IRQ generator v1.1

FPGA core guide

Generated using
Vivado 2017.1
June 2018

Contents

Change history

Version	Author	Date	Change
1.0	Jan Lipponen	11.6.2018	Initial release
1.0	Jan Lipponen	25.6.2018	Thorough device tree documentation
1.1	Jan Lipponen	28.6.2018	Control register changes and bug fixes

1 Introduction

The IRQ generator FPGA core is developed using Xilinx Zynq 7000 series devices and the Xilinx Vivado Design Suite environment. The core enables level-sensitive type interrupt generation to interrupt lines between the PL (programmable logic) and the PS (processing system). The core is controlled through AXI4-Lite interface.

Features:

- AXI4 compliant
- Up to 16 interrupt lines (level-sensitive)
- Interrupt latency calculation
- 4 32-bit control/status registers
- Delivered as Verilog (.v) source

Device and interface support	
Supported devices	Zynq-7000
Supported user interfaces	AXI4-Lite
Core provided files	
Design files	Verilog
Testbench files	VHDL
Constraints file	-
Device tree entry	Linux device tree (.dts) entry
SW support	
SW driver files	Linux kernel driver module
Tested environment	
Vivado design suite	2017.1
Linux kernel	4.6, 4.9
SoC	Z-7010, Z-7020
Yocto	Rocko
meta-xilinx	Rocko

2 Overview

The IRQ generator consist of three main parts; the AXI4-Lite slave interface logic, the control and status registers and the IRQ generation logic. These functional blocks can be seen in the Figure 1. The core consists of two Verilog source files; the top level module *irq_generator.v* and the *irqgen_controller.v* submodule.

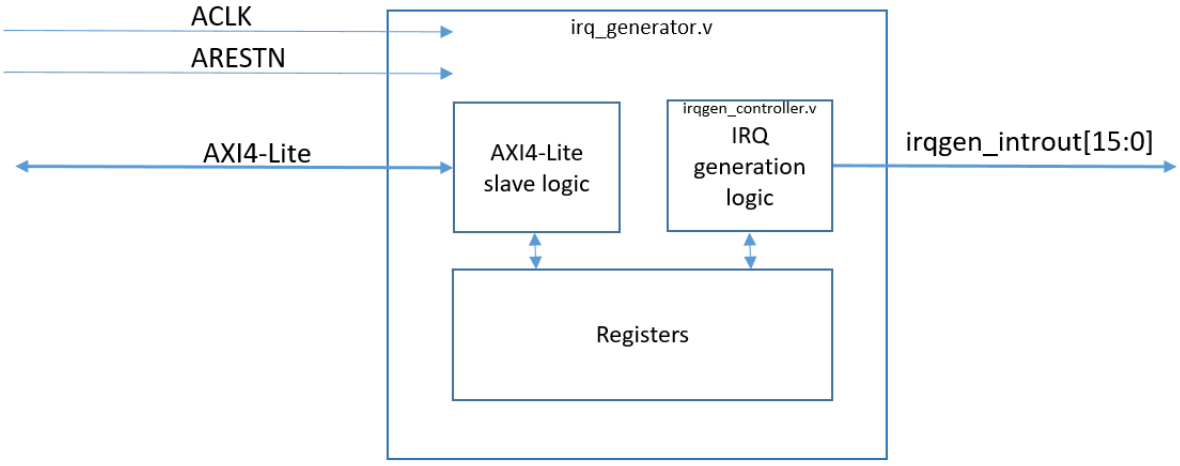

Figure 1: IRQ generator block diagram

The IRQ generation logic enables generation of user specified amount of high level-sensitive type interrupts. The core will keep an interrupt line asserted until the core is accessed from an interrupt handler and the interrupt is cleared accordingly. This procedure is specified in *Generic Interrupt Controller Architecture Specification*, by ARM [1, p. 38]. This is also briefly mentioned in the Zynq TRM (technical reference manual) page 229:

"For an interrupt of level sensitivity type, the requesting source must provide a mechanism for the interrupt handler to clear the interrupt after the interrupt has been acknowledged. This requirement applies to any IRQF2P[n] (from PL) with a high level sensitivity type." [2, p. 229]

The IRQ generator core can be programmed to generate interrupts to one interrupt line at the time. After each served interrupt, the latency is stored in a 32-bit register. The latency counter starts when an interrupt is asserted to an interrupt line and stops when the core is accessed by the user and the line gets deasserted. The latency register will be overwritten after next successfully handled interrupt.

3 Register space

All the IRQ generator registers use Little Endian format, as shown in the Figure 2.

