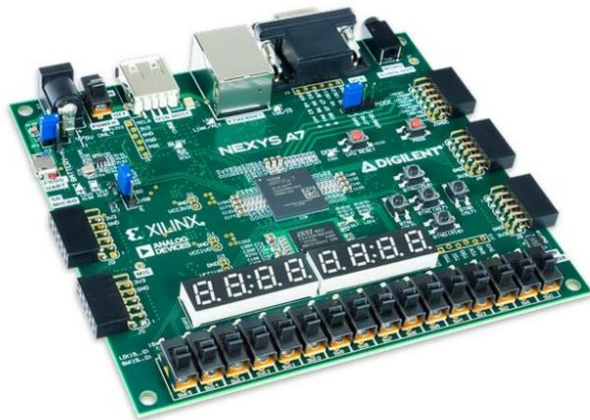
Hellenic Mediterranean University

School of Engineering

Electronic Engineering Department



Vivado for Hardware Development

Application development while exploring the Digilent Nexys A7 board capabilities, using VHDL and Xilinx's Vivado Design Suite.

FINAL PROJECT

of

Arnout Van Daele

Supervisor: **Dr Eng. Nikolaos S. Petrakis, Lecturer HMU**

Chania, Greece,

July 2021

Table of Contents

Table of Contents i
Abstract ii
Περίληψη ii
1. Introduction 1
2. About the development card Digilent Nexys A7 3
2.1. Nexys A7 3
2.2. Features 3
2.3. Purchasing Options 5
2.4. Functional Description 6
2.4.1. Power Supplies 6
2.4.2. Protection 7
2.5. FPGA Configuration 7
2.5.1. JTAG Configuration 8
2.5.2. Quad-SPI Configuration 9
2.5.3. USB Host and Micro SD Programming 9
2.6. Memory 10
2.6.1. DDR2 10
2.6.2. Quad-SPI Flash 11
2.7. Ethernet PHY 12
2.8. Oscillators/Clocks 13
2.9. USB-UART Bridge (Serial Port) 13
2.10. USB HID Host 14
2.10.1. HID Controller 15
2.10.2. Keyboard 16
2.10.3. Mouse 17
2.11. VGA Port 17
2.12. Basic I/O 18
2.12.1. Seven-Segment Display 20
2.12.2. Tri-Color LED 21
2.13. Pmod Ports 22
2.13.1. Dual Analog/Digital Pmod 23
2.14. MicroSD Slot 23
2.15. Temperature Sensor 24
2.15.1. I^2C Interface 24
2.16. Accelerometer 24
2.16.1. SPI Interface 25
2.16.2. Interrupts 25
2.17. Microphone 25
2.18. Built-In Self-Test 25
3. VHDL and the Xilinx Vivado Design Suite 27
3.1. Introduction 27
3.2. Different levels of representation and abstraction 27
3.3. Basic Structure of a VHDL file 28
3.4. Lexical Elements of VHDL 32
3.5. Data Objects: Signals, Variables and Constants 34
3.5.1. Signal 34
3.5.2. Variable 34
3.5.3. Constant 35
3.6. Data types 35
3.7. Operators 37
3.8. A 2bit counter as an example 37
3.9. Vivado Design Suite 41
4. Application development examples 42
4.1. 4 bit adder 42
4.1.1. A word on multiplexing 47
4.2. Temperature Sensor 47
4.3. Accelerometer 50
5. Conclusions 52
References 53

Abstract

The purpose of this final project is to delve into the design of digital systems using a hardware description language and FPGAs (Field Programmable Gate Arrays), taking into account both advances in science and limited access to financial resources. To this end, the VHDL hardware description language was systematically studied and a short user guide was compiled, which includes the basic features of the language to be used. The Xilinx Vivado software package was then tested, which is a complete circuit design and gateway implementation (FPGA) environment. The design and simulation steps were described in detail. A Digilent Nexys A7 development board containing the Xilinx Artix-7 family FPGA was used to complete and test the circuits. Exercises were then developed on this board as samples using the seven-segment displays, the built-in temperature sensor and the built-in accelerometer.

Περίληψη

Στόχος της παρούσας πτυχιακής εργασίας είναι η εμβάθυνση στη σχεδίαση ψηφιακών συστημάτων χρησιμοποιώντας μία γλώσσα περιγραφής υλικού και κυκλώματα FPGA (Field Programmable Gate Array), λαμβάνοντας υπόψη τόσο την πρόοδο στον συγκεκριμένο επιστημονικό τομέα, όσο και την περιορισμένη πρόσβαση σε οικονομικούς πόρους. Προς την κατεύθυνση αυτή, λοιπόν, μελετήθηκε συστηματικά η γλώσσα περιγραφής υλικού VHDL και συντάχθηκε ένας σύντομος οδηγός χρήσης, ο οποίος περιλαμβάνει τα βασικά χαρακτηριστικά τής γλώσσας που θα χρησιμοποιηθούν στο εργαστήριο. Στην συνέχεια δοκιμάστηκε το λογισμικό πακέτο Vivado της Xilinx, το οποίο είναι ένα πλήρες περιβάλλον σχεδίασης κυκλωμάτων και υλοποίησης με παρατάξεις πυλών (FPGA). Περιγράφηκαν αναλυτικά τα βήματα σχεδίασης και προσομοίωσης. Για την ολοκλήρωση και δοκιμαστική λειτουργία των κυκλωμάτων χρησιμοποιήθηκε μια αναπτυξιακή πλακέτα Nexys A7 της Digilent που περιέχει το FPGA της οικογένειας Artix-7 της Xilinx. Ακολούθως αναπτύχθηκαν σε αυτήν την πλακέτα, ως δείγματα, ασκήσεις χρησιμοποιώντας τέσσερα ψηφία 7 τομέων (seven segment displays), τον ενσωματωμένο αισθητήρα θερμοκρασίας και το ενσωματωμένο επιταχυνσιόμετρο.

1. Introduction

The evolution of technology has led to the development of reprogrammable devices that have completely changed the process of designing digital systems. Such devices are the FPGAs that appeared around 1980 and have now been established for simple to complex implementations.

FPGAs are digital integrated circuits that contain programmable digital logic blocks and programmable interfaces. By programming the logic components, the logic functions that express the operation of a digital circuit are implemented. FPGA application development is due to the flexibility and speed they provide and is usually done using hardware description languages such as VHDL and/or Verilog. Such applications concern various fields such as cryptography, bioinformatics, defense systems, etc.

The ever-increasing complexity observed in the design and construction of digital systems requires the use of design tools that facilitate this process, which is why such a tool will be studied in this dissertation. The five chapters of the work give the reader the opportunity to expand their knowledge of the VHDL language, Xilinx VIVADO design environment and FPGA implementation.

More specifically, after this chapter which is a brief introduction, the second chapter presents the features and capabilities of the Nexys A7 development card that contains the FPGA of the Artix-7 family of Xilinx and is compatible with Vivado. This card includes slide switches, buttons, LEDs, 7-segment displays, temperature sensor (ADT7420) and analog accelerometer (ADXL362), which were used in the examples.

The third chapter presents a brief guide to the VHDL hardware description language. First a brief historical overview is made and the differences in relation to the common programming languages are pointed out. The structure of the code in VHDL, verbal elements and syntax, representation of numbers and characters, operators, objects and data types are then analyzed. Finally, it describes how we can describe combinational and/or synchronous sequential circuits with VHDL and how to design hierarchical circuits. A guide to Xilinx's Vivado Design Suite follows, describing the design and implementation steps. Specifically, it describes the process from writing the code in VHDL to programming the FPGA.

The fourth chapter presents three designs that were made in the design environment of Vivado and their implementation in the Nexys A7 development board mentioned above.

The fifth chapter mentions the conclusions we reached after the completion of this work as well as suggestions for further extensions.

Of course, there is also a list of bibliographic sources that includes the books and websites that were used to prepare the work.

The reason why I chose to deal with this topic is the familiarity I gained with VHDL during my studies and the desire to gain knowledge, experience and skills in digital design applications using modern design tools and development cards with fantastic possibilities.

At this point I consider it my duty to thank my supervising professor Dr. Eng. Nikolaos Petrakis for the valuable collaboration I had with him as well as for the decisive guidance and support he offered me at all stages of the preparation of my final project.

2. About the development card Digilent Nexys A7

2.1. Nexys A7

The Nexys A7 board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA, generous external memories, and collection of USB, Ethernet, and other ports, the Nexys A7 can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys A7 to be used for a wide range of designs without needing any other components.

2.2. Features

Artix-7 FPGA

- 15,850 Programmable logic slices, each with four 6-input LUTs and 8 flip-flops (*8,150 slices)
- 4,860 Kbits of fast block RAM (*2,700 Kbits)
- Six clock management tiles, each with phase-locked loop (PLL)
- 240 DSP slices (*120 DSPs)
- Internal clock speeds exceeding 450 MHz
- Dual-channel, 1 MSPS internal analog-digital converter (XADC)

Memory

- 128MiB DDR2
- Serial Flash
- microSD card slot

Power

- Powered from USB or any 4.5V-5.5V external power source

USB and Ethernet

- 10/100 Ethernet PHY
- USB-JTAG programming circuitry
- USB-UART bridge
- USB HID Host for mice, keyboards and memory sticks

Simple User Input/Output

- 16 Switches
- 16 LEDs
- Two RGB LEDs

- Two 4-digit 7-segment displays

Audio and Video

- 12-bit VGA output
- PWM audio output
- PDM microphone

Additional Sensors

- 3-axis accelerometer
- Temperature sensor

Expansion Connectors

- Pmod connector for XADC signals
- Four Pmod connectors providing 32 total FPGA I/O

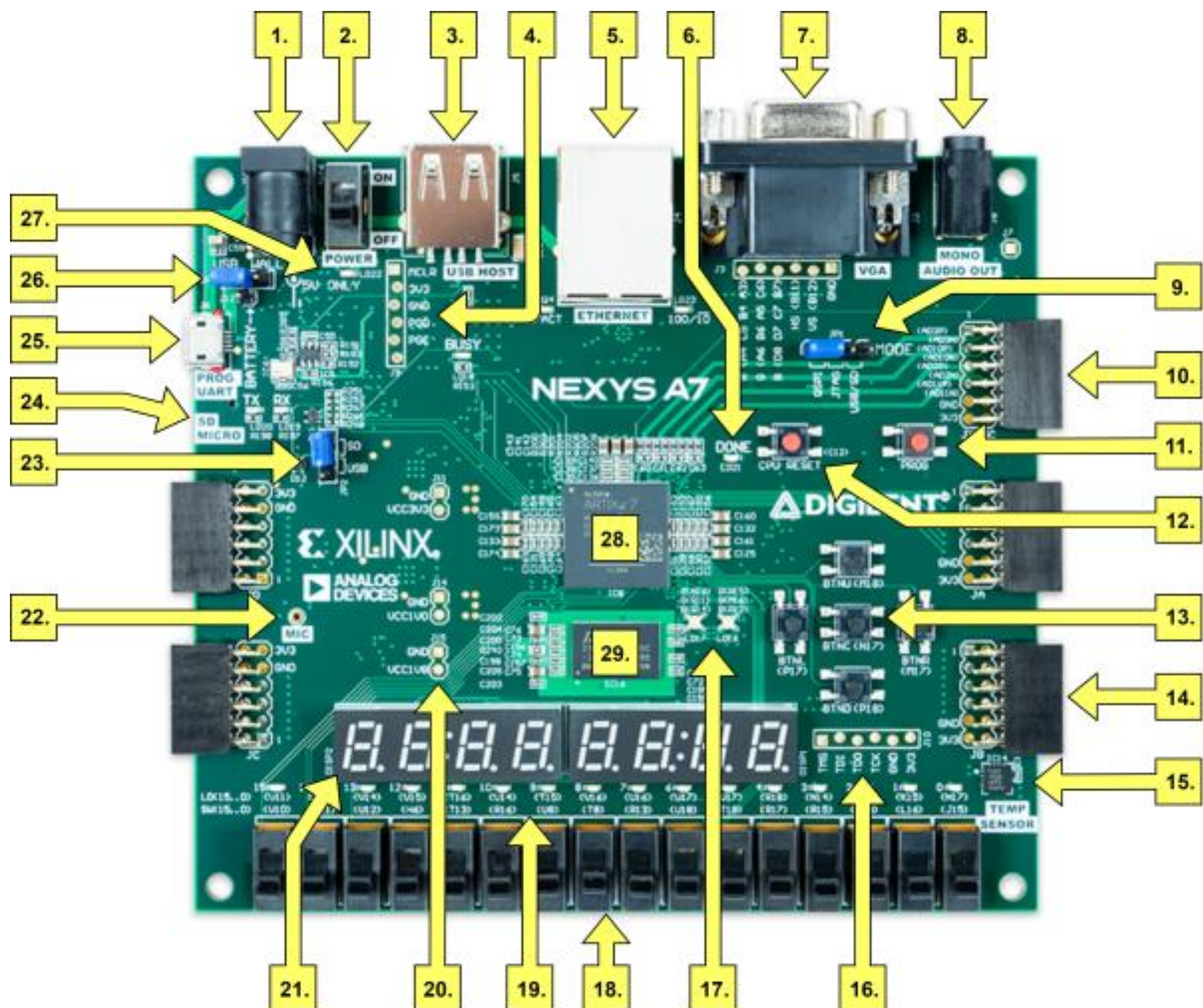


Image 2.1: Nexys A7 Feature Callout

The Nexys A7-100T is compatible with Xilinx's Vivado® Design Suite as well as the ISE® toolset, which includes ChipScope™ and EDK. Xilinx ISE has been discontinued in favor of Vivado® Design Suite. The Nexys A7-50T variant is compatible only with Vivado® Design Suite. Xilinx offers free WebPACK™ versions of these toolsets, so designs can be implemented at no additional cost. The Nexys A7 is not supported by the Digilent Adept Utility.

