OPB Interrupt Controller (v1.00c)

DS473 December 1, 2005 **Product Specification**

Introduction

An Interrupt Controller is composed of a bus-centric wrapper containing 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 the OPB (On-chip Peripheral Bus) Bus. It can be used in embedded PowerPC™ systems (Virtex-II Pro devices), and in MicroBlaze™ soft processor systems.

In this document, IntC and Simple IntC are used interchangeably to refer to functionality or interface signals common to all variations of the Simple Interrupt Controller.

Features

- Priority between interrupt requests is determined by vector position. The least significant bit (LSB, in this case bit 0) has the highest priority.
- Modular design provides a core interrupt controller functionality instantiated within a bus interface design
- 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 configurable up to the width of data bus

LogiCORE™ Facts		
Core Specifics		
Supported Device Family	QPro™-R Virtex™-II, QPro Virtex-II, Spartan™-II, Spartan-IIE, Spartan-3, Spartan-3E, Virtex, Virtex-II, Virtex-II Pro, Virtex-4, Virtex-E	
Version of Core	opb_intc	v1.00c
Resources Used		
	Min	Max
I/O	55	116
LUTs	42	395
FFs	63	342
Block RAMs	0	0
Provided with Core		
Documentation	Product Specification	
Design File Formats	VHDL	
Constraints File	N/A	
Verification	N/A	
Instantiation Template	N/A	
Reference Designs	None	
Design Tool Requirements		
Xilinx Implementation Tools	5.1i or later	
Verification	N/A	
Simulation	ModelSim SE/EE 5.6e or later	
Synthesis	XST	
Support		
Support provided by Xilinx, Inc.		

Features (contd)

- Easily cascaded to provide additional interrupt inputs
- Interrupt Enable Register for selectively disabling individual interrupt inputs
- Master Enable Register for disabling 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

Functional Description

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 and 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. Interrupts may be programmed 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.

The two external interrupt inputs may be programmed 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 because 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-cacheable or cacheable 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 1 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.

A 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.

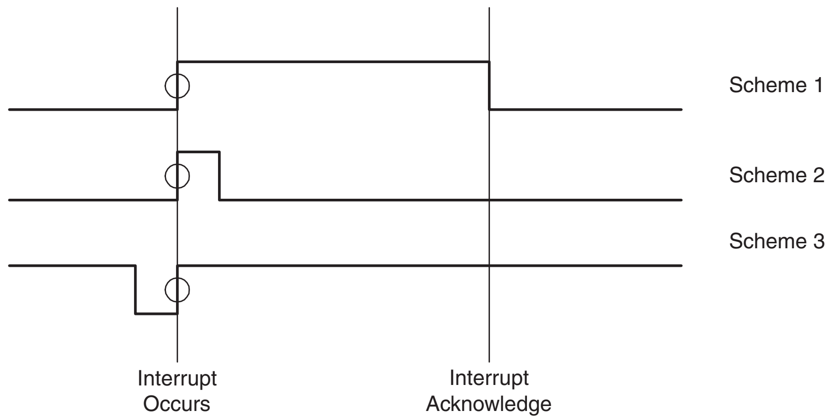

Figure 1: **Schemes for Generating Edges**

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.

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 cleared 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.

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.

Because 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 2.

