MSE - TSM_EmbHardw

Extend the System with Parallel Port and Timers

TABLE OF CONTENTS

Introduction

This document guides you through several step in order to implement a parallel port (PIO) as a peripheral for the SoPC. Unlikely of first exercise we will not use an IP form the library. You will design the peripheral by your own instead. Therefore we are going to describe it's behavior and interface by using VHDL. While extending the system we will add a timer as well. Timers provide a particular feature. Remember how we delayed the count loop in first lab. This method is very bad practice particularly in embedded software development. To avoid such **poor** delay mechanisms we will generate a timer interrupt which triggers our counter.

- Read the course slides describing how to develop a IP core
- Develop the interface and address map
- Design and test the core. If OK, hook it up to the system
- Read-out i.e the buttons by using the PIO
- Add a timer read it's data-sheet to get necessary information to configure
- Extend firmware to use PIO as well as the timer

1 Parallel Port Design

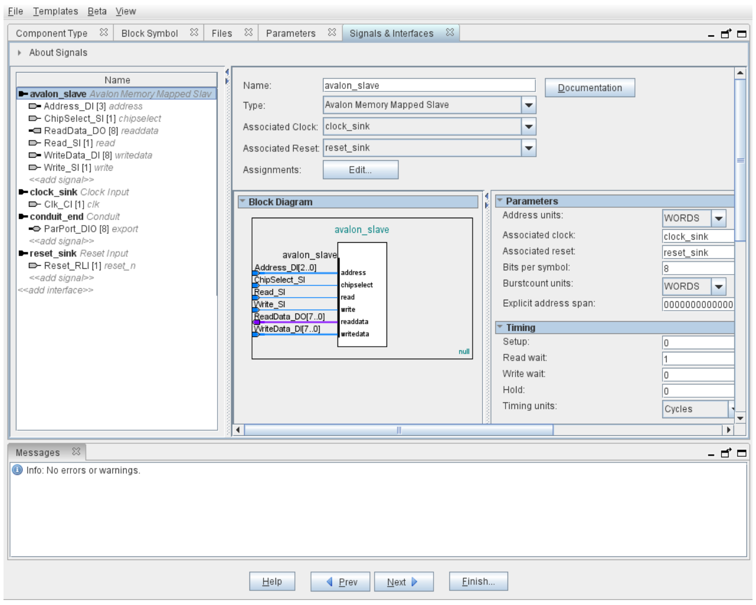
1.1 VHDL description

Create a VHDL file for the parallel port – name it *parPort.vhdl*, and describe in there the entity and architecture. Refer to the course slides for this. *(You may want copy the lab1 files. Feel free to do so. ↪ **Copying means, open lab1 and save as lab2, otherwise naming problems may occur.**)*

1.2 Attach the VHDL Bloc to the SoPC

- Write the VHDL by using your favorite editor. If you have no favorite use Emacs with EHT's[1] VHDL extension. On Debian is the installation as simple as run following command : *(sudo apt-get update && sudo apt-get install emacs emacs-goodies-el)*
- In Qsys: Go to "Project" ↪ "new component"
- In the "Component Type" tab introduce a significant name for new IP.
- In the "Files" tab include the VHDL file(s) as synthesis file and click on "**Analyse Synthesis Files**". You should end up with the window "Analyzing Synthesis Files Completed" free of errors. In case of errors correct those first. You will also end up with error messages on the "Component Editor" window. This is normal, they will be fixed later. **Do not add any file for simulation.**
- In the "Signals & Interfaces" tab, you'll find at the left a list of interfaces (in bold), each composed of different signals. These interfaces and signals have been automatically infered from their name, but usually they are wrong! It is only you that knows their correct functionality. Add (or remove) interfaces in order to end up with an `Avalon Slave`, a `Clock` input, a `Reset` input and a `Conduit_end` (external signal). See in figure 1 for more specific information.
- Each signal must be associated to an interface. Move them to match your expected port structure. Check the functionality of ports in order to make sure that the infered functionality matches with the code functionality.
- At the end you should end up with a view similar to the one shown in figure 1.
- When selecting the Avalon interface, verify that the timing fits with the type of access defined in the VHDL (for instance, number of clock cycles per read)[2]. Finally, check that every interface has a valid associated **clock** and **reset**.
- When finished, connect it to your SoPC by following the same procedure as first lab, you can keep or replace the previously used parallel port. Keep in mind that peripherals are accessible from the data bus and not from the instruction bus!
  NOTE: Whenever you upgrade your "component", delete the peripheral from the SoPC and insert it again.
- Back to Quartus, upgrade the schematic, and rerun the synthesis and P&R.

