h e p i a

Haute école du paysage, d'ingénierie
et d'architecture de Genève

Prof. A. Upegui

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences
Western Switzerland

# Building your own SoPC peripherals : Parallel Port and LCD interface

## Introduction

This documents lists step by step a sequence of operations in order to implement a parallel port (GPIO) as a peripheral for your System on Programmable Chip. The peripheral architecture is described in VHDL.

Afterwards, you'll be able to follow the same procedure for an LCD interface that will permit to display an image previously stored in memory.

## Parallel Port design - Week 3

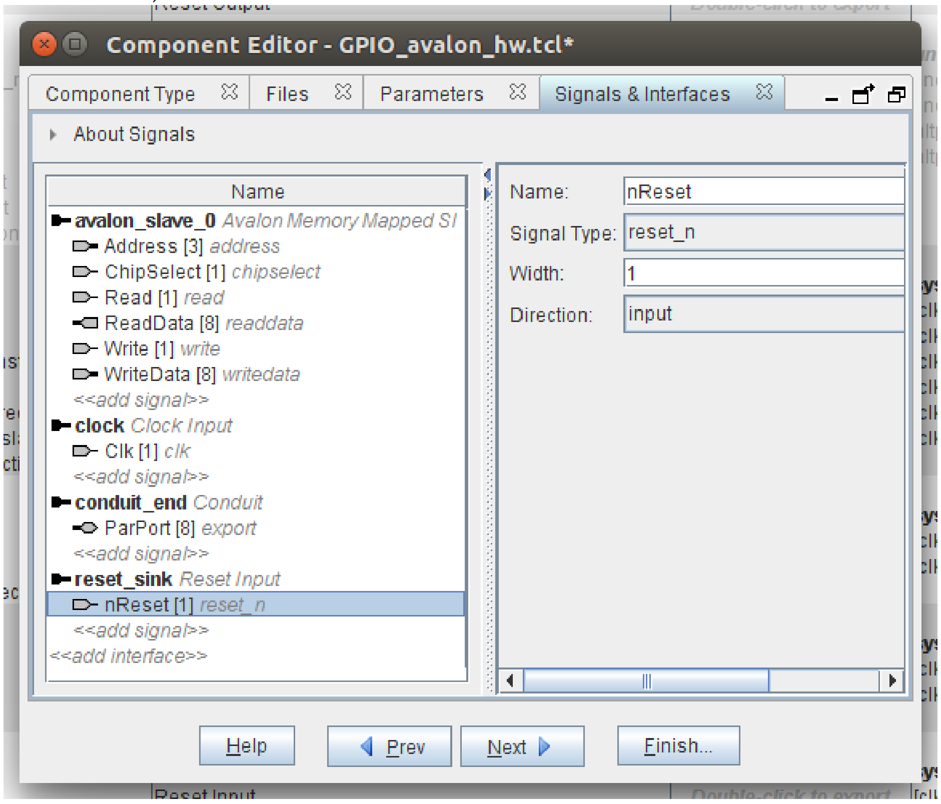
### VHDL description

Create a VHDL file for a 32-bit parallel port (GPIO.vhd), and describe in there the entity and architecture.
We will use it to drive the 32 LEDs present in the board. Refer to the course slides for this.

### Attach your VHDL bloc to your SoPC

- In Qsys, Go to "Files" => "new component"
- In the "Component Type" tab introduce a significative name.
- In the "Files" tab include your VHDL file as synthesis file and click on "Analyse Synthesis Files". You should end up with the window "Analyzing Synthesis Files Completed" free of errors, if it is not your case read the messages and fix your VHDL.
  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 who 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).
- 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 your code functionality.