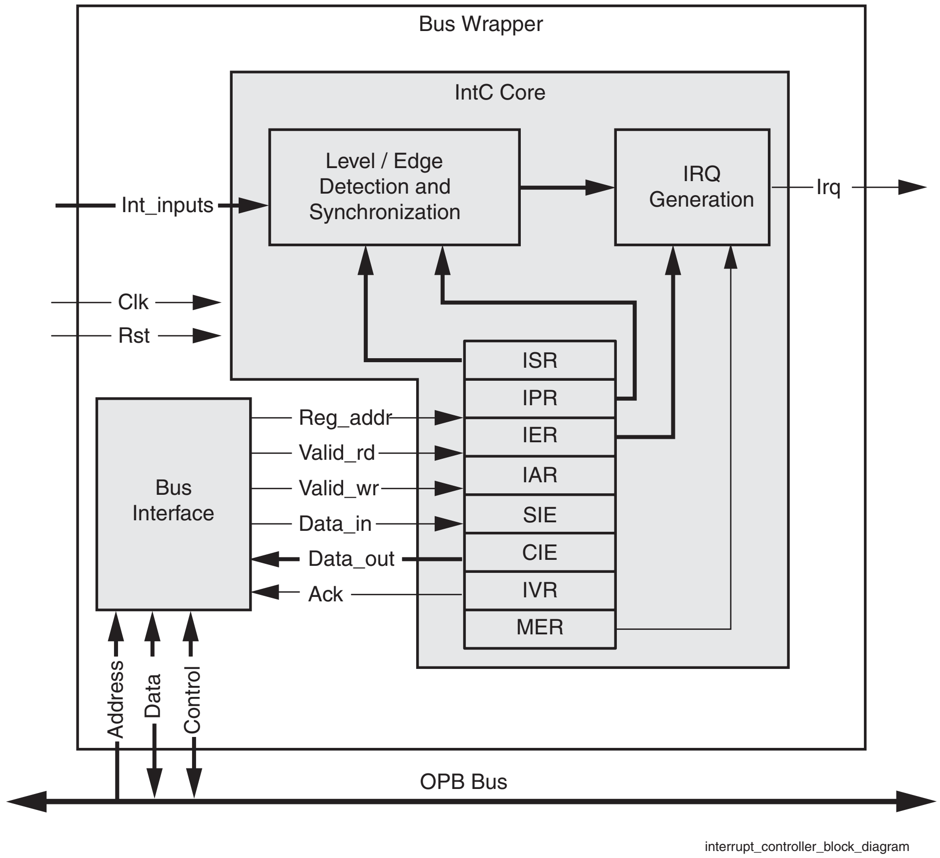

Figure 2: **Interrupt Controller Block Diagram**

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.

Because OPB addresses are byte addresses, OPB IntC register offsets are located at integral multiples of four from the base address. Table 1 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 3 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 4.