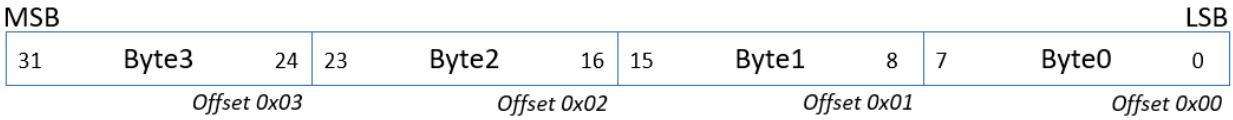

Figure 2: Little Endian 32-bit register example

All the IRQ generator AXI4-Lite accessible registers are listed on the Table 1. All the offsets are relative to the assigned base address of the AXI4-Lite interface, managed by Vivado.

Table 1: IRQ generator register address map

Address offset	Name	Access (write/read)	Description
0x0000	IRQ_GEN_CTRL_REG	write/read	Core control register
0x0004	IRQ_GEN_GENIRQ_REG	write	IRQ generation control register
0x0008	IRQ_GEN_IRQ_COUNT_REG	read	Total IRQ count status register
0x000C	IRQ_GEN_LATENCY_REG	read	Last served IRQ latency status register

3.1 IRQ_GEN_CTRL_REG – Offset 0x0000

The IRQ generator control register (0x0000) provides fields seen in the Figure 3 for the core control and IRQ handling. Explanation of these fields are listed on the Table 2.

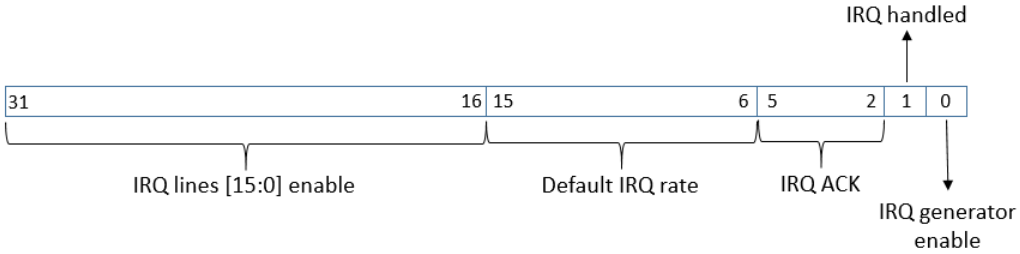

Figure 3: IRQ_GEN_CTRL_REG register

Before using the IRQ generator core it needs to be enabled by writing to the “IRQ generator enable” bit as 1.

Table 2: IRQ_GEN_CTRL_REG register details

Bits	Field name	Default value	Description
0	IRQ generator enable	0b0	When written to 1, the IRQ generator is able to generate IRQs by programming the IRQ_GEN_GENIRQ_REG.
1	IRQ handled	0b0	This bit should be written to 1 from the interrupt handler stating that the interrupt is served. This bit is always written back to zero on next clock cycle.
2 to 5	IRQ ACK	0b0000	This field should be written to matching interrupt line (0 to 15) from the interrupt handler at the same write as the “IRQ handled” bit.
6 to 15	Default IRQ delay	0b0000000000	Not supported, writing to these bits has no effect.
16 to 31	IRQ lines [15:0] enable	0x0000	Not supported, writing to these bits has no effect.

3.2 IRQ_GEN_GENIRQ_REG – Offset 0x0004

The IRQ generator generate IRQs register (0x0004) is used to program the core to generate specified amount of IRQs to an IRQ line (0 to 15). Fields of this register can be seen in Figure 4 and explanations of these fields are listed on the Table 3.

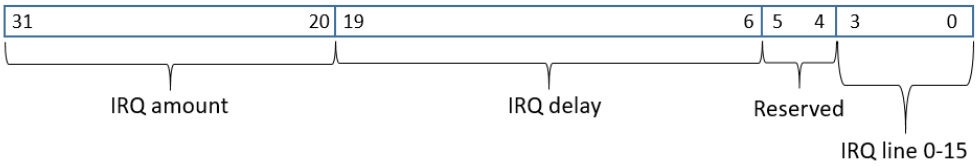

Figure 4: IRQ_GEN_GENIRQ_REG register.

IRQ generation is started by writing a valid amount (1 to 65535) of IRQs to the “IRQ amount” field. If no other values are specified, the core will generate desired amount of IRQs to the line 0 with IRQ delay 0. IRQ delay is equal to amount of clock edges of delay before new IRQ is asserted after the previous one is served.

