GPIO

Application Note for Altera DE2 Development and Education Board

Robert Miller
ECE 492 • University of Alberta • 21 March, 2013

GPIO

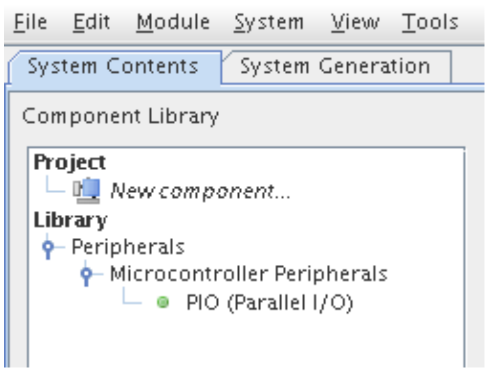
Application Note for Altera DE2 Development and Education Board

GENERAL PURPOSE INPUT/OUTPUT

The Altera DE2 board has two General Purpose Input/Output (GPIO) ports which can be used as input or output. The GPIO pins can be used to read from different types of sensors, and for writing output for digital control of various systems. There are also various GPIO peripherals that can be interfaced with the Altera DE2. This application note will guide you through the basic setup of GPIO pins using Altera SOPC Builder 10.1sp1, Quartus II Version 10.1, and the Nios II Integrated Development Environment.

1. Adding a PIO instance in SOPC Builder

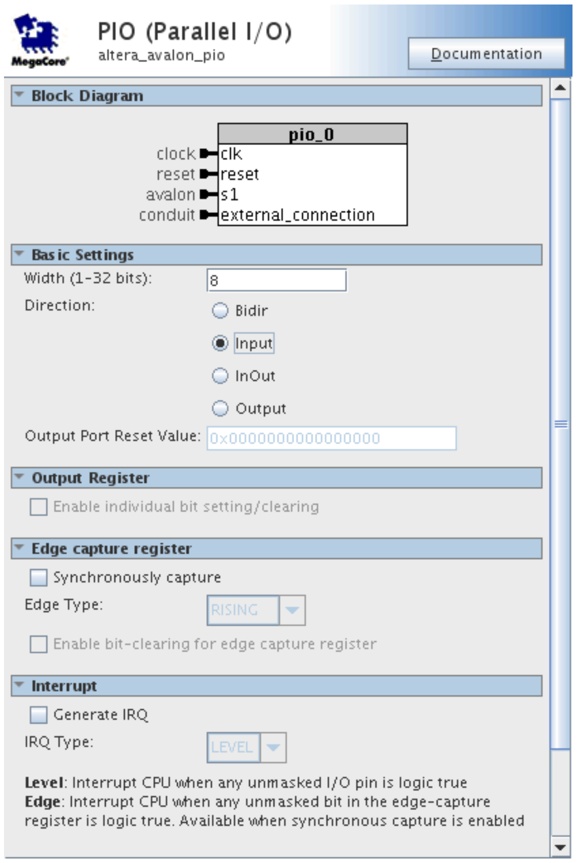
Launch Altera SOPC Builder and add a PIO instance by navigating to Library > Peripherals > Microcontroller Peripherals > PIO (Parallel I/O). Alternately you can type PIO in the search box.



After pressing ‘Add...’ or double clicking the PIO, you will be presented with a screen to customize the instance.

In the ‘Basic Settings’ you can choose the bit width of your PIO (1-32 bits - how many pins do you want to use) and the direction for these pins (in, out, bidirectional).

If you are planning to use your GPIO as an output signal (i.e drive a motor, control a CMOS etc.) set the direction to output and then click on ‘Finish’. However, if you are using the GPIO as input, you have a few more options to choose from such as the edge type you want to capture or if you want to use this input to trigger an interrupt request.



Basic Settings

Width (1-32 bits): 8

Direction: ( ) Bidir [✓ Input] ( ) InOut ( ) Output

Output Port Reset Value: 0x0000000000000000

Output Register

☐ Enable individual bit setting/clearing

Edge capture register

☐ Synchronously capture

Edge Type: RISING

☐ Enable bit-clearing for edge capture register

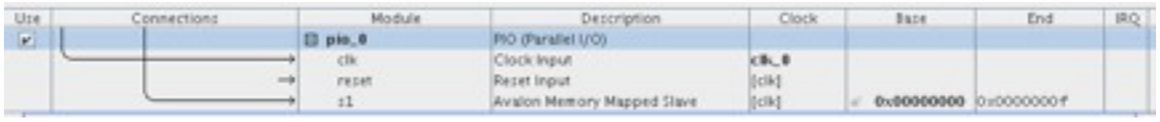
Interrupt

☐ Generate IRQ

IRQ Type: LEVEL

Level: Interrupt CPU when any unmasked I/O pin is logic true
Edge: Interrupt CPU when any unmasked bit in the edge-capture register is logic true. Available when synchronous capture is enabled

Once you are done customizing your instance of a PIO, you can click on ‘Finish’ and SOPC builder will add the module to your design.