[1]More details are given by the course slides.
[2]Remember we designed two different read implementations one with zero delay and an other with a one clock cycle delay. It depends on the implementation which is the right choice but implementation relays on specs and capability so the right choice should be clear from the beginning.



```
Name
avalon_slave Avalon Memory Mapped Slav
    Address_DI [3] address
    ChipSelect_SI [1] chipselect
    ReadData_DO [8] readdata
    Read_SI [1] read
    WriteData_DI [8] writedata
    Write_SI [1] write
    <<add signal>>
clock_sink Clock Input
    Clk_CI [1] clk
conduit_end Conduit
    ParPort_DIO [8] export
    <<add signal>>
reset_sink Reset Input
    Reset_RLI [1] reset_n
    <<add signal>>
<<add interface>>
```

Name:	avalon_slave
Type:	Avalon Memory Mapped Slave
Associated Clock:	clock_sink
Associated Reset:	reset_sink
Assignments:	Edit...

Parameters

Address units:	WORDS
Associated clock:	clock_sink
Associated reset:	reset_sink
Bits per symbol:	8
Burstcount units:	WORDS
Explicit address span:	0000000000000

Timing

Setup:	0
Read wait:	1
Write wait:	0
Hold:	0
Timing units:	Cycles

```
Info: No errors or warnings.
```

Figure 1: User IP, port interface (Avalon and FPGA IO)

1.3 Write the C Program

Back to Eclipse, regenerate the BSP first[3]! Use the functions `IOWR_8DIRECT` and `IORD_8DIRECT` defined in the **file "`io.h`"** in order to access the peripheral registers that you defined earlier.

- During the initialization set the port as output and then reproduce the counter sequence.

Keep in mind that the "BASE" address of the peripheral is defined in the **file "`system.h`"** generated by the BSP-Wizard. The address offset is specific by the address map defined in the VHDL parallel port architecture. See the address map table to get information on which address a certain functionality (register) is located.

- If it doesn't work debug the system with Eclipse and the SignalTap Analyzer.

2 Timer and Interruption (IRQ)

In order to count based on a predictable pulse we extend the system by a timer. The timer should generate a interrupt. Based on the interrupt we increment the count value. That will end in a much better counter implementation. Because we will avoid the busy wait, implemented in lab1. The busy wait is bad because we block the CPU while waiting hence we can not use the processor to compute other thing while being blocked.

2.1 In Qsys

In Qsys insert a component "Interval Timer". Parametrize the timer in order to fix the period to 10 ms, with a 32 bit counter, with start/stop control, fixed period, and readable snapshot. Attach it to the SoPC. Make sure the timer **IRQ is connected to the Nios II** processor.
Like any peripheral the timer can be controlled through a register model. Please refer to the document "Embedded Peripherals IP User Guide", for a detailed description of the register model. (Available online `http://www.altera.com/literature/ug/ug_embedded_ip.pdf`).

[3]If hardware system change. You must regenerate BSP. The BSP is the connection between hard- and software it consists information to address mapping, IP drivers etc.

- Tables 28-4 and 28-6 consists all the information needed to enable the timer. Keep in mind that the timer registers are 16-bit in order to access them correctly[4].

2.2 Using Interruption in C

For using interruptions in the firmware (Eclipse code), you must include the IRQ library. *(Libraries related to Nios II system features are in sub-folder "sys" located.)*

- `#include "sys/alt_irq.h"`

In the main function initialize the interrupt controller first. It is well known initialization must be done early hence call the init-function right at the begin of main. Follow the procedure below to do so:

- `alt_irq_register(TIMER_IRQ, &counter,(alt_isr_func)timer_interrupt_handler);`
- This function will map the IRQ receiver in `TIMER_IRQ` to the interruption handler `timer_interrupt_handler`.
- Afterwards, lets describe the interrupt handler, here is an incomplete example:

```c
// ...
void timer_interrupt_handler(void *context, alt_u32 id){
  volatile int *counter_ptr = (volatile int *)context
  (*counter_ptr)++; // increase the counter;
  // write counter value on the parallel port;
  IOWR_8DIRECT(PARALLEL_PORT_BASE, 1, *counter_ptr);
  // acknowledge IRQ on the timer;
  //...
}
// ...
```

You have to start the timer and enable IRQs from the main function, and you must also acknowledge IRQs from the timer interrupt handler. Use the following function, for instance:

- `IOWR_16DIRECT(TIMER_BASE,4,0x4);`

Have a look to the control register description in data-sheet in order to enable/disable interruptions and other functionality.

[4]Remember course slides about register remapping