Play project

1 Introduction

This project is meant as an example of how one can read and write to address-mapped registers implemented in hardware for a SPEC board. In short, the Simple PCIe FMC Carrier (SPEC) is an FMC (FPGA Mezzanine Card) carrier card which can be introduced in one of the PCIe slots in your PC. Using the FPGA on the SPEC, one can control the various peripheral devices on-board the SPEC, as well as the potential mezzanine card connected to the SPEC. More detailed information about the SPEC can be found here.

As we can see from the link above, the SPEC card has a Gennum GN4124 PCIe interface on it; this is a PCIe interface chip that provides a bridge to various FPGA logic, thus giving us the capability of controlling this logic from a PC.

Let us start by describing the folder structure for the project, followed by a description of the hardware design. Then, we'll talk a bit about the software. We will finish by going through a short example of running the software.

2 Folder structure for the project

The folder structure for our simple project, called *play* is as shown below:

play/

→ doc/

→ hdl/

  → design/

  → tb/

→ sim/

→ syn/

→ sw/

In summary, here's what each folder contains:

doc/	The present document
hdl/	HDL files for the design
hdl/design/	HDL modules generally used for synthesis
hdl/tb/	Testbenches for various design modules
sim/	Simulation-specific files, such as simulation scripts, simulator log files, etc.
syn/	Synthesis-specific files, such as the main project file for Xilinx ISE, .ucf files, etc.
sw/	Python scripts for running tests on the FPGA design

3 Design description

A simplified block diagram of the design is presented in Fig. 1. It contains a few VHDL modules communicating via Wishbone buses. The *gn4124_core* component is used to decode the signals from the Gennum chip and send them via the Wishbone interface. In this purpose, the Gennum core has one Wishbone master controlling signals to an address decoder slave (the *addr_dec* component). This address decoder acts as a Wishbone master on a separate bus, which has the *led_ctrl* and *irq_controller* components as slaves.

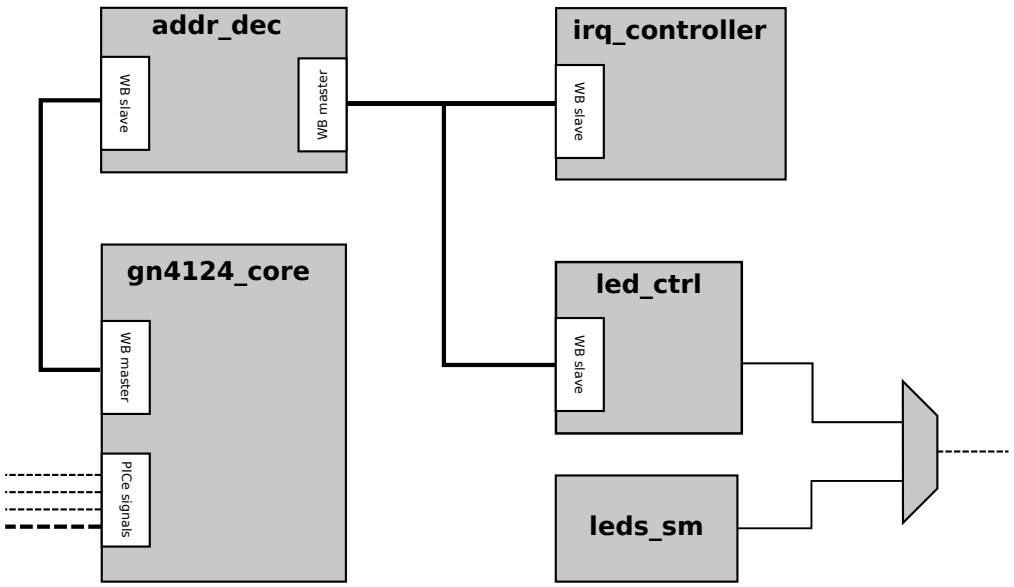

Figure 1: Design block diagram

Signals from the user (via the Gennum chip and core) will be decoded by the address decoder and the appropriate Wishbone slave selected. A very simple address mapping is used throughout this project, as seen in Tabel 1. By this mapping, a write to address *0x10* sets LED1A on the SPEC card front panel, and a write to address *0x14* sets LED1B on the front panel on.

