Altium Designer™

FPGA Design Training Module

Module 5

FPGA Design Training Module

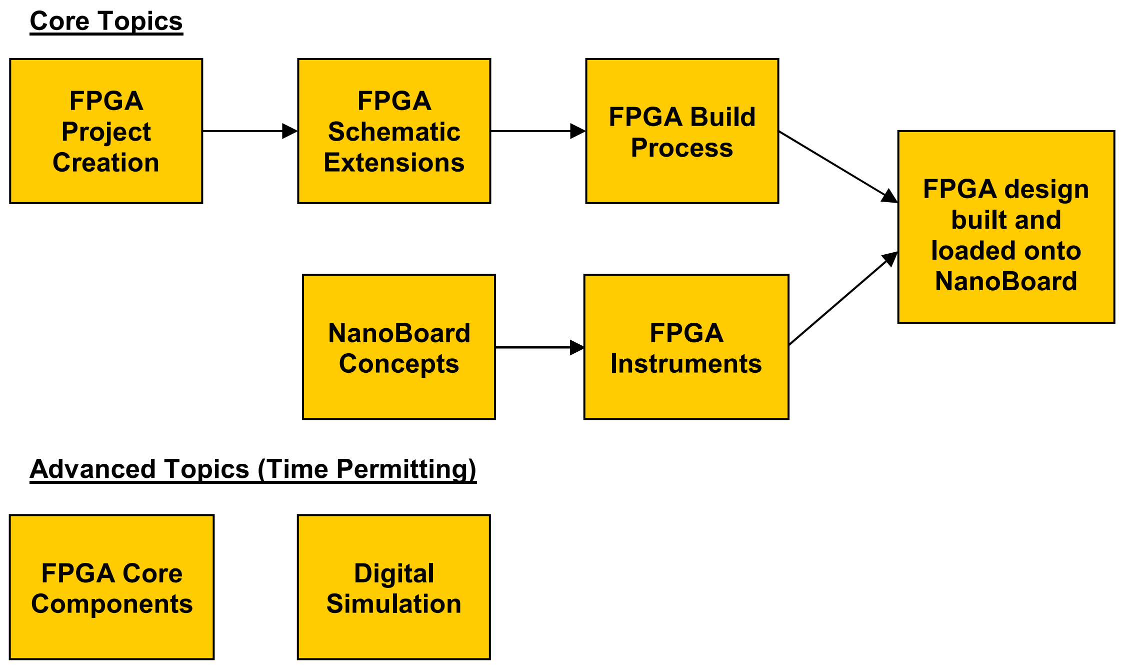
1 FPGA Design

The primary objective of this day of training is to make participants proficient in the process of developing, downloading and running an FPGA design on the NanoBoard. We will go through the FPGA design framework and demonstrate just how simple FPGA design is with Altium Designer.

1.1 Learning Objectives

- To be competent in developing FPGA designs using standard FPGA-based libraries and the schematic capture environment
- To understand and be able to make use of the FPGA build process
- To be familiar with the peripheral capabilities of the NanoBoard and know how to incorporate their use in custom FPGA designs.
- To appreciate the different communication mechanisms used by the software to control and probe a running FPGA design.
- To be competent with the use of virtual instruments in an FPGA design.

1.2 Topic Outline



.

Figure 1. Topic Outline for the FPGA Design training day.

2 Introduction to FPGA Design

2.1 FPGAwhats???

FPGA: Field Programmable Gate Array. Conceptually it can be considered as an array of Configurable Logic Blocks (CLBs) that can be connected together through a vast interconnection matrix to form complex digital circuits.

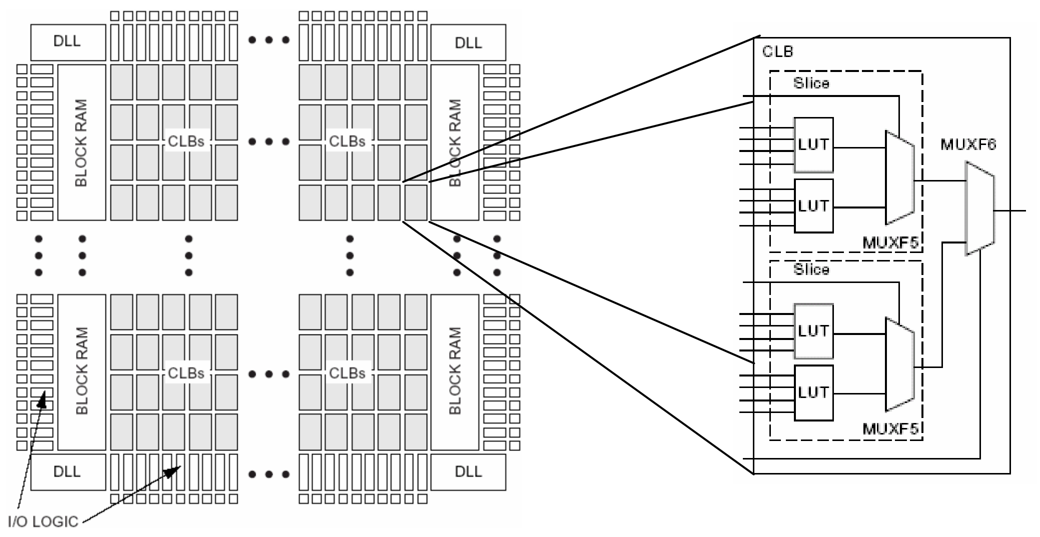


Figure 2. Exploded view of a typical FPGA

FPGAs have traditionally found use in high-speed custom digital applications where designs tend to be more constrained by performance rather than cost. The explosion of integration and reduction in price has led to the more recent widespread use of FPGAs in common embedded applications. FPGAs, along with their non-volatile cousins CPLDs (Complex Programmable Logic Devices), are emerging as the next digital revolution that will bring about change in much the same way that microprocessors did.

With current high-end devices exceeding 1000 pins and topping 1 billion transistors, the complexity of these devices is such that it would be impossible to program them without the assistance of high-level design tools. Altera and Xilinx both offer high-end EDA tool suites designed specifically to support their own devices however they also offer free versions aimed at supporting the bulk of FPGA development. Both Altera and Xilinx understand the importance of tool availability to increased silicon sales and they both seem committed to supporting a free version of their tools for some time to come.

Through the use of EDA tools, developers can design their custom digital circuits using either schematic based techniques, VHDL or a mixture of both. Prior to the Altium Designer system, vendor independent FPGA development tools were extremely expensive. Furthermore they were only useful for circuits that resided within the FPGA device. Once the design was extended to include a PCB and ancillary circuits, a separate EDA tool was needed. Altium Designer has changed all of this by being the first EDA tool capable of offering complete schematic to PCB tool integration along with multi-vendor FPGA support.

Altium made the logical extrapolation of recent trends in the FPGA world and recognized that FPGAs are no longer just for high-end designs. By making available a number of processor cores that can be downloaded onto an FPGA device and bundling them with a complete suite of embedded software development tools, Altium Designer represents a unified PCB and embedded systems development tool. FPGAs are here to stay and Altium Designer ensures that you can make the leap to the new world of digital integration with minimal pain.

3 Creating an FPGA project

3.1 Overview

All components that will be combined together into a single FPGA design must be encapsulated within an FPGA Project.

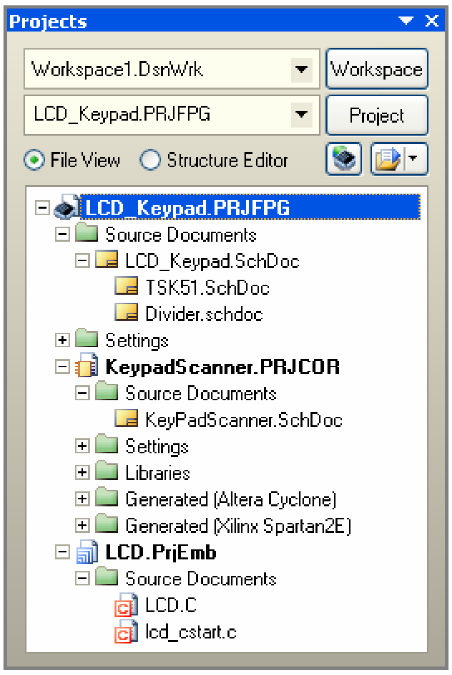
The term “Project” refers to a group of documents that combine together to form a single target. Care must be exercised when creating a project to ensure that the correct project type is selected for the desired target.

3.2 A quick word about projects and design workspaces

To the uninitiated, Altium Designer projects can appear a little confusing; especially when projects contain other projects. The important thing to remember is that each project can only have one output. If you have a design that requires several PCBs then you will need a separate PCB project for each PCB. If you have a design that uses several FPGAs then you will also need a separate FPGA project for each FPGA used on the final design.

Projects that are related together in some way can be grouped together using a type of ‘super project’ called a ***Design Workspace***. Design Workspaces are simply a convenient way of packaging one or more projects together so that all projects from a single design can be opened together.

Altium Designer supports a fully hierarchical design approach. As such it is possible for some projects to contain other projects within them. Figure 3 shows a structural view of the `LCD_Keypad` design that is distributed as an example in the Altium Designer installation. From this view we can observe the hierarchy of the different projects involved. The top-level project is an FPGA project called `LCD_Keypad` and has the filename extension PRJFPG. Within this FPGA design is an instance of the TSK51 embedded softcore. The program or software that this embedded softcore executes is contained within another project called `LCD.PrjEmb`. Furthermore the `LCD_Keypad` FPGA project also makes use of a core component called `KeyPadScanner` which has been defined as a core project (extension `.PRJCOR`).



```
LCD_Keypad.PRJFPG
  Source Documents
    LCD_Keypad.SchDoc
      TSK51.SchDoc
      Divider.schdoc
  Settings
  KeypadScanner.PRJCOR
    Source Documents
      KeyPadScanner.SchDoc
    Settings
    Libraries
    Generated (Altera Cyclone)
    Generated (Xilinx Spartan2E)
  LCD.PrjEmb
    Source Documents
      LCD.C
      lcd_cstart.c
```

Figure 3. An example of project hierarchy.

The hierarchy of projects is given below.

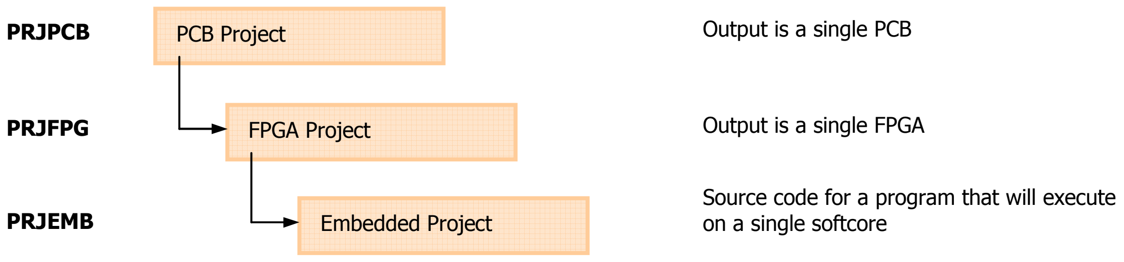


Figure 4. Possible Project Hierarchy for a design containing multiple projects

A PCB Project may contain one or more FPGA projects but never the other way around. If you think about it you will recognize that it is quite intuitive; a PCB contains FPGAs whereas an FPGA can’t contain a PCB. Similarly, an FPGA could contain one or more custom FPGA cores or microprocessor softcores. A unique Core Project will define each FPGA core component and a unique Embedded Project will define the software that executes on each of the softcores.

3.3 FPGA project

An FPGA project should be used when the target is a single FPGA. The output of an FPGA project will be a configuration bit file that can be used to program an FPGA.

The simplest way to create a project is from the **File** menu (**File » New » Project**).

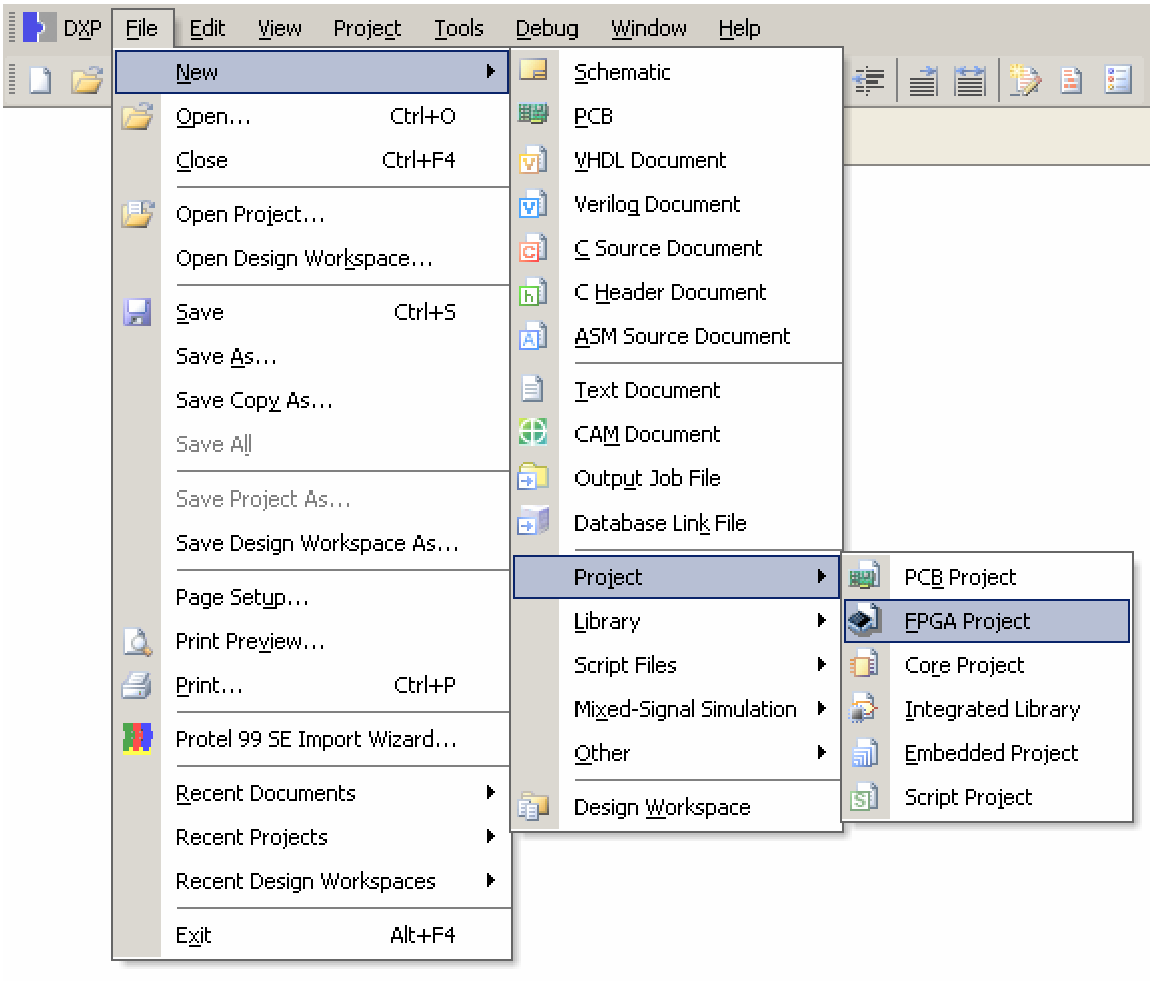


Figure 5. Creating a new FPGA project

4 FPGA schematic connectivity

4.1 Overview

Schematic documents used in FPGA designs are converted to VHDL in the process of being compiled into the design. This process is totally transparent to the user and does not require the user to know anything specific about VHDL. However the VHDL conversion process does place some requirements onto the schematic document that must be considered to ensure that the conversion process goes smoothly and that the resultant VHDL is valid.

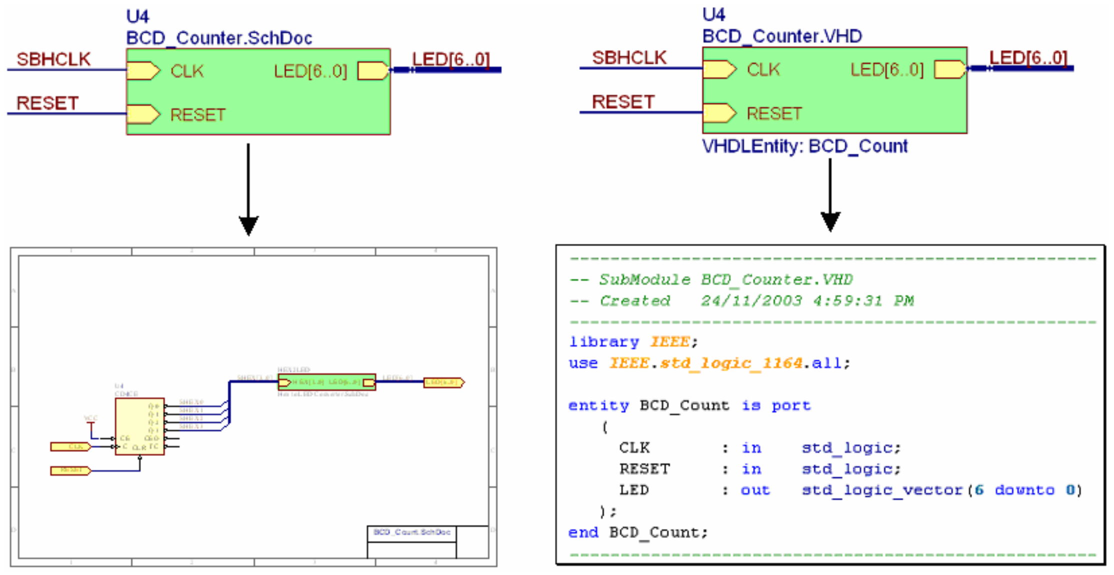
In this section we will discuss some of the extensions that have been added to the schematic environment for the purposes of servicing FPGA designs.

4.2 Wiring the design

Connectivity between the component pins is created by physical connectivity, or logical connectivity. Placing wires that connect component pins to each other creates *physical* connectivity. Placing matching net identifiers such as net labels, power ports, ports and sheet entries creates *logical* connectivity. When the design is compiled the connectivity is established, according to the net identifier scope defined for the project.

Note that while the environment supports compiling projects using either a flat or hierarchical connective structure, FPGA projects must be hierarchical.

4.3 Including VHDL in a schematic



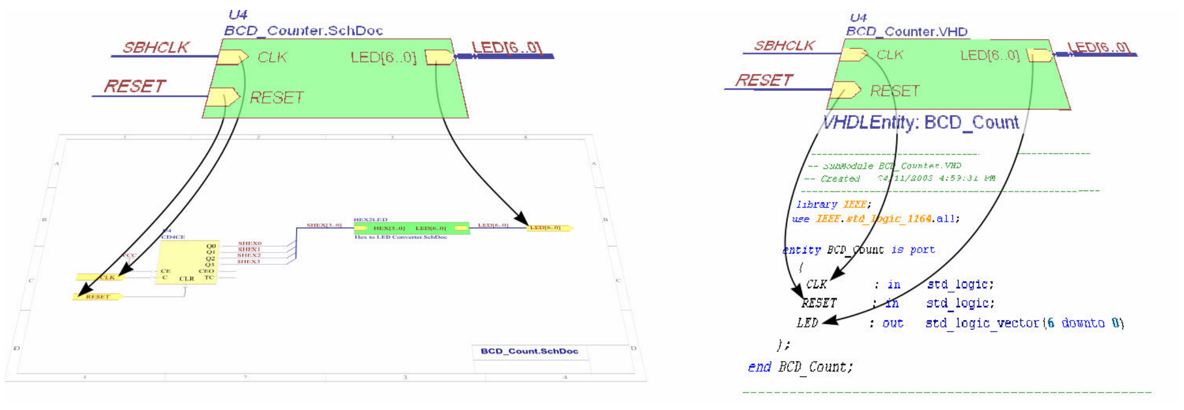
```
----------------------------------------------------------
-- SubModule BCD_Counter.VHD
-- Created   24/11/2003 4:59:31 PM
----------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity BCD_Count is port
   (
     CLK        : in    std_logic;
     RESET      : in    std_logic;
     LED        : out   std_logic_vector(6 downto 0)
   );
end BCD_Count;
----------------------------------------------------------
```

Figure 6. Linking schematic sheet symbols to lower level documents

VHDL sub-documents are referenced in the same way as schematic sub-sheets, by specifying the sub-document filename in the sheet symbol that represents it. The connectivity is from the sheet symbol to an entity declaration in the VHDL file. To reference an entity with a name that is different from the VHDL filename, include the VHDLEntity parameter in the sheet symbol whose value is the name of the Entity declared in the VHDL file (as shown above).

4.4 Establishing connectivity between documents

Hierarchical net and bus connectivity between documents obeys the standard hierarchical project connection behavior, where ports on the sub-document connect to sheet entries of the same name in the sheet symbol that represents that document, as shown below.



```
-- SubModule BCD_Counter.VHD
-- Created   04/11/2005 4:59:31 PM
---------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity BCD_Count is port
  (
   CLK       : in    std_logic;
   RESET     : in    std_logic;
   LED       : out   std_logic_vector(6 downto 0)
  );
end BCD_Count;
```

Figure 7. Connectivity between sheet symbols and lower level documents

4.5 Using buses and bus joiners

Typically there are a large number of related nets in a digital design. Buses can play an important role in managing these nets, and help present the design in a more readable form.

Buses can be re-ordered, renamed, split, and merged. To manage the mapping of nets in buses, there is a special class of component, known as a bus joiner. Bus joiners can be placed from the `FPGA Generic.IntLib` library (bus joiner names all start with the letter J). Figure 8 shows examples of using bus joiners. There are also many examples of using bus joiners in the example designs in the software.

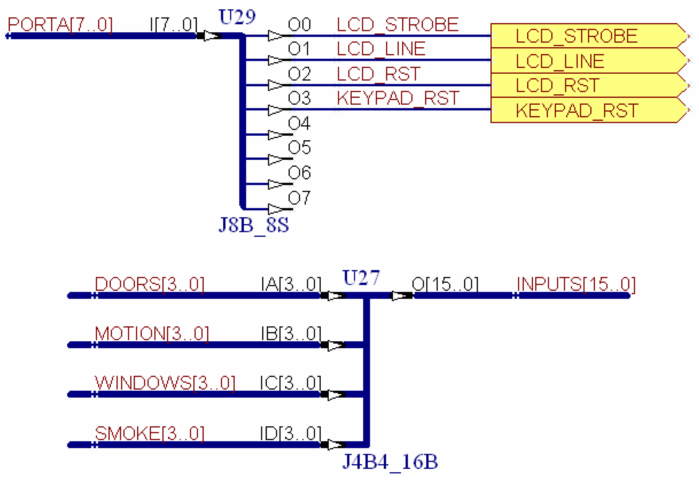


Figure 8. Examples of using bus joiners

Note that apart from the JB-type joiner, all bus joiner pins have an IO direction – use the correct joiner to maintain the IO flow. Pin IO can be displayed on sheet, enable the **Pin Direction** option in the schematic **Preferences** dialog.

The use of bus joiners in FPGA designs is a significant departure from how bus connectivity is established on other schematic documents however the benefits of bus joiners soon become clear. Nets extracted from a bus joiner need not be related in any way – ie. have the same name and differing only by number (Data[0], Data[1], Data[2], … etc). The bus joiner example above shows how a single bus can be used to route a number of LCD and Keypad signals together. Previously this was not possible in a bus.

4.5.1 Bus joiner naming convention

Bus joiners follow a standardized naming convention so that they can be easily found within the `FPGA Generic.IntLib library.`

`J<width><B/S>[Multiples]_<width><[B/S]>[Multiples]`

For example:

J8S_8B: describes a bus joiner that routes 8 single wires to a single, 8-bit bus.

J8B_8S: describes a bus joiner that routes a single, 8-bit bus into 8 single wires.

J8B_4B2: describes a bus joiner that routes a single 8-bit bus into two 4-bit busses,

J4B4_16B: describes a bus joiner that routes four, 4-bit busses into a single 16-bit bus.

4.5.2 Bus joiner splitting / merging behaviour

The basic rule is that bus joiners separate/merge the bits (or bus slice) from least significant bit (or slice) down to most significant bit (or slice).

For example, in Figure 12 U17 splits the incoming 8-bit bus on pin I[7..0] into two 4-bit bus slices, OA[3..0] and OB[3..0]. Obeying the *least to most* mapping at the slice level, the lower four bits of the input bus map to OA[3..0], and the upper four bits map to OB[3..0]. Following this through to the bit level, I0 will connect to OA0, and I7 will connect to OB3.

The joiner U27 merges the four incoming 4-bit slices into a 16-bit bus. With this joiner IA0 connects to O0, and ID3 connects to O15.

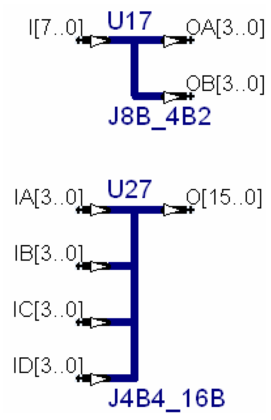


Figure 9. Bus joiners merge and split buses

4.5.3 Matching buses of different widths using the JB-type bus joiner

The JB-type bus joiner allows you to match nets in buses of different widths. It does this via 2 component parameters, IndexA and IndexB that map from one bus through to the other bus. These indices must be defined when you use a JB joiner.

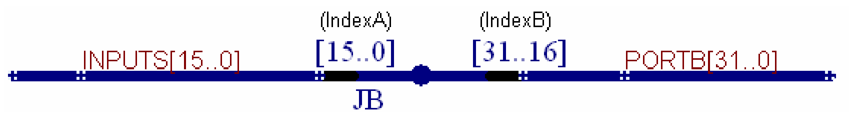


Figure 10. Join buses of different widths, and control the net-to-net mapping

Read the flow of nets through a JB-type bus joiner by matching from the nets in the attached bus, to the first index on the bus joiner, to the second index in the bus joiner, to the nets defined in the second bus net label.

Left Bus ↔ IndexA ↔ IndexB ↔ Right Bus

The rules for matching nets at each of the ↔ points are as follows:

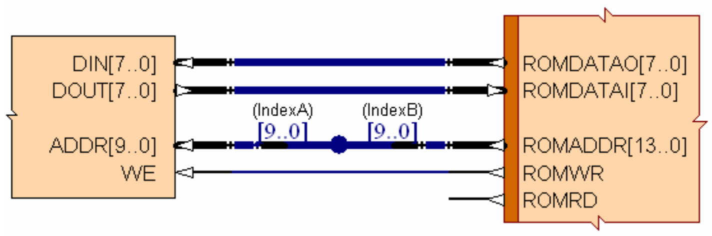


Figure 11. An example of using the JB bus joiner to achieve sub-set mapping

- If both bus ranges are descending, match by same bus index (one range must lie within the other for valid connections). In Figure 11 the matching is:

  ```
  ADDR9 ↔ IndexA9 ↔ IndexB9 ↔ ROMADDR9, thru to
  ADDR0 ↔ IndexA0 ↔ IndexB0 ↔ ROMADDR0
  ```

  (In this example ROMADDR10 thru ROMADDR13 will be unconnected)

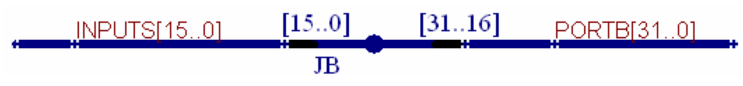


Figure 12. Using of a bus joiner for offset mapping

- In Figure 12 the matching is:

  ```
  INPUTS15 ↔ IndexA15 ↔ IndexB31 ↔ PORTB31, thru to
  ```

  INPUTS0 ⟷ IndexA0 ⟷ IndexB0 ⟷ PORTB16

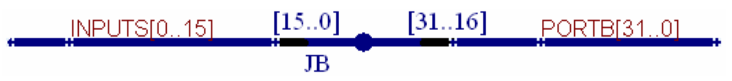


Figure 13. Using a bus joiner for range inversion

- If one bus range is descending and another is ascending, the indices are matched from left to right. In Figure 13 the matching is:

  `INPUTS0  ↔ IndexA15 ↔ IndexB31 ↔ PORTB31, thru to`

  INPUTS15 ↔ IndexA0 ↔ IndexB16 ↔ PORTB16

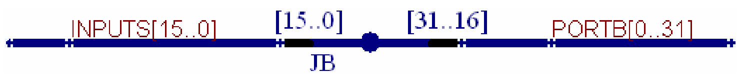


Figure 14. Another example of using a bus joiner for range inversion

- In Figure 14 the matching is:

  `INPUTS15 ↔ IndexA15 ↔ IndexB31 ↔ PORTB0, thru to`

  INPUTS0 ↔ IndexA0 ↔ IndexB16 ↔ PORTB15

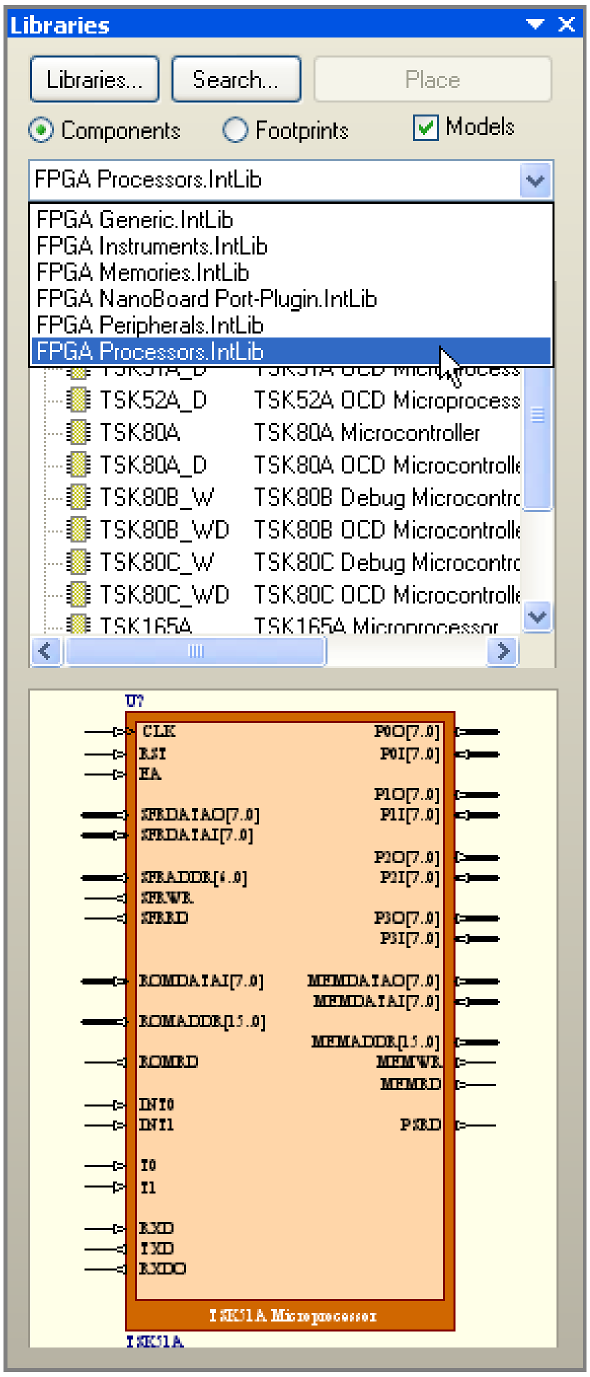
5 FPGA ready schematic components

5.1 Overview

A wide variety of FPGA-ready schematic components are included with the system, ranging from processors, to peripheral components, down to generic logic. Placing and wiring these schematic components, or writing VHDL, captures the hardware design. The FPGA-ready schematic components are like traditional PCB-ready components, except instead of the symbol being linked to a PCB footprint each is linked to a pre-synthesized EDIF model.

