MicroBlaze™ Hardware Reference Guide

March 2002

XILINX®

MicroBlaze™ Hardware Reference Guide

The following table shows the revision history for this document.

	Version	Revision
10/15/01	1.9	Initial MDK (MicroBlaze Development Kit) release.
1/14/02	2.1	MDK 2.1 release
3/02	2.2	MDK 2.2 release

Contents

Figures

The MicroBlaze Architecture

MicroBlaze Bus Interfaces

OPB Usage in Xilinx FPGAs

Microprocessor Peripheral Definition Format

On-Chip Peripheral Bus (OPB) Arbiter Design Specification

OPB Simple Interrupt Controller Specification

OPB External Memory Controller (EMC)

OPB Block RAM (BRAM) Specification

OPB ZBT Controller Design Specification

OPB UART Lite Specification

OPB JTAG_UART Specification

OPB Serial Peripheral Interface (SPI) Design Specification

OPB General Purpose Input/Output (GPIO) Specification

OPB Timebase WDT Specification

OPB Timer/Counter Specification

Tables

The MicroBlaze Architecture

MicroBlaze Bus Interfaces

OPB Usage in Xilinx FPGAs

Microprocessor Hardware Specification (MHS) Format

Microprocessor Peripheral Definition Format

On-Chip Peripheral Bus (OPB) Arbiter Design Specification

OPB Simple Interrupt Controller Specification

OPB External Memory Controller (EMC)

OPB Block RAM (BRAM) Specification

OPB ZBT Controller Design Specification

OPB UART Lite Specification

OPB JTAG_UART Specification

OPB Serial Peripheral Interface (SPI) Design Specification

OPB General Purpose Input/Output (GPIO) Specification

OPB Timebase WDT Specification

OPB Timer/Counter Specification

Overview of MicroBlaze Embedded Systems

An embedded system built around MicroBlaze™ is comprised of the following:

- MicroBlaze soft processor core
- On-chip block RAM
- Standard bus interconnects
- On-chip Peripheral Bus (OPB) peripherals.

A MicroBlaze system can range from a processor core with a minimum of local memory to a large system with many MicroBlaze processors, sizable external memory, and numerous OPB peripherals. MicroBlaze applications can range from software-based simple state machines to complex controllers for Internet appliances or other embedded applications.

Architecture Support

You can use MicroBlaze systems in the following FPGA devices:

- Virtex™ / Virtex-E / Virtex-II / Virtex-II PRO
- Spartan-II™

MicroBlaze Soft Processor Core

The MicroBlaze soft processor core is central to the MicroBlaze embedded system. This fast, efficient, 32-bit RISC processor includes the following features:

- Orthogonal instruction set
- 32 general purpose registers
- Separate instruction and data buses (Harvard architecture)
- Built-in interfaces to fast on-chip memory and to IBM's industry-standard On-chip Peripheral Bus (OPB)
- Implementations in Virtex-II and later devices support hardware multiply

Bus Interconnects

The data side and instruction side bus interfaces each have an interface to local memory (called the Local Memory Bus, or LMB) and an interface to IBM's On-chip Peripheral Bus (OPB). You can build systems that strictly adhere to a Harvard architecture, or, to share resources, you can use a single OPB in conjunction with a bus arbiter (provided as a MicroBlaze peripheral). Since system requirements differ, the MicroBlaze core is provided in six variations that supply only the LMB and OPB buses needed by your application.

The LMB bus provides guaranteed single-cycle access to on-chip block RAM. This simple, efficient, single-master bus protocol is ideal for interfacing to fast local memory. The OPB is a 32-bit wide multi-master bus that is ideal for connecting peripherals and external memory to the MicroBlaze processor core.

OPB Peripherals

OPB peripherals complete the MicroBlaze hardware system and provide functions such as the following:

- Watchdog timer
- General purpose timer/counters
- Interrupt controller
- UARTs
- General purpose I/O
- Memory controllers.

In addition, you can define and add peripherals for custom functions, or as an interface to a design residing in the FPGA.

The MicroBlaze Architecture

March 2002

Summary

This document describes the architecture for the MicroBlaze™ 32-bit soft processor core.

Overview

The MicroBlaze embedded soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx field programmable gate arrays (FPGAs). See Figure 1 for a block diagram depicting the MicroBlaze core.

Features

The MicroBlaze embedded soft core includes the following features:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- Separate 32-bit instruction and data buses that conform to IBM's OPB (On-chip Peripheral Bus) specification
- Separate 32-bit instruction and data buses with direct connection to on-chip block RAM through a LMB (Local Memory Bus)
- 32-bit address bus
- Single issue pipeline
- Hardware multiplier (in Virtex-II and subsequent devices)

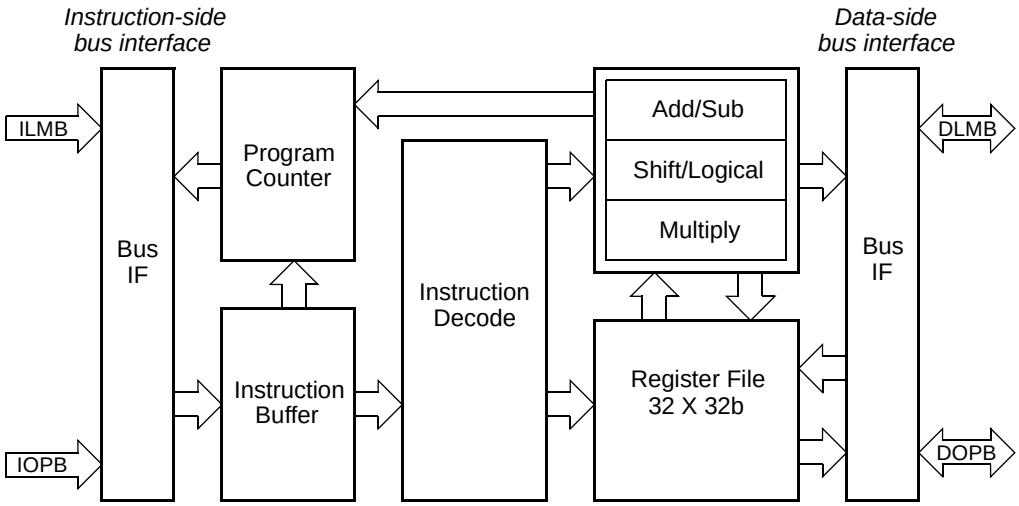


Figure 1: **MicroBlaze Core Block Diagram**

Instructions

All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit immediate operand (which can be extended to 32 bits by preceding the Type B instruction with an IMM instruction). Type B instructions have a single destination register operand. Instructions are provided in the following functional categories: arithmetic, logical, branch, load/store, and special. Table 2 lists the MicroBlaze instruction set. Refer to the MicroBlaze Instruction Set Architecture document for more information on these instructions. Table 1 describes the instruction set nomenclature used in the semantics of each instruction.

Table 1: **Instruction Set Nomenclature**

Symbol	Description
Ra	R0 - R31, General Purpose Register, source operand a
Rb	R0 - R31, General Purpose Register, source operand b
Rd	R0 - R31, General Purpose Register, destination operand,
C	Carry flag, MSR[29]
Sa	Special Purpose Register, source operand
Sd	Special Purpose Register, destination operand
s(*x*)	Sign extend argument *x* to 32-bit value
*Addr	Memory contents at location Addr (data-size aligned)

Table 2: **MicroBlaze Instruction Set Summary**

Type A	0-5	6-10	11-15	16-20	21-31	
Type B	**0-5**	**6-10**	**11-15**	**16-31**		**Semantics**
ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	00000000000	Rd := Rb + Ra
RSUB Rd,Ra,Rb	000001	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDC Rd,Ra,Rb	000010	Rd	Ra	Rb	00000000000	Rd := Rb + Ra + C
RSUBC Rd,Ra,Rb	000011	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + C
ADDK Rd,Ra,Rb	000100	Rd	Ra	Rb	00000000000	Rd := Rb + Ra
RSUBK Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDKC Rd,Ra,Rb	000110	Rd	Ra	Rb	00000000000	Rd := Rb + Ra + C
RSUBKC Rd,Ra,Rb	000111	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + C
ADDI Rd,Ra,Imm	001000	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBI Rd,Ra,Imm	001001	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIC Rd,Ra,Imm	001010	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIC Rd,Ra,Imm	001011	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
ADDIK Rd,Ra,Imm	001100	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBIK Rd,Ra,Imm	001101	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIKC Rd,Ra,Imm	001110	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIKC Rd,Ra,Imm	001111	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
MUL Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000000	Rd := Ra * Rb
MULI Rd,Ra,Imm	011000	Rd	Ra	Imm		Rd := Ra * s(Imm)
OR Rd,Ra,Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
AND Rd,Ra,Rb	100001	Rd	Ra	Rb	00000000000	Rd := Ra and Rb

Table 2: **MicroBlaze Instruction Set Summary *(Continued)***

Type A	0-5	6-10	11-15	16-20	21-31	
Type B	**0-5**	**6-10**	**11-15**	**16-31**		**Semantics**
XOR Rd,Ra,Rb	100010	Rd	Ra	Rb	00000000000	Rd := Ra xor Rb
ANDN Rd,Ra,Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{Rb}$
SRA Rd,Ra	100100	Rd	Ra	0000000000000001		Rd := Ra[0], (Ra >> 1); C := Ra[31]
SRC Rd,Ra	100100	Rd	Ra	0000000000100001		Rd := C, (Ra >> 1); C := Ra[31]
SRL Rd,Ra	100100	Rd	Ra	0000000001000001		Rd := 0, (Ra >> 1); C := Ra[31]
SEXT8 Rd,Ra	100100	Rd	Ra	0000000001100000		Rd[0:23] := Ra[24]; Rd[24:31] := Ra[24:31]
SEXT16 Rd,Ra	100100	Rd	Ra	0000000001100001		Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]
MTS Sd,Ra	100101	00000	Ra	110000000000000d		Sd := Ra , where S1 is MSR
MFS Rd,Sa	100101	Rd	00000	100000000000000a		Rd := Sa , where S0 is PC and S1 is MSR
BR Rb	100110	00000	00000	Rb	00000000000	PC := PC + Rb
BRD Rb	100110	00000	10000	Rb	00000000000	PC := PC + Rb
BRLD Rd,Rb	100110	Rd	10100	Rb	00000000000	PC := PC + Rb; Rd := PC
BRA Rb	100110	00000	01000	Rb	00000000000	PC := Rb
BRAD Rb	100110	00000	11000	Rb	00000000000	PC := Rb
BRALD Rd,Rb	100110	Rd	11100	Rb	00000000000	PC := Rb; Rd := PC
BRK Rd,Rb	100110	Rd	01100	Rb	00000000000	PC := Rb; Rd := PC; MSR[BIP] := 1
BEQ Ra,Rb	100111	00000	Ra	Rb	00000000000	if Ra = 0: PC := PC + Rb
BNE Ra,Rb	100111	00001	Ra	Rb	00000000000	if Ra /= 0: PC := PC + Rb
BLT Ra,Rb	100111	00010	Ra	Rb	00000000000	if Ra < 0: PC := PC + Rb
BLE Ra,Rb	100111	00011	Ra	Rb	00000000000	if Ra <= 0: PC := PC + Rb
BGT Ra,Rb	100111	00100	Ra	Rb	00000000000	if Ra > 0: PC := PC + Rb
BGE Ra,Rb	100111	00101	Ra	Rb	00000000000	if Ra >= 0: PC := PC + Rb
BEQD Ra,Rb	100111	10000	Ra	Rb	00000000000	if Ra = 0: PC := PC + Rb
BNED Ra,Rb	100111	10001	Ra	Rb	00000000000	if Ra /= 0: PC := PC + Rb
BLTD Ra,Rb	100111	10010	Ra	Rb	00000000000	if Ra < 0: PC := PC + Rb
BLED Ra,Rb	100111	10011	Ra	Rb	00000000000	if Ra <= 0: PC := PC + Rb
BGTD Ra,Rb	100111	10100	Ra	Rb	00000000000	if Ra > 0: PC := PC + Rb
BGED Ra,Rb	100111	10101	Ra	Rb	00000000000	if Ra >= 0: PC := PC + Rb
ORI Rd,Ra,Imm	101000	Rd	Ra	Imm		Rd := Ra or s(Imm)
ANDI Rd,Ra,Imm	101001	Rd	Ra	Imm		Rd := Ra and s(Imm)
XORI Rd,Ra,Imm	101010	Rd	Ra	Imm		Rd := Ra xor s(Imm)
ANDNI Rd,Ra,Imm	101011	Rd	Ra	Imm		Rd := Ra and $\overline{s(Imm)}$
IMM Imm	101100	00000	00000	Imm		Imm[0:15] := Imm
RTSD Ra,Imm	101101	10000	Ra	Imm		PC := Ra + s(Imm)
RTID Ra,Imm	101101	10001	Ra	Imm		PC := Ra + s(Imm); MSR[IE] := 1
RTBD Ra,Imm	101101	10010	Ra	Imm		PC := Ra + s(Imm); MSR[BIP] := 0
BRID Imm	101110	00000	10000	Imm		PC := PC + s(Imm)
BRLID Rd,Imm	101110	Rd	10100	Imm		PC := PC + s(Imm); Rd := PC
BRAI Imm	101110	00000	01000	Imm		PC := s(Imm)
BRAID Imm	101110	00000	11000	Imm		PC := s(Imm)
BRALID Rd,Imm	101110	Rd	11100	Imm		PC := s(Imm); Rd := PC
BRKI Rd,Imm	101110	Rd	01100	Imm		PC := s(Imm); Rd := PC; MSR[BIP] := 1

Table 2: **MicroBlaze Instruction Set Summary *(Continued)***

Type A	0-5	6-10	11-15	16-20	21-31	
Type B	**0-5**	**6-10**	**11-15**	**16-31**		**Semantics**
BEQI Ra,Imm	101111	00000	Ra	Imm		if Ra = 0: PC := PC + s(Imm)
BNEI Ra,Imm	101111	00001	Ra	Imm		if Ra /= 0: PC := PC + s(Imm)
BLTI Ra,Imm	101111	00010	Ra	Imm		if Ra < 0: PC := PC + s(Imm)
BLEI Ra,Imm	101111	00011	Ra	Imm		if Ra <= 0: PC := PC + s(Imm)
BGTI Ra,Imm	101111	00100	Ra	Imm		if Ra > 0: PC := PC + s(Imm)
BGEI Ra,Imm	101111	00101	Ra	Imm		if Ra >= 0: PC := PC + s(Imm)
BEQID Ra,Imm	101111	10000	Ra	Imm		if Ra = 0: PC := PC + s(Imm)
BNEID Ra,Imm	101111	10001	Ra	Imm		if Ra /= 0: PC := PC + s(Imm)
BLTID Ra,Imm	101111	10010	Ra	Imm		if Ra < 0: PC := PC + s(Imm)
BLEID Ra,Imm	101111	10011	Ra	Imm		if Ra <= 0: PC := PC + s(Imm)
BGTID Ra,Imm	101111	10100	Ra	Imm		if Ra > 0: PC := PC + s(Imm)
BGEID Ra,Imm	101111	10101	Ra	Imm		if Ra >= 0: PC := PC + s(Imm)
LBU Rd,Ra,Rb	110000	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:23] := 0, Rd[24:31] := *Addr
LHU Rd,Ra,Rb	110001	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:15] := 0, Rd[16:31] := *Addr
LW Rd,Ra,Rb	110010	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd := *Addr
SB Rd,Ra,Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[24:31]
SH Rd,Ra,Rb	110101	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[16:31]
SW Rd,Ra,Rb	110110	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd
LBUI Rd,Ra,Imm	111000	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd[0:23] := 0, Rd[24:31] := *Addr
LHUI Rd,Ra,Imm	111001	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd[0:15] := 0, Rd[16:31] := *Addr
LWI Rd,Ra,Imm	111010	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd := *Addr
SBI Rd,Ra,Imm	111100	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[24:31]
SHI Rd,Ra,Imm	111101	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[16:31]
SWI Rd,Ra,Imm	111110	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd

Registers

MicroBlaze is a fully orthogonal architecture. It has thirty-two 32-bit general purpose registers and two 32-bit special purpose registers.

General Purpose Registers (R0-R31)

The thirty-two 32-bit General Purpose Registers are numbered 0 through 31. R0 is defined to always have the value of zero. Anything written to R0 is discarded, and zero is always read.

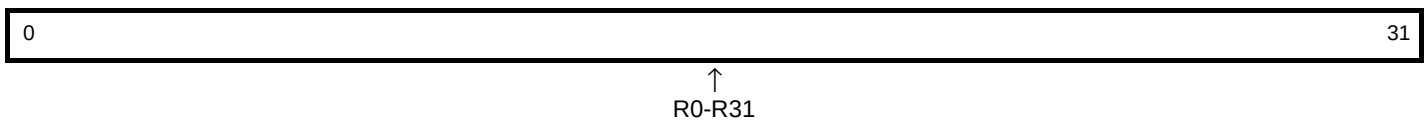


Table 3: **General Purpose Registers (R0-R31)**

Bits	Name	Description	Reset Value
0:31	R0 through R31	**General Purpose Register** R0 through R31 are 32-bit general purpose registers. R0 is always zero.	0x00000000

Special Purpose Registers

Program Counter (PC)

The Program Counter is the 32-bit address of the next instruction word to be fetched. It can be read by accessing RPC with an MFS instruction. It cannot be written to using an MTS instruction.

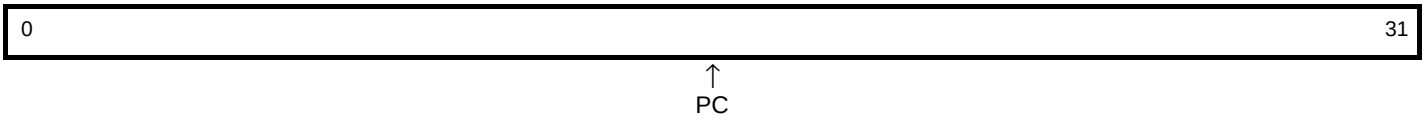


Table 4: **Program Counter (PC)**

Bits	Name	Description	Reset Value
0:31	PC	**Program Counter** Address of next instruction to fetch	0x00000000

Machine Status Register (MSR)

The Machine Status Register contains the carry flag and enables for interrupts and buslock. It can be read by accessing RMSR with an MFS instruction. When reading the MSR, bit 29 is replicated in bit 0 as the carry copy. MSR can be written to with an MTS instruction. Writes to MSR are delayed one clock cycle. When writing to MSR using MTS, the value written takes effect one clock cycle after executing the MTS instruction. Any value written to bit 0 is discarded.

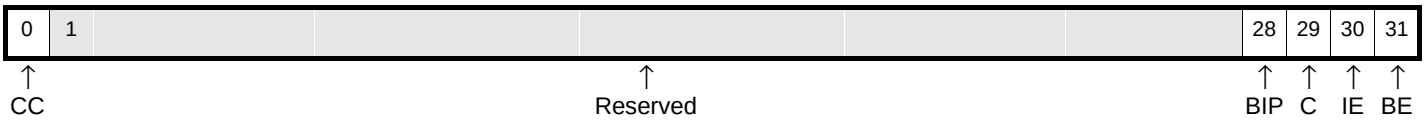


Table 5: **Machine Status Register (MSR)**

Bits	Name	Description	Reset Value
0	CC	**Arithmetic Carry Copy** Copy of the Arithmetic Carry (bit 29). Read only.	0
1:27	Reserved		
28	BIP	**Break in Progress** 0 No Break in Progress 1 Break in Progress Source of break can be software break instruction or hardware break from Ext_Brk or Ext_NM_Brk pin.	0

Table 5: **Machine Status Register (MSR) *(Continued)***

Bits	Name	Description	Reset Value
29	C	**Arithmetic Carry** 0 No Carry (Borrow) 1 Carry (No Borrow)	0
30	IE	**Interrupt Enable** 0 Interrupts disabled 1 Interrupts enabled	0
31	BE	**Buslock Enable** 0 Buslock disabled on data-side OPB 1 Buslock enabled on data-side OPB Buslock Enable does not affect operation of ILMB, DLMB, or IOPB.	0

Pipeline

This section describes the MicroBlaze pipeline architecture.

Pipeline Architecture

The MicroBlaze pipeline is a parallel pipeline, divided into three stages:

- Fetch
- Decode
- Execute

In general, each stage takes one clock cycle to complete. Consequently, it takes three clock cycles (ignoring any delays or stalls) for the instruction to complete.

cycle 1	cycle 2	cycle 3
Fetch	Decode	Execute

In the MicroBlaze parallel pipeline, each stage is active on each clock cycle. Three instructions can be executed simultaneously, one at each of the three pipeline stages. Even though it takes three clock cycles for each instruction to complete, each pipeline stage can work on other instructions in parallel with and in advance of the instruction that is completing. Within one clock cycle, one new instruction is fetched, another is decoded, and a third is completed. The pipeline effectively completes one instruction per clock cycle.

	cycle 1	cycle 2	cycle 3	cycle4	cycle5
instruction 1	Fetch	Decode	Execute		
instruction 2		Fetch	Decode	Execute	
instruction 3			Fetch	Decode	Execute

Branches

Similar to other processor pipelines, the MicroBlaze pipeline can originate control hazards that affect the pipeline execution rate. When an instruction that changes the control flow of a program (branches) is executed and completed, and eventually changes the program flow (taken branches), the previous pipeline work becomes useless. When the processor executes a taken branch, the instructions in the fetch and decode stages are not the correct ones, and must be discarded or flushed from the pipeline. The processor must refill the pipeline with the correct instructions, taking three clock cycles for a taken branch, adding a latency of two cycles for refilling the pipeline.

MicroBlaze uses two techniques to reduce the penalty of taken branches. One technique is to use delay slots and another is use of a history buffer.

Delay Slots

When the processor executes a taken branch and flushes the pipeline, it takes three clock cycles to refill the pipeline. By allowing the instruction following a branch to complete, this penalty is reduced. Instead of flushing the instructions in both the fetch and decode stages, only the fetch stage is discarded and the instruction in the decode stage is allowed to complete. This effectively produces a delayed branch or delay slot. Since the work done on the delay slot instruction is not discarded, this technique effectively reduces the branch penalty from two clock cycles to one. Branch instructions that allow execution of the subsequent instruction in the delay slot are denoted by a D in the instruction mnemonic. For example, the BNE instruction does not execute the subsequent instruction in the delay slot, whereas BNED does execute the next instruction in the delay slot before control is transferred to the branch location.

Load/Store Architecture

MicroBlaze can access memory in the following three data sizes:

- Byte (8 bits)
- Halfword (16 bits)
- Word (32 bits)

Memory accesses are always data-size aligned. For halfword accesses, the least significant address bit is forced to 0. Similarly, for word accesses, the two least significant address bits are forced to 0.

MicroBlaze is a Big-Endian processor and uses the Big-Endian address and labeling conventions shown in Figure 2 when accessing memory. The following abbreviations are used:

- MSByte: Most Significant Byte
- LSByte: Least Significant Byte
- MSBit: Most Significant Bit
- LSBit: Least Significant Bit

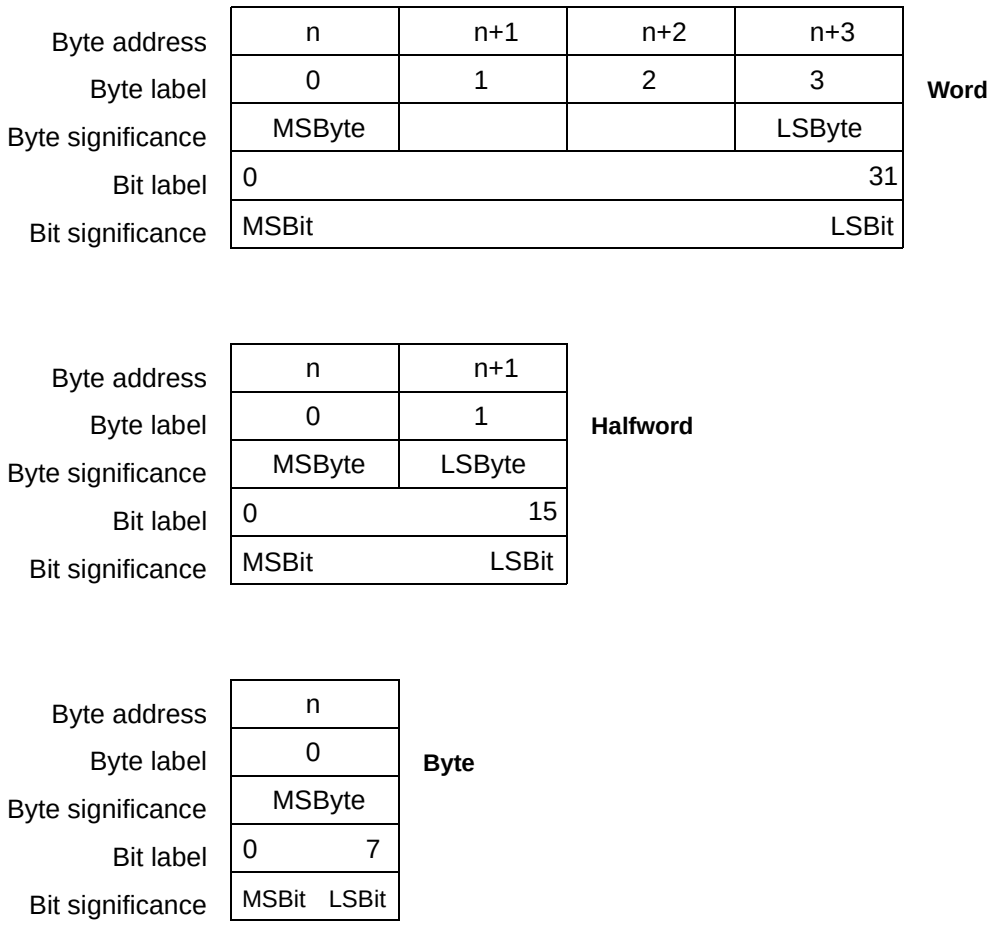


Figure 2: **Big-Endian Data Types**

Interrupts, Exceptions, and Breaks

When a Reset or a Debug_Rst occurs, MicroBlaze starts executing from address 0. PC and MSR are reset to the default values. When an Ext_Brk occurs, MicroBlaze starts executing from address 0x18 and stores the return address in register 16. An Ext_Brk is not executed if the BIP bit in MSR is active (equal to 1). When an Ext_NM_Brk occurs, MicroBlaze starts executing from address 0x18 and stores the return address in register 16. This occurs independent of the BIP bit value in MSR.

Interrupts

When an interrupt occurs, MicroBlaze stops the current execution to handle the interrupt request. MicroBlaze branches to address 0x00000010 and uses the General Purpose Register 14 to store the address of the instruction that was to be executed when the interrupt occurred. It also disables future interrupts by clearing the Interrupt Enable flag in the Machine Status Register (setting bit 30 to 0 in MSR). The instruction located at the address where the current PC points to is not executed. Interrupts do not occur if the BIP bit in the MSR register is active (equal to 1).

Equivalent Pseudocode

```
r14 ← PC
PC ← 0x00000010
MSR[IE] ← 0
```

Exceptions

When an exception occurs, MicroBlaze stops the current execution to handle the exception. MicroBlaze branches to address 0x00000008 and uses the General Purpose Register 17 to store the address of the instruction that was to be executed when the exception occurred. The instruction located at the address where the current PC points to is not executed.

Equivalent Pseudocode

```
r17 ← PC
PC ← 0x00000008
```

Breaks

There are two kinds of breaks:

- Software (internal) breaks
- Hardware (external) breaks

Software Breaks

To perform a software break, use the brk and brki instructions. Refer to the Instruction Set Architecture documentation for more information on software breaks.

Hardware Breaks

Hardware breaks are performed by asserting the external break signal. When a hardware break occurs, MicroBlaze stops the current execution to handle the break. MicroBlaze branches to address 0x00000018 and uses the General Purpose Register 16 to store the address of the instruction that was to be executed when the break occurred. MicroBlaze also disables future breaks by setting the Break In Progress (BIP) flag in the Machine Status Register (setting bit 28 to 1 in MSR). The instruction located at the address where the current PC points to is not executed.

Hardware breaks are only handled when there is no break in progress (the Break In Progress flag is set to 0). The Break In Progress flag has higher precedence than the Interrupt Enabled flag. While no interrupts are handled when the Break In Progress flag is set, breaks that occur when interrupts are disabled are handled immediately. However, it is important to note that non-maskable hardware breaks are always handled immediately.

Equivalent Pseudocode1

```
r16 ← PC
PC ← 0x00000018
MSR[IE] ← 1
```

MicroBlaze Bus Interfaces

March 2002

Summary

This document describes the MicroBlaze™ Local Memory Bus (LMB) and On-chip Peripheral Bus (OPB) interfaces.

Overview

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. Each bus interface unit is further split into a Local Memory Bus (LMB) and IBM's On-chip Peripheral Bus (OPB). The LMB provides single-cycle access to on-chip dual-port block RAM. The OPB interface provides a connection to both on- and off-chip peripherals and memory. .

Features

The MicroBlaze bus interfaces include the following features:

- OPB V2.0 bus interface with byte-enable support (see IBM's *64-Bit On-Chip Peripheral Bus, Architectural Specifications, Version 2.0*)
- LMB provides simple synchronous protocol for efficient block RAM transfers
- LMB provides guaranteed performance of 125 MHz for local memory subsystem

Bus Configurations

The block diagram in Figure 1 depicts the MicroBlaze core with the bus interfaces defined as follows:

DOPB: Data interface, On-chip Peripheral Bus
DLMB: Data interface, Local Memory Bus (BRAM only)
IOPB: Instruction interface, On-chip Peripheral Bus
ILMB: Instruction interface, Local Memory Bus (BRAM only)
Core: Miscellaneous signals (Clock, Reset, Interrupt)

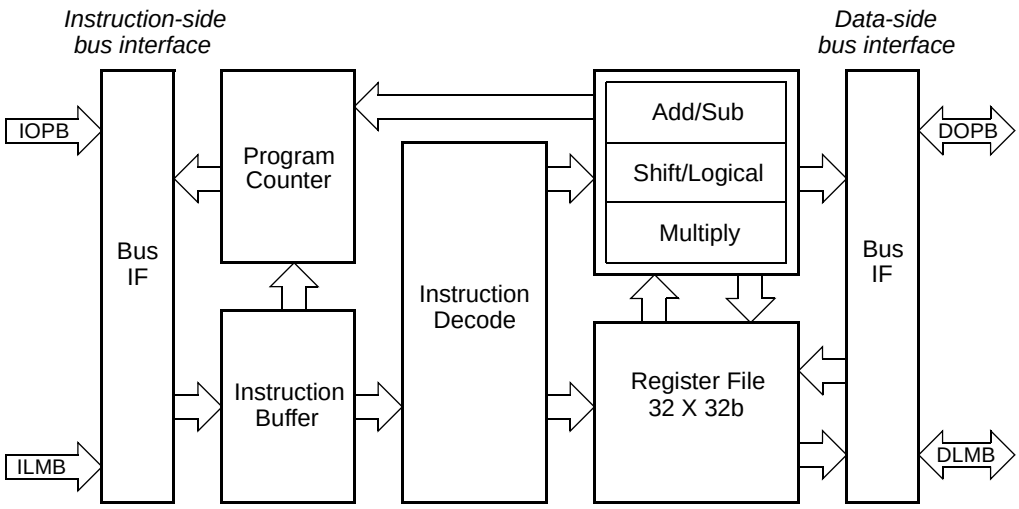


Figure 1: **MicroBlaze Core Block Diagram**

MicroBlaze bus interfaces are available in six configurations, as shown in the following figure.

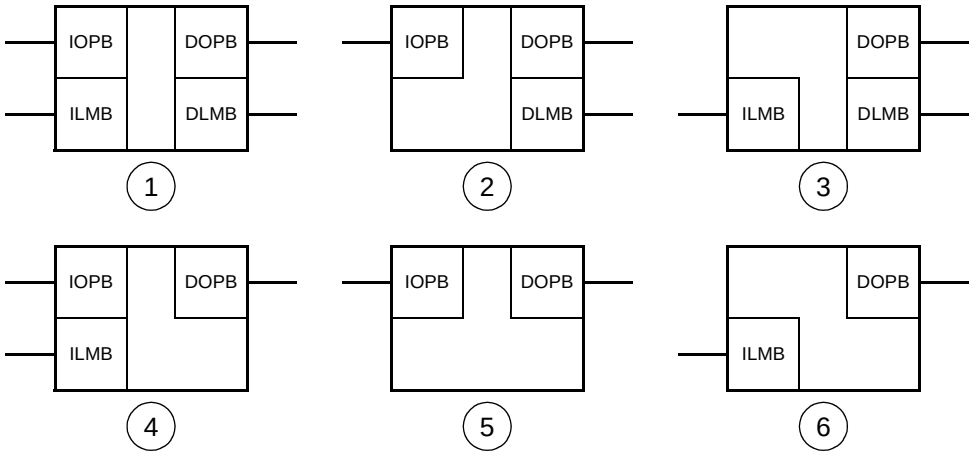


Figure 2: **MicroBlaze Bus Configurations**

The optimal configuration for your application depends on code size and data spaces, and if you require fast access to internal block RAM. The performance implications and supported memory models for each configuration is shown in the following table:

Table 1: **MicroBlaze Bus Configurations**

	Configuration	Core Fmax	Debug available	Memory Models Supported
1	IOPB+ILMB+DOPB+DLMB	110	SW/JTAG	Large external instruction memory, Fast internal instruction memory (BRAM), Large external data memory, Fast internal data memory (BRAM)
2	IOPB+DOPB+DLMB	125	SW/JTAG	Large external instruction memory, Large external data memory, Fast internal data memory (BRAM)
3	ILMB+DOPB+DLMB	125	SW/JTAG	Fast internal instruction memory (BRAM), Large external data memory, Fast internal data memory (BRAM)
4	IOPB+ILMB+DOPB	110	JTAG for ILMB memory[1] SW/for IOPB memory	Large external instruction memory, Fast internal instruction memory (BRAM), Large external data memory,
5	IOPB+DOPB	125	SW/JTAG	Large external instruction memory, Large external data memory,
6	ILMB+DOPB	125	JTAG[1]	Fast internal instruction memory (BRAM), Large external data memory,

1. ILMB memory can be debugged via a software resident monitor if the second port of the dual-ported ILMB BRAM is connected to an OPB BRAM memory controller. See Figure 6 and Figure 8.

Typical Peripheral Placement

This section provides typical peripheral placement and usage for each of the six configurations. Because there are many options for interconnecting a MicroBlaze system, you should use the following examples as *guidelines* for selecting a configuration closest to your application.

Configuration 1

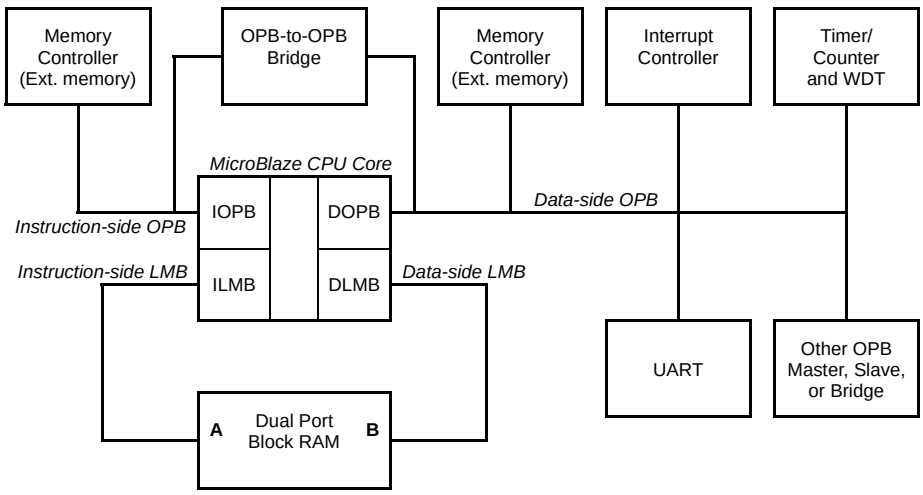


Figure 3: **Configuration 1: IOPB+ILMB+DOPB+DLMB**

Purpose

Use this configuration when your application requires more instruction and data memory than is available in the on-chip block RAM (BRAM). Critical sections of instruction and data memory can be allocated to the faster ILMB BRAM to improve your application's performance. Depending on how much data memory is required, the data-side memory controller may not be present. The data-side OPB is also used for other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes.

Characteristics

Because of the extra logic required to implement two buses per side, the maximum clock rate of the CPU may be slightly less than configurations with one bus per side. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.

Configuration 2

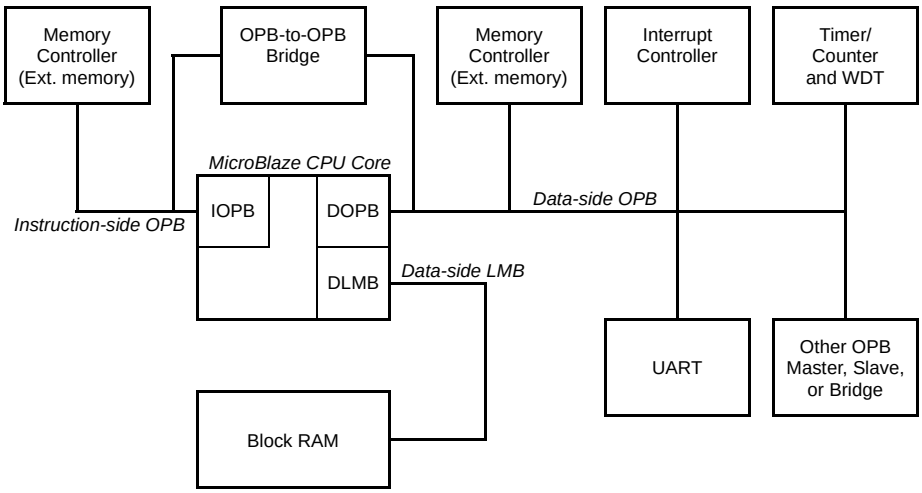


Figure 4: **Configuration 2: IOPB+DOPB+DLMB**

Purpose

Use this configuration when your application requires more instruction and data memory than is available in the on-chip BRAM. In this configuration, all of the instruction memory is resident in off-chip memory or on-chip memory on the instruction-side OPB. Depending on how much data memory is required, the data-side memory controller may not be present. The data-side OPB is also used for other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes.

Characteristics

This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. Instruction fetches on the OPB, however, are slower than fetches from BRAM on the LMB. Overall processor performance is lower than implementations using LMB unless a large percentage of code is run from the internal instruction history buffer. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.

Configuration 3

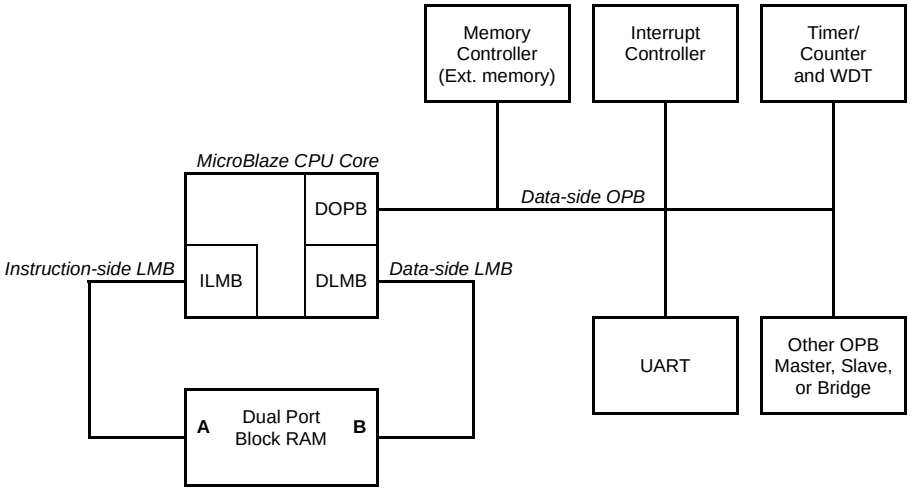


Figure 5: **Configuration 3: ILMB+DOPB+DLMB**

Purpose

Use this configuration when your application code fits into the on-chip BRAM, but more memory may be required for data memory. Critical sections of data memory can be allocated to the faster DLMB BRAM to improve your application's performance. Depending on how much data memory is required, the data-side memory controller may not be present. The data-side OPB is also used for other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals.

Typical Applications

- Data-intensive controllers
- Small to medium state machines

Characteristics

This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. The instruction-side LMB provides two-cycle pipelined read access from the BRAM for an effective access rate of one instruction per clock. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.

Configuration 4

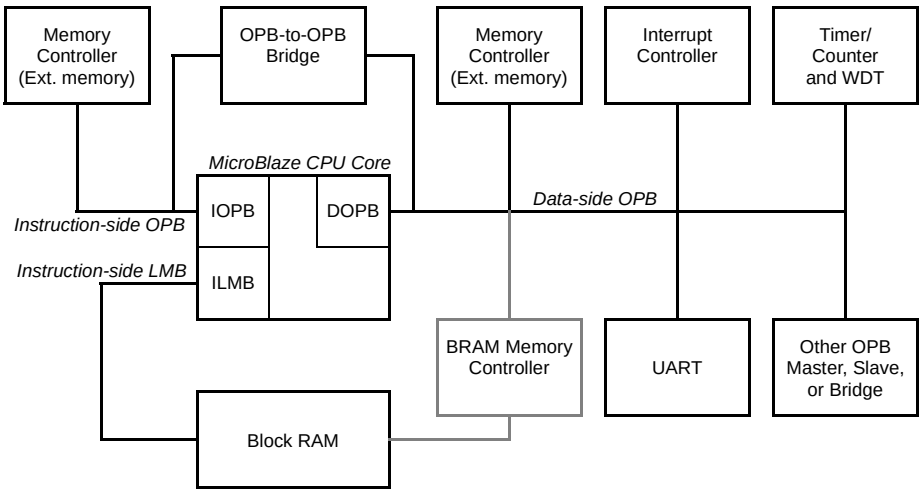


Figure 6: **Configuration 4: IOPB+ILMB+DOPB**

Purpose

Use this configuration when your application requires more instruction and data memory than is available in the on-chip BRAM. Critical sections of instruction memory can be allocated to the faster ILMB BRAM to improve your application's performance. The data-side OPB is used for one or more external memory controllers and other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes

Characteristics

Because of the extra logic required to implement two buses per side, the maximum clock rate of the CPU may be slightly less than configurations with one bus per side. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging. However, software-based debugging of code in the ILMB BRAM can only be performed if a BRAM memory controller is included on the D-side OPB bus to provide write access to the LMB BRAM.

Configuration 5

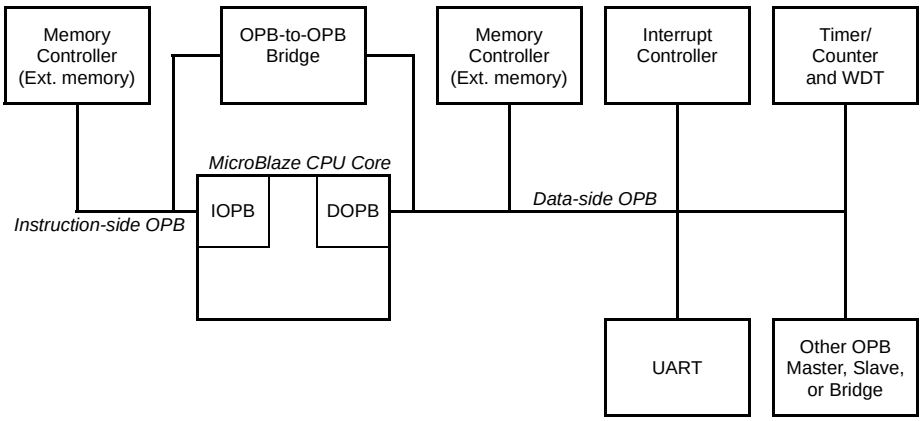


Figure 7: **Configuration 5: IOPB+DOPB**

Purpose

Use this configuration when your application requires external instruction and data memory. In this configuration, all of the instruction and data memory is resident in off-chip memory or on-chip memory on the OPB buses. The data-side OPB is used for one or more external memory controllers and other peripherals such as UARTs, timers, general purpose I/O, BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes

Characteristics

This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. However, instruction fetches on the OPB are slower than fetches from BRAM on the LMB. Overall processor performance is lower than implementations using LMB unless a large percentage of code is run from the internal instruction history buffer. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.

Configuration 6

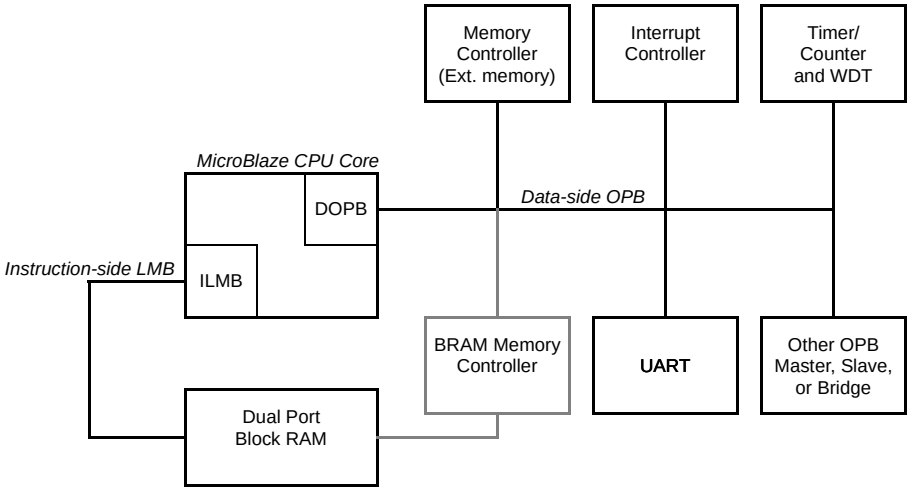


Figure 8: **Configuration 6: ILMB+DOPB**

Purpose

Use this configuration when your application code fits into the on-chip ILMB BRAM, but more memory may be required for data memory. The data-side OPB is used for one or more external memory controllers and other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals.

Typical Applications

- Minimal controllers
- Small to medium state machines

Characteristics

This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. The instruction-side LMB provides two-cycle pipelined read access from the BRAM for an effective access rate of one instruction per clock. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging. However, software-based debugging of code in the ILMB BRAM can only be performed if a BRAM memory controller is included on the D-side OPB bus to provide write access to the LMB BRAM.

Bit and Byte Labeling

The MicroBlaze buses are labeled using a big-endian naming convention. The bit and byte labeling for the MicroBlaze data types is shown in the following figure:

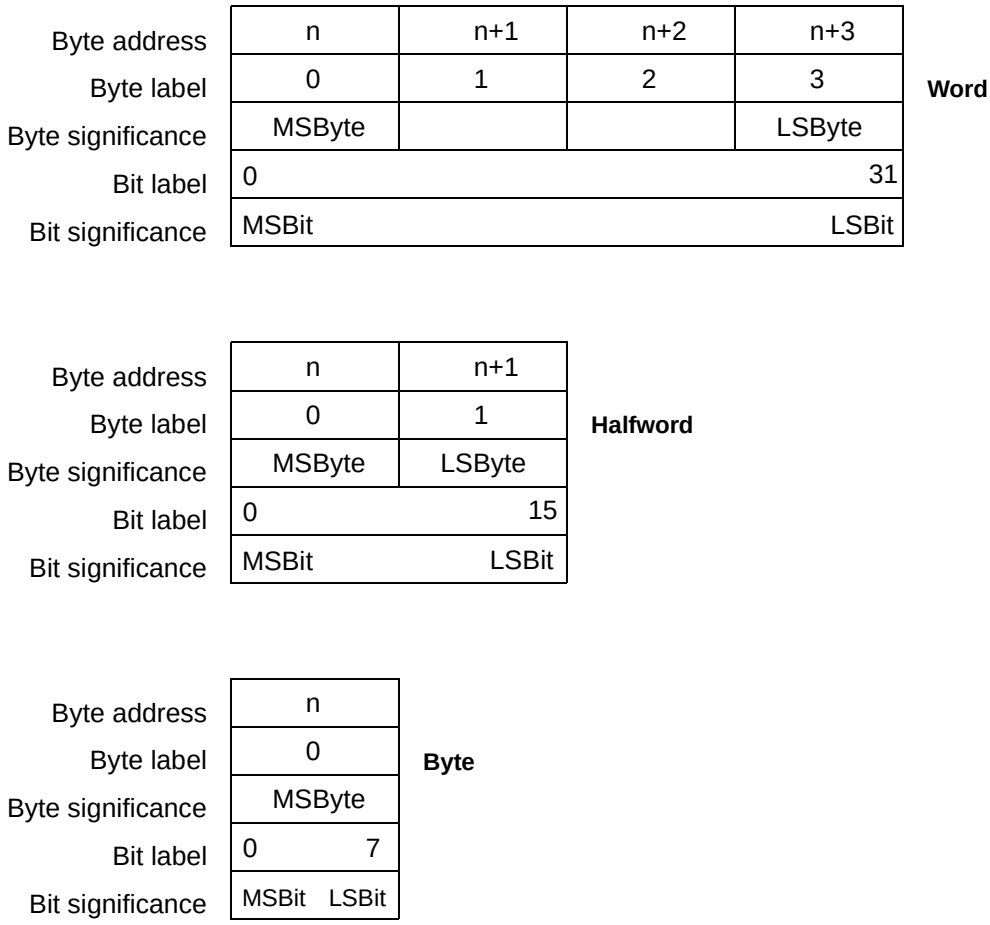


Figure 9: **MicroBlaze Big-Endian Data Types**

Core I/O

The MicroBlaze core implements separate buses for instruction fetch and data access, denoted the I side and D side buses, respectively. These buses are split into the following two bus types:

- OPB V2.0 compliant bus for OPB peripherals and memory controllers
- Local Memory Bus used exclusively for high-speed access to internal block RAM (BRAM).

All core I/O signals are listed in Table 2. Page numbers prefaced by *OPB* reference IBM's *64-Bit On-Chip Peripheral Bus, Architectural Specifications, Version 2.0*.

The core interfaces shown in the following table are defined as follows:

DOPB: Data interface, On-chip Peripheral Bus
DLMB: Data interface, Local Memory Bus (BRAM only)
IOPB: Instruction interface, On-chip Peripheral Bus
ILMB: Instruction interface, Local Memory Bus (BRAM only)
Core: Miscellaneous signals

Table 2: **Summary of MicroBlaze Core I/O**

Signal	Interface	I/O	Description	Page
DM_ABus[0:31]	DOPB	O	Data interface OPB address bus	OPB-11
DM_BE[0:3]	DOPB	O	Data interface OPB byte enables	OPB-16
DM_busLock	DOPB	O	Data interface OPB buslock	OPB-9
DM_DBus[0:31]	DOPB	O	Data interface OPB write data bus	OPB-13
DM_request	DOPB	O	Data interface OPB bus request	OPB-8
DM_RNW	DOPB	O	Data interface OPB read, not write	OPB-12
DM_select	DOPB	O	Data interface OPB select	OPB-12
DM_seqAddr	DOPB	O	Data interface OPB sequential address	OPB-13
DOPB_DBus[0:31]	DOPB	I	Data interface OPB read data bus	OPB-13
DOPB_errAck	DOPB	I	Data interface OPB error acknowledge	OPB-15
DOPB_MGrant	DOPB	I	Data interface OPB bus grant	OPB-9
DOPB_retry	DOPB	I	Data interface OPB bus cycle retry	OPB-10
DOPB_timeout	DOPB	I	Data interface OPB timeout error	OPB-10
DOPB_xferAck	DOPB	I	Data interface OPB transfer acknowledge	OPB-14
IM_ABus[0:31]	IOPB	O	Instruction interface OPB address bus	OPB-11
IM_BE[0:3]	IOPB	O	Instruction interface OPB byte enables	OPB-16
IM_busLock	IOPB	O	Instruction interface OPB buslock	OPB-9
IM_DBus[0:31]	IOPB	O	Instruction interface OPB write data bus (always 0x00000000)	OPB-13
IM_request	IOPB	O	Instruction interface OPB bus request	OPB-8
IM_RNW	IOPB	O	Instruction interface OPB read, not write (tied to ’0’)	OPB-12
IM_select	IOPB	O	Instruction interface OPB select	OPB-12
IM_seqAddr	IOPB	O	Instruction interface OPB sequential address	OPB-13
IOPB_DBus[0:31]	IOPB	I	Instruction interface OPB read data bus	OPB-13
IOPB_errAck	IOPB	I	Instruction interface OPB error acknowledge	OPB-15
IOPB_MGrant	IOPB	I	Instruction interface OPB bus grant	OPB-9
IOPB_retry	IOPB	I	Instruction interface OPB bus cycle retry	OPB-10
IOPB_timeout	IOPB	I	Instruction interface OPB timeout error	OPB-10
IOPB_xferAck	IOPB	I	Instruction interface OPB transfer acknowledge	OPB-12
Data_Addr[0:31]	DLMB	O	Data interface LB address bus	26
Byte_Enable[0:3]	DLMB	O	Data interface LB byte enables	26
Data_Write[0:31]	DLMB	O	Data interface LB write data bus	27
D_AS	DLMB	O	Data interface LB address strobe	27
Read_Strobe	DLMB	O	Data interface LB read strobe	27
Write_Strobe	DLMB	O	Data interface LB write strobe	27
Data_Read[0:31]	DLMB	I	Data interface LB read data bus	27

Table 2: **Summary of MicroBlaze Core I/O *(Continued)***

Signal	Interface	I/O	Description	Page
DReady	DLMB	I	Data interface LB data ready	27
Instr_Addr[0:31]	ILMB	O	Instruction interface LB address bus	26
I_AS	ILMB	O	Instruction interface LB address strobe	27
IFetch	ILMB	O	Instruction interface LB instruction fetch	27
Instr[0:31]	ILMB	I	Instruction interface LB read data bus	27
IReady	ILMB	I	Instruction interface LB data ready	27
Interrupt	Core	I	Interrupt	
Reset	Core	I	Core reset	
Clk	Core	I	Clock	
Debug_Rst	Core	I	Reset signal from OPB JTAG UART	
Ext_BRK	Core	I	Break signal from OPB JTAG UART	
Ext_NM_BRK	Core	I	Non-maskable break signal from OPB JTAG UART	

Bus Organization

OPB Bus Configuration

The MicroBlaze OPB interfaces are organized as byte-enable capable only masters. The byte-enable architecture is an optional subset of the OPB V2.0 specification and is ideal for low-overhead FPGA implementations such as MicroBlaze.

The OPB data bus interconnects are illustrated in Figure 10. The write data bus (from masters and bridges) is separated from the read data bus (from slaves and bridges) to break up the bus OR logic. In minimal cases this can completely eliminate the OR logic for the read or write data buses. Optionally, you can "OR" together the read and write buses to create the correct functionality for the OPB bus monitor. Note that the instruction-side OPB contains a write data bus (tied to 0x00000000) and a RNW signal (tied to logic 1) so that its interface remains consistent with the data-side OPB. These signals are constant and generally are minimized in implementation.

A multi-ported slave is used instead of a bridge in the example shown in Figure 11. This could represent a memory controller with a connection to both the IOPB and the DOPB. In this case, the bus multiplexing and prioritization must be done in the slave. The advantage of this approach is that a separate I-to-D bridge and an OPB arbiter on the instruction side are not required. The arbiter function must still exist in the slave device.

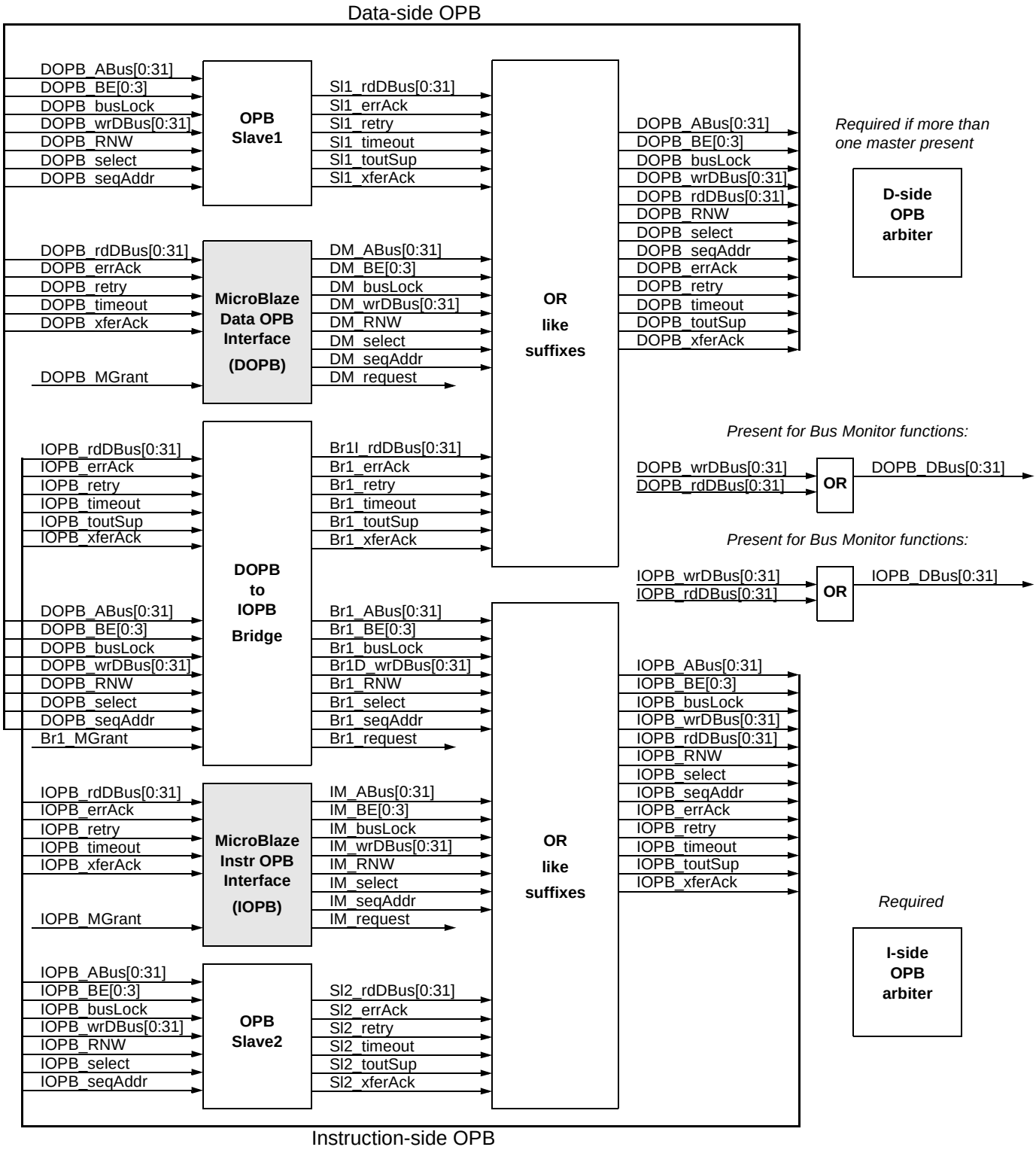


Figure 10: **OPB Interconnection (breaking up read and write buses)**

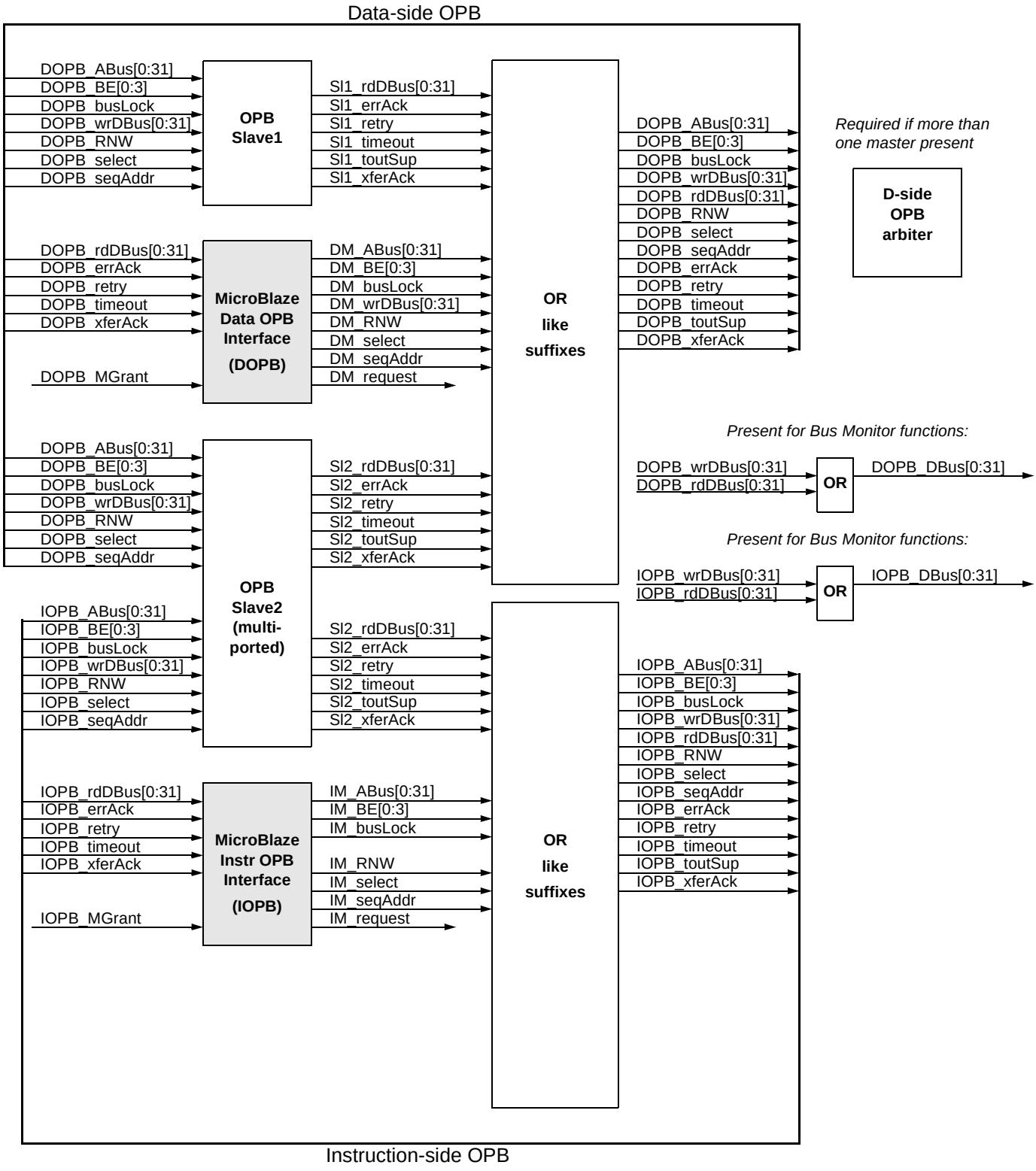


Figure 11: **OPB Interconnection (with multi-ported slave and no bridge)**

LMB Bus Definition

The Local Memory Bus (LMB) is a synchronous bus used primarily to access on-chip block RAM. It uses a minimum number of control signals and a simple protocol to ensure that local block RAM is accessed in a single clock cycle. LMB signals and definitions are shown in the following table. All LMB signals are high true.