Use	Connections	Module	Description	Clock	Base	End	IRQ
✓		pio_0	PIO (Parallel I/O)				
		clk	Clock Input	clk_0			
		reset	Reset Input	[clk]			
		s1	Avalon Memory Mapped Slave	[clk]	0x00000000	0x0000000f	

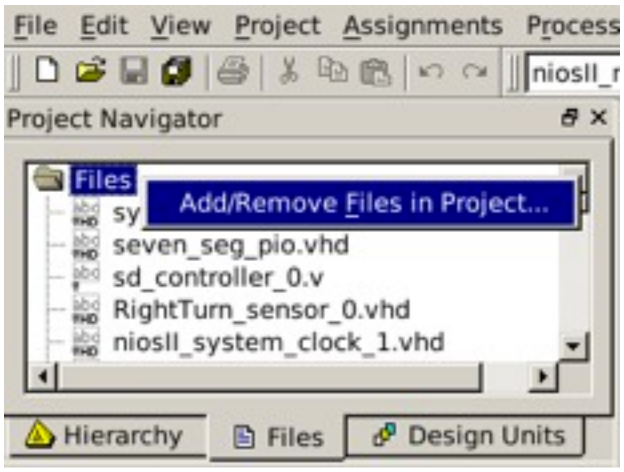
You may now right click on the default module name ‘pio_0’ and rename it as you wish. For the remainder of this document, we will assume that our example is a 12 bit output PIO called ‘loadController’. You can further customize the base address (or let SOPC builder auto assign) and the IRQ # if you setup your instance for interrupts in the previous step.

When you have completed adding all of your desired modules, you can save and generate your design.

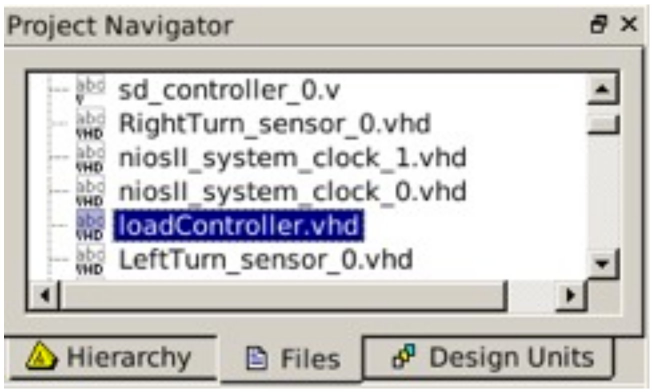
2. Setting up the GPIO instance in Quartus II

Now that SOPC Builder has generated all of the necessary files for you, all you need to do is finish wiring them up in your top-level file in Quartus II.

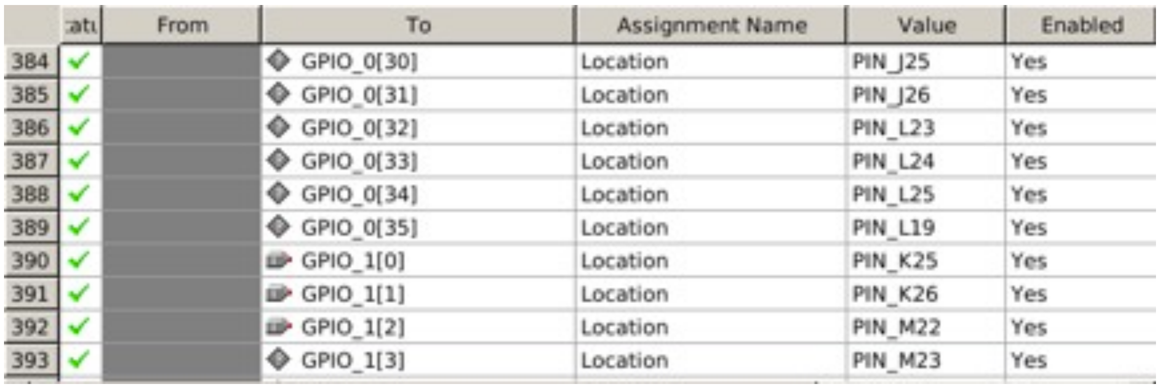
First, you must add all of the newly generated VHD files to your project. In the 'Project Navigator' pane, select the 'Files' Tab. Right click on the 'Files' folder and then select 'Add/Remove Files in Project...'.