As well as components that you use to implement your design, the available FPGA libraries include components for the virtual instruments, and the components that are mounted on the NanoBoard and are accessible via the pins on the FPGA. The role of each type of component is described below.

Help for all FPGA-ready components can be accessed by pressing the F1 key whilst the component is selected in the library list.



Libraries

Libraries... Search... Place

[✓ Components] Footprints [✓ Models]

FPGA Processors.IntLib

FPGA Generic.IntLib
FPGA Instruments.IntLib
FPGA Memories.IntLib
FPGA NanoBoard Port-Plugin.IntLib
FPGA Peripherals.IntLib
FPGA Processors.IntLib

TSK51A_D	TSK51A OCD Microprocess
TSK52A_D	TSK52A OCD Microprocess
TSK80A	TSK80A Microcontroller
TSK80A_D	TSK80A OCD Microcontrolle
TSK80B_W	TSK80B Debug Microcontro
TSK80B_WD	TSK80B OCD Microcontrolle
TSK80C_W	TSK80C Debug Microcontro
TSK80C_WD	TSK80C OCD Microcontrolle
TSK165A	TSK165A Microprocessor

Figure 15. The libraries panel

5.2 Processor cores

Softcore processors can be placed from the `\Program Files\Altium Designer 6\Library\Fpga\FPGA Processors.IntLib` library. At the time of release of this manual, the following processors and related embedded software tools are supported:

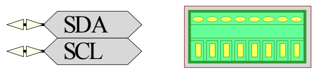
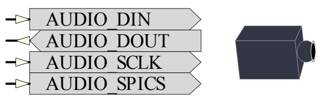
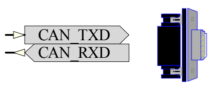
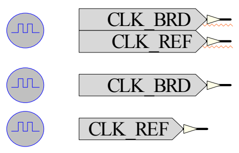
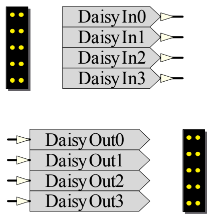
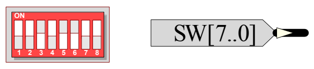
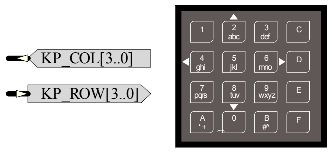
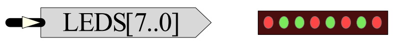
- TSK165 – Microchip 165x family instruction set compatible MCU
- TSK51/52 – 8051 instruction set compatible MCU
- TSK80 – Z80 instruction set compatible MCU
- PPC405A – Embedded Power PC Core available on some Virtex FPGAs.
- TSK3000 – 32-bit RISC processor.

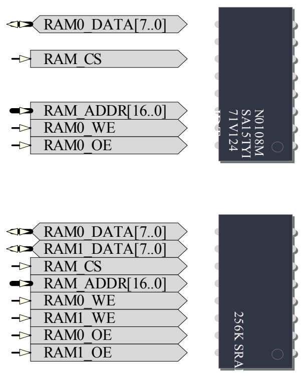
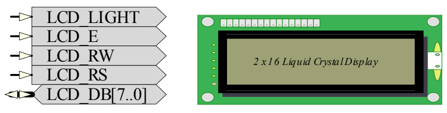
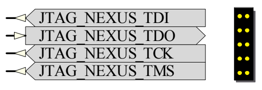
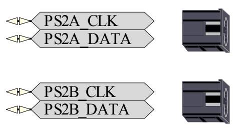
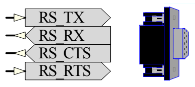
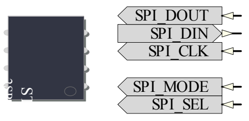
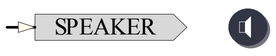
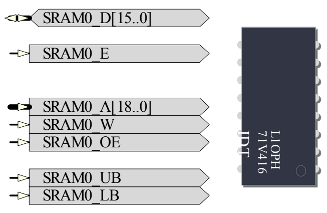
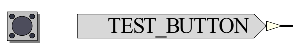
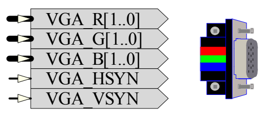
There is also full embedded tool support for:

- Xilinx MicroBlaze soft core, which requires the appropriate Xilinx device and license to use
- Xilinx Virtex-2 Pro based PowerPC 405
- AMCC PowerPC 405 discrete processor family
- ARM7, ARM9, ARM9E & ARM10E families, supported in the Sharp BlueStreak (ARM20T) discrete processor family

5.3 NanoBoard port plugins

Hardware resources on the NanoBoard can be accessed via the use of components from the `\Program Files\Altium Designer 6\Library\Fpga\FPGA NanoBoard Port-Plugin.IntLib` library. A summary of the components is given below:

NanoBoard Port-Plugin.IntLib		
Name in Library	**Description**	**Symbol**
ADCDAC_I2C	The MAX1037 ADC converter provides four multiplexed analogue inputs selectable by application software via the I2C connection.	
AUDIO_DAC	The NanoBoard provides and SPI-based 8-bit audio codec, together with relevant analogue pre and post conditioning circuitry (Maxim part number MAX1104).	
CANCNTR	The CAN Port is connected to a Microchip MCP2551 CAN transceiver IC.	
CLOCK_SUPPLY CLOCK_BOARD CLOCK_REFERENCE	An SPI-based system clock generator provides a fixed 20MHz clock (CLK_REF) and a user-programmable clock providing frequencies from 6 to 200MHz (CLK_BRD).	
DAISYIN_SLAVEIO DAISYOUT_MASTERIO	The NanoBoard can be connected in a daisychain configuration, allowing multiple NanoBoard applications to be controlled by the software. The Master and Slave I/O headers can be used to provide an application-defined communication resource between daughterboard applications on separate NanoBoards in a daisychain.	 DaisyIn0 DaisyIn1 DaisyIn2 DaisyIn3 DaisyOut0 DaisyOut1 DaisyOut2 DaisyOut3
DIPSWITCH	A standard DIP switch is wired as an active low device to the NanoBoard daughter board.	
KEYPAD	The keypad array consists of 16 miniature pushbuttons arranged in a 4 x 4 matrix. The keypad is organized so that a row-column scanning process can read the status of each key.	
LED	A set of 8 active high LEDs are connected to separate pins of the daughter board and can be driven by a user application.	

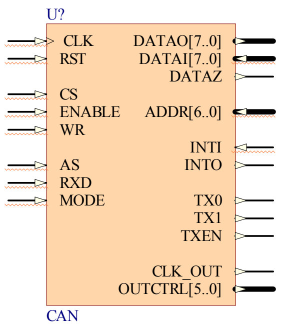
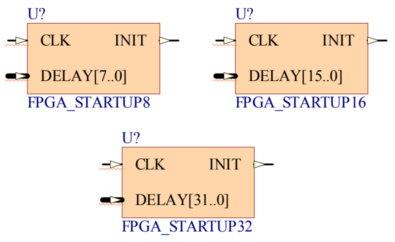
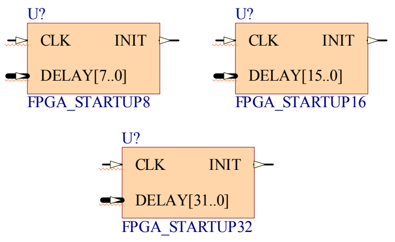
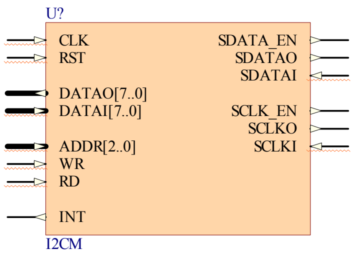
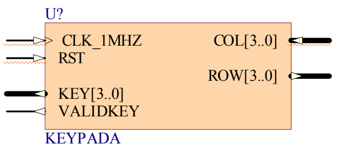
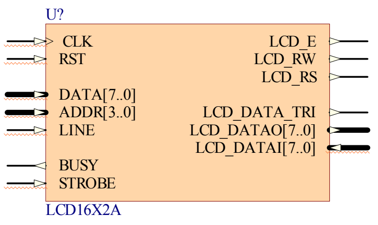
NanoBoard Port-Plugin.IntLib		
Name in Library	**Description**	**Symbol**
MEMORY0 MEMORY1 MEMORY256KX8	Two 128k x 8 static RAM devices are included on the NanoBoard and wired directly to I/O pins on the daughter board. The SRAM devices have a common Chip Select and Address signals but separate 8-bit data bus and Read/Write signals.	 RAM0_DATA[7..0] RAM_CS RAM_ADDR[16..0] RAM0_WE RAM0_OE RAM0_DATA[7..0] RAM1_DATA[7..0] RAM_CS RAM_ADDR[16..0] RAM0_WE RAM1_WE RAM0_OE RAM1_OE
LCD LCD_MEMORY0 LCD_MEMORY1 LCD_MEMORY256KX8	The NanoBoard contains a 16-character by 2-line industry standard LCD with LED backlight.	 LCD_LIGHT LCD_E LCD_RW LCD_RS LCD_DB[7..0] 2 x 16 Liquid Crystal Display
NEXUS_JTAG_ CONNECTOR	The Nexus, or soft devices chain, is implemented in the FPGA design by the inclusion of this connector.	 JTAG_NEXUS_TDI JTAG_NEXUS_TDO JTAG_NEXUS_TCK JTAG_NEXUS_TMS
PS2A PS2B	The NanoBoard features two PS2 ports – PS2A nominally may be used for a keyboard and PS2B may be used for a mouse.	
RS232CNTR	A standard DTE RS232 port is provided with TXD, RXD, RTS and CTS signals connected.	
SERIALFMEMORY	Two ST M25P40 low-cost 4-Mbit serial flash RAM devices are installed as a non-volatile memory source.	
SPEAKER	The NanoBoard's magnetic audio transducer can operate as a beeper when driven by a square-wave signal or can be pulse-width modulated to produce more complex sounds.	
SRAM_DAUGHTER0 SRAM_DAUGHTER1	Provision has been made for connection of to up to two 128k x 8 SRAM devices placed on a NanoBoard daughter board.	 SRAM0_D[15..0] SRAM0_E SRAM0_A[18..0] SRAM0_W SRAM0_OE SRAM0_UB SRAM0_LB
TEST_BUTTON	Although this button is labeled 'Test', it has no intrinsic function and can be used for any purpose by the user application.	
VGACNTR	The VGA port provides a VGA compatible RGB video monitor port. With two-bits per color channel, the VGA port is capable of representing 64 colors.	 VGA_R[1..0] VGA_G[1..0] VGA_B[1..0] VGA_HSYN VGA_VSYN

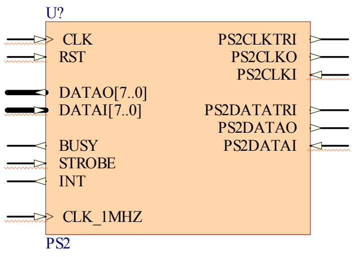
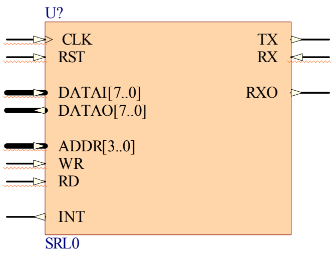
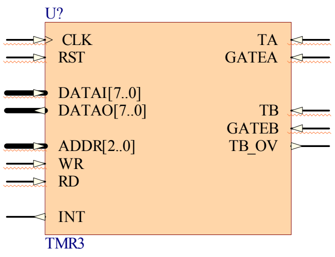
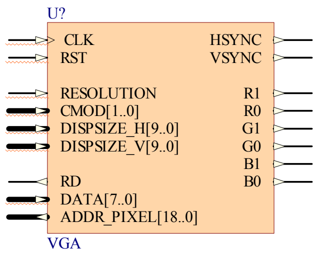
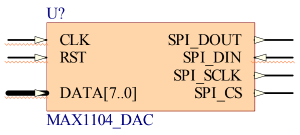
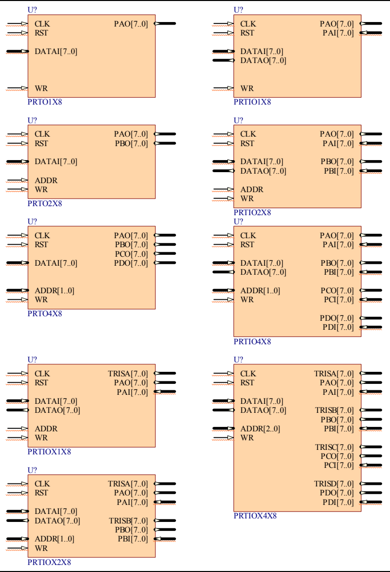
5.4 Peripheral Components

Many of the hardware resources present on the NanoBoard come with peripheral modules that can be included in the FPGA design to ease interfacing to the external port.

Peripherals can be placed from the `\Program Files\Altium Designer 6\Library\Fpga\FPGA Peripherals.IntLib` library.

FPGA Peripherals.IntLib

Name in Library	**Description**	**Symbol**
CAN	**CAN Controller** – parallel to serial interface, implementing a Controller Area Network serial communications bus on the serial side. The CAN serial bus provides high bit rate, high noise immunity and error detection. The Controller implements the BOSCH CAN 2.0B Data Link Layer Protocol. The CAN controller can be used in conjunction with the CAN interface hardware on the NanoBoard.	 U? CLK RST CS ENABLE WR AS RXD MODE DATAO[7..0] DATAI[7..0] DATAZ ADDR[6..0] INTI INTO TX0 TX1 TXEN CLK_OUT OUTCTRL[5..0] CAN
EMAC EMAC_MD EMAC_MD_W EMAC_W	**Ethernet Media Access Controller** – provides an 8-bit IEEE802.3 compliant interface between a processor and a standard Physical Layer device (PHY).	 U? CLK INIT DELAY[7..0] FPGA_STARTUP8 U? CLK INIT DELAY[15..0] FPGA_STARTUP16 U? CLK INIT DELAY[31..0] FPGA_STARTUP32
FPGA_STARTUP8 FPGA_STARTUP16 FPGA_STARTUP32	**FPGA Startup** – user-definable power-up delay, used to implement power-on reset. An internal counter starts on power up, counting the number of clock cycles specified by the Delay pin, the output pin being asserted when the count is reached.	 U? CLK INIT DELAY[7..0] FPGA_STARTUP8 U? CLK INIT DELAY[15..0] FPGA_STARTUP16 U? CLK INIT DELAY[31..0] FPGA_STARTUP32
I2CM	**I2C** – parallel to serial interface, implementing an Inter-Integrated Circuit (I2C) 2-wire serial bus on the serial side. Controllers only support a single master I2C serial bus system. The I2C controller can be used in conjunction with the I2C interface hardware on the NanoBoard.	 U? CLK RST DATAO[7..0] DATAI[7..0] ADDR[2..0] WR RD INT SDATA_EN SDATAO SDATAI SCLK_EN SCLKO SCLKI I2CM
KEYPADA	**Keypad Controller** – 4 x 4 keypad scanner with de-bounce. Can be used in a polled or interrupt driven system. Also available in either Wishbone or non-Wishbone variants. The Keypad controller can be used in conjunction with the keypad on the NanoBoard.	 U? CLK_1MHZ RST KEY[3..0] VALIDKEY COL[3..0] ROW[3..0] KEYPADA
LCD16X2A	**LCD Controller** – easy to use controller for a 2 line by 16-character LCD module. The LCD controller can be used in conjunction with the LCD display on the NanoBoard.	 U? CLK RST DATA[7..0] ADDR[3..0] LINE BUSY STROBE LCD_E LCD_RW LCD_RS LCD_DATA_TRI LCD_DATAO[7..0] LCD_DATAI[7..0] LCD16X2A

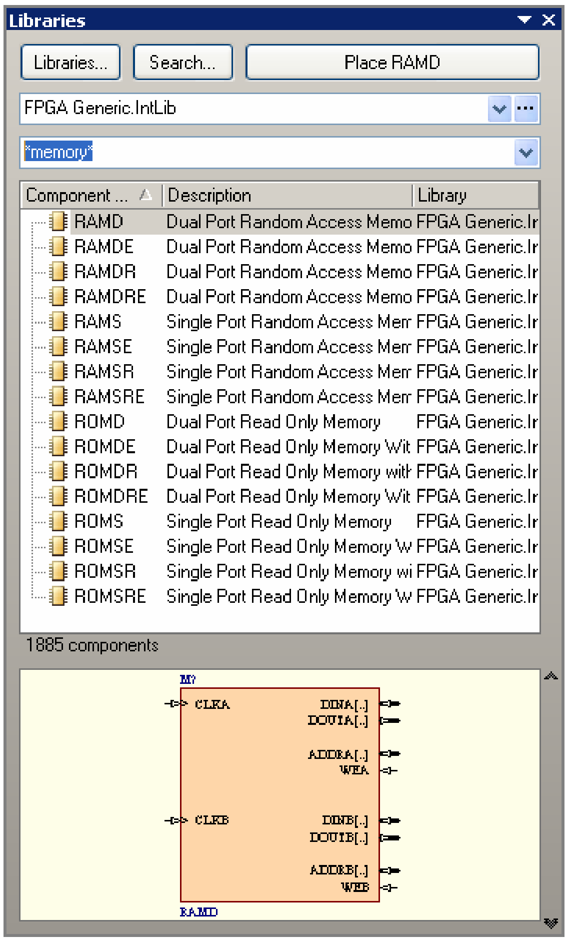
FPGA Peripherals.IntLib		
Name in Library	**Description**	**Symbol**
PS2	**PS2 Controller** – parallel to serial interface providing a bi-directional, synchronous serial interface between a host MCU and a PS/2 device (keyboard or mouse). The PS2 controller can be used in conjunction with either of the two sets of PS2 interface hardware on the NanoBoard.	 U? CLK RST DATAO[7..0] DATAI[7..0] BUSY STROBE INT CLK_1MHZ PS2CLKTRI PS2CLKO PS2CLKI PS2DATATRI PS2DATAO PS2DATAI PS2
SRL0	**SRL0** – simple parallel to serial interface, full duplex, single byte buffering. The SRL0 can be used in conjunction with the RS-232 interface hardware on the NanoBoard.	 U? CLK RST DATAI[7..0] DATAO[7..0] ADDR[3..0] WR RD INT TX RX RXO SRL0
TMR3	**TMR3** – dual timer unit, 16, 13 and 8-bit timer/counter modes.	 U? CLK RST DATAI[7..0] DATAO[7..0] ADDR[2..0] WR RD INT TA GATEA TB GATEB TB_OV TMR3
VGA	**VGA** – VGA controller that creates a simple method of implementing a VGA interface, presenting video memory as a flat address space. Supports VGA and SVGA resolutions, and B&W, 16 and 64 color. Outputs digital RGB and H+V sync. The VGA controller can be used in conjunction with the VGA output on the NanoBoard.	 U? CLK RST RESOLUTION CMOD[1..0] DISPSIZE_H[9..0] DISPSIZE_V[9..0] RD DATA[7..0] ADDR_PIXEL[18..0] HSYNC VSYNC R1 R0 G1 G0 B1 B0 VGA
MAX1104_DAC	**DAC** – This digital to analogue controller module provides a simple interface to the MAX1104 8-bit CODEC device on the NanoBoard.	 U? CLK RST DATA[7..0] SPI_DOUT SPI_DIN SPI_SCLK SPI_CS MAX1104_DAC
PRTIO1X8 PRTIO2X8 PRTIO4X8 PRTIOX1X8 PRTIOX2X8 PRTIOX4X8 PRTO1X8 PRTO2X8 PRTO4X8	**PRTx** – The PRTx Parallel Port Unit is simply a register interface for storing data to be transferred to/from another device in a design. For example when used with a microcontroller such as the TSK80x, which does not have any on-core port interfaces, the unit provides a valuable new extension to the processor's feature set. **PRTOx** – output only port devices **PRTIOx** – I/O port devices **PRTIOXx** – I/O port devices with additional tristate buffer enable output for each port.	 PRTO1X8 PRTIO1X8 PRTO2X8 PRTIO2X8 PRTO4X8 PRTIO4X8 PRTIOX1X8 PRTIOX4X8 PRTIOX2X8

5.5 Generic components

Generic components can be placed from the library `\Program Files\Altium Designer 6\Library\Fpga\FPGA Generic.IntLib`. This library is included to implement the interface logic in your design. It includes pin-wide and bus-wide versions for many components, simplifying the wiring complexity when working with buses. As well as a broad range of logic functions, the generic library also includes pullup and pulldown components as well as a range of bus joiners, used to manage the merging, splitting and renaming of buses.

For a definition of the naming convention used in the generic library and a complete listing of available devices, refer to the document: `CR0118 FPGA Generic Library Guide.pdf`.

Wild card characters can be used to filter when searching the component library.



Libraries

Libraries... Search... Place RAMD

FPGA Generic.IntLib

memory

Component ...	Description	Library
RAMD	Dual Port Random Access Memo	FPGA Generic.Ir
RAMDE	Dual Port Random Access Memo	FPGA Generic.Ir
RAMDR	Dual Port Random Access Memo	FPGA Generic.Ir
RAMDRE	Dual Port Random Access Memo	FPGA Generic.Ir
RAMS	Single Port Random Access Mem	FPGA Generic.Ir
RAMSE	Single Port Random Access Mem	FPGA Generic.Ir
RAMSR	Single Port Random Access Mem	FPGA Generic.Ir
RAMSRE	Single Port Random Access Mem	FPGA Generic.Ir
ROMD	Dual Port Read Only Memory	FPGA Generic.Ir
ROMDE	Dual Port Read Only Memory Wit	FPGA Generic.Ir
ROMDR	Dual Port Read Only Memory with	FPGA Generic.Ir
ROMDRE	Dual Port Read Only Memory Wit	FPGA Generic.Ir
ROMS	Single Port Read Only Memory	FPGA Generic.Ir
ROMSE	Single Port Read Only Memory W	FPGA Generic.Ir
ROMSR	Single Port Read Only Memory wi	FPGA Generic.Ir
ROMSRE	Single Port Read Only Memory W	FPGA Generic.Ir

1885 components

Figure 16. Using wildcards to quickly find a specific component in the Generic Library

5.6 Vendor macro and primitive libraries

If vendor independence is not required, there are also complete primitive and macro libraries for the currently supported vendors/device families. These libraries can be found in the respective Actel, Altera, Lattice and Xilinx sub-folders in `\Program Files\Altium Designer 6\Library\`. The macro and primitive library names end with the string `*FPGA.IntLib`. Note that some vendors require you to use primitive and macro libraries that match the target device. Designs that include vendor components *cannot* be re-targeted to another vendor's device.

5.7 Exercise 1 – Create a PWM.

In this exercise we will create our first FPGA design. In order to complete this task you will need to use the following components from their respective libraries:

1. Open a new FPGA Project. Save it as MyPWM.PrjFpg
2. Add a new schematic to your project and save it as MyPWM.SchDoc

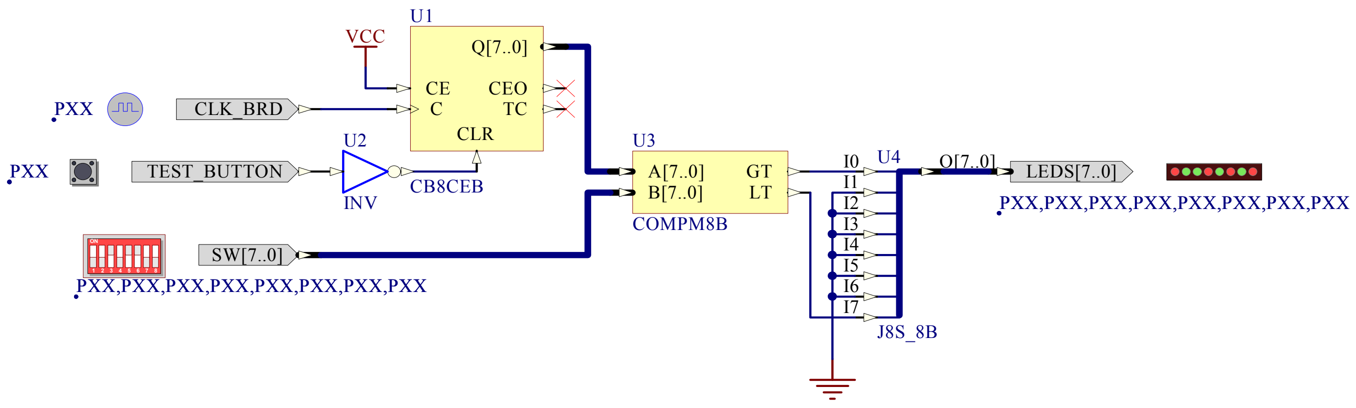


Figure 17. Place and wire the components to create the Pulse Width Modulator

3. Using components from the two libraries `FPGA Generic.IntLib` and FPGA NanoBoard Port-Plugin.IntLib, place and wire the schematic shown in Figure 17.

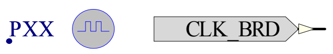
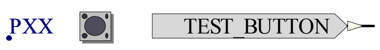
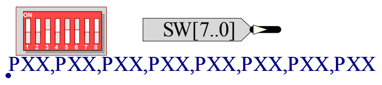
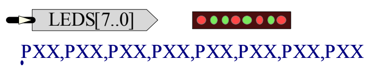
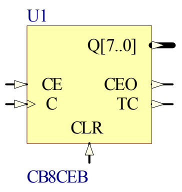
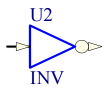
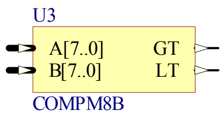
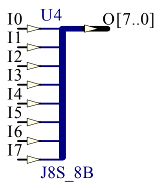
Component	Library	Name in Library
	FPGA NanoBoard Port-Plugin.IntLib	CLOCK_BOARD
	FPGA NanoBoard Port-Plugin.IntLib	TEST_BUTTON
	FPGA NanoBoard Port-Plugin.IntLib	DIPSWITCH
	FPGA NanoBoard Port-Plugin.IntLib	LED
	FPGA Generic.IntLib	CB8CEB
	FPGA Generic.IntLib	INV
	FPGA Generic.IntLib	COMPM8B
 I0 U4 Q[7..0] I1 I2 I3 I4 I5 I6 I7 J8S_8B	FPGA Generic.IntLib	J8S_8B

Figure 18. Save your work – we will continue with this schematic soon

6 Targeting the design

The schematic that we have just created contains all of the connectivity that must occur internally on our FPGA device but we still need some further information to map the ports on the FPGA schematic to physical pins on an actual FPGA device. This process is called targeting our design.

6.1 Constraint files

Rather than storing device and implementation specific data such as pin allocations and electrical properties in the source VHDL or Schematic documents, this information is stored in separate files – called *Constraint files*. This decoupling of the logical definition of an FPGA design from its physical implementation allows for quick and easy re-targeting of a single design to multiple devices and PCB layouts.

Below we see a conceptual representation of an FPGA design sitting inside an FPGA device. The red lines indicate the port-to-pin mappings that would be handled by the constraint file.

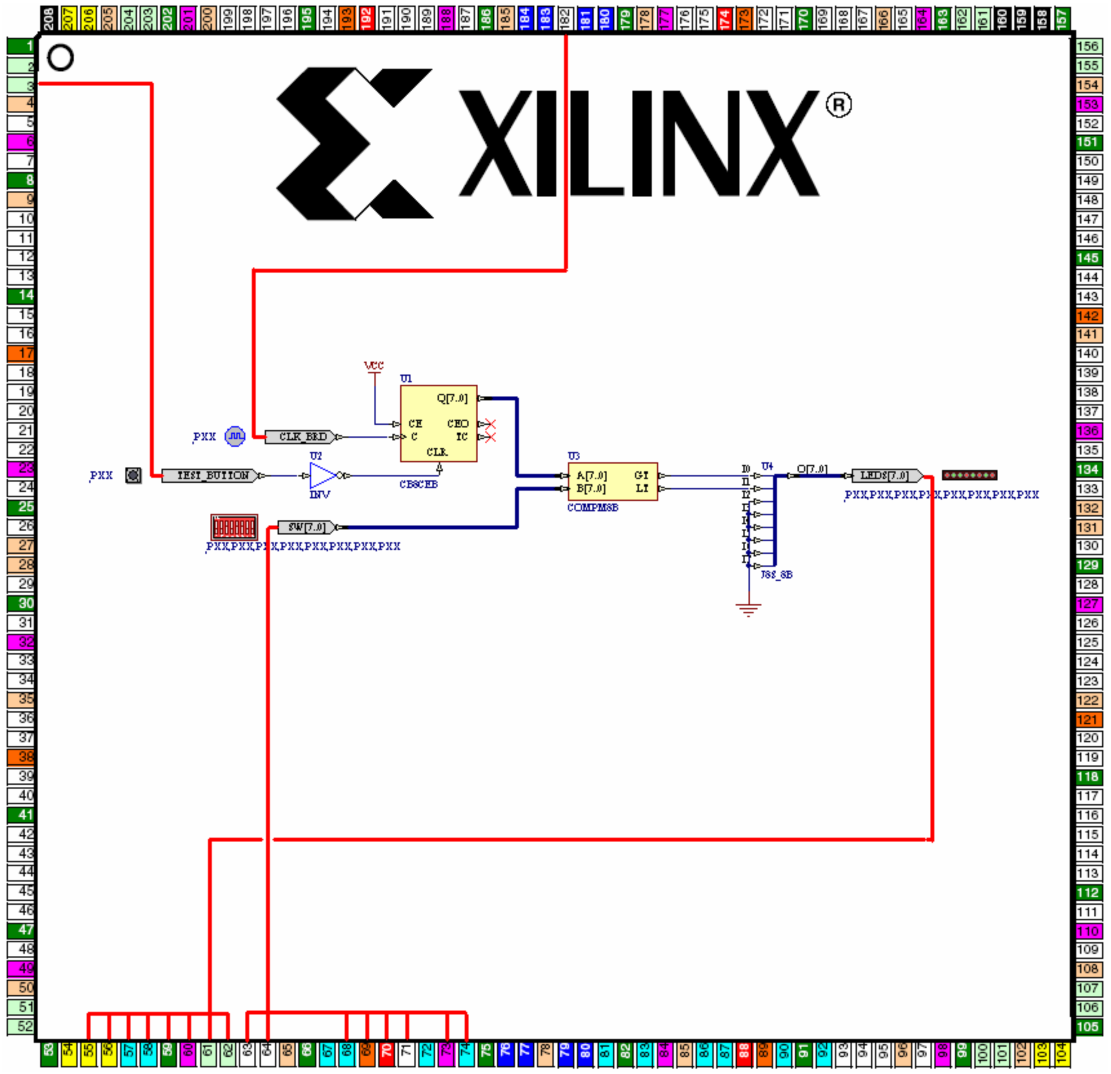


Figure 19. Conceptual view showing the linkage of ports on an FPGA schematic routed to physical device pins.

6.2 Creating a new constraint file

A constraint file can be added to a project by right-clicking the FPGA project in the **Projects** panel and selecting **Add New to Project » Constraint File**. A shell constraint file will be created.