- At the end you should end up with a view similar to next figure (with 32 bit buses instead of 8) :



Component Editor - GPIO_avalon_hw.tcl*

Component Type | Files | Parameters | Signals & Interfaces

About Signals

```
Name
avalon_slave_0 Avalon Memory Mapped Sl
    Address [3] address
    ChipSelect [1] chipselect
    Read [1] read
    ReadData [8] readdata
    Write [1] write
    WriteData [8] writedata
    <<add signal>>
clock Clock Input
    Clk [1] clk
    <<add signal>>
conduit_end Conduit
    ParPort [8] export
    <<add signal>>
reset_sink Reset Input
    nReset [1] reset_n
    <<add signal>>
<<add interface>>
```

| | |
|---|---|
| Name: | nReset |
| Signal Type: | reset_n |
| Width: | 1 |
| Direction: | input |

- When selecting the Avalon interface, verify that the timing fits with the type of access defined in your VHDL (for instance, number of clock cycles per read). 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 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 your schematic, and rerun the synthesis and P&R.

## Write your C program

Back to Eclipse, Think about regenerating your BSP !

Use the functions IOWR_32DIRECT and IORD_32DIRECT defined in io.h in order to access the peripheral registers that you have defined.
During the initialization set your port as output and then reproduce the counter sequence.

Keep in mind that the "BASE" address of your peripheral is defined in the "system.h" file generated by your bsp. The "offset" is specific to the addressing that you defined in your VHDL code.

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

## Timer and interruptions - Week 3

In order to allow your LED based counter to count periodically you must include an interval counter in your SoPC.

### 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 your 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”, page 209, for a detailed description of the register model. (available in `www.altera.com/literature/ug/ug_embedded_ip.pdf`).

Tables 194, 196, and 197 give all the information you need to enable the timer. Keep in mind that timer registers are 16-bit in order to address them correctly.

### Using interruptions in C

For using Interruptions in the Nios II / Eclipse code, you must start by including the library

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

In your main, at the begining, initialise your interruption with :

```
alt_irq_register(TIMER_0_IRQ,(void*)2,(alt_isr_func)timer_interrupt);
```

This function will map the IRQ receiver in TIMER_0_IRQ to the interruption handler `timer_interrupt`.

Afterwards, lets describe the interruption handler, here is an incomplete example :

```
 void timer_interrupt(void *context, alt_u32 id){
   counter ++; // increase the counter;
                        // write counter value on the parallel port;
                        // acknowledge IRQ on the timer;
}
```

You have to start your timer and enable IRQs from your main function, and you must also acknowledge IRQs from your timer interruption handler.

You can use, for instance, the function `IOWR_16DIRECT(TIMER_0_BASE,4,0x4);` for initialising the counter without interruptions. Have a look to the control register of the timer on the datasheet in order to enable interruptions and other functionalities.

## LCD interface design - Week 4

### Architecture definition

- Read the datasheet of the LCD interface. Pay special attention to the time diagram and the control registers.
- Identify the interface signals between your avalon interface and the LCD controller. You can have a look to the pin assignment tcl file for this. Respect signals' names !
- **ON A PAPER ! : propose a register model (bus Avalon) and a schematic.**
  Clearly identify the inputs and outputs on your interface. Include an output *waitrequest* on your avalon interface. Your Avalon bus doesn't has the right to perform a new access until the writing on the LCD has been completed
- Write the VHDL code for Avalon interface and the logic required for generating the correct timing of the LCD interface signals.
- Simulate with Modelsim, Verify your time-diagram.
- Add to Qsys and generate your SoPC.

### Using your interface

- Initialize your LCD :
  Refer to the initialisation exemple on the moodle webpage.
- On eclipse : send some pixels. For doing so use the command allowing to write to the graphic memory followed by two intricated loops for swapping every possible value for x and y. Generate a nice pattern.
- A real image cannot be generated as a pattern. It must be stored somewhere in memory as an array and your processor must first read it from memory to write it on the LCD. Send some pixels from the data memory. For doing so, initialize a bidimensional array with some values, and then send the vector values to the LCD interface.

If after a correct simulation your LCD doesn't work, debug the interface with Eclipse and the SignalTap Analyser : verify that the behavior on the real circuit matches the simulated behavior.