Callout	Component Description	Callout	Component Description
1	Power jack	**16**	JTAG port for (optional) external cable
2	Power switch	**17**	Tri-color (RGB) LEDs
3	USB host connector	**18**	Slide switches (16)
4	PIC24 programming port (factory use)	**19**	LEDs (16)
5	Ethernet connector	**20**	Power supply test point(s)
6	FPGA programming done LED	**21**	Eight digit 7-seg display
7	VGA connector	**22**	Microphone
8	Audio connector	**23**	External configuration jumper (SD / USB)
9	Programming mode jumper	**24**	MicroSD card slot
10	Analog signal Pmod port (XADC)	**25**	Shared UART/ JTAG USB port
11	FPGA configuration reset button	**26**	Power select jumper and battery header
12	CPU reset button (for soft cores)	**27**	Power-good LED
13	Five pushbuttons	**28**	Xilinx Artix-7 FPGA
14	Pmod port(s)	**29**	DDR2 memory
15	Temperature sensor		

2.3. Purchasing Options

The Nexys A7 can be purchased with either a XC7A100T or XC7A50T FPGA loaded. These two Nexys A7 product variants are referred to as the Nexys A7-100T and Nexys A7-50T, respectively. When Digilent documentation describes functionality that is common to both of these variants, they are referred to collectively as the "Nexys A7". When describing something that is only common to a specific variant, the variant will be explicitly called out by its name.

The only difference between the Nexys A7-100T and Nexys A7-50T is the size of the Artix-7 part. The Artix-7 FPGAs both have the same capabilities, but the XC7100T has about a 2 times larger internal FPGA than the XC750T. The differences between the two variants are summarized below:

Product Variant	Nexys A7-100T	Nexys A7-50T
FPGA Part Number	XC7A100T-1CSG324C	XC7A50T-1CSG324I
Look-up Tables (LUTs)	63,400	32,600
Flip-Flops	126,800	65,200
Block RAM	4,860 Kb	2,700 Kb
DSP Slices	240	120
Clock Management Tiles	6	5

2.4. Functional Description

2.4.1. Power Supplies

The Nexys A7 board can receive power from the Digilent USB-JTAG port (J6) or from an external power supply. Jumper JP3 (near the power jack) determines which source is used.

All Nexys A7 power supplies can be turned on and off by a single logic-level power switch (SW16). A power-good LED (LD22), driven by the “power good” output of the ADP2118 supply, indicates that the supplies are turned on and operating normally. An overview of the Nexys A7 power circuit is shown in Figure 2.2.

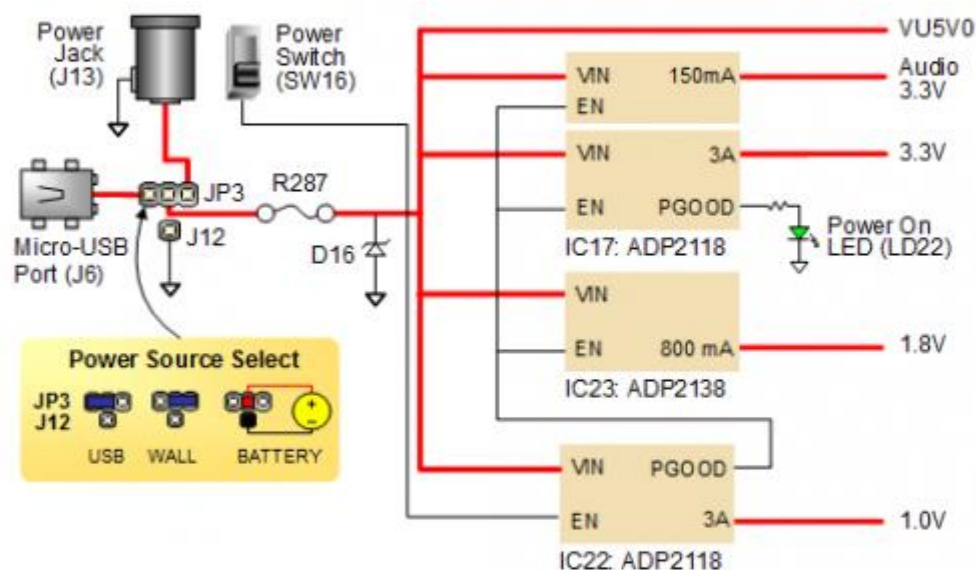


Image 2.2: Nexys A7 Power Circuit

The USB port can deliver enough power for the vast majority of designs. In order to power the board from USB port set jumper JP3 to “USB”. Our out-of-box demo draws ~400mA of current from the 5V input rail. A few demanding applications, including any that drive multiple peripheral boards, might require more power than the USB port can provide. Also, some applications may need to run without being connected to a PC’s USB port. In these instances, an external power supply or battery pack can be used.

An external power supply can be used by plugging into to the power jack (J13) and setting jumper JP3 to “WALL”. The supply must use a coax, center-positive 2.1mm internal-diameter plug, and deliver 4.5VDC to 5.5VDC and at least 1A of current (i.e., at least 5W of power). Many suitable supplies can be purchased from Digilent, through Digi-Key, or other catalog vendors.

An external battery pack can be used by connecting the battery’s positive terminal to the center pin of JP3 and the negative terminal to the pin labeled J12, directly below JP3. Since the main regulator on the Nexys A7 cannot accommodate input voltages over 5.5VDC, an external battery pack must be

limited to 5.5VDC. The minimum voltage of the battery pack depends on the application: if the USB Host function (J5) is used, at least 4.6V needs to be provided. In other cases, the minimum voltage is 3.6V.

Voltage regulator circuits from Analog Devices create the required 3.3V, 1.8V, and 1.0V supplies from the main power input. Table 2.1 provides additional information. Typical currents depend strongly on FPGA configuration and the values provided are typical of medium size/speed designs.

Supply	Circuits	Device	Current (max/typical)
3.3V	FPGA I/O, USB ports, Clocks, RAM I/O, Ethernet, SD slot, Sensors, Flash	IC17: ADP2118	3A/ 0.1 to 1.5A
1.0V	FPGA Core	IC22: ADP2118	3A/ 0.2 to 1.3A
1.8V	DDR2, FPGA Auxiliary and RAM	IC23: ADP2118	0.8A/ 0.5A

Table 2.1 Nexys A7 power supplies.

2.4.2. Protection

The Nexys A7 features overcurrent and overvoltage protection on the input power rail. A 3.5A fuse (R287) and a 5V Zener diode (D16) provide a non-resettable protection for other on-board integrated circuits, as displayed in Figure 2. Applying power outside of the specs outlined in this document is not covered by warranty. If this happens, either or both might get permanently damaged. The damaged parts are not user-replaceable.

.

2.5. FPGA Configuration

After power-on, the Artix-7 FPGA must be configured (or programmed) before it can perform any functions. You can configure the FPGA in one of four ways:

1. A PC can use the Digilent USB-JTAG circuitry (portJ6, labeled “PROG”) to program the FPGA any time the power is on.
2. A file stored in the nonvolatile serial (SPI) flash device can be transferred to the FPGA using the SPI port.
3. A programming file can be transferred to the FPGA from a micro SD card.
4. A programming file can be transferred from a USB memory stick attached to the USB HID port.

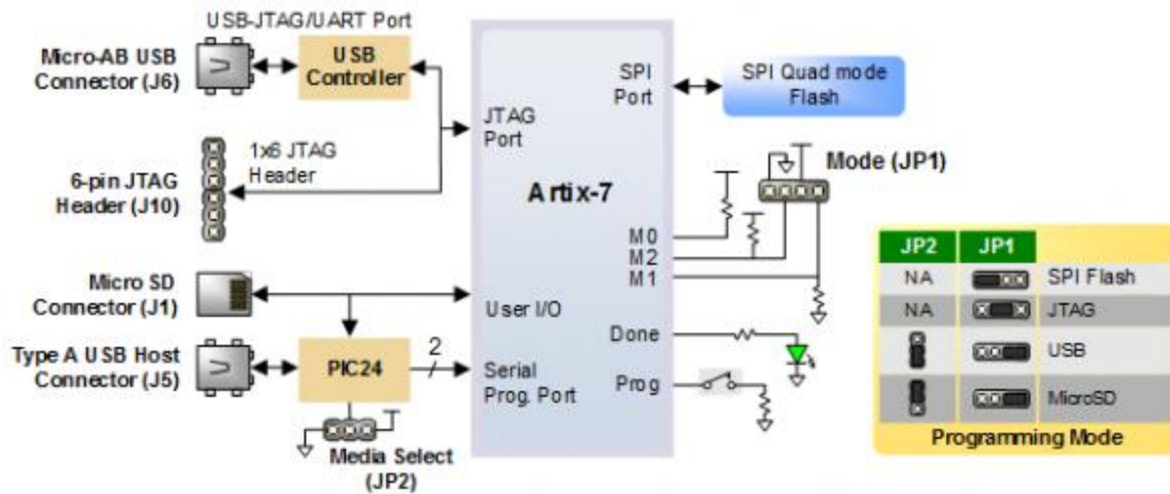


Image 2.3: Nexys A7 Configuration Options

Figure 2.3 shows the different options available for configuring the FPGA. An on-board “mode” jumper (JP1) and a media selection jumper (JP2) select between the programming modes.

The FPGA configuration data is stored in files called bitstreams that have the .bit file extension. The ISE or Vivado software from Xilinx can create bitstreams from VHDL, Verilog®, or schematic-based source files (in the ISE toolset, EDK is used for MicroBlaze™ embedded processor-based designs).

Bitstreams are stored in SRAM-based memory cells within the FPGA. This data defines the FPGA’s logic functions and circuit connections, and it remains valid until it is erased by removing board power, by pressing the reset button attached to the PROG input, or by writing a new configuration file using the JTAG port.

An Artix-7 100T bitstream is typically 30,606,304 bits and can take a long time to transfer. The time it takes to program the Nexys A7 can be decreased by compressing the bitstream before programming, and then allowing the FPGA to decompress the bitstream itself during configuration. Depending on design complexity, compression ratios of 10x can be achieved. Bitstream compression can be enabled within the Xilinx tools (ISE or Vivado) to occur during generation. For instructions on how to do this, consult the Xilinx documentation for the toolset being used. After being successfully programmed, the FPGA will cause the “DONE” LED to illuminate. Pressing the “PROG” button at any time will reset the configuration memory in the FPGA. After being reset, the FPGA will immediately attempt to reprogram itself from whatever method has been selected by the programming mode jumpers.

The following sections provide greater detail about programming the Nexys A7 using the different methods available.

2.5.1. JTAG Configuration

The Xilinx tools typically communicate with FPGAs using the Test Access Port and Boundary-Scan Architecture, commonly referred to as JTAG. During JTAG programming, a .bit file is transferred from the PC to the FPGA using the onboard Digilent USB-JTAG circuitry (port J6) or an external JTAG programmer, such as the Digilent JTAG-HS2, attached to port J10. You can perform JTAG programming

any time after the Nexys A7 has been powered on, regardless of what the mode jumper (JP1) is set to. If the FPGA is already configured, then the existing configuration is overwritten with the bitstream being transmitted over JTAG. Setting the mode jumper to the JTAG setting (seen in Figure 3) is useful to prevent the FPGA from being configured from any other bitstream source until a JTAG programming occurs.

Programming the Nexys A7 with an uncompressed bitstream using the on-board USB-JTAG circuitry usually takes around five seconds. JTAG programming can be done using the hardware server in Vivado or the iMPACT tool included with ISE and the Lab Tools version of Vivado. The demonstration project available at www.digilentinc.com gives an in-depth tutorial on how to program your board.

2.5.2. Quad-SPI Configuration

Since the FPGA on the Nexys A7 is volatile, it relies on the Quad-SPI flash memory to store the configuration between power cycles. This configuration mode is called Master SPI. The blank FPGA takes the role of master and reads the configuration file out of the flash device upon power-up. To that effect, a configuration file needs to be downloaded first to the flash. When programming a nonvolatile flash device, a bitstream file is transferred to the flash in a two-step process. First, the FPGA is programmed with a circuit that can program flash devices, and then data is transferred to the flash device via the FPGA circuit (this complexity is hidden from the user by the Xilinx tools). This is called indirect programming. After the flash device has been programmed, it can automatically configure the FPGA at a subsequent power-on or reset event as determined by the mode jumper setting. Programming files stored in the flash device will remain until they are overwritten, regardless of power-cycle events.

Programming the flash can take as long as four to five minutes, which is mostly due to the lengthy erase process inherent to the memory technology. Once written however, FPGA configuration can be very fast—less than a second. Bitstream compression, SPI bus width, and configuration rate are factors controlled by the Xilinx tools that can affect configuration speed. The Nexys A7 supports x1, x2, and x4 bus widths and data rates of up to 50 MHz for Quad-SPI programming.

Quad-SPI programming can be done using the iMPACT tool included with ISE or the Lab Tools version of Vivado.

2.5.3. USB Host and Micro SD Programming

You can program the FPGA from a pen drive attached to the USB Host port (J5) or a microSD card inserted into J1 by doing the following:

1. Format the storage device (Pen drive or microSD card) with a FAT32 file system.
2. Place a single .bit configuration file in the root directory of the storage device.
3. Attach the storage device to the Nexys A7.
4. Set the JP1 Programming Mode jumper on the Nexys A7 to “USB/SD”.
5. Select the desired storage device using JP2.

6. Push the PROG button or power-cycle the Nexys A7.

The FPGA will automatically configure with the .bit file on the selected storage device. Any .bit files that are not built for the proper Artix-7 device will be rejected by the FPGA.