6.3 Editing a constraint file

Constraint file additions / modifications can be made by manually editing the constraint file or by using the **Design » Add/Modify Constraint** menu.

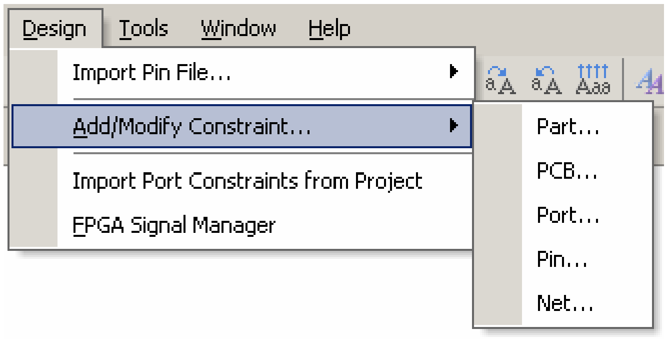
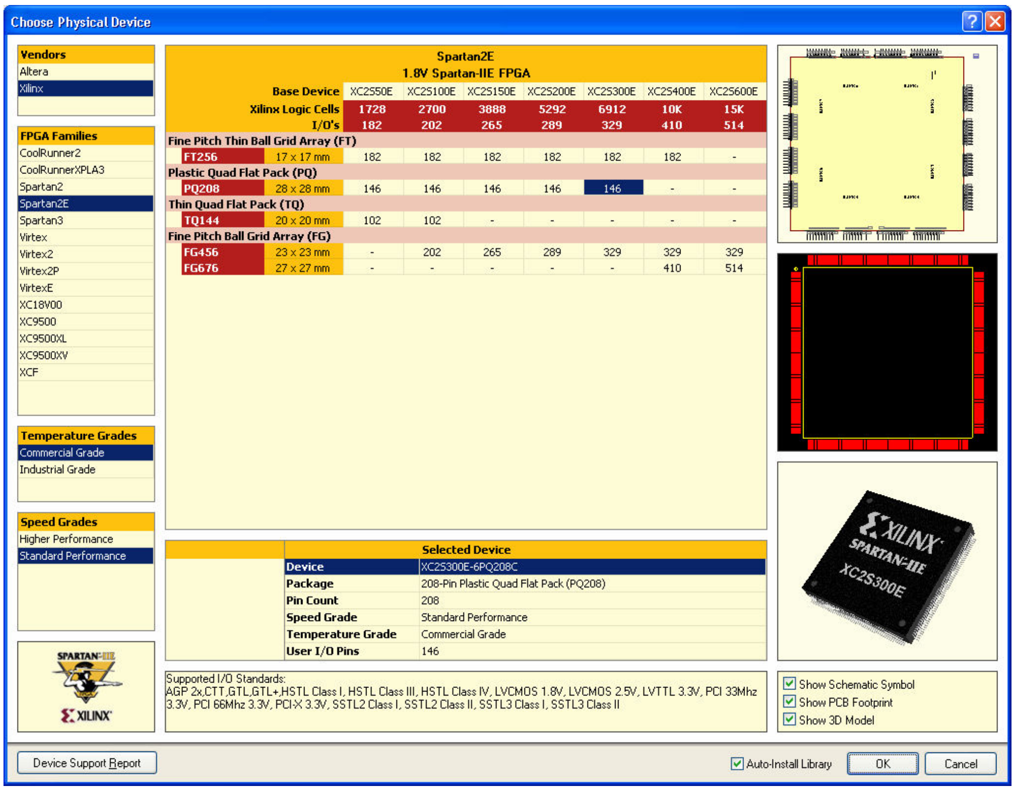


Figure 20. Add/Modify Constraint… menu options

The two main activities that will be performed on a newly created constraint file are specifying the part (device) and applying port constraints.

6.3.1 Specifying the part (device)

The design can be constrained to a specific device by selecting **Add/Modify Constraint » Part** and selecting the desired part from the **Choose Physical Device** dialog:



Spartan2E
1.8V Spartan-IIE FPGA

		XC2S50E	XC2S100E	XC2S150E	XC2S200E	XC2S300E	XC2S400E	XC2S600E
Xilinx Logic Cells		1728	2700	3888	5292	6912	10K	15K
I/O's		182	202	265	289	329	410	514
Fine Pitch Thin Ball Grid Array (FT)								
FT256	17 x 17 mm	182	182	182	182	182	182	-
Plastic Quad Flat Pack (PQ)								
PQ208	28 x 28 mm	146	146	146	146	146	-	-
Thin Quad Flat Pack (TQ)								
TQ144	20 x 20 mm	102	102	-	-	-	-	-
Fine Pitch Ball Grid Array (FG)								
FG456	23 x 23 mm	-	202	265	289	329	329	329
FG676	27 x 27 mm	-	-	-	-	-	410	514

Selected Device	
Device	XC2S300E-6PQ208C
Package	208-Pin Plastic Quad Flat Pack (PQ208)
Pin Count	208
Speed Grade	Standard Performance
Temperature Grade	Commercial Grade
User I/O Pins	146

Supported I/O Standards:
AGP 2x,CTT,GTL,GTL+,HSTL Class I, HSTL Class III, HSTL Class IV, LVCMOS 1.8V, LVCMOS 2.5V, LVTTL 3.3V, PCI 33Mhz 3.3V, PCI 66Mhz 3.3V, PCI-X 3.3V, SSTL2 Class I, SSTL2 Class II, SSTL3 Class I, SSTL3 Class II

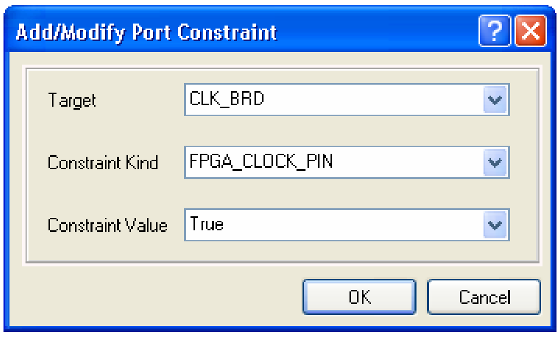
Figure 21. Choose Physical Device dialog box.

Select the Vendor, Family, Device and Temperature/Speed grades as desired and click OK. A line similar to the one below will be automatically inserted into the constraint file:

```
Record=Constraint | TargetKind=Part | TargetId=XC2S300E-6PQ208C
```

6.3.2 Specifying port constraints

Use the **Add/Modify Constraint » Port** to apply a constraint to a port in the FPGA project.


Add/Modify Port Constraint

Target: CLK_BRD

Constraint Kind: FPGA_CLOCK_PIN

Constraint Value: True

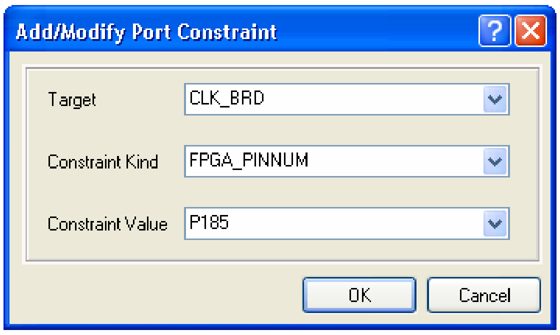
Figure 22. Add/Modify Port Constraint dialog box.

Selecting OK from the dialog box in Figure 22 will cause the following constraint to be added to the constraint file:

```
Record=Constraint | TargetKind=Port | TargetId=CLK_BRD | FPGA_CLOCK_PIN=True
```

This constraint will ensure that the Vendor FPGA tools route the CLK_BRD port to a specialized clock pin on the target device.

Alternatively, the `FPGA_PINNUM` constraint can be specified to lock the port to a specific pin on the target device.


Add/Modify Port Constraint

Target: CLK_BRD

Constraint Kind: FPGA_PINNUM

Constraint Value: P185

Figure 23. Add/Modify Port Constraint dialog box.

Selecting **OK** from the dialog box in Figure 23 will add the constraint **`FPGA_PINNUM`**`=P185` to the `CLK_BRD` port constraint.

6.3.3 Applying further constraints

Additional constraints other than those discussed here are available. Consult the **TR0103 Constraint File Reference.pdf** document for more information.

To summarize:

- **Constraint files** store implementation specific information such as device pin allocations and electrical properties.
- A **Configuration** is a grouping of one or more constraint files and describes how the FPGA project should be built.

6.4 NanoBoard constraint files

Constraint files for use with the NanoBoard daughter board modules can be found in the `\Program Files\Altium Designer 6\Library\Fpga` directory. To protect these system files from inadvertent modification, it is advisable to make this directory 'read only'.

6.5 Configurations

A *Configuration* is a set of one or more constraint files that must be used to target a design for a specific output. The migration of a design from prototype to production will often involve several PCB iterations and possibly even different FPGA devices. In this case, a separate configuration would be used to bring together constraint file information for each design iteration. Each new configuration (and its associated constraint file(s)) is stored with the project and can be recalled at any time.

Because configurations can contain multiple constraint files, it can sometimes be helpful to split constraint information across multiple constraint files. Usually one would separate the constraint files according to the class of information they contain:

6.5.1 Device and board constraint information:

The specific FPGA device must be identified and ports defined in the top level FPGA design must be mapped to specific pin numbers.

6.5.2 Device resource constraint information:

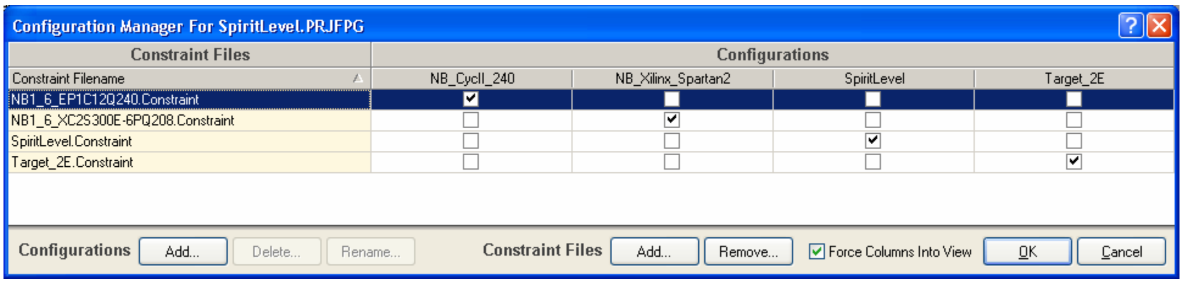
In some designs it may be advantageous to make use of vendor specific resources that are unique to a given FPGA device. Some examples are hardware multiplication units, clock multipliers and memory resources.

6.5.3 Project or design constraint information:

This would include requirements which are associated with the logic of the design, as well as constrains on its timing. For example, specifying that a particular logical port must be allocated to global clock net, and must be able to run at a certain speed.

6.6 Configuration Manager

The grouping of multiple constraints into a single configuration is managed via the *Configuration Manager*; accessible by right-clicking the FPGA project in the Projects panel and selecting **Configuration Manager** from the menu.



Configuration Manager For SpiritLevel.PRJFPG

Constraint Files	Configurations			
Constraint Filename	NB_CycII_240	NB_Xilinx_Spartan2	SpiritLevel	Target_2E
NB1_6_EP1C12Q240.Constraint	✓			
NB1_6_XC2S300E-6PQ208.Constraint		✓		
SpiritLevel.Constraint			✓	
Target_2E.Constraint				✓

Figure 24. *Configuration Manager showing multiple configurations and constraint files.*

Figure 24 shows the *Configuration Manager* dialog for a project that contains multiple configurations and constraint files. The Constraint files are listed in the left column and can be included in a Configuration (listed as the headings in the four right columns) by placing a tick at the row/column intersection point. Although this example only shows one constraint file being used in each of the configurations, there is no reason why a constraint file can't be used by more than one configuration nor is there any reason why a configuration can't make use of multiple constraint files.

6.7 Exercise 2 – Configuring MyPWM

1. Right click on the MyPWM.PRJFPG in the **Projects** panel and select *Configuration Manager* from the menu. The *Configuration Manager* should be empty as depicted in Figure 25.

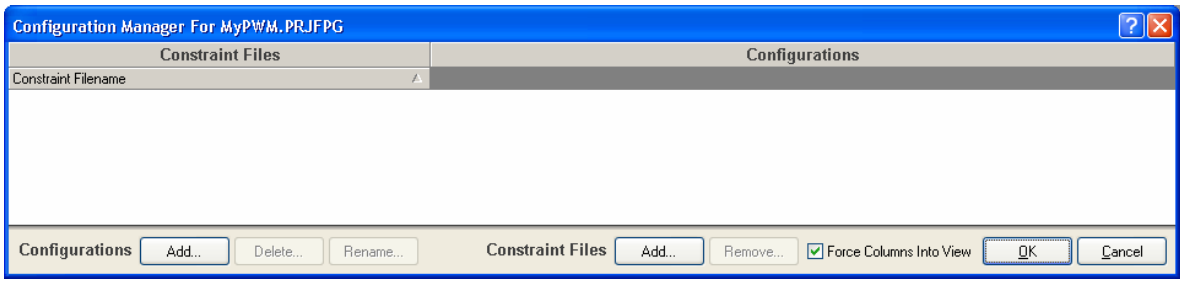


Figure 25. Configuration Manager with no Constraint Files or Configurations defined.

2. Because we are targeting our design for the NanoBoard, we will be using an existing constraint file that has been previously defined for the Spartan-II daughter board. Select the Add button next to the Constraint Files label. The Choose Constraint files to add to Project dialog box will be displayed. By default it should open in the Altium Designer 6\Library\FPGA directory. If it hasn't defaulted to this location then navigate to it.
3. Select the constraint file labelled NB1_6_XC2S300E-6PQ208.Constraint and click Open. You should see the same as Figure 26.

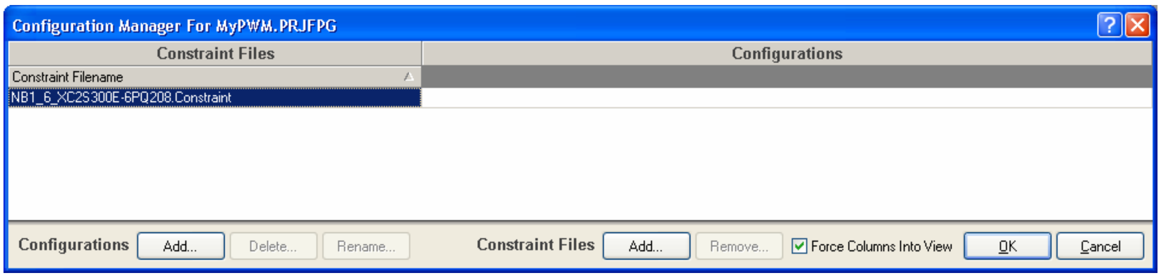


Figure 26. Configuration Manager with Spartan-II daughter board constraint file present.

4. We shall now create a configuration that will make use of this constraint file. Select the **Add** button located next to the **Configurations** label.

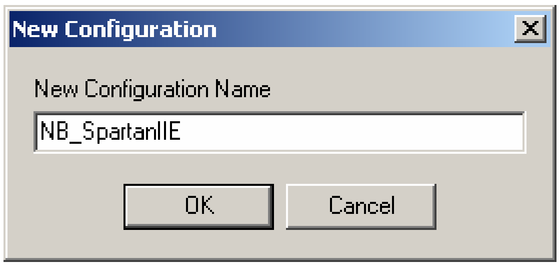


Figure 27. Specifying a new configuration.

5. Call the new configuration `NB_SpartanIIE` and select **OK**.
6. Click on the checkbox to link the `NB1_6_XC2S300E-6PQ208.Constraint` file to the `NB_SpartanIIE` configuration.

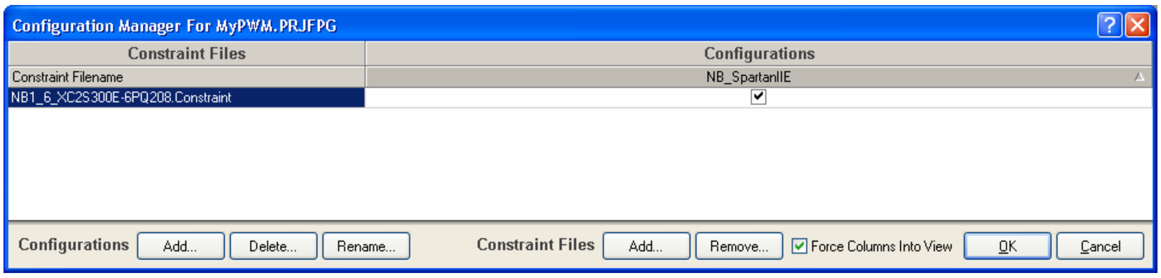


Figure 28. Configuration Manager with a constraint file and configuration specified.

7. Select **OK** to close the *Configuration Manager*.
8. Notice how the newly added constraint file is now present under the **Settings** folder in the **Projects** panel.
9. Save your project if you haven't already done so.

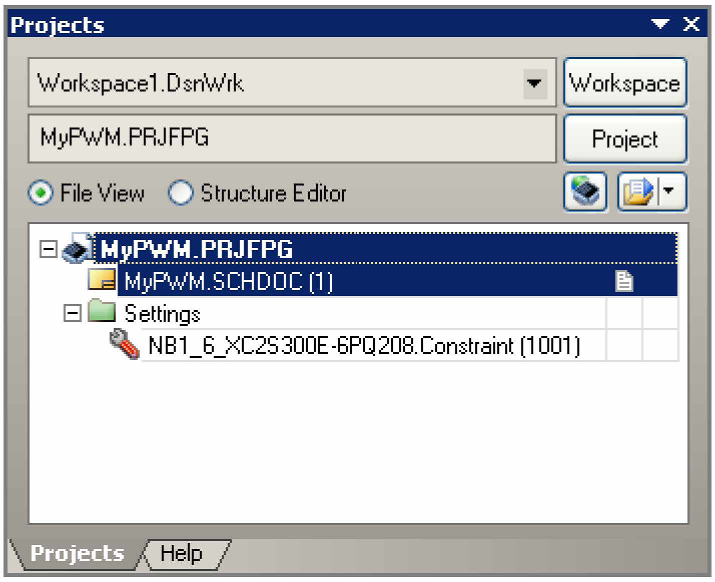


Figure 29. Projects panel after a constraint file has been added.

7 Running the design

Having just *configured* our design for the NanoBoard the next step is to build and run the design on the NanoBoard.

7.1 Overview

Before an FPGA design can be downloaded onto its target hardware, it must first undergo a multi-stage build process. This process is akin to the compilation process that software undergoes in order to create a self-contained program. In this section we will discuss the various steps necessary to build an FPGA design to the point where it is ready to be downloaded onto the target device.

7.2 Controlling the build process

The process of converting a schematic or VHDL description of a digital circuit into a *bit* file that can be downloaded onto an FPGA is quite complex. Fortunately, Altium Designer goes to great lengths to ensure that navigation through this process is as easy as possible. As a vendor independent FPGA development tool, Altium Designer provides a transparent interface to the vendor specific back end tools. Currently Altium Designer supports interaction with Actel Designer (Actel), Quartus II (Altera) and ISE (Xilinx) to perform FPGA processing. This is all handled seamlessly through the **Devices View** (**View » Devices**). The **Devices View** provides the central location to control the process of taking the design from the capture state through to implementing it in an FPGA.

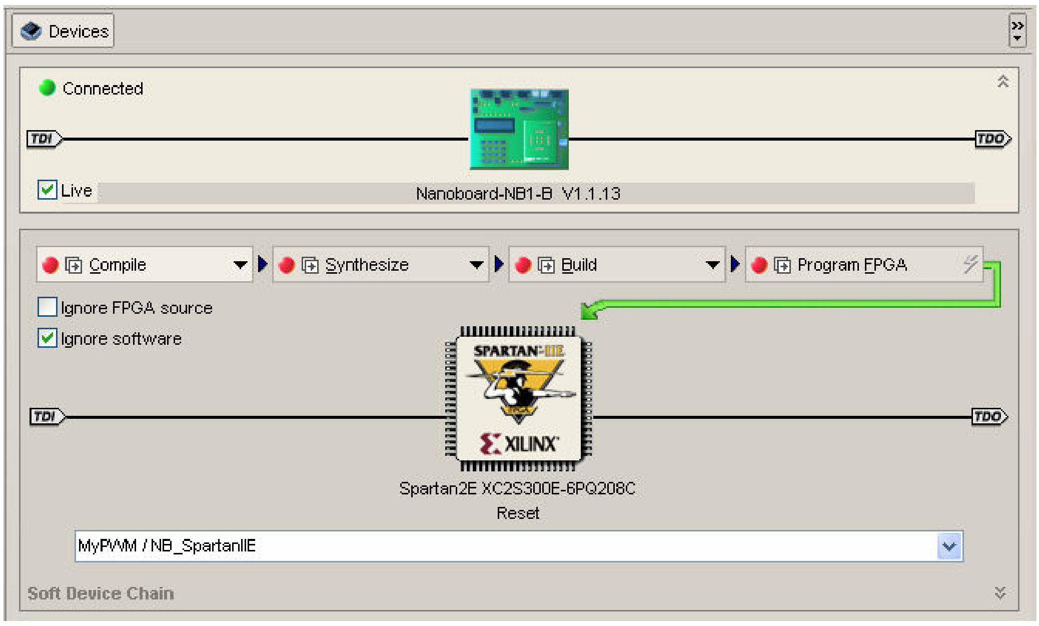


Figure 30. Devices view of an FPGA design that is yet to be processed.

The **Devices View** is accessible by clicking the **Devices View** button or by selecting **View » Devices View** from the main menu.

When run in the live mode, Altium Designer is intelligent enough to detect which daughter board device is present on the NanoBoard. In the above instance, it has detected that the Spartan2E daughter board is installed. With this information, it then searches the current project's configuration list to see if any configurations match this device. If more than one configuration is found, the drop down list below the device icon will be populated with a list of valid configurations. If no configuration can be found, the list will display the following:

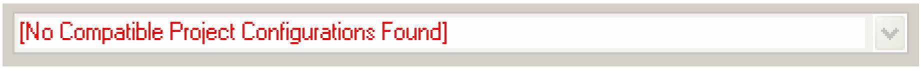

[No Compatible Project Configurations Found]

Figure 31. This message indicates that the project is not configured to target the available FPGA.

Assuming a valid configuration can be found, the simplest way to build and download a design onto the NanoBoard is to left-click on the **Program FPGA** button. This will invoke the appropriate build processes that need to be run. In the above example where no previous builds have taken place, all processes will need to be run. In other situations where a project has just been modified, it may be necessary for only a subset of the build processes to run.

7.3 Understanding the build process

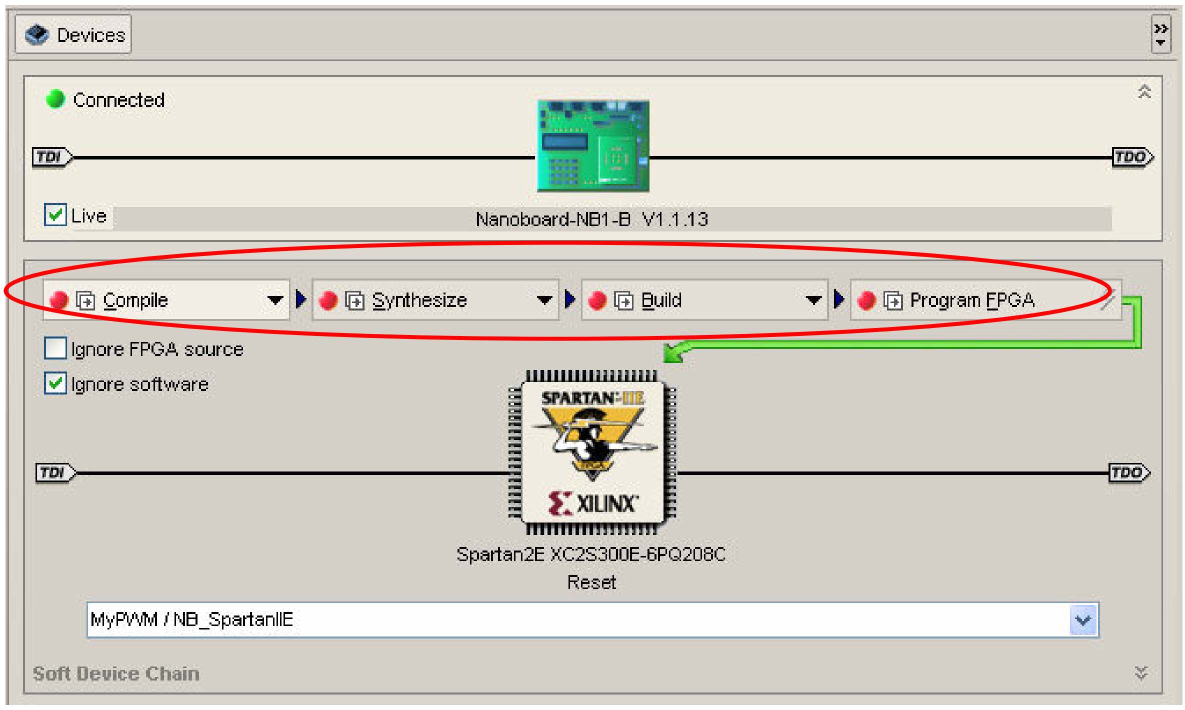


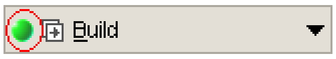
Figure 32. Navigating through the Build Process flow.

Building an FPGA project requires processing through a number of stages. Navigation through the build process is accomplished via the four steps circled in Figure 32. The function of each stage will be explained shortly.

7.4 Button regions

Each of the main buttons displayed in the Build Flow have several regions that provide information or control over the individual build stage.

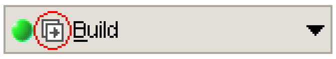
7.4.1 Status LED



The colored indicator tells you the status of that particular step in the overall build flow.

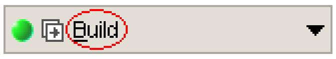
●	Grey	-	**Not Available**. The step or stage cannot be run.
●	Red	-	**Missing**. The step or stage has not been previously run.
●	Yellow	-	**Out of Date**. A source file has changed and the step or stage must be run again in order to obtain up to date file(s).
●	Blue	-	**Running**. The step or stage is currently being executed.
●	Orange	-	**Cancelled**. The step or stage has been halted by user intervention.
●	Magenta	-	**Failed**. An error has occurred while running the current step of the stage.
●	Green	-	**Up to Date**. The step or stage has been run and the generated file(s) are up to date.

7.4.2 Run all



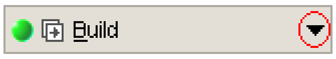
Clicking on the 'arrow' icon will force the current stage and all prior stages to run regardless of whether they have run to completion previously. Selecting this icon will force a totally clean build even if the design has been partially built.

7.4.3 Run



Selecting the 'label' region will run the current stage and any previous dependant stages that are not up to date. This is the quickest way to build a design as it only builds those portions of the design that actually require it.

7.4.4 Show sub-stages



Selecting the 'down arrow' will expose a drop down list of the various sub-stages for the current build stage. The status of the various sub-stages is indicated by the color of the status 'LED'. Where a sub-stage has failed, the associated report file can be examined to help determine the cause of the failure.

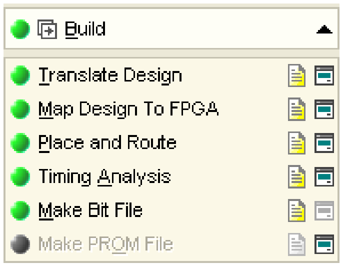


Figure 33. Sub-stages available under the main build stage.

7.5 Accessing stage reports / outputs

All generated output files are stored in a folder with the same name as the configuration used for the associated project. This folder is located in accordance with the output path defined in the **Options** tab of the *Options for Project* dialog (**Project » Project Options**). In general only files that are created as part of the build process should be located here. This ensures that projects can be compacted by deleting this directory without fear of loss of important information.

Where a report is available upon running a stage step, clicking on the associated 🗎 icon can access it. Use this feature to access detailed information relating to why a specific stage may have failed to build.

7.6 Build stages

We will now explain the different stages in the build process.

7.6.1 Compile

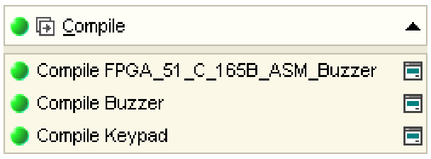


Figure 34. Compile stage of the process flow.

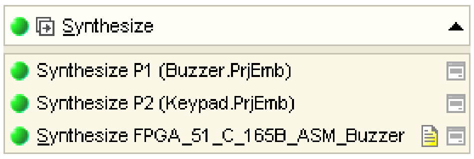
This stage of the process flow is used to perform a compile of the source documents in the associated FPGA project. If the design includes any microcontroller cores, the associated embedded projects are also compiled – producing a Hex file in each case.

This stage can be run with the **Devices** view configured in either Live or Not Live mode.

The compile process is identical to that performed from the associated Project menu. Running this stage can verify that the captured source is free of electrical, drafting and coding errors.

Note: The source FPGA (and embedded) project(s) must be compiled – either from the **Projects** panel or by running the Compile stage in the **Devices** view – in order to see Nexus-enabled device entries in the Soft Devices region of the **Devices** view.

7.6.2 Synthesize


Synthesize

Synthesize P1 (Buzzer.PrjEmb)

Synthesize P2 (Keypad.PrjEmb)

Synthesize FPGA_51_C_165B_ASM_Buzzer

Figure 35. Synthesize stage of the process flow.

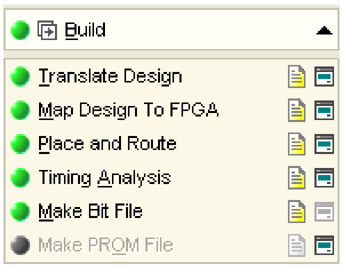
This stage of the process flow is used to synthesize the compiled FPGA project, as well as any other components that need to be generated and synthesized to specific device architectures. The vendor place and route tools subsequently use the synthesis files generated, during the build stage of the flow. Running this stage will determine whether the design is synthesizable or not.

This stage can be run with the **Devices** view configured in either Live or Not Live mode.

The actual steps involved in providing a top-level EDIF netlist and satellite synthesis model files for use by the next stage in the process flow can be summarized as follows:

- The cores for any design/device specific blocks used in the FPGA design will be auto-generated and synthesized (e.g. a block of RAM wired to an OCD-version micro controller for use as external Program memory space). These synthesized models will contain compiled information from the embedded project (Hex file).
- The main FPGA design is then synthesized. An intermediate VHDL file for each schematic sheet in the design will be generated and a top-level EDIF netlist created using these and any additional VHDL source files.
- For the particular physical device chosen, synthesized model files associated with components in the design will be searched for and copied to the relevant output folder. Both System and User presynthesized models are supported.
- The top-level folder for System presynthesized models is the `\Program Files\Altium Designer 6\Library\Edif` folder, which is sub-divided by Vendor and then further by device family.
- The top-level folder for user presynthesized models is defined in the **FPGA – Synthesis** page of the *Preferences* dialog, accessed under the **Tools** menu.
- The following list summarizes the order (top to bottom = first to last) in which folders are searched when looking for a synthesized model associated with a component in the design:
  - FPGA project folder
  - User models top folder\Vendor folder\Family folder
  - User models top folder\Vendor folder
  - User models top folder
  - System models top folder (Edif)\Vendor Folder\Family folder
  - System models top folder (Edif)\Vendor folder
  - System models top folder (Edif).