Table 1: Address map for the *play* project

0x00	IRQ_MULTI_IRQ
0x04	IRQ_SRC
0x08	IRQ_EN_MASK
0x0C	*reserved*
0x10	LED1A
0x14	LED1B

In order to test and play with IRQ's, an IRQ controller is also included. The controller supports up to 32 interrupt sources by enabling the appropriate bit in the 32-bit IRQ_EN_MASK register. In our project, a write to LED1A triggers interrupt source 0, and a write to LED1B triggers interrupt source 1. Reading the IRQ_SRC register indicates which interrupt source was triggered, and writing the appropriate bit in the register clears the interrupt source.

Last, the (inappropriately-named) *leds_sm* block is a simple block providing a binary counter on the two front panel LEDs (LED1A and LED1B) ofx the SPEC board. The binary up-counter counts roughly once per second, providing information to the user that the bitstream downloaded to the FPGA works as intended.

As seen in Fig. 1, the output of the *leds_sm* block is multiplexed with the output of the *led_ctrl*

block. The multiplexer output is switched and remains connected to the output of the *led_ctrl* block as soon as the user initiates a command to write to one of the LED registers.

This design is used as a reference design for the *hdlmake* tool. If you haven't used *hdlmake* before, the tutorial on how to run it using this project can be found here.

4 Software

The *sw/python/* folder contains a couple of scripts useful for this project. The *bitstream_load.py* script is useful for downloading a bitstream (*note:* this should have a *.bin* extension) to your FPGA. The *play_leds.py* script is the topic of this section, and is used to read and write to the registers listed in Table 1. Both these scripts use methods defined inside classes of the PTS tool. You can download PTS by cloning its git repository; the following Linux command would do this:

```
git clone git://ohwr.org/misc/pts.git <folder of choice>
```

Note: You will need *git* to be able to run this command. Follow the instructions here if you don't have git installed.

Now let's take a look at the contents of the *sw/python/* folder. You can find a script called *play_leds.py*; running this script would ask you if you want to read from or write to the board, and which register you wish to read/write. As stated in the previous section, writing to a LED register would cause an interrupt. The following is an example output of running the script and writing to one of the LED registers:

```
Mask: 0x3
(w)rite/(r)ead/(q)uit? r
READ
reg? 0
value: 0x0
(w)rite/(r)ead/(q)uit? w
WRITE
reg? 0
val? 1
interrupt!
src : 0x1
wrote to LED 1
value 0x1
(w)rite/(r)ead/(q)uit? r
READ
reg? 0
value: 0x1
(w)rite/(r)ead/(q)uit? w
WRITE
reg? 4
val? 1
interrupt!
src : 0x2
wrote to LED 2
value 0x1
(w)rite/(r)ead/(q)uit? r
READ
reg? 4
value: 0x1
(w)rite/(r)ead/(q)uit? q
quitting
```

Note that the script outputs LED 1 for LED1A and LED 2 for LED1B.

Now let us take a look at what's inside the script. First of all we create a *Gennum* object and enable interrupts with a method inside the *Gennum* class:

```
# Create a SPEC object using RawRabbit driver
spec = rr.Gennum()
spec.irqena()
```

Then, using the *CCSR* class in the *csr* module of PTS, we create a couple of register classes, using the base addresses of the IRQ controller and respectively LED controller:

```
# Define the registers' base addresses
irq_regs = CCSR(spec, 0)
led_regs = CCSR(spec, 16)
```

Since writing to LED registers LED1A and LED1B create interrupts on sources 0 and respectively 1, writing *0x3* to the IRQ_EN_MASK register enables these two interrupts:

```
# Enable interrupt on LED write
irq_regs.wr_reg(IRQ_EN_MASK, 0x3)
msk = irq_regs.rd_reg(IRQ_EN_MASK)
```

Assuming basic knowledge of Python or object-oriented programming, the rest of the program should be pretty easy for you to understand. One final thing that should be pointed out is that you should feed the *wr_reg* method of the *CCSR* class addresses that are *offsets from the base address*. We already saw an example of this in the output of the *play_leds.py* script, where instead of giving address 16 for LED1A, we gave it address 0, and when we wanted to light LED1B, we gave it address 4.

Hopefully, you now have a basic understanding of how to access registers using PTS. Many modules exist for accessing different types of peripherals on the SPEC, you should have downloaded these when you cloned the *git* repository. Check them out for more details and ideas. You can also check out the *test/* folder under your PTS clone folder for examples on using PTS.