The Auxiliary Function Status, or “BUSY” LED, gives visual feedback on the state of the configuration process when the FPGA is not yet programmed:

- When steadily lit, the auxiliary microcontroller is either booting up or currently reading the configuration medium (microSD or pen drive) and downloading a bitstream to the FPGA.
- A slow pulse means the microcontroller is waiting for a configuration medium to be plugged in.
- In case of an error during configuration, the LED will blink rapidly.

When the FPGA has been successfully configured, the behavior of the LED is application-specific. For example, if a USB keyboard is plugged in, a rapid blink will signal the receipt of an HID input report from the keyboard.

2.6. Memory

The Nexys A7 board contains two external memories: a 1Gib (128MiB) DDR2 SDRAM and a 128Mib (16MiB) non-volatile serial Flash device. The DDR2 modules are integrated on-board and connect to the FPGA using the industry standard interface. The serial Flash is on a dedicated quad-mode (x4) SPI bus. The connections and pin assignments between the FPGA and external memories are shown below.

2.6.1. DDR2

The Nexys A7 includes one Micron MT47H64M16HR-25:H DDR2 memory component, creating a single rank, 16-bit wide interface. It is routed to a 1.8V-powered HR (High Range) FPGA bank with 50 ohm controlled single-ended trace impedance. 50 Ohm internal terminations in the FPGA are used to match the trace characteristics. Similarly, on the memory side, on-die terminations (ODT) are used for impedance matching.

For proper operation of the memory, a memory controller and physical layer (PHY) interface needs to be included in the FPGA design. There are two recommended ways to do that, which are outlined below and differ in complexity and design flexibility.

The straightforward way is to use the Digilent-provided DDR-to-SRAM adapter module which instantiates the memory controller and uses an asynchronous SRAM bus for interfacing with user logic. This module provides backward compatibility with projects written for older Nexys-line boards featuring a CellularRAM instead of DDR2. It trades memory bandwidth for simplicity.

More advanced users or those who wish to learn more about DDR SDRAM technology may want to use the Xilinx 7-series memory interface solutions core generated by the MIG (Memory Interface Generator) Wizard. Depending on the tool used (ISE, EDK or Vivado), the MIG Wizard can generate a native FIFO-style or an AXI4 interface to connect to user logic. This workflow allows the customization of several DDR parameters optimized for the particular application. Table 2.5 below lists the MIG Wizard settings optimized for the Nexys A7.

Setting	**Value**
Memory type	DDR2 SDRAM
Max. clock period	3000ps (667Mbps data rate)
Recommended clock period (for easy clock generation)	3077ps (650Mbps data rate)
Memory part	MT47H64M16HR-25E
Data width	16
Data mask	Enabled
Chip Select pin	Enabled
Rtt (nominal) – On-die termination	50ohms
Internal Vref	Enabled
Internal termination impedance	50ohms

Table 2.2: DDR2 settings for the Nexys A7.

Although the FPGA, memory IC, and the board itself are capable of the maximum data rate of 667Mbps, the limitations in the clock generation primitives restrict the clock frequencies that can be generated from the 100 MHz system clock. Thus, for simplicity, the next highest data rate of 650Mbps is recommended.

The MIG Wizard will require the fixed pin-out of the memory signals to be entered and validated before generating the IP core. For your convenience, an importable UCF file is provided on the Digilent website to speed up the process.

For more details on the Xilinx memory interface solutions, refer to the 7 Series FPGAs Memory Interface Solutions User Guide (ug586).

2.6.2. Quad-SPI Flash

FPGA configuration files can be written to the Quad-SPI Flash (Spansion part number S25FL128S), and mode settings are available to cause the FPGA to automatically read a configuration from this device at power on. An Artix-7 100T configuration file requires just less than four MiB (mebibyte) of memory, leaving about 77% of the flash device available for user data. Or, if the FPGA is getting configured from another source, the whole memory can be used for custom data.

The contents of the memory can be manipulated by issuing certain commands on the SPI bus. The implementation of this protocol is outside the scope of this document. All signals in the SPI bus except SCK are general-purpose user I/O pins after FPGA configuration. SCK is an exception because it remains a dedicated pin even after configuration. Access to this pin is provided through a special FPGA primitive called STARTUPE2.

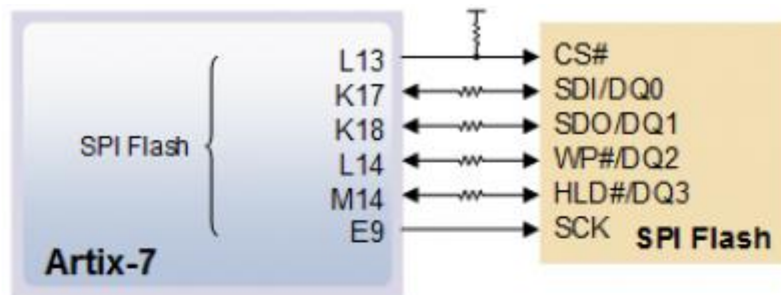


Image 2.4: Nexys A7 SPI Flash Pin-out

2.7. Ethernet PHY

The Nexys A7 board includes an SMSC 10/100 Ethernet PHY (SMSC part number LAN8720A) paired with an RJ-45 Ethernet jack with integrated magnetics. The SMSC PHY uses the RMII interface and supports 10/100 Mb/s. Figure 4.1 illustrates the pin connections between the Artix-7 and the Ethernet PHY. At power-on reset, the PHY is set to the following defaults:

- RMII mode interface
- Auto-negotiation enabled, advertising all 10/100 mode capable
- PHY address=00001

Two on-board LEDs (LD23 = LED2, LD24 = LED1) connected to the PHY provide link status and data activity feedback. See the PHY datasheet for details.

EDK-based designs can access the PHY using either the axi_ethernetlite (AXI EthernetLite) IP core or the axi_ethernet (Tri Mode Ethernet MAC) IP core. A mii_to_rmii core (Ethernet PHY MII to Reduced MII) needs to be inserted to convert the MAC interface from MII to RMII. Also, a 50 MHz clock needs to be generated for the mii_to_rmii core and the CLKIN pin of the external PHY. To account for skew introduced by the mii_to_rmii core, generate each clock individually, with the external PHY clock having a 45 degree phase shift relative to the mii_to_rmii Ref_Clk. An EDK demonstration project that properly uses the Ethernet PHY can be found on the Nexys A7 product page at www.digilentinc.com.

ISE designs can use the IP Core Generator wizard to create an Ethernet MAC controller IP core.

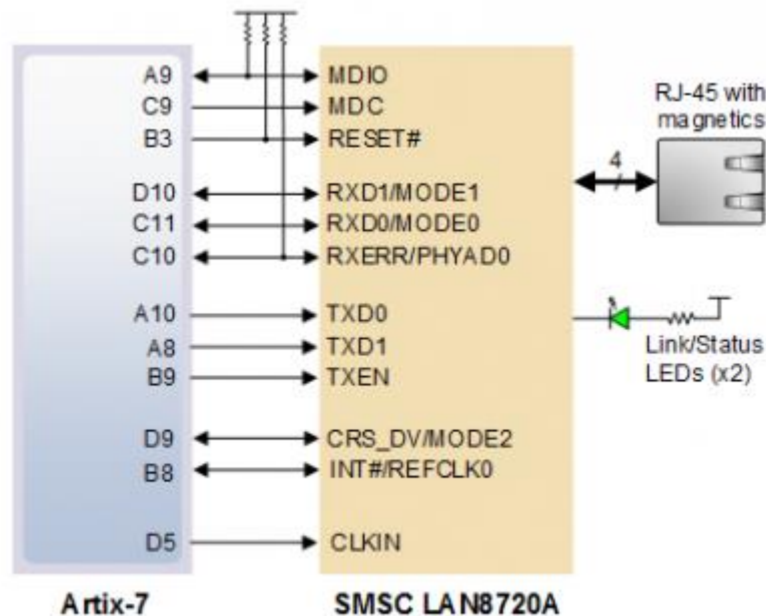


Image 2.5: Pin Connections between the Artix-7 and the Ethernet PHY

2.8. Oscillators/Clocks

The Nexys A7 board includes a single 100 MHz crystal oscillator connected to pin E3 (E3 is a MRCC input on bank 35). The input clock can drive MMCMs or PLLs to generate clocks of various frequencies and with known phase relationships that may be needed throughout a design. Some rules restrict which MMCMs and PLLs may be driven by the 100 MHz input clock. For a full description of these rules and of the capabilities of the Artix-7 clocking resources, refer to the “7 Series FPGAs Clocking Resources User Guide” available from Xilinx.

Xilinx offers the Clocking Wizard IP core to help users generate the different clocks required for a specific design. This wizard will properly instantiate the needed MMCMs and PLLs based on the desired frequencies and phase relationships specified by the user. The wizard will then output an easy-to-use wrapper component around these clocking resources that can be inserted into the user’s design. The clocking wizard can be accessed from within the Project Navigator or Core Generator tools.

2.9. USB-UART Bridge (Serial Port)

The Nexys A7 includes an FTDI FT2232HQ USB-UART bridge (attached to connector J6) that allows you use PC applications to communicate with the board using standard Windows COM port commands. Free USB-COM port drivers, available from www.ftdichip.com under the “Virtual Com Port” or VCP heading, convert USB packets to UART/serial port data. Serial port data is exchanged with the FPGA using a two-wire serial port (TXD/RXD) and optional hardware flow control (RTS/CTS). After the drivers are installed, I/O commands can be used from the PC directed to the COM port to produce serial data traffic on the C4 and D4 FPGA pins.

Two on-board status LEDs provide visual feedback on traffic flowing through the port: the transmit LED (LD20) and the receive LED (LD19). Signal names that imply direction are from the point-of-view of the DTE (Data Terminal Equipment), in this case the PC.

The FT2232HQ is also used as the controller for the Digilent USB-JTAG circuitry, but the USB-UART and USB-JTAG functions behave entirely independent of one another. Programmers interested in using the UART functionality of the FT2232 within their design do not need to worry about the JTAG circuitry interfering with the UART data transfers, and vice-versa. The combination of these two features into a single device allows the Nexys A7 to be programmed, communicated with via UART, and powered from a computer attached with a single Micro USB cable.

The connections between the FT2232HQ and the Artix-7 are shown in Figure 2.6.

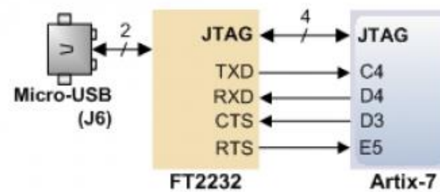


Image 2.6: Nexys A7 FT2322HQ Connections

2.10. USB HID Host

The Auxiliary Function microcontroller (Microchip PIC24FJ128) provides the Nexys A7 with USB Embedded HID host capability. After power-up, the microcontroller is in configuration mode, either downloading a bitstream to the FPGA, or waiting to be programmed from other sources. Once the FPGA is programmed, the microcontroller switches to application mode, which is USB HID Host in this case. Firmware in the microcontroller can drive a mouse or a keyboard attached to the type A USB connector at J5 labeled “USB Host”. Hub support is not currently available, so only a single mouse or a single keyboard can be used. Only keyboards and mice supporting the Boot HID interface are supported. The PIC24 drives several signals into the FPGA – two are used to implement a standard PS/2 interface for communication with a mouse or keyboard, and the others are connected to the FPGA’s two-wire serial programming port, so the FPGA can be programmed from a file stored on a USB pen drive or microSD card.

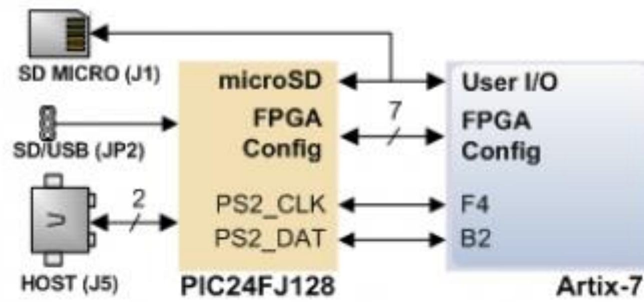
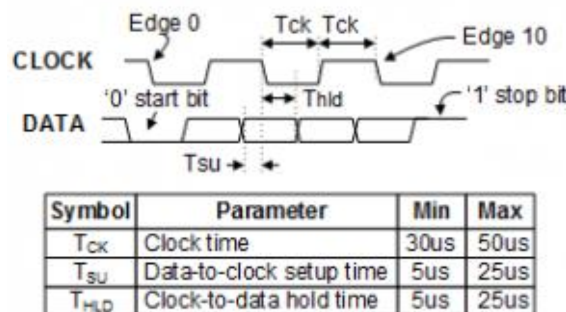


Image 2.7: Nexys A7 PIC24 Connections

2.10.1. HID Controller

The Auxiliary Function microcontroller hides the USB HID protocol from the FPGA and emulates an old-style PS/2 bus. The microcontroller behaves just like a PS/2 keyboard or mouse would. This means new designs can re-use existing PS/2 IP cores. Mice and keyboards that use the PS/2 protocol use a two-wire serial bus (clock and data) to communicate with a host. On the Nexys A7, the microcontroller emulates a PS/2 device while the FPGA plays the role of the host. Both the mouse and the keyboard use 11-bit words that include a start bit, data byte (LSB first), odd parity, and stop bit, but the data packets are organized differently, and the keyboard interface allows bi-directional data transfers (so the host device can illuminate state LEDs on the keyboard). Bus timings are shown in Figure 2.8.



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

Image 2.8: PS/2 Device-to-Host Timing Diagram

The clock and data signals are only driven when data transfers occur; otherwise, they are held in the idle state at high-impedance (open-drain drivers). This requires that when the PS/2 signals are used in a design, internal pull-ups must be enabled in the FPGA on the data and clock pins. The clock signal is normally driven by the device, but may be held low by the host in special cases. The timings define signal requirements for mouse-to-host communications and bi-directional keyboard communications. A PS/2 interface circuit can be implemented in the FPGA to create a keyboard or mouse interface.