7.6.3 Build



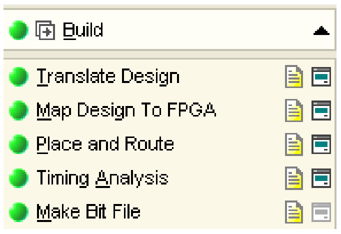


Figure 36. Build stage of the process flow for Xilinx (left) and Altera (right) devices.

This stage of the process flow is used to run the vendor place and route tools. This stage can be run with the **Devices** view configured in either live or not live mode.

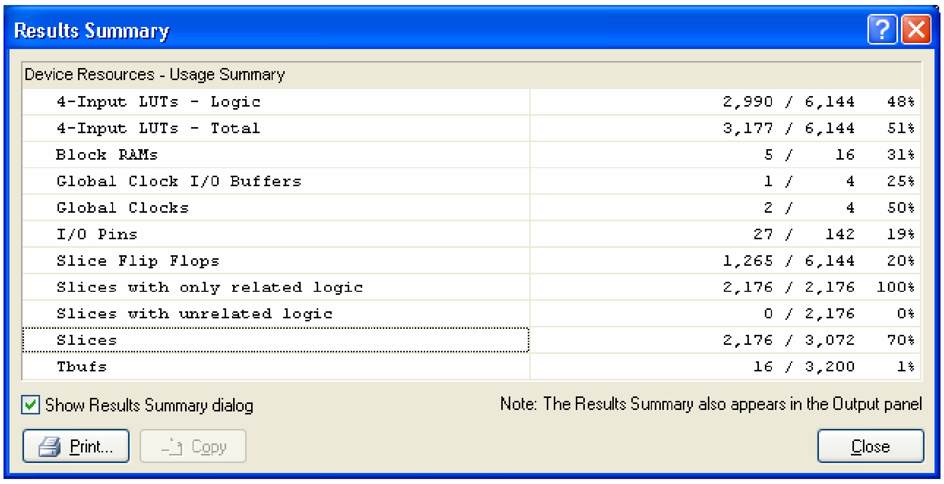
Running the tools at this stage can verify if a design will indeed fit inside the chosen physical device. You may also wish to run the Vendor tools if you want to obtain pin assignments for importing back into the relevant constraint file.

The end result of running this stage is the generation of an FPGA programming file that will ultimately be used to programming the physical device with the design. There are essentially five main stages to the build process:

- **Translate Design** – uses the top-level EDIF netlist and synthesized model files, obtained from the synthesis stage of the process flow, to create a file in Native Generic Database (NGD) format – i.e. vendor tool project file
- **Map Design to FPGA** – maps the design to FPGA primitives
- **Place and Route** - takes the low-level description of the design (from the mapping stage) and works out how to place the required logic inside the FPGA. Once arranged, the required interconnections are routed
- **Timing Analysis** – performs a timing analysis of the design, in accordance with any timing constraints that have been defined. If there are no specified constraints, default enumeration will be used
- **Make Bit File** – generates the programming file that is required for downloading the design to the physical device.

When targeting a Xilinx device, an additional stage is available – **Make PROM File**. This stage is used when you want to generate a configuration file for subsequent download to a Xilinx configuration device on a Production board.

After the Build stage has completed, the **Results Summary** dialog will appear (Figure 20). This dialog provides summary information with respect to resource usage within the target device. Information can be copied and printed from the dialog. The dialog can be disabled from opening, should you wish, as the information is readily available in the **Output** panel or from the report files produced during the build.


Results Summary

Device Resources - Usage Summary

4-Input LUTs - Logic	2,990 / 6,144	48%
4-Input LUTs - Total	3,177 / 6,144	51%
Block RAMs	5 / 16	31%
Global Clock I/O Buffers	1 / 4	25%
Global Clocks	2 / 4	50%
I/O Pins	27 / 142	19%
Slice Flip Flops	1,265 / 6,144	20%
Slices with only related logic	2,176 / 2,176	100%
Slices with unrelated logic	0 / 2,176	0%
Slices	2,176 / 3,072	70%
Tbufs	16 / 3,200	1%

[✓] Show Results Summary dialog

Note: The Results Summary also appears in the Output panel

Figure 37. Summarizing resource usage for the chosen device.

7.6.4 Program

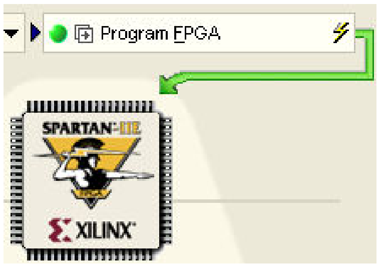


Figure 38. Program FPGA stage of the process flow.

This stage of the process flow is used to download the design into the physical FPGA device on a NanoBoard or production board. This stage is only available when the **Devices** view is configured in **Live** mode.

This stage of the flow can only be used once the previous three stages have been run successfully and an FPGA programming file has been generated.

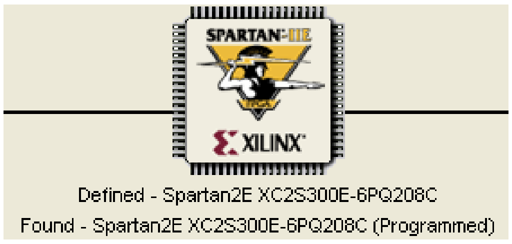

SPARTAN-IIE
XILINX
Defined - Spartan2E XC2S300E-6PQ208C
Found - Spartan2E XC2S300E-6PQ208C (Programmed)

Figure 39. Successful programming of the physical FPGA device.

As the programming file is downloaded to the device via the JTAG link, the progress will be shown in the Status bar. Once successfully downloaded, the text underneath the device will change from ‘Reset’ to ‘Programmed’ (Figure 39) and any Nexus-enabled devices on the soft chain will be displayed as ‘Running’ (Figure 40).

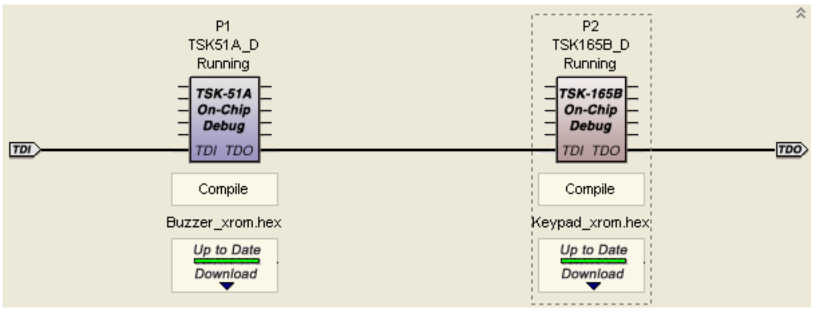


Figure 40. Soft devices running after successful program download.

7.7 Configuring a build stage

Should you wish to configure any of the specific options associated with each of the different sub-stages in the FPGA build flow you can do so by clicking on the appropriate configuration icon.

Consider the case where you want to generate a PROM file for subsequent download to a Xilinx configuration device on a production board. In the process flow associated to the targeted FPGA device, expand the build section. The last entry in the build menu is **Make PROM File** (Figure 41).

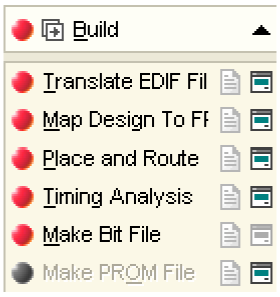
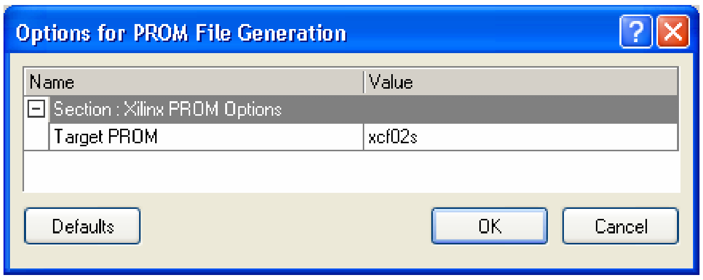


Figure 41. Accessing the command to generate a PROM file.

Clicking on the icon, to the far right of this menu entry, will open the **Options for PROM File Generation** dialog (Figure 42). From here you can choose the non-volatile configuration device that will be used by the production board to store the FPGA configuration.


Options for PROM File Generation

Name	Value
Section : Xilinx PROM Options	
Target PROM	xcf02s

Figure 42. Accessing the options dialog for PROM file generation.

7.8 How Altium Designer interacts with back-end vendor tools

If you are already familiar with the build flows offered by Altera and Xilinx, you will be familiar with one or both of the following panels:

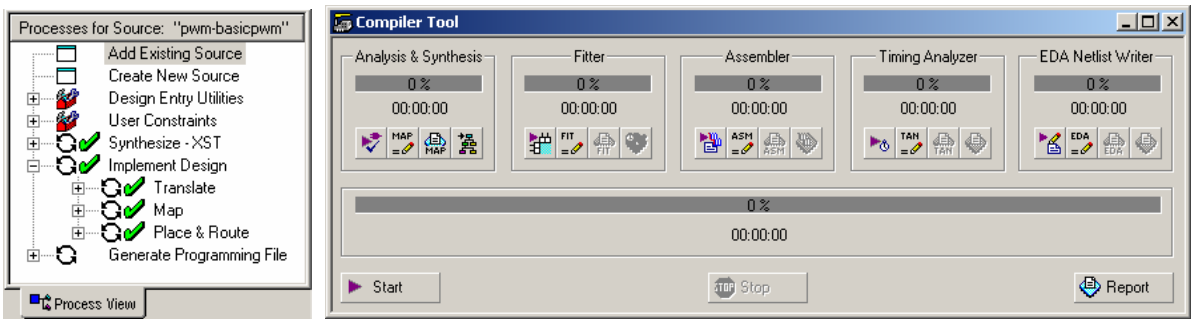


Figure 43. Xilinx (left) and Altera (right) vendor tool interfaces.

Although Altium Designer has its own VHDL synthesizer, it is reliant on back-end vendor tools to implement the design on a specific device. This makes entire sense, as it is the device vendors who have the most intimate knowledge of their specific devices and who have already developed well-proven targeting technologies.

Most vendor specific tools have been developed in a modular fashion and contain a number of separate executable programs for each phase of the implementation process. The vendor GUIs that are presented to the user are co-coordinating programs that simple pass the appropriate parameters to back-end, command-line programs.

When it comes to FPGA targeting, Altium Designer operates in a similar fashion in that it acts as a coordinator of back-end, vendor-specific programs. Parameters that need to be passed from the Altium Designer front-end to the vendor-specific back-end programs are handled by a series of text-based script files. Users who are already familiar with the back-end processing tools may find some use in accessing these script files should they wish to modify or 'tweak' interaction with back-end processing tools. This however is considered a highly advanced topic and one that should be handled cautiously. Ensure backups are taken prior to modification.

The files controlling interaction with vendor-specific back-end tools can be found in the `System` directory under the `Altium Designer 6` install directory. The naming convention used for these files is:

Device[Options | Script]_<vendor>[_<tool> | <family>].txt

i.e. `DeviceOptions_Xilinx_PAR.txt` controls the default options for Xilinx's Place and Route tool.

7.9 Exercise 3 – Run MyPWM on the NanoBoard

In this exercise we shall take our previously developed PWM design and run it on the NanoBoard.

1. Ensure that the NanoBoard is correctly connected to the PC, the XC2S300E (Xilinx) daughter board is loaded, and the NanoBoard is switched on.
2. Open the Devices View and ensure the Live checkbox is ticked.
3. Click on the 'label' region of the Program FPGA button in the FPGA Build flow. The design will begin building and may take a moment or two to complete.
4. If any build errors occur, diagnose and rectify the error and attempt the build process again.
5. Once downloaded, verify the operation of the design by switching different DIP switches off and on. You should notice a change in the LED illumination.

8 Embedded instruments

8.1 Overview

So far we have built our PWM FPGA design and run it on the NanoBoard. Fortunately this design provided an output on the LEDs that enabled us to immediately verify that the circuit was performing as we expected. But how do we verify other designs? In this section we will introduce the range of embedded instruments that can be integrated into FPGA designs to facilitate on-chip testing and debugging.

8.2 On-Chip debugging

A big concern of many embedded systems designers transitioning to FPGA based design is the issue of debugging; how does one see *inside* an FPGA circuit to diagnose a fault? What these people fail to recognize, however, is that the flexibility of FPGA devices enables typical test and measurement virtual instruments to be wired *inside* the device leading to far easier debugging than what has previously been possible.

The Altium Designer system includes a host of virtual instruments that can be utilized to gain visibility into the hardware and quickly diagnose illusive bugs. These instruments can be found in the `FPGA Instruments.IntLib` integrated library. The 'hardware' portion of the instrument is placed and wired on the schematic like other components. Once the design has been built, real time interaction with each instrument is possible from the **Devices View**.

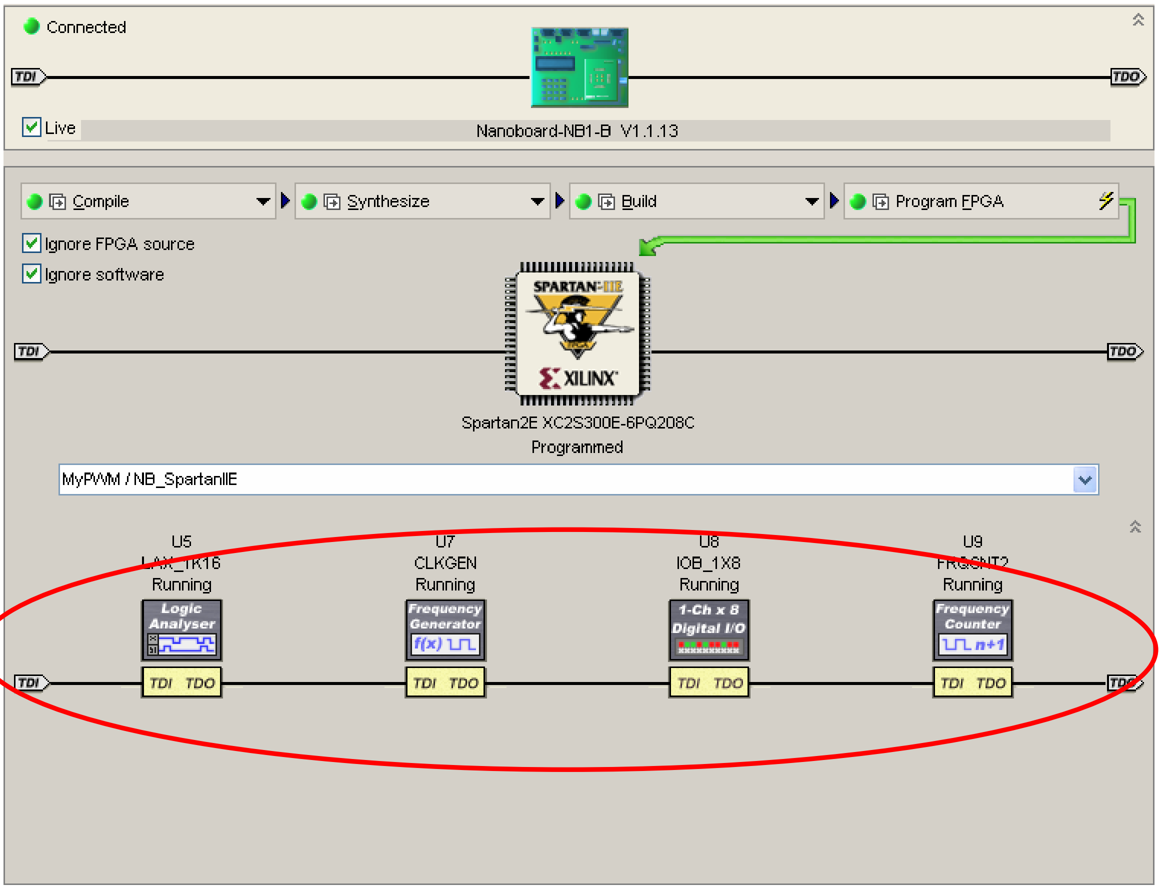


Figure 44. Embedded instruments displayed in the devices view.

The controls for the individual embedded instruments can be accessed by double-clicking their associated icon in the **Devices View**.

8.3 CLKGEN

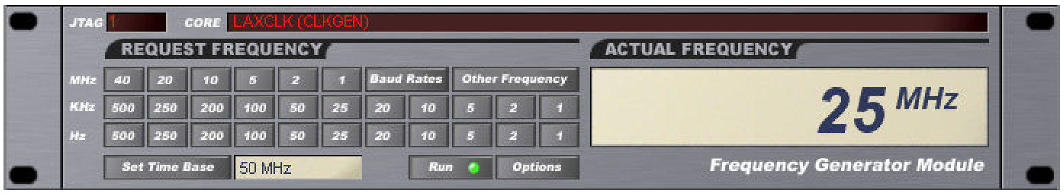
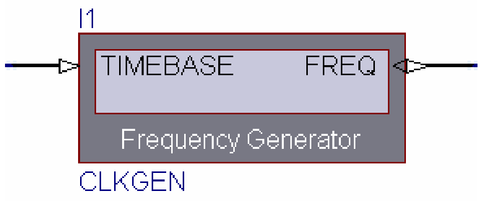


Figure 45. Frequency generator, used to generate the specified frequency



The frequency generator outputs a 50% duty cycle square wave, of the specified frequency. Clicking the appropriate button can choose a number of predefined frequencies, or a custom frequency can be selected using the **Other Frequency** button. If the specified frequency cannot be generated the closest possible is generated and the error shown on the display. Note that when the frequency generator is instantiated in the FPGA it will not be running, you must click the Run button to generate an output.

8.4 FRQCNT2

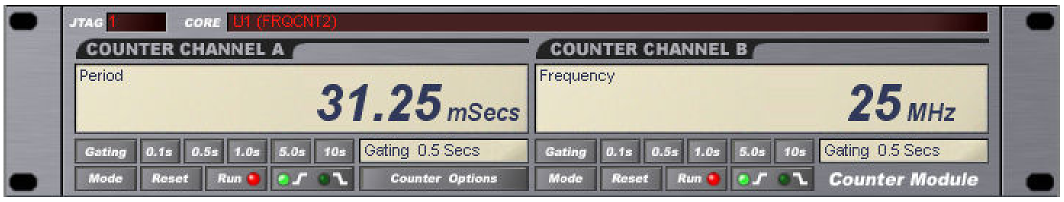
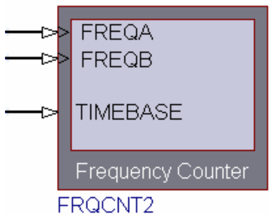


Figure 46. Frequency counter, used to measure frequency in the design



The frequency counter is a dual input counter that can display the measured signal in 3 different modes; as a frequency, period, or number of pulses.

8.5 IOB_x

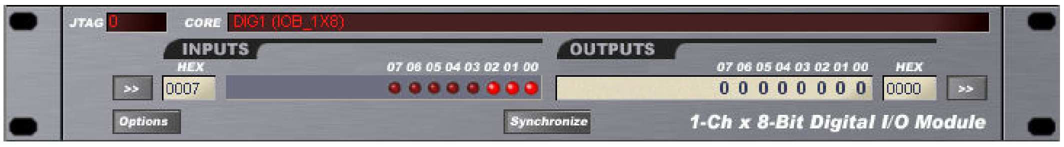
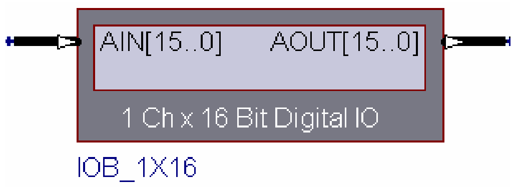


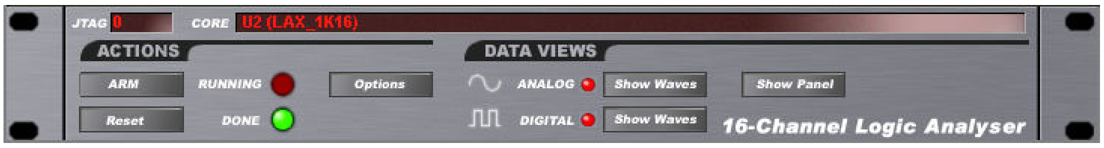
Figure 47. Digital IO module, used to monitor and control nodes in the design

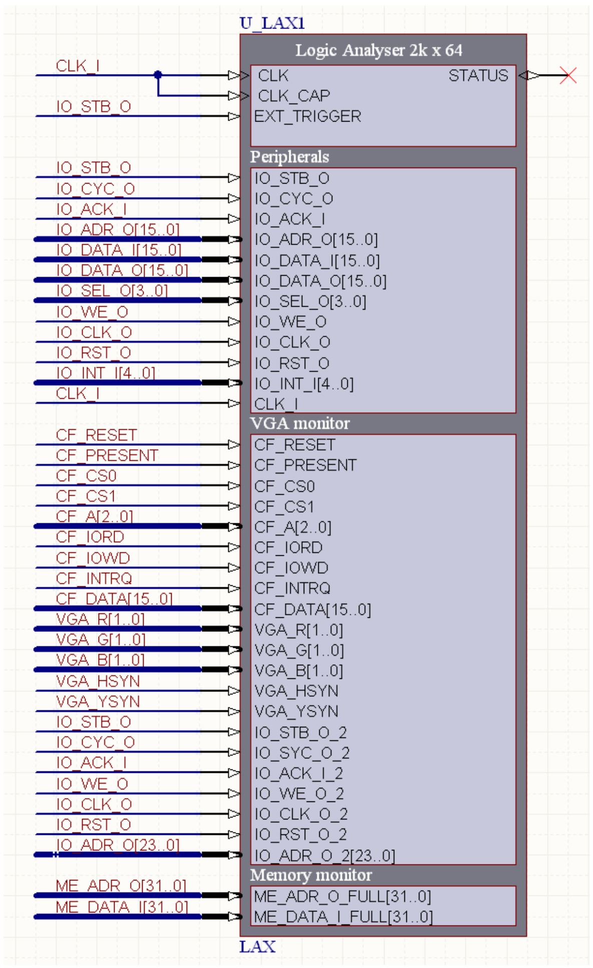


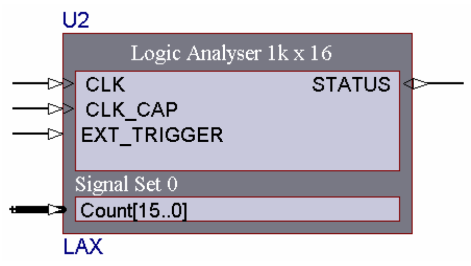
The digital I/O is a general-purpose tool that can be used for both monitoring and activating nodes in the circuit. It is available in either 8-bit wide or 16-bit wide variants, with 1 to 4 channels.

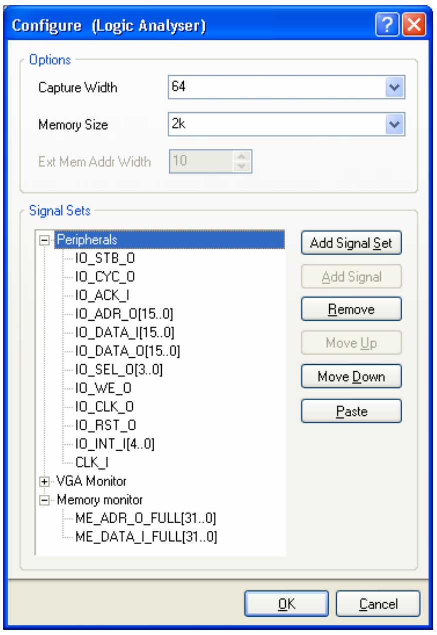
Each input bit presents as a LED, and the set of 8 or 16 bits also presents as a HEX value. Outputs can be set or cleared individually by clicking the appropriate bit in the Outputs display. Alternatively typing in a new HEX value in the HEX field can alter the entire byte or word. If a HEX value is entered you must click the >> button to output it. The Synchronize button can be used to transfer the current input value to the outputs.

8.6 LAX_x








Configure (Logic Analyser)

Options

Capture Width: 64

Memory Size: 2k

Ext Mem Addr Width: 10

Signal Sets

- Peripherals
  - IO_STB_O
  - IO_CYC_O
  - IO_ACK_I
  - IO_ADR_O[15..0]
  - IO_DATA_I[15..0]
  - IO_DATA_O[15..0]
  - IO_SEL_O[3..0]
  - IO_WE_O
  - IO_CLK_O
  - IO_RST_O
  - IO_INT_I[4..0]
  - CLK_I
- VGA Monitor
- Memory monitor
  - ME_ADR_O_FULL[31..0]
  - ME_DATA_I_FULL[31..0]

Add Signal Set | Add Signal | Remove | Move Up | Move Down | Paste

OK | Cancel

Figure 48. The logic analyzer instrument at the top, with two variations of the configurable LAX shown below it. The LAX component on the left has been configured to accept 3 different sets of 64 signals (signal sets), the one on the right has one signal set of 16 bits. The Configure dialog is used to set the capture width, memory size and the signal sets.

The logic analyzer allows you to capture multiple snapshots of multiple nodes in your design. Use the LAX to monitor multiple nets in the design and display the results as a digital or an analog waveform.

The LAX is a configurable component. Configure it to simultaneously capture 8, 16, 31 or 64 bits. The number of capture snapshots is defined by the amount of capture memory; this ranges from 1K to 4K of internal storage memory (using internal FPGA memory resources). It can also be configured to use external memory. This requires you to wire it to FPGA memory resources or to off-chip memory (eg, NanoBoard Memory).

After placing the configurable LAX from the library, right-click on the symbol and select **Configure** from the floating menu to open the *Configure (logic analyzer)* dialog, where you can define the **Capture Width**, **Memory Size** and the **Signal Sets**.

The Configurable LAX includes an internal multiplexer, this allows you to switch from one signal set to another at run time, display the capture data of interest. You can also trigger off one signal set while observing results of another set.

Note that the FPGA Instruments library includes a number of LAX components. The LAX component is the configurable version, all others are legacy versions.

8.6.1 Waveform display features

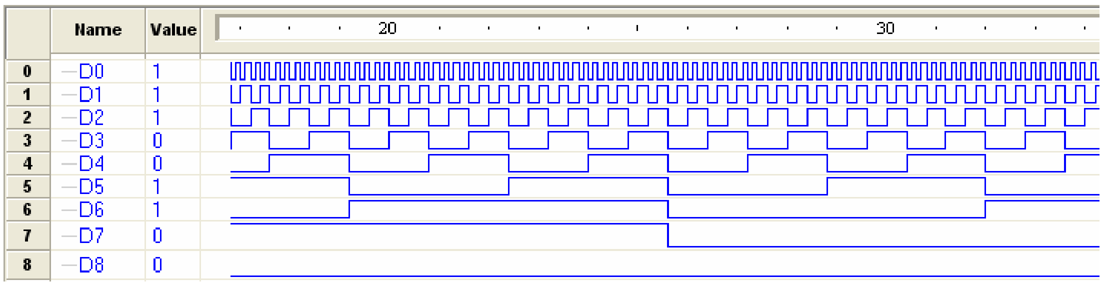


Figure 49. Digital waveform capture results from the logic analyzer

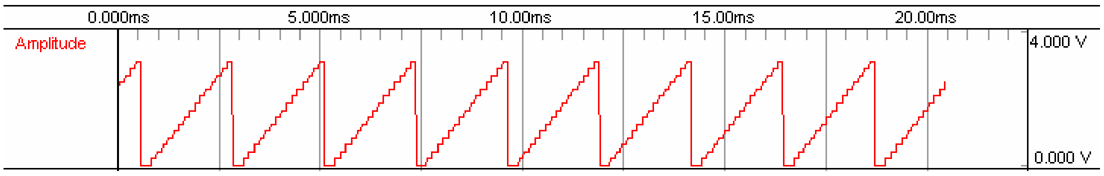


Figure 50. Analog waveform capture results from the logic analyzer

The capture results are displayed in the instrument panel. There are also two waveform display modes. The first is a digital mode, where each capture bit is displayed as a separate waveform and the capture events define the timeline. Note that the capture clock must be set in the logic analyzer options for the timeline to be calculated correctly. Click the **Show Digital Waves** button to display the digital waveform.

The second waveform mode is an analog mode, where the value on all the logic analyzer inputs is displayed as a voltage, for each capture event. The voltage range is from zero to the maximum possible count value, scaled to a default of 3.3V. Click the **Show Analog Waves** button to display the analog waveform.

8.6.2 Zooming in and out

In both the analog and digital waveform viewers it is possible to zoom in and out by hitting the **Page Up** or **Page Down** keys respectively

8.6.3 Continuous display mode

Waveforms captured by the logic analyzer can be displayed as a single pass or as a continuously updated display. Continuous updates can be enabled / disabled from the logic analyzer toolbar.

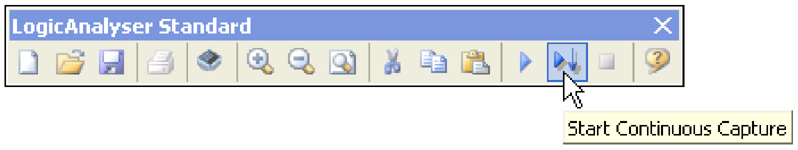


Figure 51. Enabling the continuous capture mode.

8.7 Exercise 4 – Using embedded instruments

A working design of a PWM circuit complete with embedded instruments has been prepared to illustrate the features of FPGA instruments. Your instructor will tell you where to find it on your local hard drive.

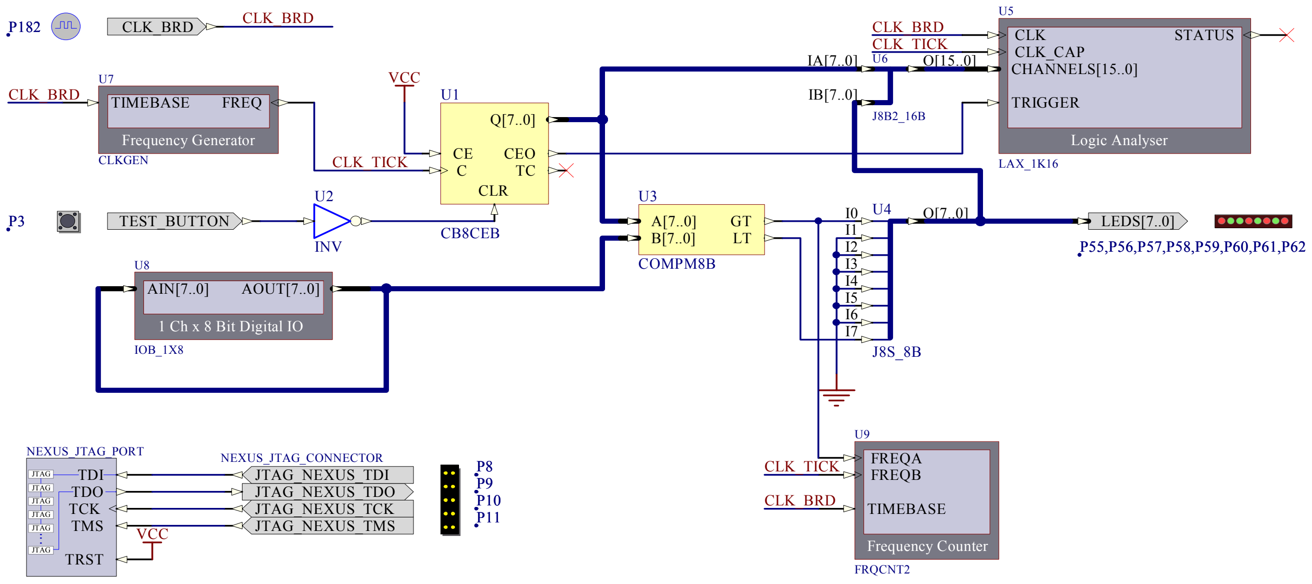


Figure 52. PWM circuit with several embedded instruments connected.