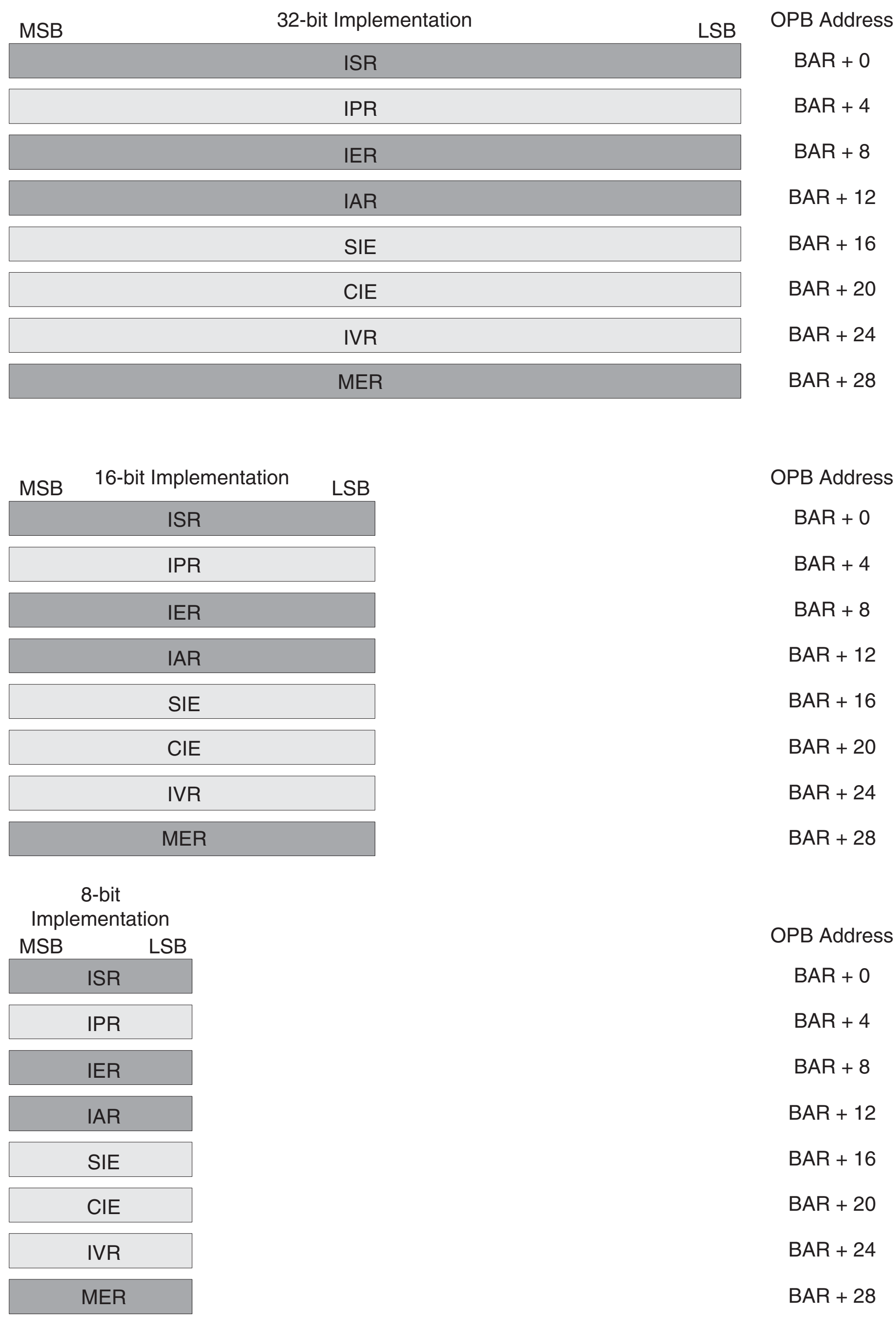

Figure 3: **OPB-based Register Offsets and Alignment**

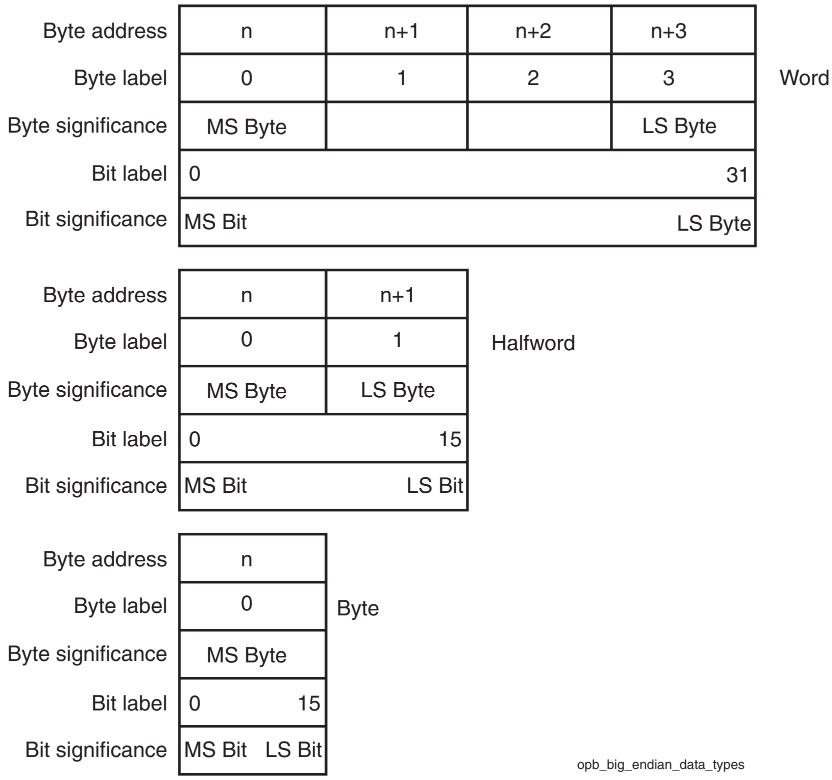

Figure 4: **OPB Big-Endian Data Types**

IntC Registers

The eight registers visible to the programmer are shown in Table 1 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 1: **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)

Table 1: **IntC Registers and Base Address Offsets**

Register Name	Abbreviation	OPB Offset
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 Interrupt Status Register (ISR) is shown in the following diagram and the bits are described in Table 2.

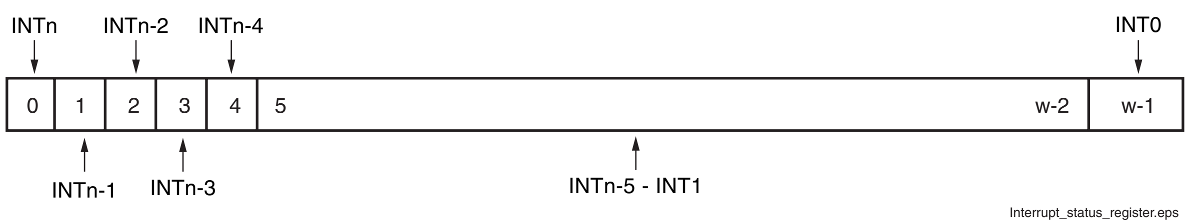

Figure 5: **Interrupt Status Register (ISR)**

Table 2: **Interrupt Status Register**

Bits	Name	Description	Reset Value
0 to (w –1)	INTn – INT0 ($n \le w - 1$) 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

Interrupt Pending Register (IPR) is shown in the following diagram and the bits are described in Table 3.