When a keyboard or mouse is connected to the Nexys A7, a “self-test passed” command (0xAA) is sent to the host. After this, commands may be issued to the device. Since both the keyboard and the mouse use the same PS/2 port, one can tell the type of device connected using the device ID. This ID can be read by issuing a Read ID command (0xF2). Also, a mouse sends its ID (0x00) right after the “self-test passed” command, which distinguishes it from a keyboard.

2.10.2. Keyboard

PS/2-style keyboards use scan codes to communicate key press data. Each key is assigned a code that is sent whenever the key is pressed. If the key is held down, the scan code will be sent repeatedly about once every 100ms. When a key is released, an F0 key-up code is sent, followed by the scan code of the released key. If a key can be shifted to produce a new character (like a capital letter), then a shift character is sent in addition to the scan code and the host must determine which ASCII character to use. Some keys, called extended keys, send an E0 ahead of the scan code (and they may send more than one scan code). When an extended key is released, an E0 F0 key-up code is sent, followed by the scan code. Scan codes for most keys are shown in Figure 2.9.

Image 2.9: Keyboard Scan Codes

A host device can also send data to the keyboard. Table 2.3 shows a list of some common commands a host might send.

The keyboard can send data to the host only when both the data and clock lines are high (or idle). Because the host is the bus master, the keyboard must check to see whether the host is sending data before driving the bus. To facilitate this, the clock line is used as a “clear to send” signal. If the host drives the clock line low, the keyboard must not send any data until the clock is released. The keyboard sends data to the host in 11-bit words that contain a ‘0’ start bit, followed by 8-bits of scan code (LSB first), followed by an odd parity bit, and terminated with a ‘1’ stop bit. The keyboard generates 11 clock transitions (at 20 to 30 KHz) when the data is sent, and data is valid on the falling edge of the clock.

Command	**Action**
ED	Set Num Lock, Caps Lock, and Scroll Lock LEDs. Keyboard returns FA after receiving ED, then host sends a byte to set LED status: bit 0 sets Scroll Lock, bit 1 sets Num Lock, and bit 2 sets Caps lock. Bits 3 to 7 are ignored.

EE	Echo (test). Keyboard returns EE after receiving EE
F3	Set scan code repeat rate. Keyboard returns F3 on receiving FA, then host sends second byte to set the repeat rate.
FE	Resend. FE directs keyboard to re-send most recent scan code.
FF	Reset. Resets the keyboard.

Table 2.3: Keyboard Commands

2.10.3. Mouse

Once entered in stream mode and data reporting is enabled, the mouse outputs a clock and data signal when it is moved; otherwise, these signals remain at logic '1.' Each time the mouse is moved, three 11-bit words are sent from the mouse to the host device, as shown in Figure 2.10. Each of the 11-bit words contains a '0' start bit, followed by 8 bits of data (LSB first), followed by an odd parity bit, and terminated with a '1' stop bit. Thus, each data transmission contains 33 bits, where bits 0, 11, and 22 are '0' start bits, and bits 11, 21, and 33 are '1' stop bits. The three 8-bit data fields contain movement data, as shown in Figure 2.10. Data is valid at the falling edge of the clock, and the clock period is 20 to 30 KHz.

The mouse assumes a relative coordinate system wherein moving the mouse to the right generates a positive number in the X field, and moving to the left generates a negative number. Likewise, moving the mouse up generates a positive number in the Y field, and moving down represents a negative number (the XS and YS bits in the status byte are the sign bits – a '1' indicates a negative number). The magnitude of the X and Y numbers represent the rate of mouse movement; the larger the number, the faster the mouse is moving (the XV and YV bits in the status byte are movement overflow indicators. A '1' means overflow has occurred). If the mouse moves continuously, the 33-bit transmissions are repeated every 50ms or so. The L and R fields in the status byte indicate Left and Right button presses (a '1' indicates the button is being pressed).

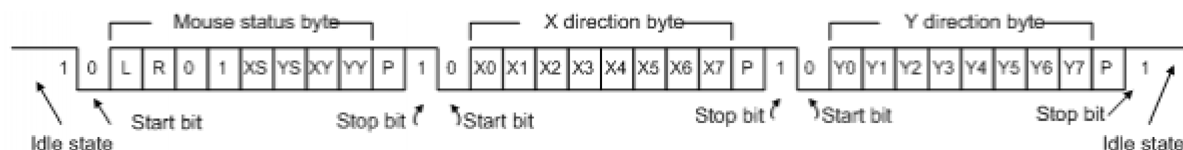


Image 2.10: Mouse Data Format

2.11. VGA Port

The Nexys A7 board uses 14 FPGA signals to create a VGA port with 4 bits-per-color and the two standard sync signals (HS – Horizontal Sync, and VS – Vertical Sync). The color signals use resistor-divider circuits that work in conjunction with the 75-ohm termination resistance of the VGA display to create 16 signal levels each on the red, green, and blue VGA signals. This circuit, shown in Figure 8.1, produces video color signals that proceed in equal increments between 0V (fully off) and 0.7V (fully on). Using this circuit, 4096 different colors can be displayed, one for each unique 12-bit pattern. A video controller circuit must be created in the FPGA to drive the sync and color signals with the correct timing in order to produce a working display system.

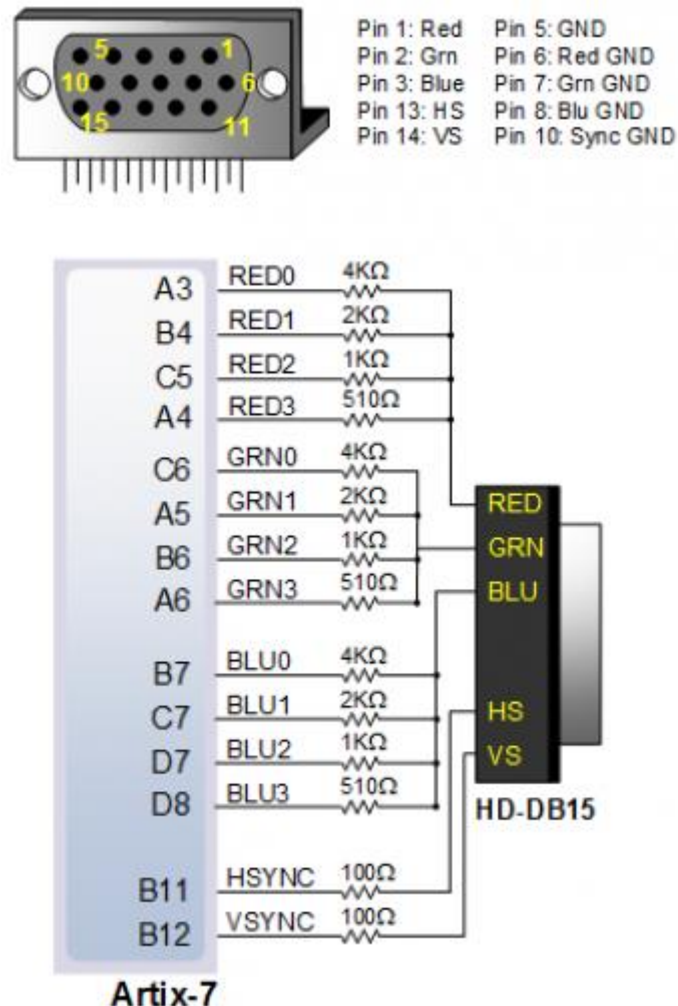


Image 2.11: Nexys A7 VGA Interface

2.12. Basic I/O

The Nexys A7 board includes two tri-color LEDs, sixteen slide switches, six push buttons, sixteen individual LEDs, and an eight-digit seven-segment display, as shown in Figure 2.12. The pushbuttons and slide switches are connected to the FPGA via series resistors to prevent damage from inadvertent short circuits (a short circuit could occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output). The five pushbuttons arranged in a plus-sign configuration are “momentary” switches that normally generate a low output when they are at rest, and a high output only when they are pressed. The red pushbutton labeled “CPU RESET,” on the other hand, generates a high output when at rest and a low output when pressed. The CPU RESET button is intended to be used in EDK designs to reset the processor, but you can also use it as a general purpose pushbutton. Slide switches generate constant high or low inputs depending on their position.

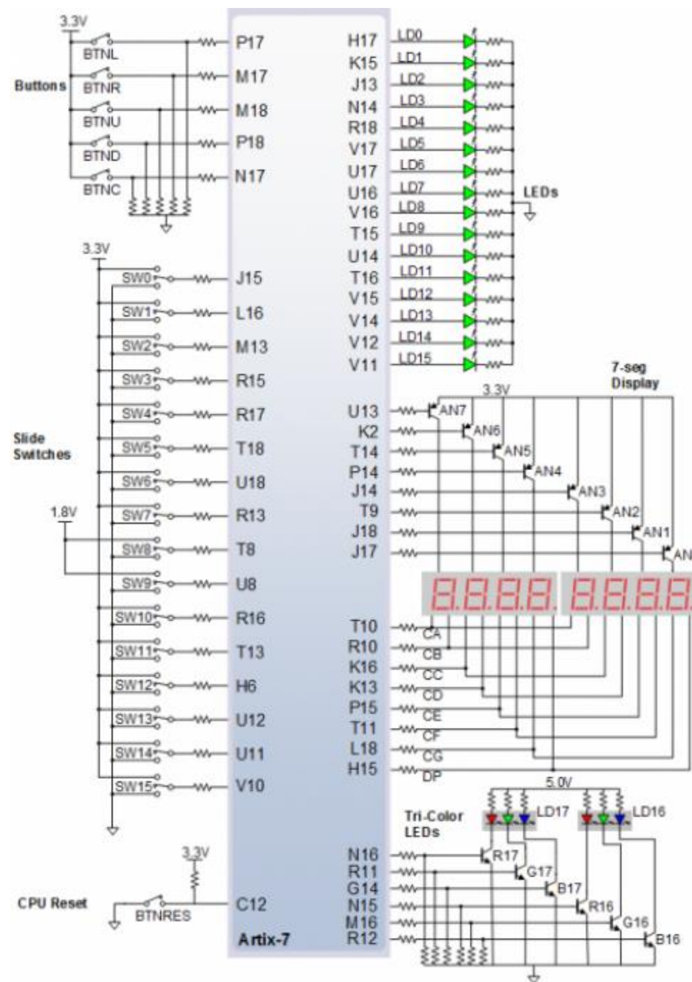


Image 2.12: General Purpose I/O Devices on the Nexys A7

The sixteen individual high-efficiency LEDs are anode-connected to the FPGA via 330-ohm resistors, so they will turn on when a logic high voltage is applied to their respective I/O pin. Additional LEDs that are not user-accessible indicate power-on, FPGA programming status, and USB and Ethernet port status.

2.12.1. Seven-Segment Display

The Nexys A7 board contains two four-digit common anode seven-segment LED displays, configured to behave like a single eight-digit display. Each of the eight digits is composed of seven segments arranged in a "figure 8" pattern, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark, as shown in Figure 2.13. Of these 128 possible patterns, the ten corresponding to the decimal digits are the most useful.

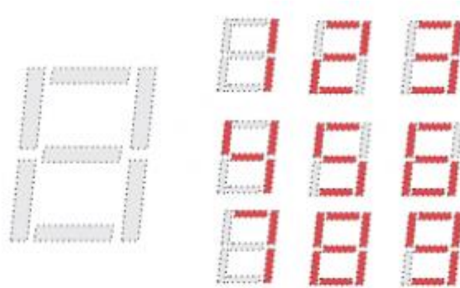


Image 2.13: An Un-illuminated Seven-Segment Display and Nine Illumination Patterns Corresponding to Decimal Digits

The anodes of the seven LEDs forming each digit are tied together into one "common anode" circuit node, but the LED cathodes remain separate, as shown in Fig 2.14. The common anode signals are available as eight "digit enable" input signals to the 8-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG. For example, the eight "D" cathodes from the eight digits are grouped together into a single circuit node called "CD." These seven cathode signals are available as inputs to the 8-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.

To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Nexys A7 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..7 and the CA..G/DP signals are driven low when active.

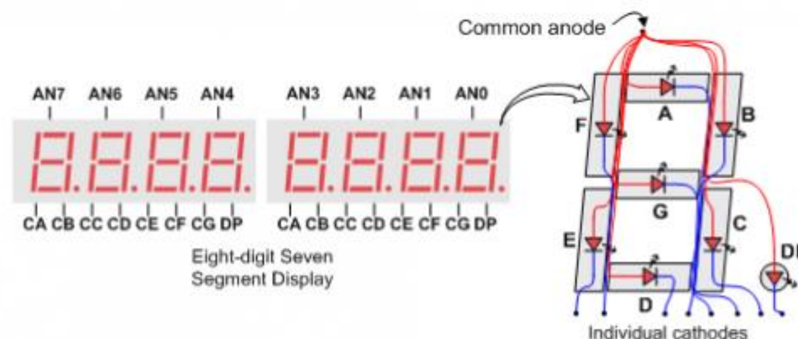


Image 2.14: Common Anode Circuit Node

A scanning display controller circuit can be used to show an eight-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-eighth of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update, or “refresh”, rate is slowed to around 45Hz, a flicker can be noticed in the display.

For each of the four digits to appear bright and continuously illuminated, all eight digits should be driven once every 1 to 16ms, for a refresh frequency of about 1 KHz to 60Hz. For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/8 of the refresh cycle, or 2ms. The controller must drive low the cathodes with the correct pattern when the corresponding anode signal is driven high. To illustrate the process, if AN0 is asserted while CB and CC are asserted, then a “1” will be displayed in digit position 1. Then, if AN1 is asserted while CA, CB, and CC are asserted, a “7” will be displayed in digit position 2. If AN0, CB, and CC are driven for 4ms, and then AN1, CA, CB, and CC are driven for 4ms in an endless succession, the display will show “71” in the first two digits. An example timing diagram for a four-digit controller is shown in Figure 2.15.