Table 3: **LMB Bus Signals**

Signal	Data Interface	Instr. Interface	Type	Description
Addr[0:31]	Data_Addr[0:31]	Instr_Addr[0:31]	O	Address bus
Byte_Enable[0:3]	Byte_Enable[0:3]	*not used*	O	Byte enables
Data_Write[0:31]	Data_Write[0:31]	*not used*	O	Write data bus
AS	D_AS	I_AS	O	Address strobe
Read_Strobe	Read_Strobe	IFetch	O	Read in progress
Write_Strobe	Write_Strobe	*not used*	O	Write in progress
Data_Read[0:31]	Data_Read[0:31]	Instr[0:31]	I	Read data bus
Ready	DReady	IReady	I	Ready for next transfer
Clk	Clk	Clk	I	Bus clock

Addr[0:31]

The address bus is an output from the core and indicates the memory address that is being accessed by the current transfer. It is valid only when AS is high. In multicycle accesses (accesses requiring more than one clock cycle to complete), Addr[0:31] is valid only in the first clock cycle of the transfer.

Byte_Enable[0:3]

The byte enable signals are outputs from the core and indicate which byte lanes of the data bus contain valid data. Byte_Enable[0:3] is valid only when AS is high. In multicycle accesses (accesses requiring more than one clock cycle to complete), Byte_Enable[0:3] is valid only in the first clock cycle of the transfer. Valid values for Byte_Enable[0:3] are shown in the following table:

Table 4: **Valid Values for Byte_Enable[0:3]**

	Byte Lanes Used			
Byte_Enable[0:3]	**Data[0:7]**	**Data[8:15]**	**Data[16:23]**	**Data[24:31]**
0000				
0001				x
0010			x	
0100		x		
1000	x			
0011			x	x
1100	x	x		
1111	x	x	x	x

Data_Write[0:31]

The write data bus is an output from the core and contains the data that is written to memory. It becomes valid when AS is high and goes invalid in the clock cycle after Ready is sampled high. Only the byte lanes specified by Byte_Enable[0:3] contain valid data.

AS

The address strobe is an output from the core and indicates the start of a transfer and qualifies the address bus and the byte enables. It is high only in the first clock cycle of the transfer, after which it goes low and remains low until the start of the next transfer.

Read_Strobe

The read strobe is an output from the core and indicates that a read transfer is in progress. This signal goes high in the first clock cycle of the transfer, and remains high until the clock cycle after Ready is sampled high. If a new read transfer is started in the clock cycle after Ready is high, then Read_Strobe remains high.

Write_Strobe

The write strobe is an output from the core and indicates that a write transfer is in progress. This signal goes high in the first clock cycle of the transfer, and remains high until the clock cycle after Ready is sampled high. If a new write transfer is started in the clock cycle after Ready is high, then Write_Strobe remains high.

Data_Read[0:31]

The read data bus is an input to the core and contains data read from memory. Data_Read[0:31] is valid on the rising edge of the clock when Ready is high.

Ready

The Ready signal is an input to the core and indicates completion of the current transfer and that the next transfer can begin in the following clock cycle. It is sampled on the rising edge of the clock. For reads, this signal indicates the Data_Read[0:31] bus is valid, and for writes it indicates that the Data_Write[0:31] bus has been written to local memory.

Clk

All operations on the LMB are synchronous to the MicroBlaze core clock.

LMB Bus Operations

The following diagrams provide examples of LMB bus operations.

Generic Write Operation

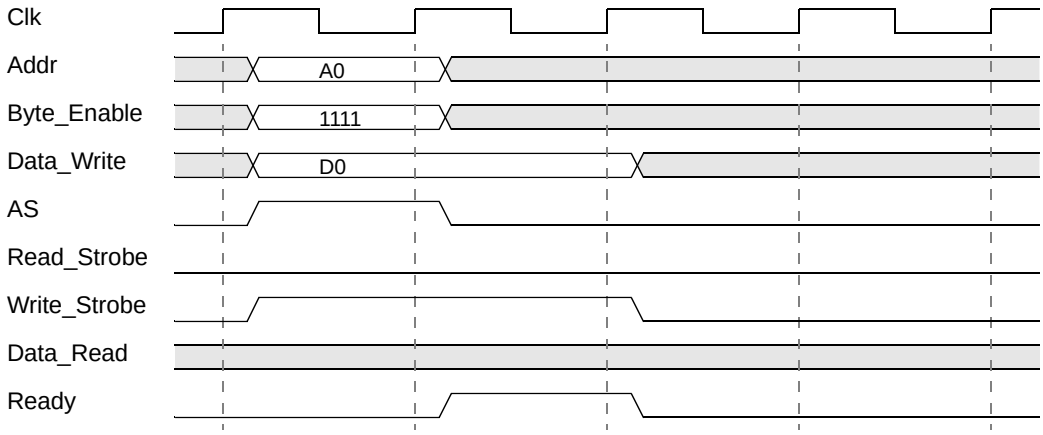


Figure 12: **LMB Generic Write Operation**

Generic Read Operation

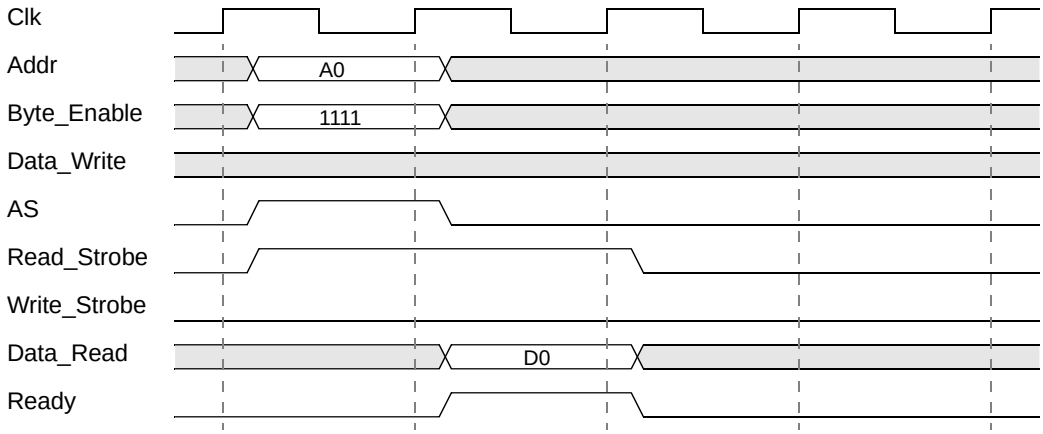


Figure 13: **LMB Generic Read Operation**

Back-to-Back Write Operation (Typical LMB access - 2 clocks per write)

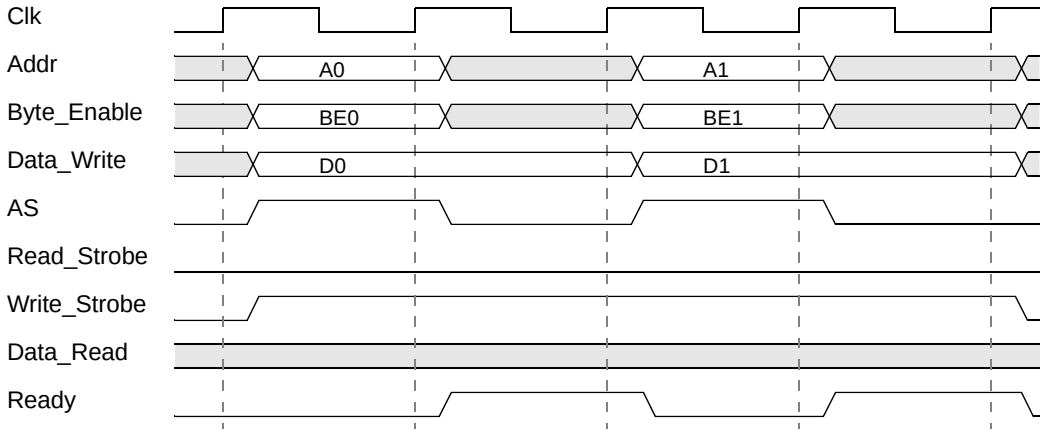


Figure 14: **LMB Back-to-Back Write Operation**

Single Cycle Back-to-Back Read Operation (Typical I-side access - 1 clock per read)

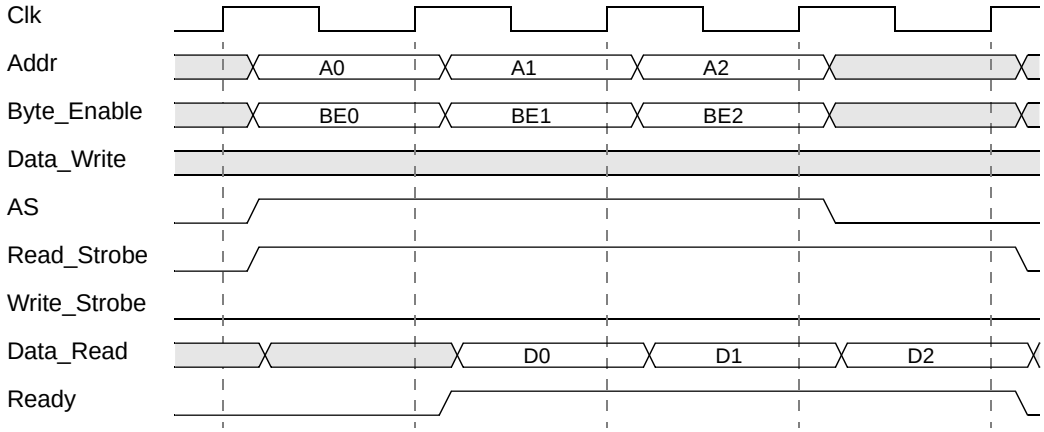


Figure 15: **LMB Single Cycle Back-to-Back Read Operation**

Back-to-Back Mixed Read/Write Operation (Typical D-side timing)

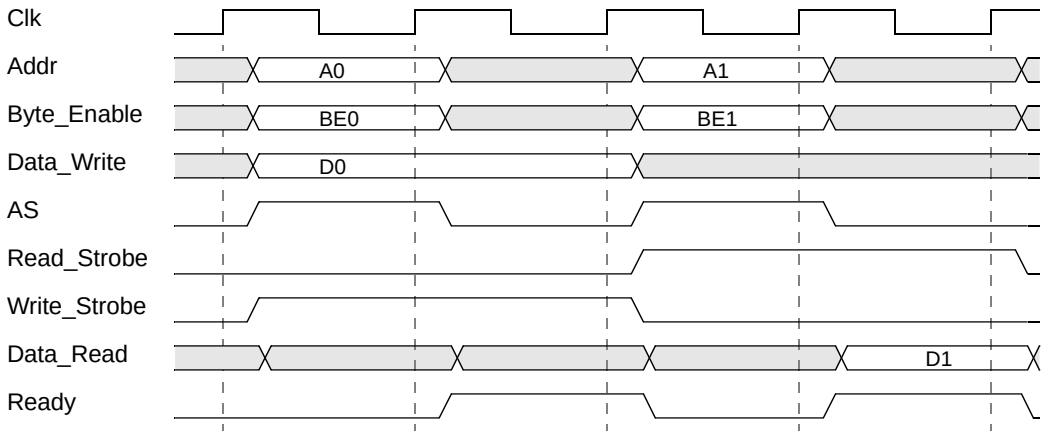


Figure 16: **Back-to-Back Mixed Read/Write Operation**

Read and Write Data Steering

The MicroBlaze data-side bus interface performs the read steering and write steering required to support the following transfers:

- byte, halfword, and word transfers to word devices
- byte and halfword transfers to halfword devices
- byte transfers to byte devices

MicroBlaze does not support transfers that are larger than the addressed device. These types of transfers require dynamic bus sizing and conversion cycles that are not supported by the MicroBlaze bus interface. Data steering for read cycles is shown in Table 5, and data steering for write cycles is shown in Table 6

Table 5: **Read Data Steering (load to Register rD)**

			Register rD Data			
Address [30:31]	**Byte_Enable [0:3]**	**Transfer Size**	**rD[0:7]**	**rD[8:15]**	**rD[16:23]**	**rD[24:31]**
11	0001	byte				Byte3
10	0010	byte				Byte2
01	0100	byte				Byte1
00	1000	byte				Byte0
10	0011	halfword			Byte2	Byte3
00	1100	halfword			Byte0	Byte1
00	1111	word	Byte0	Byte1	Byte2	Byte3

Table 6: **Write Data Steering (store from Register rD)**

			Write Data Bus Bytes			
Address [30:31]	**Byte_Enable [0:3]**	**Transfer Size**	Byte0	Byte1	Byte2	Byte3
11	0001	byte				**rD[24:31]**
10	0010	byte			**rD[24:31]**	
01	0100	byte		**rD[24:31]**		
00	1000	byte	**rD[24:31]**			
10	0011	halfword			**rD[16:23]**	**rD[24:31]**
00	1100	halfword	**rD[16:23]**	**rD[24:31]**		
00	1111	word	**rD[0:7]**	**rD[8:15]**	**rD[16:23]**	**rD[24:31]**

Note that other OPB masters may have more restrictive requirements for byte lane placement than those allowed by MicroBlaze. OPB slave devices are typically attached "left-justified" with byte devices attached to the most-significant byte lane, and halfword devices attached to the most significant halfword lane. The MicroBlaze steering logic fully supports this attachment method.

Implementation

Parameterization

The following characteristics of the MicroBlaze bus interface can be parameterized:

- Data Interface options: OPB only, LMB+OPB
- Instruction Interface options: LMB only, LMB+OPB, OPB only

OPB Usage in Xilinx FPGAs

March 2002

Summary

This document describes how to use the IBM On-chip Peripheral Bus (OPB) in Xilinx FPGAs. This document provides guidelines and simplifications for efficient FPGA implementations, and the subset of signals used in Xilinx-developed OPB devices.

Overview

For detailed information on the IBM OPB, refer to IBM's *On-Chip Peripheral Bus, Architecture Specifications, Version 2.1*:

OpbBus.pdf

The OPB is one element of IBM's CoreConnect architecture, and is a general-purpose synchronous bus designed for easy connection of on-chip peripheral devices. The OPB includes the following features:

- 32-bit or 64-bit data bus
- Up to 64-bit address
- Supports 8-bit, 16-bit, 32-bit, and 64-bit slaves
- Supports 32-bit and 64-bit masters
- Dynamic bus sizing with byte, halfword, fullword, and doubleword transfers
- Optional Byte Enable support
- Distributed multiplexer bus instead of 3-state drivers
- Single cycle transfers between OPB master and OPB slaves (not including arbitration)
- Support for sequential address protocol
- 16-cycle bus time-out (provided by arbiter)
- Slave time-out suppress capability
- Support for multiple OPB bus masters
- Support for bus parking
- Support for bus locking
- Support for slave-requested retry
- Bus arbitration overlapped with last cycle of bus transfers

The OPB is a full-featured bus architecture with many features that increase bus performance. Most of these features map well to the FPGA architecture, however, some can result in the inefficient use of FPGA resources or can lower system clock rates. Consequently, Xilinx uses an efficient *subset* of the OPB for Xilinx-developed OPB devices. However, because of the flexible nature of FPGAs, you can also implement systems utilizing OPB devices that are fully OPB V2.1 compliant.

Xilinx OPB Usage

OPB Options

Legacy Devices

Previous to OPB V2.0, there was a single signaling protocol for OPB data transfers. This protocol (which is also present in OPB V2.0 and later specifications) supports dynamic bus sizing through the use of transfer qualifiers and acknowledge signals. The transfer qualifiers

denote the size of the transfer initiated by the master, and the acknowledge signals indicate the size of the transfer from the slave. Devices that support this type of dynamic bus sizing are called *legacy devices*.

Byte-enable Devices

Starting with OPB V2.0, IBM introduced an optional, alternate transfer protocol based on Byte Enables. In the byte-enable architecture, each byte lane of the data bus has an associated byte enable signal. For each transfer, the byte enable signals indicate which byte lanes have valid data. This eliminates the need for separate transfer qualifiers that indicate the transfer size since all size information is contained in the byte enable signals. The byte-enable architecture, by itself, does not permit dynamic bus sizing, since there is only one acknowledge signal for each transfer. The OPB V2.0 specification (and later) allows you to build systems that are legacy-only, byte-enable only, or mixed. Devices that only support the byte-enable signaling are called *byte-enable devices*.

OPB V2.0 Devices

Devices that support both byte-enable signaling and legacy signaling are called *OPB V2.0 devices*. Systems that have both legacy signaling and byte-enable signaling can perform dynamic bus sizing. Note that legacy devices do not support byte-enable transfers.

Xilinx OPB Devices

These various transfer protocols have several implications for Xilinx OPB device implementations.

Conversion Cycles

Dynamic bus sizing (as supported by legacy devices) results in *conversion cycles*, which are extra transfer cycles that re-transfer data when the master-initiated transfer is larger than the slave response. For example, in a legacy system, if a master writes a 32-bit word to a slave, and the 8-bit device slave responds that it only accepted 8-bits of the transfer, then the master must perform three additional conversion cycles to transfer all of the data to the slave. Generating conversion cycles requires more logic, increases the complexity of the master, and is not an efficient use of FPGA resources. The byte-enable architecture provides a simple alternative to this problem, and is easier to implement in an FPGA.

Write Mirroring and Read Steering

Another consequence of supporting devices smaller than the bus size is *write mirroring* and *read steering*. In the OPB specification, devices smaller than the bus size are always left-justified (aligned toward the most significant side of the bus) so that the byte lanes associated with the smaller devices are easily determined. For example, a byte-wide peripheral is always located on the most-significant byte of the bus. The peripheral writes and reads data using this byte-lane. You can simplify the design of OPB masters by using a byte-enable only, no-write-mirroring architecture. A small degree of added complexity is required for peripherals that are smaller than the bus size if OPB masters do not mirror data.

Ideal FPGA Implementation of OPB-based System

The ideal FPGA implementation of an OPB-based system has the following features:

- Requires no conversion cycles
- Uses only the byte-enable architecture as specified in the OPB specification
- Does not require masters to mirror write data

These characteristics help determine how Xilinx-developed OPB devices are implemented. The detailed specifications that describe how the OPB is used in Xilinx intellectual property are provided in the next section.

Specifications for OPB Usage in Xilinx-developed OPB Devices

Xilinx-developed OPB devices adhere to the following OPB usage rules:

- The width of the OPB data buses and address buses is 32 bits. Note that some peripherals may parameterize these widths, but currently only 32-bit buses are supported. Peripherals that are smaller than 32-bits can be attached to the OPB with a corresponding restriction in addressing. For example, an 8-bit peripheral at base address A can be attached to byte lane 0, but can only be addressed at A, A+4, A+8, etc.
- All OPB devices (masters and slaves) are byte-enable devices. These devices do not support the legacy data transfer signals and therefore do not support dynamic bus sizing. OPB masters do not mirror data to unused byte lanes. See Figure 1 for the byte lane usage for aligned transfers.
- All OPB devices (masters and slaves) are required to output logic zero when they are inactive. This eliminates the need for the Mn_DBusEn and Sln_DBusEn signals external to the master or slave. The enable function is still implemented within the device.
- The byte-enables and the least-significant address bits will be driven by all masters and will contain consistent information. Examples of byte lane usage for aligned transfers are shown below in Figure 1.

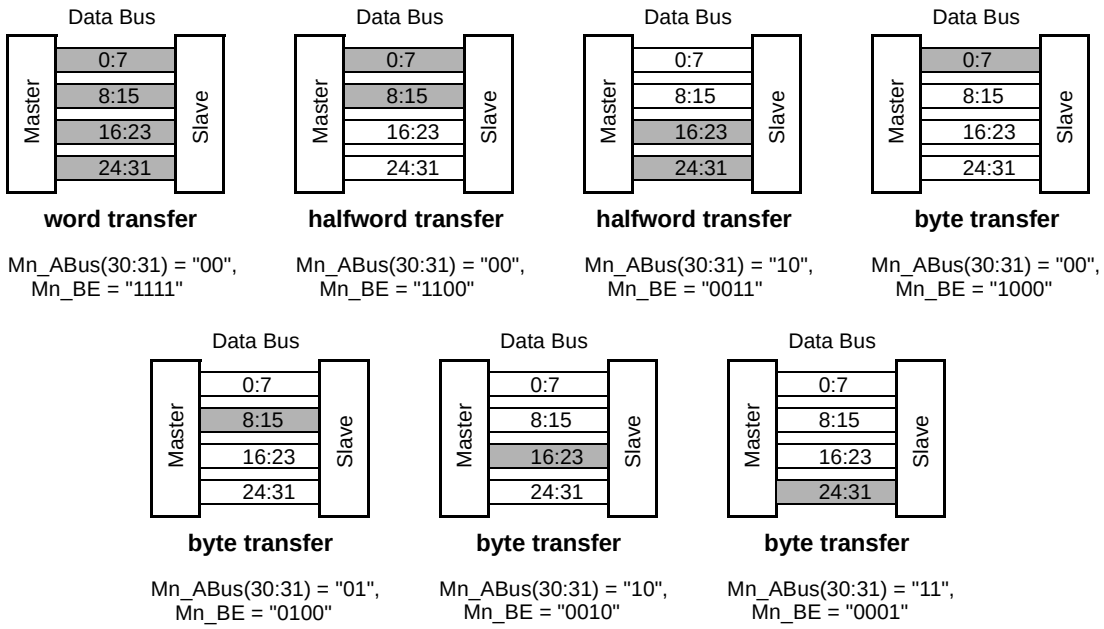


Figure 1: **Byte lane usage for aligned transfers**

- All OPB slave devices that require a continuous address space (i.e. use of all byte lanes) will implement an attachment to the OPB bus that is as wide as the OPB data width, regardless of device width. This eliminates the need for left justification on the OPB bus and eliminates the need for masters to mirror write data. As an example, consider an 8-bit memory device that must be addressed at consecutive byte addresses being attached to a 32-bit OPB. The 8-bit memory device must implement a 32-bit wide attachment to the OPB; in the bus attachment, data is steered from the proper byte lane into the 8-bit device for writes, and from the 8-bit device onto the proper byte lane for reads. The simplest way to accomplish this is with a multiplexer for steering the writes, and a connection from the 8-bit device to all byte lanes (essentially mirroring to all byte lanes) for reads.
- By convention, registers in all OPB slave devices are aligned to word boundaries (lowest two address bits are "00"), regardless of the size of the data in the register or the size of the peripheral.

- Master and Slave I/O: OPB masters will adhere exactly to the signal set shown in Table 1. OPB slaves will adhere exactly to the signal set shown in Table 2. Devices that are both master and slave will adhere exactly to the signal set shown in Table 3. Page numbers referenced in the tables apply to both the OPB V2.0 specification and the OPB V2.1 specification, both from IBM. All signals shown must be present, except for the one signal shown as optional (<Master>_DBus[0:31] for devices that are both master and slave). No additional signals for OPB interconnection may be added. The naming convention is as follows: <Master> represents a master name or acronym that starts with an upper-case letter, <Slave> represents a slave name or acronym that starts with an upper-case letter. <nOPB> represents an OPB identifier (for masters or slaves with more than OPB attachment) and must start with an uppercase letter and end with upper-case "OPB". For devices with a single OPB attachment, the <nOPB> identifier should default to "OPB" (for example, OPB_ABus). All other parts of the signal name must be referenced exactly as shown (including case).

Table 1: **Summary of OPB Master-only I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_Clk	I	OPB Clock	
<nOPB>_Rst	I	OPB Reset	
<Master>_ABus[0:31]	O	Master address bus	OPB-11
<Master>_BE[0:3]	O	Master byte enables	OPB-16
<Master>_busLock	O	Master buslock	OPB-9
<Master>_DBus[0:31]	O	Master write data bus	OPB-13
<Master>_request	O	Master bus request	OPB-8
<Master>_RNW	O	Master read, not write	OPB-12
<Master>_select	O	Master select	OPB-12
<Master>_seqAddr	O	Master sequential address	OPB-13
<nOPB>_DBus[0:31]	I	OPB read data bus	OPB-13
<nOPB>_errAck	I	OPB error acknowledge	OPB-15
<nOPB>_MGrant	I	OPB bus grant	OPB-9
<nOPB>_retry	I	OPB bus cycle retry	OPB-10
<nOPB>_timeout	I	OPB timeout error	OPB-10
<nOPB>_xferAck	I	OPB transfer acknowledge	OPB-14

Table 2: **Summary of OPB Slave-only I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_Clk	I	OPB Clock	
<nOPB>_Rst	I	OPB Reset	
<Slave>_DBus[0:31]	O	Slave data bus	OPB-11
<Slave>_errAck	O	Slave error acknowledge	OPB-15
<Slave>_retry	O	Slave retry	OPB-10
<Slave>_toutSup	O	Slave timeout suppress	OPB-15

Table 2: **Summary of OPB Slave-only I/O *(Continued)***

Signal	I/O	Description	Page (in Ref. 1)
<Slave>_xferAck	O	Slave transfer acknowledge	OPB-14
<nOPB>_ABus[0:31]	I	OPB address bus	OPB-11
<nOPB>_BE	I	OPB byte enable	OPB-16
<nOPB>_DBus[0:31]	I	OPB data bus	OPB-13
<nOPB>_RNW	I	OPB read/not write	OPB-12
<nOPB>_select	I	OPB select	OPB-12
<nOPB>_seqAddr	I	OPB sequential address	OPB-13

Table 3: **Summary of OPB Master/Slave Device I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_Clk	I	OPB Clock	
<nOPB>_Rst	I	OPB Reset	
<Master>_ABus[0:31]	O	Master address bus	OPB-11
<Master>_BE[0:3]	O	Master byte enables	OPB-16
<Master>_busLock	O	Master buslock	OPB-9
<Master>_DBus[0:31]	O	Master write data bus **(optional)**	OPB-13
<Master>_request	O	Master bus request	OPB-8
<Master>_RNW	O	Master read, not write	OPB-12
<Master>_select	O	Master select	OPB-12
<Master>_seqAddr	O	Master sequential address	OPB-13
<nOPB>_DBus[0:31]	I	OPB read data bus	OPB-13
<nOPB>_errAck	I	OPB error acknowledge	OPB-15
<nOPB>_MGrant	I	OPB bus grant	OPB-9
<nOPB>_retry	I	OPB bus cycle retry	OPB-10
<nOPB>_timeout	I	OPB timeout error	OPB-10
<nOPB>_xferAck	I	OPB transfer acknowledge	OPB-14
<Slave>_DBus[0:31]	O	Slave data bus (may optionally function as master write data bus if <Master>_DBus not present)	OPB-11
<Slave>_errAck	O	Slave error acknowledge	OPB-15
<Slave>_retry	O	Slave retry	OPB-10
<Slave>_toutSup	O	Slave timeout suppress	OPB-15
<Slave>_xferAck	O	Slave transfer acknowledge	OPB-14
<nOPB>_ABus[0:31]	I	OPB address bus	OPB-11
<nOPB>_BE	I	OPB byte enable	OPB-16

Table 3: **Summary of OPB Master/Slave Device I/O *(Continued)***

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_RNW	I	OPB read/not write	OPB-12
<nOPB>_select	I	OPB select	OPB-12
<nOPB>_seqAddr	I	OPB sequential address	OPB-13

Notes on the signal sets:

- Xilinx-developed OPB devices do not support dynamic bus sizing and therefore **do not** use the following legacy signals: Mn_dwXfer, Mn_fwXfer, Mn_hwXfer, Sln_dwAck, Sln_fwAck, and Sln_hwAck.
- Since Xilinx-developed OPB devices are byte-enable only, the Mn_beXfer and Sln_beAck signals are not required and so are not used.
- The signals required for masters and slaves are separate from the signals present in the OPB interconnect. The OPB interconnect (the OR gates and other logic required to connect OPB devices) supports the full OPB V2.1 specification (i.e. all signals are present). Thus the OPB interconnect does not limit a design to byte-enable devices and supports designs in which a mix of byte-enable, legacy, and OPB V2.0 devices are present. The bus interconnect will not limit the use of any feature of the V2.1 specification.

Legacy OPB Devices

Although byte-enable devices are the preferred and most efficient OPB devices in Xilinx devices, some designs may also use legacy OPB devices or fully V2.0 compliant devices. However, a legacy device cannot communicate directly with a byte-enable device because they use different signal sets. An interface layer between the byte-enable device and the legacy device is required. This interface is called the Byte Enable Interface (BEIF) device.

Mixed Systems

The system shown below represents a design with a mix of byte-enable, legacy, and OPB V2.0 devices. The BEIF device converts the legacy-type signals to byte-enable-type signals and vice versa.

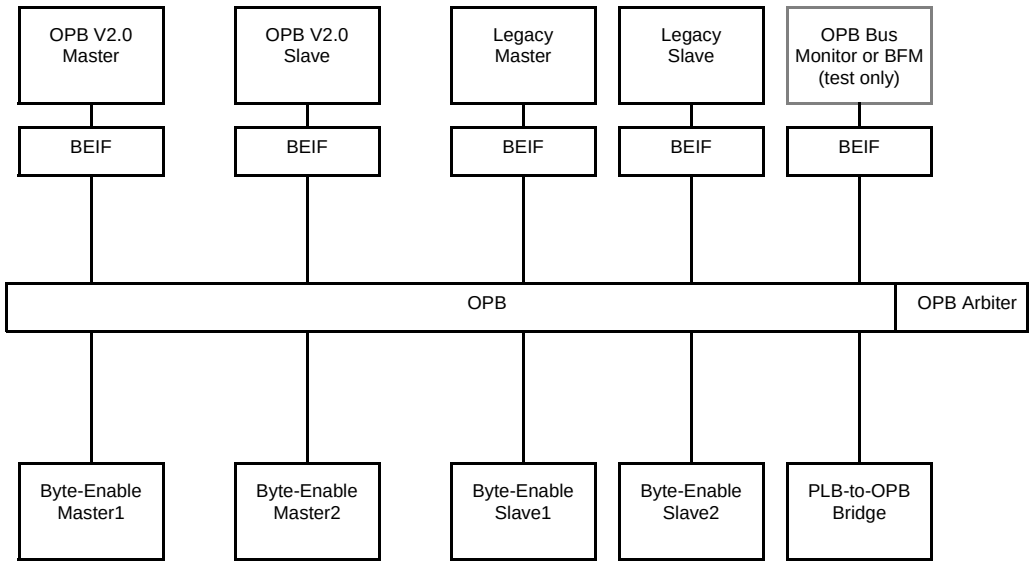


Figure 2: **OPB Interconnect with Mixed Device Types**

The BEIF device contains the following logic, not all of which must be used in all situations:

- Signal translation for byte-enable device to legacy device transfers: **<Master>_BE** is translated to the appropriate <Master>_hwXfer, <Master>_fwXfer, and <Master>_dwXfer. **<nOPB>_BE** is translated to the appropriate <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer. <Slave>_hwXfer, <Slave>_fwXfer, and <Slave>_dwXfer are translated to **<Slave>_xferAck**. <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer are translated to **<nOPB>_xferAck**. The correct lower address bits are also generated.
- Signal translation for legacy device to byte-enable device transfers: <Master>_hwXfer, <Master>_fwXfer, and <Master>_dwXfer are translated to **<Master>_BE** . <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer are translated to **<nOPB>_BE** . **<Slave>_xferAck** is translated to <Slave>_hwXfer, <Slave>_fwXfer, and <Slave>_dwXfer. **<nOPB>_xferAck** is translated to <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer.
- Mirroring and steering logic.
- Conversion cycle generator for byte-enable device to legacy device transfers.

With this architecture, systems that do not require full V2.1 features (for example, systems that contain only Xilinx IP) do not need to instantiate the BEIF and hence optimally use the available FPGA resources. Systems that require legacy or OPB V2.0 devices must instantiate the BEIF, although the most costly part of the BEIF (the conversion cycle generator) only needs to be instantiated if conversion cycles are possible (not all slaves will cause generation of conversion cycles).

OPB Usage Notes

The following are general notes on OPB usage that apply primarily to mixed systems:

- Conversion cycles are only required when a master generates a transfer request to a slave that is larger than the slave's width *and* the slave is capable of indicating that it

accepted a smaller transfer than the master requested hence requiring with a conversion cycle.

- Byte-enable masters cannot directly generate conversion cycles. They require a conversion cycle generator in the Byte Enable Interface (BEIF) device. This is because byte-enable masters do not receive any size information in the acknowledge from the slave.
- Byte-enable slaves cannot cause generation of conversion cycles. A consequence of this is that any master accessing a byte-enable slave can only transfer data up to the size of the slave. Transfers larger than the slave size will result in either 1) no response from the slave (time-out), 2) an errAck from the slave, or 3) lost data; the actual result depends on how the decode and acknowledge logic is implemented in the slave.
- Conversion cycle generator logic in the BEIF is required only for byte-enable device to legacy/OPB V2.0 device transfers.
- Write mirroring and read steering in the V2.1 specification is based on left-justified peripherals. A more complex slave attachment can be used instead of left justification.

OPB Comparison

The following table illustrates the major embedded processor bus architectures used in Xilinx FPGAs and lists some of their characteristics. Each bus has different capabilities in terms of data transfer rates, multi-master capability, and data bursting. The use of a particular bus is dictated by the processor used, the data bandwidth required in the application, and availability of peripherals. The OPB is a general-purpose peripheral bus that can be effectively used in many design situations.

PLB - Processor Local Bus (IBM). **PLB Reference**
OPB - On-chip Peripheral Bus (IBM). **OPB Reference**
OCM - On-chip Memory interface (IBM). **OCM Reference**
LMB - Local Memory Bus (Xilinx). **MicroBlaze Bus Interfaces**
DCR - Device Control Register bus (IBM). **DCR Reference**

Table 4: **Comparison of buses used in Xilinx embedded processor systems**

Feature	CoreConnect Buses			Other Buses	
	PLB	OPB	DCR	OCM	LMB
Processor family	PPC405	PPC405, MicroBlaze	PPC405	PPC405	MicroBlaze
Data bus width	64	32	32	32	32
Address bus width	32	32	10	32	32
Clock rate, MHz (max)[1]	100	125	125	375	125
Masters (max)	8	16	1	1	1
Masters (typical)	2-8	2-8	1	1	1
Slaves (max)	limited only by hardware resources			1	1
Slaves (typical)	2-6	2-8	1-8	1	1
Data rate (peak)[2]	1600 MB/s	500 MB/s	500 MB/s	500 MB/s	500 MB/s
Data rate (typical)[3]	533 MB/s[4]	167 MB/s[5]	100 MB/s[8]	333 MB/s[6]	333 MB/s[7]
Concurrent read/write	Yes	No	No	No	No
Address pipelining	Yes	No	No	No	No
Bus locking	Yes	Yes	No	No	No
Retry	Yes	Yes	No	No	No
Timeout	Yes	Yes	No	No	No
Fixed burst	Yes	No	No	No	No
Variable burst	Yes	No	No	No	No
Cache fill	Yes	No	No	No	No
Target word first	Yes	No	No	No	No
FPGA resource usage	High	Medium	Low	Low	Low
Compiler support for load/store	Yes	Yes	No	Yes	Yes

Notes:

1. Maximum clock rates are estimates and are presented for comparison only. The actual maximum clock rate for each bus is dependent on device family, device speed grade, design complexity, and other factors.
2. Peak data rate is the maximum theoretical data transfer rate at the clock rate shown for each bus.
3. The typical data rates are intended to illustrate data rates that are representative of actual system configurations. The typical data is highly dependent on the application software and system hardware configuration.
4. Assumes primarily cache-line fills, minimal read/write concurrency (66.7% bus utilization).
5. Assumes minimal use of sequential address capabilities and 3 clock cycles per OPB transfer.
6. The OCM controller operates at the PPC405 core clock rate, but its data transfer rate is limited by the access time of the on-chip memory. The typical data rate assumes 66.7% bus utilization.
7. Assumes 66.7% bus utilization.
8. Assumes DCR operates at same clock rate as PLB and each DCR access requires 5 clock cycles. The number of clock cycles per DCR transfer is dependent on how many DCR devices are present in the system. Each additional DCR device adds latency to all DCR transfers.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/17/01	1.0	Initial Xilinx version.
10/19/01	1.1	Minor editorial changes. Added links to bus references.
12/10/01	1.2	Changed Figure 2 and other minor edits.
3/20/02	1.3	Updated for MDK 2.2

March 2002

Microprocessor Hardware Specification (MHS) Format

Summary

This document describes the Microprocessor Hardware Specification (MHS) format for MicroBlaze.

Overview

In the initial phase of MicroBlaze platform design, you create an MHS (Microprocessor Hardware Specification) file that is used by the Platform Generator. This file defines your platform configuration, and includes the following:

- Peripherals
- One of six configurations of the MicroBlaze bus interfaces
- Connectivity of the system
- Address space

MHS Syntax

In general, MHS file syntax is case insensitive, however, signal and attribute names are case sensitive. Attribute settings in the MHS file have priority over the equivalent attribute setting in the Microprocessor Peripheral Definition (MPD) file. Refer to the *Microprocessor Peripheral Definition Format* document for more information on MPD file syntax.

Comments

You can insert comments in the MHS file without disrupting processing. Comments begin with a pound sign (#) and continue to the end of the line.

Peripheral Type

There are two types of peripherals:

- master
- slave

Peripheral names are in lower-case.

Use the following format at the beginning of a peripheral definition:

```
SELECT peripheral_type peripheral_name
```

Use the following format for a master peripheral:

```
SELECT master peripheral_name
```

Use the following format for a slave peripheral:

```
SELECT slave peripheral_name
```

Assignment Type

There are two types of assignments:

- attribute
- signal

Use the following format for assignment statements:

```
CSET assignment_type name = value
```

Use the following format for attributes:

```
CSET attribute name = value
```

Attribute names are case-sensitive.

Use the following format for signals:

```
CSET signal name = connection
```

Signal names are case-sensitive.

Ending a Peripheral Definition

Use the following format to end a peripheral definition:

```
END
```

MHS Example

The following is an example MHS file:

```
SELECT bus opb_v20
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = myopb
CSET signal OPB_Clk = sys_clk
CSET signal SYS_Rst = sys_reset
END
SELECT slave opb_bram
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = mybram1
CSET attribute C_HIGHADDR = 0xFFFF1FFF
CSET attribute C_BASEADDR = 0xFFFF0000
CSET signal OPB_Clk = sys_clk
END
SELECT SLAVE opb_uartlite
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = myuartlite1
CSET attribute C_HIGHADDR = 0xFFFF80FF
CSET attribute C_BASEADDR = 0xFFFF8000
CSET signal RX = rx1
CSET signal TX = tx1
CSET signal OPB_Clk = sys_clk
CSET signal Interrupt = int_periph, PRIORITY=1
END
SELECT SLAVE opb_arbiter
CSET attribute HW_VER = 1.02.b
CSET attribute INSTANCE = myarbiter1
CSET attribute C_HIGHADDR = 0x00FF90FF
CSET attribute C_BASEADDR = 0x00FF9000
CSET attribute C_PARK = 0
CSET attribute C_PROC_INTRFCE = 0
CSET attribute C_REG_GRANTS = 1
CSET signal OPB_Clk = sys_clk
END
SELECT SLAVE opb_timer
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = mytimer1
CSET attribute C_HIGHADDR = 0xFFFFA0FF
CSET attribute C_BASEADDR = 0xFFFFA000
CSET signal CaptureTrig0 = CaptureTrig0
CSET signal CaptureTrig1 = CaptureTrig1
CSET signal CompareOut0 = CompareOut0
CSET signal CompareOut1 = CompareOut1
CSET signal PWM0 = PWM0
CSET signal OPB_Clk = sys_clk
```

```
CSET signal Interrupt = int_periph, PRIORITY=2
END
SELECT SLAVE opb_gpio
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = mygpio1
CSET attribute C_HIGHADDR = 0xFFFFC0FF
CSET attribute C_BASEADDR = 0xFFFFC000
CSET signal GPIO_IO = External_IO
CSET signal OPB_Clk = sys_clk
END
SELECT SLAVE opb_intc
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = myintc1
CSET attribute C_HIGHADDR = 0xFFFFD0FF
CSET attribute C_BASEADDR = 0xFFFFD000
CSET signal Irq = Interrupt
CSET signal Int = int_periph
CSET signal OPB_Clk = sys_clk
END
SELECT SLAVE opb_zbt_controller
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = myzbt1
CSET attribute C_BASEADDR = 0xFF800000
CSET attribute C_HIGHADDR = 0xFF8000FF
CSET attribute C_ZBT_ADDR_SIZE = 17
CSET signal ZBT_Clk_FB = ZBT_Clk_FB
CSET signal ZBT_Clk = ZBT_Clk
CSET signal ZBT_OE_N = ZBT_OE_N
CSET signal ZBT_ADV_LD_N = ZBT_ADV_LD_N
CSET signal ZBT_LBO_N = ZBT_LBO_N
CSET signal ZBT_CE1_N = ZBT_CE1_N
CSET signal ZBT_CE2_N = ZBT_CE2_N
CSET signal ZBT_CE2 = ZBT_CE2
CSET signal ZBT_RW_N = ZBT_RW_N
CSET signal ZBT_CKE_N = ZBT_CKE_N
CSET signal ZBT_A = ZBT_A
CSET signal ZBT_BW_N = ZBT_BW_N
CSET signal ZBT_IO = ZBT_IO
CSET signal ZBT_IOP = ZBT_IOP
CSET signal OPB_Clk = sys_clk
END
SELECT MASTER microblaze
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = microblaze1
CSET attribute CONFIGURATION = 1
CSET signal Interrupt = Interrupt
CSET signal Clk = sys_clk
CSET attribute C_LM_HIGHADDR = 0x00007fff
CSET attribute C_LM_BASEADDR = 0x00000000
END
```

MHS Peripheral Options

Peripherals defined in the MHS can have the following options:

Table 1: **MHS Peripheral Options**

Option	Values	Default	Definition
CONFIGURATION	1, 2, 3, 4, 5, 6	3	One of six configurations of the MicroBlaze Bus interface
HW_VER	1.00.a	X	Hardware version
INSTANCE		X	User-defined instance name Must be lower-case

CONFIGURATION Option

Use the CONFIGURATION option to set one of the six configurations of the MicroBlaze bus interfaces. Refer to the *MicroBlaze Bus Interfaces* document for more information.

Use the following to set the configuration:

```
CSET attribute CONFIGURATION = value
```

You can use short-hand notation or descriptive notation for the configuration value. For the short-hand notation, use an integer from 1 to 6. For the descriptive notation, use the keywords iopb, dopb, ilmb, and dlmb joined together with underscores.

The following examples show the same configuration:

```
CSET attribute CONFIGURATION = 1 # Short-hand notation
CSET attribute CONFIGURATION = iopb_dopb_ilmb_dlmb # descriptive notation
```

HW_VER Option

Use the HW_VER option to set the hardware version, as shown in the following example:

```
CSET attribute HW_VER = 1.00.a
```

The version is specified as a literal of the form 1.00.a.

INSTANCE Option

Use the INSTANCE option to set the instance name of peripheral. This option is mandatory, and the instance name *must* be specified in lower-case.

```
CSET attribute INSTANCE = my_uart0
```

MHS Signal Options

Signals defined in the MHS file can have the following options:

Table 2: **MHS Signal Options**

Option	Values	Default	Definition
PRIORITY	integer	X	Interrupt priority
TYPE	INTERNAL EXTERNAL	EXTERNAL	Scope of signal

PRIORITY Option

Use the PRIORITY option to set the priority of an interrupt signal:

```
CSET signal Interrupt = interrupt_bus, PRIORITY=num
```

The highest priority is “1”. The *num* value is the priority of the interrupt signal among all interrupts.

TYPE Option

Use the TYPE option to set the scope of a signal:

```
CSET signal signal_name = connection, TYPE=type_value
```

The type value is either EXTERNAL or INTERNAL. By default, only OPB and LMB signals are defined as INTERNAL. All other signals are defined as EXTERNAL.

Design Considerations

This section provides general design considerations.

Defining Memory Size

Memory sizes are based on C_BASEADDR and C_HIGHADDR settings. Use the following format when defining memory size:

```
CSET attribute C_HIGHADDR= 0xFFFF00FF
CSET attribute C_BASEADDR= 0xFFFF0000
```

All memory sizes must be 2^n where n is a positive integer, and 2^n boundary overlaps are not allowed.

Defining Local Memory Size

Local Memory (LM) size is based on C_LM_BASEADDR and C_LM_HIGHDADDR settings, and only predefined sizes of LM are allowed. Otherwise, MUX stages must be used to build bigger memories, and this can slow memory access to LM. For Virtex/Virtex-E/Spartan-II devices, the maximum allowed memory size is 16 KBytes, which uses 32 select BlockRAMs. For Virtex-II and Virtex-II PRO devices, the maximum allowed memory size is 64 KBytes, which also uses 32 select BlockRAMs. Verify that your FPGA device resources can adequately accommodate your local memory instruction and data sizes.

Table 3: **Local Memory Sizes**

Architecture	Memory Size (KBytes)
Spartan-II	2, 4, 8, 16
Virtex	2, 4, 8, 16
Virtex-E	2, 4, 8, 16
Virtex-II	8, 16, 32, 64
Virtex-II PRO	8, 16, 32, 64

Use the following format to define LM size:

```
CSET attribute C_LM_HIGHADDR= 0x00001FFF
CSET attribute C_LM_BASEADDR= 0x00000000
```

LM must begin at address 0x00000000.

Internal Signals

Use the TYPE=INTERNAL attribute to set internal signals, as shown in the following example:

```
CSET signal mysignal = internal_connection, TYPE=INTERNAL
```

By default, only OPB and LMB signals are defined as INTERNAL. All other signals are defined as EXTERNAL. External signals are available through the port-declaration of the top-level module. All points of connection to the internal signal must have the TYPE=INTERNAL option.

Interrupt Signals

Interrupt signals are set with a priority, and the highest priority is “1”. If there is only one interrupt defined in the platform, then you may be able to connect it directly to the MicroBlaze

processor. The MicroBlaze processor's interrupt is level sensitive. Consequently, any other level sensitive interrupt line from a peripheral can be connected directly. However, if the peripheral's interrupt line is edge sensitive, then you must use the interrupt controller. If you connect an edge sensitive signal to a level sensitive signal, you may miss an interrupt.

Use the following format to set interrupt signals:

```
CSET signal mysignal = interrupt_bus, PRIORITY=n
```

Power Signals

Power signals are signals that are constantly driven with either VCC or GND.

Use the following format to set power signals:

```
CSET signal mysignal = power_signal
```

In this example, *power_signal* is either "net_vcc" or "net_gnd". Platform Generator expands "net_vcc" or "net_gnd" to the appropriate vector size.

Microprocessor Peripheral Definition Format

March 2002

Summary

This document describes the Microprocessor Peripheral Definition (MPD) format for MicroBlaze.

Overview

The Platform Generator allows you to partition your peripherals into one or more reusable modules. The MPD file provides peripheral information to the Platform Generator, and has the following characteristics:

- Lists ports and default connectivity for the OPB interface (as defined by IBM). For example, the MPD file can include information that maps a signal UART_xferAck to Sl_xferAck.
- Can contain attributes set by you
- Supplied by the IP provider
- Any MPD option is overwritten by the equivalent MHS assignment (refer to the *Microprocessor Hardware Specification Format* document for more details)
- Individual peripheral documentation contains information on all MPD file options

Depending on your peripheral design, you may need to use Black-Box Description (BBD) or Peripheral Analyze Order (PAO) files. The BBD file manages file locations of optimized hardware netlists for the black-box sections of the peripheral design. The PAO file contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation.

For more information on the Platform Generator, refer to the *MicroBlaze Software Reference Guide*.

Load Path

Refer to Figure 1 for a depiction of the peripheral directory structure. On a UNIX system, the OPB peripherals reside in the following location:

$MICROBLAZE/hw/coregen

On a PC, the OPB peripherals reside in the following location:

%MICROBLAZE%\hw\coregen

To specify additional directories, you can use one of the following options:

- Current directory (where Platform Generator was launched; not where the MHS resides)
- Set the Platform Generator -P option, or the XIL_MYPERIPHERALS environment variable

Platform Generator uses a search priority mechanism to locate peripherals, as follows:

1. Search current directory
2. Search $XIL_MYPERIPHERALS/opb_peripherals (UNIX) or %XIL_MYPERIPHERALS%\opb_peripherals (PC)
3. Search $MICROBLAZE/hw/coregen (UNIX) or %MICROBLAZE%\hw\coregen (PC)

The first two search areas (1 and 2) have the same underlying directory structure. The third search area has the CORE Generator directory structure. For search areas 1 and 2, the peripheral name is the name of the root directory. From the root directory, the underlying directory structure is as follows:

data
hdl
netlist
simmodels

For example, if the XIL_MYPERIPHERALS environment is set, then the MPD, BBD, and PAO files are found in the following location:

$XIL_MYPERIPHERALS/opb_peripherals/<peripheral>/data (UNIX)

%XIL_MYPERIPHERALS%\opb_peripherals\<peripheral>\data (PC)

The VHDL files are found in the following location:

$XIL_MYPERIPHERALS/opb_peripherals/<peripheral>/hdl/vhdl (UNIX)

%XIL_MYPERIPHERALS%\opb_peripherals\<peripheral>\hdl\vhdl (PC)

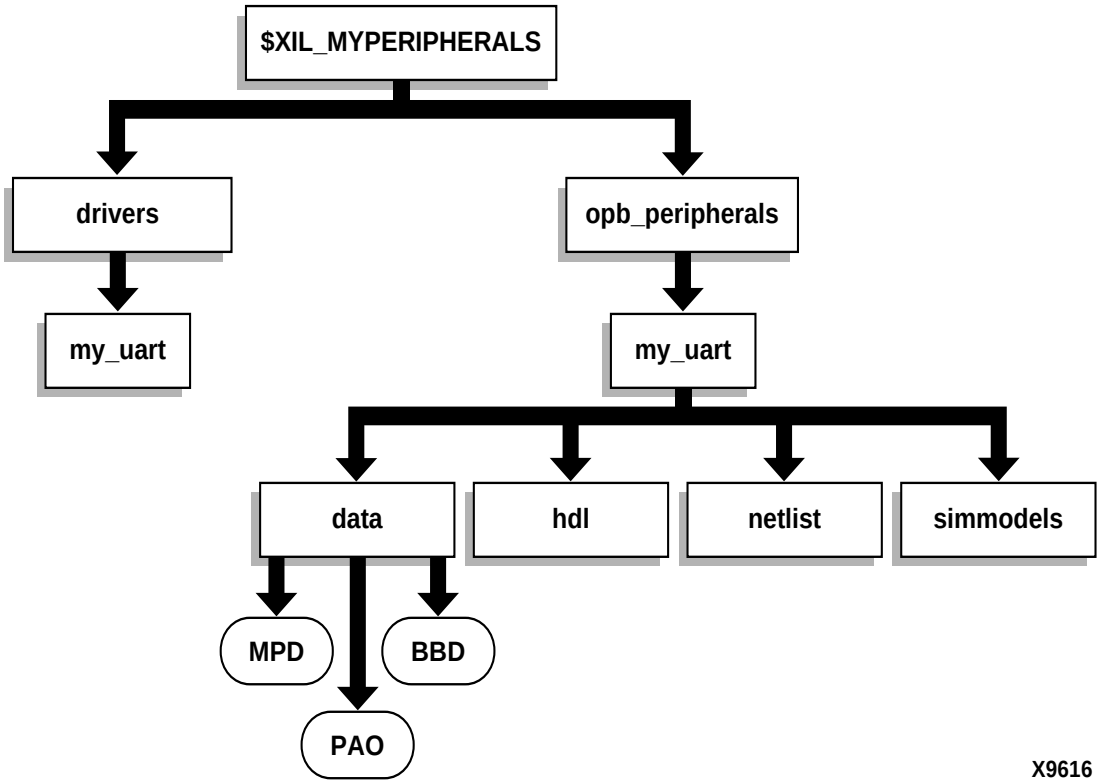


Figure 1: Peripheral Directory Structure

Using Versions

You can create multiple versions of your peripheral. The version is specified as a literal of the form 1.00.a. At the MHS level, use the HW_VER attribute to set the hardware version. The Platform Generator concatenates a "_v" and translates periods to underscores. The peripheral name and HW_VER are joined together to form a name for a search level in the load path. For example, if your peripheral is version 1.00.a, then the MPD, BBD, and PAO files are found in the following location:

$XIL_MYPERIPHERALS/opb_peripherals/<peripheral>_v1_00_a/data (UNIX)

%XIL_MYPERIPHERALS%\opb_peripherals\<peripheral>_v1_00_a\data (PC)

MPD Syntax

The MPD file is supplied by the IP provider and provides peripheral information to the Platform Generator. This file lists ports and default connectivity to the OPB interface. Attributes that you set in this file are mapped to generics for VHDL or parameters for Verilog.

Comments

You can insert comments in the MPD file without disrupting processing. The following are guidelines for inserting comments:

- Precede comments with the pound sign (#)
- Comments can continue to the end of the line
- Comments can be anywhere on the line

Format

Use the following format at the beginning of a peripheral definition:

```
SELECT ip_type peripheral_name
```

Peripheral names are lower-case. The SELECT keyword signifies the beginning of a new peripheral.

Use the following format for assignment statements:

```
CSET type name = value
```

Use the following format to end a peripheral definition:

```
END
```

IP Type

There are six types of IP:

- master
- slave
- master_slave
- bus
- bridge
- ip

Use the following format for a master peripheral:

```
SELECT master peripheral_name
```

Use the following format for a slave peripheral:

```
SELECT slave peripheral_name
```

Peripheral names are in lower-case.

Assignment Type

There are two types of assignments:

- attribute
- signal

Attribute

Use the following format for attributes:

```
CSET attribute name = value, data_type
```

Attribute names are case-sensitive. The MPD syntax requires the attribute data-type for VHDL. This requirement will be eliminated in subsequent releases because Verilog does not require a data-type for its parameters. This will allow the MPD to be written generically without modification for a specific HDL language.

Signal

Use the following format for signals:

```
CSET signal name = default_connection, direction, bus_width
```

Signal names are case-sensitive.

Signal Direction

Signals have three modes. Signal mode indicates its driver direction, and if the port can be read from within the peripheral.

The three modes and their accepted values are as follows:

- input - [input, in, i]
- output - [output, out, o]
- inout - [inout, io]

MPD Example

The following is an example MPD file:

```
SELECT slave opb_gpio
# Generics for vhdl or parameters for verilog
cset attribute C_BASEADDR         = 0x20000000, std_logic_vector
cset attribute C_HIGHADDR         = 0x200000FF, std_logic_vector
cset attribute C_OPB_DWIDTH       = 32, integer
cset attribute C_OPB_AWIDTH       = 32, integer
cset attribute C_GPIO_WIDTH       = 32, integer
cset attribute C_ALL_INPUTS       = 0,  integer
# Global ports
CSET signal OPB_Clk       = "", in
CSET signal OPB_Rst       = OPB_Rst, in
# OPB signals
CSET signal OPB_ABus      = OPB_ABus, in, [0:C_OPB_AWIDTH-1]
CSET signal OPB_BE        = OPB_BE, in,   [0:C_OPB_DWIDTH/8-1]
CSET signal OPB_DBus      = OPB_DBus, in, [0:C_OPB_DWIDTH-1]
CSET signal OPB_RNW       = OPB_RNW, in
CSET signal OPB_select    = OPB_select, in
CSET signal OPB_seqAddr   = OPB_seqAddr, in
CSET signal GPIO_DBus     = Sl_DBus, out, [0:C_OPB_DWIDTH-1]
CSET signal GPIO_errAck   = Sl_errAck, out
CSET signal GPIO_retry    = Sl_retry, out
CSET signal GPIO_toutSup  = Sl_toutSup, out
CSET signal GPIO_xferAck  = Sl_xferAck, out
# gpio signals
CSET signal GPIO_IO       = "", inout, [0:C_GPIO_WIDTH-1], ENABLE=MULTI
END
```

MPD Attribute Naming Conventions

This section provides syntax rules for attribute names for IP and systems. MPD attributes correlate to generics for VHDL or parameters for Verilog. The attribute name must be HDL (VHDL, Verilog) compliant. VHDL and Verilog have certain naming rules and conventions that must be followed.

The Platform Generator automatically expands and populates certain reserved attributes. This can help prevent errors when your peripheral requires information on the platform that is generated. The following table lists the reserved attribute names:

Table 1: **Reserved Peripheral Attribute Names**

Attribute	Description	Automatic Expansion
C_FAMILY	FPGA Device Family	X
C_BASEADDR	Base address of peripheral	
C_HIGHADDR	High address of peripheral	
C_LM_BASEADDR	Base address of Local Memory	
C_LM_HIGHADDR	High address of Local Memory	
C_NUM_MASTERS	Number of masters	X
C_NUM_SLAVES	Number of slaves	X
C_NUM_INTR_INPUTS	Number of interrupt signals	X
C_OPB_AWIDTH	OPB Address width	X
C_OPB_DWIDTH	OPB Data width	X

C_FAMILY Attribute

The C_FAMILY attribute defines the FPGA device family. This attribute is automatically populated by the Platform Generator.

Format

```
CSET attribute C_FAMILY = family, string
```

Where the *family* is spartan2, virtex, virtexe, or virtex2.

C_BASEADDR Attribute

The C_BASEADDR attribute defines the base address of the peripheral. This attribute is not populated by the Platform Generator.

Format

```
CSET attribute C_BASEADDR = base, std_logic_vector(0 to 31)
```

Where *base* is a hexadecimal value.

C_HIGHADDR Attribute

The C_HIGHADDR attribute defines the base address of the peripheral. This attribute is not populated by the Platform Generator.

Format

```
CSET attribute C_HIGHADDR = high, std_logic_vector(0 to 31)
```

Where *high* is a hexadecimal value.

C_NUM_MASTERS Attribute

The C_NUM_MASTERS attribute defines the number of masters in the system. This attribute is automatically populated by the Platform Generator.

Format

```
CSET attribute C_NUM_MASTERS = num, integer
```

Where *num* is an integer value.

C_NUM_SLAVES Attribute

The C_NUM_SLAVES attribute defines the number of slaves in the system. This attribute is automatically populated by the Platform Generator.

Format

```
CSET attribute C_NUM_SLAVES = num, integer
```

Where *num* is an integer value.

C_NUM_INTR_INPUTS Attribute

The C_NUM_INTR_INPUTS attribute defines the number of interrupt inputs. This attribute is automatically populated by the Platform Generator.

Format

```
CSET attribute C_NUM_INTR_INPUTS = num, integer
```

Where *num* is an integer value.

C_OPB_AWIDTH Attribute

The C_NUM_AWIDTH attribute defines the OPB address width. This attribute is automatically populated by the Platform Generator.

Format

```
CSET attribute C_OPB_AWIDTH = num, integer
```

Where *num* is an integer value.

C_OPB_DWIDTH Attribute

The C_NUM_DWIDTH attribute defines the OPB data width. This attribute is automatically populated by the Platform Generator.

Format

```
CSET attribute C_OPB_DWIDTH = num, integer
```

Where *num* is an integer value.

MPD Signal Naming Conventions

This section provides naming conventions for OPB signal names. These conventions are flexible to accommodate MicroBlaze and other systems that have more than one OPB and more than one OPB port per component.

For peripheral OPB ports, the names must start with a capital letter, and must be HDL (VHDL or Verilog) compliant. As with any language, VHDL and Verilog have certain naming rules and conventions that you must follow.

Global Ports

The names for the global ports of a peripheral (such as clock and reset signals) are standardized. The name for the clock signal is OPB_Clk, and the reset signal is OPB_Rst. You can use any name for other global ports (such as the interrupt signal).

Master OPB Ports

OPB V2.0 naming conventions should be followed for that part of the identifier following the last underscore in the name.

OPB Master Outputs

For interconnection to the OPB, all masters must provide the following outputs for each connection to the OPB.

```
<Mn>_ABus
<Mn>_BE
<Mn>_busLock
<Mn>_DBus
<Mn>_request
<Mn>_RNW
<Mn>_select
<Mn>_seqAddr
```

Where <*Mn*> is a meaningful name or acronym for the master output. An additional requirement on <*Mn*> is that it must not contain the string, “OPB” (upper or lower case or mixed case), so that master outputs are not confused with bus outputs.

```
IM_request
Bridge_request
DMAE_request
O2OB_request
```

OPB Master Inputs

For interconnection to the OPB, all masters must provide the following inputs for each connection to the OPB.

```
<nOPB>_DBus
<nOPB>_errAck
<nOPB>_MGrant
<nOPB>_retry
<nOPB>_timeout
<nOPB>_xferAck
```

Where <*nOPB*> is a meaningful name or acronym for the master input. An additional requirement on <*nOPB*> is that the last three characters must contain the string, “OPB” (upper or lower case or mixed case).

```
IOPB_DBus
DOPB_DBus
OPB_DBus
Bus1_OPB_DBus
```

Slave OPB Ports

OPB V2.0 naming conventions should be followed for that part of the identifier following the last underscore in the name.

OPB Slave Outputs

For interconnection to the OPB, all slaves must provide the following outputs for each connection to the OPB:

```
<Sln>_DBus
<Sln>_errAck
<Sln>_retry
<Sln>_toutSup
<Sln>_xferAck
```

Where <*Sln*> is a meaningful name or acronym for the slave output. An additional requirement on <*Sln*> is that it must not contain the string, “OPB” (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.

```
TMR_xferAck
UART_xferAck
INTC_xferAck
MemCon_xferAck
```

OPB Slave Inputs

For interconnection to the OPB, all slaves must provide the following inputs for each connection to the OPB:

```
<nOPB>_ABus
<nOPB>_BE
<nOPB>_DBus
<nOPB>_RNW
<nOPB>_select
<nOPB>_seqAddr
```

Where <*nOPB*> is a meaningful name or acronym for the slave input. An additional requirement on <*nOPB*> is that the last three characters must contain the string, “OPB” (upper or lower case or mixed case).

```
OPB_DBus
IOPB_DBus
DOPB_DBus
Bus1_OPB_DBus
```

MPD Reserved Signal Connections

The Platform Generator establishes connectivity of the OPB and LMB busses to peripherals through a common set of signal connections.

Global Ports

For interconnection to the global ports:

```
CSET signal OPB_Rst = OPB_Rst, in
```

Master OPB Ports

For interconnection to the OPB, all masters must provide the following connections for each connection to the OPB:

```
CSET signal <Mn>_ABus = M_ABus, out, [0:C_OPB_AWIDTH]
CSET signal <Mn>_BE = M_BE, out, [0:C_OPB_DWIDTH/8-1]
CSET signal <Mn>_busLock = M_busLock, out
CSET signal <Mn>_DBus = M_DBus, out, [0:C_OPB_DWIDTH-1]
CSET signal <Mn>_request = M_request, out
CSET signal <Mn>_RNW = M_RNW, out
CSET signal <Mn>_select = M_select, out
CSET signal <Mn>_seqAddr = M_seqAddr, out
CSET signal <nOPB>_DBus = OPB_DBus, in, [0:C_OPB_DWIDTH-1]
CSET signal <nOPB>_errAck = OPB_errAck, in
CSET signal <nOPB>_MGrant = OPB_MGrant, in
CSET signal <nOPB>_retry = OPB_retry, in
CSET signal <nOPB>_timeout = OPB_timeout, in
CSET signal <nOPB>_xferAck = OPB_xferAck, in
```

Slave OPB Ports

For interconnection to the OPB, all slaves must provide the following connections for each connection to the OPB:

```
CSET signal <Sln>_DBus = Sl_DBus, out, [0:C_OPB_DWIDTH-1]
CSET signal <Sln>_errAck = Sl_errAck, out
CSET signal <Sln>_retry = Sl_retry, out
CSET signal <Sln>_toutSup = Sl_toutSup, out
CSET signal <Sln>_xferAck = Sl_xferAck, out
CSET signal <nOPB>_ABus = OPB_ABus, in, [0:C_OPB_AWIDTH-1]
CSET signal <nOPB>_BE = OPB_BE, in, [0:C_OPB_DWIDTH/8-1]
CSET signal <nOPB>_DBus = OPB_DBus, in, [0:C_OPB_DWIDTH-1]
CSET signal <nOPB>_RNW = OPB_RNW, in
CSET signal <nOPB>_select = OPB_select, in
```

```
CSET signal <nOPB>_seqAddr = OPB_seqAddr, in
```

LMB Ports

Only the MicroBlaze processor is allowed interconnection to the LMB. The following connections are reserved:

```
CSET signal Instr_Addr = Instr_Addr, out, [0:31]
CSET signal Instr = Instr, in, [0:31]
CSET signal IFetch = IFetch, out
CSET signal I_AS = I_AS, out
CSET signal IReady = IReady, in
CSET signal Data_Addr = Data_Addr, out, [0:31]
CSET signal Data_Read = Data_Read, in, [0:31]
CSET signal Data_Write = Data_Write, out, [0:31]
CSET signal D_AS = D_AS, out
CSET signal Read_Strobe = Read_Strobe, out
CSET signal Write_Strobe = Write_Strobe, out
CSET signal DReady = DReady, in
CSET signal Byte_Enable = Byte_Enable, out, [0:3]
```

MPD Peripheral Options

Peripherals defined in the MPD file can have the following options:

Table 2: **MPD Peripheral Options**

Option	Values	Default	Definition
STYLE	BLACKBOX MIX HDL	HDL	Design style
EDIF	TRUE FALSE	FALSE	Synthesize HDL to a hardware implementation netlist
INBYTE	TRUE FALSE	X	The functions inbyte() and *peripheral*_inbyte() are defined
OUTBYTE	TRUE FALSE	X	The functions outbyte() and *peripheral*_outbyte() are defined

STYLE Option

The STYLE option defines the design composition of the peripheral.