Table 3: IRQ_GEN_GENIRQ_REG register details

Bits	Field name	Default value	Description
0 to 3	IRQ line 0-15	0b0000	This field should be written to match the desired IRQ line (0 to 15). No rewrite is needed if user wants to generate new interrupts to the same line.
4 to 5	Reserved	0b00	Writing to these bits has no effect.
6 to 19	IRQ delay	0b0000000000	This field should be written to match the desired IRQ delay (0 to 16383). This value is equal to the amount of clock cycles before the next interrupt is generated after the previous one is served successfully.
20 to 31	IRQ amount	0x0000	This field should be written to match the desired amount of IRQs to be generated (1 to 4095). Writing to this register triggers the IRQ generation.

3.3 IRQ_GEN_IRQ_COUNT_REG – Offset 0x0008

The IRQ generator IRQ count register consist of a single field; the total generated IRQ count as seen in the Figure 5 and on the Table 4.

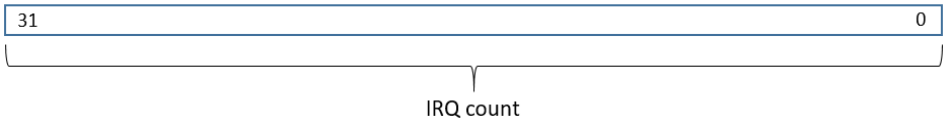

Figure 5: IRQ_GEN_IRQ_COUNT_REG register

The IRQ count field is always incremented when some IRQ line [15:0] gets asserted.

Table 4: IRQ_GEN_IRQ_COUNT_REG register details

Bits	Field name	Default value	Description
0 to 31	IRQ count	0x00000000	This read-only field contains the amount of total generated IRQs by the IRQ generator.

3.4 IRQ_GEN_LATENCY_REG – Offset 0x000C

The IRQ generator latency register consists of a single field; the latency of last successfully served IRQ as seen in the Figure 6 and on the Table 5.

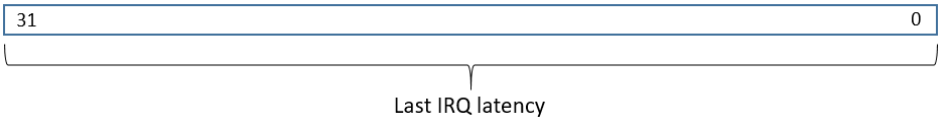

Figure 6: IRQ_GEN_LATENCY_REG register

The latency is derived by calculating the positive clock edges between an interrupt line assertion and IRQ handling by user. This value is then stored to the Last IRQ latency field – always over writing the previous latency value. With 100 MHz FPGA clock each edge is equal to 10 ns.

Table 4: IRQ_GEN_LATENCY_REG register details

Bits	Field name	Default value	Description
0 to 31	Last IRQ latency	0x00000000	This read-only field contains the latency of last successfully served IRQ in clock edges relative to the used FPGA clock.

4 Using the core

This FPGA core is intended to demonstrate the Zynq SoC interrupt capabilities with custom logic. The core is directly accessible from the PS side via the AMBA bus and the interrupt output is intended to be directly connected to the *Shared Peripheral Interrupts* port *IRQF2P* [2, p. 230]. This scheme can be seen in the Figure 7.

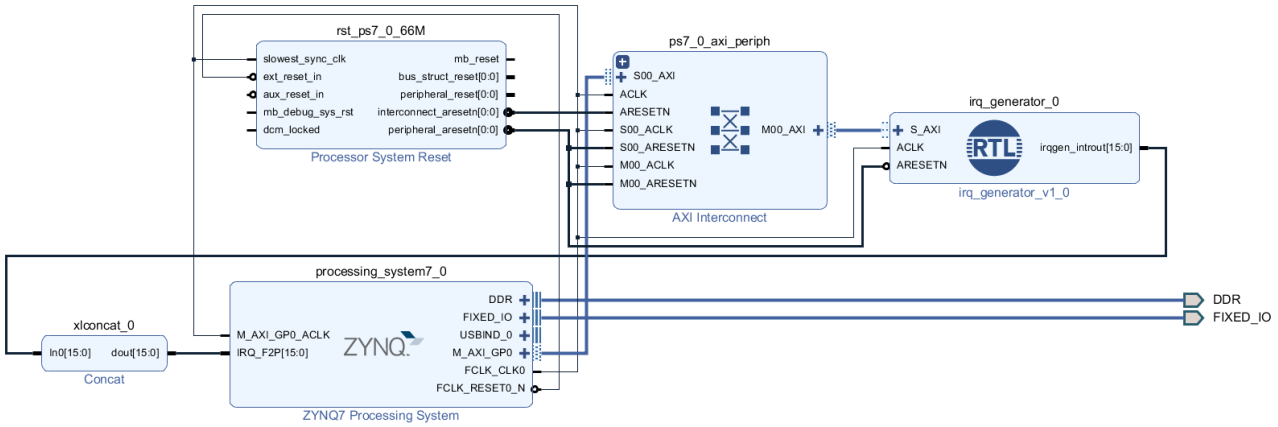

Figure 7: The IRQ generator containing Vivado block design

4.1 Programming sequence

The IRQ generator requires in minimum two register write transactions before any interrupt line is asserted. User should then serve each interrupt by writing to the control register.