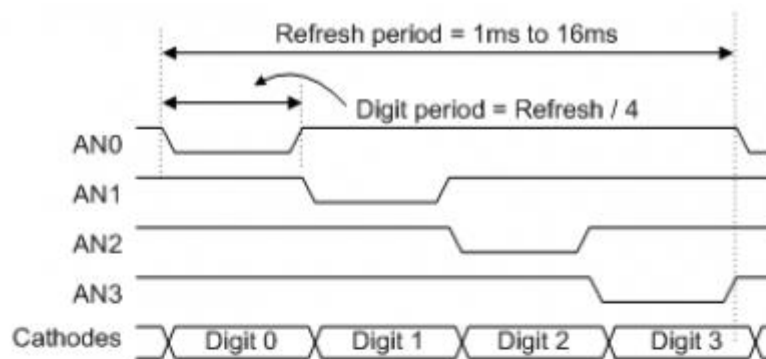


Image 2.15: Four Digit Scanning Display Controller Timing Diagram

2.12.2. Tri-Color LED

The Nexys A7 board contains two tri-color LEDs. Each tri-color LED has three input signals that drive the cathodes of three smaller internal LEDs: one red, one blue, and one green. Driving the signal corresponding to one of these colors high will illuminate the internal LED. The input signals are driven by the FPGA through a transistor, which inverts the signals. Therefore, to light up the tri-color LED, the corresponding signals need to be driven high. The tri-color LED will emit a color dependent on the combination of internal LEDs that are currently being illuminated. For example, if the red and blue signals are driven high, and green is driven low, the tri-color LED will emit a purple color.

Note: Digilent strongly recommends the use of pulse-width modulation (PWM) when driving the tri-color LEDs. Driving any of the inputs to a steady logic ‘1’ will result in the LED being illuminated at an uncomfortably bright level. You can avoid this by ensuring that none of the tri-color signals are driven

with more than a 50% duty cycle. Using PWM also greatly expands the potential color palette of the tri-color led. Individually adjusting the duty cycle of each color between 50% and 0% causes the different colors to be illuminated at different intensities, allowing virtually any color to be displayed.

2.13. Pmod Ports

The Pmod ports are arranged in a 2×6 right-angle, and are 100-mil female connectors that mate with standard 2×6 pin headers. Each 12-pin Pmod port provides two 3.3V VCC signals (pins 6 and 12), two Ground signals (pins 5 and 11), and eight logic signals, as shown in Figure 10.1. The VCC and Ground pins can deliver up to 1A of current. Pmod data signals are not matched pairs, and they are routed using best-available tracks without impedance control or delay matching. Pin assignments for the Pmod I/O connected to the FPGA are shown in Table 2.4.

Image 2.16: Pmod Connectors; Front View, as Loaded on PCB

Pmod JA	Pmod JB	Pmod JC	Pmod JD	Pmod XDAC
JA1: C17	JB1: D14	JC1: K1	JD1: H4	JXADC1: A13 (AD3P)
JA2: D18	JB2:F16	JC2: F6	JD2: H1	JXADC2: A15 (AD10P)
JA3: E18	JB3: G16	JC3:J2	JD3: G1	JXADC3: B16 (AD2P)
JA4: G17	JB4: H14	IC4: G6	JD4: G3	JXADC4: B18 (AD11P)
JA7: D17	JB7:E16	JC7: E7	JD7: H2	JXADC7: A14 (AD3N)
JA8: E17	JB8: F13	JC8: J3	JD8: G4	JXADC8: A16 (AD10N)
JA9: F18	JB9: G13	JC9: J4	JD9: G2	JXADC9: B17 (AD2N)
JA10: G18	JB10: H16	JC10: E6	JD10: F3	JXADC10: A18 (AD11N)

Table 2.4: Nexys A7 Pmod pin assignments.

2.13.1. Dual Analog/Digital Pmod

The on-board Pmod expansion connector labeled "JXADC" is wired to the auxiliary analog input pins of the FPGA. Depending on the configuration, this connector can be used to input differential analog signals to the analog-to-digital converter inside of the Artix-7 (XADC). Any or all pairs in the connector can be configured either as analog input or digital input-output.

The Dual Analog/Digital Pmod on the Nexys A7 differs from the rest in the routing of its traces. The eight data signals are grouped into four pairs, with the pairs routed closely coupled for better analog noise immunity. Furthermore, each pair has a partially loaded anti-alias filter laid out on the PCB. The filter does not have capacitors C60-C63. In designs where such filters are desired, the capacitors can be manually loaded by the user.

The XADC core within the Artix-7 is a dual channel 12-bit analog-to-digital converter capable of operating at 1 MSPS. Either channel can be driven by any of the auxiliary analog input pairs connected to the JXADC header. The XADC core is controlled and accessed from a user design via the Dynamic Reconfiguration Port (DRP). The DRP also provides access to voltage monitors that are present on each of the FPGA's power rails, and a temperature sensor that is internal to the FPGA. For more information on using the XADC core, refer to the Xilinx document titled "7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter."

2.14. MicroSD Slot

The Nexys A7 provides a microSD slot for both FPGA configuration and user access. The on-board Auxiliary Function microcontroller shares the SD card bus with the FPGA. Before the FPGA is configured the microcontroller must have access to the SD card via SPI. Once a bit file is downloaded to the FPGA (from any source), the microcontroller power cycles the SD slot and relinquishes control of the bus. This enables any SD card in the slot to reset its internal state machines and boot up in SD native bus mode. All of the SD pins on the FPGA are wired to support full SD speeds in native interface mode, as shown in Figure 2.17. The SPI is also available, if needed. Once control over the SD bus is passed from the microcontroller to the FPGA, the SD_RESET signal needs to be actively driven low by the FPGA to power the microSD card slot. For information on implementing an SD card controller, refer to the SD card specification available at www.sdcard.org.

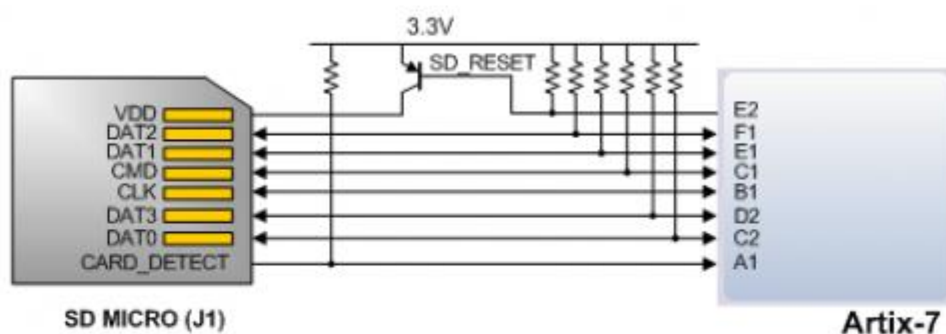


Image 2.17: Artix-7 microSD Card Connector Interface

2.15. Temperature Sensor

The Nexys A7 includes an Analog Device ADT7420 temperature sensor. The sensor provides up to 16-bit resolution with a typical accuracy better than 0.25 degrees Celsius. The interface between the temperature sensor and FPGA is shown in Figure 2.18.

Image 2.18: Temperature Sensor Interface

2.15.1. I²C Interface

The ADT7420 chip acts as a slave device using the industry standard I²C communication scheme. To communicate with ADT7420 chip, the I²C master must specify a slave address (0x4B) and a flag indicating whether the communication is a read (1) or a write (0). Once specifications are made for communication, a data transfer takes place. For ADT7420, the data transfer should consist of the address of the desired device register followed by the data to be written to the specified register. To read from a register, the master must write the desired register address to the ADT7420, then send an I²C restart condition, and send a new read request to the ADT7420. If the master does not generate a restart condition prior to attempting the read, the value written to the address register will be reset to 0x00. As some registers store 16-bit values as 8-bit register pairs, the ADT7420 will automatically increment the address register of the device when accessing certain registers, such as the temperature registers and the threshold registers. This allows for the master to use a single read or write request to access both the low and high bytes of these registers. A complete listing of registers and their behavior can be found in the ADT7420 datasheet available on the Analog Devices website.

2.16. Accelerometer

The Nexys A7 includes an Analog Device ADXL362 accelerometer. The ADXL362 is a 3-axis MEMS accelerometer that consumes less than 2μA at a 100Hz output data rate and 270nA when in motion triggered wake-up mode. Unlike accelerometers that use power duty cycling to achieve low power consumption, the ADXL362 does not alias input signals by under-sampling; it samples the full bandwidth of the sensor at all data rates. The ADXL362 always provides 12-bit output resolution; 8-bit formatted data is also provided for more efficient single-byte transfers when a lower resolution is sufficient. Measurement ranges of ±2 g, ±4 g, and ±8 g are available with a resolution of 1 mg/LSB on the ±2 g range. The FPGA can talk with the ADXL362 via SPI interface. While the ADXL362 is in Measurement Mode, it continuously measures and stores acceleration data in the X-data, Y-data, and Z-data registers. The interface between the FPGA and accelerometer can be seen in Figure 2.19.

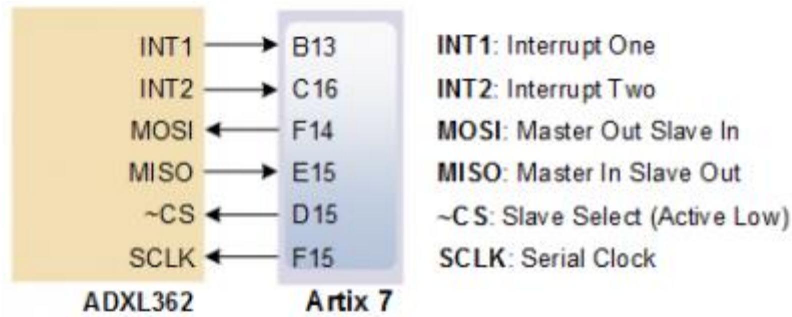


Image 2.19: Accelerometer Interface

2.16.1. SPI Interface

The ADXL362 acts as a slave device using an SPI communication scheme. The recommended SPI clock frequency ranges from 1 MHz to 5 MHz. The SPI operates in SPI mode 0 with CPOL = 0 and CPHA = 0. All communications with the device must specify a register address and a flag that indicate whether the communication is a read or a write. Actual data transfer always follows the register address and communication flag. Device configuration can be performed by writing to the control registers within the accelerometer. Access accelerometer data by reading the device registers.

2.16.2. Interrupts

Several of the built-in functions of the ADXL362 can trigger interrupts that alert the host processor of certain status conditions. Interrupts can be mapped to either (or both) of two interrupt pins (INT1, INT2). Both of these pins require internal FPGA pull-ups when used. For more details about the interrupts, see the ADXL362 datasheet.

2.17. Microphone

The Nexys A7 board includes an omnidirectional MEMS microphone. The microphone uses an Analog Device ADMP421 chip which has a high signal to noise ratio (SNR) of 61dBA and high sensitivity of -26 dBFS. It also has a flat frequency response ranging from 100Hz to 15 kHz. The digitized audio is output in the pulse density modulated (PDM) format. The component architecture is shown in Figure 2.20.

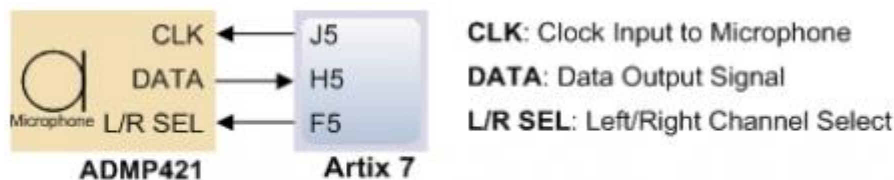


Image 2.20: Microphone Block Diagram

2.18. Built-In Self-Test

A demonstration configuration is loaded into the Quad-SPI flash device on the Nexys A7 board during manufacturing. The source code and prebuilt bitstream for this design are available for download from

the Digilent website. If the demo configuration is present in the flash and the Nexys A7 board is powered on in SPI mode, the demo project will allow basic hardware verification. Here is an overview of how this demo drives the different onboard components:

- The user LEDs are illuminated when the corresponding user switch is placed in the on position.
- The tri-color LEDs are controlled by some of the user buttons. Pressing BTNL, BTNC, or BTNR causes them to illuminate either red, green, or blue, respectively. Pressing BTND causes them to begin cycling through many colors. Repeatedly pressing BTND will turn the two LEDs on or off.
- Pressing BTNU will trigger a 5 second recording from the onboard PDM microphone. This recording is then immediately played back on the mono audio out port. The status of the recording and playback is displayed on the user LEDs. The recording is saved in the DDR2 memory.
- The VGA port displays feedback from the onboard microphone, temperature sensors, accelerometer, RGB LEDs, and USB Mouse.
- Connecting a mouse to the USB-HID Mouse port will allow the pointer on the VGA display to be controlled. Only mice compatible with the Boot Mouse HID interface are supported.
- The seven-segment display will display a moving snake pattern.[1]

3. VHDL and the Xilinx Vivado Design Suite

3.1. Introduction

The VHSIC Hardware Description Language (VHDL) is a hardware description language (HDL) that can model the behavior and structure of digital systems at multiple levels of abstraction, ranging from the system level down to that of logic gates, for design entry, documentation, and verification purposes.

VHDL is named after the United States Department of Defense program that created it, the Very High-Speed Integrated Circuits Program (VHSIC). In the early 1980s, the VHSIC Program sought a new HDL for use in the design of the integrated circuits it aimed to develop. The product of this effort was VHDL Version 7.2, released in 1985. The effort to standardize it as an IEEE standard began in the following year.[2]

3.2. Different levels of representation and abstraction

A digital system can be represented at different levels of abstraction [1]. This keeps the description and design of complex systems manageable. Figure 3.1 shows different levels of abstraction.