If you have only optimized hardware netlists, you must specify the BLACKBOX value within the MPD file. In this case, only the BBD file is read by the Platform Generator.

If you have a mix of optimized hardware netlists and HDL files, you must specify the MIX value within the MPD file. In this case, the PAO and BBD files are read by the Platform Generator.

If you have only HDL files, you must specify the HDL value within the MPD file. In this case, only the PAO file is read by the Platform Generator.

Format

```
SELECT IP peripheral_name, STYLE=value
```

Where *value* is BLACKBOX, MIX, or HDL. The default value is HDL.

EDIF Option

In hierarchal mode, this option directs the Platform Generator to write an EDIF file for the peripheral. In flatten mode, the EDIF option is ignored since the entire system is synthesized to an EDIF file.

Format

```
SELECT IP peripheral_name, EDIF=TRUE
```

INBYTE or OUTBYTE Option

The INBYTE and OUTBYTE options indicate that the peripheral can act as an input or output device, respectively. These options indicate to the Library Generator (libgen) that the peripheral has the peripheral_name_inbyte(), inbyte(), peripheral_name_outbyte(), and outbyte() functions defined.

The function outbyte() is used to write a single byte to the designated output device. The function inbyte() is used to read a singe byte from the designated input device.

Format

```
SELECT IP peripheral_name, INBYTE=boolean_value, OUTBYTE=boolean_value
```

Where *boolean_value* is either TRUE or FALSE.

Refer to the MicroBlaze Libraries documentation for more information.

MPD Signal Options

Signals defined in the MPD file can have the following options:

Table 3: **MPD Signal Options**

Option	Values	Default	Definition
BUS	*string*	X	Bus ownership of a signal
EDGE	RISING FALLING	X	Interrupt edge sensitivity
ENABLE	MULTI SINGLE	SINGLE	3-state enable control
ENDIAN	BIG LITTLE	BIG	Endianess
INITIALVAL	VCC GND	GND	Driver value on unconnected inputs
LEVEL	HIGH LOW	X	Interrupt level sensitivity
TYPE	INTERNAL EXTERNAL	EXTERNAL	Scope of signal

BUS Option

The bus ownership of a signal is specified by the BUS option.

Format

```
CSET signal IOPB_timeout = OPB_timeout, in,, BUS=bus_label
```

Where the *bus_label* is a string.

EDGE Option

The edge sensitivity of an interrupt signal is specified by the EDGE option.

Format

```
CSET signal Interrupt = “”, out,, EDGE=edge_value, TYPE=INTERNAL
```

Where *edge_value* is RISING or FALLING.

ENABLE Option

3-state signals can have multi-bit enable control, or a single bit enable control on the bus. This is specified with the ENABLE option.

Format

```
CSET signal mysignal = “”, inout, [0:31], ENABLE=enable_value
```

Where *enable_value* is SINGLE or MULTI. SINGLE is the default value.

Refer to the *HDL Design Considerations* section for more information on designing 3-state signals at the HDL level.

ENDIAN Option

The endianess of a signal is specified by the ENDIAN option.

Format

```
CSET signal mysignal = “”, out,[A:B], ENDIAN=endian_value
```

Where *endian_value* is BIG or LITTLE. BIG is the default value.

INITIALVAL Option

The driver val on unconnected input signals is specified by the INITIALVAL option.

Format

```
CSET signal mysignal = “”, in,, INITIALVAL=init_value
```

Where *init_value* is VCC or GND. GND is the default value.

LEVEL Option

The level sensitivity of an interrupt signal is specified by the LEVEL option.

Format

```
CSET signal Interrupt = “”, out,, LEVEL=level_value
```

Where *level_value* is HIGH or LOW.

TYPE Option

The scope of a signal is set with the TYPE option

Format

```
CSET signal mysignal = "", out,, TYPE=type_value
```

Where *type_value* is either EXTERNAL or INTERNAL. By default, only OPB and LMB signals are defined as INTERNAL. All other signals are defined as EXTERNAL.

Black-Box Description (BBD) File

The BBD (Black-Box Description) file is supplied by the IP provider and supplies input to the Platform Generator. The BBD file manages the file locations of optimized hardware netlists for the black-box sections of your peripheral design.

The value of the STYLE option in the MPD file determines whether or not you need a BBD file.

The black-box simulation netlists for HDL simulation must be moved to the simmodels directory, and the black-box hardware netlists for implementation must be moved to the netlist directory. The simmodels and netlist directories can have their own underlying directory structure, however, they must mirror each other.

Comments

You can insert comments in the BBD file without disrupting processing. Comments begin with a pound sign (#) and continue to the end of the line.

Format

The BBD format is a look-up table chart that lists netlist files. The first line is the header of the look-up table. There can be as many entries as necessary in the header to make a selection. Header entries are tailored by MPD options. The last column of the table must be the FILES column.

For implementation, the last column lists the relative path to the file from:
$XIL_MYPERIPHERALS/opb_peripherals/<*ip*>/netlist (UNIX)

%XIL_MYPERIPHERALS%\opb_peripherals\<*ip*>\netlist (PC)

For simulation, the last column lists the relative path to the file from:
$XIL_MYPERIPHERALS/opb_peripherals/<*ip*>/simmodels (UNIX)

%XIL_MYPERIPHERALS%\opb_peripherals\<*ip*>\simmodels (PC)

The netlist and simmodels directories can have their own underlying directory structure because the BBD file manages the relative file locations. However, the directories must mirror each other.

Each file is listed with the file extension of the hardware implementation netlist. Since implementation netlists have multiple file extensions (such as, .edn, .edf, .edo, .ngo), it is important to identify the format. For simulation, the Platform Generator uses the file extension .vhd for VHDL simulation and .v for Verilog.

BBD Example

The following is an example BBD file:

```
C_FAMILY CONFIGURATION FILES
virtex          1       virtex/microblaze_1.edf
virtex          2       virtex/microblaze_2.edf
virtex          3       virtex/microblaze_3.edf
virtex          4       virtex/microblaze_4.edf
virtex          5       virtex/microblaze_5.edf
virtex          6       virtex/microblaze_6.edf
spartan2        1       virtex/microblaze_1.edf
spartan2        2       virtex/microblaze_2.edf
spartan2        3       virtex/microblaze_3.edf
spartan2        4       virtex/microblaze_4.edf
spartan2        5       virtex/microblaze_5.edf
spartan2        6       virtex/microblaze_6.edf
virtexe         1       virtex/microblaze_1.edf
virtexe         2       virtex/microblaze_2.edf
virtexe         3       virtex/microblaze_3.edf
virtexe         4       virtex/microblaze_4.edf
virtexe         5       virtex/microblaze_5.edf
virtexe         6       virtex/microblaze_6.edf
virtex2         1       virtex2/microblaze_1.edf
virtex2         2       virtex2/microblaze_2.edf
virtex2         3       virtex2/microblaze_3.edf
virtex2         4       virtex2/microblaze_4.edf
virtex2         5       virtex2/microblaze_5.edf
virtex2         6       virtex2/microblaze_6.edf
```

Peripheral Analyze Order (PAO) File

A PAO (Peripheral Analyze Order) file is supplied by the IP provider and supplies information to the Platform Generator. This file contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation. The value of the STYLE option in the MPD file determines whether or not a PAO file is required. The HDL files used for synthesis must be moved to the hdl directory.

Comments

You can insert comments in the PAO file without disrupting processing. The following are guidelines for inserting comments:

- Precede comments with the pound sign (#)
- Comments can continue to the end of the line
- Comments can be anywhere on the line

Format

Use the following format:

```
lib library_name hdl_file_basename
```

Library_name specifies the unique library for the peripheral, and HDL file names are specified without a file extension. All names are in lower-case.

If your peripheral requires a certain version of a library, then the library name is given with the version appended. For example if you request version 1.00.a, then the library name is:

*library_name*_v1_00_a

PAO Example

The following is an example PAO file:

```
lib common_v1_00_a common_types_pkg
lib common_v1_00_a pselect
lib opb_gpio_v1_00_a gpio_core
lib opb_gpio_v1_00_a opb_gpio
```

HDL Design Considerations

This section includes HDL design considerations.

Scalable Data path

Using an MPD option declaration, you can automatically scale data path width. Bus expressions are evaluated as arithmetic equations.

Format

```
CSET signal name = default_connection, direction, [A:B]
```

Where A and B are an arithmetic expression.

MPD Example

The following is an example MPD file:

```
SELECT IP my_peripheral
# Generics for vhdl or parameters for verilog
CSET attribute C_BASEADDR = 0xB00000, std_logic_vector(0 to 31)
CSET attribute C_MY_PERIPH_AWIDTH = 17, integer
# Global ports
CSET signal OPB_Clk = “”, in
CSET signal OPB_Rst = “”, in
# My peripheral signals
CSET signal MY_ADDR = “”, out, [0:C_MY_PERIPH_AWIDTH-1]
# OPB signals
.
```

```
.
END
```

By default, if the vectors are larger than one bit, the Platform Generator determines the range specification on buses as either big-endian or little-endian. However, if the vector is one-bit width, then the range cannot be determined, and Platform Generator defaults to big-endian style notation.

To change this default behavior, use the ENDIAN option.

Format

```
CSET signal mysignal = “”, in, [0:0], ENDIAN=LITTLE
```

This builds the VHDL equivalent:

```
mysignal : in std_logic_vector(0 downto 0);
```

Internal Signals

Set internal signals with the TYPE=INTERNAL option.

Format

```
CSET signal mysignal = "", out,, TYPE=INTERNAL
```

By default, only OPB and LMB signals are defined as INTERNAL. All other signals are defined as EXTERNAL. External signals are available through the port-declaration in the top-level module.

Interrupt Signals

Interrupt signals are identified by the EDGE or LEVEL option.

3-state (InOut) Signals

At the MHS/MPD level, there is a listing for an inout port in the MPD file that allows you to map to it in the MHS file. In the MPD file, a 3-state signal is identified by the inout direction mode, and the port name must be ioname.

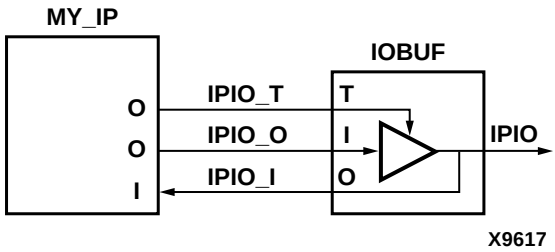


Figure 2: **IOBUF Implementation**

The Platform Generator expands the inout port in the MPD file to three ports in the port declaration section of the HDL file, and writes out the RTL code to infer the IOBUF. This port expansion occurs because if the top-level is synthesized without IO insertion, the 3-states on the inout ports are inferred as BUFTs at the CLB level. However, they should be inferred as IOBUFs at the IOB level. Platform Generator infers the 3-states at the top-level to ensure that the inout ports are always associated to the IOBUF.

Inout ports are currently defined at the top-level since the only internal signals are those defined as an input or an output. There are no inout signals defined internally that need a BUFT.

It is important to note that the 3-state enables are all active-low to allow a direct connection to the OBUFT of the IOBUF.

On-Chip Peripheral Bus (OPB) Arbiter Design Specification

March 2002

Summary

This document provides the design specification for the On-Chip Peripheral Bus (OPB) Arbiter. This document applies to the following peripherals:

opb_arbiter	v1.02c

Introduction

The OPB Arbiter design described in this document incorporates the features contained in the IBM On-chip Peripheral Bus Arbiter Core manual (version 1.5) for 32-bit implementation. This manual is referenced throughout this document and is considered the authoritative specification. Any differences between the IBM OPB Arbiter implementation and the Xilinx OPB Arbiter implementation are explained in the **Specification Exceptions** section.

The Xilinx OPB Arbiter design allows you to tailor the OPB Arbiter to suit your application by setting certain parameters to enable/disable features. In some cases, setting these parameters may cause the Xilinx OPB Arbiter design to deviate slightly from the IBM OPB Arbiter specification. These parameters are described in the **OPB Arbiter Design Parameters** section.

OPB Arbiter Overview

Features

The OPB Arbiter is a soft IP core designed for Xilinx FPGAs and contains the following features:

- Optional OPB slave interface (included in design via a design parameter)
- OPB Arbitration
  - arbitrates between 1 - 16 OPB Masters (the number of masters is parameterizable)
  - arbitration priorities among masters programmable via register write
  - priority arbitration mode configurable via a design parameter
    - Fixed priority arbitration with processor access to read/write Priority Registers
    - Dynamic priority arbitration implementing a true least recent used (LRU) algorithm
- Two bus parking modes selectable via Control Register write:
  - park on selected OPB master (specified in Control Register)
  - park on last OPB master which was granted OPB access
- Watchdog timer which asserts the OPB time-out signal if a slave response is not detected within 16 clock cycles
- Registered or combinational Grant outputs configurable via a design parameter

OPB Arbitration Protocol

OPB Bus arbitration uses the following protocol:

1. An OPB master asserts its bus request signal.
2. The OPB Arbiter receives the request and outputs an individual grant signal to each master according to its priority and the state of the other requests.
3. An OPB master samples its grant signal at the rising edge of the OPB clock. In the following cycle, the OPB master initiates a data transfer between the master and a slave by asserting its select signal.

The OPB Arbiter only issues a bus grant signal during valid arbitration cycles which are defined as either:

- Idle

  The OPB_select and OPB_busLock are deasserted, indicating that no data transfer is in progress.

- Overlapped arbitration cycle

  The OPB_xferAck is asserted, indicating the final cycle in a data transfer, and OPB_busLock is not asserted. Arbitration in this cycle allows another master to begin a transfer in the following cycle, avoiding the need for a *dead* cycle on the bus.

You can configure the Xilinx OPB Arbiter to have either registered or combinational grant outputs. Registered grant outputs are asserted one clock after each arbitration cycle resulting in one *dead* cycle on the bus. However, registered grant outputs allow the OPB bus to run at higher clock rates.

Figure 1 shows the fixed OPB arbitration protocol with combinational grant outputs, Figure 2 shows the fixed OPB arbitration protocol when the OPB Arbiter has been parameterized to have registered grant outputs.

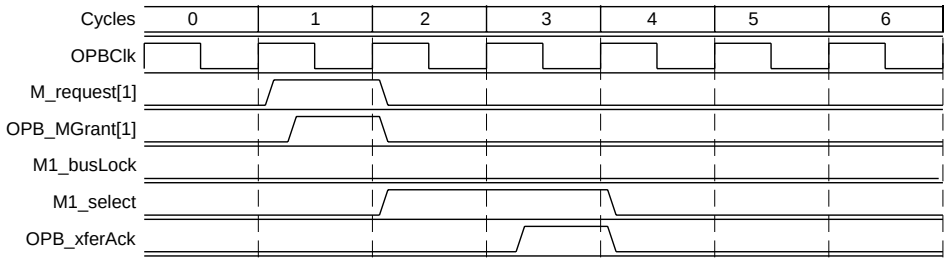


Figure 1: **OPB Fixed Bus Arbitration - Combinational Grant Outputs**

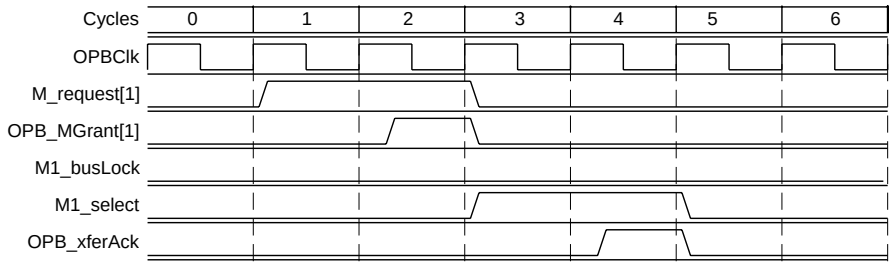


Figure 2: **OPB Fixed Bus Arbitration - Registered Grant Outputs**

An OPB master device need not deassert its request upon receipt of a bus grant signal if it has multiple bus transfer cycles to perform. Figure 3 shows an OPB arbitration cycle in which an OPB master asserts bus request continuously for four data transfer cycles. The OPB Arbiter has been parameterized for fixed priority arbitration and combinational grant outputs, therefore bus grant is asserted combinationally during valid arbitration cycles.

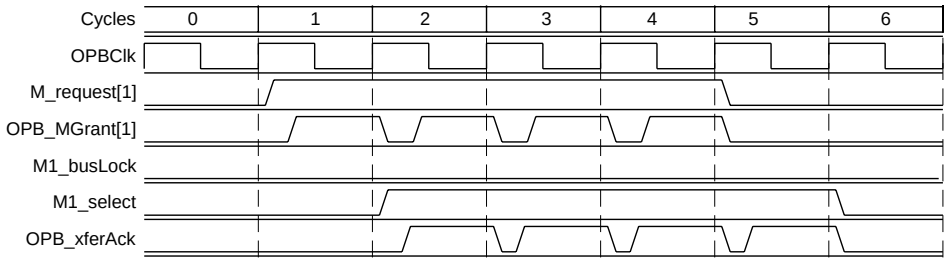


Figure 3: **Continuous Master Bus Request - Fixed priority, Combinational Grant Outputs**

When the OPB Arbiter has been parameterized for registered grant outputs and fixed priority, the bus grants are registered as shown in the following figure.

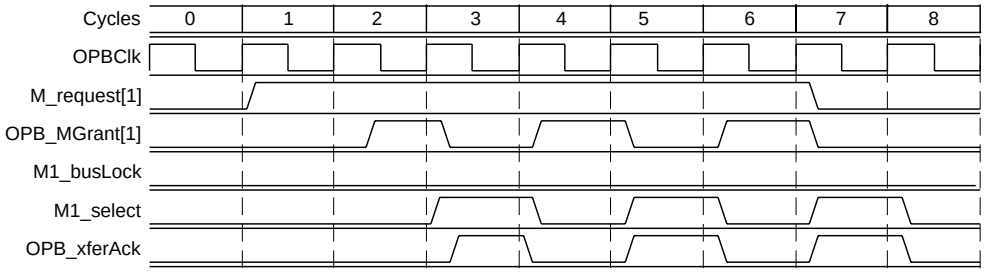


Figure 4: **Continuous Master Bus Request - Fixed priority, Registered Grant Outputs**

Even if an OPB master asserts request continuously, it will not necessarily receive a valid grant signal. Other OPB masters with higher bus priority may request the OPB and will be granted the bus according to OPB arbiter priority. If an OPB master device needs a non-interruptible sequence of bus cycles, it can use the bus lock signal for this purpose. Bus locking is described later in this document.

The OPB Arbiter supports both dynamic priority arbitration, implementing a Least Recently Used (LRU) algorithm, and fixed priority arbitration, both are described in more detail later in this document. Figure 5 shows multiple bus request or overlapped bus arbitration when the OPB arbiter is using fixed priority arbitration and combinational grant outputs. Both OPB Master 1 and OPB Master 2 simultaneously request the bus. Master 1 has a higher priority and is granted the bus. During cycle 2, Master 1 completes its first transaction and Master 2 is granted the bus for cycle 3. Thus, during cycle 2, the arbitration for the bus is overlapped with a data transfer. This overlapped bus arbitration improves the bandwidth of the bus.

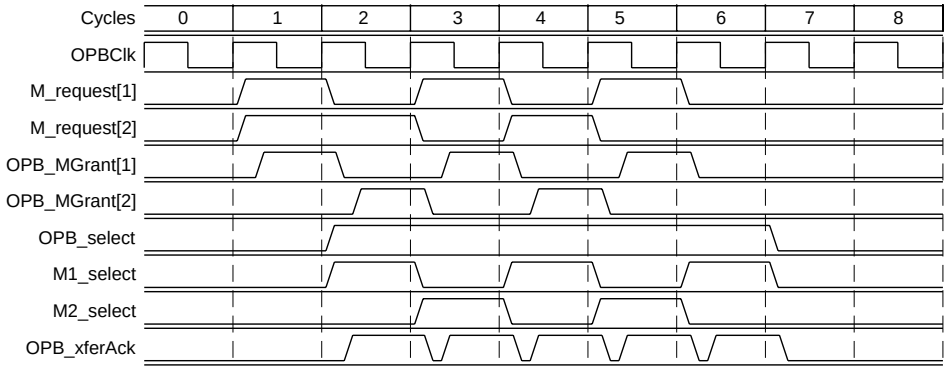


Figure 5: **Multiple Bus Requests - Fixed Priority Arbitration, Combinational Grant Outputs**

With registered grant outputs, there is a cycle between bus grant signals as shown in Figure 6. Using registered grant outputs from the OPB arbiter reduces the number of logic levels between registers and allows the OPB bus to run at a higher clock rate.

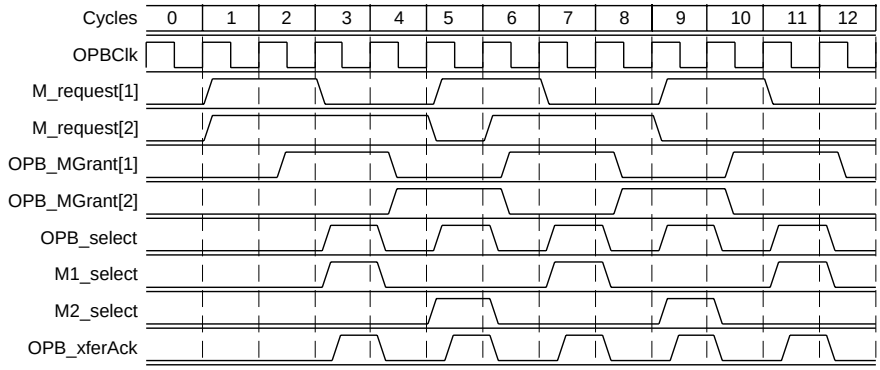


Figure 6: **Multiple Bus Requests - Fixed Priority Arbitration, Registered Grant Outputs**

OPB Arbiter Design Parameters

To obtain an OPB Arbiter that is uniquely tailored to your system, you can parameterize certain features in the OPB Arbiter design. This allows you to have a high performance design that only utilizes the resources required by your system. The features that you can parameterize in the Xilinx OPB Arbiter design are shown in the following table.

Table 1: **OPB Arbiter Design Parameters**

Grouping/Number		Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
Arbiter Features	G1	Number of OPB Masters	C_NUM_MASTERS	1(1) - 16	4	integer
	G2	Priority Mode	C_DYNAM_PRIORITY	1 = Dynamic 0 = Fixed	0 = Fixed	integer
	G3	Registered Grant Outputs	C_REG_GRANTS	1 = Registered Grant Outputs 0 = Combinational Grant Outputs(2)	1 = Registered Grant Outputs	integer
	G4	Bus Parking	C_PARK	1 = Bus parking supported(3) 0 = Bus parking not supported	0 = Bus parking not supported	integer
OPB Interface	G5	OPB Slave Interface	C_PROC_INTRFCE	1 = OPB slave interface supported 0 = OPB slave interface not supported(4)	0 = OPB slave interface not supported	integer
	G6	OPB Arbiter Base Address	C_BASEADDR	Valid Address Range(7)	None(5)	std_logic_vector
	G7	OPB Data Bus Width	C_OPB_DWIDTH	8,16,32,64,128	32	integer
	G8	OPB Address Bus Width	C_OPB_AWIDTH	16 - 32	32	integer
	G9	Device Block ID(6)	C_DEV_BLK_ID	See note 6.	0	integer
	G10	Module Identification Register Enable(6)	C_DEV_MIR_ENABLE	See note 6.	0	integer
	G11	OPB High Address	C_HIGHADDR	Address range must be a power of 2 and >= 0x1FF(7)	None(5)	integer

Notes:

1. When C_NUM_MASTERS = 1, no arbitration is necessary, however, the watchdog timer is included in the arbiter and is needed for the OPB. In this case, all other parameters are meaningless.
2. C_REG_GRANTS should only be set to 0 (indicating that the Grant outputs are combinational) if the desired OPB frequency is less than the f_{MAX} specified for this parameter.
3. When bus parking is supported, the parking mode (park on last master or park on master id) is set in the OPB Arbiter Control Register. If C_PROC_INTRFCE is 0, the parking mode is park on last master.
4. When C_PROC_INTRFCE is 0, none of the OPB Arbiter registers are accessible.
5. No default value will be specified for C_BASEADDR to insure that the actual value is set, i.e. if the value is not set, a compiler error will be generated.
6. Address range specified by C_BASEADDR and C_HIGHADDR must be at least 0x1FF and must be a power of 2.

Allowable Parameter Combinations

The only restriction on parameter combinations in the Xilinx OPB Arbiter design is that the address range specified by C_BASEADDR and C_HIGHADDR is a power of 2. To allow for the register offset within the OPB Arbiter design, the range specified by C_BASEADDR and C_HIGHADDR must be at least 0x1FF.

The clock frequency of the OPB specified in the Platform Builder tool must be low enough to allow for combinational Grant outputs. Therefore, the parameter C_REG_GRANTS can only be set to 0 if the value of the OPB frequency is less than the f_{MAX} specified for this parameter.

The number of OPB masters can be parameterized to 1 master. Though no arbitration is necessary when there is only one OPB master, the OPB Arbiter contains the watchdog timer for the OPB and is therefore needed in the system. When there is only 1 OPB master, there will be no Control Register or Priority Registers, therefore, there will be no OPB slave interface on the OPB Arbiter. Since there will not be an OPB slave interface, the OPB Arbiter, when parameterized for 1 OPB master, will not have a Configuration ROM (CROM) entry. When C_NUM_MASTERS is set to 1, all other parameters are meaningless. Also, when C_PROC_INTRFCE is set to 0, the OPB Arbiter registers are not accessible and there is no CROM entry for the OPB Arbiter.

OPB Arbiter I/O Signals

The I/O signals for the OPB Arbiter are listed in Table 2. The interfaces referenced in this table are shown in Figure 7 in the OPB Arbiter block diagram.

Table 2: **OPB Arbiter I/O Signals**

Grouping		Signal Name	Interface	I/O	Initial State	Description	Page
OPB Slave Signals	P1	ARB_DBus(0:C_OPB_DWIDTH-1)	IPIF	O	0	Arbiter output data bus	76
	P2	ARB_xferAck	IPIF	O	0	Arbiter transfer acknowledge	76
	P3	ARB_Retry	IPIF	O	0	Arbiter retry	76
	P4	ARB_ToutSup	IPIF	O	0	Arbiter timeout suppress	76
	P5	ARB_ErrAck	IPIF	O	0	Arbiter error acknowledge	76
	P6	OPB_ABus(0:C_OPB_AWIDTH-1)	IPIF	I		OPB address bus	76
	P7	OPB_BE(0:C_OPB_DWIDTH/8-1)	IPIF	I		OPB byte enables	76
	P8	OPB_DBus(0:C_OPB_DWIDTH-1)	IPIF	I		OPB data bus	76
	P9	OPB_RNW	IPIF	I		Read not Write (OR of all master RNW signals)	76
	P10	OPB_seqAddr	IPIF	I		OPB sequential address	76
Arbitration Signals	P11	M_request[0:C_NUM_MASTERS-1](1)	Arbitration Logic	I		Request from OPB Masters	80
	P12	OPB_xferAck	Arbitration Logic	I		Transfer Acknowledge indicating end of data transfer cycle (OR of all slave xferAcks)	80
	P13	OPB_select	Arbitration Logic Watchdog Timer	I		Master has taken control of the bus (OR of all master selects)	80,86
	P14	OPB_retry	Watchdog Timer	I		Bus cycle retry (OR of all slave retrys)	86
	P15	OPB_toutSup	Watchdog Timer	I		Suppress timeout (OR of all slave toutSups)	86
	P16	OPB_timeout	Watchdog Timer	O	0	Timeout signal for OPB	86
	P17	OPB_busLock	Park/Lock Logic	I		Bus lock (OR of all master buslocks)	82
	P18	OPB_MGrant[0:C_NUM_MASTERS-1](1)	Park/Lock Logic	O	0	Grant to OPB Masters	82
System	P19	OPB_Clk	System	I		System clock	
	P20	OPB_Rst	System	I		System Reset (active high)	

Notes:
1. Name has been modified slightly from that in the IBM OPB Arbiter specification to support parameterization of the number of masters

The signal, ARB_DBusEn, is not an output of the Xilinx OPB Arbiter as the IPIF module internally gates the OPB Arbiter data bus with the enable signal.

The following signals listed in the IBM OPB Arbiter core are not supported:

- ARB_sleepReq -the FPGA implementation of the OPB bus will not support sleep modes
- LSSD_AClk - FPGA implementation does not support scan
- LSSD_BClk - FPGA implementation does not support scan
- LSSD_CClk - FPGA implementation does not support scan
- LSSD_scanGate - FPGA implementation does not support scan
- LSSD_scanIn - FPGA implementation does not support scan
- LSSD_scanOut - FPGA implementation does not support scan

Parameter - Port Dependencies

The width of many of the OPB Arbiter signals depends on the number of OPB masters in the design. In addition, when certain features are parameterized away, the related input signals are unconnected and the related output signals are set to a constant values. The dependencies between the OPB Arbiter design parameters and I/O signals are shown in Table 3.

Table 3: **Parameter-Port Dependencies**

		Name	Affects	Depends	Relationship Description
Design Parameters	G1	C_NUM_MASTERS	G2-G8 P1,P2 P6-P11 P17,P18	G4, G5, G7	The width of the request and grant buses are set by the number of masters in the design. The only logic present in the OPB Arbiter design when C_NUM_MASTERS=1 is the watchdog timer, therefore, many parameters and I/O signals are unconnected in this case. If C_OPB_DWIDTH=8 and C_PARK=1 and C_PROC_INTRFCE=1 then C_NUM_MASTERS must be <=4 because there is only 2 bits for the park master id in the Control Register.
	G2	C_DYNAM_PRIORITY		G1	Unconnected if C_NUM_MASTERS=1.
	G3	C_REG_GRANTS		G1	Unconnected if C_NUM_MASTERS=1.
	G4	C_PARK	G1	G1	Unconnected if C_NUM_MASTERS=1.
	G5	C_PROC_INTRFCE	G1,G6-G8, G11 P1, P2 P6-P10	G1	Unconnected if C_NUM_MASTERS=1.
	G6	C_BASEADDR		G1,G5, G11	Unconnected if C_PROC_INTRFCE=0 or C_NUM_MASTERS=1. Range specified by C_BASEADDR and C_HIGHADDR must be at least 0x1FF.
	G7	C_OPB_DWIDTH	G1, P1, P8	G1,G5	Unconnected if C_PROC_INTRFCE=0 or C_NUM_MASTERS=1.
	G8	C_OPB_AWIDTH	P6,P7	G1,G5	Unconnected if C_PROC_INTRFCE=0 or C_NUM_MASTERS=1.
	G9	C_DEV_BLK_ID		G1,G5	Unconnected if C_PROC_INTRFCE=0 or C_NUM_MASTERS=1.
	G10	C_DEV_MIR_ENABLE		G1,G5	Unconnected if C_PROC_INTRFCE=0 or C_NUM_MASTERS=1.
	G11	C_HIGHADDR		G1,G5, G6	Unconnected if C_PROC_INTRFCE=0 or C_NUM_MASTERS=1. Range specified by C_BASEADDR and C_HIGHADDR must be at least 0x1FF.

Table 3: **Parameter-Port Dependencies *(Continued)***

		Name	Affects	Depends	Relationship Description
I/O Signals	P1	ARB_DBus(0:C_OPB_DWIDTH-1)		G1,G5, G7	Width varies with the size of the OPB Data bus. This output is grounded if C_PROC_INTRFCE = 0 or C_NUM_MASTERS=1.
	P2	ARB_xferAck		G1,G5	This output is grounded if C_PROC_INTRFCE = 0 or C_NUM_MASTERS=1.
	P3	ARB_Retry			This output is always grounded.
	P4	ARB_ToutSup			This output is always grounded.
	P5	ARB_ErrAck			This output is always grounded.
	P6	OPB_ABus(0:C_OPB_AWIDTH-1)		G1,G5, G8	Width varies with the size of the OPB Address bus. This input is unconnected if C_PROC_INTRFCE = 0 or C_NUM_MASTERS = 1.
	P7	OPB_BE(0:C_OPB_DWIDTH/8-1)		G1,G5, G8	Width varies with the size of the OPB Address bus. This input is unconnected if C_PROC_INTRFCE = 0 or C_NUM_MASTERS = 1.
	P8	OPB_DBUS(0:C_OPB_DWIDTH-1)		G1,G5, G7	Width varies with the size of the OPB Data bus. This input is unconnected if C_PROC_INTRFCE = 0 or C_NUM_MASTERS = 1.
	P9	OPB_RNW		G1,G5	This input is unconnected if C_PROC_INTRFCE = 0 or C_NUM_MASTERS = 1.
	P10	OPB_seqAddr		G1,G5	This input is unconnected if C_PROC_INTRFCE = 0 or C_NUM_MASTERS = 1.
	P11	M_request(0:C_NUM_MASTERS-1)		G1	Width varies with the number of OPB masters. This input is unconnected if C_NUM_MASTERS=1.
	P12	OPB_xferAck			
	P13	OPB_select			
	P14	OPB_retry			
	P15	OPB_toutSup			
	P16	OPB_timeout			
	P17	OPB_busLock		G1	This input is unconnected if C_NUM_MASTERS=1.
	P18	OPB_MGrant(0:C_NUM_MASTERS-1)		G1	Width varies with the number of OPB masters. This output is set to 1 if C_NUM_MASTERS = 1.
	P19	OPB_Clk			
	P20	OPB_Rst			

OPB Arbiter Register Descriptions

The OPB Arbiter contains addressable registers for read/write operations as shown in Table 4 if the design has been parameterized to support a processor interface. The base address for these registers is set in the parameter C_BASEADDR. The registers are located at an offset of 0x00000100 from C_BASEADDR. Each register is addressable on a 32-bit boundary.

Each priority level has a unique Priority Register which contains the master id for the master at that priority level. The Priority Registers are readable and writable by the processor. The number of priority levels and hence the number of Priority Registers will vary with the parameter C_NUM_MASTERS.

Table 4 shows all of the OPB Arbiter registers and their addresses when the maximum number of masters has been selected. In this case, 17 registers are required.

Table 4: **OPB Arbiter Registers**

Register Name	OPB Address	Access
Control Register	C_BASEADDR + 0x100	Read/Write
LVL0 Priority Register	C_BASEADDR + 0x104	Read/Write
LVL1 Priority Register	C_BASEADDR + 0x108	Read/Write
LVL2 Priority Register	C_BASEADDR + 0x10C	Read/Write
LVL3 Priority Register	C_BASEADDR + 0x110	Read/Write
LVL4 Priority Register	C_BASEADDR + 0x114	Read/Write
LVL5 Priority Register	C_BASEADDR + 0x118	Read/Write
LVL6 Priority Register	C_BASEADDR + 0x11C	Read/Write
LVL7 Priority Register	C_BASEADDR + 0x120	Read/Write
LVL8 Priority Register	C_BASEADDR + 0x124	Read/Write
LVL9 Priority Register	C_BASEADDR + 0x128	Read/Write
LVl10 Priority Register	C_BASEADDR + 0x12C	Read/Write
LVL11 Priority Register	C_BASEADDR + 0x130	Read/Write
LVL12 Priority Register	C_BASEADDR + 0x134	Read/Write
LVL13 Priority Register	C_BASEADDR + 0x138	Read/Write
LVL14 Priority Register	C_BASEADDR + 0x13C	Read/Write
LVL15 Priority Register	C_BASEADDR + 0x140	Read/Write

The register definitions and address locations of the Xilinx OPB Arbiter deviate from the IBM OPB Arbiter specification. This deviation is necessary to support parameterization of the number of OPB Masters. See section **Specification Exceptions** for more information.

OPB Arbiter Control Register

The OPB Arbiter Control Register is shown in Table 5 for the case where the OPB Arbiter has been parameterized for fixed priority arbitration without bus parking with a data width of 32 bits. The reset values for the DPE, DPERW, PEN, and PENRW bits vary based on the parameterization of the arbitration and bus parking scheme.

Table 5: **OPB Arbiter Control Register**

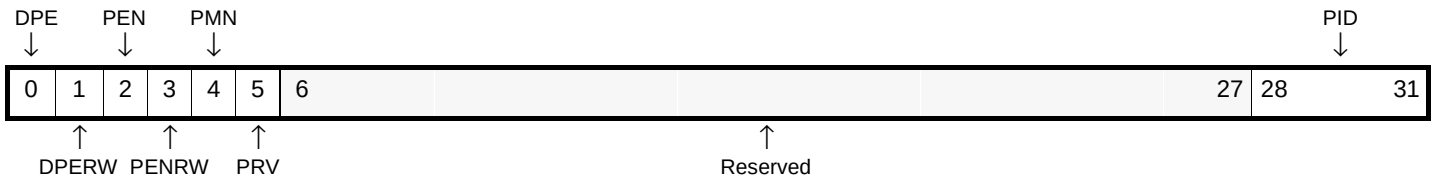


Table 6 shows the Control Register bit definitions. The Control Register definition deviates from the IBM OPB Arbiter specification. This deviation is necessary due to the fact that the Park Master ID (PID) field will vary in width based on the number of masters supported in the design. This field has been shifted so that it is LSB aligned and can be read easily by the software as an integer value. Also, the DPERW and PENRW bits have been added to provide information to the processor about the parameters chosen for the design and the accessibility of the DPE and PEN bits. Since the processor requires more than one OPB bus cycle to update the

master's priority levels, the bit Priority Registers Valid (PRV), has been added to indicate that the Priority Registers are being modified and are not valid.

Table 6: **OPB Arbiter Control Register Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0	DPE	Read(1) Read/Write(2)	'0'(1) '1'(2)	**Dynamic Priority Enable.** Enables dynamic priority arbitration algorithm and update of the Priority Register. • '0' - dynamic priority arbitration disabled • '1' - dynamic priority arbitration enabled
1	DPERW	Read	'0'(1) '1'(2)	**Dynamic Priority Enable Bit Read/Write.** This bit informs the software as to the access of the DPE bit. If the OPB Arbiter is parameterized to only support fixed priority arbitration, the DPE bit is always set to'0' to reflect that dynamic priority arbitration is not available. • '0' - DPE bit is read only • '1' - DPE bit is read/write
2	PEN	Read/Write	'0'(3) '1'(4)	**Park Enable.** Enables parking on a master when no other masters have requests asserted. • '0' - parking disabled • '1' - parking enabled
3	PENRW	Read	'0'(3) '1'(4)	**Park Enable Bit Read/Write.** This bit informs the software as to the access of the PEN bit. If the OPB Arbiter is parameterized to not support bus parking, the PEN bit is always set to'0' to reflect that bus parking is not available. • '0' - PEN bit is read only • '1' - PEN bit is read/write
4	PMN	Read/Write	'0'	**Park On Master Not Last.** When parking is enabled, this bit determines if the arbiter parks on the master who was last granted the bus or on the master specified by the Parked Master ID bits. • '0' - park on the master who had just been granted the bus • '1' - park on the master specified by the Parked Master ID bits

Table 6: **OPB Arbiter Control Register Bit Definitions *(Continued)***

Bit(s)	Name	Core Access	Reset Value	Description
5	PRV	Read/Write	'1'	**Priority Registers Valid.** This bit indicates that the Priority Registers all contain unique master IDs. This bit is negated by the processor before the processor modifies the Priority Registers and is asserted by the processor when the modifications are complete. Whenever this bit is negated, the OPB Arbiter uses the masters' IDs as their priority levels to perform bus arbitration.
6:C_OPB_DWIDTH - $\log_2$(C_NUM_MASTERS) - 1	Reserved			
C_OPB_DWIDTH - $\log_2$(C_NUM_MASTERS): C_OPB_DWIDTH - 1	PID	Read/Write	"0000"(5)	**Parked Master ID.** These bits contain the ID of the master to park on if parking is enabled and the Park On Master Not Last bit is set.

Notes:
1. OPB Arbiter parameterized to support fixed priority arbitration
2. OPB Arbiter parameterized to support dynamic priority arbitration
3. OPB Arbiter parameterized to not support bus parking
4. OPB Arbiter parameterized to support bus parking
5. The number of bits required by the PID field will vary with the number of masters supported by the OPB Arbiter. A field width of 4 is shown here for the default value.

If the OPB Arbiter has been parameterized to only support fixed priority arbitration, the DPE bit is set to '0' and is read-only by the processor core. The DPERW bit is then set to '0' and reflects the fact that the OPB Arbiter only supports fixed priority arbitration. If the OPB Arbiter has been parameterized to support dynamic priority arbitration, the DPE bit can be read from and written to by the processor. The DPERW bit is set to '1' and reflects the fact that the priority mode of the arbiter can be controlled by the DPE bit. Although the OPB Arbiter has been parameterized to support dynamic priority arbitration, the OPB Arbiter can be put in a fixed priority arbitration mode by negating the DPE bit. As a result, both priority modes are available and supported.

If the OPB Arbiter has been parameterized to not support bus parking, the PEN bit is set to '0'. The PENRW bit is then set to '0' and reflects the fact that the PEN bit is read-only by the processor core. If the OPB Arbiter has been parameterized to support bus parking, the PEN bit can be read from and written to by the processor. The PENRW bit is set to '1' and reflects the fact that the bus parking mode of the arbiter can be controlled by the PEN, PMN, and PID bits of the Control Register. Although the OPB Arbiter has been parameterized to support bus parking, bus parking can be disabled by negating the PEN bit. Also, if the OPB Arbiter has been parameterized to not include a processor or OPB slave interface, the Control Register is not accessible. As a result, the parking mode will be set to park on the last master which is the default value in the Control Register of the PMN bit. In order to park on the master whose ID is contained in the Control Register, the OPB Arbiter must be parameterized to support a processor interface so that the Control Register can be set up appropriately.

Since the processor will require multiple bus cycles to update the masters' priority levels in the Priority Registers, there will exist some period of time in which the Priority Registers will not contain unique master IDs. Though the arbiter will still function properly in this circumstance, a particular master will not have a priority level associated with it and therefore will never receive a grant if it issues a request. This could cause the OPB to "hang". Therefore, whenever the processor begins to modify the priority levels of the masters, it first negates the PRV bit in the Control Register, indicating to the arbitration logic that the Priority Register values should not

be used in bus arbitration. In this case, the arbitration logic will use the masters' IDs as their priority level until the PRV bit has been asserted.

The OPB Arbiter Control Register can be accessed from the OPB bus only if the OPB Arbiter is parameterized to support a processor interface (C_PROC_INTRFCE=1).

OPB Arbiter Priority Registers

Each Priority Register holds the master ID of the OPB master at that priority level as shown in Table 8. Each master's relative priority is determined by its ID's location within these registers. The LVL0 Priority Register holds the ID of the master with highest priority (level 0), the LVL1 Priority Register holds the ID of the master with the next highest priority (level 1), and so on. The LVL[C_NUM_MASTERS-1] Priority Register holds the ID of the master with lowest priority. These registers are reset to values which assign level 0 (highest) priority to Master 0, level 1 (next highest) priority to Master 1, level 2 priority to Master 2, etc.

The number of bits required by the master ID within each Priority Register will vary based on the number of masters support by the OPB Arbiter. The master ID will always be aligned to the LSB position within the register so that the master ID will appear as an integer when accessed by software.

Notes:

1. The IBM OPB Arbiter refers to the priority levels as High, Medium High, Medium Low, and Low since it only implemented arbitration among 4 masters. Since the Xilinx implementation of the OPB arbiter supports parameterization of the number of OPB masters in the system, numbers are used to represent priority levels instead of text descriptors. Level 0 will always remain the highest priority level regardless of the number of masters implemented. The higher the level number, the lower the priority.

Table 7: **OPB Arbiter OPB Arbiter LVLn Priority Register**

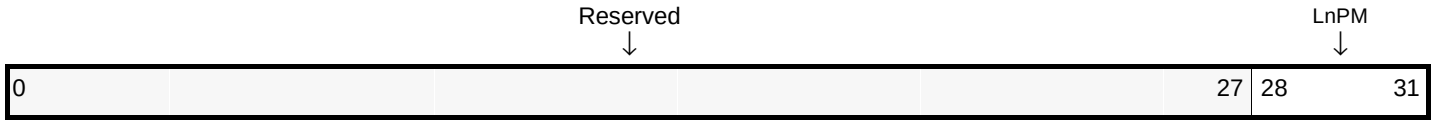


Table 8: **OPB Arbiter LVLn Priority Register Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0 : C_OPB_DWIDTH - log2(C_NUM_MASTERS) -1	Reserved			
C_OPB_DWIDTH - log2(C_NUM_MASTERS) : C_OPB_DWIDTH-1	LnPM	Read/Write	mmmm(1)	**Level n Priority Master ID.** This field contains the ID of the master at level n priority. (2)

Notes:

1. "mmmm" represents the bit encoding of the Master ID at this priority level
1. n = 0 - C_NUM_MASTERS - 1

Each master ID must be contained in one of the Priority Registers, otherwise that master's request will be ignored by the arbiter (since it has no priority value) and a grant to that master will never be asserted. This could cause the bus to stall since there is no mechanism in the OPB specification for a master to timeout while waiting for a grant. Therefore, each Priority Register must contain a unique master ID and all master IDs must be contained in one of the Priority Registers. Since the processor can not update the Priority Registers in a single OPB transaction, there may be several clock cycles in which a particular master ID is not contained within a Priority Register. Therefore, the Priority Registers Valid (PRV) bit in the Control Register is used to indicate that the values of the Priority Registers are being modified and should not be used in OPB arbitration. The processor negates this bit before modifying the

Priority Registers and then asserts this bit when the modification is complete. The processor will insure that whenever the PRV bit is asserted, all master IDs have been assigned a priority. When the PRV bit is negated, the OPB arbiter will assign priority based on the master ID, i.e, Master 0 will have level 0 priority (highest), Master 1 will have level 1 priority, and Master n will have level n priority. Once the PRV bit has been asserted, the values in the Priority Registers will again be used to determine OPB ownership.

The Priority Register can be accessed from the OPB for read and write operations. Note that regardless of the priority mode selected for the OPB Arbiter (even if the OPB Arbiter has been parameterized to fixed priority arbitration), the processor core can set the desired priority levels of the OPB masters by writing to these registers.

OPB Arbiter Block Diagram

The top-level block diagram for the OPB Arbiter is shown in Figure 7. The IPIF block is the OPB bus interface block that handles the OPB bus protocol for reading and writing the Priority Registers and the Control Register within the OPB Arbiter. The ARB2BUS data mux contains the data multiplexor required to output data to the OPB bus during a read cycle. The Arbitration Logic block determines which incoming request has the highest priority and the Park /Lock Logic block determines which master should be granted the bus based on this priority as well as whether the bus is locked or if bus parking is enabled. The Watchdog Timer asserts the OPB timeout signal if a slave response (OPB_xferAck, OPB_retry, or OPB_toutSup) has not been received within 16 clock cycles of the master taking control of the bus.

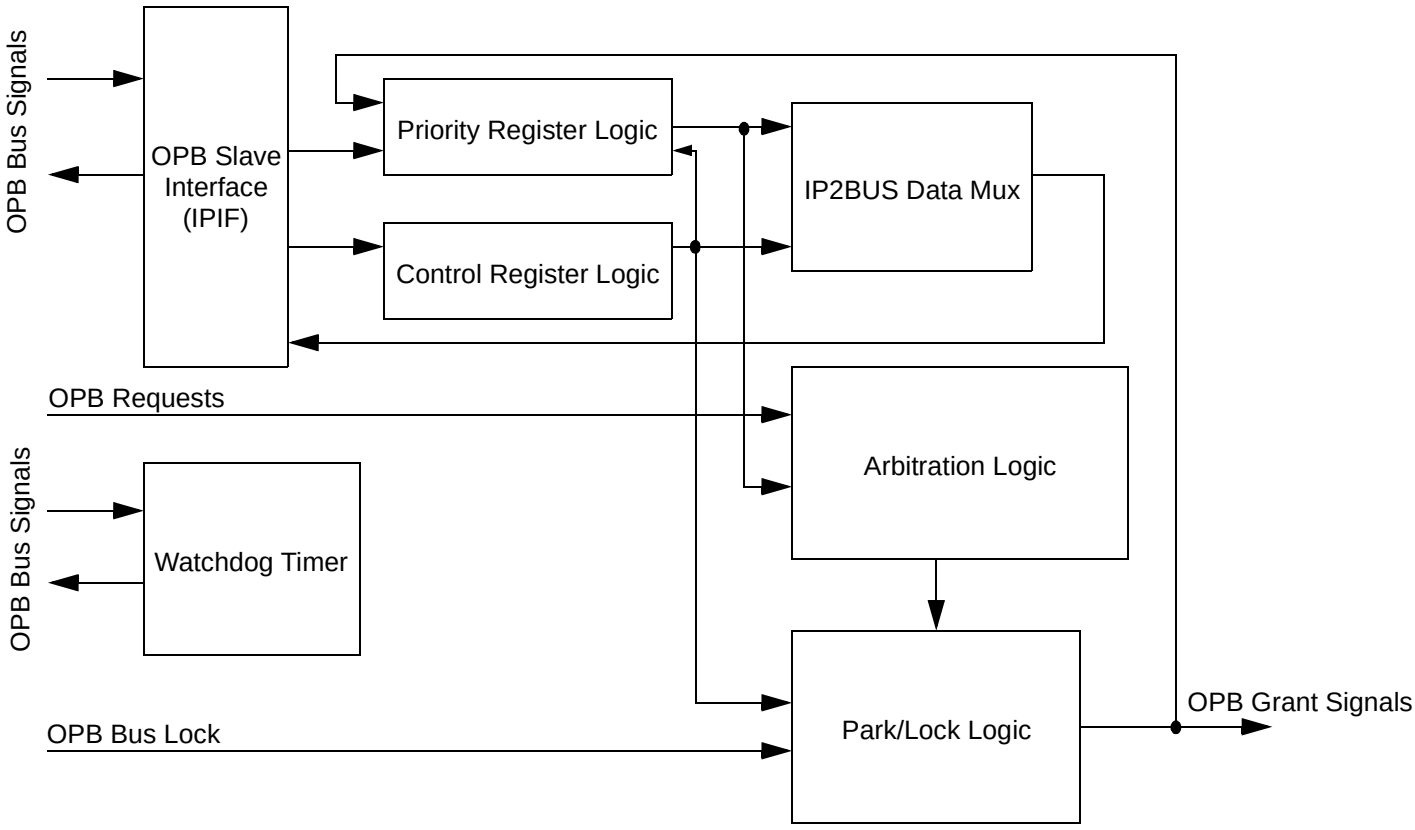


Figure 7: **OPB Arbiter Top-level Block Diagram**

When C_NUM_MASTERS=1, the only logic in the OPB Arbiter is the Watchdog Timer. The OPB Grant signal is set to VCC and all OPB Bus output signals are set to GND.

The following sections describe each module in the block diagram.

OPB Slave Interface (IPIF)

The IPIF block implements a slave interface to the OPB and is only present in the design if C_PROC_INTRFCE=1 and C_NUM_MASTERS>1. Its address in the OPB memory map is determined by setting the parameter C_BASEADDR. All registers are addressed by an offset to C_BASEADDR as shown in Table 4.

The IPIF block outputs a register write clock enable and a register read clock enable for the register which was addressed depending on the type of data transfer specified by the master. When the data transfer is complete, the IPIF block generates the transfer acknowledge.

Control Register Logic

The Control Register Logic block simply contains the OPB Arbiter Control Register described in section **OPB Arbiter Control Register** and is only present in the design if C_NUM_MASTERS > 1.

Priority Register Logic

The Priority Register logic contains the Priority Registers and the logic to update the priority of the OPB masters. Descriptions of the OPB Arbiter Priority Registers are found in section **OPB Arbiter Register Descriptions**. The Priority Registers are only present in the design if C_NUM_MASTERS>1.

Priority Register Update Logic

Fixed Priority Parameterization (C_DYNAM_PRIORITY=0)

When the OPB Arbiter is parameterized to support only fixed arbitration, the dynamic priority enable bit in the Control Register is permanently disabled. The Priority Registers are loaded at reset with the value of the Master ID which matches the priority level of the register. For example, the LVL0 Priority Register is loaded with '0' to represent the ID of Master 0. Likewise, the LVLn Priority Register is loaded with the bit encoding of n to represent the ID of Master n as shown in Table 8.

The Priority Registers can be loaded with different Master IDs by writing to the Priority Registers (if C_PROC_INTRFCE=1), therefore, the priorities of the OPB Masters can be changed as desired by software. The Priority Registers Valid (PRV) bit in the Control Register is negated whenever the processor modifies the Priority Registers and is asserted whenever the modification is complete. The ID of the masters are used to determine OPB ownership whenever the PRV bit is negated. The relative priorities of the OPB Masters are then determined by the connection of master devices to the request/grant signals. The values of the Priority Registers are used in OPB arbitration whenever the PRV bit is asserted.

Fixed priority arbitration for a 4 OPB Master system with combinational grant outputs is shown in Figure 8. Figure 9 shows the same system when configured with registered grant outputs with the master requests separated by an additional clock.

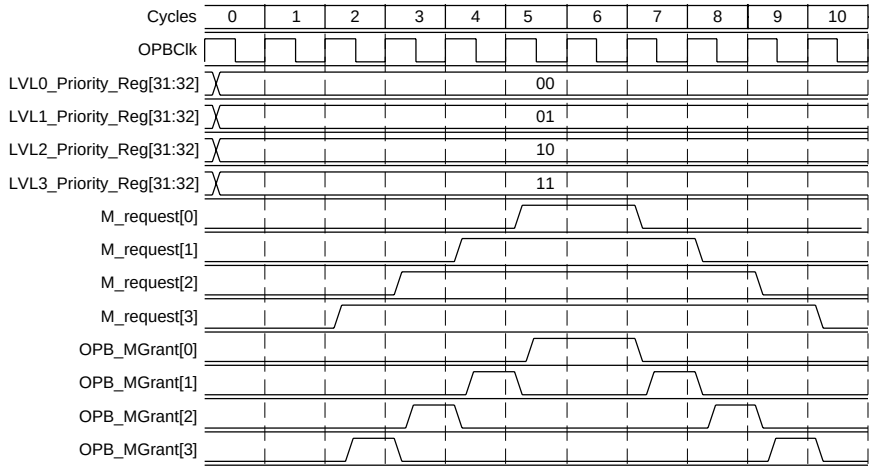


Figure 8: **Fixed Priority Arbitration, Combination Grant Outputs for 4 OPB Masters**

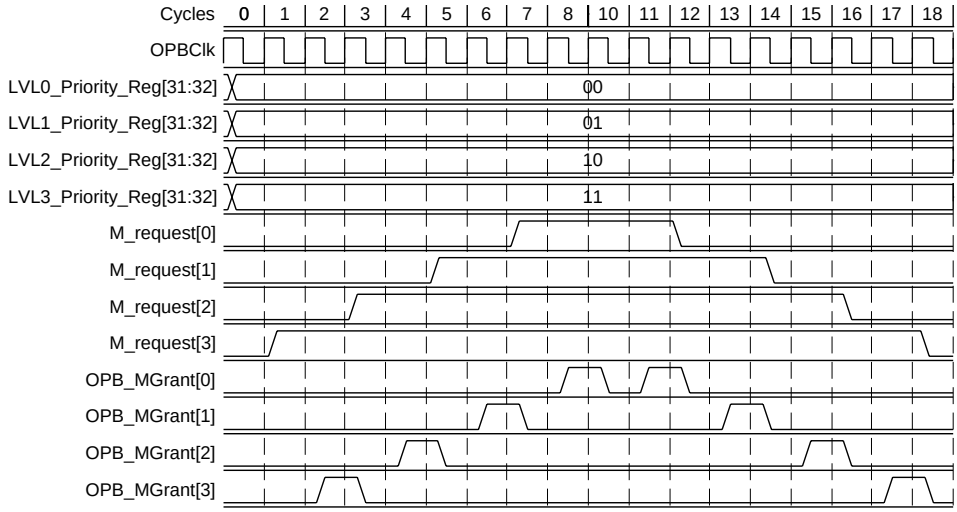


Figure 9: **Fixed Priority Arbitration, Registered Grant Outputs for 4 OPB Masters**

Dynamic Priority Parameterization (C_DYNAM_PRIORITY=1)

When the OPB Arbiter is parameterized to support dynamic priority arbitration, dynamic priority arbitration mode is enabled at reset or at any other time by writing a “1” to the DPE bit of the Control Register. Disabling dynamic priority arbitration mode by setting the DPE bit to “0” puts the OPB Arbiter into a fixed priority arbitration mode. This effectively freezes the values of the Priority registers (unless updated by an OPB write to the registers) and thus its ordering of arbitration priorities among the attached master devices. Setting the master priorities by software and not allowing them to update results in a static assignment of priority among the OPB masters.

Upon reset, the Priority registers contains the reset values as described in Table 8 and the dynamic priority bit is enabled. When dynamic priority arbitration mode is enabled, the contents of the Priority Registers are reordered after every request-grant cycle, moving the ID of the most recently granted master to the lowest Priority register and moving all other master IDs up one level of priority. The dynamic priority arbitration mode operation results in an implementation of the least recently used (LRU) algorithm. The lowest priority master ID will be

the one which was granted the bus most recently, and the highest priority master ID will be the one which was granted the bus the least in the recent past.

The values to be loaded into the Priority registers are either the values written to the register from the OPB or a shift of the master ID from the next lowest Priority register. In the case of the low Priority register, the ID of the master last granted the bus is loaded into this register. The master ID of the master granted the bus is loaded into the lowest Priority register and the IDs in all other Priority registers up to the priority of the one just granted move up one position in priority. The Priority registers above the one just granted hold their master ID values.

A pipeline register exists between the arbitration logic and Priority Register update logic which delays the master grant signals by an additional clock. Therefore, if the OPB Arbiter is configured for dynamic priority arbitration and registered grant outputs, the master grant signals will be output 2 clocks after a valid arbitration cycle. Dynamic priority arbitration for a 4 OPB master system with combinational grant outputs is shown in Figure 10. Figure 11 shows dynamic priority arbitration for a 4 master OPB system with registered grant outputs.

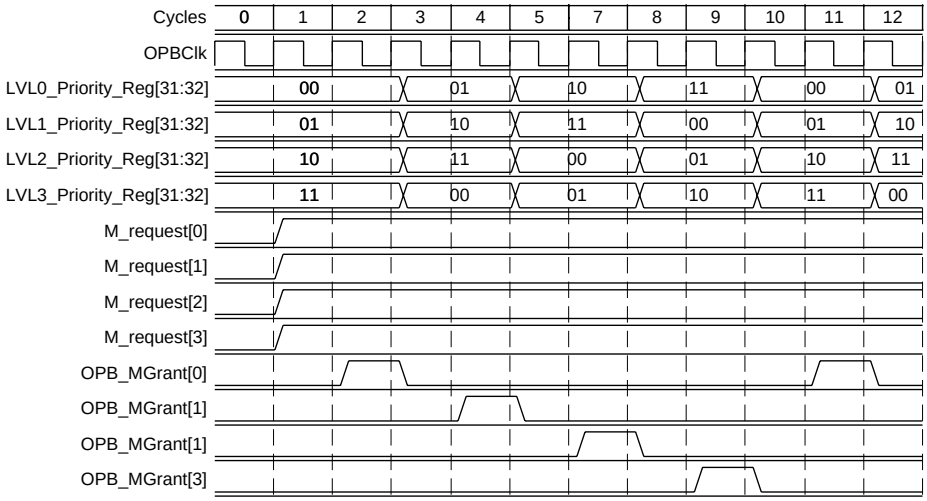


Figure 10: **Dynamic Priority Arbitration, Combinational Grant Outputs- 4 OPB Masters**

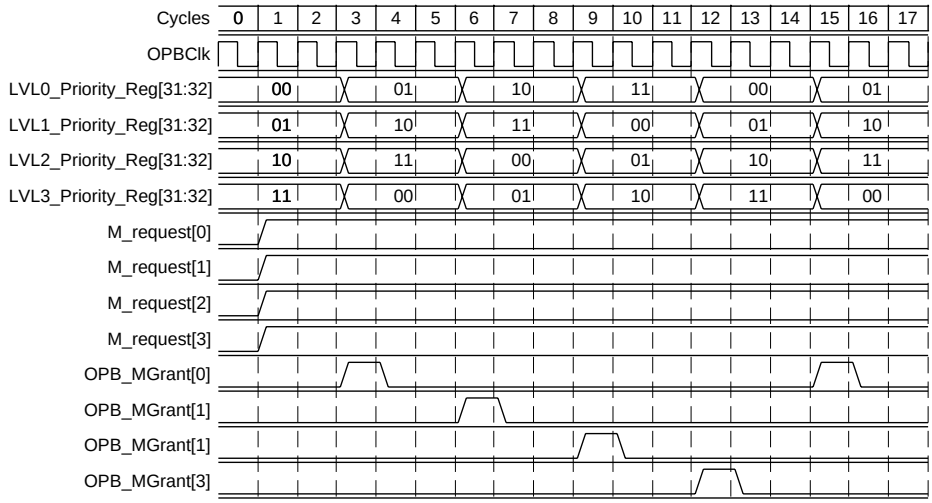


Figure 11: **Dynamic Priority Arbitration, Registered Grant Outputs- 4 OPB Masters**

Block Diagram

The block diagram for the Priority registers and the register update logic is shown in Figure 12. The gray shaded blocks represent the Priority Register update logic which is only present when the OPB Arbiter is parameterized to support Dynamic Priority arbitration.