Generating IRQs:

1. Enable the IRQ generator by writing the lowest bit of the IRQ_GEN_CTRL_REG (0x0000) to 1.
2. Write the desired amount of interrupts to the IRQ_GEN_GENIRQ_REG (0x0004) highest 16 bits (16 to 31). Optionally write the desired interrupt line to the 4 lowest bits (0 to 3) and desired IRQ delay to the bits (6 to 15), both being 0 by default.

Handling IRQs:

1. Write the “IRQ handled” field of the IRQ_GEN_CTRL_REG (0x0000) as 1 (bit 1). At the same time the interrupt line should be stated by writing the line 0-15 to the “IRQ ACK” field (2 to 5). Also, the “IRQ generator enable” (bit 0) should be written to 1 to keep the IRQ generator enabled.

5 Test bench

The test bench includes to VHDL source files; the top level design (irq_generator_tb.vhd) instantiating the DUV (irq_generator.v) and an AXI4-Lite master module (axi_master.vhd) for input generation. This scheme can be seen in the Figure 8.

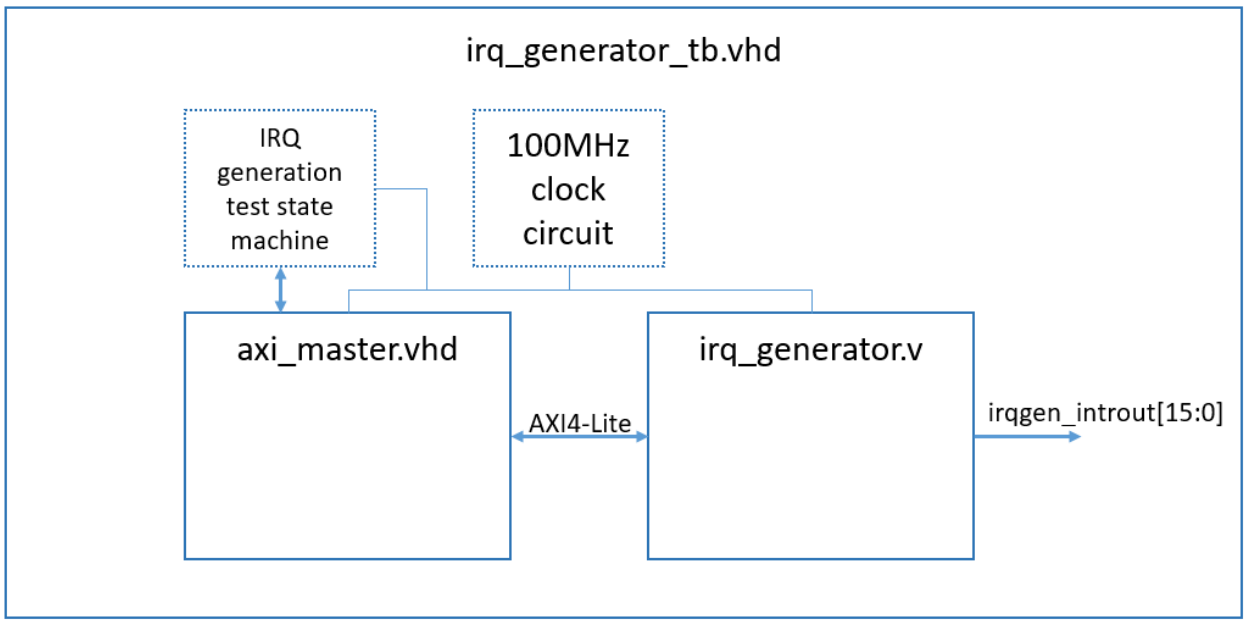

Figure 8: The IRQ generator test bench block design

The test bench implemented state machine enables generation and handling of predefined amount of IRQs to one IRQ line defined by the *gen_irqs_c* constant.

6 Linux device tree entry

Registering the IRQF2P interrupts to a Linux kernel requires a device tree entry for the IRQ generator core with the information about the used IRQs and address space of the device. The device tree entry for the IRQ generator device should be placed under the AMBA bus node.