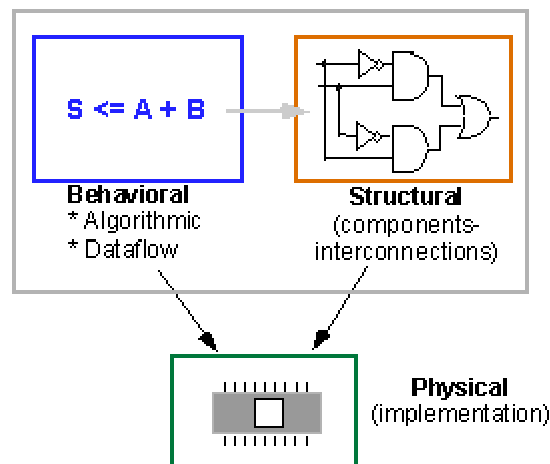


Image 3.1: Different levels of abstraction: Behavioral, Structural and Physical

The highest level of abstraction is the behavioral level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level. As an example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

The structural level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is

usually closer to the physical realization of a system. For the example above, the structural representation is shown in Figure 3.2 below.

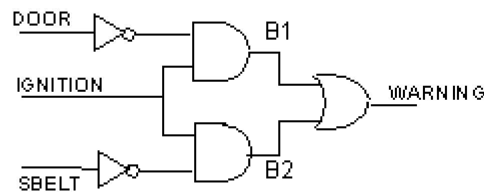


Image 3.2: Structural representation of a "buzzer" circuit.

VHDL allows one to describe a digital system at the structural or the behavioral level. The behavioral level can be further divided into two kinds of styles: Data flow and Algorithmic. The dataflow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer level). The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input. On the other hand, sequential statements are executed in the sequence that they are specified. VHDL allows both concurrent and sequential signal assignments that will determine the manner in which they are executed. Examples of both representations will be given later.

3.3. Basic Structure of a VHDL file

A digital system in VHDL consists of a design entity that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an *entity declaration* and an *architecture body*. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Figure 3.3 below. In a typical design there will be many such entities connected together to perform the desired function.

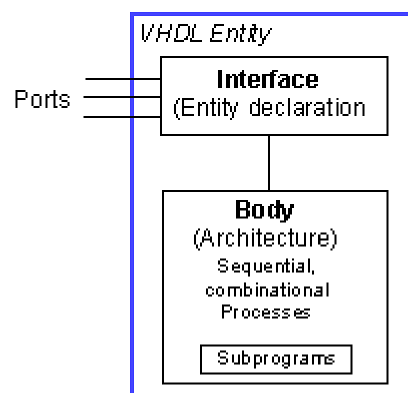


Image 3.3: A VHDL entity consisting of an interface (entity declaration) and a body (architectural description).

VHDL uses reserved keywords that cannot be used as signal names or identifiers. Keywords and user-defined identifiers are case insensitive.
Lines with comments start with two adjacent hyphens (--) and will be ignored by the compiler. VHDL also ignores line breaks and extra spaces. VHDL is a strongly typed language which implies that one has always to declare the type of every object that can have a value, such as signals, constants and variables.

The entity declaration defines the NAME of the entity and lists the input and output ports. The general form is as follows,

```
entity NAME_OF_ENTITY is [ generic generic_declarations);]
     port (signal_names: mode type;
            signal_names: mode type;
                :
            signal_names: mode type);
end [NAME_OF_ENTITY] ;
```

a) Entity Declaration

An entity always starts with the keyword *entity*, followed by its name and the keyword *is*. Next are the port declarations using the keyword *port*. An entity declaration always ends with the keyword *end*, optionally followed by the name of the entity.

- The NAME_OF_ENTITY is a user-selected identifier
- signal_names consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.
- **mode**: is one of the reserved words to indicate the signal direction:
  - **in** – indicates that the signal is an input
  - **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.
  - **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity's architecture
  - **inout** – the signal can be an input or an output.
- *type*: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.
  - *bit* – can have the value 0 and 1
  - *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7)
  - *std_logic*, *std_ulogic*, *std_logic_vector*, *std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
  - *boolean* – can have the value TRUE and FALSE
  - *integer* – can have a range of integer values

  - *real* – can have a range of real values
  - *character* – any printing character
  - *time* – to indicate time
- **generic**: generic declarations are optional and determine the local constants used for timing and sizing (e.g. bus widths) the entity. A generic can have a default value. The syntax for a generic follows,

```
generic (
constant_name: type [:=value] ;
constant_name: type [:=value] ;
:
constant_name: type [:=value] );
```

For the example of Figure 3.2 above, the entity declaration looks as follows.

```
-- comments: example of the buzzer circuit of fig. 2
entity BUZZER is
     port (DOOR, IGNITION, SBELT: in std_logic;
      WARNING: out std_logic);
end BUZZER;
```

The entity is called BUZZER and has three input ports, DOOR, IGNITION and SBELT and one output port, WARNING. Notice the use and placement of semicolons! The name BUZZER is an identifier. Inputs are denoted by the keyword in, and outputs by the keyword out. Since VHDL is a strongly typed language, each port has a defined type. In this case, we specified the std_logic type. This is the preferred type of digital signals. In contrast to the bit type that can only have the values '1' and '0', the std_logic and std_ulogic types can have nine values. This is important to describe a digital system accurately including the binary values 0 and 1, as well as the unknown value X, the uninitialized value U, "-" for don't care, Z for high impedance, and several symbols to indicate the signal strength (e.g. L for weak 0, H for weak 1, W for weak unknown - see section on Enumerated Types). The std_logic type is defined in the std_logic_1164 package of the IEEE library. The type defines the set of values an object can have. This has the advantage that it helps with the creation of models and helps reduce errors. For instance, if one tries to assign an illegal value to an object, the compiler will flag the error.

b) Architecture body

The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

The architecture body looks as follows,

```
architecture architecture_name of NAME_OF_ENTITY is
-- Declarations
      -- components declarations
      -- signal declarations
      -- constant declarations
      -- function declarations
      -- procedure declarations
      -- type declarations

           :

begin
-- Statements

    :

end architecture_name;
```

The architecture body for the example of Figure 3.2, described at the behavioral level, is given below,

```
Architecture behavioral  of BUZZER is
begin
  WARNING <=(not DOOR and IGNITION) or (not SBELT and IGNITION);
end behavioral;
```

The header line of the architecture body defines the architecture name, e.g. *behavioral*, and associates it with the entity, BUZZER. The architecture name can be any legal identifier. The main body of the architecture starts with the keyword begin and gives the Boolean expression of the function. We will see later that a behavioral model can be described in several other ways. The “<= ” symbol represents an assignment operator and assigns the value of the expression on the right to the signal on the left. The architecture body ends with an end keyword followed by the architecture name.

c) Library and Packages: **library** and **use** keywords

A library can be considered as a place where the compiler stores information about a design project. A VHDL package is a file or module that contains declarations of commonly used objects, data type, component declarations, signal, procedures and functions that can be shared among different VHDL models.

Mentioned earlier, std_logic is defined in the package ieee.std_logic_1164 in the ieee library. In order to use the std_logic one needs to specify the library and package. This is done at the beginning of the VHDL file using the library and the use keywords as follows:

```
library  ieee;
use  ieee.std_logic_1164.all;
```

The .all extension indicates to use all of the ieee.std_logic_1164 package.

The Xilinx Foundation Express comes with several packages.

ieee Library:

- std_logic_1164 package: defines the standard datatypes
- std_logic_arith package: provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std_ulogic, std_logic and std_logic_vector types
- std_logic_unsigned
- std_logic_misc package: defines supplemental types, subtypes, constants and functions for the std_logic_1164 package.

To use any of these one must include the library and use clause:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

3.4. Lexical Elements of VHDL

a) Identifiers

Identifiers are user-defined words used to name objects in VHDL models. When choosing an identifier one needs to follow these basic rules:

- May contain only alpha-numeric characters (A to Z, a to z, 0-9) and the underscore (_) character
- The first character must be a letter and the last one cannot be an underscore.
- An identifier cannot include two consecutive underscores.
- An identifier is case insensitive (ex. And2 and AND2 or and2 refer to the same object)
- An identifier can be of any length.

b) Keywords (Reserved words)

Certain identifiers are used by the system as keywords for special use such as specific constructs. These keywords cannot be used as identifiers for signals or objects we define. We have seen several of these reserved words already such as in, out, or, and, port, map, end, etc.

c) Numbers

The default number representation is the decimal system. VHDL allows integer literals and real literals. Integer literals consist of whole numbers without a decimal point, while real literals always include a decimal point. Exponential notation is allowed using the letter “E” or “e”. For integer literals the exponent must always be positive. Examples are:

Integer literals: 12 10 256E3 12e+6
Real literals: 1.2 256.24 3.14E-2

The number –12 is a combination of a negation operator and an integer literal.

To express a number in a base different from the base “10”, one uses the following convention: base#number#. A few examples follow.

Base 2: 2#10010# (representing the decimal number “18”)
Base 16: 16#12#
Base 8: 8#22#

Base 2: 2#11101# (representing the decimal number “29”)
Base 16: 16#1D#
Base 8: 8#35#

To make the readability of large numbers easier, one can insert underscores in the numbers as long as the underscore is not used at the beginning or the end.

2#1001_1101_1100_0010#
215_123

d) Characters, Strings and Bit Strings

To use a character literal in a VHDL code, one puts it in a single quotation mark, as shown in the examples below:

‘a’, ‘B’, ‘,’

On the other hand, a string of characters are placed in double quotation marks as shown in the following examples:

“This is a string”,
“To use a double quotation mark inside a string, use two double quotation marks”
“This is a “”String””.”

Any printing character can be included inside a string.

A bit-string represents a sequence of bit values. In order to indicate that this is a bit string, one places the ‘B’ in front of the string: B”1001”. One can also use strings in the hexagonal or octal base by using the X or O specifiers, respectively. Some examples are:

Binary: B”1100_1001”, b”1001011”
Hexagonal: X”C9”, X”4b”
Octal: O”311”, o”113”

3.5. Data Objects: Signals, Variables and Constants

A data object is created by an *object declaration* and has a *value* and *type* associated with it. An object can be a Constant, Variable, Signal or a File. Up to now we have seen signals that were used as input or output ports or internal nets. Signals can be considered wires in a schematic that can have a current value and future values, and that are a function of the signal assignment statements. On the other hand, Variables and Constants are used to model the behavior of a circuit and are used in processes, procedures and functions, similarly as they would be in a programming language. Following is a brief discussion of each class of objects

3.5.1. Signal

Signals are declared *outside* the process using the following statement:

```
signal list_of_signal_names: type [ := initial value] ;
```

Examples:

```
signal SUM, CARRY: std_logic;
signal CLOCK: bit;
signal TRIGGER: integer :=0;
signal DATA_BUS: bit_vector (0 to 7);
signal VALUE: integer range 0 to 100;
```

Signals are updated when their signal assignment statement is executed, after a certain delay, as illustrated below,

```
SUM <= (A xor B) after 2 ns;
```

If no delay is specified, the signal will be updated after a delta delay. One can also specify multiple waveforms using multiple events as illustrated below,

```
signal waveform : std_logic;
waveform <= ‘0’, ‘1’ after 5ns, ‘0’ after 10ns, ‘1’ after 20 ns;
```

3.5.2. Variable

A variable can have a single value, as with a constant, but a variable can be updated using a variable assignment statement. The variable is updated without any delay as soon as the statement is executed.

Variables must be declared inside a process (and are local to the process). The variable declaration is as follows:

```
variable list_of_variable_names: type [ := initial value] ;
```

A few examples follow:

```
variable CNTR_BIT: bit :=0;
variable VAR1: boolean :=FALSE;
variable SUM: integer range 0 to 256 :=16;
variable STS_BIT: bit_vector (7 downto 0);
```

The variable SUM, in the example above, is an integer that has a range from 0 to 256 with initial value of 16 at the start of the simulation. The fourth example defines a bit vector or 8 elements: STS_BIT(7), STS_BIT(6),… STS_BIT(0).

A variable can be updated using a variable assignment statement such as

```
Variable_name := expression;
```

As soon as the expression is executed, the variable is updated *without any* delay.

3.5.3. Constant

A constant can have a single value of a given type and cannot be changed during the simulation. A constant is declared as follows,

```
constant list_of_name_of_constant: type [ := initial value] ;
```

where the initial value is optional. Constants can be declared at the start of an architecture and can then be used anywhere within the architecture. Constants declared within a process can only be used inside that specific process.

```
constant  RISE_FALL_TME: time := 2 ns;
constant  DELAY1: time := 4 ns;
constant  RISE_TIME, FALL_TIME: time:= 1 ns;
constant  DATA_BUS: integer:= 16;
```

3.6. Data types

Each data object has a type associated with it. The type defines the set of values that the object can have and the set of operations that are allowed on it. The notion of *type* is key to VHDL since it is a strongly typed language that requires each object to be of a certain type. In general one is not allowed to assign a value of one type to an object of another data type (e.g. assigning an integer to a bit type is not allowed). There are four classes of data types: scalar, composite, access and file types. The scalar types represent a single value and are ordered so that relational operations can be performed on them. The scalar type includes integer, real, and enumerated types of Boolean and Character. Examples of these will be given further on.

a) Data Types defined in the Standard package

VHDL has several predefined types in the standard package as shown in the table below. To use this package the user has to include the following clause:

```
library std, work;
use std.standard.all;
```

Type	Range of values	Example
bit	'0', '1'	signal A: bit :=1;
bit_vector	an array with each element of type bit	signal INBUS: bit_vector(7 downto 0);
boolean	FALSE, TRUE	variable TEST: Boolean :=FALSE'
character	any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. '#')	variable VAL: character :='$';
file_open_kind*	read_mode, write_mode, append_mode	
file_open_status*	open_ok, status_error, name_error, mode_error	
integer	range is implementation dependent but includes at least $-(2^{31} - 1)$ to $+(2^{31} - 1)$	constant CONST1: integer :=129;
natural	integer starting with 0 up to the max specified in the implementation	variable VAR1: natural :=2;
positive	integer starting from 1 up the max specified in the implementation	variable VAR2: positive :=2;
real*	floating point number in the range of -1.0×10^{38} to $+1.0 \times 10^{38}$ (can be implementation dependent. *Not supported by the Foundation synthesis program.*	variable VAR3: real :=+64.2E12;
severity_level	note, warning, error, failure	
string	array of which each element is of the type character	variable VAR4: string(1 to 12):= "@$#ABC*()_%Z";
time*	an integer number of which the range is implementation defined; units can be	variable DELAY: time :=5 ns;

	expressed in sec, ms, us, ns, ps, fs, min and hr. . *Not supported by the Foundation synthesis program*	

*Table 3.1: Types defined in the package **Standard** of the **std** library*

3.7. Operators

VHDL supports different classes of operators that operate on signals, variables and constants. The different classes of operators are summarized below.