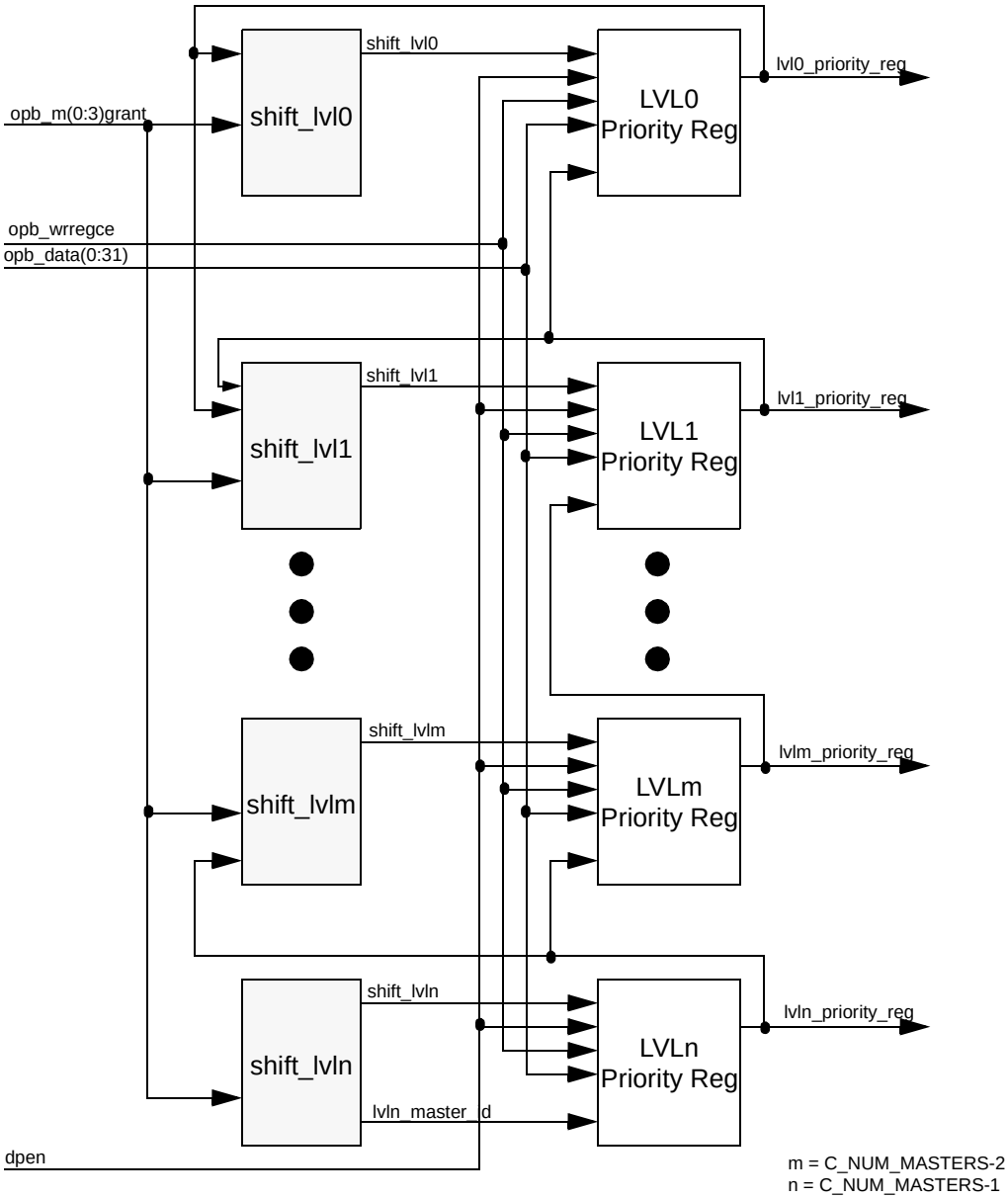


Figure 12: **Priority Register Logic**

When the OPB Arbiter has been parameterized to support a processor interface (C_PROC_INTRFCE=1), the Priority registers can still be loaded by the processor, allowing the processor to change the priorities of the OPB Masters. Also, the arbiter can be set to operate in a fixed priority mode by the processor writing to the Control Register and negating the DPE bit. However, the pipeline registers between the arbitration logic and the register update logic are still present, thereby delaying the grant outputs by an additional clock cycle.

ARB2BUS Data Mux

When a read of the OPB Arbiter Priority Registers or the OPB Arbiter Control Register is requested on the OPB, the ARB2BUS Data Mux outputs the requested data to the IPIF which sends this data to the OPB with the required protocol. This logic is only present in the design if C_PROC_INTRFCE=1 and C_NUM_MASTERS>1.

Arbitration Logic

Figure 13 depicts the functional block diagram of arbitration logic for the OPB arbiter. This logic is only present in the design if C_NUM_MASTERS>1. The pipeline registers are only present in the arbitration logic if C_DYNAM_PRIORITY = 1.

All master request signals are input to the Prioritize Request block which consists of multiplexors which prioritize the master's requests into the signals lvl0_req, lvl1_req, lvlm_req, and lvln_req based on the requesting masters' priorities if PRV =1 or the master IDs if PRV = 0. (n = C_NUM_MASTERS-1, m= C_NUM_MASTERS-2)

The prioritized request signals are then input into the priority encoder which determines which priority grant is asserted, i.e., grant_lvl0, grant_lvl1, grant_lvlm, and grant_lvln.

The prioritized grant signals are then input to the Assign Grants block to determine which master's grant signal is asserted based on the priority of that master, again determined by examination of the master IDs in the Priority Registers if PRV = 1, or the master IDs if PRV = 0. The master's priority code selects the appropriate prioritized grant signal to be output to that master. These intermediate grant signals are then registered if the OPB Arbiter is configured to support dynamic priority arbitration to reduce the number of logic levels between the arbitration logic and the Priority Register update logic. Since the Priority Register update logic is not present when the OPB Arbiter is not configured to support dynamic priority arbitration, these pipeline registers are not necessary and therefore are not present in the design.

The arbitration logic also contains the logic for detecting valid arbitration cycles which is input to the Park/Lock logic. Valid arbitration cycles are defined as when either the OPB_select signal is deasserted indicating no data transfer is in progress or when OPB_XferAck is asserted, indicating the final cycle in a data transfer and OPB_busLock is not asserted.

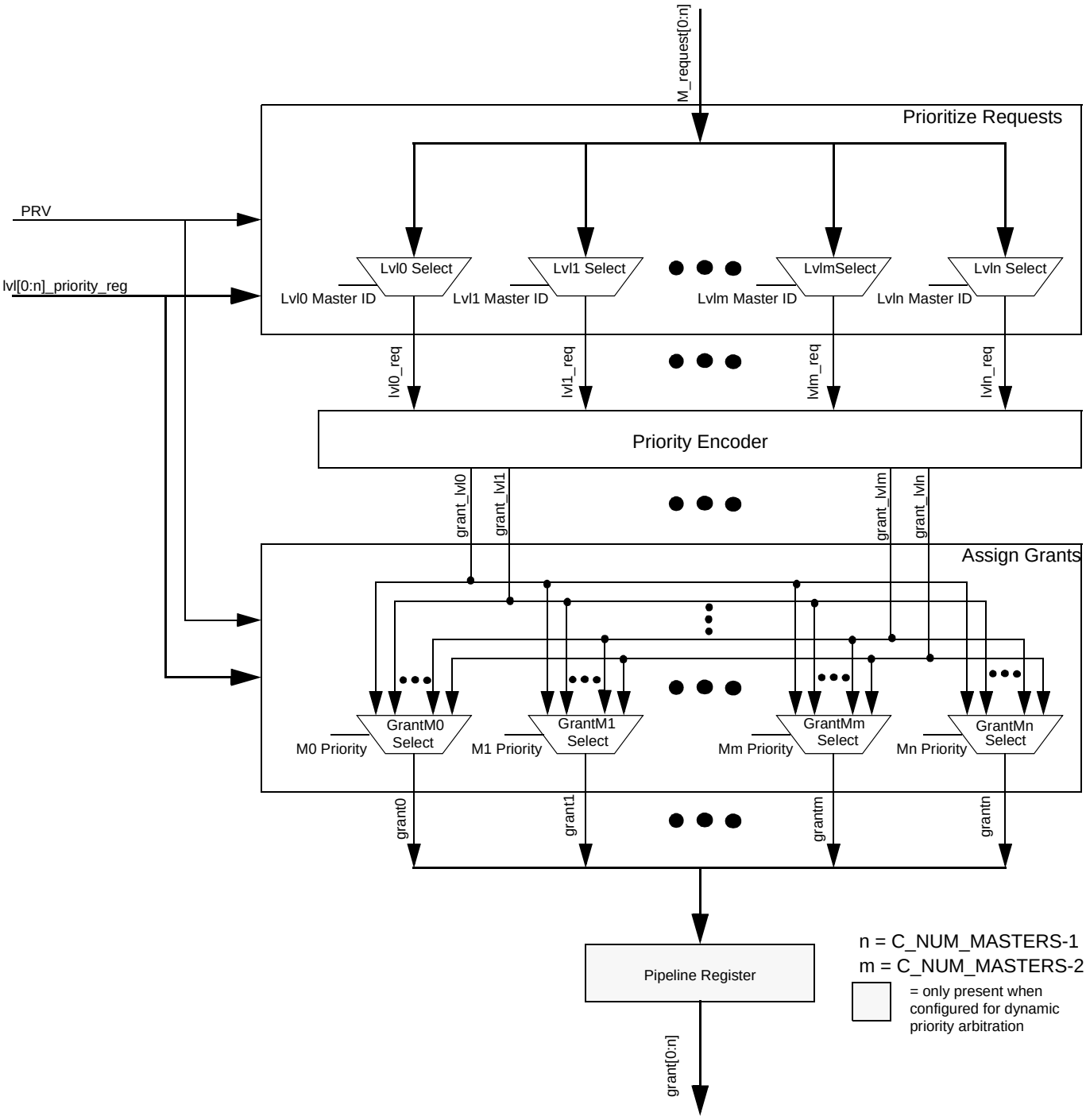


Figure 13: **Arbitration Logic**

Park/Lock Logic

The Park/Lock logic processes the raw or registered grant outputs from the Arbitration Logic and is only present in the design if C_NUM_MASTERS>1. It provides for the parking (if C_PARK=1) and locking functionality of the OPB Arbiter and generates the final grant signals which are sent to the OPB master devices during valid arbitration cycles as shown in Figure 14. The OPB Arbiter can be parameterized to register the master grant signals by setting the parameter C_REG_GRANTS to 1.

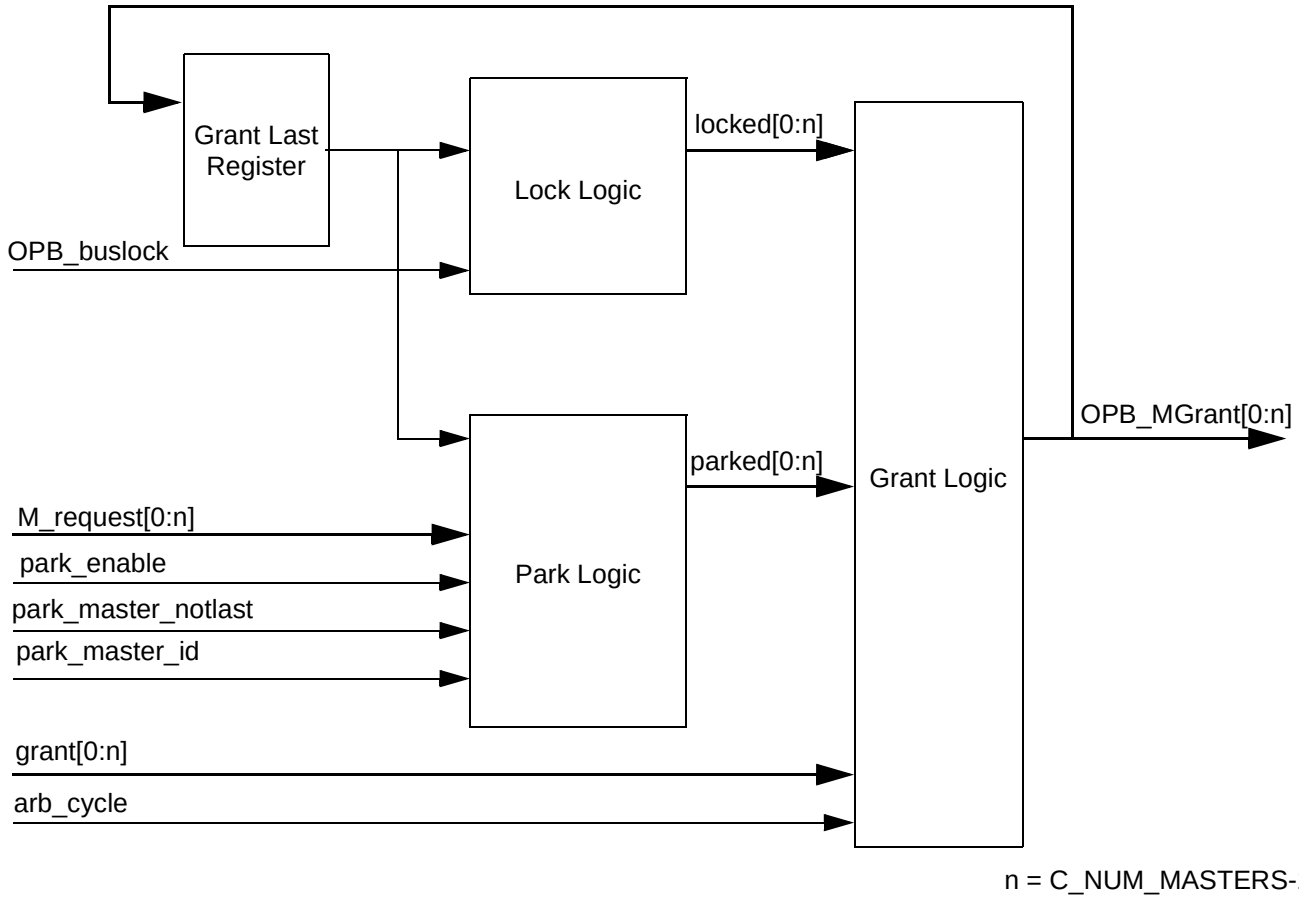


Figure 14: **Park/Lock Logic**

Grant Last Register

The Grant Last Register holds the state of the grant outputs from the last request/grant arbitration cycle. This information is used to determine which master currently has control of the bus to implement bus locking and park on last master parking.

Lock Logic

If an OPB master asserts the bus lock signal upon assuming control of the bus, the OPB arbiter will continue to grant the OPB to the master which locked the bus. Bus lock signals from all attached masters are ORed together to form OPB_busLock, which is an input to the OPB Arbiter. When OPB_busLock is asserted, bus arbitration is locked to the last granted master (as indicated by the Grant Last register). All other master Grant outputs are gated off and will not be asserted, regardless of the state of the Request inputs or the programmed priorities.

When the OPB bus is locked, bus request and grant signals have no effect on bus arbitration. The OPB master may proceed with data transfer cycles while asserting bus lock without engaging in bus arbitration and without regard to the state of the request and grant signals. Grant signals will be generated if the master asserts its request signal. The locked master's

grant signal will be asserted in response to its request signal during valid arbitration cycles. However, the locked master need not assert its request or receive an asserted grant signal to control the bus. The master owns the bus by virtue of asserting its bus lock signal after being granted the bus and before another valid arbitration cycle.The master which asserted bus lock will retain control of the bus until bus lock is deasserted for at least one complete cycle. The OPB arbiter will detect the bus lock signal and will continue to grant the bus to the current master, regardless of other (higher priority) requests. Figure 15 shows the OPB bus lock operation when the OPB Arbiter is configured for fixed priority arbitration and combinational grant outputs. Figure 16 shows the OPB bus lock operation when the OPB Arbiter is configured for fixed priority arbitration and registered grant outputs. Note that the bus grant signal is asserted one clock later.

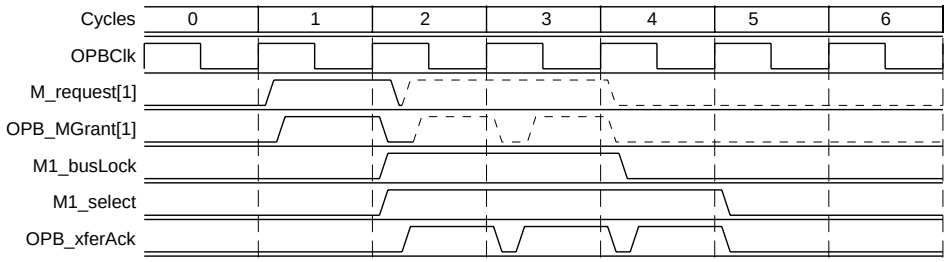


Figure 15: **Bus Locking - Fixed Priority, Combinational Grant Outputs**

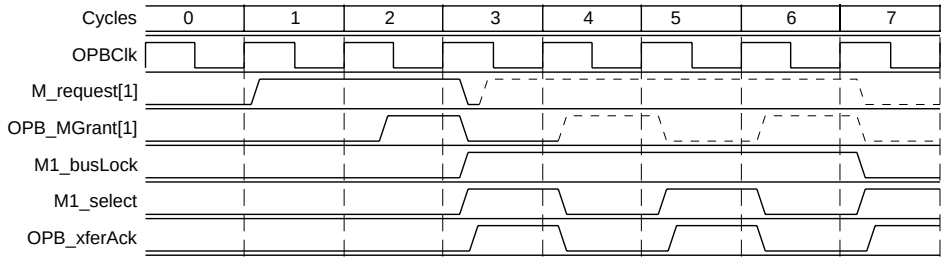


Figure 16: **Bus Locking - Fixed Priority, Registered Grant Outputs**

Park Logic

When C_PARK=1, the OPB Arbiter supports bus parking. Bus parking is when a master's grant signal is asserted during valid arbitration cycles when no other master devices are requesting. This reduces latency for the parked master eliminating the need for a request/grant cycle when initiating a new OPB transfer.

Asserting a grant signal for parking is considered an arbitration, since it determines which device controls the bus. If dynamic priority mode is enabled, the ID of the parked master will be shifted to the lowest priority slot of the Priority Register. The master will remain parked (its grant signal asserted) so long as no other master asserts a request signal. If the parked master and another master assert request at the same time, the parked master will control the bus because the bus was parked on this master even though this master is at a lowest priority. Figure 17 illustrates this behavior when the OPB Arbiter is configured to support fixed priority arbitration and combinational grant outputs. Figure 18 illustrates this behavior when the OPB Arbiter is configured to support dynamic priority arbitration when the OPB Arbiter is configured to support dynamic priority arbitration and registered grant outputs.

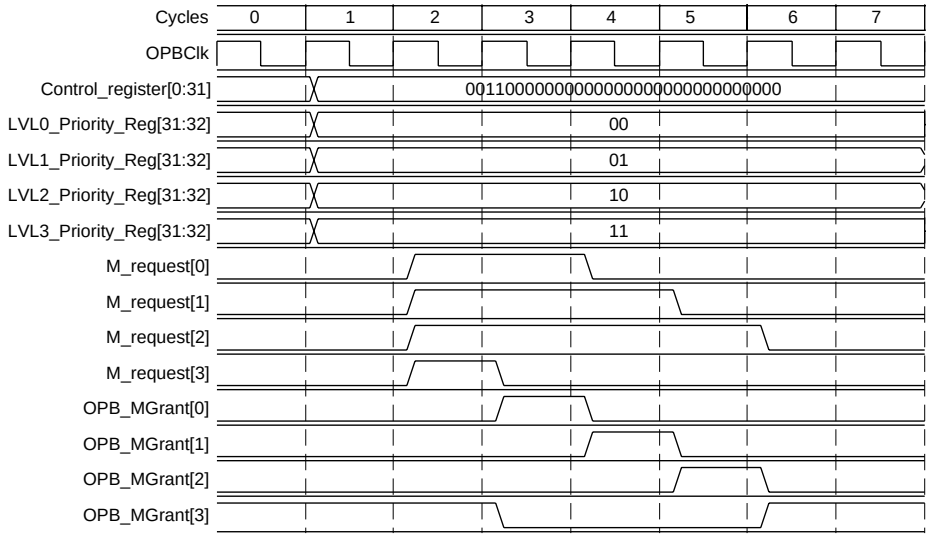


Figure 17: **Bus Parking - Fixed Priority Arbitration, Combinational Grant Outputs**

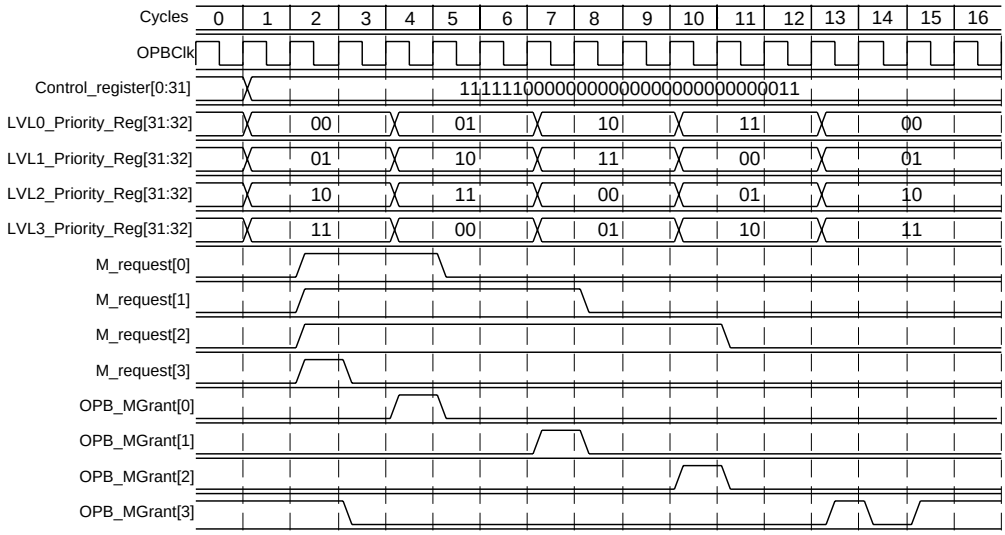


Figure 18: **Bus Parking - Dynamic Priority Arbitration, Registered Grant Outputs**

When the grant outputs are registered, the parked master's grant will assert once, negate, and then stay asserted. This allows the internal arbitration state machine a clock cycle to check if OPB_select is asserted before parking. This case occurs when a request from a master is aborted, but the grant is in the internal pipeline.

When C_PARK = 1, bus parking is enabled or disabled by the value of the Park Enable control bit in the OPB Arbiter Control Register (see Table 6). The bus can either be parked on the master who was last granted the bus, or on a specified master as indicated by the Park Master ID bits in the OPB Arbiter Control Register. If bus parking is not desired, the park logic can be eliminated by setting C_PARK=0.

Park on Master Not Last

When bus parking is enabled (bit PEN = 1 in the OPB Arbiter Control Register and C_PARK=1) and Park on Master Not Last is selected (bit PMNL = 1 in the OPB Arbiter Control Register), the bus will be parked on the master whose ID is contained in the Park Master ID (PMID) field of the OPB Arbiter Control Register. This master's grant signal will be asserted during valid arbitration cycles when no other master's request signal is asserted. Figure 19 shows bus parking on the

master specified in the Control Register for a 4 OPB master system when the OPB Arbiter is parameterized for fixed priority arbitration and combinational grant outputs.

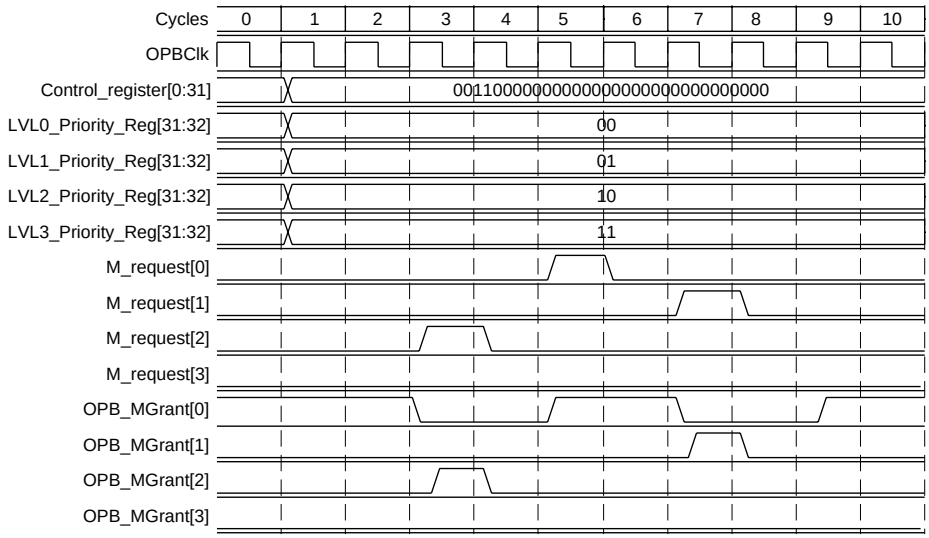


Figure 19: **Bus Parking on Master Not Last - Fixed Priority Arbitration, Combinational Grant Outputs**

Park on Last Master

When bus parking is enabled (bit PEN = 1 in the OPB Arbiter Control Register and C_PARK=1) and Park on Master Not Last is negated (bit PMNL = 0 in the OPB Arbiter Control Register), the bus will be parked on the master which was most recently granted the bus as indicated by the Grant Last register. This master's grant signal will be asserted during valid arbitration cycles when no other master's request signal is asserted. Figure 20 shows bus parking on the last master with the OPB Arbiter parameterized for fixed priority arbitration and combinational grant outputs for a 4 OPB Master system.

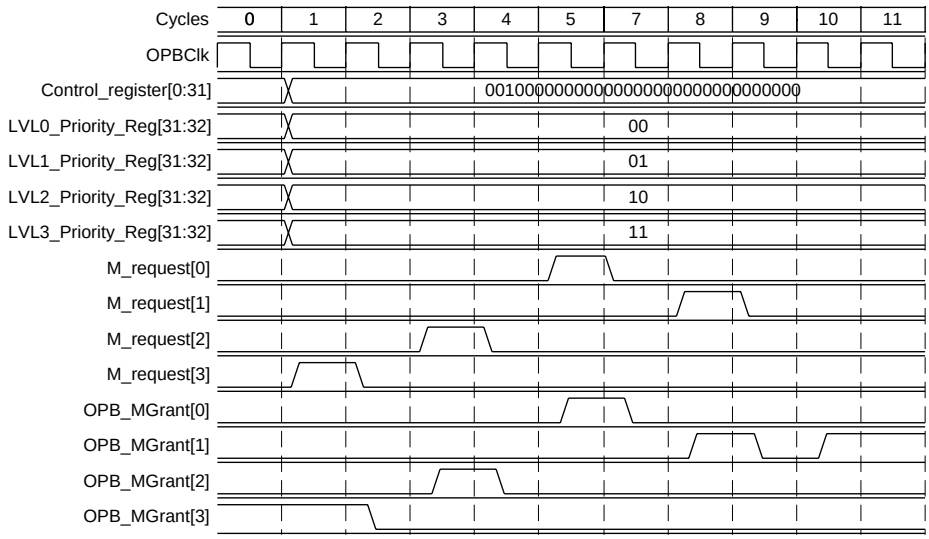


Figure 20: **Bus Parking on Last Master - Fixed Priority Arbitration, Combinational Grant Outputs**

Grant Logic

The Grant Logic block determines the final grant signals to the OPB Masters based on the intermediate grant signals from the arbitration logic and the results of the park and lock logic. The OPB Arbiter can be parameterized so that the grant signals are either registered outputs are combinational outputs. Registering the grant signals allows for higher OPB clock frequencies at the cost of a 1-cycle arbitration latency. Combinational grant outputs allow the grant signals to be asserted within the same clock cycle as the Master request signals, however, the overall clock frequency of the OPB will be affected. Basic OPB arbitration using registered grant outputs is shown in Figure 2.

Watchdog Timer

The watchdog timer generates the OPB_timeout signal within the 16th cycle following the assertion of OPB_select if there is no response from a slave (OPB_xferAck or OPB_retry) and if toutSup is not asserted by an addressed slave device to suppress the timeout. This logic is always present in the OPB Arbiter design.

Upon assertion of OPB_timeout, the master device which initiated the transfer cycle must terminate the transfer by deasserting Mn_select in the cycle following the assertion of OPB_timeout as shown in Figure 21. If OPB_busLock is not asserted, the OPB Arbiter will perform a bus arbitration in the cycle in which OPB_select is deasserted. If OPB_busLock is asserted, the requesting master retains control of the OPB, but must still deassert Mn_select following the assertion of OPB_timeout for at least one cycle.

If OPB_xferAck or OPB_retry are asserted in the 16th cycle following the assertion of Mn_select coincident to the assertion of OPB_timeout, the master device should ignore OPB_timeout, and respond to the slave's OPB_xferAck or OPB_retry signal.

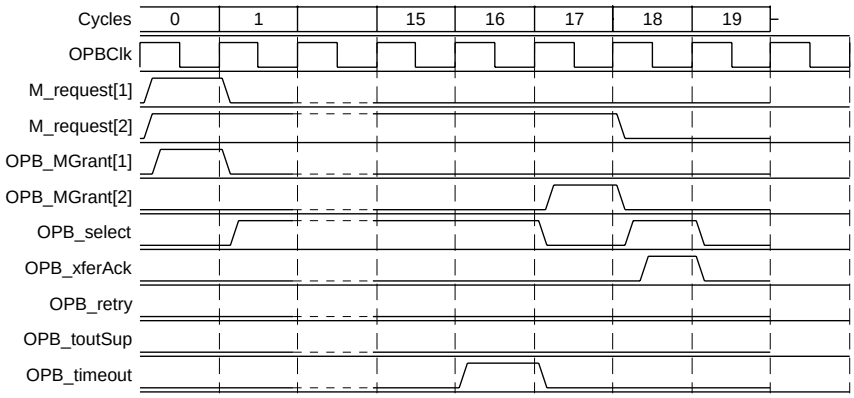


Figure 21: **OPB Timeout Error**

To prevent a bus timeout, an OPB slave must assert OPB_toutSup (OR of all slave's Sln_ToutSup) within 16 cycles from the assertion of OPB_select. OPB_toutSup will be used by the OPB Arbiter to suppress the assertion of OPB_timeout and to suspend the timeout counter. When OPB_toutSup is asserted, the timeout counter holds its current value. When OPB_toutSup is negated , the timeout counter resumes counting. OPB timeout error suppression is shown in Figure 22.

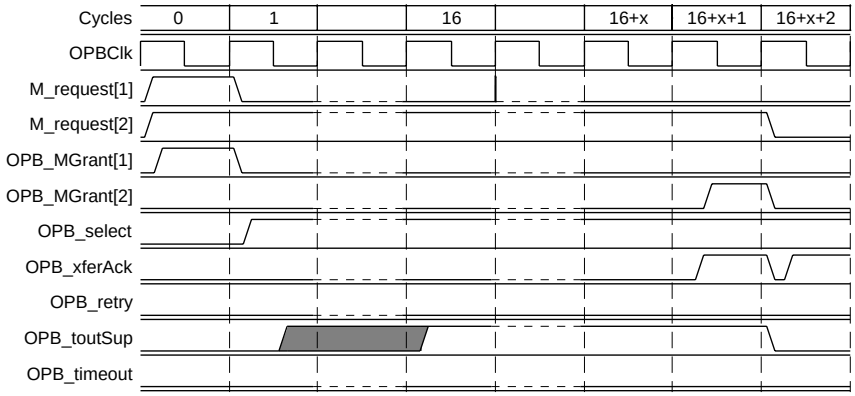


Figure 22: **OPB Timeout Error Suppression**

Design Implementation

Device Utilization and Performance Benchmarks

Since the OPB Arbiter is a module that will be used with other design pieces in the FPGA, the utilization and timing numbers reported in this section are just estimates. As the OPB Arbiter is combined with other pieces of the FPGA design, the utilization of FPGA resources and timing of the OPB Arbiter design will vary from the results reported here.

In order to analyze the OPB Arbiter's timing within the FPGA, a design was created that instantiated the OPB Arbiter with registers on all of the OPB Arbiter inputs and outputs. This allowed a constraint to be placed on the clock net for the OPB Arbiter to yield more realistic timing results. The f_{MAX} parameter shown in Table 9 was calculated with registers on the OPB Arbiter inputs and outputs. However, the resource utilizations reported in Table 9 do not include the registers on the OPB Arbiter inputs and outputs.

The OPB Arbiter benchmarks are shown in Table 9 for a Virtex™-II -5 FPGA using multi-pass place and route.

Table 9: **OPB Arbiter FPGA Performance and Resource Utilization Benchmarks (Virtex™-II -5)**

Parameter Values						Device Resources			f_{MAX} (MHz)
C_NUM_MASTERS	C_DYNAM_PRIORITY	C_PARK	C_PROC_INTRFCE	C_REG_GRANTS	C_OPB_DWIDTH/ C_OPB_AWIDTH	Slices	Slice Flip-Flops	4-input LUTs	f_{MAX}
1	N/A	N/A	N/A	N/A	N/A	4	4	6	290
2	0	0	0	0	32/32	11	7	18	223
2	1	1	1	1	32/32	76	76	87	163
4	0	0	0	0	32/32	27	13	39	153
4	1	0	0	0	32/32	42	21	69	167
4	0	1	0	0	32/32	42	17	59	150
4	0	0	1	0	32/32	75	63	95	136
4	0	0	0	1	32/32	30	18	40	173
4	1	1	1	1	32/32	130	97	163	126
8	0	0	0	0	32/32	93	31	132	122

Table 9: **OPB Arbiter FPGA Performance and Resource Utilization Benchmarks (Virtex™-II -5) *(Continued)***

8	1	1	1	1	32/32	278	145	398	100
16	0	0	0	0	32/32	441	84	654	100
16	1	1	1	1	32/32	904	252	1477	82

Notes:
1. These benchmark designs contain only the OPB Arbiter with registered inputs/outputs without any additional logic. Benchmark numbers approach the performance ceiling rather than representing performance under typical user conditions.
2. Different OPB data widths and OPB address widths are verified as part of the IPIF verification.
3. Device resource numbers do not include the registers for the OPB Arbiter I/O.
4. Max frequency calculated with registers on the OPB Arbiter I/O.

Specification Exceptions

Register Definitions and Addressing

To support parameterization of the number of masters, the Xilinx OPB Arbiter uses a separate Priority Register for each priority level. Since the number of Priority Registers can vary, the OPB Arbiter Control Register is placed at the base address of the OPB Arbiter so that its location doesn't vary.

Since the number of bits required for the master IDs will vary with the number of masters, the fields in the Control Register and the Priority Registers that contain master IDs are LSB aligned. The bit ordering of these registers is therefore different than that specified in the IBM OPB Arbiter specification. The Control Register also contains additional bits (DPERW, PENRW, PRV) not in the IBM OPB Arbiter specification. The DPERW bit indicates whether the DPE bit can be modified, the PENRW bit indicates whether the PEN bit can be modified, and the PRV bit indicates whether the Priority Registers contain valid data, i.e, all master IDs are contained in a Priority Register.

I/O Signals

The signal ARB_DbusEn is no longer an I/O signal. The gating of the OPB Arbiter's data bus with the enable signal is done internally within the IPIF Module.

The master request signals and master grant signals have been combined into a bus with an index that varies with the number of masters. This modification more easily supports the parameterization of the number of masters supported by the Xilinx OPB Arbiter. Table 12 summarizes the I/O signal name modifications and variations.

Table 12: **Xilinx OPB Arbiter I/O Signal Variations**

IBM OPB Arbiter Signal Name	Xilinx OPB Arbiter Signal Name
ARB_DBusEn	no longer an I/O signal
ARB_XferAck	Sl_XferAck
M0_request, M1_request, M2_request, M3_request	M_request[0:C_NUM_MASTERS-1]
OPB_M0Grant, OPB_M1Grant, OPB_M2Grant, OPB_M3Grant	OPB_MGrant[0:C_NUM_MASTERS-1]

Priority Level Nomenclature

The IBM OPB Arbiter refers to the priority levels as High, Medium High, Medium Low, and Low since it only implemented arbitration among 4 OPB masters. Since the Xilinx implementation of the OPB arbiter supports parameterization of the number of OPB masters in the system, it was decided that numbers should be used to represent priority levels instead of text descriptors. Level 0 will always remain the highest priority level regardless of the number of masters implemented. The higher the level number, the lower the priority.

Grant Outputs

The IBM OPB Arbiter only outputs combinational bus grant signals for both fixed and dynamic priority arbitration. The Xilinx OPB Arbiter can be configured to output combinational or registered bus grants. Also, when the Xilinx OPB Arbiter is configured to support dynamic priority arbitration, the bus grants will be output one clock after the arbitration cycle due to a pipeline register between the arbitration logic and the Priority Register update logic.

Bus Parking

When utilizing dynamic priority arbitration and the bus is parked on a particular bus master, this bus master is moved to lowest bus priority. In the IBM OPB Arbiter, if another master requests the bus at the same time as the parked bus master, the higher priority master will gain control of the bus. In the Xilinx OPB Arbiter, the parked bus master will gain control of the bus, even though this master is at a lower priority.

Clock and Power Management

The IBM OPB Arbiter Core supports clock and power management by gating clocks to all internal registers and providing a sleep request signal to a central clock and power management unit in the system. This sleep request signal is asserted by the IBM OPB Arbiter to indicate when it is permissible to shut off clocks to the arbiter. These functions are not supported in the Xilinx implementation of the OPB Arbiter, therefore the following I/O signal is not used:

- ARB_sleepReq -the FPGA implementation of the OPB bus will not support sleep modes

Scan Test Chains

The IBM OPB Arbiter contains an internal scan chain for testing and verification purposes. Xilinx FPGAs support boundary scan testing but the internal flip-flops within the architecture do not provide for an internal scan chain. Therefore, the internal scan chain implemented in the IBM OPB Arbiter is not supported in the Xilinx implementation and the following I/O signals are not used:

- LSSD_AClk - FPGA implementation does not support scan
- LSSD_BClk - FPGA implementation does not support scan
- LSSD_CClk - FPGA implementation does not support scan
- LSSD_scanGate - FPGA implementation does not support scan
- LSSD_scanIn - FPGA implementation does not support scan
- LSSD_scanOut - FPGA implementation does not support scan

Reference Documents

The following documents contain reference information important to understanding the OPB Arbiter design:

- IBM 64-Bit On-Chip Peripheral Bus Architectural Specification (v2.0)
- IBM On-Chip Peripheral Bus Arbiter Core User's Manual (v1.5), 32-Bit Implementation

OPB Simple Interrupt Controller Specification

March 2002

Summary

This document describes the specifications for a Simple Interrupt Controller for use in Xilinx FPGAs. This document applies to the following peripherals:

opb_intc	v1.00b

Overview

A Simple Interrupt Controller is composed of a bus-centric wrapper that contains the IntC core and a bus interface. The IntC core is a simple, parameterized interrupt controller that, along with the appropriate bus interface, attaches to either the OPB (On-chip Peripheral Bus) or the DCR (Device Control Register) Bus. It can be used in embedded PowerPC systems (Virtex-II PRO devices), and in MicroBlaze soft processor systems. There are currently two versions of the Simple Interrupt Controller:

- OPB IntC (OPB interface)
- DCR IntC (DCR interface)

In this document IntC and Simple IntC are used interchangeably to refer to functionality or interface signals that are common to all variations of the Simple Interrupt Controller. However, when its necessary to make a distinction, the interrupt controller is referred to as OPB IntC or DCR IntC.

Features

A Simple Interrupt Controller has the following features:

- Modular design provides a core interrupt controller functionality that is instantiated within a bus interface design (currently the OPB and DCR buses are supported)
- OPB V2.0 bus interface with byte-enable support (IBM SA-14-2528-01 64-bit On-chip Peripheral Bus Architecture Specifications, Version 2.0)
- Supports data bus widths of 8-bits, 16-bits, or 32-bits for OPB interface
- Number of interrupt inputs is configurable up to the width of the data bus
- Easily cascaded to provide additional interrupt inputs
- Interrupt Enable Register for selectively disabling individual interrupt inputs
- Master Enable Register for disabling the interrupt request output
- Each input is configurable for edge or level sensitivity — edge sensitivity can be configured for rising or falling; level sensitivity can be active-high or -low
- Automatic edge synchronization when inputs are configured for edge sensitivity
- Output interrupt request pin is configurable for edge or level generation — edge generation configurable for rising or falling; level generation configurable for active-high or -low

Interrupt Controller Overview

Interrupt controllers are used to expand the number of interrupt inputs a computer system has available to the CPU and, optionally, provide a priority encoding scheme. Modern CPUs provide one or more interrupt request input pins that allow external devices to request service

by the CPU. There are two main interrupt request mechanisms used by CPUs. Auto vectoring interrupt schemes provide an interrupt request signal to the processor and during the interrupt acknowledge cycle, the interrupt controller provides all or some portion of the address of the interrupt service routine. Hard vector interrupt schemes provide one or more fixed locations in memory, one for each interrupt request input, or one location for all interrupt inputs. In either case, some interrupt controllers allow you to program the polarity of the interrupt inputs and whether they are level or edge sensitive. Some may allow the priority of an interrupt to be programmed as well. Some popular embedded processors and their associated interrupt controllers / mechanisms are described in the following paragraphs.

Intel 8051

As in many single chip solutions, the interrupt controller for the 8051 is embedded into the functionality of the CPU and on-chip peripherals. The 8051 micro controller utilizes a hard vector approach for handling interrupts. There is a unique, hard vector address associated with external interrupt 0, timer 0, external interrupt 1, timer 1 and the serial port. You can program interrupts for high or low priority. There is an interrupt enable bit for each interrupt source as well as a bit for enabling or disabling all interrupts. You can program the two external interrupt inputs to be either edge sensitive (falling edge) or level sensitive (active low).

Zilog Z80

The Z80 supports both hard vector and auto vector modes for interrupts. The Non-Maskable Interrupt (NMI) input utilizes a hard vector and cannot be disabled by software. This interrupt is an active low, level sensitive interrupt. The other interrupt input (INT), which is also an active low, level sensitive interrupt, supports three different modes. Mode 0 provides compatibility with the 8080 microprocessor. During an interrupt acknowledge cycle the interrupting device jams a restart instruction onto the data bus, which causes program execution to continue at one of eight hard coded locations. Mode 1 is a hard vector interrupt with a single hard vector, similar to the NMI but at a different location. Mode 2 is a fully auto vectored interrupt mode. In this mode the interrupt controller is actually distributed between the processor and the Z80 family peripherals. During an interrupt acknowledge cycle the interrupting device places the low eight bits of the interrupt service routine address on the data bus. The processor provides the upper eight bits from a dedicated register that is loaded by software. NMI always has a higher priority than INT. Multiple devices can be attached to either interrupt input using a wired-or configuration. Additionally, in Mode 2, devices on the INT input can be daisy-chained to provide additional interrupt priorities. The INT input can be masked by software.

Motorola 68332

The 68332 has seven active low, level sensitive interrupt request inputs (IRQ1 to IRQ7). These inputs correspond to the seven interrupt request levels of the CPU32 core. Devices (internal or external) request service by activating a particular interrupt level. If that level is not masked then the processor acquires the appropriate interrupt vector number and obtains the service routine address from a 256 location interrupt vector table, indexed by the interrupt vector number. The interrupt vector number is determined on a per device basis, and is either at a fixed location relative to the interrupt request level or is supplied by the interrupting device as part of the interrupt acknowledge cycle. Interrupt request level seven is non-maskable and the other six levels can be masked by software. Interrupt request level one has the lowest priority and interrupt request level seven has the highest priority. All the peripherals on-chip can be programmed to request an interrupt on any of the seven levels. Interrupt request levels one through six behave as level sensitive interrupts in that as long as that level is active interrupt requests will be generated. The level must be maintained by the interrupting device until the interrupt has been acknowledged by the processor. Interrupt request level seven behaves like an edge sensitive interrupt since only one interrupt request is generated each time that level is entered and the level must be exited and re-entered to generate another interrupt.

MIPS

MIPS CPUs have eight interrupt sources, six of which are for hardware interrupts and the remaining two are for software interrupts. There is no priority, all interrupts are considered

equal. Each interrupt can be masked by the software and there is a global interrupt enable. MIPS only supports a hard vector mechanism but the vector address can be programmed to be in a non-cachable or cachable memory segment. An external interrupt controller could provide additional interrupt inputs and a priority encoding scheme but there is no mechanism for providing auto vectoring. As a result, the job of prioritizing interrupts and branching to the proper interrupt service routine is done by the software.

ARM

The ARM architecture provides two external interrupt inputs: FIQ and IRQ. FIQ is higher priority than IRQ but both can be masked by software. Each interrupt has a hard vector associated with it and its own status register and subset of general purpose registers. An external interrupt controller could provide additional interrupt inputs with priority but there is no auto vectoring capability.

IBM PowerPC 405GP Universal Interrupt Controller (UIC)

The UIC for the PowerPC 405GP provides 19 internal and 7 external interrupts. Eighteen of the internal interrupts are active high, level sensitive. The other internal interrupt is edge sensitive and active on the rising edge. The seven external interrupts are programmable as to polarity and sensitivity. Each interrupt source can be programmed to source the critical or non-critical input to the 405 core. All interrupts can be masked and the current interrupt state can be read by the processor. The UIC supports prioritized auto vectoring for the critical interrupts, either through a vector table or the actual address of the service routine. The UIC does not support vector generation for the non-critical interrupts, relying instead on the interrupt vector generation mechanism within the 405 core. This mechanism is similar to a hard vector, except that the vector used is programmable by the software.

Simple IntC

The interrupt controller described in this document is intended for use in a hard vector interrupt system. It does not directly provide an auto vectoring capability. However, it does provide a vector number that can be used in a software based vectoring scheme. Basic terminology and pros and cons of edge and level sensitive inputs are described in the remainder of this section. The functionality of the IntC is described in the sections that follow.

Edge Sensitive Interrupts

Figure 23 illustrates the three main types of edge generation schemes, using rising edges for the active edge in this example. In all three schemes, the device generating the interrupt provides an active edge and some time later the generator produces an inactive edge in preparation for generating a new interrupt request. In the first scheme, the inactive edge is depicted as occurring when the interrupt is acknowledged. This is identical to generating a level sensitive interrupt. The second scheme shows the inactive edge occurring immediately after the active edge. The third scheme shows the inactive edge occurring immediately before the active edge. All three schemes are possible and should be detected by the interrupt detection circuitry without missing an interrupt or causing spurious interrupts.One potential problem with edge sensitive interrupt schemes is their susceptibility to noise glitches. Also, it may be more difficult to remember and propagate multiple interrupts when the interrupt service routine does not handle all active interrupts. In non-auto vectoring interrupt designs, it may be necessary for the software interrupt handler to service the highest priority interrupt and then check the status for any additional interrupts that may have arrived before returning from the interrupt handler. Synchronization logic is usually necessary to avoid metastability problems with asynchronous inputs.

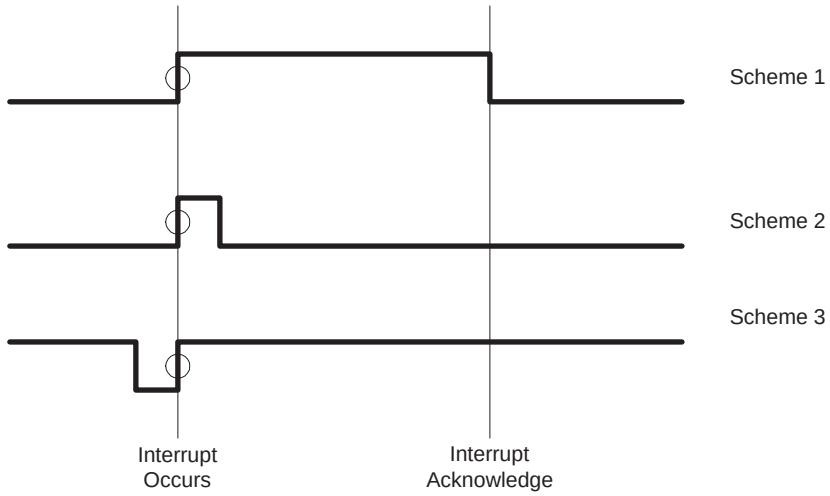


Figure 23: **Schemes for Generating Edges**

Level Sensitive Interrupts

In principle, level sensitive interrupts are somewhat simpler to manage. They are simpler to propagate when multiple sources are present and usually don't require additional synchronization logic. The major problem with level sensitive interrupts stems from their inherent susceptibility to spurious interrupts, and to missed interrupts due to problems that arise when trying to avoid spurious interrupts.

Simple Interrupt Controller Organization

The Simple IntC is organized into the following three functional units:

- Interrupt detection and request generation
- Programmer registers
- Bus interface

Interrupt Detection

Interrupt detection can be configured for either level or edge detection for each interrupt input. If edge detection is chosen, synchronization registers are also included. Interrupt request generation is also configurable as either a pulse output for an edge sensitive request or as a level output that is reset when the interrupt is acknowledged.

Programmer Registers

The interrupt controller contains the following programmer accessible registers:

- Interrupt Status Register (ISR) is a read/write register that, when read, indicates which interrupt inputs are active (pre enable bits). Writing to the ISR allows software to generate interrupts until the HIE bit has been enabled.
- Interrupt Pending Register (IPR) is a read only register that provides an indication of interrupts that are active and enabled (post enable bits). The IPR is an optional register in the simple IntC and can be parameterized away to reduce FPGA resources required by an IntC.
- Interrupt Enable Register (IER) is a read/write register whose contents are used to enable selected interrupts.
- Interrupt Acknowledge Register (IAR) is not an actual register. It is a write-only location used to clear interrupt requests.

  Note The next two address locations are not registers, but provide helper functions that make setting and clearing IER bits easier.
- Set Interrupt Enables (SIE) is a write only location that provides the ability to set selected bits within the IER in one atomic operation, rather than requiring a read/modify/write sequence.
- Clear Interrupt Enables (CIE) is a write-only location that provides the ability to clear selected bits within the IER in a single atomic operation. Both SIE and CIE are optional in the Simple IntC and can be parameterized out of the design to reduce FPGA resource consumption by the IntC.
- Interrupt Vector Register (IVR) is a read-only register that contains the ordinal value of the highest priority interrupt that is active and enabled. The IVR is optional and can be parameterized out of the design to reduce IntC FPGA resources.
- Master Enable Register (MER) is a read/write, two-bit register used to enable or disable the IRQ output and to enable hardware interrupts (when hardware interrupts are enabled, software interrupts are disabled until the IntC is reset).

Bus Interface

The core interrupt controller functionality is designed with a simple bus interconnect interface. For a particular bus interface, all that is required is a top level (bus centric) wrapper that instantiates the IntC core and the desired bus interface module. There are currently two types of bus interfaces available, providing either an OPB IntC or a DCR IntC.

The On-chip Peripheral Bus (OPB) interface provides a slave interface on the OPB for transferring data between the OPB IntC and the processor. The OPB IntC registers are memory mapped into the OPB address space and data transfers occur using OPB byte enables. The register addresses are fixed on four byte boundaries and the registers and the data transfers to and from them are always as wide as the data bus.

The number of interrupt inputs is configurable up to the width of the data bus, which is also set by a configuration parameter. In either bus interface, the base address for the registers is set by a configuration parameter. Since the inputs and the output are configurable, several Simple IntC instances can be cascaded to provide any number of interrupt inputs, regardless of the data bus width. A block diagram of the IntC is shown in Figure 24.

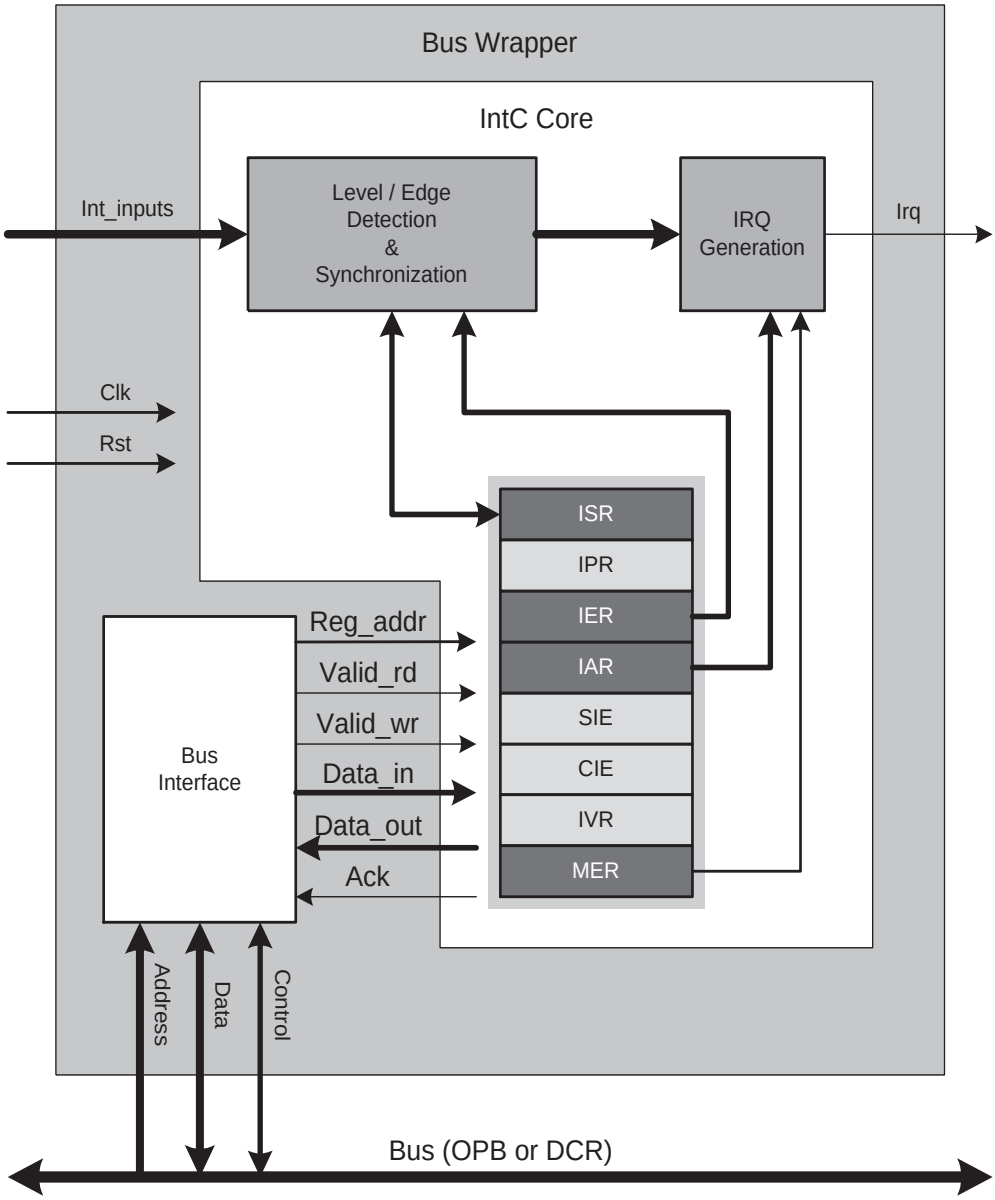


Figure 24: **Interrupt Controller Organization**

Programming Model

Register Data Types and Organization

All IntC registers are accessed through the OPB bus interface. The base address for these registers is provided by a configuration parameter. For an OPB IntC each register is accessed on a 4-byte boundary offset from the base address, regardless of the width of the registers, providing conformance to the OPB-IPIF register location convention. Since OPB addresses are byte addresses, OPB IntC register offsets are located at integral multiples of four from the base address. Table 13 illustrates the registers and their offsets from the base address for an OPB IntC. Normally, an OPB IntC is configured to be a 32-bit, 16-bit, or an 8-bit OPB peripheral that corresponds to the width of the processor data bus width. Figure 26 shows the address offsets and alignment for the OPB IntC for these three bus widths. The IntC registers are read as big-endian data. The bit and byte labeling for big-endian data types is shown in Figure 25.

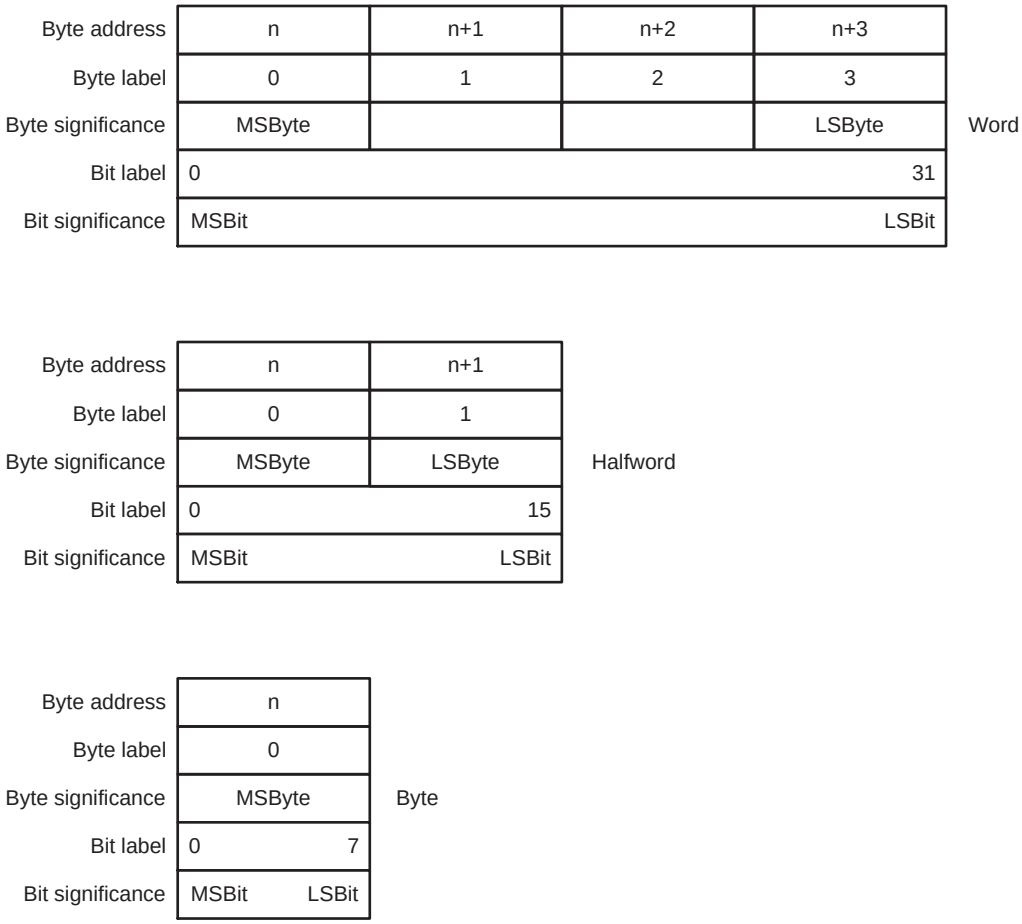


Figure 25: **Data Types**

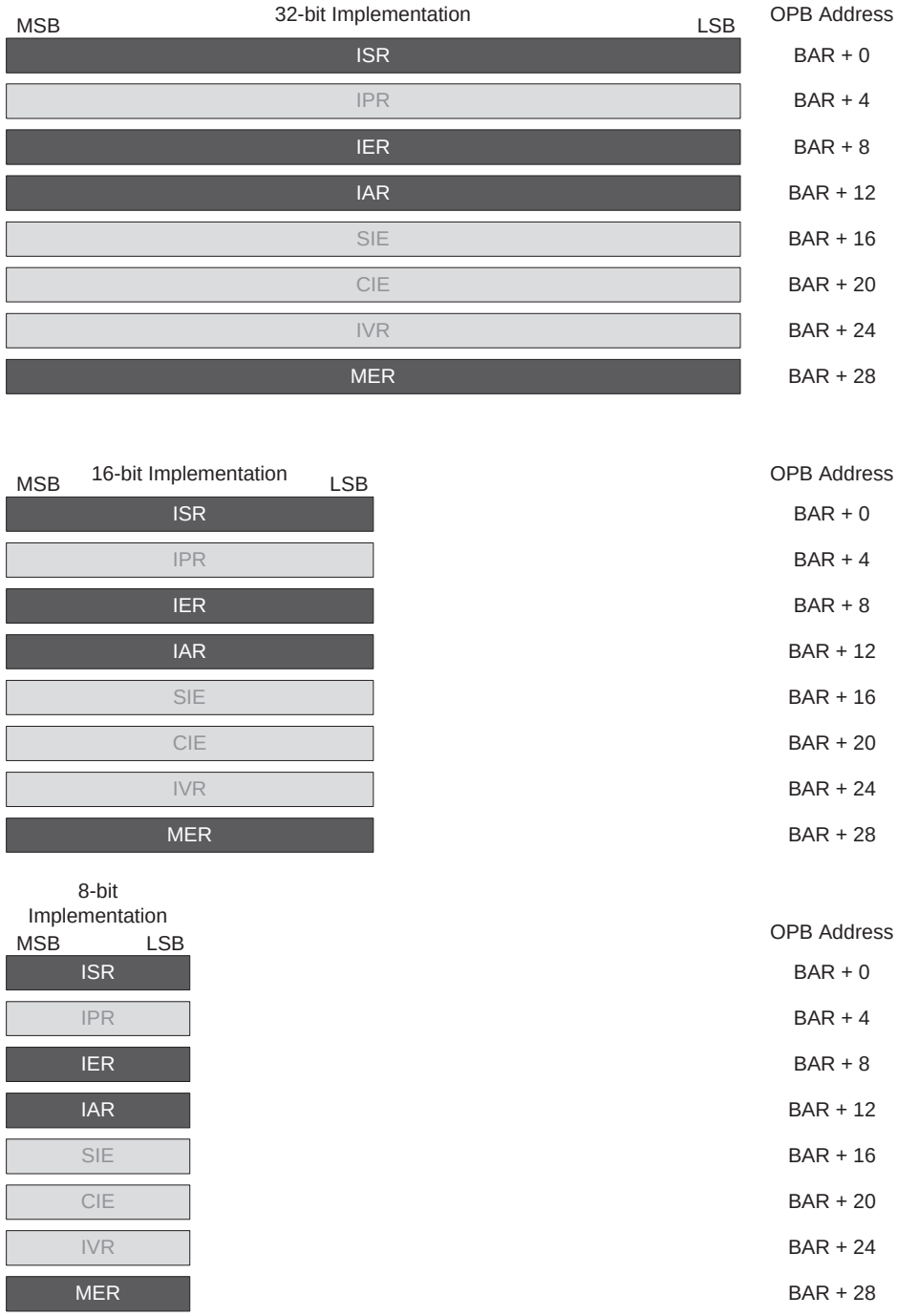


Figure 26: **OPB-based Register Offsets and Alignment**

IntC Registers

The eight registers visible to the programmer are shown in Table 13 and described in this section. In the diagrams and tables that follow, *w* refers to the width of the data bus (DB).

Note If the number of interrupt inputs is less than the data bus width, the inputs will start with INT0. INT0 maps to the LSB of the ISR, IPR, IER, IAR, SIE, and CIE, and additional inputs correspond sequentially to successive bits to the left.

Unless stated otherwise any register bits that are not mapped to inputs return zero when read and do nothing when written.

Table 13: **IntC Registers and Base Address Offsets**

Register Name	Abbreviation	OPB Offset
Interrupt Status Register	ISR	0 (00h)
Interrupt Pending Register	IPR	4 (04h)
Interrupt Enable Register	IER	8 (08h)
Interrupt Acknowledge Register	IAR	12 (0Ch)
Set Interrupt Enable Bits	SIE	16 (10h)
Clear Interrupt Enable Bits	CIE	20 (14h)
Interrupt Vector Register	IVR	24 (18h)
Master Enable Register	MER	28 (1Ch)

Interrupt Status Register (ISR)

When read, the contents of this register indicate the presence or absence of an active interrupt signal regardless of the state of the interrupt enable bits. Each bit in this register that is set to a 1 indicates an active interrupt signal on the corresponding interrupt input. Bits that are 0 are not active. The ISR register is writable by software until the Hardware Interrupt Enable (HIE) bit in the MER has been set. Once that bit has been set, software can no longer write to the ISR. Given these restrictions, when this register is written to, any data bits that are set to 1 will activate the corresponding interrupt, just as if a hardware input became active. Data bits that are zero have no effect. This allows software to generate interrupts for test purposes until the HIE bit has been set. Once HIE has been set (enabling the hardware interrupt inputs), then writing to this register does nothing. If there are fewer interrupt inputs than the width of the data bus, writing a 1 to a non-existing interrupt input does nothing and reading it will return zero. The ISR is shown in the following diagram and the bits are described in Table 14.