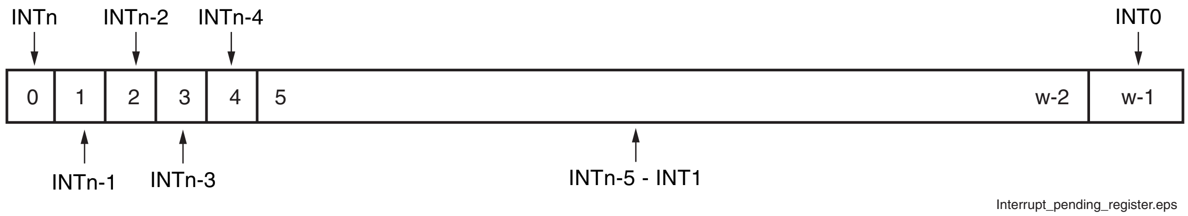

Figure 6: **Interrupt Pending Register (IPR)**

Table 3: **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, or masks, the corresponding interrupt input signal. Note however, that disabling an interrupt input is not the same as clearing it. Disabling an active interrupt blocks that interrupt from reaching the IRQ output, but as soon as it is re-enabled the interrupt will immediately generate a request on the IRQ output. An interrupt must be cleared by writing to the Interrupt Acknowledge Register as described below. Reading the IER 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 Interrupt Enable Register (IER) is shown in the following diagram and the bits are described in Table 4.

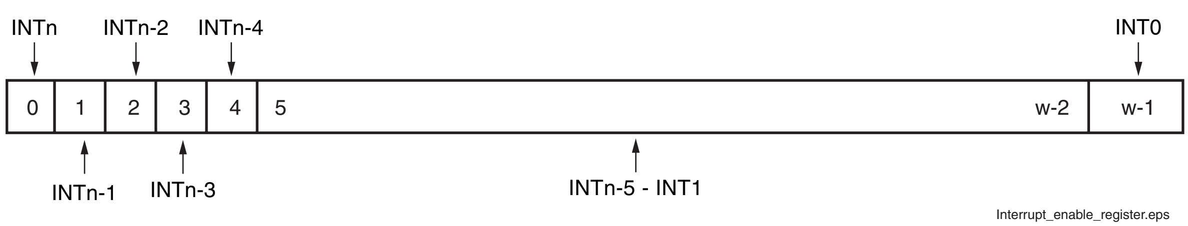

Figure 7: **Interrupt Enable Register (IER)**

Table 4: **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. An interrupt input that is active and masked by writing a 0 to the corresponding bit in the IER will remain active until cleared by acknowledging it. Unmasking an active interrupt will cause an interrupt request output to be generated (if the ME bit in the MER is set).

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 Interrupt Acknowledge Register (IAR) is shown in the following diagram and the bits are described in Table 5.

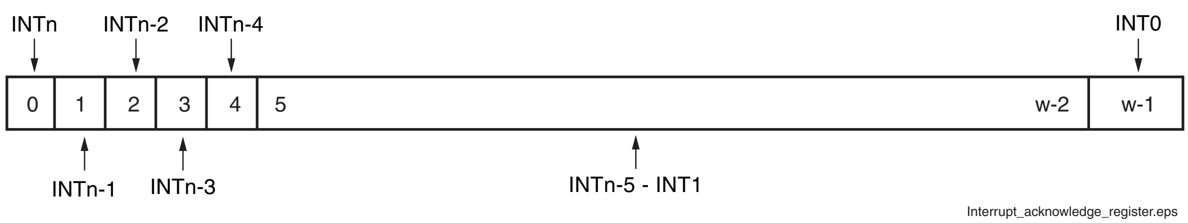

Figure 8: **Interrupt Acknowledge Register (IAR)**

Table 5: **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 Set Interrupt Enables (SIE) register is shown in the following diagram and the bits are described in Table 6.

