rising edge of the clock and gets the D bus value. Finally, after a hold time, it de-asserts the chip-select signal too. As stated before, the RPW and WPW fields of the CSR are unused when the WE field is set to '1'.

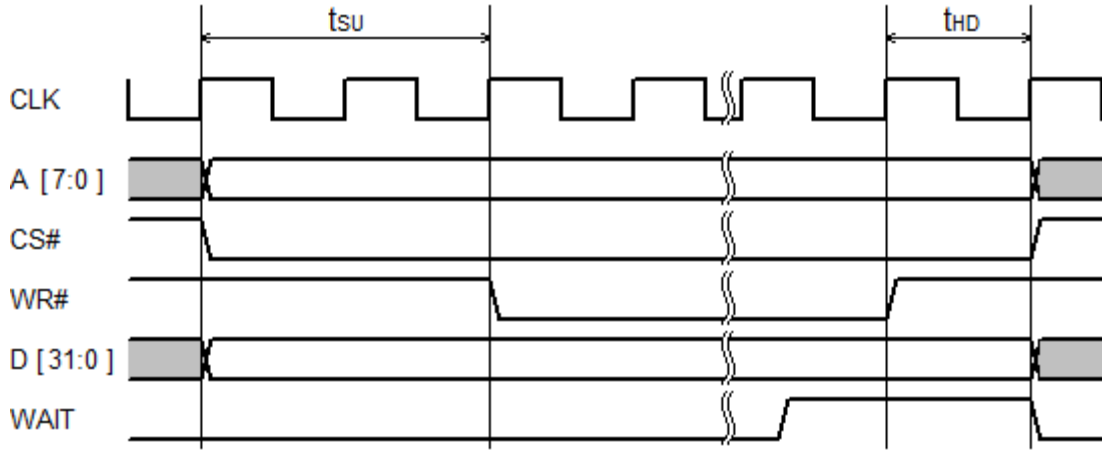



**Figure 5: Undefined pulse width write example.**

The same applies to a write cycle, apart from the CSR fields used:

- WE = '0'
- WP = '1'
- WSU = “001”
- WHD = “000”

Another difference is that obviously the D bus is written at the beginning of the cycle instead of being read.

IMPORTANT: Be sure that the IOBUS slave is correctly designed to properly generate the WAIT signal, or otherwise the master will wait forever and the system will hang!

### *2.2.1. Interrupt management*

As stated before, when interrupts are enabled in the CSR, and the IOBUS INT pin value is equal to the IP field of the CSR, the "irq" pin of the IRQ interface is asserted. At this point, if interrupts are enabled in the master of the MEM interface, then this is interrupted and eventually performs some actions to handle the exception.

IMPORTANT: always remember to clear the source of the interrupt in your ISR. As well, be sure to design your slave properly, the way that its interrupt sources can be cleared by means of an IOBUS cycle. Otherwise, the system will be stuck in the ISR forever!