Class						
1. Logical operators	**and**	**or**	**nand**	**nor**	**xor**	**xnor**
2. Relational operators	=	/=	<	<=	>	>=
3. Shift operators	**sll**	**srl**	**sla**	**sra**	**rol**	**ror**
4.Addition operators	+	=	&			
5. Unary operators	+	-				
6. Multiplying op.	*	/	**mod**	**rem**		
7. Miscellaneous op.	**	**abs**	**not**			

Table 3.2: Different classes of operators

The order of precedence is the highest for the operators of class 7, followed by class 6 with the lowest precedence for class 1. Unless parentheses are used, the operators with the highest precedence are applied first. Operators of the same class have the same precedence and are applied from left to right in an expression.[6][9]

3.8. A 2bit counter as an example

As an example of a simple project in VHDL, we can take a look at a 2 bit counter that has been written during this thesis as a small exercise.

The counter will consist of 2 LEDs on the board which will count from 0 to 3 in binary, and keep repeating when reaching the end value (which is 3).

The project consists of several files, which are the following: prescaler.vhd, Counter.vhd, Counter_system.vhd, TB_prescaler, TB_Counter and Counter.xdc

```
entity prescaler is
    Port ( clk_in : in STD_LOGIC;
           reset_prescaler : in STD_LOGIC;
           clk_out : out STD_LOGIC);
end prescaler;

architecture Behavioral of prescaler is
    signal co : std_logic;
begin
    clock_prescaler : process (clk_in, reset_prescaler)
        variable count : integer range 0 to 50_000_000;
    begin
        if reset_prescaler = '1' then
            co <= '0';
            count := 0;
        elsif clk_in'event and clk_in = '1' then
            count := count + 1;
            if count = 50_000_000 then
                count := 0;
                co <= not co;
            end if;
        end if;
    end process clock_prescaler;

    clk_out <= co;

end Behavioral;
```

Image 3.4: Code of prescaler.vhd

In this file, we take in the clock of the Nexys A7 board, which is 100MHz. This value is way too high to be able to see the counting with the human eye, so we slow it down to 1Hz. Then we lay this 1Hz signal to the output of the component.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity Counter is
    Port ( clk : in STD_LOGIC;
           rst_counter : in STD_LOGIC;
           countup : in STD_LOGIC;
           q : out unsigned (1 downto 0));
end Counter;

architecture Behavioral of Counter is
    signal count: unsigned (1 downto 0);
begin
    two_bit_counter: process (clk, rst_counter)
    begin
        if rst_counter = '1' then
            count <= (others => '0');
        elsif clk 'event and clk = '1' then
            if countup = '1' then
                count <= count + 1;
            end if;
        end if;
    end process two_bit_counter;

    q <= count;

end Behavioral;
```

Image 3.5: Code of Counter.vhd

In this file, we create the counting component. It will take in a clock signal, a reset signal and a signal to know when it can count. The output of the component will be a 2 bit vector which will hold the count value. As long as *countup* has the value ‘1’, the counter will count up once every rising clock edge. The counter can be reset by giving it a ‘1’ on the *rst_counter* input signal.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity Counter_system is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           countup : in STD_LOGIC;
           q : out unsigned (1 downto 0));
end Counter_system;

architecture Behavioral of Counter_system is
    signal clk_out : std_logic;

begin

    Count : entity work.Counter
        port map (
            clk => clk_out,
            rst_counter => rst,
            countup => countup,
            q => q
            );

    Clk_gen : entity work.prescaler
        port map (
            reset_prescaler => rst,
            clk_in => clk,
            clk_out => clk_out
            );

end Behavioral;
```

Image 3.6: Code of Counter_system.vhd

This is the top module of the project and will connect the different signals to each other.

```
## This file is a general .xdc for the Nexys A7-100T
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clk }];


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { countup }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
#set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
#set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
#set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
#set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
#set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { q[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { q[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
#set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
```

Image 3.7: Small shot of the Counter.xdc file

In this file, we connect the signals of the top module to the pins of the FPGA, to connect them to different components (LEDs, buttons, switches, …) on the Nexys A7 board.

3.9. Vivado Design Suite

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL (Hardware Description Language) designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. Vivado represents a ground-up rewrite and re-thinking of the entire design flow.[3][4]

The Vivado® Design Suite is designed to improve productivity. The tool suite is architected to increase the overall productivity for designing, integrating, and implementing systems using the Xilinx® UltraScale™ and 7 series devices, Zynq® UltraScale+™ MPSoC device, and Zynq®-7000 SoC. Xilinx devices are now much larger and come with a variety of new technology, including stacked silicon interconnect (SSI) technology, up to 28 gigabyte (GB) high speed I/O interfaces, hardened microprocessors and peripherals, analog mixed signal, and more. These larger and more complex devices create multidimensional design challenges, when handled incorrectly, that can prevent the achievement of faster time-to-market and increased productivity. With the Vivado Design Suite, the user can accelerate design implementation with place and route tools that analytically optimize for multiple and concurrent design metrics, such as timing, congestion, total wire length, utilization and power. The Vivado Design Suite provides the user with design analysis capabilities at each design stage. This allows for design and tool setting modifications earlier in the design processes where they have less overall schedule impact, thus reducing design iterations and accelerating productivity.[5]

4. Application development examples

4.1. 4 bit adder

The exercise given by Dr Eng. S. Petrakis was the following:

Design an implementation of a four-bit adder with registers for the operands. Use four (4) seven-segment LED displays, eight slide switches and three push buttons. The aim of this work is to design a four-bit adder with carry and display both, the operands and the results, in hexadecimal representation. It is proposed to create a top module called "Adder" as well as individual modules as follows:

A. a 2 to 4 decoder (DEC2x4),
B. a 2-bit counter (counter2bits),
C. a quadruple multiplexer 4 in 1 (MUX4x1x4),
D. a converter from hexadecimal to seven-segment display, for the control of the operation of the 7-segment display (convHex7seg),
E. a four-bit adder with carry-in and carry-out (Adder4bits),
F. a frequency divider (prescaler240Hz), and
G. of a four-bit register with parallel loading (register4bits). You will need two of these for the operands.

In the development card there are two groups of four seven-segment displays each. Choose one group and make sure that the two 7-segment displays on the left represent the operands and the two on the right the sum (all in the hexadecimal numbering system). In addition to the reset button, use one button for the input carry and one to load the values into the registers.

To start this project, a schematic was drawn to figure out the needed components:

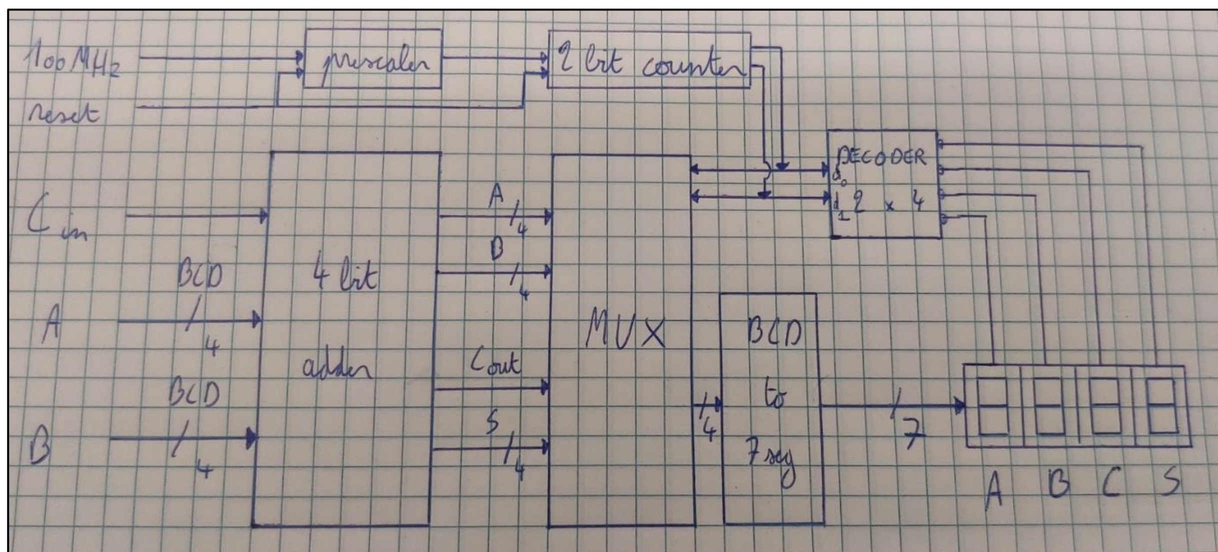


Image 4.1: Schematic of the 4 bit adder project

This schematic does not have the registers, as those were added later on during the development of this project.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity four_bit_adder is
    Port ( A: in STD_LOGIC_VECTOR(3 downto 0);
           B: in STD_LOGIC_VECTOR(3 downto 0);
           Cin: in STD_LOGIC;
           S: out STD_LOGIC_VECTOR(3 downto 0);
           Cout: out std_logic
           );
end four_bit_adder;

architecture str of four_bit_adder is
    component fa is
    port( a, b, c : in std_logic;
          co : out std_logic;
          s : out std_logic
          );
    end component;

signal temp1, temp2, temp3 : std_logic;

begin
    fa0:fa port map(a=>A(0), b=>B(0), c=>Cin,   co=>temp1, s=>S(0));
    fa1:fa port map(a=>A(1), b=>B(1), c=>temp1, co=>temp2, s=>S(1));
    fa2:fa port map(a=>A(2), b=>B(2), c=>temp2, co=>temp3, s=>S(2));
    fa3:fa port map(a=>A(3), b=>B(3), c=>temp3, co=>Cout,  s=>S(3));
end str;
```

Image 4.2: four_bit_adder component

In this component, we describe the 4 bit adder. It has 3 inputs (A, B and Cin) and 2 outputs (S and Cout). When we want to test this component before uploading it to the Nexys A7 board, we can write a simulation component, also called a testbench, to be able to simulate the code that was just written.

```
BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: four_bit_adder PORT MAP (
          A => A,
          B => B,
          Cin => Cin,
          S => S,
          Cout => Cout
        );

   stim_proc_A: process
   begin
      A <= "0100";
      wait for 100 ns;
      A <= "0111";
      wait;
   end process;


   stim_proc_B: process
   begin
      B <= "1111";
      wait for 100 ns;
      B <= "0011";
      wait;
   end process;


   stim_proc_Cin: process
   begin
        Cin <= '0';
      wait for 100 ns;
      wait;
   end process;
end Behavioral;
```

Image 4.3: Small shot of four_bit_adder testbench

When we then press the simulate button, we get the following output:

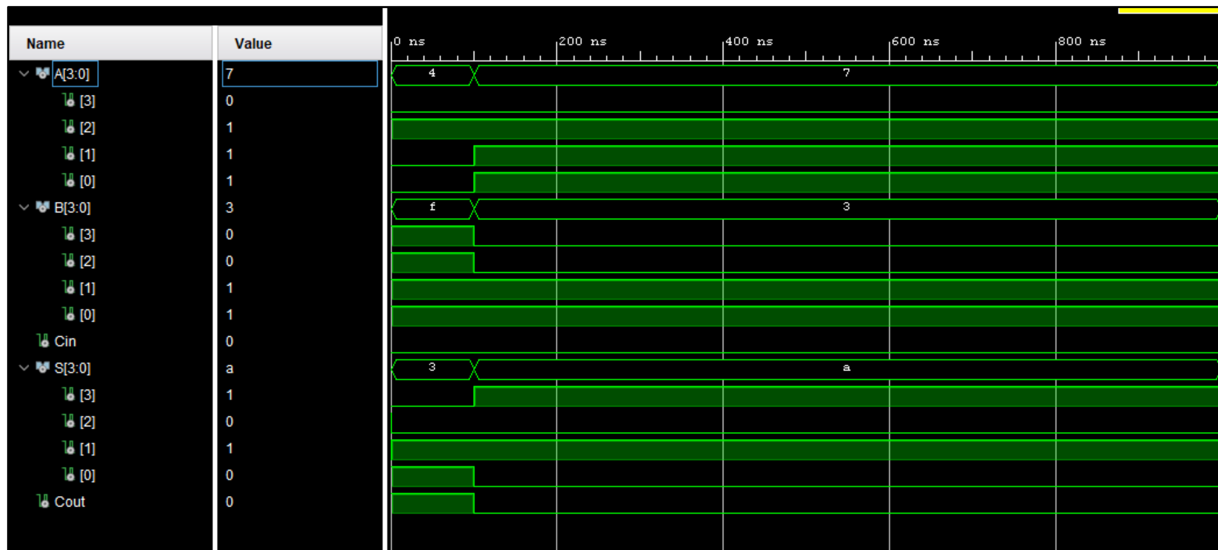


Image 4.4: Output of the four_bit_adder testbench

Here, we can check and confirm if the code we had just written provides the output we are expecting, or if there is a mistake in the code. It is way quicker than uploading the code to the Nexys A7 board every time we change a small portion of the code and provides detailed output of what is happening.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multiplexer is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           C : in STD_LOGIC_VECTOR (3 downto 0);
           D : in STD_LOGIC_VECTOR (3 downto 0);
           S0 : in STD_LOGIC;
           S1 : in STD_LOGIC;
           Z : out STD_LOGIC_VECTOR (3 downto 0));
end multiplexer;

architecture Behavioral of multiplexer is

begin

    mux: process (A, B, C, D, S0, S1)
    begin
        if    (S0 = '0' and S1 = '0') then
            Z <= A;
        elsif (S0 = '1' and S1 = '0') then
            Z <= B;
        elsif (S0 = '0' and S1 = '1') then
            Z <= C;
        else
            Z <= D;
        end if;
    end process;
end Behavioral;
```

Image 4.5: multiplexer component

On figure 4.5, we can see the multiplexer component that was needed for this project. It has 6 inputs (A, B, C, D, S0 and S1) and 1 output (Z). It will take the signals from the adder as input and put them at the output one by one to be able to display them on the 7-segment display.