1. Open the provided project and download it to your NanoBoard.
2. Follow on your own circuit as the instructor discusses the various embedded instruments.
3. Double-click the NanoBoard icon in the **Devices View** to open the instrument rack for the NanoBoard and set its clock frequency to 50MHz.

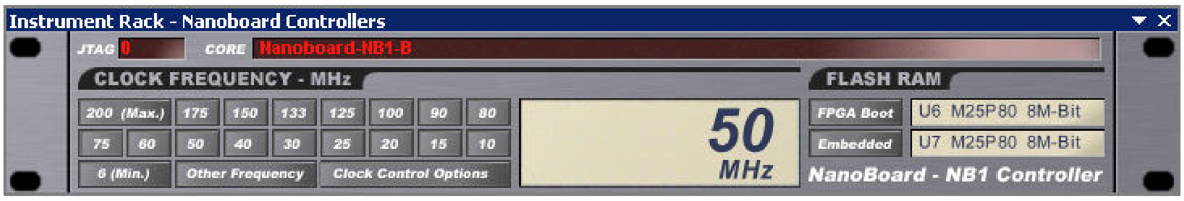
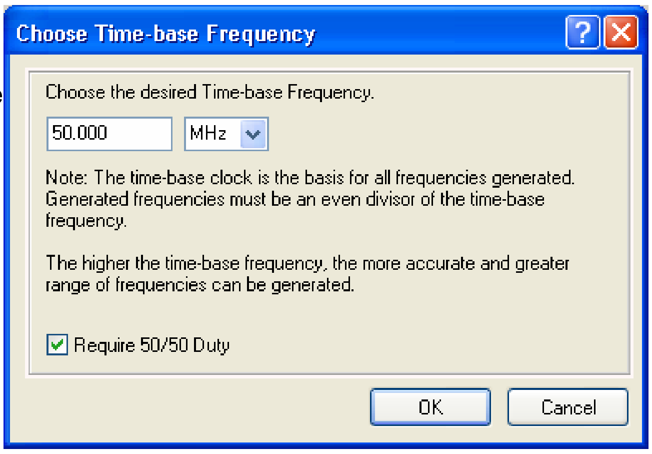


Figure 53. NanoBoard controller.

4. Open the frequency generator's instrument panel. If the time base indicated in the window next to the **Set Time Base** button is not 50 MHz then press the **Set Time Base** button to open a dialog box that will enable you to set it correctly. The **Require 50/50 Duty** checkbox should be checked.
5. The frequency generator should be set to 1MHz as indicated in Figure 55.



Choose Time-base Frequency

Choose the desired Time-base Frequency.

50.000 MHz

Note: The time-base clock is the basis for all frequencies generated. Generated frequencies must be an even divisor of the time-base frequency.

The higher the time-base frequency, the more accurate and greater range of frequencies can be generated.

☑ Require 50/50 Duty

OK Cancel

Figure 54. Counter options dialog

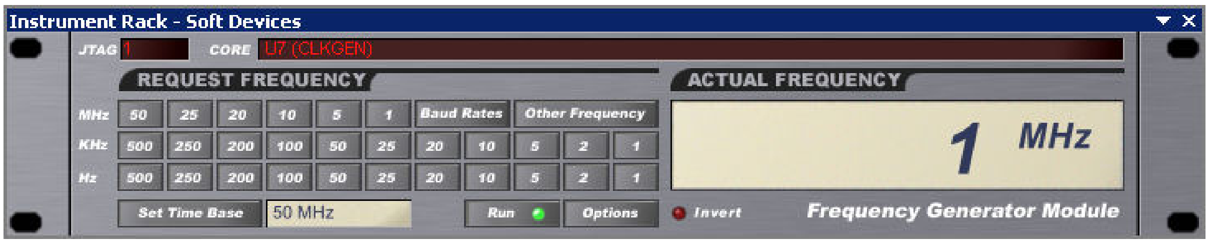
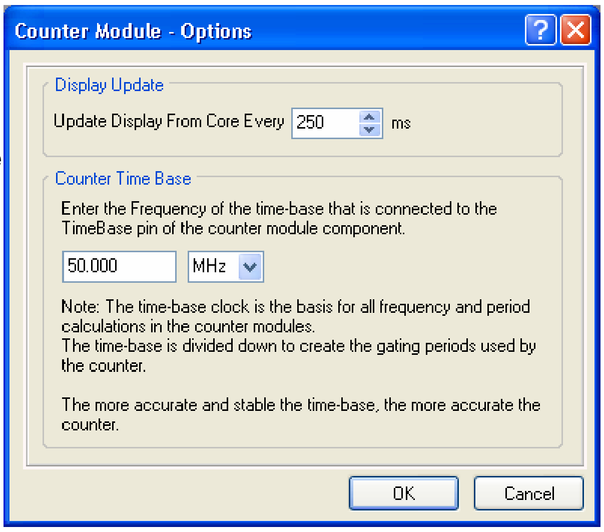


Figure 55. Frequency generator Panel

6. Open the frequency counter's instrument panel. Select the **Counter Options** button on the frequency counter module and make sure the **Counter Time Base** is also set to 50MHz (the same as the NanoBoard clock frequency), as shown in Figure 56. Press OK.
7. Use the **Mode** button as necessary on each channel of the frequency counter module to toggle the display mode between frequency, period or count. You should get the same display as depicted in Figure 57.


Counter Module - Options

Display Update

Update Display From Core Every 250 ms

Counter Time Base

Enter the Frequency of the time-base that is connected to the TimeBase pin of the counter module component.

50.000 MHz

Note: The time-base clock is the basis for all frequency and period calculations in the counter modules.
The time-base is divided down to create the gating periods used by the counter.

The more accurate and stable the time-base, the more accurate the counter.

OK Cancel

Figure 56. Counter options dialog

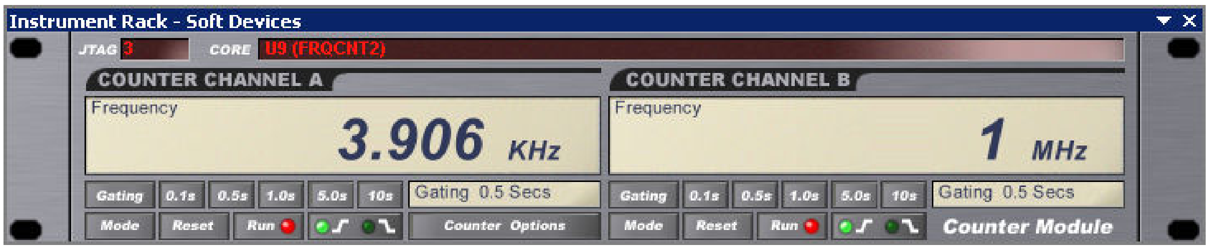


Figure 57. Frequency counter control panel

8. Open the Digital IOB's instrument panel.

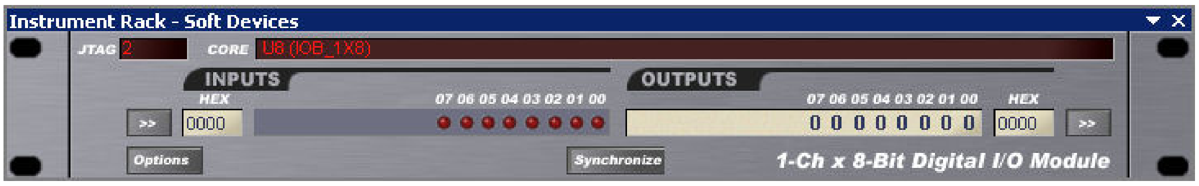


Figure 58. Digital IOB instrument control panel

9. Modify the **Outputs** of the IOB module and observe changes in the LEDs.
10. Adjust the output frequency of the frequency generator module to a lower frequency; try 1KHz. Observe the impact this has on the LEDs. Modify the **Outputs** of the IOB and observe further changes in the LEDs.
11. Adjust the output frequency of the frequency generator module back to 1MHz.
12. Open the logic analyser's instrument control panel.

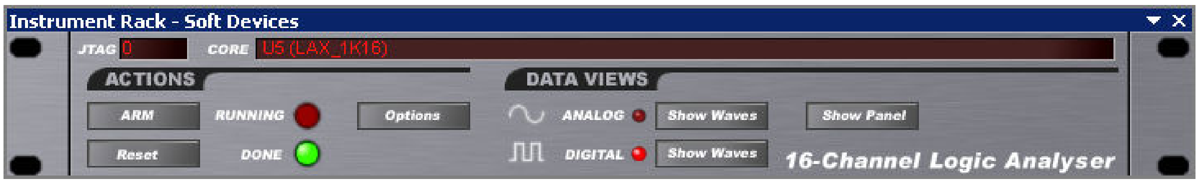
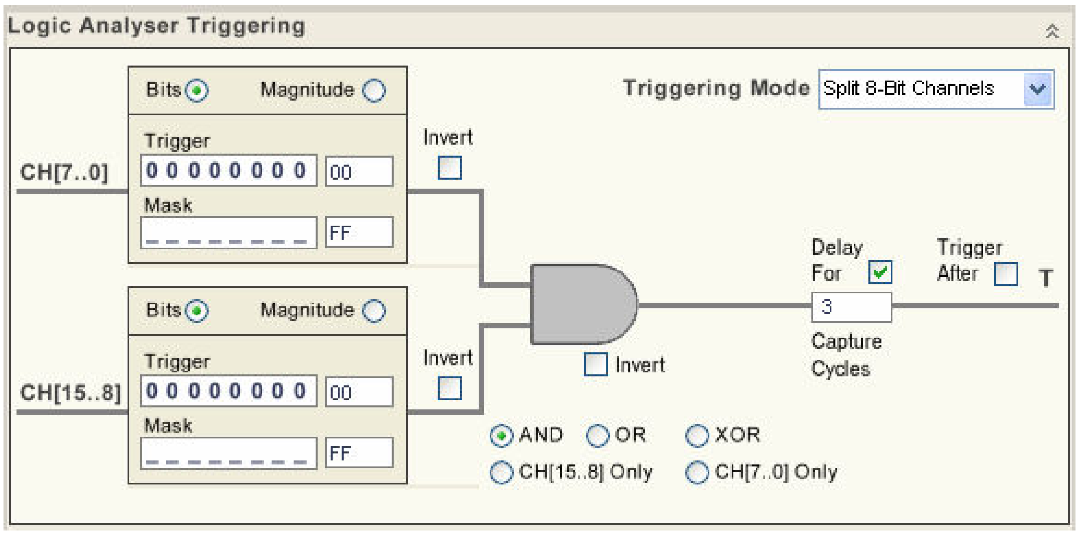


Figure 59. Logic analyser instrument control panel

13. Select **Show Panel** on the logic analyser. Set the panel up as depicted in Figure 60


Logic Analyser Triggering

Triggering Mode: Split 8-Bit Channels

CH[7..0]: [✓ Bits] Magnitude; Trigger 0 0 0 0 0 0 0 0 / 00; Mask - - - - - - - - / FF; Invert ☐

CH[15..8]: [✓ Bits] Magnitude; Trigger 0 0 0 0 0 0 0 0 / 00; Mask - - - - - - - - / FF; Invert ☐

Invert ☐

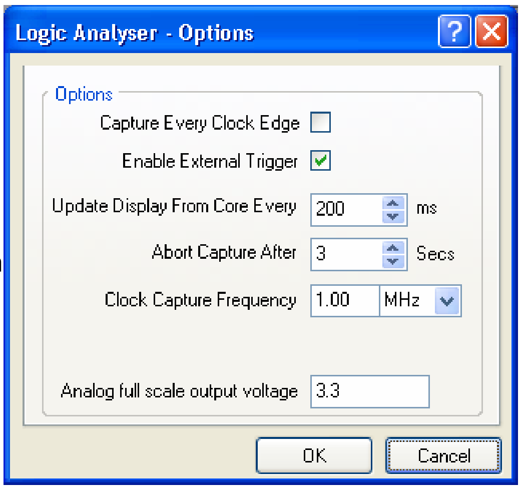
Delay For ✓ 3 Capture Cycles

Trigger After ☐ T

[✓ AND] OR XOR CH[15..8] Only CH[7..0] Only

Figure 60. Logic analyser triggering options.

14. Select **Options** on the logic analyser. Set the clock capture frequency to 1MHz – the same as the frequency generator module. Adjust the other controls to be the same as shown in Figure 61.
15. Select **Arm** and observe the waveform displayed in the waveform viewer. Select continuous mode and adjust the IOB output. Observe the change in the PWM mark-to-space ratio.


Logic Analyser - Options

Options

Capture Every Clock Edge ☐

Enable External Trigger ✓

Update Display From Core Every 200 ms

Abort Capture After 3 Secs

Clock Capture Frequency 1.00 MHz

Analog full scale output voltage 3.3

OK Cancel

Figure 61. Logic analyser setup options.

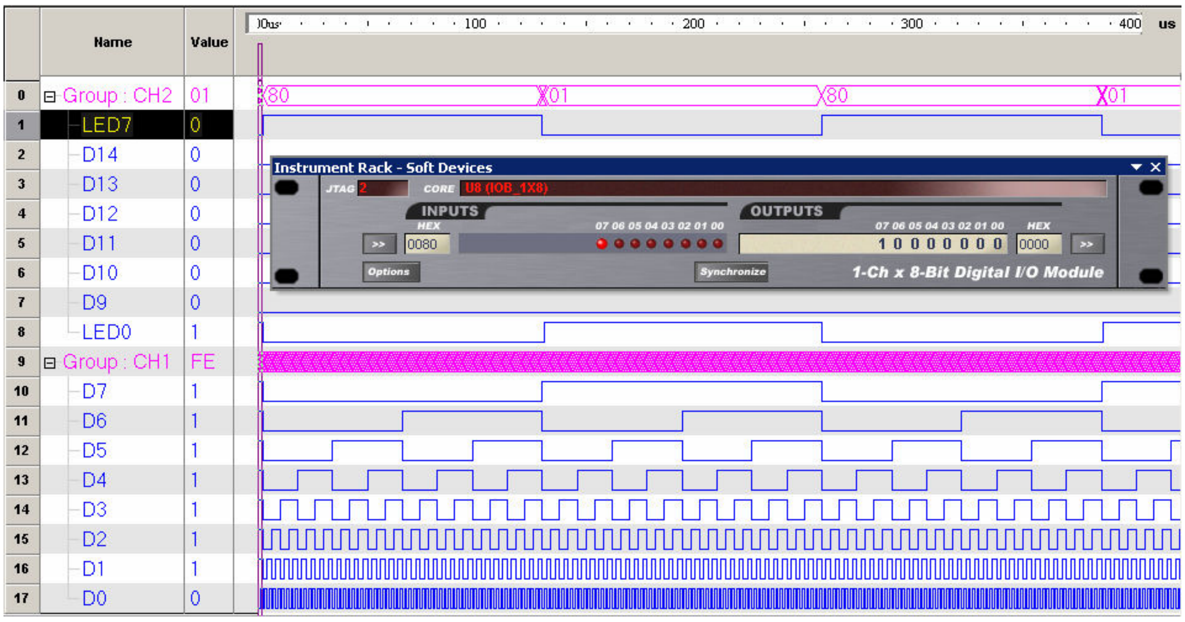


Figure 62. Logic analyser waveform with bit-7 of the IOB set.

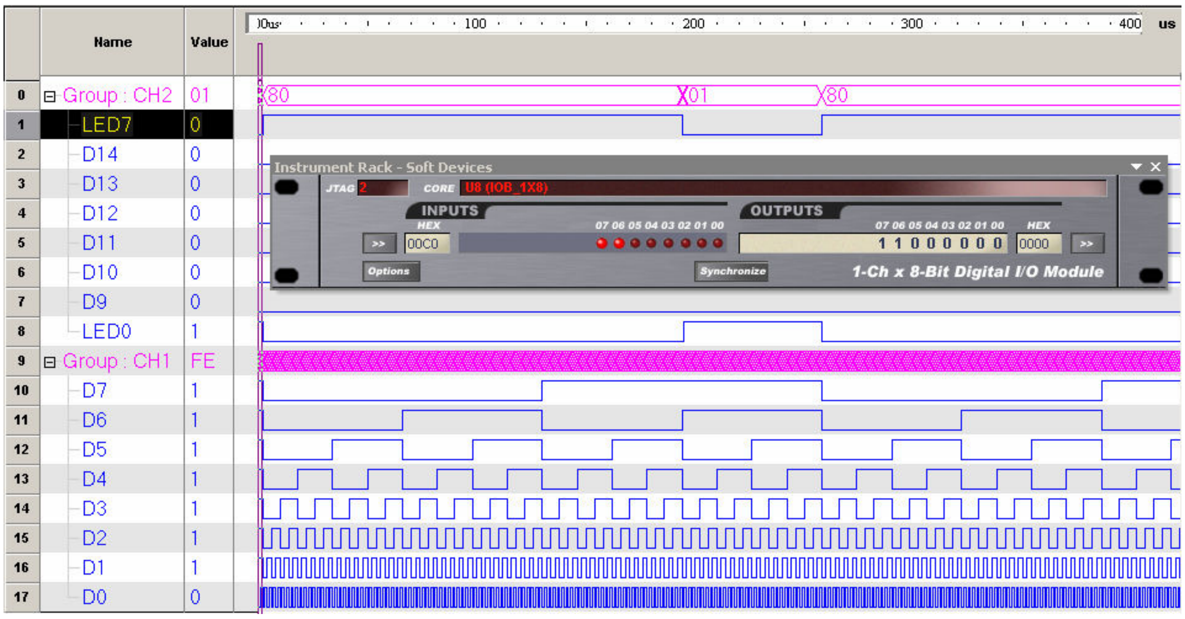
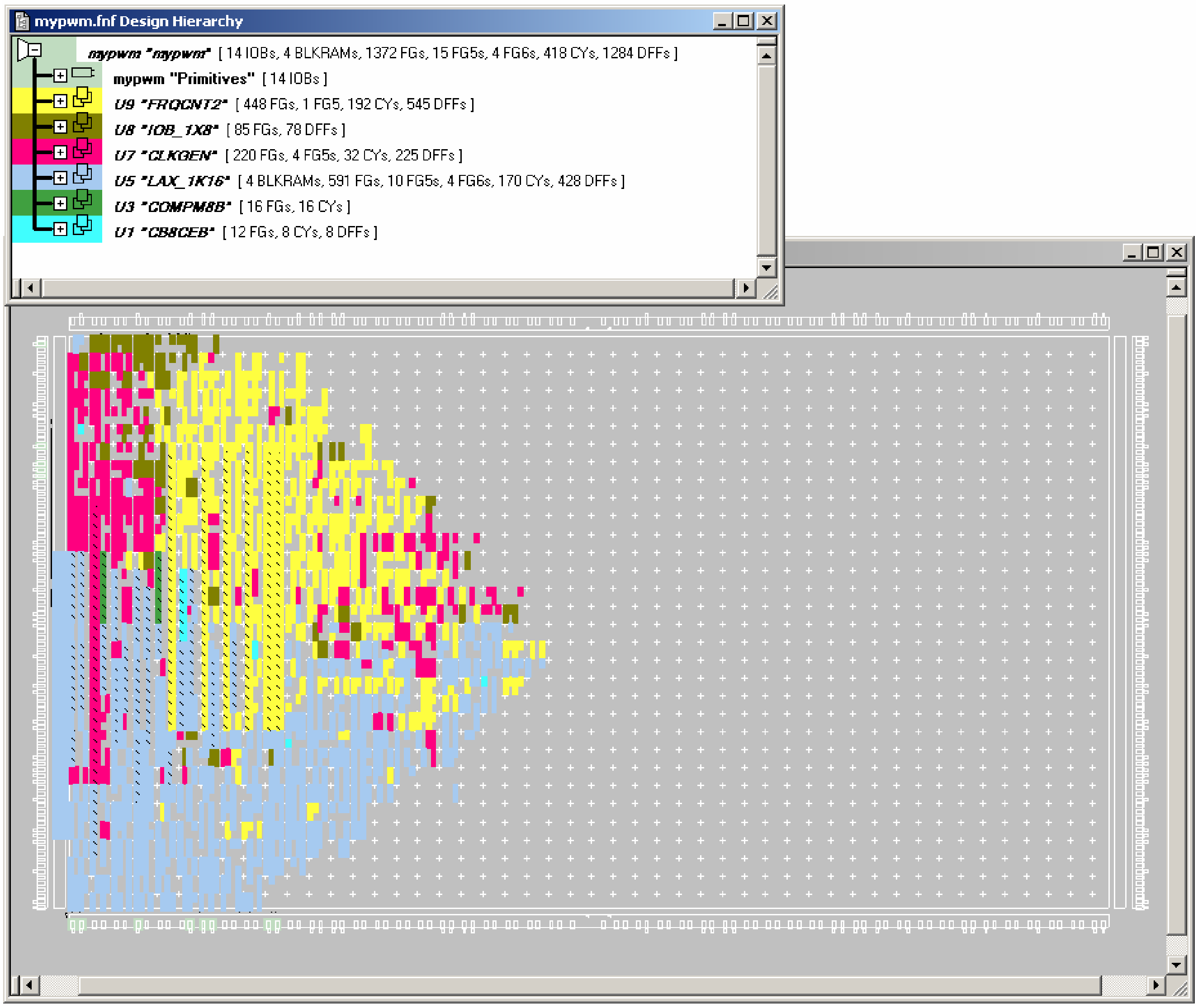


Figure 63. Logic analyser waveform with bits 6 & 7 of the IOB set.

8.8 Where are the Instruments?

The important differentiator between Altium Designer's embedded instruments and other simulation-based virtual instruments is that Altium Designer's embedded instruments are true physical devices that are downloaded into the FPGA device as part of the design. The information provided to the designer by the embedded instruments can be relied upon as it is taken from real physical measurements taken on chip.

Figure 64 illustrates this point as it shows the FPGA real estate used by the embedded instruments.



```
mypwm "mypwm" [ 14 IOBs, 4 BLKRAMs, 1372 FGs, 15 FG5s, 4 FG6s, 418 CYs, 1284 DFFs ]
  mypwm "Primitives" [ 14 IOBs ]
  U9 "FRQCNT2" [ 448 FGs, 1 FG5, 192 CYs, 545 DFFs ]
  U8 "IOB_1X8" [ 85 FGs, 78 DFFs ]
  U7 "CLKGEN" [ 220 FGs, 4 FG5s, 32 CYs, 225 DFFs ]
  U5 "LAX_1K16" [ 4 BLKRAMs, 591 FGs, 10 FG5s, 4 FG6s, 170 CYs, 428 DFFs ]
  U3 "COMPM8B" [ 16 FGs, 16 CYs ]
  U1 "CB8CEB" [ 12 FGs, 8 CYs, 8 DFFs ]
```

Figure 64. Floorplan of MyPWM_withInstruments.SchDoc after it has been placed onto an FPGA.

8.9 Enabling embedded instruments

The NanoBoard hardware incorporates the entire infrastructure necessary to support Embedded Instruments and allow them to communicate with the host PC. All virtual instruments communicate with the host PC via a 'soft' JTAG chain that conforms to the Nexus standard. To enable Nexus on the NanoBoard, the `NEXUS_JTAG_PORT` and `NEXUS_JTAG_ CONNECTOR` must be placed onto the top level schematic. These respective components can be found in the `FPGA Generic.IntLib` and `FPGA NanoBoard Port-Plugin.IntLib` Integrated Libraries.

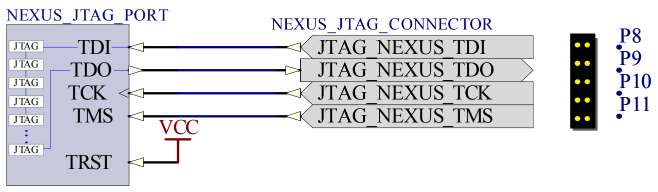


Figure 65. NEXUS JTAG Port and NEXUS JTAG Connector.

To be able to use embedded instruments in custom designs, it is necessary to reserve 4 device pins for the `NEXUS_JTAG_CONNECTOR` and ensure that sufficient device resources are present to accommodate the virtual instruments in the device. The JTAG soft chain and other communications chains present on the NanoBoard will be discussed further in the next section.

9 Interacting with the NanoBoard

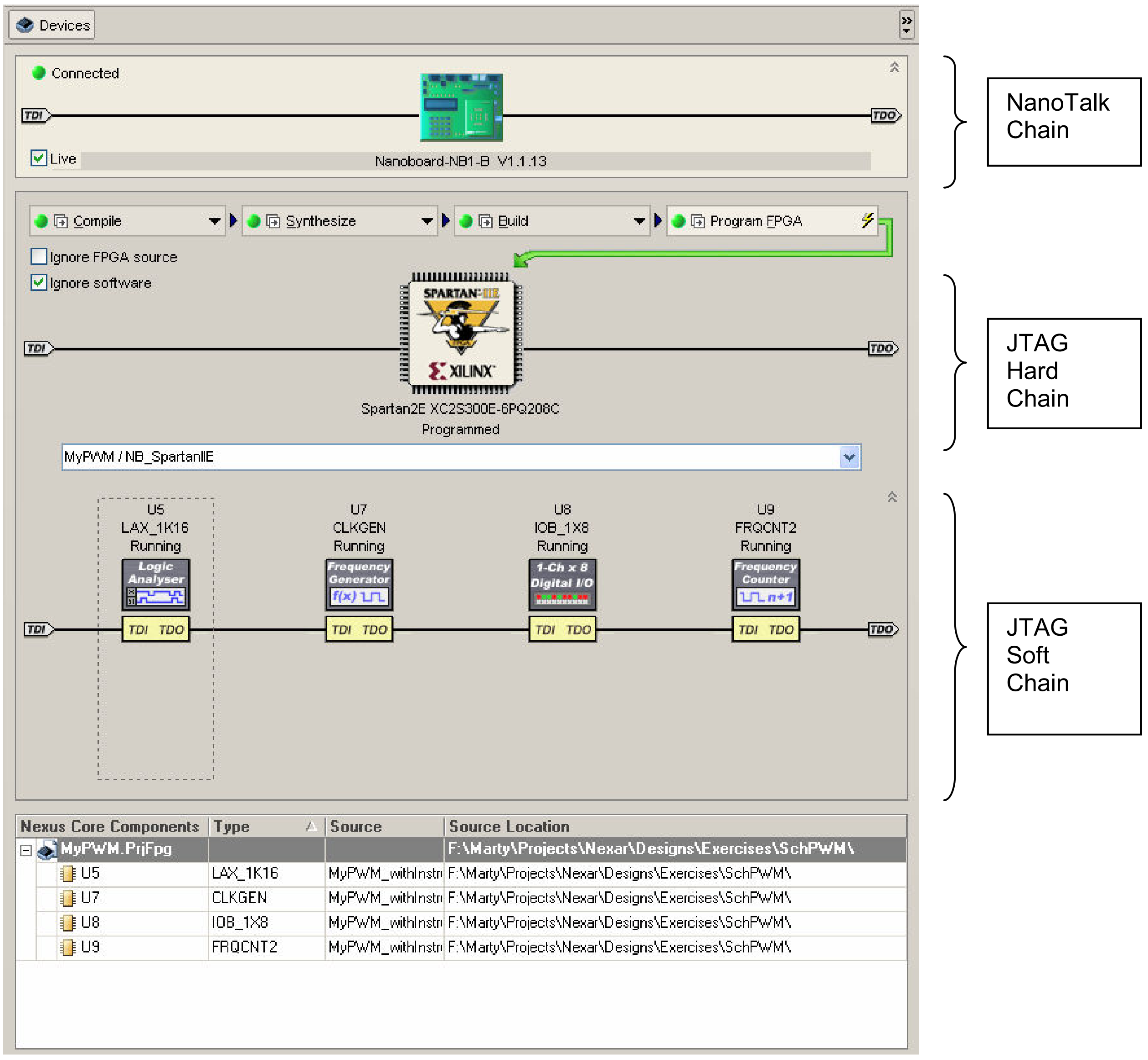
9.1 Overview

The NanoBoard is pivotal to rapid embedded systems development with Altium Designer. It contains a range of peripherals and expansion capabilities to allow it to adapt to a broad cross section of embedded projects. In this section we will discuss the concepts necessary for a designer to make effective use the NanoBoard's potential.

9.2 NanoBoard communications

The NanoBoard contains 3 primary communication channels. A complete understanding of these channels is not necessary to begin using the tool suite however it may be of interest to developers keen to make use of Altium Designer's debugging capabilities on their own custom designs.

The primary point of user control of the NanoBoard is via the **Devices View**. This view provides an easy-to-use visualization of the various communications chains active on the NanoBoard.



Nexus Core Components	Type	Source	Source Location
MyPWM.PrjFpg			F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\
U5	LAX_1K16	MyPWM_withInstr	F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\
U7	CLKGEN	MyPWM_withInstr	F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\
U8	IOB_1X8	MyPWM_withInstr	F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\
U9	FRQCNT2	MyPWM_withInstr	F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\

Figure 66. Devices view with its various communications channels highlighted.

9.2.1 NanoTalk chain

NanoTalk is the proprietary communications protocol developed by Altium to enable multiple NanoBoards to communicate with one another. The 10 pin NanoTalk headers can be found on both the left and right edges at the upper end of the NanoBoard. Communications via this channel is totally transparent to the user. There should be no need to interact with this standard.

9.2.2 JTAG Hard Chain

The JTAG Hard Chain is a serial communications channel that connects physical devices together. JTAG devices can be connected end on end by connecting the TDO pin of an upstream device to the TDI pin of a downstream device. The hard JTAG chain is visible in the middle portion of the **Devices View**. Usually this is where an FPGA will be located however if you also have other devices that are connected to the JTAG chain such as a configuration device then these will be visible also.

The hard JTAG chain can be extended beyond the NanoBoard through the **User Board A** and **User Board B** connectors. When using either of these connectors, it is imperative that the JTAG chain is not broken – i.e. the TDI/TDO chain must be looped back to the NanoBoard.

9.2.3 JTAG Soft Chain

The JTAG Soft Chain is a separate JTAG channel that provides communication with the Embedded Instruments that can be incorporated into an FPGA design. This chain is labeled as a *soft* chain since it does not connect tangible physical devices together but rather connects soft or downloadable instruments that reside *inside* a hard or physical FPGA device.