ISR — Interrupt Status Register

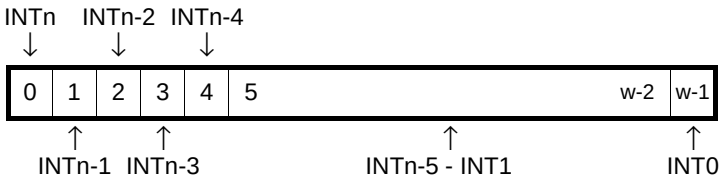


Table 14: **Interrupt Status Register**

Bits	Name	Description	Reset Value
0 to (w –1)	INTn – INT0 $(n \leq w - 1)$ where w is DB width	**Interrupt Input n – Interrupt Input 0** 0 Read – Not active; Write – No action 1 Read – Active; Write – SW interrupt	0

Interrupt Pending Register (IPR)

This is an optional register in the simple IntC and can be parameterized out of an implementation. Reading the contents of this register indicates the presence or absence of an active interrupt signal that is also enabled. Each bit in this register is the logical AND of the bits in the ISR and the IER. If there are fewer interrupt inputs than the width of the data bus, reading a non-existing interrupt input will return zero. The IPR is shown in the following diagram and the bits are described in Table 15.

IPR — Interrupt Pending Register

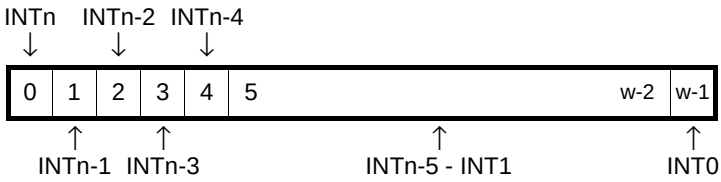


Table 15: **Interrupt Pending Register**

Bits	Name	Description	Reset Value
0 to (w – 1)	INTn – INT0 $(n \leq w - 1)$ where w is DB width	**Interrupt Input n – Interrupt Input 0** 0 – Not active 1 – Active	0

Interrupt Enable Register (IER)

This is a read/write register. Writing a 1 to a bit in this register enables the corresponding interrupt input signal. Writing a 0 to a bit disables the corresponding interrupt input signal. Reading this register indicates which interrupt inputs are enabled, where a one indicates the input is enabled and a zero indicates the input is disabled. If there are fewer interrupt inputs than the width of the data bus, writing a 1 to a non-existing interrupt input does nothing and reading it will return zero. The IER is shown in the following diagram and the bits are described in Table 16.

IER — Interrupt Enable Register

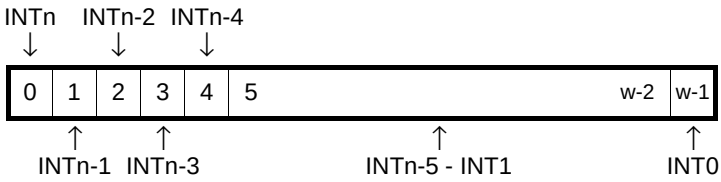


Table 16: **Interrupt Enable Register**

Bits	Name	Description	Reset Value
0 to (w – 1)	INTn – INT0 $(n \leq w - 1)$ where w is DB width	**Interrupt Input n – Interrupt Input 0** 1 – Interrupt enabled 0 – Interrupt disabled	0

Interrupt Acknowledge Register (IAR)

The IAR is a write only location that clears the interrupt request associated with selected interrupt inputs. Writing a one to a bit location in the IAR will clear the interrupt request that was generated by the corresponding interrupt input. Writing zeros does nothing as does writing a one to a bit that does not correspond to an active input or for which an interrupt input does not exist. The IAR is shown in the following diagram and the bits are described in Table 17.

IAR — Interrupt Acknowledge Register

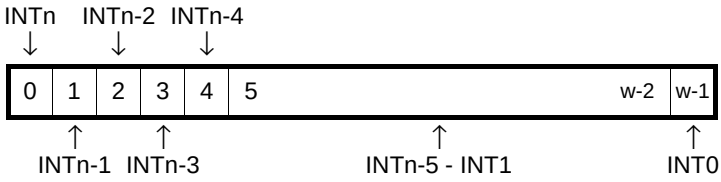


Table 17: **Interrupt Acknowledge Register**

Bits	Name	Description	Reset Value
0 to (w – 1)	INTn – INT0 $(n \leq w - 1)$ where w is DB width	**Interrupt Input n – Interrupt Input 0** 1 Clear Interrupt 0 no action	n/a

Set Interrupt Enables (SIE)

SIE is a location used to set IER bits in a single atomic operation, rather than using a read/modify/write sequence. Writing a one to a bit location in SIE will set the corresponding bit in the IER. Writing zeros does nothing, as does writing a one to a bit location that corresponds to a non-existing interrupt input. The SIE is optional in the simple IntC and can be parameterized out of the implementation. The SIE is shown in the following diagram and the bits are described in Table 18.

SIE — Set Interrupt Enables

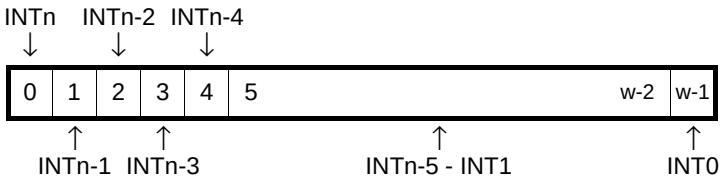


Table 18: **Set Interrupt Enables**

Bits	Name	Description	Reset Value
0 to (w – 1)	INTn – INT0 $(n \leq w - 1)$ where w is DB width	**Interrupt Input n – Interrupt Input 0** 1 Set IER bit 0 no action	n/a

Clear Interrupt Enables (CIE)

CIE is a location used to clear IER bits in a single atomic operation, rather than using a read/modify/write sequence. Writing a one to a bit location in CIE will clear the corresponding bit in the IER. Writing zeros does nothing, as does writing a one to a bit location that corresponds to a non-existing interrupt input. The CIE is also optional in the simple IntC and can be parameterized out of the implementation. The CIE is shown in the following diagram and the bits are described in Table 19.

CIE — Clear Interrupt Enables

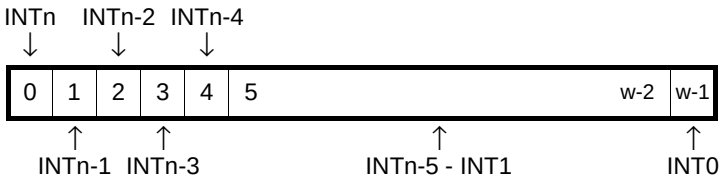


Table 19: **Clear Interrupt Enables**

Bits	Name	Description	Reset Value
0 to (w – 1)	INTn – INT0 $(n \leq w - 1)$ where w is DB width	**Interrupt Input n – Interrupt Input 0** 1 Clear IER bit 0 no action	n/a

Interrupt Vector Register (IVR)

The IVR is a read-only register and contains the ordinal value of the highest priority, enabled, active interrupt input. **INT0 (always the LSB) is the highest priority interrupt input** and each successive input to the left has a correspondingly lower interrupt priority. If no interrupt inputs are active then the IVR will contain all ones. The IVR is optional in the simple IntC and can be parameterized out of the implementation. The IVR is shown in the following diagram and described in Table 20.

IVR — Interrupt Vector Register

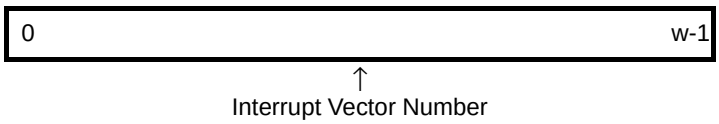


Table 20: **Interrupt Vector Register**

Bits	Name	Description	Reset Value
0 to (w – 1)	Interrupt Vector Number	Ordinal of highest priority, enabled, active interrupt input	all ones

Master Enable Register (MER)

This is a two bit, read/write register. The two bits are mapped to the two least significant bits of the location. The least significant bit contains the Master Enable (ME) bit and the next bit contains the Hardware Interrupt Enable (HIE) bit. Writing a 1 to the ME bit enables the IRQ output signal. Writing a 0 to the ME bit disables the IRQ output, effectively masking all interrupt inputs.

The HIE bit is a write once bit. At reset this bit is reset to zero, allowing software to write to the ISR to generate interrupts for testing purposes, and disabling any hardware interrupt inputs. Writing a one to this bit enables the hardware interrupt inputs and disables software generated inputs. Writing a one also disables any further changes to this bit until the device has been reset.

Writing ones or zeros to any other bit location does nothing. When read, this register will reflect the state of the ME and HIE bits. All other bits will read as zeros. The MER is shown in the following diagram and is described in Table 21.

MER — Master Enable Register

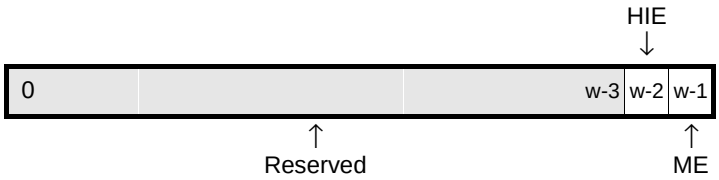


Table 21: **Master Enable Register**

Bits	Name	Description	Reset Value
0 to (w – 3)	Unused	**Not used**	0
(w – 2)	HIE	**Hardware Interrupt Enable** 0 Read – SW interrupts enabled Write – no effect 1 Read – HW interrupts enabled Write – Enable HW interrupts	0
(w – 1)	ME	**Master IRQ Enable** 0 IRQ disabled – all interrupts disabled 1 IRQ enabled – all interrupts enabled	0

Programming the IntC

This section provides an overview of software initialization and communication with an IntC.

Terminology

The number of interrupt inputs that an IntC has is set by the C_NUM_INTR_INPUTS generic described in Table 24. The first input is always Int0 and is mapped to the LSB of the registers (except IVR and MER). A valid interrupt input signal is any signal that provides the correct polarity and type of interrupt input. Examples of valid interrupt inputs are rising edges, falling edges, high levels, and low levels (hardware interrupts), or software interrupts if HIE has not been set. Each interrupt input can be selectively enabled or disabled (masked). The polarity and type of each hardware interrupt input is specified in the IntC generics C_KIND_OF_INTR, C_KIND_OF_EDGE, and C_KIND_OF_LVL (see Table 24). Software interrupts do not have any polarity or type associated with them, so, until HIE has been set, they are always valid. Any valid interrupt input signal that is applied to an enabled interrupt input will generate a corresponding interrupt request within the IntC. All interrupt requests are combined (an OR function) to form a single interrupt request output that can be enabled or disabled (masked).

Initialization and Communication

During power-up or reset, an IntC is put into a state where all interrupt inputs and the interrupt request output are disabled. In order for the IntC to accept interrupts and request service, the following steps are required:

1. Each bit in the IER corresponding to an interrupt input must be set to a one. This allows the IntC to begin accepting interrupt input signals. Int0 has the highest priority, and it corresponds to the least significant bit (LSB) in the IER.
2. The MER must be programmed based on the intended use of the IntC. There are two bits in the MER: the Hardware Interrupt Enable (HIE) and the Master IRQ Enable (ME). The ME bit must be set to enable the interrupt request output.
3. If software testing is to be performed, the HIE bit must remain at its reset value of zero. Software testing can now proceed by writing a one to any bit position in the ISR that corresponds to an existing interrupt input. A corresponding interrupt request is generated if that interrupt is enabled, and interrupt handling proceeds normally.
4. Once software testing has been completed, or if software testing is not performed, a one is written to the HIE bit, which enables the hardware interrupt inputs and disables any further software generated interrupts.
5. After a one has been written to the HIE bit, any further writes to this bit have no effect. This feature prevents stray pointers from accidentally generating unwanted interrupt requests, while still allowing self-test software to perform system tests at power-up or after a reset.

Reading the ISR indicates which inputs are active. If present, the IPR indicates which enabled inputs are active. Reading the optional IVR provides the ordinal value of the highest priority interrupt that is enabled and active. For example, if the IVR is present, and a valid interrupt signal has occurred on the Int3 interrupt input and nothing is active on Int2, Int1, and Int0, reading the IVR will provide a value of three. If Int0 becomes active then reading the IVR provides a value of zero. If no interrupts are active or it is not present, reading the IVR returns all ones.

Acknowledging an interrupt is achieved by writing a one to the corresponding bit location in the IAR. An interrupt acknowledge clears the corresponding interrupt request. However, if a valid interrupt signal remains on that input (another edge occurs or an active level still exists on the corresponding interrupt input), a new interrupt request output is generated. Also, all interrupt requests are combined to form the Irq output so any remaining interrupt requests that have not been acknowledged will cause a new interrupt request output to be generated.

The software can disable the interrupt request output at any time by writing a zero to the ME bit in the MER. This effectively masks all interrupts for that IntC. Alternatively, interrupt inputs can be selectively masked by writing a zero to each bit location in the IER that corresponds to an

input that is to be masked. If present, SIE and CIE provide a convenient way to enable or disable (mask) an interrupt input without having to read, mask off, and then write back the IER. Writing a one to any bit location(s) in the SIE sets the corresponding bit(s) in the IER without affecting any other IER bits. Writing a one to any bit location(s) in the CIE clears the corresponding bit(s) in the IER without affecting any other IER bits.

Implementation

The IntC is implemented to minimize area. Consequently, all configurable elements within the design are based on generics (parameters) and any unused or unselected capabilities are not implemented (see the **Parameterization** section).

I/O Summary

The following tables provide information on I/O signals. I/Os that are common for all IntC types are shown in Table 22. Table 23 shows I/Os that are specific to an OPB IntC.

Table 22: **Core IntC I/O Summary**

Port Name	Direction	Description	Type	Range
Intr	in	Interrupt intputs	Std_Logic_Vector	C_NUM_INTR_INPUTS – 1 downto 0
Irq	out	IntC interrupt request output	Std_Logic	n/a

Table 23: **OPB IntC I/O Summary**

Port Name	Direction	Description	Type	Range
OPB_Clk	in	OPB clock	Std_Logic	n/a
OPB_Rst	in	OPB reset, active high	Std_Logic	n/a
OPB_select	in	OPB select	Std_Logic	n/a
OPB_ABus	in	OPB address bus	Std_Logic_Vector	0 to C_OPB_AWIDTH – 1
OPB_RNW	in	OPB read not write enable (read high, write low)	Std_Logic	n/a
OPB_BE	in	OPB byte enables	Std_Logic_Vector	0 to (C_OPB_DWIDTH / 8) – 1
OPB_DBus	in	OPB data bus (OPB to IntC)	Std_Logic_Vector	0 to C_OPB_DWIDTH – 1
IntC_DBus	out	IntC data bus (IntC to OPB)	Std_Logic_Vector	0 to C_OPB_DWIDTH – 1
IntC_xferAck	out	IntC transfer acknowledge	Std_Logic	n/a
IntC_ErrAck	out	IntC error acknowledge	Std_Logic	n/a
OPB_seqAddr	in	OPB sequential address enable	Std_Logic	n/a
IntC_toutSup	out	IntC timeout suppress	Std_Logic	n/a
IntC_retry	out	IntC retry request	Std_Logic	n/a

Parameterization

The following characteristics of the IntC are parameterizable:

- Base address for the Simple IntC registers and the upper address of the memory space occupied by the IntC (C_BASEADDR, C_HIGHADDR).
- Edge or level sensitivity on interrupt inputs as well as the polarity (C_KIND_OF_INTR, C_KIND_OF_EDGE, C_KIND_OF_LVL).
- Edge (pulse) or level IRQ generation, and the polarity of the IRQ output (C_IRQ_IS_LEVEL, C_IRQ_ACTIVE).

- Address bus width (C_OPB_AWIDTH).
- Bus interface: Normally 8-bit, 16-bit or 32-bit data widths for the OPB IntC (C_OPB_DWIDTH).
- The number of interrupt inputs is parameterizable up to the width of the data bus (C_NUM_INTR_INPUTS).
- The presence of the IPR (C_HAS_IPR).
- The presence of the SIE and CIE (C_HAS_SIE, C_HAS_CIE).
- The presence of the IVR (C_HAS_IVR).

Table 24 lists the top level generics (parameters) that are common to all variations of an IntC.

Table 24: **Generics (Parameters) Common to all IntC Instantiations**

Generic Name	Description	Type	Valid Values
C_FAMILY	Target FPGA family type (not currently used).	String	"spartan2", "spartan2e", "virtex", "virtexe", "virtex2", "virtex2pro"
C_Y	Row placement directive (not currently used).	Integer	Any valid row value for the selected target family.
C_X	Column placement directive (not currently used).	Integer	Any valid column value for the selected target family.
C_U_SET	User set for grouping (not currently used).	String	"intc"
C_BASEADDR	Base address for accessing the IntC registers.	Std_Logic_Vector	Any valid 32-byte boundary address for the IntC instance. [1]
C_HIGHADDR	Upper address value of the memory map entry for the IntC. Used in conjunction with C_BASEADDR to determine the number of upper address bits to use for address decoding.	Std_Logic_Vector	Any valid address for the IntC instance that is at least 32 bytes (8 words) greater than C_BASEADDR. [2]
C_NUM_INTR_INPUTS	Number of interrupt inputs.	Integer	1 up to the width of the data bus.
C_KIND_OF_INTR	Type of interrupt for each input X = none 1 = edge 0 = level.	Std_Logic_Vector	A little-endian vector the same width as the data bus containing a 0 or 1 in each position corresponding to an interrupt input.
C_KIND_OF_EDGE	Type of each edge sensitive input X = n/a 1 = rising 0 = falling.	Std_Logic_Vector	A little-endian vector the same width as the data bus containing a 0 or 1 in each position corresponding to an interrupt input.
C_KIND_OF_LVL	Type of each level sensitive input X = n/a 1 = high 0 = low.	Std_Logic_Vector	A little-endian vector the same width as the data bus containing a 0 or 1 in each position corresponding to an interrupt input.

Table 24: **Generics (Parameters) Common to all IntC Instantiations**

Generic Name	Description	Type	Valid Values
C_HAS_IPR	Indicates the presence of IPR.	Integer	0 = not present 1 = present
C_HAS_SIE	Indicates the presence of SIE.	Integer	0 = not present 1 = present
C_HAS_CIE	Indicates the presence of CIE.	Integer	0 = not present 1 = present
C_HAS_IVR	Indicates the presence of IVR.	Integer	0 = not present 1 = present
C_IRQ_IS_LEVEL	Indicates whether the Irq output uses level (or edge) generation.	Integer	0 = edge generation 1 = level generation
C_IRQ_ACTIVE	Indicates the sense of the Irq output	Std_Logic	'0' = falling / low '1' = rising / high

Notes:

1. C_BASEADDR must begin on a 32-byte address boundary for OPB and an 8-word boundary for DCR. For an OPB IntC this means the low 5 address bits must be zero. For a DCR IntC this means the low three address bits must be zero.
2. C_HIGHADDR is required to be at least C_BASEADDR + 31 for an OPB IntC or C_BASEADDR + 7 for a DCR IntC in order to provide space for the eight 32-bit addresses used by the simple IntC registers. However, a bigger memory map space allocated to the simple IntC will reduce the FPGA resources required for decoding the address. For example:
   C_BASEADDR = 0x70800000
   C_HIGHADDR = 0x7080001F
   provides the maximum address decode resolution for an OPB IntC, requiring the upper 27 address bits to be decoded. This choice will increase the number of FPGA resources required for implementation and may adversely affect the maximum operating frequency of the system. Conversely,
   C_BASEADDR = 0x70800000
   C_HIGHADDR = 0x7FFFFFFF
   will significantly reduce the address decoding logic for an OPB IntC (only the 4 upper address bits), resulting in a smaller and faster implementation. A similar situation exists for the DCR IntC with the exception that the addresses are only ten bits wide, so the maximum address decode resolution for a DCR IntC requires seven upper address bits to be decoded.

Table 25 lists the top level generics (parameters) that are present in an OPB IntC.

Table 25: **Generics (Parameters) for an OPB IntC**

Generic Name	Description	Type	Valid Values
C_OPB_AWIDTH	Width of the OPB address bus.	Integer	32
C_OPB_DWIDTH	Width of the OPB data buses.	Integer	8, 16, or 32

OPB External Memory Controller (EMC)

March 2002

Summary

This document provides the design specification for the External Memory Controller (EMC) Intellectual Property (IP) solution. This document applies to the following peripherals:

opb_memcon	v1.00a

Introduction

This specification defines the architecture and interface requirements for the EMC. This module supports data transfers between the On-chip Peripheral Bus (OPB) and external memory devices such as SRAM and Flash devices. Example devices for use with this controller are the Integrated Device Technology, Inc. IDT71V416S SRAM and the Intel 28F128J3A StrataFlash Memory Devices. The EMC module is organized to be an OPB slave-only device, which differs from the IBM EBC specification.

The Xilinx EMC design allows you to tailor the EMC to suit your application by setting certain parameters to enable or disable features. Quick links are provided to the following sections:

- **EMC Parameters**
- **EMC I/O Signals**
- **EMC Address Map and Register Descriptions**
- **Connecting to Memory (SRAM and StrataFlash)**

EMC Overview

Features

The EMC is a soft IP core designed for Xilinx FPGAs and has the following features:

- Parameterized for up to a total of eight memory (SRAM / Flash) banks
  - Separate base addresses and address range for each bank of memory
- Separate Control Register for each bank of memory to control memory mode
- OPB V2.0 bus interface with byte-enable support
- Supports 128-bit, 64-bit, 32-bit, 16-bit, and 8-bit bus interfaces
- Supports memory width of 128-bits, 64 bits, 32-bits, 16 bits, or 8 bits
- Memory width is independent of OPB bus width (memory width must be less than or equal to OPB bus width)
- Configurable wait states for read, write, read in page, read recovery before write, and write recovery before read
- Optional faster access for in-page read accesses (page size 8 bytes)
- System clock frequency of up to 133 MHz

EMC Background

The EMC implements the hardware and software functionality of the IBM External Bus Controller operating as a slave on the OPB. The EMC module receives instructions from the OPB to read and write to external SRAMs and Flash. The implementation does not include the features of the External Master Interface as described in the IBM External Bus Controller Functional Specification (v. C12E0623_EBC). Additional features that are not implemented are as follows:

- Attachment of an external master device to gain access to all OPB slave devices
- Device Control Register (DCR) interface
- Sharing of the I/O pins on the module with the SDRAM controller IBM specification number (v. C12E0622_HSPLB_MC).

The OPB Memory Controller provides an interface between the OPB and one to eight external banks of memory components, such as the Intel® StrataFlash™ or the IDT71V416S SRAM memory. The controller supports OPB data bus widths of 8 to 128 bits, and memory subsystem widths of 8 to 128 bits. You can configure the controller to support page-mode reads that can be up to six times faster than non-page reads. The in-page detection logic is automatically configured out of the controller if page mode is not required.

The Flash memory controller is organized much like an SRAM interface. This controller assumes that the Flash programming circuitry is built into the Flash components and that the command interface to the Flash is handled in software. The controller provides basic read/write control signals and the ability to configure the access times for read, read-in-page, write, and recovery times when switching from read to write or write to read.

This controller supports the OPB V2.0 byte enable architecture. Any access size up to the width of the OPB data bus is permitted. Limitations may apply when the memory components limit the allowed transfer types, such as memory devices configured in 16-bit or 32-bit mode with no byte enable support.

EMC Parameters

Certain features can be parameterized in the EMC design to allow you to obtain an EMC that is uniquely tailored to your system. This allows you to configure a design that only utilizes the resources required by your system, and operates with the best possible performance. The features that can be parameterized in the Xilinx EMC design are shown in Table 26.

Table 26: **EMC Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
Number of Memory Banks	C_NUM_BANKS_MEM	1 - 8	2	integer
OPB Clock Period	C_OPB_CLOCK_PERIOD_PS	Integer number of picoseconds	40000	integer
Control Register Bank Base Address	C_BASEADDR	Valid Address Range(3)	None(4)	std_logic_vector
Control Register Bank High Address	C_HIGHADDR	Address range must be a power of 2 and > 0x01F(3)	None(4)	std_logic_vector
Flash/SRAM Base Address x = 0 to 7	C_MEMx(1,2)_BASEADDR	Valid Address Range(3)	None(4)	std_logic_vector
Flash/SRAM High Address x = 0 to 7	C_MEMx(1,2)_HIGHADDR	Address range must be a power of 2 and ≤ OPB Address Space(3)	None(4)	std_logic_vector
OPB Data Bus Width	C_OPB_DWIDTH	32,64	64	integer

Table 26: **EMC Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
OPB Address Bus Width	C_OPB_AWIDTH	8 - 32	32	integer
Width of Data Bus to Memory Devices	C_MEM_WIDTH	≤ C_OPB_DWIDTH	64	integer
Address access time for reads when EMCCRx[0] = 0 or EMCCRx[1] = 1 and access is out of page(6)	C_RD_ADDR_TO_OUT_SLOW_PS_x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer
Address valid to end of write when EMCCRx[0] = 0(5,6)	C_WR_ADDR_TO_OUT_SLOW_PS_x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer
Minimum time that write enable goes low (follows address being driven out). Applies to all writes.(5,6)	C_WR_MIN_PULSE_WIDTH_PS_ x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer
Address access time for reads when EMCCRx[0] = 1 or EMCCRx[1] = 1 and access is in page(6)	C_RD_ADDR_TO_OUT_FAST_PS_x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer
Address valid to end of write when EMCCRx[0] = 1(5,6)	C_WR_ADDR_TO_OUT_FAST_PS_x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer
Delay inserted before Write Enable goes low if previous access was Read. (6)	C_RD_RECOVERY_BEFORE_WR_PS_x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer
Delay inserted before Output Enable goes low if previous access was Write. (6)	C_WR_RECOVERY_BEFORE_RD_PS_x(1)	Integer number of picoseconds	Refer to Data Sheet of particular Memory Device	integer

Notes:

3. x = values for memory banks 0 to 7
4. This design can accommodate up to 8 Banks of Flash and/or SRAM. The address range generics are designated as C_MEM0_BASEADDR, C_MEM1_BASEADDR, C_MEM0_HIGHADDR, C_MEM1_HIGHADDR, etc.
5. Address range specified by C_BASEADDR and C_HIGHADDR must be a power of 2 and >=0x01F. C_MEMx(4)_BASEADDR and C_MEMx_HIGHADDR must be a power of 2 and less than or equal to the OPB address space.
6. No default value is specified for C_BASEADDR and C_HIGHADDR AND C_MEMx_BASEADDR, C_MEMx_HIGHADDR to insure that the actual value is set; if the value is not set, a compiler error is generated. These generics must be a power of 2 and encompass the memory size for C_MEMx_BASEADDR, C_MEMx_HIGHADDR.
7. Write enable low time is the maximum of C_WR_ADDR_TO_OUT_FAST/SLOW_PS and C_WR_MIN_PULSE_WIDTH.
8. As specified by the memory device data sheet.

EMC I/O Signals

The I/O signals for the EMC are listed in Table 27.

Table 27: **EMC I/O Signals**

Signal Name	Interface	I/O	Description
OPB_Clk	OPB	I	OPB Clock
OPB_Rst	OPB	I	OPB Reset
OPB_ABus(0:C_OPB_AWIDTH-1)	OPB	I	OPB Address Bus
OPB_BE(0:C_OPB_DWIDTH/8-1)	OPB	I	OPB Byte Enables
OPB_DBus(0:C_OPB_DWIDTH-1)	OPB	I	OPB Data Bus
OPB_RNW	OPB	I	OPB Read, Not Write
OPB_select	OPB	I	OPB Select
OPB_seqAddr	OPB	I	OPB Sequential Address
MemCon_DBus(0:C_OPB_DWIDTH-1)	OPB	O	Memory Controller Data Bus
MemCon_errAck	OPB	O	Memory Controller Error Acknowledge
MemCon_retry	OPB	O	Memory Controller Retry
MemCon_toutSup	OPB	O	Memory Controller Timeout Suppress
MemCon_xferAck	OPB	O	Memory Controller Transfer Acknowledge
Mem_STS(0:C_NUM_BANKS_MEM-1	IP Core	I	Memory Status Signal
Mem_DQ_I(0:C_MEM_WIDTH-1)	IP Core	I	Memory Input Data Bus
Mem_DQ_O(0:C_MEM_WIDTH-1)	IP Core	O	Memory Output Data Bus
Mem_DQ_T	IP Core	O	Memory Output 3-state Signal
Mem_A(0:C_OPB_AWDITH-1)	IP Core	O	Memory Address Bus
Mem_RPN	IP Core	O	Memory Reset/Power Down
Mem_CEN(0:C_NUM_BANKS_MEM-1)	IP Core	O	Memory Chip Enables
Mem_OEN	IP Core	O	Memory Output Enable
Mem_WEN	IP Core	O	Memory Write Enable
Mem_QWEN(0:(C_MEM_WIDTH/8) -1)	IP Core	O	Memory Qualified Write Enables
Mem_BEN(0:(C_MEM_WIDTH/8) -1)	IP Core	O	Memory Byte Enables

OPB Timing

This section describes the basic read and write timing for the OPB. For detailed descriptions, refer to the IBM OPB Specification(v2.0). An OPB cycle is initiated with a master request. The highest priority request is granted the bus, and in the next cycle the master asserts the bus select signal and begins the transfer. The transfer is completed with the return of transfer acknowledge, retry, or error acknowledge.

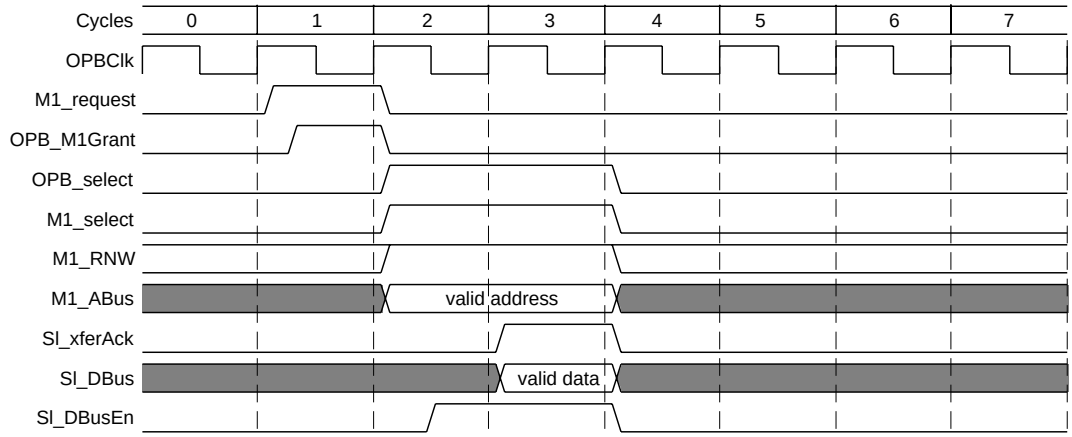


Figure 27: **Basic OPB Data Transfer**

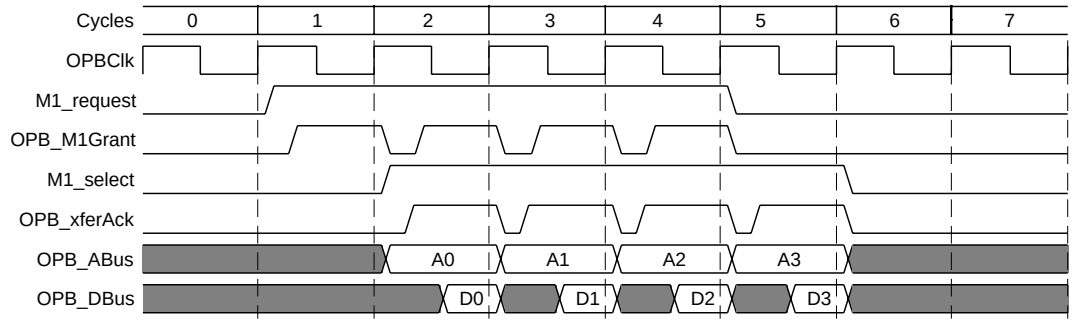


Figure 28: **OPB Data Transfer with Continuous Master Request**

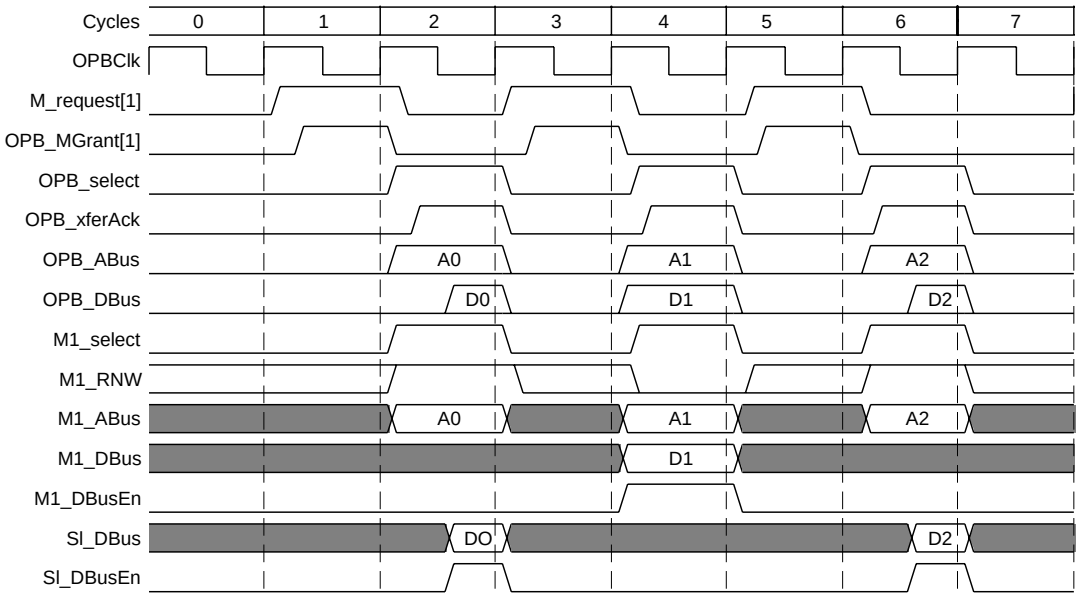


Figure 29: **OPB Full-Word Read/Write**

EMC Address Map and Register Descriptions

The EMC supports up to 8 banks of SRAM and/or Flash memory. Each memory bank has an independent base address and address range. The address range of a bank of memory is restricted to be a power of 2. If the desired address range is represented by 2^n, then the *n* least significant bits of the base address must be 0. For example, a memory bank with an addressable range of 16M (2^{24}) bytes could have a base address of 0xFF000000 and a high address of 0xFFFFFFFF. A memory bank with an addressable range of 64K (2^{16}) bytes could have a base address of 0xABCD0000 and a high address of 0xABCDFFFF. The addresses for each bank of memory are shown in Table 28.

Table 28: **EMC Memory Banks**

Memory	Base Address	High Address	Access
Bank 0	C_MEM0_BASEADDR	C_MEM0_HIGHADDR	R/W
Bank 1	C_MEM1_BASEADDR	C_MEM1_HIGHADDR	R/W
Bank 2	C_MEM2_BASEADDR	C_MEM2_HIGHADDR	R/W
Bank 3	C_MEM3_BASEADDR	C_MEM3_HIGHADDR	R/W
Bank 4	C_MEM4_BASEADDR	C_MEM4_HIGHADDR	R/W
Bank 5	C_MEM5_BASEADDR	C_MEM5_HIGHADDR	R/W
Bank 6	C_MEM6_BASEADDR	C_MEM6_HIGHADDR	R/W
Bank 7	C_MEM7_BASEADDR	C_MEM7_HIGHADDR	R/W

The EMC contains addressable control registers for write operations as shown in Table 29. The base address for these registers is set in the parameter C_BASEADDR which represents the address of the Memory Bank Control Register.

Table 29 shows all of the EMC control registers and addresses when all memory banks are used. Only control registers for memory banks that are used are present in the design.

Table 29: **EMC Control Registers**

Register Name	OPB Address	Access
MEM0 Control Register (EMCCR0)	C_BASEADDR + 0x00	Write
MEM1 Control Register (EMCCR1)	C_BASEADDR + 0x04	Write
MEM2 Control Register (EMCCR2)	C_BASEADDR + 0x08	Write
MEM3 Control Register (EMCCR3)	C_BASEADDR + 0x0C	Write
MEM4 Control Register (EMCCR4)	C_BASEADDR + 0x10	Write
MEM5 Control Register (EMCCR5)	C_BASEADDR + 0x14	Write
MEM6 Control Register (EMCCR6)	C_BASEADDR + 0x18	Write
MEM7 Control Register (EMCCR7)	C_BASEADDR + 0x1C	Write

Notes:
1. This design can accommodate up to 8 Banks of Flash and/or SRAM, and are therefore designated as C_MEM0_BASEADDR, C_MEM1_BASEADDR, etc.
2. Each bank of memory's control register is the same as described below.

Note The register definitions and address locations of the Xilinx EMC deviate from the IBM EBC specification. This deviation is necessary as a slave-only implementation for the EBC and because there is no DCR interface.

EMC Control Register (EMCCR)

The EMC Control Register is shown below. The first row of the table is the bit location, the second row contains the reset value for that bit.

Table 30 shows the Control Register bit definitions. The Control Register definition deviates from the IBM EBC specification because the DCR is not supported in the design.

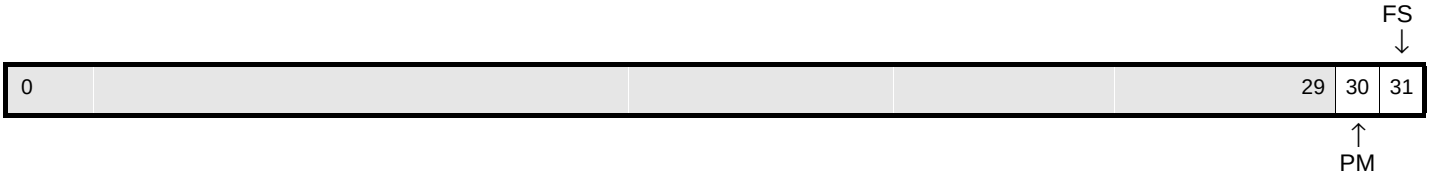


Table 30: **EMC Control Register Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0 - 29	Reserved			
30	PM	Read/Write	'0'(1)	**Page Mode Enable.** Determines whether or not in-page detection logic is created with a corresponding decrease in read access time for in-page reads. • '0' - Page Mode is Disabled • '1' - Page Mode is Enabled
31	FS	Read/Write	'0'(1)	**Fast/Slow Mode Enable.** Determines the number of Wait States required based on the input timing parameters. • '0' - Slow Access Time • '1' - Fast Access Time

Table 31: **EMC Control Register Bit Functionality**

Read/Write	PM Enable	FS Enable	Function
Read	0	0	Slow access
	0	1	Fast access
	1	X	Fast in page, slow not in page
Write	X	0	Slow access
	X	1	Fast access

EMC Block Diagram

Memory Data Types and Organization

Depending on the size of the bus attached to the processor, memory can be accessed through the EMC as follows:

- byte (8 bits)
- halfword (2 bytes)
- word (4 bytes)
- doubleword (8 bytes)
- 128-bit (16 bytes)

For the OPB, data is organized as big-endian. The bit and byte labeling for the big-endian data types is shown in Figure 30.

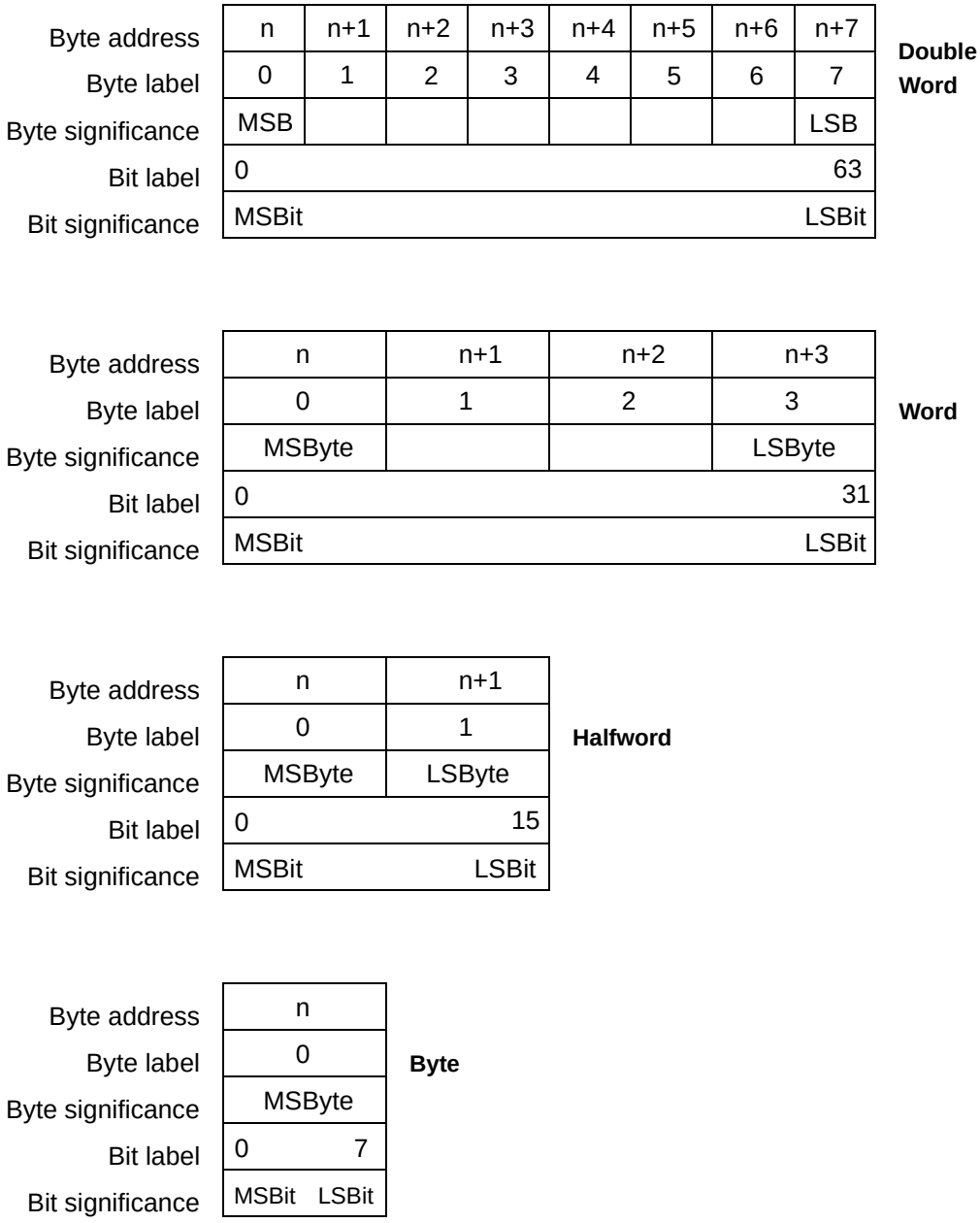


Figure 30: **Big-Endian Data Types**

Figure 31 depicts the Memory Control Block Diagram implemented in the EMC.

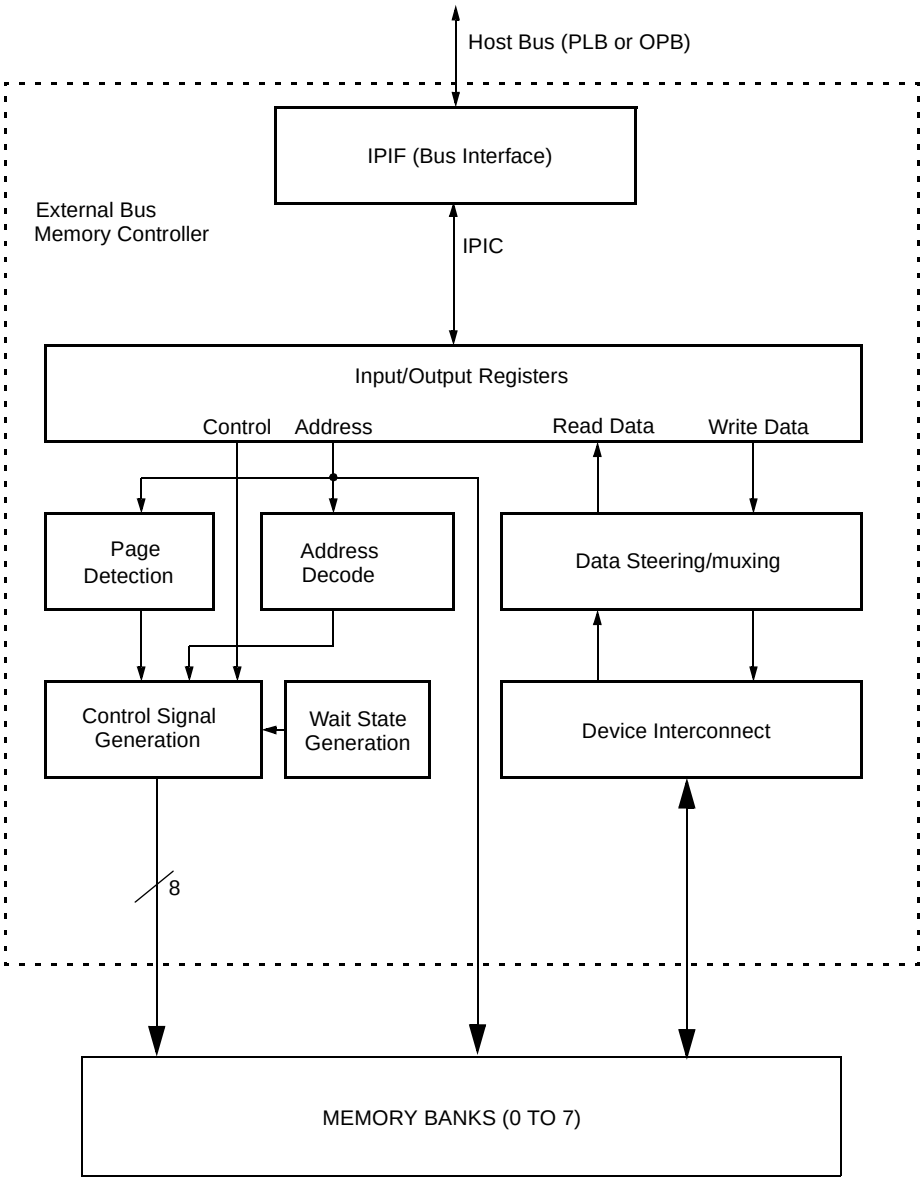


Figure 31: **EMC Memory Control Block Diagram**

Figure 32 depicts the Memory Control State Diagram implemented in the EMC.

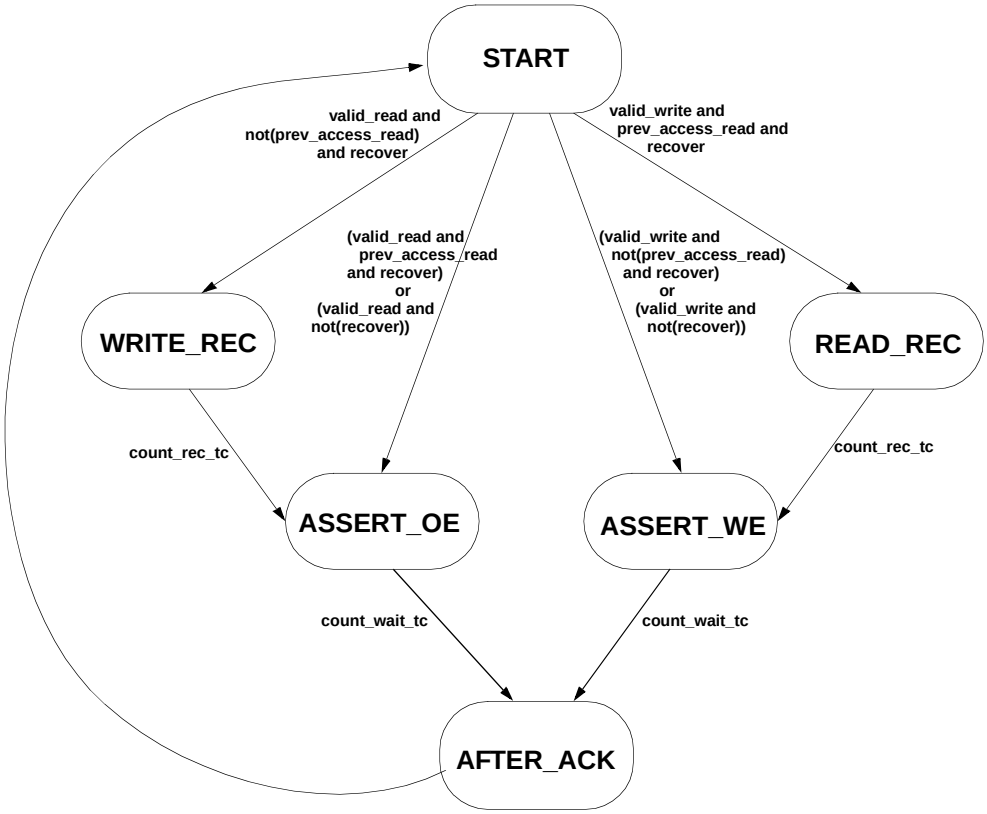


Figure 32: **EMC Memory Control State Diagram**

Memory Controller Operation

Basic Timing for Memory

The Memory Controller is designed to connect to a variety of memory subsystem configurations. For detailed descriptions on the timing and protocol of the IDT 71V416S SRAM and the Intel 28F128J3 StrataFlash, refer to the appropriate data sheet. However, basic read and write timing diagrams are listed below. Figure 33 and Figure 34 illustrate the basic read and write functions for the SRAM. Table 32 defines the symbols used in the figures for the SRAM.

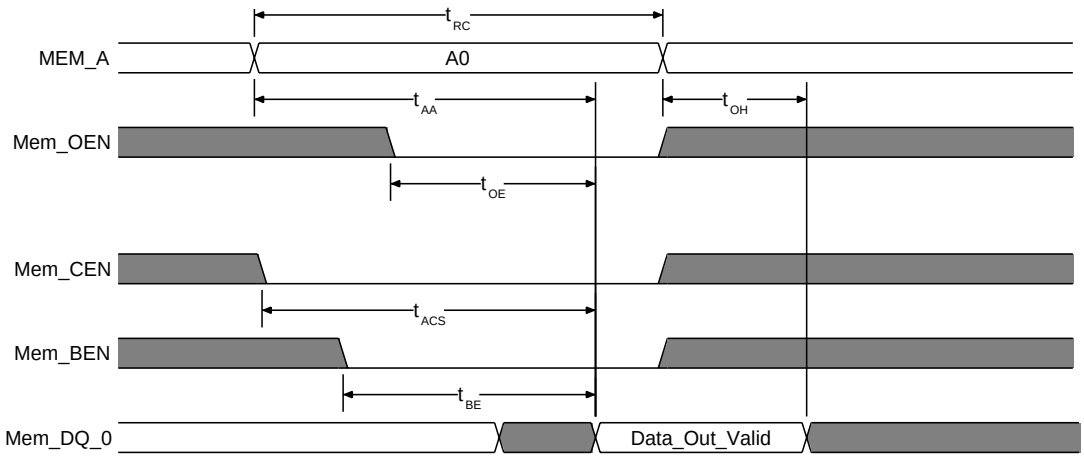


Figure 33: **Timing Waveform for SRAM Read Cycle**

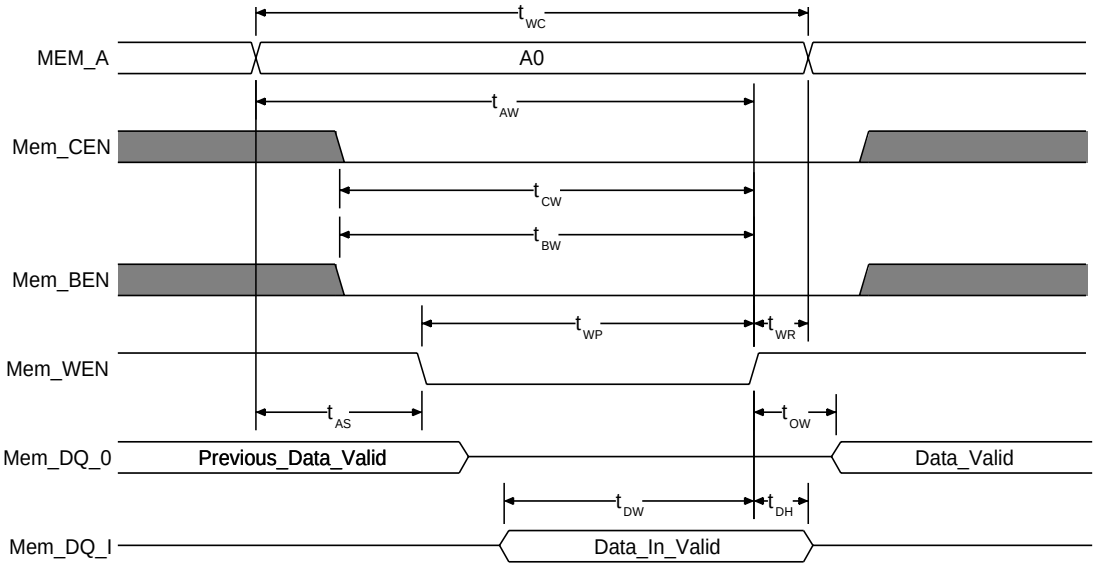


Figure 34: **Timing Waveform for SRAM Write Cycle**

Table 32: **SRAM Parameter Description**

Symbol	Parameter(1)
READ CYCLE	
t_{RC}	Read Cycle Time
t_{AA}	Address Access Time
t_{ACS}	Chip Select Access Time
t_{OE}	Output Enable Low to Output Valid
t_{OH}	Output Hold from Address Change
t_{BE}	Byte Enable Low to Output Valid
WRITE CYCLE	
t_{WC}	Write Cycle Time
t_{AW}	Address Valid to End of Write
t_{CW}	Chip Select Low to End of Write
t_{BW}	Byte Enable Low to End of Write
t_{AS}	Address Set-up Time
t_{WR}	Address Hold from End of Write
t_{WP}	Write Pulse Width
t_{DW}	Data Valid to End of Write
t_{DH}	Data Hold Time
t_{OW}	Write Enable High to Output Low-Z

Notes:
1. Refer to IDT71V416S Data Sheet for specific timing parameters.
2. WEN is HIGH for Read Cycle
3. Address must be valid prior to or coincident with the later CEN, BEN transition LOW; otherwise t_{AA} is the limiting parameter
4. Write Cycle Timing is WEN controlled.

Figure 35 and Figure 36 illustrate the basic read and write functions for StrataFlash. Table 33 defines the symbols used in the figures for StrataFlash.

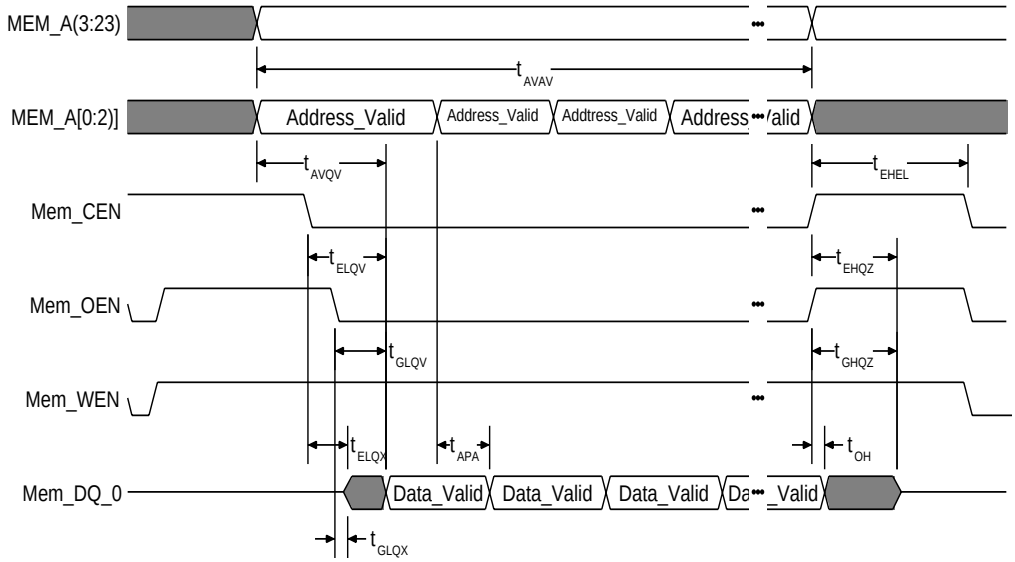


Figure 35: **Waveform for Page-mode and Standard Word/byte Read Operation**

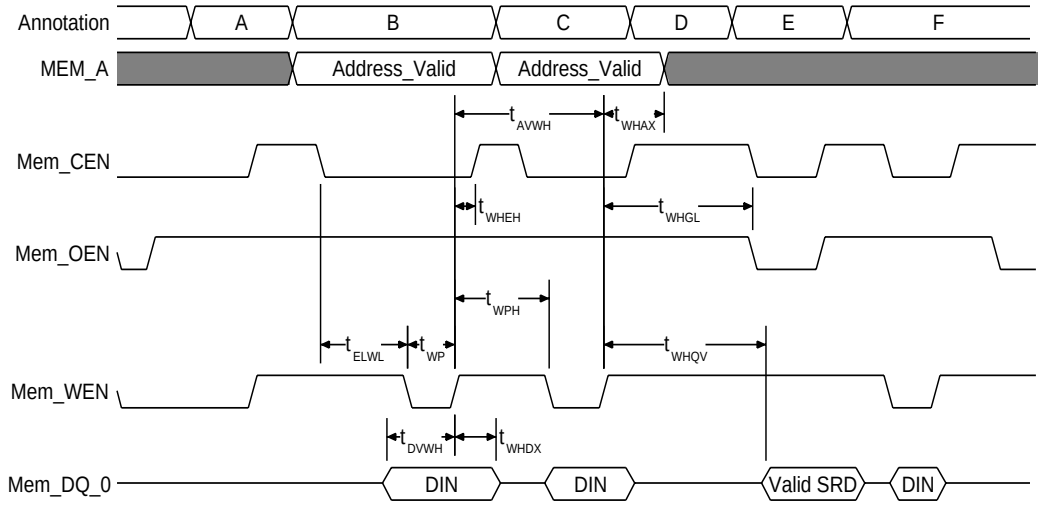


Figure 36: **Waveform for Write Operations**

Table 33: **StrataFlash Parameter Description**

Symbol	Parameter(1)
READ ONLY	
t_{AVAV}	Read/Write Cycle Time
t_{AVQV}	Address to Output Delay
t_{ELQV}	CEN to Output Delay
t_{GLQV}	OEN to Non-Array Output Delay
t_{ELQX}	CEN to Output Low-Z
t_{GLQX}	OEN to Output Low-Z
t_{EHQZ}	CEN High to Output in High-Z

Table 33: **StrataFlash Parameter Description**

Symbol	Parameter[1]
t_{GHQZ}	OEN High to Output in High-Z
t_{OH}	Output Hold from Address, CEN, or OEN Change, Whichever occurs first
t_{EHEL}	CEN High to CEN Low
t_{APA}	Page Address Access Time
Write Operations	
A	Power-up and standby
B	Write block erase, write buffer, or program set-up
C	Write block erase or write buffer confirm, or valid address and data
D	Automated erase delay
E	Read status register or query data
F	Write read array command
t_{ELWL}	CEN(WEN) Low to WEN(CEN) Going Low
t_{WP}	Write Pulse Width
t_{DVWH}	Data Setup to WEN(CEN) Going High
t_{AVWH}	Address Setup to WEN(CEN) Going High
t_{WHEH}	CEN(WEN) Hold from WEN(CEN) High
t_{WHDX}	Data Hold from WEN(CEN) High
t_{WHAX}	Address Hold from WEN(CEN) High
t_{WPH}	Write Pulse High
t_{WHGL}	Write Recovery before Read

Notes:
1. Refer to Intel 28F128J3A Data Sheet for specific timing parameters.

Connecting to Memory

The three primary considerations for connecting the controller to memory devices are the width of the OPB data bus, the width of the memory subsystem, and the number of memory devices used. The width of the memory subsystem is the maximum width of data that can be read from or written to the memory subsystem. The memory width must be less than or equal to the OPB data bus width.

The data and address signals at the memory controller are labeled with big-endian bit labeling (for example, D(0:31), D(0) is the MSB). Most memory devices are either endian-agnostic (they can be connected either way) or little-endian D(31:0) with D(31) as the MSB.

Note Exercise caution with the connections to the external memory devices to avoid incorrect data and address connections. The following tables show the correct mapping of memory controller pins to memory device pins.

Table 34: **Variables used in Defining Memory Subsystem**

Variable	Allowed Range	Definition
BN	0 to 7	Memory bank number
DN	0 to 127	Memory device number within a bank. The memory device attached to the *most significant bit* in the memory subsystem is 0; device numbers increase toward the least significant bit.
MW	8 to 128	Width in bits of memory subsystem
DW	1 to 128	Width in bits of data bus for memory device
MAW	1 to 32	Width in bits of address bus for memory device
AU	1 to 128	Width in bits of smallest addressable data word on the memory device
AS	≥ 0	Address shift for address bus = $\log_2$(MW*AU/DW/8)
HAW	1 to 64	Width of host address bus (e.g. OPB or PLB) in bits

Table 35: **Memory Controller to Memory Interconnect**

Description	EMC Signal (MSB:LSB)	Memory Device Signal (MSB:LSB)
Data bus	Mem_DQ(DN*DW:(DN+1)*DW-1)	D(DW-1:0)
Address bus	Mem_A(HAW-MAW-AS:HAW-AS-1)	A(MAW-1:0)
Chip Enable, low-true	MEM_CEN(BN)	CEN
Output Enable, low-true	MEM_OEN	OEN
Write Enable, low-true	MEM_WEN	WEN (for devices that have byte enables or do not require byte enables)
Byte-Enable-Qualified Write Enable, low-true	MEM_QWEN(INT(DN*DW/8))	WEN (for devices that require byte enables and do not have them)
Byte Enable, low-true	MEM_BEN(INT(DN*DW/8): INT((DN+1)*DW/8-1))	BEN(DW/8-1:0)

Example Memory Connections

Example 1

This example shows the connection to 32-bit memory using 2 IDT71V416S SRAM parts.

Table 36: **Variables for Simple SRAM Example**

Variable	Value	Definition
BN	0	Memory bank number
DN	0 to 1	Memory device number within a bank. The memory device attached to the *most significant bit* in the memory subsystem is 0; device numbers increase toward the least significant bit.
MW	32	Width in bits of memory subsystem
DW	16	Width in bits of data bus for memory device

Table 36: **Variables for Simple SRAM Example *(Continued)***

Variable	Value	Definition
MAW	18	Width in bits of address bus for memory device
AU	16	Width in bits of smallest addressable data word on the memory device
AS	2	Address shift for address bus = $\log_2$(MW*AU/DW/8)
HAW	32	Width of host address bus (e.g. OPB or PLB) in bits

Table 37: **Connection to 32-bit Memory using 2 IDT71V416S Parts**

DN	Description	EMC Signal (MSB:LSB)	Memory Device Signal (MSB:LSB)
0	Data bus	Mem_DQ(0:15)	I/O(15:0)
	Address bus	Mem_A(12:29)	A(17:0)
	Chip Enable, low-true	MEM_CEN(0)	$\overline{CS}$
	Output Enable, low-true	MEM_OEN	$\overline{OE}$
	Write Enable, low-true	MEM_WEN	$\overline{WE}$
	Byte Enable, low-true	MEM_BEN(0:1)	$\overline{BHE}$:$\overline{BLE}$
1	Data bus	Mem_DQ(16:31)	I/O(15:0)
	Address bus	Mem_A(12:29)	A(17:0)
	Chip Enable, low-true	MEM_CEN(0)	$\overline{CS}$
	Output Enable, low-true	MEM_OEN	$\overline{OE}$
	Write Enable, low-true	MEM_WEN	$\overline{WE}$
	Byte Enable, low-true	MEM_BEN(2:3)	$\overline{BHE}$:$\overline{BLE}$

Example 2

This example shows the connection to 2 banks of 64-bit memory using 4 IDT71V416S SRAM parts per bank.