The whole working of this 4-bit adder to 7 segment works on the 2 bit counter, which is driven by the prescaler. An explanation for the working of the prescaler was given in chapter 3. The prescaler will drive the 2-bit counter, which in turn will drive the multiplexer and decoder. The code of the decoder can be seen on figure 4.6.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decoder is
    Port ( d : in STD_LOGIC_VECTOR(1 downto 0);
           y : out STD_LOGIC_VECTOR(3 downto 0)
           );
end decoder;

architecture Behavioral of decoder is

begin

    y(0) <= not (not d(0) and not d(1));
    y(1) <= not (    d(0) and not d(1));
    y(2) <= not (not d(0) and     d(1));
    y(3) <= not (    d(0) and     d(1));


end Behavioral;
```

Image 4.6: decoder component

This decoder component has 1 input (d, a 2-bit logic vector) and 1 output (y, a 4-bit logic vector). It will take in the 2-bit counter value and will in turn put one of 4 bits on the output high, one by one. Using this decoder, we can turn on one 7 segment display at a time, so we can multiplex the 4 used 7 segment displays. A little explanation on multiplexing will follow later. This way, only 7 pins of the microcontroller are used to control the cathodes of the segments, while 4 pins (1 pin for each used 7 segment display) are used to control the common anode of each display.

4.1.1. A word on multiplexing

Multiplexed displays are electronic display devices where the entire display is not driven at one time. Instead, sub-units of the display are multiplexed, this means, driven one at a time, but this happens so quickly, it makes the viewer believe the entire display is continuously active. [7][8]

4.2. Temperature Sensor

The exercise provided was the following: Use the embedded temperature sensor of the development card and display the temperature on seven-segment displays with decimal values in Celsius grades.

To start, the components TempSensorCtl and TWICtl, provided by Digilent, were used to interface with the temperature sensor on the Nexys A7 board. This made it much easier to work with the temperature sensor, as there was no need to write our own interfacing module.

The components were gathered from the Digilent GitHub. Here we can find the whole project that is played on the Nexys A7 board whenever the user powers it on the first time before writing code to it.

The multiplexer and decoder from the 4 bit adder were used to be able to drive the 7 segment displays, this in turn also saved a lot of time and work. But the difference in this project, was that the values have to be displayed in decimal, and not hexadecimal. To make this work, we had to find a solution for when a value is higher than 9, and in turn would need 2 seven segment displays to be displayed.

The solution for this, was to split the value at the decimal point, and recombine these 2 sides at the end. This was done through 2 components; binary_to_BCD_lp and binary_to_BCD_rp, where *lp* stands for *left part* and *rp* for *right part*.

```
entity binary_to_BCD_rp is
    Port ( input : in STD_LOGIC_VECTOR (3 downto 0);
           output : out STD_LOGIC_VECTOR (3 downto 0));
end binary_to_BCD_rp;

architecture Behavioral of binary_to_BCD_rp is
    signal right_part : STD_LOGIC_VECTOR (3 downto 0);
begin
    process (input)
    begin
        case input is when "0000" => right_part <= "0000"; --0
                      when "0001" => right_part <= "0001"; --1
                      when "0010" => right_part <= "0001"; --1
                      when "0011" => right_part <= "0010"; --2
                      when "0100" => right_part <= "0011"; --3
                      when "0101" => right_part <= "0011"; --3
                      when "0110" => right_part <= "0100"; --4
                      when "0111" => right_part <= "0100"; --4
                      when "1000" => right_part <= "0101"; --5
                      when "1001" => right_part <= "0110"; --6
                      when "1010" => right_part <= "0110"; --6
                      when "1011" => right_part <= "0111"; --7
                      when "1100" => right_part <= "1000"; --8
                      when "1101" => right_part <= "1000"; --8
                      when "1110" => right_part <= "1001"; --9
                      when "1111" => right_part <= "1001"; --9
                      when others => right_part <= "0000"; --null
        end case;
    end process;

    output <= right_part;

end Behavioral;
```

Image 4.7: binary_to_BCD_rp component

In figure 4.7, we see the right part component. The input (4 bits) is converted to a decimal value and then delivered at the output.

HEXADECIMAL							decimal
0	0					0,0000	0,0
1	1				0,0625	0,0625	0,1
2	2			0,125		0,1250	0,1
3	3			0,125	0,0625	0,1875	0,2
4	4		0,25			0,2500	0,3
5	5		0,25		0,0625	0,3125	0,3
6	6		0,25	0,125		0,3750	0,4
7	7		0,25	0,125	0,0625	0,4375	0,4
8	8	0,5				0,5000	0,5
9	9	0,5			0,0625	0,5625	0,6
A	10	0,5		0,125		0,6250	0,6
B	11	0,5		0,125	0,0625	0,6875	0,7
C	12	0,5	0,25			0,7500	0,8
D	13	0,5	0,25		0,0625	0,8125	0,8
E	14	0,5	0,25	0,125		0,8750	0,9
F	15	0,5	0,25	0,125	0,0625	0,9375	0,9

Image 4.8: Hexadecimal to decimal conversion

The calculation for the conversion from hexadecimal to decimal can be seen on figure 4.8. An example can be:

HEX Value: D -> 1101
So: 1 * 0,5 + 1 * 0,25 + 0 * 0,125 + 1 * 0,0625 = 0,8125 (0,8 rounded)

A working project can be seen on figure 4.7. Here, the values were still in hexadecimal, as this image was taken during the development of the project.

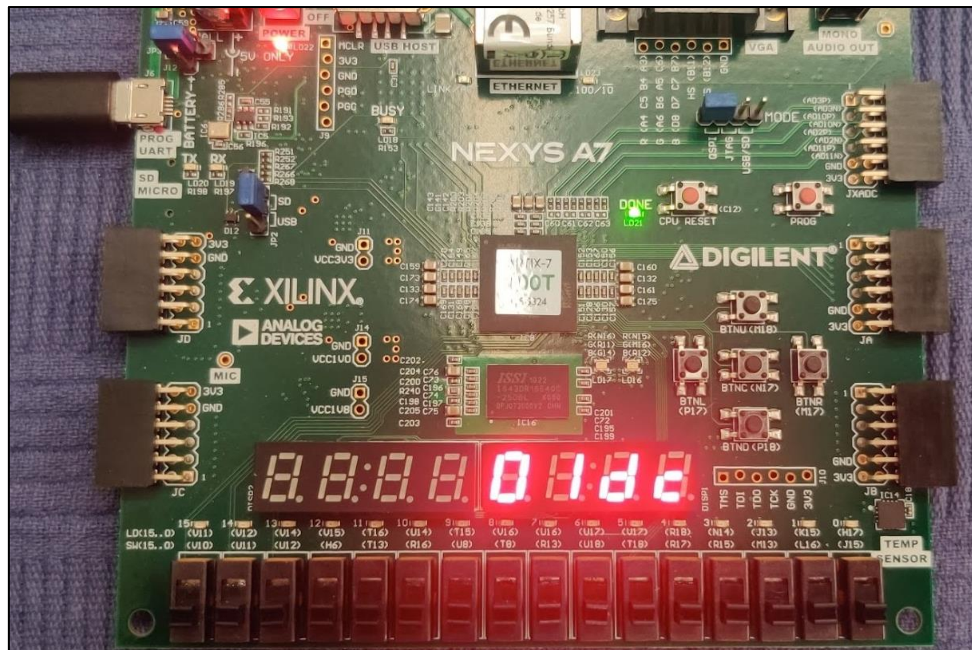


Image 4.9: temperature reading in hexadecimal

To be able to read this temperature, the user performs the following calculations:

Hex value: 0x1DC
Binary value: 0 0000 0001 1101 1100
Calculation: 16^1 * 1 + 16^0 * 13 + 16^-1 * 12 = 29,75
Decimal value: 29,75

After it was made sure the sensor readings were correct, the conversion from hexadecimal to decimal was made, as this was not an easy task, it was done in steps.

4.3. Accelerometer

The exercise provided was the following:

Use the embedded accelerometer of the development card and turn on or off specific LEDs depending of the inclination of the card in comparison with the horizontal position.

Again, components provided by Digilent were used to make it easier to interface with the accelerometer sensor. These files are ADXL362Ctrl and SPI_If.

As the board tilts to one side or the other, the LEDs on each side, and the LEDs in the middle, to show if the board was sitting horizontal, would be used to show the position.

To make this easy, the LEDs were mapped to a 16 bit array, each LED could on or off, 1 or 0. The accelerometer sensor outputs a value from -2^{11} to $2^{11} - 1$. This means the output will be from -2048 to 2047.

If the value would be smaller than -1000, the left most LEDs would light up. If it was higher than 1000, the most right LEDs would light up, and if the value was around 0, it would be the middle LEDs.

```
entity acc_to_leds is
    Port ( input : in STD_LOGIC_VECTOR (11 downto 0); --output accelerometer in 12 bits (-2^11 <=> 2^11 -1)
           output : out STD_LOGIC_VECTOR (15 downto 0)); --16 leds in total
end acc_to_leds;

architecture Behavioral of acc_to_leds is

    signal leds_out : STD_LOGIC_VECTOR (15 downto 0);
begin
    process (input)
    variable input_int : integer := TO_INTEGER(UNSIGNED(input));
        begin
            if (input_int < -1000) then
                leds_out <= "1100000000000000";
            elsif (input_int > -2) AND (input_int < 2) then
                leds_out <= "0000000110000000";
            elsif (input_int > 1000) then
                leds_out <= "0000000000000011";
            end if;
    end process;

    output <= leds_out;

end Behavioral;
```

Image 4.10: acc_to_leds component

On figure 4.10, we can see the implementation of this.

While in theory, this seemed to be a perfect solution, this didn’t work in practice. Only one side of LEDs was lit up, and didn’t turn off while moving the board around. Because of time limitations, no solution to this was found.

5. Conclusions

Reading the above, it becomes clear that the introduction of FPGAs in the implementation of digital systems requires the use of efficient design tools. Leading companies in the production of FPGA devices are Xilinx, Intel (acquired by Altera) and Lattice Semiconductor, each of which has its own commercial design tool. Specifically, Xilinx has the Vivado Design Suite which replaced the obsolete ISE.

The purpose of this final project was to study the features and capabilities offered by Digilent's Nexys A7 development board that contains Xilinx FPGA Artix-7 and is compatible with both ISE and Vivado (free Webpack version). Among other things, the development board has 16 slide switches, 16 LEDs, 8 7-segment displays, 5 buttons and incorporates multiple peripherals such as temperature sensor, digital microphone etc. which make it possible to implement applications without the use of other components.

So, using Vivado, version 2020.2, three exercises were developed, which were implemented in Xilinx's FPGA (XC7A100T-1CSG324C). The board has a variety of peripherals and ports whose use could be tested with the development of other new exercises.

Indicatively, the analog temperature sensor (ADT7420) can be used which measures temperatures from -40 ° C to 150 ° C and communicates via the I²C standard. It has 14 8-bit registers, of which 9 are temperature registers. After the appropriate registrations have been made in some registers, it is then possible to read the values of the temperature registers TEMP_H, TEMP_L and display these values in the LEDs, in the 7-segment displays or even on a screen via the VGA connection of the board.

Another exercise could be to use the microphone containing the analog ADMP421 where the digitized sound is received via pulse modulation (PDM). It would even be interesting to use the two tricolor LEDs of the development board, which are driven via pulse width modulation (PWM).

Also, where appropriate, a mouse and / or keyboard that can be connected via the USB port provided could be used.

Finally, unlimited possibilities for various applications are provided by the existing PMOD ports, in case the laboratory supplies the appropriate modules, such as a Global Positioning Receiver (Pmod GPS: GPS Receiver), or an ultrasonic distance meter (Pmod MAbotixar: Max Ultrasonic Range Finder) etc.

References

Books

[9] V. A. Pedroni, Circuit Design and Simulation with VHDL. The MIT Press, 2010

Webpages

[1] https://reference.digilentinc.com/programmable-logic/nexys-a7/reference-manual

[2] https://en.wikipedia.org/wiki/VHDL

[3] https://en.wikipedia.org/wiki/Xilinx_Vivado

[4] https://www.xilinx.com/products/design-tools/vivado.html

[6] https://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html

[7] https://en.wikipedia.org/wiki/Multiplexing

[8] https://en.wikipedia.org/wiki/Multiplexed_display

Other

[5] Xilinx Document Navigator : Vivado Design Suite User Guide: Getting Started