9.3 Technical background

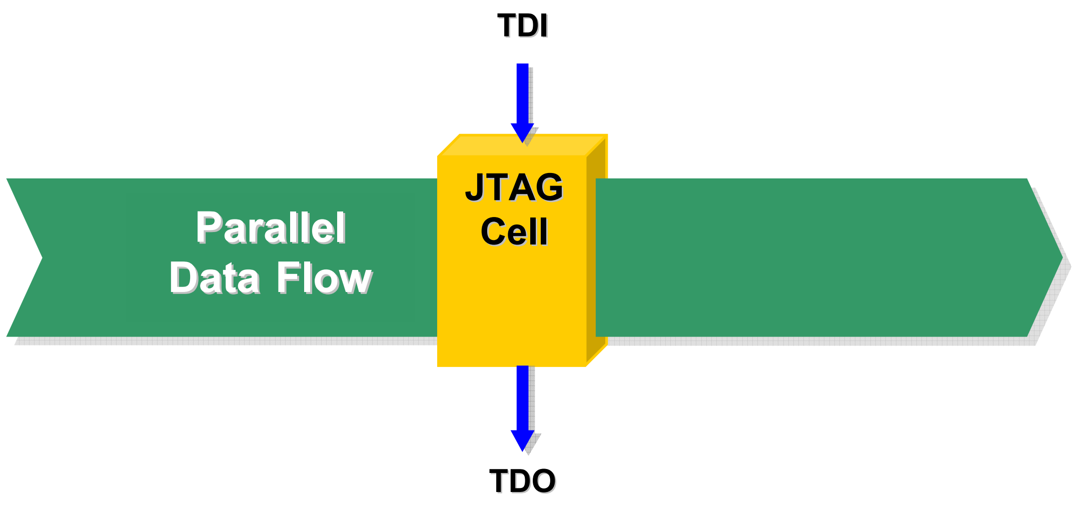


Figure 67. Conceptual View of JTAG data flows.

9.3.1 JTAG in depth

The acronym JTAG stands for <u>J</u>oint <u>T</u>est <u>A</u>pplication <u>G</u>roup and is synonymous with IEEE 1149.1. The standard defines a Test Access Port (TAP), boundary scan architecture and communications protocol that allows automated test equipment to interact with hardware devices. Essentially it enables you to place a device into a test mode and then control the state of each of the device's pins or run a built-in self-test on that device. The flexibility of the JTAG standard has also lead to its usage in programming (configuring) devices such as FPGAs and microprocessors.

At minimum, JTAG requires that the following pins are defined on a JTAG device:

TCK: Test Clock Input

TMS: Test Mode Select

- TDI: Test Data Input
- TDO: Test Data Output

TCK controls the data rate of data being clocked into and out of the device. A rising TCK edge is used by the device to sample incoming data on its TDI pin and by the host to sample outgoing data on the devices TDO pin.

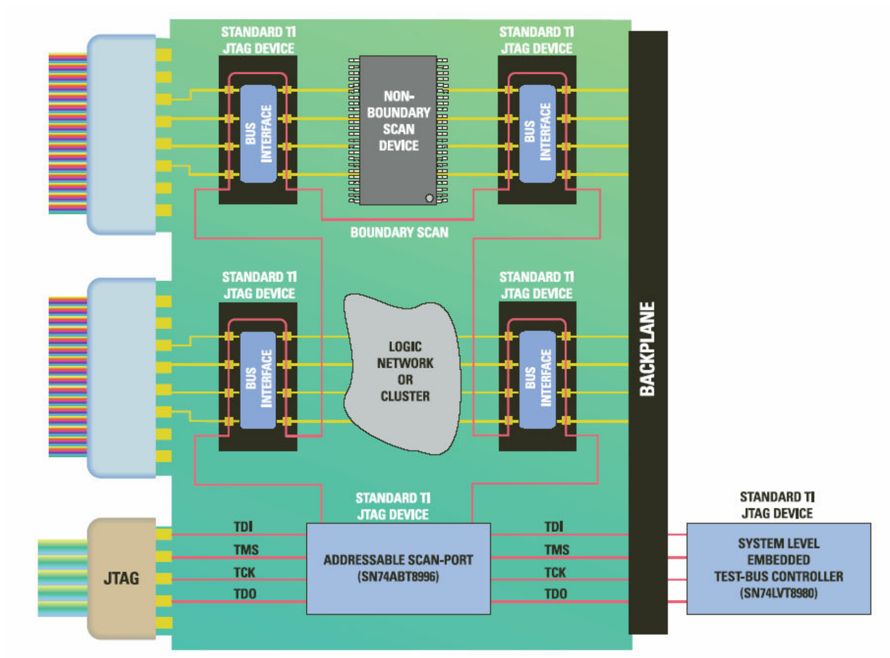


Figure 68. Using JTAG Chain to connect multiple JTAG devices together in a digital design.

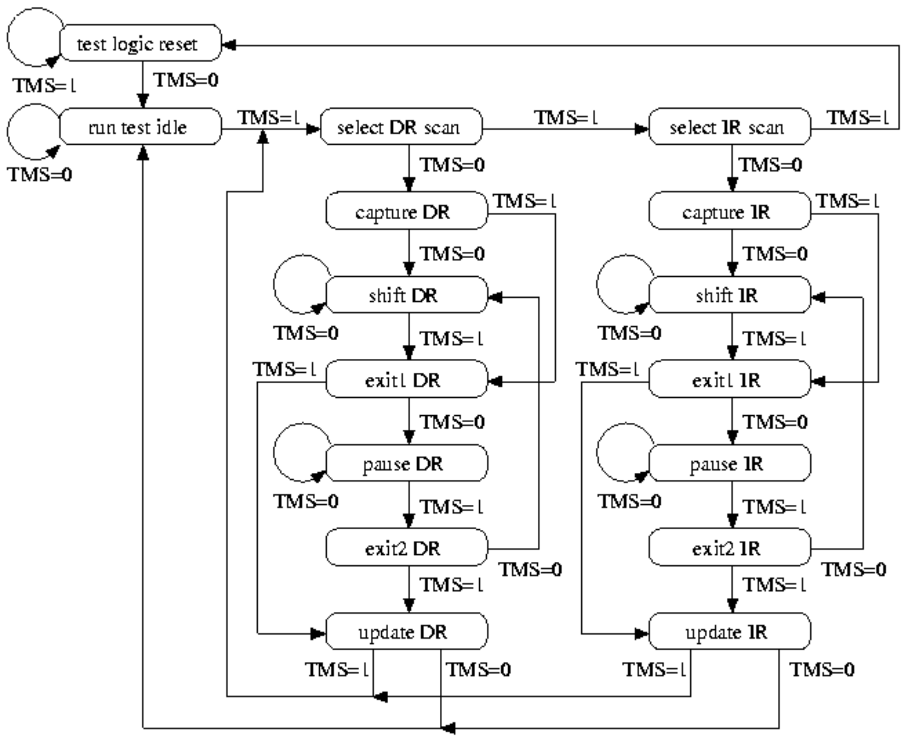


Figure 69. JTAG Test Access Port (TAP) State Machine.

The Test Access Port (TAP) Controller is a state machine that controls access to two internal registers – the Instruction Register (IR) and the Data Register (DR). Data fed into the device via TDI or out of the device via TDO can only ever access one of these two registers at any given time. The register being accessed is determined by which state the TAP controller is in. Traversal through the TAP controller state machine is governed by TMS.

9.3.2 Nexus 5001

The flexibility of JTAG for hardware debugging purposes has flowed over into the software domain. In the same way that test engineers have sought a standardized method for testing silicon, software engineers have also sought a standardized means for debugging their programs.

In 1998, the Global Embedded Debug Interface Standard (GEDIS) Consortium was formed. In late 1999 the group moved operations into the IEEE-ISTO and changed their name to the Nexus 5001 Forum and released V1.0 of IEEE-ISTO – 1999. In December 2003, V2.0 was released.

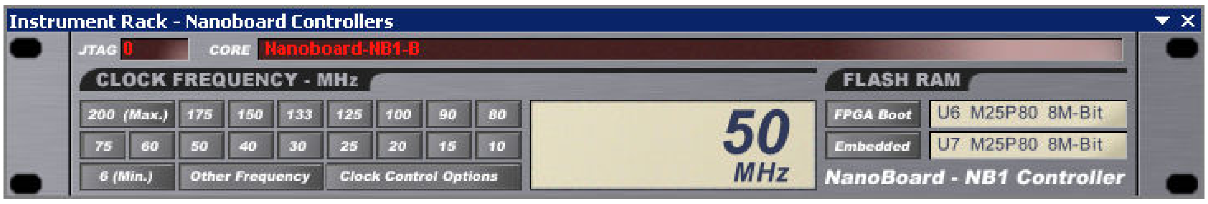
The Nexus 5001 standard provides a standardized mechanism for debug tools to interact with target systems and perform typical debugging operations such as setting breakpoints and analyzing variables, etc. There are 4 classes of Nexus compliance – each with differing levels of supported functionality. The lowest level uses JTAG as the low-level communications conduit.

The implementation of Nexus 5001 on the NanoBoard has been labeled as the JTAG Soft Chain. It is a serial chain just like the hard chain however rather than connecting physical devices together, it connects virtual devices together. These devices include the set of virtual instruments that are supplied with Altium Designer and described in the following chapter. Control of devices on the Soft Chain can be performed from the **Devices View** – Soft Chain Devices are located towards the bottom of the **Devices View** under the Hard Chain.

As with the JTAG Hard Chain, the Soft Chain can be taken off the NanoBoard via the **User Board A** and **User Board B** connectors. This provides the means for target systems to also include virtual instruments and to benefit from the Altium Designer development environment. Similarly to the Hard Chain, it is imperative that a complete loop be maintained between the Soft Chain TDI and TDO connections.

9.4 The NanoBoard controller

The NanoBoard Controller can be accessed by double-clicking on the NanoBoard icon in the **Devices View**.


Instrument Rack - Nanoboard Controllers

JTAG 0 CORE Nanoboard-NB1-B

CLOCK FREQUENCY - MHz

200 (Max.)	175	150	133	125	100	90	80
75	60	50	40	30	25	20	15	10
6 (Min.)	Other Frequency	Clock Control Options					

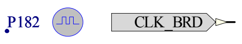
50 MHz

FLASH RAM

FPGA Boot	U6 M25P80 8M-Bit
Embedded	U7 M25P80 8M-Bit

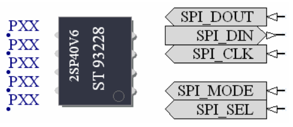
NanoBoard - NB1 Controller

Figure 70. The NanoBoard Controller Instrument Rack.



The Clock Frequency indicated in the window will be supplied to the `CLK_BRD` port on the NanoBoard. Accessing this clock on custom designs is as simple as placing the CLOCK_BOARD component from the `FPGA NanoBoard Port-Plugin.IntLib` Library.

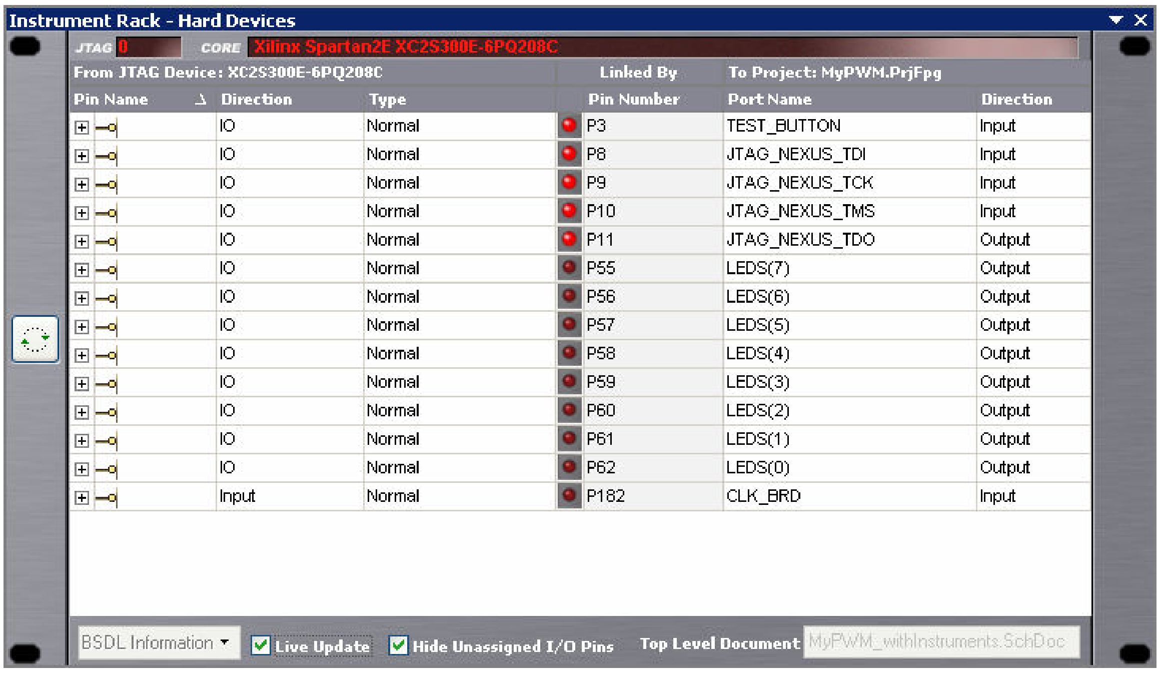
Selecting a non-standard frequency is possible by clicking the **Other Frequency** button. The NanoBoard clock system employs a serially programmable clock source (part number ICS307-02) that is capable of synthesizing any clock frequency between 6 and 200MHz. Advanced access to the Clock Control IC registers is available through the **Clock Control Options** button. A datasheet for this device is available from the ICS website http://www.icst.com/products/pdf/ics3070102.pdf. An online form useful for calculating settings for the clock control IC is also available at http://www.icst.com/products/ics307inputForm.html.



To the right of the NanoBoard Controller is a section with the heading Flash RAM. The **FPGA Boot** button affords the facility to store a daughter board configuration file that will get automatically loaded into the daughter board on power up. The **Embedded** button exposes memory that can be used by the user application to store non-volatle user data. The Embedded Memory device is accessible via the SERIALFMEMORY component in the `FPGA NanoBoard Port-Plugin.IntLib` Library.

9.5 FPGA I/O view

Double-clicking a device icon in the JTAG Hard chain displays the Instrument Rack for that device.


Instrument Rack - Hard Devices
JTAG 0
CORE Xilinx Spartan2E XC2S300E-6PQ208C
BSDL Information
Live Update
Hide Unassigned I/O Pins
Top Level Document MyPWM_withInstruments.SchDoc

From JTAG Device: XC2S300E-6PQ208C — Linked By — To Project: MyPWM.PrjFpg

Pin Name	Direction	Type	Pin Number	Port Name	Direction
	IO	Normal	P3	TEST_BUTTON	Input
	IO	Normal	P8	JTAG_NEXUS_TDI	Input
	IO	Normal	P9	JTAG_NEXUS_TCK	Input
	IO	Normal	P10	JTAG_NEXUS_TMS	Input
	IO	Normal	P11	JTAG_NEXUS_TDO	Output
	IO	Normal	P55	LEDS(7)	Output
	IO	Normal	P56	LEDS(6)	Output
	IO	Normal	P57	LEDS(5)	Output
	IO	Normal	P58	LEDS(4)	Output
	IO	Normal	P59	LEDS(3)	Output
	IO	Normal	P60	LEDS(2)	Output
	IO	Normal	P61	LEDS(1)	Output
	IO	Normal	P62	LEDS(0)	Output
	Input	Normal	P182	CLK_BRD	Input

Figure 71. The FPGA I/O Instrument Rack.

This interface enables the developer to see in real time the flow of signals across the device's pins. This can be particularly useful when ensuring that signals are being correctly propagated to and from the device.

Placing a tick in the **Live Update** checkbox will cause the display to update in real time. Alternatively, leaving the **Live Update** checkbox clear and selecting the update icon will cause signal information to be latched to the display and held.

Check **Hide Unassigned I/O Pins** to remove clutter from the display.

The **BSDL Information** drop down list should only need to be accessed for devices which are unknown to Altium Designer. In this case, you will need to provide the location of the vendor supplied BSDL file for the device you are viewing.

The FPGA IO instrument rack is available for all devices on the JTAG Hard Chain – including devices on a user board that is connected to the JTAG Hard Chain.

9.6 Live cross probing

Probe directives can be placed on the FPGA schematic on any I/O net and will update in real time as long as the Hard Devices Instrument Panel is displayed. Use the **Place » Directives » Probe** to place a cross probe on one of the I/O nets.

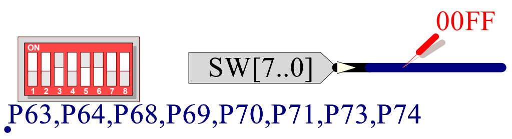


Figure 72. Using Live Cross Probing.

9.7 Exercise 5 – View MyPWM on the NanoBoard

1. Reload your circuit from Exercise 3 again and run it on the NanoBoard.
2. Open the FPGA IO Instrument Rack.
3. Check the Hide Unassigned I/O Pins checkbox and the Live Update checkboxes.
4. Observe the change in switch states and LEDs as you toggle the NanoBoard DIP switches.
5. Use the **Place » Directives » Probe** option to place a probe point on the bus connected to the DIP Switches. Observe the probe value as the DIP Switches are changed on the NanoBoard.

10 Creating a core component

10.1 Core project

Altium Designer provides the ability to encapsulate an entire FPGA circuit into a single component that can be used as a building block in other projects. These self-contained blocks are called core components and offer the advantage of design reuse and design security. Core components can be synthesized for a target FPGA and made available to others without exposing the underlying IP.

A Core project is used to create an FPGA component that may be used multiple times within one or across many FPGA projects. The output of a Core project behaves in a similar fashion to a library component in that it becomes an elemental unit that is used as a component in larger designs.

A Core project is useful when you wish to make some functionality available to a broad user base but you do not want to expose the IP used to implement the functionality.

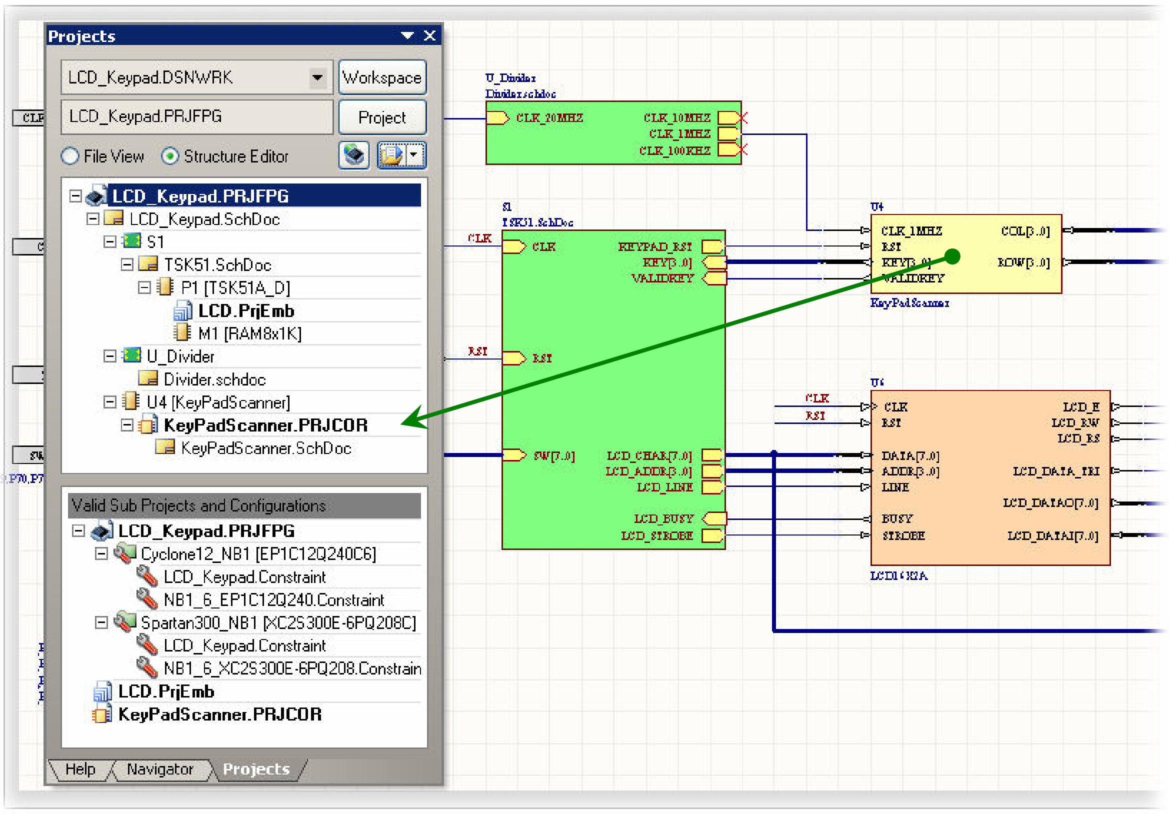


Figure 73. Using a Core Component in an FPGA Project.

10.2 Creating a core component from an FPGA project.

It is possible to create a core component from scratch however often we wish to create a core component from an existing FPGA design or project. In either case a blank core project must first be created. If the core component is to be based on an existing design then use **Project » Add Existing to Project** to add the relevant VHDL and / or schematic documents to the project. If the core component is being created from scratch then its source documents will need to be created in the same way that an FPGA project is built.

10.3 A word about EDIF

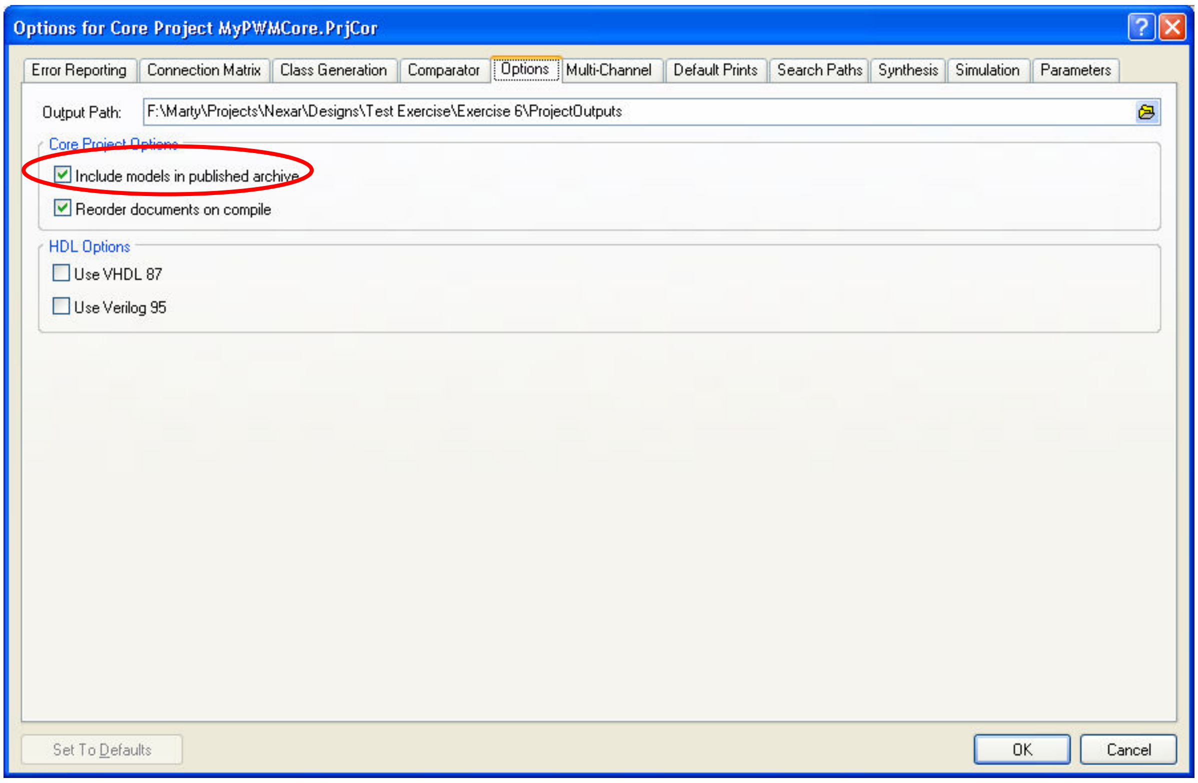
EDIF is an acronym for Electronic Design Interchange Format. It was originally developed as a standardized format for transferring integrated circuit design information between vendor tools. Altium Designer creates EDIF files as part of the synthesis process and these files are then passed to the vendor back end tools for complete FPGA place and route.

Although EDIF files conform to a standard, the information within a given EDIF file may contain vendor specific constructs. EDIF files can not, therefore be considered as vendor independent.

It is also worth noting that although EDIF files do offer some form of IP protection, they are readable by humans and can be deciphered with little effort. They should not be relied upon to maintain IP protection.

10.4 Setting up the core project

Once the core project has been created it is important to make available its EDIF models when you eventually 'publish' it. Make sure the **Include models in published archive** checkbox is ticked in the **Options** tab of the **Project Options** dialog.



Options for Core Project MyPWMCore.PrjCor

Output Path: F:\Marty\Projects\Nexar\Designs\Test Exercise\Exercise 6\ProjectOutputs

Core Project Options

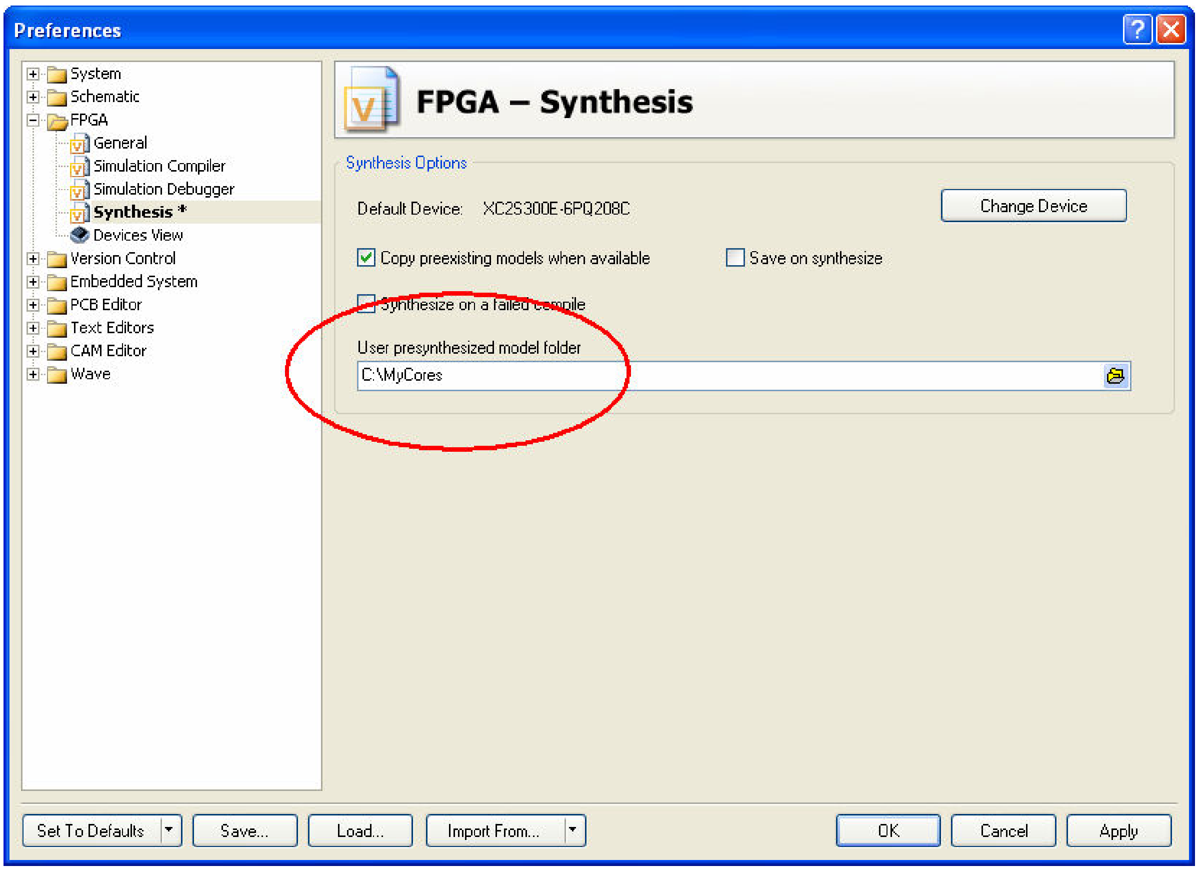
- ☑ Include models in published archive
- ☑ Reorder documents on compile

HDL Options

- ☐ Use VHDL 87
- ☐ Use Verilog 95

Figure 74. Setting options for a core component.

You must now specify the folder on your hard disk that you wish the EDIF models to be saved into. This folder will be searched along with the standard system EDIF folders (\Altium Designer 6\Library\EDIF) when you synthesize any design. It is good practice to keep EDIF models generated from core projects in a single location for easier searching. The location of the EDIF model folder is specified in the **Preferences – FPGA – Synthesis** dialog.



FPGA – Synthesis

Synthesis Options

Default Device: XC2S300E-6PQ208C

☑ Copy preexisting models when available ☐ Save on synthesize

☐ Synthesize on a failed compile

User presynthesized model folder

C:\MyCores

Figure 75. Specifying the location of core component models.

10.5 Constrain / configure

The concept of constraint files and configurations is central to the flexibility of Altium Designer. They provide a mechanism to allow FPGA circuits to be developed independent of the final physical implementation. Rather than storing device and implementation specific data such as pin allocations and electrical properties in the source VHDL or schematic documents, this information is stored in separate files – called *Constraint files*. This decoupling of the logical definition of an FPGA design from its physical implementation allows for quick and easy re-targeting of a single design to multiple devices and PCB layouts.

There are a number of classes of configuration information pertinent to different aspects of an FPGA project:

10.5.1 Device and board considerations:

The specific FPGA device must be identified and ports defined in the top level FPGA design must be mapped to specific pin numbers.

10.5.2 Device resource considerations:

In some designs it may be advantageous to make use of vendor specific resources that are unique to a given FPGA device. Some examples are hardware multiplication units, clock multipliers and memory resources.

10.5.3 Project or design considerations:

This would include requirements which are associated with the logic of the design, as well as constrains on its timing. For example, specifying that a particular logical port must be allocated to global clock net, and must be able to run at a certain speed.

A *configuration* is a set of one or more constraint files that must be used to target a design for a specific output. The migration of a design from prototype, refinement and production will often involve several PCB iterations and possibly even different devices. In this case, a separate configuration would be used to bring together constraint file information for each design iteration. Each new configuration (and its associated constraint file(s)) is stored with the project and can be recalled at any time.