Table 38: **Variables for Two Banks of SRAM**

Variable	Value	Definition
BN	0 to 1	Memory bank number
DN	0 to 3	Memory device number within a bank. The memory device attached to the *most significant bit* in the memory subsystem is 0; device numbers increase toward the least significant bit.
MW	64	Width in bits of memory subsystem
DW	16	Width in bits of data bus for memory device
MAW	18	Width in bits of address bus for memory device

Table 38: **Variables for Two Banks of SRAM *(Continued)***

Variable	Value	Definition
AU	16	Width in bits of smallest addressable data word on the memory device
AS	3	Address shift for address bus = $\log_2$(MW*AU/DW/8)
HAW	32	Width of host address bus (e.g. OPB or PLB) in bits

Table 39: **Connection to 64-bit Memory using 8 IDT71V416S Parts**

BN	DN	Description	EMC Signal (MSB:LSB)	Memory Device Signal (MSB:LSB)
0	0	Data bus	Mem_DQ(0:15)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(0)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(0:1)	$\overline{BHE}$:$\overline{BLE}$
	1	Data bus	Mem_DQ(16:31)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(0)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(2:3)	$\overline{BHE}$:$\overline{BLE}$
	2	Data bus	Mem_DQ(32:47)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(0)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(4:5)	$\overline{BHE}$:$\overline{BLE}$
	3	Data bus	Mem_DQ(48:63)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(0)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(6:7)	$\overline{BHE}$:$\overline{BLE}$

Table 39: **Connection to 64-bit Memory using 8 IDT71V416S Parts *(Continued)***

BN	DN	Description	EMC Signal (MSB:LSB)	Memory Device Signal (MSB:LSB)
1	0	Data bus	Mem_DQ(0:15)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(1)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(0:1)	$\overline{BHE}$:$\overline{BLE}$
	1	Data bus	Mem_DQ(16:31)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(1)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(2:3)	$\overline{BHE}$:$\overline{BLE}$
	2	Data bus	Mem_DQ(32:47)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(1)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(4:5)	$\overline{BHE}$:$\overline{BLE}$
	3	Data bus	Mem_DQ(48:63)	I/O(15:0)
		Address bus	Mem_A(11:28)	A(17:0)
		Chip Enable, low-true	MEM_CEN(1)	$\overline{CS}$
		Output Enable, low-true	MEM_OEN	$\overline{OE}$
		Write Enable, low-true	MEM_WEN	$\overline{WE}$
		Byte Enable, low-true	MEM_BEN(6:7)	$\overline{BHE}$:$\overline{BLE}$

Connecting to Intel StrataFlash

Because StrataFlash parts contain an identifier register, a status register, and a command interface, the bit label ordering is critical to proper functioning. The following tables show examples of how to connect the big-endian EMC buses to the little-endian StrataFlash parts. The proper connection ordering is also indicated in a more general form in Table 35. StrataFlash parts have a x8 mode and a x16 mode, selectable with the BYTE# input pin. To calculate the proper address shift, the minimum addressable word is 8 bits for both x8 and x16 mode, since A0 always selects a byte.

Example 3

This example shows the connection to 32-bit memory using 2 StrataFlash parts in x16 mode (supports byte read, but no byte write; smallest data type that can be written is 16-bit data)

Table 40: **Variables for StrataFlash (x16 mode) Example**

Variable	Value	Definition
BN	0	Memory bank number
DN	0 to 1	Memory device number within a bank. The memory device attached to the **most significant bit** in the memory subsystem is **0**; device numbers increase toward the least significant bit.
MW	32	Width in bits of memory subsystem
DW	16	Width in bits of data bus for memory device
MAW	24	Width in bits of address bus for memory device
AU	8	Width in bits of smallest addressable data word on the memory device
AS	1	Address shift for address bus = $\log_2$(MW*AU/DW/8)
HAW	32	Width of host address bus (e.g. OPB or PLB) in bits

Table 41: **Connection to 32-bit Memory using 2 StrataFlash Parts**

DN	Description	EMC Signal (MSB:LSB)	StrataFlash Signal (MSB:LSB)
0	Data bus	Mem_DQ(0:15)	DQ(15:0)
	Address bus	Mem_A(7:30)	A(23:0)
	Chip Enable, low-true	GND,GND,MEM_CEN(0)	CE(2:0)
	Output Enable, low-true	MEM_OEN	OE#
	Write Enable, low-true	MEM_QWEN(0)	WE#
	Reset/Power down, low-true	MEM_RPN	RP#
	Status, low-true	MEM_STS(0)	STS
	Byte mode select, low-true	N/A - tie to VCC	BYTE#
	Program enable, high-true	N/A - tie to VCC	V_{PEN}
1	Data bus	Mem_DQ(16:31)	DQ(15:0)
	Address bus	Mem_A(7:30)	A(23:0)
	Chip Enable, low-true	GND,GND,MEM_CEN(0)	CE(2:0)
	Output Enable, low-true	MEM_OEN	OE#
	Write Enable, low-true	MEM_QWEN(2)	WE#
	Reset/Power down, low-true	MEM_RPN	RP#
	Status, low-true	MEM_STS(0)	STS
	Byte mode select, low-true	N/A - tie to VCC	BYTE#
	Program enable, high-true	N/A - tie to VCC	V_{PEN}

Example 4

This example shows the connection to 32-bit memory using 4 StrataFlash parts in x8 mode (supports byte reads and writes):

Table 42: **Variables for StrataFlash (x8 mode) Example**

Variable	Value	Definition
BN	0	Memory bank number
DN	0 to 3	Memory device number within a bank. The memory device attached to the **most significant bit** in the memory subsystem is **0**; device numbers increase toward the least significant bit.
MW	32	Width in bits of memory subsystem
DW	8	Width in bits of data bus for memory device
MAW	24	Width in bits of address bus for memory device
AU	8	Width in bits of smallest addressable data word on the memory device
AS	2	Address shift for address bus = $\log_2$(MW*AU/DW/8)
HAW	32	Width of host address bus (e.g. OPB or PLB) in bits

Table 43: **Connection to 32-bit Memory using 4 StrataFlash Parts**

DN	Description	EMC Signal (MSB:LSB)	StrataFlash Signal (MSB:LSB)
0	Data bus	Mem_DQ(0:7)	DQ(7:0)(1)
	Address bus	Mem_A(8:29)	A(23:0)
	Chip Enable, low-true	GND,GND,MEM_CEN(0)	CE(2:0)
	Output Enable, low-true	MEM_OEN	OE#
	Write Enable, low-true	MEM_QWEN(0)	WE#
	Reset/Power down, low-true	MEM_RPN	RP#
	Status, low-true	MEM_STS(0)	STS
	Byte mode select, low-true	N/A - tie to GND	BYTE#
	Program enable, high-true	N/A - tie to VCC	V_{PEN}
1	Data bus	Mem_DQ(8:15)	DQ(7:0)(1)
	Address bus	Mem_A(8:29)	A(23:0)
	Chip Enable, low-true	GND,GND,MEM_CEN(0)	CE(2:0)
	Output Enable, low-true	MEM_OEN	OE#
	Write Enable, low-true	MEM_QWEN(1)	WE#
	Reset/Power down, low-true	MEM_RPN	RP#
	Status, low-true	MEM_STS(0)	STS
	Byte mode select, low-true	N/A - tie to GND	BYTE#
	Program enable, high-true	N/A - tie to VCC	V_{PEN}

Table 43: **Connection to 32-bit Memory using 4 StrataFlash Parts**

DN	Description	EMC Signal (MSB:LSB)	StrataFlash Signal (MSB:LSB)
2	Data bus	Mem_DQ(16:23)	DQ(7:0)(1)
	Address bus	Mem_A(8:29)	A(23:0)
	Chip Enable, low-true	GND,GND,MEM_CEN(0)	CE(2:0)
	Output Enable, low-true	MEM_OEN	OE#
	Write Enable, low-true	MEM_QWEN(2)	WE#
	Reset/Power down, low-true	MEM_RPN	RP#
	Status, low-true	MEM_STS(0)	STS
	Byte mode select, low-true	N/A - tie to GND	BYTE#
	Program enable, high-true	N/A - tie to VCC	V_{PEN}
3	Data bus	Mem_DQ(24:31)	DQ(7:0)(1)
	Address bus	Mem_A(8:29)	A(23:0)
	Chip Enable, low-true	GND,GND,MEM_CEN(0)	CE(2:0)
	Output Enable, low-true	MEM_OEN	OE#
	Write Enable, low-true	MEM_QWEN(3)	WE#
	Reset/Power down, low-true	MEM_RPN	RP#
	Status, low-true	MEM_STS(0)	STS
	Byte mode select, low-true	N/A - tie to GND	BYTE#
	Program enable, high-true	N/A - tie to VCC	V_{PEN}

Notes:
1. In x8 configuration, DQ(15:8) are not used and should be treated according to manufacturer's data sheet.

OPB ZBT Controller Design Specification

March 2002

Summary

This document describes the specifications for a ZBT controller core for the On-chip Peripheral Bus (OPB). This document applies to the following peripherals:

opb_zbt_controller	v1.00a

Overview

The ZBT Memory Controller is a 32-bit peripheral that attaches to the OPB (On-chip Peripheral Bus) and has the following features:

Features

- OPB V2.0 bus interface with byte-enable support
- Supports 32-bit bus interfaces
- Supports memory width of 32-bits

Operation

The OPB ZBT Controller provides an interface between the OPB and external ZBT memories. The controller supports OPB data bus widths of 32bits, and memory subsystem widths of 32 bits. This controller supports the OPB V2.0 byte enable architecture.

OPB ZBT Controller Parameters

To allow you to obtain an ZBT Controller that is uniquely tailored for your system, certain features can be parameterized in the ZBT Controller design. This allows your to configure a design that only utilizes the resources required by your system, and operates with the best possible performance. The features that can be parameterized in the Xilinx ZBT Controller design are shown in Table 1.

Table 1: **ZBT Controller Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
ZBT Memory Base Address	C_BASEADDR	Valid Address Range(2)	None(1)	std_logic_vector
ZBT Memory Address Width	C_ZBT_ADDR_SIZE	2-31	17	integer
Implement ZBT Clock synchronization	C_EXTERNAL_DLL	0 = Do implement 1 = Do not implement	0	integer

Notes:

1. Address range specified by C_BASEADDR must be a power of 2
2. No default value is specified for C_BASEADDR to insure that the actual value is set; if the value is not set, a compiler error is generated. These generics must be a power of 2.

ZBT Controller I/O Signals

The I/O signals for the ZBT Controller are listed in Table 2.

Table 2: **ZBT COntroller I/O Signals**

Signal Name	Interface	I/O	Description
OPB_Clk	OPB	I	OPB Clock
OPB_Rst	OPB	I	OPB Reset
OPB_ABus(0:31)	OPB	I	OPB Address Bus
OPB_BE(0:3)	OPB	I	OPB Byte Enables
OPB_DBus(0:31)	OPB	I	OPB Data Bus
OPB_RNW	OPB	I	OPB Read, Not Write
OPB_select	OPB	I	OPB Select
OPB_seqAddr	OPB	I	OPB Sequential Address
ZBT_DBus(0:31zbt)	OPB	O	ZBT Controller Data Bus
ZBT_errAck	OPB	O	ZBT Controller Error Acknowledge
ZBT_retry	OPB	O	ZBT Controller Retry
ZBT_toutSup	OPB	O	ZBT Controller Timeout Suppress
ZBT_xferAck	OPB	O	ZBT Controller Transfer Acknowledge
ZBT_Clk_FB	IP Core	I	Feedback ZBT Clock for clock synchronization
ZBT_Clk_FBOut	IP Core	O	Feedback ZBT Clock for clock synchronization
ZBT_Clk	IP Core	O	ZBT Memory clock
ZBT_CKE_N	IP Core	O	ZBT Memory clock enable
ZBT_OE_N	IP Core	O	ZBT Memory output enable
ZBT_ADV_LD_N	IP Core	O	ZBT Memory Control signal
ZBT_LBO_N	IP Core	O	ZBT Memory Control signal
ZBT_CE1_N	IP Core	O	ZBT Memory Select signal
ZBT_CE2_N	IP Core	O	ZBT Memory Select signal
ZBT_CE2	IP Core	O	ZBT Memory Select signal
ZBT_RW_N	IP Core	O	ZBT Memory Read/Write signal
ZBT_A(0:C_ZBT_ADDR_SIZE-1)	IP Core	O	ZBT Memory Address
ZBT_BW_N(0:3)	IP Core	O	ZBT Memory Byte Enable
ZBT_IO_I(0:31)	IP Core	I	ZBT Memory Input Data Bus
ZBT_IO_O(0:31)	IP Core	O	ZBT Memory Output Data Bus
ZBT_IO_T	IP Core	O	ZBT Memory Output 3-state Signal
ZBT_IOP_I(1:4)	IP Core	I	ZBT Memory Input Parity Data Bus
ZBT_IOP_O(1:4)	IP Core	O	ZBT Memory Output Parity Data Bus
ZBT_IOP_T	IP Core	O	ZBT Memory Output Parity 3-state Signal

Connecting to Memory

The following table shows how to connect a ZBT controller and SAMSUNG K7N163601M-OC15 (512kword x 32). The ZBT controller does not support sleep mode, burst mode, parity checking, or parity generating.

Table 3: **Signal Connection**

ZBT Controller	SAMSUNG K7N163601M-OC15
ZBT_A(0:19)	A(0:19)
ZBT_Clk	CLK
ZBT_CKE_N	$\overline{\text{CKE}}$
ZBT_RW_N	$\overline{\text{WE}}$
ZBT_ADV_LD_N	ADV
ZBT_OE_N	$\overline{\text{OE}}$
ZBT_CE1_N	$\overline{\text{CS1}}$
ZBT_IO(24:31)	DQA(0:7)
ZBT_IO(16:23)	DQB(0:7)
ZBT_IO(8:15)	DQC(0:7)
ZBT_IO(0:7)	DQD(0:7)
ZBT_BW_N(4)	$\overline{\text{BWA}}$
ZBT_BW_N(2)	$\overline{\text{BWB}}$
ZBT_BW_N(1)	$\overline{\text{BWC}}$
ZBT_BW_N(0)	$\overline{\text{BWD}}$
(Need to be tied to VCC)	CS2
(Need to be tied to GND)	$\overline{\text{CS2}}$
(Need to be tied to GND)	ZZ
(Need to be tied to GND)	$\overline{\text{LBO}}$

Address Mapping

The generic C_ZBT_ADDR_SIZE specifies the number of address signals to the ZBT memory. Since all accesses are word, the generic specifies the number of address bits for word accesses. For example, 512 KWord memory needs 19 address bits (2^19 = 512288). The address decoding uses OPB_A(0 to 29-C_ZBT_ADDR_SIZE) and the generic C_BASEADDR to determine if the OPB access is to the ZBT memory.

Timing Diagrams

A read uses five OPB clock cycles to complete, and a write uses 2 OPB clock cycles. The write is faster because pipelining is used for the ZBT controller and the ZBT memories. Figure 1 shows a simple read access. Figure 2 shows two successive writes to illustrate the write pipelining.

Figure 1: **Read Cycle**

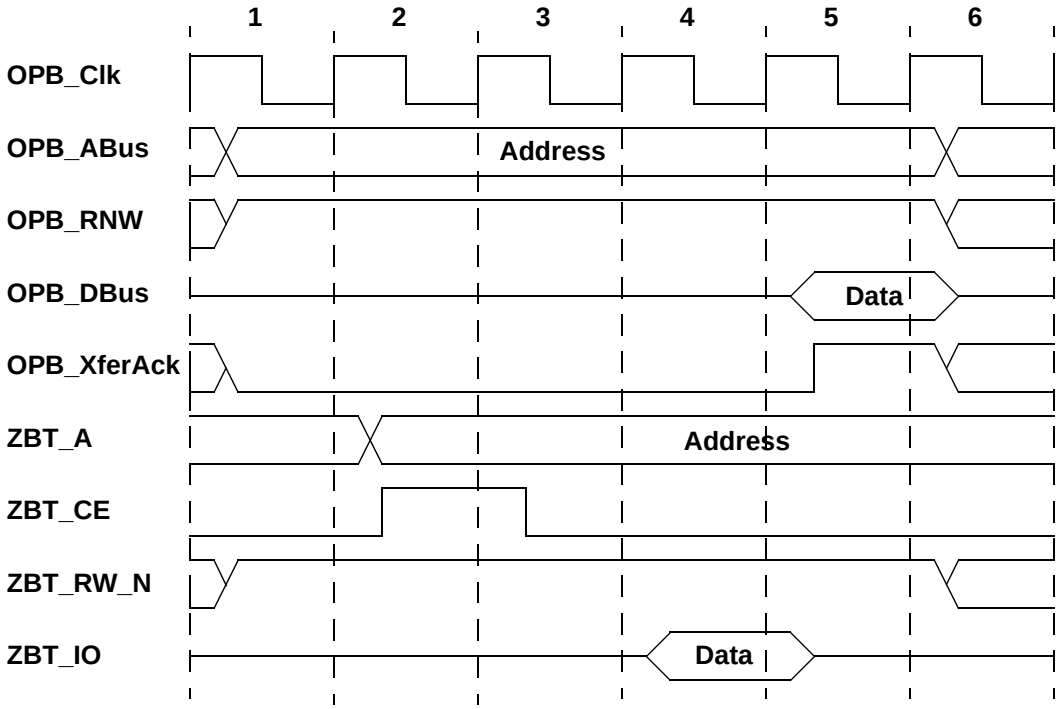
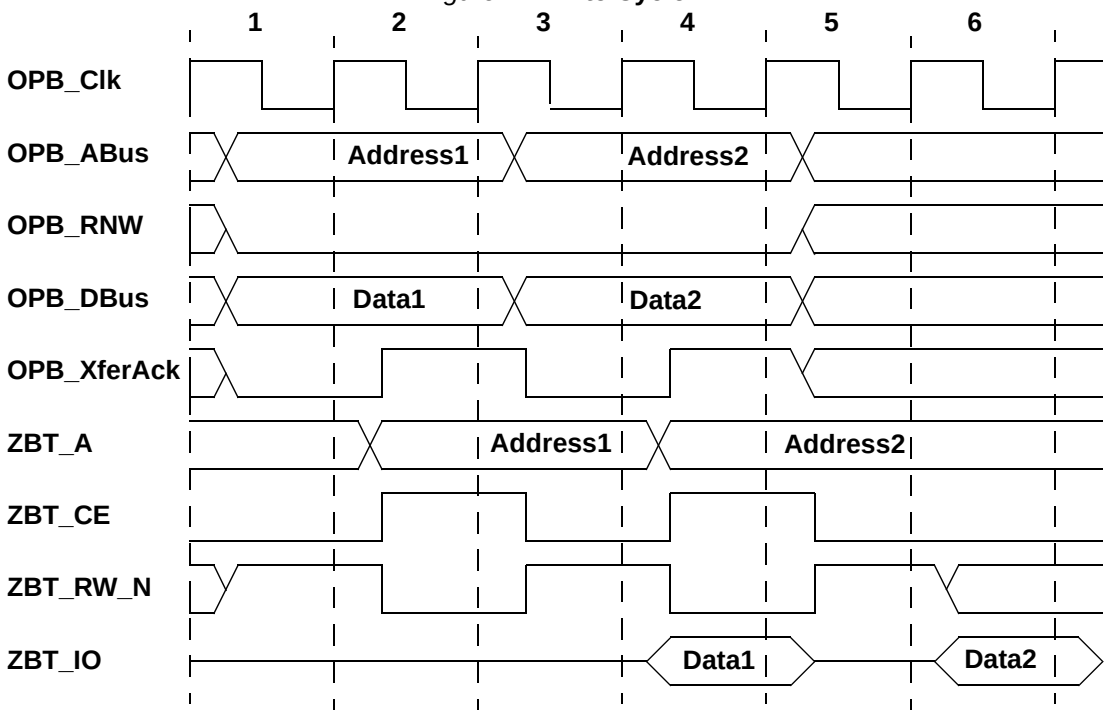


Figure 2: **Write Cycle**

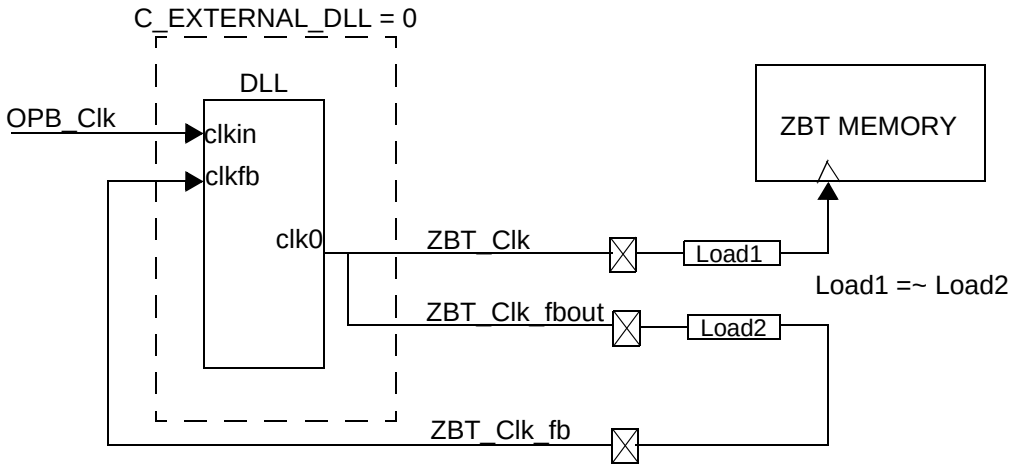


Clock Handling

Since ZBT memories are clocked and all timing on signals to the memories is referencing the ZBT clock, careful handling of the clock is important. Since all signals to the ZBT are driven from the OPB_Clk, the ZBT clock must be synchronized to the OPB_Clk. You can do this with a DLL, as shown in Figure 3

The controller has a generic C_EXTERNAL_DLL that specifies if the peripheral implements the DLL. If there is more than one ZBT controller in a design, the number of DLLs can exceed the number of available DLLs. Consequently, a more centralized handling of the clocks is required. In this case, one DLL synchronizes all ZBT clocks and each of the controllers must inhibit the implementing of the DLL (C_EXTERNAL_DLL = 1). This centralized DLL drives all ZBT clocks, and requires that the loads on each of the ZBT clocks is equal.

Figure 3: **Clock Synchronization**



Programming Model

Register Data Types and Organization

The ZBT controller is organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in Figure 4.

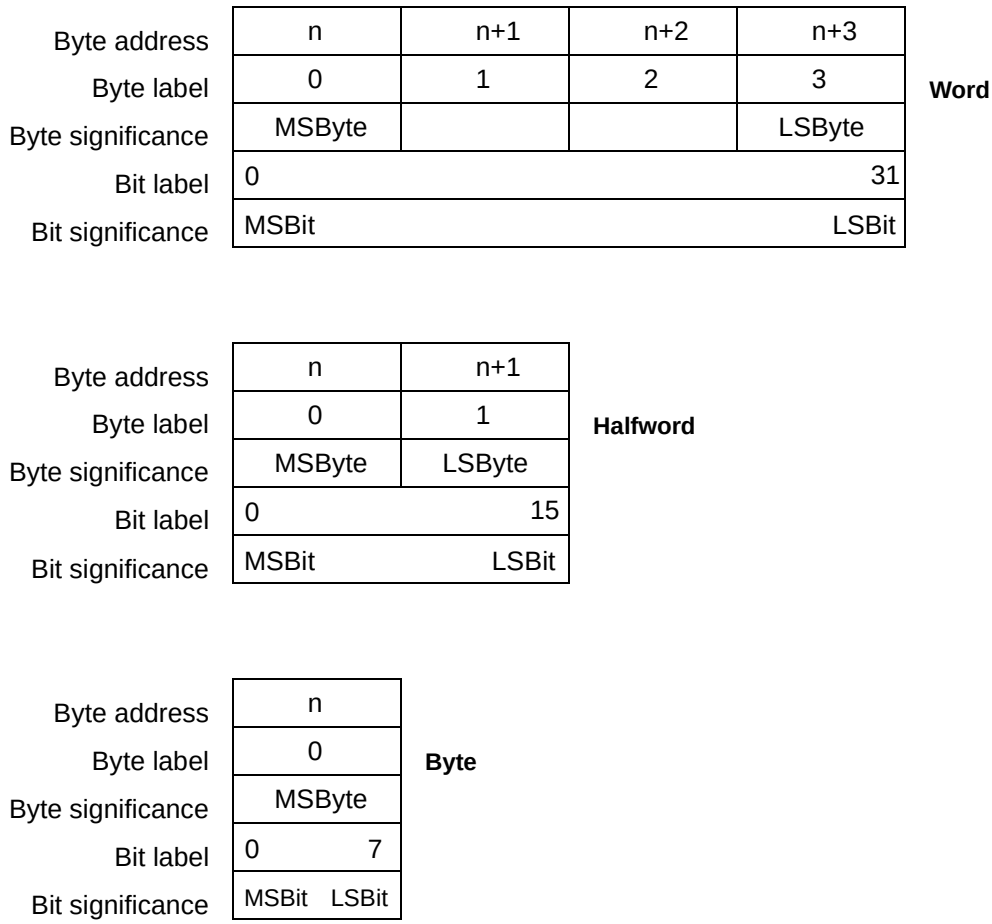


Figure 4: **Big-Endian Data Types**

Implementation

Design Tips

To achieve the highest fmax on the ZBT controller, the Xilinx implementation tools must force flip-flops to the IO pads. This option is turned off by default.

OPB Block RAM (BRAM) Specification

March 2002

Summary

This document describes the specifications for an OPB BRAM core for the MicroBlaze soft processor and other embedded processors. This document applies to the following peripherals:

opb_bram	v1.00a

Overview

The OPB BRAM is a module that attaches to the OPB (On-chip Peripheral Bus), and has the following features:

Features

- OPB V2.0 bus interface with byte-enable support
- Number of BRAMs is configurable
- Handles byte, half-word and word transfers
- Other port of the BRAM is available for customer designs
- Handles Virtex, Virtex-E, Spartan-II, Virtex-II and Virtex-II PRO type of BRAM

OPB_BRAM Parameters

To allow you to obtain an OPB_BRAM that is uniquely tailored for your system, certain features can be parameterized in the OPB_BRAM design. This allows you to configure a design that only utilizes the resources required by your system, and operates with the best possible performance. The features that can be parameterized in Xilinx OPB_BRAM designs are shown in Table 1.

Table 1: **OPB_BRAM Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
OPB_BRAM Registers Base Address	C_BASEADDR	Valid Address Range(2)	None(1)	std_logic_vector
OPB_BRAM Registers HIGH Address	C_HIGHADDR	Valid Address Range(2)	None(1)	std_logic_vector
Target Family	C_FAMILY	Xilinx FPGA families	virtex2	strings
OPB Data Bus Width	C_OPB_DWIDTH	32	32	integer
OPBAddress Bus Width	C_OPB_AWIDTH	8 - 32	32	integer

Notes:

1. Address range specified by C_BASEADDR and C_HIGHADDR must be a power of 2
2. No default value is specified for C_BASEADDR and C_HIGHADDR to insure that the actual value is set; if the value is not set, a compiler error is generated. These generics must be a power of 2.

OPB_BRAM I/O Signals

The I/O signals for the OPB_BRAM are listed in Table 2.

Table 2: **OPB_BRAM I/O Signals**

Signal Name	Interface	I/O	Description
OPB_Clk	OPB	I	OPB Clock
OPB_Rst	OPB	I	OPB Reset
OPB_ABus[0:C_OPB_AWIDTH-1]	OPB	I	OPB Address Bus
OPB_BE[0:C_OPB_DWIDTH/8-1]	OPB	I	OPB Byte Enables
OPB_DBus[0:C_OPB_DWIDTH-1]	OPB	I	OPB Data Bus
OPB_RNW	OPB	I	OPB Read, Not Write
OPB_select	OPB	I	OPB Select
OPB_seqAddr	OPB	I	OPB Sequential Address
OPB_BRAM_DBus[0:C_OPB_DWIDTH-1]	OPB	O	OPB_BRAM Data Bus
OPB_BRAM_errAck	OPB	O	OPB_BRAM Error Acknowledge
OPB_BRAM_retry	OPB	O	OPB_BRAM Retry
OPB_BRAM_toutSup	OPB	O	OPB_BRAM Timeout Suppress
OPB_BRAM_xferAck	OPB	O	OPB_BRAM Transfer Acknowledge
BRAM_Clk	Other Port	I	Other Port Clock
BRAM_Addr[0:31]	Other Port	I	Other Port Address Bus
BRAM_We[0:3]	Other Port	I	Other Port Byte Enables
BRAM_Write_Data[0:31]	Other Port	I	Other Port Write Data Bus
BRAM_Read_Data[0:31]	Other Port	O	Other Port Read Data Bus

Programming Model

Supported Memory Sizes

The following sizes are supported for Virtex, Virtex-E, and Spartan-II:

- 4 BRAMs => 2 Kbyte
- 8 BRAMs => 4 Kbyte
- 16 BRAMs => 8 Kbyte
- 32 BRAMs => 16 Kbyte

The following sizes are supported for Virtex-II and Virtex-II PRO:

- 4 BRAMs => 8 Kbyte
- 8 BRAMs => 16 Kbyte
- 16 BRAMs => 32 Kbyte
- 32 BRAMs => 64 Kbyte

Register Data Types and Organization

The BRAMs are organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in Figure 1.

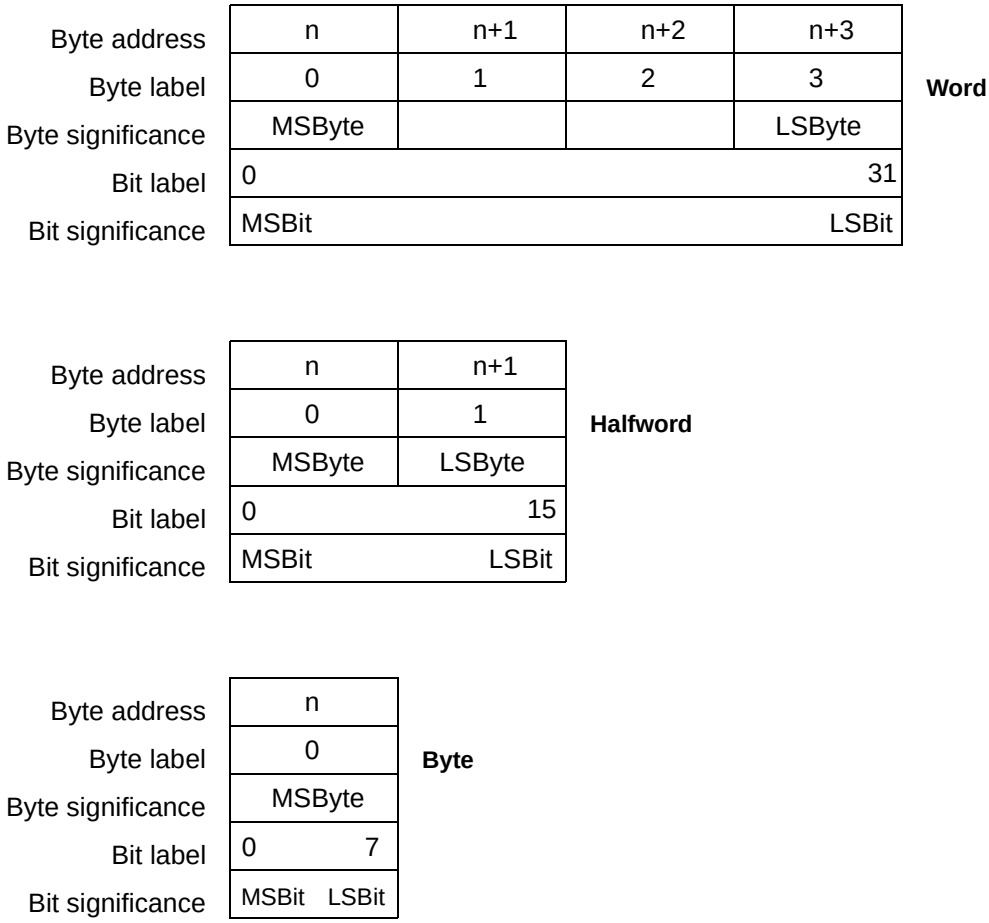


Figure 1: **Big-Endian Data Types**

OPB UART Lite Specification

March 2002

Summary

This document describes the specifications for a UART core for the On-chip Peripheral Bus (OPB). This document applies to the following peripherals:

opb_uartlite	v1.00a

Overview

The UART Lite is a module that attaches to the OPB (On-chip Peripheral Bus) and has the following features:

Features

- OPB V2.0 bus interface with byte-enable support
- Supports 8-bit bus interfaces
- One transmit and one receive channel (full duplex)
- 16-character transmit FIFO and 16-character receive FIFO
- Number of databits in a character is configurable (5-8)
- Parity; can be configured for odd or even
- Configurable baud rate

UART Lite Parameters

To allow you to obtain a UART Lite that is uniquely tailored for your system, certain features can be parameterized in a UART Lite design. This allows you to configure a design that only utilizes the resources required by your system, and operates with the best possible performance. The features that can be parameterized in the Xilinx UART Lite design are shown in Table 1.

Table 1: **UART Lite Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
UART Lite Registers Base Address	C_BASEADDR	Valid Address Range(2)	None(1)	std_logic_vector
UART Lite Registers High Address	C_HIGHADDR	Valid Address Range(2)	None(1)	std_logic_vector
Target Family	C_FAMILY	Xilinx FPGA families	virtex2	strings
OPB Bus Width	C_OPB_AWIDTH	32	32	integer
OPB Data Bus Width	C_OPB_DWIDTH	32	32	integer
C_CLK_FREQ	Clock frequency of the OPB system clock driving the UART Lite peripheral in Hz.	integer (ex. 125000000)	125_000_000	integer
C_BAUDRATE	Baud rate of the UART Lite in bits per second.	integer (ex. 9600)	19_200	integer

Table 1: **UART Lite Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
C_DATA_BITS	The number of data bits in the serial frame.	integer (5 to 8)	8	integer
C_USE_PARITY	Determines whether parity is used or not.	Integer 1 = use parity, 0 = do not use parity.	1	integer
C_ODD_PARITY	If parity is used, determines whether parity is odd or even	integer 1= odd parity, 0 = even parity.	1	integer

UART Lite I/O Signals

The I/O signals for the UART Lite are listed in Table 2.

Table 2: **UART Lite I/O Signals**

Signal Name	Interface	I/O	Description
OPB_Clk	OPB	I	OPB Clock
OPB_Rst	OPB	I	OPB Reset
OPB_ABus[0:31]	OPB	I	OPB Address Bus
OPB_BE[0:3]	OPB	I	OPB Byte Enables
OPB_DBus[0:31]	OPB	I	OPB Data Bus
OPB_RNW	OPB	I	OPB Read, Not Write
OPB_select	OPB	I	OPB Select
OPB_seqAddr	OPB	I	OPB Sequential Address
UART_DBus[0:31]	OPB	O	UART Data Bus
UART_errAck	OPB	O	UART Error Acknowledge
UART_retry	OPB	O	UART Retry
UART_toutSup	OPB	O	UART Timeout Suppress
UART_xferAck	OPB	O	UART Transfer Acknowledge
Interrupt	Interrupt	O	UART Interrupt
RX	External	I	Receive Data
TX	External	O	Transmit Data

JTAG_UART Address Map and Register Descriptions

Register Data Types and Organization

Registers in the UART Lite are accessed as one of three types: byte (8 bits), halfword (2 bytes), and word (4 bytes). All register accesses are on word boundaries to conform to the OPB-IPIF register location convention. The addresses of the UART Lite registers are provided in the **Address Map** section.

The UART Lite registers are organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in Figure 1.

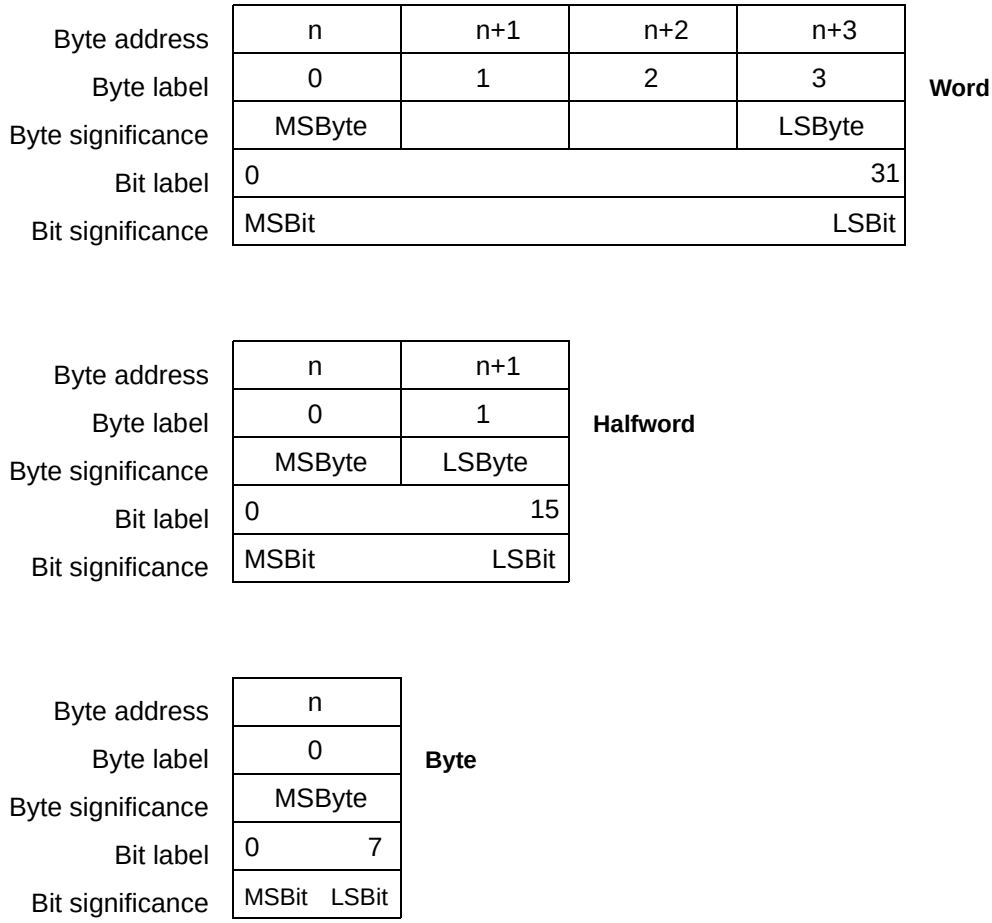


Figure 1: **Big-Endian Data Types**

Registers of the UART Lite

Information on the following registers used in assembly language programming are described in this section.

Receive FIFO	Read character from Receive FIFO
Transmit FIFO	Write character into Transmit FIFO
Status	Read from Status Register
Control	Write to Control Register

Figure 2: **UART Lite Register Set**

Status Register (STATREG)

The Status register contains the status of the receive and transmit FIFO, if interrupts are enabled, and if there are any errors.

Table 3: **Status Register**

Bits	Name	Description	Reset Value
0-23	Reserved	Not used	0
24	PAR_ERROR	Parity Error Indicates that a parity error has occurred since the last time the status register was read. If the UART is configured without any parity handling, this bit will always be '0'. The received character will be written into the receive FIFO. The bit will be cleared when the status register is read 0 No parity error has occurred 1 A parity error has occurred	0
25	FRAME_ERROR	Frame Error Indicates that a frame error has occurred since the last time the status register was read. Frame Error is defined as detection of a stop bit with the value '0'. The receive character will be ignored and NOT written to the receive FIFO. The bit will be cleared when the status register is read 0 No Frame error has occurred 1 A frame error has occurred	0
26	OVERUN_ERROR	Overrun Error Indicates that a overrun error has occurred since the last time the status register was read. Overrun is when a new character has been received but the receive fifo is full. The received character will be ignored and NOT written into the receive FIFO. The bit will be cleared when the status register is read 0 No interrupt has occurred 1 Interrupt has occurred	0
27	INTR_ENABLED	Interrupts is enabled Indicates that interrupts is enabled 0 Interrupt is disabled 1 Interrupt is enabled	0

Table 3: **Status Register *(Continued)***

Bits	Name	Description	Reset Value
28	TX_FIFO_FULL	Transmit FIFO is full Indicates if the transmit FIFO is full. 0 Transmit FIFO is not full 1 Transmit FIFO is full	
29	TX_FIFO_EMPTY	Transmit FIFO is empty Indicates if the transmit FIFO is empty. 0 Transmit FIFO is not empty 1 Transmit FIFO is empty	
30	RX_FIFO_FULL	Receive FIFO is full Indicates if the receive FIFO is full. 0 Receive FIFO is not full 1 Receive FIFO is full	
31	RX_FIFO_VALID_DATA	Receive FIFO is has valid data Indicates if the receive FIFO has valid data. 0 Receive FIFO is empty 1 Receive FIFO has valid data	

Control Register (CTRL_REG)

The Control register contains the UART Lite control.

Table 4: **Control Register (CTRL_REG)**

Bits	Name	Description	Reset Value
0-26	Reserved	Not used	0
27	ENABLE_INTR	Enable Interrupt for the UART 0 Disable interrupt signal 1 Enable interrupt signal	0
28-29	Reserved	Not used	0
30	RST_RX_FIFO	Reset/Clear the receive FIFO When written to with a '1' the receive FIFO is cleared. 0 Do nothing 1 Clear the receive FIFO	0
31	RST_TX_FIFO	Reset/Clear the transmit FIFO When written to with a '1' the transmit FIFO is cleared. 0 Do nothing 1 Clear the transmit FIFO	0

Address Map

UART_BASE_ADDRESS + 0: Read from Receive FIFO

UART_BASE_ADDRESS + 4: Write to transmit FIFO

UART_BASE_ADDRESS + 8: Read from Status Register

UART_BASE_ADDRESS + 12: Write to Control Register

Interrupts

If interrupts are enabled, an interrupt is generated when one of the following conditions is true:

1. When there exists any valid character in the receive FIFO, the interrupt stays active until the receive FIFO is empty.

2. When the transmit FIFO goes from not empty to empty, such as when the last character in the transmit FIFO is transmitted, the interrupt is only active one clock cycle.

Design Implementation

Device Utilization and Performance Benchmarks

The following table shows approximate resource utilization and performance benchmarks for the OPB UART Lite. The estimates shown are not guaranteed and can vary with FPGA family and speed grade, implementation parameters, user timing constraints, and implementation tool version. Only parameters that affect resource utilization are shown in the following table.

Table 5: **OPB UART Lite Performance and Resource Utilization Benchmarks (Virtex-II 2V1000-5)**

Parameter Values							Device Resources		f_{MAX} (MHz)
Address Bits in Decode	C_AWIDTH	C_CLK_FREQ	C_BAUDRATE	C_DATA_BITS	C_USE_PARITY	C_ODD_PARITY	Flip-Flops	4-input LUTs	f_{MAX}
24	32	100_000_000	19_200	5	FALSE	FALSE	48	88	
24	32	100_000_000	19_200	6	FALSE	FALSE	49	92	
24	32	100_000_000	19_200	7	FALSE	FALSE	50	95	
24	32	100_000_000	19_200	8	FALSE	FALSE	51	100	
24	32	40_000_000	38_400	8	FALSE	FALSE	49	97	158
24	32	100_000_000	19_200	6	TRUE	FALSE	57	108	137
24	32	100_000_000	19_200	7	TRUE	FALSE	57	108	137

March 2002

OPB Serial Peripheral Interface (SPI) Design Specification

Summary

This document presents specifications for the VHDL implementation of Motorola's Serial Peripheral Interface (SPI) in a Xilinx FPGA. The original specifications closely followed Motorola's M68HC11-Rev. 4.0 Reference Manual, and this document emphasizes the M68HC-11 specifications. However, the design was enhanced with a number of exceptions and enhancements as described in this document. The default mode of operation has been changed to a manual slave select operation (not included in the M68HC11 specification). This document applies to the following peripherals:

opb_spi	v1.00b

Introduction

The Serial Peripheral Interface (SPI) is a full-duplex, synchronous channel that supports a four-wire interface (receive, transmit, clock and slave select) between one master and one slave. The original specifications followed closely Motorola's M68HC11-Rev. 4.0 Reference Manual. There are difference from the 68HC11 specification that should be reviewed when utilizing this SPI Assembly, see **Specification Exceptions**.

The Version B specification has extended functionality, including a manual slave select mode. This mode allows you to manually control the slave select line directly by the data written to the slave select register. This allows transfers of an arbitrary number of bytes without toggling the slave select line until all bytes are transferred. In this mode, you must toggle the slave select by writing the appropriate data to the slave select register. The manual slave select mode is the default mode of operation.

This parameterized module permits additional slaves with automatic generation of the required decoding of the individual slave select outputs by the master. Additional masters can be added as well; however, means to detect all possible conflicts are not implemented with this interface standard, but rather require the software to arbitrate bus control in order to eliminate conflicts. At this time only SPI slave devices are allowed off-chip. This is an artifact of software master control arbitration which can not be guaranteed if off-chip masters were allowed and is due to issues with asynchronous external clocks as well. Essentially any number of internal slave and master SPI devices is allowed. The actual number is limited by the performance that is desired.

NOTICE

SPI Device Features

- Four signal interface (MOSI, MISO, SCK, and $\overline{SS}$)-- $\overline{SS}$ bit for each slave on the SPI bus
- Three signal in/out (in, out, 3-state) for implementing 3-state SPI device in/outs to support multi-master configuration within the FPGA
- Full-duplex operation
- Works with N times 8-bit data characters in default configuration. The default mode implements manual control of the $\overline{SS}$ output via data written to the slave select register which appears directly on the $\overline{SS}$ output when the master is enabled. This mode can be used only with external slave devices. In addition, an optional operation where the $\overline{SS}$ output is toggled automatically with each 8-bit character transfer by the master device internal state machine can be selected via a bit in the command register for SPI master devices.
- Supports back-to-back character transmission and reception
- Master and slave SPI modes supported
- Multi-master environment supported (implemented with 3-state drivers and requires software arbitration for possible conflict)
- Multi-slave environment supported (automatic generation of additional master slave select signals)
- Continuous transfer mode for automatic scanning of a peripheral
- Supports maximum clock rates of up to one-half the OPB clock rate in both master and slave modes when both SPI devices are in the same FPGA part (routing constraints of SPI bus signals must be incorporated in map/par process). In anticipation of remote master operation, slaves operation supports one-fourth the OPB clock rate (artifact of asynchronous SCK clock relative to the OPB clock which requires clock synchronization).
- Parameterizable baud rate generator
- Programmable clock phase and polarity
- External ports (selected via a parameter) for off-chip slave interconnects (off-chip masters not supported)
- Optional transmit and receive FIFOs (implemented as a pair only)
- Local loopback capability for testing

The Xilinx SPI design allows you to tailor the SPI Assembly to suit your application by setting certain parameters to enable or disable features. The parameterizable features of the design are discussed in the **SPI Configuration Parameters** section.

The basic SPI device consists of a register module and the SPI module. Optional FIFOs and support are discussed in a later section. The register block includes all memory mapped registers (as shown in Figure 1) and resides on the Xilinx OPB. As shown in Figure 3, the SPI module consists of transmitter and receiver sections, a parameterized baud rate generator (BRG) and a control unit. The registers are an 8-bit status register, an 8-bit control register, an N-bit slave select register and a pair of 8-bit transmit/receiver registers. In the 68HC11 implementation, the transmit register is transparent to the shift register and the receive register is double buffered with the shift register. In this implementation without FIFOs, both the transmit and receive register are double buffered. Hardware prevents data transfer from the transmit buffer to the shift register while an SPI transfer is in progress, consequently, the write collision error described in the MC68HC11 Reference Manual can not occur and the WCOL flag is not supported. All registers are accessed directly from the Xilinx OPB which is a subset of IBM's 64-bit OPB utilizing byte enables (see IBM's 64-Bit On-Chip Peripheral Bus document for details). As shown in Figure 1, optional FIFOs can be implemented on both receive and transmit paths.

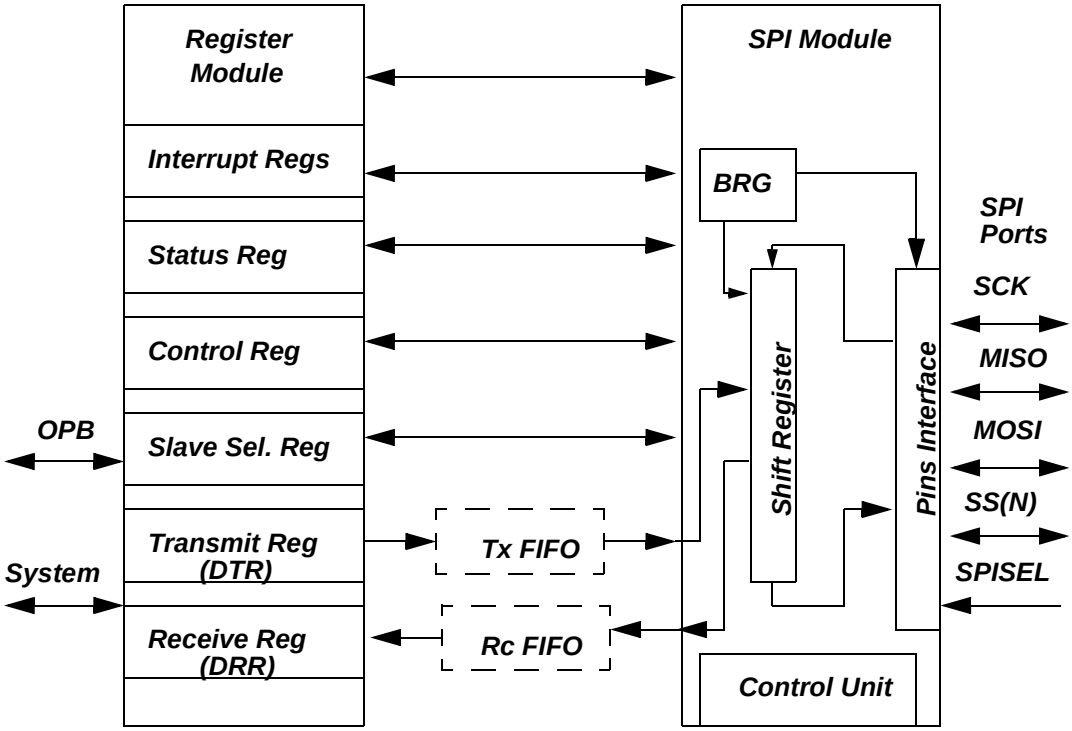


Figure 1: **SPI Assembly Top-level Block Diagram**

SPI Protocol with Automatic Slave Select Assertion

This section describes the SPI protocol where Slave Select (SS(N)) is asserted automatically by the SPI Master device (i.e CR(24) = 0). This is not the default configuration, but was the only mode of operation in the first specification of this device. This operation follows closely the specification in Motorola 68HC11 Reference Manual. For more details and timing diagrams, please refer to the description of the SPI bus in the Motorola 68HC11 Reference Manual.

The SPI bus to a given slave device (N-th device) consists of four wires, Serial Clock (SCK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and Slave Select ($\overline{SS}$(N)), SCK, MOSI, and MISO are shared signals for all slaves and masters. Each master SPI device has the functionality to generates an active-low, one-hot encoded $\overline{SS}$(N) vector where each bit is assigned as a $\overline{SS}$-signal to each slave SPI device. It is possible for SPI Master/Slave devices to be both internal to the FPGA and SPI Slave devices to be external to the FPGA package; when external slave devices are to be on the SPI bus, SPI pins are automatically instantiated. This option is chosen via Platform Generator. Multiple SPI Master/Slave devices are diagrammed in Figure 2. When a SPI device is slave-only, then the slave select register and multiple SS(N) outputs are not included in that device implementation. Slave-only option is selected via Platform Generator.

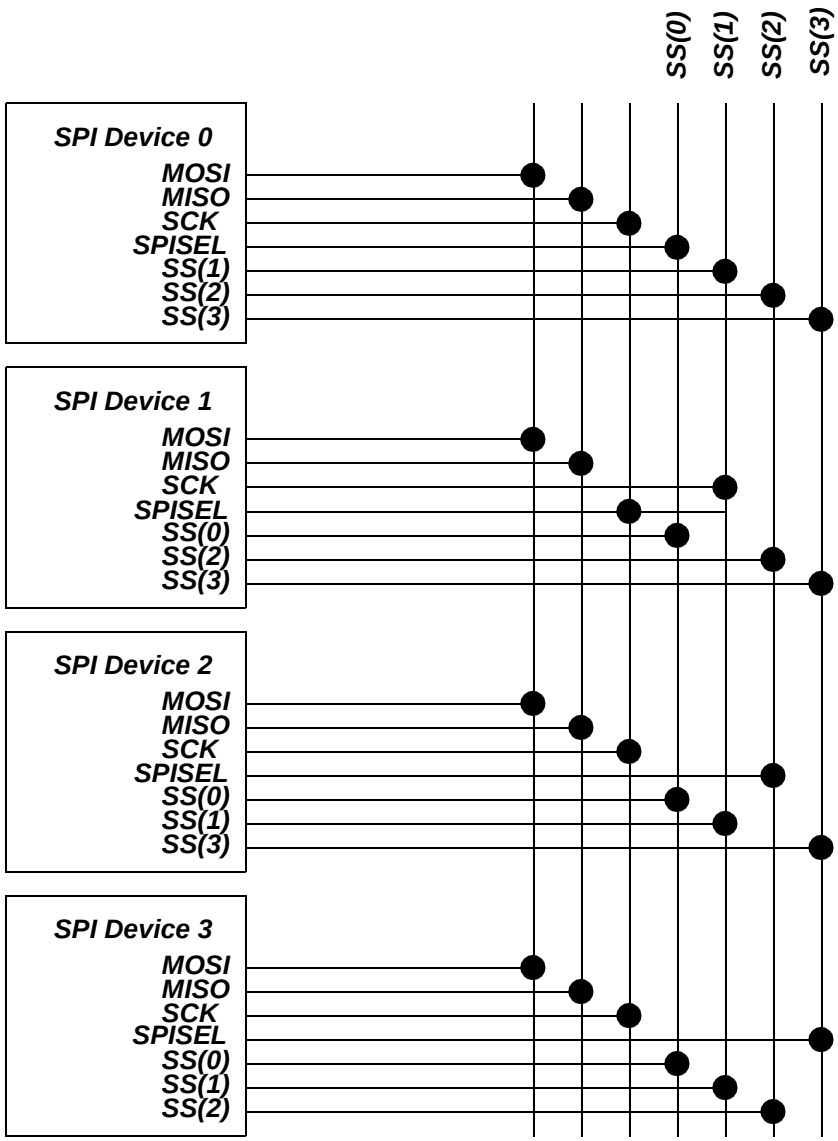


Figure 2: **Multi-master Configuration Block Diagram. Slave only devices, which are not shown, have only SPISEL local slave select port and do not have SS(N) remote slave select port.**

The SCK signal is driven by the SPI Master controlling the bus and regulates the flow of data. The master must be configured at the time of system configuring to transmit data at a one of four baud rates. The 68HC11 SPI specification prescribes baud rate selection via bits in the control register; however, in this FPGA implementation, the baud rate is selected via a parameter that fixes the baud rate at the time of system configuration. The FPGA permits reconfiguration by resetting the parameters and rebuilding the system. This approach was adopted to reduce FPGA resource requirements.

One bit of data is transferred per SCK clock period. Data is shifted on one edge of SCK and is sampled on the opposite edge when the data is stable. Consistent with the 68HC11 SPI specification, selection of clock polarity and a choice of two fundamentally different clocking protocols on an 8-bit oriented data transfer is possible via bits in the control register.

The clock phase and polarity (or idle state) can be modified for SPI data transfers with programmable bits in the control register. The clock polarity (CPOL) bit selects an active high (i.e. idles low) or active low clock (i.e. idle high). The clock phase (CPHA) bit can be set to select one of two fundamentally different transfer formats. If CPHA = '0', data is valid on the first SCK edge (rising or falling) after SS(N) has been asserted. If CPHA = '1', data is valid on the second SCK edge (rising or falling) after SS(N) has asserted. Determination of whether the edge of interest is rising or falling edge depends on the idle state (i.e. CPOL setting). The clock phase and polarity must be identical for the master SPI device and the selected slave device. Figure 3 and Figure 4 shows four possible clock behaviors and these diagrams are discussed in more detail below.

Both the MOSI and MISO port behaviors are different depending on the whether the SPI device is configured as a master or a slave. When configured as a master, the MOSI port is a serial data output port and the MISO is a serial data input port. The opposite is true when the device is configured as a slave; the MISO port is a slave serial data output port and the MOSI is a serial data input port. There may be only one master and one slave transmitting data at any given time. The bus architecture provides limited contention error detection (i.e. multiple devices driving the shared MISO and MOSI signals) and requires the software to provide arbitration to prevent most possible contention errors.

All SCK, MOSI, and MISO pins of all devices are respectively hardwired together. For all transactions, a single SPI device is configured as a master and all other SPI devices on the SPI bus are configured as slaves. The single master drives the SCK and MOSI pins to the SCK and MOSI pins of the slaves. The uniquely selected slave device drives data out its MISO pin to the MISO master pin to realize full-duplex communication.

The Nth bit of the SS(N) signal selects the Nth SPI slave with an active-low signal. All other slave devices ignore both SCK and MOSI signals. In addition, the non-selected slaves (i.e. SS pin high) maintain their MISO pin so as to not interfere with SPI bus activities.

When external slave SPI devices are implemented, SCK, MOSI, and MISO, as well as the needed SS(N) signals, are brought out to pins. All signals are true 3-state bus signals and erroneous external bus activity can corrupt internal transfers when both internal and external devices are present. You must insure that external pull-up or pull-down of external SPI 3-state signals are consistent with the sink/source capability of the FPGA IO drivers. Recall that the IO drivers can configured for different drive strengths as well as integral pull-ups. The 3-state signals for multiple external slaves can be implemented however the system designer desires, but the external bus must follow the SPI 68HC11 specifications.

Figure 3 shows the timing diagram for an SPI data transfer when the clock phase, CPHA, is set to '0'. The waveforms are shown for both positive and negative clock polarities of SCK (i.e. CPOL = 1 and = 0). Recall the with CPHA=1, data is valid on the first clock edge (rising or falling depending on CPOL). Therefore, SCK signal remains in the idle state until one-half period following assertion of the slave select line which denotes the start of a transaction. Since assertion of the SS(N) line denotes the start of a transfer, it must be de-asserted and reasserted for sequential byte transfers to the same slave device.

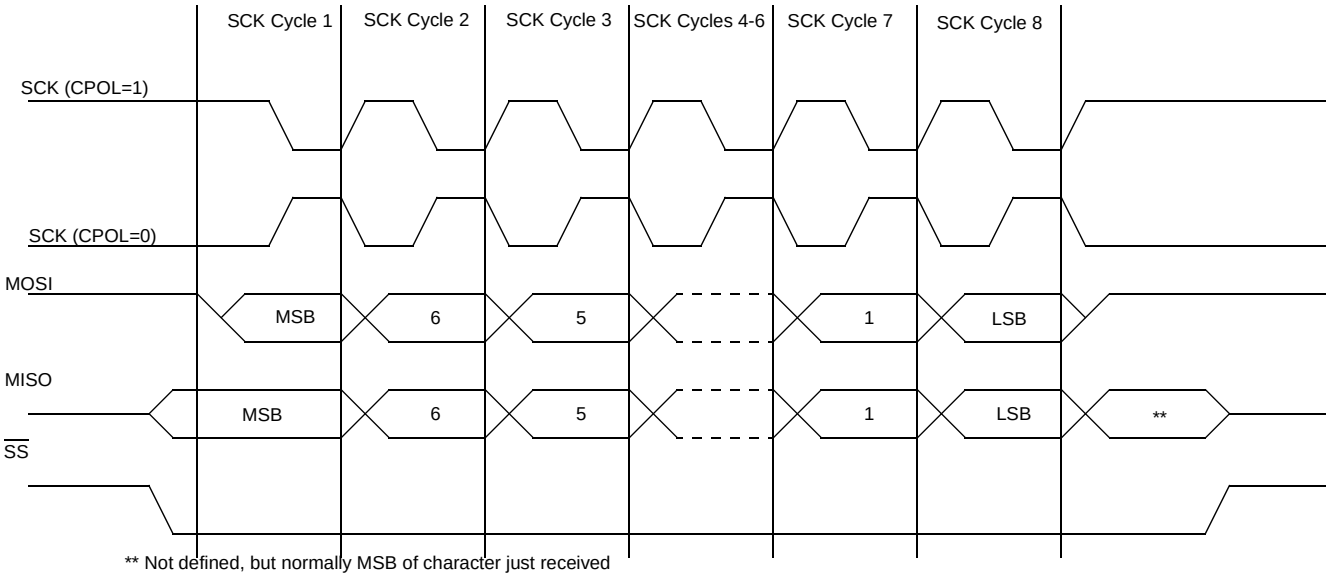


Figure 3: **Data Transfer on the SPI Bus with CPHA=0 and CR(24)=0 for 8-bit character.**

Figure 4 shows the timing diagram for an SPI transfer when the clock phase, CPHA, is set to '1'. Waveforms are shown for both positive and negative clock polarities of SCK. The first SCK cycle begins with a transition of SCK-signal from its idle state and this denotes the start of the data transfer. Because the clock transition from idle denotes the start of a transfer, the 68HC11 spec notes that $\overline{SS}$(N) line may remain active low between successive transfers. The spec text goes on to state that this format is useful in systems with a single master and single slave. In the context of the 68HC11spec, transmit data is placed directly in the shift register upon a write to the transmit register. Hence, it is the user's responsibility to insure that the data is properly loaded in the slave shift register prior to the first SCK edge. Recall that in this implementation the transmit register is double-buffered with the shift register, therefore, additional logic would be required to monitor the transmit register and asynchronously load the shift register (asynchronously in the sense of SCK) prior to the first shift whenever a write to the transmit register was performed.

In this implementation, it was chosen instead to toggle the SS-signal for all CPHA configurations and not support permitting SPISEL being held low. It is required that all $\overline{SS}$-signals be routed between SPI devices internally to the FPGA. The result of toggling of the $\overline{SS}$-signal is a minimization of FPGA resources.