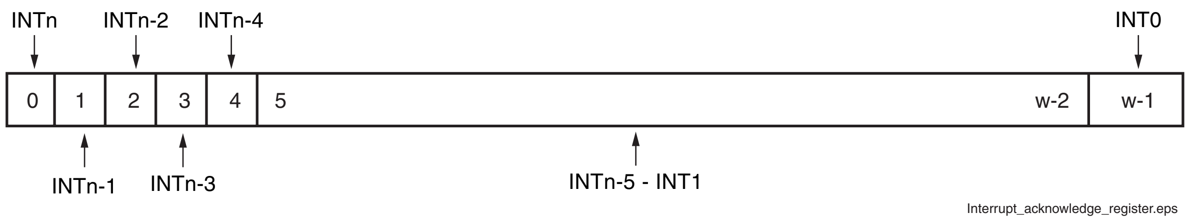

Figure 9: **Set Interrupt Enables (SIE) Register**

Table 6: **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 Clear Interrupt Enables (CIE) register is shown in the following diagram and the bits are described in Table 7.

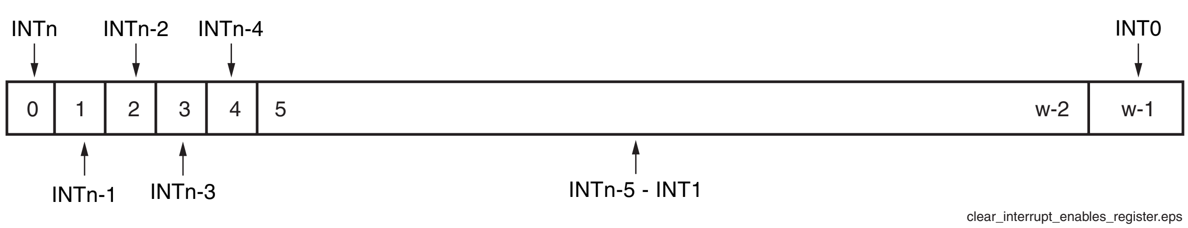

Figure 10: **Clear Interrupt Enables (CIE) Register**

Table 7: **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 Interrupt Vector Register (IVR) is shown in the following diagram and described in Table 8.

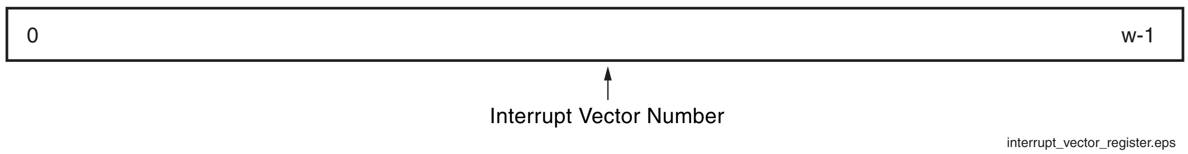

Figure 11: **Interrupt Vector Register (IVR)**

Table 8: **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 Master Enable Register (MER) is shown in the following diagram and is described in Table 9.

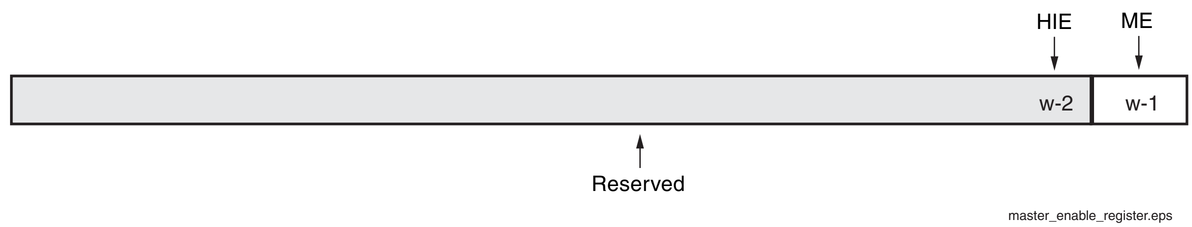

Figure 12: **Master Enable Register (MER)**

Table 9: **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 12. 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 12).

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. Note that disabling an interrupt by masking it (writing a zero to the IER) does not clear the interrupt. That interrupt will remain active but blocked until it is unmasked or cleared.

An interrupt acknowledge bit 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 10. Table 11 shows I/Os that are specific to an OPB IntC.