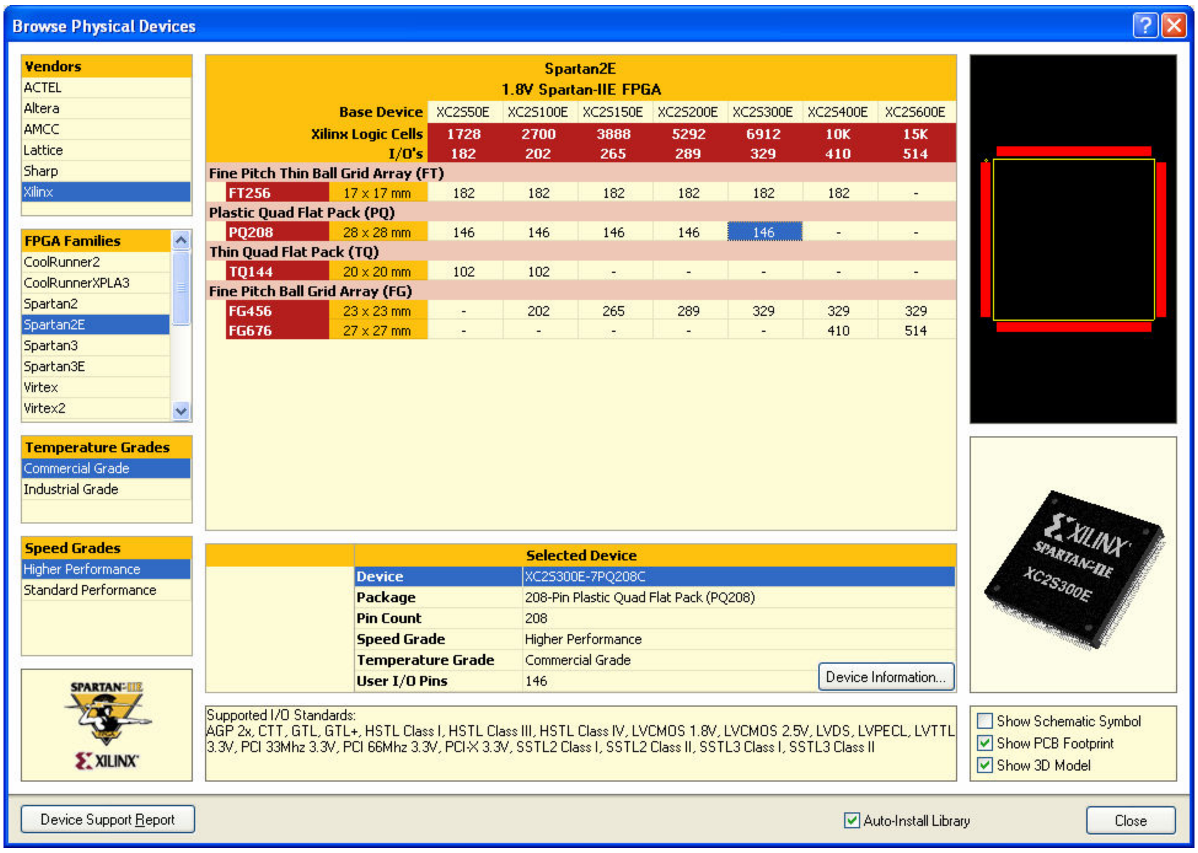
To summarize:

- **Constraint files** store implementation specific information such as device pin allocations and electrical properties.
- A **Configuration** is a grouping of one or more constraint files and describes how the FPGA project should be built.

10.6 Creating a new constraint file.

Before a configuration can be built, a constraint file must exist. Constraint files have the extension `.Constraint`. Constraint files for use with the NanoBoard daughter-board modules can be found in the `\Program Files\Altium Designer 6\Library\Fpga` directory. In general it is advisable to take a copy of these files and store it in your project directory before adding it to the project. This way the project is kept self-contained and any edits you may inadvertently make will not affect the supplied constraint file.

- To add your own, new constraint file, right click on the project name in the **Projects** panel and select **Add New To Project » Constraint File**.
- A new blank constraint file will appear. To specify the target device select **Design » Add/Modify Constraint » Part** and the *Browse Physical Devices* dialog will open.



Vendors
ACTEL
Altera
AMCC
Lattice
Sharp
Xilinx

FPGA Families
CoolRunner2
CoolRunnerXPLA3
Spartan2
Spartan2E
Spartan3
Spartan3E
Virtex
Virtex2

Temperature Grades
Commercial Grade
Industrial Grade

Speed Grades
Higher Performance
Standard Performance

Spartan2E
1.8V Spartan-IIE FPGA

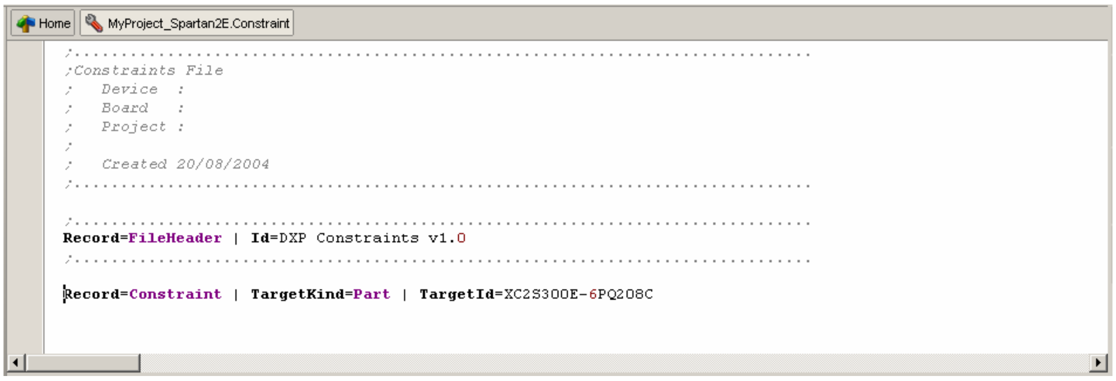
	Base Device	XC2S50E	XC2S100E	XC2S150E	XC2S200E	XC2S300E	XC2S400E	XC2S600E
	Xilinx Logic Cells	1728	2700	3888	5292	6912	10K	15K
	I/O's	182	202	265	289	329	410	514
Fine Pitch Thin Ball Grid Array (FT)								
FT256	17 x 17 mm	182	182	182	182	182	182	-
Plastic Quad Flat Pack (PQ)								
PQ208	28 x 28 mm	146	146	146	146	146	-	-
Thin Quad Flat Pack (TQ)								
TQ144	20 x 20 mm	102	102	-	-	-	-	-
Fine Pitch Ball Grid Array (FG)								
FG456	23 x 23 mm	-	202	265	289	329	329	329
FG676	27 x 27 mm	-	-	-	-	-	410	514

Selected Device	
Device	XC2S300E-7PQ208C
Package	208-Pin Plastic Quad Flat Pack (PQ208)
Pin Count	208
Speed Grade	Higher Performance
Temperature Grade	Commercial Grade
User I/O Pins	146

Supported I/O Standards:
AGP 2x, CTT, GTL, GTL+, HSTL Class I, HSTL Class III, HSTL Class IV, LVCMOS 1.8V, LVCMOS 2.5V, LVDS, LVPECL, LVTTL 3.3V, PCI 33Mhz 3.3V, PCI 66Mhz 3.3V, PCI-X 3.3V, SSTL2 Class I, SSTL2 Class II, SSTL3 Class I, SSTL3 Class II

☐ Show Schematic Symbol
☑ Show PCB Footprint
☑ Show 3D Model
☑ Auto-Install Library

Figure 76. The Browse Physical Devices dialog, where you select the target FPGA.



```
;...............................................................................
;Constraints File
;   Device  :
;   Board   :
;   Project :
;
;   Created 20/08/2004
;...............................................................................

;...............................................................................
Record=FileHeader | Id=DXP Constraints v1.0
;...............................................................................

Record=Constraint | TargetKind=Part | TargetId=XC2S300E-6PQ208C
```

Figure 77. Basic constraint file.

- Select the vendor, family, device and temperature/speed grades as desired and click OK. A line similar to the one above will be automatically inserted into the constraint file:
- Save the constraint file. Typically it would be named to reflect its role – for example if the target device was a Xilinx Spartan-3 FPGA mounted on your project PCB you might call it `MyProject_Spartan2E.Constraint`. You will notice the constraint file has been added to the project under the settings tab.

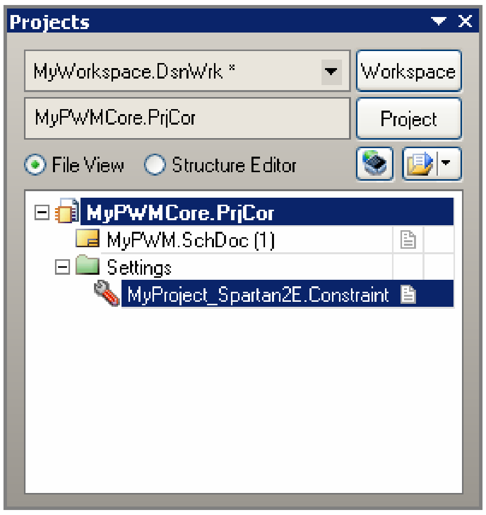


Figure 78. Project with constraint File.

10.7 Creating a configuration

As previously mentioned, configurations group a number of constraint files together to create a set of instructions for the FPGA build process. To define a configuration …

- Right-click on the project name in the **Projects** panel and select **Configuration Manager**

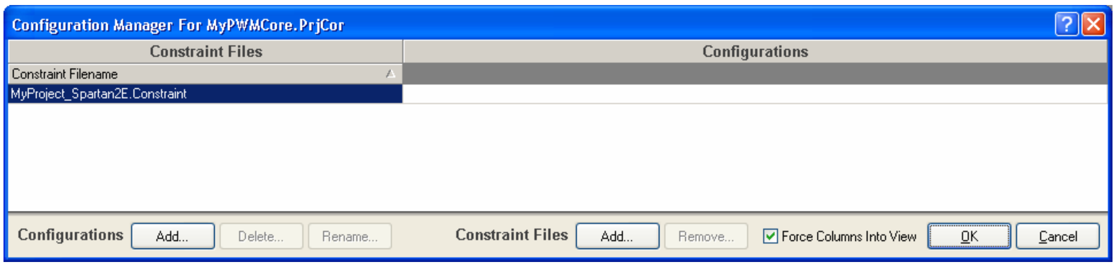
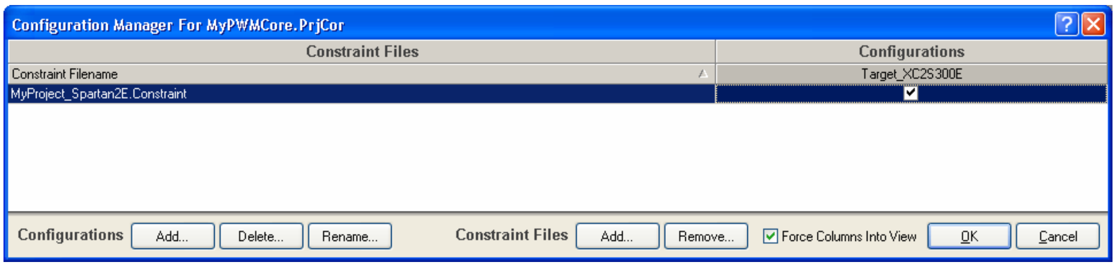


Figure 79. Specifying a configuration using the configuration manager.

- If you have just recently created a new constraints file, you will see it listed under the Constraint Filename. Existing constraint files that currently aren't in the project can be added by selecting the **Add** button next to the **Constraint Files** text.
- To define a new configuration, select the **Add** button next to the **Configurations** text. A dialog will appear requesting you to provide a name for the new configuration. The name can be arbitrary but it is helpful if it provides some indication as to what the configuration is for.
- Having defined the new configuration, you may now assign constraint files to it by ticking their associated checkbox. Here we have assigned the constraint file **MyProject_Spartan2E** to the **Target_XC2S300E** configuration.

- Click OK to save the configuration.

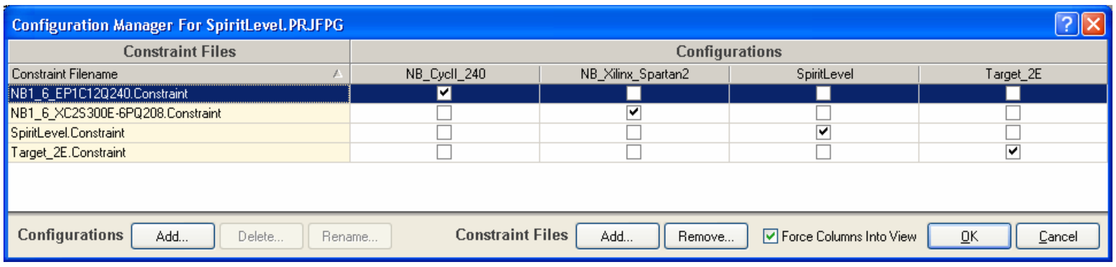


Configuration Manager For MyPWMCore.PrjCor

Constraint Files	Configurations
Constraint Filename	Target_XC2S300E
MyProject_Spartan2E.Constraint	✓

Figure 80. Specifying a configuration using the configuration manager.

Although the above simplistic example only had one constraints file and one configuration, the power of configurations really becomes apparent as the design matures. Below we see how a design has been targeted to multiple platforms:



Configuration Manager For SpiritLevel.PRJFPG

Constraint Files	Configurations			
Constraint Filename	NB_CycII_240	NB_Xilinx_Spartan2	SpiritLevel	Target_2E
NB1_6_EP1C12Q240.Constraint	✓	☐	☐	☐
NB1_6_XC2S300E-6PQ208.Constraint	☐	✓	☐	☐
SpiritLevel.Constraint	☐	☐	✓	☐
Target_2E.Constraint	☐	☐	☐	✓

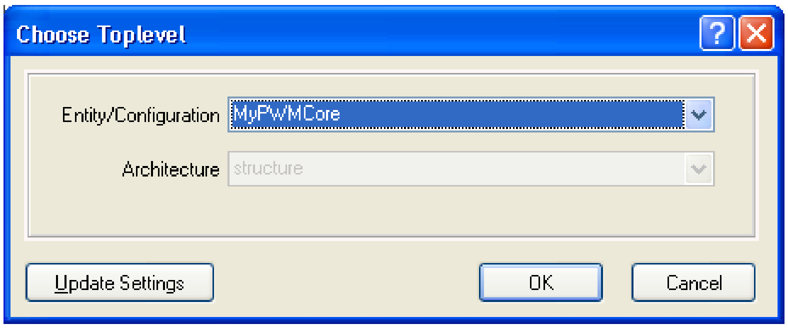
Figure 81. Example of a project with multiple configurations defined.

Configurations can be updated or modified as desired at any time throughout the project's development by returning to the *Configuration Manager* dialog.

10.8 Synthesize

Now that we have defined a configuration we are ready to synthesize the core for the target.

- With the top level FPGA document open select **Design » Synthesize**. If we had defined more than one configuration and wished to synthesize all configurations at once we could select **Design » Synthesize All Configurations.**



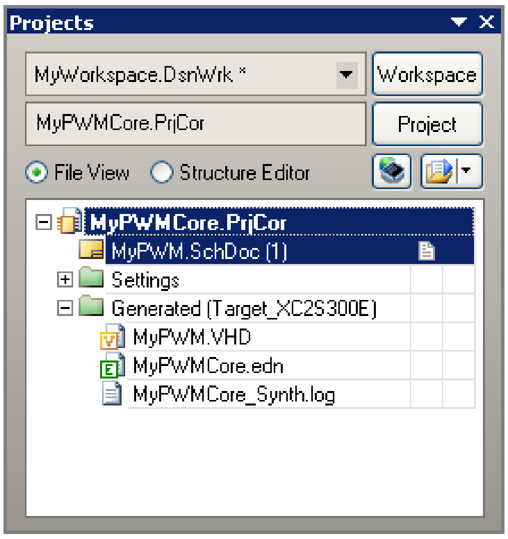
Choose Toplevel

Entity/Configuration: MyPWMCore

Architecture: structure

Figure 82. Specifying an FPGA project's top level document.

- If you have not already nominated the top level entity/configuration in the **Synthesis** tab of the **Options for Core Project**, the **Choose Toplevel** dialog will appear. Enter the core project name or select from the dropdown list and click **OK** to continue.
- The project will be synthesized resulting in the generation of VHDL files for the schematic, EDIF files for the schematic wiring and parts, and a synthesis log file. These will all be located under the *Generated* folder in the project panel.



```
Projects
MyWorkspace.DsnWrk *
MyPWMCore.PrjCor
MyPWMCore.PrjCor
  MyPWM.SchDoc (1)
  Settings
  Generated (Target_XC2S300E)
    MyPWM.VHD
    MyPWMCore.edn
    MyPWMCore_Synth.log
```

Figure 83. Files generated after synthesizing the design

- You will observe the configuration name in brackets beside the *Generated* Folder. Had we synthesized more than one configuration then a separate *Generated* folder would have appeared for each of the defined configurations.
- Confirm that the synthesis process completed successfully by observing the synthesis log file. A line towards the bottom of the report should indicate whether any errors were encountered.

10.9 Publish

Now we can *publish* the core project. This will zip together (archive) all the EDIF files in the core project's Project Outputs folder and then copy this to the user EDIF models folder that was specified earlier.

- Select **Design » Publish**. If the error message "cannot find 'working folder'" appears, make sure you have set up the **Use presynthesized model folder** option in the **FPGA Preferences** dialog.
- Check the **Messages** panel to ensure the design has been successfully published.
- Save the core project file.

10.10 Creating a core schematic symbol

The core project has been successfully synthesized and published. It would be possible at this point for other personnel to make use of your core through a VHDL instantiation process. This can be a messy affair. A far simpler option would be for them to use a schematic symbol that is linked to your core and associated EDIF files. To do this, we need to create our own schematic symbol from the core component.

- With the top level FPGA document open select **Design » Generate Symbol**.

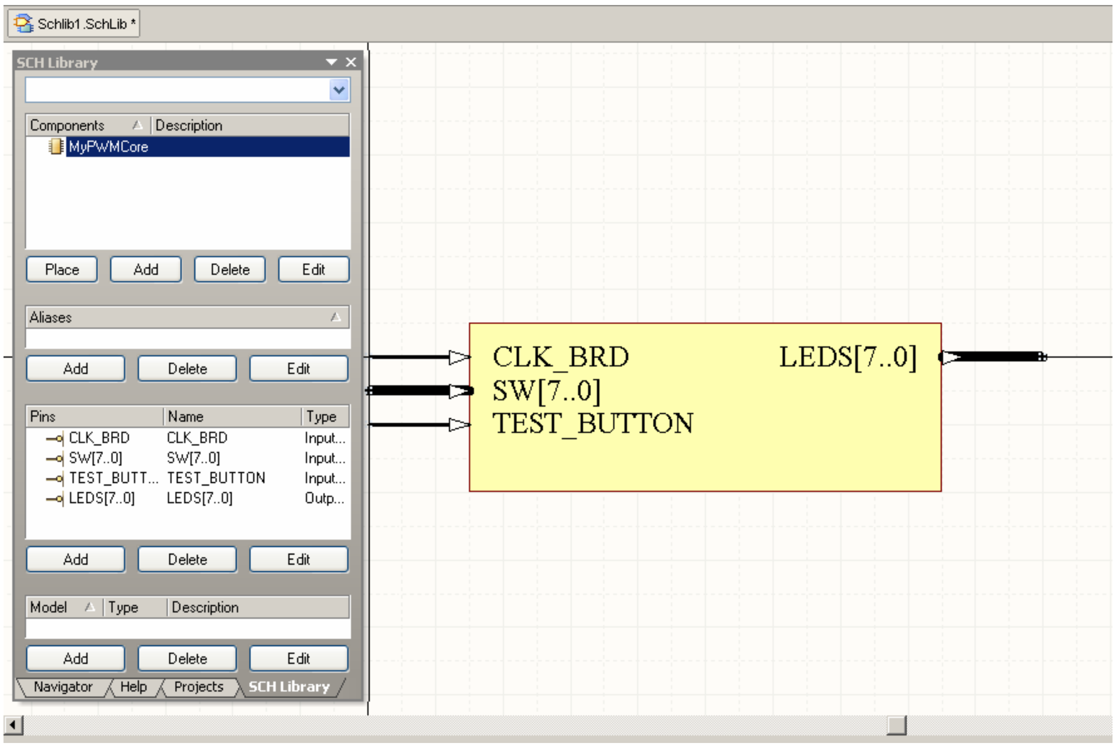
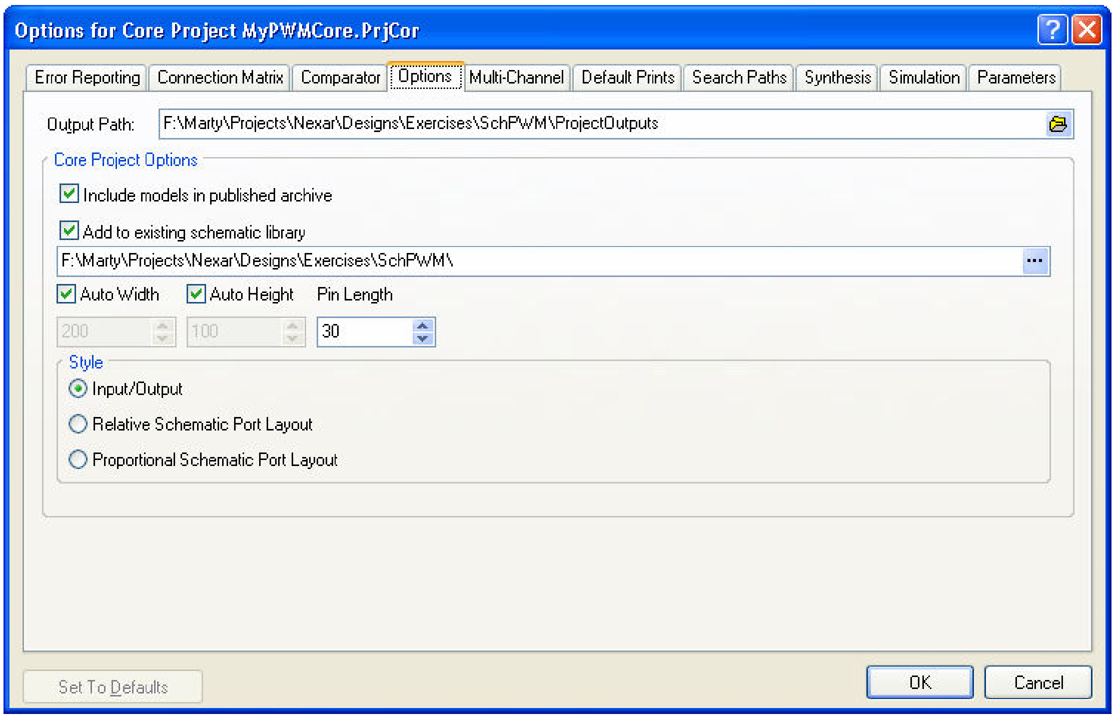


Figure 84. Creating a core component symbol.

- A new schematic library (Schlib1.SchLib) will be automatically created and opened to display the generated symbol. By default the component name will take on the same name as the core project name. Options controlling the new component's appearance and style can be controlled from the **Options** tab of the **Project » Project Options** dialog.


Options for Core Project MyPWMCore.PrjCor

Error Reporting | Connection Matrix | Comparator | Options | Multi-Channel | Default Prints | Search Paths | Synthesis | Simulation | Parameters

Output Path: F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\ProjectOutputs

Core Project Options

- [✓] Include models in published archive
- [✓] Add to existing schematic library

F:\Marty\Projects\Nexar\Designs\Exercises\SchPWM\

[✓] Auto Width 200 — [✓] Auto Height 100 — Pin Length 30

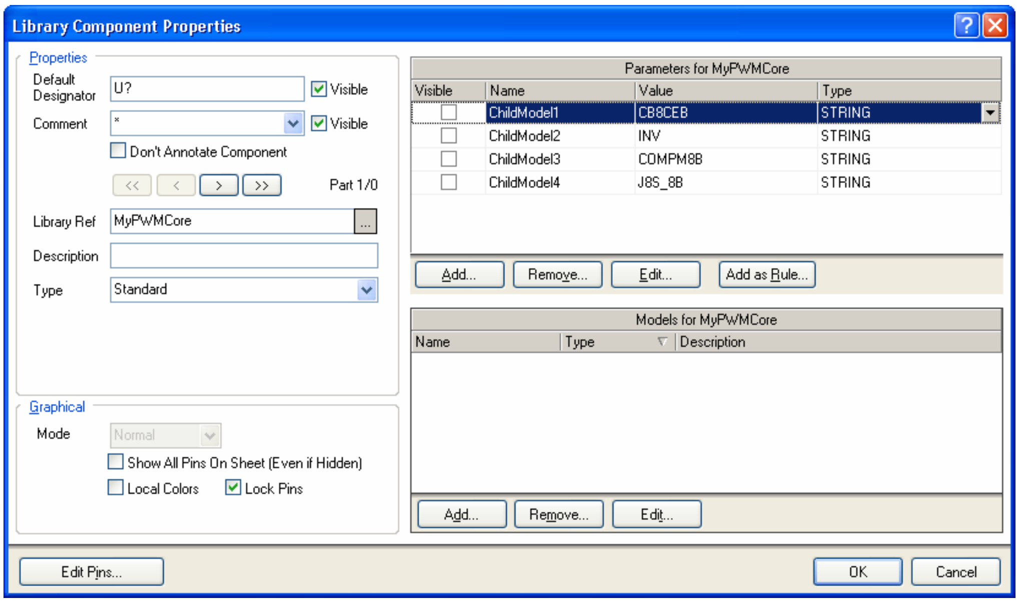
Style

- [✓ Input/Output]
- Relative Schematic Port Layout
- Proportional Schematic Port Layout

Set To Defaults — OK — Cancel

Figure 85. Specifying core component options.

- From within the library editor, select the component in the **Library** panel and select the **Edit** button. The **Library Component Properties** dialog will be displayed. Note that several parameters have been added to indicate which child models are required to be retrieved from the published EDIF zip files.


Library Component Properties

Properties

Default Designator: U? [✓] Visible

Comment: * [✓] Visible

[ ] Don't Annotate Component

Part 1/0

Library Ref: MyPWMCore

Description:

Type: Standard

Parameters for MyPWMCore

Visible	Name	Value	Type
☐	ChildModel1	CB8CEB	STRING
☐	ChildModel2	INV	STRING
☐	ChildModel3	COMPM8B	STRING
☐	ChildModel4	J8S_8B	STRING

Add... — Remove... — Edit... — Add as Rule...

Models for MyPWMCore

Name	Type	Description

Add... — Remove... — Edit...

Graphical

Mode: Normal

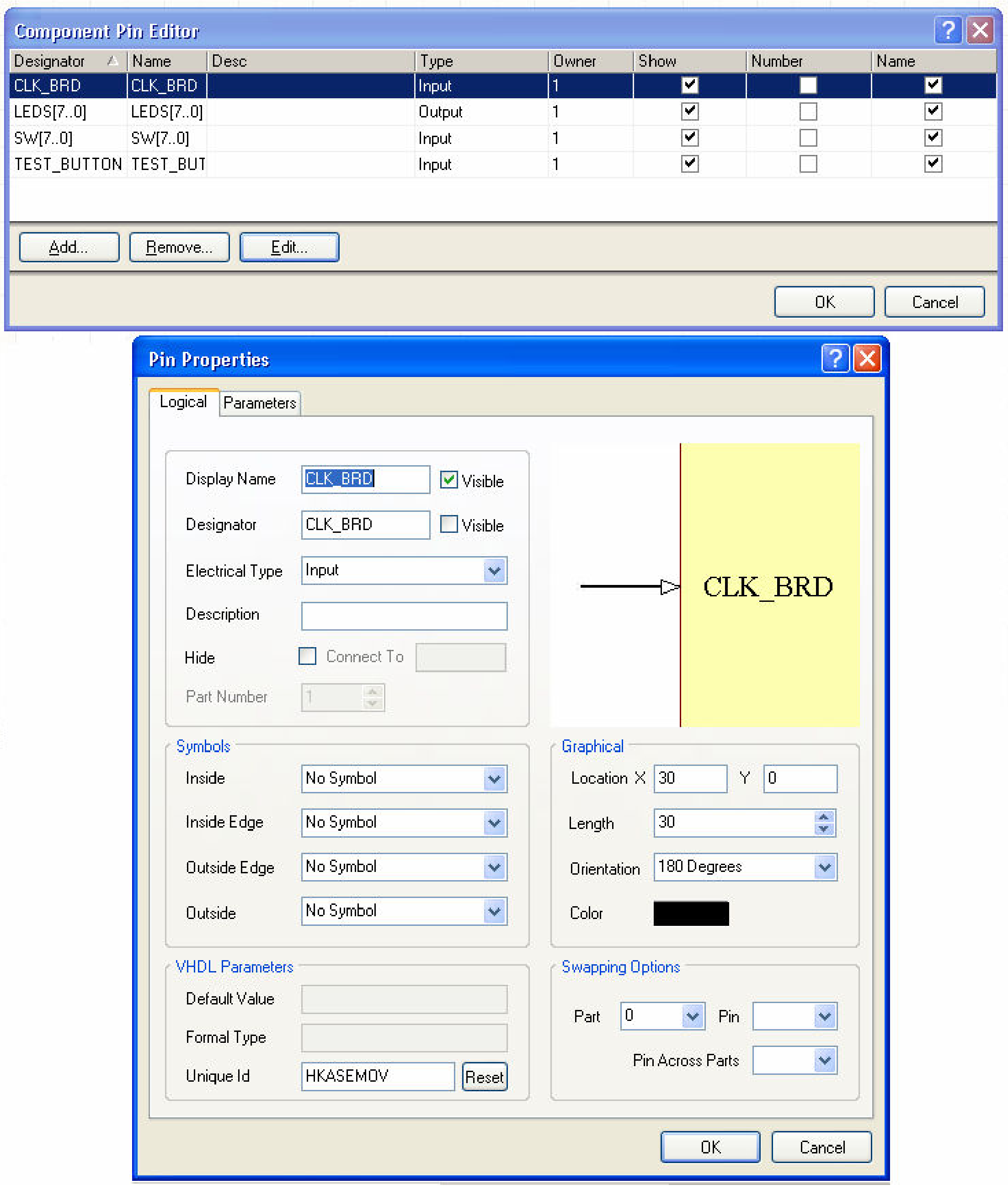
[ ] Show All Pins On Sheet (Even if Hidden)

[ ] Local Colors — [✓] Lock Pins

Edit Pins... — OK — Cancel

Figure 86. Specifying the properties of the newly created core component symbol.

- Clicking on the **Edit Pins** button will enable further modification of the properties and appearance of the schematic symbol.
- From the schematic library editor, adjust the symbol properties as appropriate and save the component. Save the library before exiting.


Component Pin Editor

Designator	Name	Desc	Type	Owner	Show	Number	Name
CLK_BRD	CLK_BRD		Input	1	✓	☐	✓
LEDS[7..0]	LEDS[7..0]		Output	1	✓	☐	✓
SW[7..0]	SW[7..0]		Input	1	✓	☐	✓
TEST_BUTTON	TEST_BUT		Input	1	✓	☐	✓

Pin Properties

Logical | Parameters

Display Name: CLK_BRD ✓ Visible
Designator: CLK_BRD ☐ Visible
Electrical Type: Input
Description:
Hide ☐ Connect To
Part Number: 1

Symbols
Inside: No Symbol
Inside Edge: No Symbol
Outside Edge: No Symbol
Outside: No Symbol

Graphical
Location X 30 Y 0
Length 30
Orientation 180 Degrees
Color ■

VHDL Parameters
Default Value
Formal Type
Unique Id HKASEMOV

Swapping Options
Part 0 Pin
Pin Across Parts

Figure 87. Editing the core component pins.

10.11 Using a core component

When a core component is synthesized and published, the EDIF model is archived into the location specified in the **FPGA Preferences** dialog. Any project that subsequently uses the core component must ensure that the EDIF archive can be found within the search path. The search sequence for EDIF models is:

$project_dir

$user_edif\$vendor\$family

$user_edif\$vendor

$user_edif

$system_edif\$vendor\$family

$system_edif\$vendor

$system_edif

Note that the search locations includes the project directory which makes it useful if you need to transfer the design to another PC that does not have the user EDIF models location defined.