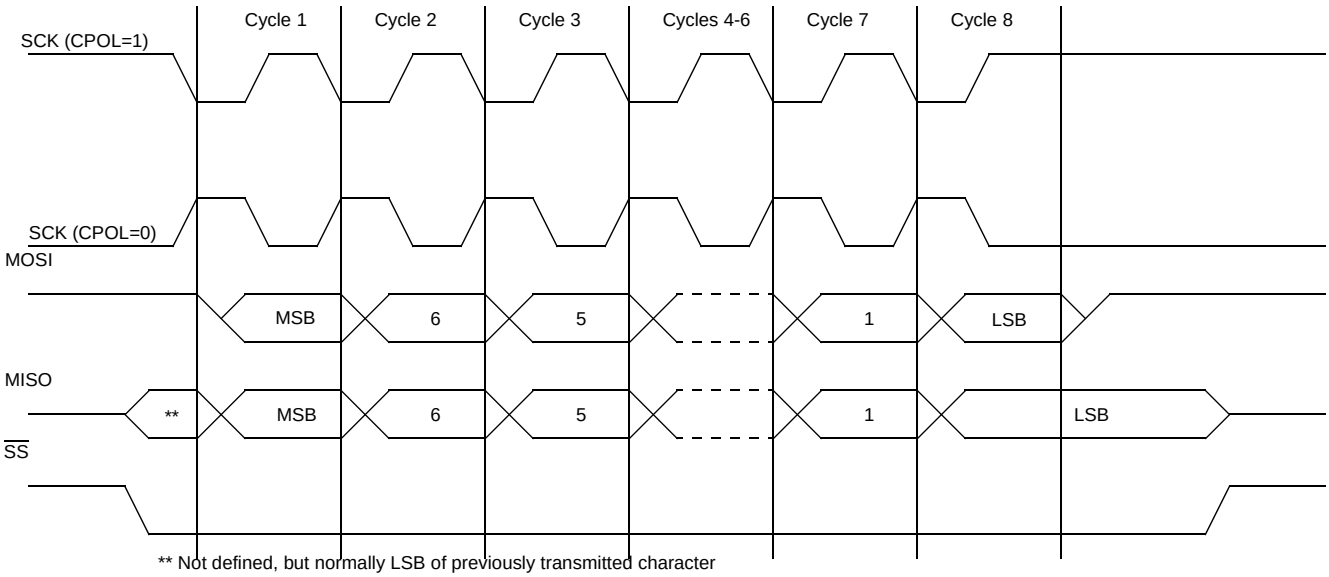


Figure 4: **Data Transfer on SPI Bus with CPHA=1 and CR(24)=0 for 8-bit character.**

All SPI transfers are full-duplex where an 8-bit data character is transferred from the master to the slave and an independent 8-bit data character is transferred from the slave to the master. This can be viewed as an 8-bit shift register in the SPI Master device and another 8-bit shift register in a SPI slave device that are connected as a circular 16-bit shift register.

The SPI specification details the timing and waveforms for byte transfers where the msb is shifted out first on the SPI bus, but does not dictate the data content or sequence of data in the sense of address information or other information type. All data written to the transmit register will be transmitted on the SPI bus, and all data received on the SPI bus will be stored in a receive register for the user logic to interpret.

Transfer Beginning Period

The definition of the transfer beginning period is taken directly from the M68HC11 Reference Manual; this manual can be referenced for more details that are not reproduced herein. All SPI transfers are started and controlled by a master SPI device.

As a slave, the processor considers a transfer to begin with the first SCK edge (CPHA=1)or the falling edge of $\overline{SS}$(N) (CPHA=0). In either CPHA format, a transfer can be aborted by taking the $\overline{SS}$(N) signal high, which causes the SPI slave logic and bit counters to be reset. In this implementation, the software driver can deselect all slaves (i.e. SS(N) is all ones) to abort a transaction, and it is, by design, the software responsibility to inhibit the user from changing slaves during a SPI single transfer or buffer transfer. However, the hardware does allow such a change in slave select.

Recall that in this implementation, the transmit register is double-buffered with the shift register. Furthermore it is required that SS be asserted in all modes. In slave configuration, the data is transmitted from the transmit register on the first OPB rising clock edge following SS-signal being asserted if data is available in the register or FIFO. If data is not available, then the under-run interrupt is asserted.

Transfer Ending Period

The definition of the transfer ending period is taken directly from the M68HC11 Reference Manual; this manual can be referenced for more details which are not reproduced herein. As stated in the manual, a SPI transfer is technically complete when the SPIF flag is set, but depending on the configuration of the SPI system, there may be additional tasks to be

performed before the system can consider the transfer complete. In this VHDL implementation and when configured without FIFOs, the equivalent of the SPIF bit is the bit 30 in the status register which is the Rc_Full bit. Setting of this bit denotes the end of a transfer and that data is available in the receive register. In this configuration, bit-27 of the interrupt register, which is the Data Receive Register Full interrupt is asserted as well. The data received is clocked in the receive register on the same clock edge as interrupt(27) being asserted.

When the SPI device is configured as a master without FIFOs, status register bits 31 and 28 are cleared, status register bits 29 and 30 are set, and interrupt bits 27 and 30 are set on the first rising OPB clock edge after at the end of the eighth SCK cycle. Note that the end of the eighth SCK cycle is a transition on SCK for CPHA=0, but is not denoted by a transition on SCK for CPHA=1. However, the master internal clock provides this SCK edge which prompts the setting/clearing of the bits noted.

When the SPI device is configured as a slave, setting/clearing of the bits discussed above for a master coincides with the setting/clearing of the master bits for both cases of CPHA=0 and CPHA=1. Recall that for CPHA=1 (i.e. no SCK edge denoting the end of the eighth clock period) the slave has no way of knowing when the end of the eighth SCK period occurs unless an OPB clock period counter was included in the SPI slave device. In this design, a counter was implemented which permits the simultaneous setting of status bits and interrupts for both master and slave SPI devices. It is noted that in the case of an external SPI slave device, SCK can be asynchronous with the internal clock of the external slave device, hence, this vhdl design cannot be used with external slaves that do not utilize the OPB clock.

When the SPI assembly is configured with FIFOs and a series of consecutive SPI 8-bit character transfers are performed, status bits and interrupts do indicate completion of the first SPI and the last. The only way to monitor when intermediate transfers are completed is to monitor the receive FIFO occupancy number. There is also an interrupt when the transmit FIFO is less than half full. Complete details on interrupts is discussed in a later section.

Optional FIFOs

One option to the system designer via Platform Generator is to include FIFOs in the SPI assembly as shown in Figure 1. Since SPI is full-duplex both transmit and receive FIFOs are instantiated as a pair. When FIFOs are implemented, the slave select address is required to be the same for all data buffered in the FIFOs. This is required because a FIFO for the slave select address is not implemented. Both transmit and receive FIFOS are 16 bytes deep and are accessed via single OPB transactions since burst mode is not supported.

The transmit FIFO is write-only. When data is written in the FIFO, the occupancy number is incremented and when a SPI transfer is completed, the number is decremented. As a consequence of this operation, aborted SPI transfers still has the data available for a retry of the transmission. The occupancy number is a read-only register. If a write of data is attempted when the FIFO is full, then no acknowledgement is given and a bus timeout will occur. Interrupts associated with the transmit FIFO include Data Transmit FIFO Empty, Transmit FIFO Half Empty, and Transmit FIFO Under-run. See the later section on interrupts for details.

The receive FIFO is read-only. When data is read from the FIFO, the occupancy number is decremented and when a SPI transfer is completed, the number is incremented. If a read is attempted when the FIFO is empty, then no acknowledgement is given and a bus timeout will occur. Data is automatically written to the FIFO from the SPI module shift register after the completion of a SPI transfer. If the receive FIFO is full and more data is received, then a Receive FIFO Overflow interrupt is issued. When this happens, all data attempted to be written to the full receive FIFO by the SPI module is lost. The other interrupt associated with the receive FIFO is the Receive FIFO Full interrupt.

SPI transfers, when the SPI assembly is configured with FIFOs, can be started in two different ways depending on when the enable bit in the control register is set. If the enable bit is set prior to the first data being loaded in the FIFO, then the SPI transfer begins immediately after the write to the master transmit FIFO. If the FIFO is emptied via SPI transfers before additional bytes are written to the transmit FIFO, an interrupt will be asserted. When the OPB-SPI SCK frequency ratio is sufficiently small, this scenario is highly probable. Alternatively, the FIFO can

be load with up to 16 bytes and then the enable bit can be set which starts the SPI transfer. In this case, an interrupt is issued after all bytes are transferred. In all cases, more data can be written to the transmit FIFOs to increase the number of bytes transferred before emptying the FIFOs.

Local Master Loopback Mode

Local master loopback mode, although not included in the 68HC11 Reference Manual, has been implemented to expedite testing. When this mode is selected via setting the Loop bit in the control register, the transmitter output is internally connected to the receiver input. The receiver and transmitter operate normally, except that received data (from remote slave) is ignored. This mode is meaningful only when the SPI device is configured as a master.

Hardware Error Detection

The SPI architecture relies mainly on software controlled bus arbitration for multi-master configurations to avoid conflicts and errors, but limited error detection is implemented in the SPI hardware. The first error detection mechanism in hardware to be discussed is contention error detection that detects when an SPI device configured as a master is selected (i.e. its $\overline{SS}$-bit is asserted) by another SPI device simultaneously configured as master. As noted before, the master being selected as a slave immediately drives its outputs as necessary to avoid hardware damage due to simultaneous drive contention. Simultaneously with driving outputs to a safe condition, the master sets the mode-fault error (MODF) bit in the status register. This bit is automatically cleared by reading the status register. Following a MODF error, the master must be disabled and re-enabled with correct data. This may require clearing the FIFOs when configured with FIFOs.

A similar error detection mechanism in hardware has been implemented for SPI slave devices. The error detected is when a SPI device configured as a slave but is not enabled and is selected (i.e. its $\overline{SS}$-bit is asserted) by another SPI device. When this condition is detected, interrupt bit 30 is set by a strobe to the interrupt register if the interrupt module is included in the assembly. This error detection does not exist if the interrupt module is not included in the SPI assembly.

Under-run and over-run conditions error detection is provided as well. Under-run conditions can happen only in slave mode of operation, where a master commands a transfer but the slave does not have data in the transmit register or FIFO for transfer. When a such a request is made, the slave under-run interrupt is asserted and the slave shift register is loaded with all zeros for transmission. Over-run can happen to both master and slave devices where a transfer occurs when the receive register or FIFO is full. When such a transfer occurs, the data received in that transfer is not registered (i.e. it is lost) and the over-run interrupt bit-26 is asserted.

Software Freeze Command Operation

The software freeze command impacts only master operation and not slave operation. When a freeze command is asserted (i.e. Freeze signal goes high), the master completes any transfer in progress, but does not initiate a subsequent SPI transfer until the freeze signal is pulled low. Operation is identical to as if the SPI master transmit register/FIFO is empty when the Freeze signal is asserted.

SPI Protocol with Manual Slave Select Assertion

This section briefly describes the SPI protocol where Slave Select (SS(N)) is manually asserted by the user (i.e CR(24) = 1). This is the default configuration. The mode is provided to permit transfers of an arbitrary number of bytes without toggling Slave Select until all the bytes are transferred. In this mode where CR(24) = 1, the data in the Slave Select Register appears directly on the SS(N) output. As described earlier SCK must be stable before assertion of Slave Select, therefore, when manual slave select mode is utilized, the SPI master must be enabled first (CR(24) = 1) to assert SCK to the idle state prior to asserting Slave Select. Note that the Master Transfer Inhibit bit (CR(23)) can be utilized to inhibit Master transactions until the Slave Select is asserted manually and all data registers are initialized as desired.

When the above rules are followed, the timing is essentially as presented for the automatic Slave Select assertion with the exception that assertion of Slave Select is under the user control and the number of bytes transferred is controlled by the user. Note that the Master Transfer Inhibit can be asserted to permit loading of registers or FIFOs as needed. This can be utilized before the first transaction and after any transaction that is allowed to complete.

SPI Configuration Parameters

Table 1 lists parameters required to be defined in configuring the SPI assembly via Platform Generator. Although the OPB data bus is currently 32-bits and planned to go to only 64-bits, the IPIF in the SPI assembly is parameterized to allow data bus widths less than 32-bits and up to 16 bytes. This will permit use of the SPI attachment in other applications beyond CoreConnect related applications.

Table 1: **Parameters to Configure the SPI Assembly**

Group	# Label	Feature	Parameter (generic) Name	Allowed Values	Default value to GUI	Constraint and VHDL type
OPB	G1	Platform Builder assigned device ID number	C_DEV_BLK_ID	See Platform Builder specification	4	type: integer
	G2	Enable/Disable Model ID register	C_DEV_MIR_ENABLE	non-zero = included; zero = not included	0	type: integer
	G3	Base address for assembly (IPIF and SPI module)	C_BASEADDR	Vector of length C_OPB_AWIDTH	None	type: std_logic_vector
	G4	Permits alias of address space by making greater than X7F	C_HIGHADDR	C_BASEADDR + any 2**n-1 value greater than X7F	C_BASEADDR + X7F	type: std_logic_vector Default value in GUI assumes C_IP_REG_BAR_OFFSET = X60
	G5	Include interrupt module required for multiple interrupt conditions	C_INTERRUPT_PRESENT	non-zero = included; zero = not included	Set to 1 Disabled	type: integer (Drivers require this to be 1; see text); Possible future option
	G6	OPB address bus width	C_OPB_AWIDTH		32	type: integer Includes bits for byte addressing
	G7	OPB databus width	C_OPB_DWIDTH		32	type: integer
SPI	G8	Both receive and transmit FIFOs	C_FIFO_EXIST	non-zero = included; zero = not included	1 (i.e. included)	type: integer

Table 1: **Parameters to Configure the SPI Assembly**

Group	# Label	Feature	Parameter (generic) Name	Allowed Values	Default value to GUI	Constraint and VHDL type
	G9	OPB to SPI SCK frequencies ratio	C_OPB_SCK_RATIO	2, 4, 16, 32, NX16 for N=1,2,3,...,128	2	type: integer
	G10	Slave-only mode	C_SPI_SLAVE_ONLY	non-zero = included; zero = not included	Set to 0 Disabled	type: integer; Not implemented at this time.
	G11	Number of off-chip Slave Select bits in SS-vector	C_NUM_OFFCHIP_SS_BITS	0 to the Number of bits in OPB databus	Set to 0 Disabled	type: integer; Must be less than or equal to C_NUM_SS_BITS.
	G12	Total number of Slave Select bits in SS-vector	C_NUM_SS_BITS	1 up to the number of bits in OPB databus	1	type: integer; Must be less than or equal to the number of bits in the OPB databus.

SPI Assembly I/O Signals

The I/O signals for the SPI Assembly are listed in Figure 2. The interfaces referenced in this table are shown in Figure 1

Table 2: **SPI Assembly I/O Signals**

Group	# label	Signal Name	External Interface	I/O	Signal Description
System	P1	OPB_Rst	System	I	Reset signal
	P2	IP2INTC_Irpt	Interrupt controller	O	Interrupt signal to interrupt controller
	P3	Freeze	System	I	Software freeze command signal
OPB	P4	OPB_Clk	OPB	I	OPB Bus clock
	P5	OPB_select	OPB	I	Select signal from OPB
	P6	OPB_RNW	OPB	I	OPB read/write
	P7	OPB_seqAddr	OPB	I	OPB sequential address
	P8	OPB_BE	OPB	I	OPB byte enable
	P9	OPB_ABus	OPB	I	OPB address bus
	P10	OPB_DBus	OPB	I	OPB data bus
	P11	SPI_DBus	OPB OR-logic	O	Data output bus (gated)
	P12	SPI_xferAck	OPB OR-logic	O	Attachment transfer acknowledgement

Table 2: **SPI Assembly I/O Signals**

Group	# label	Signal Name	External Interface	I/O	Signal Description
	P13	SPI_errAck	OPB OR-logic	O	Attachment bus error (tied low)
	P14	SPI_toutSup	OPB OR-logic	O	Attachment time-out suppress (tied low)
	P15	SPI_retry	OPB OR-logic	O	Attachment retry (tied low)
SPI	P16	SCK_I	All SPI devices	I	SPI Bus Clock Input
	P17	SCK_O	All SPI devices	O	SPI Bus Clock Output
	P18	SCK_T	All SPI devices	O	SPI Bus Clock 3-state Enable (3-state when high)
	P19	MOSI_I	All SPI devices	I	Master out, Slave in Input
	P20	MOSI_O	All SPI devices	O	Master out, Slave in Output
	P21	MOSI_T	All SPI devices	O	Master out, Slave in 3-state Enable (3-state when high)
	P22	MISO_I	All SPI devices	I	Master in, slave out Input
	P23	MISO_O	All SPI devices	O	Master in, slave out Output
	P24	MISO_T	All SPI devices	O	Master in, slave out 3-state Enable (3-state when high)
	P25	SPISEL	All SPI master devices	I	Local SPI slave select active low input
	P26	$\overline{SS}$_I	None--Included for tool requirements	I	Input of slave select vector of length N Input where there are N SPI devices, but not connected
	P27	$\overline{SS}$_O	Each bit is interfaced to a SPI device	O	One-hot encoded, active low slave select vector of length N Output
	P28	$\overline{SS}$_T	Each bit is interfaced to a SPI device	O	Single 3-state control signal for slave select vector of length N (3-state when high)

Port and Parameter Dependencies

Table 3 lists dependencies of ports and parameters on each parameter.

Table 3: **Port and Parameter Dependencies for Slave Attachment**

Group	# label	Parameter (generic) name	Affects	Depends	Relationship Description
OPB	G1	C_DEV_BLK_ID	None	None	
	G2	C_DEV_MIR_ENABLE	None	None	
	G3	C_BASEADDR	None	G6	Must be of length C_OPB_AWIDTH where lower order bits are zeros. The number of lower order bits that are zero is equal to the number of continuous lower order bits in C_HIGHADDR that are non-zero and not equal to the same bit in C_BASEADDR.
	G4	C_HIGHADDR	None	G3, G6	Included at request of Platform Generator Designers
	G5	C_INTERRUPT_PRESENT	None	None	Software drivers available only when present (i.e. 1)
	G6	C_OPB_AWIDTH	P9,G3-4	None	Sets OPB address bus interface width
	G7	C_OPB_DWIDTH	P8,P10	None	Sets OPB data bus and byte enable interface widths
SPI	G8	C_FIFO_EXIST	None	None	
	G9	C_OPB_SCK_RATIO	P16	None	Determines OPB to SCK frequency ratio
	G10	C_SPI_SLAVE_ONLY	P16-18, P20	None	Not implemented at this time, but will be in the future.
	G11	C_NUM_OFFCHIP_SS_BITS	Number of pins assigned	G12	Defines the number of bits in the $\overline{SS}$-vector that go off-chip (i.e. off-chip SPI slave devices). Must be less than or equal to C_NUM_SS_BITS
	G12	C_NUM_SS_BITS	P26-28	None	Defines the number of bits in the $\overline{SS}$-vector.

SPI Register Descriptions

The SPI assembly contains addressable registers for read/write operations as shown in Table 4. The base address is set by the parameter C_BASEADDR and all SPI register addresses are calculated by an offset from C_BASEADDR. The first SPI register is at offset C_IP_REG_BAR_OFFSET.

The IPIF also contains the interrupt registers and IP reset module which are both optional registers. Addresses of both register sets are calculated by an offset from C_BASEADDR.

All bit indices are relative the OPB bus. The highest bit index is the lsb. In this document, bit assignment will be made assuming a 32-bit OPB; assignment for either wider or narrower buses follows the same convention.

Table 4 shows addresses of all the SPI registers, the IPIF interrupt registers and the IP Reset module. The transmit FIFO occupancy and Receive FIFO occupancy only exist when the SPI assembly is configured via parameters to include FIFOs. Note that the transmit and receive registers have independent addresses which differs from the 68HC11 specification where the same address is to be assigned to the two independent registers.

Table 4: **SPI Assembly Registers and Offset from BAR**

Register Name	OPB Address	Access
Interrupt Global Enable Register	C_BASEADDR + **0X1C**	Read/Write
Interrupt Register	C_BASEADDR + **0X20**	Read/Write "1" to clear
Interrupt Enable Register	C_BASEADDR + **0X28**	Read/Write
Reset Module	C_BASEADDR + **0X40**	Read/Write
Control Register (CR)	C_BASEADDR + 0X060	Read/Write
Status Register (SR)	C_BASEADDR + 0X064	Read
Data Transmit Register (DTR) whether it is a single register or the transmit FIFO	C_BASEADDR + 0X068	Write
Data Receive Register (DRR) whether it is a single register or the receive FIFO	C_BASEADDR + 0X06**C**	Read
Slave Select Register (SSR)	C_BASEADDR + 0X070	Read/Write
Transmit FIFO Occupancy	C_BASEADDR + 0X074	Read Does not exist if FIFOs are not present
Receive FIFO Occupancy	C_BASEADDR + 0X07**8**	Read Does not exist if FIFOs are not present

SPI Interrupt Registers

Interrupt Module Specifications

The interrupt registers are in the interrupt model which is instantiated in the IPIF module of the SPI assembly.

The SPI assembly has the option to include the interrupt module which permits multiple conditions for interrupt or to not include the module and have an interrupt strobe occur upon only the completion of a transfer. Of course, all status register bits are available for detailed information independent of the interrupt choice. The reason for this option is that the interrupt module utilizes about 25 LUTs that can be eliminated if multiple interrupt functionality is

sacrificed. This option was implemented prior to optimizing the interrupt controller which, at the time, yielded much greater savings. If the interrupt module is not implemented, then the single interrupt condition is when the receive register or FIFO is full. At this time, software will not be developed utilizing the single interrupt scheme and it will be the responsibility of the user to provide software.

Interrupt Global Enable Descriptions

A global enable is provided, to globally enable, or of more utility, globally disable interrupts from the SPI device. This bit is essentially ANDed with the input to the interrupt controller. Bit assignment is shown in Table 6. Unlike most other registers, this bit is the MSB on the OPB. This bit is read/write and cleared upon reset.

Table 5: **Interrupt Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
0	Interrupt Global Enable	Read/Write	0	**Interrupt Global Enable-** OPB bit (0) is the Interrupt Global Enable bit. Enables all individually enabled interrupts to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
1-31		Read	0	**Unassigned-**

Interrupt Signal Descriptions

When the interrupt module is included, up to seven unique interrupt conditions are possible depending upon whether the system is configured with FIFOs or not. A system without FIFOs has 6 interrupts that are sent to the IPIF interrupt module where some are redundant as discussed below. The IPIF interrupt module has a register that can enable each interrupt independently. Bit assignment in the Interrupt register for a 32-bit databus is shown in Table 6. The interrupt register is read-only and bits are cleared by writing a '1' to the bit(s) being cleared. All bits are cleared upon reset.

Table 6: **Interrupt Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
31	MODF	Read/Write '1' to clear	0	**MODF-** Interrupt(31) is the Mode-Fault Error Flag- This interrupt is generated if the $\overline{SS}$ signal goes active while the SPI device is configured as a master. This bit is set immediately by upon $\overline{SS}$ going active by a transition of the MODF status bit from low to high.
30	Slv_MODF	Read/Write '1' to clear	0	**Slv_MODF-** Interrupt(30) is the slave Mode-Fault Error Flag- This interrupt is generated if the $\overline{SS}$ signal goes active while the SPI device is configured as a slave but is not enabled. This bit is set immediately upon $\overline{SS}$ going active and continually set if $\overline{SS}$ is active and the device is not enabled

Table 6: **Interrupt Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
29	DTR Empty	Read/Write '1' to clear	0	**Data Transmit Register (FIFO) Empty-** Interrupt(29) is the Data Transmit Register(FIFO) Empty interrupt. Without FIFOs, this bit is set at the end of a SPI byte transfer by a one-clock period strobe to the interrupt register. With FIFOs, this bit is set at the end of the SPI byte transfer when the transmit FIFO is emptied by a one-clock period strobe to the interrupt register. See section on definition of end of transfer. In the context of the 68HC11 Reference Manual, when configured without FIFOs, this interrupt is equivalent in information content to the complement of SPI transfer complete flag ($\overline{\text{SPIF}}$) interrupt bit.
28	DTR Under-run	Read/Write '1' to clear	0	**Transmit Register/FIFO Under-run-** Interrupt(28) is the Transmit Register/FIFO under-run interrupt. This bit is set by a one-clock period strobe to the interrupt register when data is request from an "empty" transmit register/FIFO by the SPI state machine in order to perform a SPI transfer. This can occur only when the SPI device is in slave mode. All zeros are loaded in the shift register and transmitted by the slave in an under-run condition.
27	DRR Full	Read/Write '1' to clear	0	**Data Receive Register/ FIFO Full-** Interrupt(27) is the Data Receive Register Full interrupt. Without FIFOs, this bit is set at the end of a SPI byte transfer by a one-clock period strobe to the interrupt register. With FIFOs, this bit is set at the end of the SPI byte transfer when the receive FIFO has been filled by a one-clock period strobe to the interrupt register.
26	DRR Over-run	Read/Write '1' to clear	0	**Receive Register/FIFO Over-run-** Interrupt(26) is the Receive FIFO over-run interrupt. This bit is set by a one-clock period strobe to the interrupt register when an attempt to write data to a full receive register or FIFO is made by the SPI state machine in order to complete a SPI transfer. This can occur when the SPI device is in either master or slave mode.
25	Tx FIFO Near Empty	Read/Write '1' to clear	0	**Transmit FIFO Less Than or Equal to Half Empty-** Interrupt(25) is the Transmit FIFO near empty interrupt. This bit is set by a one-clock period strobe to the interrupt register when the occupancy value is decremented from '1000' to '0111'. Note that "0111' means there are 8 characters in the FIFO to be transmitted. This interrupt exists only if the SPI Assembly is configured with FIFOs.

Table 6: **Interrupt Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
24		Read	0	**Unassigned-**

Interrupt Enable Descriptions

The SPI assembly has interrupt enable features. Bit assignment in the Interrupt enable register is shown in Table 7. The interrupt enable register is read/write. All bits are cleared upon reset.

Table 7: **Interrupt Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
31	MODF set enable	Read/Write	0	**Mode-Fault Error Flag Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
30	Slv_MODF enable	Read/Write	0	**Slv_MODF Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
29	DTR Empty enable	Read/Write	0	**Data Transmit Register (FIFO) Empty Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
28	DTR Under-run enable	Read/Write	0	**Transmit FIFO Under-run Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
27	DRR Full enable	Read/Write	0	**Data Receive Register Full/Receive Full FIFO Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
26	DRR Over-run enable	Read/Write	0	**Receive FIFO Over-run Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
25	Tx FIFO Near Empty enable	Read/Write	0	**Transmit FIFO Less Than or Equal to Half Empty Enable-** Enables this interrupt to be passed to the interrupt controller. • '0' - Not enabled. • '1' - Enabled.
24		Read/Write	0	**Unassigned-**

SPI Assembly Reset Descriptions

The IPIF module in the SPI assembly has instantiated in it the IP reset module. This module permits software reset of the SPI module independent of other modules in the system and has another register for test purposes.

SPI Control Register (CR)

Bit assignment in the SPI Control Register is shown in Table 8. Bit assignment was made to follow the assignment pattern of Xilinx IPIF specifications and, when possible, follow the 68HC11 assignment pattern.

Table 8: **SPI Control Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Core Access	Reset Value	Description
31	LOOP	Read/Write	0	**Local loopback mode-**Enables local loopback operation and is functional only in master mode. • ’0’ - Normal operation. • ’1’ - Loopback mode. The transmitter output is internally connected to the receiver input. The receiver and transmitter operate normally, except that received data (from remote slave) is ignored. Note that the interrupt enable bit which resides in the this bit position of the 68HC11 specification resides in the interrupt enable register in this implementation; see exceptions.
30	SPE	Read/Write	0	**SPI System Enable select bit-**Setting this bit high enables the SPI devices as noted below. • ’0’ - SPI System OFF. Both master and slave outputs are in "3-state" and slave inputs ignored. • ’1’ - SPI System ON. Master outputs active (e.g. MOSI and SCK in idle state) and slave outputs will become active if $\overline{SS}$ becomes asserted. Master will start transfer when transmit data is available.
29	MSTR	Read/Write if master-slave option is implemented .Read Only if slave-only option is implemented	0	**MSTR-** Setting this bit configures the SPI device as a slave or master if that option is implemented via parameters. If slave-only option is implemented, then this bit is fixed to ’0’. • ’0’ - Slave configuration. • ’1’ - Master configuration.
28	CPOL	Read/Write	0	**Clock Polarity select bit-**Setting this bit defines clock polarity. • ’0’ - Active high clock; SCK idles low. • ’1’ - Active low clock; SCK idles high.

Table 8: **SPI Control Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Core Access	Reset Value	Description
27	CPHA	Read/Write	0	**Clock Phase select bit-**Setting this bit selects one of two fundamentally different transfer formats. See timing diagrams and discussion of diagrams.
26	Tx FIFO Reset	Read/Write	0	**Transmit FIFO Reset.** This bit forces a reset of the FIFO pointer and asserts FIFO empty flag. FIFO contents are unchanged. This bit is reset automatically reset to '0' one OPB clock period after set to '1'. This bit is unassigned when the SPI assembly in not configured with FIFOs. • '0' - Transmit FIFO normal operation • '1' - Reset transmit FIFO pointer
25	Rx FIFO Reset	Read/Write	0	**Receive FIFO Reset.** This bit forces a reset of the FIFO pointer and asserts FIFO empty flag. FIFO contents are unchanged. This bit is reset automatically reset to '0' one OPB clock period after set to '1'. This bit is unassigned when the SPI assembly in not configured with FIFOs. • '0' - Transmit FIFO normal operation • '1' - Reset receive FIFO pointer
24	Manual $\overline{SS}$ Assertion Enable	Read/Write	1	• **Manual $\overline{SS}$ Assertion Enable-** This bit forces the data in the Slave Select register to be asserted on the $\overline{SS}$ output anytime the device is configured as a master and the device is enabled (SPE asserted). This bit has no effect on slave operation. • '0' - $\overline{SS}$ assertion by Master state machine • '1' - $\overline{SS}$ follows data in Slave Select register
23	Master Transaction Inhibit	Read/Write	1	**Master Transaction Inhibit-** This bit inhibits Master transactions in the same way Freeze functions. This bit has no effect on slave operation. • '0' - Master transactions enabled • '1' - Master transactions inhibited

SPI Status Register (SR)

Bit assignment in the SPI Status Register is shown in Table 9 Bit assignment was made to follow the assignment pattern of Xilinx IPIF specifications and, when possible, follow the 68HC11 assignment pattern. The status register is read-only. Note reset default values.

Table 9: **SPI Status Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
31	Rc_Empty	Read	1	**Receive Empty-** When a receive FIFO exists, this bit will be set high when the receive FIFO is empty; the occupancy of the FIFO is decremented with each FIFO read operation. When FIFOs don't exist, this bit is set high when the receive register has been read. This bit is cleared at the end of a successful SPI transfer.
30	Rc_Full	Read	0	**Receive Full-** When a receive FIFO exists this bit will be set high when the receive FIFO is full; the occupancy of the FIFO is incremented with the completion of each SPI transaction. When FIFOs don't exist, this bit is set high when an SPI transfer has completed. When FIFOs don't exist Rc_Empty and Rc_Full are complements.
29	Tx_Empty	Read	1	**Transmit Empty-** When a transmit FIFO exists, this bit will be set high when the transmit FIFO is empty; the occupancy of the FIFO is decremented with the completion of each SPI transfer. When FIFOs don't exist, this bit is cleared with the completion of a SPI transfer. Either with or without FIFOs, this bit is cleared upon an OPB write to the FIFO or transmit register.
28	Tx_Full	Read	0	**Transmit Full-** When a transmit FIFO exists this bit will be set high when the transmit FIFO is full. When FIFOs don't exist, this bit is set high when an OPB write to the register has been made and it is cleared when the SPI transfer has completed.
27	MODF	Read	0	**Mode-Fault Error Flag-** This flag is set if the SS signal goes active while the SPI device is configured as a master. MODF is automatically cleared by reading the SR. MODF does generate an interrupt with a single-cycle strobe when the MODF bit transitions from a low to a high. • '0' - No error. • '1' - Error condition detected.
26		Read	0	**Unassigned-**
25		Read	0	**Unassigned-**
24		Read	0	**Unassigned-**

Data Transmit Register (DTR)

This register is write only and contains data to be transmitted on the SPI bus. It is double buffered with the shift register. The data is transferred from the register to the shift register following enable bit being set high in master mode or following SPISEL being active in slave mode. If a transfer is in progress, the data in the DTR is loaded in the shift register as soon as the data in the shift register is transferred to the receive register (DRR) and a new transfer starts. The data in the DTR is held in the DTR until a subsequent write overwrites the data. Table 10 shows specifics of the data format.

When a Transmit FIFO exists, data is written directly in the FIFO and the register is simply in the FIFO. The pointer is decremented at the time of completion of each SPI transfer.

The hardware that forwards data from the register or FIFO to shift register will never cause a write collision error. Attempting to write to a full register or FIFO will not result in a write acknowledgement for the OPB transaction, but rather an OPB timeout.

Table 10: **SPI Data Transmit Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
24-31	D0 - D7	Write Only	0x00	SPI Transmit Data

Data Receive Register (DRR)

This double buffer receive register contains the data received from the SPI bus. The received data is placed in this register after each complete transfer if the register is empty. The SPI architecture does not provide a means for a slave to throttle traffic on the bus; consequently, the DRR is update following each completed transaction only if the DRR was read prior to the last SPI transfer. If the DRR was not read (i.e. is full), then the most recently transferred data will be lost and a receive over-run interrupt will occur. The same condition can occur with a master SPI device as well. For both master and slave SPI devices with a receive FIFO, the data is buffered in the FIFO. If a SPI transfers occur with the FIFO full, then the most recently transferred data will be lost and a receive over-run interrupt will occur. The receive FIFO is read only. If an attempt to read an empty receive register or FIFO is made, then an OPB timeout error will occur because an acknowledgement will not be issued. Table 11 shows specifics of the data format.

Table 11: **SPI Data Receive Register Bit Definitions (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
24-31	D0 - D7	Read Only	0x00	SPI Receive Data

Slave Select Register (SSR)

This field contains an N-length vector that specifies the slave that the local master is to communicate with. This vector is a active-low, one-hot encoded vector ($\overline{SS}$(N)). Actual assignment of slaves to specific bits is performed by Platform Generator. This register is read/write Table 12 shows specifics of the data format. The index of $\overline{SS}$(N) increments in the opposite direction to that of the OPB bit index. OPB bit index 31 is bit zero of $\overline{SS}$(N), OPB bit index 30 is bit index 1 of $\overline{SS}$(N) and so on. The reason for reversing the order of index incrementing is considerations in software driver development.

Table 12: **SPI Slave Select Address Register Bits (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
(31-N+1) to 31	Slave Address	Read/ Write	all '1's	Active-low, one-hot encoded slave select vector of length N-bits. N must be less than or equal to the databus width. Note that $\overline{SS}$(N) increments in the opposite direction to that of the OPB bit index

Transmit FIFO Occupancy Register (Tx_FIFO_OCY)

This field contains the occupancy number greater than one for the Transmit FIFO when the SPI assembly is configured with FIFOs. The actual occupancy is the binary value plus 1. This register is read only and does not exist when the assembly is configured without FIFOs. The Transmit FIFO Empty Interrupt or Status Bit is the only reliable way to determine if the FIFO is empty; reading this register cannot be used to determine if the FIFO is empty. An attempt to write to this register does not yield an acknowledgement, but rather an OPB timeout. Table 13 shows specifics of the data format.

Table 13: **Transmit FIFO Occupancy Register Bits (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
24-27	Reserved	Read	0x0	Reserved.
28-31	Occupancy Value	Read		Bit 4 is the MSB. The binary value plus 1 yields the occupancy.

Receive FIFO Occupancy Register (Rc_FIFO_OCY)

This field contains the occupancy number greater than one for the Receive FIFO when the SPI assembly is configured with FIFOs. The actual occupancy is the binary value plus 1. This register is read only and does not exist when the assembly is configured without FIFOs. The Receive FIFO Empty Status Bit is the only reliable way to determine if the FIFO is empty; reading this register cannot be used to determine if the FIFO is empty. An attempt to write to this register does not yield an acknowledgement, but rather an OPB timeout. Table 14 shows specifics of the data format.

Table 14: **Receive FIFO Occupancy Register Bits (Bit assignment assumes 32-bit bus)**

Bit(s)	Name	Access	Reset Value	Description
24-27	Reserved	Read	0x0	Reserved
28-31	Occupancy Value	Read		Bit 4 is the MSB. The binary value plus 1 yields the occupancy.

Design Implementation

Target Technology

The intended target technology is a Virtex™ II FPGA.

Device Utilization and Performance Benchmarks

This section will be updated when the design has been completed. It will contain the resources and timing for various values of the parameters.

Since the SPI Assembly is a module that will be used with other design pieces in the FPGA, the utilization and timing numbers reported in this section are just estimates. As the SPI Assembly is combined with other FPGA designs, the utilization of FPGA resources and timing of the SPI Assembly will vary from the results reported here.

In order to analyze the SPI timing within the FPGA, a design was created that instantiated the SPI Assembly with registers on all of the SPI ports. This allowed a constraint to be placed on the clock net for the SPI Assembly to yield more realistic timing results. Using this method, the clock frequency register-to-register varied from 110-125 MHz depending on the actual place and route of the design in the FPGA.

The SPI Assembly benchmarks are shown in Table 15 for a VirtexII -5 FPGA.

Table 15: ***SPI Assembly*** **FPGA Performance and Resource Utilization Benchmarks (VirtexII -5)**

Parameter Values (For Example)				Device Resources			f_{MAX}	
C_GEN1	C_GEN2	C_GEN3	C_GEN4	Slices	Slice Flip-Flops	4-input LUTs	f_{MAX_CMB}	f_{MAX_REG}

Notes:
1. These benchmark designs contain only the IIC with registered inputs/outputs without any additional logic. Benchmark numbers approach the performance ceiling rather than representing performance under typical user conditions.

Device resource utilization can be estimated by the following formula:

TBD

Flow Description

This section provides information on setting the SPI registers to initiate and complete bus transactions.

SPI Master Device with or without FIFOs where the slave select vector is asserted manually via command register bit(24) assertion

This flow permits the transmittal of N-bytes in a single toggling of the slave select vector (default mode). Follow these steps to successfully complete an SPI transaction:

1. Start from proper state including SPI bus arbitration
2. Configure master interrupt enable registers as desired.
3. Configure target slave SPI device as required.
4. Write initial data to master transmit register/FIFO. This assumes that the SPI Master is disabled.
5. Insure Slave Select Register has all ones.

6. Write configuration data to master SPI device CR as desired including setting bit 24 for manual asserting of SS-vector and setting both enable bit and Master Transfer Inhibit bit. This initializes SCK and MOSI but inhibits transfer.
7. Write to Slave Select Register to manual assert of SS-vector.
8. Write the above configuration data to master SPI device CR, but clear inhibit bit which starts transfer.
9. Wait for interrupt (typically interrupt(30)) or poll status for completion. Wait time will depend on clock ratio.
10. Set Master Transaction Inhibit bit to service interrupt request. Writing new data to master register/FIFOs and slave device then clear Master Transaction Inhibit bit to continue N 8-bit character transfer. Note that an overrun of the receive register/FIFO can occur if the receive register/FIFOs are not read properly. Also note that SCK will have "stretched" idle levels between byte transfers (or groups of byte transfers if utilizing FIFOs) and that MOSI can transition at end of a byte transfer (or group of transfers) but will be stable at least one-half SCK period prior to sampling edge of SCK.
11. Repeat previous two steps until all data is transferred
12. Write all ones to slave select register or exit manual slave select assert mode to deassert SS-vector while SCK and MOSI are in the idle state.
13. Disable devices as desired.

SPI Master and Slave Devices without FIFOs performing one 8-bit transfers (optional mode)

Follow these steps to successfully complete an SPI transaction:

1. Start from proper state including SPI bus arbitration.
2. Configure master and slave interrupt enable registers as desired.
3. Write configuration data to master SPI device CR as required.
4. Write configuration data to slave SPI device CR as required.
5. Write the active-low, one-hot encoded slave select address to the master SS-register.
6. Write data to slave transmit register as required.
7. Write data to master transmit register to start transfer.
8. Wait for interrupt (typically interrupt(30)) or poll status for completion.
9. Read interrupt registers of both master and slave SPI devices as required.
10. Perform interrupt requests as required.
11. Read status registers of both master and slave SPI devices as required.
12. Perform actions as required or dictated by status register data.

SPI Master and Slave Devices where Registers/FIFOs are filled before SPI transfer is started and multiple discrete 8-bit transfers are transferred (optional mode)

Follow these steps to successfully complete an SPI transaction:

1. Start from proper state including SPI bus arbitration
2. Configure master and slave interrupt enable registers as desired.
3. Write configuration data to master SPI device CR as required.
4. Write configuration data to slave SPI device CR as required.

5. Write the active-low, one-hot encoded slave select address to the master SS-register.
6. Write all data to slave transmit Register/FIFO as required.
7. Write all data to master transmit Register/FIFO.
8. Write enable bit to master control register which starts transfer.
9. Wait for interrupt (typically interrupt(30)) or poll status for completion.
10. Read interrupt registers of both master and slave SPI devices as required.
11. Perform interrupt requests as required.
12. Read status registers of both master and slave SPI devices as required.
13. Perform actions as required or dictated by status register data.

SPI Master and Slave Devices with FIFOs where some initial data is written to FIFOs, the SPI transfer is started, data is written to the FIFOs as fast or faster than the SPI transfer and multiple discrete 8-bit transfers are transferred (optional mode)

Follow these steps to successfully complete an SPI transaction:

1. Start from proper state including SPI bus arbitration
2. Configure master and slave interrupt enable registers as desired.
3. Write configuration data to master SPI device CR as required.
4. Write configuration data to slave SPI device CR as required.
5. Write the active-low, one-hot encoded slave select address to the master SS-register.
6. Write initial data to slave transmit FIFO as required.
7. Write initial data to master transmit FIFO.
8. Write enable bit to master control register which starts transfer.
9. Continue writing data to both master and slave FIFOs.
10. Wait for interrupt (typically interrupt(30)) or poll status for completion.
11. Read interrupt registers of both master and slave SPI devices as required.
12. Perform interrupt requests as required.
13. Read status registers of both master and slave SPI devices as required.
14. Perform actions as required or dictated by status register data.

Platform Generator Considerations

Platform Generator is the tool that will allow processor systems to be configured using building blocks of IP. Based on the configuration of the system and the IP in the system, the Platform Generator tool will create the Configuration ROM (CROM) with information about each IP block and will set the parameters for each IP block based on the system configuration.

Certain system parameters will be input into Platform Generator such as the OPB clock frequency and the ratio of OPB to SCK frequencies (C_OPB_SCK_RATIO) which will affect system performance.

Platform Generator must also instantiate 3-state I/O pins for SCK, MOSI and MISO when the parameter C_NUM_OFFCHIP_SS_BITS is non-zero. In addition, a number of slave select bits (given by C_NUM_OFFCHIP_SS_BITS) must connected to I/0 for off-chip slave selection.

Specification Exceptions

Exceptions to the Motorola's M68HC11-Rev. 4.0 Reference Manual

A slave mode-fault error interrupt was added to provide an interrupt if a SPI device is configured as a slave and is selected when not enabled.

In this design, the data transmit and data receive registers have independent addresses. This is an exception to the 68HC11 specification which calls for the two registers to have the same address.

All $\overline{SS}$-signals are required to be routed between SPI devices internally to the FPGA. This is because toggling of the $\overline{SS}$-signal is utilized in slaves to minimize FPGA resources.

Manual control of the $\overline{SS}$-signals is provided by setting bit 24 in the command register. When this bit is set, the vector in the slave select register is asserted when the device is configured as a master and is enabled. When this mode is enabled, multiple bytes can be transferred without toggling the $\overline{SS}$-vector.

A control bit is provided to inhibit master transfers. This bit is effective in any master mode, but has main utility in manual control of the SS-signals.

In this implementation without FIFOs, both the transmit and receive register are double buffered. Hardware prevents data transfer from the transmit buffer to the shift register while an SPI transfer is in progress, consequently, the write collision error described in the MC68HC11 Reference Manual can not occur. In the 68HC11 implementation, the transmit register is transparent to the shift register which necessitates the write collision error (WCOL) detection hardware; however, it is not required or implemented in this implementation.

The interrupt enable bit (SPIE) defined by the 68HC11 specifications which resides in the 68HC11 control register has been moved to the interrupt bit-wise enable register

In the position of the SPIE bit is a bit to select local master loopback mode for testing. This is not specified in the 68HC11 specification, but is in the 8260 specification and it is supported in this implementation.

An option is implemented in the this FPGA design to implement FIFOs on both transmit and receive (Full Duplex only).

An option is implemented in this FPGA design to select slave-only mode; however, this option is not part of the 68HC11 specification. This was implemented to reduce FPGA resource required when slave-only operation is desired.

The baud rate generator is specified by Motorola to be programmable via bits in the control register; however, in this FPGA design the baud rate generator is programmable via parameters in the VHDL implementation. Furthermore, in addition to the prescribed ratios of 2, 4, 16 and 32, all integer multiples of 16 up to 2048 are allowed.

Reference Documents

The following documents contain reference information important to understanding the SPI design:

- [1] Motorola's M68HC11-Rev. 4.0 Reference Manual
- Motorola's MPC8260 PowerQUICC II™ User's Manual 4/1999 Rev. 0

OPB General Purpose Input/Output (GPIO) Specification

March 2002

Summary

This document describes the specifications for a general purpose input/output core for the OPB bus. This document applies to the following peripherals:

opb_gpio	v1.00a

Overview

The GPIO (General Purpose Input/Output) is a 32-bit peripheral that attaches to the OPB (On-chip Peripheral Bus), and has the following features:

Features

- OPB V2.0 bus interface with byte-enable support
- Supports 32-bit, 16-bit, and 8-bit bus interfaces
- Each GPIO bit dynamically programmable as input or output
- Number of GPIO bits configurable up to size of data bus interface
- Can be configured as inputs-only to reduce resource utilization

GPIO Organization

The GPIO is a simple peripheral consisting of two registers and a multiplexer for reading register contents and the GPIO I/O signals. The GPIO block diagram is shown in the following figure:

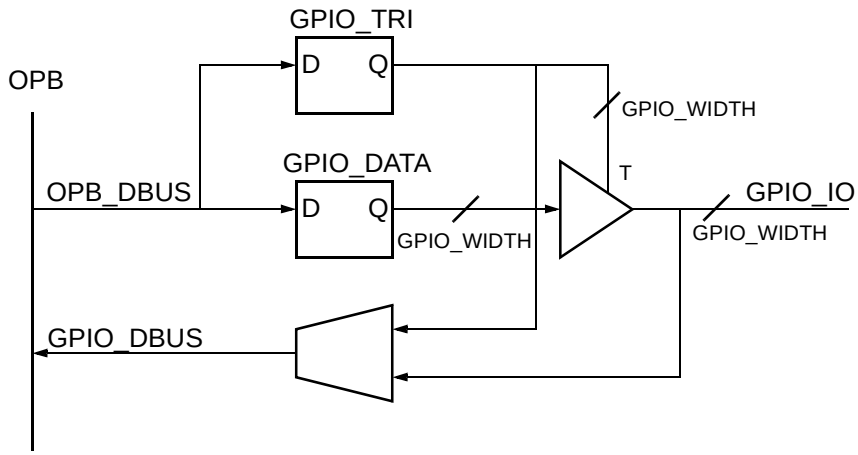


Figure 1: **GPIO Block Diagram**

Programming Model

Register Data Types and Organization

Registers in the GPIO are accessed as one of the following types:

- Byte (8 bits)
- Half word (2 bytes)
- Word (4 bytes)

Configuration

The following table shows GPIO configurations and access type.

Table 1: **GPIO Configuration and Access Type**

Configuration	Access Type
32-bit slave OPB peripheral	Word
16-bit peripheral	Half word
8-bit peripheral	Byte
32-bit, 16-bit, or 8-bit peripheral	All register accesses are on word boundaries to conform to the OPB-IPIF register location convention

The addresses of the GPIO registers when configured as a 32-bit OPB slave are shown in the following table:

Table 2: **GPIO Register Address Map (32-bit OPB)**

Register	Address (Hex)	Size	Type	Description
GPIO_DATA	0x00	Word	R/W	GPIO Data Register
GPIO_TRI	0x04	Word	R/W	GPIO Three-state Register

The GPIO registers are organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in the following figure:

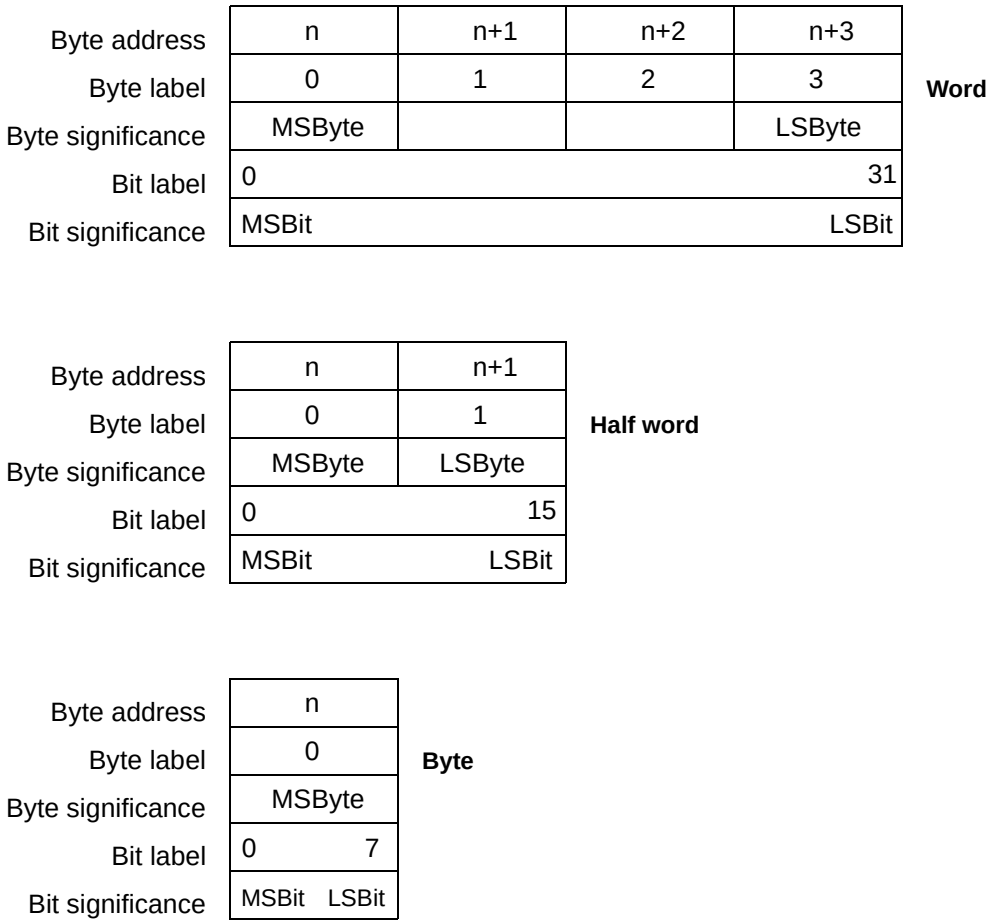


Figure 2: **Big-Endian Data Types**

Registers of the GPIO

Information on the registers used in assembly language programming are described in this section.

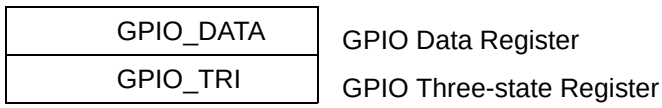


Figure 3: **GPIO Register Set**

Address Map

Table 3: **GPIO Register Address Map (32-bit OPB)**

Register	Address (Hex)	Size	Type	Description
GPIO_DATA	0x00	Word	R/W	GPIO Data Register
GPIO_TRI	0x04	Word	R/W	GPIO Three-state Register

GPIO Data Register (GPIO_DATA)

Description

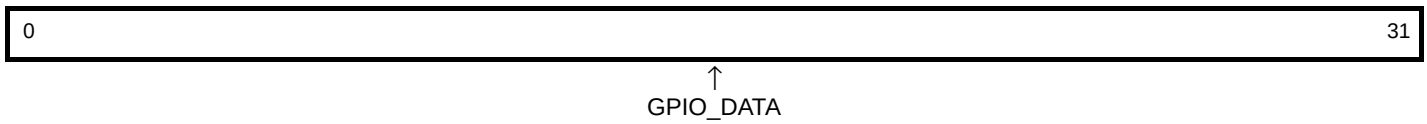


Table 4: **GPIO_DATA Register**

Bits	Name	Description	Reset Value
0:31	GPIO_DATA	GPIO Data *For I/O programmed as inputs:* R: reads value on input pin W: no effect *For I/O programmed as outputs:* R: reads value in GPIO data register W: writes value to GPIO data register and output pin	0

GPIO Three-state Register (GPIO_TRI)

Description

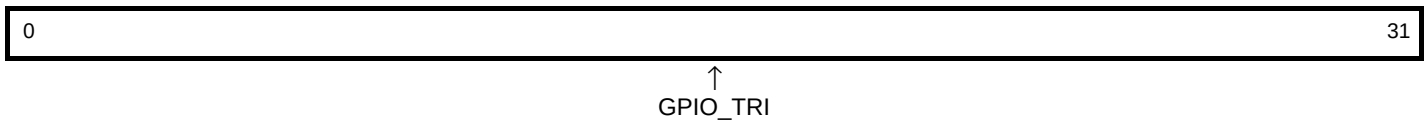


Table 5: **GPIO_TRI Register**

Bits	Name	Description	Reset Value
0:31	GPIO_TRI	GPIO Three-state Control (Bit Direction). Each I/O pin of the GPIO is individually programmable as an input or output. For each bit: 0 I/O pin configured as output 1 I/O pin configured as input	all bits = 1

Operation

GPIO Operation

A write to the GPIO_DATA register causes the written data to appear on the GPIO I/O port for I/Os that are configured as outputs. The GPIO_DATA register reads either the content of the GPIO_DATA register (for I/Os configured as outputs), or the GPIO I/O port (for I/Os configured as inputs).

The GPIO_TRI register configures the I/O as either input or output. Each bit of the I/O port has a corresponding bit in the GPIO_TRI register. Each I/O bit can be individually configured as input or output. If only inputs are required, the C_ALL_INPUTS parameter can be set to true. As a result, the GPIO_TRI register and the read multiplexer are removed from the logic to reduce resource utilization.

Implementation

I/O Summary

Table 6: **Summary of GPIO I/O (32b OPB interface)**

Signal	Interface	I/O	Description	Page
OPB_Clk	OPB	I	OPB Clock	
OPB_Rst	OPB	I	OPB Reset	
OPB_ABus[0:31]	OPB	I	OPB Address Bus	
OPB_BE[0:3]	OPB	I	OPB Byte Enables	
OPB_DBus[0:31]	OPB	I	OPB Data Bus	
OPB_RNW	OPB	I	OPB Read, Not Write	
OPB_select	OPB	I	OPB Select	
OPB_seqAddr	OPB	I	OPB Sequential Address	
GPIO_DBus[0:31]	OPB	O	GPIO Data Bus	
GPIO_errAck	OPB	O	GPIO Error Acknowledge	
GPIO_retry	OPB	O	GPIO Retry	
GPIO_toutSup	OPB	O	GPIO Timeout Suppress	
GPIO_xferAck	OPB	O	GPIO Transfer Acknowledge	
GPIO_IO[0:31]	Ext.	I/O	General Purpose Input/Outputs. Number of I/O bits is configurable at FPGA configuration, direction of each I/O bit is programmable at run-time.	

MPD File Parameters

The opb_gpio.mpd (Microprocessor Peripheral Definition) file contains a list of the peripheral's parameters that are fixed at FPGA configuration time. The parameters are described in the following table:

Table 7: **MPD Parameters**

Parameter	Description	Type
C_OPB_AWIDTH	Width of the address bus attached to the peripheral	integer
C_OPB_DWIDTH	Width of the data bus attached to the peripheral	integer
C_BASEADDR	Indicates the base address of this peripheral expressed as a std_logic_vector	std_logic_vector (0 to C_AWIDTH-1)
C_HIGHADDR	Indicates the highest address occupied by this peripheral expressed as a standard logic vector	std_logic_vector (0 to C_AWIDTH-1)
C_GPIO_WIDTH	Width of the GPIO bus (number of GPIO bits used)	integer
C_ALL_INPUTS	Indicates that all I/O are configured as inputs; results in lower resource utilization if only inputs are needed 0: I/O are programmable as input or output. 1: All I/O are inputs	integer

Parameterization

The following characteristics of the GPIO can be parameterized:

- Base address for the GPIO registers
- Width of OPB data bus attached to the peripheral
- Width of OPB address bus attached to the peripheral
- Number of GPIO bits
- I/Os are input-only or programmable as input or output

OPB Timebase WDT Specification

March 2002

Summary

This document describes the specifications for a 32-bit free-running timebase and watchdog timer core for the OPB bus. This document applies to the following peripherals:

opb_timebase_wdt	v1.00a

Overview

The TBWDT (TimeBase WatchDog Timer) is a 32-bit peripheral that attaches to the OPB (On-chip Peripheral Bus), and has the following features:

Features

- OPB V2.0 bus interface with byte-enable support
- Supports 32-bit, 16-bit, and 8-bit bus interfaces
- Watchdog timer (WDT) with selectable timeout period and interrupt
- Configurable WDT enable: enable-once or enable-disable
- One 32-bit free-running timebase counter with rollover interrupt

Timebase WDT Organization

The TBWDT block diagram is shown in the following figure:

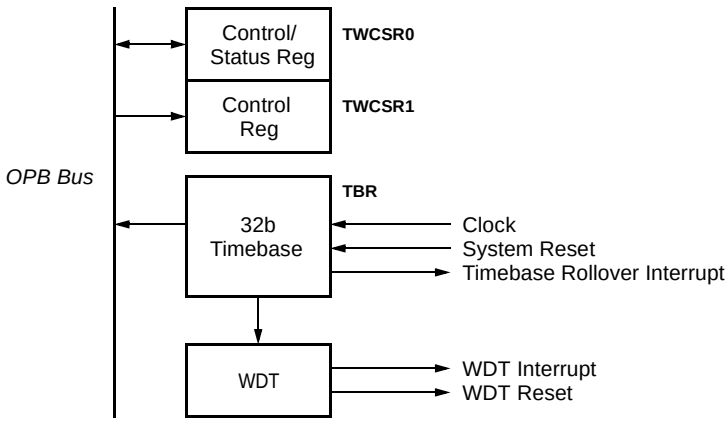


Figure 1: **Timebase/WDT Organization**

The TBWDT has the following characteristics:

- Consists of a free-running 32-bit timebase counter that is used for both general purpose timing and the WDT facility
- The timebase counter always counts up from system reset and is read-only
- The WDT timeout interval is determined by which bit in the timebase is used as input to the WDT state machine
- The WDT uses a dual-expiration architecture

After one expiration of the timeout interval an interrupt is generated and the WDT state bit is set to one in the status register. If the state bit is not cleared (by writing a 1 to the state bit) before the next expiration of the timeout interval, a WDT reset is generated. A WDT reset sets the WDT reset status bit in the status register so that the application code can determine if the last system reset was a WDT reset.

- The WDT can only be disabled by writing to two distinct addresses, reducing the possibly of inadvertently disabling the WDT in the application code

Programming Model

Register Data Types and Organization

TBWDT registers are accessed as one of the following types:

- Byte (8 bits)
- Half word (2 bytes)
- Word (4 bytes)

Configuration

The following table shows TBWDT configurations and access type.

Table 1: **TBWDT Configuration and Access Type**

Configuration	Access Type
32-bit slave OPB peripheral	Word
16-bit peripheral	Half word
8-bit peripheral	Byte
32-bit, 16-bit, or 8-bit peripheral	All register accesses are on word boundaries to conform to the OPB-IPIF register location convention

The addresses of the TBWDT registers when configured as a 32-bit OPB slave are shown in the following table:

Table 2: **TBWDT Register Address Map**

Register	Address (Hex)	Size	Type	Description
TCSR0	0x00	Word	R/W	Control/Status Register 0
TCSR1	0x04	Word	W	Control/Status Register 1 - state is mirrored in TCSR0 for read
TBR	0x08	Word	R	Timebase Register

The TBWDT registers are organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in the following figure:

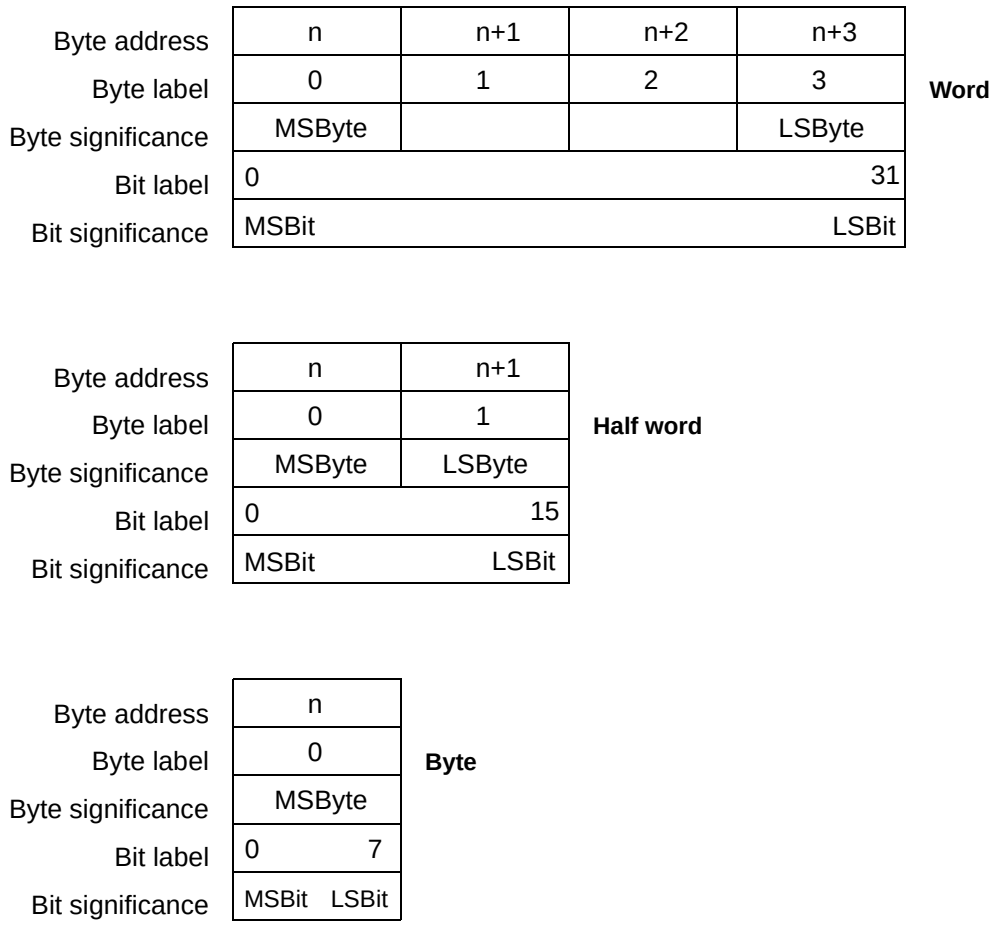


Figure 2: **Big-Endian Data Types**

Registers of the Timebase / Watchdog Timer

Registers used in assembly language programming are described in this section.

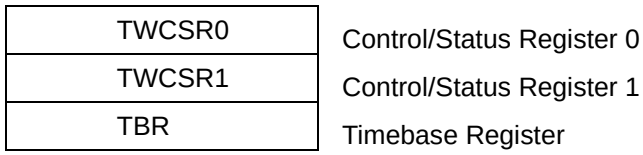


Figure 3: **TBWDT Register Set**

Address Map

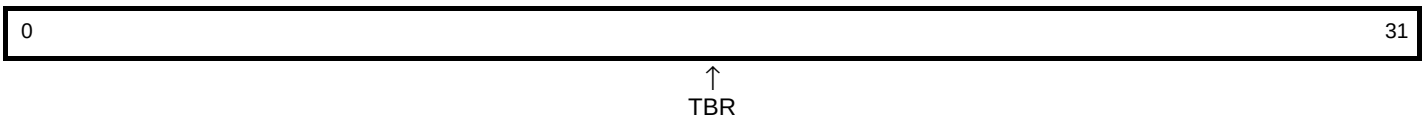
Table 3: **TBWDT Register Address Map**

Register	Address (Hex)	Size	Type	Description
TCSR0	0x00	Word	R/W	Control/Status Register 0
TCSR1	0x04	Word	W	Control/Status Register 1 - state is mirrored in TCSR0 for read
TBR	0x08	Word	R	Timebase Register

Timebase Register (TBR)

The Timebase Register is the output of a free-running incrementing counter that clocks at the input clock rate (no prescaling of the clock is done for this counter). This register is read-only and is reset by the following:

- A system reset
- Enabling the WDT after power on reset
- Enabling the WDT after the WDT has been disabled (EWDT1 and EWDT2 must both be 0 to disable the WDT). The WDT is enabled when either EWDT1 or EWDT2 are set to 1. Note that when the WDT mode is enable-once, the TBR can only be reset when the WDT is first enabled.