Table 10: **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(1)
Irq	out	IntC interrupt request output	Std_Logic	n/a

Notes:
1. Bit 0 is always the highest priority interrupt and each successive bit to the left has a corresponding lower interrupt priority.

Table 11: **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 can be parameterized:

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

Table 12: **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_BASEADDR must be a multiple of the range, where the range is C_HIGHADDR - C_BASEADDR +1.
C_NUM_INTR_INPUTS	Number of interrupt inputs.	Integer	1 up to the width of the data bus.

Table 12: **Generics (Parameters) Common to all IntC Instantiations *(Contd)***

Generic Name	Description	Type	Valid Values
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.
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. This means the low 5 address bits must be zero.
2. C_HIGHADDR is required to be at least C_BASEADDR + 31 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, 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 = 0x70000000
   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.
3. C_BASEADDR must be a multiple of the range, where the range is C_HIGHADDR - C_BASEADDR +1.

Table 13 lists the top level generics (parameters) that are present in an OPB IntC.

Table 13: **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	32

Core Usage in EDK

If you are using this core with the EDK design tools, see solution record SR18804 for details

Design Implementation

Device Utilization and Performance Benchmarks

Table 14 contains benchmark information for a Virtex-II Pro-7 FPGA using multipass place and route.

Table 14: **OPB_Intc FPGA Performance and Resource Utilization Benchmarks (Virtex-II Pro-7)**

Parameter Values					Device Resources			f_{MAX}
C_NUM_INTR_INPUTS	C_HAS_IPR	C_HAS_SIE	C_HAS_CIE	C_HAS_IVR	Slices	Slice Flip-flops	LUTs	f_{MAX} (MHZ)
1	0	0	0	0	69	86	48	149
1	1	0	0	0	55	64	41	222
1	0	1	0	0	54	63	40	201
1	0	0	1	0	54	63	40	213
1	0	0	0	1	55	65	42	200
1	1	1	1	1	54	66	45	209
1	0	0	0	0	55	63	42	215
2	1	1	1	1	62	72	50	220
2	0	0	0	0	59	68	45	225
4	1	1	1	1	81	90	70	196
4	0	0	0	0	74	82	60	208
8	1	1	1	1	121	126	113	158
8	0	0	0	0	105	110	95	156
16	1	1	1	1	196	198	196	152

Table 14: **OPB_Intc FPGA Performance and Resource Utilization Benchmarks (Virtex-II Pro-7)** ***(Contd)***

16	0	0	0	0	168	166	159	151
32	1	1	1	1	369	342	395	151
32	0	0	0	0	307	278	315	150

Revision History

Date	Version	Revision
04/11/01	1.0	Initial Xilinx release.
05/04/01	2.0	Changed to reflect EA IntC programmer interface
05/10/01	2.1	Added HIE to MER and changed functionality of ISR to match
09/17/01	2.2	Fixed errors in figures; added I/O and parameterization tables
10/04/01	2.3	Changed C_BASE_NUM_BITS to C_HIGHADDR and added note
11/07/01	2.4	Added Programming the IntC section
12/04/01	3.0	Changed to a core IntC with bus centric wrappers (opb_intc v1.00b and dcr_intc v1.00a); separate figures and tables for OPB and DCR
12/07/01	3.1	Changed DCR signal names to match internal standardization; added type and range columns to I/O tables and type and valid values to generics tables.
03/20/02	4.0	Updated for MDK 2.2
05/23/02	4.1	Update to EDK 1.0
07/22/02/	4.2	Add XCO parameters for System Generator
07/31/02	4.3	Add additional text to XCO parameters; fixed some typos; added text for IAR operation
09/11/02	4.4	Added resource utilization table
11/05/02	4.5	Rev hardware to 1.00c
01/08/03	4.6	Update for EDK SP3
07/24/03	4.7	Update to new template
07/28/03	4.7.1	Change DS number from 207 because of duplicates
12/05/03	4.8	Add notes to table 11 to fix CR 179857
01/22/04	4.8.1	Update page 1 and add Core Usage in EDK section for CR 182552
8/18/04	4.9	Updated for Gmm; updated trademarks and supported device family listing.
1/10/04	5.0	Converted to new DS template; updated images to Xilinx graphic standards; removed all conditional text; removed all references to the DCR interface.
12/1/05	5.1	Added Spartan-3E to supported family devices.