10.12 Exercise 6 – Create a core component from MyPWM

1. Create a new core project and call it MyPWMCore.PrjCor. Note that the filename must not have spaces in it.
2. Follow the steps in Section 10.4 to ensure the settings are correct.
3. Attach the existing MyPWM.SchDoc that you created as part of exercise 3.
4. Create a project level constraint file and call it MyPWMPrj.Constraint. Add the following to this constraint file:

Record=Constraint | **TargetKind**=Port | **TargetId**=CLK_BRD | **FPGA_CLOCK_PIN**=True

Record=Constraint | **TargetKind**=Port | **TargetId**=CLK_BRD | **FPGA_CLOCK**=True

Record=Constraint | TargetKind=Port | TargetId=CLK_BRD | FPGA_CLOCK_FREQUENCY=50Mhz

Figure 88. Updates to be made to MyPWMPrj.Constraint file.

5. Create a constraint file each for an Altera Cyclone device as well as a Xilinx SpartanIIE device.
6. Create a configuration that links each of the individual device constraint files with the project constraint file.
7. Synthesize all configurations and publish the design. Check the **User Presynthesized model Folder** (as set in Section 10.4) using windows explorer and view the directories that are created and their contents.
8. Create a core schematic symbol and save it to the library MyCoreLib.SchLib.
9. Create a new FPGA project and schematic that makes use of your PWM core and test it on the NanoBoard.

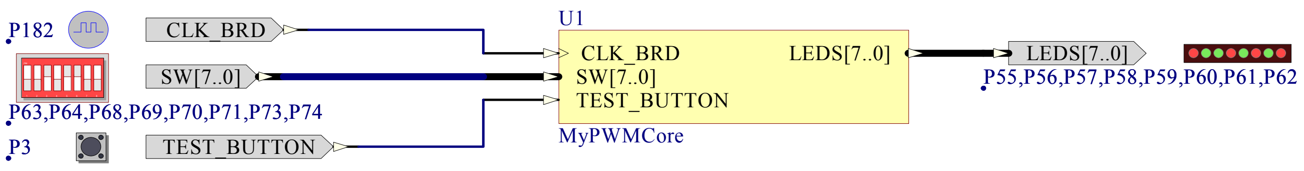


Figure 89. Test project used to test the function of MyPWMCore.

11 FPGA design simulation

Altium Designer supports behavioral simulation of VHDL designs. This is particularly useful when verifying the functional operation of digital circuits prior to implementing them inside an FPGA.

11.1 Creating a testbench

Before simulation can begin, a ***VHDL Testbench*** file must be created to drive the simulation session. Conceptually, the Testbench straddles the Design Under Test (DUT) and drives the DUT's inputs whilst observing its outputs.

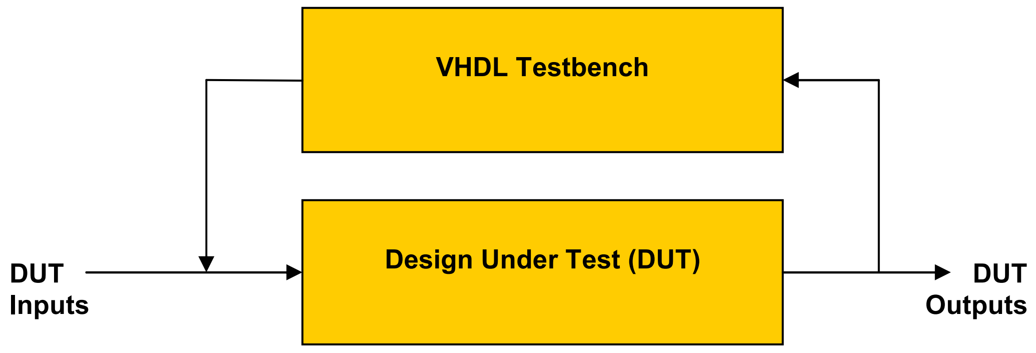


Figure 90. Conceptual view of how a VHDL testbench interacts with the Design Under Test (DUT).

Altium Designer provides a convenient method for building a VHDL Testbench based on the inputs and outputs of the nominated DUT. A shell testbench file can be automatically created by the system.

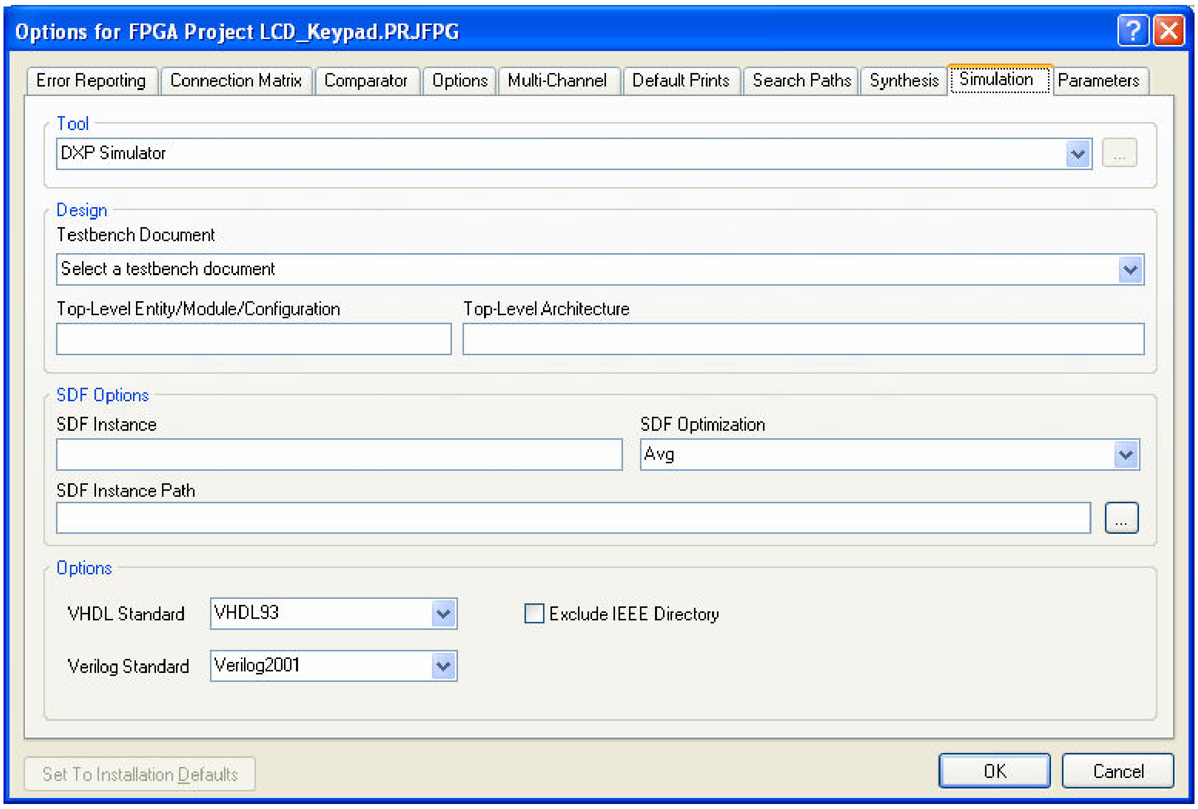
- Open a schematic document and select **Tools » Convert » Create VHDL Testbench** from the menu.
- Open a VHDL document and select **Design » Create VHDL Testbench**.

A new VHDL document will be created with the extension `.VHDTST` and will be added to the project.

Within the Testbench file will be a comment "—insert stimulus here". By placing VHDL code at this point you can control the operation of the simulation session. At a minimum, the Testbench must set all of the DUT's inputs to a known state. If the DUT requires a clock then that too must be provided by the Testbench. Most simulation errors occur as a result of the Testbench failing to properly initialize the inputs of the DUT.

11.2 Assigning the Testbench Document

Once you have created the Testbench file but before a simulation can begin, Altium Designer needs to be formally told which VHDL document in the project will be used to drive the simulation. Select **Project Options** by right clicking on the FPGA project in the **Projects** panel or use the menu to select **Project » Project Options** Select the **Simulation** tab from within the **Project Options** dialog and select the appropriate **Testbench Document** from the drop-down list.



Options for FPGA Project LCD_Keypad.PRJFPG

Tool: DXP Simulator

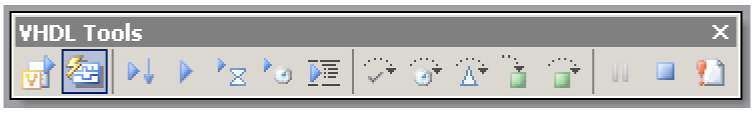
Design
Testbench Document: Select a testbench document
Top-Level Entity/Module/Configuration:
Top-Level Architecture:

SDF Options
SDF Instance:
SDF Optimization: Avg
SDF Instance Path:

Options
VHDL Standard: VHDL93
Exclude IEEE Directory: ☐
Verilog Standard: Verilog2001

Figure 91. Specifying the testbench document.

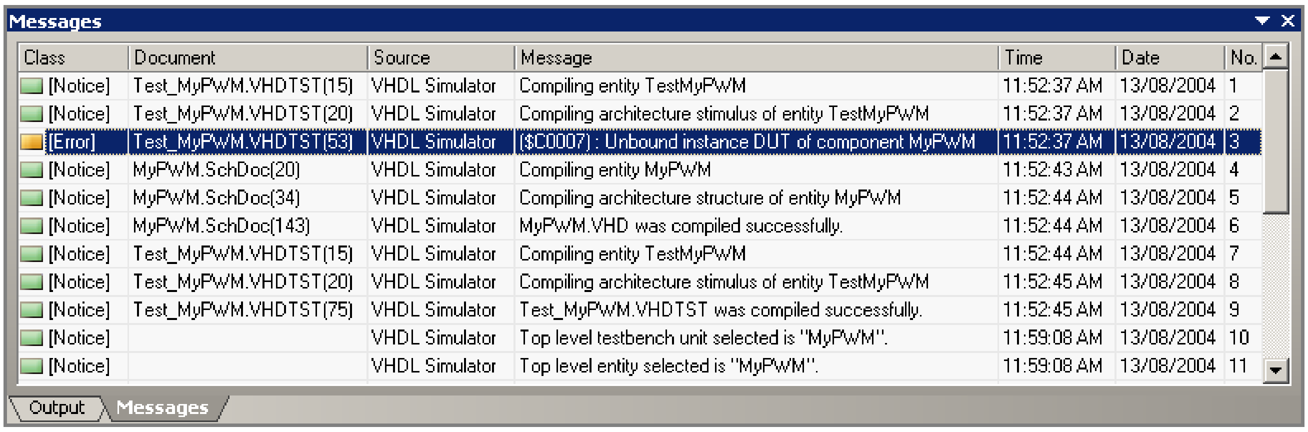
11.3 Initiating a simulation session



A simulation session can be initiated by selecting **Simulator » Simulate** from the menu or by clicking the simulation button in the VHDL Tools toolbar whilst a VHDL document is active in the main window.

11.4 Project compile order

When you first run a simulation from a testbench, Altium Designer may need to establish the compilation order of the VHDL documents. Whilst performing this process, you may see an error appear in the **Messages** panel with the message: “Unbound instance DUT of component …”. Do not be concerned as this is normal when you first run a simulation.



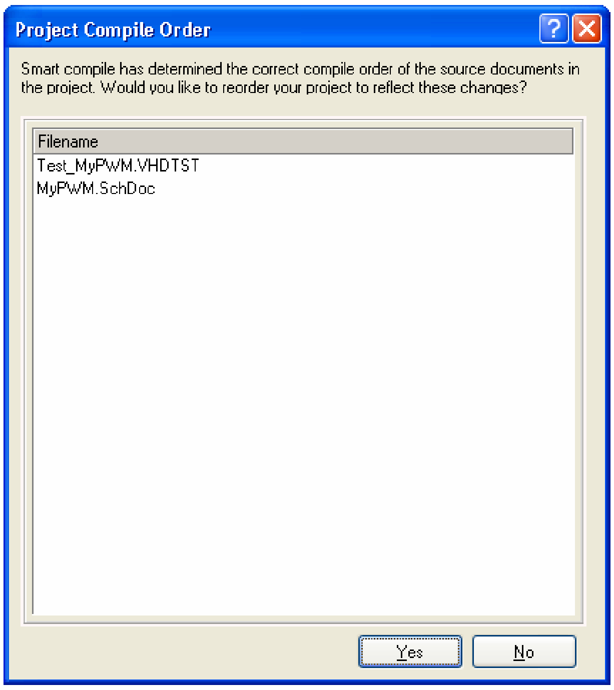
Messages

Class	Document	Source	Message	Time	Date	No.
[Notice]	Test_MyPWM.VHDTST(15)	VHDL Simulator	Compiling entity TestMyPWM	11:52:37 AM	13/08/2004	1
[Notice]	Test_MyPWM.VHDTST(20)	VHDL Simulator	Compiling architecture stimulus of entity TestMyPWM	11:52:37 AM	13/08/2004	2
[Error]	Test_MyPWM.VHDTST(53)	VHDL Simulator	($C0007) : Unbound instance DUT of component MyPWM	11:52:37 AM	13/08/2004	3
[Notice]	MyPWM.SchDoc(20)	VHDL Simulator	Compiling entity MyPWM	11:52:43 AM	13/08/2004	4
[Notice]	MyPWM.SchDoc(34)	VHDL Simulator	Compiling architecture structure of entity MyPWM	11:52:44 AM	13/08/2004	5
[Notice]	MyPWM.SchDoc(143)	VHDL Simulator	MyPWM.VHD was compiled successfully.	11:52:44 AM	13/08/2004	6
[Notice]	Test_MyPWM.VHDTST(15)	VHDL Simulator	Compiling entity TestMyPWM	11:52:44 AM	13/08/2004	7
[Notice]	Test_MyPWM.VHDTST(20)	VHDL Simulator	Compiling architecture stimulus of entity TestMyPWM	11:52:45 AM	13/08/2004	8
[Notice]	Test_MyPWM.VHDTST(75)	VHDL Simulator	Test_MyPWM.VHDTST was compiled successfully.	11:52:45 AM	13/08/2004	9
[Notice]		VHDL Simulator	Top level testbench unit selected is "MyPWM".	11:59:08 AM	13/08/2004	10
[Notice]		VHDL Simulator	Top level entity selected is "MyPWM".	11:59:08 AM	13/08/2004	11

Figure 92. Messages panel.

After a brief moment, the **Project Compile Order** dialog will appear indicating the suggested compilation order (Figure 93).

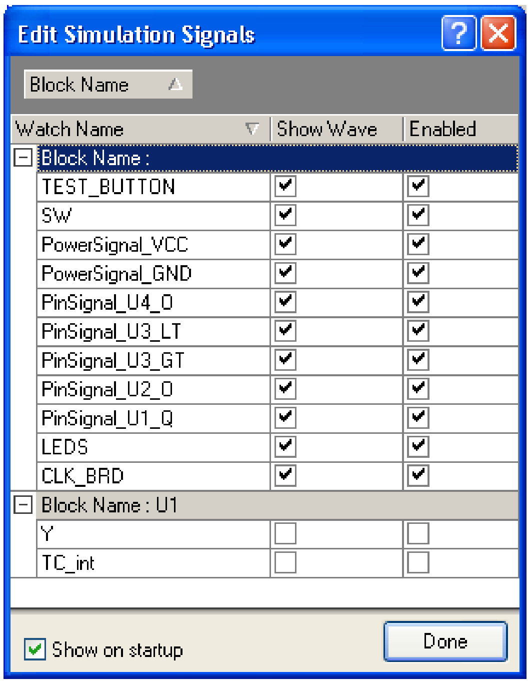
The compilation order of the project can be changed at a later time if necessary by selecting **Project » Project Order** or by right clicking on the FPGA project in the **Projects** panel and selecting **Project Order**


Project Compile Order

Smart compile has determined the correct compile order of the source documents in the project. Would you like to reorder your project to reflect these changes?

Filename
Test_MyPWM.VHDTST
MyPWM.SchDoc

Figure 93. Confirming the project compiler order.


Edit Simulation Signals

Block Name

Watch Name	Show Wave	Enabled
Block Name :		
TEST_BUTTON	✓	✓
SW	✓	✓
PowerSignal_VCC	✓	✓
PowerSignal_GND	✓	✓
PinSignal_U4_O	✓	✓
PinSignal_U3_LT	✓	✓
PinSignal_U3_GT	✓	✓
PinSignal_U2_O	✓	✓
PinSignal_U1_Q	✓	✓
LEDS	✓	✓
CLK_BRD	✓	✓
Block Name : U1		
Y	☐	☐
TC_int	☐	☐

✓ Show on startup

Figure 94. Specifying signals to display in the simulation.

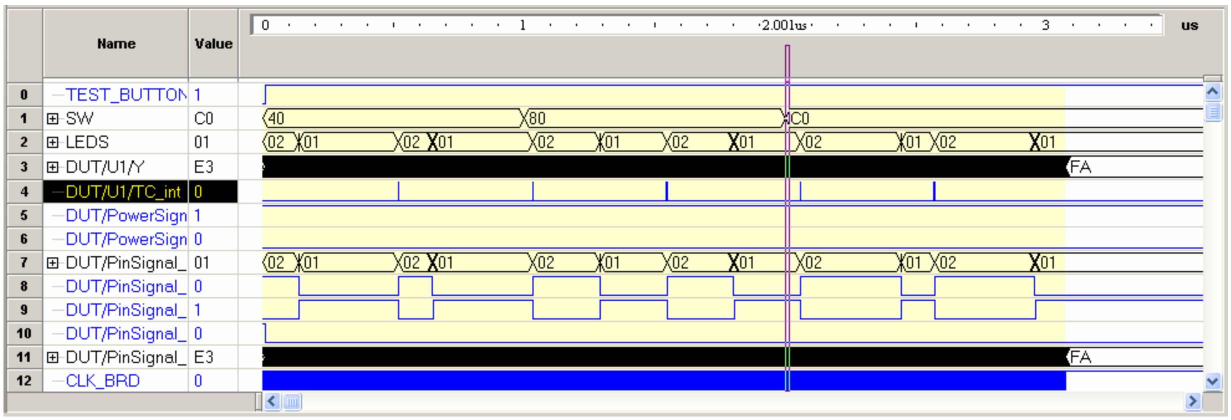
11.5 Setting up the simulation display

The Signals dialog (Figure 94) is automatically presented at the beginning of a simulation or it can be accessed via **Simulator » Signals**.

The **Watch Name** is the name of the signal declared inside the block of VHDL code.

Signals must be **Enabled** in order to be a part of the simulation. Furthermore, if they need to be displayed as part of the simulation output then **Show Wave** must also be selected.

The Waveform viewer provides a visualization of the status of each of the displayed signals.



	Name	Value
0	TEST_BUTTON	1
1	⊞ SW	C0
2	⊞ LEDS	01
3	⊞ DUT/U1/Y	E3
4	DUT/U1/TC_int	0
5	DUT/PowerSign	1
6	DUT/PowerSign	0
7	⊞ DUT/PinSignal_	01
8	DUT/PinSignal_	0
9	DUT/PinSignal_	1
10	DUT/PinSignal_	0
11	⊞ DUT/PinSignal_	E3
12	CLK_BRD	0

Figure 95. The waveform viewer.

- The icon ⊞ next to the bus name indicates a bus signal. Clicking on this icon will expand the bus into its individual signals for closer inspection.
- The time cursor (indicated by the purple vertical bar) can be dragged along the time axis via the mouse. The current position of the cursor is provided in the time bar across the top of the display.
- Zooming in or out is achieved by pressing the Page Up or Page Down keys respectively.
- The display format of the individual signals can be altered via the menu item **Tools » Format and Radix**.

11.6 Running and Debugging a Simulation

Running a simulation is reasonably straightforward with all the stepping/running functions that you might wish to use being available from the **Simulation** menu or the **VHDL Tools** toolbar.

- **Run Forever** will run the simulation indefinitely until **Stop** is pressed. This command is used to run a VHDL simulation until there are no changes occurring in the signals enabled in the simulation.
- Use the **Run (for a time step)** command to run the current simulation for a user-specified period of time (time step).
- **Run for** (the last time step) will run the simulator for the same period of time as specified in the last **Run** command.
- **Run to Time** will run the simulator to an absolute time. Selecting a time prior to where the simulation has already simulated to will cause the simulator to do nothing.
- **Run to Cursor** is useful when debugging VHDL source and will cause the simulator to run until the defined cursor location is encountered in a source VHDL document. The simulator will simulate everything up to the selected line. Make sure that the **Show execution point** option is enabled, in the **Debugging Options** region of the **FPGA – Simulation Debugger** page of the *Preferences* dialog (Figure 97).
- **Custom Step** (Run simulation to the next debug point): This command is used to run the current simulation, up to the next executable line of code in the source VHDL documents. The next executable code point can be anywhere in the code and therefore the command can be considered to be stepping through the code in parallel, rather than the sequentially-based step into and step over commands.

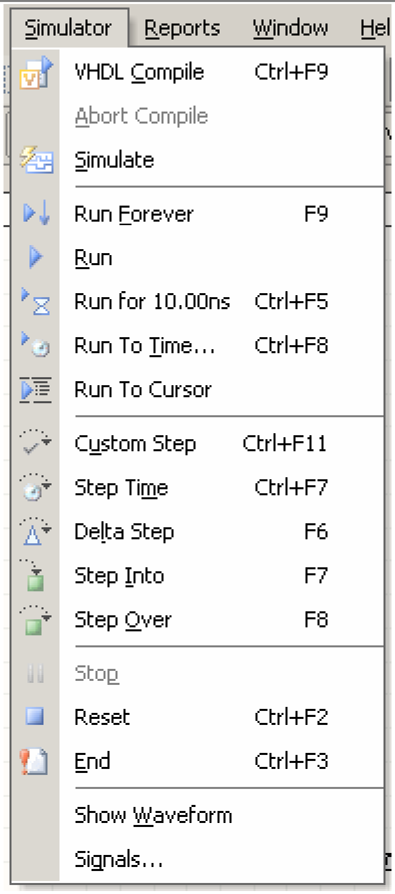
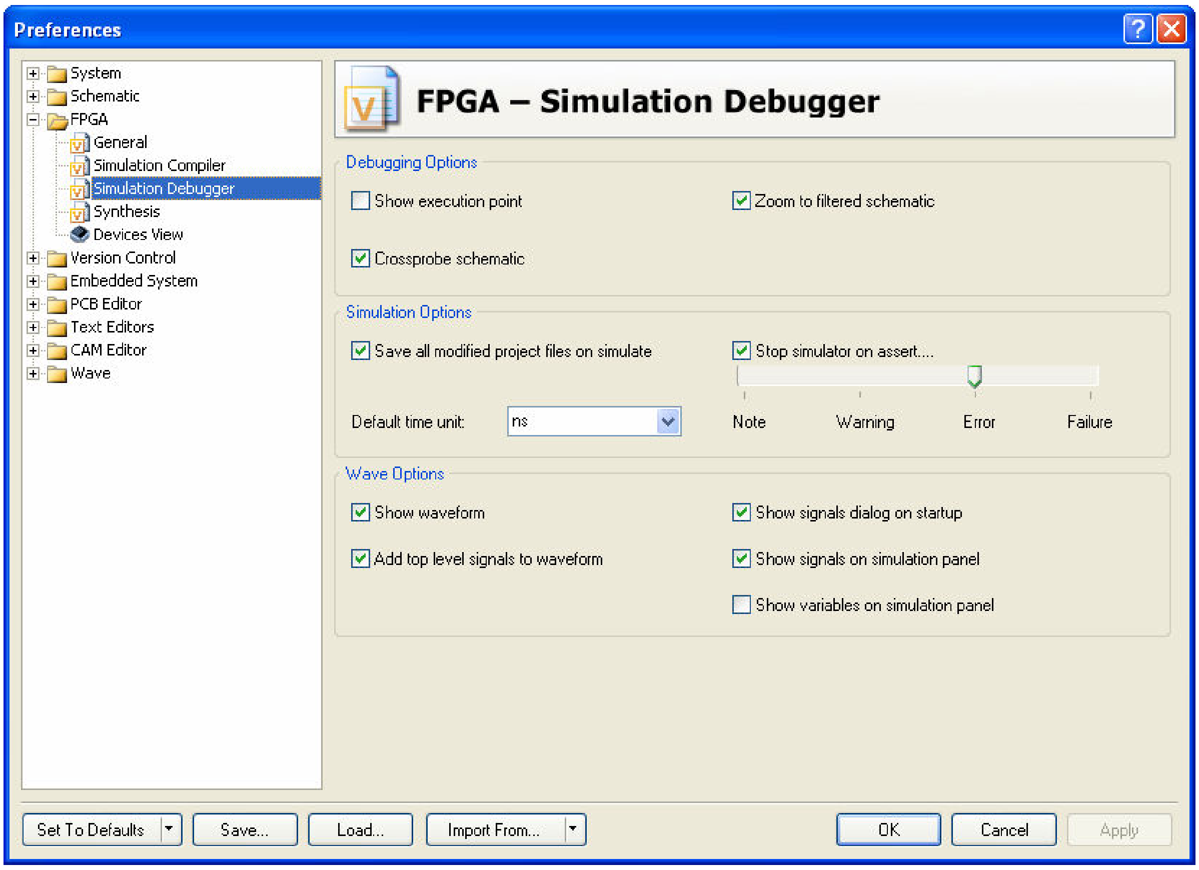


Figure 96. The simulation menu.

- **Step Time:** This command is used to run the current simulation, executing code in the source VHDL documents until time increments – i.e. all delta time events prior to the next time increment will be executed.
- **Delta Step:** This command is used to run the current simulation for a single cycle, which can be called a Delta step. A Delta step can be so small that no change in real time is seen.
- **Step Into** enables the user to single-step through the executable lines of code in the source VHDL documents. If any procedures/functions are encountered, stepping will continue into the called procedure or function.
- **Step Over** is similar to **Step Into** except that if any procedures/functions are encountered, stepping will execute the entire procedure/function as a single executable line and will not step into it.



FPGA – Simulation Debugger

Debugging Options
- [ ] Show execution point
- [✓] Zoom to filtered schematic
- [✓] Crossprobe schematic

Simulation Options
- [✓] Save all modified project files on simulate
- [✓] Stop simulator on assert.... (Note / Warning / [✓ Error] / Failure)
- Default time unit: ns

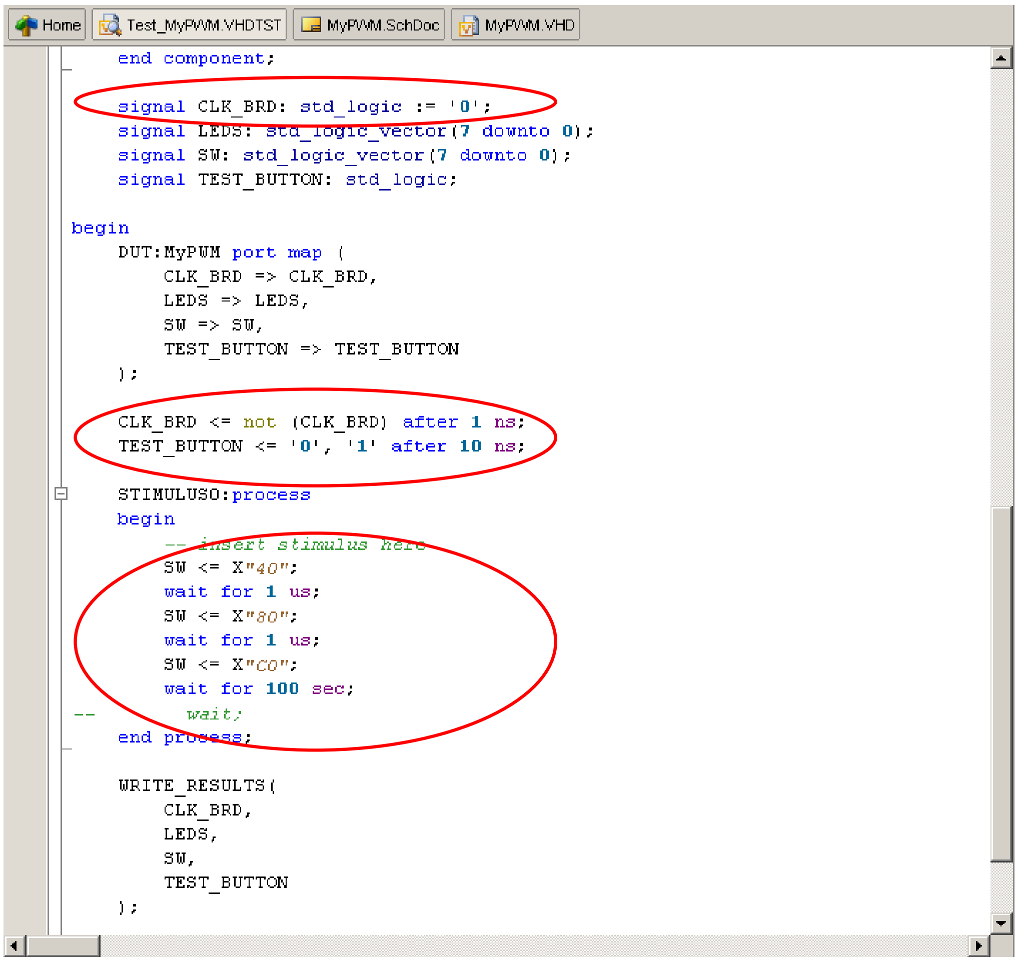
Wave Options
- [✓] Show waveform
- [✓] Show signals dialog on startup
- [✓] Add top level signals to waveform
- [✓] Show signals on simulation panel
- [ ] Show variables on simulation panel

Figure 97. The simulation debugger options in the preferences dialog.

- **Stop** will pause the simulator at its current simulation point. A paused simulation can continue to be run with any of the above commands.
- **Reset** will abort the current simulation, clear any waveforms and reset the time back to 0.
- **End** terminates the entire simulation session. Ended simulations can not be restarted other than by initiating another simulation session.

11.7 Exercise 7 – Create a testbench and simulate MyPWM

1. Open the project you created in Exercise 2 and make MyPWM.VHD the active document.
2. Select **Design » Create VHDL Testbench** from the menu.
3. Update the testbench to be the same as the code listed in Figure 98.



```
    end component;

    signal CLK_BRD: std_logic := '0';
    signal LEDS: std_logic_vector(7 downto 0);
    signal SW: std_logic_vector(7 downto 0);
    signal TEST_BUTTON: std_logic;

begin
    DUT:MyPWM port map (
        CLK_BRD => CLK_BRD,
        LEDS => LEDS,
        SW => SW,
        TEST_BUTTON => TEST_BUTTON
    );

    CLK_BRD <= not (CLK_BRD) after 1 ns;
    TEST_BUTTON <= '0', '1' after 10 ns;

    STIMULUS0:process
    begin
        -- insert stimulus here
        SW <= X"40";
        wait for 1 us;
        SW <= X"80";
        wait for 1 us;
        SW <= X"C0";
        wait for 100 sec;
--      wait;
    end process;

    WRITE_RESULTS(
        CLK_BRD,
        LEDS,
        SW,
        TEST_BUTTON
    );
```

Figure 98. Testbench code for testing MyPWM.

4. Update the testbench document, top-level entity/configuration and top-level architecture fields in the simulation tab of the **Project » Project Options** dialog.
5. Compile the testbench document and rectify any errors.
6. Run the simulation by selecting **Simulator » Simulate**.
7. Run the simulator for 2us.
8. Observe the waveforms for LEDS[0] and LEDS[1]. Is it what you expect? Try changing the PWM period by changing the value of SW in the testbench.

12 Review