Control/Status Register 0 (TCSR0)

Control/Status Register 0 contains the watchdog timer reset status, watchdog timer state, and watchdog timer enables.

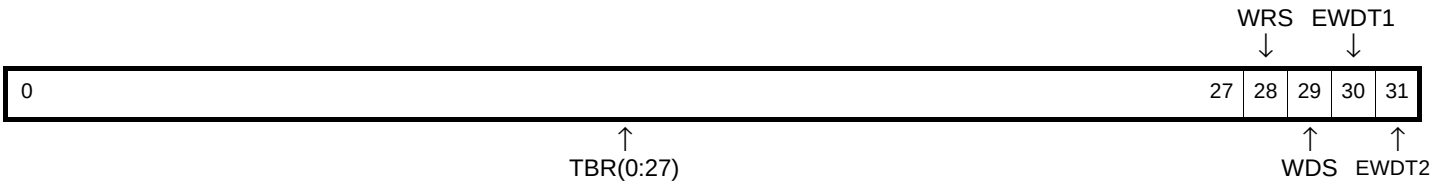


Table 4: **Control/Status Register 0 (TCSR0)**

Bits	Name	Description	Reset Value
0:27	TBR(0:27)	**Timebase Register (Most significant 28 bits)** This read-only field contains the most significant 28 bits of the timebase register. The timebase register is mirrored here so that a single read can be used to obtain the count value and the watchdog timer state if the upper 28 bits of the timebase provide sufficient timing resolution.	

Table 4: **Control/Status Register 0 (TCSR0) *(Continued)***

Bits	Name	Description	Reset Value
28	WRS	**Watchdog Reset Status** Indicates the WDT reset signal was asserted. This bit is not cleared by a system reset so that it can be read after a system reset to determine if the reset was caused by a watchdog timeout. Writing a 1 to this bit clears the watchdog reset status bit. Writing a 0 to this bit has no effect. 0 WDT reset has not occurred 1 WDT reset has occurred	
29	WDS	**Watchdog Timer State** Indicates the WDT period has expired. The WDT_Reset signal will be asserted if the WDT period expires again before this bit is cleared by software. Writing a 1 to this bit clears the watchdog timer state. Writing a 0 to this bit has no effect. 0 WDT period has not expired 1 WDT period has expired, reset will occur on next expiration	
30	EWDT1	**Enable Watchdog Timer (Enable 1)** This bit must be used in conjunction with the EWDT2 bit in the TWCSR1 register. BOTH bits must be 0 to disable the WDT. 0 Disable WDT function 1 Enable WDT function	0
31	EWDT2	**Enable Watchdog Timer (Enable 2)** This bit must be used in conjunction with the EWDT1 bit in the TCSR0 register to disable the WDT. BOTH bits must be 0 to disable the WDT. This bit is READ-ONLY in this register. The value of EWDT2 can be modified only in TWCSR1. 0 WDT function is disabled 1 WDT function is enabled	0

Control/Status Register 1 (TCSR1)

Control/Status Register 1 contains the second Watch Dog Timer (WDT) enable bit.The WDT enable must be cleared in both TCSR0 and TCSR2 to disable the WDT. If the WDT is configured as enable-once, then the WDT cannot be disabled after it has been enabled.

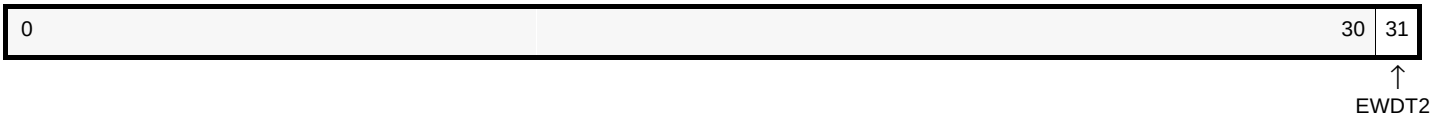


Table 5: **Control/Status Register 1 (TCSR1)**

Bits	Name	Description	Reset Value
0:30	Reserved		
31	EWDT2	**Enable Watchdog Timer (Enable 2)** This bit must be used in conjunction with the EWDT1 bit in the TCSR0 register to disable the WDT. BOTH bits must be 0 to disable the WDT. This bit is WRITE-ONLY in this register. The value of EWDT2 can be read back only in TWCSR1. 0 WDT function is disabled 1 WDT function is enabled	0

Operation

Timebase Operation

The timebase is a 32-bit up counter that is incremented by one on the rising edge of the clock provided to the TBWDT. This counter is reset to zero when the Reset input is high or when the WDT is enabled. The TBR contains the full timebase count value (32 bits). The TWCSR0 contains the most-significant 28 bits of the timebase count, as well as the WDT enable and status bits. The timing resolution from the upper 28 bits of the timebase count is T_{clk} x 16 (T_{clk} is the period of the input clock). As a result, a single access can be used to read the state of the watchdog times, as well as a reduced resolution version of the timebase.

An interrupt signal is provided that pulses high for one clock period as the counter rolls over from 0xFFFFFFFF to 0x00000000. This interrupt can be used by the software to keep track of how many timebase rollovers have occurred.

WDT Operation

The WDT timeout interval is configured by a parameter to be 2^{WDT_CLOCKS} clock cycles, where WDT_CLOCKS is any integer from 8 to 31. The WDT interval is set at FPGA configuration time and cannot be modified dynamically through a control register.

The state of the WDT is given by the WDS bit in the TWCSR0 register. If the WDT interval expires while the WDS bit is 1, the WDT reset signal is asserted. An interrupt is provided when the WDS bit is set so that the software can clear the bit before the second expiration of the WDT. The WDS bit is cleared by writing a 1 to it. Writing a 0 to the WDS bit has no effect. The WDT state diagram is shown in the following figure:

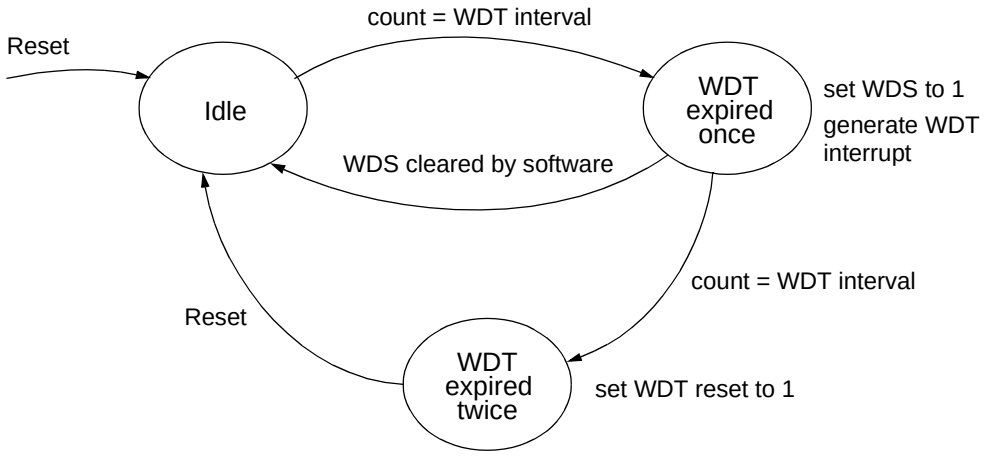


Figure 4: **WDT State Diagram**

Implementation I/O Summary

Table 6: **Summary of Timebase WDT Core I/O**

Signal	Interface	I/O	Description	Page
OPB_Clk	OPB	I	OPB Clock	
OPB_Rst	OPB	I	OPB Reset	
OPB_ABus[0:31]	OPB	I	OPB Address Bus	
OPB_BE[0:3]	OPB	I	OPB Byte Enables	
OPB_DBus[0:31]	OPB	I	OPB Data Bus	
OPB_RNW	OPB	I	OPB Read, Not Write	
OPB_select	OPB	I	OPB Select	
OPB_seqAddr	OPB	I	OPB Sequential Address	
TBWDT_DBus[0:31]	OPB	O	TBWDT Data Bus	
TBWDT_errAck	OPB	O	TBWDT Error Acknowledge	
TBWDT_retry	OPB	O	TBWDT Retry	
TBWDT_toutSup	OPB	O	TBWDT Timeout Suppress	
TBWDT_xferAck	OPB	O	TBWDT Transfer Acknowledge	
WDT_Reset	Ext.	O	Watchdog Timer Reset. Asserted upon second expiration of the WDT timeout interval.	
Timebase_Interrupt	Ext.	O	Timebase Rollover Interrupt. Asserted as a one clock period wide pulse upon rollover of the timebase from 0xFFFFFFFF to 0x00000000.	
WDT_Interrupt	Ext.	O	Watchdog Timer Interrupt. Goes high and stays high until the WDS bit is cleared in the TWCSR0 register.	

MPD File Parameters

The opb_timebase_wdt.mpd (Microprocessor Peripheral Definition) file contains a list of the peripheral's parameters that are fixed at FPGA configuration time. The parameters are described in the following table:

Table 7: **MPD Parameters**

Parameter	Description	Type
C_WDT_INTERVAL	Indicates the exponent for setting the length of the WDT interval. WDT interval = $2^{C\_WDT\_INTERVAL} \times T_{clk}$	integer
C_WDT_ENABLE_ONCE	Indicates WDT enable behavior. 0: WDT can be repeatedly enabled and disabled via software. 1: WDT can only be enabled once (no disable possible after initial enable).	integer
C_OPB_AWIDTH	The width of the address bus attached to the peripheral.	integer
C_OPB_DWIDTH	The width of the data bus attached to the peripheral.	integer
C_BASEADDR	Indicates the base address of this peripheral expressed as a std_logic_vector.	std_logic_vector (0 to C_AWIDTH-1)
C_HIGHADDR	Indicates the highest address occupied by this peripheral expressed as a standard logic vector.	std_logic_vector (0 to C_AWIDTH-1)

Device Utilization and Performance Benchmarks

The following table shows approximate resource utilization and performance benchmarks for the OPB Timer/Counter. The estimates shown are not guaranteed and can vary with FPGA family and speed grade, parameters selected for implementation, user timing constraints, and implementation tool version. Only parameters that affect resource utilization are shown in the following table.

Table 8: **OPB Timebase/WDT Performance and Resource Utilization Benchmarks (Virtex-II 2V1000-5)**

Parameter Values		Device Resources			f_{MAX} (MHz)
Address Bits in Decode	C_AWIDTH	Slices	Slice Flip-Flops	4-input LUTs	f_{MAX}
24	32		111	63	155

Parameterization

The following characteristics of the TBWDT can be parameterized:

- Base address for the TBWDT registers
- Behavior of WDT enable: enable-once or enable-many
- WDT interval
- Future parameterization: Bus interface — 8-bit, 16-bit, or 32-bit. The internal architecture of the watchdog timer/WDT remains the same across bus interface sizes

OPB Timer/Counter Specification

March 2002

Summary

This document describes the specifications for a timer/counter core for the OPB bus. This document applies to the following peripherals:

opb_timer	v1.00b

Overview

The TC (Timer/Counter) is a 32-bit timer module that attaches to the OPB (On-chip Peripheral Bus), and has the following features:

Features

- OPB V2.0 bus interface with byte-enable support
- Supports 32-bit bus interface
- Two programmable interval timers with interrupt, event generation, and event capture capabilities
- Configurable counter width
- One Pulse Width Modulation (PWM) output
- Freeze input for halting counters during software debug

Timer/Counter Organization

The TC is organized as two identical timer modules. Each timer module has an associated register (the Load Register) that is used to hold either the initial value for the counter for event generation or a capture value, depending on the mode of the timer. The TC block diagram is shown in the following figure:

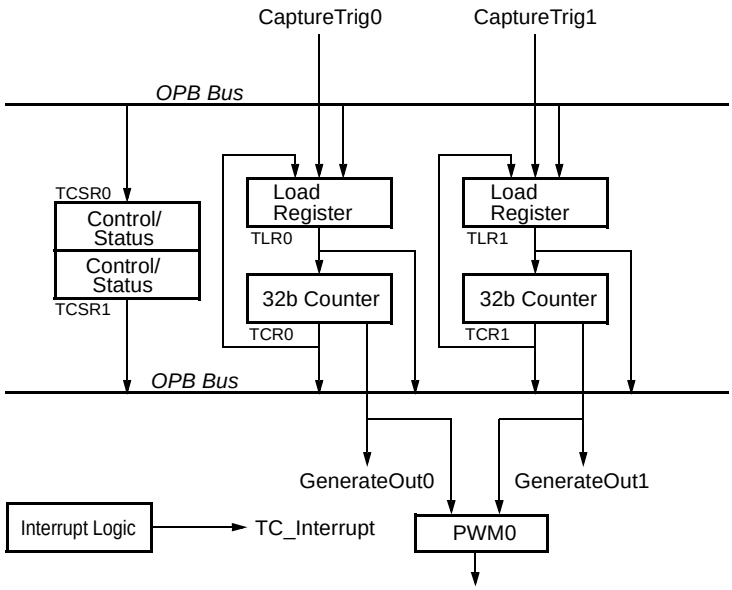


Figure 1: **Timer/Counter Organization**

The *generate value* is used to generate a single interrupt at the expiration of an interval, or a continuous series of interrupts with a programmable interval. The *capture value* is the timer value that has been latched on detection of an external event. The clock rate of the timer modules is OPB_Clk (no prescaling of the clock is performed). All of the TC interrupts are OR'ed together to generate a single external interrupt signal. The interrupt service routine reads the control/status registers to determine the source of the interrupt.

Programming Model

Timer Modes

You can use a Generate Mode, a Capture Mode, or a Pulse Width Modulation (PWM) Mode with the two timer/counter modules.

Generate Mode

In Generate Mode, the value in the Load Register is loaded into the counter and the counter begins to count (up or down, selectable by the UDT bit in TCSR) when it is enabled. On transition of the carry out of the counter, the counter stops or automatically reloads the generate value from the Load Register and continues counting (selectable by the ARHT bit in TCSR). The TINT bit is set in TCSR and, if enabled, the external GenerateOut signal is driven to 1 for one clock cycle. If enabled, the interrupt signal for the timer is driven to 1 for one clock cycle. This mode is useful for generating repetitive interrupts or external signals with a specified interval.

Characteristics

Generate Mode has the following characteristics:

- The value loaded into the Load Register is called the generate value.
- On startup, the generate value in the Load Register must be loaded into the counter by setting the Load bit in the TCSR. This applies whether the counter is set up to Auto Reload or Hold when the interval has expired. Setting the Load bit to '1' loads the counter with the value in the Load Register. The Load bit must be cleared before the counter is enabled for proper operation.
- When the ARHT bit (Auto Reload/Hold) is set to '1' and the counter rolls over from all '1's to all '0's (when counting up), or from all '0's to all '1's (when counting down), the generate value in the Load Register will be automatically reloaded into the counter and the counter will continue to count. If the GenerateOut signal is enabled (bit GENT in the TCSR), an output pulse will be generated (one clock period in width). This is useful for generating a repetitive pulse train with a specified period.
- When the ARHT bit (Auto Reload/Hold) is set to '0' and the counter rolls over from all '1's to all '0's (when counting up), or from all '0's to all '1's (when counting down), the counter will hold at the current value and will not reload the generate value. If the generate out signal is enabled (bit GENT in the TCSR), an output pulse will be generated (one clock period in width). This is useful for a one-shot pulse that is to be generated after a specified period of time.
- The counter can be set up to count either up or down (bit UDT in the TCSR). If the counter is set up as a down counter, the generate value is the number of clocks in the timing interval. The period of the GenerateOut signal is the generate value times the clock period.
- When the counter is set to count down,
  $\text{TIMING\_INTERVAL} = (\text{TLRx} + 2) \times \text{OPB\_CLOCK\_PERIOD}$.
- When the counter is set to count up,
  $\text{TIMING\_INTERVAL} = (\text{MAX\_COUNT} - \text{TLRx} + 2) \times \text{OPB\_CLOCK\_PERIOD}$,
  where MAX_COUNT is the maximum count value of the counter, such as 0xFFFFFFFF for a 32-bit counter.
- The GenerateOout signals can be configured as high-true or low-true.

Capture Mode

In Capture Mode, the value of the counter is stored in the Load Register when the external capture signal is asserted. The TINT bit is also set in TCSR on detection of the capture event. The counter can be configured as an up or down counter for this mode (selectable by the UDT bit in TCSR). The ARHT bit controls whether the capture value is overwritten with a new capture value before the previous TINT flag is cleared. This mode is useful for time tagging external events while simultaneously generating an interrupt.

Characteristics

Capture Mode has the following characteristics:

- The capture signal can be configured to be low-true or high true.
- The capture signal is sampled within the TC with the OPB_Clk. The capture event is defined as the transition on the sampled signal to the asserted state. For example, if the capture signal is defined to be high-true, then the capture event is when the sampled (synchronized to the OPB_Clk) signal transitions from ’0’ to ’1’.
- When the capture event occurs, the counter value is written to the Load Register. This value is called the capture value.
- When the ARHT bit (Auto Reload/Hold) is set to ’0’ and the capture event occurs, the capture value is written to the Load Register. The Load Register will hold this capture value until the Load Register is read. If the Load Register is not read, subsequent capture events will not update the Load Register and will be lost.
- When the ARHT bit (Auto Reload/Hold) is set to ’1’ and the capture event occurs, the capture value is always written to the Load Register. Subsequent capture events will update the Load Register and will overwrite the previous value, whether it has been read or not.
- The counter can be set up to count either up or down (bit UDT in the TCSR).

Pulse Width Modulation (PWM) Mode

In PWM mode, two timer/counters are used as a pair to produce an output signal (PWM0) with a specified frequency and duty factor. Timer0 sets the period and Timer1 sets the high time for the PWM0 output.

Characteristics

PWM Mode has the following characteristics:

- The mode for both Timer0 and TImer1 must be set to Generate Mode (bit MDT in the TCSR set to ’0’).
- The PWMA0 bit in TCSR0 and PWMB0 bit in TCSR1 must be set to ’1’ to enable PWM mode.
- The GenerateOut signals must be enabled in the TCSR (bit GENT set to ’1’). The PWM0 signal is generated from the GenerateOut signals of Timer0 and Timer1, so these signals must be enabled in both timer/counters.
- The assertion level of the GenerateOut signals for both timers in the pair must be set to ’1’. This is done by setting C_GEN0_ASSERT and C_GEN1_ASSERT to ’1’.
- The counter can be set to count up or down.

Setting the PWM Period and Duty Factor

The PWM period is determined by the generate value in Timer0’s Load Register (TLR0). The PWM high time is determined by the generate value in Timer1’s Load Register (TLR1). The period and duty factor are calculated as follows:

When counters are configured to count **up** (UDT = ’0’):

$$\text{PWM\_PERIOD} = (\text{TLR0} + 2) \times \text{OPB\_CLOCK\_PERIOD}$$

$$\text{PWM\_HIGH\_TIME} = (\text{TLR1} + 2) \times \text{OPB\_CLOCK\_PERIOD}$$

When counters are configured to count **down** (UDT = '1'):

$$PWM\_PERIOD = (MAX\_COUNT - TLR0 + 2) \times OPB\_CLOCK\_PERIOD$$

$$PWM\_HIGH\_TIME = (MAX\_COUNT - TLR1 + 2) \times OPB\_CLOCK\_PERIOD$$

where MAX_COUNT is the maximum count value for the counter, such as 0xFFFFFFFF for a 32-bit counter.

Interrupts

The TC interrupt signals can be enabled or disabled with the ENIT bit in the TCSR. The interrupt status bit (TINT) in the TCSR cannot be disabled and always reflects the current state of the timer interrupt. In Generate Mode, a timer interrupt is caused by the counter rolling over (the same condition used to reload the counter when ARHT is set to '1'). In Capture Mode, the interrupt event is the capture event. Characteristics of the interrupts are:

- Interrupt events can only occur when the timer is enabled. In Capture Mode, this prevents interrupts from occurring before the timer is enabled.
- The interrupt signal goes high for one clock cycle when the interrupt condition is met and the interrupt is enabled in the TCSR. The interrupt is asserted on the rising edge of the interrupt signal.
- A single interrupt signal is provided. The interrupt signal is the OR of the interrupts from the two counters. The interrupt service routine must poll the TCSR's to determine the source or sources of the interrupt.
- The interrupt status bit (TINT in the TCSR) can only be cleared by writing a '1' to it. Writing a '0' to it has no effect on the bit. Since the interrupt condition is an edge (the counter rollover or the capture event), it can cleared at any time and will not indicate an interrupt condition until the next interrupt event.

Register Data Types and Organization

TC registers are accessed as one of the following types:

- Byte (8 bits)
- Half word (2 bytes)
- Word (4 bytes)

Configuration

The following table shows TC configurations and access type.

Table 1: **TC Configuration and Access Type**

Configuration	Access Type
32-bit slave OPB peripheral	Word

The addresses of the TC registers are shown in the following table:

Table 2: **TC Register Address Map**

Register	Address (Hex)	Size	Type	Description
TCSR0	0x00	Word	R/W	Control/Status Register 0
TLR0	0x04	Word	R/W	Load Register 0
TCR0	0x08	Word	R	Timer/Counter Register 0

Table 2: **TC Register Address Map**

Register	Address (Hex)	Size	Type	Description
TCSR1	0x10	Word	R/W	Control/Status Register 1
TLR1	0x14	Word	R/W	Load Register 1
TCR1	0x18	Word	R	Timer/Counter Register 1

The TC registers are organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in the following figure:

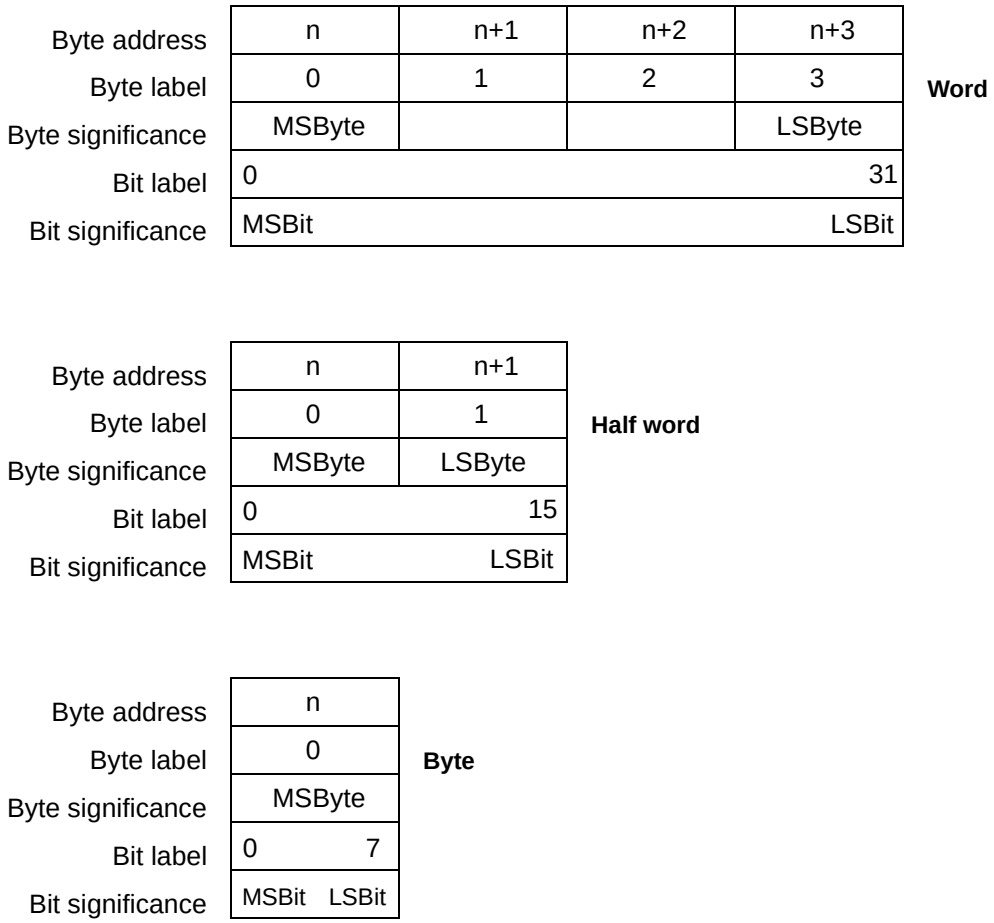
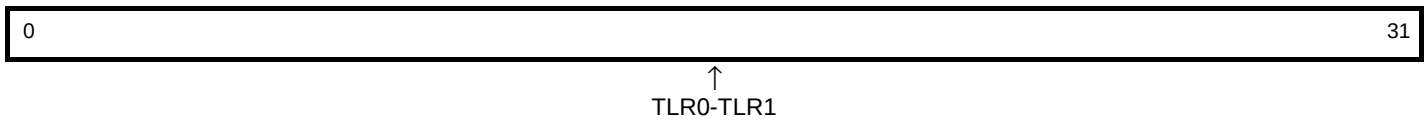


Figure 2: **Big-Endian Data Types**

Register Descriptions

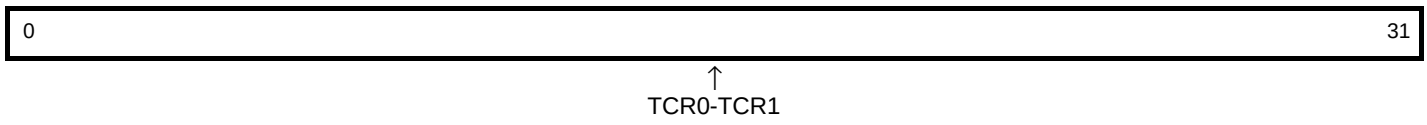
Load Register (TLR0-TLR1)

When the counter width has been configured as less than 32 bits, the Load Register value is right-justified in TLR0 and TLR1. The least-significant counter bit is always mapped to Load Register bit 31.



Timer/Counter Register (TCR0-TCR1)

When the counter width has been configured as less than 32 bits, the count value is right-justified in TCR0 and TCR1. The least-significant counter bit is always mapped to Timer/Counter Register bit 31.



Control/Status Register 0 (TCSR0)

Control/Status Register 0 contains the control and status bits for timer module 0.

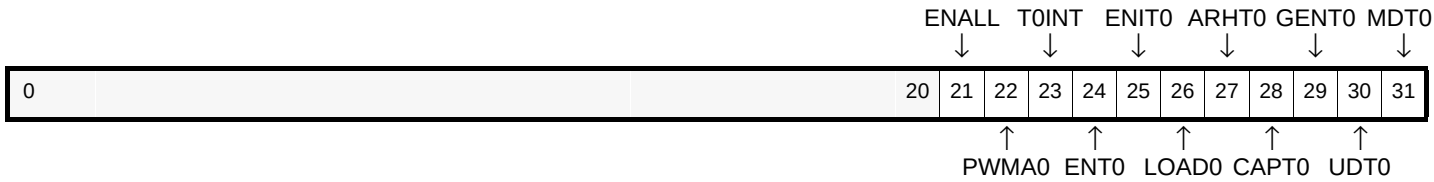


Table 3: **Control/Status Register 0 (TCSR0)**

Bits	Name	Description	Reset Value
0:20	Reserved		
21	ENALL	**Enable All Timers** 0 No effect on timers 1 Enable all timers (counters run) This bit is mirrored in all control/status registers and is used to enable all counters simultaneously. Writing a '1' to this bit sets ENALL, ENT0, and ENT1. Writing a '0' to this register clears ENALL but has no effect on ENT0 and ENT1.	0

Table 3: **Control/Status Register 0 (TCSR0) *(Continued)***

Bits	Name	Description	Reset Value
22	PWMA0	**Enable Pulse Width Modulation for Timer0** 0 Disable pulse width modulation 1 Enable pulse width modulation PWM requires using Timer0 and Timer1 together as a pair. Timer0 sets the period of the PWM output, and Timer1 sets the high time for the PWM output. For PWM Mode, MDT0 and MDT1 must be '0' and C_GEN0_ASSERT and C_GEN1_ASSERT must be '1'.	0
23	TINT0	**Timer0 Interrupt** Indicates that the condition for an interrupt on this timer has occurred. If the timer mode is capture and the timer is enabled, this bit indicates a capture has occurred. If the mode is generate, this bit indicates the counter has rolled over. Must be cleared by writing a '1'. *Read:* 0 No interrupt has occurred 1 Interrupt has occurred *Write:* 0 No change in state of T0INT 1 Clear T0INT (clear to '0')	0
24	ENT0	**Enable Timer0** 0 Disable timer (counter halts) 1 Enable timer (counter runs)	0
25	ENIT0	**Enable Interrupt for Timer0** Enables the assertion of the interrupt signal for this timer. Has no effect on the interrupt flag in TCSR0. 0 Disable interrupt signal 1 Enable interrupt signal	0
26	LOAD0	**Load Timer0** 0 No load 1 Loads timer with value in TLR0	0

Table 3: **Control/Status Register 0 (TCSR0) *(Continued)***

Bits	Name	Description	Reset Value
27	ARHT0	**Auto Reload/Hold Timer0** When the timer is in Generate Mode, this bit determines whether the counter reloads the generate value and continues running or holds at the termination value. In Capture Mode, this bit determines whether a new capture trigger overwrites the previous captured value or if the previous value is held. 0 Hold counter or capture value 1 Reload generate value or overwrite capture value	0
28	CAPT0	**Enable External Capture Trigger Timer0** 0 Disables external capture trigger 1 Enables external capture trigger	0
29	GENT0	**Enable External Generate Signal Timer0** 0 Disables external generate signal 1 Enables external generate signal	0
30	UDT0	**Up/Down Count Timer0** 0 Timer functions as up counter 1 Timer functions as down counter	0
31	MDT0	**Timer0 Mode** See the **Timer Modes** section. 0 Timer mode is generate 1 Timer mode is capture	0

Control/Status Register 1 (TCSR1)

Control/Status Register 1 contains the control and status bits for timer module 1.

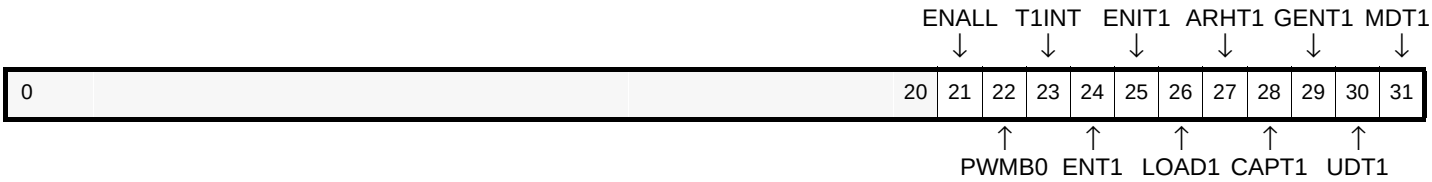


Table 4: **Control/Status Register 1 (TCSR1)**

Bits	Name	Description	Reset Value
0:20	Reserved		

Table 4: **Control/Status Register 1 (TCSR1) *(Continued)***

Bits	Name	Description	Reset Value
21	ENALL	**Enable All Timers** 0 No effect on timers 1 Enable all timers (counters run) This bit is mirrored in all control/status registers and is used to enable all counters simultaneously. Writing a '1' to this bit sets ENALL, ENT0, and ENT1. Writing a '0' to this register clears ENALL but has no effect on ENT0 and ENT1.	0
22	PWMB0	**Enable Pulse Width Modulation for Timer1** 0 Disable pulse width modulation 1 Enable pulse width modulation PWM requires using Timer0 and Timer1 together as a pair. Timer0 sets the period of the PWM output, and Timer1 sets the high time for the PWM output. For PWM Mode, MDT0 and MDT1 must be '0' and C_GEN0_ASSERT and C_GEN1_ASSERT must be '1'.	0
23	TINT1	**Timer1 Interrupt** Indicates that the condition for an interrupt on this timer has occurred. If the timer mode is capture and the timer is enabled, this bit indicates a capture has occurred. If the mode is generate, this bit indicates the counter has rolled over. Must be cleared by writing a '1'. *Read:* 0 No interrupt has occurred 1 Interrupt has occurred *Write:* 0 No change in state of T1INT 1 Clear T1INT (clear to '0')	0
24	ENT1	**Enable Timer1** 0 Disable timer (counter halts) 1 Enable timer (counter runs)	0
25	ENIT1	**Enable Interrupt for Timer1** Enables the assertion of the interrupt signal for this timer. Has no effect on the interrupt flag in TCSR1. 0 Disable interrupt signal 1 Enable interrupt signal	0

Table 4: **Control/Status Register 1 (TCSR1) *(Continued)***

Bits	Name	Description	Reset Value
26	LOAD1	**Load Timer1** 0 No load 1 Loads timer with value in TLR1	0
27	ARHT1	**Auto Reload/Hold Timer1** When the timer is in generate mode, this bit determines whether the counter reloads the generate value and continues running or holds at the termination value. In capture mode, this bit determines whether a new capture trigger overwrites the previous captured value or if the previous value is held until it is read. 0 Hold counter or capture value 1 Reload generate value or overwrite capture value	0
28	CAPT1	**Enable External Capture Trigger Timer1** 0 Disables external capture trigger 1 Enables external capture trigger	0
29	GENT1	**Enable External Generate Signal Timer1** 0 Disables external generate signal 1 Enables external generate signal	0
30	UDT1	**Up/Down Count Timer1** 0 Timer functions as up counter 1 Timer functions as down counter	0
31	MDT1	**Timer1 Mode** See the **Timer Modes** section. 0 Timer mode is generate 1 Timer mode is capture	0

Implementation I/O Summary

Table 5: **Summary of Timer Core I/O**

Signal	Interface	I/O	Description
OPB_Clk	OPB	I	OPB Clock
OPB_Rst	OPB	I	OPB Reset
OPB_ABus[0:31]	OPB	I	OPB Address Bus
OPB_BE[0:3]	OPB	I	OPB Byte Enables
OPB_DBus[0:31]	OPB	I	OPB Data Bus
OPB_RNW	OPB	I	OPB Read, Not Write

Table 5: **Summary of Timer Core I/O**

Signal	Interface	I/O	Description
OPB_select	OPB	I	OPB Select
OPB_seqAddr	OPB	I	OPB Sequential Address
TC_DBus[0:31]	OPB	O	TC Data Bus
TC_errAck	OPB	O	TC Error Acknowledge
TC_retry	OPB	O	TC Retry
TC_toutSup	OPB	O	TC Timeout Suppress
TC_xferAck	OPB	O	TC Transfer Acknowledge
CaptureTrig0	Ext.	I	Capture Trigger 0
CaptureTrig1	Ext.	I	Capture Trigger 1
GenerateOut0	Ext.	O	Generate Output 0
GenerateOut1	Ext.	O	Generate Output 1
PWM0	Ext.	O	Pulse Width Modulation Output 0
Interrupt	Ext.	O	Interrupt
Freeze	Ext.	I	Freeze Count Value

MPD File Parameters

The opb_timer.mpd (Microprocessor Peripheral Definition) file contains a list of the peripheral's parameters that are fixed at FPGA configuration time. The parameters are described in the following table:

Table 6: **MPD Parameters**

Parameter	Description	Type
C_FAMILY	FPGA family, one of virtex, virtexe, virtex2, virtex2p, spartan2, or spartan2e	string
C_COUNT_WIDTH	The width in bits of the counters in the OPB TImer/Counter	integer range 8 to 32
C_ONE_TIMER_ ONLY	0: Two timers are present 1: One timer is present (No PWM mode)	integer
C_TRIG0_ASSERT	'0': CaptureTrig0 input is low-true '1': CaptureTrig0 input is high-true	std_logic
C_COUNT_WIDTH	The width in bits of the counters	integer
C_TRIG1_ASSERT	'0': CaptureTrig1 input is low-true '1': CaptureTrig1 input is high-true	std_logic
C_GEN0_ASSERT	'0': GenerateOut0 output is low-true '1': GenerateOut0 output is high-true	std_logic
C_GEN1_ASSERT	'0': GenerateOut1 output is low-true '1': GenerateOut1 output is high-true	std_logic
C_OPB_AWIDTH	The width in bits of the address bus attached to the peripheral.	integer

Table 6: **MPD Parameters *(Continued)***

Parameter	Description	Type
C_OPB_DWIDTH	The width in bits of the data bus attached to the peripheral.	integer
C_BASEADDR	Indicates the base address of this peripheral expressed as a std_logic_vector.	std_logic_vector (0 to C_AWIDTH-1)
C_HIGHADDR	Indicates the highest address occupied by this peripheral expressed as a standard logic vector.	std_logic_vector (0 to C_AWIDTH-1)

Device Utilization and Performance Benchmarks

The following table shows approximate resource utilization and performance benchmarks for the OPB Timer/Counter. The estimates shown are not guaranteed and can vary with FPGA family and speed grade, parameters selected for implementation, user timing constraints, and implementation tool version. Only parameters that affect resource utilization are shown in the following table:

Table 7: **OPB Timer/Counter Performance and Resource Utilization Benchmarks (Virtex-II 2V1000-5)**

Parameter Values		Device Resources			f_{MAX} (MHz)
Address Bits in Decode	C_AWIDTH	Slices	Slice Flip-Flops	4-input LUTs	f_{MAX}
4	32		238	254	130
8	32		245	258	130
16	32		253	260	131
24	32		261	260	130

Parameterization

The following characteristics of the TC can be parameterized:

- Base address for the TC registers
- Assertion level for CaptureTrig and GenerateOut signals (high-true or low-true)
- Future parameterization: number of timer/counter modules

OPB JTAG_UART Specification

March 2002

Summary

This document describes the specifications for a JTAG_UART core for the On-chip Peripheral Bus (OPB). This document applies to the following peripherals:

opb_jtag_uart	v1.00b

Overview

The JTAG_UART is a module that attaches to the OPB (On-chip Peripheral Bus), and has the following features:

Features

- Mimics UART functionality to MicroBlaze but sends data over JTAG
- OPB V2.0 bus interface with byte-enable support
- Supports 8-bit bus interfaces
- One transmit and one receive channel (full duplex)
- 16-character transmit FIFO and 16-character receive FIFO
- Requires xmd or xmdterm to run on host for JTAG communication
- Able to reset system and MicroBlaze
- Able to assert break signals to MicroBlaze

JTAG_UART Parameters

To allow you to obtain a JTAG_UART that is uniquely tailored for your system, certain features can be parameterized in the JTAG_UART design. This allows you to configure a design that only utilizes the resources required by your system, and operates with the best possible performance. The features that can be parameterized in the Xilinx JTAG_UART design are shown in Table 1.

Table 1: **JTAG_UART Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
JTAG_UART Registers Base Address	C_BASEADDR	Valid Address Range(2)	None(1)	std_logic_vector
JTAG_UART Registers HIGH Address	C_HIGHADDR	Valid Address Range(2)	None(1)	std_logic_vector
Target Family	C_FAMILY	Xilinx FPGA families	virtex2	strings
OPB Data Bus Width	C_OPB_DWIDTH	32	32	integer
OPBAddress Bus Width	C_OPB_AWIDTH	8 - 32	32	integer

Notes:

1. Address range specified by C_BASEADDR and C_HIGHADDR must be a power of 2
2. No default value is specified for C_BASEADDR and C_HIGHADDR to insure that the actual value is set; if the value is not set, a compiler error is generated. These generics must be a power of 2.

JTAG_UART I/O Signals

The I/O signals for the JTAG_UART are listed in Table 2.

Table 2: **JTAG_UART I/O Signals**

Signal Name	Interface	I/O	Description
OPB_Clk	OPB	I	OPB Clock
OPB_Rst	OPB	I	OPB Reset
OPB_ABus[0:31]	OPB	I	OPB Address Bus
OPB_BE[0:3]	OPB	I	OPB Byte Enables
OPB_DBus[0:31]	OPB	I	OPB Data Bus
OPB_RNW	OPB	I	OPB Read, Not Write
OPB_select	OPB	I	OPB Select
OPB_seqAddr	OPB	I	OPB Sequential Address
JTAG_UART_DBus[0:31]	OPB	O	JTAG_UART Data Bus
JTAG_UART_errAck	OPB	O	JTAG_UART Error Acknowledge
JTAG_UART_retry	OPB	O	JTAG_UART Retry
JTAG_UART_toutSup	OPB	O	JTAG_UART Timeout Suppress
JTAG_UART_xferAck	OPB	O	JTAG_UART Transfer Acknowledge
Interrupt	Interrupt	O	JTAG_UART Interrupt
RX	External	I	Receive Data
TX	External	O	Transmit Data
Debug_SYS_Rst	Internal	O	Reset signal to OPB V2.0
Debug_Rst	Internal	O	Reset signal to MicroBlaze
Ext_BRK	Internal	0	Break signal to MicroBlaze
Ext_NM_BRK	Internal	O	Non-maskable break signal to MicroBlaze

JTAG_UART Address Map and Register Descriptions

Register Data Types and Organization

Registers in the JTAG_UART are accessed as one of three types: byte (8 bits), halfword (2 bytes), and word (4 bytes). All register accesses are on word boundaries to conform to the OPB-IPIF register location convention. The addresses of the JTAG_UART registers are shown in the **Address Map** section.

The JTAG_UART registers are organized as big-endian data. The bit and byte labeling for the big-endian data types is shown in Figure 1.

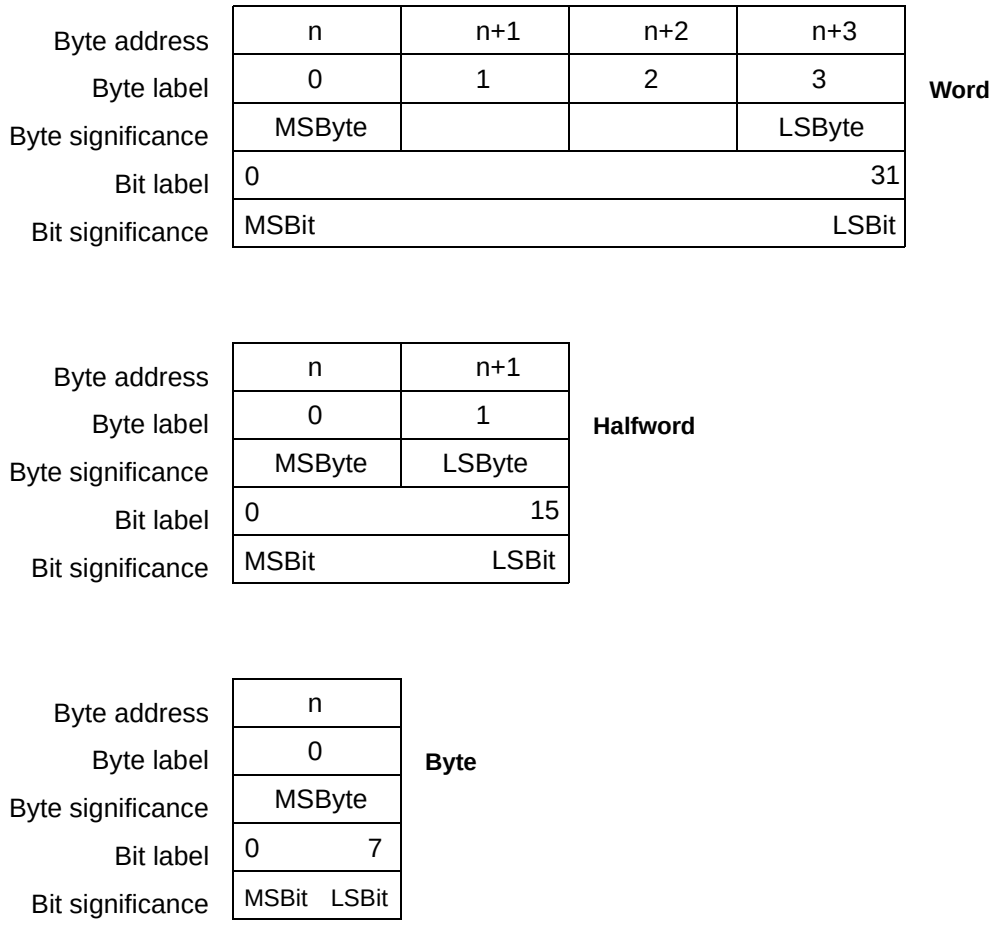


Figure 1: **Big-Endian Data Types**

Registers of the JTAG_UART

Information on the following registers used in assembly language programming are described in this section.

Receive FIFO	Read character from Receive FIFO
Transmit FIFO	Write character into Transmit FIFO
Status	Read from Status Register
Control	Write to Control Register

Figure 2: **JTAG_UART Register Set**

Status Register (STATREG)

The Status register contains the status of the receive and transmit FIFO, if interrupts are enabled, and if there are any errors.

Table 3: **Status Register**

Bits	Name	Description	Reset Value
0-26	Reserved	**Not used**	0
27	INTR_ENABLED	Interrupts is enabled Indicates that interrupts is enabled 0 Interrupt is disabled 1 Interrupt is enabled	0
28	TX_FIFO_FULL	Transmit FIFO is full Indicates if the transmit FIFO is full. 0 Transmit FIFO is not full 1 Transmit FIFO is full	
29	TX_FIFO_EMPTY	Transmit FIFO is empty Indicates if the transmit FIFO is empty. 0 Transmit FIFO is not empty 1 Transmit FIFO is empty	
30	RX_FIFO_FULL	Receive FIFO is full Indicates if the receive FIFO is full. 0 Receive FIFO is not full 1 Receive FIFO is full	
31	RX_FIFO_VALID_DATA	Receive FIFO is has valid data Indicates if the receive FIFO has valid data. 0 Receive FIFO is empty 1 Receive FIFO has valid data	

Control Register (CTRL_REG)

The Control register contains the control of the JTAG_UART.

Table 4: **Control Register (CTRL_REG)**

Bits	Name	Description	Reset Value
0-26	Reserved	Not used	0
27	ENABLE_INTR	Enable Interrupt for the JTAG_UART 0 Disable interrupt signal 1 Enable interrupt signal	0
28-29	Reserved	Not used	0
30	RST_RX_FIFO	Reset/Clear the receive FIFO When written to with a '1' the receive FIFO is cleared. 0 Do nothing 1 Clear the receive FIFO	0
31	RST_TX_FIFO	Reset/Clear the transmit FIFO When written to with a '1' the transmit FIFO is cleared. 0 Do nothing 1 Clear the transmit FIFO	0

Address Map

JTAG_UART_BASE_ADDRESS + 0: Read from Receive FIFO

JTAG_UART_BASE_ADDRESS + 4: Write to transmit FIFO

JTAG_UART_BASE_ADDRESS + 8: Read from Status Register

JTAG_UART_BASE_ADDRESS + 12: Write to Control Register

Interrupts

If interrupts are enabled, an interrupt is generated when one of the following conditions is true:

1. When there exists any valid character in the receive FIFO, the interrupt stays active until the receive FIFO is empty

2. When the transmit FIFO goes from not empty to empty, such as when the last character in the transmit FIFO is transmitted, the interrupt is only active one clock cycle.

Design Implementation

Device Utilization and Performance Benchmarks

The following table shows approximate resource utilization and performance benchmarks for the OPB JTAG_UART. The estimates shown are not guaranteed and can vary with FPGA family and speed grade, implementation parameters, user timing constraints, and implementation tool version. Only parameters that affect resource utilization are shown in the following table.

Table 5: **OPB JTAG_UART Performance and Resource Utilization Benchmarks (Virtex-II 2V1000-5)**

Parameter Values		Device Resources		f_{MAX} (MHz)
Address Bits in Decode	C_AW IDTH	Flip-Flops	4-input LUTs	f_{MAX}
24	32			

Appendix A: MicroBlaze Endianness

This chapter describes big-endian and little-endian data objects and how to use little-endian data with the big-endian MicroBlaze soft processor. This chapter includes the following sections:

- "Origin of Endian"
- "Definitions"
- "Bit Naming Conventions"
- "Data Types and Endianness"
- "VHDL Example"

Origin of Endian

The terms *Big-Endian* and *Little-Endian* come from Part I, Chapter 4, of Jonathan Swift's *Gulliver's Travels*. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

. . . our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of Lilliput and Blefuscu. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of Blefuscu; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the Big-Endians have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of Blefuscu did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet Lustrog, in the fifty-fourth Chapter of the Brundrecal, (which is their Alcoran.) This, however, is thought to be a mere Strain upon the text: For the Words are these; That all true Believers shall break their Eggs at the convenient End: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the Big-Endian Exiles have found so much Credit in the Emperor of Blefuscu's Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty

thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

Definitions

Data are stored or retrieved in memory, in byte, half word, word, or double word units. Endianness refers to the order in which data are stored and retrieved. Little-endian specifies that the least significant byte is assigned the lowest byte address. Big-endian specifies that the most significant byte is assigned the lowest byte address.

Note Endianness does not affect single byte data.

Bit Naming Conventions

The MicroBlaze architecture uses a bus and register bit naming convention in which the most significant bit (MSB) name incorporates zero (‘0’). As the significance of the bits decreases across the bus, the number in the name increases linearly so that a 32-bit vector has a least significant bit (LSB) name equal to 31. Other Xilinx interfaces such as the PCI Core use the opposite convention in which a name with a ‘0’ represents the LSB vector position.

Data Types and Endianness

Hardware supported data types for MicroBlaze are word, half word, and byte. The data organization for each type is shown in the following tables.

Table 0-1 Word Data Type

Byte address	n	n+1	n+2	n+3
Byte label	0	1	2	3
Byte significance	MSByte			LSByte
Bit label	0			31
Bit significance	MSBit			LSBit

Table 0-2 Half Word Data Type

Byte address	n	n+1
Byte label	0	1
Byte significance	MSByte	LSByte
Bit label	0	15
Bit significance	MSBit	LSBit

Table 0-3 Byte Data Type

Byte address	n	
Byte label	0	
Byte significance	MSByte	
Bit label	0	7
Bit significance	MSBit	LSBit

The following C language structure includes various scalars and character strings. The comments indicate the value assumed to be in each structure element. These values show how the bytes comprising each structure element are mapped into storage.

```
struct {
int a; /* 0x1112_1314 word */
long long b; /* 0x2122_2324_2526_2728  double word */
char *c; /* 0x3132_3334 word */
char d[7]; /* 'A','B','C','D','E','F','G' array of bytes */
short e; /* 0x5152 halfword */
int f; /* 0x6162_6364 word */
} s;
```

C structure mapping rules permit the use of padding (skipped bytes) to align scalars on desirable boundaries. The structure mapping examples show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between a and b, one byte between d and e, and two bytes between e and f. The same amount of padding is **present in both big-endian and little-endian mappings.**

Note For the MicroBlaze core, all operands in the ALU and GPRs, and all pipeline instructions are big-endian.

The big-endian mapping of "struct" is shown in the following table. (The data is highlighted in the structure mappings). Hexadecimal addresses are below the data stored at the address. The contents of each byte, as defined in the structure, are shown as a number (hexadecimal) or character (for the string elements).

Table 0-4 Big-endian Mapping

11 0x00	12 0x01	13 0x02	14 0x03	0x04	0x05	0x06	0x07
21 0x08	22 0x09	23 0x0A	24 0x0B	25 0x0C	26 0x0D	27 0x0E	28 0x0F
31 0x10	32 0x11	33 0x12	34 0x13	'A' 0x14	'B' 0x15	'C' 0x16	'D' 0x17
'E' 0x18	'F' 0x19	'G' 0x1A	0x1B	51 0x1C	52 0x1D	0x1E	0x1F
61 0x20	62 0x21	63 0x22	64 0x23	0x24	0x25	0x26	0x27

Table 0-5 Little-endian Mapping

14 0x00	13 0x01	12 0x02	11 0x03	0x04	0x05	0x06	0x07
28 0x08	27 0x09	26 0x0A	25 0x0B	24 0x0C	23 0x0D	22 0x0E	21 0x0F
34 0x10	33 0x11	32 0x12	31 0x13	'A' 0x14	'B' 0x15	'C' 0x16	'D' 0x17
'E' 0x18	'F' 0x19	'G' 0x1A	0x1B	52 0x1C	51 0x1D	0x1E	0x1F
64 0x20	63 0x21	62 0x22	61 0x23	0x24	0x25	0x26	0x27

VHDL Example

BRAM – LMB Example

LMB uses big-endian byte addressing, while the BRAM uses little-endian byte addressing. To translate data between the two busses, swap the data and address bytes.

Interface Between BRAM and MicroBlaze

```
entity Local_Memory is
  port (
    Clk   : in std_logic;
    Reset : in boolean;

    -- Instruction Bus
    Instr_Addr : in  std_logic_vector(0 to 31);
    Instr      : out std_logic_vector(0 to 31);
    IFetch     : in  std_logic;
    I_AS       : in  std_logic;
    IReady     : out std_logic;

    -- ports to "Decode_I"
    Data_Addr    : in  std_logic_vector(0 to 31);
    Data_Read    : out std_logic_vector(0 to 31);
    Data_Write   : in  std_logic_vector(0 to 31);
    D_AS         : in  std_logic;
    Read_Strobe  : in  std_logic;
    Write_Strobe : in  std_logic;
    DReady       : out std_logic;
    Byte_Enable  : in  std_logic_vector(0 to 3)
    );

end Local_Memory;

architecture IMP of Local_Memory is
```

BRAM Component Declaration (little-endian)

```
  component mem_dp_0 is
    port (
      addra : in  std_logic_vector(9 downto 0);
      addrb : in  std_logic_vector(9 downto 0);
      clka  : in  std_logic;
      clkb  : in  std_logic;
      dinb  : in  std_logic_vector(7 downto 0);
      douta : out std_logic_vector(7 downto 0);
      doutb : out std_logic_vector(7 downto 0);
      web   : in  std_logic);
  end component mem_dp_0;
```

Swap BRAM Little-endian Data to Big-endian

```
Swap_BE_and_LE_order : process (....)
begin
    for I in addra'range loop
      addra(I) <= Instr_Addr(29-I);
    end loop;
    for I in addrb'range loop
      addrb(I) <= Data_Addr(29-I);
```

```
    end loop;
    for I in 0 to 3 loop
      for J in 0 to 7 loop
        dinb(I*8+J) <= Data_Write((3-I)*8+(7-J));
        Instr((3-I)*8+(7-J)) <= douta(I*8+J);
        Data_Read((3-I)*8+(7-J)) <= doutb(I*8+J);
      end loop;
    end loop;
  end process Swap_BE_and_LE_order;
```

BRAM Instantiation

```
mem_dp_0_I : mem_dp_0
port map (
  addra=>addra,               --[IN std_logic_VECTOR(9 downto 0)]
  addrb=>addrb,               --[IN std_logic_VECTOR(9 downto 0)]
  clka=>Clk,                  --[IN std_logic]
  clkb=>Clk,                  --[IN std_logic]
  dinb=>dinb(31 downto 24)    --[IN std_logic_VECTOR(7 downto 0)]
  douta=>douta(31 downto 24), --[OUT std_logic_VECTOR(7 downto 0)]
  doutb => doutb(31 downto 24), --[OUT std_logic_VECTOR(7 downto 0)]
  web=>we(0));                --[IN std_logic]
```

BRAM – OPB Example

OPB uses big-endian byte addressing, while the BRAM uses little-endian byte addressing. To translate data between the two buses, swap the data and address bytes.

Interface Between BRAM and MicroBlaze

```
library IEEE;
use IEEE.std_logic_1164.all;

entity OPB_BRAM is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"B000_0000";
    C_NO_BRAMS : natural            := 4;  -- Can be 4,8,16,32 only
    C_VIRTEXII    : boolean         := true
    );
  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;

    -- OPB signals
    OPB_ABus    : in std_logic_vector(0 to 31);
    OPB_BE      : in std_logic_vector(0 to 3);
    OPB_RNW     : in std_logic;
    OPB_select  : in std_logic;
    OPB_seqAddr : in std_logic;
    OPB_DBus    : in std_logic_vector(0 to 31);

    OPB_BRAM_DBus    : out std_logic_vector(0 to 31);
    OPB_BRAM_errAck  : out std_logic;
    OPB_BRAM_retry   : out std_logic;
    OPB_BRAM_toutSup : out std_logic;
    OPB_BRAM_xferAck : out std_logic;
```

```
    -- OPB_BRAM signals (other port)
    BRAM_Clk        : in  std_logic;
    BRAM_Addr       : in  std_logic_vector(0 to 31);
    BRAM_WE         : in  std_logic_vector(0 to 3);
    BRAM_Write_Data : in  std_logic_vector(0 to 31);
    BRAM_Read_Data  : out std_logic_vector(0 to 31)
    );

end entity OPB_BRAM;

architecture IMP of OPB_BRAM is
```

BRAM Component Declaration (little-endian)

```
component RAMB16_S9_S9
    port (
      DIA   : in  std_logic_vector (7 downto 0);
      DIB   : in  std_logic_vector (7 downto 0);
      DIPA  : in  std_logic_vector (0 downto 0);
      DIPB  : in  std_logic_vector (0 downto 0);
      ENA   : in  std_ulogic;
      ENB   : in  std_ulogic;
      WEA   : in  std_ulogic;
      WEB  : in  std_ulogic;
      SSRA  : in  std_ulogic;
      SSRB  : in  std_ulogic;
      CLKA  : in  std_ulogic;
      CLKB  : in  std_ulogic;
      ADDRA : in  std_logic_vector (10 downto 0);
      ADDRB : in  std_logic_vector (10 downto 0);
      DOA   : out std_logic_vector (7 downto 0);
      DOB   : out std_logic_vector (7 downto 0);
      DOPA  : out std_logic_vector (0 downto 0);
      DOPB  : out std_logic_vector (0 downto 0) );
  end component;
```

Swap BRAM Little-endian Data to Big-endian

```
  BE_to_LE : for I in 0 to 31 generate
    opb_dbus_le(I )       <= OPB_DBus(31-I);
    bram_write_data_le(I) <= BRAM_Write_Data(31-I);
    BRAM_Read_Data(I)    <= bram_Read_Data_LE(31-I);
    opb_ABus_LE(I)        <= OPB_ABus(31-I);
    bram_Addr_LE(I)       <= BRAM_Addr(31-I);
  end generate BE_to_LE;
```

BRAM Instantiation

```
All_Brams : for I in 0 to C_NO_BRAMS-1 generate

By_8 : if (C_NO_BRAMS = 4) generate

RAMB16_S9_S9_I : RAMB16_S9_S9
port map (
DIA => opb_DBUS_LE(((I+1)*8-1) downto I*8), --[in std_logic_vector(7 downto 0)]
DIB =>bram_Write_Data_LE(((I+1)*8)-1 downto I*8), --[in std_logic_vector (downto 0)]
DIPA => null_1,            -- [in  std_logic_vector (7 downto 0)]
DIPB  => null_1,           -- [in  std_logic_vector (7 downto 0)]
ENA   => '1',               -- [in  std_ulogic]
ENB   => '1',               -- [in  std_ulogic]
WEA   => opb_WE(I),         -- [in  std_ulogic]
WEB   => BRAM_WE(I),        -- [in  std_ulogic]
```

```
SSRA  => '0',               -- [in  std_ulogic]
SSRB  => '0',               -- [in  std_ulogic]
CLKA  => OPB_Clk,           -- [in  std_ulogic]
CLKB  => BRAM_Clk,         -- [in  std_ulogic]
ADDRA => opb_ABus_LE(12 downto 2),  -- [in  std_logic_vector (10 downto 0)]
ADDRB => bram_Addr_LE(12 downto 2),  -- [in  std_logic_vector (10 downto 0)]
DOA=>opb_BRAM_DBus_LE_I(((I+1)*8-1)downto I*8),--[out std_logic_vector(7 downto 0)]
DOB =>bram_Read_Data_LE(((I+1)*8-1) downto I*8),--[out std_logic_vector(7 downto 0)]
DOPA  => open,              -- [out std_logic_vector (0 downto 0)]
DOPB  => open);             -- [out std_logic_vector (0 downto 0)]
end generate By_8;
```

Appendix B: JTAG Connector

This appendix provides information on the standard 10-pin JTAG connector that can be used in a MicroBlaze system.

The following figure shows the JTAG connector pins and signal names

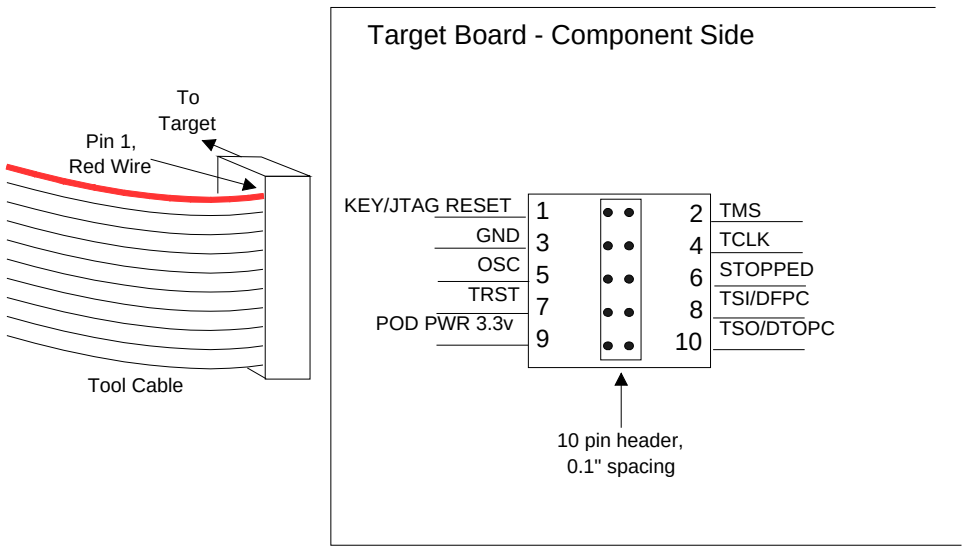


Figure 0-1 JTAG Connector

The following table lists the pins, signal names, and signal functions for the connector.

Table 0-1 JTAG Connector Signals

Pin Number	Signal Name	Function
1	KEY / JTAG REST	This signal is normally not used, and often the female part has a plug in the Pin 1 hole as the pin on the target board is cut off. Alternatively, this pin can be used for a JTAG state machine reset if needed.
3	GND	Digital signal ground.
5	OSC	Oscillator signal; normally not used, but can generate a clock up to 100 MHz.
7	TRST	Target reset. This signal is normally an open collector and active low.

Table 0-1 JTAG Connector Signals

Pin Number	Signal Name	Function
9	POD POWER 3.3V	This pin is used to supply the buffers in the tool with 3.3 or other low voltage. By powering the buffers with a target voltage, it is possible to interface to many different logic levels. The power flows from the target board to the tool to power only the buffers that drive the JTAG signals to the target board.
2	TMS	The JTAG TMS line that is used to control the JTAG tap state machine.
4	TCLK	The JTAG clock.
6	STOPPED	Signal normally not used, but the tool can optionally detect that the target system is not running if this signal is held high.
8	TSI / DFPC	JTAG data input to the target silicon from the tool.
10	TSO / DTOPC	The JTAG TSO signal that is an output from the target board silicon to the tool.

Index

O

P

R

S

T

U

V

W

Z