The entry for the IRQ generator should specify the base address and range for the device, define the used IRQs and ACK values, the compatible string and the interrupt parent:

```
irq_gen@43C00000 {
            compatible = "wapice,irq-gen";
            interrupt-parent = <0x4>;
            #interrupt-cells = <0x3>;
            interrupts = <0x0 0x1d 0x4 0x0 0x1e 0x4 0x0 0x1f 0x4 0x0 0x20 0x4
                          0x0 0x21 0x4 0x0 0x22 0x4 0x0 0x23 0x4 0x0 0x24 0x4
                          0x0 0x34 0x4 0x0 0x35 0x4 0x0 0x36 0x4 0x0 0x37 0x4
                          0x0 0x38 0x4 0x0 0x39 0x4 0x0 0x3a 0x4 0x0 0x3b 0x4>;
            wapice,intrack = <ACK1d ACK1e ACK1f ACK20 ACK21 ACK22 ACK23 ACK24
                              ACK34 ACK35 ACK36 ACK37 ACK38 ACK39 ACK3a ACK3b>;
            reg = <0x43c00000 0x10000>;
};
```

- *compatible*
  The *compatible* property is used for device driver selection. Using this string the platform driver can read data out of the device tree. [3, p. 13]

- *interrupt-parent*
  The *interrupt-parent* property should be used by all devices capable of interrupt generation. It is a *phandle* value pointing to an interrupt controller node in the device tree. [3, p. 17]

- *#interrupt-cells*
  The *#interrupt-cells* property specifies the amount of unsigned integer values needed to encode an *interrupt specifier*, each describing one interrupt source [3, p. 18]. The ARM generic interrupt controller uses 3 cells to specify an interrupt: 1st cell for interrupt type (0 for SPIs (shared peripheral interrupt) and 1 for PPIs (private peripheral interrupt) interrupts), 2nd cell for the interrupt number and the 3rd for interrupt type (1 for low-to-high edge, 2 for high-to-low edge, 4 for active high level-sensitive and 8 for active low level-sensitive) [4].

- *interrupts*
  The *interrupts* property lists the *interrupt specifiers* for all used interrupts [3, p. 19].

- *wapice,intrack*
  The *wapice,intrack* is a custom property holding the interrupt acknowledgment values needed for interrupt clearing from the device (value needed to be written to the “IRQ ACK” field in the IRQ_GEN_CTRL_REG register). They are listed in the same order as their corresponding interrupts in the *interrupts* property. These values are defined in a separate dt-bindings header file *irq_gen_bindings.h*.

- *reg*
  The *reg* property specifies the base address and the length of the device's address space. [3, p. 15]

The interrupt numbers specified by the 2nd cell of each *interrupt specifier* are not trivially resolvable. The Zynq technical reference manual lists the SPI interrupts in the Table 7-4 and the IRQF2P[15:0] port is mapped to IRQ IDs 61-68 and 84-91. However, these values cannot be used in the device tree directly.

The ARM GIC architecture specification v1.0 defines the interrupt IDs for SPI interrupts to be in the range [32, 1019], while the bottom IDs [0, 31] are reserved for SGIs (software generated interrupt) [0, 15] and for PPIs [16, 31] [1, p. 20]. These values actually matches the IRQ IDs listed in the Zynq TRM, but the device tree expects the SPI and PPI interrupt numbers to start from value 0; the SPI interrupt numbers are defined to be in the range [0, 987] and the PPI interrupts in the range [0, 15] [4]. Overlapping ranges are possible because the interrupt type is first selected in the 1st cell of each *interrupt specifier*.

Due to the usage of interrupt numbering starting from 0 in the device tree the interrupt IDs specified by the ARM GIC architecture specification needs to be adjusted accordingly. 32 needs to be extracted from SPI type IDs and 16 from PPI type IDs: the lowest SPI IRQ ID (32) maps to the interrupt number 0 as does the lowest PPI IRQ ID (16).

References

[1] Arm Holdings, ARM Generic Interrupt Controller Architecture Specification 1.0, B.b ed., Arm Holdings, 2013, p. 214.

[2] Xilinx, Inc., Zynq-7000 All Programmable SoC Technical Reference Manual, 1.11 ed., Xilinx, Inc., 2016, p. 1863.

[3] devicetree.org, Devicetree Specification Release v0.2, 2017, p. 57.

[4] Linux Kernel Organization, Inc, "ARM Generic Interrupt Controller," Dec 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/devicetree/bindings/interrupt-controller/arm%2Cgic.txt. [Accessed Jun 2018].