Then next window will allow you to browse for your instance of the module, it will be named as whatever you renamed it to in SOPC builder. In this example, it is loadController.vhd. Select your file and then click on 'Add' > 'Apply' > 'OK'. If done successfully, you should now see your module instance in the Project Navigator pane.

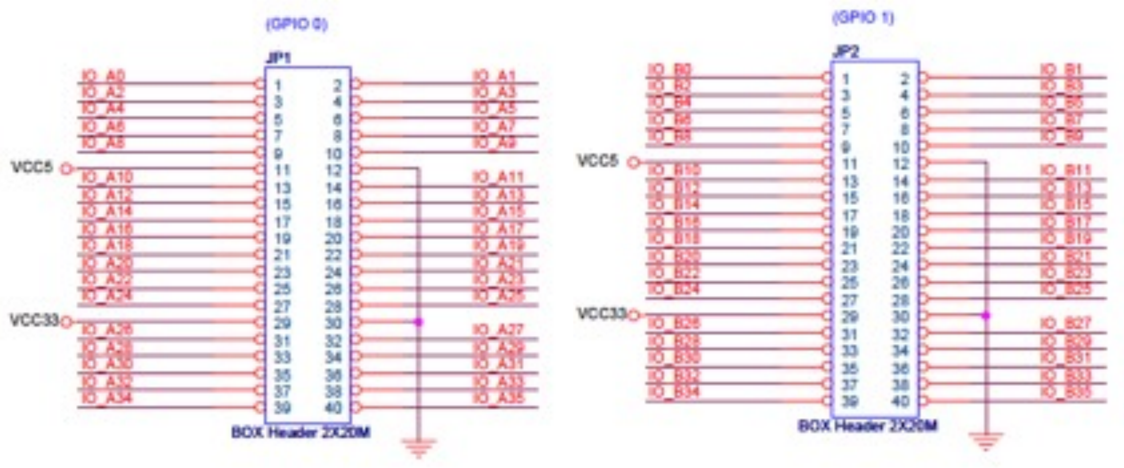


The GPIO pins need to be available to our design via pin assignments. If you have already imported all of the pin assignments in Quartus II then it will include the GPIO assignments. You can double check by clicking on ‘Assignments’ > ‘Assignment Editor’ (Ctrl+Shift+A). In the Assignment Editor you should be able to find your GPIO_0 and GPIO_1 with 36 pins each.



	:atı	From	To	Assignment Name	Value	Enabled
384	✓		GPIO_0[30]	Location	PIN_J25	Yes
385	✓		GPIO_0[31]	Location	PIN_J26	Yes
386	✓		GPIO_0[32]	Location	PIN_L23	Yes
387	✓		GPIO_0[33]	Location	PIN_L24	Yes
388	✓		GPIO_0[34]	Location	PIN_L25	Yes
389	✓		GPIO_0[35]	Location	PIN_L19	Yes
390	✓		GPIO_1[0]	Location	PIN_K25	Yes
391	✓		GPIO_1[1]	Location	PIN_K26	Yes
392	✓		GPIO_1[2]	Location	PIN_M22	Yes
393	✓		GPIO_1[3]	Location	PIN_M23	Yes

Note that both the GPIO_0 and GPIO_1 physically have 40 pins each, but we are only able to customize 36 of these pins at runtime. This is because 4 pins on each of the GPIO headers are reserved and cannot be changed. Pins 12 & 30 are ground, pin 11 is always at VCC 5V and similarly, pin 29 is at VCC 3.3V.



Consult the Altera DE2 schematics for further information.

Making the connections in Quartus II should be fairly trivial. First you declare the GPIO that you wish to use. For this example, we will setup GPIO_0 for our 12 bit loadController. In your top-level's entity you would add the following:

```
entity niosII_microc_fp is
        port
        (...
         -- LOAD CONTROLLER Outputs
        GPIO_0          :       out             std_logic_vector (35 downto 0);
        ...);
```

Next you will need to add the corresponding signal to your top-level's architecture. If you open up the *system.vhd file in your project, you will find the signal name for your module instance. In this example, we find the signal we want in the niosII_system.vhd files test bench statement (close to the bottom of the file). Look in your *system.vhd for 'architecture europa of test_bench' and inside there you will find the 'component niosII_system' declaration'. Scroll down until you find the name of your module instance. For example:

```
-- the_loadController
signal out_port_from_the_loadController : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
```

This is the signal you want to copy into your top-level's architecture > component declaration as shown below.

```
architecture structure of niosII_microc_fp is
component niosII_system
        port
        (...
          -- Load Controller
          out_port_from_the_loadController : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
         ...);
end component;
...
end structure;
```

The last piece is to connect the signal to the GPIO pins. This is also done inside the top-level's architecture. Scroll down from where you just added the signal and you will find the begin statement. Note that we are using explicit declaration of each bit which is not only good for troubleshooting but also gives you flexibility to use any combination of the pins as you see fit.

```
begin
...
       -- FOR GPIO Output for load controller
       out_port_from_the_loadController(0) => GPIO_0(0),
       out_port_from_the_loadController(1) => GPIO_0(1),
       out_port_from_the_loadController(2) => GPIO_0(2),
       ...and so on...
```

Earlier we declared the GPIO_0 as a 36 bit vector, by doing so we can now assign to any of the available 36 pins on the GPIO. If you are using the default pin assignments then note that Pin 1 will correspond to GPIO_0(0) and will continue this convention (offset of 1) up until pin 10. Since pin 11 and 12 are reserved, our offset changes and GPIO_0(10) will correspond to pin 13. This convention continues up until the next set of reserved pins, 29 and 30, at which point GPIO_0(26) will correspond to pin 31 and so on up to pin 40.

You are now finished setting up the GPIO in Quartus II. You can compile your design and move on to the Nios II IDE if required.

3. Accessing the GPIO pins from NIOS IDE

Once you have a project setup in the Nios IDE, your system.h file should have your module instance definitions that match what you did in SOPC Builder. For example, the base address should match #define LOADCONTROLLER_BASE 0x01909490. You will use this defined variable name to access the GPIO pins.

If everything matches (which it should as this is read straight from the *system.ptf file) then you are ready to write some code to control your GPIO.

We will utilize the two macros IORD and IOWR from “io.h” to read and write to our base address. The following is the syntax to use for IORD and IOWR:

IORD(base, offset)
IOWR(base, offset, data)
	base is the defined variable name of your module instance. For our example ‘LOADCONTROLLER_BASE’
	
	Offset is the word offset of the register you are accessing in the peripheral. The word size is assumed to be 32-bit so offsets 0, 1, 2, 3,... map to byte offsets 0, 4, 8, 12,....

data size oriented version of these macros:
IORD_8DIRECT(base, offset)
IORD_16DIRECT(base, offset)
IORD_32DIRECT(base, offset)
IOWR_8DIRECT(base, offset, data)
IOWR_16DIRECT(base, offset, data)
IOWR_32DIRECT(base, offset, data)
	Note that the difference here is that 'offset' is in bytes. The _8 _16 and _32 dictates the width of the access giving you more control over the access size (1, 2, or 4 bytes at a time). This is important when you access a slave port that contains byte enables and has multiple values stored in a single wide register.

For further information you may wish to consult the Altera forums as they are a wealth of information and you can find specific examples.

"These macros perform data cache bypassing. They are typically used for peripheral accesses so that the data doesn't become cached." ..."The IORD and IOWR macros treat the offset as a four byte word offset. Here are some examples:

IOWR(0, 4, 1234). -> writes 1234 to base 0 + word offset 4 (byte address 0 + 4x4= 16)
IORD(12, 2) -> reads from base 12 + word offset 2 (byte address 12+2x4 = 20)

In general the byte offset is 'base + offset x 4'. The access size is always 4 bytes which is why I don't recommend using IORD and IOWR and use the 8/16/32DIRECT ones instead which always use byte offsets." -BadOmen - SuperModerator Altera Forum

In our example, we are using our loadController to send a 0V or 3.3V to the gate of a CMOS for digital control of a circuit. Therefore, to turn pins 1-3 'on=3.3V' and the rest 'off = 0V' the code would look as follows:

```
IOWR_16DIRECT(LOADCONTROLLER_BASE, 0, 6);
```

This code will write 16 bits to the base address. Zero is used as the offset in our case. 6 translates to 0000000000000111 in binary with each bit mapping to the corresponding GPIO pins as defined earlier in our top-level. For example GPIO_0(0), GPIO_0(1), GPIO_0(2) would now be 'high/on' outputing 3.3V and the rest would be 'low/off' at 0V.

References

1. http://en.wikipedia.org/wiki/General_Purpose_Input/Output

2. http://www.alteraforum.com/forum/

3. http://users.ece.gatech.edu/~hamblen/DE2/DE2_Schematic.pdf

Image Credits

1. DE2 board image on title page used without permission from http://www.altera.com/education/univ/images/boards/de2.jpg